Uppsala Universitet

Natural Computation Methods for Machine Learning

Self-Driving Car in Unity

UPPSALA
UNIVERSITET

August 31, 2020

Venkata Sai Teja Mogillapalle, Dhanush Kumar Akunuri

Contents

1 Introduction **2**

2 Background **3**
2.1 Q-Learning 3
2.2 Deep Q-Learning 3
2.2.1 Experience Replay 4
2.2.2 Epsilon greedy Strategy 4
2.2.3 Fixed Targets Model 4
2.3 Double-Deep Q-Learning 5

3 Environment **5**
3.1 Unity 5
3.2 Donkey Car 5

4 Implementation **6**
4.1 Pre-Processing 6
4.2 Reward Function 8
4.3 Convolutional Neural Network 9

5 Results and Comparison **10**

6 Future Work **12**

List of References **13**

1 Introduction

Self-driving cars have gained a lot of attention during the last decade. Many big companies are coming forward to develop a complete human free self-driving car. There are some solutions which use supervised learning for developing a self-driving car. This can lead to inclusion of human bias for training the model and this might not work well in the real world. Reinforcement learning would work great for this type of problems. The agent/car doesn't need to have any prior information of the environment and it can learn everything by itself. But the car can learn how to drive only after exploring and exploiting the environment and learning from the mistakes. Training our car in the real-world would be extremely dangerous as they may prone to many accidents. Virtual Environments like Unity would be a great help for this. We can use the virtual environments for training the self-driving car and the developed model can be transferred to the real world. This is much better than training the car from scratch in the real world.

In this project we have used reinforcement learning algorithms called Deep Q-learning and double Deep Q-learning for training the car in Unity environment and compared the results of the two algorithms. CNN has been used as the Neural network model for the deep Q-learning. We have used a very simple environment which doesn't have any obstacles but the track/path has many turns and the model is trained on many different paths. The aim of the project is to make the car stay on the track for a long time without going off-track while the speed is kept constant.

2 Background

The goal of any reinforcement learning algorithm is to find an optimal policy which maximizes the expected return(sum of rewards) received by the agent.

2.1 Q-Learning

Q-Learning is a Reinforcement learning technique which is used to find an optimal policy in the case of MDP (Marchv Decision Process). The standard q-learning stores the Q-values in Q-tables.
The Q-learning update rule is given by:

$$Q(s,a) = Q(s,a) + \eta(r + \gamma max_{a' \in A} Q(s',a') - Q(s,a)) \tag{1}$$

where,

- Q(s, a) is the value of the current state-action pair.
- $'\eta'$ is the learning rate, it defines how much we accept the new value compared to the old value. This the value which essentially gets added to our current value and helps us move towards the direction of the latest update.
- $'\gamma'$ is the discount factor, it quantifies the importance we are giving for the future reward. it is also used to approximate the noise in the future rewards. Varies in the range from 0 to if it is close to zero then the agent will consider immediate rewards. if not, then it will consider future rewards.
- 'r' is the reward.
- $max_{a' \in A} Q(s',a')$ is the maximum value of all possible state-action pairs found in the state the agent is entering.
- The updated Q(s, a) will be the old value Q(s, a) plus the error, $E = r + \gamma max_{a' \in A} Q(s',a') - Q(s,a)$ multiplied by the learning rate . So in a sense the agent learns by estimating the new value of Q(s, a) by considering the error between the current state Q(s, a) at time t and the next state Q(s',a') at time t + 1.

2.2 Deep Q-Learning

Q-Learning works fine in the case of small state-space environment. But in the case of complex environments, the performance drops off greatly. A deep neural network can be used to approximate the Q-values instead of Q-tables. This process of combining Q-learning and deep neural networks for estimating the Q-values is

known as Deep Q learning. CNN(Convolutional Neural Network) is used as the deep neural network for our project as we would be dealing with image inputs. We shall add several techniques on top of the deep Q-learning for better performance.

