

# Reinforcement Learning

Throughout this course you are going to learn the fundamental concepts which underpin Reinforcement Learning - an exciting branch of machine learning which has real-world applications from training computers how to play computer games through to training autonomous vehicles.

## Getting Started

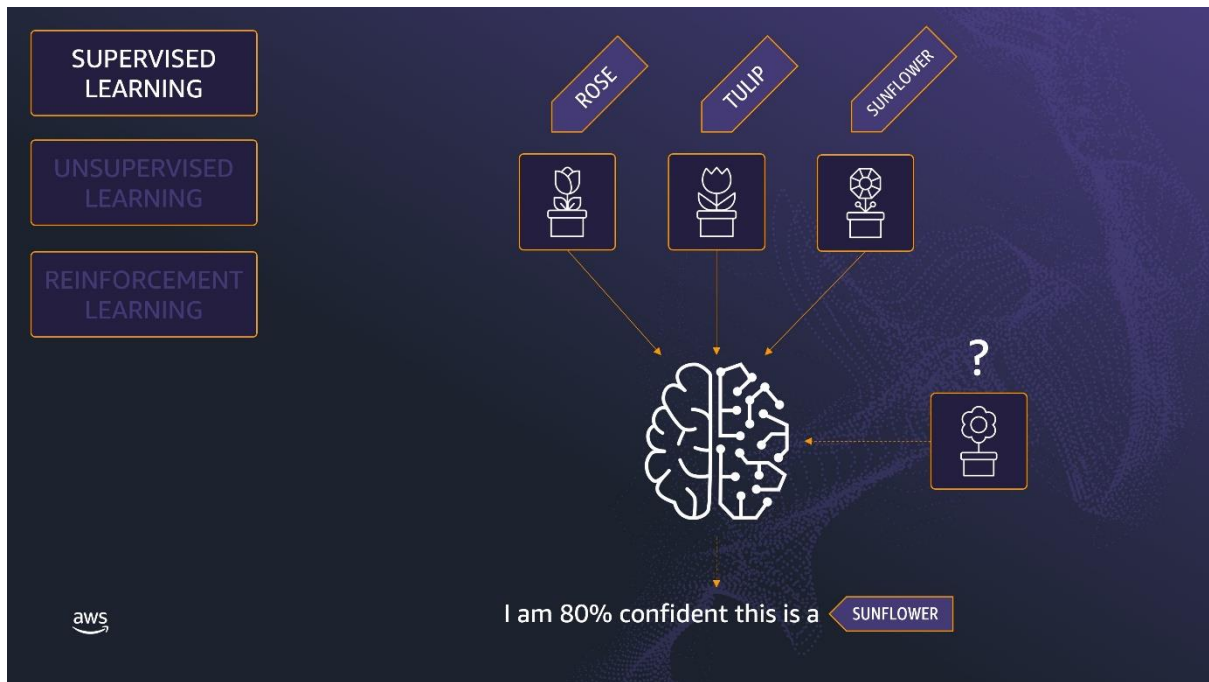
### Machine Learning Refresher

#### What is Machine Learning ?

Machine learning is part of the broader field known as artificial intelligence. This involves simulating human intelligence and decision making by building algorithms to take and process information which inform future decisions. Machine learning involves teaching an algorithm how to learn without being explicitly programmed to do so.

#### Supervised learning

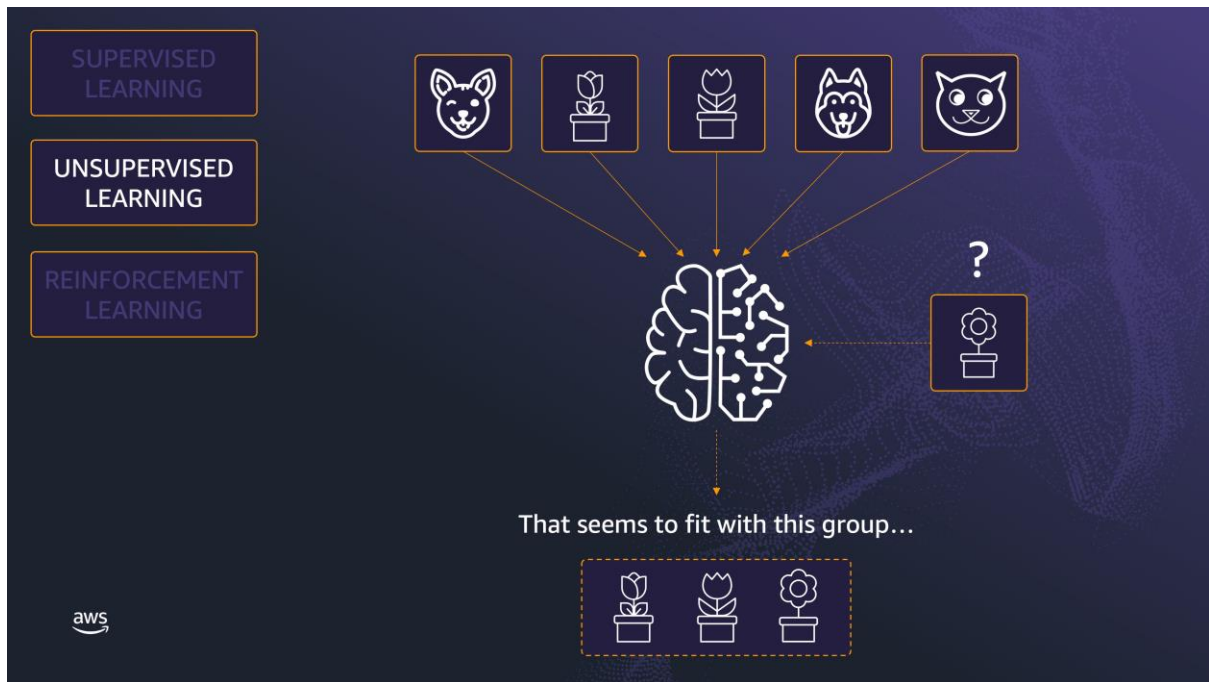
In supervised learning, every training sample from the dataset has a corresponding label associated with it, which essentially tells the machine learning algorithm what the training sample is. As a result, the algorithm can then learn from this data to predict labels for unseen data in the future.



In this example, the algorithm is being trained to identify flowers. So it would be given training data with images of flowers along with kind of flower each image contains (i.e. the label). The algorithm then uses this training data to learn and identify flowers in unseen images it may be provided in the future.

## Unsupervised learning

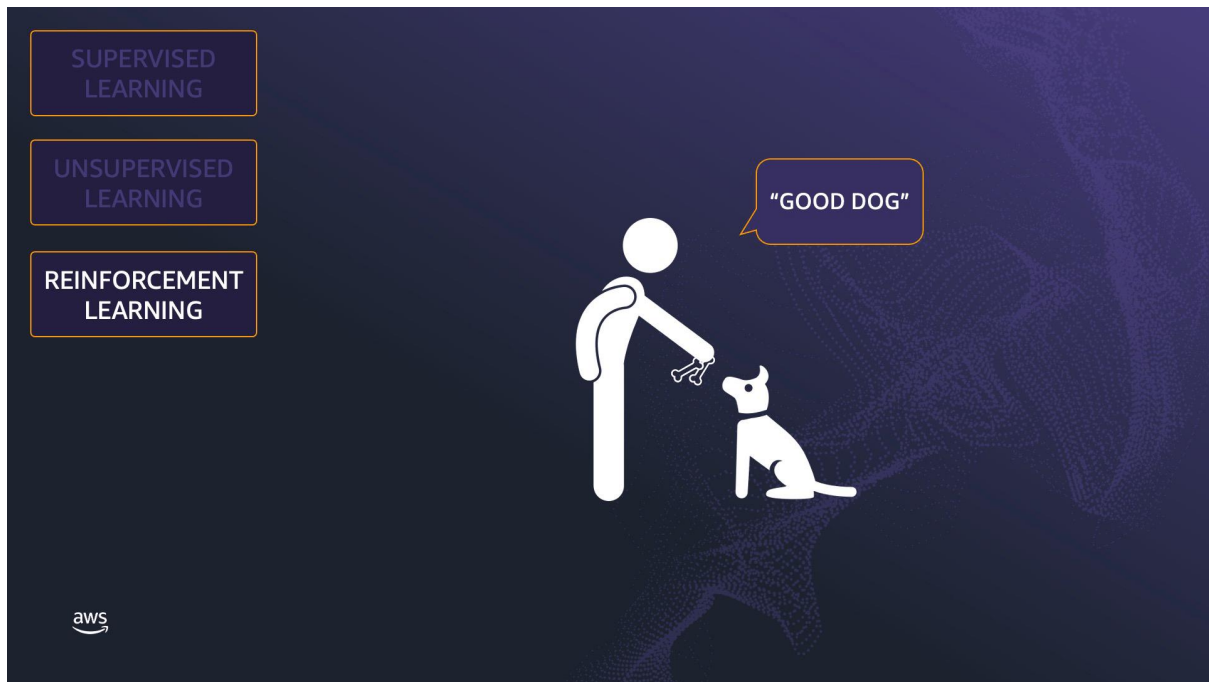
In unsupervised learning, there are no labels for the training data. The machine learning algorithm tries to learn the underlying patterns or distributions that govern the data.



In this example, the training data given to the machine learning algorithm does not contain labels to predict. Instead, the algorithm must identify patterns in the data itself. This can often be a benefit since it allows you to use massive datasets where labels are often not available.

