Rohan Singh
116166331

Project Proposal ENPM690

Abhinav Modi
116346849

Title

Reinforcement Learning for Autonomous Decision Making in Dense Traffic

Abstract

Through this project, we aim to address the problem of lane changing decisions and speed-up/speed-down decisions in cars. We aim to use a reinforcement learning algorithm to train an agent, a car, driving on a highway with traffic, to reach a high velocity while avoiding collisions with neighbouring vehicles. Driving on the right side of the road is also rewarded. We proposed to use 2 versions of the Deep Q-Learning algorithms, an unmodified "vanilla" DQN and Neural Fitted Q-Learning.

Introduction

With the advent of self-driving technology for vehicles, there has been an increasing need for accurate autonomous decision making algorithms. Although designing such algorithms for practical application is a complex task, we decided to focus our work on lane change decisions, training and testing our algorithm on simulation.

There are previous works on lane departure algorithms like the MOBIL model, but we aim to use Reinforcement learning for this problem, hoping to "teach" the agent(the car) to recognize/predict behaviours of surrounding cars(which may be difficult/complex to characterize with any traditional methods) to make a more informed decision. Moreover, we also took this opportunity to discuss the limitations of plain Deep Q-Learning by comparing it with Fitted Q-Learning, which provides more empirical guarantees of convergence.

We hope this work helps with existing self-driving research, and also provide a comparison of the above mentioned DQN algorithms.

Background/Related Work

Quick Overview of traditional Q-Learning

Q-learning is a model free reinforcement learning algorithm that seeks to find the best action to take given the current state. This algorithm tries to find the policy that maximizes future rewards, by estimating the "quality" (Q-value) of an action, and thus following a policy that chooses the best action (max q-value) from any given state. These Q-values are stored in a table with Q-values for all action-state pairs.
In classical Q-learning, the update rule is given by

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha(c(s,a) + \gamma \min_b Q_k(s',b)) \tag{1}$$

where s denotes the current state, a is the action that is applied, and s' is the resulting state. c is the reward function, α is the learning rate that has to be decreased in the course of learning in order to fulfill the conditions of stochastic approximation and γ is a discounting factor. As mentioned in [1], it can be shown, that under mild assumptions Q-learning converges for finite state and action spaces, as long as every state action pair is updated infinitely often. Then, in the limit, the optimal Q-function is reached.
Typically, the update is performed on-line in a sample-by-sample manner, that is, every time a new transition is made, the value function is updated.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q
    Take action a, observe r, s'
    Update
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  Until s is terminal

Figure 1: Q-Learning

Deep Q-Learning

In the above described algorithm, the Q-function can be approximated by a neural network(in place of keeping record of Q values in a table). Since no direct assignment of Q-values like in a table based representation can be made, instead, an error function is introduced, that aims to measure the difference between the current Q-value and the new value that should be assigned. The error measure is like

$$error = Q(s,a) - (c(s,a) + \gamma \min_b Q(s',b)) \tag{2}$$

At this point, common gradient descent techniques (like the 'backpropagation' learning rule) can be applied to adjust the weights of a neural network in order to minimize the error.
Like above, this update rule is typically applied after each new sample. As mentioned in [1],

