# AMAZON DEEPRACER 2020 – BATTLE OF SCHOOLS

By Amazon AWS and EagleDream

TEAM

## THE FAST AND THE CURIOUS

Mayur Jawalkar
Sam Kuzio
Kunjan Mhaske
Sourabh Hanamsheth
Paritosh Borkar
Thomas Morris

# Table of Contents

## Introduction:

Amazon DeepRacer 1/18$^{th}$ Scale Autonomous Car Racing Competition gives the developers and students an interesting and fun way to get started with reinforcement learning (RL), experiment with new RL algorithms and simulation-to-real domain transfer methods, and experience RL in the real world. AWS DeepRacer is the first autonomous scale car specifically developed to help developers get hands-on with reinforcement learning. The goal of this competition is to develop an algorithm that let the small autonomous car drive on the real-world track using Reinforcement Learning on Deep Neural Networks with 3D racing simulation while training and evaluating. Developers can train, evaluate, tune RL models on online 3D racing simulation and deploy their model onto AWS DeepRacer car for real-world autonomous experience and compete in the competition.

## Objective:

To develop the reward function that utilizes the inputs from the car's environment and behavior to decide whether give rewards or not, based on how it is performing in real time and complete the track in minimum time.

## Reinforcement Learning (RL) :

RL is an advanced machine learning technique that takes a very different approach to training models than other machine learning methods. According to AWS, its super power is that it learns very complex behaviors without requiring any labeled training data, and can make short term decisions while optimizing for a longer term goal. In the DeepRacer, an agent (the car) interacts in an environment (the track). Depending on its policy, the car gets a reward. Then, update the policy in order to maximize the reward function.

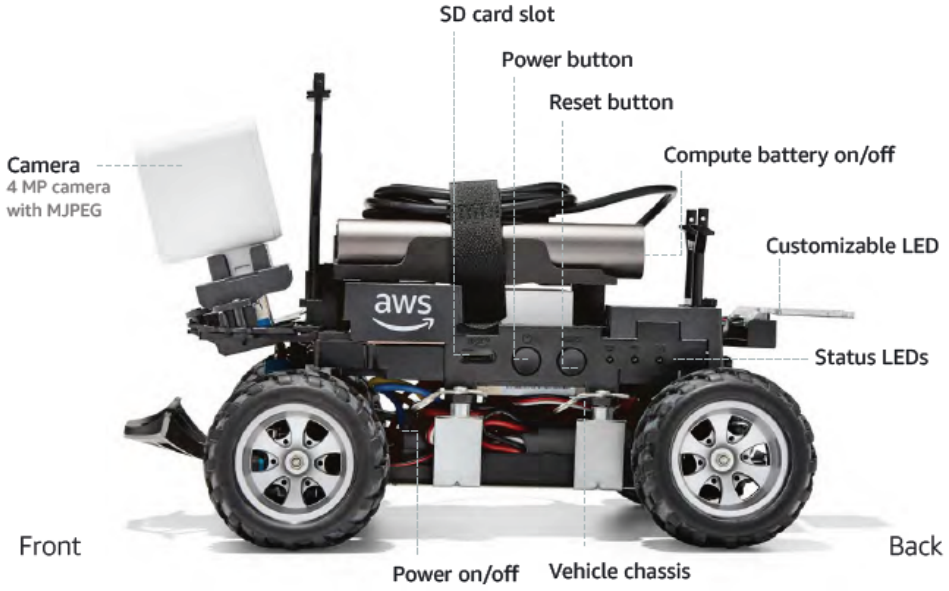
In DeepRacer, the agent gets rewards from the actions it takes as a response to the environment. In reinforcement learning, the reward function plays a critical role in training your models. The reward function provides the logic that tells your agent how good or bad an outcome was by assigning a value to it. These rewards are used in the training algorithm, which optimizes the model to choose actions that lead to the maximum expected cumulative rewards. The reward function is used during training to incentivize the driving behavior you want the agent to exhibit when you ultimately stop training the model and race it.

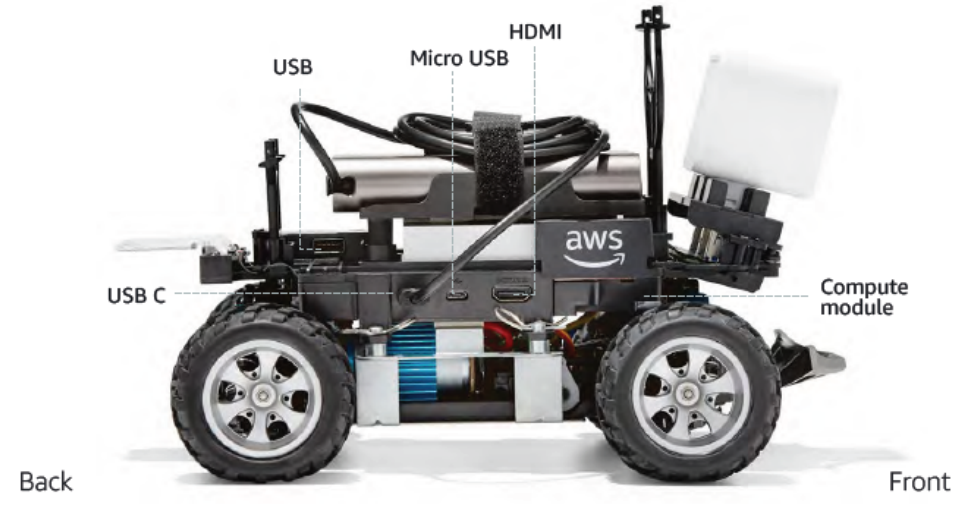
In practice the reward is calculated during each step in training. That is after each action is taken a reward is calculated based on the outcome. Each step is recorded (state, action, next state, reward) and used to train the model.

