Self-Driving Car Using Deep Q-Network

Pranitha Keerthi
Master's of Science in Computer Science

Saivenkat Thatikonda
Master's of Science in Computer Science

Abstract— **By leveraging the power of neural networks, deep reinforcement learning has achieved optimal excellency in many of the field's like Atari games, self-driving cars etc. Deep reinforcement learning is poised to revolutionize the field of AI and represents a step towards building autonomous systems with a higher-level understanding. Currently, deep learning is enabling reinforcement learning to scale to problems that were previously intractable. In this paper, we present our approach in building a self-driving car using deep Q-learning that is simple Q-learning combined with an artificial neural network. Firstly, we start by explaining the basics of Q-Learning that is how it is derived from bellman's equation, uses Markov decision process for action selection process and is updated by temporal difference. Secondly, we explain how it is combined with neural network, making it a deep Q- learning method. Further, we explain our approach in implementing self-driving car using deep Q- learning. We also compare the performance of the car agent with Deep Q-learning and Deep Q-learning with experience replay to emphasize how important experience replay is to learn complicated environments. Moreover, we will also show the evaluation of results while tuning the parameters like rewards, neural network architecture etc.**

I. Introduction

Q-learning [1][5] is a standout amongst the most prevalent reinforcement learning algorithms to come up with a policy in different reward driven domain problems like Atari games, self-driving cars, robot control etc. Simple Q-learning is a technique which was derived from the Bellman's dynamic programming recursive equation [3], it helps us to calculate the goodness of an action for a state. Artificial neural network are the main tools for machine learning, since many years neural network is in the center of attention when autonomous driving is to be implemented. Neural networks consist of input, hidden (consisting of randomized weights), and output layers. By making the neural network predict the Q- values for all the actions of a state we build a deep Q- network.

The main goal of this project is to build a self-driving car which can learn how to drive itself from a given initial position to the goal state within the boundaries of the given road and pass through any obstacles that comes in its way. Every year new network architecture and improvements to existing systems for autonomous cars are presented by a wide range of scientists all over the globe. Big tech companies such as Google and Facebook as well as huge automakers such as Tesla and BMW invest billions in the research of deep learning, which pushes the development to unexpected and unseen levels, which is our motivation for this project. The environment for a self-driving car is dynamic, i.e. random events like obstacles can occur suddenly, that is why we chose deep Q-learning algorithm [5][7] because of its excellent performance in complicated as well as dynamic environments.

A. Domain

1. *Markov decision process* [2][5]

Markov rule states – *"the probability of reaching the next state depends only on the present state but not on the previous states and actions."* A transition in reinforcement learning is defined as a tuple of 4 (S, A, R, S') where s is the present state of the agent, a is the action it performed in the transition, r is the reward it gets for the action at that state and s' is the next state.

$$P\ (s_{t+1}|s_0,\ a_0,\ s_1,\ a_1,\ ...,\ s_t,\ a_t) = P\ (s_{t+1}|s_t,\ a_t) \qquad (1)$$

In the above given tuple (S, A, R, S'), where *s* is the set of all the states of the environment at present time *t*. In any environment we can represent the state of the agent using a vector of encoded values; *a* is the set of actions the agent can perform at time *t*. We can define the transition rule as- "*the probability of s_{t+1} given state s_t, action a (P (s_{t+1} | s_t, a)) is given by the probability distribution of future states, r is the reward function the agent gets after performing an action at a state."*

2. *Bellman's Equation [3] and Temporal Difference* [4]

Bellman's principle of optimality states that *"an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."* At the beginning t = 0, all the Q-values are initialized to 0. Now let's suppose we are at time t, in a certain state s_t. We play the action at and we get the reward r_t. Bellman equation states that the value of current state (value here means the goodness of being in that state not the Q-value) is equal to the sum of reward the agent gets and the discounting factor multiplied by maximum of all the values of the states it can reach doing the action,

$$V(s) = r_t + \gamma\ max_a\ (Q\ (a,\ s_{t+1})) \qquad (2)$$

Temporal Difference Learning methods can be used to estimate the value functions (value functions are state-action pair functions that estimate how good a particular action will be in a given state, or what the return for that action is expected to be), i.e. it is the difference between the value of the present state at the present time with present information and the value of the current state which we get when we compute it in the future with different information we get in the future time.

