AWS DeepRacer

Developer Guide

AWS DeepRacer: Developer Guide

Table of Contents

What Is AWS DeepRacer? 1
- The AWS DeepRacer Console 1
- The AWS DeepRacer Vehicle 1
- The AWS DeepRacer League 2
- An Integrated Learning System 2
- Concepts and Terminology 3
  - Racing Event Terminology 6
- Pricing 6

How It Works 7
- Reinforcement Learning 7
- Action Space and Reward Function 9
- Training Algorithm 10
- AWS DeepRacer Service Architecture 12
- AWS DeepRacer Workflow 13
- Simulated-to-Real Performance Gaps 14

Get Started 15
- Set Up Account Resources 15
- Train Your First Model 15
- Evaluate Models in Simulation 20

Train and Evaluate Models 24
- Understanding Racing Types and Enabling Sensors 25
  - Choose Sensors 25
  - Configure Your Agent for Training 27
  - Tailor Training for Time Trials 28
  - Tailor Training for Object Avoidance Races 29
  - Tailor Training for Head-to-Head Races 30
- Train and Evaluate Models Using AWS DeepRacer Console 31
  - Create Your Reward Function 31
  - Explore Action Space 33
  - Tune Hyperparameters 34
  - Examine Training Job Progress 37
  - Clone a Trained Model 38
  - Evaluate Models in Simulations 39
  - Log Events to CloudWatch Logs 39
  - Optimize Training for Real Environments 41
- Train and Evaluate Models Using Amazon SageMaker Notebooks 43
  - Create a Notebook 43
  - Initialize the Notebook Instance 44
  - Set Up the Environment 48
  - Train Your AWS DeepRacer Model 51
- Reward Function Reference 58
  - Reward Function Input Parameters 58
  - Reward Function Examples 73

Operate Your Vehicle 77
- Get to Know Your Vehicle 77
  - Inspect Your Vehicle 78
  - Charge and Install Batteries 79
  - Test Compute Module 80
  - Turn Off Your Device 81
  - LED Indicators 81
  - Vehicle Spare Parts 83
- Set Up Your Vehicle 91
  - Get Ready to Set Up Wi-Fi 91
  - Set Up Wi-Fi and Update Software 92

Launch Device Console 92
Calibrate Your Vehicle 94
Upload Your Model 100
Drive Your Vehicle 101
Drive Your AWS DeepRacer Vehicle Manually 101
Drive Your AWS DeepRacer Vehicle Autonomously 102
Inspect and Manage Vehicle Settings 103
View Vehicle Logs 107
Build Your Physical Track 109
Materials and Tools 109
Materials You May Need 109
Tools You May Need 109
Lay Your Track 110
Dimensional Requirements 110
Model Performance Considerations 111
Steps to Build the Track 111
Track Design Templates 114
Single-Turn Track Template 115
S-Curve Track Template 115
Loop Track Template 116
re:Invent 2018 Track Template 116
Championship Cup 2019 Track Template 118
Participate in Virtual Races 119
Racing Event Types 119
Joining an Online AWS-sponsored or Community-Sponsored Race 119
Join a Virtual Circuit Race 120
Join a Community Race 123
Organize a Community Race 126
Manage Community Races 128
Security 130
AWS DeepRacer-Dependent Services 130
Required IAM Roles 132
Troubleshoot Common Issues 133
Why Can't I Connect to the Device Console with USB Connection between My Computer and Vehicle? 133
How to Switch AWS DeepRacer Compute Module Power Source from Battery to a Power Outlet 136
How to Connect Your AWS DeepRacer to Your Wi-Fi Network 138
How to Charge the Vehicle's Drive Module Battery 138
How to Charge the Vehicle's Compute Module Battery 139
My Battery Is Charged but My Vehicle Doesn't Move 140
How to Maintain Your Vehicle's Connection 142
How to Troubleshoot Wi-Fi Connection If Your Vehicle's Wi-Fi LED Indicator Flashes Blue, Then Turns Red for Two Seconds, and Finally Off 142
What Does It Mean When the Vehicle's Wi-Fi or Power LED Indicator Flashes Blue? 143
How Can I Connect to the Vehicle's Device Console Using its Hostname? 143
How to Connect to Vehicle's Device Console Using Its IP Address 143
How to Get Your Device's Mac Address 143
How to Recover the Device Controller Default Password 144
How to Manually Update Your Device 146
How to Diagnose and Resolve Common Device Operational Issues 147
Why Doesn't the Video Player on the Device Console Show the Video Stream from My Vehicle's Camera? 147
Why Doesn't My AWS DeepRacer Vehicle Move? 147
Why Don't I See the Latest Device Update? How Do I Get the Latest Update? 148
Why Isn't My AWS DeepRacer Vehicle Connected to My Wi-Fi Network? 148
Why Does the AWS DeepRacer Device Console Page Take a Long Time to Load? 148
Why Does a Model Fail to Perform Well When Deployed to an AWS DeepRacer Vehicle? 148

Restore Vehicle to Factory Settings 149
Preparing for Factory Reset 149
Restore to Factory Settings 163
Document History 164
AWS glossary 165

What Is AWS DeepRacer?

AWS DeepRacer is an integrated learning system for users of all levels to learn and explore reinforcement learning (p. 3) and to experiment and build autonomous driving applications. It consists of the following components:

- AWS DeepRacer Console: an AWS Machine Learning service to train and evaluate reinforcement learning models (p. 24)in a simulated autonomous-driving environment.
- AWS DeepRacer Vehicle: a 1/18th scale RC car capable of running inference on a trained AWS DeepRacer model (p. 77) for autonomous driving.
- AWS DeepRacer League: the world's first global, autonomous racing league. Race for prizes, glory, and a chance to advance to the Championship Cup.

Topics

- The AWS DeepRacer Console (p. 1)
- The AWS DeepRacer Vehicle (p. 1)
- The AWS DeepRacer League (p. 2)
- AWS DeepRacer As an Integrated Learning System (p. 2)
- AWS DeepRacer Concepts and Terminology (p. 3)
- Pricing (p. 6)

The AWS DeepRacer Console

The AWS DeepRacer console is a graphical user interface to interact with the AWS DeepRacer service. You can use the console to train a reinforcement learning model and to evaluate the model performance in the AWS DeepRacer simulator built upon AWS RoboMaker. In the console, you can also download a trained model for deployment to your AWS DeepRacer vehicle for autonomous driving in a physical environment.

In summary, the AWS DeepRacer console supports the following features:

- Create a training job to train a reinforcement learning model with a specified reward function, optimization algorithm, environment, and hyperparameters.
- Choose a simulated track to train and evaluate a model by using Amazon SageMaker and AWS RoboMaker.
- Clone a trained model to improve training by tuning hyperparameters to optimize your model's performance.
- Download a trained model for deployment to your AWS DeepRacer vehicle so it can drive in a physical environment.
- Submit your model to a virtual race and have its performance ranked against other models in a virtual leaderboard.

The AWS DeepRacer Vehicle

The AWS DeepRacer vehicle is a Wi-Fi enabled, physical vehicle that can drive itself on a physical track by using a reinforcement learning model.

- You can manually control the vehicle, or deploy a model for the vehicle to drive autonomously.
- The autonomous mode runs inference on the vehicle's compute module. Inference uses images that are captured from the camera that is mounted on the front.
- A Wi-Fi connection allows the vehicle to download software. The connection also allows the user to access the device console to operate the vehicle by using a computer or mobile device.

The AWS DeepRacer League

The AWS DeepRacer League League is an important component of AWS DeepRacer. The AWS DeepRacer League is intended to foster communal learning and collaborative exploration through sharing and competition.

With the AWS DeepRacer League, you can have your development effort compared with other AWS DeepRacer developers in a physical or virtual racing event. Not only do you get a chance to win prizes, you also have a way to measure your reinforcement learning model. You can create opportunities to share your insights with other participants, to learn from each other, and to inspire each other.

Join a race or learn how to train a model in the League.

AWS DeepRacer As an Integrated Learning System

Reinforcement learning, especially deep reinforcement learning, has proven effective in solving a wide array of autonomous decision-making problems. It has applications in financial trading, data center cooling, fleet logistics, and autonomous racing, to name a few.

Reinforcement learning has the potential to solve real-world problems. However, it has a steep learning curve because of the extensive technological scope and depth. Real-world experimentation requires that you construct a physical agent, e.g., an autonomous racing car. It also requires that you secure a physical environment, e.g., a driving track or public road. The environment can be costly, hazardous, and time-consuming. These requirements go beyond merely understanding reinforcement learning.

To help reduce the learning curve, AWS DeepRacer simplifies the process in three ways:

- By offering a wizard to guide training and evaluating reinforcement learning models. The wizard includes pre-defined environments, states, actions, and customizable reward functions.
- By providing a simulator to emulate interactions between a virtual agent (p. 4) and a virtual environment.
- By offering an AWS DeepRacer vehicle as a physical agent. Use the vehicle to evaluate a trained model in a physical environment. This closely resembles a real-world use case.

If you are a seasoned machine learning practitioner, you will find AWS DeepRacer a welcome opportunity to build reinforcement learning models for autonomous racing in both virtual and physical environments. To summarize, use AWS DeepRacer to create reinforcement learning models for autonomous racing with the following steps:

1. Train a custom reinforcement learning model for autonomous racing. Do this by using the AWS DeepRacer console integrated with Amazon SageMaker and AWS RoboMaker.
2. Use the AWS DeepRacer simulator to evaluate a model and test autonomous racing in a virtual environment.
3. Deploy a trained model to AWS DeepRacer model vehicles to test autonomous racing in a physical environment.

AWS DeepRacer Concepts and Terminology

AWS DeepRacer builds on the following concepts and uses the following terminology.

AWS DeepRacer

Also referred to as AWS DeepRacer vehicle. One type of AWS DeepRacer vehicle is an AWS DeepRacer car that is a 1/18th scale model car. It has a mounted camera and an on-board compute module. The compute module runs inference in order to drive itself along a track. The compute module and the vehicle chassis are powered by dedicated batteries known as the compute battery and the drive battery, respectively.

AWS DeepRacer service

An AWS Machine Learning service for exploring reinforcement learning that is focused on autonomous racing. The AWS DeepRacer service supports the following features:

1. Train a reinforcement learning model on the cloud.
2. Evaluate a trained model in the AWS DeepRacer console.
3. Submit a trained model to a virtual race and, if qualified, have its performance posted to the event's leaderboard.
4. Clone a trained model continue training for improved performances.
5. Download the trained model artifacts for uploading to an AWS DeepRacer vehicle.
6. Place the vehicle on a physical track for autonomous driving and evaluate the model for real-world performances.
7. Run the AWS DeepRacer League for racing events on physical tracks.
8. Remove unnecessary charges by deleting models that you don't need.

Reinforcement learning

A machine learning method that is focused on autonomous decision making by an agent in order to achieve specified goals through interactions with an environment. In reinforcement learning, learning is achieved through trial and error and training does not require labeled input. Training relies on the reward hypothesis. The hypothesis is that all goals can be achieved by maximizing a future reward after action sequences. In reinforcement learning, designing the reward function is important. The better the reward function is crafted, the better the agent can decide what actions to take to reach the goal.

For autonomous racing, the agent is a vehicle. The environment includes traveling routes and traffic conditions. The goal is for the vehicle to reach its destination quickly without accidents. Rewards are scores used to encourage safe and speedy travel to the destination. The scores penalize dangerous and wasteful driving.

To encourage learning during training, the learning agent must be allowed to sometimes pursue actions that might not result in rewards. This is referred to as the exploration and exploitation trade-off. It helps reduce or remove the likelihood that the agent might be misguided into false destinations.

For a more formal definition, see reinforcement learning on Wikipedia.

Reinforcement learning model

The environment in which an agent acts establishes three things: The states that the agent has, the actions that the agent can take, and the rewards that are received by taking action. The strategy with which the agent decides its action is referred to as a policy. The policy takes the environment state as input and outputs the action to take. In reinforcement learning, the policy is often represented by a deep neural network and we refer to this as the reinforcement learning

model. Each training job generates one model. A model can be generated even if the training job is stopped early. A model is immutable, which means it cannot be modified and overwritten after it's created.

AWS DeepRacer simulator

A virtual environment built on AWS RoboMaker for visualizing training and evaluating AWS DeepRacer models.

AWS DeepRacer vehicle

See AWS DeepRacer (p. 3)

AWS DeepRacer car

A type of AWS DeepRacer vehicle (p. 4) that is a 1/18th scale model car.

Leaderboard

A *leaderboard* is a ranked list of AWS DeepRacer vehicle performances in an AWS DeepRacer League racing event. The race can be a virtual event, carried out in the simulated environment, or a physical event, carried out in a real-world environment. The performance metric is the average lap time submitted by AWS DeepRacer users who have evaluated their trained models on a track identical or similar to the given track of the race.

If a vehicle completes three laps consecutively, then it qualifies to be ranked on a leaderboard. The average lap time for the first three consecutive laps is submitted to the leaderboard.

Machine-learning frameworks

The software libraries used to build machine learning algorithms. Supported frameworks for AWS DeepRacer include Tensorflow.

Policy network

A policy network is a neural network that is trained. The policy network takes video images as input and predicts the next action for the agent. Depending on the algorithm, it may also evaluate the value of current state of the agent.

Optimization algorithm

An optimization algorithm is the algorithm used to train a model. For supervised training, the algorithm is optimized by minimizing a loss function with a particular strategy to update weights. For reinforcement learning, the algorithm is optimized by maximizing the expected future rewards with a particular reward function.

Neural network

A collection of connected units or nodes that are used to build an information model based on biological systems. Also referred to as artificial neural network. Each node is called an artificial neuron and mimics a biological neuron in that it receives an input (stimulus), becomes activated if the input signal is strong enough (activation), and produces an output predicated upon the input and activation. It's widely used in machine learning because an artificial neural network can serve as a general-purpose approximation to any function. Teaching machine to learn becomes finding the optimal function approximation for the given input and output. In deep reinforcement learning, the neural network represents the policy and is often referred to as the policy network. Training the policy network amounts to iterating through steps that involve generating experiences based on the current policy followed by optimizing the policy network with the newly generated experiences. The process continues until certain performance metrics meet required criteria.

Hyperparameters

Algorithm-dependent variables that control the performance of training a neural network. An example hyperparameter is the learning rate that controls how much new experiences are counted

for in learning at each step. A larger learning rate makes a faster training but may make the trained model lower quality. Hyperparameters are empirical and require systematic tuning for each training.

AWS DeepRacer Track

A path or course on which an AWS DeepRacer vehicle drives. The track can exist in either a simulated or real-world, physical environment. You use a simulated environment for training an AWS DeepRacer model on a virtual track. The AWS DeepRacer console makes virtual tracks available. You use a real-world environment for running an AWS DeepRacer vehicle on a physical track. The AWS DeepRacer League provides physical tracks for event participants to compete. You must create your own physical track if you want to run your AWS DeepRacer vehicle in any other situation.

Reward function

An algorithm within a learning model that tells the agent whether the action performed resulted in:

- A good outcome that should be reinforced.
- A neutral outcome.
- A bad outcome that should be discouraged.

The reward function is a key part of reinforcement learning. It determines the behavior that the agent learns by incentivizing specific actions over others. The user provides the reward function by using Python. This reward function is used by an optimizing algorithm to train the reinforcement learning model.

Experience episode

A period in which the agent collects experiences as training data from the environment by running from a given starting point to completing the track or going off the track. Different episodes can have different lengths. Also referred to as an episode or experience-generating episode.

Experience iteration

Consecutive experiences between each policy iteration that performs updates of the policy network weights. Also referred to as an experience-generating iteration, the size can be set in one of the hyperparameters for training. At the end of each experience iteration, the collected episodes are added to an experience replay or buffer. The neural network is updated by using random samples of the experiences.

Policy iteration

Any number of passes through the randomly sampled training data to update the policy neural network weights during gradient ascent. A single pass through the training data to update the weights is also known as an epoch. Policy iteration is also referred to as policy-updating iteration.

Training job

A workload that trains a reinforcement learning model and creates trained model artifacts on which to run inference. Each training job has two sub-processes:

1. Start the agent to follow the current policy. The agent explores the environment in a number of *episodes* (p. 5) and creates training data. This data generation is an iterative process itself.
2. Apply the new training data to compute new policy gradients. Update the network weights and continue training. Repeat Step 1 until a stop condition is met.

Each training job produces a trained model and outputs the model artifacts to a specified data store.

Evaluation job

A workload that tests the performance of a model. Performance is measured by given metrics after the training job is done. The standard AWS DeepRacer performance metric is the driving time that an agent takes to complete a lap on a track. Another metric is the percentage of the lap completed.

Racing Event Terminology

League/Competition

In the context of AWS DeepRacer League events, the terms league and competition relate to the competition structure. AWS sponsors the AWS DeepRacer League, which means we own it, design it, and execute it. A competition has a start and end date.

Season

A competition can repeat in subsequent years. We call these different seasons (for example, the 2019 season or 2020 season). Rules can change from season to season, but are typically consistent within a season. Terms and conditions for the AWS DeepRacer League can vary from season to season.

Circuit

In the AWS DeepRacer League, you can choose to race in the Summit Circuit or in the Virtual Circuit. The Summit Circuit refers to in-person races happening at selected AWS Summits. The Virtual Circuit refers to the online races happening in the AWS DeepRacer console.

Event

As defined by the rules, an event is an AWS DeepRacer League occurance where you can participate in a race. It can be in-person, like the Summit Circuit, or online, like the virtual circuit. An event has a start and end date. Virtual Circuit events will typically last a month, whereas Summit Circuit events will typically last a day. There can be many events in a season, and some rules, such as how we rank those participating in an event, select who wins, and what happens thereafter are subject to change.

Race type

There are flexible options for race types available at each event. In the Summit Circuit, you can choose to race in a time-trial (TT) race, or in a head-to-head (H2H) race, or both. In the Virtual Circuit, you can race in a time-trial (TT), object-avoidance (OA), or head-to-head (H2H) race, or all three. Beginning in 2020, the default Virtual Circuit program includes TT, OA, and H2H. However, we may add a fourth bonus race at a particular event. Alternatively, an event host could choose to only offer a TT race. Each race type may also specify the number of laps, how racers are ranked and so on.

Pricing

When you use the AWS DeepRacer service console you will be charged based on your usage. Your monthly AWS billing statement will show a charge from each of the AWS services used by AWS DeepRacer to train models, evaluate models, or store models and metadata, such as log files.

To get you started, AWS DeepRacer provides a Free Tier to first time AWS DeepRacer users, that should cover the first 7 hours of training. This is enough time to train and tune your first model and enter the AWS DeepRacer League. There is no cost for submitting a model to take part in any AWS DeepRacer League virtual event.

For details about pricing see the AWS DeepRacer service detail page.

How AWS DeepRacer Works

AWS DeepRacer vehicle is a 1/18th scale vehicle that can autonomously drive along a track by itself or race against another vehicle. The vehicle can be equipped with various sensors that include a front-facing camera, stereo cameras, radars or a LiDAR. The sensors collect data about the environment the vehicle operates in. Different sensors provide the view at different scales.

AWS DeepRacer uses reinforcement learning to enable autonomous driving for the AWS DeepRacer vehicle. To achieve this, you train and evaluate a reinforcement learning model in a virtual environment with a simulated track. After the training, you upload the trained model artifacts to your AWS DeepRacer vehicle. You can then set the vehicle for autonomous driving in a physical environment with a real track.

Training a reinforcement learning model can be challenging, especially if you're new to the field. AWS DeepRacer simplifies the process by integrating required components together and providing easy-to-follow wizard-like task templates. However, it's helpful to have a good understanding of the basics of reinforcement learning training implemented in AWS DeepRacer.

Topics

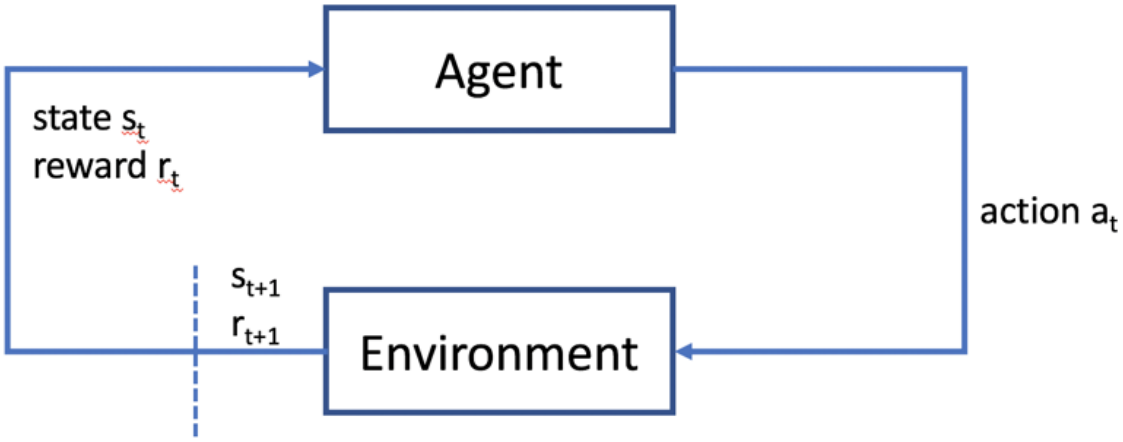
- Reinforcement Learning in AWS DeepRacer (p. 7)
- AWS DeepRacer Action Space and Reward Function (p. 9)
- AWS DeepRacer Training Algorithm (p. 10)
- AWS DeepRacer Service Architecture (p. 12)
- AWS DeepRacer Solution Workflow (p. 13)
- Simulated-to-Real Performance Gaps (p. 14)

Reinforcement Learning in AWS DeepRacer

In reinforcement learning, an *agent*, such as a physical or virtual AWS DeepRacer vehicle, with an objective to achieve an intended goal interacts with an *environment* to maximize the agent's total reward. The agent takes an *action*, guided by a strategy referred to as a *policy*, at a given environment *state* and reaches a new state. There is an immediate *reward* associated with any action. The reward is a measure of the desirability of the action. This immediate reward is considered to be returned by the environment.

The goal of the reinforcement learning in AWS DeepRacer is to learn the optimal policy in a given environment. Learning is an iterative process of trials and errors. The agent takes the random initial action to arrive at a new state. Then the agent iterates the step from the new state to the next one. Over time, the agent discovers actions that lead to the maximum long-term rewards. The interaction of the agent from an initial state to a terminal state is called an *episode*.

The following sketch illustrates this learning process:



The *agent* embodies a neural network that represents a function to approximate the agent's policy. The image from the vehicle's front camera is the environment *state* and the agent *action* is defined by the agent's speed and steering angles.

The agent receives positive *rewards* if it stays on-track to finish the race and negative rewards for going off-track. An *episode* starts with the agent somewhere on the race track and finishes when the agent either goes off-track or completes a lap.

> **Note**
> Strictly speaking, the environment state refers to everything relevant to the problem. For example, the vehicle's position on the track as well as the shape of the track. The image fed through 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 than state. For simplicity, we use *state* and *observation* interchangeably throughout this documentation.

Training the agent in a simulated environment has the following advantages:

- The simulation can estimate how much progress the agent has made and identify when it goes off the track to compute a reward.
- The simulation relieves the trainer from tedious chores to reset the vehicle each time it goes off the track, as is done in a physical environment.
- The simulation can speed up training.
- The simulation provides better controls of the environment conditions, e.g. selecting different tracks, backgrounds, and vehicle conditions.

The alternative to reinforcement learning is *supervised learning*, also referred to as *imitation learning*. Here a known dataset (of [image, action] tuples) collected from a given environment is used to train the agent. Models that are trained through imitation learning can be applied to autonomous driving. They work well only when the images from the camera look similar to the images in the training dataset. For robust driving, the training dataset must be comprehensive. In contrast, reinforcement learning does not require such extensive labeling efforts and can be trained entirely in simulation. Because reinforcement learning starts with random actions, the agent learns a variety of environment and track conditions. This makes the trained model robust .

AWS DeepRacer Action Space and Reward Function

For autonomous driving, the AWS DeepRacer vehicle receives input images streamed at 15 frames per second from the front camera. The raw input is downsized to 160x120 pixels in size and converted to grayscale images.

Responding to an input observation, the vehicle reacts with a well-defined action of specific speed and steering angle. The actions are converted to low-level motor controls. The possible actions a vehicle can take is defined by an action space of the dimensions in speed and steering angle. An action space can be discrete or continuous. AWS DeepRacer uses a discrete action space.

For a discrete action space of finite actions, the range is defined by the maximum speed and the absolute value of the maximum steering angles. The granularities define the number of speeds and steering angles the agent has.

For example, the AWS DeepRacer default action space has the following actions you can use to train an AWS DeepRacer model.

The default AWS DeepRacer action space

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

This default action space is characterized by the following ranges and granularities:

The default action space characteristics

Property	Value	
Maximum steering angle	30 degrees	
Steering angle granularity	5	
Maximum speed	0.8 m/s	
Speed granularity	2	

To influence behavior, we can explore a reward function to assign immediate rewards to the actions in this action space. For example, AWS DeepRacer has a basic reward function by default to encourage the

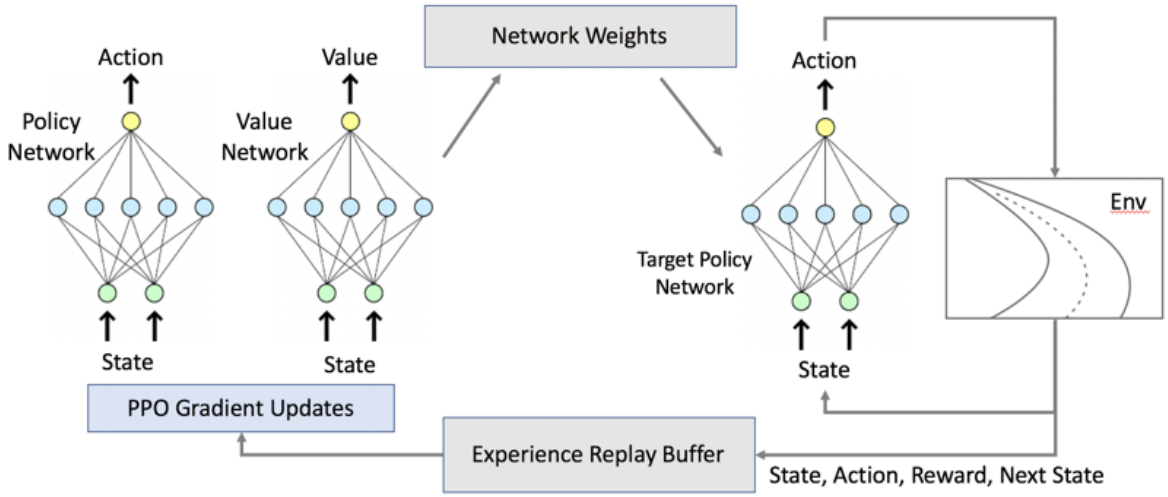
agent to stay as close to the center line as possible. The agent avoids steering close to the edge of the track and going off the track with even a slight turn. For details of this default reward function in the AWS DeepRacer console, see AWS DeepRacer reward function example (p. 73).

In addition to the default action space, you can also explore a custom action space (p. 33) and a custom reward function (p. 31) to train your models.

You can apply a trained model to an AWS DeepRacer vehicle by mapping the maximum speed (*0.8 m/s*) and maximum steering angles (*30 degrees*) used in training to the corresponding maximum physical values. This is called vehicle calibration (p. 94).

AWS DeepRacer Training Algorithm

AWS DeepRacer uses the Proximal Policy Optimization (PPO) algorithm to train the reinforcement learning model. PPO uses two neural networks during training: a policy network and a value network. The policy network (also called actor network) decides which action to take given an image as input. The value network (also called critic network) estimates the cumulative reward we are likely to get given the image as input. Only the policy network interacts with the simulator and gets deployed to the real agent, namely an AWS DeepRacer vehicle.



Below we explain how the actor and critic work together mathematically.

PPO is a derivative of the policy gradient method. In the most basic form, the policy gradient method trains the agent to move along a track by searching for the optimal policy $\#^*(a|s;\ \#^*)$. The optimization aims at maximizing a policy score function $J(\#)$ that can be expressed in terms of the immediate reward $r(s,a)$ of taking action (a) in state (s) averaged over the state probability distribution $\#(s)$ and the action probability distribution ($\#(a|s;\ \#)$):

$$J(\theta) = \sum_{s \in S} \rho(s) \sum_{a \in A} \pi(a|s;\ \theta)\, r(s,a)$$

The optimal policy, as represented by the optimal policy network weights $\#^*$, can be expressed as follows:

$$\theta^* = argmax_\theta(J(\theta))$$

The maximization can proceed by following the policy gradient ascent over episodes of training data `(s, a, r)`:

$$\theta_{\tau+1} = \theta_\tau + \alpha \nabla_\theta J(\theta)$$

where # is known as the learning rate and `#_#J(#_#)` is the policy gradient with respect to # evaluated at step #.

In terms of the total future reward:

$$R(\tau) = R(s_\tau, r_\tau) = \sum_{t=0}^{H} \gamma^t \, r(s_{\tau+1}, r_{\tau+1})$$

where γ is the discount factor ranging between 0 and 1, and τ maps to an *experience* (s_τ, a_τ, r_τ) at step τ, and the summation includes experiences in an episode that starts from time $t = 0$ and ends at time $t = H$ when the agent goes off-track or reaches to the finish line, the score function becomes the expected total future reward averaged over the policy distribution π across many episodes of experiences:

$$J(\theta) = \sum_\tau \pi(\theta, \tau) \, R(\tau)$$

From this definition of `J(#)`, the policy weight updates can be expressed as follows:

$$\nabla_\theta J(\theta_\tau) = \sum_\tau \pi(\theta, \tau) \; \nabla_\theta \log\big(\pi(\theta, \tau)\big) \; R(\tau) \; \approx \; \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{H} \nabla_\theta \log\Big(\pi\big(a_{i,t} | s_{i,t};\, \theta\big)\Big) \; R\big(a_{i,t}, s_{i,t}\big)$$

Here, averaging over π is approximated by sample averaging over N of episodes each of which consists of possibly unequal H number of experiences.

The update rule for a policy network weight then becomes:

$$\Delta\theta_{i,t} = \alpha \, \nabla_\theta \log\Big(\pi\big(a_{i,t} | s_{i,t};\, \theta\big)\Big) \; R\big(a_{i,t}, s_{i,t}\big)$$

The policy gradient method outlined above is of limited utility in practice, because the score function $R(s_{i,t}, a_{i,t})$ has high variance as the agent can take many different paths from a given state. To get around this, one uses a critic network (#) to estimate the score function. To illustrate this, let $V_{\#}(s)$ the value of the critic network, where s describes a state and # the value network weights. To train this value network, the estimated value ($y_{i,t}$) of state s at step t in episode i is estimated to be the immediate reward taking action $a_{i,t}$ at state $s_{i,t}$ plus the discounted total future value of the state s at the next step $t+1$ in the same episode:

$$y_{i,t} \approx r(a_{i,t}, s_{i,t}) + \gamma V_{\phi}(s_{i,t+1})$$

The loss function for the value network weights is:

$$L(\phi) = \frac{1}{2}\sum_{i,t}\left(V_{\phi}(s_{i,t}) - y_{i,t}\right)^2$$

Using the estimated values, the policy gradient for updating the policy network weights # becomes:

$$\nabla_{\theta} J(\theta_{\tau}) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{H} \nabla_{\theta} \log\left(\pi(a_{i,t}|s_{i,t};\, \theta)\right)\left(r(a_{i,t}, s_{i,t}) + \gamma V_{\phi}(s_{i,t+1}) - V_{\phi}(s_{i,t})\right)$$

This formulation changes the policy network weights such that it encourages actions that give higher rewards than prior estimate and discourages otherwise.

Every reinforcement learning algorithm needs to balance between exploration and exploitation. The agent needs to explore the state and action space to learn which actions lead to high rewards in unexplored state space. The agent should also exploit by taking the actions that leads to high rewards so that the model converges to a stable solution. In our algorithm, the policy network outputs the probability of taking each action and during training the action is chosen by sampling from this probability distribution (e.g. an action with probability 0.5 will be chosen half the time). During evaluation, the agent picks with action with the highest probability.

In addition to the above actor-critic framework, PPO uses importance sampling with clipping, adds a Gauss-Markov noise to encourage exploration and uses generalized advantage estimation. To learn more, see the original paper.

AWS DeepRacer Service Architecture

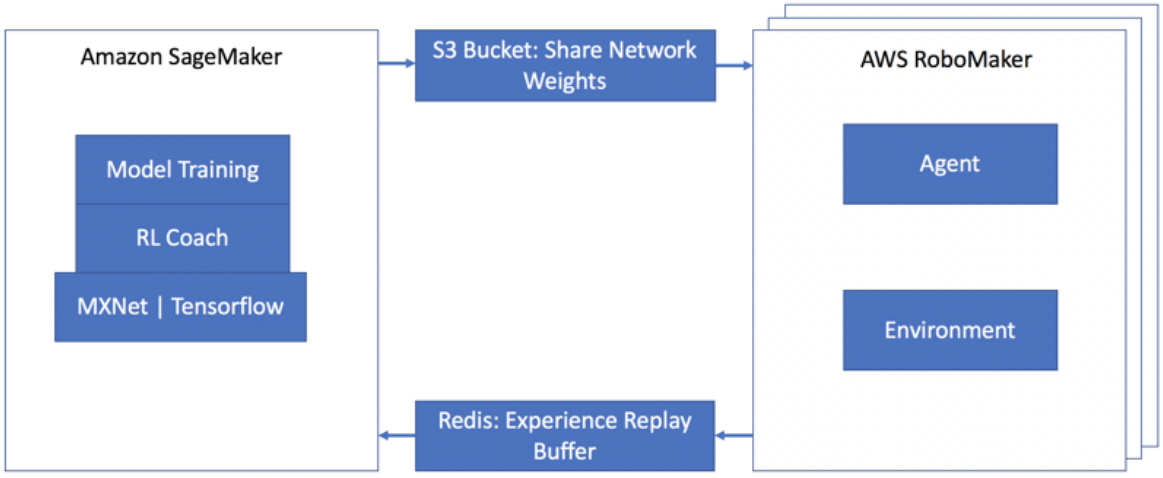
The AWS DeepRacer service is built upon Amazon SageMaker, AWS RoboMaker and other AWS services such as Amazon S3.

Amazon SageMaker is an AWS machine learning platform to train machine learning models in general. AWS DeepRacer uses it to train reinforcement learning model in particular. AWS RoboMaker is a cloud service to develop, test and deploy robotic solutions in general. AWS DeepRacer uses it create the virtual agent and its interactive environment. Amazon S3 is an economical general-purpose cloud storage solution. AWS DeepRacer uses it to store trained model artifacts. In addition, AWS DeepRacer uses Redis, an in-memory database, as an experience buffer to select training data from for training the policy neural network.

Within the AWS DeepRacer architecture, AWS RoboMaker creates a simulated environment for the agent to drive along a specified track. The agent moves according to the policy network model that has been trained up to a certain time in Amazon SageMaker. Each run starts from the starting line to an end state which can be the finishing line or off the track and is known as an episode. For each episode, the course is divided into segments of a fixed number of steps. In each segment experiences, defined as an ordered list of the tuples of (*state*, *action*, *reward*, *new state*) associated with individual steps, are cached in Redis as an experience buffer. Amazon SageMaker then randomly draws from the experience buffer training data in batches and feeds the input data to the neural network to update the weights. It then stores the updated model in Amazon S3 for Amazon SageMaker to use in order to generate more experiences. The cycle continues until training stops.

In the beginning before the first model is trained for the first time, Amazon SageMaker initializes the experience buffer with random actions.

The following diagram illustrates this architecture.



This setup allows running multiple simulations to train a model on multiple segments of a single track at the same time or to train the model for multiple tracks simultaneously.

AWS DeepRacer Solution Workflow

Training an AWS DeepRacer model involves the following general tasks:

1. The AWS DeepRacer service initializes the simulation with a virtual track, an agent representing the vehicle, and the background. The agent embodies a policy neural network that can be tuned with hyper-parameters as defined in the PPO algorithm (p. 10).
2. The agent acts (as specified with a steering angle and a speed) based on a given state (represented by an image from the front camera).
3. The simulated environment updates the agent's position based on the agent action and returns a reward and an updated camera image. The experiences collected in the form of state, action, reward, and new state are used to update the neural network periodically. The updated network models are used to create more experiences.
4. You can monitor the training in progress along the simulated track with a first-person view as seen by the agent. You can display metrics such as rewards per episode, the loss function value, the entropy of the policy. CPU or memory utilization can also be displayed as training progresses. In addition, detailed logs are recorded for analysis and debugging.
5. The AWS DeepRacer service periodically saves the neural network model to persistent storage.
6. The training stops based on a time limit.

7. You can evaluate the trained model in a simulator. To do this submit the trained model for time trials for a selected number runs on the selected track.

After the model is successfully trained and evaluated, it can be uploaded to a physical agent (an AWS DeepRacer vehicle). The process involves the following steps:

1. Download the trained model from its persistent storage (an Amazon S3 bucket).
2. Use the vehicle's device control console to upload the trained model to the vehicle. Use the console to calibrate the vehicle for mapping the simulated action space to the physical action space. You can also use the console to check the throttling parity, view the front camera feed, load a model into the inference engine, and watch the vehicle driving on a real track.

   The vehicle's device control console is a web server hosted on the vehicle's compute module. The console is accessible from the vehicle IP address with a connected Wi-Fi network and a web browser on a computer or a mobile device.
3. Experiment with the vehicle driving under different lighting, battery levels, and surface textures and colors.

   The vehicle's performance in a physical environment may not match the performance in a simulated environment due to model limitations or insufficient training. The phenomenon is referred to as the *sim2real* performance gap. To reduce the gap, see the section called “Simulated-to-Real Performance Gaps” (p. 14).

Simulated-to-Real Performance Gaps

Because the simulation cannot capture all aspects of the real world accurately, the models trained in simulation may not work well in the real world. Such discrepancies are often referred to as simulated-to-real (*sim2real*) performance gaps.

Efforts have been made in AWS DeepRacer to minimize the *sim2real* performance gap. For example, the simulated agent is programmed to take about 10 actions per second. This matches the frequency the AWS DeepRacer vehicle runs inference with, about 10 inferences per second. As another example, at the start of each episode in training, the agent's position is randomized. This maximizes the likelihood that the agent learns all parts of the track evenly.

To help reduce *real2sim* performance gaps, make sure to use the same or similar color, shape and dimensions for both the simulated and real tracks. To reduce visual distractions, use barricades around the real track. Also, carefully calibrate the ranges of the vehicle's speed and steering angles so that the action space used in training matches the real world. Evaluating model performance in a different simulation track than the one used in training can show the extent of the *real2real* performance gap.

For more information about how to reduce the *sim2real* gap when training an AWS DeepRacer model, see the section called “Optimize Training for Real Environments” (p. 41).

Get Started with AWS DeepRacer

To get started with AWS DeepRacer, let's first walk through the steps to use the AWS DeepRacer console to configure an agent with appropriate sensors for your autonomous driving requirements, to train a reinforcement learning model for the agent with the specified sensors, and to evaluate the trained model to ascertain the quality of the model.

Topics

- Set Up Account Resources for AWS DeepRacer (p. 15)
- Train Your First AWS DeepRacer Model (p. 15)
- Evaluate Your AWS DeepRacer Models in Simulation (p. 20)

Set Up Account Resources for AWS DeepRacer

To use AWS DeepRacer console and other AWS services, you need an AWS account. If you don't have an account, visit aws.amazon.com and choose **Create an AWS Account**. For detailed instructions, see Create and Activate an AWS Account.

As a best practice, you should also create an AWS Identity and Access Management (IAM) user with administrator permissions and use that for all work that does not require root credentials. Create a password for console access, and access keys to use command line tools. For instructions, see Creating Your First IAM Admin User and Group in the *IAM User Guide*.

After you've created your AWS account, sign in to the AWS DeepRacer console, with your AWS account or user credentials. Then, follow the steps below to have the required AWS resources created.

To create the required resources for your account

1. From the main navigation pane on the console, choose **Get started with reinforcement learning**.
2. On the **Get started with reinforcement learning** page, under **Step 0: Create account resources**, choose **Create resources**.

You need to create account resources one time only. Later, you can choose **Reset resources** to sync up any resource updates.

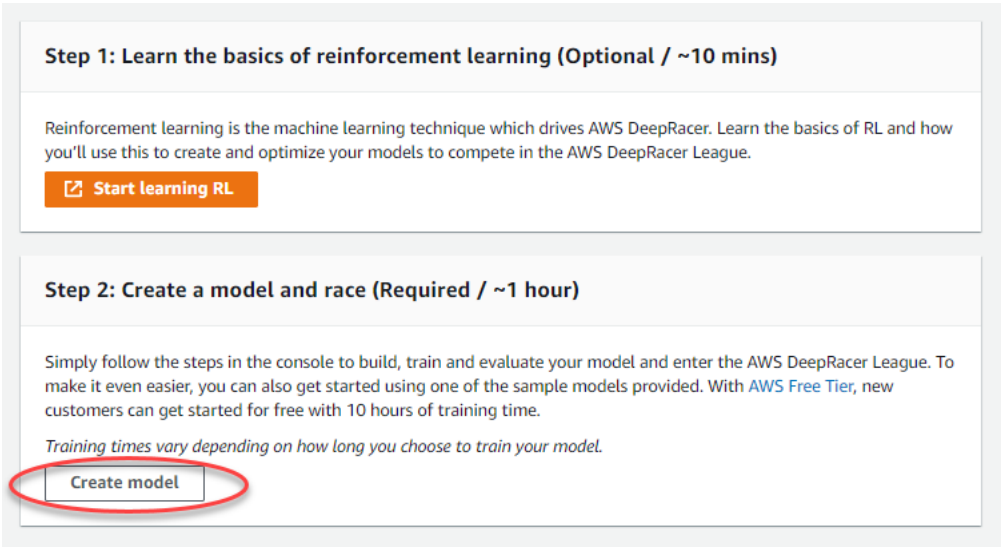
You're now ready to configure an agent for training a AWS DeepRacer model. The topics in this section assume that you've signed into the AWS DeepRacer console.

Train Your First AWS DeepRacer Model

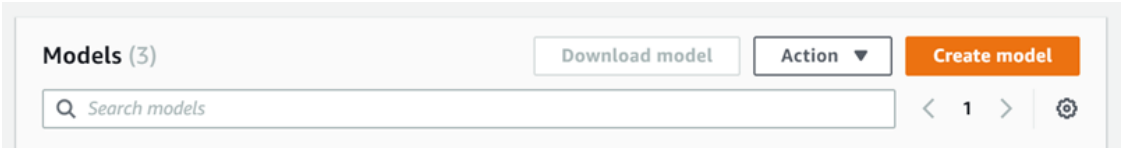
To get started quickly using AWS DeepRacer to explore reinforcement learning and its application to autonomous driving, we'll walk you through how to train your first model using the AWS DeepRacer console.

To train a reinforcement learning model using the AWS DeepRacer console

1. If this is your first time using AWS DeepRacer, choose **Get started** from the service landing page or choose **Get started with reinforcement learning** from the main navigation pane.
2. On the **Get started with reinforcement learning** page, under **Step 2: Create a model and race**, choose **Create model**.



Alternatively, on the AWS DeepRacer home page, choose **Your models** from the main navigation pane to open the **Your models** page. On the **Your models** page, choose **Create model**.

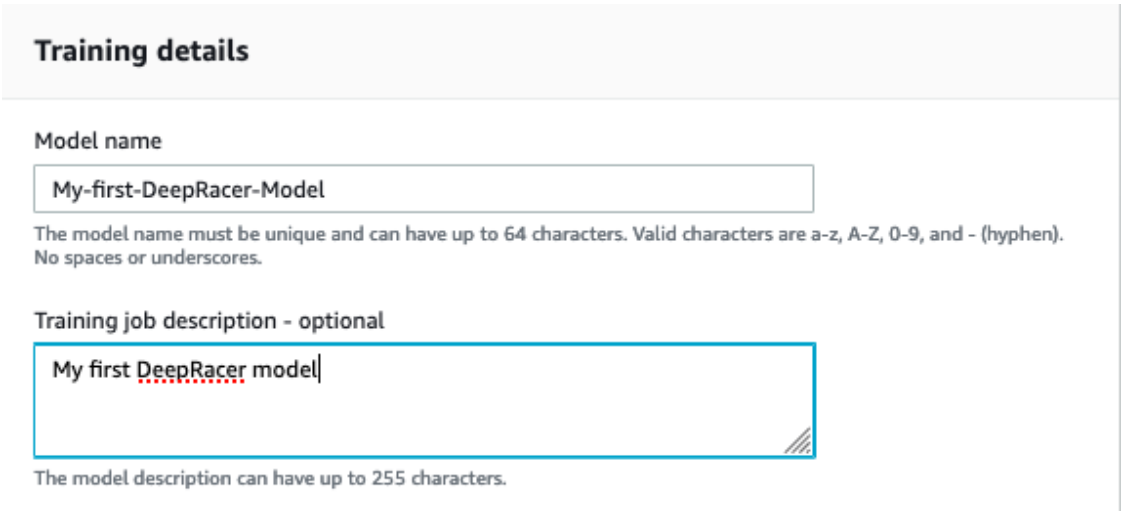


3. On the **Create model** page, under **Account resources**, if you don't have the required account resources, choose **Create resources**. If there is any issue with the account resources, choose **Reset resources**.

   For more information about the required IAM roles and policies, see Required IAM Roles for AWS DeepRacer to Call Dependent AWS Services (p. 132).

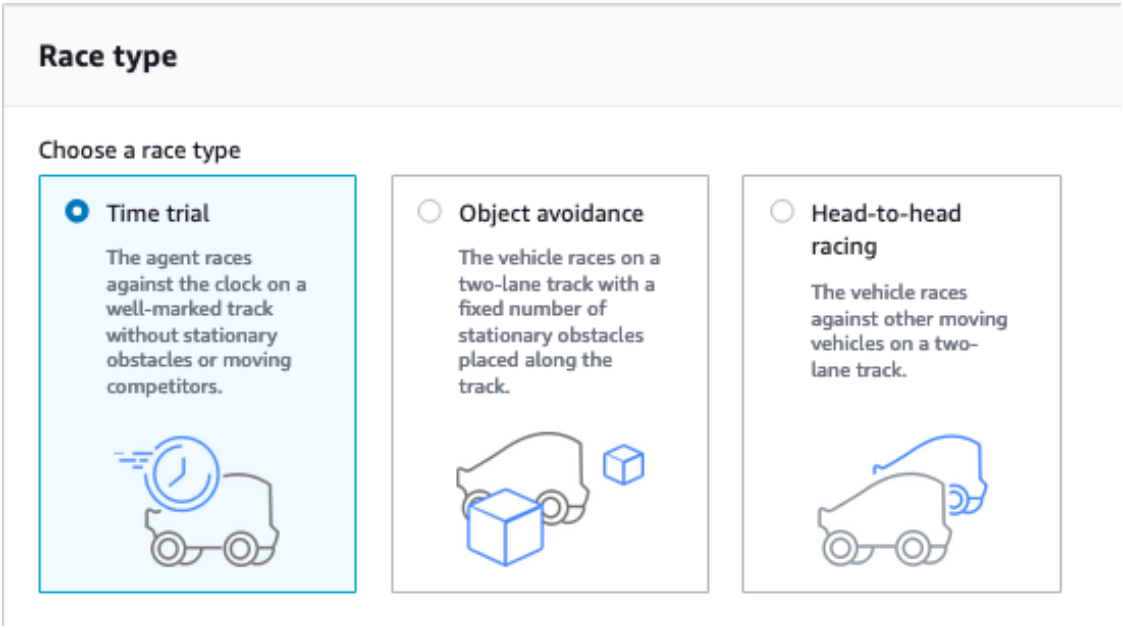
4. On the **Create model** page, under **Training details**, type a name for the model in **Model name** and, optionally, provide a summary description of the model in **Training job description**.

   You'll use the model name to reference this model when submitting it to a leaderboard of an AWS-sponsored or community-organized racing event or when cloning to continue the training.



5. On the **Create model** page, under **Environment simulation**, choose a track as a virtual environment to train your AWS DeepRacer agent. Then, choose **Next**.

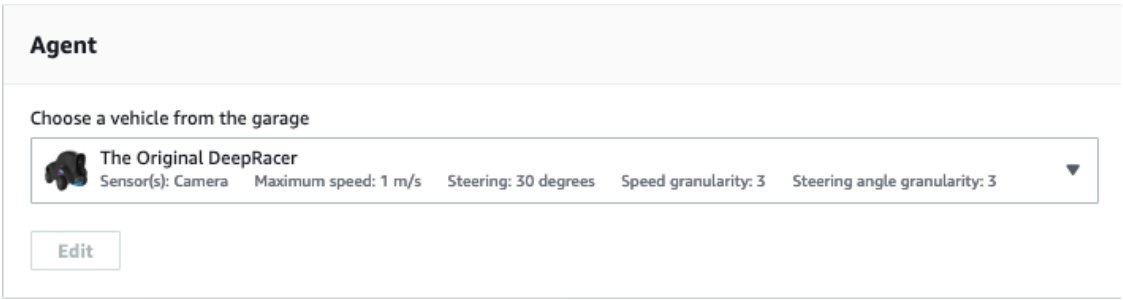
   For your first run, choose a track with a simple shape and smooth turns. In later iterations, you can choose more complex tracks to progressively improve your models. To train a model for a particular racing event, choose the track most similar to the event track.
6. On the **Create model** page, choose **Next**.
7. On the **Create Model** page, under **Race type**, choose a training type.



   For your first run, choose **Time trial**. The agent with the default sensor configuration with a single-lens camera is suitable for this type of racing without modifications. For more information, see the section called “Tailor Training for Time Trials” (p. 28).

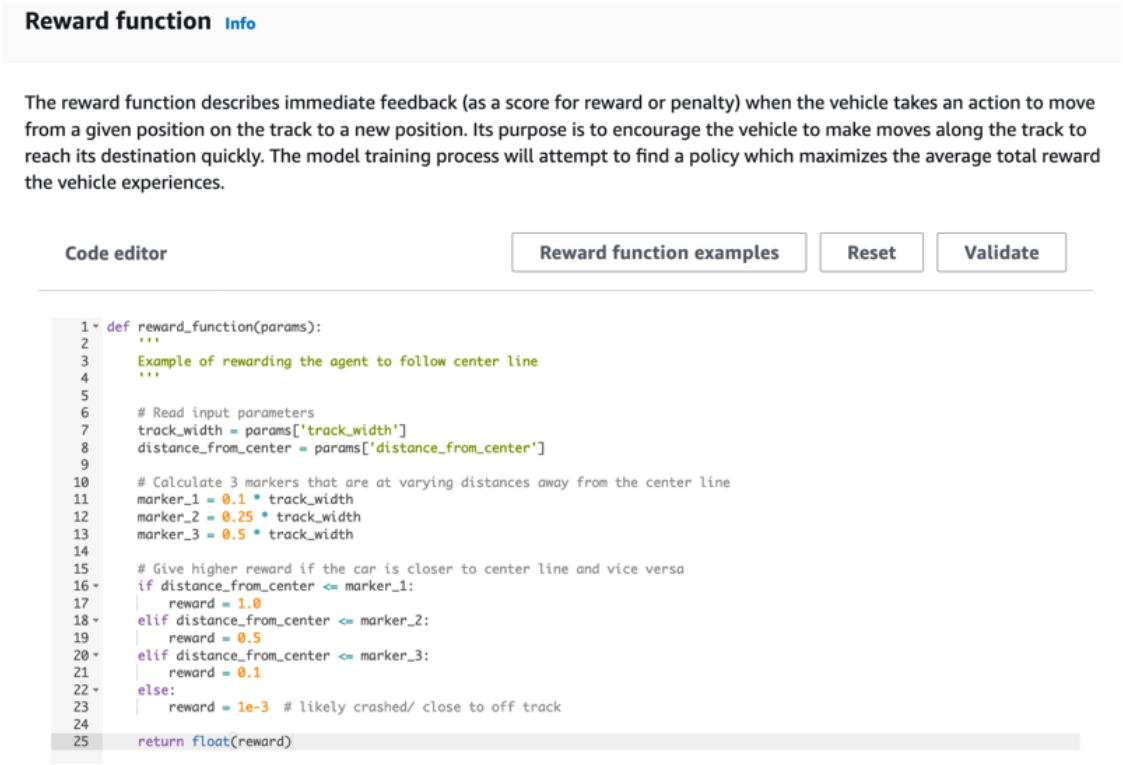
   For later runs, you can choose **Object avoidance** to go around stationary obstacles placed at fixed or random locations along the chosen track. The agent should be configured with at least a double-lens front-facing stereo camera for such applications, although a single-lens front-facing camera can be used for avoiding stationary obstacles on fixed locations. For more information, see the section called “Tailor Training for Object Avoidance Races” (p. 29).

   For more ambitious runs, choose **Head-to-head racing** to race against up to 4 bot vehicles moving at a constant speed. In addition to either a single-lens camera or a stereo camera, the agent should be configured with a LiDAR unit to enable detecting and avoiding blind spots while passing other moving vehicles or stationary obstacles. For more information, see the section called “Tailor Training for Head-to-Head Races” (p. 30).
8. On the **Create model** page, under **Agent**, choose **The Original DeepRacer** for your first model.



The **Edit** button is unavailable because the default agent is not configurable. For a custom agent, the **Edit** option will be available for you to modify the agent configuration to meet the racing criteria for the chosen race type.

9. On the **Create model** page, choose **Next**.
10. On the **Create model** page, under **Reward function**, use the default reward function example as-is for your first model.



Later on, you can choose **Reward function examples** to select another example function and then choose **Use code** to accept the selected reward function.

There are four example functions you can start with. They illustrate how to follow the track center (default), how to keep the agent inside the track borders, and how to prevent zig-zag driving, and how to avoid crashing into stationary obstacles or other moving vehicles.

To learn more about the reward function, see the section called “Reward Function Reference” (p. 58).

11. On the **Create model** page, under **Training algorithm and hyperparameters**, use the default hyperparameter values as-is.

    Later on, to improve training performance, expand **Hyperparameters** and modify the default hyperparameter values as follows:

    a. For **Gradient descent batch size**, choose available options (p. 34).
    b. For **Number of epochs**, set a valid value (p. 34).
    c. For **Learning rate**, set a valid value (p. 34).
    d. For **Entropy**, set a valid value (p. 34).
    e. For **Discount factor**, set a valid value (p. 34).
    f. For **Loss type**, choose available options (p. 34).

g. For **Number of experience episodes between each policy-updating iteration**, set a valid value (p. 34).

For more information about hyperparameters, see Systematically Tune Hyperparameters (p. 34).

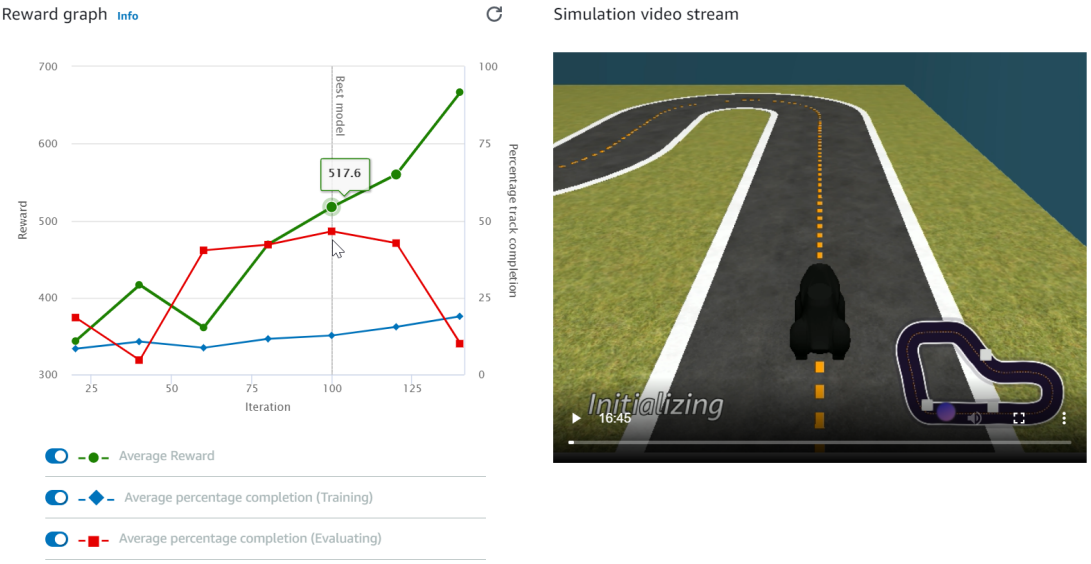
12. On the **Create model** page, under **Stop conditions**, leave the default **Maximum time** value as-is or set a new value to terminate the training job, to help prevent long-running (and possible run-away) training jobs.

