AWS DeepRacer

AWS DeepRacer is the fastest way to get rolling with machine learning, literally. Get hands-on with a fully autonomous 1/18th scale race car driven by reinforcement learning, 3D racing simulator, and global racing league.

AWS DeepRacer is a 1/18th scale race car which gives you an interesting and fun way to get started with reinforcement learning (RL). RL is an advanced machine learning (ML) technique which takes a very different approach to training models than other machine learning methods. 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.

With AWS DeepRacer, you now have a way to get hands-on with RL, experiment, and learn through autonomous driving. You can get started with the virtual car and tracks in the cloud-based 3D racing simulator, and for a real-world experience, you can deploy your trained models onto AWS DeepRacer and race your friends, or take part in the global AWS DeepRacer League. Developers, the race is on.

Deep Racer in AWS Console

The AWS Deepracer console is now available from AWS. If you login to the console and search for Deepracer.

Where does the Reward come from? Reinforcement learning algorithms are geared for optimization of cumulative rewards.

The model will learn which action (and then subsequent actions) will result in the highest cumulative reward on the way to the goal.

The critical part to make your reinforcement learning model work is the reward function. In general you design your reward function to act like an incentive plan.

Reward function parameters for AWS DeepRacer In AWS DeepRacer, the reward function is a Python function which is given certain parameters that describe the current state and returns a numeric reward value.

The parameters passed to the reward function describe various aspects of the state of the vehicle, such as its position and orientation on the track, its observed speed, steering angle and more.

We will explore some of these parameters and how they describe the vehicle as it drives around the track:

1. Position on track - The parameters x and y describe the position of the vehicle in meters, measured from the lower-left corner of the environment.
2. Heading - The heading parameter describes the orientation of the vehicle in degrees, measured counter-clockwise from the X-axis of the coordinate system.
3. Track width - The track_width parameter is the width of the track in meters.
4. Waypoints - The waypoints parameter is an ordered list of milestones placed along the track center. Each waypoint in waypoints is a pair [x, y] of coordinates in meters, measured in the same coordinate system as the car's position.

5. Distance from center line - The distance_from_center parameter measures the displacement of the vehicle from the center of the track. The is_left_of_center parameter is a boolean describing whether the vehicle is to the left of the center line of the track.
6. All wheels on track - The all_wheels_on_track parameter is a boolean (true / false) which is true if all four wheels of the vehicle are inside the track borders, and false if any wheel is outside the track.
7. Speed - The speed parameter measures the observed speed of the vehicle, measured in meters per second.
8. Steering angle - The steering_angle parameter measures the steering angle of the vehicle, measured in degrees. This value is negative if the vehicle is steering right, and positive if the vehicle is steering left.

Training the Model

Link to Documentation:
https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-console-train-evaluate-models.html

If you clone a previously trained model as the starting point of a new round of training, you could improve training efficiency. To do this, modify the hyperparameters to make use of already learned knowledge.

- On the Models page, choose a trained model and then choose Clone from the Action drop-down menu list.
- For Model details, do the following:
  - Type RL_model_1 in Model name, if you don't want a name to be generated for the cloned model.
  - Optionally, give a description for the to-be-cloned model in Model description - optional.

Hyperparameters	**Description**
Gradient descent batch size	The number recent vehicle experiences sampled at random from an experience buffer and used for updating the underlying deep-learning neural network weights. Random sampling helps reduce correlations inherent in the input data. Use a larger batch size to promote more stable and smooth updates to the neural network weights, but be aware of the possibility that the training may be longer or slower.
Number of epochs	The number of passes through the training data to update the neural network weights during gradient descent. The training data corresponds to random samples from the experience buffer. Use a larger number of epochs to promote more stable updates, but expect a slower training. When the batch size is small, you can use a smaller number of epochs
Learning rate	During each update, a portion of the new weight can be from the gradient-descent (or ascent) contribution and the rest from the existing weight value. The learning rate controls how much a gradient-descent (or ascent) update contributes to the network weights. Use a higher learning rate to include more gradient-descent contributions for faster training, but be aware of the possibility that the expected reward may not converge if the learning rate is too large.
Entropy	A degree of uncertainty used to determine when to add randomness to the policy distribution. The added uncertainty helps the AWS DeepRacer vehicle explore the action space more broadly. A larger entropy value encourages the vehicle to explore the action space more thoroughly.

Hyperparameters	Description
Discount factor	A factor specifies how much of the future rewards contribute to the expected reward. The larger the Discount factor value is, the farther out contributions the vehicle considers to make a move and the slower the training. With the discount factor of 0.9, the vehicle includes rewards from an order of 10 future steps to make a move. With the discount factor of 0.999, the vehicle considers rewards from an order of 1000 future steps to make a move. The recommended discount factor values are 0.99, 0.999 and 0.9999.
Loss type	Type of the objective function used to update the network weights. A good training algorithm should make incremental changes to the agent's strategy so that it gradually transitions from taking random actions to taking strategic actions to increase reward. But if it makes too big a change then the training becomes unstable and the agent ends up not learning. The Huber loss and Mean squared error loss types behave similarly for small updates. But as the updates become larger, Huber loss takes smaller increments compared to Mean squared error loss. When you have convergence problems, use the Huber loss type. When convergence is good and you want to train faster, use the Mean squared error loss type.
Number of experience episodes between each policy-updating iteration	The size of the experience buffer used to draw training data from for learning policy network weights. An experience episode is a period in which the agent starts from a given starting point and ends up completing the track or going off the track. It consists of a sequence of experiences. Different episodes can have different lengths. For simple reinforcement-learning problems, a small experience buffer may be sufficient and learning is fast. For more complex problems that have more local maxima, a larger experience buffer is necessary to provide more uncorrelated data points. In this case, training is slower but more stable. The recommended values are 10, 20 and 40.

