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

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.

Summary

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.

Traffic signaling

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.

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.

However, before we get started we'll need to look more closely at how reinforcement learning works.

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

In the previous section we discussed policies in reinforcement learning, particularly deterministic policies and stochastic policies. 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

Now that we have some understanding of how machine learning algorithms work, particularly policies, 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.

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.

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.

Training an AWS DeepRacer model

Well done on making it this far! You should now have a good understanding of the fundamentals of reinforcement learning. This lesson will cover training your very first DeepRacer model.

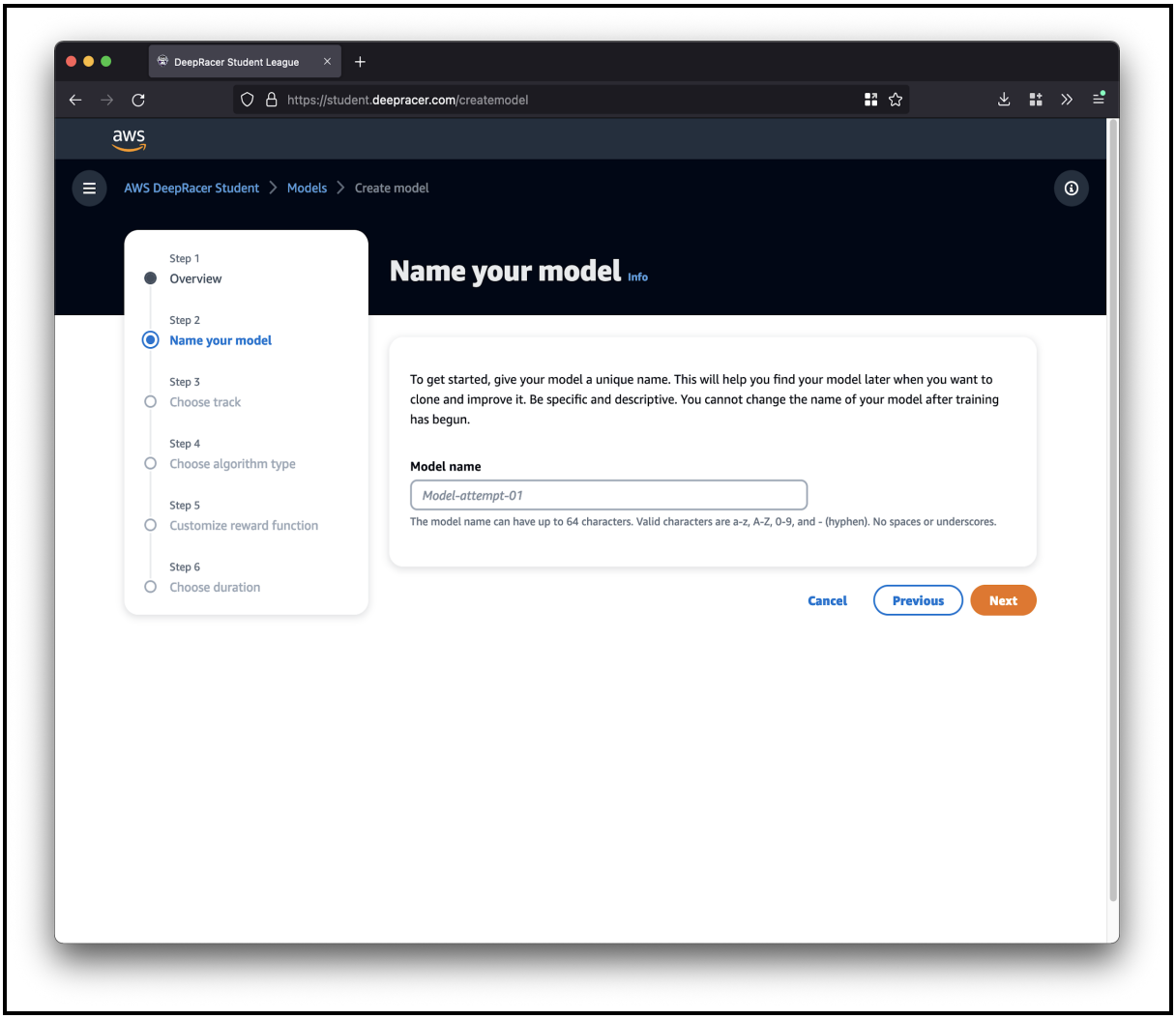
Summary

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.

