NOTES 2

Glossary

- **Action:** For every state, an agent needs to take an action toward achieving its goal.
- **Agent:** The piece of software you are training is called an agent. It makes decisions in an environment to reach a goal.
- **Discriminator**: A neural network trained to differentiate between real and synthetic data.
- **Discriminator loss:** Evaluates how well the discriminator differentiates between real and fake data.
- **Edit event:** When a note is either added or removed from your input track during inference.
- **Environment:** The environment is the surrounding area within which the agent interacts.
- **Exploration versus exploitation:** An agent should exploit known information from previous experiences to achieve higher
  cumulative rewards, but it also needs to explore to gain new experiences that can be used in choosing the best actions in the future.
- **Generator:** A neural network that learns to create new data resembling the source data on which it was trained.
- **Generator loss:** Measures how far the output data deviates from the real data present in the training dataset.
- **Hidden layer:** A layer that occurs between the *output* and *input* layers. Hidden layers are tailored to a specific task.
- **Input layer:** The first layer in a neural network. This layer receives all data that passes through the neural network.
- **Output layer:** The last layer in a neural network.
  This layer is where the predictions are generated based on the
  information captured in the hidden layers.

- **Piano roll:** A two-dimensional piano roll matrix that represents input tracks. Time is on the horizontal axis and pitch is on the vertical axis.
- **Reward:** Feedback is given to an agent for each action it takes in a given state. This feedback is a numerical reward.

Let’s take a look at some examples.

- RL is great at **playing games**:
  - **Go** (board game) was mastered by the AlphaGo Zero software.
  - **Atari classic video** games are commonly used as a learning tool for creating and testing RL software.
  - **StarCraft II,** the real-time strategy video game, was mastered by the AlphaStar software.
- RL is used in **video game level design**:
  - Video game level design determines how complex each stage of a game is and directly affects how boring, frustrating, or fun it is to play that game.
  - Video game companies create an agent that plays the game over and over again to collect data that can be visualized on graphs.
  - This visual data gives designers a quick way to assess how easy or difficult it is for a player to make progress, which enables them to find that “just right” balance between boredom and frustration faster.
- RL is used in **wind energy optimization**:
  - RL models can also be used to power robotics in physical devices.
  - When multiple turbines work together in a wind farm, the turbines in the front, which receive the wind first, can cause poor wind conditions for the turbines behind them. This is called **wake turbulence** and it reduces the amount of energy that is captured and converted into electrical power.

    - Wind energy organizations around the world use reinforcement learning to test solutions. Their models respond to changing wind conditions by changing the angle of the turbine blades. When the upstream turbines slow down it helps the downstream turbines capture more energy.
- Other examples of real-world RL include:
    - **Industrial robotics**
    - **Fraud detection**
    - **Stock trading**
    - **Autonomous driving**

Some examples of real-world RL include: Industrial robotics, fraud detection, stock trading, and autonomous driving

Agent

- The piece of software you are training is called an agent.
- It makes decisions in an environment to reach a goal.
- In AWS DeepRacer, the agent is the AWS DeepRacer car and its goal is to finish * laps around the track as fast as it can while, in some cases, avoiding obstacles.

Environment

- The environment is the surrounding area within which our agent interacts.

- For AWS DeepRacer, this is a track in our simulator or in real life.

State

- The state is defined by the current position within the environment that is visible, or known, to an agent.
- In AWS DeepRacer's case, each state is an image captured by its camera.
- The car's initial state is the starting line of the track and its terminal state is when the car finishes a lap, bumps into an obstacle, or drives off the track.

Action

- For every state, an agent needs to take an action toward achieving its goal.
- An AWS DeepRacer car approaching a turn can choose to accelerate or brake and turn left, right, or go straight.