$$TD_t\,(a_t, s_t) = r_t + \gamma\, max_a\,(Q\,(a, s_{t+1})) - Q\,(a_t, s_t) \tag{3}$$

3. *Q-Learning* [1][5]

Q-Learning can be defined as a form of model-free reinforcement learning, a method of asynchronous dynamic programming (DP) based on Bellman's equation. The learning is proceeded using temporal differences (TD), i.e.

$$Q_t\,(a_t, s_t) = Q_{t-1}(a_t, s_t) + \alpha TD_t\,(a_t, s_t) \tag{4}$$

where $TD_t\,(a_t, s_t)$ can be calculated as,

$$TD_t\,(a_t, s_t) = r_t + \gamma\, max_a\,(Q\,(a, s_{t+1})) - Q\,(a_t, s_t) \tag{5}$$

In the above equations α is the learning rate it can be any value between 0 and 1, γ is the discounting factor which is heuristic to know how far we are from the goal, a is the actions available at the present state and t is present value whereas t-1 is the past value. Using the temporal difference, we update the Q-value of every state as a function of time. Since the Q-value is getting updated by seeing the maximum of future value recursively it gives an optimal plan for the agent to reach the goal state.

4. *Deep Q-learning* [5]

Q-learning process when combined with a neural network makes it a deep Q-learning process. In Deep Q-learning we send an agent state representation into the neural network and architect the neural network in such a way that it outputs the Q-values of all the actions of the state sent into the neural network.

Transition in a deep Q-Learning is defined as a tuple of 4 (s, r, a, s'), where *s* is the state of the agent, *a* is the action given by the neural network by doing a SoftMax over Q-values, *r* is the reward the agent got for performing the action and *s'* is the state it reaches after performing the action.

Experience Replay [9] can be defined as collection of the transitions as experiences made by the agent for the learning process in a deep Q-network by replaying those experiences.

Learning process: By sending a state into the neural network we get predicted Q-values for each and every action of that state and by applying a SoftMax function on the Q-values we get a predicted action which has a maximum Q-value. Now we calculate the target Q-value for the same action in that state by using simple Q-learning (using eq.4), i.e. the reward it gets after performing that action plus the discounted factor multiplied by the best Q-value of the next state. Since we have the predicted Q-value and the target Q-value the neural network will now do its magic in the form of gradient descent and starts to learn by minimizing the loss between the prediction and the target.

As explained above, we use the learning phase of the DQN to make our car learn what actions to take at a given state. We leave the car for exploring the environment by making transitions, i.e. moving from one state to another by performing an action with a reward for that action. All the transitions are there after stored in experience memory. Every time the experience memory accumulates some n number of transitions, the learning process described above is implemented for a batch of transitions.

II. Literature Review

There wasn't any paper that implemented what we were trying to do. But there are several papers that helped us in finding the algorithm which we can implement to achieve our goal and also few related works that carried out the deep Q-network algorithm to achieve their goal.

The main paper that we reviewed for our project is "Deep Reinforcement Learning for Autonomous Driving" [5]. This paper was motivated by the successful demonstrations in learning of Atari games like chess and Go by Google DeepMind. The authors in this paper propose a framework for autonomous driving using deep reinforcement learning. The authors of the paper start by explaining a short review about deep reinforcement learning and then move ahead by explaining about their framework, incorporation of recurrent neural networks for information integration, enabling the car to handle partially observable scenarios. The approach was tested in an open source 3D car racing simulator called TORCS. The paper state that their simulation results demonstrate learning of autonomous maneuvering in a scenario of complex road curvatures and simple interactions of other vehicles. The network is trained end-end [6] following the same objective of the DQN. The result of the approach is a successful lane keeping behavior with speed limit. This paper was a huge asset for us to understand how DQN works, how it can be carried out for autonomous car and mainly, this paper helped us in understanding how to train an agent for lane keeping behavior.

