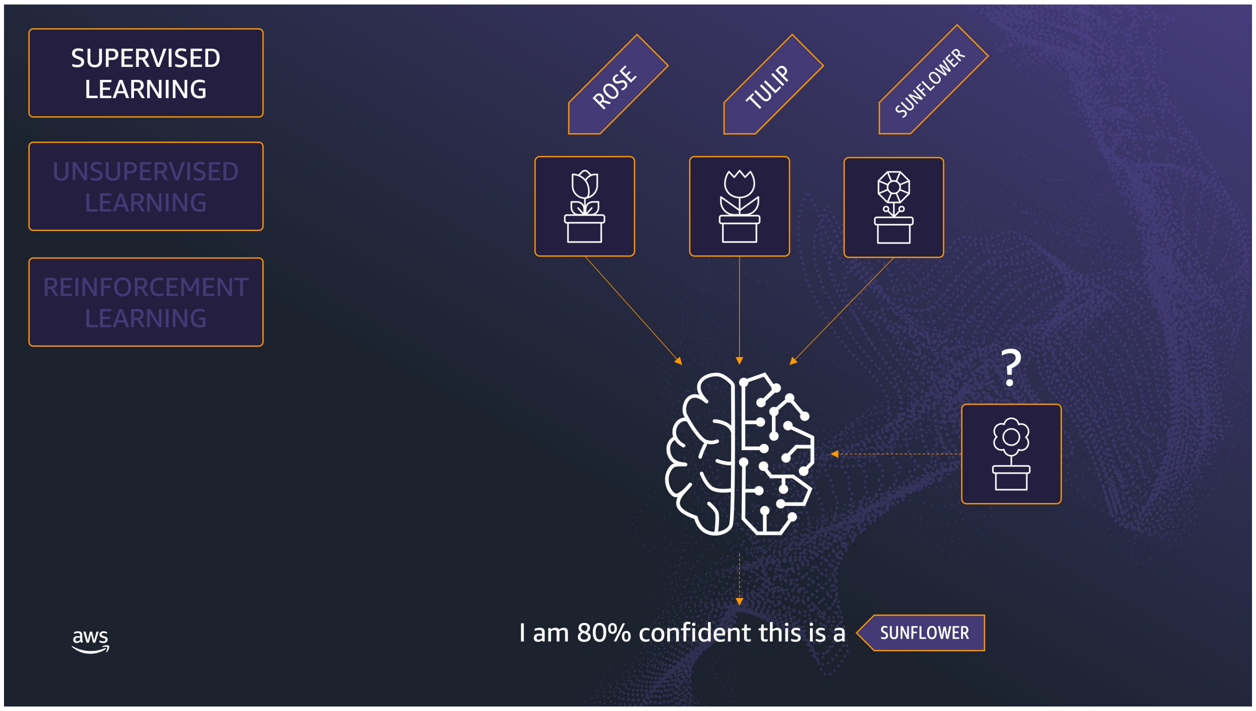
AWS DEEPRACER RL NOTES

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.

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.
Supervised learning



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.

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.
Unsupervised learning

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 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.
Reinforcement learning

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.

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 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.

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.

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.

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.

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.

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.

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.

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 https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-reward-function-input.html

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.

HIGHLY IMPORTANT GUIDE FOR THE DEEPRACER COMPETITIONS

https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-reward-function-input.html

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.

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.

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.

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.