## Reinforcement learning

Reinforcement learning is very different to supervised and unsupervised learning. In reinforcement learning, the algorithm learns from experience and experimentation. Essentially, it learns from trial and error.



In this example, we are training a dog. The dog will try and do different things in response to commands you may issue, such as “sit” or “stay”, and when it does the right thing you provide a treat, like a doggie biscuit. Over time the dog learns that to get a reward it needs to correctly follow your commands.

## Wrap Up

In this lesson you have completed a refresher of the three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning.

The rest of this course will be focusing on reinforcement learning and how you can use this to get started with AWS DeepRacer. See you on the track!

# Introduction to Reinforcement Learning

## What is Reinforcement Learning?

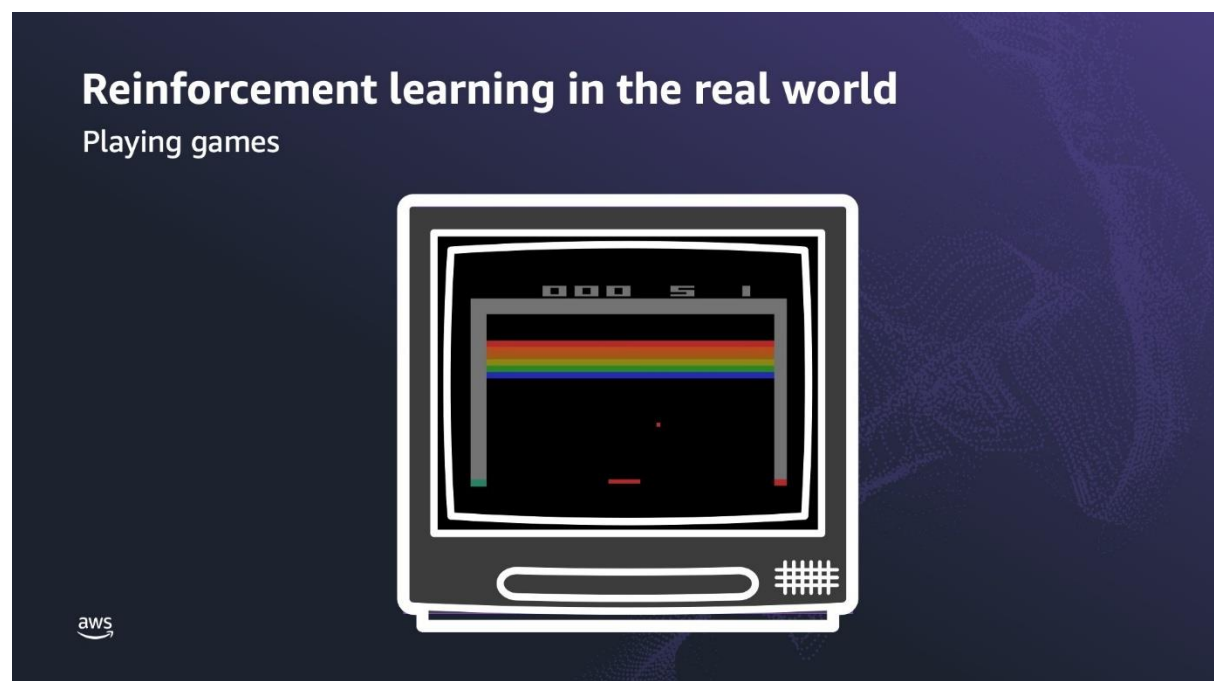
Reinforcement learning consists of several key concepts:

- **Agent** is the entity being trained. In our example, this is a dog.
- **Environment** is the “world” in which the agent interacts, such as a park.
- **Actions** are performed by the agent in the environment, such as running around, sitting, or playing ball.
- **Rewards** are issued to the agent for performing good actions.

Keep these terms in mind as you continue your journey with reinforcement learning. They will come up frequently and are very important.

Let's look at some examples of how reinforcement learning can be applied towards real world problems.

## Playing Games



Playing games is a classic example of applied reinforcement learning.

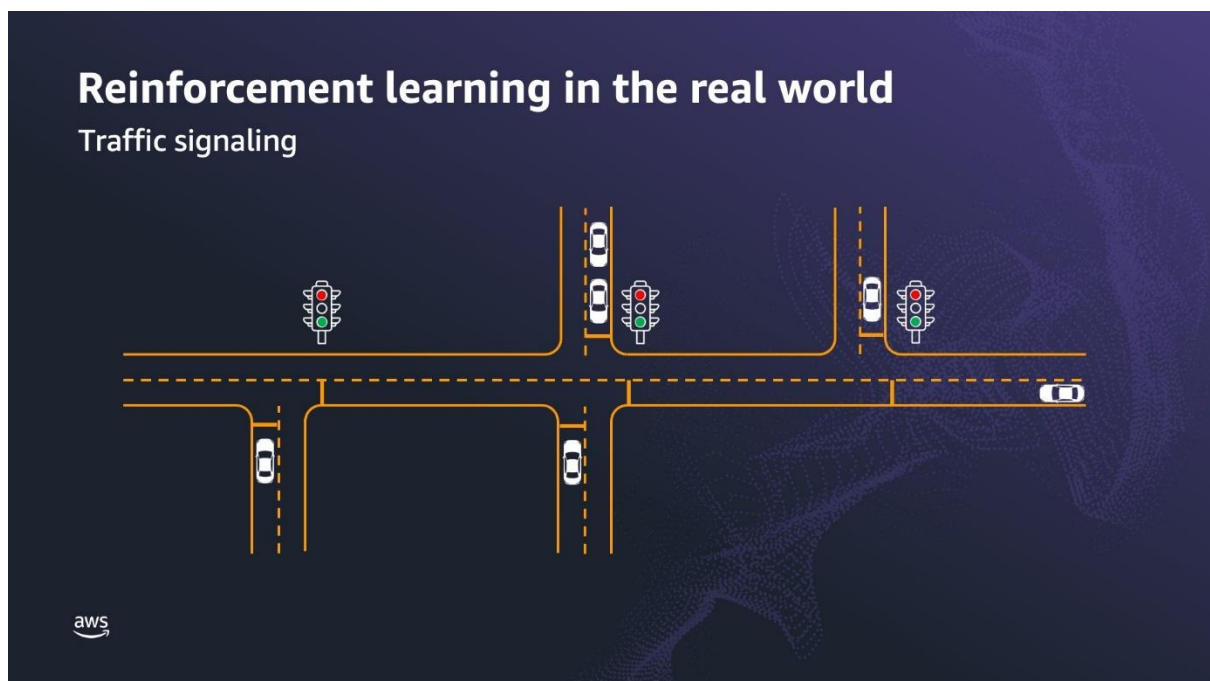
Let's use the game Breakout as an example. The objective of the game is to control the paddle and direct the ball to hit the bricks and make them disappear. A reinforcement learning model has no idea what the purpose of the game is, but by being rewarded for

good behavior (in this case, hitting a brick with the ball) it learns over time that it should do that to maximise reward.

In this situation, the:

- **Agent** is the paddle;
- **Environment** is the game scenes with the bricks and boundaries;
- **Actions** are the movement of the paddle; and
- **Rewards** are issued by the reinforcement learning model based upon the number of bricks hit with the ball.

## Traffic Signalling



Another use case for reinforcement learning is controlling and coordinating traffic signals to minimize traffic congestion.

How many times have you driven down a road filled with traffic lights and have to stop at every intersection as the lights are not coordinated? Using reinforcement learning, the model wants to maximise its total reward which is done through ensuring that the traffic signals change to keep maximum possible traffic flow.

In this use-case, the:

- **Agent** is the traffic light control system;
- **Environment** is the road network;

- **Actions** are changing the traffic light signals (red-yellow-green); and
- **Rewards** are issued by the reinforcement learning model based upon traffic flow and throughput in the road network.

## Autonomous Vehicles



A final example of reinforcement learning is for self-driving, autonomous, cars.

It's obviously preferable for cars to stay on the road, not run into anything, and travel at a reasonable speed to get the passengers to their destination. A reinforcement learning model can be rewarded for doing these things and will learn over time that it can maximize rewards by doing these things.

In this case, the:

- **Agent** is the car (or, more correctly, the self-driving software running on the car);
- **Environment** is the roads and surrounds on which the car is driving;
- **Actions** are things such as steering angle and speed; and
- **Rewards** are issued by the reinforcement learning model based upon how successfully the car stays on the road and drives to the destination.

## Wrap up

In this lesson you learned more about reinforcement learning and the key steps in developing a reinforcement learning model.



# Reinforcement Learning with a Mechanical Computer

## Reinforcement learning with a Mechanical Computer

This chapter uses the game of Hexapawn and a simple mechanical computer to provide insight into how reinforcement learning works. The computer is made out of 24 matchboxes that represent every state in the game and all possible moves from within each state. The matchboxes are filled with a few colored beads that each represent a move possible in that state. The computer is punished when it makes a poor decision by removing the bead corresponding to that move so that it can not be repeated. Through additional turns and negative feedback, the computer learns to optimize its moves and eventually win games against human players.

To learn more about Hexapawn, you can find the original article written by AI researcher Martin Gardner here: “How to build a game-learning machine and then teach it to play and win” by Martin Gardner, Scientific American: <http://cs.williams.edu/~freund/cs136-073/GardnerHexapawn.pdf>.

## Wrap Up

In this lesson we learned about reinforcement learning using a mechanical computer.