    When experimenting in the early phase of training, you should start with a small value for this parameter and then progressively train for longer amounts of time.

13. On the **Create model** page, choose **Create model** to start creating the model and provisioning the training job instance.
14. After the submission, watch your training job being initialized and then run.

    The initialization process takes about 6 minutes to change status from **Initializing** to **In progress**.

15. Watch the **Reward graph** and **Simulation video stream** to observe the progress of your training job. You can choose the refresh button next to **Reward graph** periodically to refresh the **Reward graph** until the training job is complete.



The training job is running on the AWS Cloud, so you don't need to keep the AWS DeepRacer console open during training. However, you can come back to the console to check on your model at any point while the job is in progress.

If the **Simulation video stream** window or the **Reward graph** display becomes unresponsive, refresh the browser page to get the training progress updated.

After the training job stops, you can proceed to evaluate the model trained thus far. To do so, follow the next steps (p. 20).

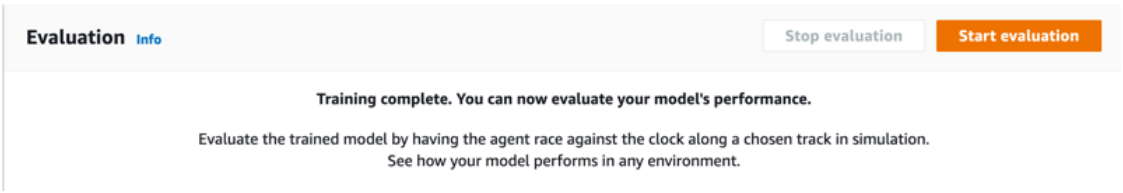
Evaluate Your AWS DeepRacer Models in Simulation

After your training job is complete, you should evaluate the trained model to assess its convergency behavior. The evaluation proceeds by completing a number of trials on a chosen track and having the agent move on the track according to likely actions inferred by the trained model. The performance metrics include a percentage of track completion and the time running on each track from start to finish or going off-track.

To evaluate your trained model, you can use the AWS DeepRacer console. To do so, follow the steps in this topic.

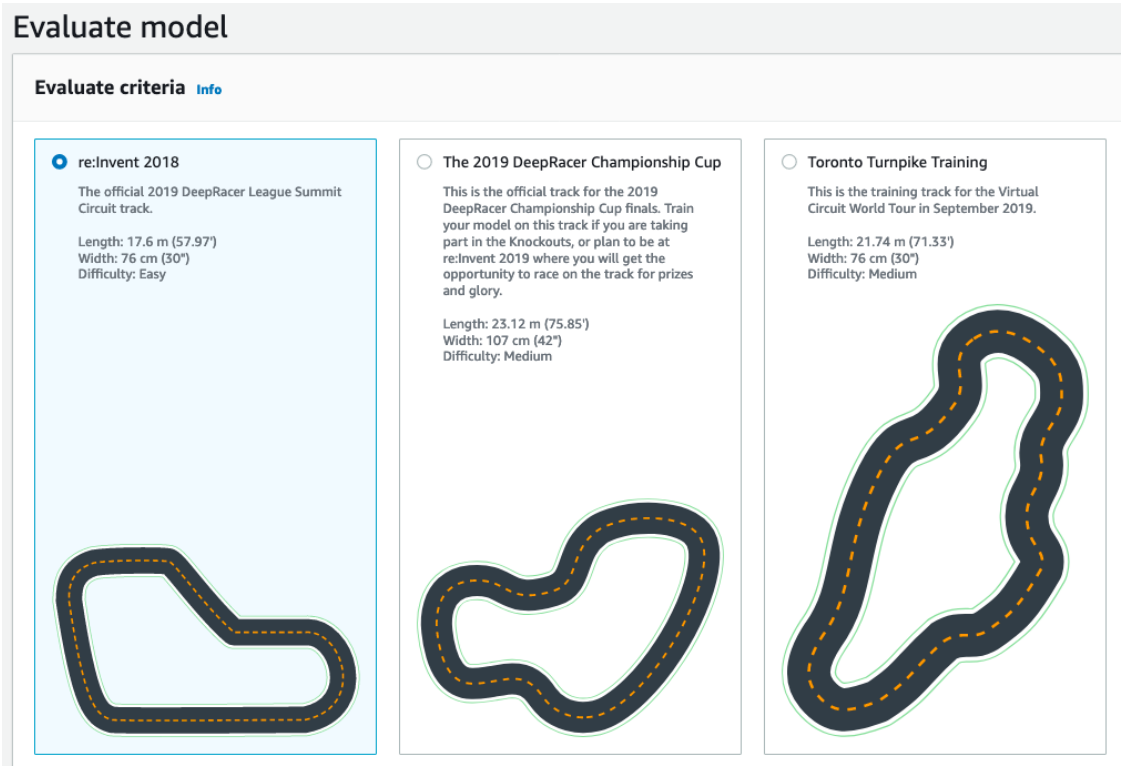
To evaluate a trained model in the AWS DeepRacer console

1. Open the AWS DeepRacer console at https://console.aws.amazon.com/deepracer.
2. From the main navigation pane, choose **Models** and then choose the model you just trained from the **Models** list to open the model details page.
3. In **Evaluation**, choose **Start evaluation**.



   You can start an evaluation after your training job status changes to **Completed** or the model's status changes to **Ready** if the training job wasn't completed.

   A model is ready when the training job is complete. If the training wasn't completed, the model can also be in a **Ready** state if it's trained up to the failing point.
4. On the **Evaluate model** page, under **Evaluate criteria**, choose a track under **Evaluation criteria**.

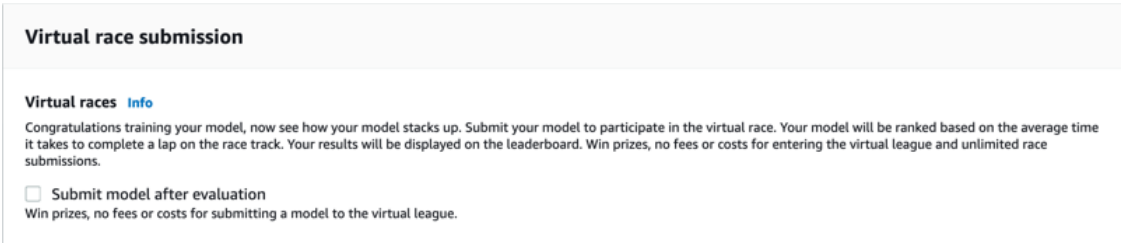


Typically, you want to choose a track that is the same as or similar to the one you used in training the model (p. 15). You can choose any track for evaluating your model, however, you can expect the best performance on a track most similar to the one used in training.

To see if your model generalizes well, choose an evaluation track different from the one used in training.

5. On the **Evaluate model** page, under **Evaluate criteria**, choose the number of trials you want to use to evaluate the model.
6. On the **Evaluate model** page, under **Race type**, choose the racing type that you chose to train the model.

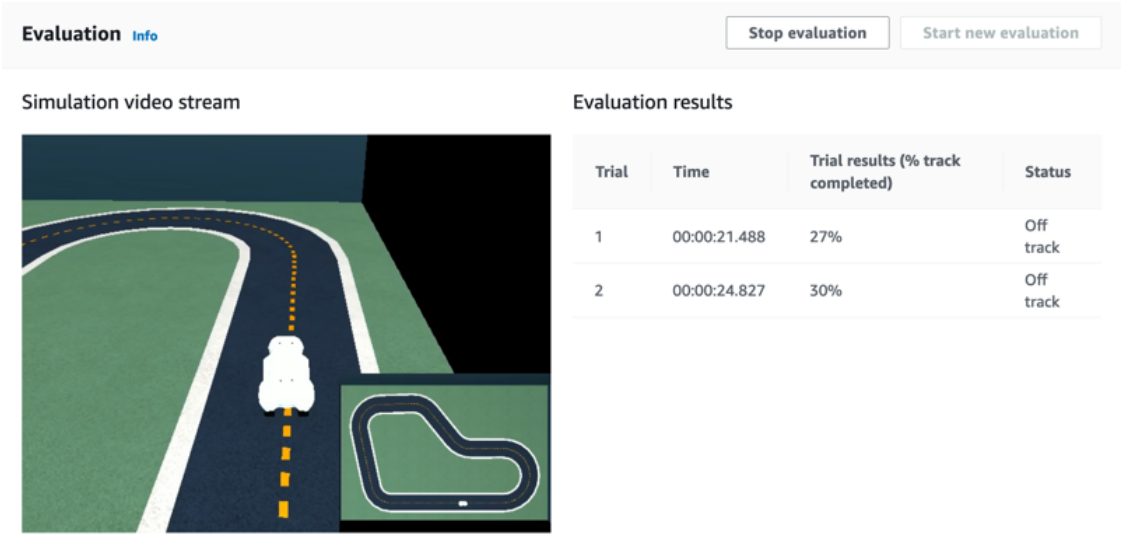
   For evaluation you can choose a race type different from the race type used in training. For example, you can train a model for head-to-head races and then evaluate it for time trials. In general, the model must generalize well if the training race type differs from the evaluation race type. For your first run, you should use the same race type for both evaluation and training.
7. On the **Evaluate model** page, under **Virtual Race Submission**, for your first model, turn off the **Submit model after evaluation** option. Later, if you want to participate in a racing event, leave this option enabled.



8. On the **Evaluate model** page, choose **Start evaluation** to start creating and initializing the evaluation job.

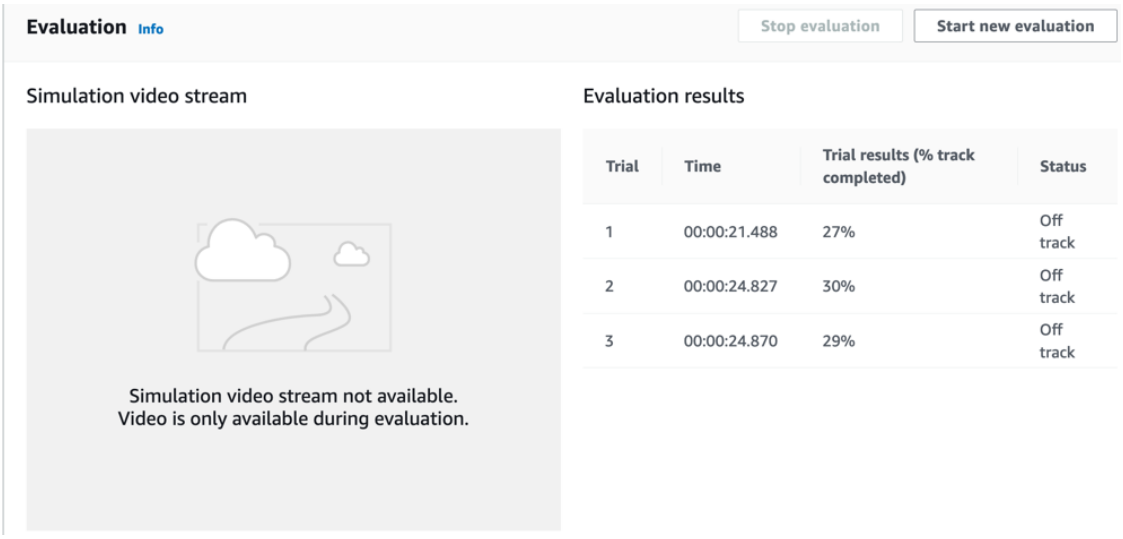
This initialization process takes about 3 minutes to complete.

9. As the evaluation progresses, the evaluation results, including the trial time and track completion rate, are displayed under **Evaluation** after each trial. In the **Simulation video stream** window, you can watch how the agent performs on the chosen track.



You can stop an evaluation job before it completes. To stop the evaluation job, choose **Stop evaluation** on the upper-right corner of the **Evaluation** card and then confirm to stop the evaluation.

10. After the evaluation job is complete, examine the performance metrics of all the trials under **Evaluation results**. The accompanying simulation video stream is no longer available.



For this particular evaluation job, the trained model fails to complete any trial. As the first run, this is not unusual. Possible reasons include that the training didn't converge and the training needs more time, the action space needs to be enlarged to give the agent more room to react, or the reward function needs to be updated to handle varying environments.

You can continue to improve the model by cloning a previously trained one, modifying the reward function, tuning hyperparameters, and then iterating the process until the total reward converges

and the performance metrics improve. For more information on how to improve the training, see *Train and Evaluate Models* (p. 24).

To transfer your completely trained model to your AWS DeepRacer vehicle for driving in a physical environment, you need to download the model artifacts. To do so, choose **Download model** on the model's details page. If your AWS DeepRacer physical vehicle doesn't support new sensors and your model has been trained with the new sensor types, you'll get an error message when you use the model on your AWS DeepRacer vehicle in a real-world environment. For more information about testing an AWS DeepRacer model with a physical agent, see *Operate Your Vehicle* (p. 77).

If you've trained your model on a track identical or similar to the one specified in an AWS DeepRacer League racing event or an AWS DeepRacer community race, you can submit the model to the virtual races in the AWS DeepRacer console. To do this, follow **Official DeepRacer virtual circuit** or **Community races** on the main navigation pane. For more information, see *Participate in Virtual Races* (p. 119).

To train a model for obstacle avoidance or head-to-head racing, you may need to add new sensors to the agent and the physical vehicle. For more information, see the section called “Understanding Racing Types and Enabling Sensors” (p. 25).

Train and Evaluate AWS DeepRacer Models

When your AWS DeepRacer vehicle drives itself along a track, it captures environmental states with the camera mounted on the front and takes actions in response to the observations. Your AWS DeepRacer model is a function that maps the observations and actions to the expected reward. To train your model is to find or learn the function that maximize the expected reward so that the optimized model prescribes what actions (speed and steering angle pairs) your vehicle can take to move itself along the track from start to finish.

In practice, the function is represented by a neural network and training the network involves finding the optimal network weights given sequences of observed environmental states and the responding vehicle's actions. The underlying criteria of optimality are described by the model's reward function that encourages the vehicle to make legal and productive moves without causing traffic accidents or infractions. A simple reward function could return a reward of 0 if the vehicle is on the track, -1 if it's off the track, and +1 if it reaches the finish line. With this reward function, the vehicle gets penalized for going off the track and rewarded for reaching the destination. This can be a good reward function if time or speed is not an issue.

Suppose that you're interested in having the vehicle drive as fast as it can without getting off a straight track. As the vehicle speeds up and down, the vehicle may steer left or right to avoid obstacles or to remain inside. Making too big a turn at a high speed could easily lead the vehicle off the track. Making too small a turn may not help avoid colliding with an obstacle or another vehicle. Generally speaking, optimal actions would be to make a bigger turn at a lower speed or to steer less along a sharper curve. To encourage this behavior, your reward function must assign a positive score to reward smaller turns at a higher speed and/or a negative score to punish bigger turns at a higher speed. Similarly, the reward function can return a positive reward for speeding up along a straighter course or speeding down when it's near an obstacle.

The reward function is an important part of your AWS DeepRacer model. You must provide it when training your AWS DeepRacer model. The training involves repeated episodes along the track from start to end. In an episode the agent interacts with the track to learn the optimal course of actions by maximizing the expected cumulative reward. At the end, the training produces a reinforcement learning model. After the training, the agent executes autonomous driving by running inference on the model to take an optimal action in any given state. This can be done in either the simulated environment with a virtual agent or a real-world environment with a physical agent, such as an AWS DeepRacer scale vehicle.

To train a reinforcement learning model in practice, you must choose a learning algorithm. Currently, the AWS DeepRacer console supports only the proximal policy optimization (PPO) algorithm for faster training performances. You can then choose a deep-learning framework supporting the chosen algorithm, unless you want to write one from scratch. AWS DeepRacer integrates with Amazon SageMaker to make some popular deep-learning frameworks, such as TensorFlow, readily available in the AWS DeepRacer console. Using a framework simplifies configuring and executing training jobs and lets you focus on creating and enhancing reward functions specific to your problems.

Training a reinforcement learning model is an iterative process. First, it's challenging to define a reward function to cover all important behaviors of an agent in an environment at once. Second, hyperparameters are often tuned to ensure satisfactory training performance. Both require experimentation. A prudent approach is to start with a simple reward function and then progressively enhance it. AWS DeepRacer facilitates this iterative process by enabling you to clone a trained model

and then use it to jump-start the next round of training. At each iteration you can introduce one or a few more sophisticated treatments to the reward function to handle previously ignored variables or you can systematically adjust hyperparameters until the result converges.

As with general practice in machine learning, you must evaluate a trained reinforcement learning model to ascertain its efficacy before deploying it to a physical agent for running inference in a real-world situation. For autonomous driving, the evaluation can be based on how often a vehicle stays on a given track from start to finish or how fast it can finish the course without getting off the track. The AWS DeepRacer simulation runs in the AWS RoboMaker simulator and lets you run the evaluation and post the performance metrics for comparison with models trained by other AWS DeepRacer users on a leaderboard (p. 119).

Topics

- Understanding Racing Types and Enabling Sensors Supported by AWS DeepRacer (p. 25)
- Train and Evaluate AWS DeepRacer Models Using the AWS DeepRacer Console (p. 31)
- Train and Evaluate AWS DeepRacer Models Using Amazon SageMaker Notebooks (p. 43)
- AWS DeepRacer Reward Function Reference (p. 58)

Understanding Racing Types and Enabling Sensors Supported by AWS DeepRacer

In AWS DeepRacer League, you can participate in the following types of racing events:

- **Time trial**: race against the clock on an unobstructed track and aim to get the fastest lap time possible.
- **Object avoidance**: race against the clock on a track with stationary obstacles and aim to get the fastest lap time possible.
- **Head-to-head racing**: race against one or more other vehicles on the same track and aim to cross the finish line before other vehicles.

AWS DeepRacer community races currently supports time trials only.

You should experiment with different sensors on your AWS DeepRacer vehicle to provide it with sufficient capabilities to observe its surroundings for a given race type. The next section describes the AWS DeepRacer-supported sensors (p. 25) that can enable the supported types of autonomous racing events.

Topics

- Choose Sensors for AWS DeepRacer Racing Types (p. 25)
- Configure Agent for Training AWS DeepRacer Models (p. 27)
- Tailor AWS DeepRacer Training for Time Trials (p. 28)
- Tailor AWS DeepRacer Training for Object Avoidance Races (p. 29)
- Tailor AWS DeepRacer Training for Head-to-Head Races (p. 30)

Choose Sensors for AWS DeepRacer Racing Types

Your AWS DeepRacer vehicle comes with a front-facing monocular camera as the default sensor. You can add another front-facing monocular camera to make front-facing stereo cameras or to supplement either the monocular camera or stereo cameras with a LiDAR unit.

The following list summarizes the functional capabilities of AWS DeepRacer-supported sensors, together with brief cost-and-benefit analyses:

Front-facing camera

A single-lens front-facing camera can capture images of the environment in front of the host vehicle, including track borders and shapes. It's the least expensive sensor and is suitable to handle simpler autonomous driving tasks, such as obstacle-free time trials on well-marked tracks. With proper training, it can avoid stationary obstacles on fixed locations on the track. However, the obstacle location information is built into the trained model and, as the result, the model is likely to be overfitted and may not generalize to other obstacle placements. With stationary objects placed at random locations or other moving vehicles on the track, the model is unlikely to converge.

In the real world, the AWS DeepRacer vehicle comes with a single-lens front-facing camera as the default sensor. The camera has 120-degree wide angle lens and captures RGB images that are then converted to grey-scale images of 160 x 120 pixels at 15 frames per second (fps). These sensor properties are preserved in the simulator to maximize the chance that the trained model transfers well from simulation to the real world.

Front-facing stereo camera

A stereo camera has two or more lenses that capture images with the same resolution and frequency. Images from the both lens are used to determine the depth of observed objects. The depth information from a stereo camera is valuable for the host vehicle to avoid crashing into the obstacles or other vehicles in the front, especially under more dynamic environment. However, added depth information makes trainings to converge more slowly.

On the AWS DeepRacer physical vehicle, the double-lens stereo camera is constructed by adding another single-lens camera and mounting each camera on the left and right sides of the vehicle. The AWS DeepRacer software synchronizes image captures from both cameras. The captured images are converted into greyscale, stacked, and fed into the neural network for inferencing. The same mechanism is duplicated in the simulator in order to train the model to generalize well to a real-world environment.

LiDAR sensor

A LiDAR sensor uses rotating lasers to send out pulses of light outside the visible spectrum and time how long it takes each pulse to return. The direction of and distance to the objects that a specific pulse hits are recorded as a point in a large 3D map centered around the LiDAR unit.

For example, LiDAR helps detect blind spots of the host vehicle to avoid collisions while the vehicle changes lanes. By combining LiDAR with mono or stereo cameras, you enable the host vehicle to capture sufficient information to take appropriate actions. However, a LiDAR sensor costs more compared to cameras. The neural network must learn how to interpret the LiDAR data. Thus, trainings will take longer to converge.

On the AWS DeepRacer physical vehicle a LiDAR sensor is mounted on the rear and tilted down by 6 degrees. It rotates at the angular velocity of 10 rotations per second and has a range of 15cm to 2m. It can detect objects behind and beside the host vehicle as well as tall objects unobstructed by the vehicle parts in the front. The angel and range are chosen to make the LiDAR unit less susceptible to environmental noise.

You can configure your AWS DeepRacer vehicle with the following combination of the supported sensors:

- Front-facing single-lens camera only.

  This configuration is good for time trials, as well as obstacle avoidance with objects at fixed locations.
- Front-facing stereo camera only.

  This configuration is good for obstacle avoidance with objects at fixed or random loations.

- Front-facing single-lens camera with LiDAR.

  This configuration is good for obstacle avoidance or head-to-head racing.
- Front-facing stereo camera with LiDAR.

  This configuration is good for obstacle avoidance or head-to-head racing, but probably not most economical for time trials.

As you add more sensors to make your AWS DeepRacer vehicle to go from time trials to object avoidance to head-to-head racing, the vehicle collects more data about the environment to feed into the underlying neural network in training. This makes training more challenging because the model is required to handle increased complexities. In the end, your tasks of learning to train models become more demanding.

To learn progressively, you should start training for time trials first before moving on to object avoidance and then to head-to-head racing. You'll find more detailed recommendations in the next section.

Configure Agent for Training AWS DeepRacer Models

To train a reinforcement learning model for AWS DeepRacer vehicle to race in obstacle avoidance or head-to-head racing, you need to configure the agent with appropriate sensors. For simple time trials, you could use the default agent configured with a single-lens camera. In configuring the agent you can customize the action space and choose a neural network topology so that they work better with the selected sensors to meet the intended driving requirements. In addition, you can change the agent's appearance for visual identification during training.

After you configure it, the agent configuration is recorded as part of the model's metadata for training and evaluation. For evaluation, the agent automatically retrieves the recorded configuration to use the specified sensors, action space, and neural network technology.

This section walks you through the steps to configure an agent in the AWS DeepRacer console.

To configure an AWS DeepRacer agent in the AWS DeepRacer console

1. Sign in to the AWS DeepRacer console.
2. On the primary navigation pane, choose **Garage**.
3. For the first time you use **Garage**, you're presented with the **WELCOME TO THE GARAGE** dialog box. Choose **>** or **<** browse through the introduction to various sensors supported for the AWS DeepRacer vehicle or choose **X** to close the dialog box. You can find this introductory information on the help panel in **Garage**.
4. On the **Garage** page, choose **Build new vehicle**.
5. On the **Mod your own vehicle** page, under **Mod specifications**, choose one or more sensors to try and learn the best combination that can meet your intended racing types.

   To train for your AWS DeepRacer vehicle time trials, choose **Camera**. For obstacle avoidance or head-to-head racing, you want to use other sensor types. To choose **Stereo camera**, make sure you have acquired an additional single-lens camera. AWS DeepRacer makes the stereo camera out two single-lens cameras. You can have either a single-lens camera or a double-lens stereo cameras on one vehicle. In either case, you can add a LiDAR sensor to the agent if you just want the trained model to be able to detect and avoid blind spots in obstacle avoidance or head-to-head racing.
6. On the **Garage** page and under **Neural network topologies**, choose a supported network topology.

   In general, a deeper neural network (with more layers) is more suitable for driving on more complicated tracks with sharp curves and numerous turns, for racing to avoid stationary obstacles, or for competing against other moving vehicles. But a deeper neural network is more costly to train

and the model takes longer to converge. On the other hand, a shallower network (with fewer layers) costs less and takes a shorter time to train. The trained model is capable of handling simpler track conditions or driving requirements, such as time trials on a obstacle-free track without competitors.

Specifically, AWS DeepRacer supports **3-layer CNN** or **5-layer CNN**.

7. On the **Garage** page, choose **Next** to proceed to setting up the agent's action space.
8. On the **Action space** page, leave the default settings for your first training. For subsequent trainings, experiment with different settings for the steering angle, top speed, and their granularities. Then, choose **Next**.
9. On the **Color your vehicle to stand out in the crowd** page, enter a name in **Name your DeepRacer** and then choose a color for the agent from the **Vehicle color** list. Then, choose **Submit**.
10. On the **Garage** page, examine the settings of the new agent. To make further modifications, choose **Mod vehicle** and repeat the previous steps starting at **Step 4**.

Now, your agent is ready for training.

Tailor AWS DeepRacer Training for Time Trials

If this is your first time to use AWS DeepRacer, you should start with a simple time trial to become familiar with how to train AWS DeepRacer models to drive your vehicle. This way, you get a gentler introduction to basic concepts of reward function, agent, environment, etc. Your goal is to train a model to make the vehicle stay on the track and finish a lap as fast as possible. You can then deploy the trained model to your AWS DeepRacer vehicle to test driving on a physical track without any additional sensors.

To train a model for this scenario, you can choose the default agent from **Garage** on the AWS DeepRacer console. The default agent has been configured with a single front-facing camera, a default action space and a default neural network topology. It is helpful to start training an AWS DeepRacer model with the default agent before moving on to more sophisticated ones.

To train your model with the default agent, follow the recommendations below.

1. Start training your model with a simple track of more regular shapes and of less sharp turns. Use the default reward function. And train the model for 30 minutes. After the training job is completed, evaluate your model on the same track to watch if the agent can finish a lap.
2. Read about the reward function parameters (p. 58). Continue the training with different incentives to reward the agent to go faster. Lengthen the training time for the next model to 1 - 2 hours. Compare the reward graph between the first training and this second one. Keep experimenting until the reward graph stops improving.
3. Read more about action space (p. 9). Train the model the 3rd time by increasing the top speed (e.g., 1 m/s). To modify the action space, you must build in **Garage** a new agent, when you get the chance to make the modification. When updating the top speed of your agent, be aware of that the higher the top speed, the faster the agent can complete the track in evaluation and the faster your AWS DeepRacer vehicle can finish a lap on a physical track. However, a higher top speed often means a longer time for the training to converge because the agent is more likely to overshoot on a curve and thus get off track. You may want to decrease granularities to give the agent more rooms to accelerate or decelerate and further tweak the reward function in other ways to help training converge faster. After the training converges, evaluate the 3rd model to see if the lap time improves. Keep exploring until there is no more improvement.
4. Choose a more complicated track and repeat **Step 1** to **Step 3**. Evaluate your model on a track that is different from the one you used to train on to see how the model can generalize to different virtual tracks generalize to real-world environments (p. 14).
5. (Optional) Experiment with different values of the hyperparameters (p. 34) to improve the training process and repeat **Step 1** to **Step 3**.

6. (Optional) Examine and analyze the AWS DeepRacer logs. For sample code that you can use to analyze the logs, see https://github.com/aws-samples/aws-deepracer-workshops/tree/master/log-analysis.

Tailor AWS DeepRacer Training for Object Avoidance Races

After you become familiar with time trials and have trained a few converged models, move on to the next more demanding challenge—obstacle avoidance. Here, your goal is to train a model that can complete a lap as fast as possible without going off track, while avoiding crashing into the objects placed on the track. This is obviously a harder problem for the agent to learn, and training takes longer to converge.

The AWS DeepRacer console supports two types of obstacle avoidance training: obstacles can be placed at fixed or random locations along the track. With fixed locations, the obstacles remain fixed to the same place throughout the training job. With random locations, the obstacles change their respective places at random from episode to episode.

It is easier for trainings to converge for location-fixed obstacle avoidance because the system has less degrees of freedom. However, models can overfit when the location information is built in to the trained models. As a result, the models may be overfitted and may not generalize well. For randomly positioned obstacle avoidance, it's harder for trainings to converge because the agent must keep learning to avoid crashing into obstacles at locations it hasn't seen before. However, models trained with this option tend to generalize better and transfer well to the real-world races. To begin, have obstacles placed at fixed locations, get familiar with the behaviors, and then tackle the random locations.

In the AWS DeepRacer simulator, the obstacles are cuboid boxes with the same dimensions (9.5" (L) x 15.25" (W) x 10/5" (H)) as the AWS DeepRacer vehicle's package box. This makes it simpler to transfer the trained model from the simulator to the real world if you place the packaging box as an obstacle on the physical track.

To experiment with obstacle avoidance, follow the recommended practice outlined in the steps below:

1. Use the default agent or experiment with new sensors and action spaces by customizing an existing agent or building a new one. You should limit the top speed to below 0.8 m/s and the speed granularity to 1 or 2 levels.

   Start training a model for around 3 hours with 2 objects at fixed locations. Use the example reward function and train the model on the track that you will be racing on, or a track that closely resembles that track. The **2019 Championship Cup** track is a simple track, which makes it a good choice for summit race preparation. Evaluate the model on the same track with the same number of obstacles. Watch how the total expected reward converges, if at all.
2. Read about the reward function parameters (p. 58). Experiment with variations of your reward function. Increase the obstacle number to 4. Train the agent to see if the training converges in the same amount of training time. If it doesn't, tweak your reward function again, lower the top speed or reduce the number of obstacles, and the train the agent again. Repeat experimenting until there is no more significant improvement.
3. Now, move on to training avoiding obstacles at random locations. You'll need to configure the agent with additional sensors, which are available from **Garage** in the AWS DeepRacer console. You can use a stereo camera. Or you can combine a LiDAR unit with either a single-lens camera or a stereo camera, but should expect a longer training time. Set the action space with a relatively low top speed (e.g. 2 m/s) for the training to converge quicker. For the network architecture, use a shallow neural network, which has been found sufficient for obstacle avoidance.
4. Start training for 4 hours the new agent for obstacle avoidance with 4 randomly placed objects on an simple track. Then evaluate your model on the same track to see if it can finish laps with

randomly positioned obstacles. If not, you may want to tweak your reward function, try different sensors and have longer training time. As another tip, you can try cloning an existing model to continue training to leverage previously learned experience.

5. (Optional) Choose a higher top speed for the action space or have more obstacles randomly placed along the track. Experiment with different combination of sensors and tweak the reward functions and hyperparameter values. Experiment with the **5-layer CNN** network topology. Then, retrain the model to determine how they affect convergence of the training.

Tailor AWS DeepRacer Training for Head-to-Head Races

Having gone through training obstacle avoidance, you're now ready to tackle the next level of challenge: training models for head-to-head races. Unlike the obstacle avoidance events, head-to-head racing has a dynamic environment with moving vehicles. Your goal is to train models for your AWS DeepRacer vehicle to compete against other moving vehicles in order to reach the finish line first without going off track or crashing to any of other vehicles. In the AWS DeepRacer console you can train a head-to-head racing model by having your agent to compete against 1-4 bot vehicles. Generally speaking, you should have more obstacles placed on a longer track.

Each bot vehicle follows a predefined path at constant speed. You can enable it to change lanes or to remains on its starting lane. Similar to training for obstacle avoidance, you can have the bot vehicles evenly distributed across the track on both lanes. The console limits you to have up to 4 bot vehicles on the track. Having more competing vehicles on the track provides the learning agent with more opportunities to encounter more varied situations with the other vehicles. This way, it learns more in one training job and the agent gets trained faster. However, each training is likely to take longer to converge.

To train an agent with bot vehicles, you should set the top speed of the agent's action space higher than the (constant) speed of the bot vehicles so that the agent has more passing opportunities during training. As a good starting point, you should set the agent's top speed at 0.8 m/s and the bot vehicle's moving speed at 0.4 m/s. If you enable the bots to change lanes, the training becomes more challenging because the agent must learn not only how to avoid crashing into a moving vehicle in the front on the same lane but also how to avoid crashing into another moving vehicle in the front on the other lane. You can set the bots to change lanes at random intervals. The length of an interval is randomly selected from a range of time (e.g. 1s to 5s) that you specify before starting the training job. This lane-changing behavior is more similar to the real-world head-to-head racing behaviors and the trained agent should generate better. However, it takes longer to train the model to converge.

Follow these suggested steps to iterate your training for head-to-head racing:

1. In **Garage** of the AWS DeepRacer console, build a new training agent configured with both stereo cameras and a LiDAR unit. It is possible to train a relatively good model using only stereo camera against bot vehicles. LiDAR helps reduce blind spots when the agent changes lanes. Do not set the top speed too high. A good starting point is 1 m/s.
2. To train for head-to-head racing against bot vehicles, start with two bot vehicles. Set the bot's moving speed lower than your agent’s top speed (e.g. 0.5 m/s if the agent's top speed is 1 m/s). Disable the lane-changing option, and then choose the training agent you just created. Use one of the reward function examples or make minimally necessary modifications, and then train for 3 hours. Use the track that you will be racing on, or a track that closely resembles that track. The **2019 Championship Cup** track is a simple track, which makes it a good choice for summit race preparation. After the training is complete, evaluate the trained model on the same track.
3. For more challenging tasks, clone your trained model for a second head-to-head racing model. Proceed to either experiment with more bot vehicles or enable lane-changing options. Start with slow lane-changing operations at random intervals longer than 2 seconds. You may also want to experiment with custom reward functions. In general, your custom reward function logic can be similar to those for obstacle avoidance, if you don't take into consideration a balance between

surpassing other vehicles and staying on track. Depends on how good your previous model is, you may need to train another 3 to 6 hours. Evaluate your models and see how the model performs.

Train and Evaluate AWS DeepRacer Models Using the AWS DeepRacer Console

To train a reinforcement learning model, you can use the AWS DeepRacer console. In the console, create a training job, choose a supported framework and an available algorithm, add a reward function, and configure training settings. You can also watch training proceed in a simulator. You can find the step-by-step instructions in the section called “Train Your First Model” (p. 15).

This section explains how to train and evaluate an AWS DeepRacer model. It also shows how to create and improve a reward function, how an action space affects model performance, and how hyperparameters affect training performance. You can also learn how to clone a training model to extend a training session, how to use the simulator to evaluate training performance, and how to address some of the simulation to real-world challenges.

Topics

- Create Your Reward Function (p. 31)
- Explore Action Space to Train a Robust Model (p. 33)
- Systematically Tune Hyperparameters (p. 34)
- Examine AWS DeepRacer Training Job Progress (p. 37)
- Clone a Trained Model to Start a New Training Pass (p. 38)
- Evaluate AWS DeepRacer Models in Simulations (p. 39)
- Log AWS DeepRacer Events to CloudWatch Logs (p. 39)
- Optimize Training AWS DeepRacer Models for Real Environments (p. 41)

Create Your Reward Function

A reward function (p. 58) describes immediate feedback (as a reward or penalty score) when your AWS DeepRacer vehicle moves from one position on the track to a new position. The function's purpose is to encourage the vehicle to make moves along the track to reach a destination quickly without accident or infraction. A desirable move earns a higher score for the action or its target state. An illegal or wasteful move earns a lower score. When training an AWS DeepRacer model, the reward function is the only application-specific part.

In general, you design your reward function to act like an incentive plan. Different incentive strategies could result in different vehicle behaviors. To make the vehicle drive faster, the function should give rewards for the vehicle to follow the track. The function should dispense penalties when the vehicle takes too long to finish a lap or goes off the track. To avoid zig-zag driving patterns, it could reward the vehicle to steer less on straighter portions of the track. The reward function might give positive scores when the vehicle passes certain milestones, as measured by `waypoints` (p. 58). This could alleviate waiting or driving in the wrong direction. It is also likely that you would change the reward function to account for the track conditions. However, the more your reward function takes into account environment-specific information, the more likely your trained model is over-fitted and less general. To make your model more generally applicable, you can explore action space (p. 33).

If an incentive plan is not carefully considered, it can lead to unintended consequences of opposite effect. This is possible because the immediate feedback is a necessary but not sufficient condition for reinforcement learning. An individual immediate reward by itself also can’t determine if the move is desirable. At a given position, a move can earn a high reward. A subsequent move could go off the track

and earn a low score. In such case, the vehicle should avoid the move of the high score at that position. Only when all future moves from a given position yield a high score on average should the move to the next position be deemed desirable. Future feedback is discounted at a rate that allows for only a small number of future moves or positions to be included in the average reward calculation.

A good practice to create a reward function (p. 58) is to start with a simple one that covers basic scenarios. You can enhance the function to handle more actions. Let's now look at some simple reward functions.

Topics

- Simple Reward Function Examples (p. 32)
- Enhance Your Reward Function (p. 32)

Simple Reward Function Examples

We can start building the reward function by first considering the most basic situation. The situation is driving on a straight track from start to finish without going off the track. In this scenario, the reward function logic depends only `on_track` and `progress`. As a trial, you could start with the following logic:

```
def reward_function(params):
    if not params["all_wheels_on_track"]:
        reward = -1
    else if params["progress"] == 1 :
        reward = 10
    return reward
```

This logic penalizes the agent when it drives itself off the track. It rewards the agent when it drives to the finishing line. It's reasonable for achieving the stated goal. However, the agent roams freely between the starting point and the finishing line, including driving backwards on the track. Not only could the training take a long time to complete, but also the trained model would lead to less efficient driving when deployed to a real-world vehicle.

In practice, an agent learns more effectively if it can do so bit-by-bit throughout the course of training. This implies that a reward function should give out smaller rewards step by step along the track. For the agent to drive on the straight track, we can improve the reward function as follows:

```
def reward_function(params):
    if not params["all_wheels_on_track"]:
        reward = -1
    else:
        reward = params["progress"]
    return reward
```

With this function, the agent gets more reward the closer it reaches the finishing line. This should reduce or eliminate unproductive trials of driving backwards. In general, we want the reward function to distribute the reward more evenly over the action space. Creating an effective reward function can be a challenging undertaking. You should start with a simple one and progressively enhance or improve the function. With systematic experimentation, the function can become more robust and efficient.

Enhance Your Reward Function

After you have successfully trained your AWS DeepRacer model for the simple straight track, the AWS DeepRacer vehicle (virtual or physical) can drive itself without going off the track. If you let the vehicle run on a looped track, it won't stay on the track. The reward function has ignored the actions to make turns to follow the track.

To make your vehicle handle those actions, you must enhance the reward function. The function must give a reward when the agent makes a permissible turn and produce a penalty if the agent makes an illegal turn. Then, you're ready to start another round of training. To take advantage of the prior training, you can start the new training by cloning the previously trained model, passing along the previously learned knowledge. You can follow this pattern to gradually add more features to the reward function to train your AWS DeepRacer vehicle to drive in increasingly more complex environments.

For more advanced reward functions, see the following examples:

- the section called “Example 1: Follow the Center Line in Time Trials” (p. 73)
- the section called “Example 2: Stay Inside the Two Borders in Time Trials” (p. 74)
- the section called “Example 3: Prevent Zig-Zag in Time Trials” (p. 74)
- the section called “Example 4: Stay On One Lane without Crashing into Stationary Obstacles or Moving Vehicles” (p. 75)

Explore Action Space to Train a Robust Model

As a general rule, train your model to be as robust as possible so that you can apply it to as many environments as possible. A robust model is one that can be applied to a wide range of track shapes and conditions. Generally speaking, a robust model is not "smart" because its reward function does not have the ability to contain explicit environment-specific knowledge. Otherwise, your model is likely to be applicable only to an environment similar to the trained one.

Explicitly incorporating environment-specific information into the reward function amounts to feature engineering. Feature engineering helps reduce training time and can be useful in solutions tailor made to a particular environment. To train a model of the general applicability though, you should refrain from attempting a lot of feature engineering.

For example, when training a model on a circular track, you can't expect to obtain a trained model applicable to any non-circular track if you have such geometric properties explicitly incorporated into the reward function.

How would you go about training a model as robust as possible while keeping the reward function as simple as possible? One way is to explore the action space spanning the actions your agent can take. Another is to experiment with hyperparameters (p. 34) of underlying training algorithm. Often times, you do both. Here, we focus on how to explore the action space to train a robust model for your AWS DeepRacer vehicle.

In training an AWS DeepRacer model, an action (`a`) is a combination of speed (`t` meters per second) and steering angle (`s` in degrees). The action space of the agent defines the ranges of speed and steering angle the agent can take. For a discrete action space of `m` number of speeds, $(v_1, .., v_n)$ and `n` number of steering angles, $(s_1, .., s_m)$, there are `m*n` possible actions in the action space:

$$
\begin{aligned}
&a_1: && (v_1, s_1) \\
&\ldots \\
&a_n: && (v_1, s_n) \\
&\\
&\ldots \\
&a_{(i-1)*n+j}: && (v_i, s_j) \\
&\ldots \\
&\\
&a_{(m-1)*n+1}: && (v_m, s_1) \\
&\ldots \\
&a_{m*n}: && (v_m, s_n)
\end{aligned}
$$

The actual values of (v_i, s_j) depend on the ranges of v_{max} and $|s_{max}|$ and are not uniformly distributed.

Each time you begin training or iterating your AWS DeepRacer model, you must first specify the `n`, `m`, v_{max} and $|s_{max}|$ or agree to using their default values. Based on your choice, the AWS DeepRacer service generates the available actions your agent can choose in training. The generated actions are not uniformly distributed over the action space.

In general, a larger number of actions and larger action ranges give your agent more room or options to react to more varied track conditions, such as a curved track with irregular turning angles or directions. The more options available to the agent, the more readily it can handle track variations. As a result, you can expect that the trained model to be more widely applicable, even when using a simple reward function .

For example, your agent can learn quickly to handle straight-line track using a coarse-grained action space with small number of speeds and steering angles. On a curved track, this coarse-grained action space is likely to cause the agent to overshoot and go off the track while it turns. This is because there are not enough options at its disposal in order to adjust its speed or steering. Increase the number of speeds or the number of steering angles or both, the agent should become more capable of maneuvering the curves while keeping on the track. Similarly, if your agent moves in a zig-zag fashion, you can try to increase the number of steering ranges to reduce drastic turns at any given step.

When the action space is too large, training performance may suffer, because it takes longer to explore the action space. Be sure to balance the benefits of a model's general applicability against its training performance requirements. This optimization involves systematic experimentation.

Systematically Tune Hyperparameters

One way to improve your model's performance is to enact a better or more effective training process. For example, to obtain a robust model, training must provide your agent more or less evenly distributed sampling over the agent's action space. This requires a sufficient mix of exploration and exploitation. Variables affecting this include the amount of training data used (`number of episodes between each training` and `batch size`), how fast the agent can learn (`learning rate`), the portion of exploration (`entropy`). To make training practical, you may want to speed the learning process. Variables affecting this include `learning rate`, `batch size`, `number of epochs` and `discount factor`.

The variables affecting the training process are known as hyperparameters of the training. These algorithm attributes are not properties of the underlying model. Unfortunately, hyperparameters are empirical in nature. Their optimal values are not known for all practical purposes and require systematic experimentation to derive.

Before discussing the hyperparameters that can be adjusted to tune the performance of training your AWS DeepRacer model, let's define the following terminology.

Data point

A data point, also known as an *experience*, it is a tuple of (*s,a,r,s'*), where *s* stands for an observation (or state) captured by the camera, *a* for an action taken by the vehicle, *r* for the expected reward incurred by the said action, and *s'* for the new observation after the action is taken.

Episode

An episode is a period in which the vehicle starts from a given starting point and ends up completing the track or going off the track. It embodies a sequence of experiences. Different episodes can have different lengths.

Experience buffer

An experience buffer consists of a number of ordered data points collected over fixed number of episodes of varying lengths during training. For AWS DeepRacer, it corresponds to images captured by the camera mounted on your AWS DeepRacer vehicle and actions taken by the vehicle and serves as the source from which input is drawn for updating the underlying (policy and value) neural networks.

Batch

A batch is an ordered list of experiences, representing a portion of simulation over a period of time, used to update the policy network weights. It is a subset of the experience buffer.

Training data

A training data is a set of batches sampled at random from an experience buffer and used for training the policy network weights.

Algorithmic hyperparameters and their effects

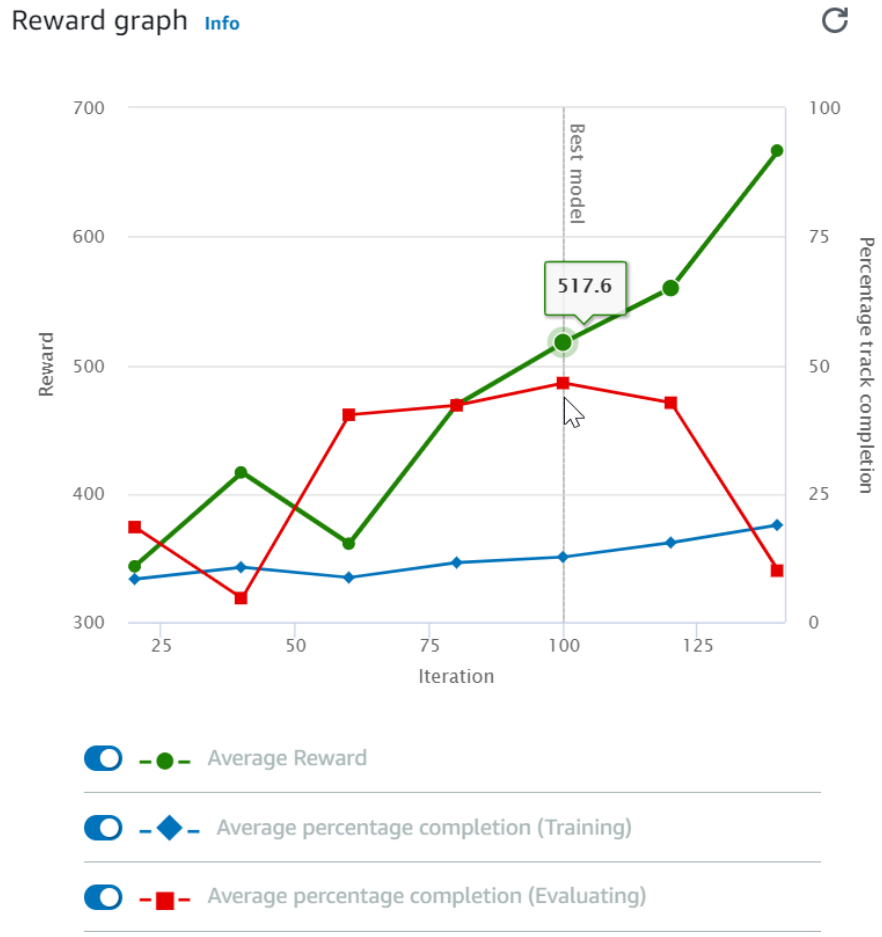
Hyperparamete	Description
Gradient descent batch size	The number recent vehicle experiences sampled at random from an experience buffer and used for updating the underlying deep-learning neural network weights. Random sampling helps reduce correlations inherent in the input data. Use a larger batch size to promote more stable and smooth updates to the neural network weights, but be aware of the possibility that the training may be longer or slower. Required Yes Valid values Positive integer of `(32, 64, 128, 256, 512)` Default value `64`
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. Use a larger number of epochs to promote more stable updates, but expect a slower training. When the batch size is small, you can use a smaller number of epochs Required No Valid values Positive integer between `[3 – 10]` Default value `3`
Learning rate	During each update, a portion of the new weight can be from the gradient-descent (or ascent) contribution and the rest from the existing weight value. The learning rate controls how much a gradient-descent (or ascent) update contributes to the network weights. Use a higher learning rate to include more gradient-descent contributions for faster training, but be aware of the possibility that the expected reward may not converge if the learning rate is too large. Required No Valid values Real number between `0.00000001` (or 10^{-8}) and `0.001` (or 10^{-3})

Hyperparamete	Description
	Default value `0.0003`
`Entropy`	A degree of uncertainty used to determine when to add randomness to the policy distribution. The added uncertainty helps the AWS DeepRacer vehicle explore the action space more broadly. A larger entropy value encourages the vehicle to explore the action space more thoroughly. Required No Valid values Real number between `0` and `1`. Default value `0.01`
Discount factor	A factor specifies how much of the future rewards contribute to the expected reward. The larger the **Discount factor** value is, the farther out contributions the vehicle considers to make a move and the slower the training. With the discount factor of 0.9, the vehicle includes rewards from an order of 10 future steps to make a move. With the discount factor of 0.999, the vehicle considers rewards from an order of 1000 future steps to make a move. The recommended discount factor values are 0.99, 0.999 and 0.9999. Required No Valid values Real number between `0` and `1`. Default value `0.999`
Loss type	Type of the objective function used to update the network weights. A good training algorithm should make incremental changes to the agent's strategy so that it gradually transitions from taking random actions to taking strategic actions to increase reward. But if it makes too big a change then the training becomes unstable and the agent ends up not learning. The **Huber loss** and **Mean squared error loss** types behave similarly for small updates. But as the updates become larger, **Huber loss** takes smaller increments compared to **Mean squared error loss**. When you have convergence problems, use the **Huber loss** type. When convergence is good and you want to train faster, use the **Mean squared error loss** type. Required No Valid values (**Huber loss**, **Mean squared error loss**) Default value **Huber loss**

Hyperparamete	Description
Number of experience episodes between each policy-updating iteration	The size of the experience buffer used to draw training data from for learning policy network weights. An experience episode is a period in which the agent starts from a given starting point and ends up completing the track or going off the track. It consists of a sequence of experiences. Different episodes can have different lengths. For simple reinforcement-learning problems, a small experience buffer may be sufficient and learning is fast. For more complex problems that have more local maxima, a larger experience buffer is necessary to provide more uncorrelated data points. In this case, training is slower but more stable. The recommended values are 10, 20 and 40. Required No Valid values Integer between `5` and `100` Default value `20`

Examine AWS DeepRacer Training Job Progress

After starting your training job, you can examine the training metrics of rewards and track completion per episode to ascertain the training job's performance of your model. On the AWS DeepRacer console, the metrics are displayed in the **Reward graph**, as shown in the following illustration.



You can choose to view the reward gained per episode, the averaged reward per iteration, the progress per episode, the averaged progress per iteration or any combination of them. To do so, toggle the **Reward (Episode, Average)** or **Progress (Episode, Average)** switches at the bottom of **Reward graph**. The reward and progress per episode are displayed as scattered plots in different colors. The averaged reward and track completion are displayed by line plots and start after the first iteration.

The range of rewards is shown on the left side of the graph and the range of progress (0-100) is on the right side. To read the exact value of of a training metric, move the mouse near to the data point on the graph.

The graphs are automatically updated every 10 seconds while training is under way. You can choose the refresh button to manually update the metric display.

A training job is good if the averaged reward and track completion show trends to converge. In particular, the model has likely converged if the progress per episode continuously reach 100% and the reward levels out. If not, clone the model and retrain it.

Clone a Trained Model to Start a New Training Pass

If you clone a previously trained model as the starting point of a new round of training, you could improve training efficiency. To do this, modify the hyperparameters to make use of already learned knowledge.

In this section, you learn how to clone a trained model using the AWS DeepRacer console.

To iterate training the reinforcement learning model using the AWS DeepRacer console

1. Sign in to the AWS DeepRacer console, if you're not already signed in.
2. On the **Models** page, choose a trained model and then choose **Clone** from the **Action** drop-down menu list.
3. For **Model details**, do the following:
   a. Type `RL_model_1` in **Model name**, if you don't want a name to be generated for the cloned model.
   b. Optionally, give a description for the to-be-cloned model in **Model description - optional**.
4. For **Environment simulation**, choose another track option.
5. For **Reward function**, choose one of the available reward function examples. Modify the reward function. For example, consider steering.
6. Expand **Algorithm settings** and try different options. For example, change the **Gradient descent batch size** value from 32 to 64 or increase the **Learning rate** to speed up the training.
7. Experiment with difference choices of the **Stop conditions**.
8. Choose **Start training** to begin new round of training.

As with training a robust machine learning model in general, it is important that you conduct systematic experimentation to come up with the best solution.

Evaluate AWS DeepRacer Models in Simulations

To evaluate a model is to test the performance of a trained model. In AWS DeepRacer, the standard performance metric is the average time of finishing three consecutive laps. Using this metric, of any two models, one is better if it can make the agent go faster on the same track than the other one.

In general, evaluating a model involves the following tasks:

1. Configure and start an evaluation job.
2. Observe the evaluation in progress while the job is running. This can be done in the AWS DeepRacer simulator.
3. Inspect the evaluation summary after the evaluation job is done. You can terminate an evaluation job in progress at any time.
4. Optionally, submit the evaluation result to an eligible AWS DeepRacer leaderboard (p. 119). The ranking on the leaderboard lets you know how well your model performs against other participants.

Test an AWS DeepRacer model with an AWS DeepRacer vehicle driving on a physical track, see *Operate Your Vehicle* (p. 77).

Log AWS DeepRacer Events to CloudWatch Logs

For diagnostic purposes, AWS DeepRacer reports certain runtime events to CloudWatch Logs during training and evaluation.

The events are logged in job-specific log streams. For a training job, the log stream appears under the `/aws/sagemaker/TrainingJobs` log group. For a simulation job, the log stream appears under the `/aws/robomaker/SimulationJobs` log group. For an evaluation job submitted to a leaderboard in the AWS DeepRacer League Virtual Circuit, the log stream appears under the `/aws/deepracer/leaderboard/SimulationJobs` log group. For the reward function execution, the log stream appears under the `/aws/lambda/AWS-DeepRacer-Test-Reward-Function` log group.

Most of the log entries are self-explanatory, except for those starting with "SIM_TRACE_LOG". An example of this log entry is shown as follows:

```
SIM_TRACE_LOG:0,14,3.1729,0.6200,-0.2606,-0.26,0.50,2,0.5000,False,True,1.4878,1,17.67,1563406790.24001
```

The event items correspond to the following data values, respectively:

```
SIM_TRACE_LOG: episode, step, x-coordinate, y-
coordinate, heading, steering_angle, speed, action_taken, reward, job_completed, all_wheels_on_track, p
  closest_waypoint_index, track_length, time.time()
```

To access the AWS DeepRacer logs, you can use the CloudWatch console , the AWS CLI or an AWS SDK.

To view AWS DeepRacer logs using the AWS CLI

1. Open a terminal window.
2. Type the following command:

```
aws logs get-log-events \
    --log-group-name a-deepracer-log-group-name \
    --log-stream-name a-deepracer-log-stream-name
```

The command returns a result similar to the following output:

```
{
    "events": [
        {
            "timestamp": 1563406819300,
            "message":
 "SIM_TRACE_LOG:2,155,7.3941,1.0048,0.0182,-0.52,1.00,1,0.0010,False,False,14.7310,16,1
7.67,1563406818.939216",
            "ingestionTime": 1563406819310
        },

        ...

        {
            "timestamp": 1563407217100,
            "message":
 "SIM_TRACE_LOG:39,218,5.6879,0.3078,-0.1135,0.52,1.00,9,0.0000,True,False,20.7185,9,17.67,15634072
            "ingestionTime": 1563407217108
        },
        {
            "timestamp": 1563407218143,
            "message": "Training> Name=main_level/agent, Worker=0, Episode=40, Total
 reward=61.93, Steps=4315, Training iteration=0",
            "ingestionTime": 1563407218150
        }
    ],
    "nextForwardToken": "f/34865146013350625778794700014105997464971505654143647744",
    "nextBackwardToken": "b/34865137118854508561245373892407536877673471318173089813"
}
```

To view AWS DeepRacer logs in the CloudWatch Logs console:

1. Sign in to the CloudWatch console.
2. Choose **Logs** from the main navigation pane.

3. Choose an appropriate log group.

   To help quickly find the AWS DeepRacer-specific event logs, type one of the aforementioned log group names in the **Filter** box.
4. Choose a log stream to open the log file.

   To quickly locate the most recent log stream in a given log group, sort the list by **Last Event Time**.

