Uppsala Universitet

Natural Computation Methods for Machine Learning

Self-Driving Car in Unity

UPPSALA UNIVERSITET

June 1, 2020

Venkata Sai Teja Mogillapalle, Dhanush Kumar Akunuri

Contents

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. This is extremely difficult to implement and it is extremely dangerous to make the real self-driving car which is still in the training phase to test in the real world. Implementing self-driving cars in virtual environments like Unity would be a great start if the aim is to develop a fully functional self-driving car. We use Reinforcement learning algorithms to make the car learn by itself (instead of some supervised learning) and try to move in a simple track with lots of turns on the way. The aim of the project is to make the car stay on the track for a long time without going off-track.

2 Environment

2.1 Open AI and GYM

Open AI GYM is an open-source toolkit mainly used for reinforcement learning. It supports teaching an agent from walking to playing games, like ping pong, etc [1]. Open AI GYM supports many high computational libraries like Keras, TensorFlow and Theano. This toolkit has many environments designed for implementing Reinforcement learning such as Donkey Cars, Atari, Box2d. OpenAI GYM lets the agent learn through episodes, where each episode ends when the agent reaches the terminal state.

2.2 Donkey Car

Donkey car is an open-source DIY self-driving platform for small RC cars which is built around the self-driving sandbox donkey simulator. This simulator is developed using the Unity Game Engine and Open AI-GYM, it uses their internal physics and graphics and also it is flexible for making changes in the environment. Meanwhile Donkey car has 3 scenes by default created by [2] in which one of it is shown in fig(2.2).we used Websocket protocol which enables a bidirectional connection between the client and the server with very low latency. Here the donkey car acts as a client which pushes states and rewards to the client whereas the server (Python) pushes steering angle and throttle actions.

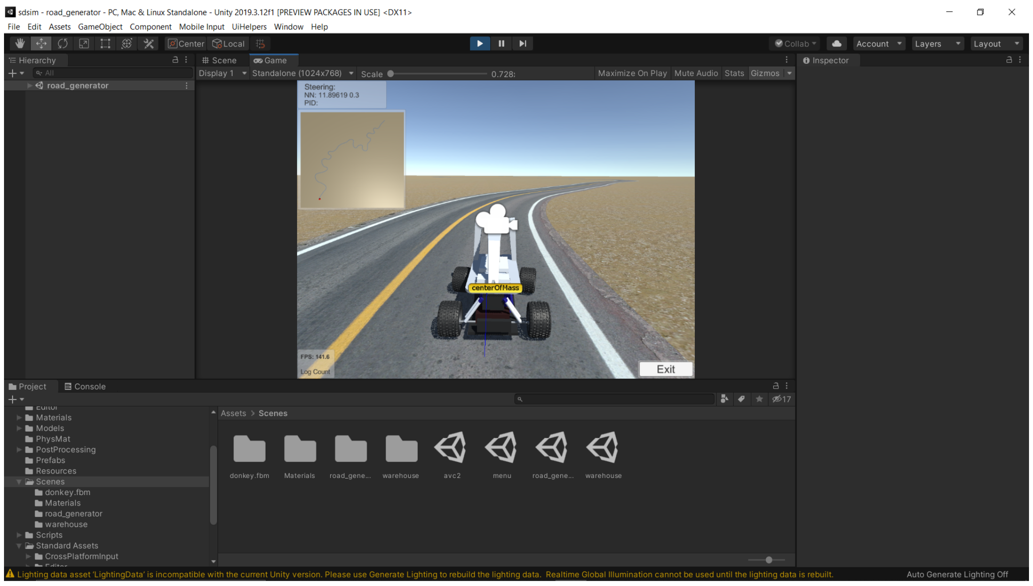
sdsim - road_generator - PC, Mac & Linux Standalone - Unity 2019.3.12f1 [PREVIEW PACKAGES IN USE] <DX11>
File Edit Assets GameObject Component Mobile Input UiHelpers Window Help
Hierarchy
road_generator
Scene
Game
Inspector
Steering:
NN: 11.89619 0.3
PID:
centerOfMass
Exit
Project
Console
Assets > Scenes
donkey.fbm
Materials
road_gene...
warehouse
avc2
menu
road_gene...
warehouse
Auto Generate Lighting Off

Figure 1: Donkey car running in Unity.

3 Background

3.1 Q-Learning

Q-Learning is a Reinforcement Learning which is used to find the optimal policy in the case of MDP (Marchv Decision Process). The usual q-learning stores the Q-values in Q-tables.

3.2 Deep Q-Learning

Q-Learning works fine in the case of small state-space environment. But in the case of very complex environments, the performance drops off greatly. We can modify the algorithm of Q-learning by using deep-neural networks for estimating the Q-values. This process of combining Q-learning and deep neural networks for estimating the Q-values is known as Deep Q learning. We will do several changes for Deep Q-learning to make it suit for our environment and train better and faster. The first modification is that instead of using deep neural network, we shall be using a CNN as we have images as inputs to the model. And also, several modifications to the traditional deep Q learning are made.

3.2.1 Experience Replay

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

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

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 and it is called 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.

3.2.2 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 which helps in the training process. This technique of using different target network was introduced by Timothy et al. 2019 [3]. 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 values of the main network shifts, the values of the target network will also shift with the same amount and we shall never be able to reduce the error. Because of this, value estimations can get out of control and might produce too huge values.

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

3.3 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 [4] 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 step.
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.

$$Y_t^{DOUBLEDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_{\alpha} Q(s_{t+1}, a; \theta_t), \theta_t^-) \qquad (2)$$

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

4 Implementation

4.1 Pre-Processing

The input for the neural network is an image which is captured by the camera but it is computationally expensive to use a full-sized colour image. Also, the output of the agent is completely dependent on the location and orientation of the lines present on the roads. 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. Remove tiny details and noise using gaussian blur.
2. Converted colour channel image to greyscale.
3. Detect the edges using canny edge detection, which is often used to detect edges by looking for quick changes in colour between a pixel and its neighbour [5]. 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 [5]. 'HoughLinesP' is a function in OpenCV which returns an array of lines organized by endpoints.
5. The Hough transform gives 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.

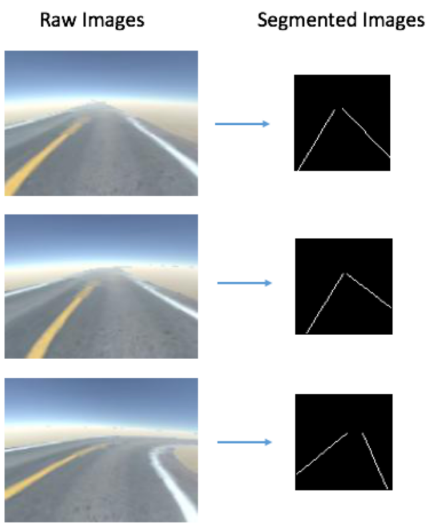

Figure 2: Images on the left are captured by camera, the right ones are pre-processed.

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. We 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.

4.2.1 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.

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.

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 running for longer time. The plot for the following is shown below :

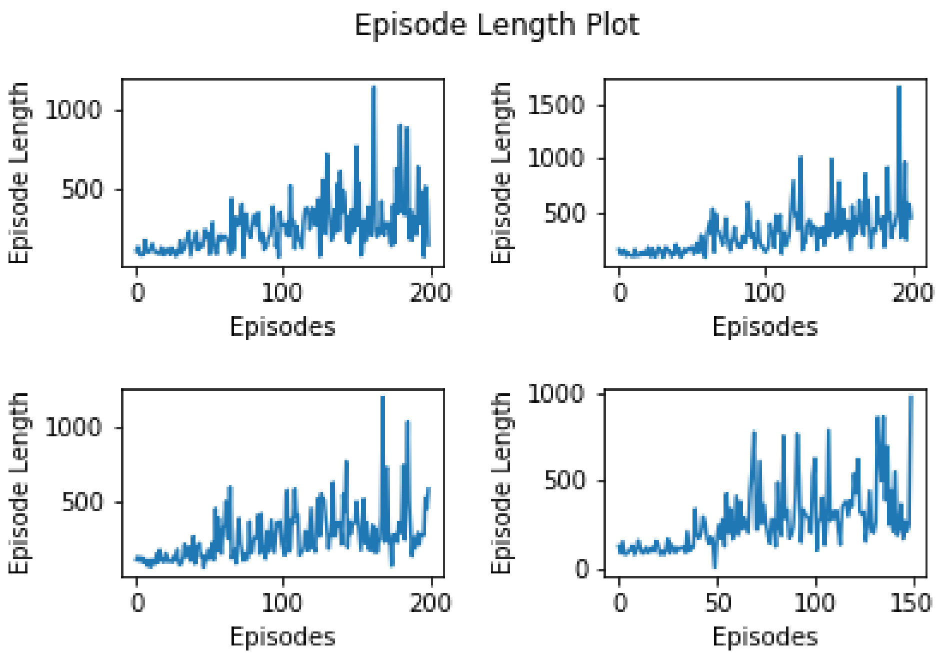

Figure 3: Top left plot shows the episode length plotted for deep q learning implementation for first reward function. Top right is the same with reward 2. Bottom left plot is the double deep q learning with reward 1 function. Bottom right is with ddqn and reward 2.

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:

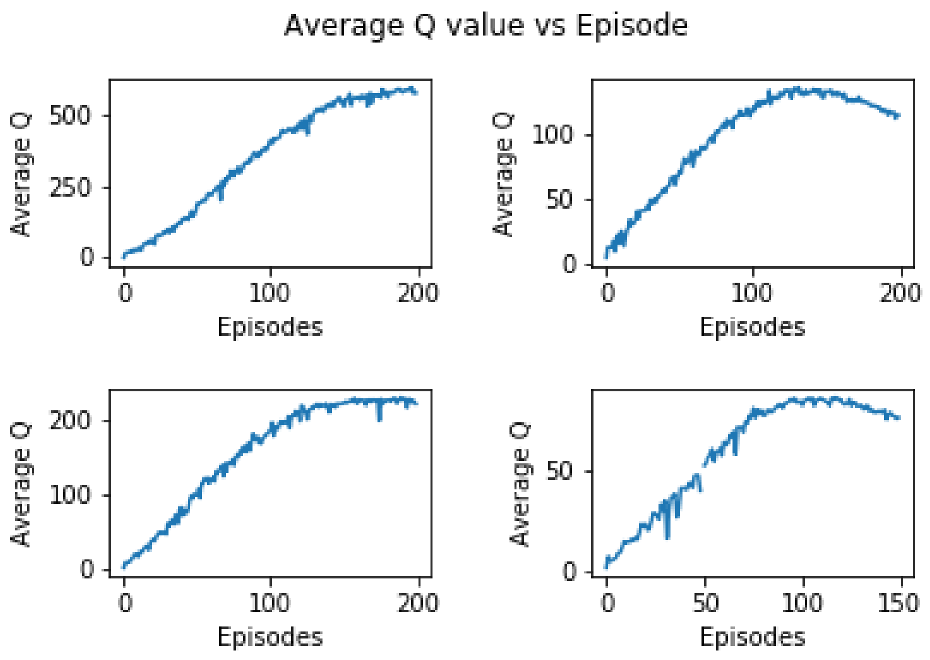

Figure 4: Top left plot shows the Average q plotted for deep q learning implementation for first reward function. Top right is the same with reward 2. Bottom left plot is the double deep q learning with reward 1 function. Bottom right is with ddqn and reward 2.

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 plots for both deep q learning and double deep q learning curves were almost flattened after 100 episodes and for reward function 1, it took a little longer for getting almost flattened curve.
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. All the different training set-ups 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).

6 Future Work

In our paper, we only implemented the deep q-learning and double deep-q learning models for finding 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 the car move even faster and better. We can also test by adding some obstacle 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] GYM. Getting started with gym, 2016.

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

[3] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap, Jonathan J. Hunt. Continuous control with deep reinforcement learning, 2019.

[4] Arthur Guez Hado van Hasselt and Google DeepMind David Silver. Deep reinforcement learning with double q-learning, 2016.

[5] Laurent Desegur. A lane detection approach for self-driving vehicles, 2017.