Deep Reinforcement Learning for Self-Driving Using DDQN

Bella Abdelouahab

Abstract :

The purpose of this research is to use Double Deep Q-Network Agents to evaluate the task of a self-driving car. For this purpose several approaches to Q-learning will be studied. The DDQN agent performs an action in the environment (car) and acquires the new distance from the walls using vehicle's sensors which are placed onboard. The goal of the policy is to drive as fast as possible while avoiding crashes. An environment will be designed using the pyglet simulator. The findings demonstrate that using a DQN agent to self-driving a car in this environment may be an effective strategy.

Keywords: Self-Driving · Reinforcement Learning · Double Deep Q-learning Network · Artificial Neural Network · Pyglet Simulator.

1. BACK GROUND

Machine learning :

Machine learning is a subset of artificial intelligence (AI) that is designed in such a manner that it can automatically learn and grow from experience without having to be coded totally manually. For various tasks, there are a variety of different machine learning techniques to apply; these strategies range in difficulty from the most simple to the most complicated. A major approach in the field of machine learning is to use the best technique to generate the best outcomes. Machine learning, in general, necessitates a vast amount of data that must be processed and assessed. While machine learning often produces more accurate results in terms of identifying profitable possibilities or potential hazards, it does so at the expense of a lot of resources and training time.

Reinforcement learning:

Reinforcement learning occurs when an algorithm provides feedback on individual actions with a desired end state but no apparent optimum path. A model is asked to make a sequence of decisions as part of reinforcement learning. Each action that is taken will result in a reward, which will either encourage the model to continue completing the activities or will have the opposite effect. Reinforcement machine learning algorithms are a type of learning approach that

involves interacting with the environment, and the model's target, the [1]agent, will try to maximize its reward via trial and error.

Q-learning:

Q-Learning is one of the more fundamental reinforcement learning algorithms. In contrast to a model-based approach, Q-Learning makes the agent learn policies (agent's way of behaving at a given time) directly. A positive or negative reward must be included in the policy table, Q table, for every action done by a state.

Artificial Neural Networks:

Artificial neural networks, or ANN, are computing systems that are inspired by biological neural networks that can execute a variety of tasks using a large quantity of data. To get the best outcomes from changing inputs[3], several algorithms are employed to comprehend the relationships in a given collection of data. Different models are employed to forecast future results given the data, and the network is taught to provide the required outputs. The nodes are joined in such a way that it functions similarly to a human brain. To cluster and categorize the data, several correlations and hidden patterns in the raw data are exploited.

Deep Q-learning Network:

DQN came to prominence a few years ago when they broke the Atari code for DeepMind. The idea was to combine Q-learning with a deep neural network to estimate the Q-function. in order to make accurate predictions on future decisions without the usage of Q-Tables. The Q table is a lookup table in which each item reflects a combination of states and actions, which is excellent for simpler circumstances, as previously stated. However, in case of large and complex environments it may take a long time to get a high reward so basically the agent gets used to low but good scores and considers these high scores as out-layers .

Double Deep Q-Network:

To solve the problem mentioned before, a few tricks had to be introduced to assist the network and enable it to train consistently.

The initial trick was to include [2]**experience replay** in the game. This is a memory buffer that is sampled from when updating to break sequence dependence by storing previously observed state-action-reward-next state tuples.

The system's second trick was to add a **target network**. This target network is simply a duplicate of the neural network we're training, and the algorithm updates it on a regular basis. This is significant because it provides a generally steady baseline for performance evaluation. We don't have labeled data to inform us when we're right or incorrect in reinforcement learning; instead, we have a reward signal. If we're maximizing a reward, the greater the better in our algorithm's eyes, but we have no idea how huge it can go.

RL agent, for example, is chugging along, picking up a few goodies here and there as it completes its mission. There are a lot of 0's and 1's, then it receives a massive reward of 10 all of a sudden. Is that satisfactory? Well, it's certainly better than anything we've seen so far, but perhaps if we had taken a different action, we might have received a 50 or 100 reward instead. We just do not know.

Because of its second Q-function approximator, double Q-learning may learn a superior approximation. We can do the same thing, except instead of utilizing a second, independent DQN, we can use our target network to unbias our predictions. This is quite easy to do by using the target network to estimate the value of our action.

2. STRUCTURES

TensorFlow Framework:

TensorFlow is a free and open-source software framework for dataflow and differentiable programming that may be used to solve a variety of problems. It's a symbolic math library that's also utilized in neural networks and other machine learning applications. At Google, it's utilized for both research and manufacturing. TensorFlow was created for internal Google use by the Google Brain team. On November 9, 2015, it was distributed under the Apache License 2.0.