The paper "Playing Atari with Deep Reinforcement Learning" [8] was used to get an idea about the environment, i.e. how an environment can be represented and how to make an agent learn based upon the future rewards. This paper uses convolutional neural network, trained with a variant of Q-learning, with input as raw pixels to achieve a goal as a value estimating future rewards. This approach was applied to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or the learning algorithm. This paper demonstrates that a convolutional neural network can overcome the challenges of deep learning, due to data distribution changes in RL an algorithm learns new behaviors which can be problematic for deep learning methods that assume a fixed underlying distribution, to learn successful control policies from raw data in complex RL environments. The evaluation metric of this approach is the total reward the agent collects in an episode or game averaged over a number of games, which is computed periodically during training. It is observed that the average total reward metrics tend to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits. The method lacks in any theoretical convergence guarantees but it was able to train large neural networks using a reinforcement learning signal and stochastic gradient descent in a stable manner. The result of this approaches is it surpasses a human expert on three of the games played against the Atari AI developed in the paper.

Another paper that we reviewed for our project is the “Human-level control through deep reinforcement learning” [7]. The authors in this paper utilized ongoing advances in preparing deep neural networks to build up a novel artificial agent, named "a deep network", a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks- a central goal of general artificial intelligence that has eluded previous efforts. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. The authors tested the agent on the challenging domain of classic Atari 2006 games. In addition to the learned agents, the authors also report score for professional human games tester playing under controlled conditions. The paper demonstrated that the deep Q-network agent, receiving minimal prior knowledge, i.e. only the pixels, and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human game’s tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel a diverse array of challenging tasks. This paper was the main motivation for us to use deep Q-network for our project as it was already experimented across 49 different games and the result was deep Q-network ending up giving better results compared to other previous algorithms.

III. Methodology

Firstly, we are going to talk about environment setup, followed by the rewards for its actions. Then we are going to talk about the implementation of deep q-networks in our environment and also explain the architecture of neural network. Further, the implementation of learning process with experience replay will be discussed in depth. Finally, we will explain how the front-end interface is visually updated using the action information it gets from the DQN and how the whole system works.

Environment setup (Implementing the front-end UI):

To visualize the environment of the self-driving car we created a front-end user interface using a python package KIWI. Initially, the UI is just a blank 2D Canvas on which a virtual car will be moving. One can imagine the blank canvas as a M*N grid of 0’s. The UI is designed in such a way that a user can paint obstacles or road on it dynamically while the car is moving (as shown in Fig. 1). Whenever a user paints something on the canvas the 0 will be converted into 1 representing that there is an obstacle at that point in the canvas, making it a 2D map of pixels.

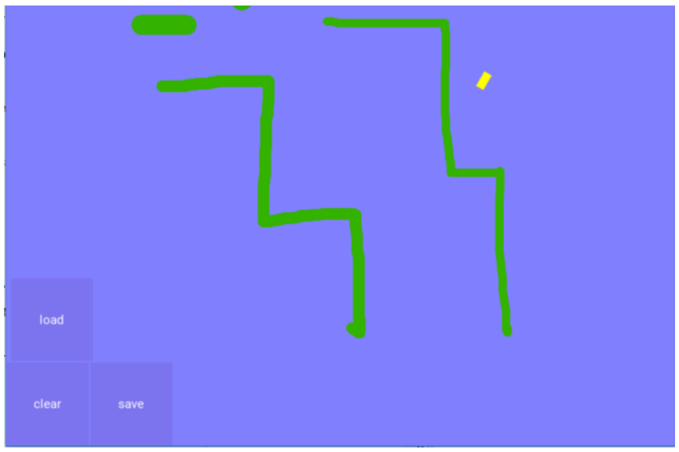

Figure 1: UI of our environment

State representation: The state representation of the car should be created by holding the essence of its surroundings with the directions of its moves, with respect to the goal.

