EECS 495
Deep Reinforcement Learning from Scratch

Train DonkeyCar in Unity Simulator with Reinforcement Learning

Siddharth Bhola
Rekha Goverthanam

1. Motivation

The most common methods to train the car to perform self-driving are behavioral cloning and line following. On a high level, behavioral cloning works by using a convolutional neural network to learn a mapping between car images (taken by the front camera) and steering angle and throttle values through supervised learning. The other method, line following, works by using computer vision techniques to track the middle line and utilizes a PID controller to get the car to follow the line. We thought that self-driving simulation is a perfect use-case for applying an advanced Reinforcement Learning algorithm called Deep Q-Learning.

2. Environment details:

DonkeyCar: An opensource DIY self-driving platform for small scale cars consisting of both hardware and software stack which allows us to run our custom trained models in RC cars.

Donkey Car Simulator for interaction between Unity and Python:
Tawn Kramer has created a high fidelity DonkeyCar simulator in Unity. This simulator comes with python code to communicate with Unity which is done through WebSocket protocol which allows bidirectional communication between client and server. In our case, our python "server" can push messages directly to Unity (e.g. steering and throttle actions), and our Unity "client" can also push information (e.g. states and rewards) back to the python server. We are using the "Generated Road Scene" available in this simulator which creates a randomly generated road so that you can have miles of curves on different road surfaces.

All the library dependencies are added in requirements.txt and can be run by this command: `pip install -r requirements.txt`

3 . Double Deep Q Learning (DDQN)

Introduction
In Deep-Q Learning, the Target *y_i* and *Q(s,a)* are estimated separately by two different neural networks, which are often called the Target, and Q-Networks. However, Deep Q-learning is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer

overestimated to underestimated values, as it can be seen in the calculation of the TD-Target ***y_i*** *below*:

$$y_i := \mathbb{E}_{a' \sim \pi} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \right]$$

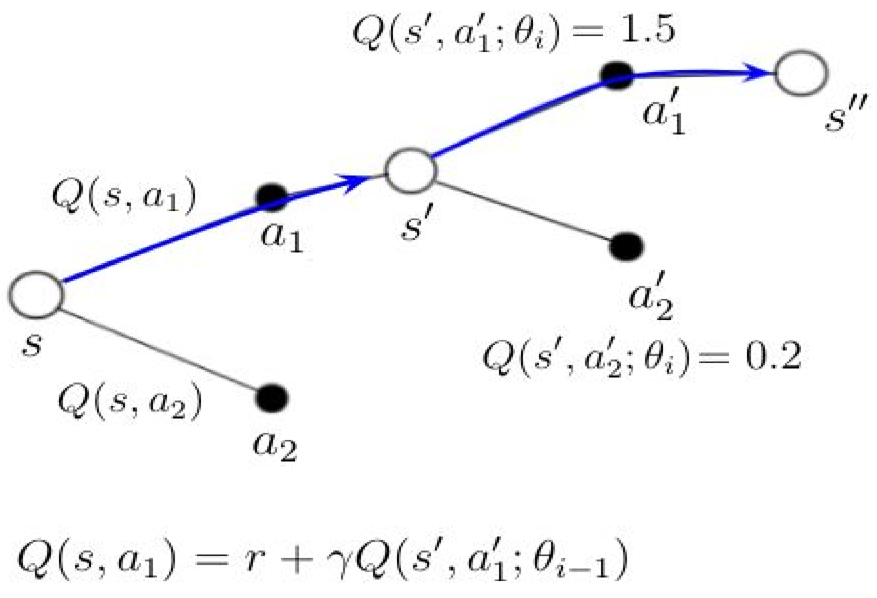
The idea of **Double Q-learning** is to reduce overestimations by decomposing the *max* operation in the target into **action selection** and **action evaluation**. The Target-Network calculates ***Q(s, a_i)*** for each possible action ***a_i*** in state ***s***. The greedy policy decides upon the highest values ***Q(s, a_i)*** which action ***a_i*** to select. This means that the Target-Network **selects** the action ***a_i*** and at the same time evaluates its quality by calculating ***Q(s, a_i).*** Double Q-Learning tries to decouple this both procedures from each other.