Optimize Training AWS DeepRacer Models for Real Environments

Many factors affect the real-world performance of a trained model, including the choice of the action space (p. 33), reward function (p. 31), hyperparameters (p. 34) used in the training, and vehicle calibration (p. 94) as well as real-world track (p. 109) conditions. In addition, the simulation is only an (often crude) approximation of the real world. They make it a challenge to train a model in simulation, to apply it to the real world, and to achieve a satisfactory performance.

Training a model to give a solid real-world performance often requires numerous iterations of exploring the reward function (p. 31), action spaces (p. 33), hyperparameters (p. 34), and evaluation (p. 39) in simulation and testing (p. 101) in a real environment. The last step involves the so-called *simulation-to-real world* (*sim2real*) transfer and can feel unwieldy.

To help tackle the *sim2real* challenges, heed the following considerations:

- Make sure that your vehicle is well calibrated.

  This is important because the simulated environment is most likely a partial representation of the real environment. Besides, the agent takes an action based on the current track condition, as captured by an image from the camera, at each step. It cannot see far enough to plan its route at a fast speed. To accommodate this, the simulation imposes limits on the speed and steering. To ensure the trained model works in the real world, the vehicle must be properly calibrated to match this and other simulation settings. For more information for calibrating your vehicle, see the section called “Calibrate Your Vehicle” (p. 94).
- Test your vehicle with the default model first.

  Your AWS DeepRacer vehicle comes with a pre-trained model loaded into its inference engine. Before testing your own model in the real world, verify that the vehicle performs reasonably well with the default model. If not, check the physical track setup. Testing a model in an incorrectly built physical track is likely to lead to a poor performance. In such cases, reconfigure or repair your track before starting or resuming testing.

  Note
  When running your AWS DeepRacer vehicle, actions are inferred according to the trained policy network without invoking the reward function.
- Make sure the model works in simulation.

  If your model doesn't work well in the real world, it's possible that either the model or track is defective. To sort out the root causes, you should first evaluate the model in simulations (p. 39) to check if the simulated agent can finish at least one loop without getting off the track. You can do so by inspecting the convergence of the rewards while observing the agent's trajectory in the simulator. If the reward reaches the maximum when the simulated agents completes a loop without faltering, the model is likely to be a good one.
- Do not over train the model.

Continuing training after the model has consistently completed the track in simulation will cause overfitting in the model. An over-trained model won't perform well in the real world because it can't handle even minor variations between the simulated track and the real environment.

- Use multiple models from different iterations.

  A typical training session produces a range of models that fall between being underfitted and being overfitted. Because there are no a priori criteria to determine a model that is just right, you should pick a few model candidates from the time when the agent completes a single loop in the simulator to the point where it performs loops consistently.
- Start slow and increase the driving speed gradually in testing.

  When testing the model deployed to your vehicle, start with a small maximum speed value. For example, you can set the testing speed limit to be <10% of the training speed limit. Then gradually increase the testing speed limit until the vehicle starts moving. You set the testing speed limit when calibrating the vehicle using the device control console. If the vehicle goes too fast, i.e. the speed exceeds those seen during training in simulator, the model is not likely to perform well on the real track.
- Test a model with your vehicle in different starting positions.

  The model learns to take a certain path in simulation and can be sensitive to its position within the track. You should start the vehicle tests with different positions within the track boundaries (from left to center to right) to see if the model performs well from certain positions. Most models tend to make the vehicle stay close to either side of one of the white lines. To help analyze the vehicle's path, plot the vehicle's positions (x, y) step by step from the simulation to identify likely paths to be taken by your vehicle in a real environment.
- Start testing with a straight track.

  A straight track is much easier to navigate compared to a curved track. Starting your test with a straight track is useful to weed out poor models quickly. If a vehicle cannot follow a straight track most of the time, the model will not perform well on curved tracks, either.
- Watch out for the behavior where the vehicle takes only one type of actions,

  When your vehicle can manage to take only one type of actions, e.g., to steer the vehicle to the left only, the model is likely over-fitted or under-fitted. With given model parameters, too many iterations in training could make the model over-fitted. Too few iterations could make it under-fitted.
- Watch out for vehicle's ability to correct its path along a track border.

  A good model makes the vehicle to correct itself when nearing the track borders. Most well-trained models have this capability. If the vehicle can correct itself on both the track borders, the model is considered to be more robust and of a higher quality.
- Watch out for inconsistent behaviors exhibited by the vehicle.

  A policy model represents a probability distribution for taking an action in a given state. With the trained model loaded to its inference engine, a vehicle will pick the most probable action, one step at time, according to the model's prescription. If the action probabilities are evenly distributed, the vehicle can take any of the actions of the equal or closely similar probabilities. This will lead to an erratic driving behavior. For example, when the vehicle follows a straight path sometimes (e.g., half the time) and makes unnecessary turns at other times, the model is either under-fitted or over-fitted.
- Watch out for only one type of turns (left or right) made by the vehicle.

  If the vehicle takes left turns very well but fails to manage steering right, or, similarly, if the vehicle takes only right turns well, but not left steering, you need to carefully calibrate or recalibrate your vehicle's steering. Alternatively, you can try to use a model that is trained with the settings close to the physical settings under testing.
- Watch out for the vehicle's making sudden turns and go off-track.

If the vehicle follows the path correctly most of the way, but suddenly veers off the track, it is likely due to distractions in the environment. Most common distractions include unexpected or unintended light reflections. In such cases, use barriers around the track or other means to reduce glaring lights.

Train and Evaluate AWS DeepRacer Models Using Amazon SageMaker Notebooks

The AWS DeepRacer console provides you with an integrated experience to train and evaluate your AWS DeepRacer models. It's integrated because AWS DeepRacer uses Amazon SageMaker and AWS RoboMaker behind the scenes. The integration includes detailed reinforcement learning tasks and makes training more readily accessible to beginners.

If you're an experienced user of Amazon SageMaker or if you're determined to learn how to use Amazon SageMaker and AWS RoboMaker to train and evaluate your AWS DeepRacer models, then you can manually create an Amazon SageMaker notebook. You can then clone a reinforcement learning sample notebook instance and use it as a template to perform the predefined tasks that train and evaluate an AWS DeepRacer model.

After the training, you can copy the trained model artifacts to your AWS DeepRacer vehicle for test runs in a physical environment.

The tutorial presents step-by-step instructions to walk you through these tasks.

Topics

- Create an Amazon SageMaker Notebook (p. 43)
- Initialize the Amazon SageMaker Notebook Instance (p. 44)
- Set Up the Training Environment (p. 48)
- Train Your AWS DeepRacer Model (p. 51)

Create an Amazon SageMaker Notebook

To train an AWS DeepRacer model directly on Amazon SageMaker, follow the steps below and create an Amazon SageMaker notebook instance.

To create an Amazon SageMaker notebook instance to train and evaluate your AWS DeepRacer models

1. Sign in to the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker. Choose one of the supported regions.
2. From the navigation pane, choose **Notebook instances** and then choose **Create notebook instance**.
3. On the **Create notebook instance** page, do the following:

   a. Type a name. For example, `my-deepracer-model`) for the **Notebook instance name**.

   b. If the **IAM role** drop-down menu is not populated with an existing IAM role, choose **Create a new role**, **Enter a custom IAM role ARN**, or **Use existing role** and then follow the instructions.

   c. Leave the default choices for all other options and then choose **Create notebook instance**.

   For more information, see creating an Amazon SageMaker notebook instance.

4. Wait for the notebook instance's **Status** to change from `Pending` to `InService`. Then choose **Open Jupyter**.
5. On the **Jupyter** page (which is the home page of the newly created notebook), do the following:

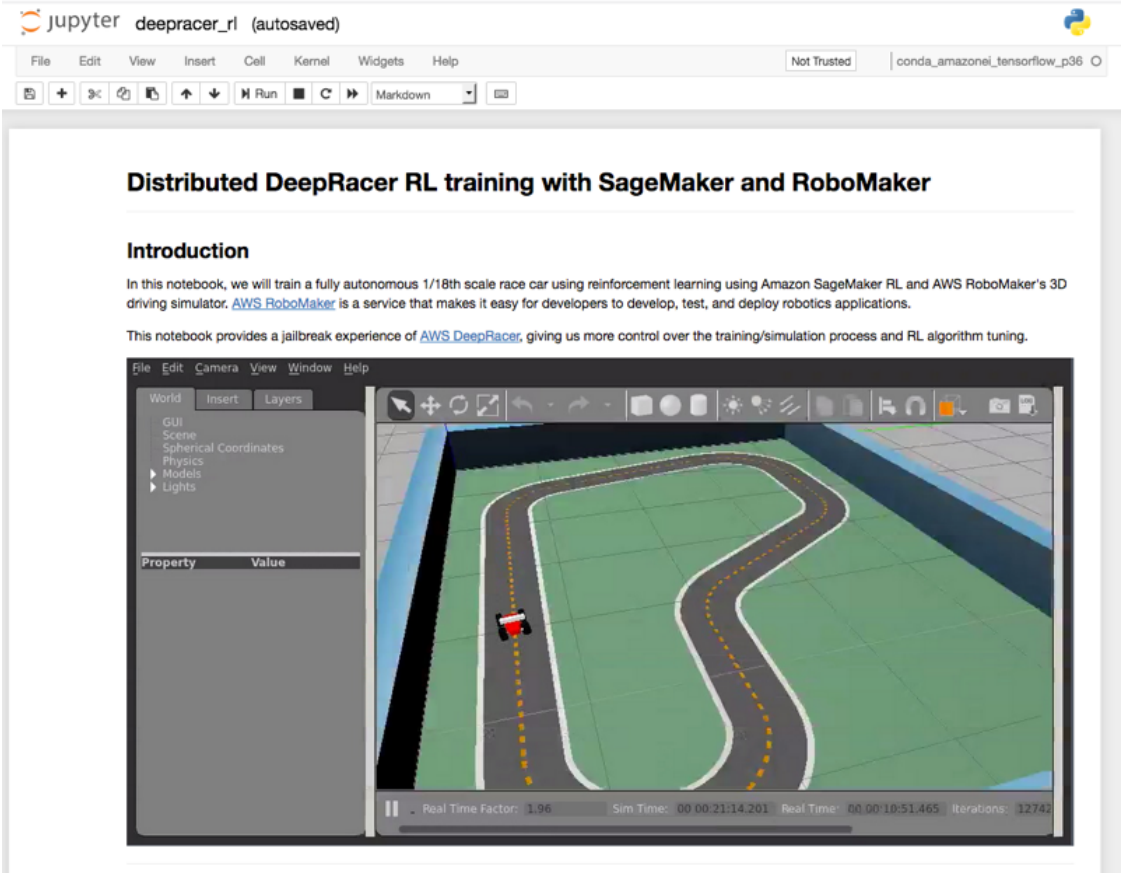
   a. Choose the **SageMaker Examples** tab.

   b. Expand the **Reinforcement Learning** example group from the example collection.

   c. For this exercise, choose **Use** next to the **deepracer_rl.ipynb** item.

   d. On the **Create a copy in your home directory** dialog, choose **Create copy**.

   At this point, the notebook instance is running and you can begin to train the model.



   You are charged for a running instance according to the selected instance type. To avoid being charged for a running instance when you're not ready to use it, shut down the instance.

Initialize the Amazon SageMaker Notebook Instance

To use an Amazon SageMaker notebook instance to train your AWS DeepRacer model, first properly initialize the instance for the required job. The initialization includes the following.

- Import required libraries.
- Set up training environment.
- Grant access permissions for Amazon SageMaker and AWS RoboMaker.
- Provision a docker container to host training and evaluation jobs.

- Configure VPC for Amazon SageMaker and AWS RoboMaker to interact with each other.

Follow the steps below for detailed instructions to initialize a notebook instance.

To initialize an Amazon SageMaker notebook instance

1. To import the required library to do training, choose the notebook instance's first code block. For example, choose the one under the **Imports** heading. Next, choose **Run** from the notebook's menu bar to execute the code block. You can use the `Shift-Enter` key-command shortcuts to start running the code block.

```
In [ ]: import boto3
        import sagemaker
        import sys
        import os
        import re
        import numpy as np
        import subprocess
        sys.path.append("common")
        from misc import get_execution_role, wait_for_s3_object
        from docker_utils import build_and_push_docker_image
        from sagemaker.rl import RLEstimator, RLToolkit, RLFramework
        from time import gmtime, strftime
        import time
        from IPython.display import Markdown
        from markdown_helper import *
```

   Before the code execution starts, the code block status shows `In [ ]`. When the execution is under way, the status becomes `In [*]`. After the code execution is complete, the status becomes `In [`*n*`]`, where *n* corresponds to the order of invocations. Because the importation code cell is the first, `n=1`. If you run the command again after the first run, the status becomes `In [2]`.

   For asynchronous execution, the code cell returns immediately to show the completed status. For synchronous executions, subsequent calls are blocked until the current code cell execution is completed when the status turns from `In [*]` to `In [`*n*`]`.

2. To initialize the basic parameters, run the **Initializing basic parameters** code block as-is.

Initializing basic parameters

```
In [ ]: # Select the instance type
        instance_type = "ml.c4.2xlarge"
        #instance_type = "ml.p2.xlarge"
        #instance_type = "ml.c5.4xlarge"

        # Starting SageMaker session
        sage_session = sagemaker.session.Session()

        # Create unique job name.
        job_name_prefix = 'deepracer-notebook'

        # Duration of job in seconds (1 hours)
        job_duration_in_seconds = 3600

        # AWS Region
        aws_region = sage_session.boto_region_name
        if aws_region not in ["us-west-2", "us-east-1", "eu-west-1"]:
            raise Exception("This notebook uses RoboMaker which is available only in US East (N. Virginia),"
                            "US West (Oregon) and EU (Ireland). Please switch to one of these regions.")
```

   The example notebook instance sets the job duration for 1 hour by default. To speed up or extend the training, you can decrease or increase the `job_duration_in_seconds` value before running the code cell.

3. To set up the training output storage, choose the code block under **Setup S3 bucket**, and then choose **Run** from the notebook instance menu or press the `Shift+Enter` keys.

Setup S3 bucket

Set up the linkage and authentication to the S3 bucket that we want to use for checkpoint and metadata.

In [ ]:
```
# S3 bucket
s3_bucket = sage_session.default_bucket()

# SDK appends the job name and output folder
s3_output_path = 's3://{}/'.format(s3_bucket)

#Ensure that the S3 prefix contains the keyword 'sagemaker'
s3_prefix = job_name_prefix + "-sagemaker-" + strftime("%y%m%d-%H%M%S", gmtime())

# Get the AWS account id of this account
sts = boto3.client("sts")
account_id = sts.get_caller_identity()['Account']

print("Using s3 bucket {}".format(s3_bucket))
print("Model checkpoints and other metadata will be stored at: \ns3://{}/{}".format(s3_bucket, s3_prefix))
```

When the execution completes, you can verify this bucket in Amazon S3 console.

To view the `s3_output_path` variable value, append `print(s3_output_path)` to the above code cell and rerun the code.

4. To set up appropriate permissions for this notebook instance to access the S3 storage for output by Amazon SageMaker, run the code cell under **Create an IAM role**.

Create an IAM role

Either get the execution role when running from a SageMaker notebook `role = sagemaker.get_execution_role()` or, when running from local machine, use utils method `role = get_execution_role('role_name')` to create an execution role.

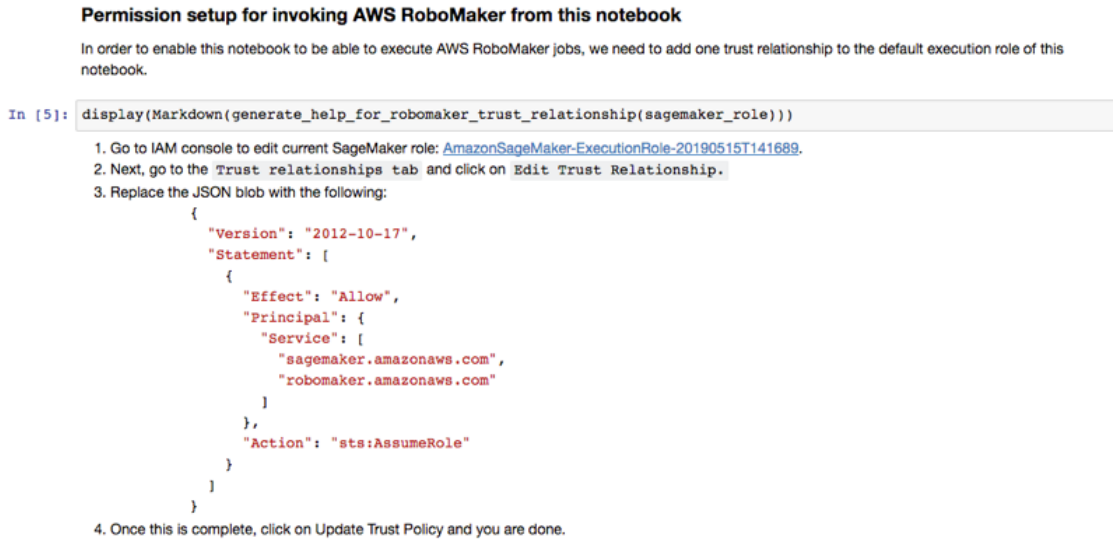
In [ ]:
```
try:
    sagemaker_role = sagemaker.get_execution_role()
except:
    sagemaker_role = get_execution_role('sagemaker')

print("Using Sagemaker IAM role arn: \n{}".format(sagemaker_role))
```

When executed, this code block creates a new IAM role containing the following IAM policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*"
            ]
        }
    ]
}
```

The created IAM role has Amazon SageMaker as its trusted entity.

5. To set up appropriate permissions for this notebook instance to invoke AWS RoboMaker to simulate the training environment, run the code cell under **Permission setup for invoking AWS RoboMaker from this notebook** and follow the instructions thereafter to add `robomaker.amazonaws.com` as another trusted entity of the previously created IAM role.



Permission setup for invoking AWS RoboMaker from this notebook

In order to enable this notebook to be able to execute AWS RoboMaker jobs, we need to add one trust relationship to the default execution role of this notebook.

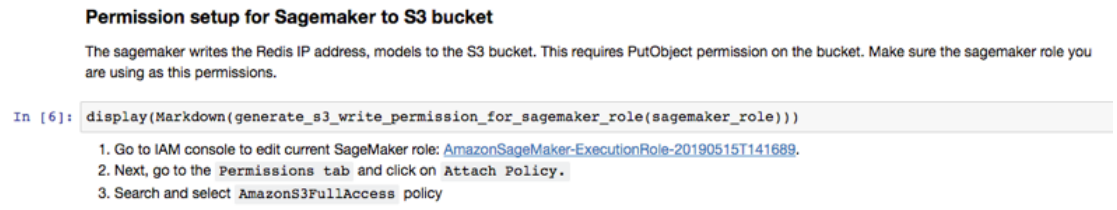
```
In [5]: display(Markdown(generate_help_for_robomaker_trust_relationship(sagemaker_role)))
```

1. Go to IAM console to edit current SageMaker role: AmazonSageMaker-ExecutionRole-20190515T141689.
2. Next, go to the `Trust relationships tab` and click on `Edit Trust Relationship.`
3. Replace the JSON blob with the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "sagemaker.amazonaws.com",
          "robomaker.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

4. Once this is complete, click on Update Trust Policy and you are done.

6. To set up the required permissions for Amazon SageMaker to access the S3 storage, run the code cell under **Permission setup for SageMaker to S3 bucket** and folow the instructions thereafter to attach the **AmazonS3FullAccess** policy to the IAM role previously created.



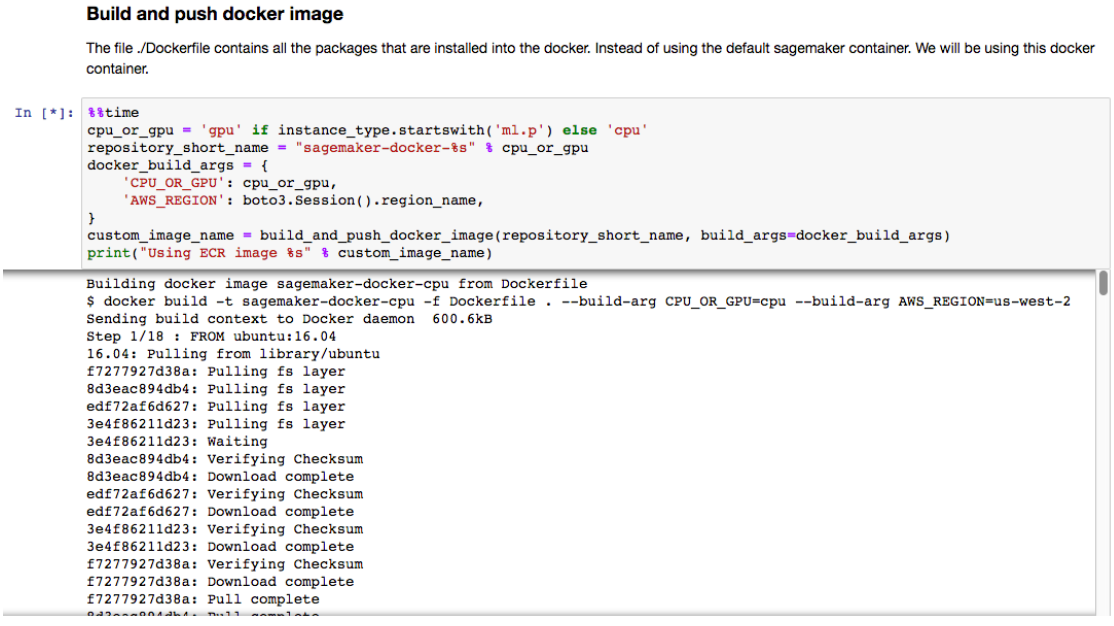
Permission setup for Sagemaker to S3 bucket

The sagemaker writes the Redis IP address, models to the S3 bucket. This requires PutObject permission on the bucket. Make sure the sagemaker role you are using as this permissions.

```
In [6]: display(Markdown(generate_s3_write_permission_for_sagemaker_role(sagemaker_role)))
```

1. Go to IAM console to edit current SageMaker role: AmazonSageMaker-ExecutionRole-20190515T141689.
2. Next, go to the `Permissions tab` and click on `Attach Policy.`
3. Search and select `AmazonS3FullAccess` policy

7. To provision a docker container for running our training and evalution jobs, run the code cell under **Build and push docker image**.



Build and push docker image

The file ./Dockerfile contains all the packages that are installed into the docker. Instead of using the default sagemaker container. We will be using this docker container.

```
In [*]: %%time
cpu_or_gpu = 'gpu' if instance_type.startswith('ml.p') else 'cpu'
repository_short_name = "sagemaker-docker-%s" % cpu_or_gpu
docker_build_args = {
    'CPU_OR_GPU': cpu_or_gpu,
    'AWS_REGION': boto3.Session().region_name,
}
custom_image_name = build_and_push_docker_image(repository_short_name, build_args=docker_build_args)
print("Using ECR image %s" % custom_image_name)
```

```
Building docker image sagemaker-docker-cpu from Dockerfile
$ docker build -t sagemaker-docker-cpu -f Dockerfile . --build-arg CPU_OR_GPU=cpu --build-arg AWS_REGION=us-west-2
Sending build context to Docker daemon  600.6kB
Step 1/18 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
f7277927d38a: Pulling fs layer
8d3eac894db4: Pulling fs layer
edf72af6d627: Pulling fs layer
3e4f86211d23: Pulling fs layer
3e4f86211d23: Waiting
8d3eac894db4: Verifying Checksum
8d3eac894db4: Download complete
edf72af6d627: Verifying Checksum
edf72af6d627: Download complete
3e4f86211d23: Verifying Checksum
3e4f86211d23: Download complete
f7277927d38a: Verifying Checksum
f7277927d38a: Download complete
f7277927d38a: Pull complete
```

Building and pushing the docker image takes some time to finish.

8. To enable VPC mode for Amazon SageMaker and AWS RoboMaker to communicate with each other over network, run the code cell under **Configure VPC**. By default, the notebook instance uses your default VPC, security group, and subnets to configure the VPC mode. If you don't want open VPC for other traffic, make sure to set the **Inbound Rules** and **Outbound Rules** for the specified security group to allow incoming traffic from itself only.

9. To enable the SageMaker training job to access S3 resources, run the code cell under **Create Route Table** to create a VPC S3 endpoint.

Create Route Table

A SageMaker job running in VPC mode cannot access S3 resources. So, we need to create a VPC S3 endpoint to allow S3 access from SageMaker container. To learn more about the VPC mode, please visit this link.

In [9]:

```
#TODO: Explain to customer what CREATE_ROUTE_TABLE is doing
CREATE_ROUTE_TABLE = True

def create_vpc_endpoint_table():
    print("Creating ")
    try:
        route_tables = [route_table["RouteTableId"] for route_table in ec2.describe_route_tables()['RouteTables']\
                        if route_table['VpcId'] == deepracer_vpc]
    except Exception as e:
        if "UnauthorizedOperation" in str(e):
            display(Markdown(generate_help_for_s3_endpoint_permissions(sagemaker_role)))
        else:
            display(Markdown(create_s3_endpoint_manually(aws_region, deepracer_vpc)))
        raise e

    print("Trying to attach S3 endpoints to the following route tables:", route_tables)

    if not route_tables:
        raise Exception(("No route tables were found. Please follow the VPC S3 endpoint creation "
                         "guide by clicking the above link."))
    try:
        ec2.create_vpc_endpoint(DryRun=False,
                                VpcEndpointType="Gateway",
                                VpcId=deepracer_vpc,
                                ServiceName="com.amazonaws.{}.s3".format(aws_region),
                                RouteTableIds=route_tables)
        print("S3 endpoint created successfully!")
    except Exception as e:
        if "RouteAlreadyExists" in str(e):
            print("S3 endpoint already exists.")
        elif "UnauthorizedOperation" in str(e):
            display(Markdown(generate_help_for_s3_endpoint_permissions(role)))
            raise e
        else:
            display(Markdown(create_s3_endpoint_manually(aws_region, default_vpc)))
            raise e

if CREATE_ROUTE_TABLE:
    create_vpc_endpoint_table()
```

```
Creating
Trying to attach S3 endpoints to the following route tables: ['rtb-0f4c136a']
S3 endpoint already exists.
```

At this point, you're done with initializing the training and are ready to move on to set up the training environment (p. 48).

Set Up the Training Environment

Setting up the environment for training your AWS DeepRacer model involves selecting a race track, a reward function and the associated action space, as well as hyperparameters used for training.

The notebook uses the default settings for these. To view the default settings, uncomment relevant parts and then run the code cell under **Configure the preset for RL algorithm**. For example, to view the code listing of the reward function, run the code cell as follows:

Configure the preset for RL algorithm

The parameters that configure the RL training job are defined in `src/markov/presets/` . Using the preset file, you can define agent parameters to select the specific agent algorithm. We suggest using Clipped PPO for this example.
You can edit this file to modify algorithm parameters like learning_rate, neural network structure, batch_size, discount factor etc.

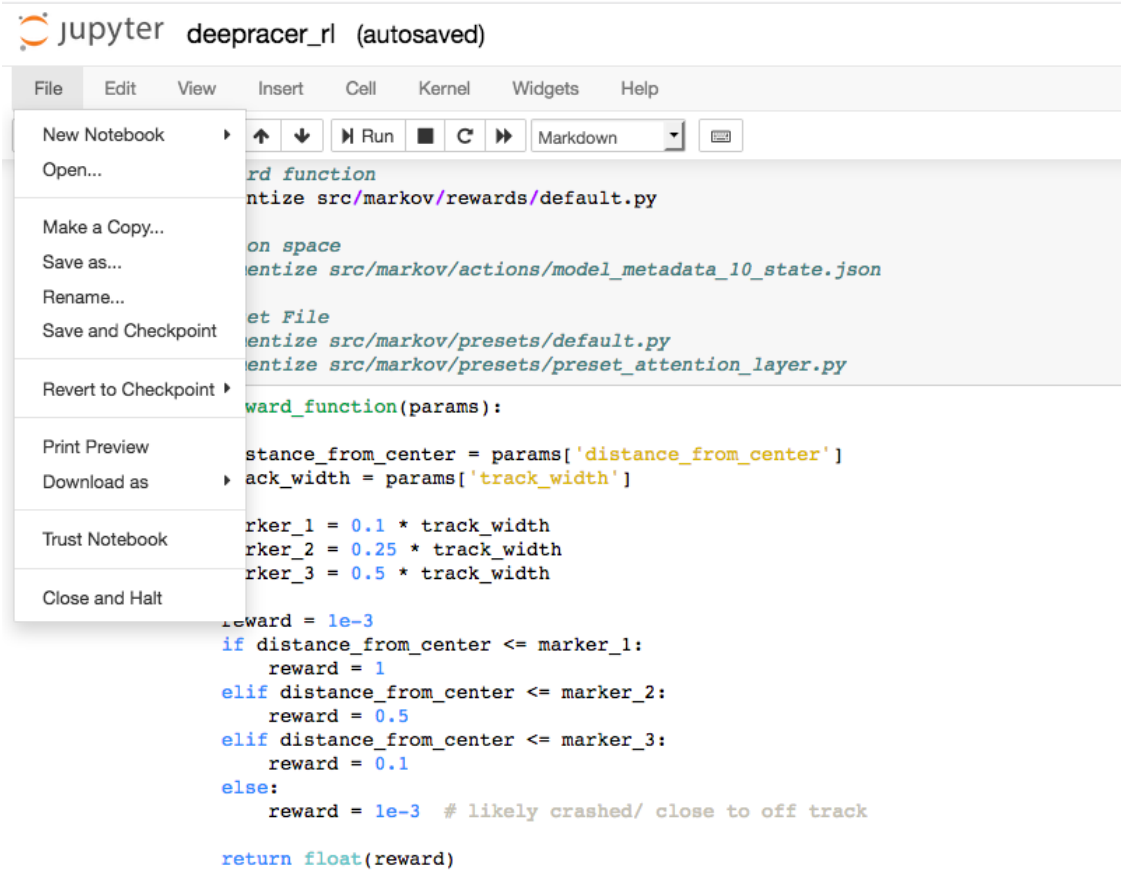
In [10]:
```
# Uncomment the pygmentize code lines to see the code

# Environmental File
#!pygmentize src/markov/environments/deepracer_racetrack_env.py

# Reward function
!pygmentize src/markov/rewards/default.py

# Action space
#!pygmentize src/markov/actions/model_metadata_10_state.json

# Preset File
#!pygmentize src/markov/presets/default.py
#!pygmentize src/markov/presets/preset_attention_layer.py
```

```
def reward_function(params):

    distance_from_center = params['distance_from_center']
    track_width = params['track_width']

    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    reward = 1e-3
    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 float(reward)
```

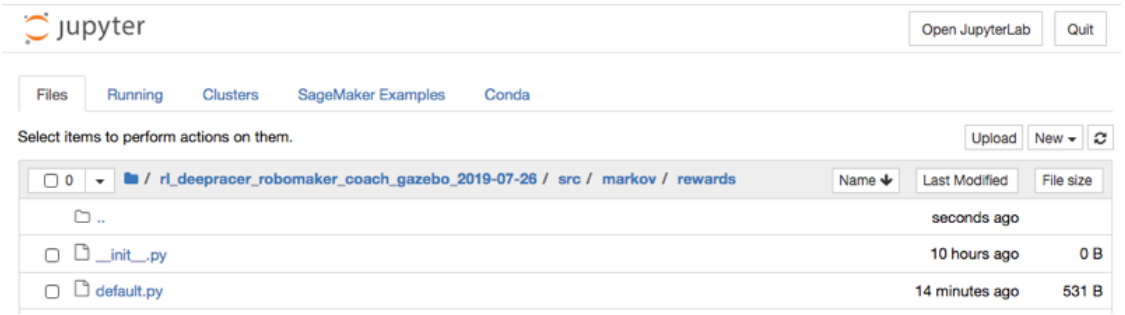
If you decide to use the default settings, copy the files to the S3 bucket. To modify any of the files, follow the steps below, changing the file name and directory for anything other than the default reward function.

To modify the reward function in the *default.py* file:

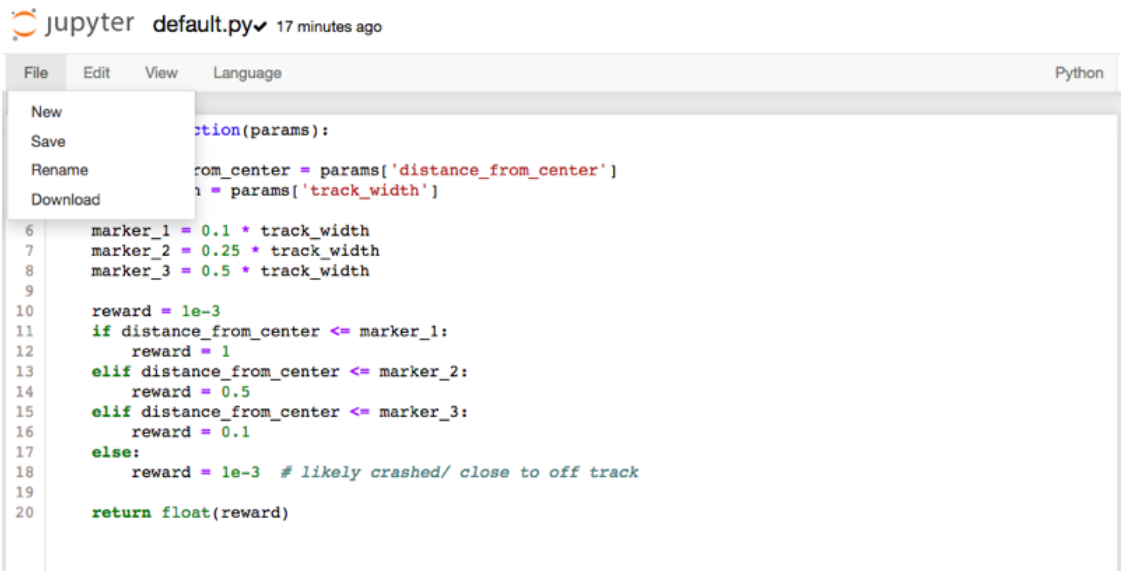
1. Choose **File** menu on the top of the notebook instance page and then choose **Open...**.



2. Navigate to the *src/markov/rewards* folder and choose *default.py* to open the file.



3. Edit the file as you see fit. After finishing editing the file, choose **File->Save** to save the update.



Notice that the environment file is shared by both Amazon SageMaker and AWS RoboMaker, also known as nodes. When it's used by Amazon SageMaker, the `node_type` is `SAGEMAKER_TRAINING_WORKER`. When it's used by AWS RoboMaker, the `node_type` is `SIMULATION_WORKER`.

Train Your AWS DeepRacer Model

Training your model with Amazon SageMaker and AWS RoboMaker amounts to executing the code in the *training_worker.py* file under the notebook's *src* directory. The *training_worker.py* file is designated as the entry point of your training job.

The training process involves using AWS RoboMaker to emulate driving experiences in the environment, relaying the experiences at fixed intervals to Amazon SageMaker as input to train the deep neural network, and updating the network weights to an S3 location.

While the training is in progress, you can have specified training metrics logged to Amazon CloudWatch Logs or displayed to the AWS RoboMaker terminal.

To train your AWS DeepRacer model

1. Run the code cell under **Copy custom files to S3 bucket so that sagemaker & robomaker can pick it up** copy the environment files to S3 .

Copy custom files to S3 bucket so that sagemaker & robomaker can pick it up

```
In [11]: s3_location = "s3://%s/%s" % (s3_bucket, s3_prefix)
         print(s3_location)

         # Clean up the previously uploaded files
         !aws s3 rm --recursive {s3_location}

         # Make any changes to the environment and preset files below and upload these files
         !aws s3 cp src/markov/environments/deepracer_racetrack_env.py {s3_location}/environments/deepra

         !aws s3 cp src/markov/rewards/default.py {s3_location}/rewards/reward_function.py

         !aws s3 cp src/markov/actions/model_metadata_10_state.json {s3_location}/model_metadata.json

         !aws s3 cp src/markov/presets/default.py {s3_location}/presets/preset.py
         #!aws s3 cp src/markov/presets/preset_attention_layer.py {s3_location}/presets/preset.py
```

```
s3://sagemaker-us-west-2-738575810317/deepracer-notebook-sagemaker-190726-210602
upload: src/markov/environments/deepracer_racetrack_env.py to s3://sagemaker-us-west-2-738575
810317/deepracer-notebook-sagemaker-190726-210602/environments/deepracer_racetrack_env.py
upload: src/markov/rewards/default.py to s3://sagemaker-us-west-2-738575810317/deepracer-note
book-sagemaker-190726-210602/rewards/reward_function.py
upload: src/markov/actions/model_metadata_10_state.json to s3://sagemaker-us-west-2-738575810
317/deepracer-notebook-sagemaker-190726-210602/model_metadata.json
upload: src/markov/presets/default.py to s3://sagemaker-us-west-2-738575810317/deepracer-note
book-sagemaker-190726-210602/presets/preset.py
```

2. To start an Amazon SageMaker job to train your AWS DeepRacer model, do the following:

   a. Run the first code cell under **Train the RL model using the Python SDK Script mode** to define training metrics to watch in either CloudWatch Logs or in an AWS RoboMaker console window.

```
In [60]: metric_definitions = [
             # Training> Name=main_level/agent, Worker=0, Episode=19, Total reward=-102.88, Steps=19019,
             {'Name': 'reward-training',
              'Regex': '^Training>.*Total reward=(.*?),'},

             # Policy training> Surrogate loss=-0.32664725184440613, KL divergence=7.255815035023261e-06
             {'Name': 'ppo-surrogate-loss',
              'Regex': '^Policy training>.*Surrogate loss=(.*?),'},
              {'Name': 'ppo-entropy',
              'Regex': '^Policy training>.*Entropy=(.*?),'},

             # Testing> Name=main_level/agent, Worker=0, Episode=19, Total reward=1359.12, Steps=20015,
             {'Name': 'reward-testing',
              'Regex': '^Testing>.*Total reward=(.*?),'},
         ]
```

   You can watch the specified metrics to monitor the training and to find out the effectiveness of your chosen reward function in CloudWatch Logs or using an AWS RoboMaker terminal.

   b. Run the second code cell under **Train the RL model using the Python SDK Script mode** to start an Amazon SageMaker training job for your model.

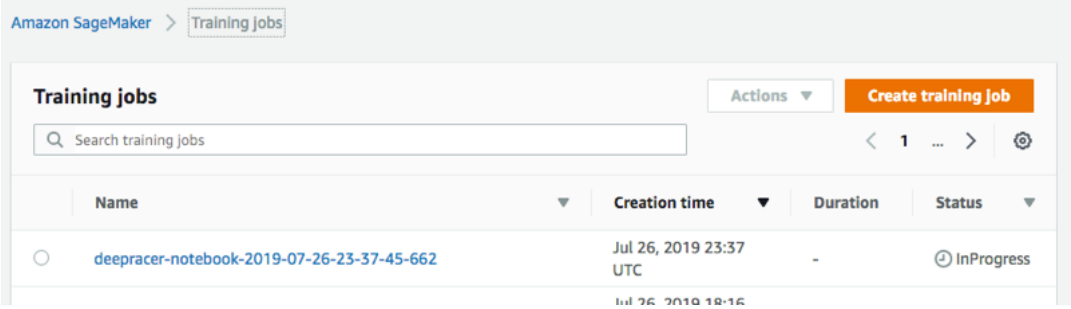
We use the RLEstimator for training RL jobs.

1. Specify the source directory which has the environment file, preset and training code.
2. Specify the entry point as the training code
3. Specify the choice of RL toolkit and framework. This automatically resolves to the ECR path for the RL Container.
4. Define the training parameters such as the instance count, instance type, job name, s3_bucket and s3_prefix for storing model checkpoints and metadata. **Only 1 training instance is supported for now.**
5. Set the RLCOACH_PRESET as "deepracer" for this example.
6. Define the metrics definitions that you are interested in capturing in your logs. These can also be visualized in CloudWatch and SageMaker Notebooks.

```
In [21]: estimator = RLEstimator(entry_point="training_worker.py",
                                 source_dir='src',
                                 image_name=custom_image_name,
                                 dependencies=["common/"],
                                 role=sagemaker_role,
                                 train_instance_type=instance_type,
                                 train_instance_count=1,
                                 output_path=s3_output_path,
                                 base_job_name=job_name_prefix,
                                 metric_definitions=metric_definitions,
                                 train_max_run=job_duration_in_seconds,
                                 hyperparameters={
                                     "s3_bucket": s3_bucket,
                                     "s3_prefix": s3_prefix,
                                     "aws_region": aws_region,
                                     "preset_s3_key": "%s/presets/preset.py"% s3_prefix,
                                     "model_metadata_s3_key": "%s/model_metadata.json" % s3_prefix,
                                     "environment_s3_key": "%s/environments/deepracer_racetrack_env.py"
                                 },
                                 subnets=deepracer_subnets,
                                 security_group_ids=deepracer_security_groups,
                             )

estimator.fit(wait=False)
job_name = estimator.latest_training_job.job_name
print("Training job: %s" % job_name)
```

```
Training job: deepracer-notebook-2019-07-26-23-37-45-662
```

This Amazon SageMaker training job uses the TensorFlow framework and runs on a specified EC2 compute instance type. The output lists the job name. You can track the status of this training job in Amazon SageMaker.



3. To create an environment simulation job in AWS RoboMaker, run the code cells under **Start the RoboMaker job** and **Create Simulation Application**.
4. To start the simulation on AWS RoboMaker and share the simulated data, run the code cell under **Launch the Simulation job on RoboMaker**.

Launch the Simulation job on RoboMaker

We create AWS RoboMaker Simulation Jobs that simulates the environment and shares this data with SageMaker for training.

```
In [26]: num_simulation_workers = 1

envriron_vars = {
    "WORLD_NAME": "reinvent_base",
    "KINESIS_VIDEO_STREAM_NAME": "SilverstoneStream",
    "SAGEMAKER_SHARED_S3_BUCKET": s3_bucket,
    "SAGEMAKER_SHARED_S3_PREFIX": s3_prefix,
    "TRAINING_JOB_ARN": job_name,
    "APP_REGION": aws_region,
    "METRIC_NAME": "TrainingRewardScore",
    "METRIC_NAMESPACE": "AWSDeepRacer",
    "REWARD_FILE_S3_KEY": "%s/rewards/reward_function.py" % s3_prefix,
    "MODEL_METADATA_FILE_S3_KEY": "%s/model_metadata.json" % s3_prefix,
    "METRICS_S3_BUCKET": s3_bucket,
    "METRICS_S3_OBJECT_KEY": s3_bucket + "/training_metrics.json",
    "TARGET_REWARD_SCORE": "None",
    "NUMBER_OF_EPISODES": "0",
    "ROBOMAKER_SIMULATION_JOB_ACCOUNT_ID": account_id
}

simulation_application = {"application":simulation_app_arn,
                          "launchConfig": {"packageName": "deepracer_simulation_environment",
                                           "launchFile": "distributed_training.launch",
                                           "environmentVariables": envriron_vars}
                         }


vpcConfig = {"subnets": deepracer_subnets,
             "securityGroups": deepracer_security_groups,
             "assignPublicIp": True}

client_request_token = strftime("%Y-%m-%d-%H-%M-%S", gmtime())

responses = []
for job_no in range(num_simulation_workers):
    response =  robomaker.create_simulation_job(iamRole=sagemaker_role,
                                                clientRequestToken=client_request_token,
                                                maxJobDurationInSeconds=job_duration_in_seconds,
                                                failureBehavior="Continue",
                                                simulationApplications=[simulation_application],
                                                vpcConfig=vpcConfig
                                                )
    responses.append(response)

print("Created the following jobs:")
job_arns = [response["arn"] for response in responses]
for response in responses:
    print("Job ARN", response["arn"])
```

```
Created the following jobs:
Job ARN arn:aws:robomaker:us-west-2:738575810317:simulation-job/sim-02nk1r56hq67
```

5. To watch the simulations in AWS RoboMaker, run the code cell under **Visualizing the simulations in RoboMaker** and then choose the **Simulation 1** link from the output.

```
In [67]: display(Markdown(generate_robomaker_links(job_arns, aws_region)))
```

Click on the following links for visualization of simulation jobs on RoboMaker Console

- Simulation 1

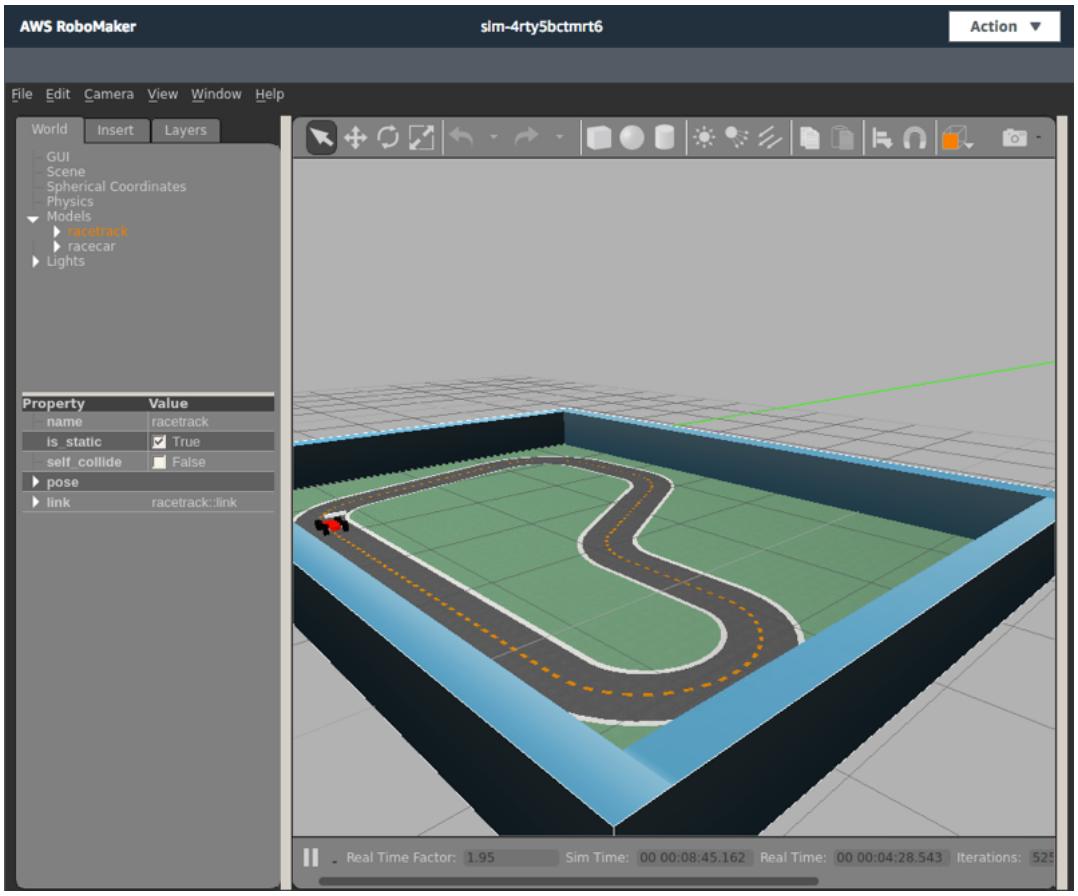
Alternatively, you can go to the AWS RoboMaker console directly to open the simulation job.

After the simulation job is initialized, the AWS RoboMaker console makes available the following visualization utilities:

- **Gazebo**: an emulation of 3D worlds for simulating autonomous vehicle in the chosen track.
- **rqt**: Qt-based framework and plugins for ROS GUI development.

- **ivis**: ROS visualizer for displaying the field of vision as captured by the vehicle's front-facing camera.
- **Terminal**: A terminal application to provide command line access on the simulation job host.

a. To view your vehicle learning in the 3D simulation, double-click or tap **Gazebo**.

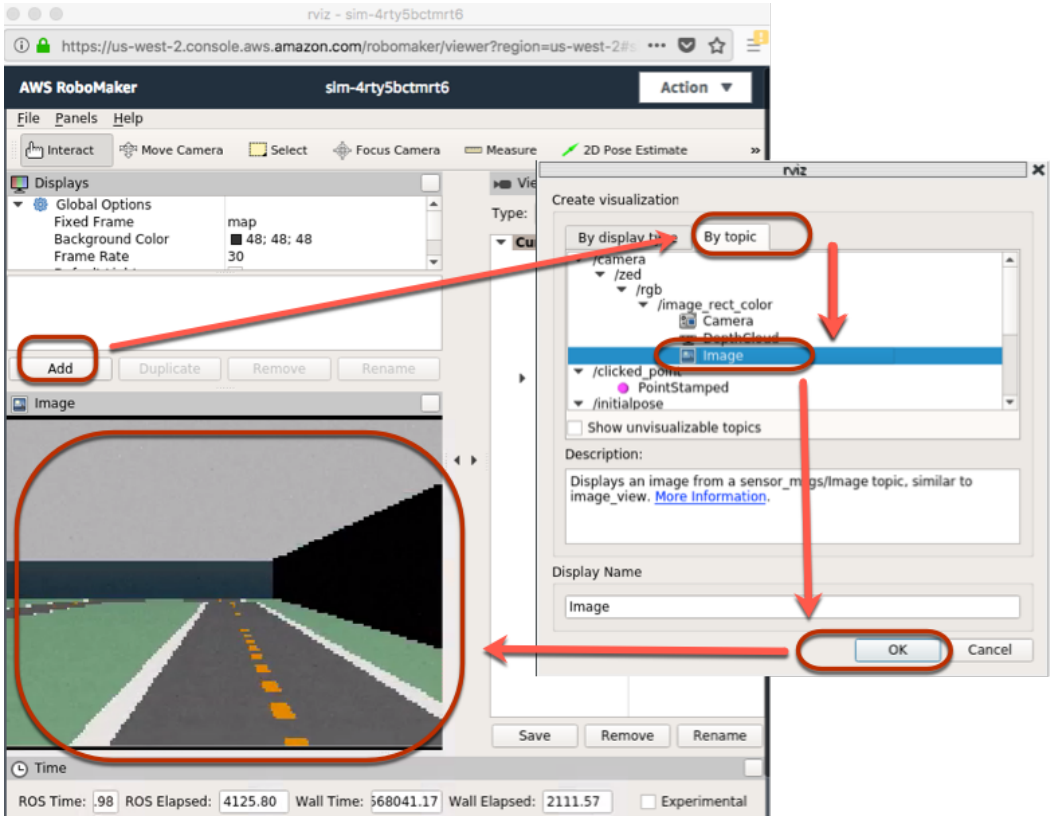


You watch the simulated vehicle navigate along the track in repeated trials starting from the starting point and ending at going off-track or reaching the finishing line. In the beginning, the vehicle can stay on the track briefly. As time goes on, it learns to stay on the track longer.

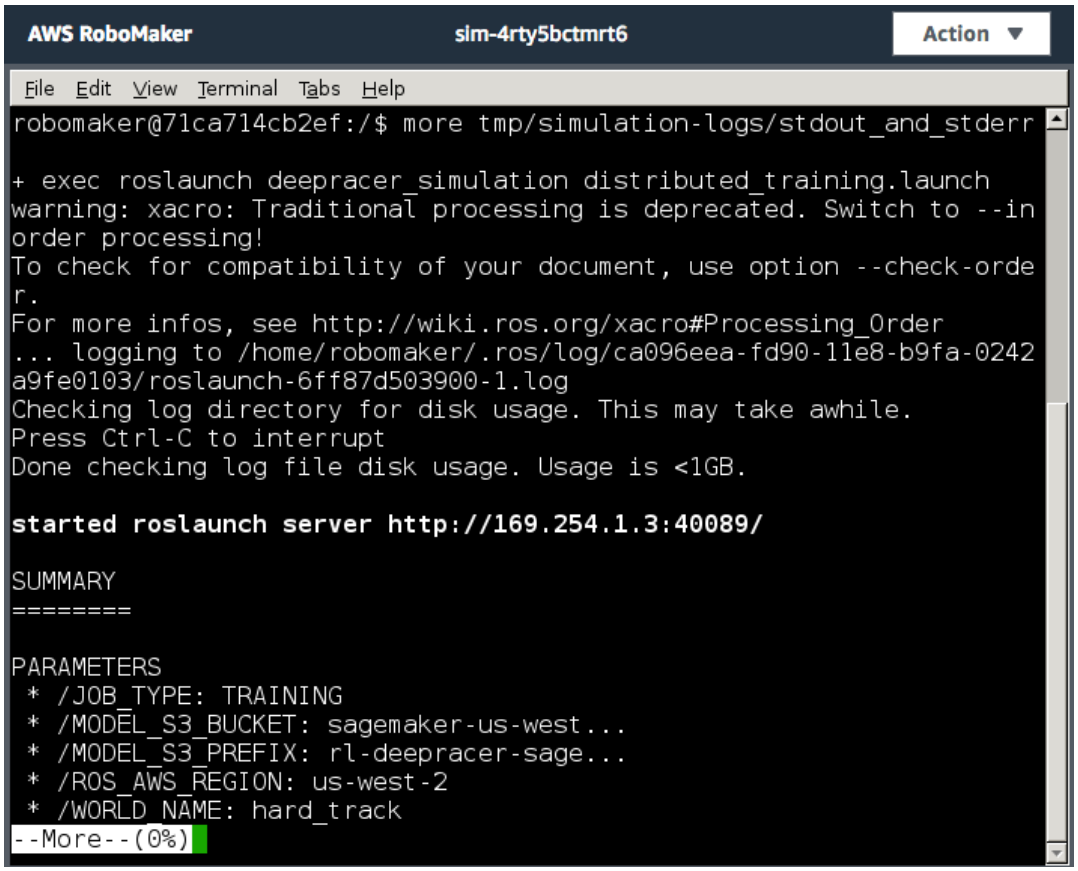
b. To access **rqt** utilities, double-click or tap **rqt** and choose a plugin.

For more information about the plugins, see AWS RoboMaker plugins.

c. To view the front-facing vision of the vehicle, double-click or tap **rvis**. Choose **Add** to create a visualization. And then choose the **By topic** tab, scroll down to choose **/camera/zed/rgb/image_rec_color/Image**, choose OK.



d. To use the terminal, double-click or tap **Terminal** to open a terminal window on the simulation job host and type appropriate shell command.

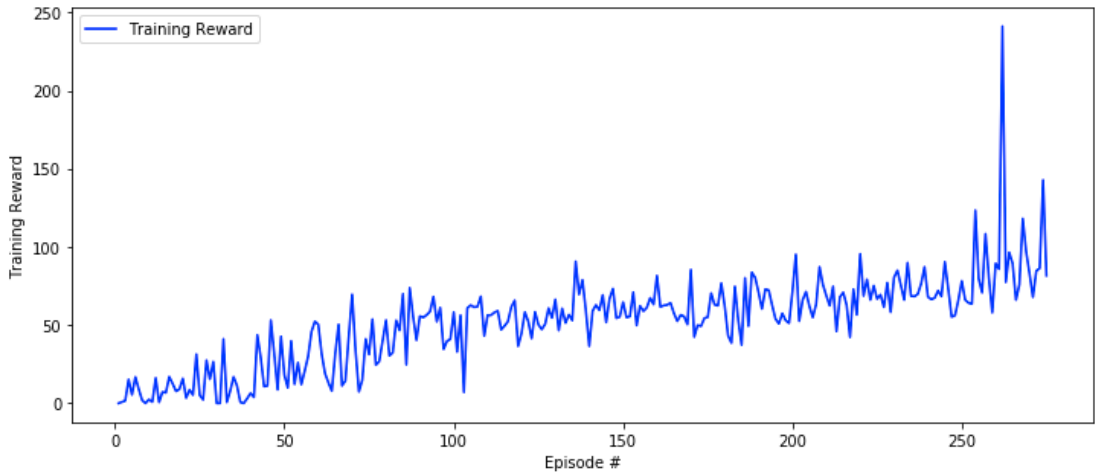


With the simulation job host terminal opened, you can call Linux shell commands to view (`more` or `tail`) the logs or performing other operations.

To view the reward of the last 10 steps in the simulation logs, you can type the following shell command in the terminal:

```
tail /tmp/simulation-logs/stdout_and_stderr
```

6. To visualize the training performance, run the two code cells under **Plot metrics for training job**. When all is done successfully, you see a plot of **Training reward** vs **Episode #** similar to the following.



In this particular example, the training reward appears to start to plateau. Perhaps more data are needed to verify if it's true. If the training job is running, you can run the code cell under **Plot metrics for training job** again to include more recent data into the plot. If they persist, the onset of large fluctuations can indicate certain deficiency in the reward function. Thus, you might update the reward function definition. In any case, you need to collect more data with more training.

After training has elapsed the specified amount of time, you can locate the trained model artifacts in the training job's S3 bucket, e.g., *s3://<bucket>/<sagemaker-training-job-name>/output/model.tar.gz*. Download the model artifacts file, copy it to a USB drive and then transfer the file to your AWS DeepRacer vehicle's compute module.

- To clean up when you're done with training and no longer need the AWS RoboMaker and Amazon SageMaker resources, run the two code cells under **Clean Up**.
- To evaluate the model that has been trained thus far, run the code cell under **Evaluation**.

  If successful, a simulation job is created for the task in AWS RoboMaker. Make note of the job name in the output below the code cell. You may need it to open the simulation job in the AWS RoboMaker console. This simulation job is similar to the simulation job for training. It provides the same utilities for you view the evaluation in progress in the AWS RoboMaker console. In particular, you can watch the evaluation trials in **Gazebo**.
- When you're done with evaluating the model and want to terminate the simulation application, run the code cell under **Clean Up Simulation Application Resource**.

AWS DeepRacer Reward Function Reference

The following is the technical reference of the AWS DeepRacer reward function.

Topics

- Input Parameters of the AWS DeepRacer Reward Function (p. 58)
- AWS DeepRacer Reward Function Examples (p. 73)

Input Parameters of the AWS DeepRacer Reward Function

The AWS DeepRacer reward function takes a dictionary object as the input.