## Agent – DeepRacer Car:

The Amazon DeepRacer vehicle is a Wi-Fi enabled, physical vehicle that can drive itself on a physical track by using a reinforcement learning model.

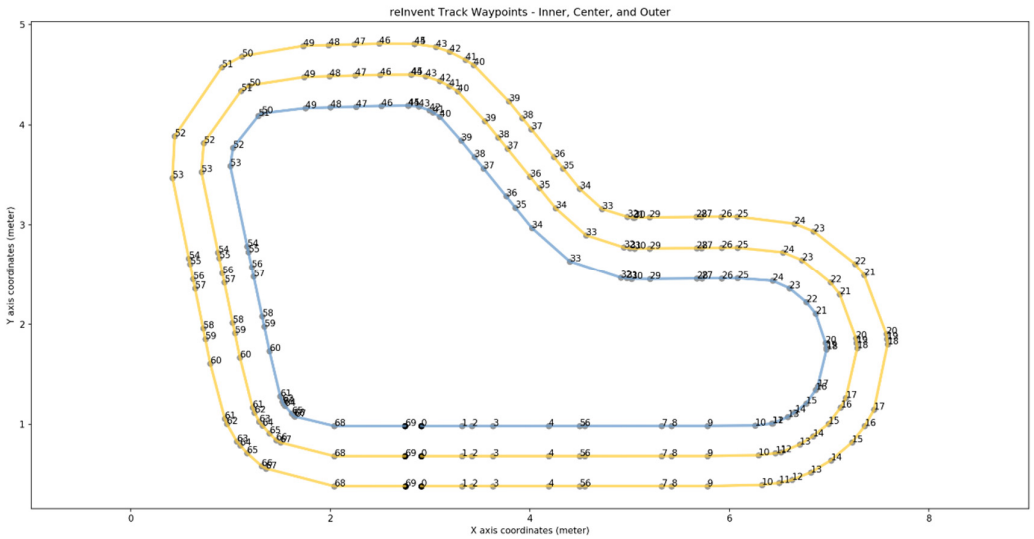
**Car**: 1/18th scale 4WD with monster truck chassis
**CPU**: Intel Atom™ Processor
**Memory**: 4GB RAM
**Storage**: 32GB (expandable)
**Wi-Fi**: 802.11ac
**Camera**: 4 MP camera with MJPEG
**Software**: Ubuntu OS 16.04.3 LTS, Intel® OpenVINO™ toolkit, ROS Kinetic
**Drive battery**: 7.4V/1100mAh lithium polymer
**Compute battery**: 13600mAh USB-C PD
**Ports**: 4x USB-A, 1x USB-C, 1x Micro-USB, 1x HDMI
**Sensors**: Integrated accelerometer and gyroscope



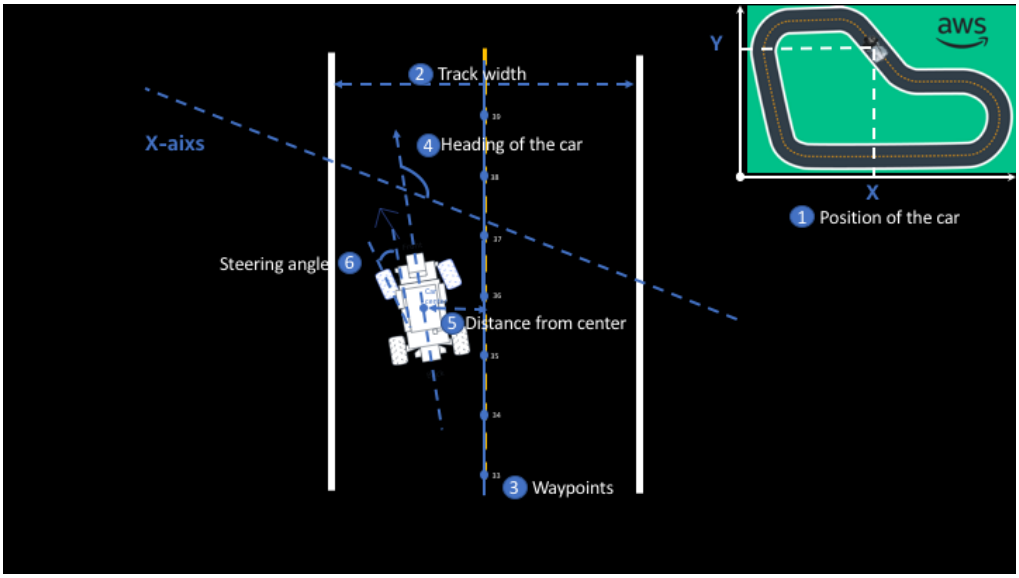



## Environment – Simulation Track and Real-World Track:

Here is a visualization of the waypoints used for the track used in the competition. We only have access to the centerline waypoints in the reward function.

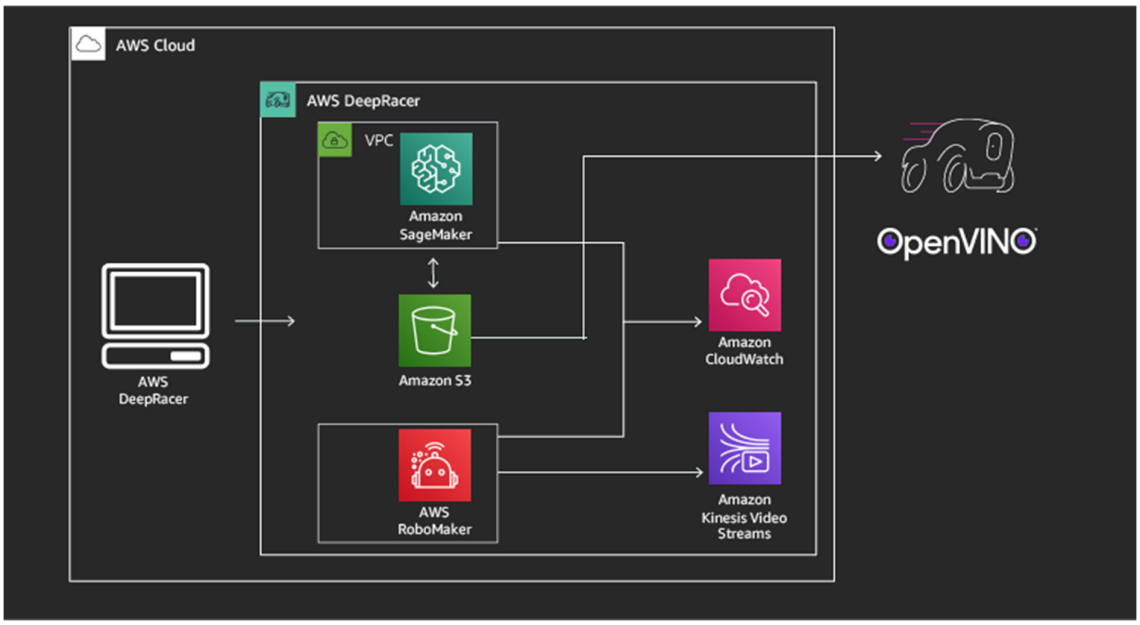



Here is a visual explanation of some of the reward function parameters:



➢ **How it works?**

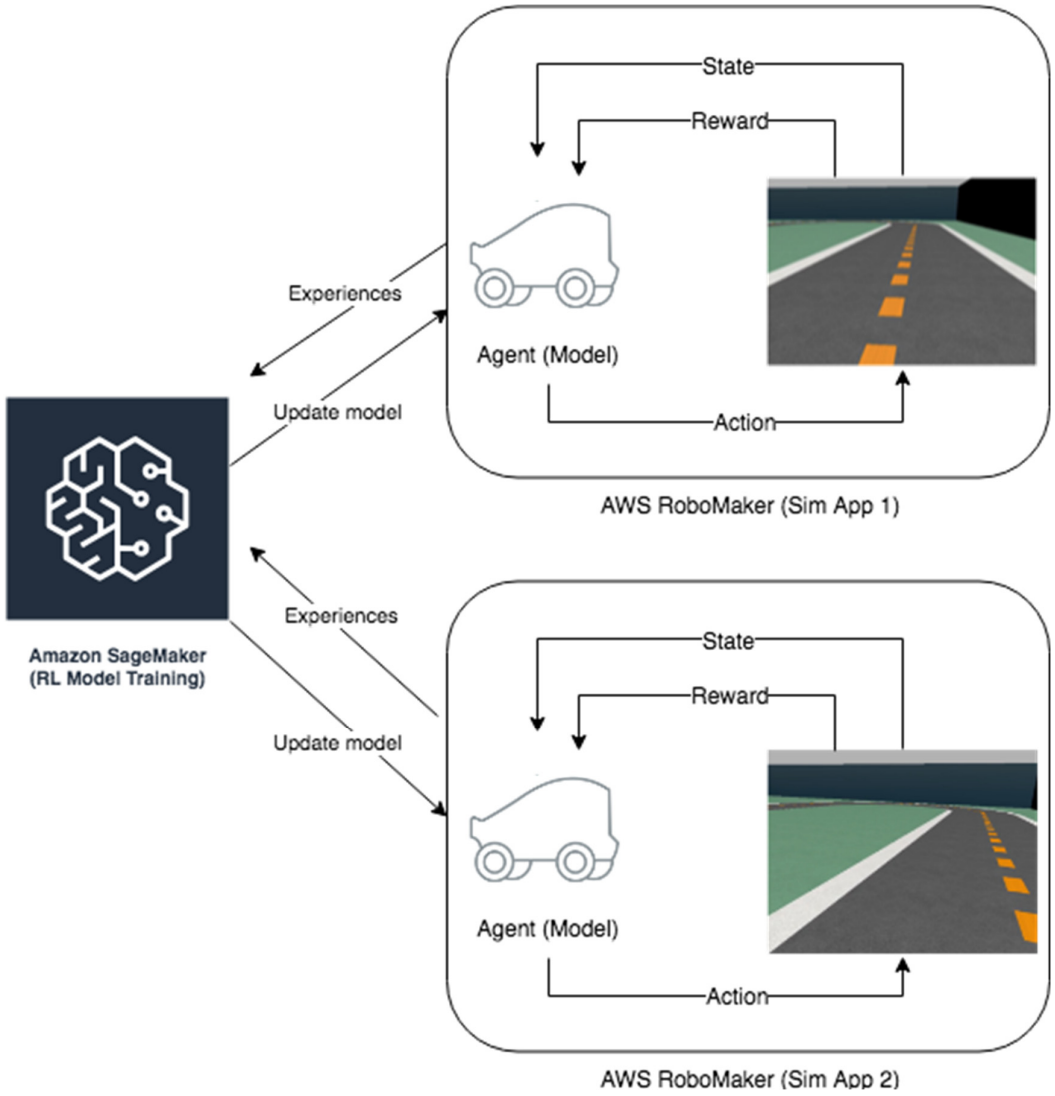
AWS Machine Learning ecosystem offers an interactive learning system for users of all levels to acquire and refine their skill set in machine learning in general and reinforcement learning in particular. We can use the AWS DeepRacer console to train and evaluate deep reinforcement learning models in simulated autonomous-driving environment and then deploy them to an AWS DeepRacer vehicle for real world autonomous driving.




The tasks include how to define a rewards function, customize the action space, set up the training configuration, and creating a training job using Amazon SageMaker, AWS RoboMaker to provide the racing simulation, Amazon Kinesis Video Streams for video streaming of virtual simulation footage to observe the training and evaluation, Amazon CloudWatch to monitor the training logs and Amazon S3 to store the model.

Training is done through an iterative process of simulation to gather experience, followed by training on the experience to update the model, followed by simulation using the new model to get new experience, followed by training on the new experience to update the model and so forth.

Initially the model does not have any knowledge of which actions will lead to good outcomes. It will choose actions at random as it explores the environment. Over time it will learn which actions are better and start to exploit these. How quickly it exploits or how long it explores is a trade-off that needed to make by developers.




The car has a limited range of actions like speed up, move left/right slightly or strongly etc. and a Deep Neural Network trained on custom reward function that predicts which of these actions is the most appropriate and gives good rewards depending on the image as an input from camera on car.

While training, the car makes a decided number of attempts (using Hyperparameter and Training time) in a virtual simulator and give the car reward after each attempt. The results of this rewards backpropagate in the Deep Neural Network to update every weight in every steps.

## Parameters:

The following list contains the variables that can be used in the reward function.

```
"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
"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.
"offtrack": Boolean,                     # Boolean flag to indicate whether an 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_len.
"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 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
```

## Hyperparameters:

These are the hyperparameters for RL optimization algorithm:

| Hyperparameter | Impact on training |
|---|---|
| Batch size | Batch size determines the number of steps, randomly sampled from the most recent experience, we will use to train our network. Thus it is the amount of experience you use to update your network. |
| Number of Epochs | This is the number of times that our optimization algorithm will run through the batch and update the weights of our network, each time that training is performed. |
| Learning rate | The learning rate determines the size of the updates to network weights. |
| Exploration | This is the method used to provide the exploration/exploitation trade-off. Thus to what extent will your model explore vs exploit. |
| Entropy | Entropy controls the amount of exploration, by adding noise to the probability distribution of actions. Smaller entropy implies less exploration. |
| Discount factor | This determines how far forward the model should look in time to value its actions. In our basic environment 0.9 looks about 20 to 30 steps ahead, and it can take 100s of steps to make a turn. |
| Loss type | Loss functions used during our weight update process. |
| Number of episodes between each training | This determines how much experience we want to obtain, in the form of episodes, before we run training. After each training step, we will use the newly trained model to obtain more episodes and iteratively continue the process. |

# Training:

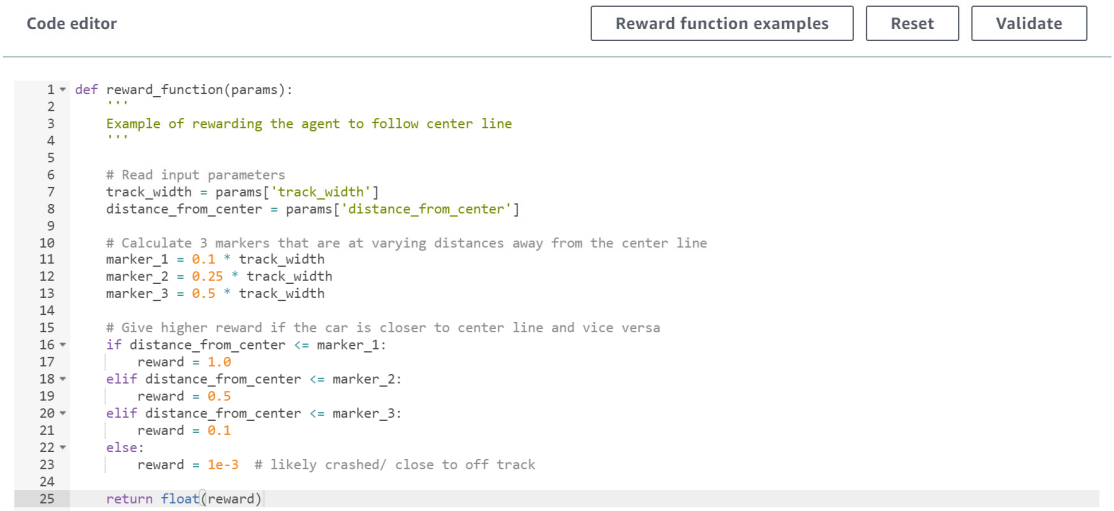
## ➢ Get familiar with the environment

Initially, we decided to train our car using a default/sample model provided by AWS to get familiar with the AWS working environment. This model was monitoring whether the car is on the track using distance of the car from the center of the track. In this model, the car was getting a good reward if it stays close to the center line of the track. Otherwise it was getting a very less reward. We trained our model for 4-6 hours. Using this model, we were able to keep our car on the track most of the time and it was completing the track in around 50-60 seconds.

Although, this model was not one of the models which resulted in significant outcomes in terms of racing event, it helped us to get comfortable with the working environment. Learning outcomes from this model were very crucial to move ahead with next models. We learnt about the following things.

