AMAZON WEB SERVICE

AWS DEEPRACER STUDENT

TRAINING A REINFORCEMENT LEARNING MODEL

Reinforcement Learning

Reinforcement learning problems involve learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. In an essential way they are closed-loop problems because the learning system's actions influence its later inputs.

The three most important distinguishing features of reinforcement learning problems- being closed-loop in an essential way, not having direct instructions as to what actions to take, and where the consequences of actions, including reward signals. We therefore consider reinforcement learning to be a third machine learning paradigm, alongside of supervised learning, unsupervised learning, and perhaps other paradigms as well.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation.Another key feature of reinforcement learning is that it explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment.

Reinforcement learning takes the opposite tack, starting with a complete, interactive, goal-seeking agent. When reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environment models are acquired and improved. When reinforcement learning involves supervised learning, it does so for specific reasons that determine which capabilities are critical and which are not.

Reinforcement Learning

Reinforcement learning is part of a decades-long trend within artificial intelligence and machine learning toward greater integration with statistics, optimization, and other mathematical subjects. Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems.

All involve interaction between an active decision-making agent and its environment, within which the agent seeks to achieve a goal despite un- certainty about its environment. At the same time,the agent must monitor its environment frequently and react appropriately.

Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a policy, a reward signal, a value function, and, optionally, a model of the environment.

1. A policy defines the learning agent's way of behaving at a given time.
2. A reward signal defines the goal in a reinforcement learning problem.
3. Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run.
4. The fourth and final element of some reinforcement learning systems is a model of the environment.

Types of Reinforcement learning

o **Positive –**

Positive Reinforcement is defined as when an event, occurs due to a particular behaviour, increases the strength and the frequency of the behaviour. In other words, it has a positive effect on behavior.

Advantages of reinforcement learning are:

- Maximizes Performance
- Sustain Change for a long period of time
- Too much Reinforcement can lead to an overload of states which can diminish the results