ANN Model:

Since our goal policy as mentioned before is to go as fast as possible avoiding crushes, we will use a sample artificial neural network constructed of 5 layers, the input layer contains 11 neuron wish represent the distances for the walls of the truck, the output layer contains 5 neurons divided as follow (forward, backward, left, right and go straight), the hidden layers are 3 layers each on contains 256 neurons.

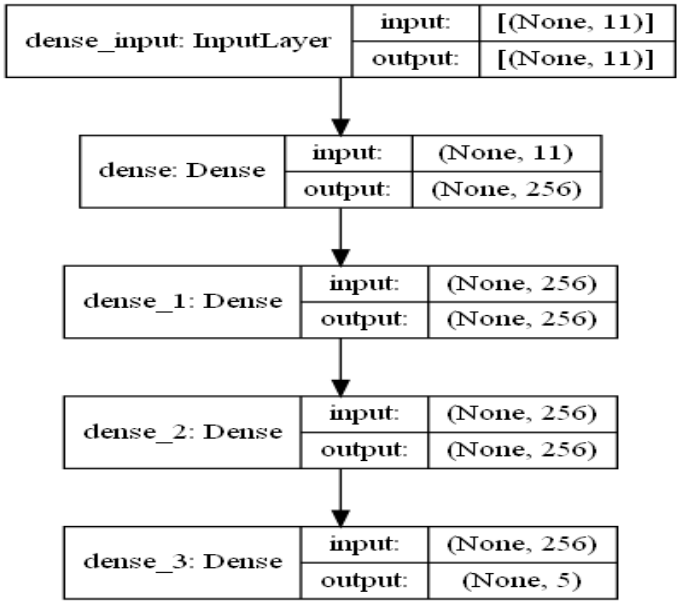
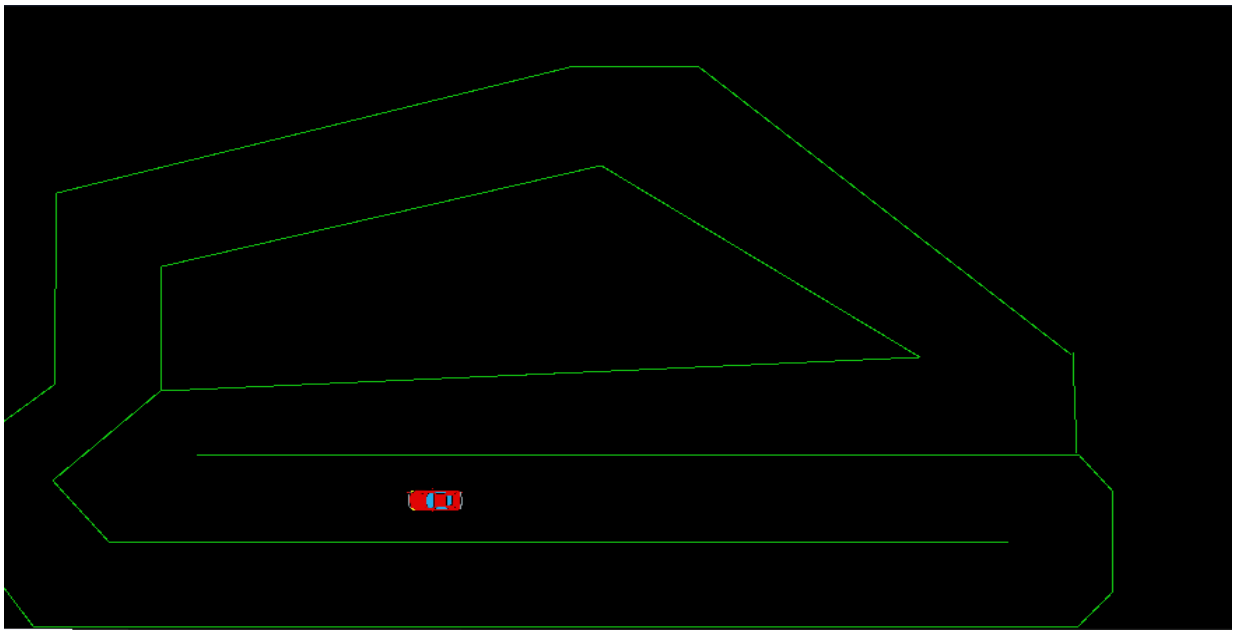