# Reinforcement Learning with AWS DeepRacer

## Reinforcement Learning with AWS DeepRacer

### Applying reinforcement learning to AWS DeepRacer



### Summary

AWS DeepRacer is a 1/18th scale racing car, with the objective being to drive around a track as fast as possible. To achieve this goal, AWS DeepRacer uses reinforcement learning.

- The **agent** is the AWS DeepRacer car (or, more specifically, the software running on the car);
- The agent wants to achieve the goal of finishing laps around the track as fast as possible, so the track is the **environment**.
- The agent knows about the environment through the **state** which is the portion of the environment known to the agent. In the case of AWS DeepRacer, it is the images being captured by the camera.
- Once the agent knows its state in the environment, it can perform **actions** in the environment to help it achieve its goal. In the case of DeepRacer, this might be accelerating, braking, turning left, turning right, or going straight.
- The agent then receives feedback in the form of a **reward** about how well that action contributed towards achieving its goal.

- And all this happens within an **episode**. This can be thought of as a cycle of the agent performing an action in the environment (based upon the state it has observed) and then receiving feedback in the form of a reward which informs future actions it might take.

## Wrap up

In this lesson we looked at how reinforcement learning is implemented in AWS DeepRacer, and also how the various components of reinforcement learning relate specifically to AWS DeepRacer.

# Training your first AWS DeepRacer Model

## Training an AWS DeepRacer model

### **Summary**

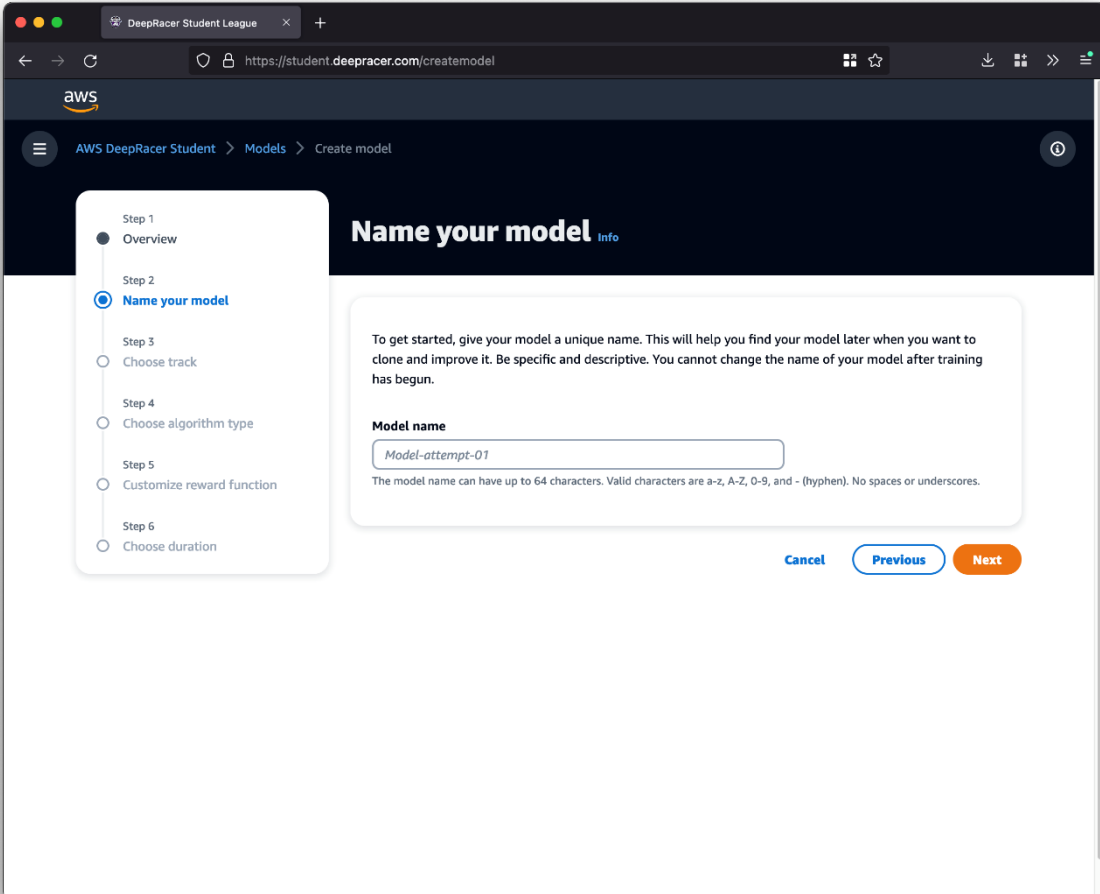
Creating a model in AWS DeepRacer Student is a six-step process:

- Name your model
- Choose track
- Choose algorithm type
- Customize reward function
- Choose duration
- Train your model

The following sections will review these six steps which were covered in the video.

## Step 1: Name your Model

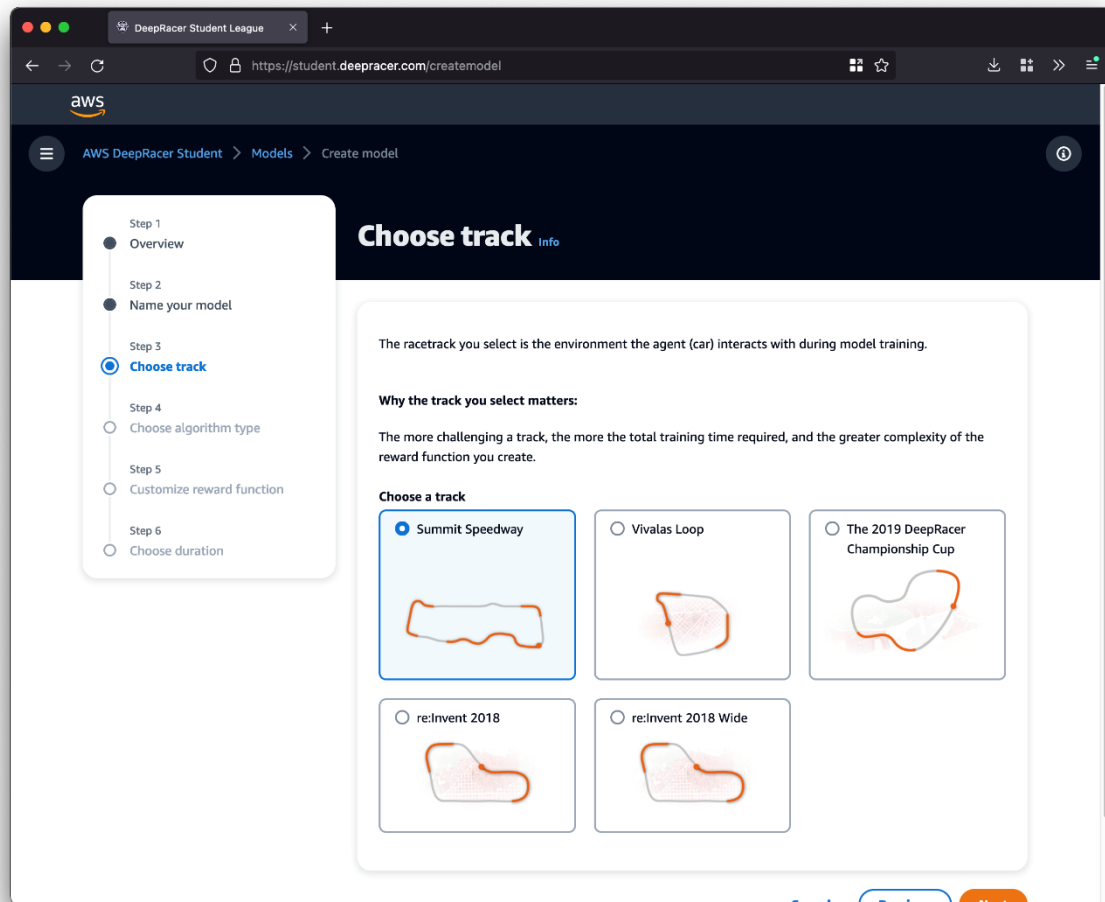
As the first step you will need to name your model. Make sure you name your model such that it is both specific and descriptive, as you are likely going to have quite a few models across the season.



The screenshot shows the AWS DeepRacer Student League interface in a web browser. The browser tab is titled 'DeepRacer Student League' and the address bar shows 'https://student.deepracer.com/createmodel'. The AWS logo is in the top left. The breadcrumb navigation is 'AWS DeepRacer Student > Models > Create model'. On the left, a vertical sidebar lists six steps: Step 1 Overview, Step 2 Name your model (selected with a blue circle), Step 3 Choose track, Step 4 Choose algorithm type, Step 5 Customize reward function, and Step 6 Choose duration. The main content area has a dark header with 'Name your model' and an 'Info' icon. Below this, a text box explains: 'To get started, give your model a unique name. This will help you find your model later when you want to clone and improve it. Be specific and descriptive. You cannot change the name of your model after training has begun.' A 'Model name' input field contains 'Model-attempt-01'. Below the field, a note states: 'The model name can have up to 64 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen). No spaces or underscores.' At the bottom right are three buttons: 'Cancel', 'Previous' (highlighted with a blue border), and 'Next' (highlighted in orange).

My suggestion is using the track or race name and a version number - so for this demonstration, I am going to use *SummitSpeedway-V1* (as I am going to be training on that track) and I would then increment the version number with each clone of the model.