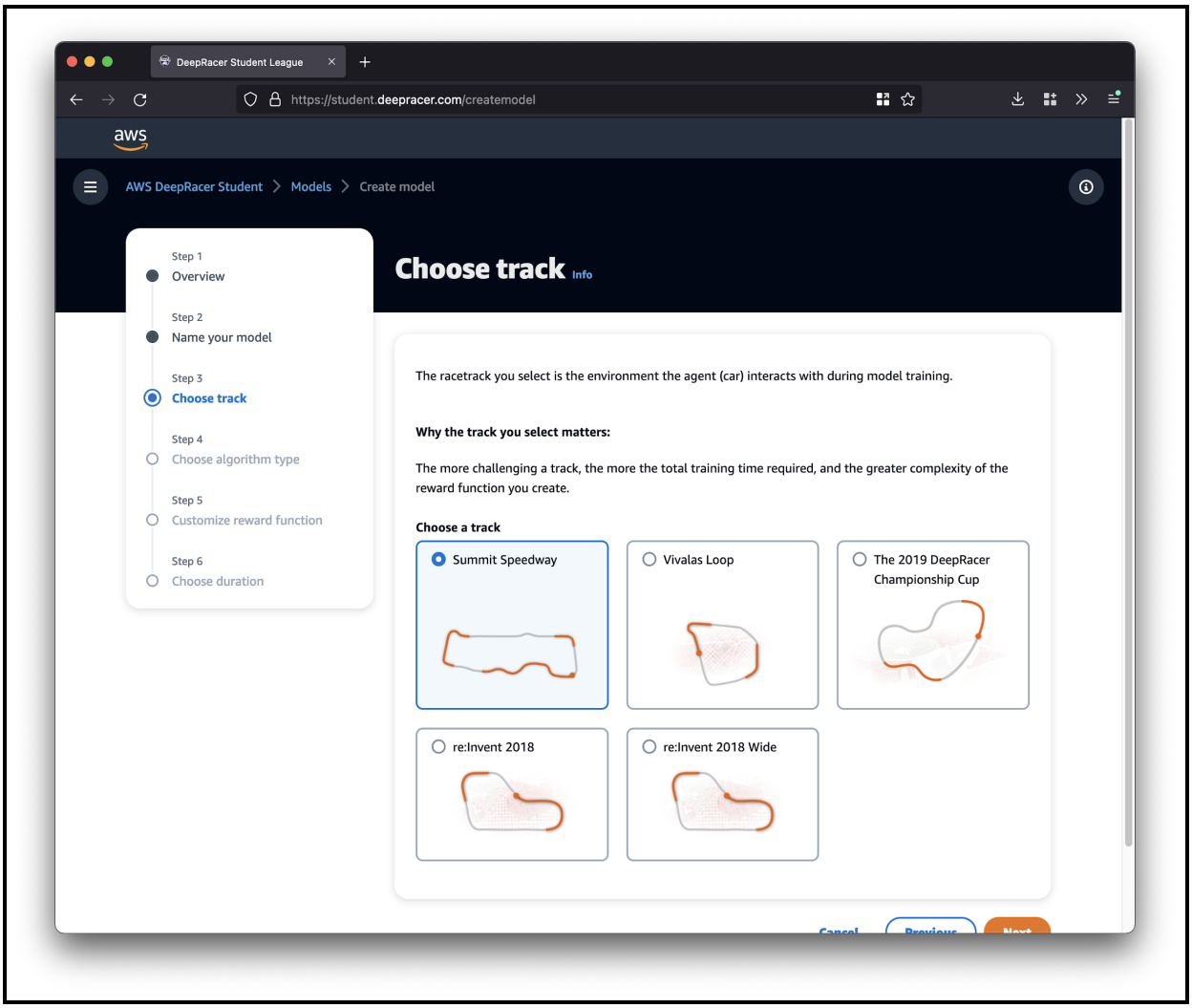
Name your model



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.