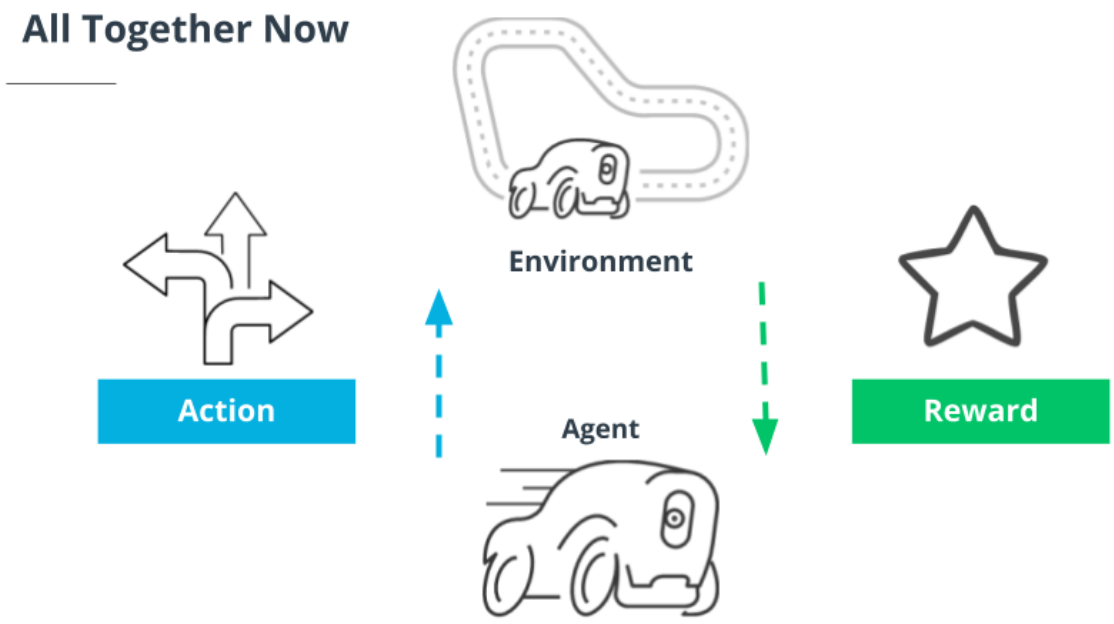
Reward

- Feedback is given to an agent for each action it takes in a given state.
- This feedback is a numerical reward.
- A reward function is an incentive plan that assigns scores as rewards to different zones on the track.

Episode

- An episode represents a period of trial and error when an agent makes decisions and gets feedback from its environment.
- For AWS DeepRacer, an episode begins at the initial state, when the car leaves the starting position, and ends at the terminal state, when it finishes a lap, bumps into an obstacle, or drives off the track.

In a reinforcement learning model, an **agent** learns in an interactive real-time **environment** by trial and error using feedback from its own **actions**. Feedback is given in the form of **rewards**.



In
 a reinforcement learning model, an agent learns in an interactive real-time environment by trial and error using feedback from its own actions. Feedback is given in the form of rewards.

Putting Your Spin on AWS DeepRacer: The Practitioner's Role in RL

Summary

AWS DeepRacer may be autonomous, but you still have an important role
 to play in the success of your model. In this section, we introduce the **training algorithm, action space, hyperparameters,** and **reward function** and discuss how your ideas make a difference.

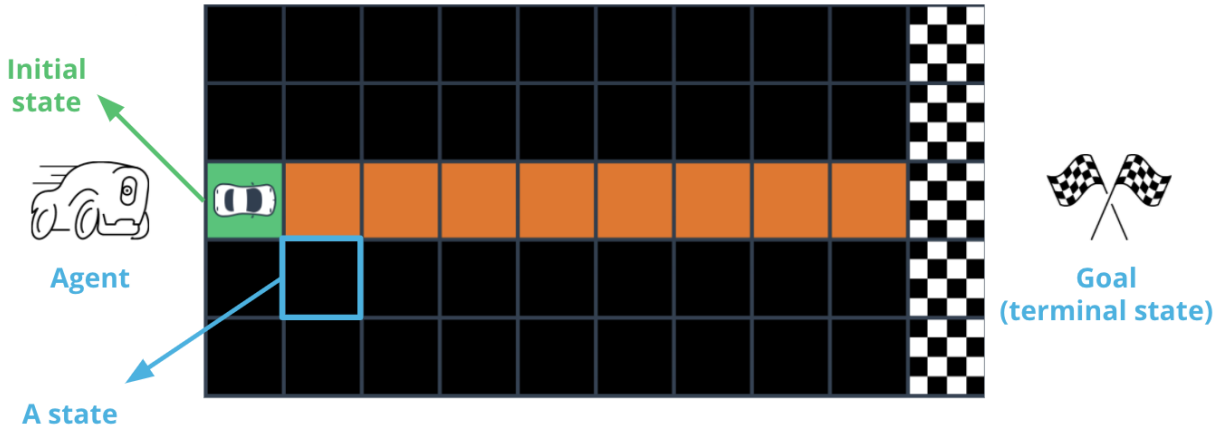
- An *algorithm* is a set of instructions that tells a computer what to do. ML is special because it enables computers to learn without being explicitly programmed to do so.
- The *training algorithm* defines your model's learning objective, which is to maximize total cumulative reward. Different algorithms have different strategies for going about this.

    - A *soft actor critic (SAC)* embraces exploration and is data-efficient, but can lack stability.
    - A *proximal policy optimization* (PPO) is stable but data-hungry.
- An *action space* is the set of all valid actions, or choices, available to an agent as it interacts with an environment.
    - *Discrete action space* represents all of an agent's possible actions for each state in a finite set of steering angle and throttle value combinations.
    - *Continuous action space* allows the agent to select an action from a range of values that you define for each sta te.
- *Hyperparameters* are variables that control the performance of your agent during training. There is a variety of different categories with which to experiment. Change the values to increase or decrease the influence of different parts of your model.
    - For example, the *learning rate* is a hyperparameter that controls how many new experiences are counted in learning at each step. A higher learning rate results in faster training but may reduce the model’s quality.
- The *reward function*'s purpose is to encourage the agent to reach its goal. Figuring out how to reward which actions is one of your most important jobs.

Putting Reinforcement Learning into Action with AWS DeepRacer

Summary

This video put the concepts we've learned into action by imagining the reward function as a grid mapped over the race track in AWS DeepRacer’s training environment, and visualizing it as metrics plotted on a graph. It also introduced the trade-off between exploration and exploitation, an important challenge unique to this type of machine learning.



Imagine the **reward function** as a grid mapped over the track with each square representing a **state**.

Each
square is a state. The green square is the starting position, or initial state, and the finish line is the goal, or terminal state.

Key points to remember about **reward functions:**

- Each state on the grid is assigned a score by your reward function. You incentivize behavior that supports your car's goal of completing fast laps by giving the highest numbers to the parts of the track on which you want it to drive.
- The reward function is the actual code you'll write to help your agent determine if the action it just took was good or bad, and how good or bad it was.