## Step 2: Choose Track

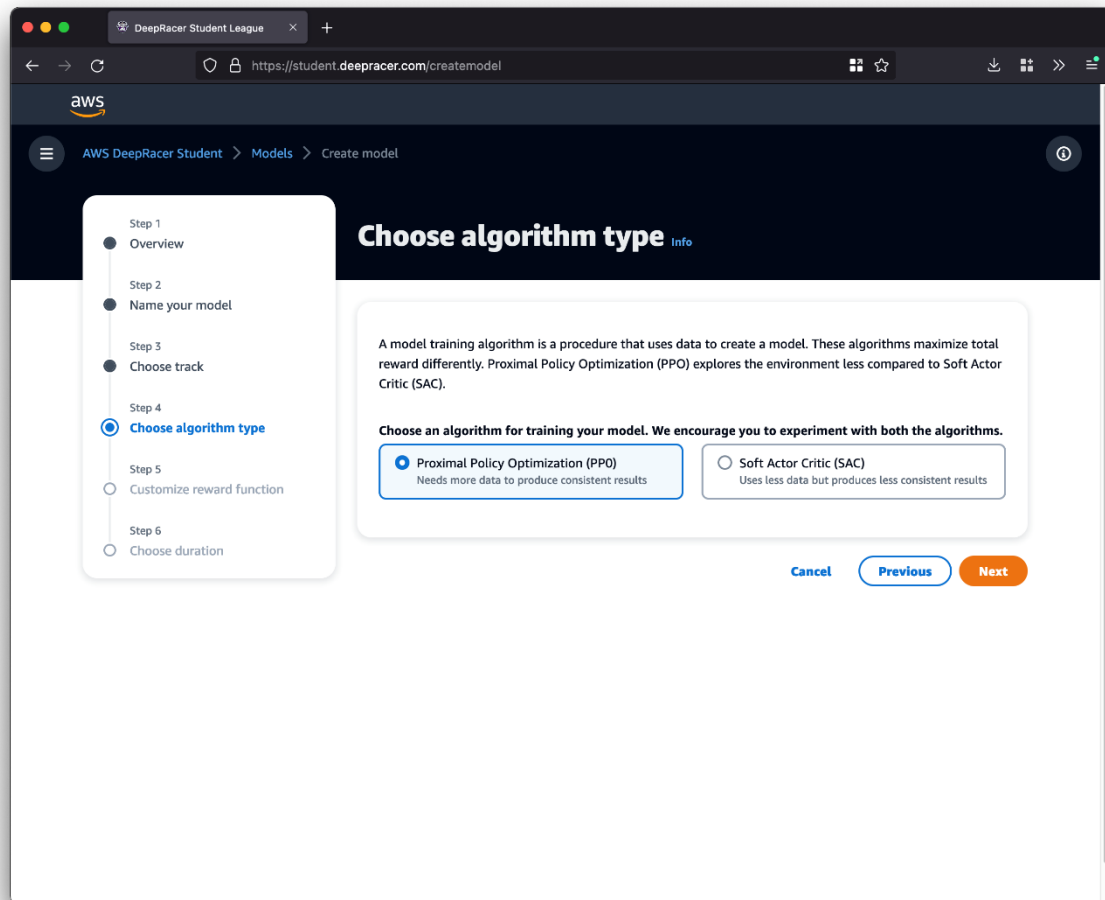


Now you need to select the track to use for training your model.

Note that the more challenging the track, the more training time you will need - and possibly also a more complex reward function.

As a rule of thumb, if you are competing in the Student League then train on the track which the competition is using. In this example, I am training for a race on the Summit Speedway track.

## Step 3: Choose Algorithm Type



In this step you can choose the reinforcement learning training algorithm to use. This is where things start to get really interesting, as we are now exercising some control over how our model learns.

In this lesson we are just concerned about getting a training job up and running, so choose either PPO or SAC but don't worry too much about which one you select. In the next lesson we will deep dive into the two algorithms.

## Step 4; Customize the Reward Function

The screenshot shows the AWS DeepRacer Student League interface. The sidebar on the left lists the steps: Step 1 Overview, Step 2 Name your model, Step 3 Choose track, Step 4 Choose algorithm type, Step 5 Customize reward function (selected), and Step 6 Choose duration. The main content area is titled 'Customize reward function' and includes an 'Info' link. It explains that a reward function is a Python code that describes immediate feedback in the form of a reward or penalty to move from a given position on the track to a new position. It also states the purpose of a reward function: to encourage the vehicle to make moves along the track quickly to reach its destination. Three sample reward functions are presented: 'Follow the centerline' (rewarded for staying close to the centerline), 'Stay within borders' (rewarded for staying within the track borders), and 'Prevent zig-zag' (rewarded for smooth driving). A 'Walk me through this code' button is provided for the 'Follow the centerline' function. The code block shows the following Python code:

```
1 def reward_function(params):
2     # Example of rewarding the agent to follow
      center line
3
4     # Read input parameters
5     track_width = params['track_width']
```

You now have the opportunity to customize the reward function. This is the piece of code which determines how much the agent should be rewarded for its actions.

There are three sample reward functions available, and we are going to deep dive into these in a future lesson - so for the moment select “Follow the centerline” which will give you a reward function that will reward the agent for staying close to the centreline of the track. This is a great starting reward function which you can build upon later.



## Step 5: Choose Duration

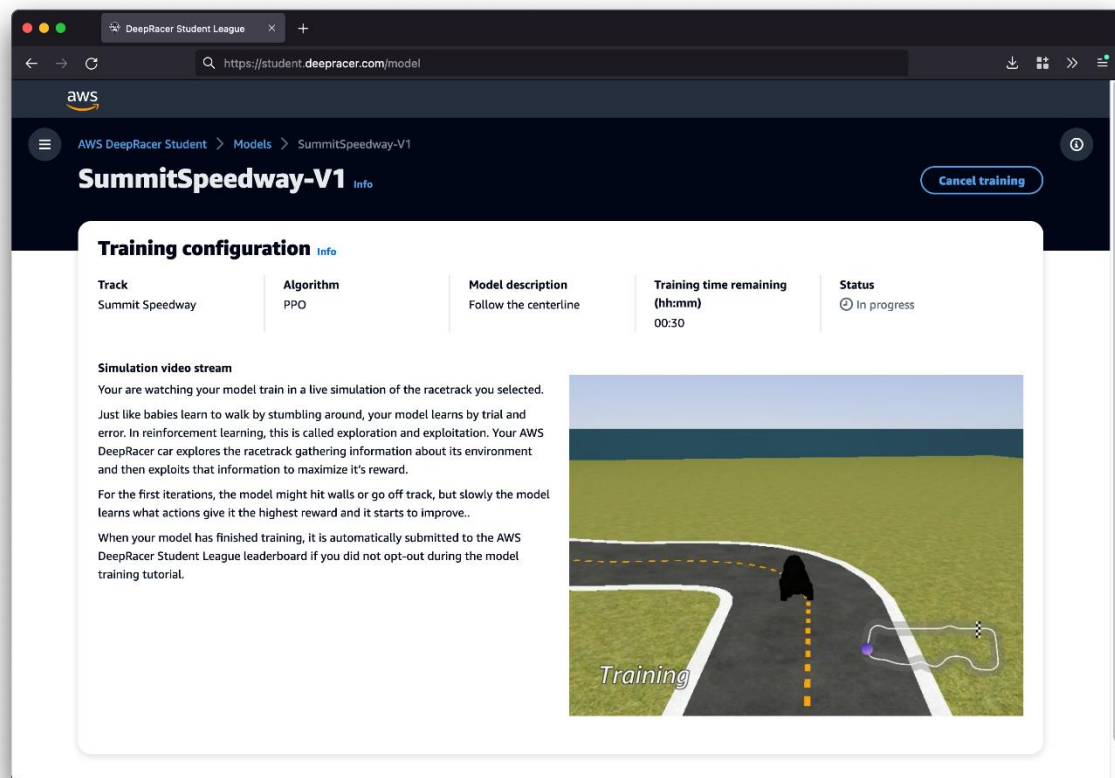
The screenshot shows the 'Choose duration' step in the AWS DeepRacer Student League interface. On the left, a vertical sidebar lists six steps: Step 1 Overview, Step 2 Name your model, Step 3 Choose track, Step 4 Choose algorithm type, Step 5 Customize reward function, and Step 6 Choose duration (which is highlighted with a blue circle). The main content area has a dark header with the AWS logo and navigation links. Below the header, the title 'Choose duration' is displayed with an 'Info' link. A paragraph explains that training duration impacts performance. A dropdown menu for 'Choose duration of model training' is set to '60 minutes', with a note that time must be between 10 and 600 minutes. A 'Model description' section includes a text area with the placeholder 'Follow the centerline' and a 255-character limit. A checkbox is checked, indicating agreement to the terms and conditions. At the bottom, there are three buttons: 'Cancel', 'Previous', and 'Train your model' (which is highlighted in orange).

Finally, you can configure the options for model training. I suggest you start by doing 60 minutes of training, and give your model a description so you can find it later on (such as a quick summary of the chosen algorithm along with reward function).

If you would like to participate in the Student League make sure you also tick the box to submit to the leaderboard - there's no harm in doing this, even for a simple model, so give it a shot. You can submit retrained and new models as many times as you like.

When you are ready to start training, select the "Train your model" button.