The state of the car is represented using 5 variables,

1. *Left Sensor*, the density of obstacles that is density of 1’s towards the left of the car.
2. *Right Sensor*, the density of obstacles or density of 1’s towards the right of the car.
3. *Front Sensor*, the density of obstacles or density of 1’s towards the front of the car.
4. *Orientation with respect to the goal*, the angle in which the car is moving with respect to the angle in which the goal is pointed towards
5. *Negative Orientation with respect to the goal*, the perpendicular direction of the orientation.

At any given state in the 2D virtual environment of car, the state of the car can be represented by making a list of all the above-mentioned variables. We calculate the density of the obstacle for any sensor by taking some n squares which are situated to the required side of the car and find out how many of them are 1. Then we divide the number of 1’s by the total number of squares considered which gives the density of the obstacles. The state representation of the environment can be represented as, *[left obstacle density, right obstacle density, front obstacle density, orientation, -orientation]*.

Actions:

The actions taken by the car at any given time in the environment can be grouped as,

1. *Turning left* (turning 20 degrees clockwise on the virtual canvas)
2. *Turning right* (turning 20 degrees counter-clockwise on the virtual canvas)
3. *Go straight* (no turning)

The virtual car in the interface gets what action to be taken when it gives its state to the neural network. The action is performed by the car and we can see the car moving in the direction of the action visually in the front end.

Rewards:

In reinforcement learning the agent should be given a reward whenever it performs an action, or whenever it reaches certain good state or a bad state by doing some action in order for it to learn to make the best action for a given state of an environment. In this environment the rewards are given as follows,

1. *-1*, when the car crashes with an obstacle
2. *-0.1*, when the car is moving further away from the goal
3. *0.1*, when the car is moving closer to the objective.

After performing an action given by the neural network, the car updates its location on the canvas and one of the events mentioned above will occur and the agent gets a reward for that.

After implementing an admissible front-end interface where a user can create obstacles and the car can move on the interface by performing actions, next the implementation details of the brain of the car (implementation of the deep Q-Network) is discussed.

Implementing Deep Q-network:

We used PyTorch, a python deep learning research platform, to implement the deep Q-network model. Deep Q-learning algorithm mentioned in the introduction of this paper was implemented by adding a neural network in such a way that it can predict the action the agent has to take by taking the input as the present state of the agent.

Neural network architecture:

The neural network architecture consists of an input layer, hidden layer, and an output layer (as shown in Fig. 2) with an activation function and an optimizer. The input layer of the neural network has 5 nodes because the state of the car is represented using a tensor of dimension 1*5, since it is the input layer when the state of the car is inputted into the neural network the 5 nodes will be in fact the 5 variables of the car state. The hidden layer has 100 nodes of randomized weights which seems reasonable for a problem like this. The output layer has 3 nodes because it has to give 3 Q-values as outputs which correspond to 3 actions of the car. Only one hidden layer was used for this project adding more and more layers might be useful when the state of the car is represented using more number variables and when there are lot of actions the car can take not just 3. An activation function ReLU was used for. Adam optimizer was used to test the optimization of the agent, stochastic gradient descent was also used for trial.

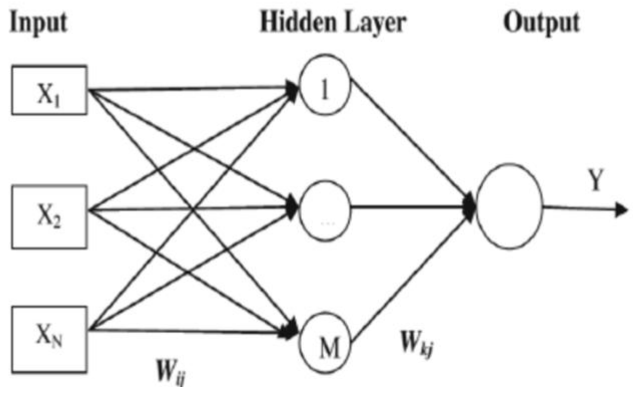

Figure 2: Neural Network Architecture

Initially the neural network hidden layer has random weights so if it is asked to predict an action given a state of the car it will not be able to give the best action possible since it did not start learning yet, but eventually with the rewards and the actions returned by the neural network using softmax function it is going to learn.