Summary

In total there are 13 parameters you can use in your reward function

- x and y - The position of the vehicle on the track
- heading - Orientation of the vehicle on the track
- waypoints - List of waypoint coordinates
- closest_waypoints - Index of the two closest waypoints to the vehicle
- progress - Percentage of track completed
- steps - Number of steps completed
- track_width - Width of the track
- distance_from_center - Distance from track center line
- is_left_of_center - Whether the vehicle is to the left of the center line
- all_wheels_on_track - Is the vehicle completely within the track boundary?
- speed - Observed speed of the vehicle
- steering_angle - Steering angle of the front wheels

For more information on these parameters and the values they can take, read the detailed documentation.

https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-console-train-evaluate-models.html#deepracer-reward-function-signature

Sample Reward Functions

Sample 1

This example created by Nick Sefiddashti shows how to add a reward based on Speed and distance from center of track.

```
  def reward_function(on_track, x, y, distance_from_center, car_orientation,
progress, steps, throttle, steering, track_width, waypoints,
closest_waypoint):
  import math
   if(on_track):
      reward = (1/((.5 * track_width) - abs(distance_from_center))) *
throttle * progress
  else:
      reward = 0
  return float(reward)
```

Sample 2

This example created by AWS and enhanced by Nick Sefiddashti shows how to add a reward for Steering around the track.

```
def reward_function(on_track, x, y, distance_from_center, car_orientation,
progress, steps, throttle, steering, track_width, waypoints,
closest_waypoint):
  '''
  @on_track (boolean) :: The vehicle is off-track if the front of the
vehicle is outside of the white
  lines
  @x (float range: [0, 1]) :: Fraction of where the car is along the x-axis.
1 indicates
  max 'x' value in the coordinate system.
  @y (float range: [0, 1]) :: Fraction of where the car is along the y-axis.
1 indicates
  max 'y' value in the coordinate system.
  @distance_from_center (float [0, track_width/2]) :: Displacement from the
center line of the track
  as defined by way points
  @car_orientation (float: [-3.14, 3.14]) :: yaw of the car with respect to
the car's x-axis in
  radians
  @progress (float: [0,1]) :: % of track complete
  @steps (int) :: numbers of steps completed
  @throttle :: (float) 0 to 1 (0 indicates stop, 1 max throttle)
  @steering :: (float) -1 to 1 (-1 is right, 1 is left)
  @track_width (float) :: width of the track (> 0)
  @waypoints (ordered list) :: list of waypoint in order; each waypoint is a
```

```
set of coordinates
  (x,y,yaw) that define a turning point
  @closest_waypoint (int) :: index of the closest waypoint (0-indexed) given
the car's x,y
  position as measured by the eucliedean distance
  @@output: @reward (float [-1e5, 1e5])
  '''
  import math
  # Example Centerline following reward function
  marker_1 = 0.1 * track_width
  marker_2 = 0.25 * track_width
  marker_3 = 0.5 * track_width
  reward = 1e-3
  if distance_from_center >= 0.0 and distance_from_center <= marker_1:
      if throttle > .75:
          if abs(steering) < .25:
              reward = 1
          else:
              reward = .9
      else:
          if abs(steering) < .25:
              reward = .8
          else:
              reward = .7
  elif distance_from_center <= marker_2:
      if throttle > .75:
          if abs(steering) < .25:
              reward = .6
          else:
              reward = .5
      else:
          if abs(steering) < .25:
              reward = .4
          else:
              reward = .3
  elif distance_from_center <= marker_3:
      if throttle > .75:
          if abs(steering) < .25:
              reward = .2
          else:
              reward = .1
      else:
          if abs(steering) < .25:
              reward = .05
          else:
              reward = 0
  else:
      reward = 1e-3  # likely crashed/ close to off track
      return float(reward)
```

Sample 3

Stay On Track In this example, we give a high reward for when the car stays on the track, and penalize if the car deviates from the track boundaries. This example uses the all_wheels_on_track, distance_from_center and track_width parameters to determine whether the car is on the track, and give a high reward if so. Since this function doesn't reward any specific kind of behavior besides staying on the track, an agent trained with this function may take a longer time to converge to any particular behavior.

```
def reward_function(params):
  '''
  Example of rewarding the agent to stay inside the two borders of the track
  '''
  # Read input parameters
  all_wheels_on_track = params['all_wheels_on_track']
  distance_from_center = params['distance_from_center']
  track_width = params['track_width']
  # Give a very low reward by default
  reward = 1e-3
  # Give a high reward if no wheels go off the track and
  # the agent is somewhere in between the track borders
  if all_wheels_on_track and (0.5*track_width - distance_from_center) >=
0.05:
      reward = 1.0
  # Always return a float value
  return float(reward)
```

Sample 4

Follow Center Line In this example we measure how far away the car is from the center of the track, and give a higher reward if the car is close to the center line. This example uses the track_width and distance_from_center parameters, and returns a decreasing reward the further the car is from the center of the track. This example is more specific about what kind of driving behavior to reward, so an agent trained with this function is likely to learn to follow the track very well. However, it is unlikely to learn any other behavior such as accelerating or braking for corners.

```
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)
```

Sample 5

No incentive An alternative strategy is to give a constant reward on each step, regardless of how the car is driving.

This example doesn't use any of the input parameters — instead it returns a constant reward of 1.0 on each step.

The agent's only incentive is to successfully finish the track, and it has no incentive to drive faster or follow any particular path. It may behave erratically.

However, since the reward function doesn't constrain the agent's behavior, it may be able to explore unexpected strategies and behaviors that turn out to perform well.

```
def reward_function(params):
  '''
  Example of no incentive
  '''
  # Always return 1 if the car does not crash
  return 1.0
```