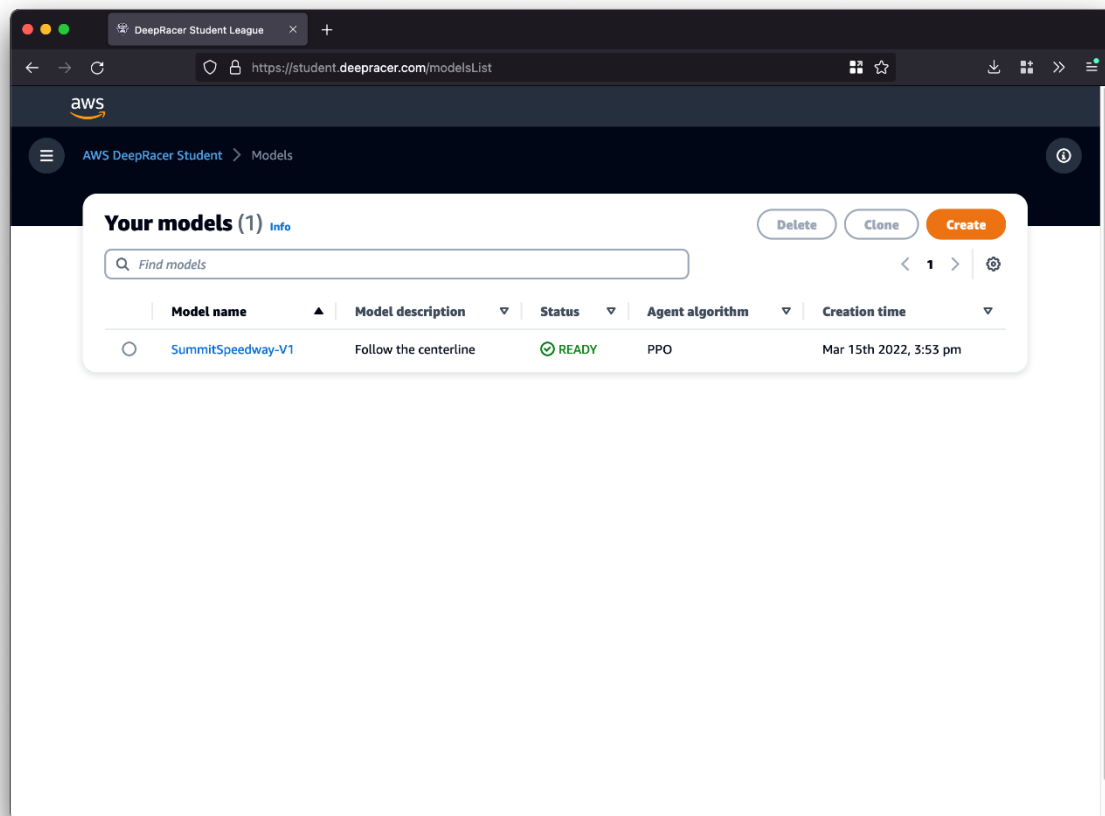
## Step 6: Training



When the training starts you will be able to see a simulated video stream of the training.

Don't worry if the model is going off-track or doing things it shouldn't, this is all part of the training process. Remember, reinforcement learning is essentially learning by trial and error. The agent is exploring the environment to gather information (called exploration) and will then use that information to try and maximize its reward (called exploitation).

Once the training has finished you can see your model by going into the "Models" section in the AWS DeepRacer Student console. You can also clone your model to continue training or perhaps modify the reward function. We will be talking more about cloning and improving an existing model in a future deep dive chapter.



## Wrap Up

Congratulations on training your first AWS DeepRacer model!

# Deep Dives

## Training Algorithms

### Introduction

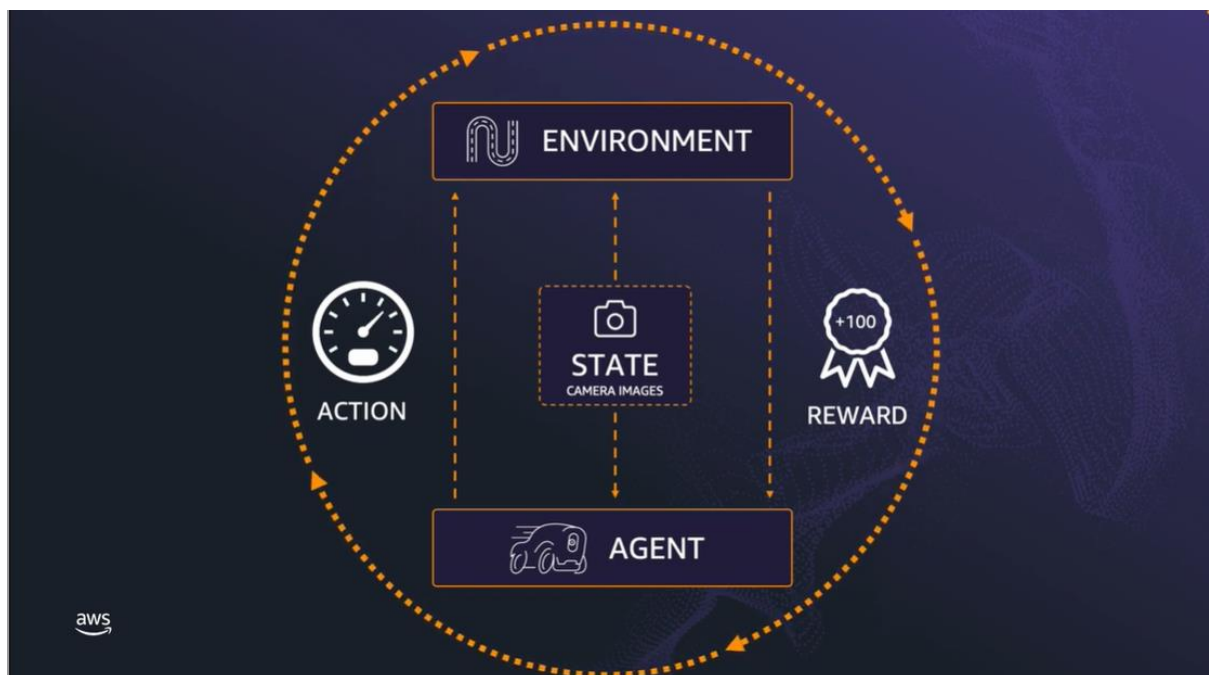
In the previous chapters we discussed that when training a machine learning model an algorithm is used. Algorithms are sets of instructions, essentially computer programs. Machine learning algorithms are special programs which learn from data. This algorithm then outputs a model which can be used to make future predictions.

AWS DeepRacer offers two training algorithms:

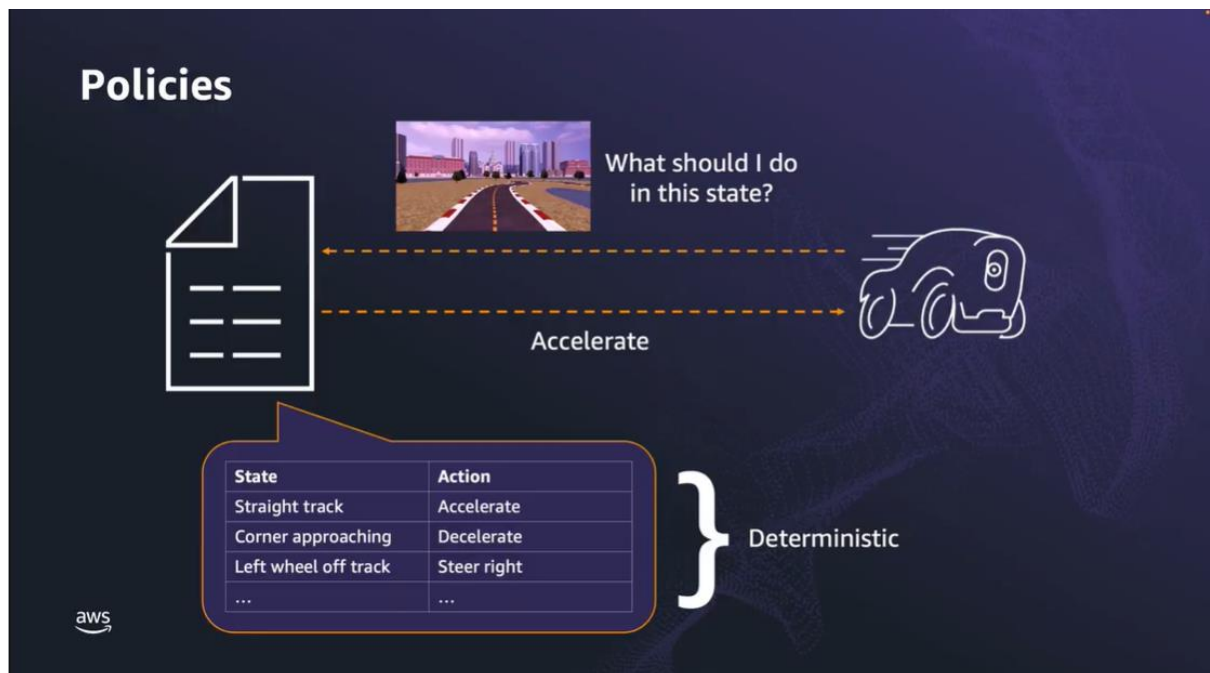
- Proximal Policy Optimization (PPO)
- Soft Actor Critic (SAC)

This chapter is going to take you through the differences between these two algorithms.