These **rewards** incentivize center-line driving.

The squares containing exes are the track edges and defined as terminal states, which tell your car it has gone off track.

Key points to remember about **exploration versus exploitation:**

- When a car first starts out, it *explores* by wandering in random directions. However, the more training an agent gets, the more it learns about an environment. This experience helps it become more confident about the actions it chooses.
- *Exploitation* means the car begins to exploit or use information from previous experiences to help it reach its goal. Different training algorithms utilize exploration and exploitation differently.

Key points to remember about the **reward graph:**

- While training your car in the AWS DeepRacer console, your training metrics are displayed on a *reward graph*.
- Plotting the total reward from each episode allows you to see how the model performs over time. The more reward your car gets, the better your model performs.

Key points to remember about **AWS DeepRacer:**

- AWS DeepRacer is a combination of a physical car and a virtual simulator in the AWS Console, the AWS DeepRacer League, and community races.
- An AWS DeepRacer device is not required to start learning: you can start now in the AWS console. The 3D simulator in the AWS console is where training and evaluation take place.

New Terms

- **Exploration versus exploitation:** An agent should exploit known information from previous experiences to achieve higher cumulative rewards, but it also needs to explore to gain new experiences that can be used in choosing the best actions in the future.

Additional Reading

- If you are interested in more tips, workshops, classes, and other resources for improving your model, you'll find a wealth of resources on the AWS DeepRacer Pit Stop page.
- For detailed step-by-step instructions and troubleshooting support, see the AWS DeepRacer Developer Documentation.
- If you're interested in reading more posts on a range of DeepRacer topics as well as staying up to date on the newest releases, check out the AWS Discussion Forums.
- If you're interested in connecting with a thriving global community of reinforcement learning racing enthusiasts, join the AWS DeepRacer Slack community.
- If you're interested in tinkering with DeepRacer's open-source device software and collaborating with robotics innovators, check out our AWS DeepRacer GitHub Organization.

Exercise: Interpret the reward graph of your first AWS DeepRacer model

Instructions

Train a model in the AWS DeepRacer console and interpret its reward graph.

Part 1: Train a reinforcement learning model using the AWS DeepRacer console

Practice the knowledge you've learned by training your first 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 **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.
4. On the **Create model** page, choose **Next**.
5. 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.
6. On the **Create model** page, under **Training algorithm and hyperparameters**, choose the **Soft Actor Critic (SAC)** or **Proximal Policy Optimization (PPO)** algorithm. In the AWS DeepRacer console, SAC models must be trained in *continuous action spaces*. PPO models can be trained in either *continuous* or *discrete action spaces*.
7. On the **Create model** page, under **Training algorithm and hyperparameters**, use the default hyperparameter values as is. Later on, to improve training performance, expand the hyperparameters and experiment with modifying the default hyperparameter values.
8. On the **Create model** page, under **Agent**, choose **The Original DeepRacer** or **The Original DeepRacer (continuous action space)** for your first model. If you use Soft Actor Critic (SAC) as your training algorithm, we filter your cars so that you can conveniently choose from a selection of compatible continuous action space agents.
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.

11. 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.
12. On the **Create model** page, choose **Create mode**l to start creating the model and provisioning the training job instance.
13. 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**.
14. 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.

Note: 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.