1. How to write a reward function?

   Writing a reward function is easy thing. We were provided with a console to write our reward function in python while creating our model.




2. How to access the parameters in the reward function?

   All the parameters were accessible using the dictionary – ‘params’, passed as an argument to the reward function. e.g.,

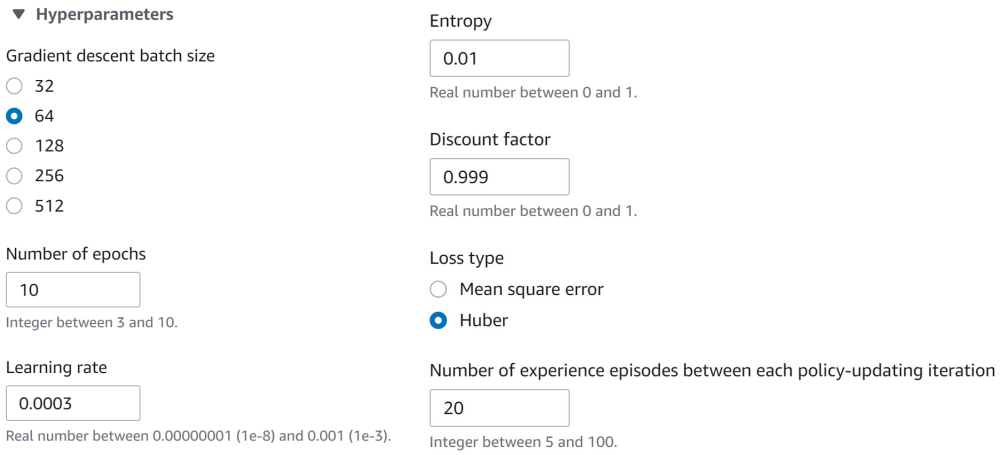
   distance_from_center = params['distance_from_center']

3. What all hyper parameters are present?

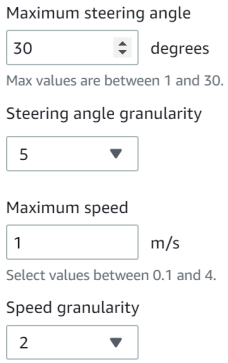
   Gradient descent batch size, Number of epochs, Learning Rate, Entropy, Discount Factor, Loss Type, and Number of experience episodes between each policy-updating iteration.

4. How to change those hyperparameters?

   We can change these hyperparameters while creating a new model.




5. How to customize your own car?
   We were able to customize our own car in the simulation environment. In the customization option, we were able to decide upon type of CNN (3 layered / 5 layered) and action space using maximum steering angle, angle granularity, maximum speed of the vehicle, and speed granularity.




**Action list**

| Action number | Steering angle | Speed |
|---|---|---|
| 0 | -30 degrees | 0.5 m/s |
| 1 | -30 degrees | 1 m/s |
| 2 | -15 degrees | 0.5 m/s |
| 3 | -15 degrees | 1 m/s |
| 4 | 0 degrees | 0.5 m/s |
| 5 | 0 degrees | 1 m/s |
| 6 | 15 degrees | 0.5 m/s |
| 7 | 15 degrees | 1 m/s |
| 8 | 30 degrees | 0.5 m/s |
| 9 | 30 degrees | 1 m/s |

After gaining the clear understanding of the environment by working on the default model, we decided to move ahead with building a strategy for our car.

- **Strategy:**

Our main aim of this event was to complete the lap in the minimum time. Hence, in order to do that our basic strategy was to '*slow down around the curves and speed up during the straight road'*.

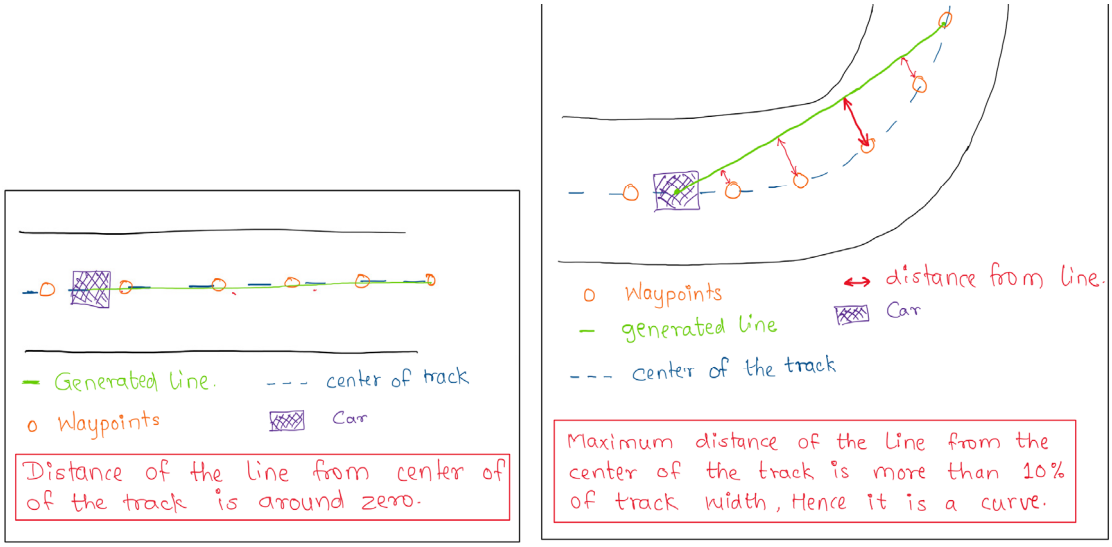
- **Challenge:**

The track was completely new for our car and there was not way to save the information of the track. Hence, after deciding the strategy, our major challenge was to identify the straight and curvy regions around the track. Without this information our strategy was useless.