### Policies



A policy defines the action that the agent should take for a given state. This could conceptually be represented as a table - given a particular state, perform this action.



This is called a **deterministic policy**, where there is a direct relationship between state and action. This is often used when the agent has a full understanding of the environment and, given a state, always performs the same action.

Consider the classic game of rock, paper, scissors. An example of a deterministic policy is always playing rock. Eventually the other players are going to realize that you are always playing rock and then adapt their strategy to win, most likely by always playing paper. So in this situation it's not optimal to use a deterministic policy.



So, we can alternatively use a **stochastic policy**. In a stochastic policy you have a range of possible actions for a state, each with a probability of being selected. When the policy is queried to return an action for a state it selects one of these actions based on the probability distribution.

This would obviously be a much better policy option for our rock, paper, scissors game as our opponents will no longer know exactly which action we will choose each time we play.

You might now be asking, with a stochastic policy how do you determine the value of being in a particular state and update the probability for the action which got us into this state? This question can also be applied to a deterministic policy; how do we pick the action to be taken for a given state?

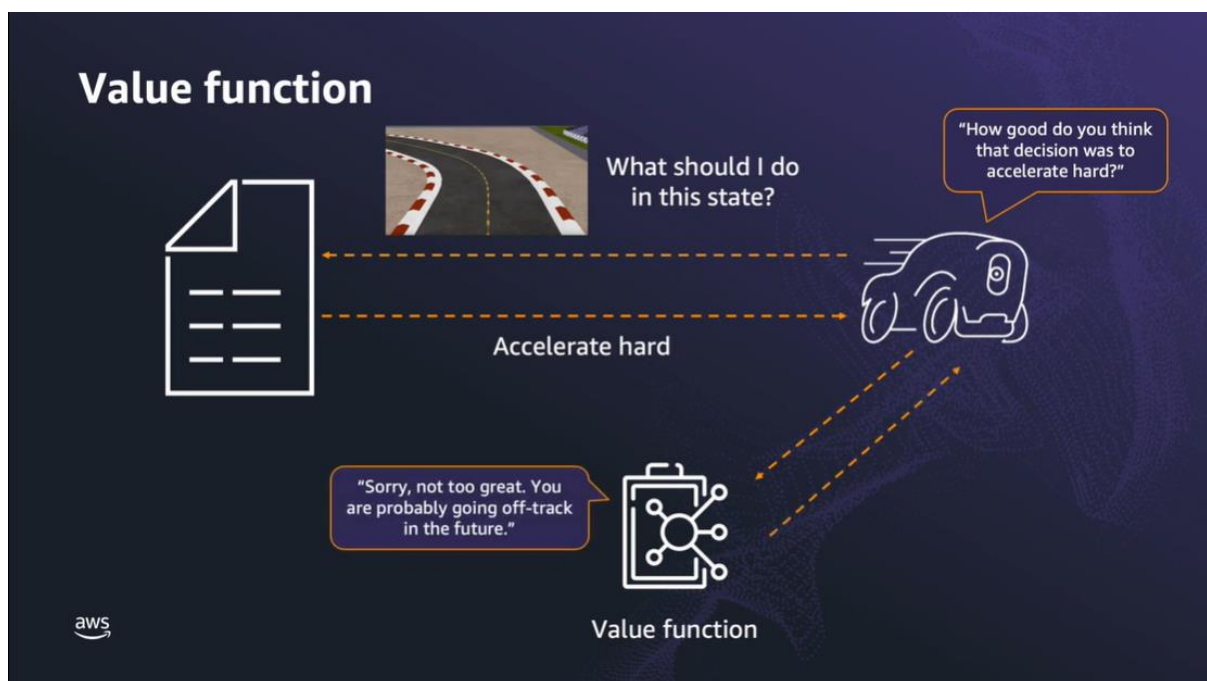
Well, we somehow need to determine how much benefit we have derived from that choice of action. We can then update our stochastic policy and either increase or decrease the probability of that chosen action being selected again in the future, or select the specific action with the highest likelihood of future benefit as in our deterministic policy.

If you said that this is based on the reward, you are correct. However, the reward only gives us feedback on the value of the single action we just chose. To truly determine the

value of that action (and resulting state) we should not only look at the current reward, but future rewards we could possibly get from being in this state.

## Value Function

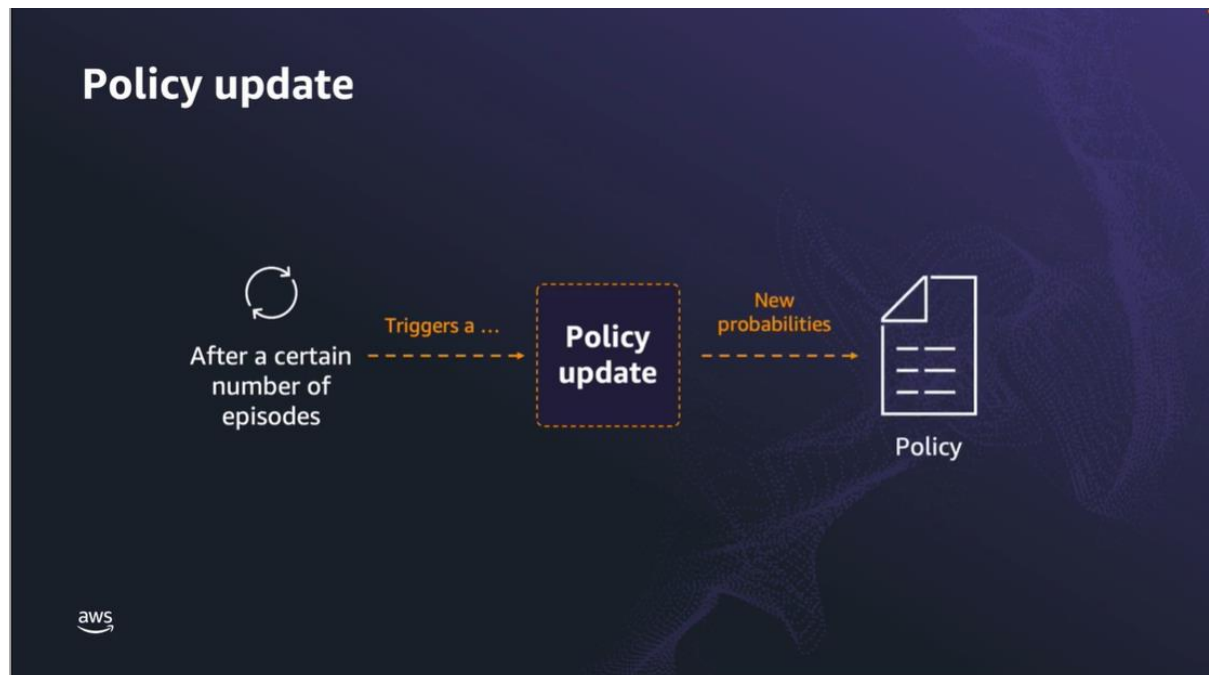
The chapter finished with the question about how we can determine possible future rewards from being in a certain state. This is done through the **value function**. Think of this as looking ahead into the future and figuring out how much reward you expect to get given your current policy.



Say the DeepRacer car (agent) is approaching a corner. The algorithm queries the policy about what to do, and it says to accelerate hard. The algorithm then asks the value function how good it thinks that decision was - but unfortunately the results are not too good, as it's likely the agent will go off-track in the future due to his hard acceleration into a corner. As a result, the value is low and the probabilities of that action can be adjusted to discourage selection of the action and getting into this state.

This is an example of how the value function is used to critique the policy, encouraging desirable actions while discouraging others.

We call this adjustment a **policy update**, and this regularly happens during training. In fact, you can even define the number of episodes that should occur before a policy update is triggered.



In practice the value function is not a known thing or a proven formula. The reinforcement learning algorithm will estimate the value function from past data and experience.

## PPO and SAC

Let's take a look at the similarities and differences between PPO and SAC in relation to how they learn.

The first thing to point out is that AWS DeepRacer uses both PPO and SAC algorithms to train stochastic policies. So they are similar in that regard. However, there is a key difference between the two algorithms.



## Training algorithms



**Proximal Policy  
Optimization (PPO)**

*On-Policy*

**Soft Actor Critic  
(SAC)**



**PPO** uses “**on-policy**” learning. This means it learns only from observations made by the current policy exploring the environment - using the most recent and relevant data. Say you are learning to drive a car, on-policy learning would be analogous to you reviewing a video of your most recent lesson and taking note of what you did well, and what needs improvement.