Part 2: Inspect your reward graph to assess your training progress

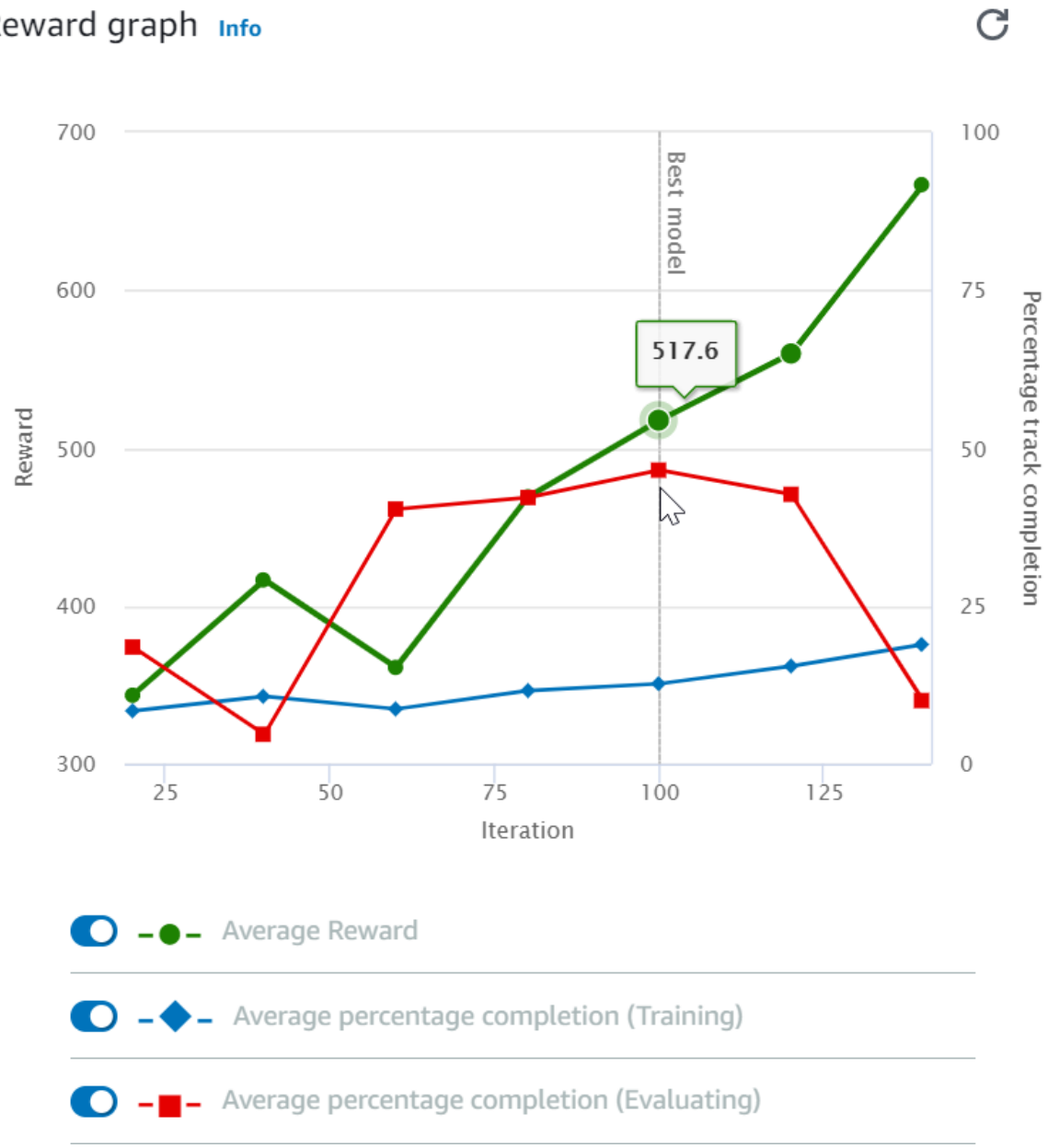
As you train and evaluate your first model, you'll want to get a sense of its quality. To do this, inspect your reward graph.

Find the following on your reward graph:

- Average reward
- Average percentage completion (training)
- Average percentage completion (evaluation)
- Best model line
- Reward primary y-axis
- Percentage track completion secondary y-axis

- Iteration x-axis

Review the solution to this exercise for ideas on how to interpret it.