➢ **Solution:**

We 11ehaviou all parameters from the provided list and figured out a way to identify the straight and curvy regions of the track using parameters like 'x', 'y' (current coordinates of the car), 'waypoints' (list of all waypoints), 'closest_waypoints' (list of the two closest waypoints), 'track_width', 'distance_from_center', etc.

We decided to draw a line of reference from the current position of the car to next waypoints by considering each waypoint one by one. After drawing this line, we checked the maximum distance of that line from the center of the track. For doing this we computed the distance of the line from every intermediate waypoint because of a known fact that all waypoints are on the center of the track. Using this approach, we can easily identify the curve because the maximum distance from the line will be high if there is a curvy region ahead and it will be nearly zero if it is a straight region ahead.



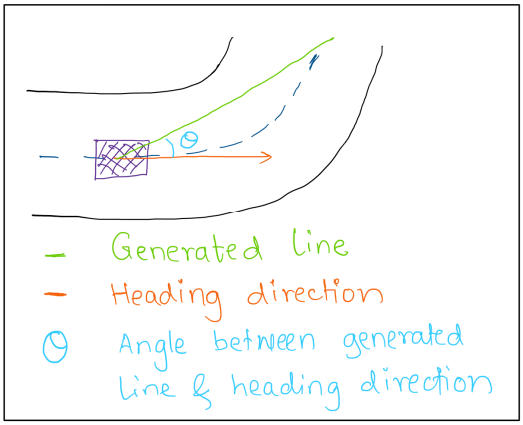

But, even with this approach we can't keep looking at infinitely long distance considering the real-time 11ehaviour of our car. With infinite vision, car will have a clear understanding of the road, but it will take time to process that much information. On the other hand, with a very short vision, car will take fast decisions but won't have a clear insight about the road. Hence, to address this trade-off, initially we decided to limit the vision of the car to next five waypoints only.

➢ **Decide Speed and Angle Reward:**

In this event we were working on a reinforcement learning model. Hence, it was crucial to decide the rewards for the behavior of the car around various regions of the track. We decided to give the maximum reward for maximum speed around the line closest to the center of the track. i.e., maximum reward for a line within 10% of the track width from the center of the track. And, give maximum reward for speeds in range 1 m/s to 3 m/s around the curves i.e., distance of the line is more than 10% of the track width from the center of the track.

```
# Determine if we are going straight or curving
if max_waypoint_distance_from_target_line < 0.1 * track_width:

    speed_reward = 1 - (speed / 5.0)

# Else if we are on a curve
else:

    if speed < 1:
        speed_reward = 0
    elif speed < 3:
        speed_reward = 1
    else:
        speed_reward = 0
```

Along with that, we also computed the angle between the angle between heading of our car and the generated line. We decided to give the maximum reward if the angle is less i.e., car is moving in along the generated line, and vise-versa.




```
# Reward based on difference in heading angle and target line angle
target_line_angle = math.degrees(math.atan(a))
heading_actual = heading
if heading < 0:
    heading_actual = heading + 360

angle_diff = abs(target_line_angle - heading_actual)

# Calculate reward based on how close the car is to being on the heading of the target line
angle_reward = 1 - (angle_diff / 180)
```

To compute the final reward we decided to take the square root of the squared sum of the angle and speed rewards.

```
# Calculate final reward
reward = math.sqrt(angle_reward**2 + speed_reward**2)
```

After training this model for 6+ hours, it was performing better in terms of following the path, but the speed was not improved. The car was still giving the best lap time of around 1 minute which was very high in order to compete in the competition.

- **Improvement in Speed:**

In the previous step, our car was not speeding up. Hence, we started trying different ways of giving speed and angle rewards. We tried increasing the threshold for a maximum distance between generated line and nearest waypoints, increase the vision by increasing the number of waypoints to look further, etc. Moreover, we tried to change the action space of the car by changing the maximum speed of the car from 1 m/s to 2, 3, 4 m/s and changing the steering angle granularity. But none of these changes were showing any significant result in terms of speed improvement.

Finally, after analyzing complete reward function, we figured out that the issue was with the method to compute the final reward. Square root of a squared sum of the speed and the angle reward was not making much sense for a model to learn the speeding behavior. We changed this method to compute the final reward by taking the average of the speed and angle reward.

After making this change and letting the model to train for 5+ hours, our model showed the significant improvement in the speed. We got the best lap time of around 24 seconds. But, with this improvement in speed, the car started showing some weird behavior in terms of following the path. It was showing the behaviors like following the path in reverse direction, frequently going off track and drifting off the corners.

Later, to fix this behavior, we decided to make use of few more parameters like is_off_track, all_wheels_on_track, is_reversed, etc. Using these Boolean parameters, we penalized the weird behaviors of the car. Also, after trying various combinations, we decided to use a car with speed granularity of 3, maximum speed of 2 m/s, maximum steering angle 30 degrees and angle granularity of 5. This model showed significant improvement in terms of path following.