## Training algorithms



**Proximal Policy  
Optimization (PPO)**

*On-Policy*

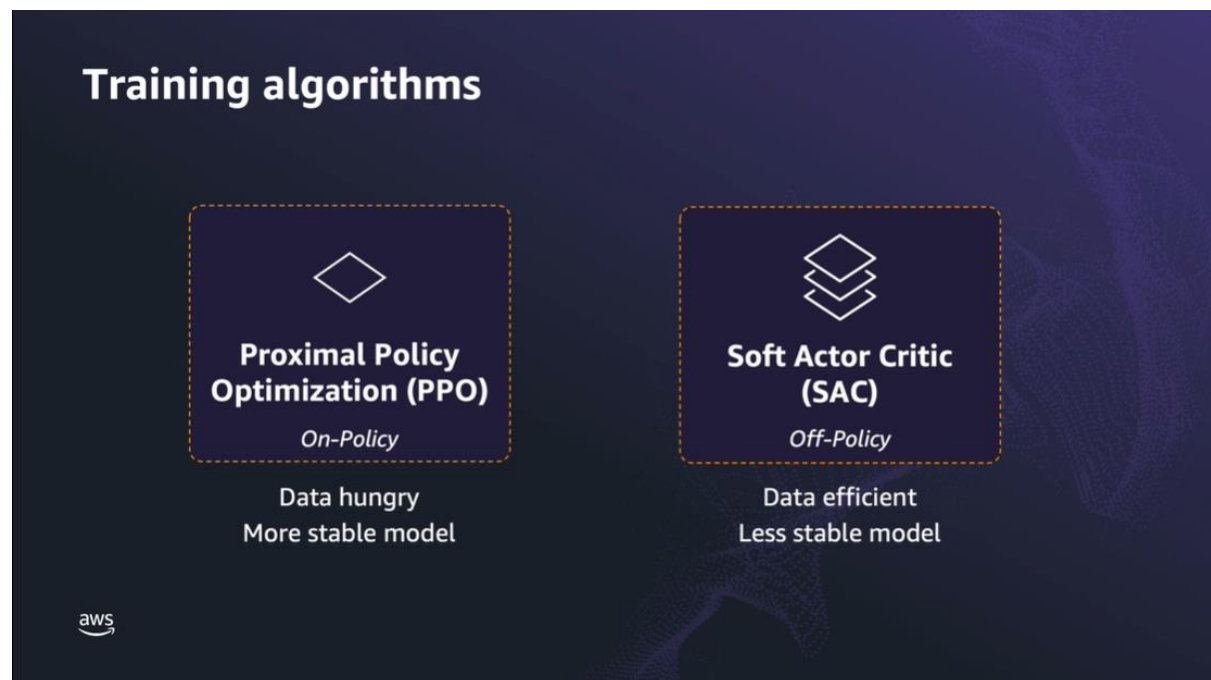


**Soft Actor Critic  
(SAC)**

*Off-Policy*



In contrast, **SAC** uses “**off-policy**” learning. This means it can use observations made from previous policies exploration of the environment - so it can also use old data. Going back to our learning to drive analogy, this would involve reviewing videos of your driving lessons from the last few weeks. Even though you have probably improved since those lessons, it can still be helpful to watch those videos in order to reinforce good and bad things. It could also include reviewing videos of other drivers to get ideas about good and bad things they might be doing.



So what are some benefits and drawbacks of each approach?

- PPO generally needs more data as it has a reasonably narrow view of the world, since it does not consider historical data - only the data in front of it during each policy update. In contrast, SAC does consider historical data so it needs less new data for each policy update.
- That said, PPO can produce a more stable model in the short-term as it only considers the most recent, relevant data - compared with SAC which might produce a less stable model in the short-term since it considers less relevant, historical data.

So which should you use? There is no right or wrong answer. SAC and PPO are two algorithms from a field which is constantly evolving and growing. Both have their benefits and either one could work best depending on the circumstance.

As you'll learn as you continue along your machine learning journey, it involves a lot of experimentation and tuning to see what is going to work best for you.

## Wrap Up

In this lesson we have deep dived into the background of reinforcement learning algorithms in terms of policies and value functions, and discussed differences between the two algorithms available in AWS DeepRacer: PPO and SAC.

# Reward Functions

## Introduction

You might recall from when you trained your first model that you can define a reward function. The purpose of the reward function is to issue a reward based upon how good, or not so good, the actions performed are at reaching the ultimate goal. In the case of AWS DeepRacer, that goal is getting around the track as quickly as possible.

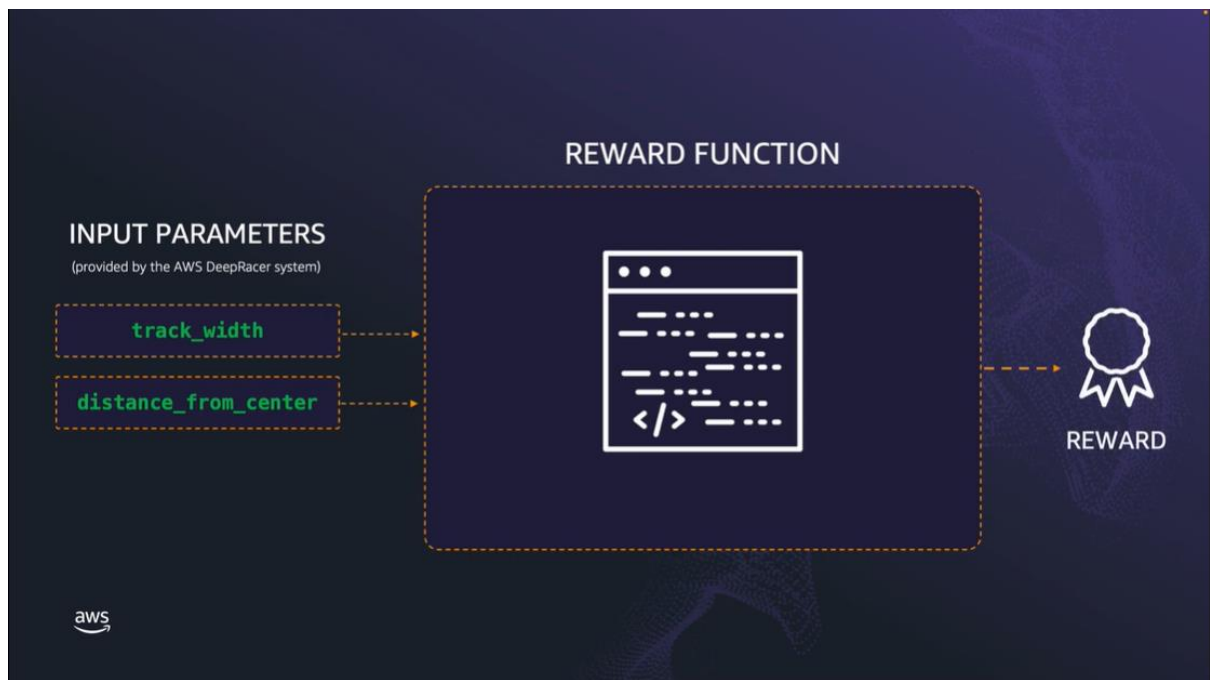
So the logical question you might be asking is - "how does the reward get calculated and issued?". Well, this one is over to you - as you have control over the reward function, which is the piece of code that determines and returns the reward value.

In the next section you will learn about more the reward function and how it works.

## The Reward Function

In order to calculate an appropriate reward you need information about the state of the agent and perhaps even the environment. These are provided to you by the AWS DeepRacer system in the form of input parameters - in other words, they are parameters for input into your reward function.

There are over 20 parameters available for use, and the reward function is simply a piece of code which uses the input parameters to do some calculations and then output a number, which is the reward.



The reward function is written in Python as a standard function, but it must be called *reward\_function* with a single parameter - which is a Python dictionary containing all the input parameters provided by the AWS DeepRacer system.

## Improving the reward Function

Just because you have trained a model doesn't mean you cannot change the reward function. You might find that the model is exhibiting behavior you want to de-incentivize, such as zig-zagging along the track. In this case you may include code to penalize the agent for that behavior.



These reward functions we looked at in this section are just examples, and you should experiment and find one which works well for you. A list of all the different input parameters available to the reward function can be found in the AWS DeepRacer Developer Guide, with full explanations and examples: [here](#).

The reward function can be as simple, or as complex, as you like - just remember, a more complex reward function doesn't necessarily mean better results.

## Wrap Up

Well done on completing this deep dive on reward functions. You should now have a good working knowledge of how the sample reward functions work, along with some ideas about how you might be able to craft your very own reward function!

As a final tip, whenever you implement a reward function make sure you select the “Validate” button. This will check to make sure your reward function doesn’t have any errors that will prevent it from running. Note, this does not provide any assurances about how good your reward function might actually be in practice - it is simply a check to make sure there's no syntax errors.

# How the AWS DeepRacer Works

## Introduction

The AWS DeepRacer device is a 1/18th scale race car. It has an onboard computer equipped with Wi-Fi and an Intel Atom processor running Ubuntu Linux 20.04. The onboard computer stores machine learning models and provides the processing power that allows the car to make decisions and navigate the race track. The AWS DeepRacer device can be equipped with up to two single-lens cameras and a LiDAR sensor.

In this chapter we are diving deep into how the AWS DeepRacer device works, starting with an overview of AWS DeepRacer. We will go over how the AWS DeepRacer device makes decisions, the challenges involved, and how the OpenVINO toolkit is used in conjunction with AWS DeepRacer to address these challenges, allowing the AWS DeepRacer device to make real-time decisions.

Let's get started by taking a closer look at how the AWS DeepRacer device can make these real-time decisions in a race.

## How does the AWS DeepRacer vehicle make decisions?

Once you have trained a model using the AWS DeepRacer console, it can be loaded onto an AWS DeepRacer device. The onboard computer on the AWS DeepRacer uses this model to make decisions, a process known as machine learning inference.

Machine learning inference involves running input data through a machine learning model which then outputs a prediction. This prediction is based upon past experience and subsequent knowledge which the model has gained through this training.

In the case of AWS DeepRacer, the machine learning model takes input in the form of images from the cameras, and data from the optional LiDAR sensor. The model then provides output as to what it thinks is the optimal steering and speed actions the car should take to stay on the track.