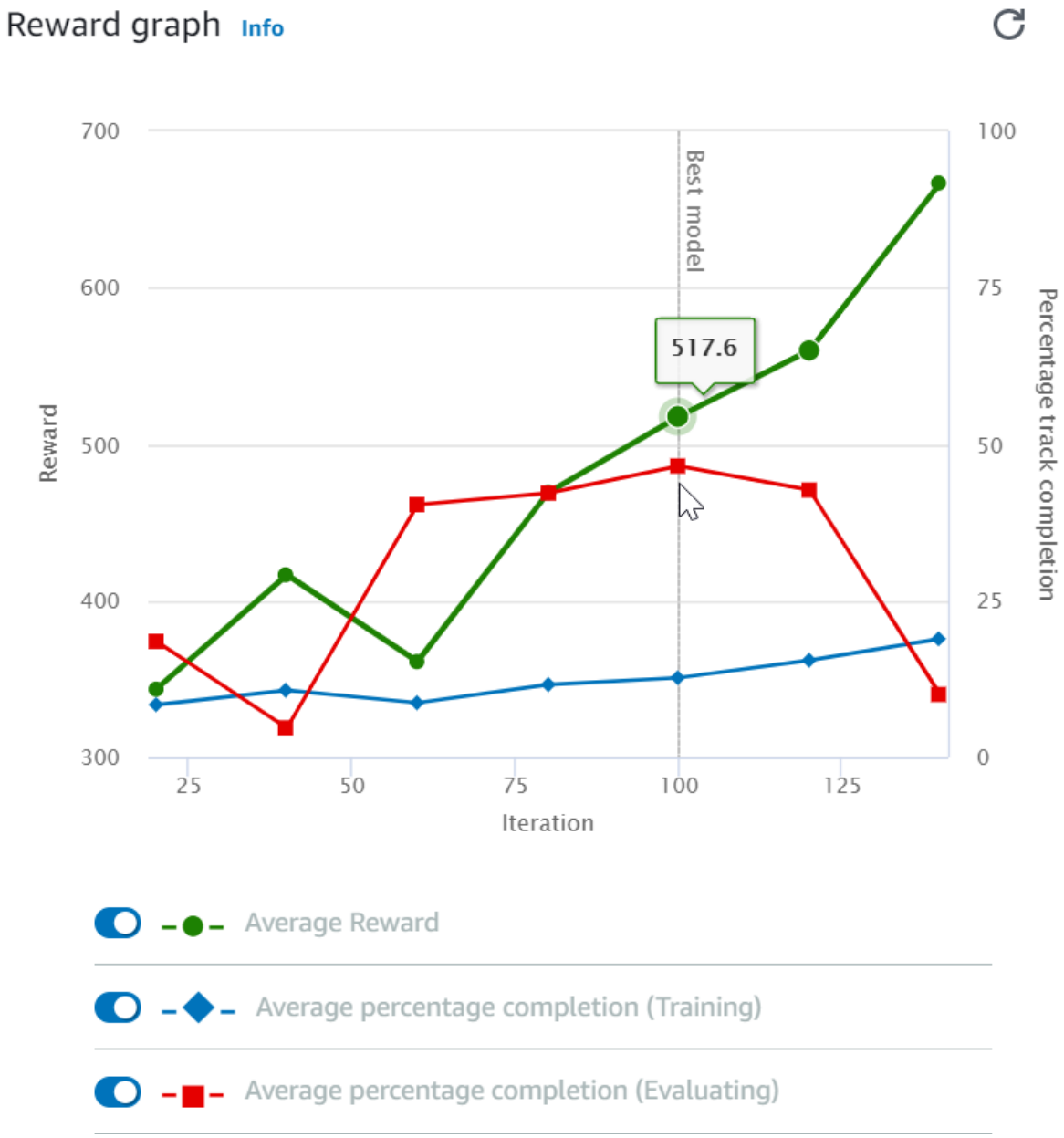


As you train and evaluate your first model, you'll want to get a sense of its quality. To do this, inspect your reward graph.

To get a sense of how well your training is going, watch the reward graph. Here is a list of its parts and what they do:

- **Average reward**
  - This graph represents the average reward the agent earns during a training iteration. The average is calculated by averaging the reward earned across all episodes in the training iteration. An episode begins at the starting line and ends when the agent completes one loop around the track or at the place the vehicle left the track or collided with an object. Toggle the switch to hide this data.
- **Average percentage completion (training)**
  - The training graph represents the average percentage of the track completed by the agent in all training episodes in the current training. It shows the performance of the vehicle while experience is being gathered.
- **Average percentage completion (evaluation)**
  - While the model is being updated, the performance of the existing model is evaluated. The evaluation graph line is the average percentage of the track completed by the agent in all episodes run during the evaluation period.
- **Best model line**
  - This line allows you to see which of your model iterations had the highest average progress during the evaluation. The checkpoint for this iteration will be stored. A checkpoint is a snapshot of a model that is captured after each training (policy-updating) iteration.
- **Reward primary y-axis**
  - This shows the reward earned during a training iteration. To read the exact value of a reward, hover your mouse over the data point on the graph.
- **Percentage track completion secondary y-axis**
  - This shows you the percentage of the track the agent completed during a training iteration.

- **Iteration x-axis**
  - This shows the number of iterations completed during your training job.



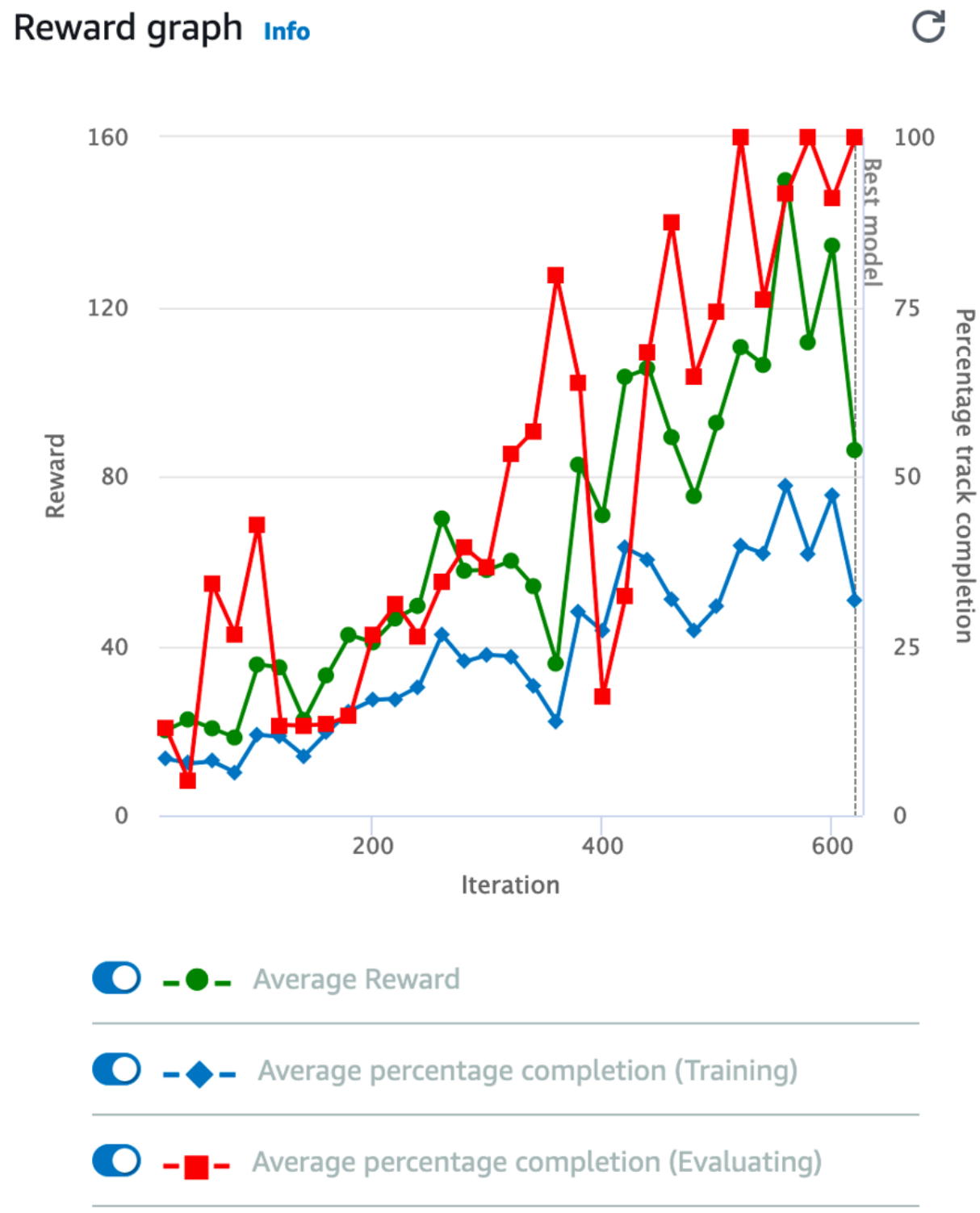
List of reward graph parts and what they do