Implementation of the learning process using experience replay:

After setting up the environment of the project, i.e. the interface and the brain in the form of DQN we let the agent explore the environment by making transitions. The agent inputs its state to the neural network architecture, that was created, which in turn predicts the action the agent needs to perform. The agent then performs this action to get to a new state and gets a respective reward but be aware the learning process for the agent has not begun yet.

In the first step of the learning process the requirement is to save all of these transitions as experiences in an experience replay memory array. After the memory has at least 500 of these transitions then the learning process is started. The implementation goes as follows, we take 100 of these transitions which are in the form of (s, a, r, s’), i.e. (presentState, action, reward, nextState) and convert them into a batch ([s0, a0, r0, s’0], [s1, a1, r1, s’1], [s2, a2, r2, s’2], …...). The 100 experiences that were chosen should have a normal distribution in order to cover all the variations of transitions.

After getting the batch of normalized experiences using experience replay memory, Q-learning is combined with the prediction power of the neural network to create a target and prediction respectively. This is implemented by putting all the present states in the batch experiences into a long tensor which is sent into the neural network to get a long tensor of predicted Q-values. As we already have the next states for all the present states, next we put them in a long tensor and send them into the neural network likewise to get the predicted Q-values. We already know that the Q-value of an action at the present state is equal to the reward it gets for that action summed up with the discounting factor multiplied by maximum of all the Q-values in the next state. Hence, the Q-values we get to a long tensor for all the present states should be equal to the reward plus discount factor multiplied by the Q-values we get to a long tensor of next states.

But the neural network isn't perfect yet, hence there will be difference of error between them. That is when the loss function and the optimization come into the picture. In the process of reducing the loss function and optimizing the weights of the neural network, to make the predicted present state Q-values equal to the target present state Q-values (calculated using the next-states), the neural network will learn and train the agent to improve its actions in the environment.

Thus, the update on the front-end interface occurs when the neural network will give an action to the car, not just a random action but the action given by the network which is learning in the background. As soon as the car gets an action from the network it moves in the direction of the action, visually we can see this on the canvas that car is moving and it gets a reward based on what happened after the action, i.e if it hits an obstacle it gets penalized badly or if it is going towards the goal or reached the goal it gets a good reward. In this way the front-end and the backend is connected and the whole system moves forward.

IV. Results

Reinforcement learning is based upon the rewards which we give to the agent upon doing a good thing or a bad thing, hence we can also evaluate the agent's performance by plotting a graph between the cumulative reward the agent accumulates as a function of a time or we can also see the reward average of the agent modelled across time. In this project as we already discussed how the rewards are given to the agent in the methodology section, we keep track of them in memory to evaluate the agent once it has taken a considerable number of steps. Now let us see some results in different conditions of the environment. There are two rewards that are being compared,

Cumulative reward: It is the reward accumulated till now after performing n number of actions.

Average reward: The ratio of cumulative reward and the number of steps it took to get that cumulative reward.

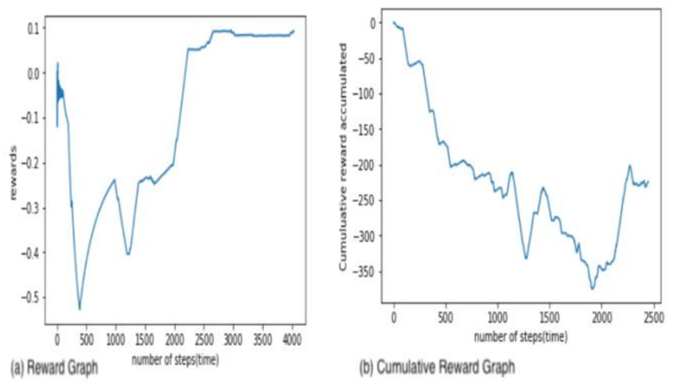

Figure 3: Number of actions vs rewards graph with no brain

In Fig. 3 we see the plots of the car moving in an environment without any learning. When there is no learning, or no brain given to the agent the neural network just gives random actions as outputs which for the car to perform. Since, there is no learning given to the car agent the cumulative reward is very low, the ideal cumulative reward curve should be an increasing curve but that is not the case here since the agent is just performing senseless actions.