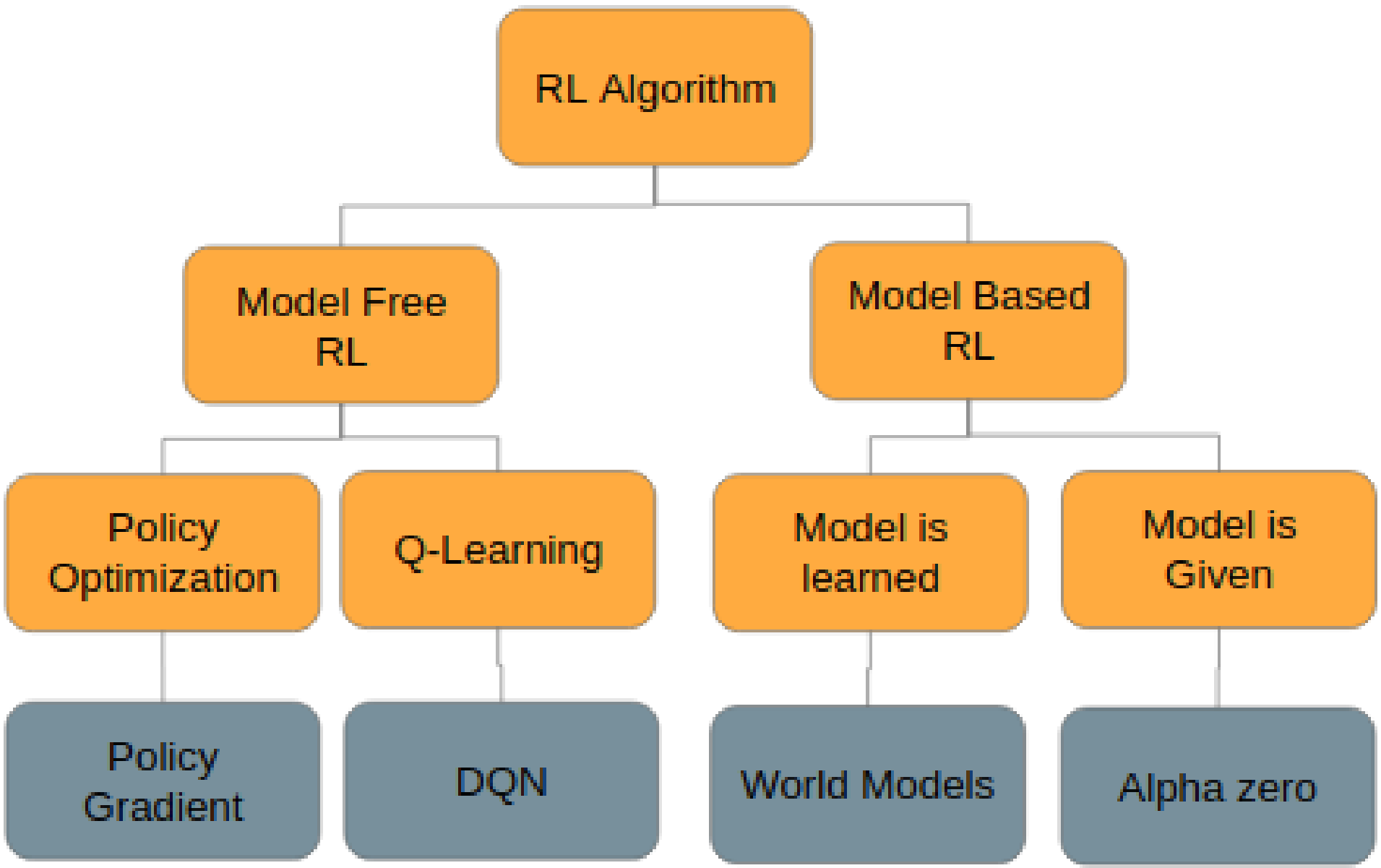
o **Negative –**

Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

Advantages of reinforcement learning:

- Increases Behavior
- Provide defiance to a minimum standard of performance
- It Only provides enough to meet up the minimum behavior

Types of Reinforcement learning



Comparison of Reinforcement learning Algorithms

Algorithm	Description	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Either	Discrete	Discrete	Sample-means
Q-learning	State–action–reward–state	Off-policy	Discrete	Discrete	Q-value
SARSA	State–action–reward–state–action	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State–action–reward–state with eligibility traces	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State–action–reward–state–action with eligibility traces	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	On-policy	Continuous	Continuous	Advantage
NAF	Q-Learning with Normalized Advantage Functions	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Off-policy	Continuous	Continuous	Advantage

Markov Process

o **Markov process:**

We start with the definition of a Markov process. A sequence of states is Markov if and only if the probability of moving to the next state St+1 depends only on the present state St and not on the previous states S1, S2, · · · , St−1. That is, for all t, P[St+1|St] = P[St+1|S1, S2, · · · , St].

We always talk about time-homogeneous Markov chain in RL, in which the probability of the transition is independent of t:

P[St+1 = s′|St = s] = P[St = s′|St−1 = s].

Formally,

Definition 1 (Markov Process). a Markov Process (or Markov Chain) is a tuple (S,P), where

• S is a (finite) set of states

• P is a state transition probability matrix. Pss′ = P[St+1 = s′|St = s].

The dynamics of the Markov process proceeds as follows: We start in some state s0, and moves to some successor state s1, drawn from Ps0s1. We then moves to s2 drawn from Ps1s2 and so on. We represent this dynamic as follows:

s0 →s1 →s2 →s3 →...

If we introduce reward, action and discount into a Markov process, we get a Markov decision process.

Policy and Value function

o **Policy Function:**
The agent's action selection is modeled as a map called policy:

$$\pi : A \times S \to [0, 1]$$
$$\pi(a, s) = \Pr(a_t = a \mid s_t = s)$$

o **State-Value Function:**
The value function is defined as the expected return starting with state , i.e. , and successively following policy . Hence, roughly speaking, the value function estimates "how good" it is to be in a given state.

$$V_\pi(s) = \mathrm{E}[R \mid s_0 = s] = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right],$$

where the random variable R denotes the **return**, and is defined as the sum of future discounted rewards:

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,$$

where r_t is the reward at step t, $\gamma \in [0, 1)$ is the discount-rate. Gamma is less than 1, so events in the distant future are weighted less than events in the immediate future.