```
def reward_function(params) :

    reward = ...

    return float(reward)
```

The `params` dictionary object contains the following key-value pairs:

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

}
```

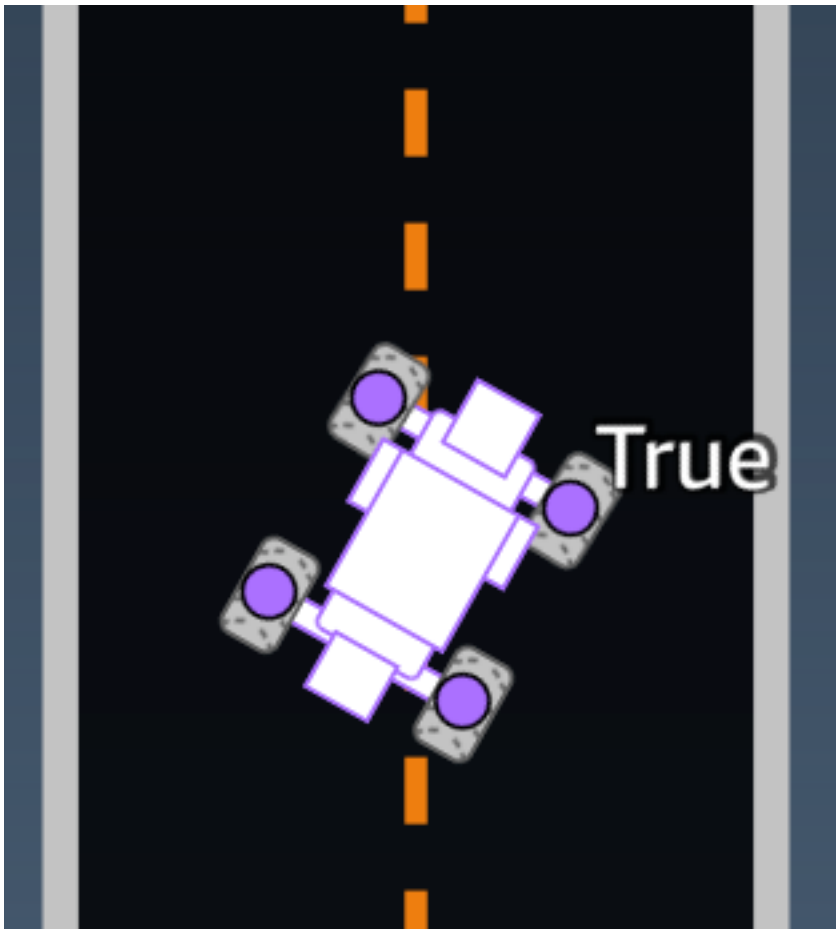
A more detailed technical reference of the input parameters is as follows.

all_wheels_on_track

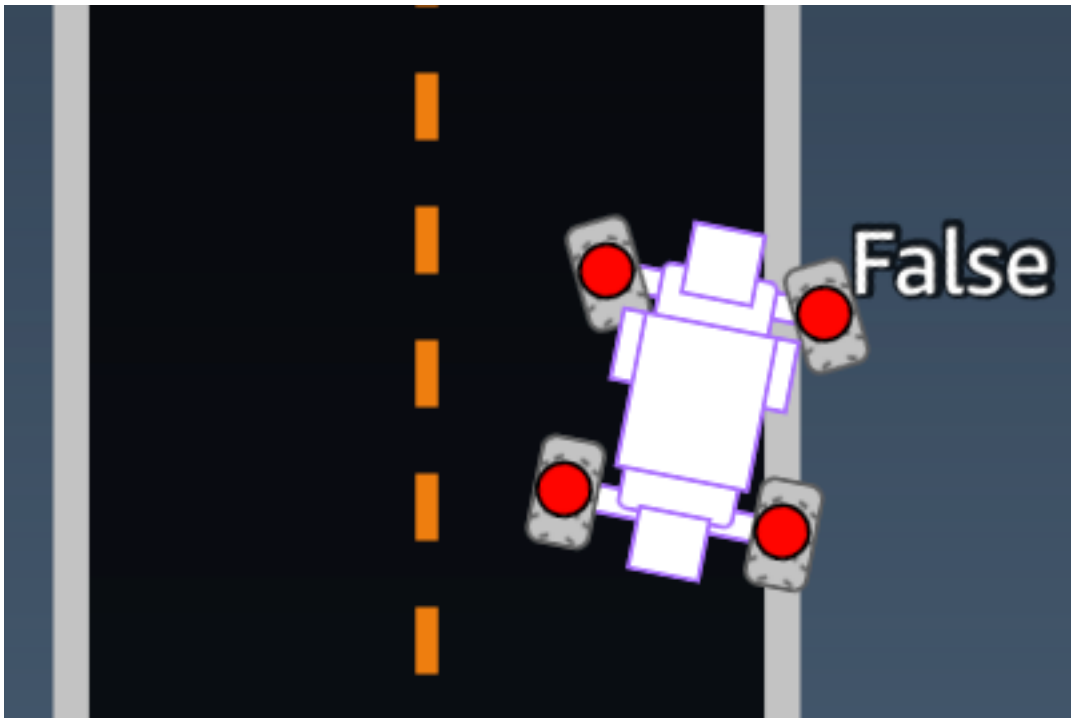
Type: `Boolean`

Range: `(True:False)`

A `Boolean` flag to indicate whether the agent is on-track or off-track. It's off-track (`False`) if any of its wheels are outside of the track borders. It's on-track (`True`) if all of the wheels are inside the two track borders. The following illustration shows that the agent is on-track.



The following illustration shows that the agent is off-track.



Example: *A reward function using the `all_wheels_on_track` parameter*

```
define reward_function(params):
    ################################################################################
    '''
    Example of using all_wheels_on_track and speed
    '''

    # Read input variables
    all_wheels_on_track = params['all_wheels_on_track']
    speed = params['speed']

    # Set the speed threshold based your action space
    SPEED_THRESHOLD = 1.0

    if not all_wheels_on_track:
        # Penalize if the car goes off track
        reward = 1e-3
    elif speed < SPEED_THRESHOLD:
        # Penalize if the car goes too slow
        reward = 0.5
    else:
        # High reward if the car stays on track and goes fast
        reward = 1.0

    return reward
```

closest_waypoints

Type: `[int, int]`

Range: `[(0:Max-1),(1:Max-1)]`

The zero-based indices of the two neighboring `waypoints` closest to the agent's current position of `(x, y)`. The distance is measured by the Euclidean distance from the center of the agent. The first

element refers to the closest waypoint behind the agent and the second element refers the closest waypoint in front of the agent. `Max` is the length of the waypoints list. In the illustration shown in waypoints (p. 72), the `closest_waypoints` would be `[16, 17]`.

Example: A reward function using the `closest_waypoints` parameter.

The following example reward function demonstrates how to use `waypoints` and `closest_waypoints` as well as `heading` to calculate immediate rewards.

```
def reward_function(params):
    ################################################################################
    '''
    Example of using waypoints and heading to make the car in the right direction
    '''

    import math

    # Read input variables
    waypoints = params['waypoints']
    closest_waypoints = params['closest_waypoints']
    heading = params['heading']

    # Initialize the reward with typical value
    reward = 1.0

    # Calculate the direction of the center line based on the closest waypoints
    next_point = waypoints[closest_waypoints[1]]
    prev_point = waypoints[closest_waypoints[0]]

    # Calculate the direction in radius, arctan2(dy, dx), the result is (-pi, pi) in
 radians
    track_direction = math.atan2(next_point[1] - prev_point[1], next_point[0] -
 prev_point[0])
    # Convert to degree
    track_direction = math.degrees(track_direction)

    # Calculate the difference between the track direction and the heading direction of the
 car
    direction_diff = abs(track_direction - heading)
    if direction_diff > 180:
        direction_diff = 360 - direction_diff

    # Penalize the reward if the difference is too large
    DIRECTION_THRESHOLD = 10.0
    if direction_diff > DIRECTION_THRESHOLD:
        reward *= 0.5

    return reward
```

closest_objects

Type: `[int, int]`

Range: `[(0:len(object_locations)-1), (0:len(object_locations)-1]`

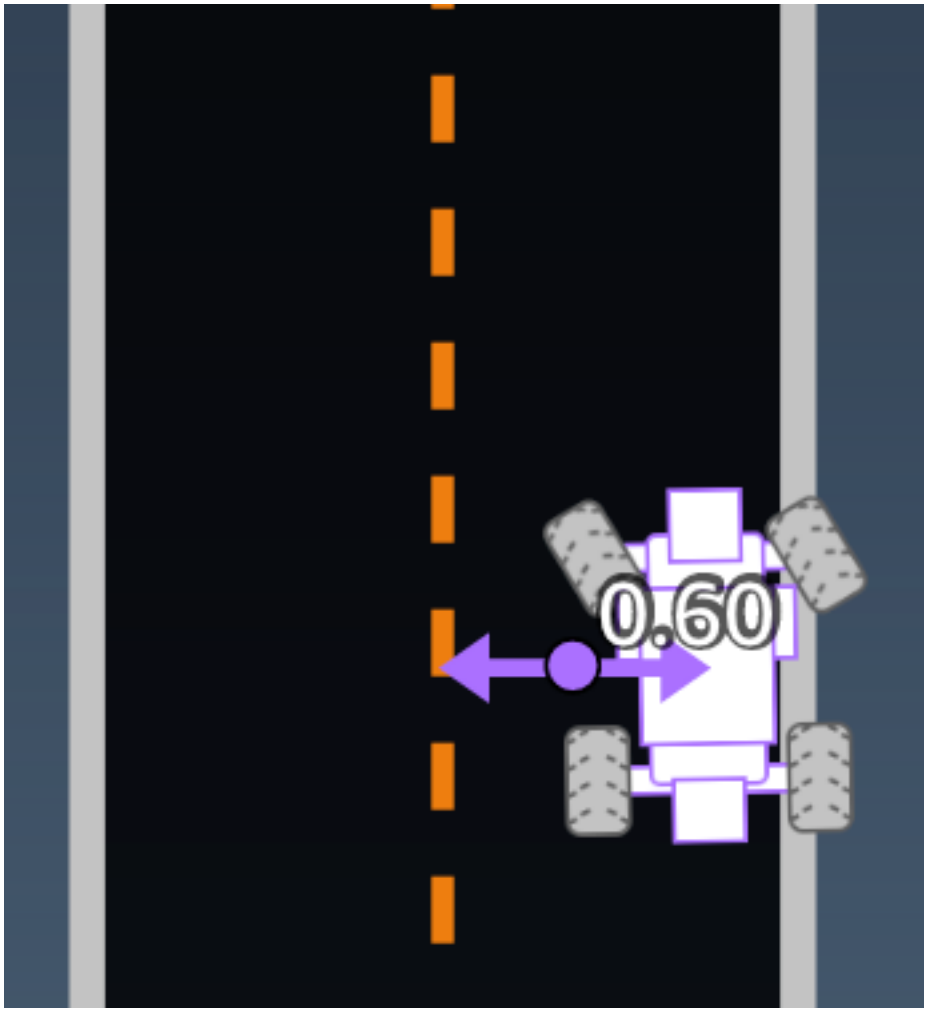
The zero-based indices of the two closest objects to the agent's current position of (x, y). The first index refers to the closest object behind the agent, and the second index refers to the closest object in front of the agent. If there is only one object, both indices are 0.

distance_from_center

Type: `float`

Range: `0:~track_width/2`

Displacement, in meters, between the agent center and the track center. The observable maximum displacement occurs when any of the agent's wheels are outside a track border and, depending on the width of the track border, can be slightly smaller or larger than half the `track_width`.



Example: *A reward function using the `distance_from_center` parameter*

```
def reward_function(params):
    ################################################################################
    '''
    Example of using distance from the center
    '''

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

    # Penalize if the car is too far away from the center
    marker_1 = 0.1 * track_width
    marker_2 = 0.5 * track_width

    if distance_from_center <= marker_1:
        reward = 1.0
    elif distance_from_center <= marker_2:
```

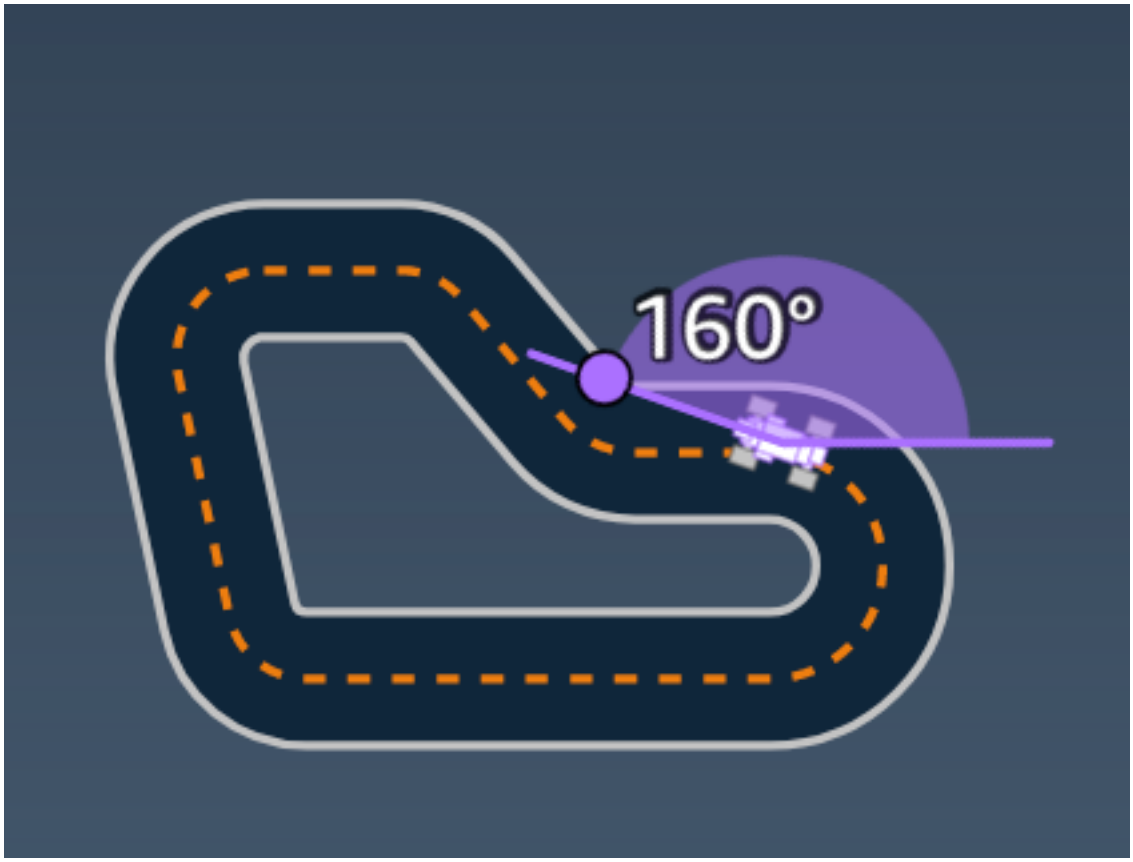
```
        reward = 0.5
    else:
        reward = 1e-3  # likely crashed/ close to off track

    return reward
```

heading

Type: `float`

Range: `-180:+180`

Heading direction, in degrees, of the agent with respect to the x-axis of the coordinate system.



Example: *A reward function using the `heading` parameter*

For more information, see `closest_waypoints` (p. 61).

is_crashed

Type: `Boolean`

Range: `(True:False)`

A Boolean flag to indicate whether the agent has crashed into another object (`True`) or not (`False`) as a termination status.

is_left_of_center

Type: `Boolean`

Range: `[True : False]`

A `Boolean` flag to indicate if the agent is on the left side to the track center (`True`) or on the right side (`False`).

is_offtrack

Type: `Boolean`

Range: `(True:False)`

A Boolean flag to indicate whether the agent has off track (True) or not (False) as a termination status.

is_reversed

Type: `Boolean`

Range: `[True:False]`

A Boolean flag to indicate if the agent is driving on clock-wise (True) or counter clock-wise (False).

It's used when you enable direction change for each episode.

objects_distance

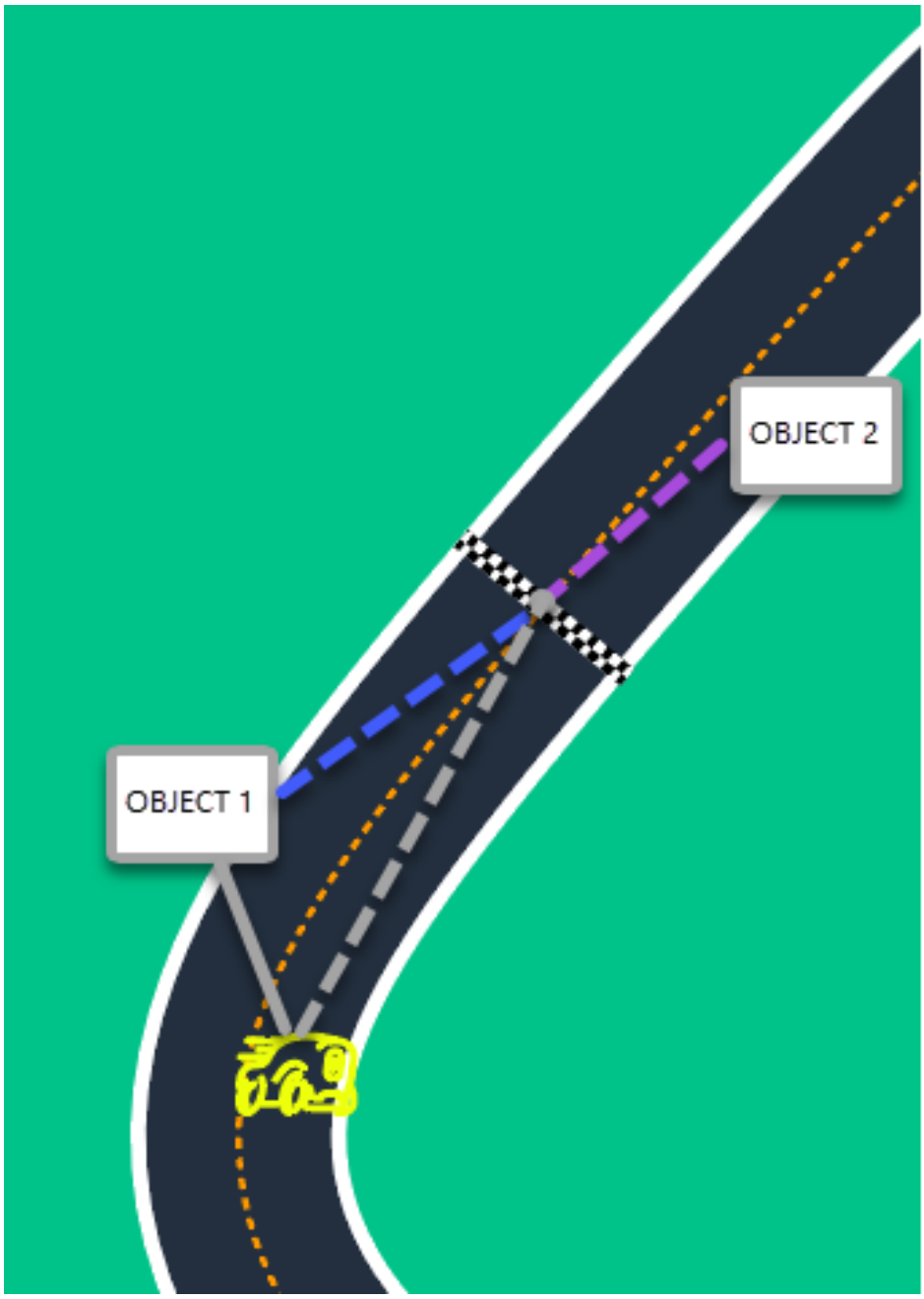
Type: `[float, … ]`

Range: `[(0:track_length), … ]`

A list of the distances between objects in the environment in relation to the starting line. The i^{th} element measures the distance in meters between the i^{th} object and the starting line along the track center line.

To index the distance between a single object and the agent, use:

```
abs(params["objects_distance"][index] -
(params["progress"]/100.0)*params["track_length"])
```



Note
abs | (var1) - (var2)| = how close the car is to an object, WHEN var1 = ["objects_distance"][index] and var2 = params["progress"]*params["track_length"]
To get an index of the closest object in front of the vehicle and the closest object behind the vehicle, use the "closest_objects" parameter.

objects_heading

Type: `[float, … ]`

Range: `[(-180:180), … ]`

List of the headings of objects in degrees. The i^{th} element measures the heading of the i^{th} object. For stationary objects, their headings are 0. For a bot vehicle, the corresponding element's value is the vehicle's heading angle.

objects_left_of_center

Type: `[Boolean, … ]`

Range: `[True|False, … ]`

List of Boolean flags. The i^{th} element value indicates whether the i^{th} object is to the left (True) or right (False) side of the track center.

objects_location

Type: `[(x,y), ...]`

Range: `[(0:N,0:N), ...]`

List of all object locations, each location is a tuple of (x, y (p. 72)).

The size of the list equals the number of objects on the track. Note the object could be the stationary obstacles, moving bot vehicles.

objects_speed

Type: `[float, … ]`

Range: `[(0:12.0), … ]`

List of speeds (meters per second) for the objects on the track. For stationary objects, their speeds are 0. For a bot vehicle, the value is the speed you set in training.

progress

Type: `float`

Range: `0:100`

Percentage of track completed.

Example: *A reward function using the `progress` parameter*

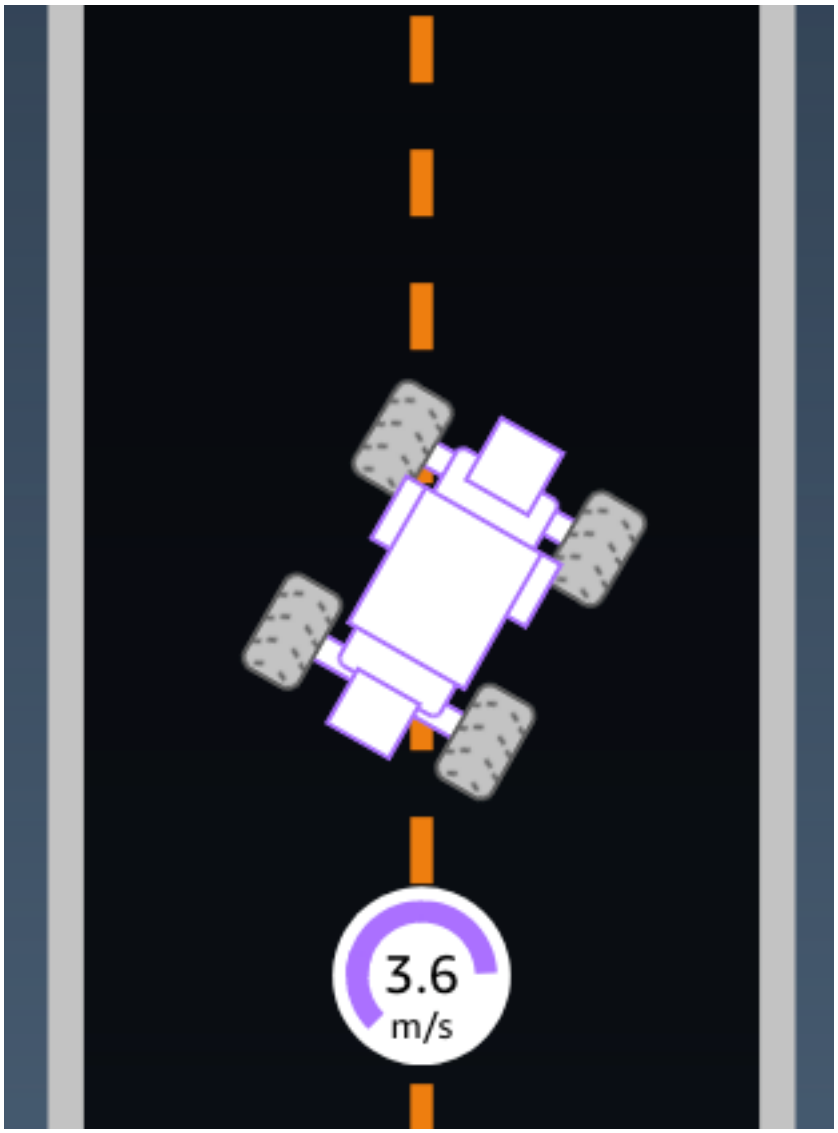
For more information, see steps (p. 70).

speed

Type: `float`

Range: `0.0:5.0`

The observed speed of the agent, in meters per second (m/s).



Example: *A reward function using the* `speed` *parameter*

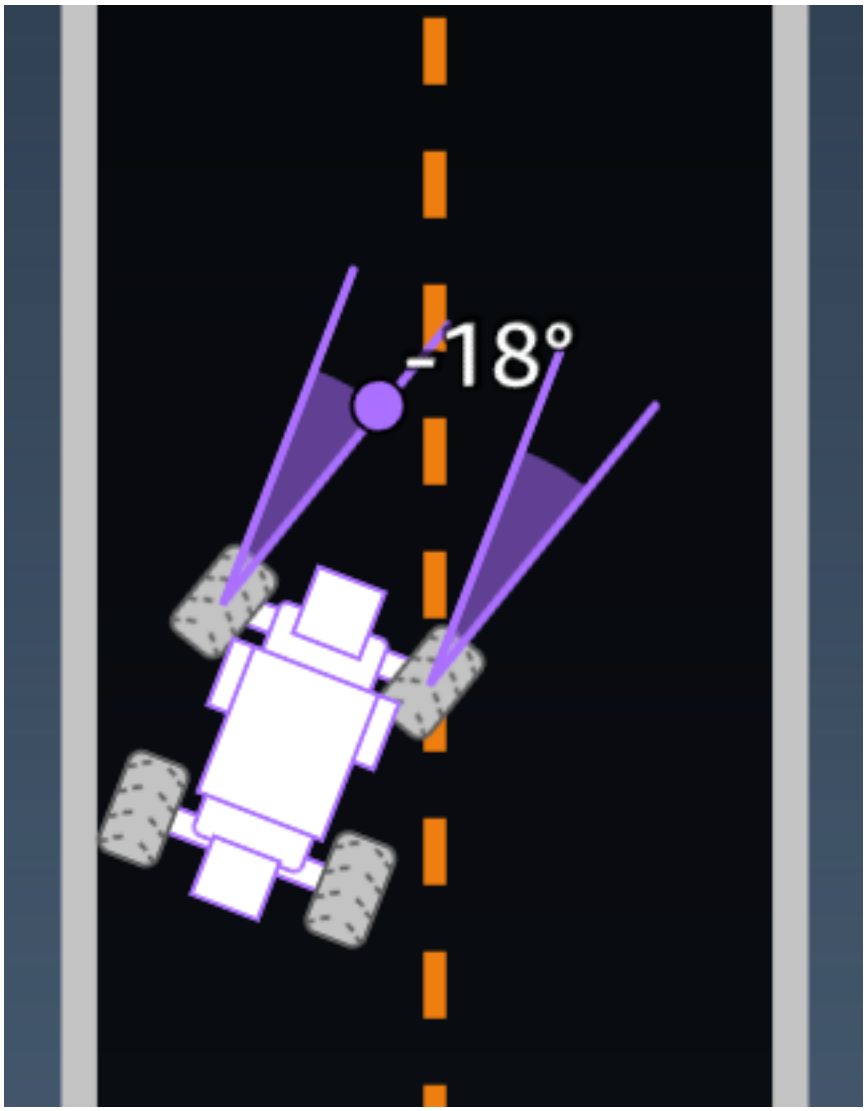
For more information, see all_wheels_on_track (p. 59).

steering_angle

Type: `float`

Range: `-30:30`

Steering angle, in degrees, of the front wheels from the center line of the agent. The negative sign (-) means steering to the right and the positive (+) sign means steering to the left. The agent center line is not necessarily parallel with the track center line as is shown in the following illustration.



Example: *A reward function using the* `steering_angle` *parameter*

```
def reward_function(params):
    '''
    Example of using steering angle
    '''

    # Read input variable
    steering = abs(params['steering_angle']) # We don't care whether it is left or right
 steering

    # Initialize the reward with typical value
    reward = 1.0

    # Penalize if car steer too much to prevent zigzag
    STEERING_THRESHOLD = 20.0
    if steering > ABS_STEERING_THRESHOLD:
        reward *= 0.8

    return reward
```

steps

Type: `int`

Range: $0:N_{step}$

Number of steps completed. A step corresponds to an action taken by the agent following the current policy.

Example: *A reward function using the `steps` parameter*

```
def reward_function(params):
    ################################################################################
    '''
    Example of using steps and progress
    '''

    # Read input variable
    steps = params['steps']
    progress = params['progress']

    # Total num of steps we want the car to finish the lap, it will vary depends on the
 track length
    TOTAL_NUM_STEPS = 300

    # Initialize the reward with typical value
    reward = 1.0

    # Give additional reward if the car pass every 100 steps faster than expected
    if (steps % 100) == 0 and progress > (steps / TOTAL_NUM_STEPS) * 100 :
        reward += 10.0

    return reward
```

track_length

Type: `float`

Range: $[0:L_{max}]$

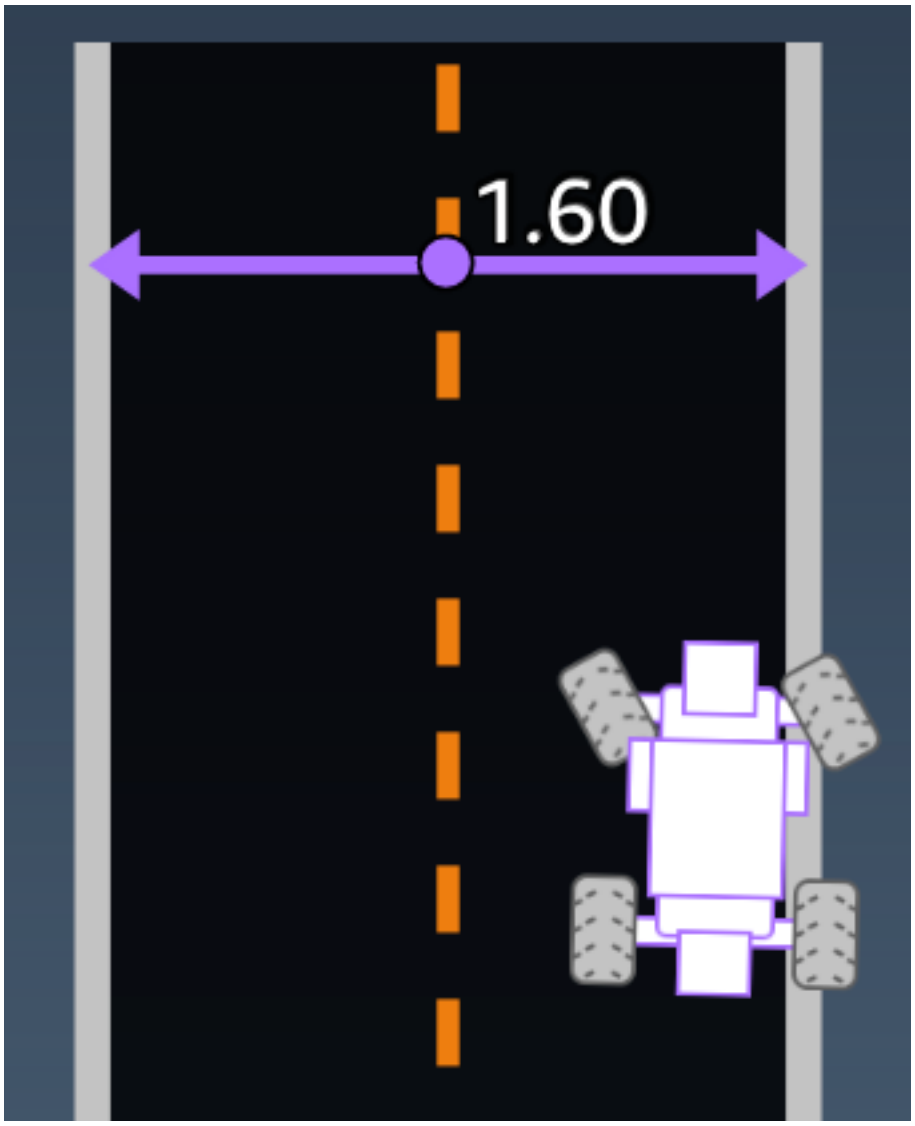
The track length in meters. L_{max} `is track-dependent.`

track_width

Type: `float`

Range: $0:D_{track}$

Track width in meters.



Example: *A reward function using the `track_width` parameter*

```
def reward_function(params):
    ################################################################################
    '''
    Example of using track width
    '''

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

    # Calculate the distance from each border
    distance_from_border = 0.5 * track_width - distance_from_center

    # Reward higher if the car stays inside the track borders
    if distance_from_border >= 0.05:
        reward *= 1.0
    else:
        reward = 1e-3 # Low reward if too close to the border or goes off the track
```

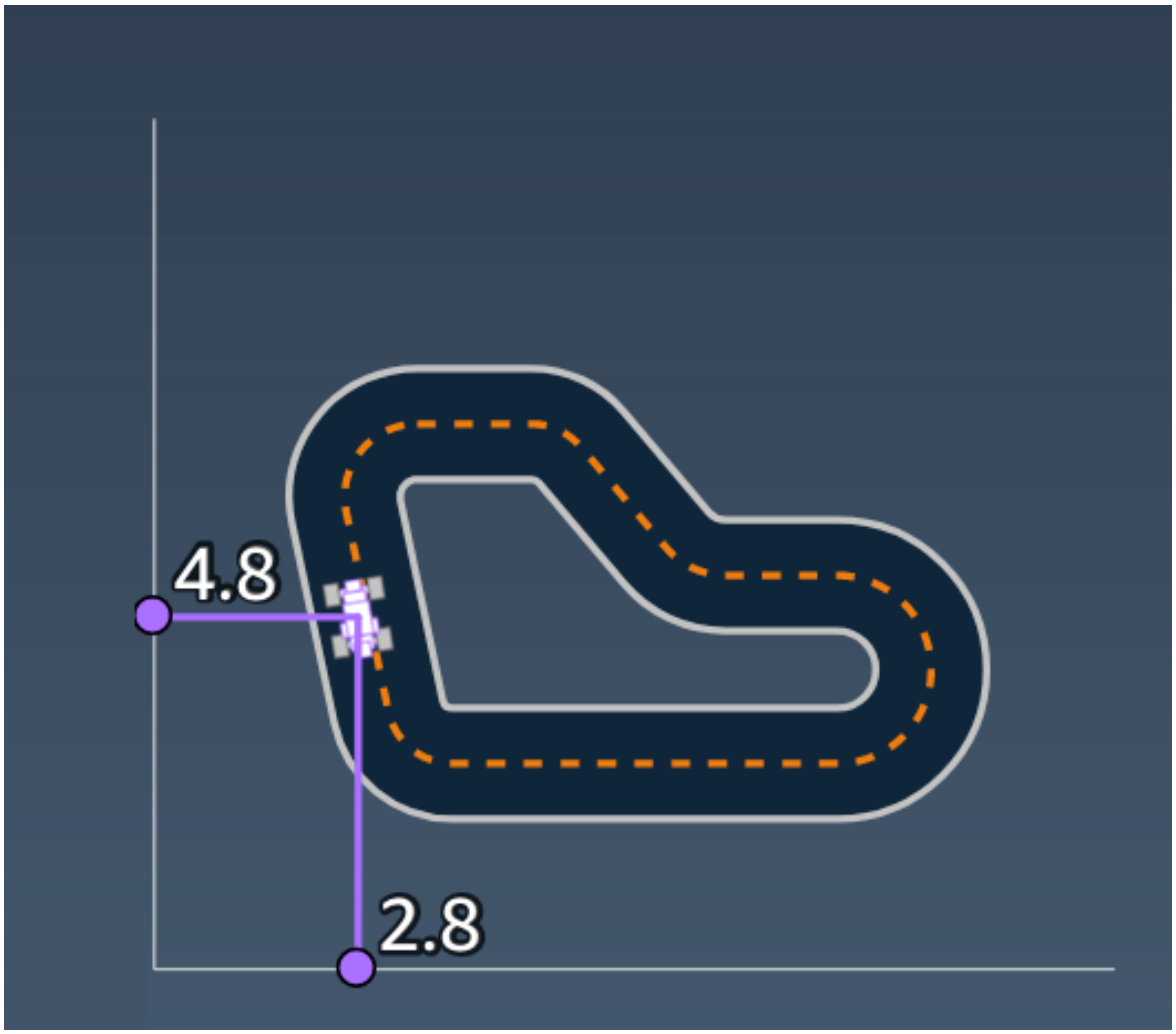
```
    return reward
```

x, y

Type: `float`

Range: `0:N`

Location, in meters, of the agent center along the x and y axes, of the simulated environment containing the track. The origin is at the lower-left corner of the simulated environment.

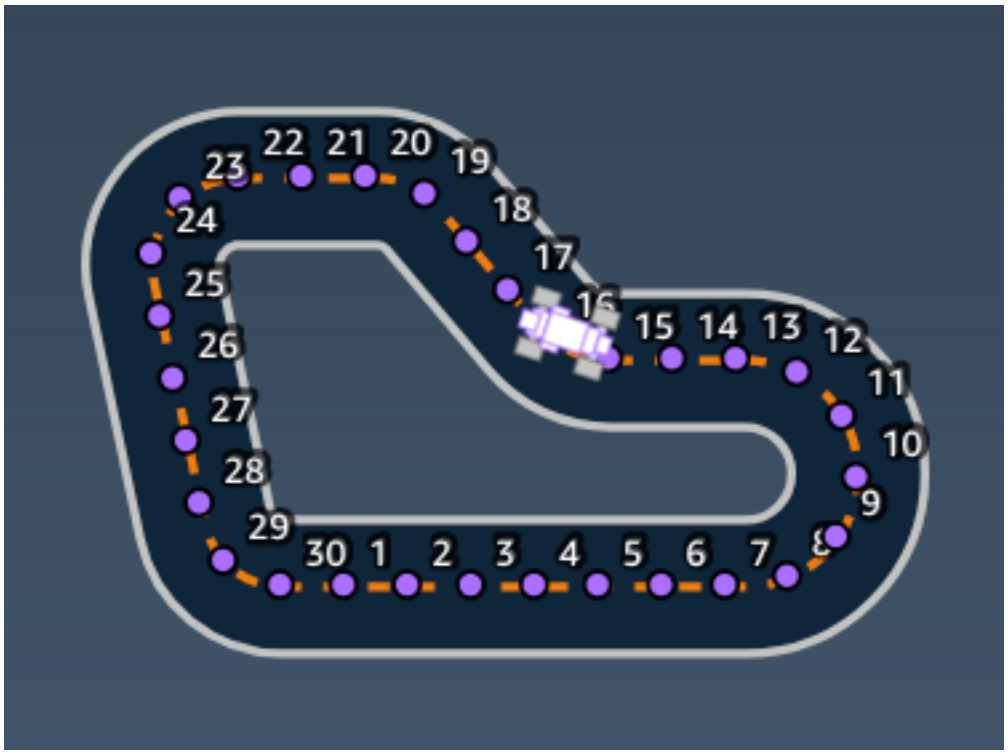


waypoints

Type: `list` of `[float, float]`

Range: `[[`$x_{w,0}$`,`$y_{w,0}$`]` … `[`$x_{w,Max-1}$`,` $y_{w,Max-1}$`]]`

An ordered list of track-dependent `Max` milestones along the track center. Each milestone is described by a coordinate of ($x_{w,i}$, $y_{w,i}$). For a looped track, the first and last waypoints are the same. For a straight or other non-looped track, the first and last waypoints are different.



Example *A reward function using the* `waypoints` *parameter*

For more information, see `closest_waypoints` (p. 61).

AWS DeepRacer Reward Function Examples

The following lists some examples of the AWS DeepRacer reward function.

Topics

- Example 1: Follow the Center Line in Time Trials (p. 73)
- Example 2: Stay Inside the Two Borders in Time Trials (p. 74)
- Example 3: Prevent Zig-Zag in Time Trials (p. 74)
- Example 4: Stay On One Lane without Crashing into Stationary Obstacles or Moving Vehicles (p. 75)

Example 1: Follow the Center Line in Time Trials

This example determines how far away the agent is from the center line, and gives higher reward if it is closer to the center of the track, encouraging the agent to closely follow the center line.

```
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
```

Example 2: Stay Inside the Two Borders in Time Trials

This example simply gives high rewards if the agent stays inside the borders, and let the agent figure out what is the best path to finish a lap. It is easy to program and understand, but likely takes longer to converge.

```
def reward_function(params):
    '''
    Example of rewarding the agent to stay inside the two borders of the track
    '''

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

    # Give a very low reward by default
    reward = 1e-3

    # Give a high reward if no wheels go off the track and
    # the car is somewhere in between the track borders
    if all_wheels_on_track and (0.5*track_width - distance_from_center) >= 0.05:
        reward = 1.0

    # Always return a float value
    return reward
```

Example 3: Prevent Zig-Zag in Time Trials

This example incentivizes the agent to follow the center line but penalizes with lower reward if it steers too much, which helps prevent zig-zag behavior. The agent learns to drive smoothly in the simulator and likely keeps the same behavior when deployed in the physical vehicle.

```
def reward_function(params):
    '''
    Example of penalize steering, which helps mitigate zig-zag behaviors
    '''

    # Read input parameters
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    steering = abs(params['steering_angle']) # Only need the absolute steering angle

    # Calculate 3 marks that are farther and father 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 = 1.0
    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

    # Steering penality threshold, change the number based on your action space setting
    ABS_STEERING_THRESHOLD = 15

    # Penalize reward if the car is steering too much
    if steering > ABS_STEERING_THRESHOLD:
        reward *= 0.8

    return float(reward)
```

Example 4: Stay On One Lane without Crashing into Stationary Obstacles or Moving Vehicles

This reward function rewards the agent to stay between the track borders and penalizes the agent for getting too close to the next object in the front. The agent can move from lane to lane to avoid crashes. The total reward is a weighted sum of the reward and penalty. The example gives more weight to the penalty term to focus more on safety by avoiding crashes. You can play with different averaging weights to train the agent with different driving behaviors and to achieve different driving performances.

```
def reward_function(params):
    '''
    Example of rewarding the agent to stay inside two borders
    and penalizing getting too close to the objects in front
    '''

    all_wheels_on_track = params['all_wheels_on_track']
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    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']

    # Initialize reward with a small number but not zero
    # because zero means off-track or crashed
    reward = 1e-3

    # Reward if the agent stays inside the two borders of the track
    if all_wheels_on_track and (0.5 * track_width - distance_from_center) >= 0.05:
        reward_lane = 1.0
    else:
        reward_lane = 1e-3

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

    # Distance to the next object
    distance_closest_object = objects_distance[next_object_index]
    # 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

    # Calculate reward by putting different weights on
    # the two aspects above
    reward += 1.0 * reward_lane + 4.0 * reward_avoid

    return reward
```

Operate Your AWS DeepRacer Vehicle

After you finish training and evaluating an AWS DeepRacer model in the AWS DeepRacer simulator, you can deploy the model to your AWS DeepRacer vehicle. You can set the vehicle to drive on a track and evaluate the model's performance in a physical environment. This mimics a real-world autonomous race.

Before driving your vehicle for the first time, you must set up the vehicle, install software updates, and calibrate its drive-chain sub-system.

To drive your vehicle on a physical track, you must have a track. For more information, see *Build Your Physical Track* (p. 109)

Topics

- Get to Know Your AWS DeepRacer Vehicle (p. 77)
- Choose a Wi-Fi Network for Your AWS DeepRacer Vehicle (p. 91)
- Launch AWS DeepRacer Vehicle's Device Console (p. 92)
- Calibrate Your AWS DeepRacer Vehicle (p. 94)
- Upload a Model to Your AWS DeepRacer Vehicle (p. 100)
- Drive Your AWS DeepRacer Vehicle (p. 101)
- Inspect and Manage Your AWS DeepRacer Vehicle Settings (p. 103)
- View Your AWS DeepRacer Vehicle Logs (p. 107)

Get to Know Your AWS DeepRacer Vehicle

Your AWS DeepRacer vehicle is a machine learning-enabled, battery-powered, and Wi-Fi-connected 1/18th-scale model four-wheel drive car with a front-mounted 4-megapixel camera and an Ubuntu-based compute module.

The vehicle can drive autonomously by running inference that is based on a reinforcement learning model in its compute module. You can also drive the vehicle manually, without deploying any reinforcement learning model. If you have not already obtained an AWS DeepRacer vehicle, you can order one here.

The AWS DeepRacer vehicle is powered by a brushed motor. The driving speed is controlled by a voltage regulator that controls how fast the motor spins. The servomechanism (servo) that operates the steering system is protected by the black cover in the AWS DeepRacer vehicle chassis.

Topics

- Inspect Your AWS DeepRacer Vehicle (p. 78)
- Charge and Install Your AWS DeepRacer Batteries (p. 79)
- Test Your AWS DeepRacer Compute Module (p. 80)
- Turn Off Your AWS DeepRacer Vehicle (p. 81)

- AWS DeepRacer Vehicle LED Indicators (p. 81)
- AWS DeepRacer vehicle spare parts (p. 83)

Inspect Your AWS DeepRacer Vehicle

When you open your AWS DeepRacer vehicle box, you should find the following components and accessories:

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.
Vehicle battery [3]	A 7.4v Li-Po battery pack to power the motor.
Vehicle battery charger [4]	Use this to charge the vehicle battery that powers the vehicle drive chain.

Components	Comments
Vehicle battery power adapter [5]	Use this to connect the vehicle battery charger to a power outlet.
Compute module power bank [6]	Use this to power the compute module that runs inference on a downloaded AWS DeepRacer reinforcement learning model.
Compute module power bank connector cable [7]	Use this USB C-to-USB C cable to connect the compute module with the power bank.
Power cord and adapter [8]	Use this to charge the compute module power bank and the compute module.
Vehicle chassis pins [9] and [11]	Use the four white pins and the spare ones (in black) to fasten the compute module to the vehicle chassis.
USB-to-µUSB cable [10]	Use this to support USB-OTG functionality.

To set up your AWS DeepRacer vehicle, you must also have the following items ready:

- A computer with a USB port and access to the internet.
- A Wi-Fi network connected to the internet.
- An AWS account.

Now follow the instructions in the next section (p. 79) to make sure your vehicle battery and the power bank are charged.

Charge and Install Your AWS DeepRacer Batteries

Your AWS DeepRacer vehicle has two power sources: the vehicle battery and the compute module power bank.

The hard-cased power bank keeps the compute module running. The compute module maintains the Wi-Fi connection, runs inference against a deployed AWS DeepRacer model, and issues a command for the vehicle to take an action.

The vehicle battery powers the motor to move the vehicle. It has a blue package with two sets of cables. The two-wired set of the red and black cables is used to connect to the vehicle's ESC and the triple-wired blue (or black), white and red cables is to connect to the charger. For driving, only the two-wired cable set should be connected to the vehicle.

After fully charged, the battery voltage will drop as the batteries discharge. When the voltage drops, the available torque also drops. As a consequence, the same speed setting will result in slower speed on the track. When the battery is fully empty, the vehicle stops moving. For autonomous driving under normal conditions, the battery usually lasts 15-25 minutes. To ensure consistent behavior, it is recommended that you charge the battery after every 15 minutes of use.

To install and charge the vehicle battery and the power bank, follow the steps below.

1. Remove your AWS DeepRacer vehicle shell.
2. Remove the four vehicle chassis pins. Carefully lift vehicle chassis while keeping wires connected.
3. To charge and install the vehicle battery, do the following:

a. To charge the battery, plug the three-wired cable set from the batter to the charger to connect the battery to the power adapter and then plug the power adapter to a wall outlet or to a USB port if a USB cable is used to charge the battery.

   For a graphical illustration of how to charge the vehicle battery using the enclosed charger, see the section called “How to Charge the Vehicle's Drive Module Battery” (p. 138).

b. After the battery is charged, plug the two-wired cable set of the vehicle battery cable into the black and red cable connector on your vehicle.

c. To secure the vehicle battery, tie the battery under the vehicle chassis with the attached straps.

   Make sure to keep all the cables inside the vehicle.

d. To check if the vehicle battery is charged, do the following:

   i. Slide the vehicle power switch to turn on the vehicle.

   ii. Listen for two short beeps.

      If you don't hear the beeps, the vehicle is not charged. Remove the battery from the vehicle and repeat Step 1 above to recharge the battery.

   iii. When not using the vehicle, slide the vehicle power switch back to turn off the vehicle battery.

4. To check the power bank charging level, do the following:

   a. Press the power button on the power bank.

   b. Check the four LED lights next to the power button to determine the charging level.

      If all the four LED lights are lit, the power bank is fully changed. If none of the LED lights are lit, the power bank needs to be charged.

   c. To charge the power bank, insert the USB C plug from the power adapter into the USB C port of the power bank. It takes some time for the power bank to be fully charged. When it is charged, repeat **Step 4** to confirm that the power bank is fully charged.

5. To install the power bank, do the following:

   a. Insert the power bank into its holder with the power button and USB C port facing the back of the vehicle.

   b. Use the strap to tie the power bank to the vehicle chassis securely.

      Note
      Do not connect the power bank to the compute module in this step.

Test Your AWS DeepRacer Compute Module

Test the compute module to verify that it can be started successfully. To test the module by using an external power source, follow the steps below:

To test your vehicle's compute module

1. Connect the compute module to a power source. Connect the power cord to the power adapter, plug the power cord to a power outlet, and insert the power adapter's USB C plug into the USB C port on the compute module.
2. Turn on the vehicle's compute module by pressing the power button on the compute module.
3. To verify the compute module's status, check that the LED lights are shown as follows:

   - Solid blue

The compute module is started, connected to the specified Wi-Fi, and ready to go.

In this state, you can log in to the compute module after you attach it to a monitor using an HDMI cable, a USB mouse and a USB keyboard. For the first-time login, use `deepracer` for both the **username** and **password**. You will then be asked to reset the password for future logins. For security reasons, choose a strong password phrase for the new password.

- Blinking red

  The compute module is in setup mode.

- Solid yellow

  The compute module is initializing.

- Solid red

  The compute module failed to connect to the Wi-Fi network.

4. When you're done with the test, press the power button on the compute module to turn it off and then unplug it from the external power source.

Turn Off Your AWS DeepRacer Vehicle

To turn off your AWS DeepRacer vehicle, unplug the vehicle from the external power source. You can also press the power button on the device until power indicator is off.

AWS DeepRacer Vehicle LED Indicators

Your AWS DeepRacer vehicle has two sets of LED indicators for the vehicle status and for customizable visual identification of your vehicle, respectively.

The details are discussed as follows.

Topics

- AWS DeepRacer Vehicle System LED Indicators (p. 82)
- AWS DeepRacer Vehicle Identification LEDs (p. 83)

AWS DeepRacer Vehicle System LED Indicators

The AWS DeepRacer vehicle system LED indicators are located on the left side of the vehicle chassis when the vehicle is in the forward position in front of you.

The three system LEDs are positioned after the **RESET** button. The first LED (on the left side of your field of view) shows the status of the system power. The second (middle) LED is reserved for future use. The last (right) LED shows the status of the Wi-Fi connection.

LED Type	Color	Status
Power	Off	There is no power supply.
	Blinking yellow	BIOS and OS are being loaded.
	Steady yellow	OS is loaded.

LED Type	Color	Status
	Steady blue	An application is running.
	Blinking blue	A software update is in progress.
	Steady red	An error is encountered while the system is being booted or an application is being started.
Wi-Fi	Off	There is no Wi-Fi connection.
	Blinking blue	The vehicle is connecting to the Wi-Fi network.
	Steady red for 2 seconds and then off	The Wi-Fi connection failed.
	Steady blue	The Wi-Fi connection is established.

AWS DeepRacer Vehicle Identification LEDs

The AWS DeepRacer vehicle custom LEDs are located at the tail of the vehicle. They're used to help identifying your vehicle in races when multiple vehicles are present. You can use the AWS DeepRacer device console to set them a supported color (p. 103) of your choosing.

AWS DeepRacer vehicle spare parts

Use this catalogue to help you find AWS DeepRacer parts for replacement or repair.

Note
The AWS DeepRacer vehicle uses WLToys A949 and A979 Remote Control (RC) car chassis.

Spare AWS DeepRacer vehicle parts

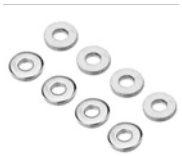
Part	Name
	Spare compute battery

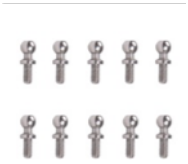
Part	Name
	Tire
	Front bumper
	Suspension arm
	Pull rod
	C style seat
	Transmission shaft
	Round head screw, M2x17.5mm
	Chassis car bottom

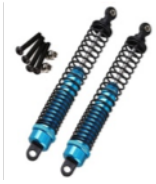
Part	Name
	Turning seat
	Rear suspension frame
	Metal hexagonal combiner set
	Gear box shell
	Differential box case
	Differential drive cup
	Front rear shelter

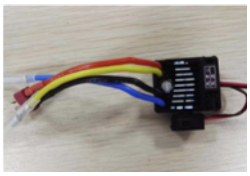
Part	Name
	Servo seat
	Central driving shaft
	Shock frame
	Servo arm
	Differential mechanism
	Reduction gear
	Motor base

Part	Name
	Lithium battery 7.4V 1100mah
	17g steering engine
	Motor heat dissipation cover
	Stub axle
	Screw gasket of motor, fixed seat
	390 motor
	Hexagon connector 4x8x3mm

Part	Name
	Hexagon connector 8x12x3.5mm
	Ball bearing 7x11x3mm
	Ball bearing 8x12x3.5mm
	Middle axle disc plate
	Screw 2.6x6mm
	Screw 2x7mm
	Screw 2.5x8mm
	Screw 2x16mm

Part	Name
	Screw 2.5x6x6mm
	Screw M3x5mm
	Ball screw 10.8x4mm
	Screw 2x6mm
	Screw 2x9.5mm
	M3 locknut
	Axle hinge pin

Part	Name
	Drive shaft
	Swing arm pin
	Screw 2*29KM
	Hair pin
	Front shock absorber
	Charger
	Metal gear
	Back shock absorber

Part	Name
	ESC

Choose a Wi-Fi Network for Your AWS DeepRacer Vehicle

The first time you open your AWS DeepRacer vehicle, you must set it up to connect to a Wi-Fi network. Complete this setup to get the vehicle's software updated and to get the IP address to access the vehicle's device console.

This section walks you through the steps to perform the following tasks:

- Connect your laptop or desktop computer to your vehicle.
- Set up the vehicle's Wi-Fi connection.
- Update the vehicle's software.
- Get the vehicle's IP address.
- Test drive the vehicle.

Use a laptop or desktop computer to perform the setup tasks. We'll refer to this setup computer as your computer, to avoid possible confusion with the vehicle's compute module, which is running the Ubuntu operating system.

After the initial setup of the Wi-Fi connection, you can follow the same instructions to choose a different Wi-Fi network.

> **Note**
> AWS DeepRacer does not support Wi-Fi network that requires active captcha verification for use sign-in.