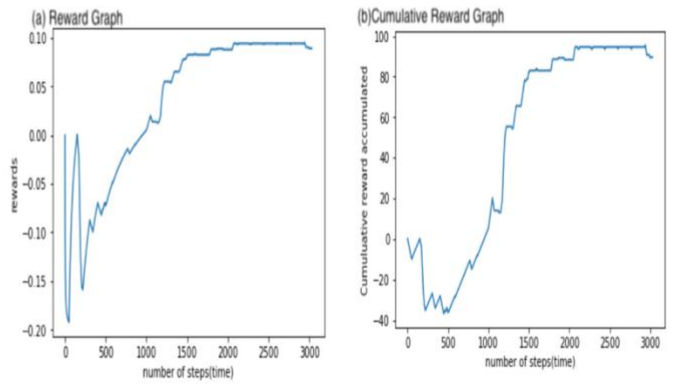

Figure 4: Number of actions vs rewards graph with brain, agent making round trips

In Fig. 4 we see the agent with learning, the plots of the average reward and cumulative rewards as function of number of steps is seen for the agent with learning respectively. This is a simple environment with no roads and obstacles, the goal of the agent that is to move from one state to the goal state. Since the environment is simple for the agent, it learns pretty quickly as we see in the fig 4a, the training started with ups and downs when the agent started to learn and then plateaued in 2000 steps after it learned which is pretty quick. The cumulative reward is an increasing curve from the starting which is the ideal nature expected from the agent.

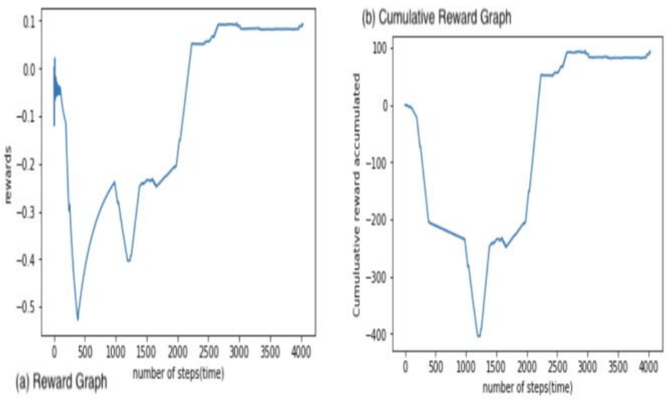

Figure 5: Number of actions vs rewards graph with a simple road

In Fig. 5 the performance of the agent in an environment with simple road is plotted. Since there is a road now, the environment is bit complicated then the environment without any road. Hence, the agent took more time that is 3000 steps for the agent to plateau the reward curve. As you can see in 5b the cumulative reward increased rapidly over time that is because the agent learned the simple road very well and is performing greatly hence accumulating so much positive reward.

Other evaluations and parameter tuning (when given +1 reward):

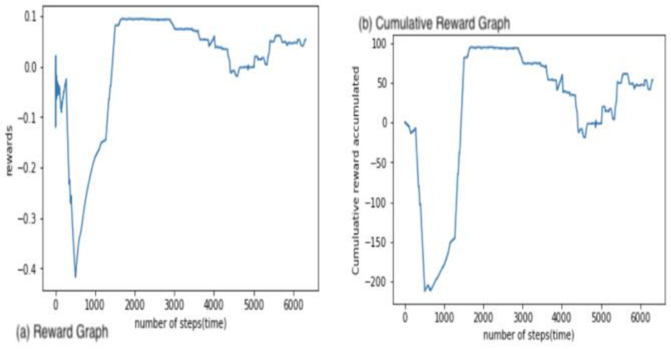

Figure 6: Number of actions vs rewards graph with a complicated road(obstacles)