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.

Choose track

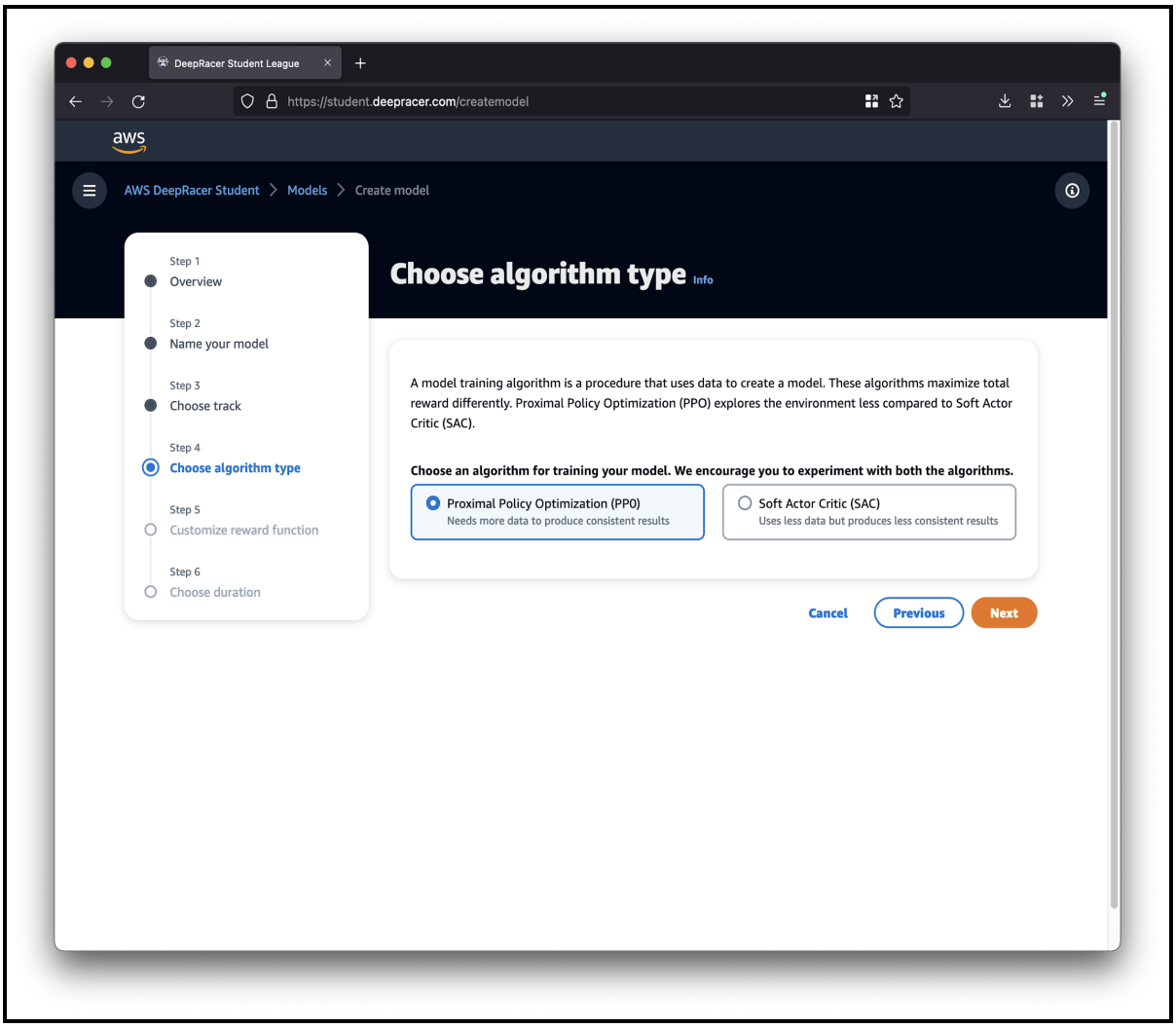


Now you need to select the track to use for training your model.