Reward Graph Interpretation

The following four examples give you a sense of how to interpret the success of your model based on the reward graph. Learning to read these graphs is as much of an art as it is a science and takes time, but reviewing the following four examples will give you a start.

Needs more training

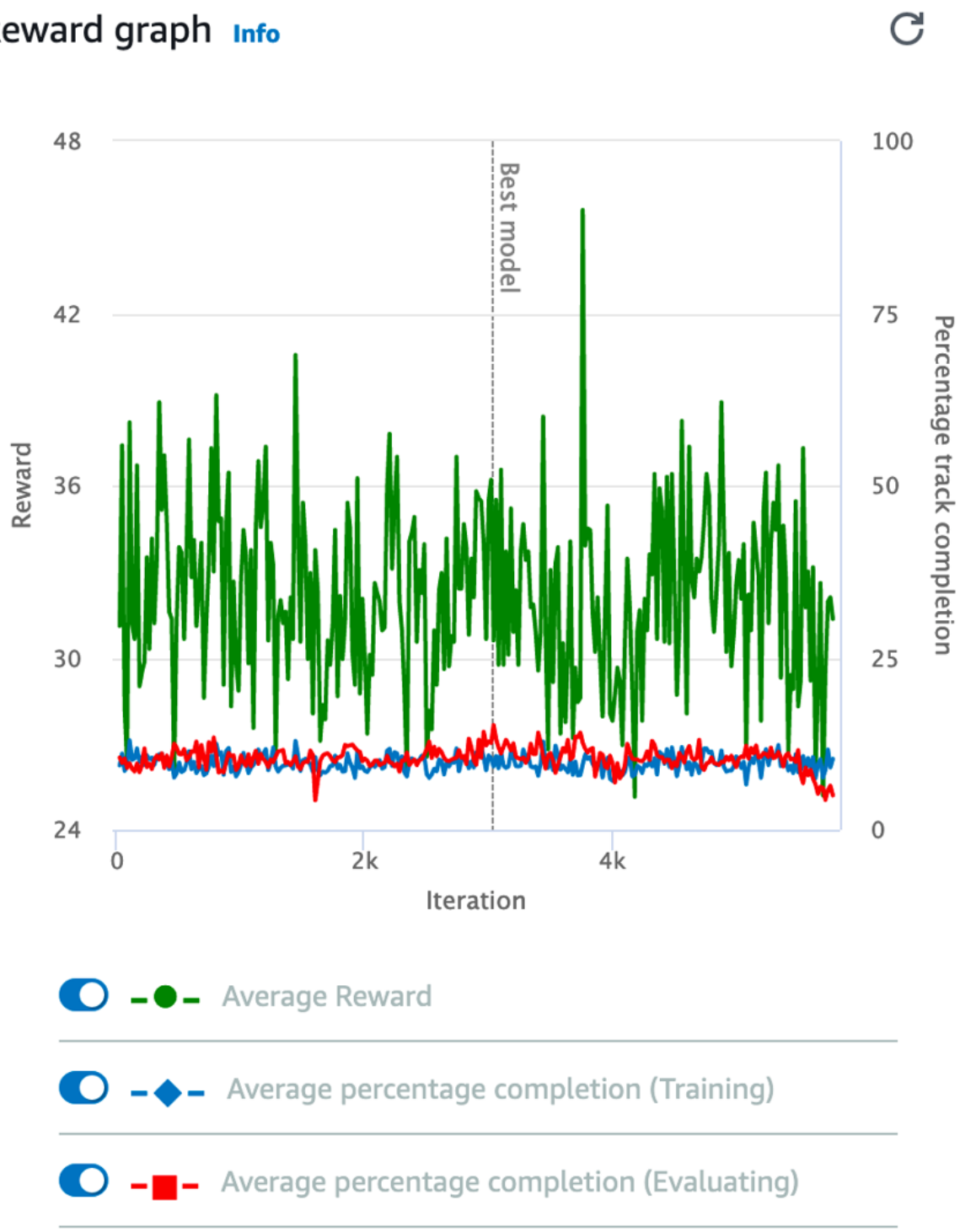
In the following example, we see there have only been 600 iterations, and the graphs are still going up. We see the evaluation completion percentage has just reached 100%, which is a good sign but isn't fully consistent yet, and the training completion graph still has a ways to go. This reward function and model are showing promise, but need more training time.



Needs more training

No improvement

In the next example, we can see that the percentage of track completions haven’t gone above around 15 percent and it's been training for quite some time—probably around 6000 iterations or so. This is not a good sign! Consider throwing this model and reward function away and trying a different strategy.