In Fig. 6 wee see the agent performance in a complicated environment with lots of obstacles and Knotty roads. We see the agent doing very bad in the starting stages and gradually improves his average reward. The agent improves as a function of time since he is learning by each and every experience he gets in the environment in the form of rewards. In the starting steps the agent did not know how to handle the complex environment that is why he was hitting the obstacles and got penalized badly for that. But he learns from his mistakes and improves gradually. The cumulative reward curve is an increasing curve and it took the agent 6000 steps to converge. It took him more time to converge for the complex road then the simple road.

Importance of experience replay:

Experience replay will allow the agent to see the n steps ahead and learn complicated environments. Without experience replay the agent would only see the near future and will only decide what to do in that case which won't work in a complex environment.

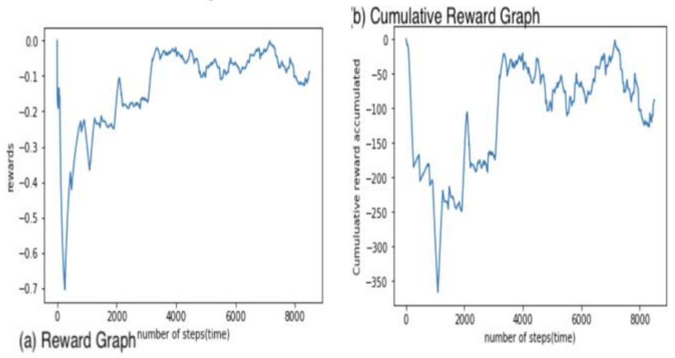

Figure 7: Number of actions vs rewards graph without experience replay

In Fig. 7 we see the agent performing without experience replay and he is doing well because it was simple environment given to it, though he decided what to do just seeing the environment one step ahead he performs well because there is complexity in the environment. Which would not be the case if the environment is complex.

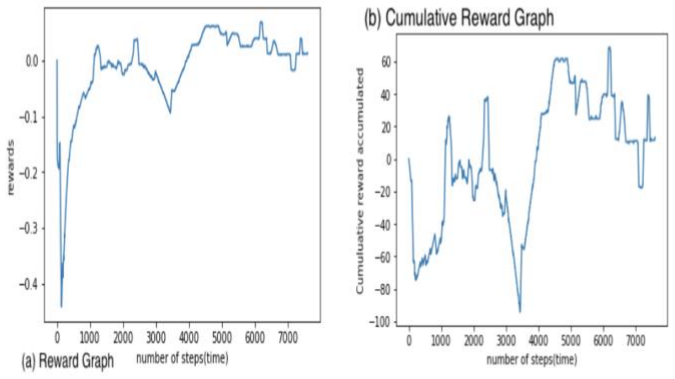

Figure 8: Number of actions vs rewards graph with experience replay

In Fig. 8 we see the agent performance with experience replay. It badly fails in the starting since it acts seeing n steps ahead from the current state, but it learns all kinds of complex roads.

We also observed that when the agent was given +1, -1 reward for going towards and away from the goal respectively, it learnt lot quicker than when the agent was given +0.1, -0.1 reward.

V. Discussions

From all the results observed a common pattern can be observed. That is in the starting of the learning process the agent fails badly and accumulates negative rewards. This happens because the agent does not know what actions to be taken but then it learns from the rewards it gets and starts learning gradually. The agent's average reward gradually increases from the start state to the middle stage because of the learning. The complexity of the environment is directly proportional to the learning time of the agent. Hence, based on the proportionality the agent might take less time or more time to learn the environment and do well consistently that it the learning will be plateaued. In our results we observed that even without experience replay the agent did well in not so complex environments, this is because the agent just looks one step ahead and decides what to do and will not encounter extraordinary situations. Hence, the agent will triumph in simple environments, but this will not be the case when there is a complex environment. Agent learns better with experience replay by looking n steps ahead.