Topics

- Get Ready to Set Up Wi-Fi Connection for Your AWS DeepRacer Vehicle (p. 91)
- Set Up Wi-Fi Connection and Update Your AWS DeepRacer Vehicle's Software (p. 92)

Get Ready to Set Up Wi-Fi Connection for Your AWS DeepRacer Vehicle

To set up your vehicle's Wi-Fi connection, connect your a laptop or desktop computer to your vehicle's compute module using the included *USB-to-USB C* cable.

To connect your computer to your vehicle's compute module, follow the steps below.

To connect your computer to your vehicle to set up the device

1. Insert the USB end of the *USB-to-USB C* cable into your computer's USB port.
2. Insert the cable's USB C end into your vehicle's USB C port.

You're now ready to proceed to setting up your vehicle's Wi-Fi connection.

Set Up Wi-Fi Connection and Update Your AWS DeepRacer Vehicle's Software

Before you follow the steps here to set up the Wi-Fi connection, be sure you complete the steps in the section called “Get Ready to Set Up Wi-Fi” (p. 91).

1. Look at the bottom of your vehicle and make note of the password printed under **Host name**. You'll need it to log in to the device control console to perform the setup.
2. On your computer, go to `https://deepracer.aws` to launch the device control console of your vehicle.
3. When prompted with a message that the connection is not private or secure, do one of the following.

   a. In Chrome, choose **Avanced** and then choose **Proceed to *<device_console_ip_address>* (unsafe)**.

   b. In Safari, choose **Details**, follow the **visit this website** link, and the choose **Visit Websites**. If prompted for your password to update the certificate trust settings, type the password and then choose **Update settings**.

   c. In Opera, choose **Continue Anyway** when warned of an invalid certificate.

   d. In Edge, choose **Details** and then choose **Go on to the webpage (Note recommended)**.

   e. In Firefox, choose **Advanced**, choose **Add Exception**, and then choose **Confirm Security Exception**.
4. Under **Unlock your AWS DeepRacer vehicle**, enter the password noted in **Step 1** and then choose **Access vehicle**.
5. On the **Connect your vehicle to your Wi-Fi network** pane, choose your Wi-Fi network name from the **Wi-Fi network name (SSID)** drop-down menu, type the password of your Wi-Fi network under **Wi-Fi password**, and choose **Connect**.
6. Wait until the Wi-Fi connection status changes from **Connecting to Wi-Fi network...** to **Connected**. Then, choose **Next**.
7. On the **Software update** pane, if a software update is required, turn on the vehicle's compute module, with the included power cord and power adapter, and then choose **Install software update**.

   Powering the vehicle with an external power source helps avoid interruption of the software update if the compute module's power bank become discharged.
8. Wait until the software update status changes from **Installing software update** to **Software update installed successfully**.
9. Note the IP address shown under **Wi-Fi network details**. You'll need it to open the vehicle's device control console after the initial setup and any subsequent modification of the Wi-Fi network settings.

Launch AWS DeepRacer Vehicle's Device Console

After you set up the vehicle's Wi-Fi connection and install required software updates, you should open the device console to verify if the vehicle's network connection is working. Subsequently, you can launch the device console to inspect, calibrate and manage the vehicle's other settings. The process involves signing in to your vehicle's device console using the IP address of your vehicle.

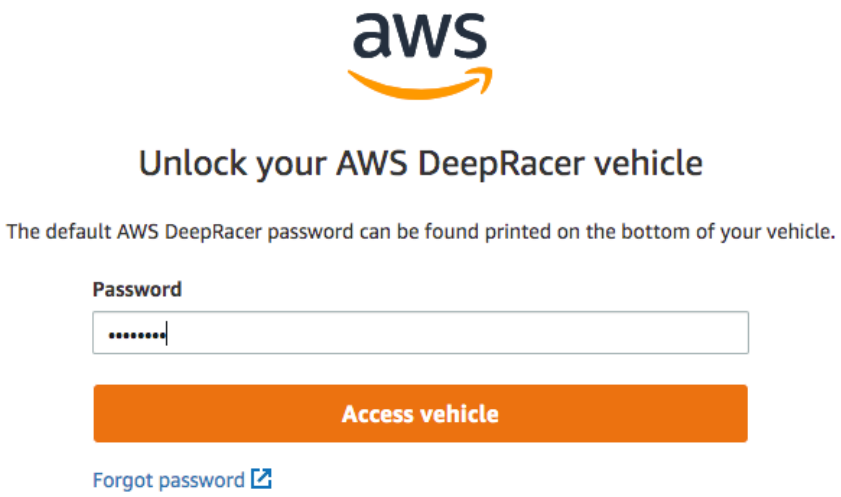
The device control console is hosted on the vehicle and is accessed with the IP address you obtained at the end of the Wi-Fi setup (p. 92) section.

To access the device console of your AWS DeepRacer vehicle through the Wi-Fi connection

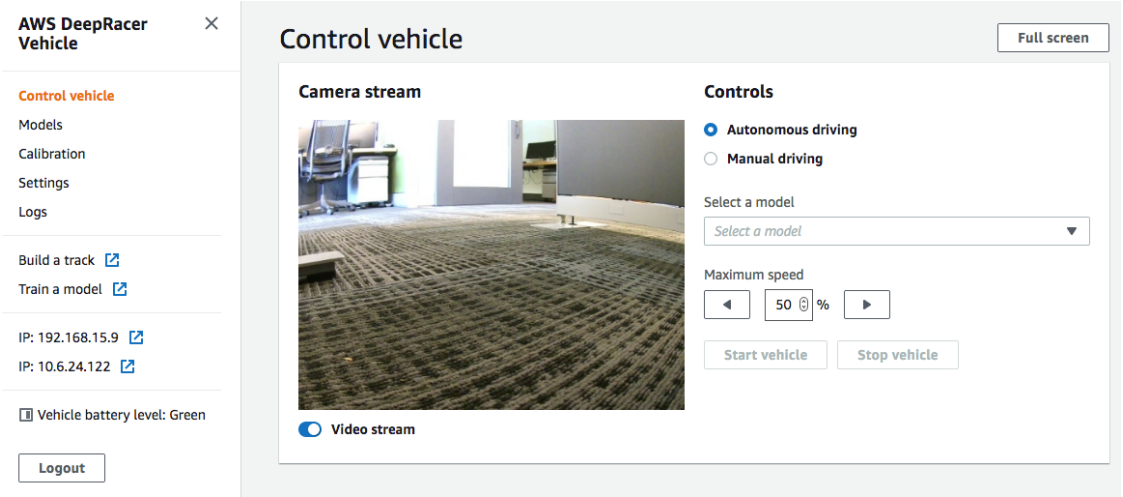
1. To access the device console of your vehicle, open a web browser on your computer, tablet or a smart phone and type your vehicle's IP address into the address bar.

   You can get this IP address when setting up the vehicle's Wi-Fi connection (p. 92). For illustration, we use `10.92.206.61` as an example.

   If you are prompted with a warning that the connection is not private or secure, ignore the message and continue to connect to the device console.
2. Under **Unlock your AWS DeepRacer vehicle**, type the device console's password in **Password** and then choose **Access vehicle**.



   You can find the default password printed on the bottom of your vehicle (under **Host Name**).
3. When you are successfully signed in, you see the device console's home page as follows.



You're now ready to calibrate and operate your vehicle. If this is your first time operating the vehicle, proceed to calibrating the vehicle (p. 94) now.

Calibrate Your AWS DeepRacer Vehicle

To achieve the best performance, it's essential that you calibrate some physical parts of your AWS DeepRacer vehicle. If you use an uncalibrated vehicle, it can add uncertainty when testing your model. If the vehicle's performance is not optimal, you might be tempted to only adjust the deep learning model code. However, you won't be able to improve the vehicle performance if the root cause is mechanical. Adjust the mechanics by calibration.

To calibrate your AWS DeepRacer vehicle, set the duty cycle range for the vehicle's electronic control system (ECS) and its servomechanism (servo), respectively. Both the servo and ECS accept pulse-width modulation (PWM)) signals as control input from the vehicle's compute module. The compute module adjusts both of the vehicle's speed and steering angle by changing the duty cycles of the PWM signals.

The maximum speed and steering angle defines the span of the action space. You can specify the maximum speed and maximum steering angle during training in simulation. When deploying the trained model to your AWS DeepRacer vehicle for driving on a real-world track, the maximum speed and steering angle of the vehicle must be calibrated to match those used in the simulation training.

To ensure that the real-world experiences match the simulated experiences, you should calibrate your vehicle to match the maximum speed and maximum steering angles between the simulation and the real world. In general, there are two ways to do this calibration:

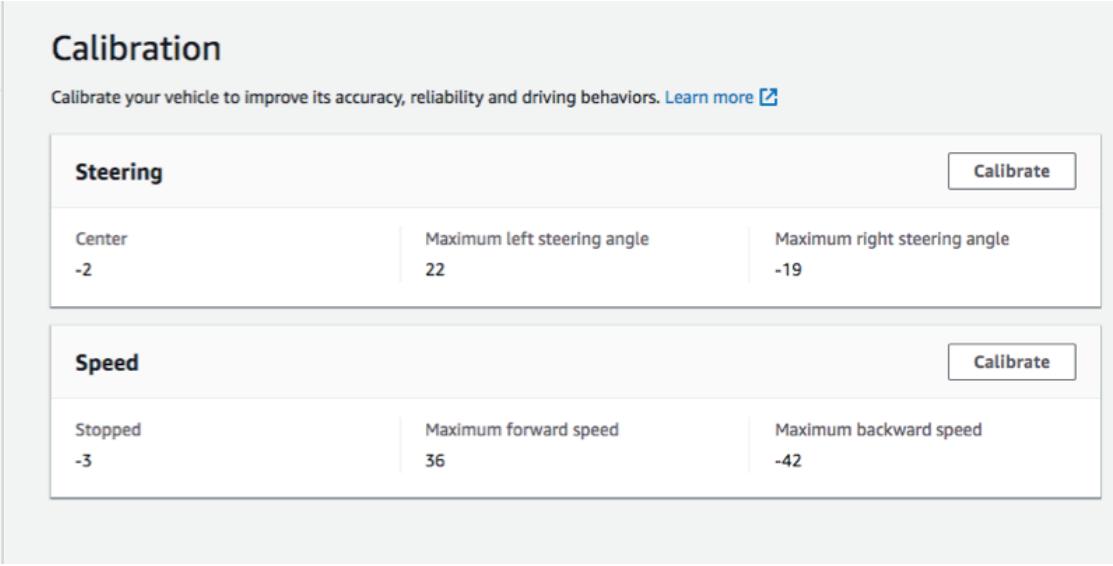
- Define the action space in training and calibrate the physical vehicle to match the settings.
- Measure the actual performance of your vehicle and change the settings of the action space in the simulation.

A robust model can handle certain differences between the simulation and the real world. However, you should experiment with either approach and iterate to find the best results.

Before starting the calibration, turn on the compute module. After it's started and the power LED has turned solid blue, turn on the vehicle battery. After you hear two short beeps and one long beep, you're ready to proceed with the calibration.

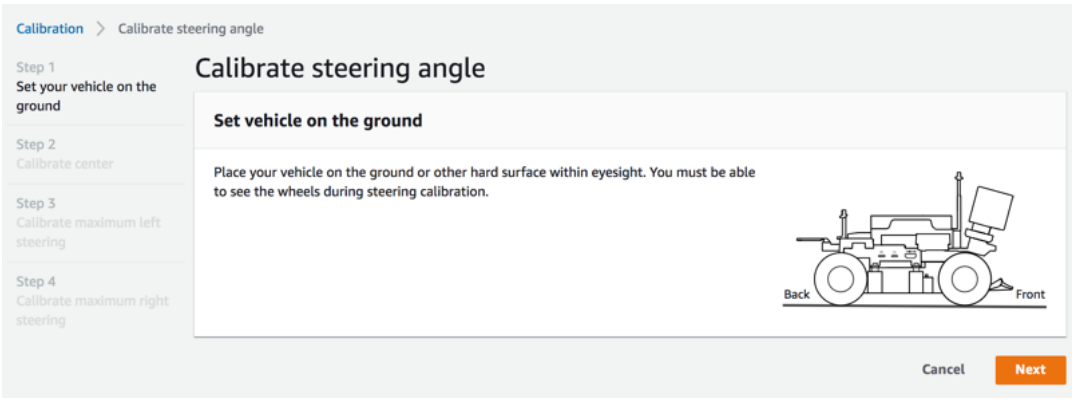
To calibrate your AWS DeepRacer vehicle to match the training settings:

1. Follow these instructions (p. 92) to access your vehicle and open the device control console.
2. Choose **Calibration** from the main navigation pane.



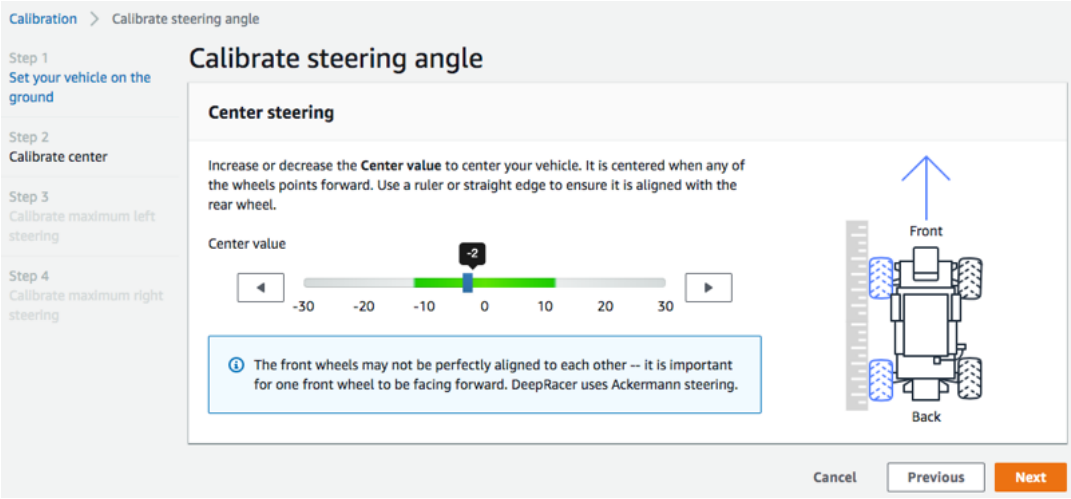
3. On the **Calibration** page, choose **Calibrate** in **Steering** and then follow the steps below to calibrate the vehicle's maximum steering angles.

   a. Set the vehicle on the ground or another hard surface where you can see the wheels during the steering calibration. Choose **Next**.



   Steering a vehicle on a track requires the much smaller steering angles than turning wheels in the air. To measure the actual steering angles of the wheels, it's important that you place the vehicle down on the track surface.

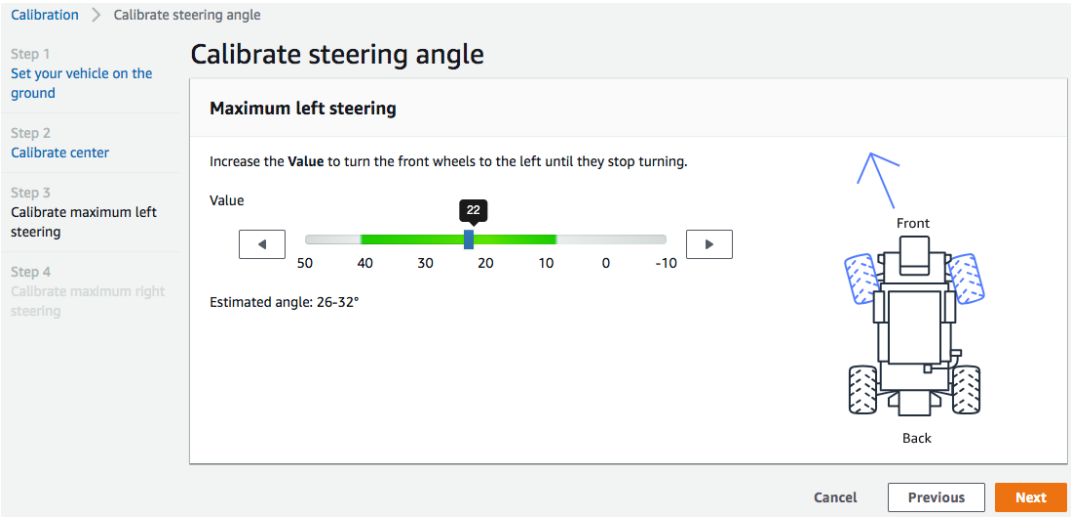
   b. Under **Center steering**, gradually move the slider or press the left or right arrow to the position where at least one of the front wheels is aligned with the rear wheel on the same side. Choose **Next**.



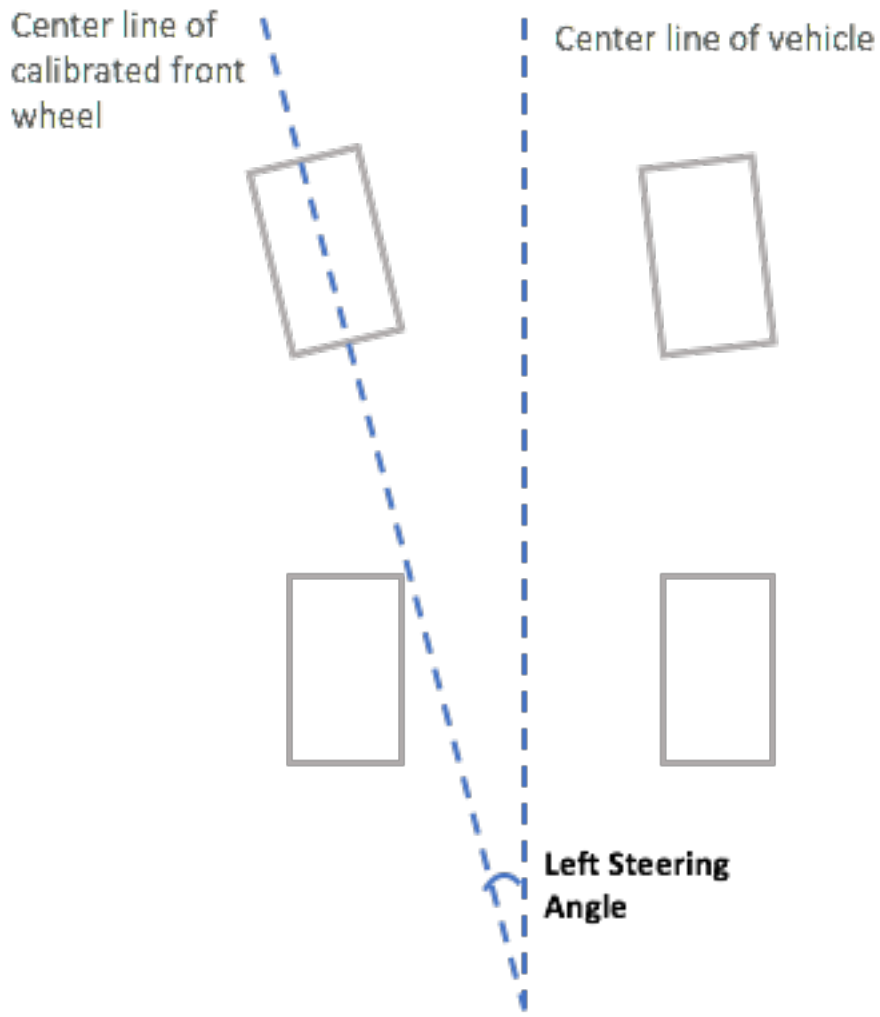
AWS DeepRacer uses Ackermann front-wheel steering to turn wheels on the inside and outside of a turn. This mean that the left and right front wheels generally turn at different angles. In AWS DeepRacer, the calibration is done on the center value. Therefore, you need to adjust the wheels on the selected side to be aligned in a straight line.

> **Note**
> Make sure to calibrate your AWS DeepRacer vehicle well (p. 94) so that it can maintain center steering as straight as possible. You can test this by manually pushing the vehicle to verify it follows a straight path.

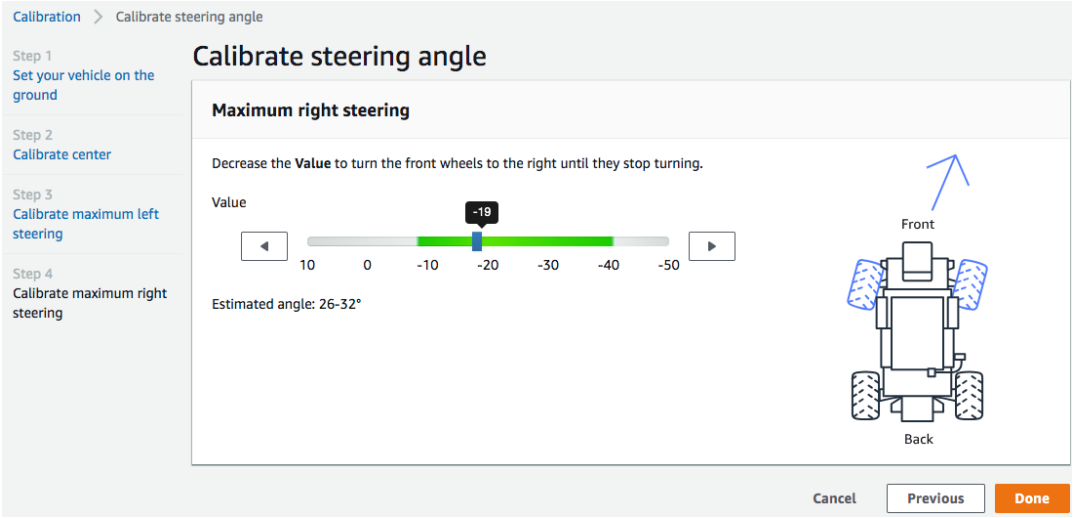
c. Under **Maximum left steering**, gradually move the slider to the left or press the left arrow until the vehicle front wheels stops turning left. There will be a quiet noise. If you hear a loud noise, you have gone too far. The position corresponds to the maximum left steering angle. If you have limited your steering angle in the simulated action space, match the corresponding value here. Choose **Next**.



To measure the actual maximum left steering angle, draw a center line for the vehicle, mark the two edge points of the selected front wheel for calibration, and draw the center line of this front wheel until it crosses over the center line of the vehicle. Use a protractor to measure the angle. See the figure below. If you want to match the actual angle in your training, you can set the same value in the action space in your next training job.



d. Under **Maximum right steering**, gradually move the slider to the right until the selected front wheels stop turning right. There will be a quiet noise. If you hear a loud noise, you have gone too far. The position corresponds to the maximum right steering angle. If you have limited your steering angle in the simulated action space, match the corresponding value here. Choose **Done**.

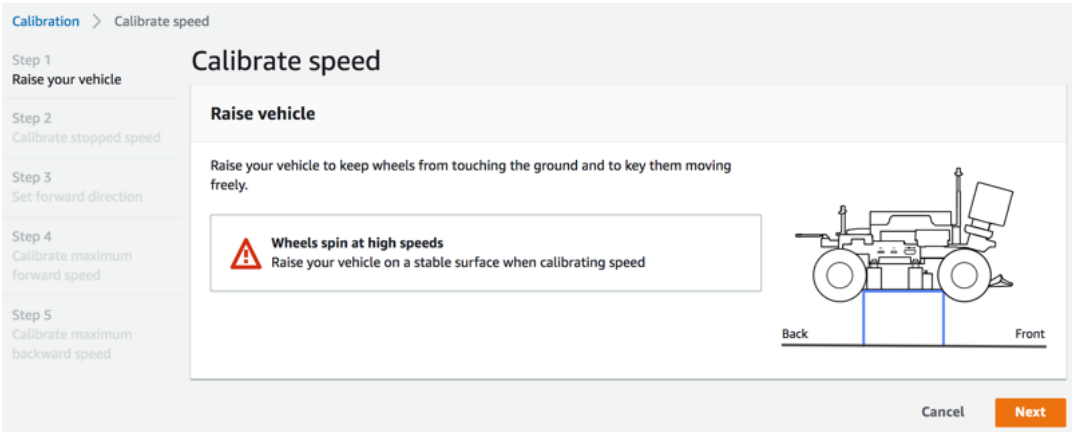


To measure the actual maximum right steering angle, follow the steps similar to those used to measure the maximum left steering angle.

This concludes the steering calibration for your AWS DeepRacer vehicle.

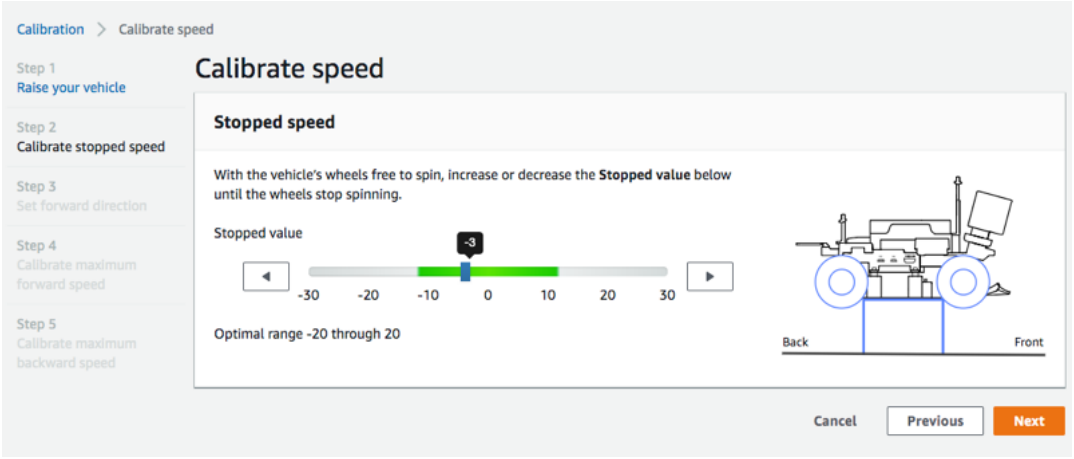
4. To calibrate the vehicle's maximum speed, choose **Calibrate** in **Speed** on the **Calibration** page and then follow the steps below.

   a. Raise the vehicle so that the wheels are free to turn. Choose **Next** on the device control console.



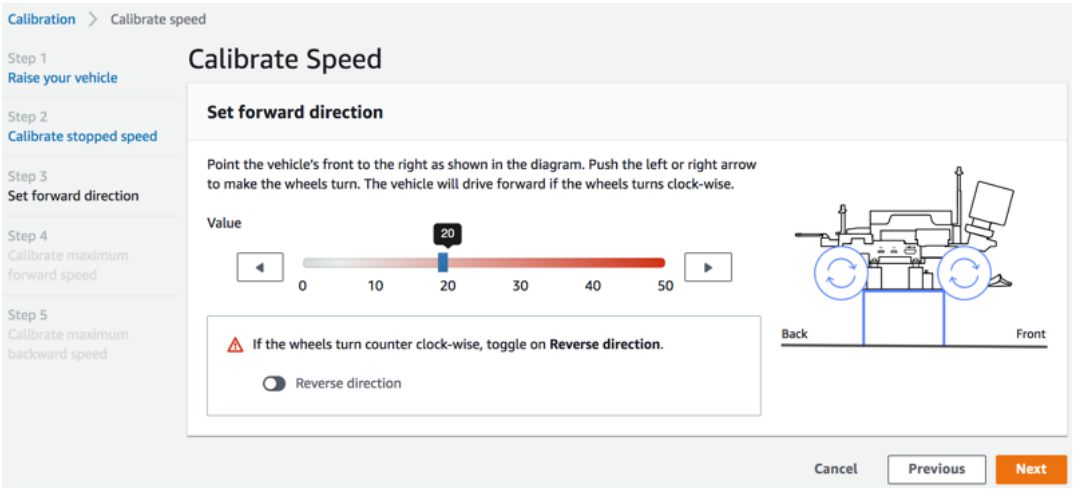
   Note
   If the vehicle's speed has been set too high, it may run too fast during calibration and cause damage to the environment, the vehicle, or others nearby. You should raise the vehicle, as instructed here, but not hold it in your hands.

   b. To calibrate the stopped speed, press the left or right arrow to gradually change **Stopped value** under **Stopped speed** on the device control console until the wheels stop turning. Choose **Next**.



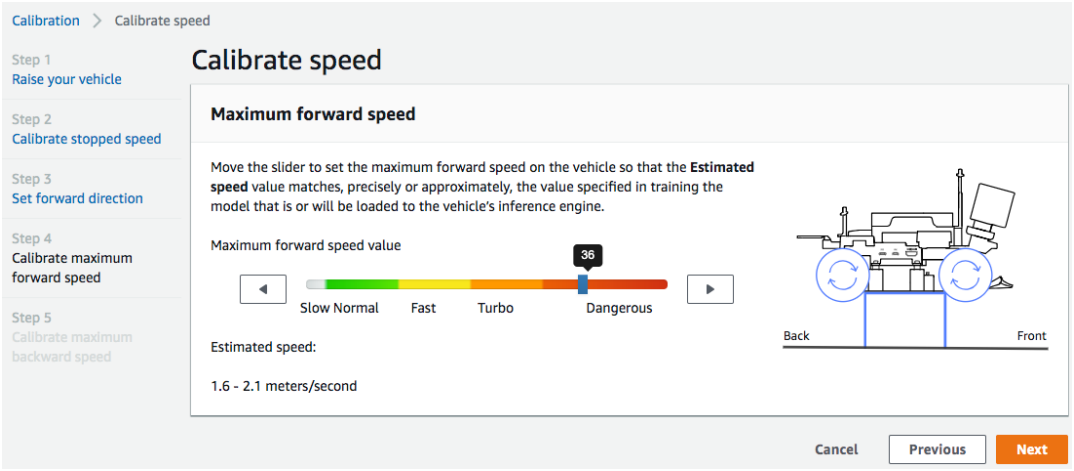
   Note
   When pressing the **Stopped value** further left or further right to the value when you start hearing noises, the wheels are about to move. The ideal zero-throttle point is the middle of the two values. For example, if you start hearing a noise at 16 on the left and at -4 on the right, the optimal stopped value should be 10.

   c. To set the vehicle's forward direction, place the vehicle as shown on the screen and the image here, and then press the left or right arrow to make the wheels turn. If the wheels turn clock-wise, the forward direction is set. If not, toggle **Reverse direction**. Choose **Next**.



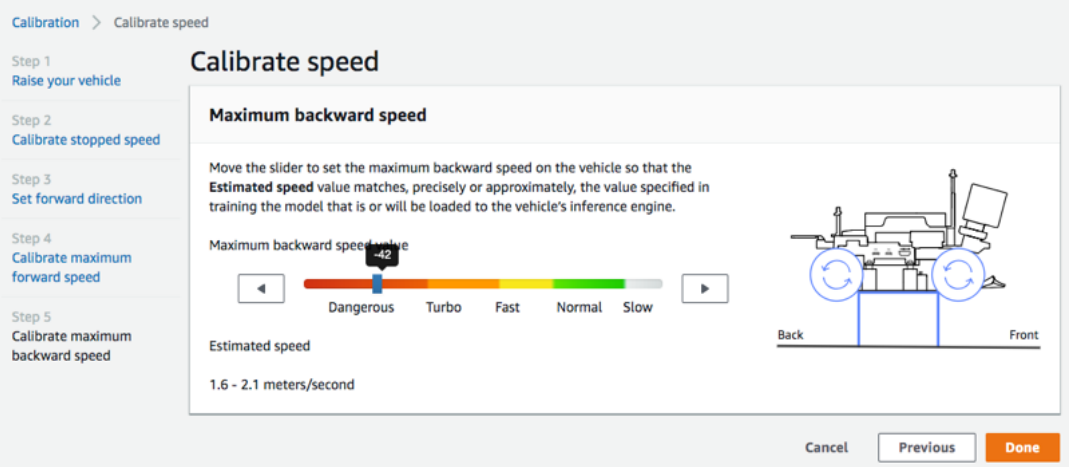
Note
Vehicles distributed at AWS re:Invent 2018 might have their forward direction set in reverse. In such a case, make sure to toggle **Reverse direction**.

d. To calibrate the maximum forward speed, under **Maximum forward speed**, gently move the slider left or right to adjust the **Maximum forward speed value** number gradually to such a positive value that the **Estimated speed** value is equal or similar to the maximum speed specified in the simulation. Choose **Next**.



Note
The actual maximum speed your vehicle depends on the friction of the track surface as well as the vehicle battery level. To make it flexible, you can set the vehicle's throttle limit to be 20-30 percent higher than the maximum speed specified for training in the simulation. Generally speaking, you should set the maximum speed value within the green area. Above that, your vehicle is likely to drive too fast at increased risk of breaking. Additionally, the action space for training doesn't support the maximum speed of more than 2 m/s.

e. To calibrate the maximum backward speed, under **Maximum backward speed**, gently move the slider left or right to adjust the **Maximum backward speed value** number gradually to such a negative value that the **Estimated speed** value is equal or similar to the maximum speed specified in the simulation. Choose **Done**.



Note
The AWS DeepRacer vehicle doesn't use backward speed in the autonomous driving mode. You can set the backward speed to any value with which you can comfortably control the vehicle's manual driving mode.

This concludes calibrating your AWS DeepRacer vehicle's maximum speed.

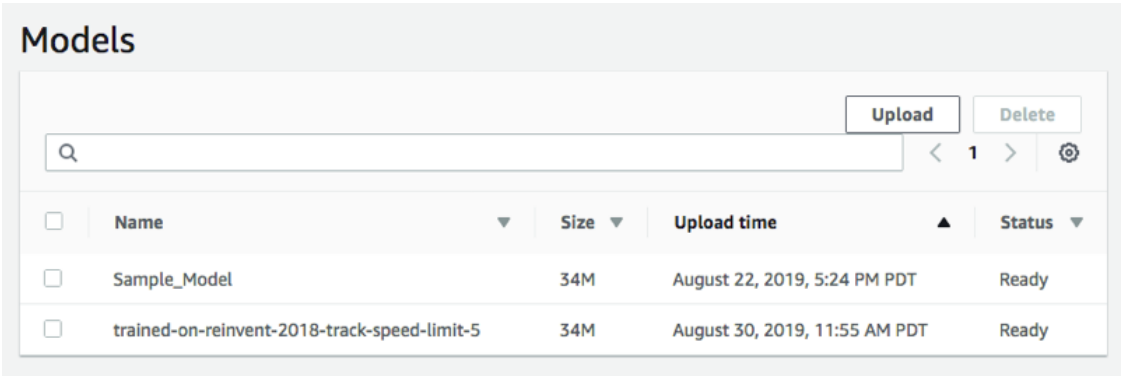
Upload a Model to Your AWS DeepRacer Vehicle

To start your AWS DeepRacer vehicle on autonomous driving, you must have uploaded at least one AWS DeepRacer model to your AWS DeepRacer vehicle.

To upload a model, you must have trained and evaluated the model (p. 31). You can train the model using the AWS DeepRacer console. After that, you need to download the model artifacts from its Amazon S3 storage to a (local or network) drive that can be accessed by your computer.

To upload a trained model to your vehicle

1. Choose **Models** from the device console's main navigation pane.



2. On the **Models** page, choose **Upload** above the **Models** list.
3. From the file picker, navigate to the drive or share where you've downloaded your model artifacts and choose the the compressed model file (of the `*.tar.gz` extension) to upload.

   Only a successfully uploaded model will be added to the **Models** list and can be available for you to load it into the vehicle's inference engine in the autonomous driving mode. For the instructions

on how to load a model into your vehicle's inference engine, see Drive Your AWS DeepRacer Vehicle Autonomously (p. 102).

Drive Your AWS DeepRacer Vehicle

After setting up your AWS DeepRacer vehicle (p. 91), you can start to drive your vehicle manually or let it drive autonomously, using the vehicle's device console.

For autonomous driving, you must have trained an AWS DeepRacer model and have the trained model artifacts deployed to the vehicle. In the autonomous racing mode, the model running in the inference engine controls the vehicle's driving directions and speed. Without a trained model downloaded to the vehicle, you can use the vehicle's device console to drive the vehicle manually.

Many factors affect the vehicle's performance in autonomous driving. They include the trained model, vehicle calibration, track conditions, such as surface frictions, color contrasts and light reflections, etc. For your vehicle to achieve an optimal performance, you must make sure that the model transfer from the simulation to the real world is as accurate, relevant and meaningful. For more information, see the section called “Optimize Training for Real Environments” (p. 41).

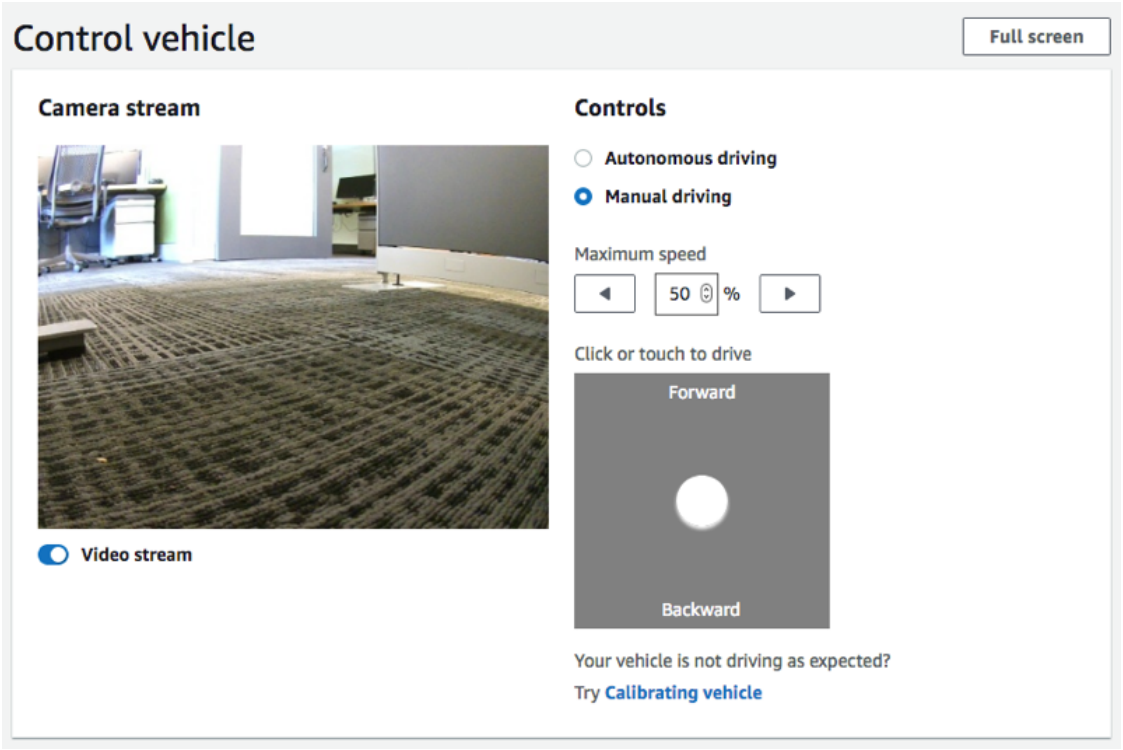
Drive Your AWS DeepRacer Vehicle Manually

If you have not trained any model or have not deployed any trained model to your AWS DeepRacer vehicle, you can't let it drive itself. But you can drive it manually.

To drive a AWS DeepRacer vehicle manually, follow the steps below.

To drive your AWS DeepRacer vehicle manually

1. With your AWS DeepRacer vehicle connected to the Wi-Fi network, follow the instructions (p. 92) to sign in to the vehicle's device control console.
2. On the **Control vehicle** page, choose **Manual driving** under **Controls**.



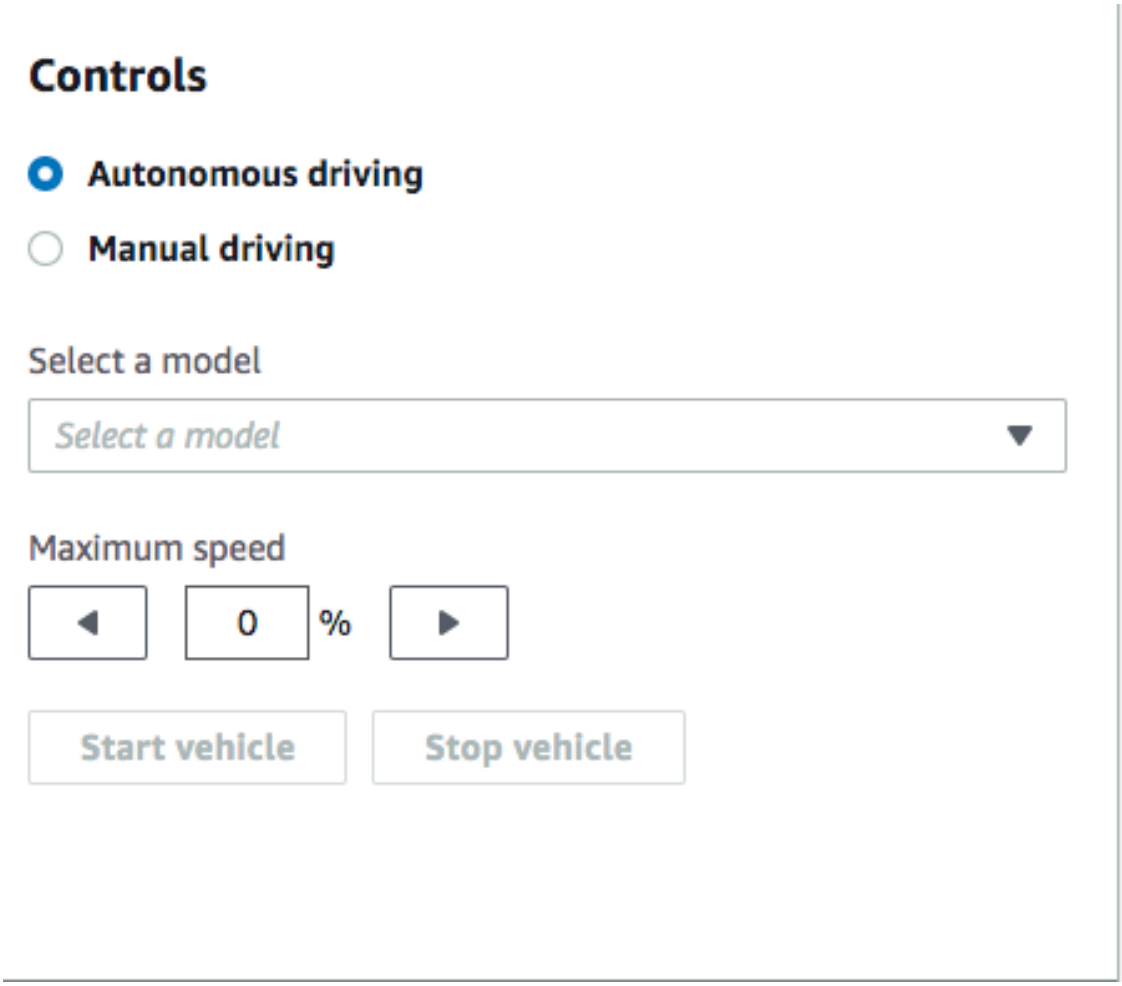
3. Under **Click or touch to drive**, click or touch a position within the driving pad to drive the vehicle. Images captured from the vehicle's front camera are displayed in the video player under **Camera stream**.
4. To turn video stream on or off on the device console while you drive the vehicle, toggle the **Video stream** option under the **Camera stream** display.
5. Repeat from **Step 3** to drive the vehicle to different locations.

Drive Your AWS DeepRacer Vehicle Autonomously

To start autonomous driving, place the vehicle on a physical track and do the following:

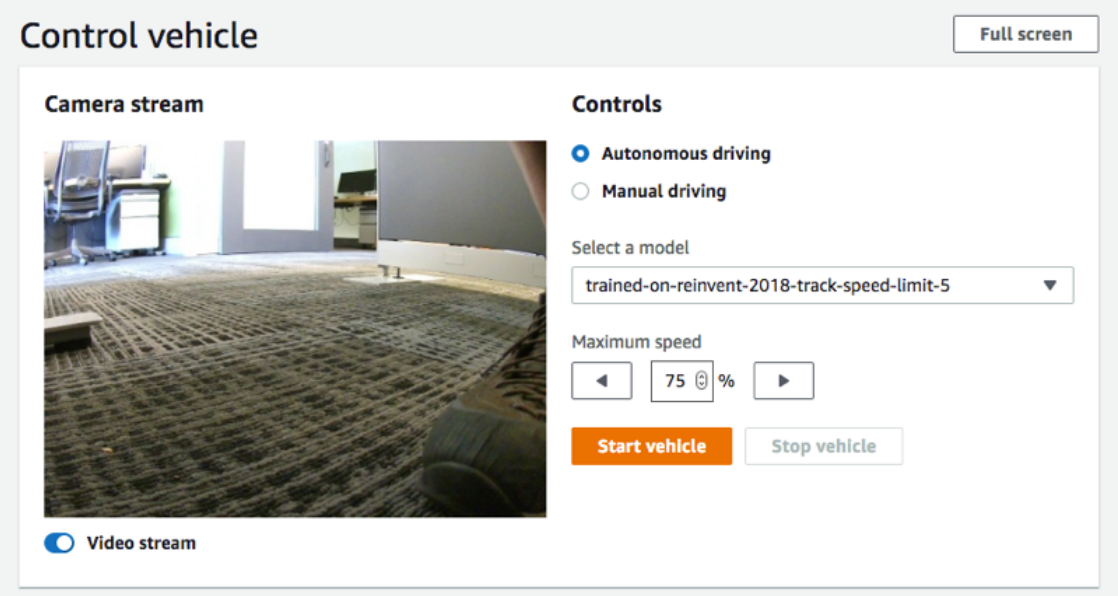
To drive your AWS DeepRacer vehicle autonomously

1. Follow the instructions (p. 92) to sign in to the vehicle's device console, and then do the following for autonomous driving:
2. On the **Control vehicle** page, choose **Autonomous driving** under **Controls**.



3. From the **Select a model** drop-down list, choose an uploaded model. Then choose **Load model**. This will start loading the model into the inference engine. The process takes about 10 seconds to complete.
4. Adjust the **Maximum speed** setting of the vehicle to be a percentage of the maximum speed used in training the model.

Certain factors, such as surface friction of the real track, can reduce the maximum speed of the vehicle from the maximum speed used in the training. You'll need to experiment to find the optimal setting.



5. Choose **Start vehicle** to set the vehicle to drive autonomously.
6. To turn video stream on or off on the device console while you drive the vehicle, toggle the **Video stream** option under the **Camera stream** display.
7. Watch the vehicle drive on the physical track or the streaming video player on the device console.
8. To stop the vehicle, choose **Stop vehicle**.

Repeat from Step 3 for another run with the same or a different model.

Inspect and Manage Your AWS DeepRacer Vehicle Settings

After the initial setup, you can use the AWS DeepRacer device control console to manage your vehicle's settings. The tasks include the following:

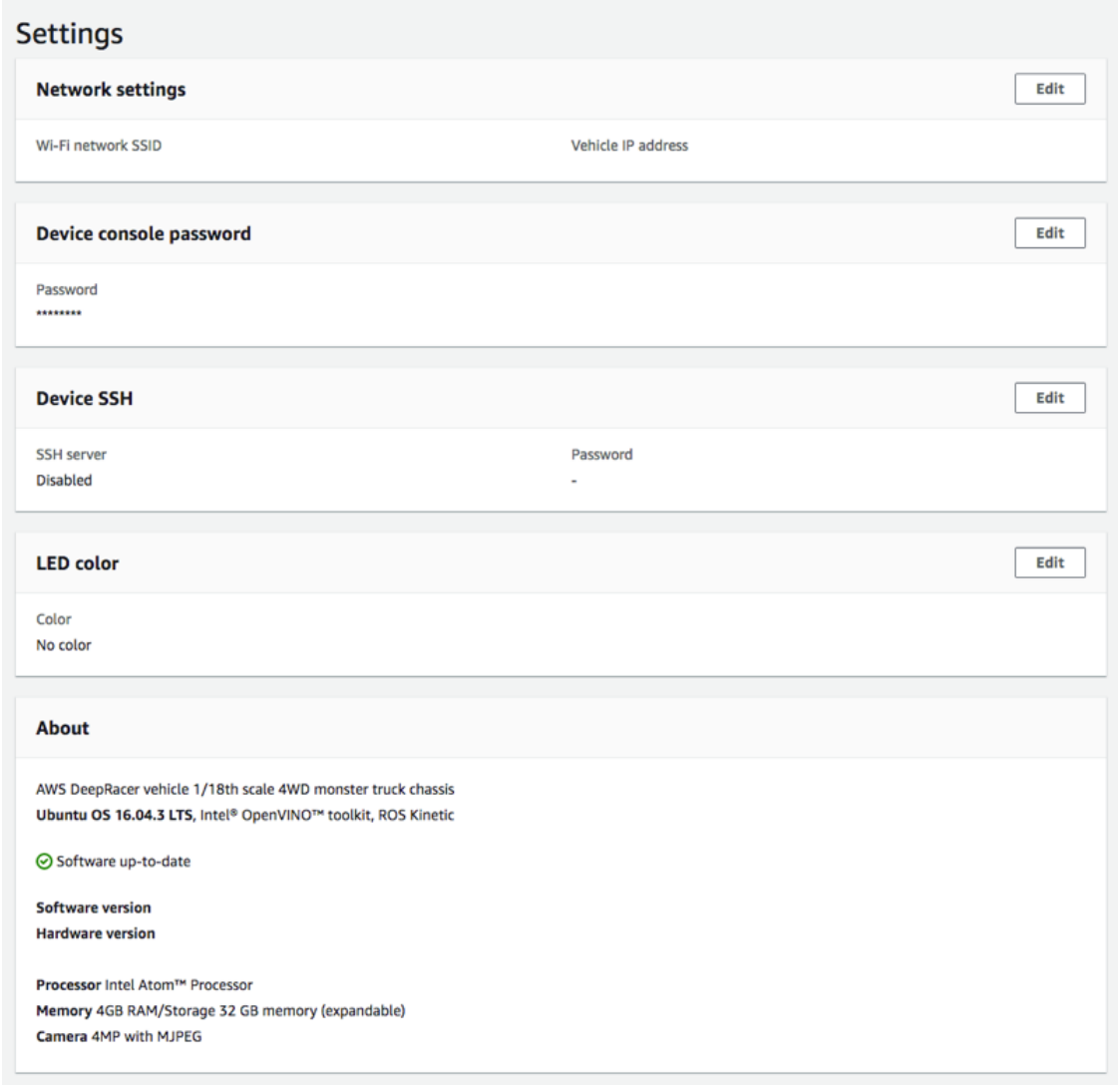
- choosing another Wi-Fi network,
- resetting the device console password,
- enabling or disabling the device SSH settings,
- configuring the vehicle's trail light LED color,
- inspecting the device software and hardware versions,
- checking the vehicle battery level.

The procedure below walks you through these tasks.

To inspect and manage your vehicle's settings

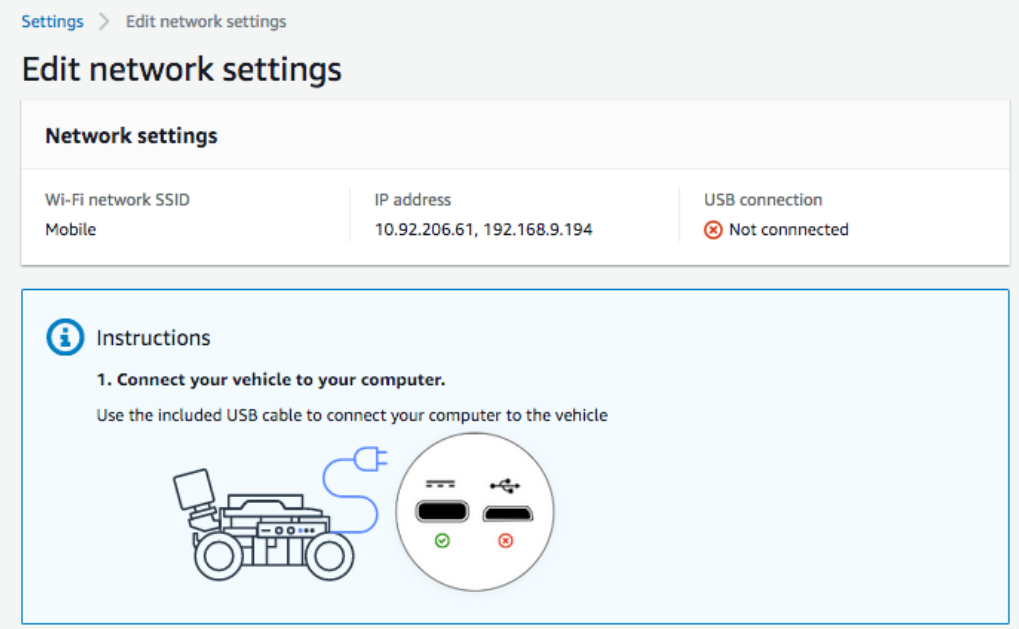
1. With your AWS DeepRacer vehicle connected to the Wi-Fi network, follow the instructions (p. 92) to sign in to the vehicle's device control console.

2. Choose **Settings** from the main navigation pane.
3. On the **Settings** page, perform one or more of the following tasks of your choosing.



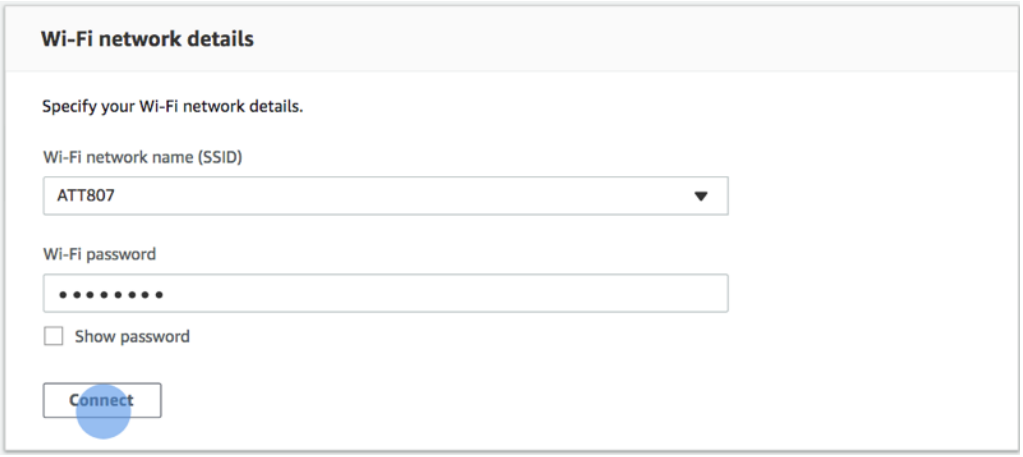
   a. To choose another Wi-Fi network, choose **Edit** for **Network settings** and then follow the steps below.

      i. Follow the instructions, shown on **Edit network settings**, to connect your vehicle to your computer using the USB-to-USB-C cable. After the **USB connection** status becomes **Connected**, choose the **Go to deepracer.aws** button to open the device console login page.



ii. On the device console login page, type the password printed on the bottom of your vehicle and then choose **Access vehicle**.

iii. Under **Wi-Fi network details**, choose a Wi-Fi network from the drop-down list, type the password of the chosen network, and then choose **Connect**.



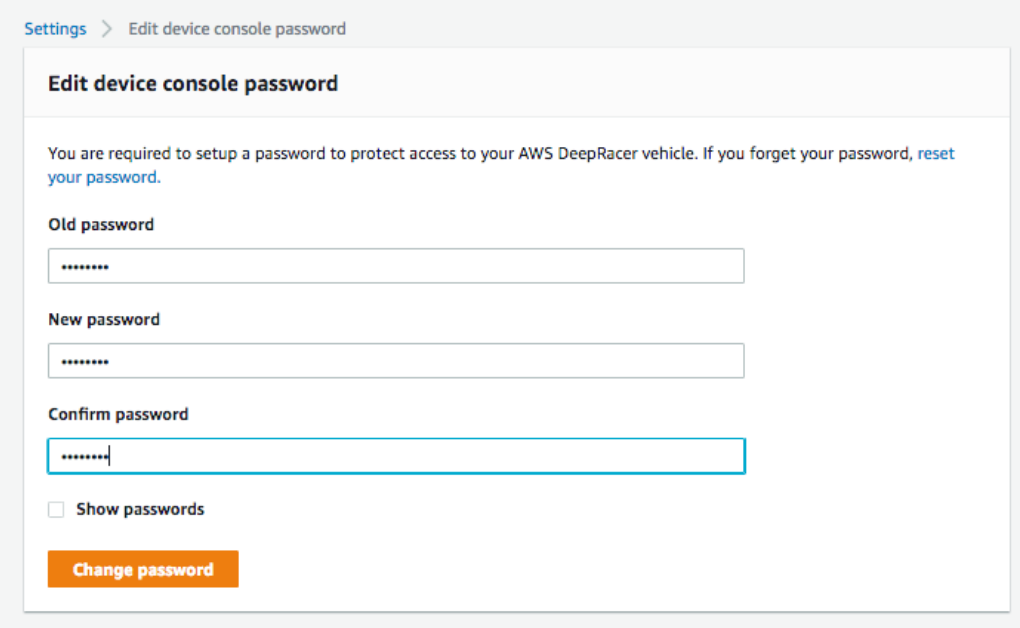
iv. After the **Vehicle status** for the Wi-Fi connection becomes **Connected**, choose **Next** to return to the **Settings** page of the device console, where you'll see a new IP address of the vehicle.

b. To reset the password for signing in to the device console, choose **Edit** for **Device console password** and then follow the steps below.