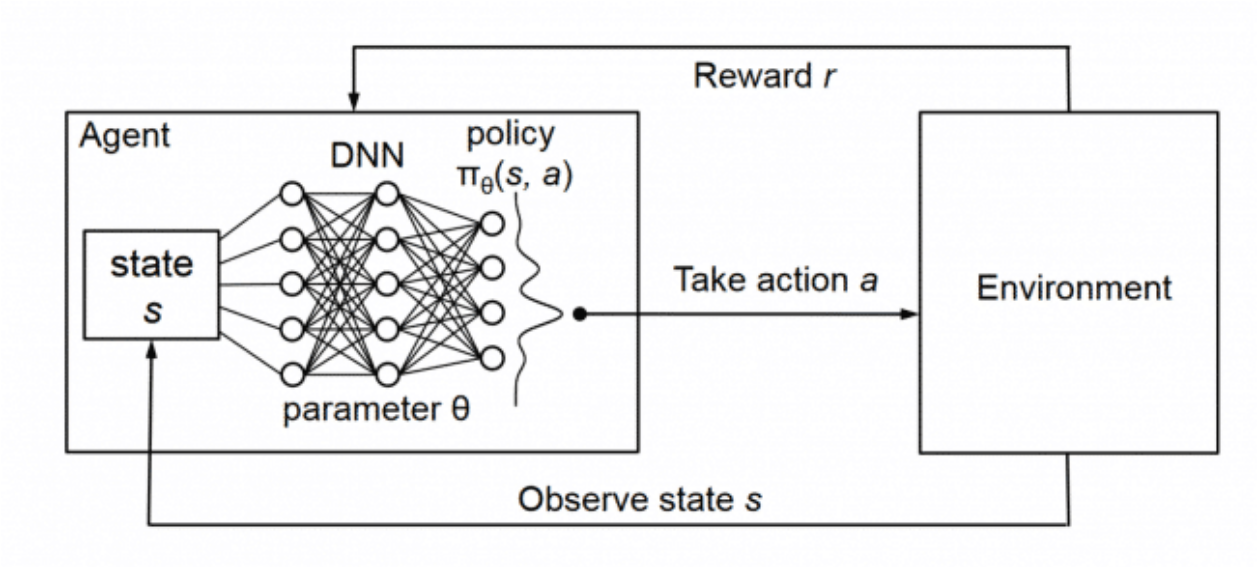

Figure 2: General Architecture for Deep Q-Learning

the problem with this on-line update rule is, that typically, several ten thousands of episodes have to be done until an optimal or near optimal policy has been found. One reason for this is, that if weights are adjusted for one certain state action pair, then unpredictable changes also occur at other places in the state-action space. Although in principle this could also have a positive effect (generalisation) in many cases, in most, this seems to be the main reason for unreliable and slow learning.

Neural Fitted Q Iteration

The basic idea underlying NFQI is the following: Instead of updating the neural value function on-line (which leads to the problems described in the previous section), the update is

performed off-line considering an entire set of transition experiences. Experiences are collected in triples of the form (s, a, s') by interacting with the (real or simulated) system. The set of experiences is called the sample set D. Experiences are collected after running the game(agents interaction with the environment) and stored in memory till it is reaches a set limit. From this memory, batches are randomly sampled and used for offline training for set number of epochs, after which new experiences are generated using the now updated network. This is followed till convergence to desired behaviour.
The consideration of the entire training information instead of on-line samples, has an important further consequence: It allows the application of advanced supervised learning methods, that converge faster and more reliably than online gradient descent methods.

Approach

We planned to complete this project in the following manner :

1. Implement environment for the required task.
2. Implement a simple network for Q learning.
3. Map reward according to desired behaviour and tune hyper-parameters while training for low density traffic first, then for high density traffic.
4. Implement additional modules for training/sampling for the Neural Fitted Q-Learning algorithm.
5. Train (while tuning hyper-parameters) for low density traffic first, then for high density traffic.
6. Compare results from both methods.

Implementation

Environment

The state-space we plan to use for the learning algorithm is represented by $S = \{s_i\}_{i[0,N]}$ where

$$s_i = \begin{bmatrix} x_i, & y_i, & v_x^i, & v_y^i, & cos\psi_i, & sin\psi_i \end{bmatrix}^T \tag{3}$$

is called the list of features for each neighboring car and s_0 defines the parameters of the ego-vehicle.