The performance of the agent varies by tuning different parameters in the environment. The parameters on which trial and error can be performed are as follows:

- Temperature adjusting (telling the SoftMax function about the trade-off on exploration and exploitation)
- Different rewards and Additional rewards (Ex. The agent would not take an optimal route. A reward that incentivized lowering the number of steps to get to an objective in comparison to the last lap was added.)
- Different neural network architecture (Ex. Layers, # of Nodes, Optimizers, Activation Functions)
- Memory size in experience replay etc.

The main challenge we encountered was, the agent struggled a lot on finding its way, it got stuck in small loops. We found that this happened due to the exploration and exploitation trade off variable T of the SoftMax function, it seems that the agent was totally inclined to exploring and sometimes got stuck in the local minimum. To overcome this problem, we adjusted the T variable such that it also took exploitation into consideration which helped the agent to find its way when it got stuck in a loop. After the T was adjusted and after a while of training (say 15 minutes or so) the agent didn't get lost. Also, with the addition of some obstacles it started getting lost again but eventually it also found its way back through the map.

Another challenge that was faced was when complicated roads were painted, i.e. extreme curves, the road became too complicated for the agent to find its way from the start state to goal state. We overcame this problem by applying variations for rewards, i.e. there was an increase in negative reward when the agent would collide with the road and by punishing it less when it's getting further away from the goal. Because of the increase in the negative reward (when colliding with road) and decrease in punishment (when it gets further away from the goal) the agent started learning more better and the actions it performed were comparatively better.

It was observed that different configurations will do better than other configurations based upon the environment, different configurations with different parameters might perform well for different kinds of problem. Hence, one should always do trial and error on parameter tuning and see what works for their environment.

Conclusion

We finally would like to conclude by saying that we successfully built a self-driving car using Deep Q-Learning algorithm combined with experience replay. We accomplished the goal of our project, the car agent learning a considerable policy to drive from a given state to the goal state by avoiding the obstacles it encounters in its way. There is a lot of scope for future work in the domain of Deep Q-learning in terms of speeding up the process of learning and enhancing the learning process. One development could be adding additional layers to the neural network can be added and observe how the agent performs when more complicated roads are painted. The 3 most hot topics which are definitely worth looking into are Prioritized experience replay(prioritizing the important experiences to the agent so that those will be replayed frequently), asynchronous methods for deep Q-learning (instead of just using one agent for exploring the environment multiple agents with a environment copy of their own will explore and learn simultaneously) and deep Q-learning with active demonstration(giving some demonstration to the agent in some complex situations so that it will learn those situations). Implementation of these three concepts in our self-driving car agent would be a great future scope to work with.

References

[1] Christopher J.C.H. Watkins; "Q - Learning"; 1992; http://www.derongliu.org/adp/adp-cdrom/Watkins1992.pdf

[2] Martin van Otterlo; "Markov Decision Processes: Concepts and Algorithms"; May 2009; https://pdfs.semanticscholar.org/968b/ab782e52faf0f7957ca0f38b9e9078454afe.pdf

[3] Richard Bellman; "The Theory of Dynamic Programming"; July 1954; https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf

[4] Richard S. Sutton; "Learning to predict by the methods of temporal differences"; August 1988; https://link.springer.com/article/10.1007/BF00115009

[5] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, Senthil Yogamani; "Deep Reinforcement Learning framework for Autonomous Driving"; 30th Conference on Neural Information Processing Systems (NIPS 2016); https://arxiv.org/pdf/1704.02532.pdf

[6] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, Senthil Yogamani; "End-to-End Deep Reinforcement Learning for Lane Keeping Assist"; April 2017; https://openreview.net/pdf?id=ByjDCQgke

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller,

Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amior Sadik, Ioannnis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis;, “Human-level control through deep reinforcement learning ”; 2015;

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller; “Playing Atari with Deep Reinforcement Learning”; DeepMind Technologies, 2013;https://arxiv.org/pdf/1312.5602v1.pdf

[9] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver; “Prioritized Experience Replay”; ICLR 2016;https://arxiv.org/pdf/1511.05952.pdf