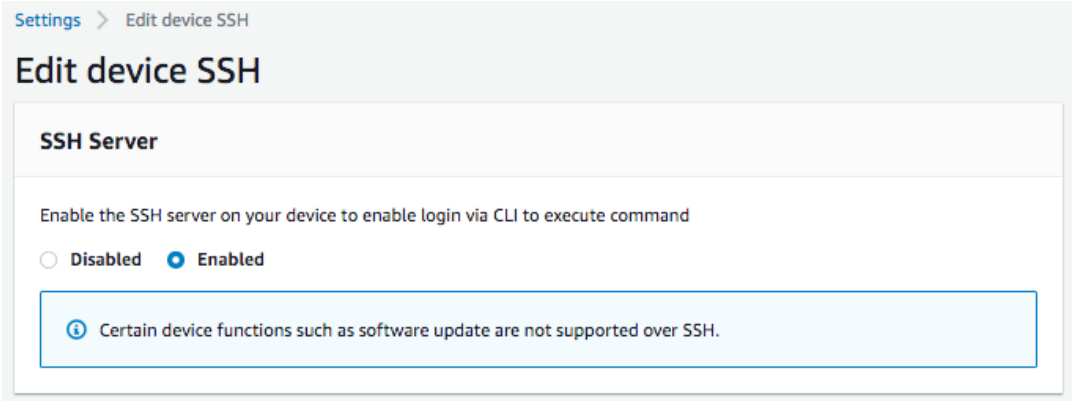
i. On **Edit device console password** page, type a new password in **New password**.

ii. Retype the new password in **Confirm password** to confirm your intention for the change. The password value must be the same before you can move on.

iii. Choose **Change password** to complete the task. This option is activated only if you have entered and confirmed a valid password value in the steps above.

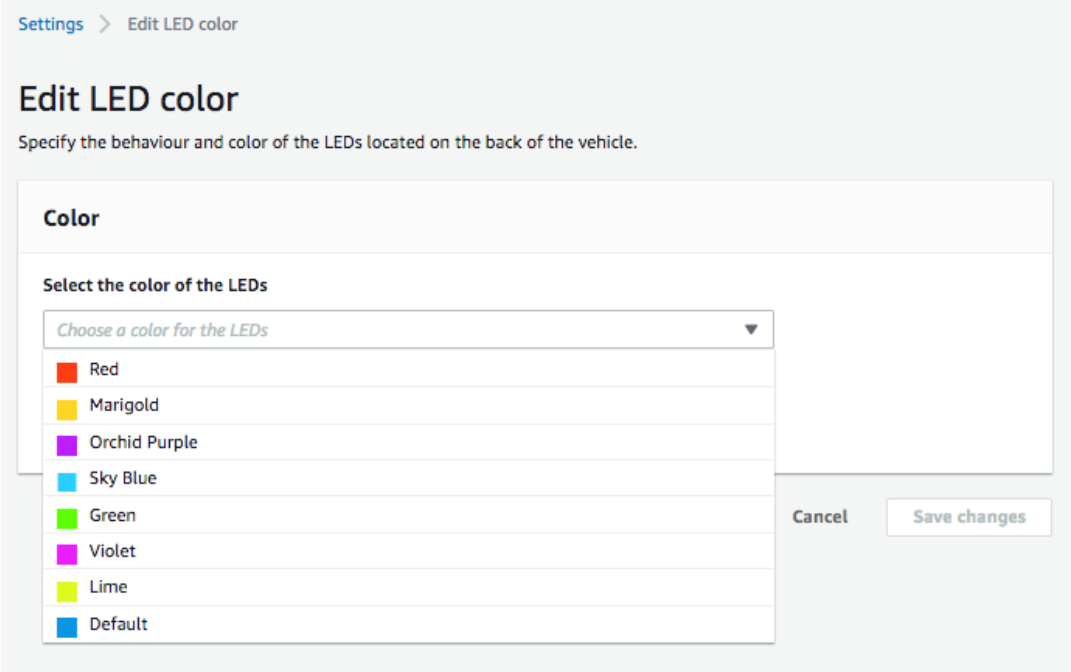


c. To enable or disable SSH connection to the vehicle, choose **Edit** for **Device SSH** and then choose **Enable** or **Disable**.



4. To change the vehicle's trail light LED color to distinguish your vehicle on a track, choose **Edit** for **LED color** on the **Settings** page and do the following.

   a. Choose an available color from the **Select the color of the LEDs** drop-down list on the **Edit LED color** page.



You should choose a color that can help identify your vehicle from other vehicles sharing the track at the same time.

b. Choose **Save changes** to complete the task.

   The **Save changes** functionality becomes active only after you have chosen a color.

5. To inspect the device software and hardware versions and to find out the system and camera configurations, check the **About** section under **Settings**.
6. To inspect the vehicle battery's charge level, check the lower part of the primary navigation pane.

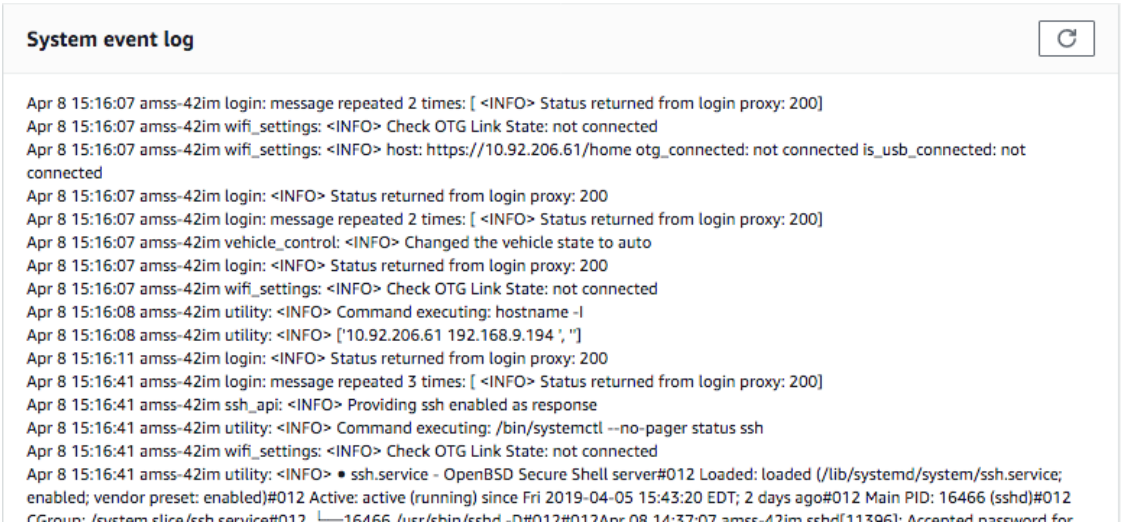
View Your AWS DeepRacer Vehicle Logs

Your AWS DeepRacer vehicle logs operational events that can be helpful for troubleshooting issues encountered in running your vehicle. There are two types of AWS DeepRacer vehicle logs:

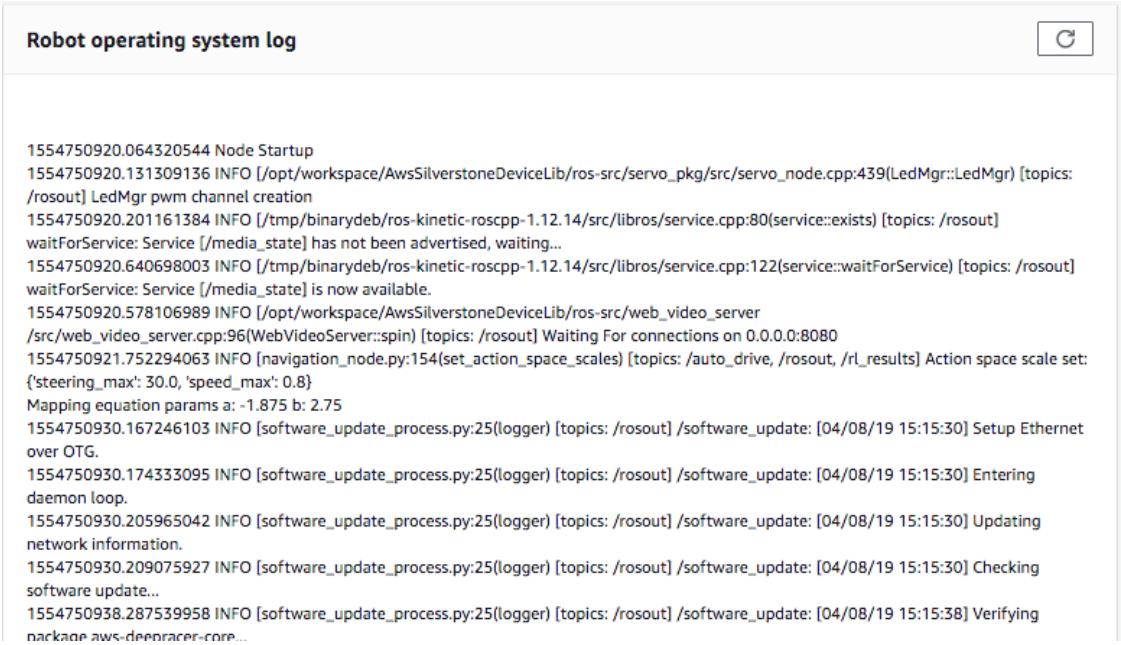
- The system event log keeps track of operations taking place in the vehicle's computer operating system, such as process managing, Wi-Fi connecting or password reset events.
- The robot operating system logs record statuses of operations taking place in the vehicle's operating system node for robotic operations, including vehicle driving, video streaming and policy inferencing operations.

To view the device logs, follow the steps below.

1. With your AWS DeepRacer vehicle connected to the Wi-Fi network, follow the instructions (p. 92) to sign in to the vehicle's device control console.
2. Choose **Logs** from the device console's main navigation pane.
3. To view the system events, scroll down the event list under **System event log**.



4. To view the robot operating system events, scroll down the event list under **Robot operating system log**.



Build Your Physical Track for AWS DeepRacer

This section describes how you can build a physical track for a AWS DeepRacer model. To drive your AWS DeepRacer autonomously and to test your reinforcement learning model in a physical environment, you need a physical track. Your track resembles the simulated track used in training and replicates the environment used to train the deployed AWS DeepRacer model.

Topics

- Track Materials and Build Tools (p. 109)
- Lay Your Track for AWS DeepRacer (p. 110)
- AWS DeepRacer Track Design Templates (p. 114)

Track Materials and Build Tools

Before you start to construct you track, get the following materials and tools ready.

Topics

- Materials You May Need (p. 109)
- Tools You May Need (p. 109)

Materials You May Need

To build a track, you need the following materials:

- For track borders:

  You can create a track with tape that is about 2-inches wide and white or off-white color against the dark-colored track surface. For a dark surface, use a white or off-white tape. For example, 1.88 inch width, pearl white duct tape or 1.88 inch (less sticky) masking tape.
- For track surface:

  You can create a track on a dark-colored hard floor such as hardwood, carpet, concrete, or asphalt felt. The latter mimics the real-world road surface with minimal reflection. Interlocked foam or rubber pads are also good options.

Tools You May Need

The following tools are either required or helpful to design and build your track:

- Tape measure and scissors

  A good tape measure and a pair of scissors are essential for building your track. If you don't already have one, you can order a tape measure here or scissors here.

- Optional design tools

  To design your own track, you might need a protractor, a ruler, a pencil, a knife and a compass.

Lay Your Track for AWS DeepRacer

When you build your track, it's a good practice to start with a simple design, such as a straight or single-turn track. Next you can move on to looped tracks. Here, we use a single-turn track as an example to walk you through the steps to construct your own track. First let's review dimensional requirements of a track.

Topics

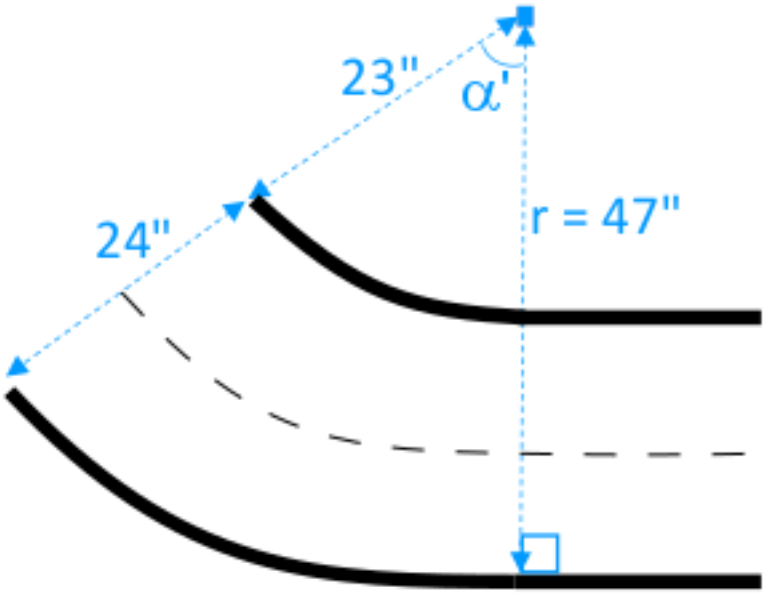
- Dimensional Requirements (p. 110)
- Considerations for Model Performance (p. 111)
- Steps to Build the Track (p. 111)

Dimensional Requirements

You can build a track of any shape as long as it meets the following requirements:

- **Minimum turning radius**:

  On a curved track, the turning radius (r) measures from the circle center to the outside border, as illustrated below.



The minimum turning radius (r_{min}) depends on the track turning angle (α) at a corner and should comply to the following limits:

- If the track's turning angle is `# ≤ 90 degrees`,

  $r_{min} \geq 25$ inches

  We recommend 30 inches.

- If the track's turning angle is `# > 90 degrees`, α

  $r_{min} \geq 30$ inches.

  We recommend 35 inches.

- **Track width**,

  The track width (w_{track}) should comply to the following limit:

  $w_{track} \geq 24 \pm 3$ inches.

- **Track surface**:

  The track surface should be smooth and of a uniform dark color. The minimum enclosing area should be `30 inches x 60 inches` in size.

  Carpeted and wood floors work well. Interlocked foam or rubber pads match the simulated environment better than wood, but this is not required. Concrete floors can be problematic due to light reflection on the surface.

- **Track barrier**

  Though not required, we recommended that you encircle the track with uniform-colored barriers that are at least 2.5 feet tall and 2 feet away from the track at all points.

Considerations for Model Performance

How you build a track can affect the reliability and performance of a trained model. The following are factors you should consider when building your own tracks.

1. Do not place any white objects on or near your track. If necessary, remove any white object from the track or its vicinity. This is because training in the simulated environment assumes that only the track borders are white.
2. Use clean and continuous tape to mark the track borders. Broken or creased track borders can affect the trained model performance.
3. Avoid using a reflective surface as the track floor. Reduce glare from bright lights. The glare from straight edges can be misinterpreted as objects or borders.
4. Do not use a track floor with line markings other than the track lines. The model might interpret the non-track lines as part of the track.
5. Place barriers around the track to help reduce distractions from background objects.

Steps to Build the Track

As an illustration, we use the most basic single-turn track. You can modify the instructions to create a more complex track such as an S-curve, a loop, or the AWS re:invent 2018 track.

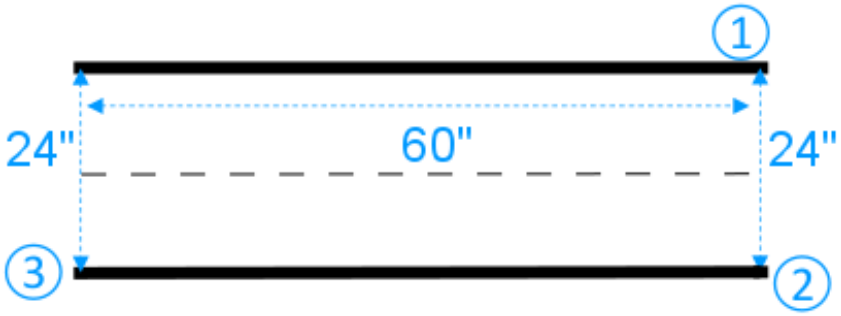
To build an AWS DeepRacer single-turn track

1. To construct the straight portion of the track, follow the steps below and refer to the diagram.

   a. Put a 60-inch long piece of tape on the floor to lay down the first border in a straight line (1).

   b. Use a tape measure to locate the second border's two end points, (2) and (3). Put them 24 inches apart from the first border's two ends.

   c. Put another 60-inch long piece of tape on the floor to lay down the second boarder to connect the two endpoints (2) and (3).

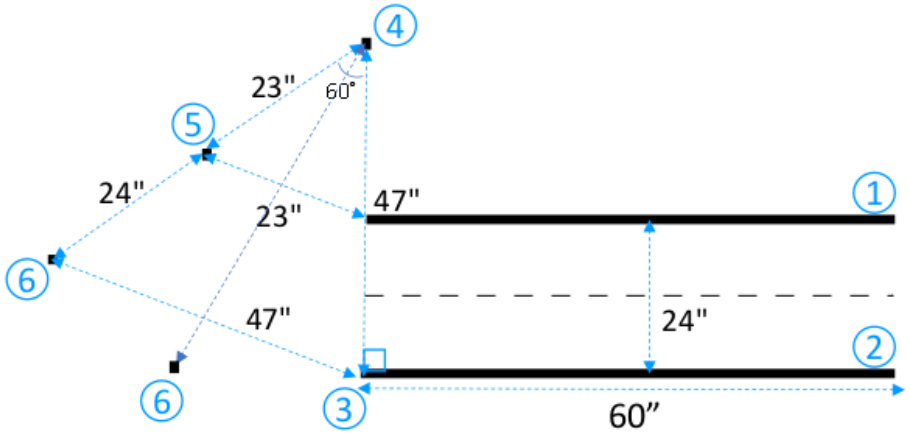


   We assume the straight track segment is 60-inches long and 24-inches wide. You can adjust the length and width to fit to your space, provided that the dimensional requirements are met.

2. To make the track to turn at a 60-degree angle, do the following and refer to the diagram:

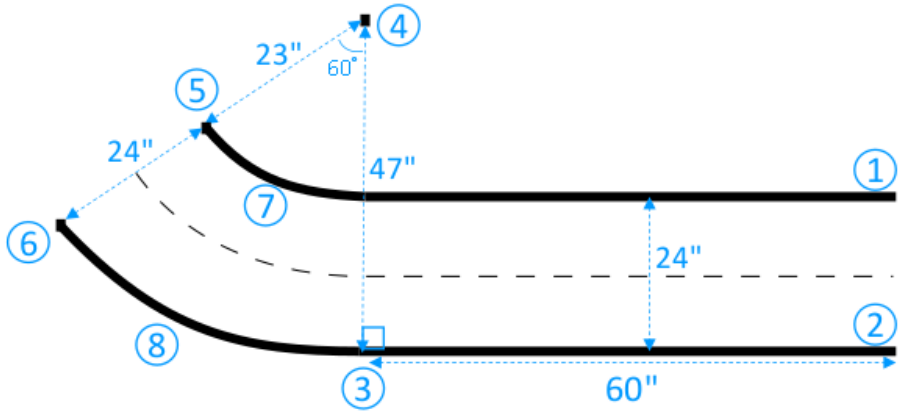
   a. Use the tape measure to locate the center (4) of the turning radius (4–3 or 4–6). Mark the center with a piece of tape.

   b. Draw an equilateral triangle. The three sides are (3–4), (4–6), and (6–3).



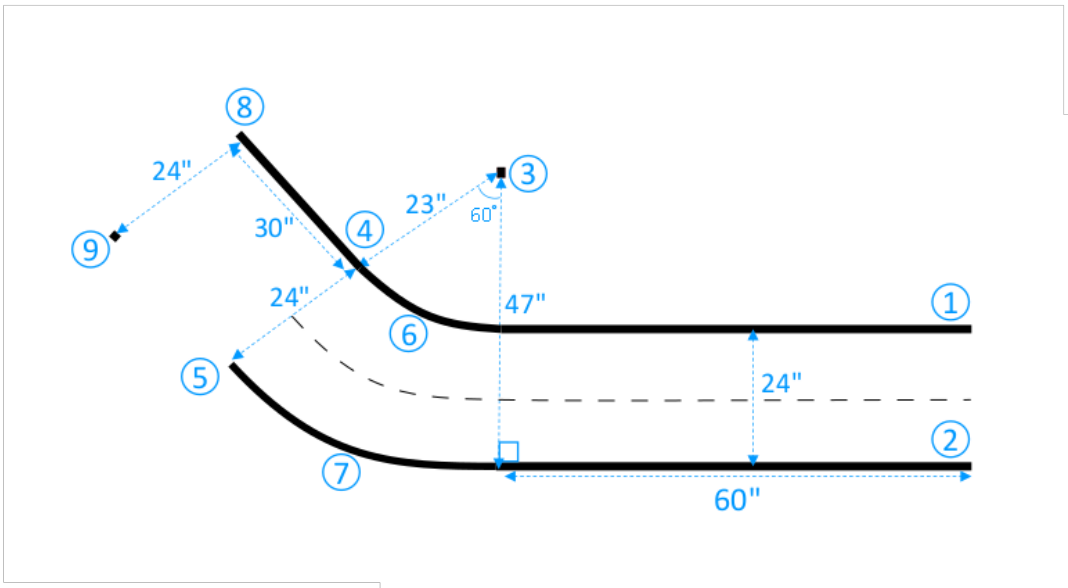
To make a 60-degree turn along the track, use the equilateral triangle (3–4–6) to determine the locations of the two final end points (5) and (6) for the curved track segment. For turns at a different angle, you can use a protractor (or a protractor app) to locate the two final ends (5) and (6) of the curved track segment. Turning radius variations are acceptable as long as the minimum turning radius requirement in Step 2 is met.

c. Put small tape segments, e.g. 4-inches each, on the floor to lay the curved border segments (7) and (8) and connect them with the straight-line borders. The two curved borders don't need to be parallel.



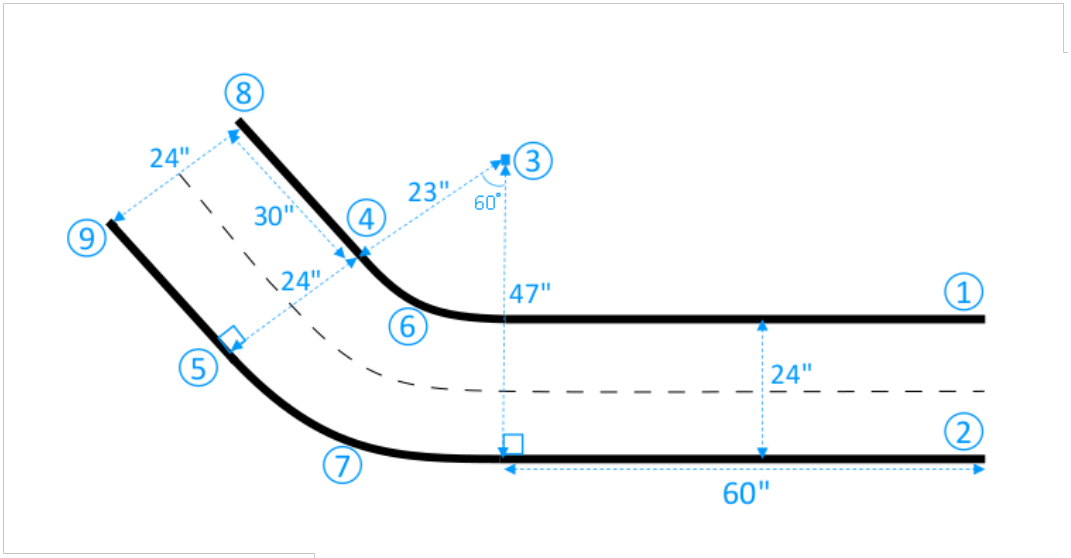
3. To extend the track with the next straight segment of 30 inches long and 24 inches wide, do the following:

a. Put a 30-inch long piece of tape on the floor to lay down the first border (4-8) perpendicular to the edge (3-5).



b. Use the tape measure to locate the ending point of the second border (9). You can customize the length of the straight lines to fit to the space you have.

c. Put another 30-inch long piece of tape on the floor to lay down the second border (5-9) perpendicular to the edge (3-5).



We assume the second straight track segment is 30 inches long and 24 inches wide. You can adjust the length and width to fit to your space, provided that the dimensional requirements are met and the dimensions are consistent with other track segments.

4. Optionally, cut tape segments of 4 inches long and then place the tape segments 2 inches apart along the track center to lay the dashed center line.

You've now finished building the single-turn track. To help your vehicle to better distinguish the drivable surfaces from non-drivable surfaces, you should paint the off-track surface a color of sufficient contrast with respect to the on-track surface color. To ensure safety, you could encircle the track with uniform-colored barriers that are at least 2.5 feet tall and 2 feet away from the track at all points.

You can apply the instructions to extend the track to more complex shapes (p. 114).

AWS DeepRacer Track Design Templates

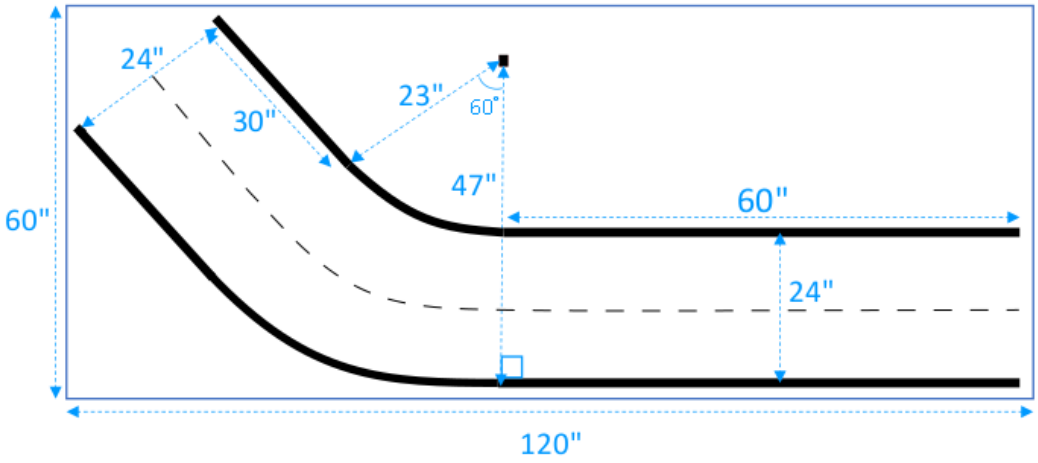
The following track design templates show AWS DeepRacer tracks that you can build by following the instructions (p. 110) presented in this section.

Topics

- AWS DeepRacer Single-Turn Track Template (p. 115)
- AWS DeepRacer S-Curve Track Template (p. 115)
- AWS DeepRacer Loop Track Template (p. 116)
- AWS DeepRacer AWS re:Invent 2018 Track Template (p. 116)
- AWS DeepRacer Championship Cup 2019 Track Template (p. 118)

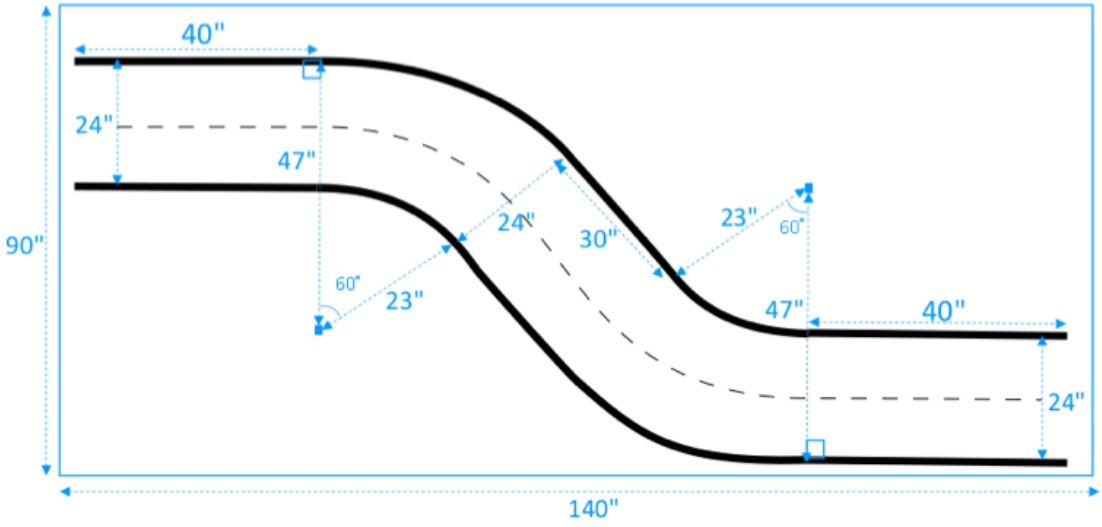
AWS DeepRacer Single-Turn Track Template

This basic track template consists of two straight track segments connected by a curved track segment. Models trained with this track should make your AWS DeepRacer vehicle drive in straight line or to make turns in one direction.



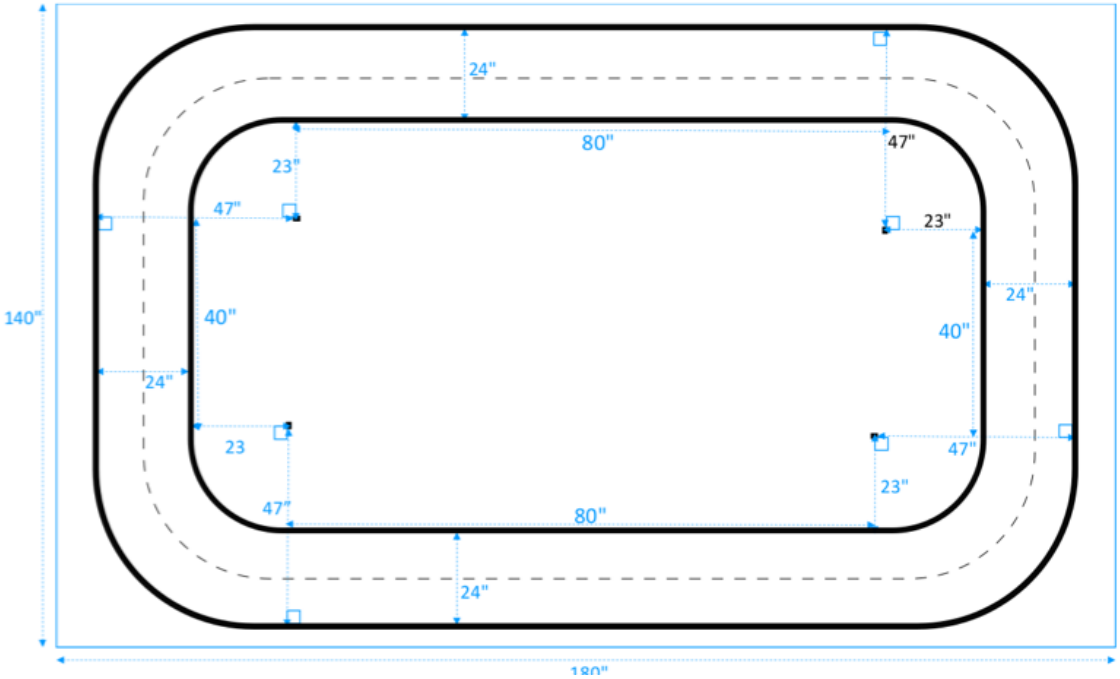
AWS DeepRacer S-Curve Track Template

The track is more complex than the single-turn track because the model needs to learn to make turns in two directions. You can easily extend the single-turn track construction instructions to this track by turning it in the opposite direction after the first turn.



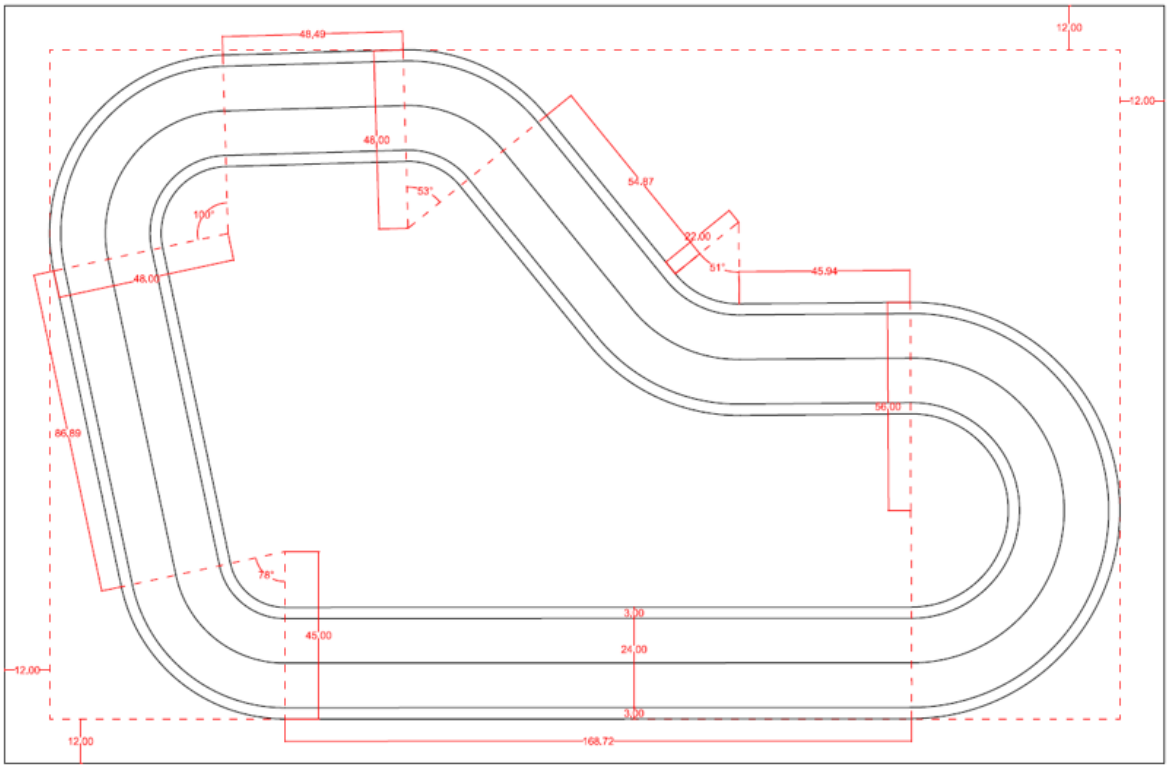
AWS DeepRacer Loop Track Template

This regular loop track is a repeating, 90-degree, single-turn track. Obviously, it requires a larger enclosing area for laying the entire track.



AWS DeepRacer AWS re:Invent 2018 Track Template

This less regular loop track was first presented at the AWS re:Invent 2018. Models trained on this track must learn how to turn left and right at various angles and how to move straight ahead in various directions.



At AWS re:Invent 2018, the track size measured 26 feet long by 17 feet wide. To change the scale, be aware that the scaled-down track may become too narrow to fit the model trained on a wider track.

To reproduce the same color production, use the following color specifications:

- Green: PMS 3395C
- Orange: PMS 137C
- Black: PMS 432C
- White: CMYK 0-0-2-9

This track was tested with the following materials for its surface:

- Carpet

  The track was printed on 8-ounce, dye-sublimated, polyester-faced carpet with latex rubberized backing. Carpet is durable, provides great performance, but is expensive.
- Vinyl

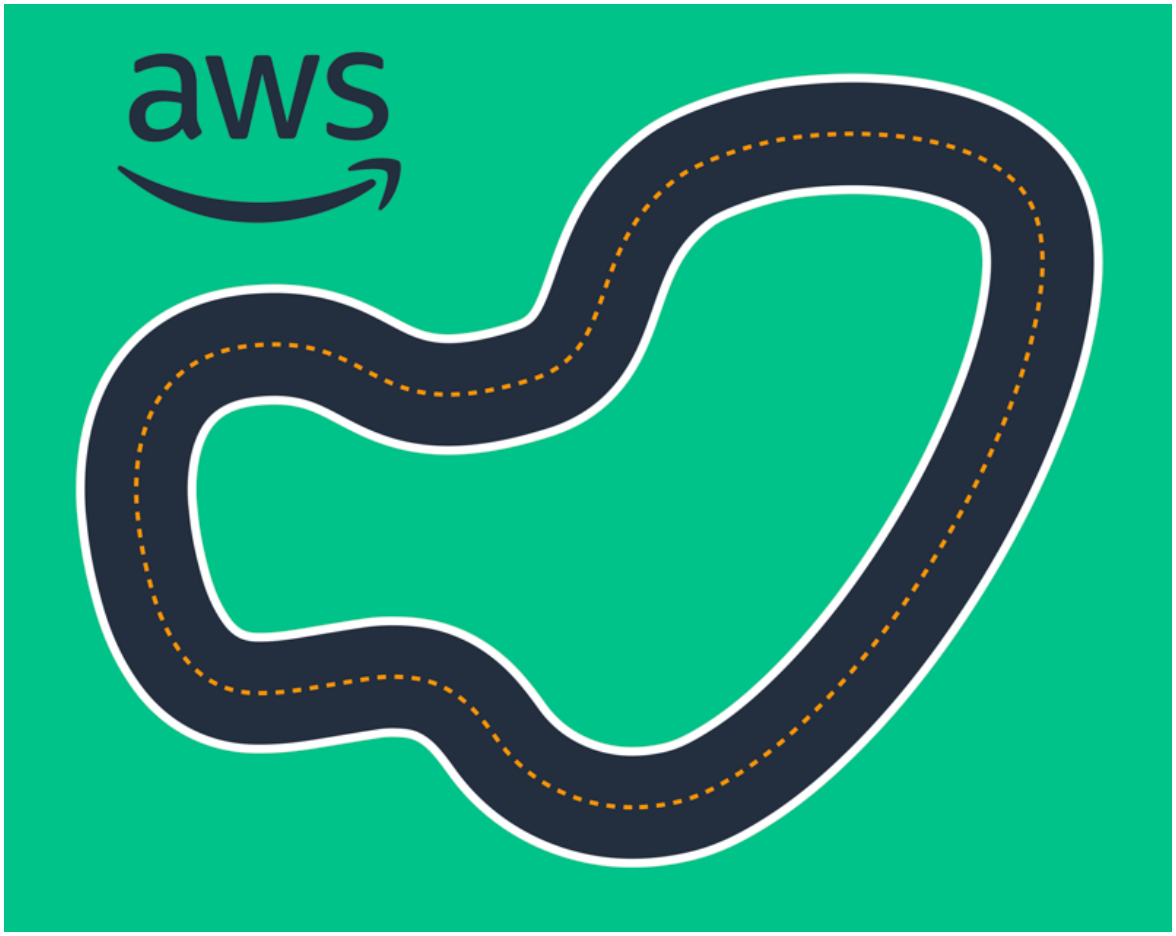
  The track was printed on 13-ounce scrim vinyl with a matte finish to reduce glare. Vinyl is typically cheaper than carpet and provides good performance. Vinyl is not as durable as carpet.

Due to its large size, the track cannot be easily printed on a single piece of material. Align track lines well when connecting pieces together.

For more detailed information, see the AWS DeepRacer re:Invent 2018 track specification.

AWS DeepRacer Championship Cup 2019 Track Template

This track was used during the AWS DeepRacer 2019 Championship Cup finals. It's intended to train models for head-to-head racing.



The track is 34 feet long and 27 feet wide. Please see dimensions in the track files below.

To print or create your own Championship Cup track, download the AWS DeepRacer Championship Cup 2019 track files here.

Participate in AWS DeepRacer Virtual Races

After successfully training and evaluating your model in simulations, you may want to compare your model's performance to other racers' models. The comparison can help you measure the fitness of your model. To support this, AWS DeepRacer lets you organize your own community races for users to participate, in addition to the AWS-sponsored AWS DeepRacer League racing events. The races can be online (virtual) or in-person. This section discusses how to participate in an AWS DeepRacer League Virtual Circuit race or a community-based virtual race.

When participating in an online race, you submit your model to the virtual leaderboard of the racing event. The AWS DeepRacer console will choose the most optimal version of your model during the training job simulation. It automatically picks your best model by cycling between training and checkpoint evaluations. If the evaluation of your model passes the specified racing criteria, the result displayed on the leaderboard and the performance ranked against other participants.

AWS DeepRacer Racing Event Types

An event can be categorized by its sponsor or organizer. Both AWS DeepRacer League and community racing events can take place in person on a physical track or online on a virtual track.

- **AWS-sponsored racing events** – Racing events sponsored by AWS are referred to as AWS DeepRacer League events and are open for any AWS DeepRacer users.
  - AWS DeepRacer League Summit Circuit races are in-person events.
  - AWS DeepRacer League Virtual Circuit are online events.

- **Community-sponsored racing events** – Racing events created by AWS DeepRacer users are called community racing events.

Joining an Online AWS-sponsored or Community-Sponsored Race

You can use the AWS DeepRacer console to enter an AWS DeepRacer League Virtual Circuit event or a community-based online race.

- Any AWS DeepRacer user can join any open online race in AWS DeepRacer League Virtual Circuit.
- Only invited users can access or participate in community racing virtual events. Users are invited when they receive an invitation link sent by the race organizer or forwarded by another race participant.

Topics

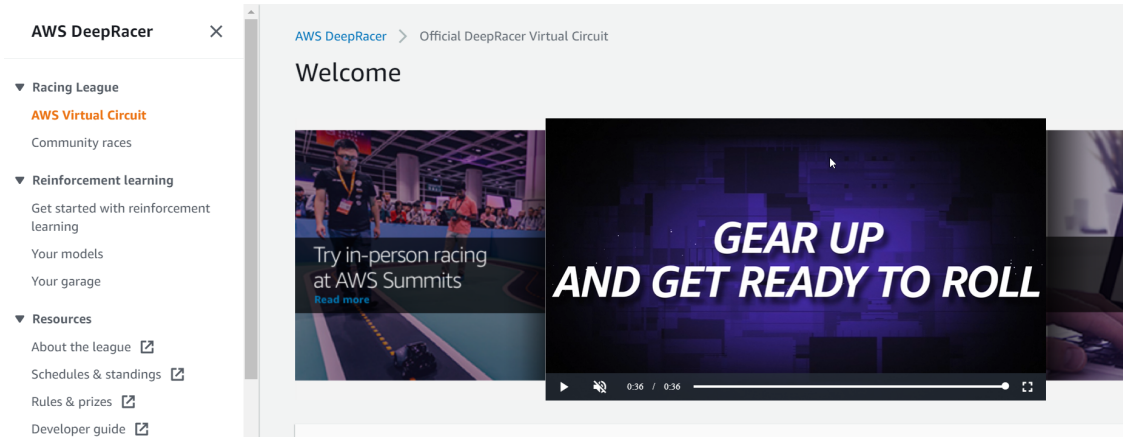
- the section called “Join a Virtual Circuit Race” (p. 120)
- the section called “Join a Community Race” (p. 123)
- the section called “Organize a Community Race” (p. 126)
- the section called “Manage Community Races” (p. 128)
- the section called “Racing Event Terminology” (p. 6)

Join an AWS DeepRacer League Virtual Circuit Race

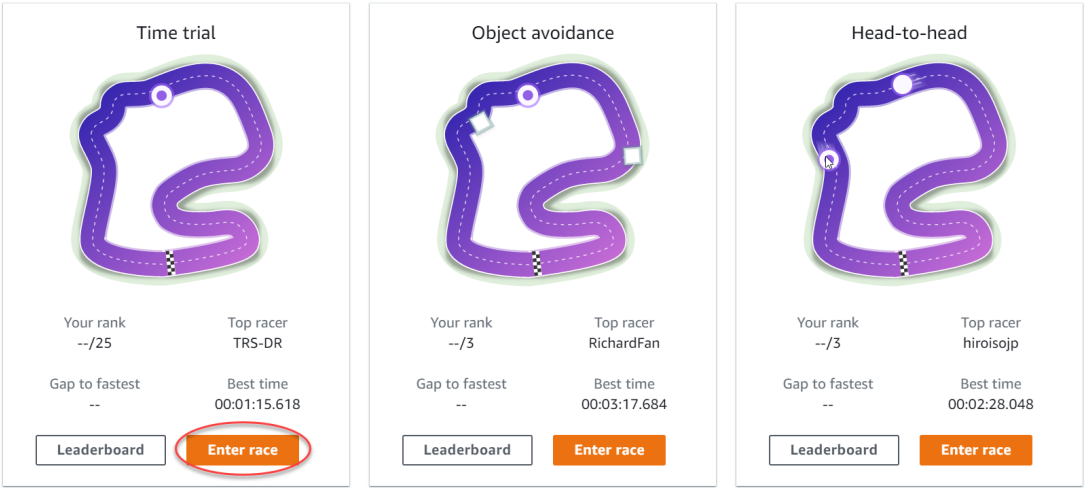
In this section, learn how to use the AWS DeepRacer console to submit your trained model to a Virtual Circuit race.

To enter in the AWS DeepRacer League Virtual Circuit

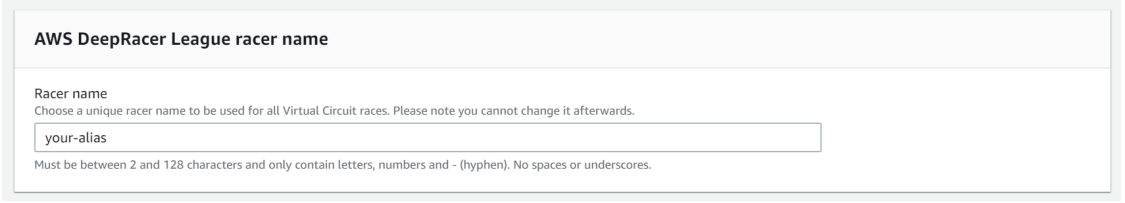
1. Sign in to the AWS DeepRacer console at https://console.aws.amazon.com/deepracer.
2. From the main navigation pane, choose **AWS Virtual Circuit**.



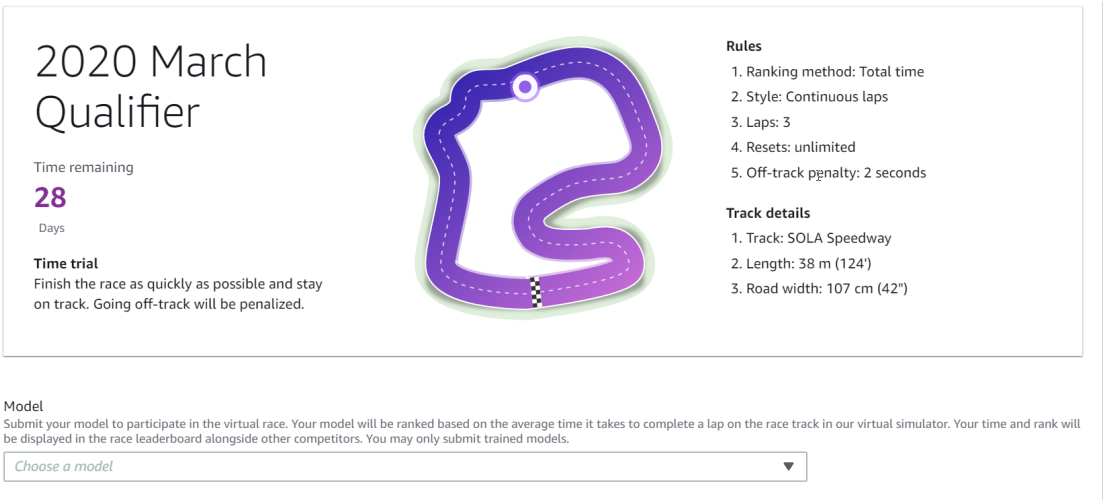
3. On the **AWS Virtual Circuit** page, select a race type, and then choose **Enter race.**



4. If this is your first time participating in an AWS DeepRacer League racing event, set your alias in **Racer name** under **AWS DeepRacer League racer name**. You can't change your racing alias after you create it.



5. Under **Model submission details**, choose your model from the **Model** list.



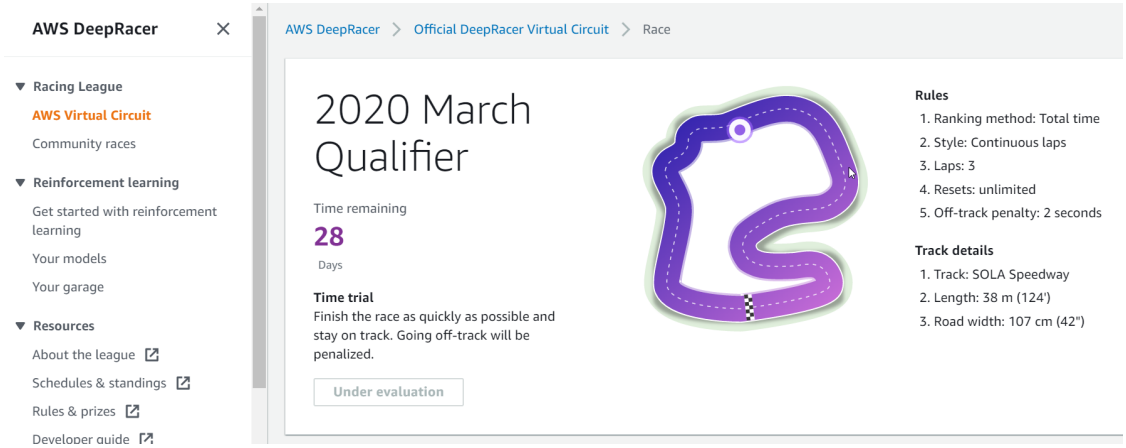
6. If this is your first time participating in an AWS DeepRacer League event, under **User terms and services**, select the **I have read and agree to the terms and conditions of the DeepRacer League** check box.

   Follow the link to read the terms and conditions before selecting this option.

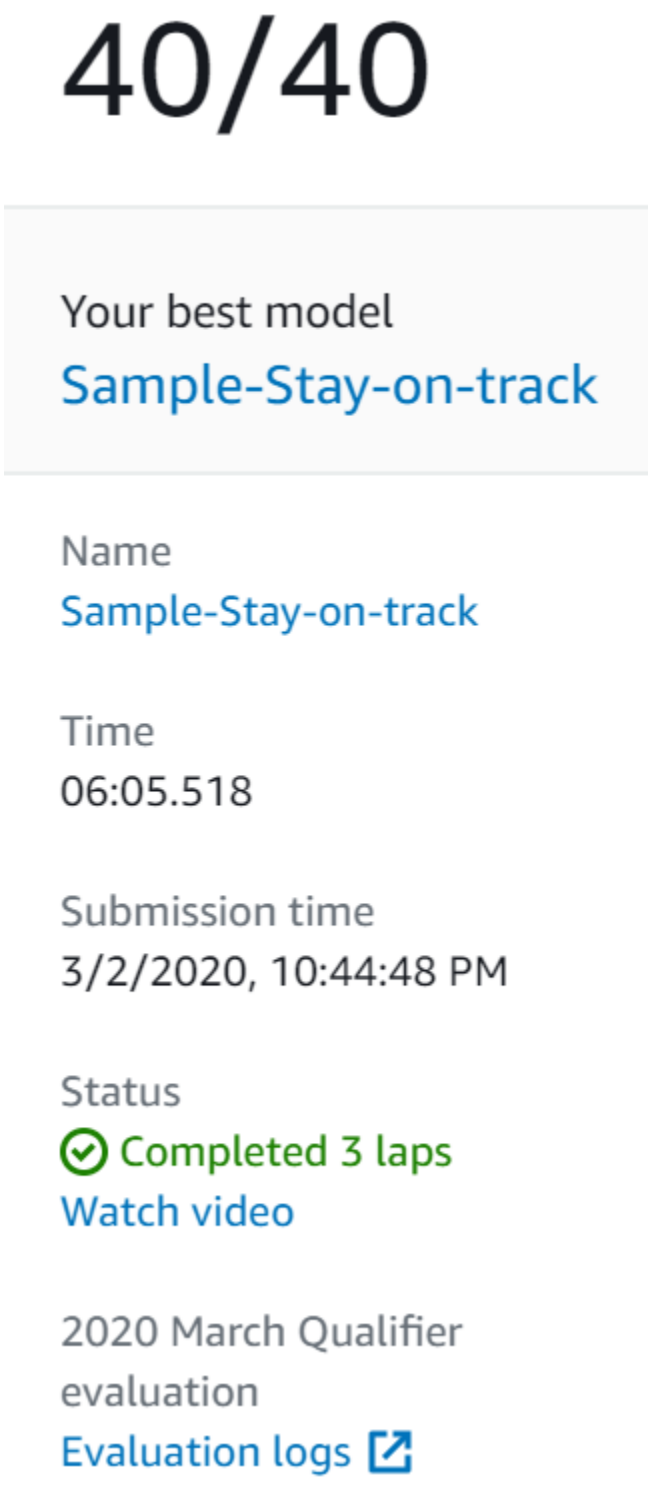
7. Choose **Submit model** to complete the submission.

   After your model is submitted, the AWS DeepRacer console starts its evaluation. The process can take up to 10 minutes.

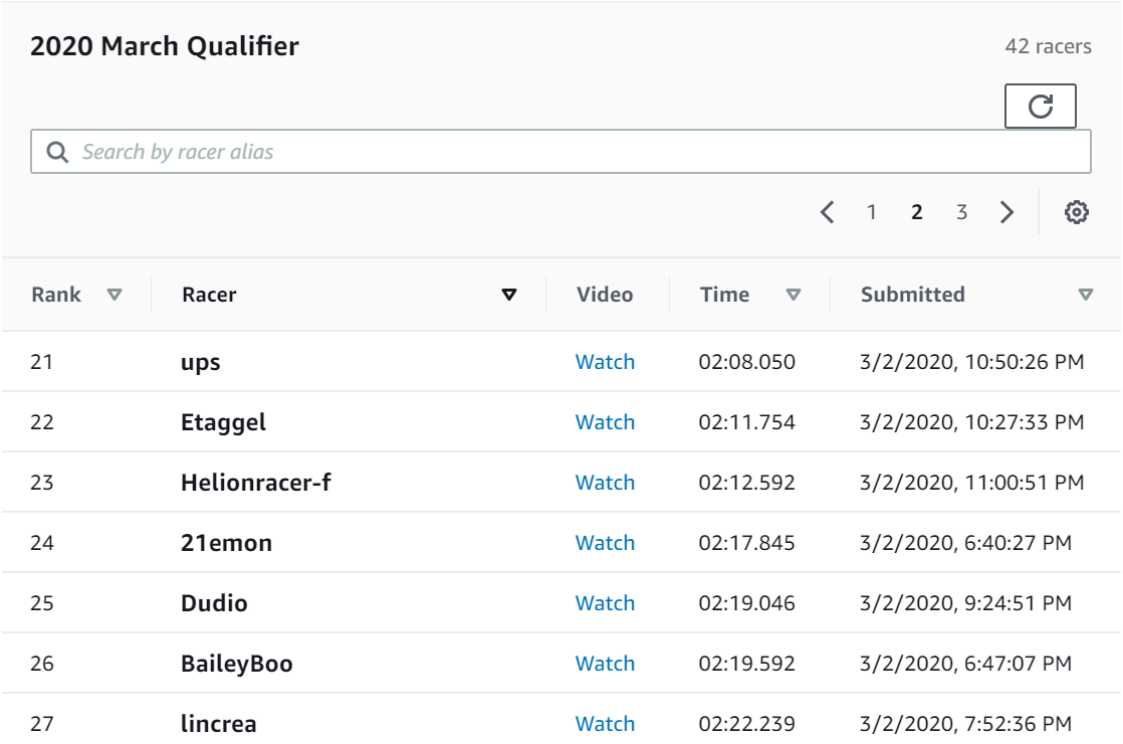
8. On the race page, inspect the race track to ensure that your model has been trained to handle the track shape.