2.2.1 Experience Replay

We used a technique called experienced replay while training our network. Experience of an agent is defined by the tuple,

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \tag{2}$$

where 's' stands for the state, 'a' stands for action, 'r' stands for reward and 't' represents the time. We store the latest 'n' values of the experience in a replay memory. Each time a small batch from the replay memory is taken for training. Selecting this small batch is done randomly. The main advantage of using the experience replay is it breaks the correlation that occurs when we try to train continuous/consecutive samples. Learning only from the consecutive samples leads to improper training of the model.

2.2.2 Epsilon greedy Strategy

For obtaining a balance between exploration and exploitation, a strategy called Epsilon greedy is used. In the method, an exploration rate ϵ will be used which is initially set to 1. When it is set to 1, our agent will only explore the environment. Epsilon value will be updated for each episode and it starts decaying every time so that it becomes greedy to exploit the environment instead of exploring it.

2.2.3 Fixed Targets Model

Instead of a single network, here we use two networks for the training process. The second network is used to generate the target values which can be used for calculating the loss(between output Q values and target Q values) during the training process. This technique of using different target network was introduced by Timothy et al. 2019 [1]. The reason for using a different network for targets is that, if we use the same network for target value calculations, during every step of training, as the weights of the main network gets updated, along with the Q-values, the target Q-values are also updated. Eachtime we update our weights, our Q-values tries to move closer to the target Q -values but the target Q-values are also updated in the same direction as we are using same network for calculating them. So, our Q-values will never converge and they sometimes lead to explosion of values. To overcome this problem, a separate network(a clone of the policy

network) for the target values is maintained. The weights of this target network are frozen with the weights of the policy network and are updated with the policy network's new weights every certain amount of timesteps. This process makes our model more stable than the deep Q-learning model with a single deep neural network.

In addition to deep Q learning, we also implement double deep Q-learning algorithm.

2.3 Double-Deep Q-Learning

Double deep q network was introduced to reduce the overestimates of the Q values generated in the deep Q network. The double deep Q network from Hasselt et al.2016 [2] proposed a simple trick for solving the problems of deep Q network. The difference between our modified deep Q network and the double deep network has only a simple change during the training. Our deep Q learning model mentioned above uses max Q-values when computing the target Q-values during our training step. But here, we chose the action from the primary network and we use our target network to calculate the corresponding Q value for the action selected. This simple step will reduce the overestimates of deep Q network and makes training much faster.

We used both the updated deep Q learning and double deep-Q learning for training our self-driving car.

3 Environment

3.1 Unity

Unity3D is a game development engine which can be used to design 3D,2D game environments and simulations.Unity has Unity Asset store where we can find assets like cars,roads, buildings and many more. The kart which we used is a pre-built 3d asset which already contains implementation of steering behaviour, braking and physics for car.

3.2 Donkey Car

Donkey car is an open-source DIY self-driving platform for small RC cars which is built on the self-driving sandbox donkey simulator. This simulator is developed using the Unity Game Engine and Open AI-GYM. Open AI GYM is an open-source toolkit mainly used for reinforcement learning. Donkey car uses their internal physics and graphics but it is also flexible for making changes in the environment.

We have used websocket protocol which enables a bi-directional communication between the client and the server with very low latency. Here the donkey car in Unity acts as a client which pushes states and rewards to the server whereas the server (Python) pushes steering angle and throttle actions. The Donkey car has 3 scenes by default created by [3] in which one of it is shown in fig(3.2).