The action space is represented using 5 integers which represent the following

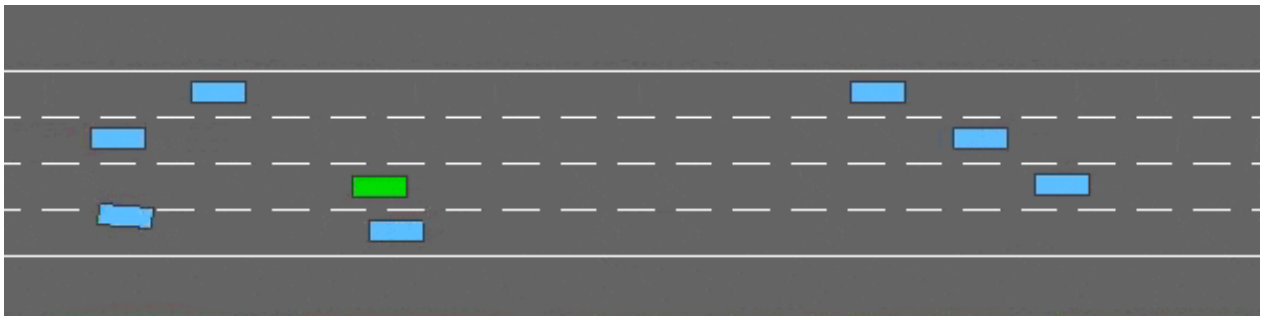

Figure 3: Snapshot of the Highway environment to be used

- 0: Switch to left lane
- 1: Don't do anything
- 2: Switch to right lane
- 3: Go faster
- 4: Go slower

The Agent(car) uses a low-level PID controller for following the decisions made by the learning algorithms. The aim is to teach the ego-vehicle, 2 behaviours i.e., vehicle acceleration and lane changing to navigate through the highway while avoiding the other vehicles and maintaining stability in traffic.

DQN

The neural network used for DQN was implemented using the pytorch library, with tensorboard for training result visualization. The network was a Multilayer Perceptron consisting of 2 hidden layers with 265 neurons in each layer (The initial and final layers were the size of the state and action vectors). The hyperparameters and other algorithm features used finally for the training are as follows:

1. Discount Factor $\gamma = 0.8$
2. Learning Rate = 1e-2
3. Loss function - L2 loss
4. Activation function - ReLU(Rectified Linear Unit)
5. Total number of epochs = 2000 (each for low and high density traffic environment)
6. Exploration-Exploitation strategy - Epsilon greedy

- Decay Rate $\tau = 6000$
- Time step t = 1
- Initial epsilon $\epsilon = 1$
- Final epsilon $\epsilon = 0.05$

NFQI

The neural network used for NFQI was also implemented using the pytorch library, with tensorboard for training result visualization. The network was a Multilayer Perceptron consisting of 2 hidden layers with 265 neurons in each layer (The initial and final layers were the size of the state and action vectors). The hyperparameters and other algorithm features used finally for the training are as follows:

1. Discount Factor $\gamma = 0.8$
2. Learning Rate = 1e-2
3. Loss function - L2 loss
4. Activation function - ReLU(Rectified Linear Unit)
5. Total Memory size = 10,000 experience tuples
6. Batch size = 1000 experience tuples
7. Total number of Regression epochs = 400
8. Exploration-Exploitation strategy - Epsilon greedy
    - Decay Rate $\tau = 2000$
    - Time step t = 1
    - Initial epsilon $\epsilon = 1$
    - Final epsilon $\epsilon = 0.05$

Reward Modelling

The reward function for the environment consits of the following:

1. Collision reward: -1 if collision 0 otherwise
2. Right lane reward: 0.1 * the index of the lane the vehicle is driving in. *The lane numbers are set as 0 for the top lane and 3 for the bottom-most lane which is the right-most lane for the vehicles*

3. High velocity reward: +0.5 for accelerating
4. Lane change reward: 0

The above mentioned rewards were the ones we began our training with. After some initial experiments we observed that the vehicle was changing its speed but not changing lanes at all. So we changed the lane change reward to encourage the agent to change lanes in case there is a vehicle ahead and there is space in the adjacent lanes. To keep the collisions to a minimum we also changed the collision reward and the final rewards were as follows:

1. Collision reward: -10 if collision 0 otherwise
2. Right lane reward: 0.2 * the index of the lane the vehicle is driving in. *The lane numbers are set as 0 for the top lane and 3 for the bottom-most lane which is the right-most lane for the vehicles*
3. High velocity reward: +1.0 for accelerating
4. Lane change reward: 0.5