9. On the race page, view your submission status under your racer name.



10. On the race page, inspect the ranking list on the leaderboard to see how your model compares with others.



2020 March Qualifier — 42 racers

Search by racer alias

1 2 3

Rank	Racer	Video	Time	Submitted
21	ups	Watch	02:08.050	3/2/2020, 10:50:26 PM
22	Etaggel	Watch	02:11.754	3/2/2020, 10:27:33 PM
23	Helionracer-f	Watch	02:12.592	3/2/2020, 11:00:51 PM
24	21emon	Watch	02:17.845	3/2/2020, 6:40:27 PM
25	Dudio	Watch	02:19.046	3/2/2020, 9:24:51 PM
26	BaileyBoo	Watch	02:19.592	3/2/2020, 6:47:07 PM
27	lincrea	Watch	02:22.239	3/2/2020, 7:52:36 PM

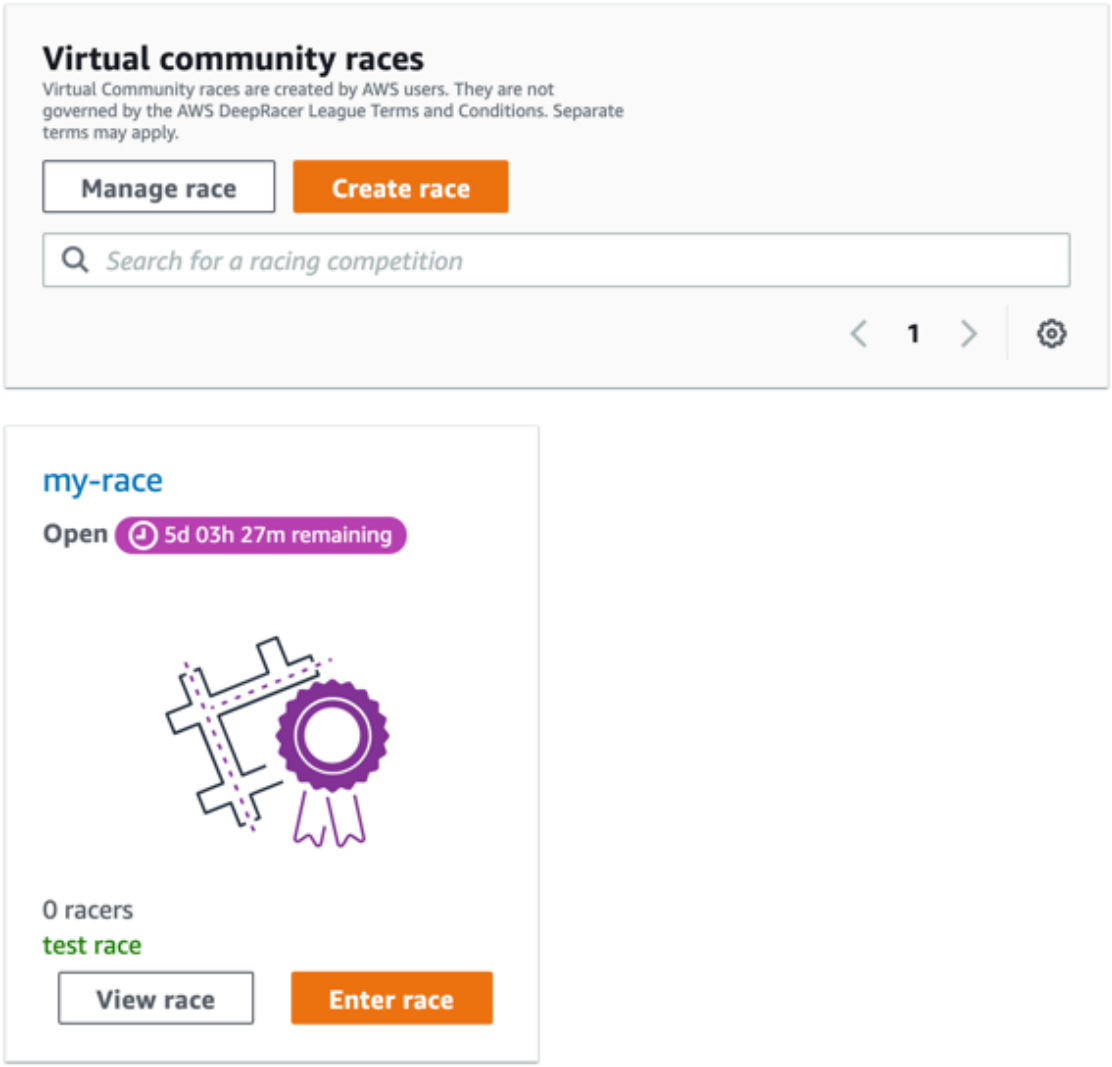
If your model can't finish the track in three consecutive trials, it is not included in the ranking list on the leaderboard. If a subsequent submission yields a faster result, your model's ranking is updated to reflect your best performance at the time.

Join an AWS DeepRacer Community Race

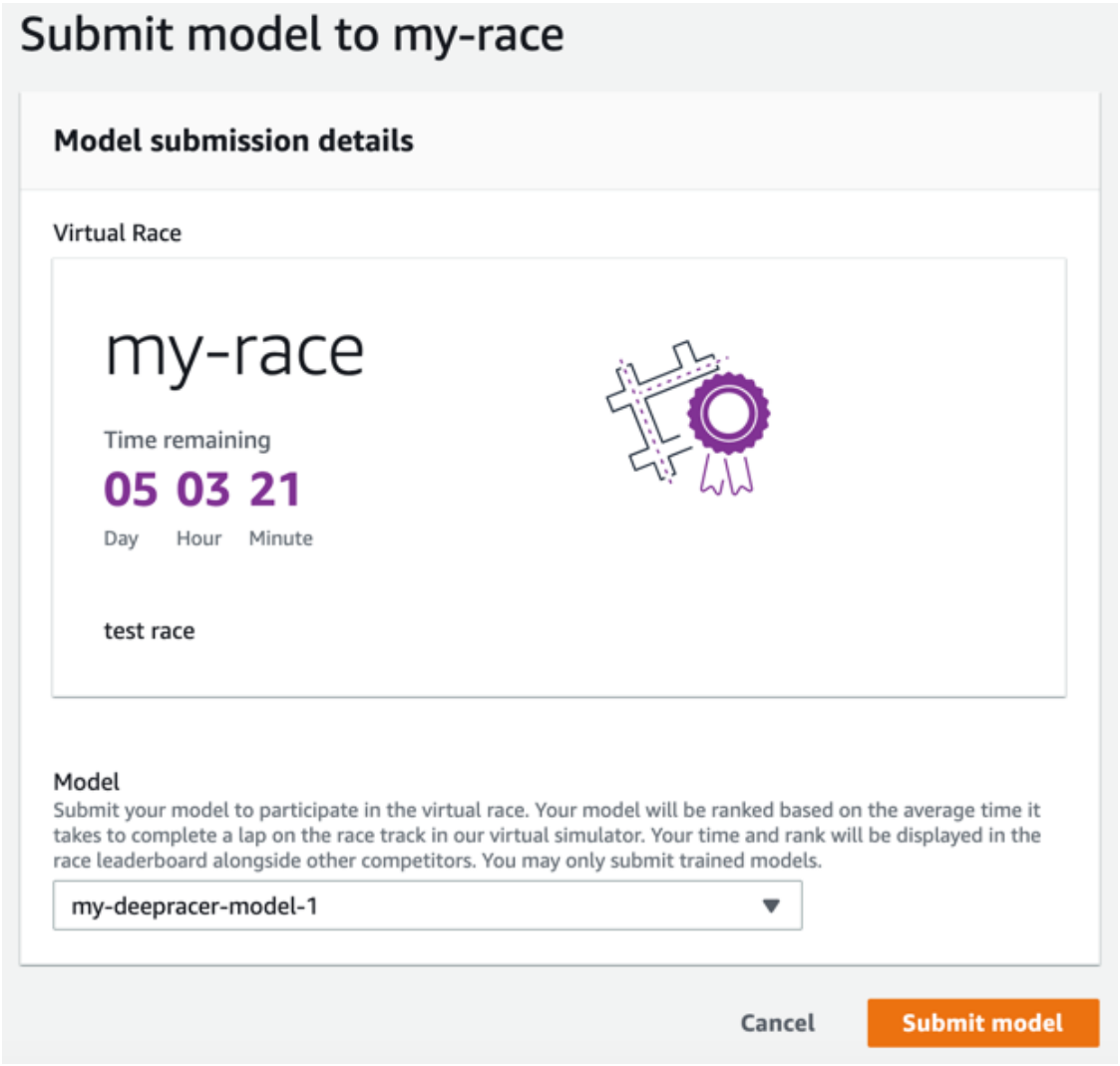
If you're invited to an active community race and you've entered an AWS DeepRacer race before, follow the steps below to join the invited race in the AWS DeepRacer console.

To join an AWS DeepRacer Community Race as a race participant

1. Sign in to your AWS account in the AWS DeepRacer console.
2. Choose **Community races** from the main navigation pane.
3. On the **Virtual community races** page and from the list of invited races, choose the race you want to enter and then choose **Enter race**.



4. On the **Submit model to *your race*** page, under **Model**, choose a trained model and then choose **Submit model**.



5. If your model is evaluated successfully against the racing criteria, watch the event's leaderboard to see how your model ranks against other participants.

If you're new to AWS and receive an invitation in an email message to join an AWS DeepRacer community race, choose the event link to go to the AWS DeepRacer console and then sign up for an AWS account before proceeding to join the race.

As a new AWS DeepRacer user or a first-time participant to any AWS DeepRacer race, follow the steps to join the invited race in the AWS DeepRacer console.

To join an AWS DeepRacer Community Race as a New AWS DeepRacer user

1. Sign in to your AWS account in the AWS DeepRacer console.
2. From the main navigation pane, under **Racing league**, choose **Community races**.
3. When prompted for your **AWS DeepRacer racer name**, type an racer name as your identification across all the AWS DeepRacer leaderboards.
4. On the invited race details page, expand **Get started racing**.
5. Choose **Get started with RL** to get a quick introduction to training ad AWS DeepRacer model for autonomous driving.

6. Train and evaluate your model for the invited race in the AWS DeepRacer console.

   For more information on training your model, see Train Your First AWS DeepRacer Model (p. 15).
7. Navigate to **Community races** to see the race you're invited to.
8. Watch your ranking on the event leaderboard, if your model is evaluated successfully against the racing criteria.

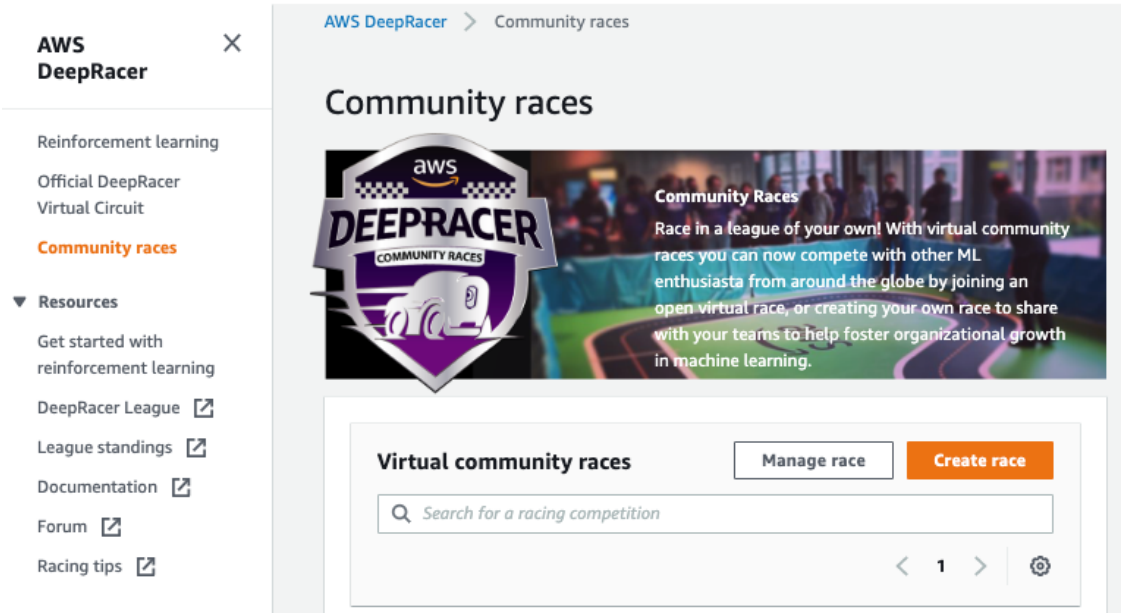
Organize an AWS DeepRacer Community Race

Community racing events are races organized by AWS DeepRacer users that are not officially sponsored by AWS.

To organize a community race for AWS DeepRacer users to join, you must create a community race and invite participants to join by sharing a race invitation link. You can do so in the AWS DeepRacer console.

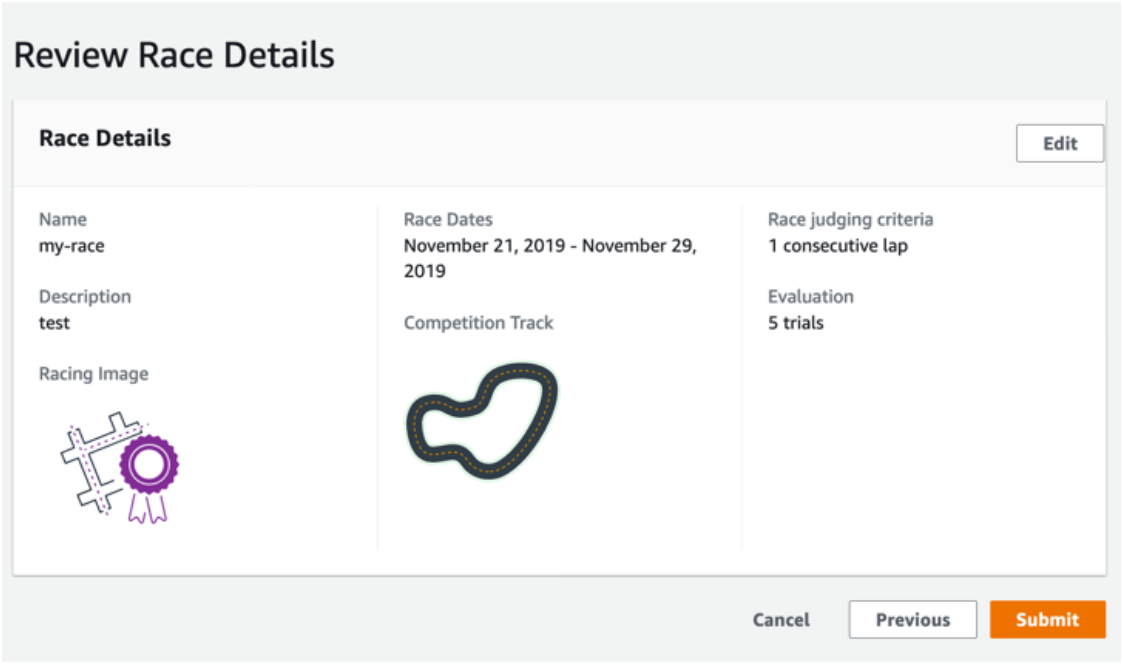
To create an AWS DeepRacer community race using the AWS DeepRacer console

1. Open the AWS DeepRacer console.
2. From the main navigation pane, choose **Community races**.
3. Under the **Community races**, choose **Create race**.

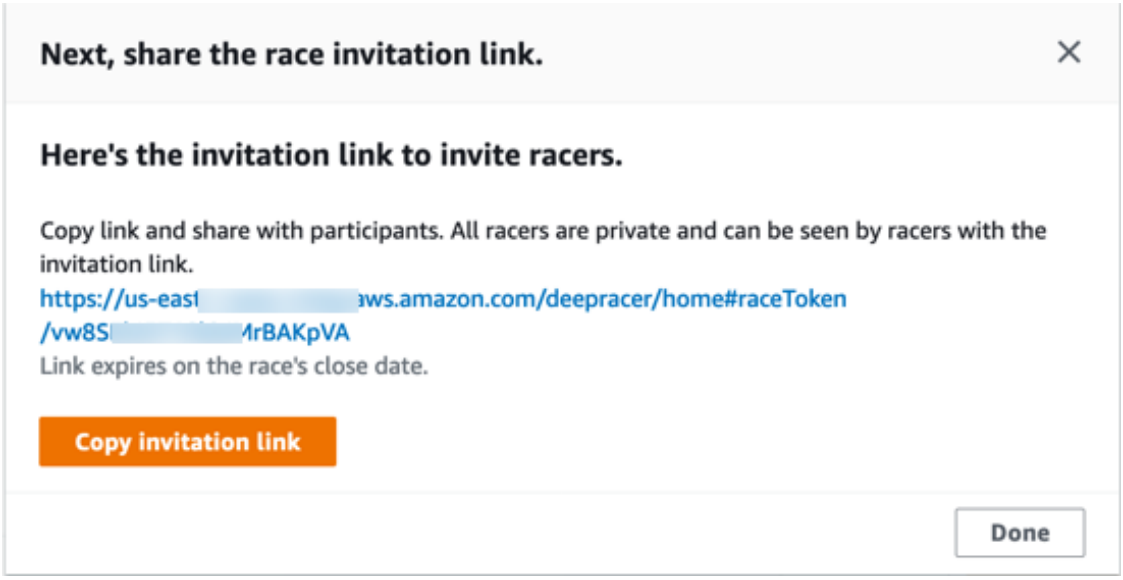


4. On the **Race Details** page, type a name for the racing event.
5. Provide a description for the event. The description should clearly summarize the goals and rules of the event for participants.
6. Specify the starting and closing dates of the event.
7. Choose a competition track. For beginners to race in your event, choose a simpler track. For advanced users, use a more challenging track.
8. (Optional) Choose a racing image.
9. Choose an entry requirement of a number of consecutive laps a racer must complete to qualify for submission of the result to the race's leaderboard. For a beginners' race, choose a smaller number. For advanced users, use a larger number.

10. Specify the number of trials in **Race evaluation** a racer must complete to qualify for the race.
11. Choose **Next**.
12. On the **Review Race Details**, choose **Submit**.



13. Copy the racing event's invitation link and share it with participants you want to invite. Then, choose **Done** to finish creating the event.



The recipients becomes invited participants when they follow the link to join the race (p. 125). A recipient can forward the link to invite another user to participate in the race. Users without the invitation link aren't aware of the event and can't access the race. However, the race organizer can deny any invited user to access the event by removing the participant from the race. This is part of the community race management tasks. The next section (p. 128) discusses these details.

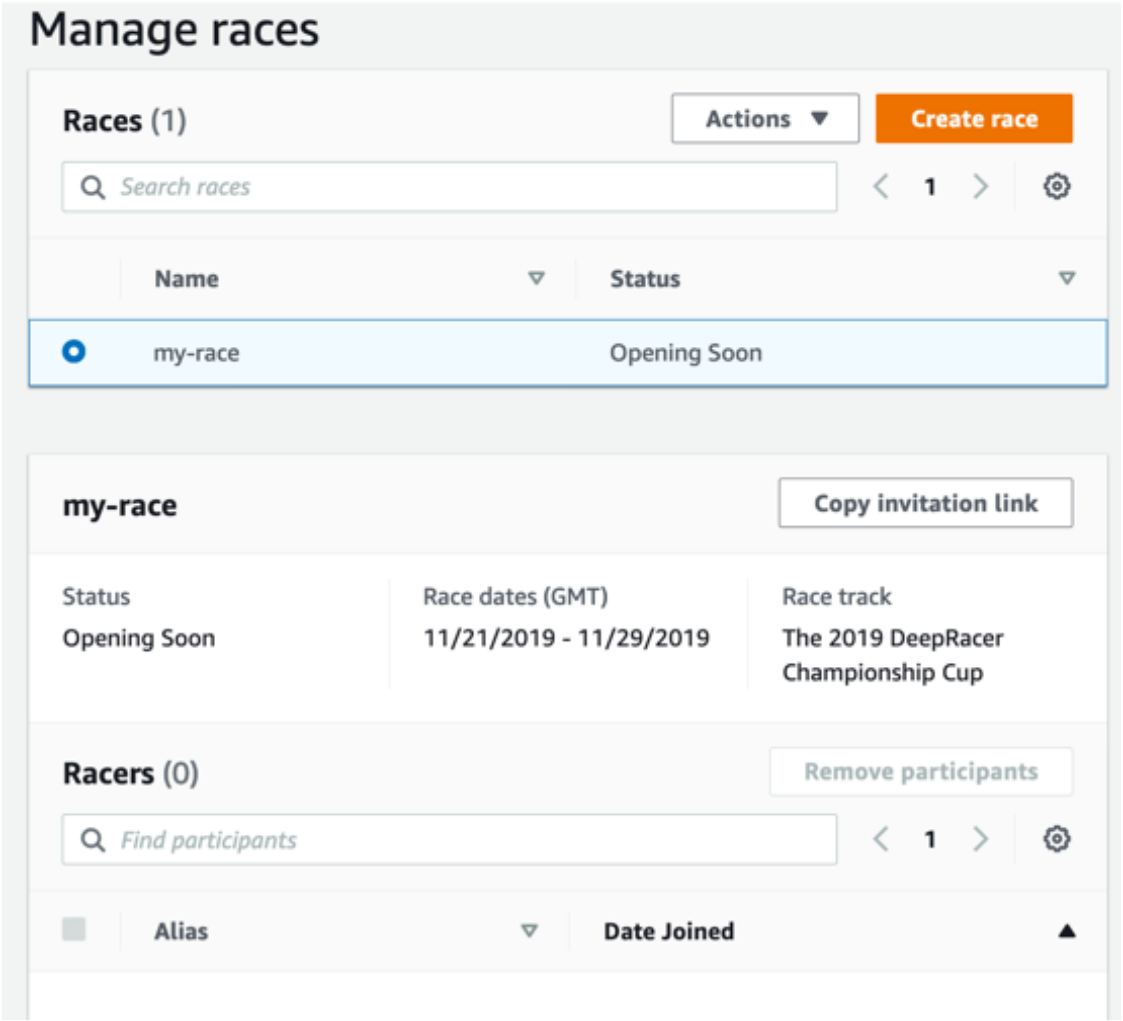
Manage an AWS DeepRacer Community Race

All community races are private and aren't visible to a user unless the user has an invitation link. Invitation links can be shared and forwarded to anyone. To join the race, participants need an AWS account. If this is their first time, users go through the account creation process before they can join the race.

As the race organizer, you can edit race details and remove participants as part of the race management.

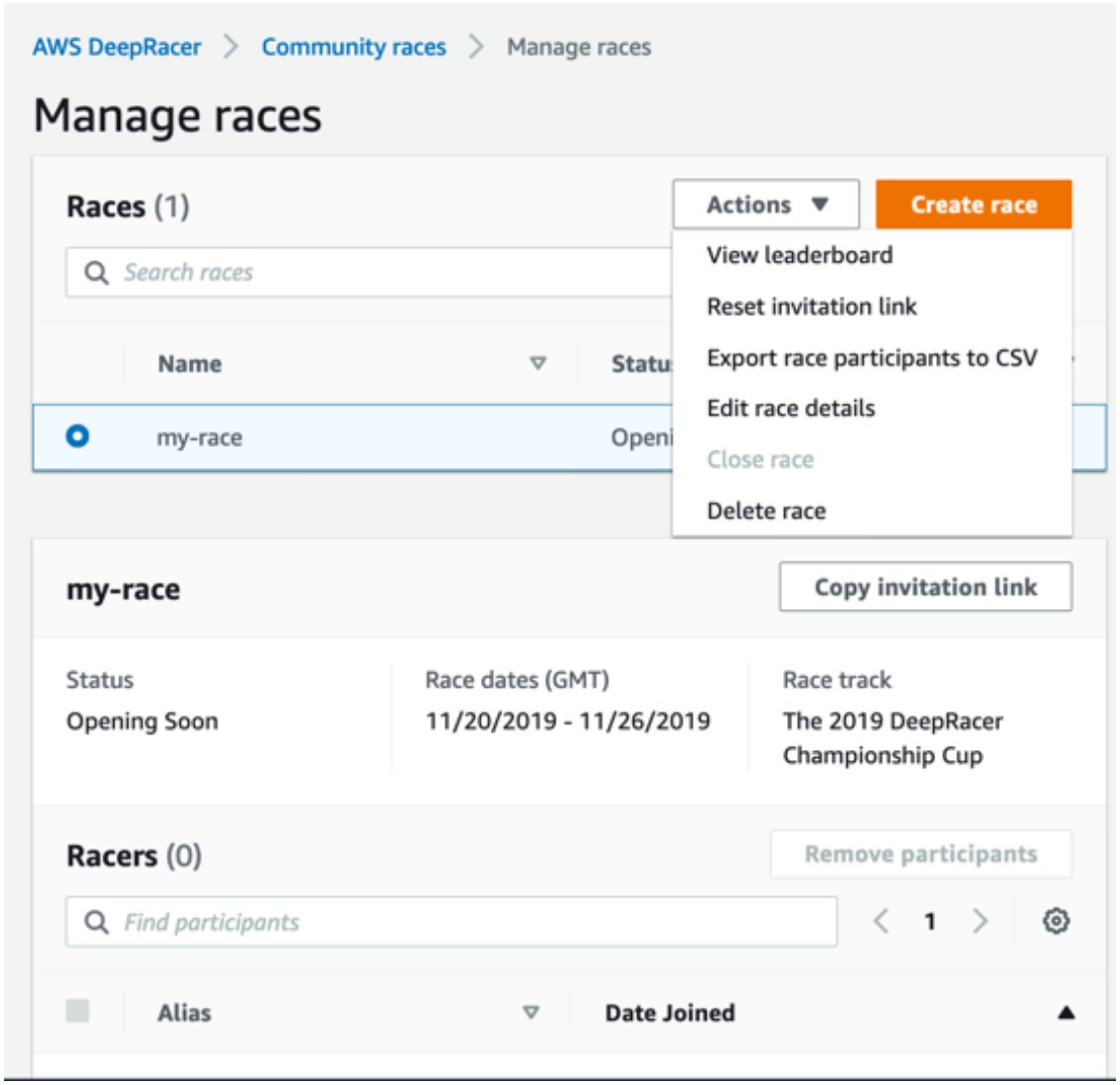
To manage an AWS DeepRacer community race using the AWS DeepRacer console

1. Sign in to the AWS DeepRacer console.
2. Choose **Community races** from the main navigation pane.
3. On the **Manage races** page, choose an event of from the **Races** list.



The next card displays the selected event's details, including the list of joined participants.

4. To edit the race details including reset the start and end dates, choose **Edit race details** from the **Actions** menu.



Follow the on-screen instructions to finish editing.

5. To view the event's leaderboard, choose **View leaderboard** from the **Actions** menu.
6. To reset the event's invitation link, choose **Reset invitation link** from the **Actions** menu.

   Optionally, you can copy the link to share it with invited participants at this step. Resetting the invitation link prevents anyone from choosing the previous link to access the race. All users who have already clicked the link and submitted a model remain in the race.
7. To end the open race, choose **Close race** from the **Actions** menu. This action ends the race immediately ahead of the specified closing date.
8. To delete the event, choose **Delete race** from the **Actions** menu. This action permanently removes this race and details from all participants' community races.
9. To remove a participant, choose one or more race participants, choose **Remove participants**, and then confirm to remove the participant.

   Removing a participant from an event revokes the user's permissions to access the racing event.

Security for AWS DeepRacer

To use AWS DeepRacer to train and evaluate reinforcement learning, your AWS account must have appropriate security permissions to access dependent AWS resources, including VPC to run training jobs and an Amazon S3 bucket to store trained model artifacts, etc.

The AWS DeepRacer console provides a 1-click solution for you to have the required security settings set up for the dependent services. This section documents the AWS services AWS DeepRacer depends as well as the the IAM roles and policy defining the required permissions to access the dependent services.

Topics

- AWS DeepRacer-Dependent AWS Services (p. 130)
- Required IAM Roles for AWS DeepRacer to Call Dependent AWS Services (p. 132)

AWS DeepRacer-Dependent AWS Services

AWS DeepRacer uses the following AWS services to manage required resources:

Amazon Simple Storage Service

To store trained model artifacts in an Amazon S3 bucket.

AWS Lambda

To create and run the reward functions.

AWS CloudFormation

To create training jobs for AWS DeepRacer models.

Amazon SageMaker

To train the AWS DeepRacer models.

AWS RoboMaker

To simulate an environment for both training and evaluation.

The dependent AWS Lambda, AWS CloudFormation, Amazon SageMaker, and AWS RoboMaker in turn use other AWS services including Amazon CloudWatch and Amazon CloudWatch Logs.

The following table shows AWS services used by AWS DeepRacer, directly or indirectly.

AWS Services that AWS DeepRacer uses directly or indirectly

AWS service principal	Comments
`application-autoscaling`	• Indirectly called by Amazon SageMaker to automatically scale its operations.
`cloudformation`	• Directly called by AWS DeepRacer to create training jobs for reinforcement learning models.
`cloudwatch`	• Directly called by AWS DeepRacer to log its operations. • Indirectly called by AWS RoboMaker to log its operations.

AWS service principal	Comments
	• Indirectly called by Amazon SageMaker to log its operations.
ec2	• Indirectly called by AWS CloudFormation and Amazon SageMaker to create and run training jobs.
ecr	• Indirectly called by AWS RoboMaker to work with Amazon Elastic Container Registry.
kinesisvideo	• Directly called by AWS DeepRacer to view cached training streams. • Indirectly called by AWS RoboMaker to cache training streams.
lambda	• Directly called by AWS DeepRacer to create and run the reward functions.
logs	• Directly called by AWS DeepRacer to log its operations. • Indirectly called by AWS Lambda to log its operations. • Indirectly called by AWS RoboMaker to log its operations.
robomaker	• Directly called by AWS DeepRacer to render a virtual reinforcement learning environment in a simulation.
s3	• Indirectly called by AWS RoboMaker to list a bucket beginning with `'deepracer'` and to read objects in the bucket, or write objects to the bucket. • Indirectly called by Amazon SageMaker to perform Amazon SageMaker-specific storage operations. • Directly called by AWS DeepRacer to create, list, and delete buckets that have names starting with "`deepracer`." Also called to download objects from the buckets, upload objects to the buckets, or delete objects from the buckets.
sagemaker	• Directly called by AWS DeepRacer to train reinforcement learning models.

To use AWS DeepRacer to call these services, you must have appropriate IAM roles with required policies attached to them. Learn the details about these policies and roles in Required IAM Roles for AWS DeepRacer to Call Dependent AWS Services (p. 132).

Required IAM Roles for AWS DeepRacer to Call Dependent AWS Services

Before you create a model, use the AWS DeepRacer console to set up resources for your account. As you do this, the AWS DeepRacer console creates the following IAM roles:

AWSDeepRacerServiceRole

Allows AWS DeepRacer to create required resources and call AWS services on your behalf.

AWSDeepRacerSageMakerAccessRole

Allows Amazon SageMaker to create required resources and call AWS services on your behalf.

AWSDeepRacerRoboMakerAccessRole

Allows AWS RoboMaker to create required resources and call AWS services on your behalf.

AWSDeepRacerLambdaAccessRole

Allows AWS Lambda functions to call AWS services on your behalf.

AWSDeepRacerCloudFormationAccessRole

Allows AWS CloudFormation to create and manage AWS stacks and resources on your behalf.

Follow the links to view detailed access permissions in the AWS IAM console.

Troubleshoot Common AWS DeepRacer Issues

Here you'll find troubleshooting tips for frequently asked questions as well as late-coming bug fixes.

Topics

- Why Can't I Connect to the Device Console with USB Connection between My Computer and Vehicle? (p. 133)
- How to Switch AWS DeepRacer Compute Module Power Source from Battery to a Power Outlet (p. 136)
- How to Connect Your AWS DeepRacer to Your Wi-Fi Network (p. 138)
- How to Charge the AWS DeepRacer Drive Module Battery (p. 138)
- How to Charge the AWS DeepRacer Compute Module Battery (p. 139)
- My Battery Is Charged but My AWS DeepRacer Vehicle Doesn't Move (p. 140)
- How to Maintain Your Vehicle's Wi-Fi Connection (p. 142)
- How to Get the MAC Address of Your AWS DeepRacer Device (p. 143)
- How to Recover Your AWS DeepRacer Device Console Default Password (p. 144)
- How to Manually Update Your AWS DeepRacer Device (p. 146)
- How to Diagnose and Resolve Common AWS DeepRacer Operational Issues (p. 147)
- Restore Your AWS DeepRacer Vehicle to Factory Settings (p. 149)

Why Can't I Connect to the Device Console with USB Connection between My Computer and Vehicle?

When setting up your vehicle for the first time, you might find it unable to open the device console (also known as the device web server, `https://deepracer.aws`, hosted on the vehicle) after connecting your AWS DeepRacer vehicle to your computer with a micro-USB/USB cable (USB is also referred to as USB-A).

Multiple causes may be behind this. Typically, you can resolve the issue with the following simple remedy.

To activate your device's USB-over-Ethernet network

1. Turn off Wi-Fi on your computer and unplug any Ethernet cable connected to it.
2. Press the **RESET** button on the vehicle to reboot the device.
3. Open the device console by navigating to `https://deepracer.aws` from a web browser on your computer.

If the previous procedure doesn't work, you can check your computer's network preferences to verify that they're properly configured to let the computer connect to the device's network, whose network name is `Deepracer`. To do this, follow the steps in the following procedure.

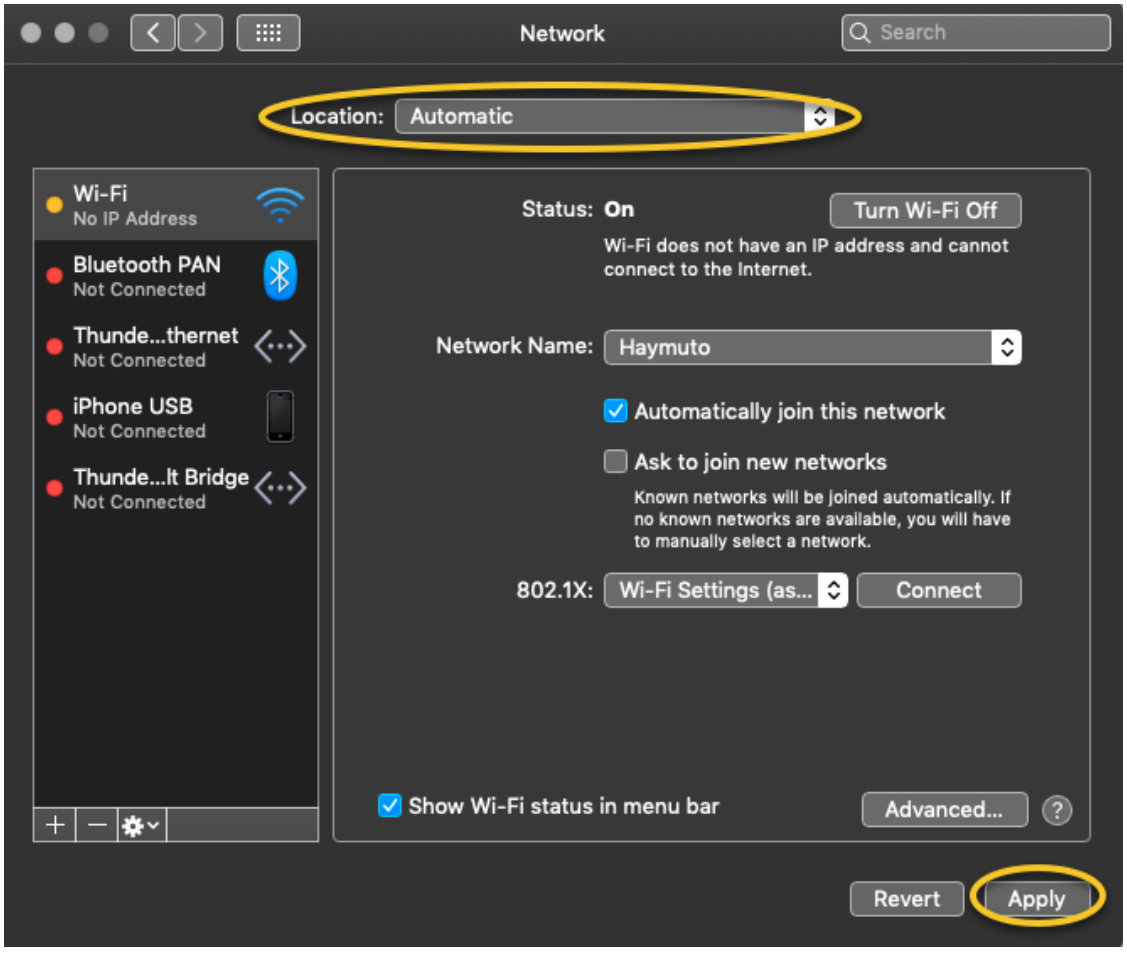
> **Note**
> The instructions below assume you're working with a MacOS computer. For other computer systems, consult with the network preferences documentation for the respective operating system and use the below instructions as a general guide.

To activate the device's USB-over-ethernet network on your MacOS computer

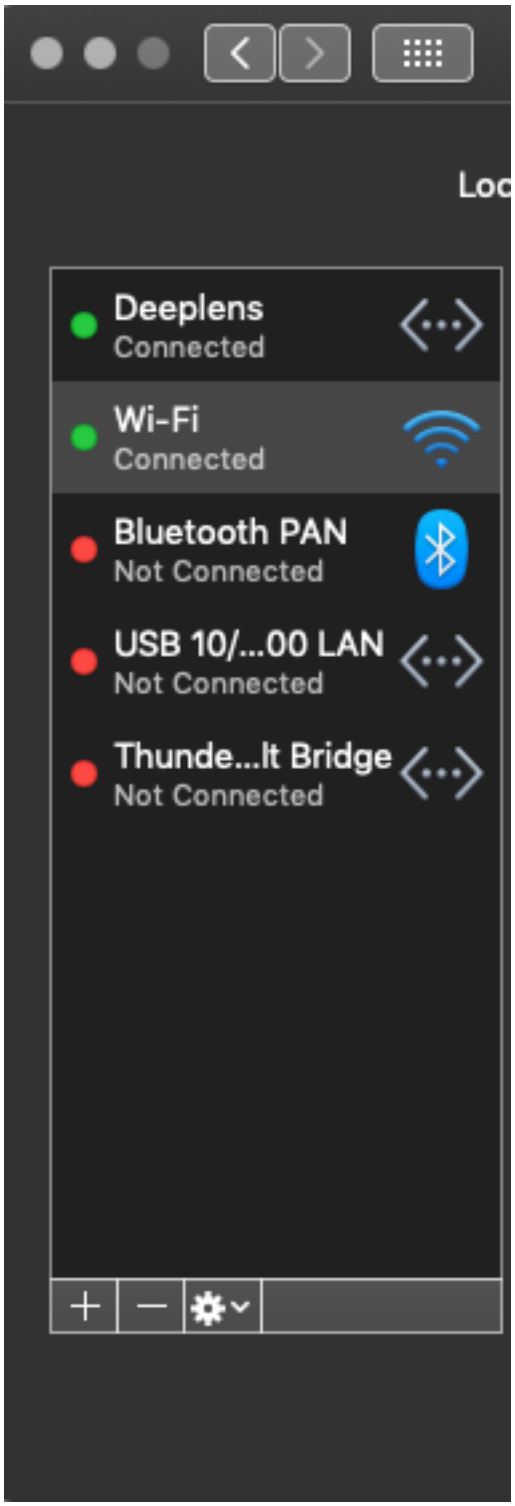
1. Choose the network icon (on the top-right corner of the display) to open **Network preferences**.

   Alternatively, choose **Command+space**, type **Network**, and then choose **Network System Preferences**.

2. Check if **Deepracer** is listed as **Connected**. If **DeepRacer** is listed but not connected, make sure the micro-USB/USB cable is tightly plugged in between the vehicle and your computer.
3. If the **Deepracer** network is not listed there or is listed but not connected when the USB cable is plugged in, choose **Automatic** from the **Location** preference and then choose **Apply**.



4. Verify that the AWS DeepRacer network is up and running as **Connected**.



5. When your computer is connected to the **Deepracer** network, refresh the `https://deepracer.aws` page on the browser, and continue with the rest of **Get Started Guide** instructions of **Connect to Wi-Fi**.

6. If the **Deepracer** network is not connected, disconnect your computer from the AWS DeepRacer vehicle and then reconnect it. When the **Deepracer** network becomes **Connected**, continue with the **Get Started Guide**d instructions.
7. If the **Deepracer** network on the device is still not connected, reboot your computer and AWS DeepRacer vehicle and repeat from **Step 1** of this procedure, if necessary.

If the above remedy still doesn't resolve the issue, the device certificate might have been corrupted. Follow the steps below to generate a new certificate for your AWS DeepRacer vehicle to repair the corrupted file.

To generate a new certificate on the AWS DeepRacer vehicle

1. Terminate the USB connection between your computer and your AWS DeepRacer vehicle by unplugging the micro-USB/USB cable.
2. Connect your AWS DeepRacer vehicle to a monitor (with a HDMI-to-HDMI cable) and to USB keyboard and mouse.
3. Log in to the AWS DeepRacer operating system. If this is the first login to the device operating system, use `deepracer` for the password, when asked for, and then proceed to change the password, as required, and use the updated password for subsequent logins.
4. Open a terminal window and type the following Shell command. You can choose the **Terminal** shortcut from **Applications -> System Tools** on the desktop to open a terminal window. Or you can use the file browser, navigate to the `/usr/bin` folder, and choose **gnome-terminal** to open it.

   ```
   sudo /opt/aws/deepracer/nginx/nginx_install_certs.sh && sudo reboot
   ```

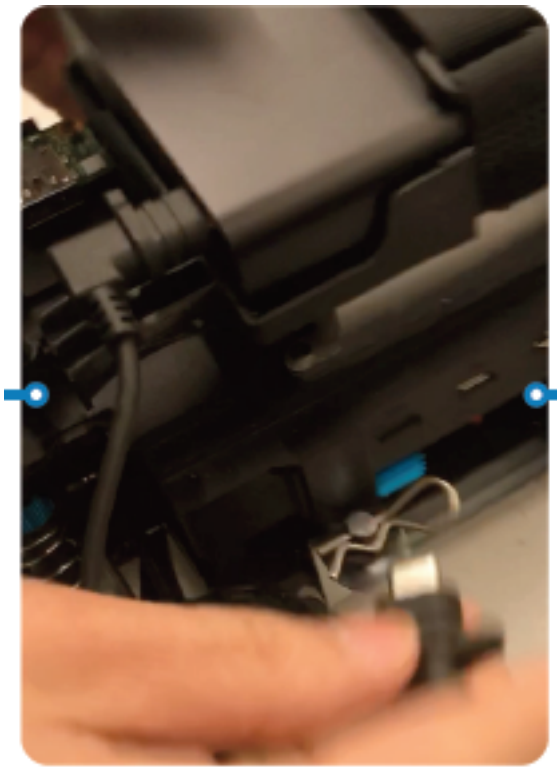
   Enter the password, which you used or updated in the previous step, when prompted.

   The above command installs a new certificate and reboots the device. It also reverts the device console's password to the default value printed at the bottom of the AWS DeepRacer vehicle.
5. Disconnect the monitor, keyboard and mouse from the vehicle and reconnect it to your computer with the micro-USB/USB cable.
6. Follow the second procedure in this topic (p. 134) to verify your computer is indeed connected to the device network before opening the device console (`https://deepracer.aws`) again and, then, continue with the **Connect to Wi-Fi** instructions in **Get Started Guide**.

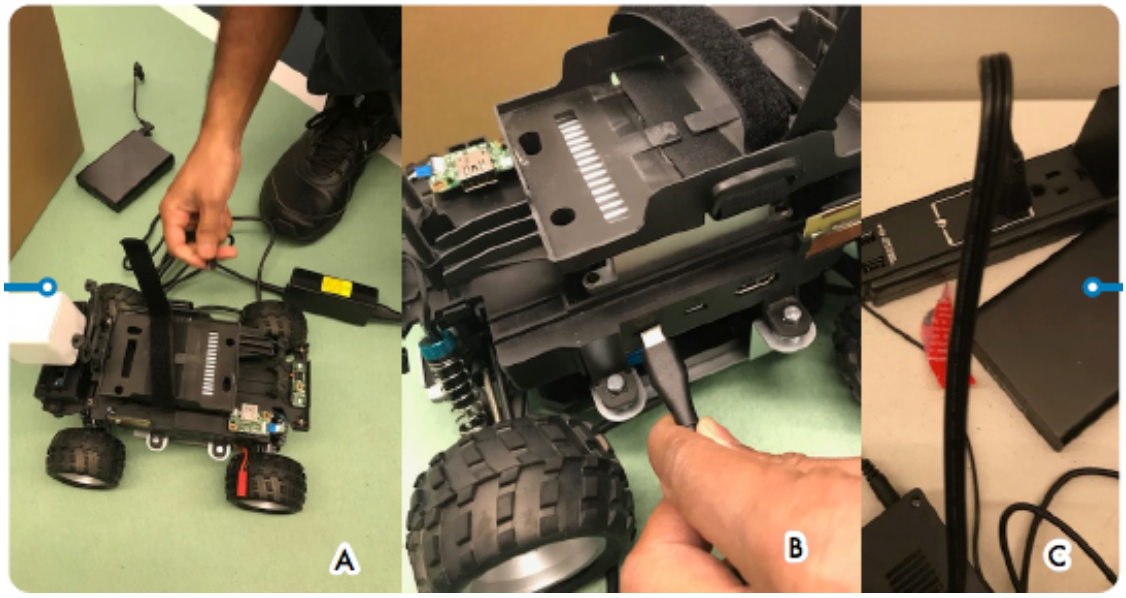
How to Switch AWS DeepRacer Compute Module Power Source from Battery to a Power Outlet

If the compute module battery level is low when you set up your AWS DeepRacer for the first time, follow the steps below to switch the compute power supply from the battery to a power outlet:

1. Unplug the USB-C cable from the vehicle's compute power port.



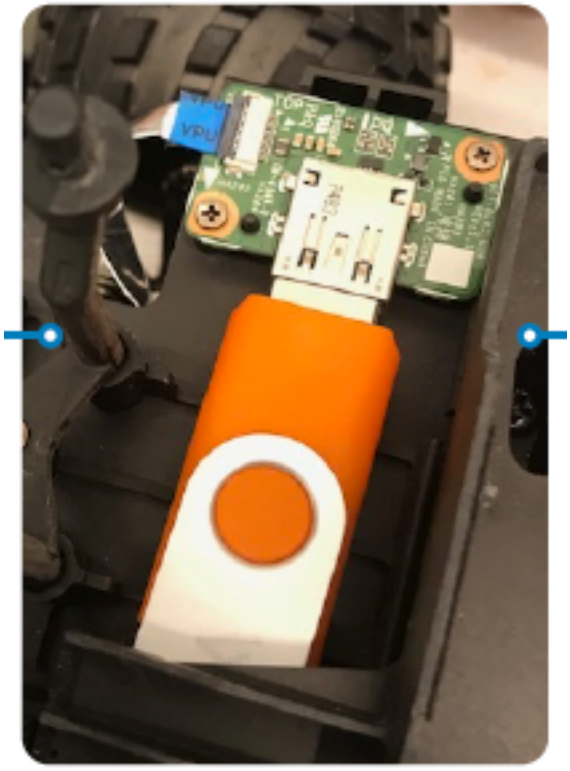
2. Attach the AC power cord and the USB-C cable to the computer module power adapter (A). Plug the power cord to a power outlet (C) and plug the USB-C cable the vehicle's computer module power port (B).



How to Connect Your AWS DeepRacer to Your Wi-Fi Network

To use your AWS DeepRacer, you must connect the vehicle to your home or office Wi-Fi network. To connect the vehicle to your Wi-Fi network, follow the steps below:

1. Have a USB flash drive on hand.
2. Plug in the USB flash drive to your computer.
3. Open a web browser on your computer and navigate to https://d1.awsstatic.com/deepracer/wifi-creds.txt to download the Wi-Fi configuration file and copy it to the USB drive.
4. Open the Wi-Fi configuration file in a text editor and type the name (SSID) and password of your Wi-Fi network in the corresponding fields.
5. Eject the USB drive from your computer and then plug it into the USB port on the back of the vehicle.



6. Watch the Wi-Fi LED on the vehicle to blink and then to turn blue. The vehicle is now connected to the Wi-Fi network. Unplug the USB drive and skip the next step.
7. If the Wi-Fi LED turns red after blinking, unplug the USB drive from the vehicle. Plug the USB drive back to your computer, verify that the configuration contains the correct network name and password, correct any mistakes or typos, save the file. Repeat Step 5.

How to Charge the AWS DeepRacer Drive Module Battery

Follow the steps below to charge your AWS DeepRacer drive module battery:

1. Optionally remove from the vehicle the drive module battery.
2. Attach the battery charger to the battery, as depicted as follows:

3. Plug the power cord of battery charger into a power outlet.

How to Charge the AWS DeepRacer Compute Module Battery

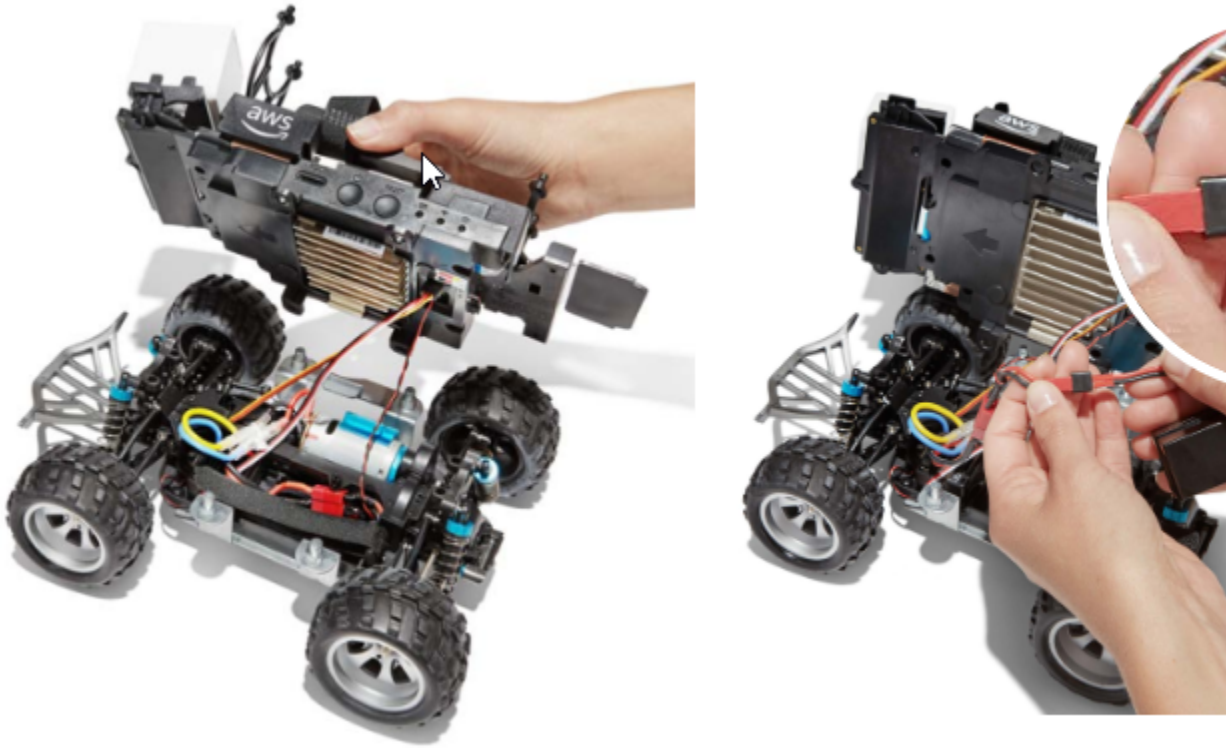
Follow the steps below to charge your AWS DeepRacer compute module battery:

1. Optionally remove the compute module battery from the vehicle.
2. Attach the compute power charger to the compute module battery.
3. Plug the power cord of the compute battery charger into a power outlet.

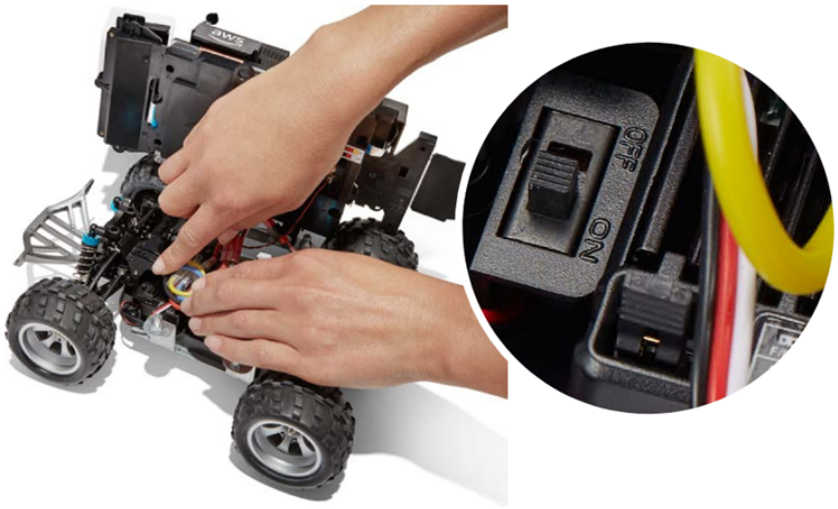
My Battery Is Charged but My AWS DeepRacer Vehicle Doesn't Move

Follow these steps if your AWS DeepRacer console is set up, your compute battery is charged, and your Wi-Fi is connected, but your vehicle still doesn't move:

1. Lift the compute module, being careful not to loosen the wires connecting it to the drive train. Make sure the vehicle battery underneath is correctly connected, red input to red adapter.



2. Turn on the vehicle drive train by pushing the switch to the "on" position. Listen for the indicator sound (two short beeps) to confirm that the car has charge. If the car powers on successfully, skip to step 4.



3. If you do not hear two beeps when you switch on your car battery, ensure that the battery is fully charged. Plug the vehicle battery's white connector cable into its charge adapter, which can be differentiated from the compute module's adapter by its red and green LED indicator lights. Connect the adapter to its charge cable and plug it into a power outlet. When both red and green lights on the vehicle battery charge adapter are lit, it indicates that the battery still needs charging.

Red light + green light = not fully charged

When only the green light is illuminated, your battery is fully charged and ready to use. Disconnect the car battery's white connector from the charge adapter, and reconnect its red connector to the car. If you removed the battery to charge it (optional) make sure to once again secure it to the drive train with the Velcro strap. Turn on the vehicle drive train by pushing its switch to the "on" position. If you still don't hear two beeps, the battery is likely damaged. Please reach out to AWSDeepRacer-Help@amazon.com.

4. Connect your car to Wi-Fi (p. 91) and open the AWS DeepRacer console in your browser. Manually drive your vehicle with the touch joystick to confirm that it can move.

REMINDER: To get the most millage out of your car battery, make sure to switch off the vehicle drive train or disconnect its battery when you are not using your AWS DeepRacer.

If your car still does not move, contact AWSDeepRacer-Help@amazon.com.

How to Maintain Your Vehicle's Wi-Fi Connection

The following troubleshooting guide provides you tips for maintaining your vehicle's connection.

How to Troubleshoot Wi-Fi Connection If Your Vehicle's Wi-Fi LED Indicator Flashes Blue, Then Turns Red for Two Seconds, and Finally Off

Check the following to verify you have the valid Wi-Fi connection settings.

- Verify that the USB drive has only one disk partition with only one *wifi-creds.txt* file on it. If multiple *wifi-creds.txt* files are found, all of them will be processed in the order they were found, which may lead to unpredictable behavior.
- Verify the Wi-Fi network's SSID and password are correctly specified in *wifi-creds.txt* file. An example of this file is shown as follows:

```
##################################################################################
#                                  AWS DeepRacer                                 #
# File name: wifi-creds.txt                                                      #
#                                                                                #
# ...                                                                            #
##################################################################################
```

```
# Provide your SSID and password below
ssid: ' MyHomeWi-Fi'
password: myWiFiPassword
```

- Verify both the field names of `ssid` and `password` in the *wifi-creds.txt* file are in lower case.
- Verify that each of the field name and value is separated by one colon (:). For example. `ssid : ' MyHomeWi-Fi'`
- Verify that the field value containing a space is enclosed by a pair of single quotes. On Mac, TextEdit or some other text editor shows single quotes as of the '...' form, but not of ‘...’. If the field value does not contain spaces, the value can be without single quotes.