No improvement

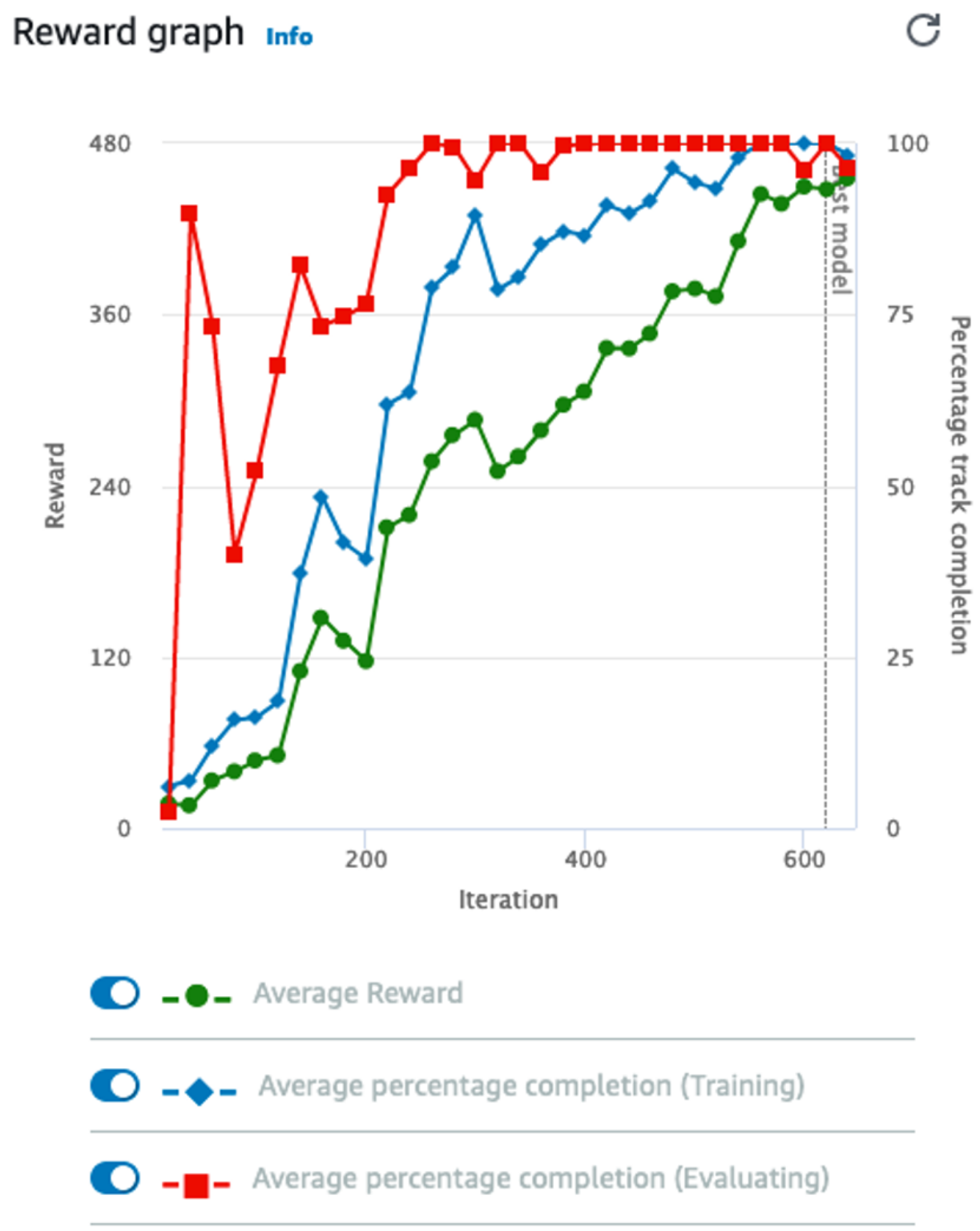
A well-trained model

In the following example graph, we see the evaluation percentage completion reached 100% a while ago, and the training percentage reached 100% roughly 100 or so iterations ago. At this point, the model is well trained. Training it further might lead to the model becoming overfit to this track.

Avoid overfitting

Overfitting or **overtraining** is a really important concept in machine learning. With AWS DeepRacer, this can become an issue when a model is trained on a specific track for too long. A good model should be able to make decisions based on the features of the road, such as the sidelines and centerlines, and be able to drive on just about any track.

An **overtrained** model, on the other hand, learns to navigate using landmarks specific to an individual track. For example, the agent turns a certain direction when it sees uniquely shaped grass in the background or a specific angle the corner of the wall makes. The resulting model will run beautifully on that specific track, but perform badly on a different virtual track, or even on the same track in a physical environment due to slight variations in angles, textures, and lighting.