Results and Discussion

The training is performed using both deep Q learning and fitted Q methods in 2 different density environments:

1. Low density environment: 40 cars, which are initialized in the highway environment with an initial spacing of 1.5*initial velocity

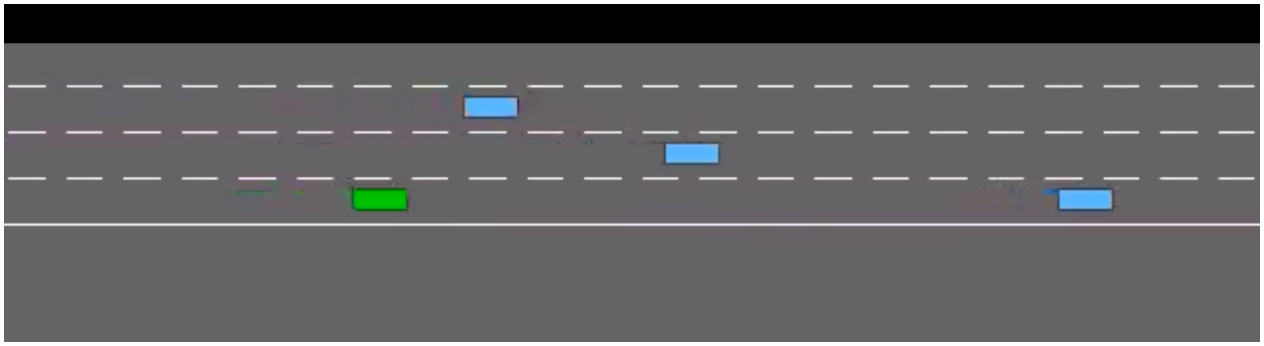

Figure 4: Low density environment

2. High density environment: 80 cars, which are initialized in the highway environment with an initial spacing of 1*initial velocity.

Figure 5: High density environment

The training happens for 2000 iterations in the low density environment and then training environment shifts to a high density environment. This change in the difficulty of the environment during training is called curriculum learning. This helps the agent generalize better in the high density environment because it has already learned a basic policy and is not starting from scratch.

Both the implemented algorithms show positive results after the initial 2000 episodes. The total reward per episode achieved by the agent increases as the agent trains for increasing number of episodes. For the DQN method, the results can be seen in the figure (6)

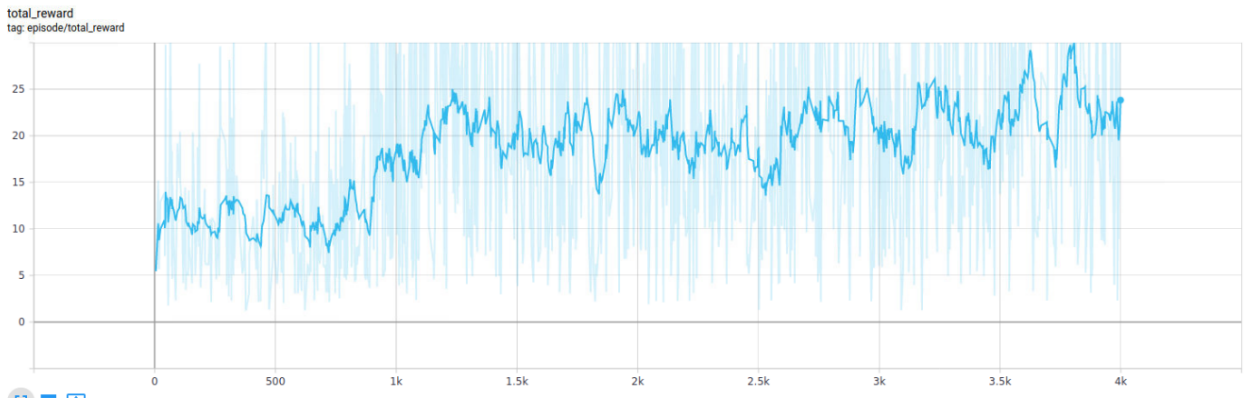

Figure 6: Total reward received by the agent for each episode for Deep Q learning

For the Fitted Q iteration method, this step couldn't be automated and thus we had to manually resume the training from the last checked point for the high density environment.