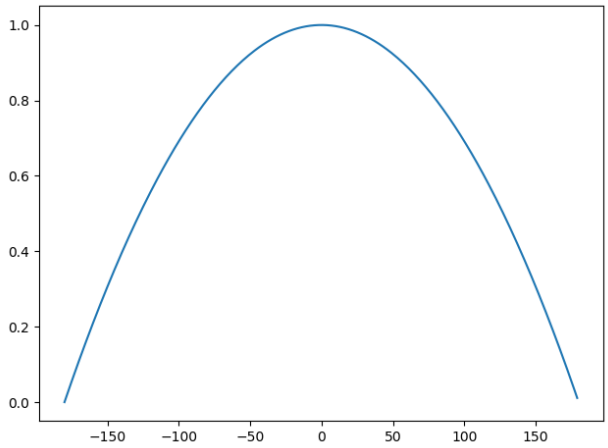
Although this model was giving the best lap time of around 24 seconds, there was still a scope to improve the lap time and reliability to make it ready for the final competition.

## ➢ Final Improvements:

To improve the behavior of the model, again we started investigating the core reward functions i.e., angle and speed reward. We noticed that previously used angle reward was showing a straight-line behavior. We tried to make it a smooth curve by changing the reward function.

```
# Calculate reward based on how close the car is to being on the heading of the target line
# angle_reward = 1 - (angle_diff / 180)
angle_reward = 1 - ((angle_diff/180)**2)
```

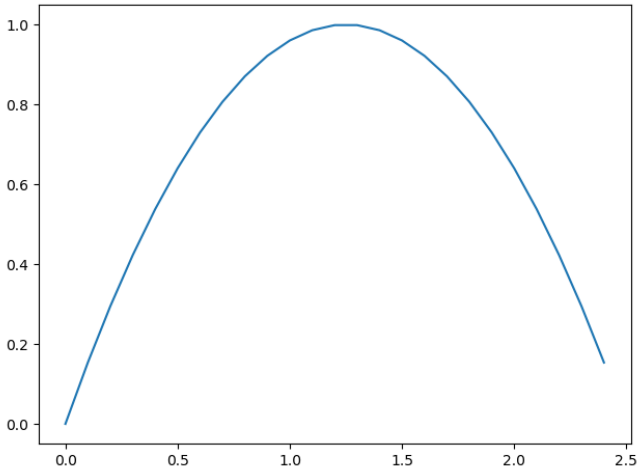
Behavior of this function can be viewed by the below curve. It gives the maximum reward for 0 degree angle and reward gradually decreases as the angle moves away from zero.



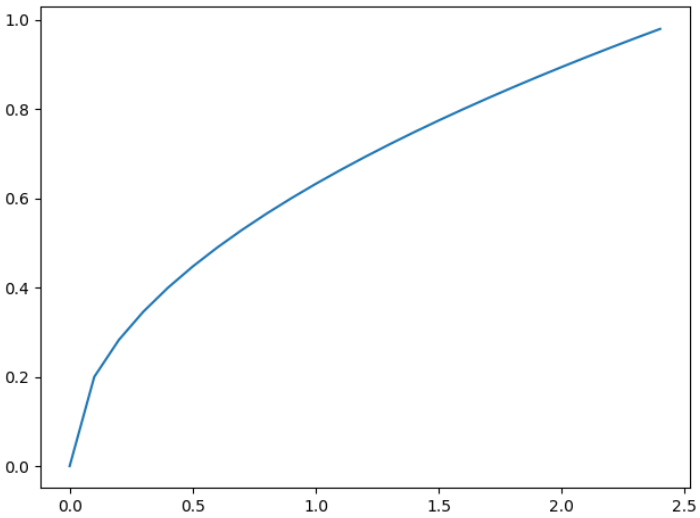

Moreover, we tried to smoothen the curves for speed reward. We changed the function to compute the speed reward to the function mentioned below.

```
# Determine if we are going straight or curving
if max_waypoint_distance_from_target_line < 0.1 * track_width:

    speed_reward = (speed / MAX_SPEED)**0.5

# Else if we are on a curve
else:
    speed_norm = speed / MAX_SPEED
    # Generate a curve based on the normalized speed that peaks at about 75% max speed.
    #speed_reward = (-8 * (speed_norm**3)) + (9.3714 * (speed_norm ** 2)) + (-1.4714 * speed_norm) + 0.1214
    # Generate a curve that peaks at 50% max speed. Should work well with speed granularity 2.
    speed_reward = (-4 * (speed_norm**2)) + (4 * speed_norm)
    speed_reward *= 0.75
```

For the curvy region: This function shows the following behavior. It gives the best reward for speed around 1.2 m/s and reward gradually goes down otherwise.




For the straight region: This function shows the following behavior. It gives the best reward for maximum speeds.




The final reward is calculated by taking the square root of the average of these two rewards. These changes show the significant improvement in the behavior of the car both in terms of speed and path following reliability after training of around 6+ hours. This model gave us the best lap time of around 12-13 seconds in simulation environment. This was our best performing model which we decided to use during the day of event.

## Event Day:

- **Practice Round:**

On the event day, we selected three of our best performing models to use on an actual track. We got around 10 minutes on the track to practice and observe the behaviors of our models in real world. During these 10 minutes we were given an i-pad which was the abstract controller of the car. Using this controller, we were able to control the maximum speed threshold of the car. After running our car around the track for couple of laps we finalized the best speed threshold range of 70% - 85%. We also observed the behavior of our car across the critical sections of the track e.g., horse-shoe curve, lighting differences, etc. At the end of this training time, we decided the strategy to use during the qualifier round.

- **Qualifier Round:**

During qualifier round, we decided to start our car on the track with the threshold of 75% and later keep on increasing the threshold by observing the behavior of our car. Our car was performing well, and we could increase the speed threshold up to 93% when we got the best lap time of 9.4 seconds. This lap time qualified us for the final round. During the time between the final event and the qualifiers we observed the behaviors of the cars of other teams and decided the strategy to use during the final round.