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.

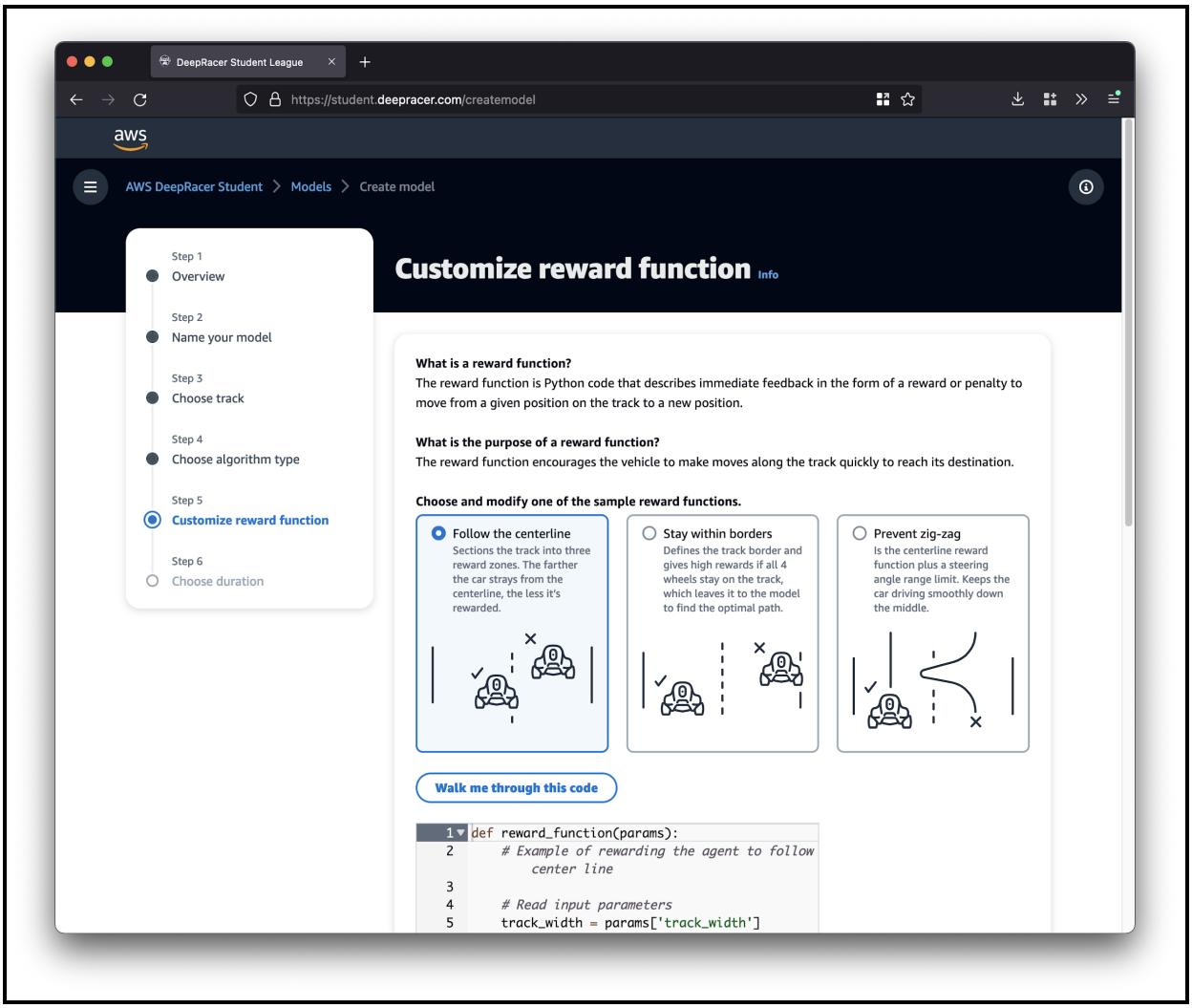
Choose algorithm type



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.