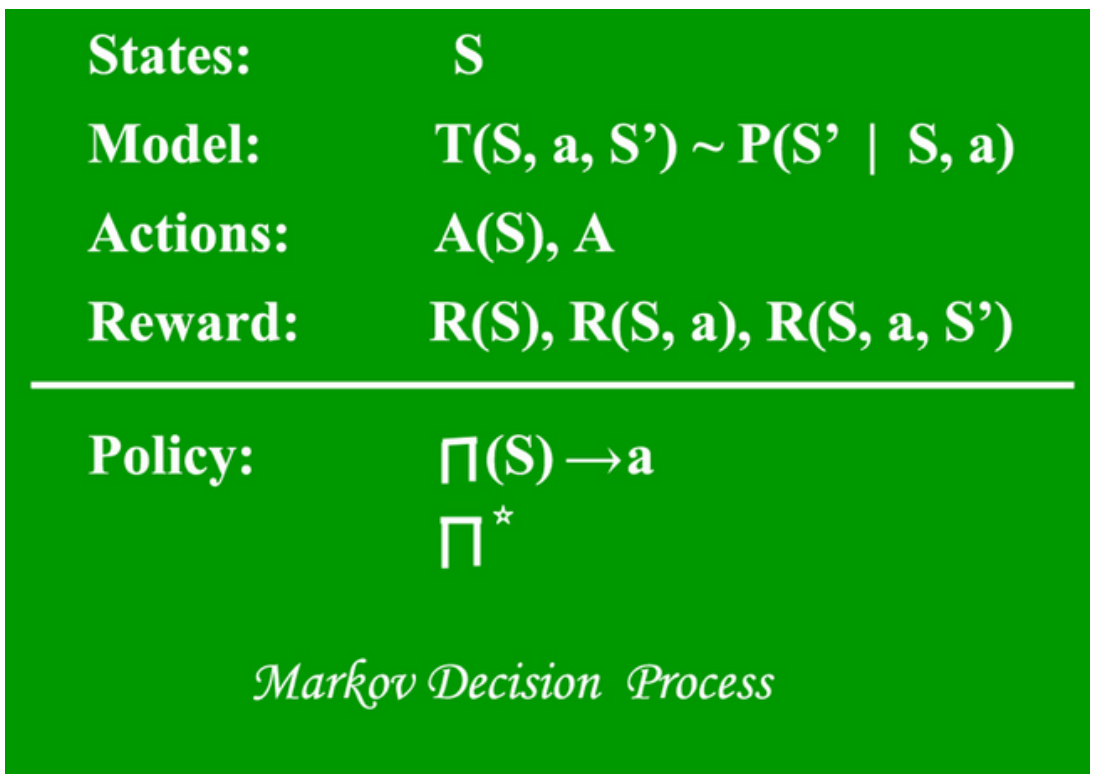
o **Value Function:**
Value function approaches attempt to find a policy that maximizes the return by maintaining a set of estimates of expected returns for some policy (usually either the "current" [on-policy] or the optimal [off-policy] one).