The graph shown in (8) is the regression loss for the batched training of the neural network for the Fitted Q iteration method.

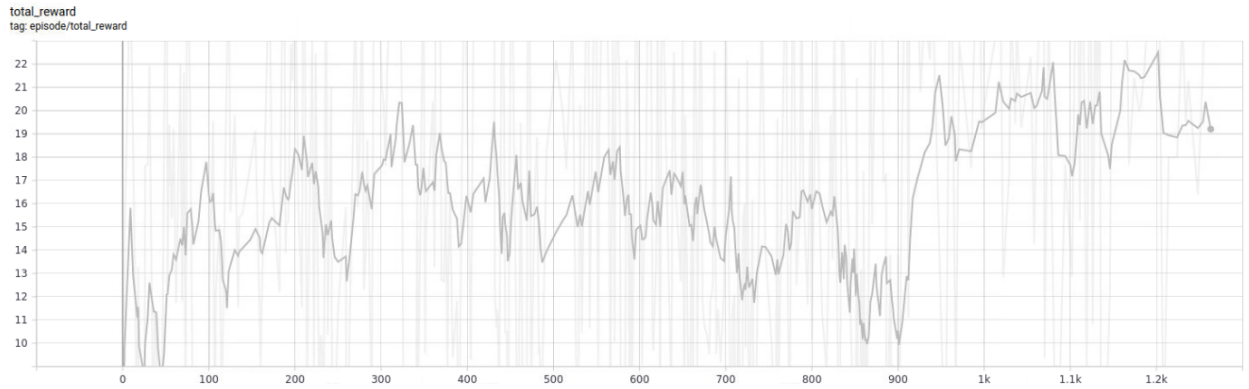

Figure 7: Total reward received by the agent for each episode for Fitted Q iteration method

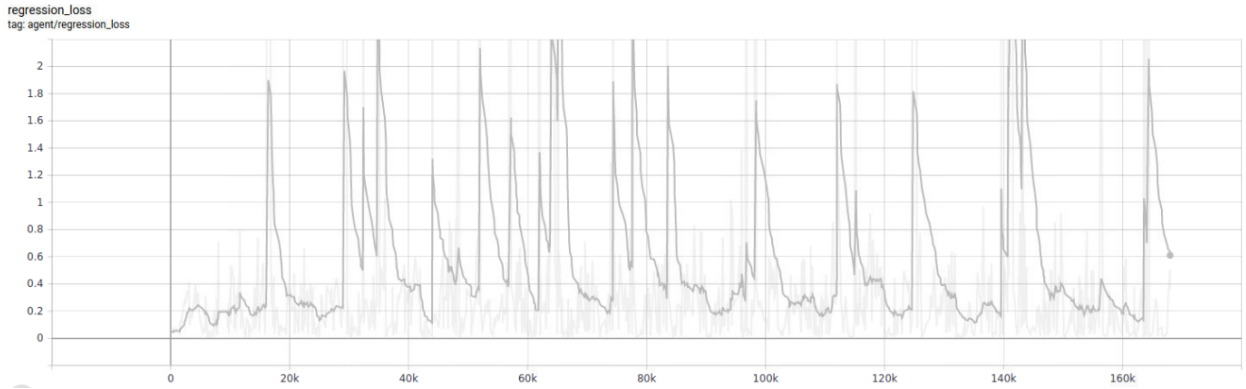

Figure 8: Regression loss per iteration for all intermediate batched training in Fitted Q Iteration method

Conclusion

We were able to train a car to make lane/speed change decisions autonomously, given the state of neighbouring vehicles(This information can be assumed to be available though a perception module installed on the car). We also provided empirical evidence of the better performance of Neural Fitted Q Iteration over plain "vanilla" Deep Q-Learning. This project helped us in exploring the deep Q reinforcement learning and its variations and how it can be used for such complex and dynamic environments. The current implementation helps achieve the goal we aimed for in this project but in many ways not sufficient for real-world use. We propose a few changes that can benefit the training process and make the environment more "real" in the next section.