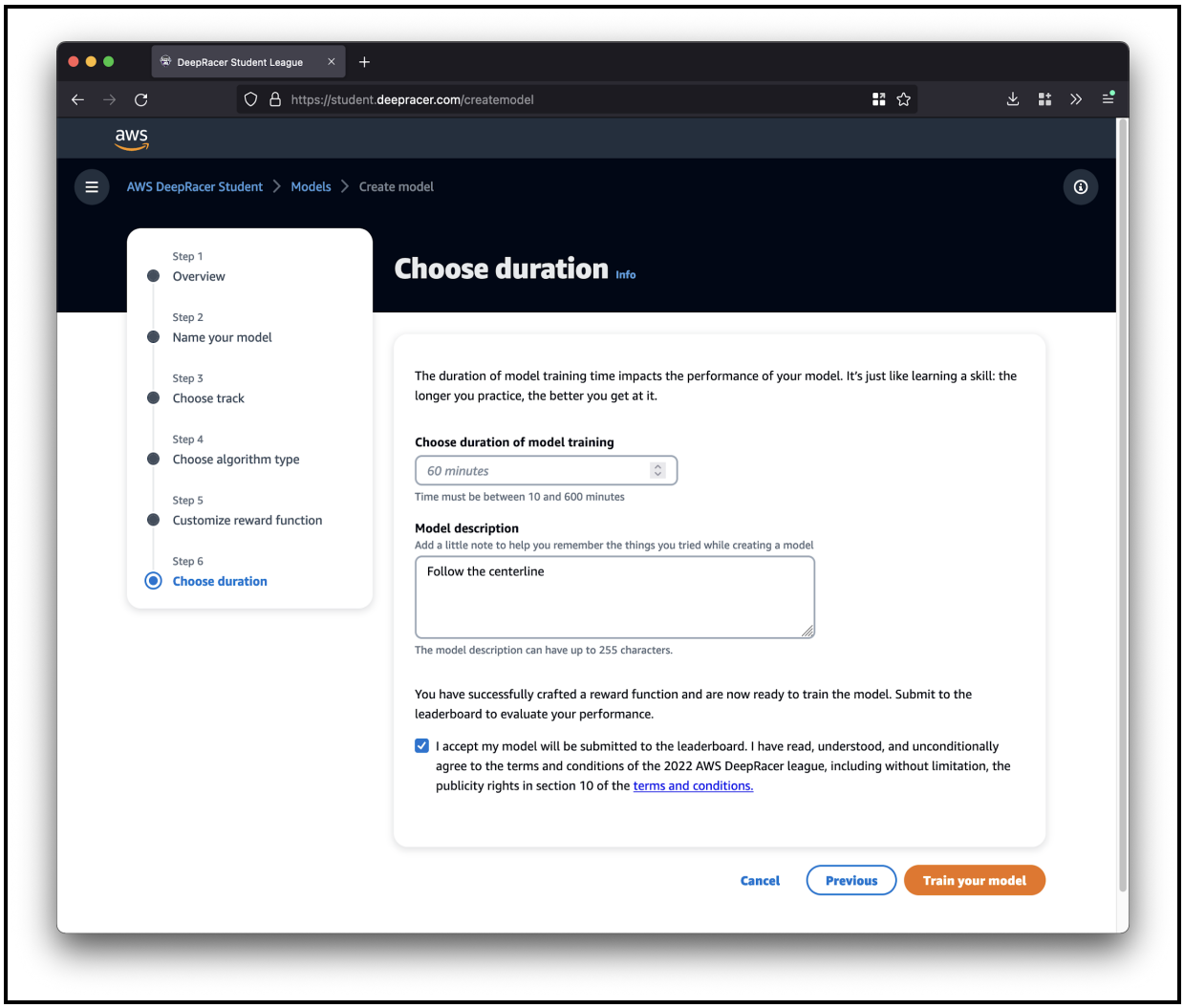
Customize reward function



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.