Markov Decision Process

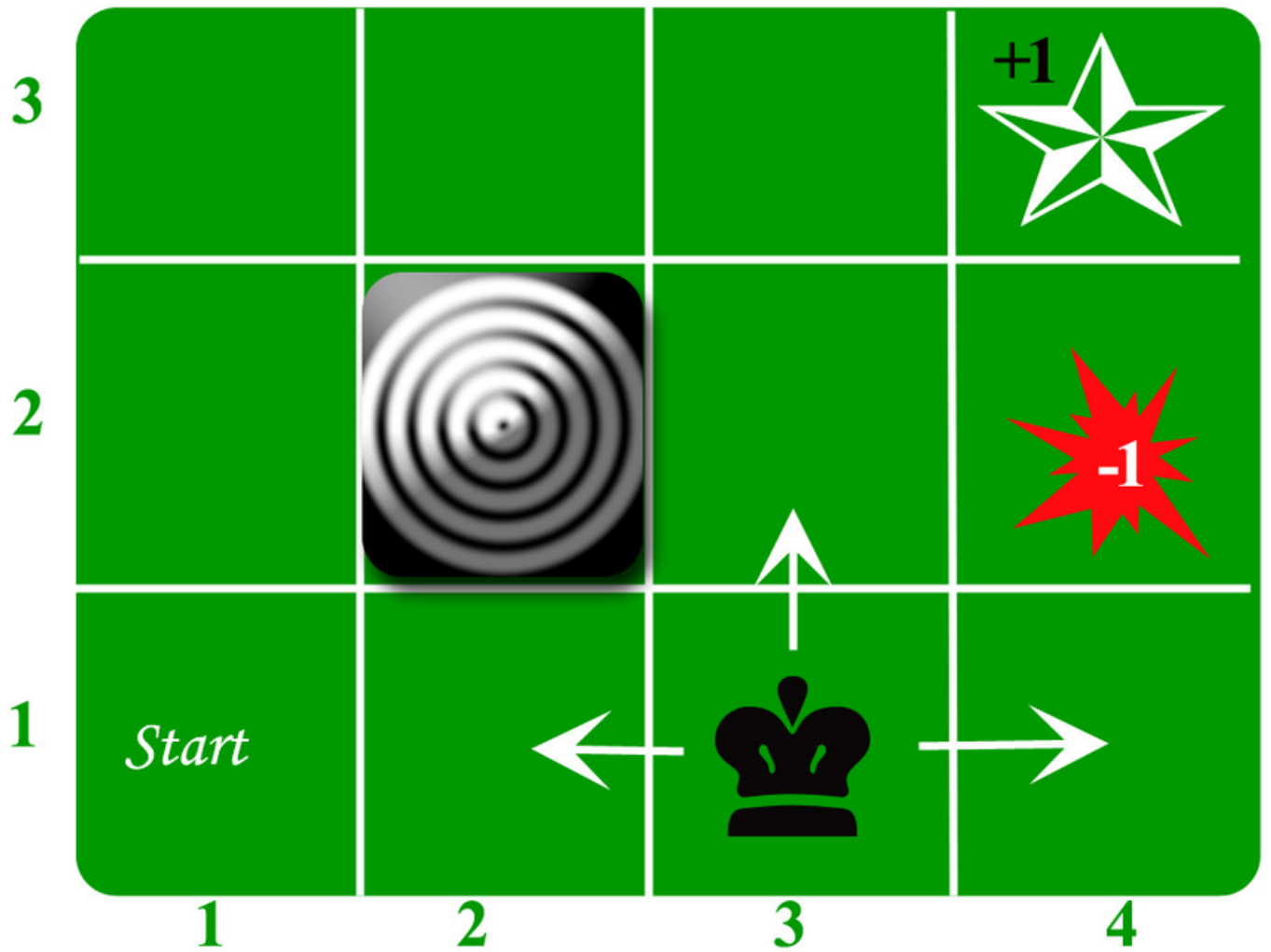
o **Markov Decision Process:** A Markov decision process is a tuple (S,A,P,γ,R), where:

- S is a finite set of states.
- A is a finite set of actions.
- P is the state transition probability matrix, $P_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$
- $\gamma \in [0, 1]$ is called the discount factor.
- $R: S \times A \rightarrow R$ is a reward function.

o The MDP is used to model the **environment in reinforcement learning.** In the MDP, the transition to the next state St+1 depends not only on the current state St, but also depends on the action At you make at the current state. Also, each state-action pair is attached with a reward function.



Markov Decision Process: Example



Markov Decision Process: Example

o An agent (one which performs tasks) lives in the grid. The above example is a 3*4 grid. The grid has a START state(grid no 1,1). The purpose of the agent is to wander around the grid to finally reach the Blue Diamond (grid no 4,3). Under all circumstances, the agent should avoid the Fire grid (orange color, grid no 4,2). Also the grid no 2,2 is a blocked grid, it acts as a wall hence the agent cannot enter it.
The agent can take any one of these actions: **UP, DOWN, LEFT, RIGHT**
Walls block the agent path, i.e., if there is a wall in the direction the agent would have taken, the agent stays in the same place. So for example, if the agent says LEFT in the START grid he would stay put in the START grid.