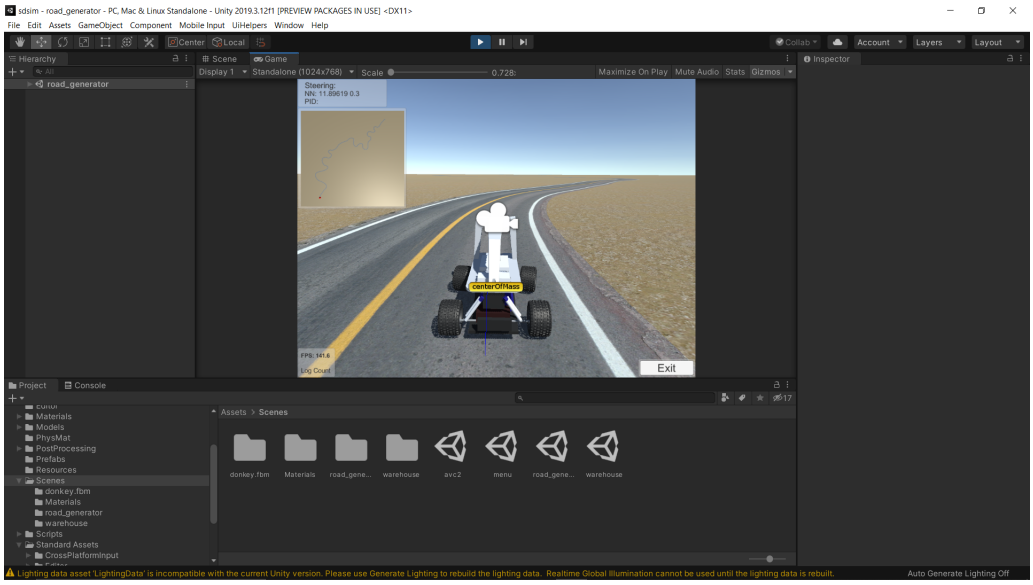

Figure 1: Donkey car running in Unity.

4 Implementation

The Reinforcement Learning agent in our project is the car. A camera is attached to the car which is used for generating the input states to our algorithm at each time step. The input images are pre-processed and passed on to our CNN which outputs the angle(action) the car has to take based on the given input and the reward function. This value is passed back to the car and it moves in the direction(angle) provided by the neural network. All the actions mentioned here happen at real-time. The input images captured undergo some pre-processing before passing to the neural network.

4.1 Pre-Processing

Since we are using images as the input states and we are generate an image at each time step, we shall be processing many hundreds, if not thousands of images. And since the images captured are color images, it becomes computationally very

expensive if we do not pre-process them. Our track contains two lines along the edges of the road and one line in the middle of the track. Our agent's current state is determined by the orientation of the lines present on the roads. So we can take the information about lines and leave the remaining data present in the images. For this, we created an image pre-processing pipeline which neglects the background, detects the lines and differentiates the lines based on its position.

The pipeline consists of the following tasks:

1. Removing tiny details and noise using gaussian blur.

2. Converting the colour image to greyscale.

3. Detecting the edges using canny edge detection. It is often used to detect edges by looking at quick changes in colors between a pixel and its neighbour [4]. We also removed the unnecessary details in the image using region of interest. Since the movement of an agent is completely dependent on its location, we ignored the background and considered only the bottom portion containing lanes.Noise removal and the greyscale conversion in previous steps have helped to differentiate the middle lines more clearly.

4. We then used probabilistic Hough lines to identify the location of lanes on the roads. It basically extracts lines passing through each of our edge points and group them by similarity [4]. 'HoughLinesP' is a function in OpenCV which returns an array of lines organized by endpoints.

5. The Hough transform might give multiple lines but we only want two distinct lanes to drive in between. This is done by arranging the lines with respect to its slope, the ones positive slope is assumed as the right lane and the negative slope as the left one. We then filter out lines with unacceptable slopes that throw off the intended slope of each line.The final image that is given as input to the neural network is as shown in the fig(4.1).

The resultant image contains 2 lines, these are images are stacked up with 4 successive frames and passed to the neural network as input.

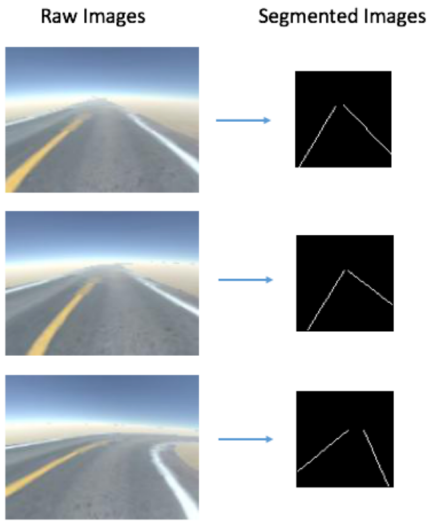

Figure 2: Images on the left are captured by camera, the right ones are pre-processed.