What Does It Mean When the Vehicle's Wi-Fi or Power LED Indicator Flashes Blue?

If the USB drive contains *wifi-creds.txt* file, the Wi-Fi LED indicator flashes blue while the vehicle is attempting to connect to the Wi-Fi network specified in the file.

If the USB drive has the `models` directory, the Power LED flashes blue while the vehicle is attempting to load the model files inside the directory.

If the USB drive has both the *wifi-creds.txt* file and the `models` directory, the vehicle will process the two sequentially, starting with an attempt to connect to Wi-Fi and then loading models.

The Wi-Fi LED might also turn red for two seconds if the Wi-Fi connection attempt fails.

How Can I Connect to the Vehicle's Device Console Using its Hostname?

When connecting to the vehicle's device console using its hostname, make sure you type: `https://`*`hostname`*`.local` in the browser, where *`hostname`* value (of the `AMSS-`*`1234`* format) is printed on the bottom of the AWS DeepRacer vehicle.)

How to Connect to Vehicle's Device Console Using Its IP Address

To connect to the device console using IP address as shown in the *device-status.txt* file (found on the USB drive), make sure the following conditions are met.

- Check your laptop or mobile devices are in the same network as the AWS DeepRacer vehicle.
- Check if you have connected to any VPN, if so, disconnect first.
- Try a different Wi-Fi network. For example, turn on personal hotspot on your phone.

How to Get the MAC Address of Your AWS DeepRacer Device

Follow the instructions below to get the MAC address of your AWS DeepRacer device:

1. Make sure that your AWS DeepRacer device is only connected to a Wi-Fi network.

2. Connect your AWS DeepRacer device to a monitor. You'll need a HDMI-to-HDMI, HDMI-to-DVI or similar cable and insert one end of the cable into the HDMI port on the vehicle's chassis and plug the other end into a supported display port on the monitor.
3. Connect a USB keyboard to your AWS DeepRacer using the USB port on the device's compute module, after the compute module is booted.
4. Type `deepracer` in the **Username** input field.
5. Type the device SSH password in the **Password** input field.

   If this is your first time to log in to the device, type `deepracer` in the **Password** input field. Reset the password, as required, before moving to the next step. You'll use the new password for future logins. For security reasons, use a complex or strong password phrase for the new password.
6. After logged in, open a Terminal window.

   You can use the Search button for the Terminal application.
7. Type the following Ubuntu shell command in the Terminal window:

   ```
   ifconfig | grep HWaddr
   ```

   The command produces an output similar to the following:

   ```
   mlan0     Link encap:Ethernet   HWaddr   01:2a:34:b5:c6:de
   ```

   The hexadecimal numbers are the device's MAC address.

How to Recover Your AWS DeepRacer Device Console Default Password

Recovering your AWS DeepRacer device console default password involves retrieving or resetting the default password. The default password is printed at the bottom of the device, as shown following.

Follow the instructions in the following procedure to recover the password for your AWS DeepRacer device web server using an Ubuntu terminal window.

1. Connect your AWS DeepRacer device to a monitor. You'll need a HDMI-to-HDMI, HDMI-to-DVI or similar cable and insert one end of the cable into the HDMI port on the vehicle's chassis and plug the other end into a supported display port on the monitor.
2. Connect a USB keyboard to your AWS DeepRacer using the USB port on the device's compute module, after the compute module is booted.
3. in the **Username**, enter `deepracer`.
4. In **Password**, enter the device SSH password.

   If this is your first time to log in to the device, enter `deepracer` in **Password**. Reset the password, as required, before moving to the next step. You'll use the new password for future logins. For security reasons, use a complex or strong password phrase for the new password.
5. After you're logged in, open a terminal window.

   You can use Search button to find the terminal window application.
6. To get the default device console password, type the following command in the terminal window:

```
$cat /sys/class/dmi/id/chassis_asset_tag
```

The command outputs the default password as its result.

7. To reset the device console password to the default, run the following Python script in the terminal window:

```
sudo python /opt/aws/deepracer/nginx/reset_default_password.py
```

How to Manually Update Your AWS DeepRacer Device

Recent changes in the AWS DeepRacer service has made certain legacy devices, such as those distributed at AWS re:Invent 2018, unable to update automatically. Follow the steps below to manually update such a device.

To manually update an AWS DeepRacer device

1. Download to your computer and unzip this manually update a AWS DeepRacer device script.

   The default name of the uncompressed file for this script is `deepracer-device-manual-update.sh`. In this topic, we'll assume you use this default script file name.
2. Copy the downloaded and uncompressed the script file (`deepracer-device-manual-update.sh`) from your computer to a USB drive.
3. Connect the device to a monitor using a HDMI-HDMI cable, to a USB keyboard, and to a USB mouse.
4. Power on the device and sign into the OS after the device is booted up.

   You'll need to set the new OS password, if this is your first sign-in to the device.
5. Plug in the USB drive into the device and copy the script file to a folder (for example, ~/Desktop) on the device.
6. From a terminal on the device, type the following command to go to the script file's folder and to add execution permission to the script file:

```
cd ~/Desktop
chmod +x deepracer-device-manual-update.sh
```

7. Type the following shell command to run the script:

```
sudo ./deepracer-device-manual-update.sh
```

8. When done with updating the device, open a web browser on your computer or a mobile device and navigate to the device IP address, e.g., 192.168.1.11 in a home network or 10.56.101.13 in an office network.

   Make sure that your device is connected to your Wi-Fi network and use a browser in the same network without tunneling through a VPN.
9. On the device console, type the password for the device console to sign in. Wait for the update screen to show up. When prompted for further updates, follow the instructions therein.

How to Diagnose and Resolve Common AWS DeepRacer Operational Issues

As you explore reinforcement learning with your AWS DeepRacer vehicle, the device may become non functional. The following troubleshooting topics help you diagnose the problems and resolve the issues.

Topics

- Why Doesn't the Video Player on the Device Console Show the Video Stream from My Vehicle's Camera? (p. 147)
- Why Doesn't My AWS DeepRacer Vehicle Move? (p. 147)
- Why Don't I See the Latest Device Update? How Do I Get the Latest Update? (p. 148)
- Why Isn't My AWS DeepRacer Vehicle Connected to My Wi-Fi Network? (p. 148)
- Why Does the AWS DeepRacer Device Console Page Take a Long Time to Load? (p. 148)
- Why Does a Model Fail to Perform Well When Deployed to an AWS DeepRacer Vehicle? (p. 148)

Why Doesn't the Video Player on the Device Console Show the Video Stream from My Vehicle's Camera?

After logging into the AWS DeepRacer device console, you don't see any live video streamed from the camera mounted on the AWS DeepRacer vehicle in the video player in **Device Controls**. The following could cause this issue:

- The camera might have a loose connection to the USB port. Unplug the camera module from the vehicle, replug it into the USB port, power off the device, and then power on the device to restart it.
- The camera might be defective. Use a known working camera from another AWS DeepRacer vehicle, if available, to test whether this is the cause.

Why Doesn't My AWS DeepRacer Vehicle Move?

You powered on your AWS DeepRacer vehicle, but you can't make it to move. The following could cause this issue:

- The vehicle's power bank is not turned on or the power bank is not connected to the vehicle. Make sure to connect the provided USB-C-to-USB C cable between the USB-C port on the power bank and the USB-C port on the vehicle chassis. Verify that the LED indicators light up, which indicates the charge levels of the power bank. If not, push the power button on the power bank, and then push the power button on the vehicle's chassis to boot up the device. The device is booted up when its tail lights light up.
- If the power bank is on and the vehicle is booted up, but the vehicle does not move in either manual or autonomous driving mode, check if the vehicle's battery under the vehicle chassis is charged and turned on. If not, recharge the vehicle battery and then turn it on after the battery is fully charged.
- The vehicle battery cable connectors are not fully plugged into the device driving module power cable connector. Make sure the cable connectors are tightly coupled.
- The battery cables are defective. Test this battery on another working vehicle, if possible, to test whether this is the cause.
- The power switch of the vehicle battery is not turned on. Turn on the power switch and make sure you hear two beeps followed by a long beep.

Why Don't I See the Latest Device Update? How Do I Get the Latest Update?

Why is my AWS DeepRacer vehicle's software outdated?

- No automatic update is performed on the device lately. You may need to perform a manual update (p. 146).
- The vehicle is not connected to the Internet. Make sure the vehicle is connected to a Wi-Fi or Ethernet network with internet access.

Why Isn't My AWS DeepRacer Vehicle Connected to My Wi-Fi Network?

When I check the network status on the vehicle's OS, I don't see the AWS DeepRacer vehicle connected to any Wi-Fi network. This could happen because of the following issues:

- No Wi-Fi has been configured for the AWS DeepRacer vehicle. Follow this setup instruction (p. 91) to set up the Wi-Fi network for your vehicle.
- The vehicle is out of the active network signal range. Make sure to operate the vehicle within the chosen Wi-Fi network range.
- The vehicle's pre-configured Wi-Fi network doesn't match the available Wi-Fi network. Follow the setup instruction (p. 91) to set up the Wi-Fi network that does not require active CAPTCHA.

Why Does the AWS DeepRacer Device Console Page Take a Long Time to Load?

When I tried to open the device console of my AWS DeepRacer vehicle, the device console page appears to take a long time to load.

- Your vehicle is down or off. Make sure the vehicle is powered on when the tail lights are on.
- The IP address of you vehicle has been changed, most likely by your network's DHCP server. To find out the vehicle's new IP address, follow these setup instructions (p. 91) to sign in to the device console with the USB-USB cable connection between your computer and the vehicle. View the new IP address in **Settings**. Alternatively, you can examine the list of devices attached to your network to discover the new IP address. If you're not a network administrator, ask the administrator to investigate this for you.

Why Does a Model Fail to Perform Well When Deployed to an AWS DeepRacer Vehicle?

After training a model and deploying its artifacts to your AWS DeepRacer vehicle, sometimes the vehicle doesn't perform as expected. What went wrong?

In general, optimizing a trained model for transfer to a physical AWS DeepRacer vehicle is a challenging learning process. It often requires iterations through trial and error. For general guidelines on best practices, see Optimize Training AWS DeepRacer Models for Real Environments (p. 41).

The following are some likely common factors affecting the model performance in your AWS DeepRacer vehicle:

- Your model has not converged in training. Clone the model to continue the training or retrain the model for a longer period of time. Make sure the agent continuously finishes laps in the simulation (that is, 100% process towards the end of the training).
- Your model was over-trained (that is, over-fitted). It fits too well to the training data, but doesn't generalize to unknown situations. Retrain the model with a more flexible or accommodating reward function (p. 31) or increase the granularities of the action space (p. 33). You should also evaluate a trained model on different tracks to see if the model generalizes well.
- Your AWS DeepRacer vehicle may not have been calibrated properly. To test whether this is true, switch to manual driving and see if the vehicle drives as expected. If it doesn't, calibrate the vehicle (p. 94).
- You are running the vehicle autonomously on a track that doesn't meet the requirements. For track requirements, see Build Your Physical Track for AWS DeepRacer (p. 109).
- There are too many objects close to the physical track, making the track significantly different from the simulated environment. Clear the track surroundings to make the physical track as close to the simulated one as possible.
- Reflection from the track surface or a near-by object can create glare to confuse the camera. Adjust lighting and avoid making the track on smooth-surfaced concrete floors or with other shiny materials.

Restore Your AWS DeepRacer Vehicle to Factory Settings

At some point after trying out AWS DeepRacer, you might want to reset the device to its factory settings for a fresh start. The first proceedure explains how to partition a USB flash drive into two parts and make the first partition bootable. It also gives you the factory restore files to download onto the second partition. The second proceedure explains how to use the partitioned flash drive and its contents to reset your AWS DeepRacer to its factory settings.

Topics

- Preparing for the Factory Reset of Your AWS DeepRacer Vehicle (p. 149)
- Restore Your AWS DeepRacer Vehicle to Factory Settings (p. 163)

Preparing for the Factory Reset of Your AWS DeepRacer Vehicle

Resetting your AWS DeepRacer completely wipes the data on the device and returns it to its factory settings. To prepare, there are steps you need to take that require additional hardware. This topic explains what you need to get started and walks you through the process.

Prerequisites

Before you get started make sure you have the following items ready:

- 1 USB flash drive, 32 GB or larger
- A computer to facilitate preparation - Choose one of the following options:
  - **Option 1:** Set up your AWS DeepRacer compute module as a Linux computer with a mouse, keyboard, monitor (connect with HDMI type A cable)
  - **Option 2:** Connect AWS DeepRacer to an Ubuntu, MacOS, or Windows computer
- An AWS DeepRacer vehicle

Important
Confirm which AWS DeepRacer hardware model you are resetting to insure you download the correct image and factory reset file

- **A**: Vehicles ordered from amazon.com have a white "AWS" logo on the chassis - use **0.0.8 BIOS** path
- **B**: Vehicles originating from re:Invent 2018 have no text on the chassis - use **0.0.6 BIOS** path

Preparation

To prepare for the factory reset, perform the following tasks.

- Format the USB drive into the following two partitions.
  - FAT32 of 2 GB
  - NTFS of at least 16 GB
- Make the USB drive bootable to start the factory reset on reboot.
  - Burn the required custom Ubuntu ISO image to the FAT32 partition.
  - Copy the required factory restore files to the NTFS partition of the USB drive.

Depending on the computer you use, specific tasks may differ from one operating system to another. We present step-by-step instructions to prepare your USB drive using Ubuntu (via the computer module of the AWS DeepRacer vehicle), MacOS, and Windows operating systems.

The instructions for using other Linux or Unix computers are similar to the Ubuntu instructions discussed in the following section. You need to replace the `apt-get` commands with the corresponding commands supported by the other Linux or Unix system that you choose to use instead.

Choose one of the following procedures according to the type of computer you use.

Partition a USB Drive and Make it Bootable by Using an Ubuntu Computer

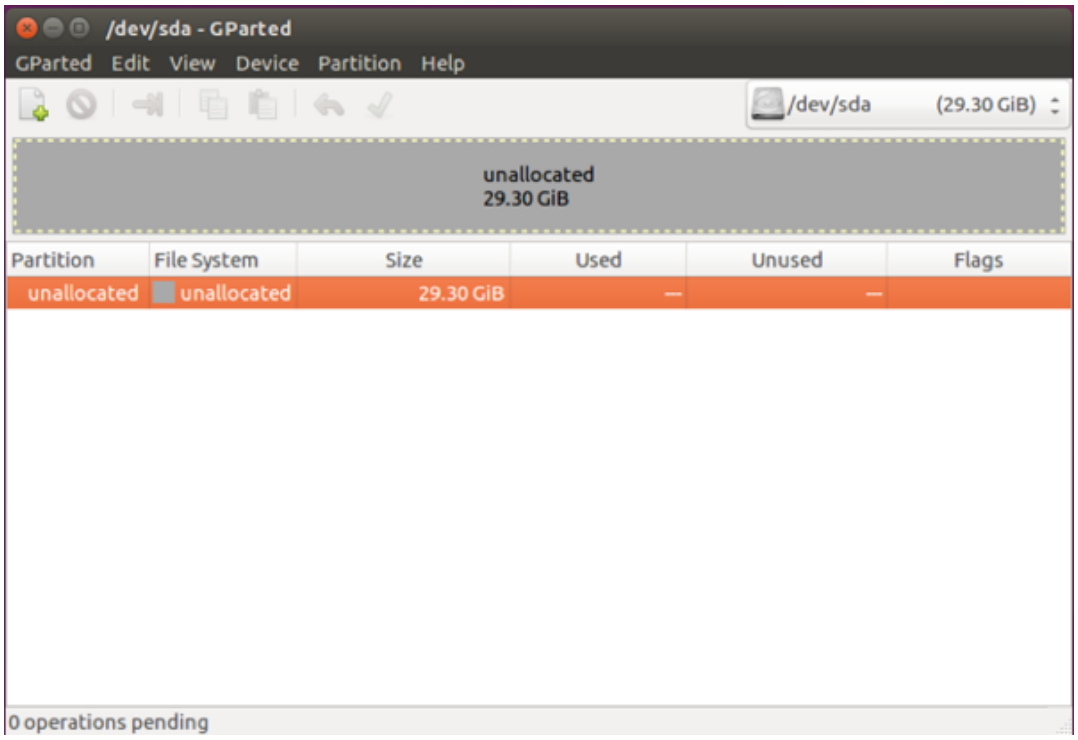
In the following steps, use the AWS DeepRacer vehicle computer module as an Ubuntu computer. The same instructions apply to a Linux computer running Ubuntu. The instructions on other flavors of Linux or Unix operating systems are similar. Just replace the `apt-get *` commands with their corresponding commands supported by the other Linux or Unix system of your choosing.

To partition the USB drive and make it bootable

1. To format the USB drive by running Ubuntu commands on the AWS DeepRacer vehicle or a computer running Ubuntu, do the following.

a. On the AWS DeepRacer vehicle compute module, run the following commands to install and launch GParted.

```
sudo apt-get update; sudo apt-get install gparted
sudo gparted
```



b. On the newly created GParted console, choose **/dev/sda** in the drop-down menu on the top-right corner and then delete all existing partitions.

   If the partitions are locked, open the context (right-click) menu and choose **unmount**.

c. To create the FAT32 partition of 2 GB capacity, choose the file icon on the top-left, set the parameters similar to the following, and then choose **Add**.

   Free space preceding: `1`

   New size:`2048`

   Free space following: `27951`

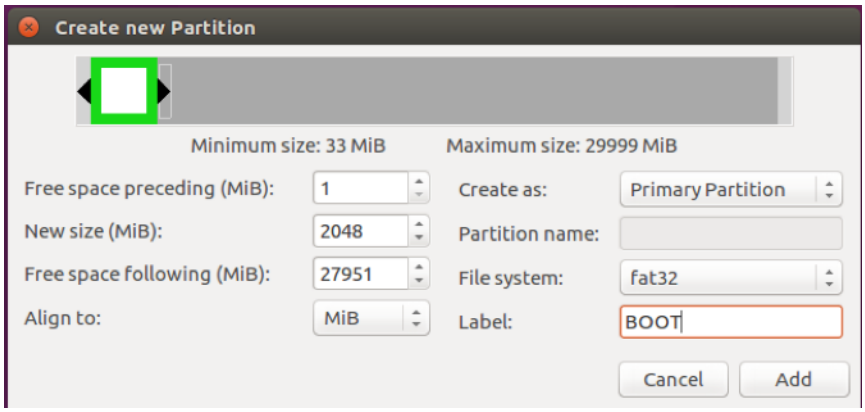
   Align to: `MiB`

   Create as: `Primary Partition`

   Partition name::

   File system: `fat32`

   Label: `BOOT`



d. To create the NTFS partition of at least 16 GB capacity, choose the file icon again, set the parameters similar to the following, and choose **Add**.

   Free space preceding: `0`

   New size:`27951`

   Free space following: `0`

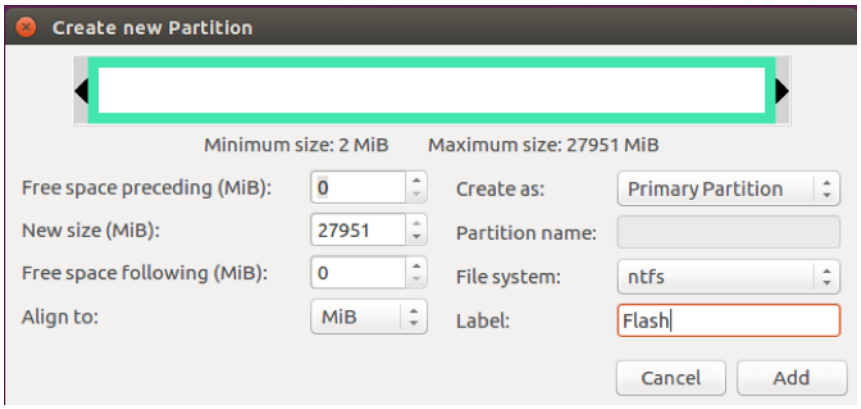
   Align to: `MiB`

   Create as: `Primary Partition`

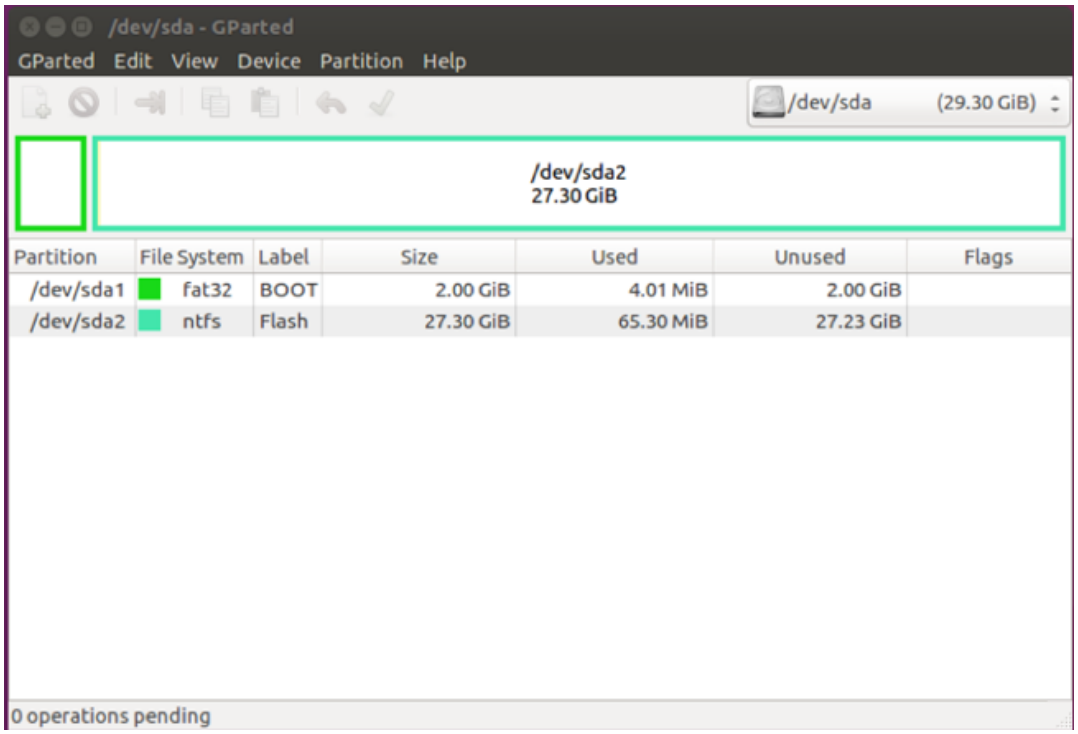
   Partition name:

   File system: `nfts`

   Label: `Flash`



e. After you've created the FAT32 and NTFS partitions, the USB drive partition information appears in the GParted console.



2. To make the USB drive bootable from the FAT32 partition, follow these steps.

   a. Download the customized Ubuntu ISO image appropriate for your AWS DeepRacer model:

      - For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS** logo on the chassis, download this customized Ubuntu ISO image (0.0.8 BIOS) .
      - For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this customized Ubuntu ISO image (0.0.6 BIOS).

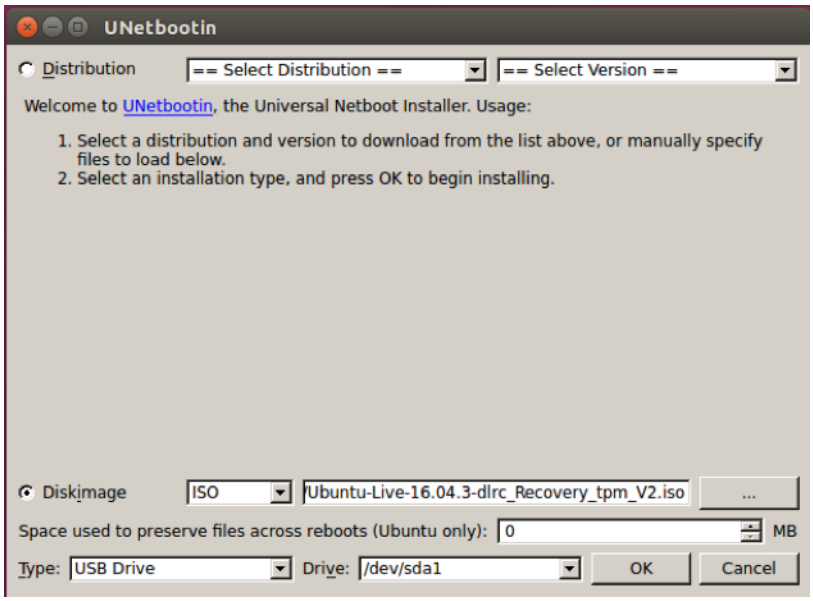
   b. Use UNetbootin on your AWS DeepRacer device, to do the following:

      i. On your AWS DeepRacer compute module, run the following command to install and launch UNetbootin.

```
sudo apt-get update; sudo apt-get install unetbootin
sudo unetbootin
```

      ii. On the UNetbootin window, do the following:

         A. Check the **Disimage** radio button.
         B. For the disk image, choose **ISO** from the drop-down menu.
         C. Open the file picker to choose the downloaded Ubuntu ISO file.
         D. For **Type**, choose **USB Drive**.
         E. For **Drive**, choose `/dev/sda1`.
         F. Choose **OK**.



Note
The customized Ubuntu image may be more recent than what's shown here. If so, use the most recent version of the Ubuntu image.

If you get a **/dev/sda1 not mounted** alert message, choose OK to close the message, unplug the USB drive, replug the drive, and then follow the steps above create the Ubuntu ISO image.

3. To copy the factory restore files to the NTFS partition of the USB drive, follow these steps.

   a. Download the compressed factory restore package appropriate for your AWS DeepRacer model:

      - For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS** logo on the chassis, download this compressed factory restore package (0.0.8 BIOS) .
      - For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this compressed factory restore package (0.0.6 BIOS).

   b. Unzip the downloaded package, and copy the uncompressed files to the second (NTFS) partition of the USB drive.

Partition a USB Drive and Make it Bootable by Using a MacOS Computer

Follow the instructions here to use a MacOS computer to prepare the USB drive for factory reset.

To partition the USB drive and make it bootable by using a MacOS computer

1. To format the USB drive, follow these steps.

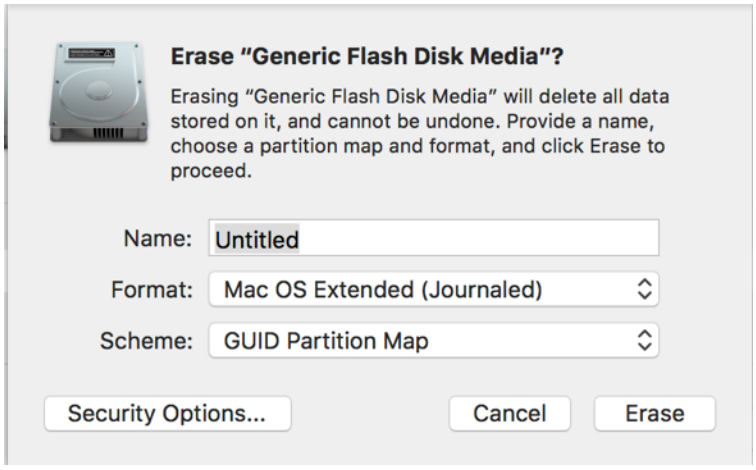
   a. Plug in the USB drive to your MacOS computer.

   b. Press `command + space` to open the search tool bar and then type `Disk Utility`.

      Alternatively, you can choose **Finder->Applications->Utilties->Disk Utility** to open the Disk Utility.

   c. Choose the plugged-in USB drive, e.g., **Generic Flash Disk** which may differ from your USB drive name, on the left pane of Disk Utility. Then choose **Erase** on the top.

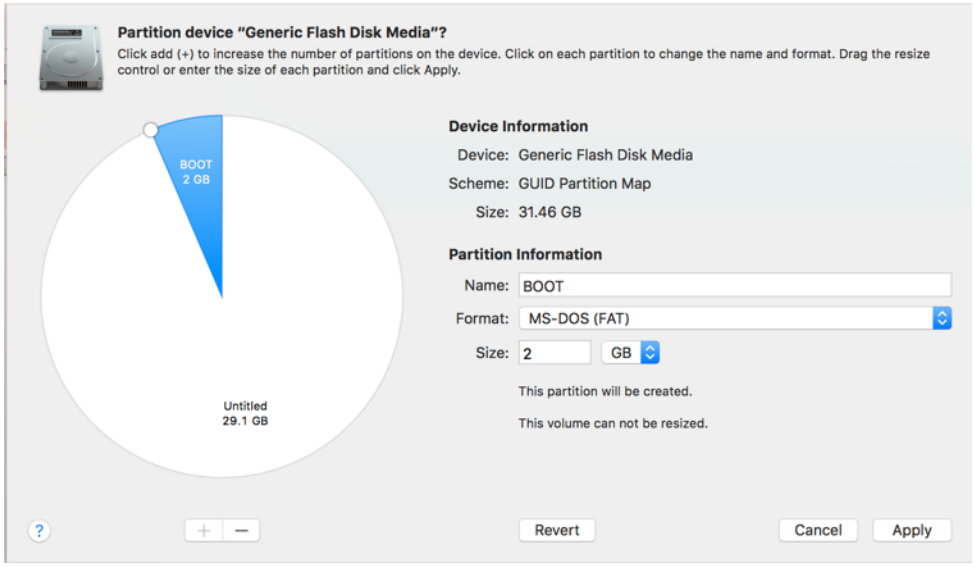
d. On the **Erase "Generic Flash Disk Media"?** page, choose **Mac OS Extended (Journaled)** for Format, choose **GUID Partition Map** for **Scheme**, and then choose **Erase**.



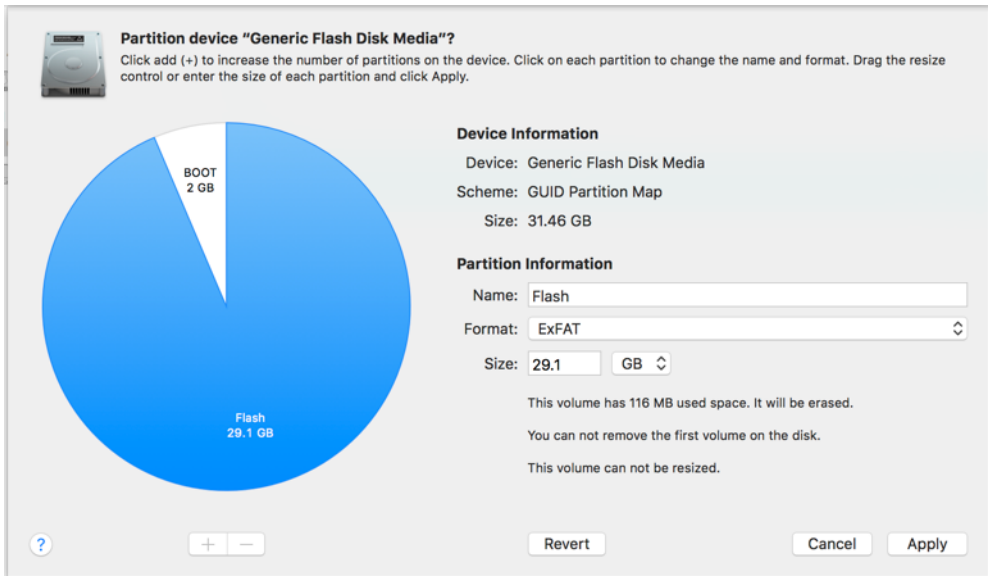
After the USB drive is refreshed, choose **Done** to continue on the **Erasing "Generic Flash Disk Media"?** dialog window.

e. On the Disk Utility console, choose the USB drive on the left navigation pane, choose **Partition** from the menu on the top, and then choose the **+** button on the **Partition device ...** pop-up.

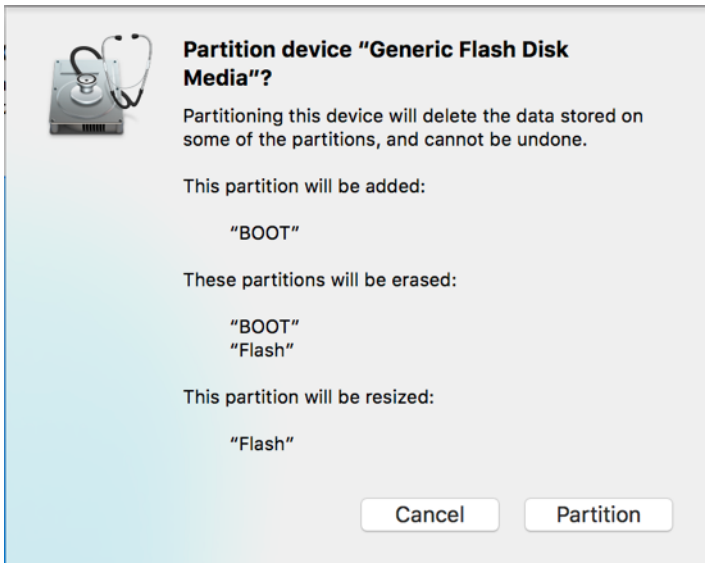
f. To create the FAT32 partition of 2 GB capacity, under **Partition Information** type `Boot` (or another name of your choosing) for **Name**, choose `MS-DOS (FAT)` for **Format**, set **Size** to `2 GB`. Do not choose **Apply** yet.



g. To create the partition for the updated AWS DeepRacer image, choose a point in the other (**Untitled**) partition. Under **Partition Information**, type `Flash` (or another name of your choosing) for **Name**, choose `ExFat` for **Format**, leave the remaining capacity (in GB) of the USB drive in **Size**. Choose **Apply**.



h. On the ensuing pop-up window, choose **Partition** to confirm creation of the specified new partitions.



i. On the Disk Utility console, choose the **BOOT** partition on the left pane, and then choose **Info** from the menu on the top. Make note of the **BSD device node** value. In this tutorial, the value is `dsa1`. You need to supply this path when making the USB drive bootable from the FAT32 partition.

2. To make the USB drive bootable from the FAT32 partition, follow these steps.

   a. Download the customized Ubuntu ISO image appropriate for your AWS DeepRacer model:

      - For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS** logo on the chassis, download this customized Ubuntu ISO image (0.0.8 BIOS) .
      - For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this customized Ubuntu ISO image (0.0.6 BIOS).

   b. Go to https://unetbootin.github.io/ to download the *UNetbootin* software. Then start the UNetbootin console.

   c. On the **UNetbootin** console, do the following:

      i. Check the **Disimage** radio button.

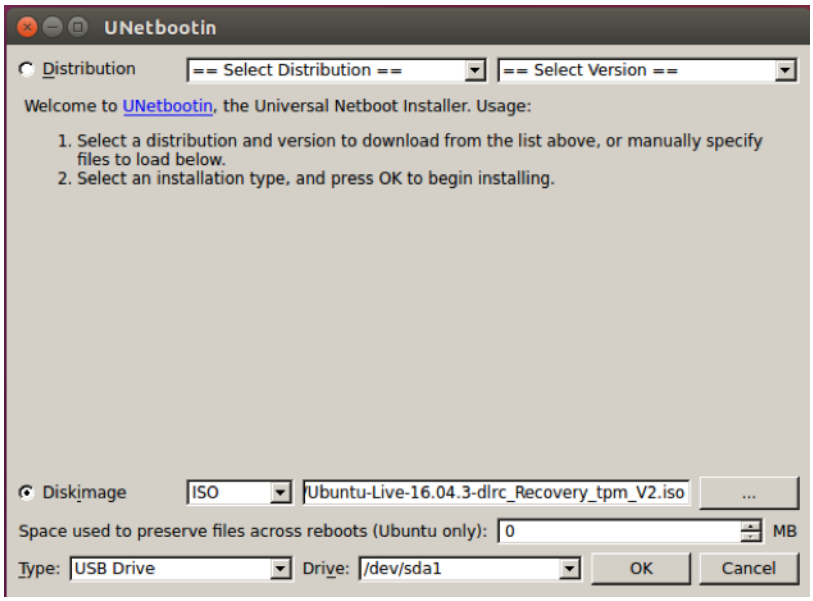
      ii. For the disk image, choose **ISO** from the drop-down menu.

      iii. Open the file picker to choose the downloaded Ubuntu ISO file.

      iv. For **Type**, choose **USB Drive**.

      v. For **Drive**, choose `/dev/sda1`.

      vi. Choose **OK**.



Note
The customized Ubuntu image may be more recent than what's shown here. If so, use the most recent version of the Ubuntu image.

If you get a **/dev/sda1 not mounted** alert message, choose OK to close the message, unplug the USB drive, replug the drive, and then follow the steps above create the Ubuntu ISO image.

3. To copy the factory restore files to the NTFS partition of the USB drive, follow these steps.

   a. Download the compressed factory restore package appropriate for your AWS DeepRacer model:

      - For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS**" logo on the chassis, download this compressed factory restore package (0.0.8 BIOS) .
      - For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this compressed factory restore package (0.0.6 BIOS).

   b. Unzip the downloaded package, and copy the uncompressed files to the second (NTFS) partition of the USB drive.

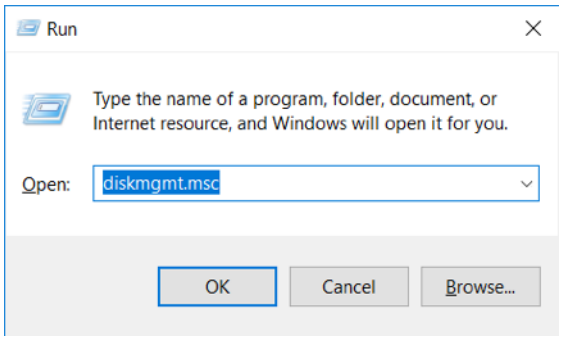
Partition a USB Drive and Make it Bootable by Using a Windows Computer

Follow the instructions here to use a Windows computer to prepare the USB drive for factory reset.

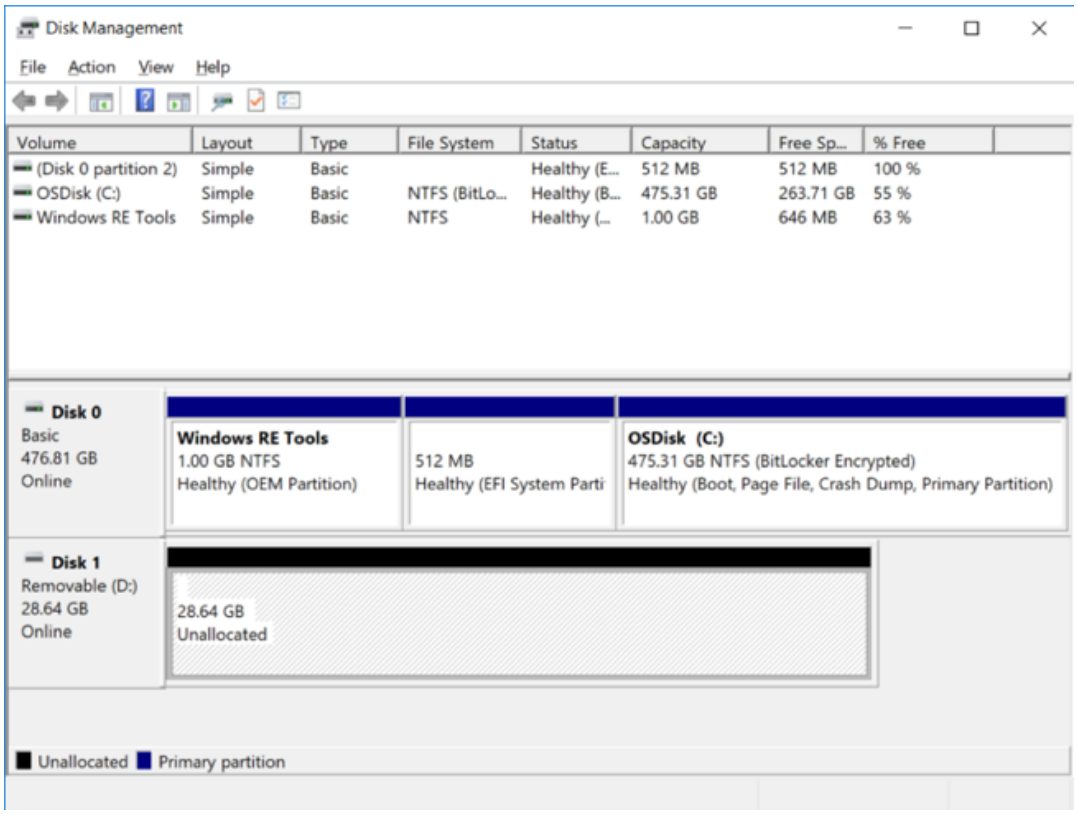
To partition the USB drive and make it bootable by using a Windows computer

1. To format the USB drive, follow these steps.

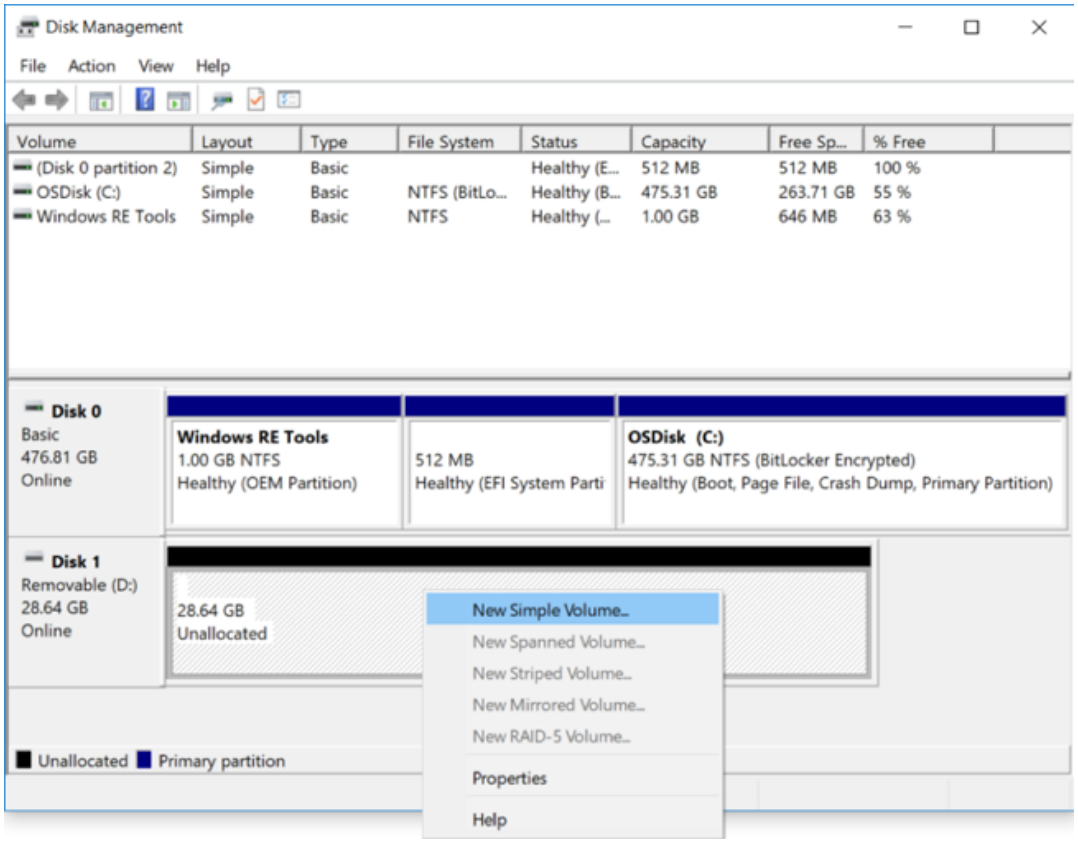
   a. Open the Windows command prompt, type `diskmgmt.msc`, and choose **OK** to launch the Windows Disk Management Console.



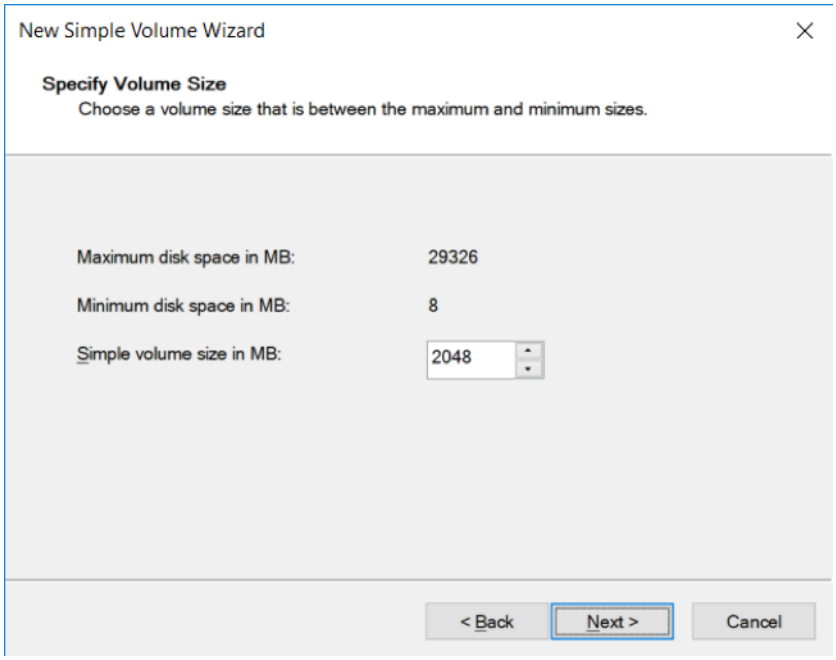
b. From the Disk Management console, choose the USB drive. Delete all the partitions and make the drive unallocated. The example in the screenshot here shows **Disk 1 Removable (D:)** as the USB drive.



c. To create the FAT32 partition of 2 GB capacity, open the Disk Management console and choose the USB drive. Open the context (right-click) menu and choose **New Simple Volume** from the context menu.

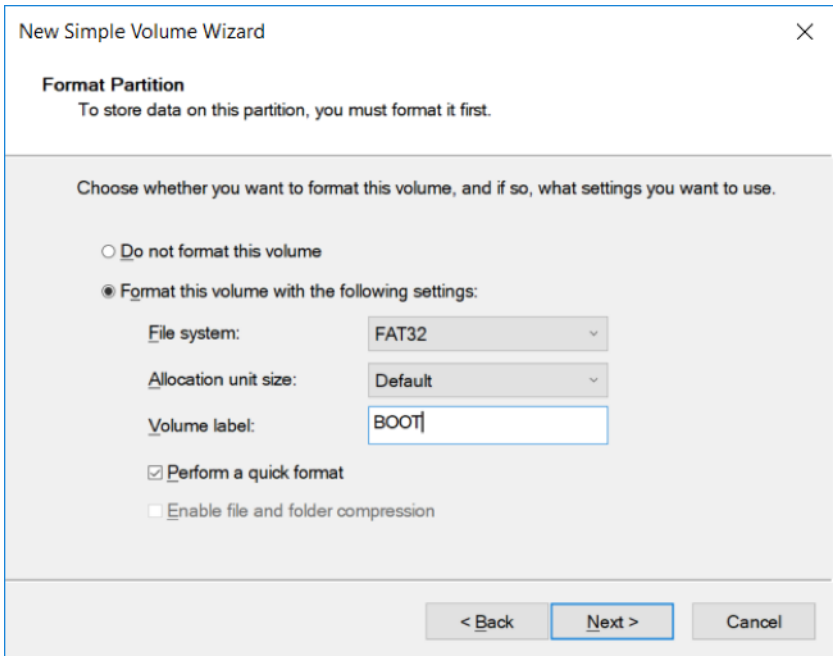


d. On the New Simple Volume Wizard, choose **2048** for **Simple volume size in MB** and then choose **Next**.

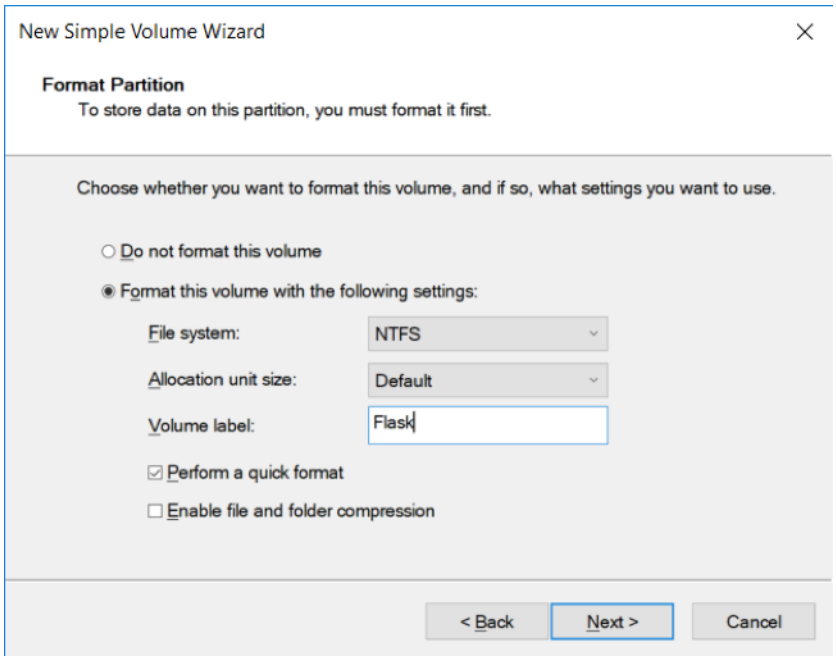


e. On the **New Simple Volume Wizard** page and under **Format Partition**, choose the **Format this volume with the following settings**. Then, choose `FAT32` for **File system**, `Default` for

Allocation unit size and any label (e.g., `BOOT`) for **Volume label**. Finally, choose **Next** to create the FAT32 partition.



f. To create the NTFS partition of the remaining disk capacity, open the Disk Management console. Choose the USB drive and open the context (right-click) menu to choose **New Simple Volume**. Choose the **Format this volume with the following settings** option. Choose NTFS for **File system**, `Default` for **Allocation unit size**, and a label (e.g, `Flask`) for **Volume label**. Finally, choose **Next** to start creating the NTFS partition.



2. To make the USB drive bootable from the FAT32 partition, follow these steps.

   a. Download the customized Ubuntu ISO image appropriate for your AWS DeepRacer model:

- For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS** logo on the chassis, download this customized Ubuntu ISO image (0.0.8 BIOS) .
- For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this customized Ubuntu ISO image (0.0.6 BIOS).

b. Go to https://unetbootin.github.io/ to download the *UNetbootin* software. Then start the UNetbootin console.

c. On the **UNetbootin** console, do the following:

i. Check the **Disimage** radio button.

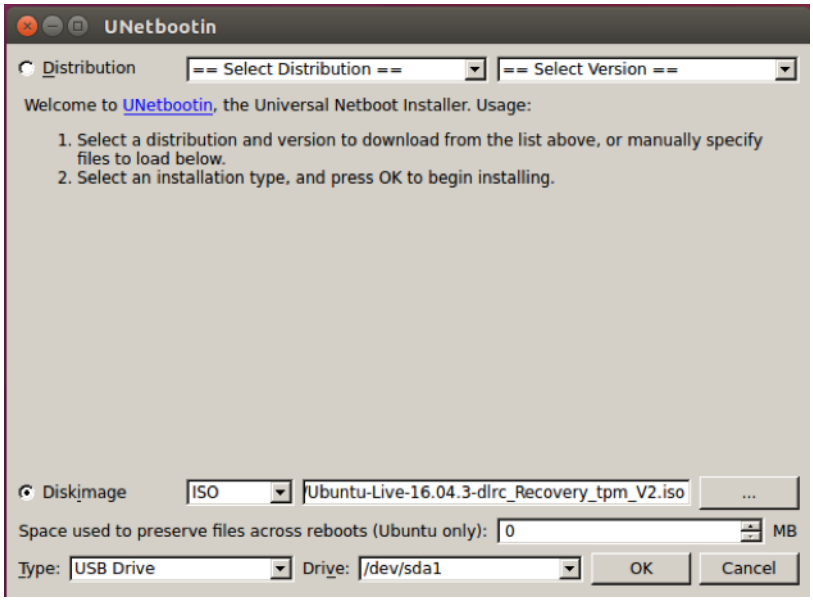
ii. For the disk image, choose **ISO** from the drop-down menu.

iii. Open the file picker to choose the downloaded Ubuntu ISO file.

iv. For **Type**, choose **USB Drive**.

v. For **Drive**, choose `/dev/sda1`.

vi. Choose **OK**.



Note
The customized Ubuntu image may be more recent than what's shown here. If so, use the most recent version of the Ubuntu image.

If you get a **/dev/sda1 not mounted** alert message, choose OK to close the message, unplug the USB drive, replug the drive, and then follow the steps above create the Ubuntu ISO image.

3. To copy the factory restore files to the NTFS partition of the USB drive, follow these steps.

a. Download the compressed factory restore package appropriate for your AWS DeepRacer model:

- For AWS DeepRacer vehicles ordered from amazon.com with a white **AWS** logo on the chassis, download this compressed factory restore package (0.0.8 BIOS) .
- For AWS DeepRacer vehicles originating from re:Invent 2018 with no text on the chassis, download this compressed factory restore package (0.0.6 BIOS).

b. Unzip the downloaded package. If your favorite tool can't unzip the file successfully, try using the PowerShell Expand-Archive command.

Restore Your AWS DeepRacer Vehicle to Factory Settings

Follow the instructions here to restore your AWS DeepRacer vehicle to its factory settings. Make sure you have made proper preparations as described in the section called “Preparing for Factory Reset” (p. 149).

> **Note**
> After the factory reset, all data stored on your AWS DeepRacer vehicle will be erased.

To restore your AWS DeepRacer vehicle to its factory settings

1. Connect your AWS DeepRacer vehicle to a monitor. You'll need an HDMI-to-HDMI, HDMI-to-DVI, or similar cable. Insert a compatable end of the cable into the HDMI port on the vehicle's chassis and plug the other end into a supported display port on the monitor.
2. Connect a USB keyboard and mouse. There are three AWS DeepRacer compute module USB ports in the front of the vehicle, on either side of, and including the port the camera is plugged into. A fourth USB port is found at the back of the vehicle. From above, the USB port is located in the space between the compute battery and the LED tail light.
3. Insert the prepared USB drive into an open port in your compute module. Turn on the power and repeatedly press the *ESC* key to enter BIOS.
4. From the BIOS window, choose **Boot From File**, then **The option with USB in it**, then **EFI**, then **BOOT**, and finally **BOOTx64.EFI**.
5. After the compute module is booted, wait for the device reset to start automatically when the power LED indicator starts to flash and a terminal window is presented to display the progress. You don't provide any further user input at this stage.

   If some error happens and the recovery fails, restart the procedure from **Step 1**. For detailed error messages, see the *result.log* file generated on the USB drive.
6. Wait for about 6 minutes for the power LED to stop flashing when the terminal closes automatically and the factory reset completes. The device then reboots itself automatically.
7. After the device is restored to factory settings, disconnect the USB drive from the vehicle's compute module.

After the factory reset, your AWS DeepRacer vehicle software is likely outdated. To update the vehicle software, go to the AWS DeepRacer device console and follow the instructions.

Document History for AWS DeepRacer Developer Guide

The following table describes the important changes to the documentation since the last release of AWS DeepRacer.

update-history-change	update-history-description	update-history-date
Updates for multi-vehicle racing and obstacle avoidance (p. 164)	AWS DeepRacer now supports new sensor types of stereo camera and LIDAR that enable multi-vehicle racing and obstacle avoidance. For more information, see the section called "Understanding Racing Types and Enabling Sensors" (p. 25).	December 2, 2019
Updates for community races (p. 164)	AWS DeepRacer now enables AWS DeepRacer users to organize their own racing events, known as community races, with private leaderboards open to invited users only. For more information, see *Participate in Virtual Races* (p. 119).	December 2, 2019
Updates for general availability (p. 164)	AWS DeepRacer now features more robust methods to train and evaluate deep learning models. The user interface is updated and explained. More options and precise data is available to build your own physical tracks. Troubleshooting information is available.	April 29, 2019
Initial release AWS DeepRacer Developer Guide (p. 164)	Initial release of the documentation to help the AWS DeepRacer user to learn reinforcement learning and explore its applications for autonomous racing, using the AWS DeepRacer console, the AWS DeepRacer simulator, and a AWS DeepRacer scale model vehicle.	November 28, 2018

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.