Future Work

Currently the state space is what we can call as a list of feature vectors. This encoding is efficient in the sense that it uses the smallest quantity of information necessary to represent the scene. However, it lacks two important properties. First, its size varies with the number of vehicles which can be problematic for the sake of function approximation which often expects constant-sized inputs. Second, we expect a driving policy to be permutation invariant, i.e. not to be dependent on the order in which other traffic participants are listed. A popular way to address this limitations is to use a spatial grid representation. Using this we can train a CNN architecture for the Deep Q learning process using the snapshot of each time instant of the environment.

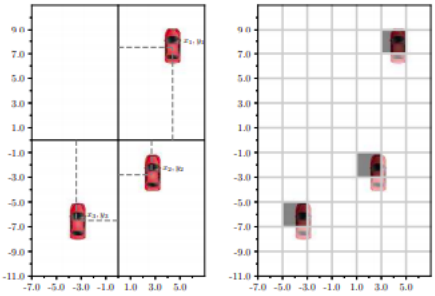

Figure 9: position(x and y) list representation(left) vs Spatial grid representation(right)

Another way to make the environment more "real" is by introducing heterogeneous agents. All the blue vehicles in the environment have the same behaviour model i.e., acceleration-deceleration rates, "politeness" towards other agents and lane changing patterns. In the figure(10) the red cars depict some aggressive cars which exhibit sudden lane changing behaviours, higher acceleration and sudden braking behaviours. This increases the chaos in the environment making it difficult for the agent to navigate through the environment. This can help the agent to learn to react to such difficult traffic situations which will be very beneficial for real-world testing and deployment.

Github link

The source code for this project can be found here: Link

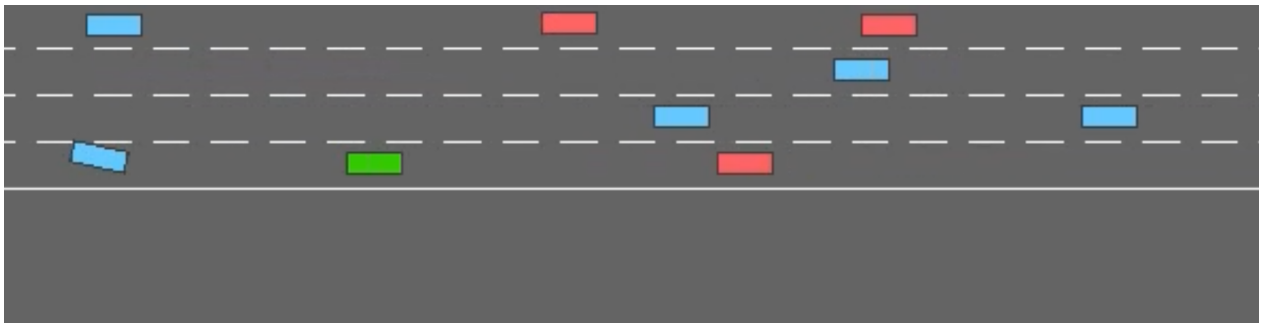

Figure 10: position(x and y) list representation(left) vs Spatial grid representation(right)

Bibliography

1. Riedmiller, M., 2005, October. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In European Conference on Machine Learning (pp. 317-328). Springer, Berlin, Heidelberg.

2. Highway environment reference link

3. Ernst, Damien, Pierre Geurts, and Louis Wehenkel. "Tree-based batch mode reinforcement learning." Journal of Machine Learning Research 6.Apr (2005): 503-556.

4. Chen, Yu Fan, et al. "Socially aware motion planning with deep reinforcement learning." 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2017.