Choose duration

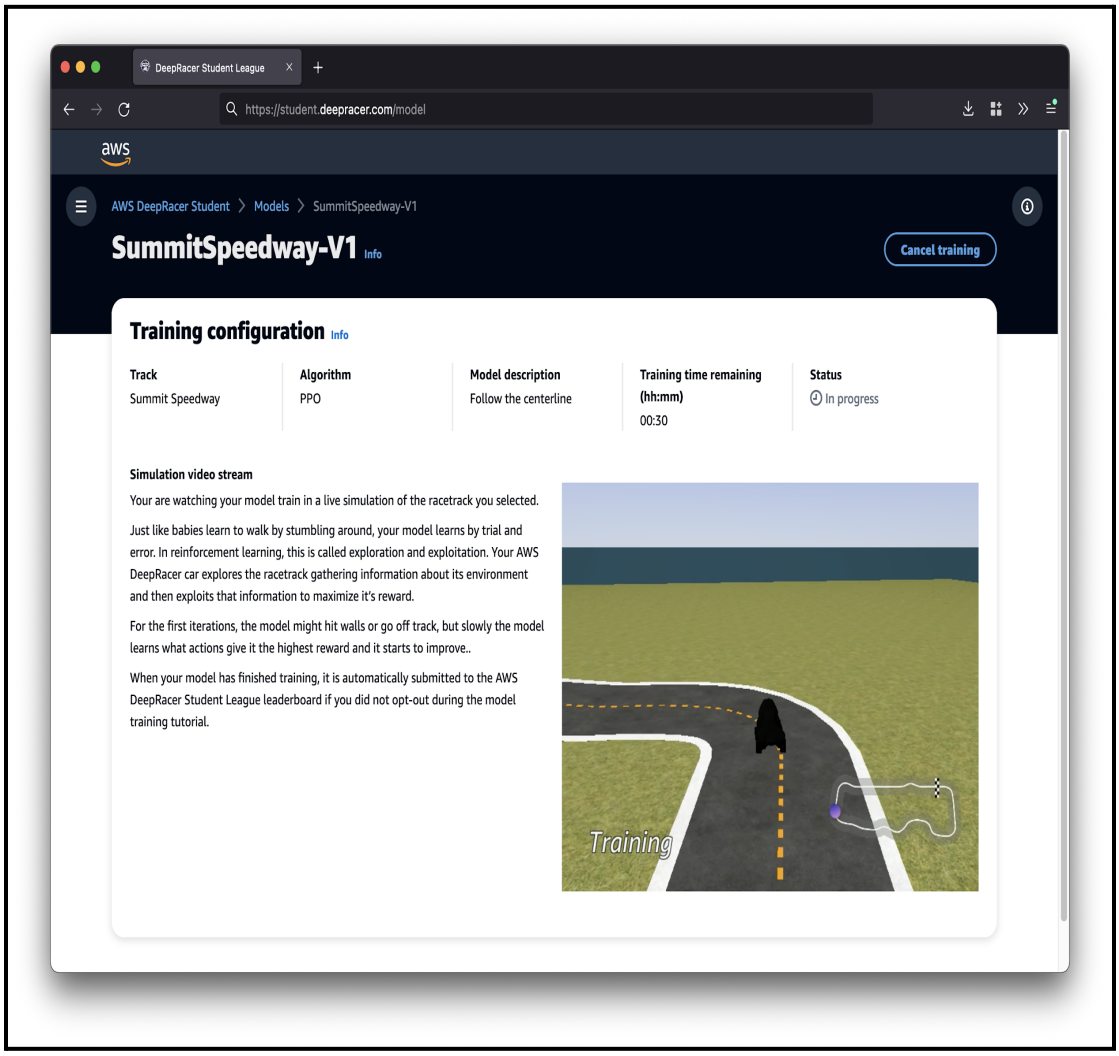


Finally, you can configure the options for model training. I suggest you start by doing 60 minutes of training, and give your model a description so you can find it later on (such as a quick summary of the chosen algorithm along with reward function).

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.