First Aim: To find the shortest sequence getting from START to the Diamond. Two such sequences can be found:

- RIGHT RIGHT UP UPRIGHT
- UP UP RIGHT RIGHT RIGHT

Let us take the second one (UP UP RIGHT RIGHT RIGHT) for the subsequent discussion. The move is now noisy. 80% of the time the intended action works correctly. 20% of the time the action agent takes causes it to move at right angles. For example, if the agent says UP the probability of going UP is 0.8 whereas the probability of going LEFT is 0.1, and the probability of going RIGHT is 0.1 (since LEFT and RIGHT are right angles to UP).

The agent receives rewards each time step:-

- Small reward each step (can be negative when can also be term as punishment, in the above example entering the Fire can have a reward of -1).
- Big rewards come at the end (good or bad).
- The goal is to Maximize the sum of rewards.

Customisations of a reward function in AWS DeepRacer

o Creating a reward function is like designing an incentive plan. Parameters are values that can be used to develop your incentive plan.

o Different incentive strategies result in different vehicle behaviors. To encourage the vehicle to drive faster, try awarding negative values when the car takes too long to finish a lap or goes off the track. To avoid zig-zag driving patterns, try defining a steering angle range limit and rewarding the car for steering less aggressively on straight sections of the track.

Editing Python code to customize your reward function

o In AWS DeepRacer Student, you can edit sample reward functions to craft a custom racing strategy for your model.

To customize your reward function

1. Customize reward function page of the AWS DeepRacer Student Create model
2. Use the code editor below the sample reward function picker to customize the reward function's input parameters using Python code.
3. Select Validate to check whether or not your code will work. Alternatively, choose Reset to start over.
4. Once you're done making changes, select Next.

Customisations of a reward function in AWS DeepRacer

In essence, reinforcement learning is modelled after the real world, in evolution, and how people and animals learn. Through experience, we humans learn what to do and what not to do in different scenarios and apply them to our own lives. RL works the same way through simulating experiences to help machines learn.
Let's frame the problem by describing some key terms, using AWS DeepRacer as an example:

- Agent/Actor: the entity performing actions in the environment (AWS DeepRacer and the models that control it)
- Actions: what the agent can do (the car can turn, accelerate, etc)
- Environment: where the agent exists (the track)
- State: the key characteristics in a given time (location on the track)
- Reward: the feedback given to the agent depending on its action in the previous state (a high reward for doing well, a low reward for doing poorly)

So here's how a basic RL model learns: an agent in an environment performs an action in its given state. The environment gives the agent a reward depending on the quality of its action. The agent then performs another action and the cycle continues. A training episode consists of the entirety of this cycle for a certain amount of actions.

