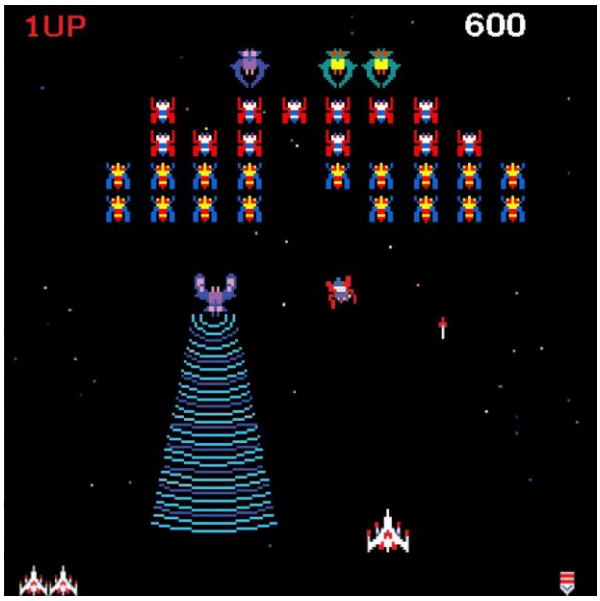
DeepRacer Lab 1/Introduction Document

Introducing the DeepRacer.

Introduction

The AWS DeepRacer is an electric vehicle designed specifically with machine learning capabilities in mind. The car is intended for use on a physical track and relies on its cameras and a network connection to a computer with a dedicated GPU. Despite looking like an RC car, the DeepRacer is intended to run autonomously via a generated algorithm. The algorithm that is used to run autonomously is generated by a special subset of **machine learning** called **reinforcement learning.**

DeepRacer vs. CyBot



The DeepRacer differs greatly from the CyBot. Mostly due to the processing power and sensors available to each machine. The DeepRacer relies on dual cameras as well as a lidar sensor and gyroscope. The DeepRacer also has wifi capabilities and a lot more processing power than the CyBot, having an Intel Atom Processor and 4 GB of RAM. The CyBot also has wifi capabilities but only the processing capabilities available to the Tiva C Series Launchpad microcontroller- an ARM Cortex M4 CPU and 256 KB of memory. The CyBot also has light sensors, left and right bump sensors, a ping sensor, and an infrared sensor. The DeepRacer needs a lot more processing power because of how complex its algorithms are. It needs to process data from its sensors and send it over wifi to a more powerful computer for processing. Based on this data it creates a new model for navigating the environment that is stored on the DeepRacer for future decision making.

What is Machine Learning?

Machine learning is process in which a model is generated that is able to predict a desired output based on some given input variables. A common example, which we will be using for this lab, is an autonomous car. In order to train a model, we first need to understand what is happening under the hood. The specific type of machine learning we will be focusing on is reinforcement learning.

What Parameter Estimation?

Parameter estimation is a good place to start in understanding machine learning. Below is an example to show that, when given an input and an output, we are able to estimate the policy that governs the system.

```
>> x = randn(100,5);
>> theta = [1 8 4 6 2];
>> y = x * theta';
>> y = y + randn(100,1)/100;
>> theta_hat = ((x'*x)^-1)*x'*y

theta_hat =

    1.0003
    8.0006
    4.0005
    5.9998
    2.0007
```

In this system, were are given 100 random sets of 5 input variables. We then define our ideal/true policy as the vector [1 8 4 6 2]. This was done arbitrarily for this example, but in the real world we would not have access to this information and would need to solely rely on our inputs and outputs. Our output is defined as x * transpose(theta) + noise which shows that even in a real environment with imperfect data, we can still estimate a policy.

We then apply a formula to estimate theta, denoted theta hat, given our input and output data. The formula used here is a pseudo-inverse formula, however there are other popular formulas such as the gradient decent formula, which can converge to a better result and can be more efficient for large data sets.

As can be seen, we are able to estimate the policy for our given system, which is the overall goal of machine learning. This example uses only 5 inputs and is quite simple, however more complex machine learning models are able to approximate many complex variables that may govern the system.