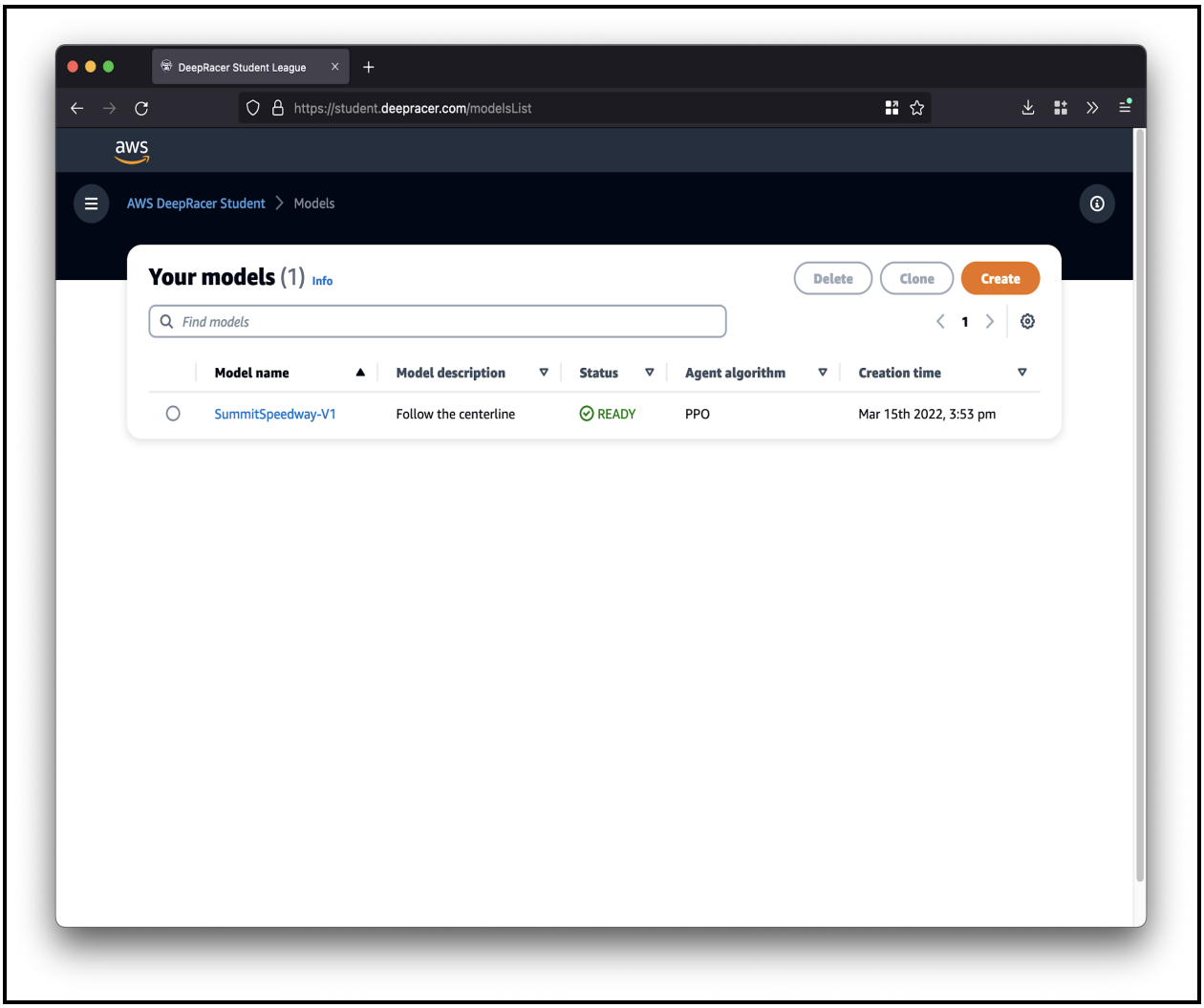
When you are ready to start training, select the "Train your model" button.

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.

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.

However, before we get started we'll need to look more closely at how reinforcement learning works.

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

In the previous section we discussed policies in reinforcement learning, particularly deterministic policies and stochastic policies. 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

Now that we have some understanding of how machine learning algorithms work, particularly policies, 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.

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.

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.

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.

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.