4.2 Reward Function

We calculated the reward function based on CTE. CTE stands for Cross Track Error which is provided by the Unity environment. It gives how far our car is from the centre of the track. We experimented with 2 different reward function for our environment. The first reward function is given by:

$$Reward1 = 1 - (abs(cte)/max\_cte) \tag{3}$$

We kept the value of max_cte as a constant value. The fist reward value always comes between [0,1].

In the case of the second reward function, we did not use the max_cte variable and we only calculated the reward based on the previous value of cte and the current cte. Previous cte refers to the cte value obtained in the previous step. So, this reward function is capable of producing both positive and negative reward values unlike the first case where we used only the positive reward values for calculation.

$$Reward2 = abs(prev\_cte) - abs(cte) \tag{4}$$

In both cases, we terminate an episode if the abs(cte) is larger than max_cte.
give a fixed negative reward in the case an episode is reset.Since we kept the value

of velocity/throttle as constant, we are not including the velocity in the reward function.
This reward function is only used for calculating the angle of the car and not the throttle value. Throttle value is always kept constant.

4.3 Convolutional Neural Network

Since the state inputs in our case are images, we will be using CNN instead of normal neural networks for estimating the Q values for each state-action pair. After pre-processing the images, we shall stack up 4 images and give them as input to the neural network instead of a single image. This process will make the network to understand the state of the environment in a better way.The architecture we used was inspired by NVIDIA paper [5].The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers.

5 Results and Comparison

We worked with 2 different scenarios (different reward functions) and with 2 different algorithms. Episode length indicates the time until which the self-driving car was running before getting reset. A Higher value of the episode length indicates that self-driving car was moving on the track for longer time.

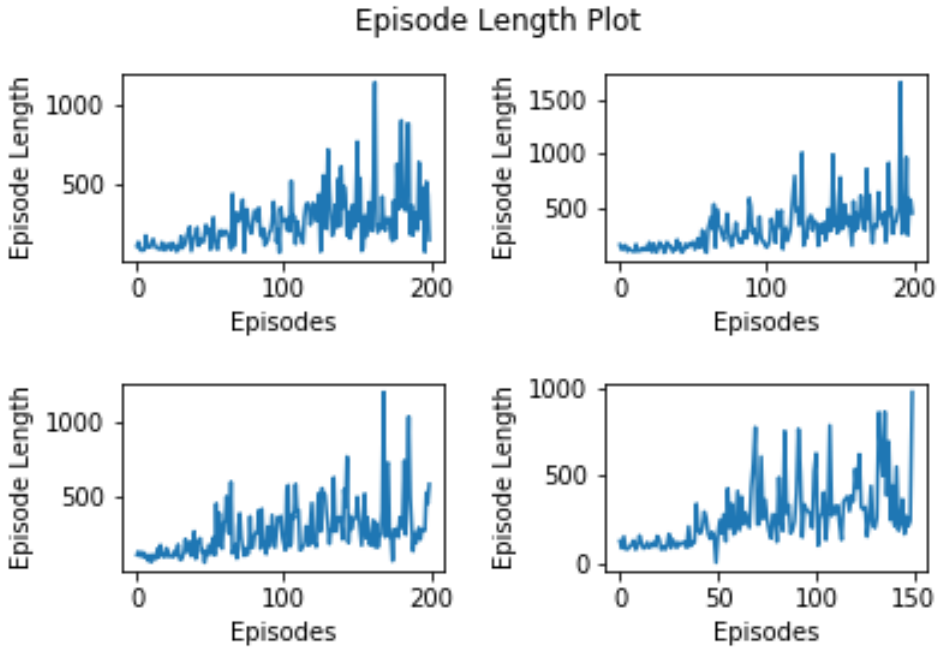

Figure 3: Top left plot shows the episode length plotted for deep q learning implementation Reward1(3). Top right is the same with reward 2. Bottom left plot is the double deep q-learning with Reward1(3) function. Bottom right is with ddqn and Reward2(4)

.