In **Double Q-Learning** the TD-Target looks as follows:

$$y_i^{\textbf{DoubleQ}} = \mathbb{E}_{a' \sim \mu} \left[r + \gamma Q(s', \arg\max_a Q(s', a; \theta_i); \theta_{i-1}) | S_t = s, A_t = a \right]$$

As you can see the max operation in the target is gone. While the Target-Network with parameters ***θ(i-1)*** evaluates the quality of the action, the action itself is determined by the Q-Network that has parameters ***θ(i).*** This procedure is in contrast to the vanilla implementation of Deep Q-Learning where the Target-Network was responsible for action selection and evaluation.

Illustration of Double Q-Learning:



$$Q(s, a_1) = r + \gamma Q(s', a'_1; \theta_{i-1})$$

An AI agent is at the start in state ***s***. He knows based on some previous calculations the qualities ***Q(s, a_1)*** and ***Q(s, a_2)*** for possible two actions in those states. He decides to take action ***a_1*** and ends up in state ***s'***.
The Q-Network calculates the qualities ***Q(s', a_1')*** and ***Q(s, a_2')*** for possible actions in this new state. Action ***a_1'*** is picked because it results in the highest quality according to the Q-Network.

State Space

Here state is the pixel images taken by the front camera of the Donkey car, we pre-processed the images and performed the following transformations

1. Resize it from **(120,160)** to **(80,80)**
2. Turn it into **grayscale**
3. **Frame stacking**: Stack 4 frames from previous time steps together

   Frame stacking is done to let the agent choose action based on the prior sequence of driving frames too and **Frame Skipping** is set to 2.

The final state is of dimension **(1,80,80,4)**

Pre-processing Images

Since we give the full pixel camera images as inputs, it might overfit to the background patterns instead of recognizing the lane lines. When we try our model in the real world, we should get the agent to neglect the background noise and just focus on the track lines. To achieve that we followed the below procedure inspired by this blog

1. Detect and extract all edges using Canny Edge Detector
2. Identify the straight lines through Hough Line Transform
3. Separate the straight lines into positive sloped and negative sloped (candidates for left and right lines of the track)
4. Reject all the straight lines that do not belong to the track utilizing slope information

Action Space

We have two actions, steering control and throttle control. But for simplicity, we set throttle value as constant, which is 0.7 and opt only to control the steering. Donkey car simulator can take continuous steering value, which ranges from -1 to 1. However, DQN can only handle discrete value, so we discretized the steering value into 15 categorical bins. In the bins where only one item is set to 1, which represents the linear value, and all other items set to 0.

Q Network Architecture

Our Q network is a **3-layer convolutional neural network** that takes $80 \times 80 \times 4$ stacked frame states as input and output 15 values representing the 15 discretized steering categories. We have used ReLu for activation. For exploration, we used epsilon which is initialized with 1 and by the end of training, it will be gradually reduced to 0.2 to let lots of actions with the maximum q-values are get picked (exploitation).

Few parameter values:

Discount_factor = 0.99
Learning_rate = 0.0001
Epsilon
    Train = 1.0
    Test = 10-6
Epsilon_min = 0.02
Batch_size = 64
Episode = 10000
Frame Skipping = 2

Below is our Q-Network Architecture

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
conv1	$80 \times 80 \times 4$	8×8	4	32	ReLu	$20 \times 20 \times 32$
conv2	$20 \times 20 \times 32$	4×4	2	64	ReLu	$9 \times 9 \times 64$
conv3	$9 \times 9 \times 64$	3×3	1	64	ReLu	$7 \times 7 \times 64$
fc4	$7 \times 7 \times 64$			512	ReLu	512
fc5	512			15	Linear	15