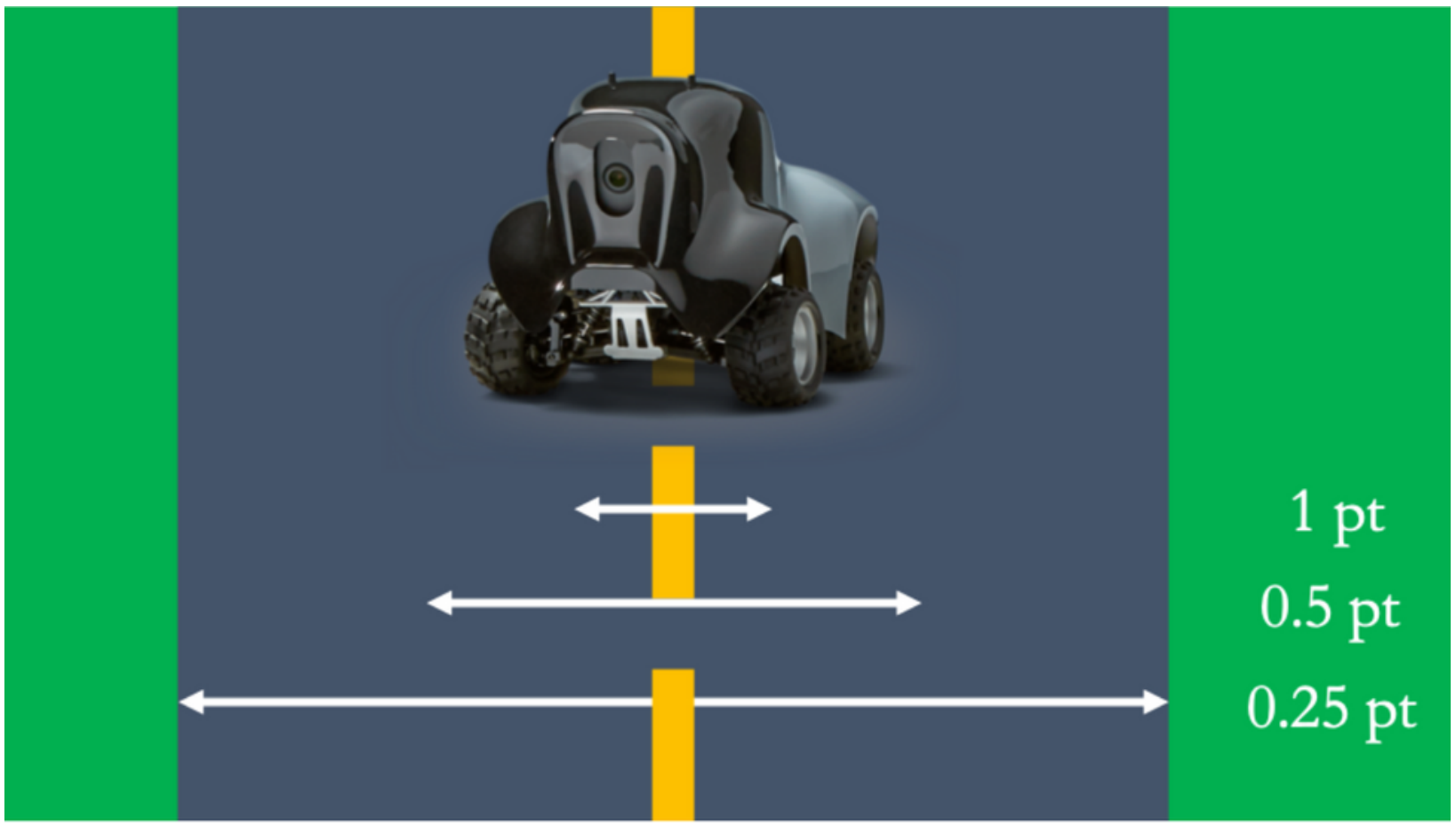
Whats in the reward?

The reward function is a very important part of an RL model. Having one that incentives optimal actions and disincentivizes poor actions is critical to have a well-trained agent. In AWS DeepRacer, we create our reward function with input parameters. These comprise of the position of the car, the distance from the center, and more. Here's an example of a basic reward function:

```
def reward_function(params):
    '''
    Example of rewarding the agent to follow center line
    '''

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

    # Calculate 3 markers that are at varying distances away from the center
line
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the car is closer to center line and vice versa
    if distance_from_center <= marker_1:
        reward = 1.0
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3  # likely crashed/ close to off track

    return float(reward)
```

Whats in the reward?

In this function, we use the input parameter params[‘track_width’] and params[‘distance_from_center’]. We create three markers and compare the car’s distance from the centre line with these markers. If the car is within 10% from the centre, it’s given a reward of 1 . If it’s within 25%, it’s given a reward of 0.5. If it’s on the track, it gets 0.1. And if it’s off the track, we give it 0.001, which amounts to basically nothing .



This reward function will give the car a high reward if it sticks to the middle of the track. Knowing this, it will avoid falling off the track, which is what we want! While this reward function doesn’t optimize for speed, at least we can rest assured that we’ll probably finish the race.

Input parameters of the AWS DeepRacer reward function

```
def reward_function(params) :

    reward = ...

    return float(reward)
```

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

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

}
```

A reward function based on closest_waypoints *parameter*

```
# Place import statement outside of function (supported libraries: math, random, numpy,
 scipy, and shapely)
# Example imports of available libraries
#
# import math
# import random
# import numpy
# import scipy
# import shapely

import math

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

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

    # Initialize the reward with typical value
    reward = 1.0

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

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

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

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

    return float(reward)
```

A reward function based on steering_angle *parameter*


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

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

    # Initialize the reward with typical value
    reward = 1.0

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

    return float(reward)
```