We can observe that episode lengths for bottom-right plot (double deep q learning with reward 2) produced more number of spikes in the plot after some 70 episodes. Which indicates that the car was able to stay on the track for a longer amount of time without getting reset

The average q value for different experimental set-ups are plotted below:

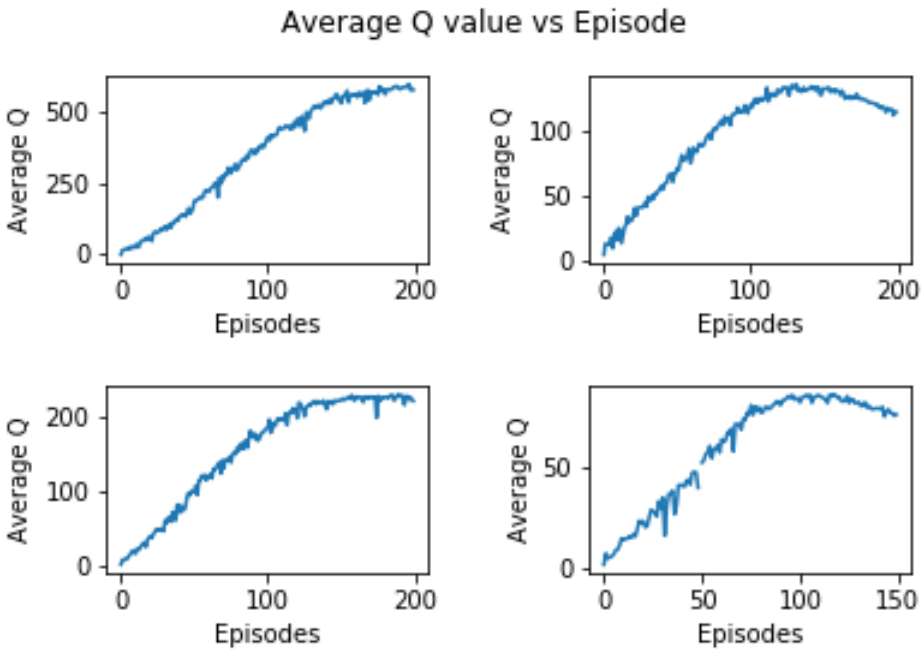

Figure 4: Top left plot shows the Average q plotted for deep q learning implementation with first reward function. Top right is the same set-up with reward 2. Bottom left plot is the double deep q learning with Reward1(3) function. Bottom right is with ddqn and Reward2(4).

Even in this case, we can see that the double deep q learning algorithm with reward function 2 is converging faster(less than 100 episodes) than the rest. We can observe that using reward function 2, the average Q values for both deep q learning and double deep q learning were almost flattened after 100 episodes and for reward function 1, it took a little longer for converging.

On average it took around 3-4 hours for each of the experiment set up to train. We trained on NVidea GTX 1060 with 6GB graphics and with 16 GB RAM and Intel i7 processor. Every training set-up took almost the same time for completing the training. We kept a limit of 200 episodes for each training set-up (except for the last set-up where we set a limit of 150 episodes because of time constraints).

6 Future Work

In our paper, we implemented the deep q-learning and double deep-q learning models for finding only the angle the car(by keeping the throttle fixed) has to take in order to stay on the track. We can extend this to add velocity component to the car and train our car to make it move faster on straight tracks and a bit slow when taking turns. We have experimented with simple tracks which contains a lot turns but we haven't included any obstacles on it. We can also test by adding some obstacles on the road and making the car avoid those obstacles. We can also train our car with some dynamic obstacle and see how the car reacts to those obstacles.

References

[1] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap, Jonathan J. Hunt. Continuous control with deep reinforcement learning. https://arxiv.org/pdf/1509.02971.pdf, 2019.

[2] Arthur Guez Hado van Hasselt and Google DeepMind David Silver. Deep reinforcement learning with double q-learning. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847, 2016.

[3] Tawn Kramer. Openai gym environments for donkey car. https://github.com/tawnkramer/gym-donkeycar, 2018.

[4] Laurent Desegur. A lane detection approach for self-driving vehicles. https://medium.com/@ldesegur/a-lane-detection-approach-for-self-driving-vehicles-c5ae1679f7ee, 2017.

[5] Daniel Dworakowski Mariusz Bojarski, Davide Del Testa. End to end learning for self-driving cars. https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf, 2016.