Rewards

Reward is a function of cross track error (cte) which is provided by the Unity environment. Cross track error measures the distance between the center of the track and car. Our shaped reward is given by the following formula:

$$reward = 1 - (abs(cte)/max\_cte)$$

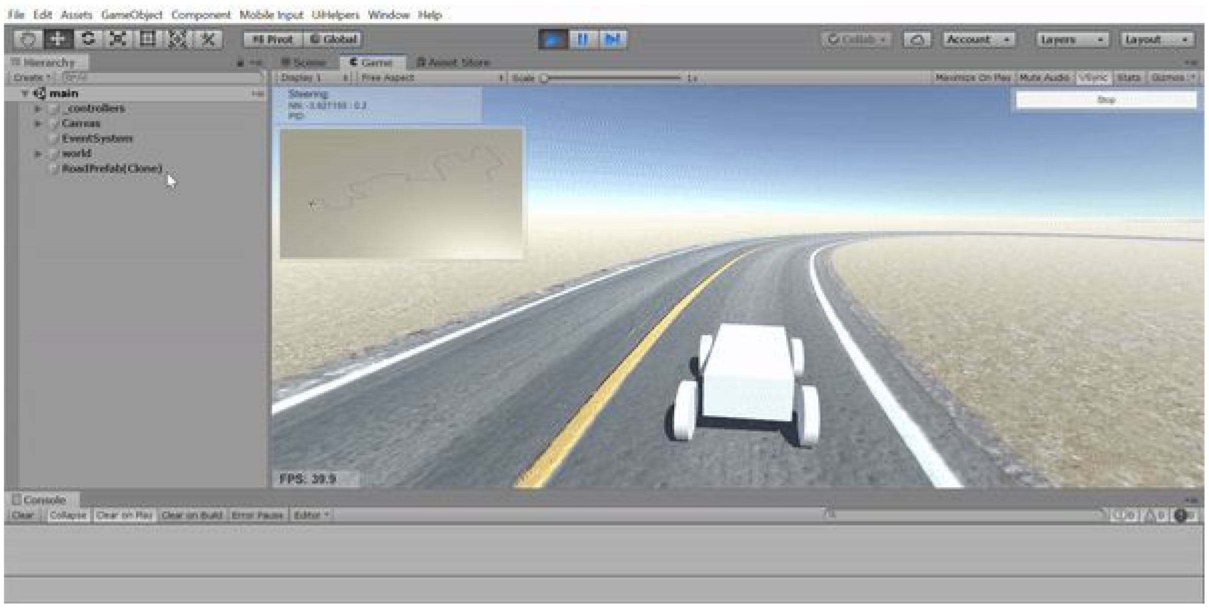
Where max_cte is just a normalizing constant so that the reward is within the range of 0 and 1. We terminate the episode if $abs(cte)$ is larger than max_cte

Other variables

Memory replay buffer (i.e. storing <state, action, reward, next_state> tuples) has a capacity of 10000. **Target Q network** is updated at the end of each episode. **Epsilon-greedy** is used for exploration.

Results

First, we trained our model for 100 episodes in a Jupyter Notebook on a single CPU. Next, we trained our model in AWS with a GPU Instance of p2.xlarge for 10000 episodes. Below is our model after training of 10000 episodes. We can notice the car learned to drive and stayed at the center of the track most of the time.



The car is a bit wriggle, we think this is because when we preprocess the image we lost some of the useful background information and line curvature information which makes the car wriggle especially when making turns. But in positive note, our model is less prone to overfitting and can even be generalized to real-world tracks.

Future Scope

In this implementation, we kept constant throttle value of 0.7 and only generating the value of steering control. In the next phase, we will try to let the agent learn to output a throttle value as well to optimize vehicle speed**.** And we will also try to transfer our trained model from the simulator to the real world.

References:

1. https://arxiv.org/pdf/1509.06461.pdf
2. https://www.intel.ai/demystifying-deep-reinforcement-learning/#gs.igrn5d
3. https://github.com/tawnkramer/sdsandbox
4. https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html
5. https://flyyufelix.github.io/2018/09/11/donkey-rl-simulation.html
6. https://docs.donkeycar.com/guide/simulator/
7. https://towardsdatascience.com/deep-double-q-learning-7fca410b193a