We decided to start with our reliable model which gave us the best lap time in qualifiers with 80% speeding threshold and then increase the threshold runtime upon looking at the behavior of the car. We also planned to be aggressive after getting one decent lap time around the track.

- **Final Round:**

When we reached at the event venue and observed the track there was a major change in the lighting condition. The light color was bright, there were bright and shady spots around the track. We observed that the performance of the other team was not exactly same as the qualifier round, probably because of lighting conditions and car calibrations. We changed our strategy to start with 75% threshold after seeing these effects on the other team's model. During our race, we faced the similar issues. We got the best lap time of around 15.2 seconds in the final race.

Although, the reason behind this behavior of the car can not be clearly debugged, we anticipate that it is because of the challenging lighting condition and car calibrations.

## Conclusion:

In this event, we learnt about reinforcement learning by practically developing various models by writing reward functions and setting hyperparameters. We observed the effect of bad reward function on the model. Our model was not showing any improvement in earlier stages because of a bad way of calculating the final reward. But this behavior was improved as soon as we fixed the reward function. Also, we observed the effect of hyperparameters on the learning behavior of the model. We observed that changing the neural network from 3-CNN to 5-CNN and increasing the action space significantly increases the learning time. The major thing which we learnt on the event day is that there is always a difference between the behavior of model in the simulation and real world. Our car was drifting around the corners in the simulated environment. But on the real track the friction was much better, and car never drifted. Also, the lighting conditions in the simulation were constant. But, on actual track our car showed bad behavior because of different lighting conditions.

Finally, we would like to thank Dr. Kinsman to provide us a chance to participate in this event as a part of academic project. It was a great learning opportunity to learn about reinforcement learning.

## References:

DeepRacer Documentation: https://github.com/awsdocs/aws-deepracer-developer-guide

DeepRacer Guidelines: https://aws.amazon.com/deepracer/getting-started/

DeepRacer Workshops: https://github.com/aws-samples/aws-deepracer-workshops

## Final Reward Function For Reference:

```
def reward_function(params):
    import math  # This uses a lot of math.

    # Read input parameters
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    steering = abs(params['steering_angle']) # Only need the absolute steering angle
    current_speed = params['speed']
    closest_waypoints_indices = params['closest_waypoints']
    next_waypoint_index = closest_waypoints_indices[1]
    waypoints = params['waypoints']
    car_pos = (params['x'], params['y'])
    speed = params['speed']
    heading = params["heading"]
    is_offtrack = params['is_offtrack']
    is_reversed = params['is_reversed']

    # Punish harshly for very bad behavior
    if is_offtrack or is_reversed:
        return float(0)

    # At the very least, head towards the next waypoint
    target_waypoint = waypoints[next_waypoint_index]

    # Keep the maximum distance between the mid-waypoints and the target line.
    # This is a proxy for how intense the curve is.
    max_waypoint_distance_from_target_line = 0

    # Assume that the next waypoint is in front of the car.
    # Also assume that the line between the car and the next waypoint is always valid.
    # Iterate over the waypoint AFTER the next waypoint up to 7 after the next waypoint
    for idx in range(next_waypoint_index + 2, next_waypoint_index + 7):

        mod_idx = idx % len(waypoints)


        b = -1

        # Slope between car and the point we are evaluating
        a = (car_pos[1] - waypoints[mod_idx][1]) / (car_pos[0] - waypoints[mod_idx][0])

        # Y intercept
        c = -(car_pos[0] * (a) + car_pos[1])

        # For each point between the next point and the curent point,
        # See if the distance between the the point and the line we are testing
        # Is greater than a threshold that will determine if we are or are not on the road.
        line_invalid = False
        for test_pt_idx in range(next_waypoint_index, idx):

            test_pt = waypoints[test_pt_idx % len(waypoints)]
            dist_to_line = abs((a * test_pt[0]) + (b * test_pt[1]) + c) / math.sqrt(a**2 + b**2)
```

```
        # Check if the line is invalid
        if dist_to_line > 0.5 * track_width:
            line_invalid = True
            break

        # Keep track of the waypoint that is furthest from the line
        if dist_to_line > max_waypoint_distance_from_target_line:
            max_waypoint_distance_from_target_line = dist_to_line
            # index_of_maximum_distance_waypoint = test_pt_idx % len(waypoints)

    if line_invalid:
        target_waypoint = idx-1 % len(waypoints)
        break

# Reward based on difference in heading angle and target line angle
target_line_angle = math.degrees(math.atan(a))
heading_actual = heading
if heading < 0:
    heading_actual = heading + 360

angle_diff = target_line_angle - heading_actual

# Calculate reward based on how close the car is to being on the heading of the target line
angle_reward = 1 - ((angle_diff/180)**2)

# The maximum speed of the car, as defined in the car's action space.
MAX_SPEED = 2.5

# Determine if we are going straight or curving
if max_waypoint_distance_from_target_line < 0.1 * track_width:

    speed_reward = (speed / MAX_SPEED)**0.5

# Else if we are on a curve
else:
    speed_norm = speed / MAX_SPEED
    # Generate a curve that peaks at 50% max speed. Should work well with speed granularity 2.
    speed_reward = (-4 * (speed_norm**2)) + (4 * speed_norm)
    speed_reward *= 0.75

# Model validation sometimes generates a math domain warning for the final
# reward calculation unless these lines are here.
if speed_reward < 0:
    speed_reward = 0
if angle_reward < 0:
    angle_reward = 0

# Calculate final reward.
reward = math.sqrt((speed_reward + (angle_reward)) / 2)

return float(reward)
```