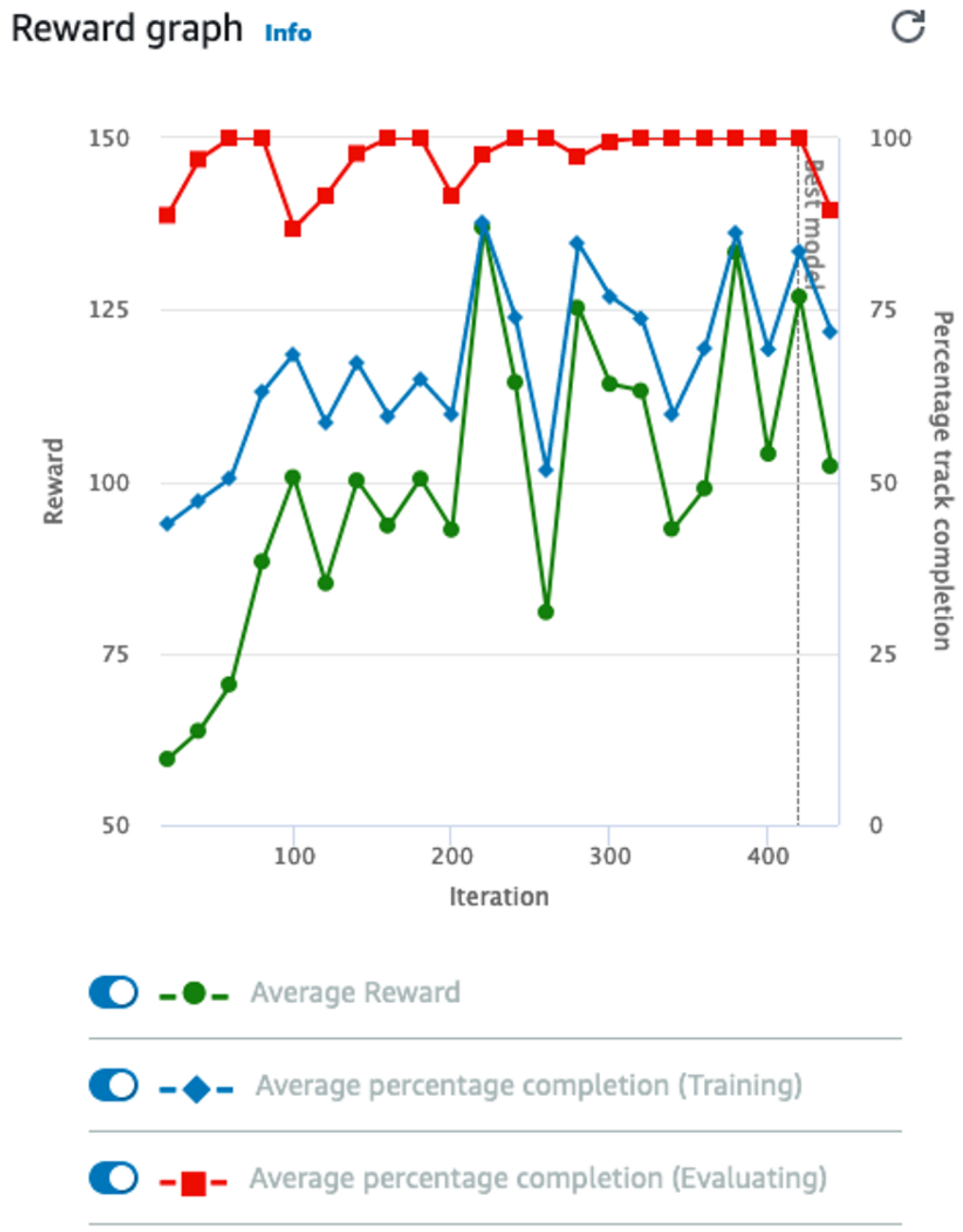


Well-trained - Avoid overfitting

Adjust hyperparameters

The AWS DeepRacer console's default **hyperparameters** are quite effective, but occasionally you may consider adjusting the training hyperparameters. The **hyperparameters** are variables that essentially act as settings for the training algorithm that control the performance of your agent during training. We learned, for example, that the **learning rate** controls how many new experiences are counted in learning at each step.

In this reward graph example, the training completion graph and the reward graph are swinging high and low. This might suggest an inability to converge, which may be helped by adjusting the learning rate. Imagine if the current weight for a given node is .03, and the optimal weight should be .035, but your learning rate was set to .01. The next training iteration would then swing past optimal to .04, and the following iteration would swing under it to .03 again. If you suspect this, you can reduce the learning rate to .001. A lower learning rate makes learning take longer but can help increase the quality of your model.



Adjust hyperparameters

Good Job and Good Luck!

Remember: training experience helps both model and reinforcement learning practitioners become a better team. Enter your model in the monthly AWS DeepRacer League races for chances to win prizes and glory while improving your machine learning development skills!

Generative AI Models

In this lesson, you will learn how to create three popular types of generative models: **generative adversarial networks (GANs), general autoregressive models,** and **transformer-based models**.

Each of these is accessible through AWS DeepComposer to give you hands-on experience with using these techniques to generate new examples of music.

Autoregressive models

Autoregressive convolutional neural networks (AR-CNNs) are used to study systems that evolve over time and assume that the likelihood of some data depends only on what has happened in the past. It's a useful way of looking at many systems, from weather prediction to stock prediction.

Generative adversarial networks (GANs)

Generative adversarial networks (GANs), are a machine learning model format that involves pitting two networks against each other to generate new content. The training algorithm swaps back and forth between training a *generator network* (responsible for producing new data) and a *discriminator network* (responsible for measuring how closely the generator network's data represents the training dataset).

Transformer-based models

Transformer-based models are most often used to study data with some sequential structure (such as the sequence of words in a sentence). Transformer-based methods are now a common modern tool for modeling natural language.

We won't cover this approach in this course but you can learn more about transformers and how AWS DeepComposer uses transformers in AWS DeepComposer learning capsules.