Another method for parameter estimation is the Stochastic Gradient Decent (SGD) method. This method follows the formula

$$\theta_j = \theta_j + \alpha\left(y^i - \hat{y}^i\right)x_j^i$$

Where θ_j is the j-th element of our true theta approximation, alpha is our learning rate (usually a small number), y^i is the i-th element of our output vector, and x_j^i is the j-th element of the i-th row of x. This equation uses the stochastic approximation of the gradient between y and $\hat{y}$ to always change θ in a way that minimizes the difference between y and $\hat{y}$.

View the Prelab.c document and try to understand how the code functions and play around with it. Change the values in θ or the amount of noise in the signal. What happens for each of these changes?

What happens when you decrease the samples in x?

What happens when you increase or decrease alpha?

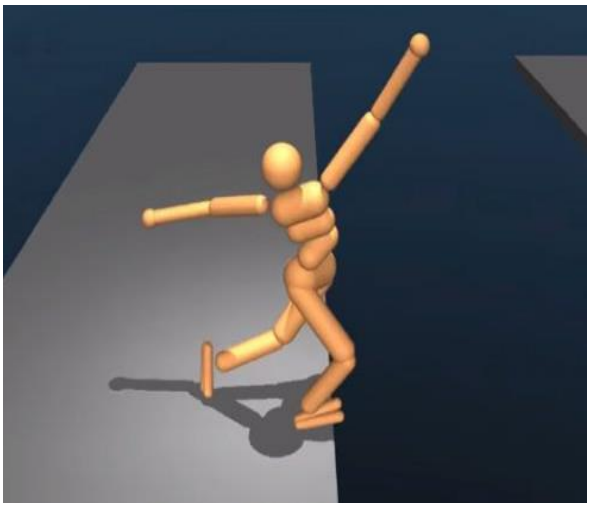
What happens if you add more entries to θ?

After seeing how the code functions, how can you apply what you have learned to your understanding of machine learning?

For more information into the SGD method of approximation, look at THIS Wikipedia article which goes more in depth into its derivation and mathematical basis. If you are very interested in the applications of the SGD in machine learning, HERE is an IEEE paper accessible through the university in which a version of the SGD, the 'Stochastic Gradient Descent with Momentum', is used in vehicle detection. Although it does not go into depth on the SGDm algorithm, it does help show that it is useful to understand as well as gives further insight into the how neural networks and convolutional neural networks function.

What is Reinforcement Learning?

Reinforcement learning is a style of machine learning is ideal for training autonomous models and storing temporal data as opposed to supervised learning, which is more ideal for models that rely on classifying and identifying information. This type of machine learning has also been used to teach computers how to play video games like Pac-Man, Galaga, and even teach computers how to walk in simulated environments.



This process requires two things on the part of the trainer. One is an environment in which the machine performs a task, and the other is a **policy**.

Reinforcement learning, at its heart, simply tries to maximize its efficiency for a given problem. We first model our agent, which in our case is the DeepRacer, as having certain states that it is able to be in. We will first assume a discrete state system, then move onto a continuous state system which is what we will use for the DeepRacer.

Discrete State System:

In a discrete system, our robot is constrained to a limited amount of choices it is allowed to make. The default action space for our robot is shown below.

AWS DeepRacer default discrete action space

Action number	Steering	Speed
0	-30 degrees	0.4 m/s
1	-30 degrees	0.8 m/s
2	-15 degrees	0.4 m/s
3	-15 degrees	0.8 m/s
4	0 degrees	0.4 m/s
5	0 degrees	0.8 m/s
6	15 degrees	0.4 m/s
7	15 degrees	0.8 m/s
8	30 degrees	0.4 m/s
9	30 degrees	0.8 m/s

This set of actions can be interpreted as a vector, which we will call A. We then interpret the environment around the bot to create another matrix of environment variables and states. These could include distance from the out of bounds line, angle relative to the track, or when and what the last state transition was. This set of variables will be called S.

The goal of our policy, which we will call π, is to create a set of equations which maximizes the reward defined in a reward function. This can be expressed by maximizing the equation $P(a_t = a \mid s_t = s)$, which maximizes the probability for a state change given our state variables s. An example of this is would be if we chose to reward our robot for staying on the middle line of the track. The robot would start with completely random states that with each training iteration would change slightly to slowly allow the robot to try multiple choices for different situations. It would slowly learn that if our current action was turning left and we were by the left boundary, the best state transition would be to turn away from the boundary. It would learn that the probability of gaining more points by doing that is high and thus our policy would be weighted in a way that next time the bot was close to the track boundary, it would turn away as our probability equation for receiving reward points would be high. With enough training, our policy will be able to recognize most environmental situations and, using the policy, determine what state transition provides the best chance of receiving reward points.