Figure 1. Model different layers

3. Environment

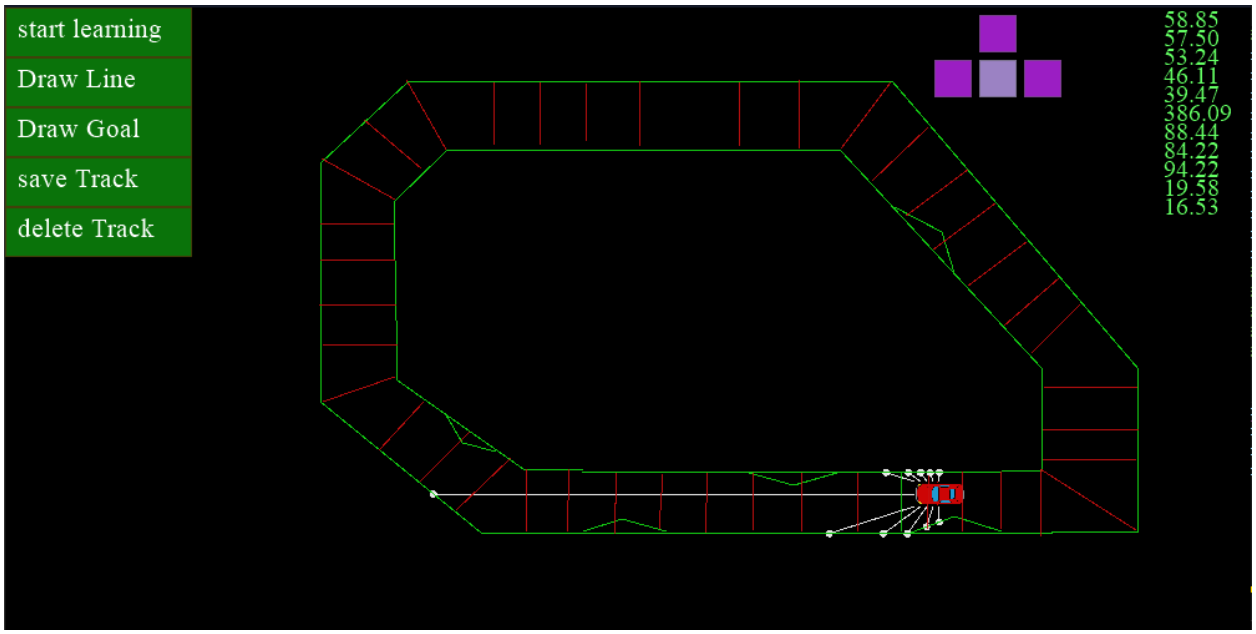
Classic environment:

Using the pyglet Framework we build a 2D environment with size of 1000x500px contain a racing truck and a car, the car can be controlled using keyboard and both the rotation angle or velocity increase as long as the pressing on the key gose



Custom environment:

The environment above considered good and easy for human to play, however since we are not using a pretrained model the model basically dose not know anything about the environment, it does not know what a truck is,what a car is, how the car moves and way more, the model has zero previous knowledge, so for the observations of the environments, we added some lines to calculate the distance from the wall (using the intersecting formula since the truck also is a bunch of lines) , this idea was inspired from how the bats sees, next we put some lines around the truck wish called reward gates, whenever the car crosses a gate the next gate gets activated and a reward claimed for this action, over time the AI will understand that it needs to go around the truck



4. Results and Evaluation

Model Configuration:

For this environment we will need specific [5]hyperparameters to control the learning process since this is a complex environment.

Epsilon : epsilon is a value between 0 and 1 which controls the average random action the agent takes during training, in our case we want the model to have more time in exploring, thus we set epsilon discount to 0.998, in addition to that we want the agent to take a random action from time to time in order to have effective and generalize training, so we set minimum

value of epsilon to 0.1.

Gamma: gamma tells us the discount factor which means that agent will reduce the contribution of estimates of reward from future states to its estimates of the values of the given state because it may or may not end up in whatever state or whatever set of action it predicts for the action values function.

Model Learning Graphe:

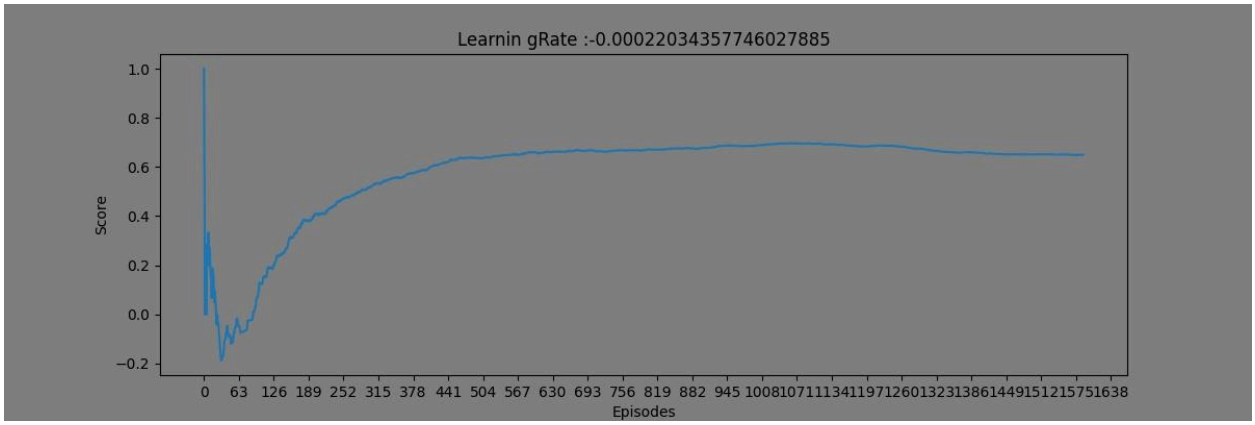
Learning rate : 0.01

Epsilon-discount : 0.9

Batch-size:128 [4]