For example, a well trained model would output that the car has to take a left turn for an image input captured at a left turn on the track - as this would maximize its chance of staying on the track and completing the lap.

The inference process can be compute intensive. In the case of AWS DeepRacer, the inference needs to occur in real-time as the car drives on the track.

We can measure the performance of inference with two metrics:

- The "inference rate" which is the number of inferences which can be done per second; and
- The "inference time" or "inference latency" which is time taken to run a single inference.

Ultimately, we want to maximize the rate of inference and therefore the number of decisions made every second. However, the rate of inference is dependent on many factors. The most obvious is the performance of the machine running the inference, such as the CPU (Central Processing Unit) speed, whether any GPU (Graphical Processing Unit) acceleration is present, and the amount of system RAM (Random Access Memory) available.

There are also other factors. One is the machine learning framework used for training. There are many different frameworks available, such as TensorFlow, PyTorch, and Apache MXNet. These frameworks provide the tools and services to build machine learning models, including the training algorithms such as PPO (Proximal Policy Optimization) and SAC (Soft Actor Critic), which you might be familiar with from chapter **3.1 Training algorithms** in this module.

You don't need to worry about these frameworks since it's all taken care of for you with AWS DeepRacer, but it is worthwhile at least having some awareness of their presence and purpose.

Now that we have a better understanding of how the AWS DeepRacer makes decisions, we will look at some of the challenges involved in making this all work in the next section.

## Challenges in real-world applications of machine learning inference

When people think of machine learning (ML) models they often imagine them running on large, powerful computers. Some real-world ML applications, like real-time traffic monitoring, need computers powerful enough to provide the high inference rates necessary for complex, high stakes tasks.



It also may be more cost-effective to centralize inferencing in one place. In these kinds of situations the data is fed from edge devices, such as traffic cameras, all across a city to a central server for processing.

However, what happens when sending the data to a central server for inference isn't a good option and we need to perform inference locally? This is the situation with AWS DeepRacer. It would take too long to send the data to a remote server, perform the inference, and return the result to the AWS DeepRacer car. By that time the vehicle might have already driven off the track! In this situation we need to perform the inference locally on the device. We call this "inference at edge".

However, this presents a challenge. The compute available on edge devices is restricted. In the case of AWS DeepRacer, it is a relatively small battery powered computer. So, we need to balance the needs of efficient and effective inference while also working within the bounds of the compute available.

In the next section, we are going to look at some general challenges of computing at the edge and learn how to optimize for your device.

## Addressing the challenges

So far we have discussed two places where inference can be performed: on large, powerful computers—like a cloud server—or on low-power devices like the AWS DeepRacer device. We call these low-power devices edge devices. Inference on these devices is called real-time inference at edge.

Latency is the time that it takes to receive a result back from a remote computer from the moment the data is sent out from your device. In most cases, inference at the edge results in lower latency because the input doesn't need to be sent out—It can all be done in one place. This is particularly important for the AWS DeepRacer device which needs to make real-time decisions. However, when you create a deep learning model for the AWS DeepRacer device, you have to consider that edge devices are generally less powerful and have less compute power than what is available in the cloud.

To infer data through a deep learning model we need an inference engine. If you think of the inference engine as the car's engine then the deep learning model is the fuel. This combination determines how fast and how efficient the car will run. You want to make sure the fuel (your model) is appropriate for the engine (the device) it is running on.

So, we need to do some more work and optimize our trained model so that it performs well when loaded onto the AWS DeepRacer device. In the next section we are going to learn how to optimize our model with the OpenVINO toolkit and AWS DeepRacer.

## Model optimization with Intel OpenVINO

In the previous section we discussed the need to optimize our model for the computer onboard the AWS DeepRacer device so that we can achieve acceptable inference performance. To do this, OpenVINO toolkit is used in conjunction with the AWS DeepRacer device.

OpenVINO stands for Open Visual Inferencing and Neural Network Optimization and is Intel's developer tool kit for machine learning inference. It contains a series of components and tools which help developers improve the inference performance of their models on Intel hardware, such as the AWS DeepRacer device, which has a compute module with an Intel Atom processor.

There are different kinds of compute hardware available today, such as CPUs and GPUs. You can think of these types of processors as different kinds of automobiles. The CPU might be similar to a family station wagon. It's a great all-around car that's good for different uses, but it probably won't win a track race. Meanwhile, the GPU is like a purpose-built four-wheel drive car. It's great for off-roading, but isn't easy to drive around town. OpenVINO toolkit helps you optimize machine learning models to work on different types and combinations of hardware.

The Intel OpenVINO toolkit has the following components: the Deep Learning Deployment Toolkit (includes the model optimizer), inference engine, and pre-trained models. It also includes advanced tools and libraries for expert programmers.

Using an optimized model and the OpenVINO toolkit inference engine, we can achieve a low inference latency on the Intel Atom processor in the AWS DeepRacer device.

In the next section, we will go into detail about how the optimization process works.

## How does the optimization process work?

There are three steps to the optimization process:

1. Convert and optimize
2. Further tuning for performance (optional)
3. Deploy the model

In the first step, we will run the model optimizer to prepare the pre-trained model for inferencing. It will generate a set of different files, known as an Intermediate Representation (IR), that the OpenVINO toolkit inference engine in AWS DeepRacer can understand and use. The model optimizer performs a number of optimizations on the model to make it run faster and more efficiently on the Intel Atom powered AWS DeepRacer. This is all done automatically when you export your model from the AWS DeepRacer Console, so that in most situations you're done and can jump directly to step three and deploy the model to the AWS DeepRacer device.

However, in some cases, if the performance isn't quite what you are expecting, there are plenty of additional tools included with the OpenVINO toolkit that advanced users can use. Think of these additional tools as you would modifying a car. Most drivers are happy with their cars as they come from the manufacturer. However, enthusiasts may wish to tune their cars or modify them in other ways to make them perform better. Investing the time to learn about and use these tools will allow you to get the best performance out of your AWS DeepRacer device.

Let's take a look at a few of these tools.

The Post-Training Optimization Toolkit (POT) is a tool included with the OpenVINO toolkit, which can be used to make the model more efficient. Think of this tool as a slider with accuracy at one end and performance at the other.

The POT allows you to tune a model for extra performance by reducing the precision of the model and therefore, sacrificing some accuracy. Some Intel processors come with accelerators to turbo charge performance of lower precision models and this is where the POT is particularly handy, because it is similar to putting the right fuel in a car to make sure you get the best performance from the engine.

There are also benchmarking and accuracy checker tools available, which will allow you to get key performance and accuracy data for a model on the target hardware, such as AWS DeepRacer. This data is useful as you go through the process of refining and optimizing your model to understand how it impacts inference performance and accuracy.

When you are done with any of the optional fine-tuning, the optimized model is ready to be uploaded to the AWS DeepRacer device to perform real-time inference using the OpenVINO toolkit inference engine.

## Conclusion

Well done on completing this deep dive into how the AWS DeepRacer device works and the unique challenges associated with running inference at edge. As you have seen, Intel's OpenVINO toolkit works hand-in-hand with AWS DeepRacer to make the real-time inference possible through model optimization. If you would like more information about the OpenVINO toolkit, see [intel.com/openVINO](https://intel.com/openVINO).

## Pro Tips from the Pros

### Pro-tips

Even though AWS DeepRacer Student League only launched in 2022, the AWS DeepRacer League has been running since 2018 - and there have been many racers competing for the top prizes across the past years.

In this section we have brought together some of the top racers from the AWS DeepRacer Pro League to give you some pro tips about how to approach the League competition and training your models.

#### *Ross Williams*

- The more vague your reward function is the longer it is going to take to train. It is not always a bad thing but it could just take a very long time.
- Small Changes can have Big impact (try small incremental changes)

#### *Henry Gotzinger*

- Be careful while making changes to the reward function. Drastic Changes can lead to instability
- Sometimes straightforward reward function is better
- Consistent Naming scheme.

#### *James Jennes*

- Select something that is based on progress the one downside to that is it can take longer to train.
- Pointing the staring at a point on centre line

#### *Robert Nuido*

- Make things simple.
- If you want to try complex things it is better to optimize them first making them easier to understand and then make little changes.

### *Roger Logan*

- You should reward your function often.
- Rewards should be constant and forever and Punishment should be temporary.
- Use waypoint to cut the corner.

### *Lars Lorentz Ludvigsen*

- To give the car forward looking abilities the model is setup to maximize not the single step reward but the sum of all future rewards.
- Discounted with the discounts. A high discount rate like 0.999 means almost all future rewards counts , a small discount rate like 0.9 means only the next few steps count.
- Single lap in deep racer can take 10-30 seconds. Imagine you complete a lap in 10 seconds that is 150 steps , 150 chances to give reward
- IN Deep Racer student the discount rate is fixed to 0.999. Since all the setps count the car might be inclined to take the slow route to get high reward in total
- As a rule of thumb fast laps should get higher reward than slow laps.

### *Jochem Lugtenburg*

- Make use of python classes in your reward function.
- Wrap you r reward function is class which will enable you to make use of information from previous steps. Compare steps and reward.

### *Andy Milluzi*

- Look at logs , look at what is being processed and look at what decisions your racer is making.

## **Wrap Up**

Congratulations on completing the Reinforcement Learning with AWS DeepRacer course! Hopefully you now have a better understanding of reinforcement learning and how this can be applied to AWS DeepRacer.

Keep checking back as new content will be added to this course regularly, covering more advanced topics and concepts.