Continuous State System:

In a continuous state system, like what the robot we are using will have, there are some tradeoffs and advantages compared to the discrete state system. The largest disadvantage is the time needed for training. As we now must consider every possible speed and steering angle combination, there is much more complexity in computing the desired state transition. As we are able to train on more advanced hardware however, this is less of an issue and the benefits of this outweigh the cons. The first benefit is training potential. As the robot can now choose any combination of speed and steering, it can find the optimal choice to maximize its reward whereas in a discrete system it may need to compromise and thus the model will be worse. The second big advantage of a continuous state system is the options allowed when making a reward function. We are now able to set the rate of change of our angle and speed in the reward function which may incentivize faster acceleration as well as the ability to stop zigzagging which can waste time. In a discrete system, the only option would be to incentivize or penalize state transitions however this also has the negative effect of then simultaneously penalizing necessary state transitions. While these cons can all be fixed using proper coding techniques and smart engineering, it is more efficient for our goal to use a continuous time state system.

How to use the DeepRacer

Step 1: DeepLearner GUI

Refer to the “Using DeepLearner” manual on getting started with the deep learning environment.

Step 2: The Reward Function

The reward function is necessary for reinforcement learning, it tells the model what outcomes are desirable, what is undesirable, and gives a weight to certain outcomes to be able to prioritize certain things. For example, here is a reward function that is given by Amazon to prioritize object avoidance:

```
# This is the default Object Avoidance Function

def reward_function(params):
    '''
    Example of rewarding the agent to stay inside two borders
    and penalizing getting too close to the objects in front
    '''

    all_wheels_on_track = params['all_wheels_on_track']
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    objects_distance = params['objects_distance']
    _, next_object_index = params['closest_objects']
    objects_left_of_center = params['objects_left_of_center']
    is_left_of_center = params['is_left_of_center']

    # Initialize reward with a small number but not zero
    # because zero means off-track or crashed
    reward = 1e-3

    # Reward if the agent stays inside the two borders of the track
    if all_wheels_on_track and (0.5 * track_width - distance_from_center) >= 0.05:
        reward_lane = 1.0
    else:
        reward_lane = 1e-3

    # Penalize if the agent is too close to the next object
    reward_avoid = 1.0

    # Distance to the next object
    distance_closest_object = objects_distance[next_object_index]
    # Decide if the agent and the next object is on the same lane
    is_same_lane = objects_left_of_center[next_object_index] == is_left_of_center

    if is_same_lane:
        if 0.5 <= distance_closest_object < 0.8:
            reward_avoid *= 0.5
        elif 0.3 <= distance_closest_object < 0.5:
            reward_avoid *= 0.2
        elif distance_closest_object < 0.3:
            reward_avoid = 1e-3 # Likely crashed

    # Calculate reward by putting different weights on
    # the two aspects above
    reward += 1.0 * reward_lane + 4.0 * reward_avoid

    return reward
```

The way that the DeepRacer uses the reward function is it will perform random actions at first, then evaluate it's actions several times each second based on this reward function. Then the data gathered is stored into a **convolutional neural network** in order to help the DeepRacer make better decisions in the future.

Task: Use the given reward function or create your own to train a DeepRacer into successfully completing a lap around the given track.

Step 3: Training and Racing

Task: Using the Reward function created in the previous step, train model for 20 minutes and save the model to race and describe how well it performs. Next, continue the training for another 20 minutes (Or longer if you prefer) and again save the model. Now, race the two models and report on the difference the extra training time made in the race.

Training Iteration	Reward graph	Lap Time	Description
1. 20 Minute Model			
2. 40+ Minute Model			

What differences did you observe?

Optional: Step 4: Race a friend

Model	Reward graph	Lap Time	Description
You			
Friend			

How were your reward functions different? Did you focus on different aspects such as staying close to the center or speed?

In what ways did your reward function outperform your friend’s reward function?

In what ways did your reward function underperform when compared to your friend's reward function?

Would you change anything about your reward function after comparing it to your friend’s?

Step 5: Ending the Program

Before you leave, please be sure to **STOP ALL TRAINING PROCESSES** and save your models. If you don’t, the server will automatically stop any processes currently running and **WILL NOT save that model**.