Inputs:11

output(number of action):5



The previous figure represents the training graph with default parameters, lets see twik some hyperparameters

Learning rate : 0.001

Epsilon-discount : 0.998

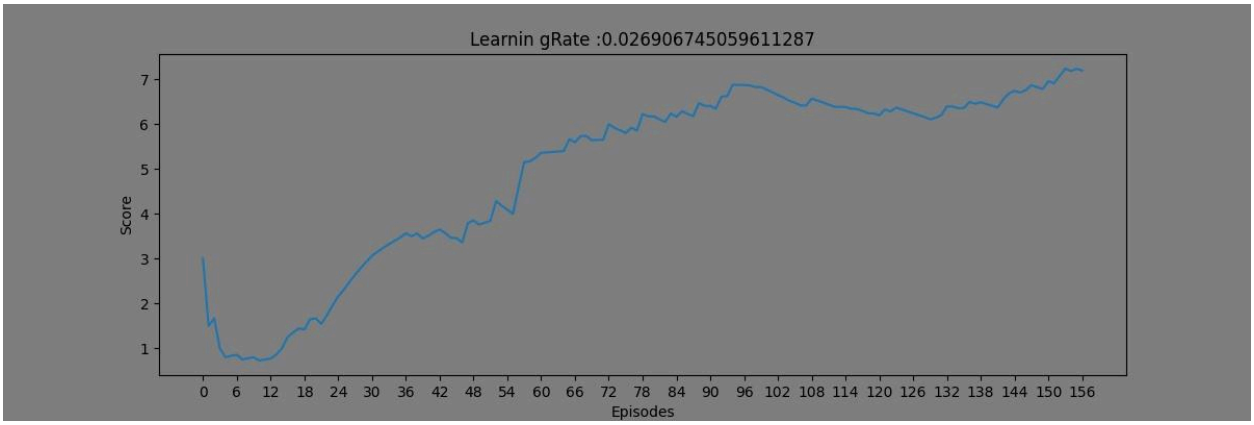
Batch-size:64[4]

Inputs:11

output(number of action):5

Gamma:0.99

And also we change number of action take per second from 10 to 60, so that agent has more time to evaluate the state



As it is being demonstrated, When compared to a model that uses a default hyperparameters, the model that uses a custom one gradually increases the rewards per episode and results in a greater reward peak.

5. Discussion

Dueling DDQNs (Dueling Learning):

Dueling deep q-learning: in this algorithm the value function V(s) and the advantage function A are used to break the Q-values into two portions (s, a).

V(s) is a value function that tells us how much reward we'll get from state s. And the advantage function A(s, a) indicates how much better one action is than the others. We can derive the Q-values by combining the value V and the advantage A for each activity.

$$Q(s,a) = V(s) + A(s,a)$$

Thus the agent can learn which states are (or are not) valuable without having to learn the effect of each action at each state. Due to this information, it is clear that adopting dueling double q-learning, rather than the traditional DDQN, would have been more productive and

advantageous in this setting.

Machine Configuration:

OS Name: Microsoft Windows 10 Pro
Processor: Intel(R) Core(TM) i5 CPU M 430 @ 2.27GHz 2.27 GHz
Total Physical Memory:6,135 MB
GPU : NVIDIA GeForce GT 320M

Training time : around 110 hours

6. Conclusion

While I've solved several environments before, this one appears to be the most difficult. There was no opportunity to use various reinforcement learning strategies because so much time was spent trying to train the custom and classic environments using a DDQN. While the results did not meet my high expectations, I learnt a lot because altering the environment and the model to evaluate changes was both entertaining and informative. Overall, the project's prognosis has remained good throughout all of the trials and testing.

The code for this article can be found here

References:

[1]:www.mathworks.com Reinforcement Learning Agents https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html

[2]:Levine, Sergey. ”Exploring deep and recurrent architectures for optimal control.” arXiv preprint arXiv:1311.1761 (2013).

[3]: the artificial neural network–based reinforcement learning

[4]: Ioffe, Sergey, and Christian Szegedy. ”Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv preprint arXiv:1502.03167 (2015).

[5] : Kizito Nyuytiymbiy (Dec 30, 2020) Hyperparameters in Machine Learning and Deep Learning from https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac