On improving traffic flow using deep reinforcement learning

N.M.T. Vijay, Vihang Agarwal, Ananya Trivedi
University of Michigan, Ann Arbor
{nmtvijay, vihang, ananyatt}@umich.edu
Github-Repo

1 Motivation

The United States being highly auto-dependent, with the 307 million people owning 150 m cars, an average American spends 17000 minutes a year behind the wheel. This implies an unnecessary investment of a substantial amount of one's time and effort and also comes with its own set of issues and troubles like potential accidents, human fatigue, time wasted behind the wheel in traffic congestion, having to drive in limited visibility etc. Self driving cars are the solution to mitigate if not completely eliminate these problems.

To realize efficient self driving agents, the task of optimizing traffic flow will be a challenge. Controlling the pace car and the lane changes for each agent takes will enable us to control traffic congestion, reducing delays and travel times. This is a complex problem due to various reasons: one is the existence of a large state space for the agent at any given instance and another is there being no definitive measure of optimality that it can be drive towards. This makes traditional motion planning methods like A* and rrt very hard to implement in this setting. We thus decided to explore Machine Learning based approaches as an alternative, specifically the family of 'Reinforcement Learning' algorithms. The use of neural networks to estimate the state-action value function in a reinforcement learning (RL) framework has recently been demonstrated to be a remarkably powerful way to learn how to succeed in a very large state space with no prior knowledge.[6]

We explore reinforcement learning as a possible solution for decision making for these fully automated driving vehicles in real urban traffic scenarios and model traffic scenarios by a simulated micro-traffic environment. The perception, control, and planning system for one of the car is all handled by a single neural network as part of a model-free, off-policy reinforcement learning process.

2 Problem Statement

For an intuitive understanding of this problem it is necessary to first look at our agent, the environment it operates in and then attempt to develop a technique for meaningfully driving it while maximizing its speed.

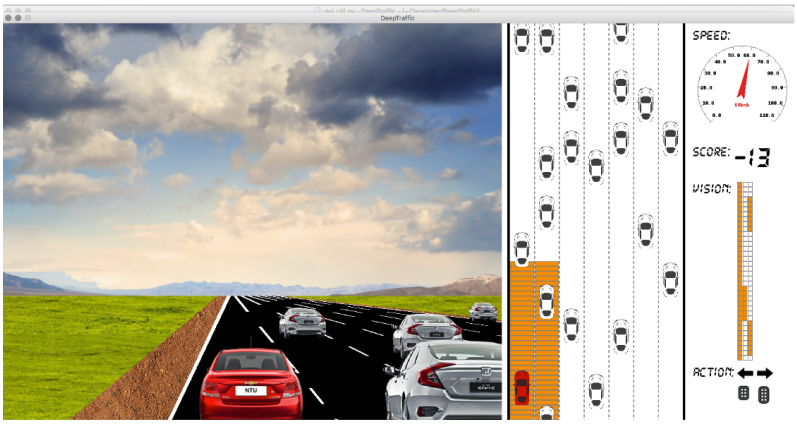
Let's consider the traffic flow simulator.

It uses a discrete occupancy grid as a simplified representation of the world. Each cell contains the speed of the car occupying it. The focus of this simulation is to learn efficient movement patterns in heavy traffic, the problem of avoiding collisions is abstracted away by using a "safety system". This safety system looks at the occupancy grid and prevents actions that would lead to a crash, for example accelerating when there already is a car in front or changing lanes with a car on the next lane.[2]

All cars have a choice of five actions, changing lanes to either side, accelerating or slowing down and simply doing nothing/keeping the same settings as before. The other cars choose the actions at random, e.g. changing lanes if the safety system permits. The random sampling of these actions is

biased towards patterns that seem realistic, for example not changing the lanes too often in general, but with a higher probability if stuck behind another car.

Further, there is one car (displayed in red) that is not using these random actions. This is the car controlled by the deep reinforcement learning agent. It is able to choose an action every 30 frames (the time it takes to make a lane change) and gets a cutout of the state map as an input to compute its actions. Knowing every agents optimal speed, the best lane to drive in, or the best route to take, traffic flow could be optimized. We cast this as a motion planning problem.



This is a python based implementation similar to the Deep-Traffic simulator MIT provides for its Deep-Traffic competition.[5]

Markov decision processes (MDPs) are a general framework to model planning and decision making problems. Environments which follow a structure where a given state conveys everything the agent needs to act optimally are called 'Markov Decision Processes (MDP). Executing an action $u \in U$, given the system is in state $x \in X$, is what will be called a policy $\pi : x \rightarrow u$. The goal of such a planning problem is to find an optimal policy (sequence of actions) π^* that maximizes the expected reward over the time horizon T. A commonly applied approach to find an optimal policy is using value iteration.

Anyhow, true system states are typically partially observable. What this means is the information available to our agent is spatially and often times even temporally limited. Partially observable Markov decision processes(POMDPs) help to accommodate this limitation by introducing the idea of a belief $bel(x_t)$ of being in a state x_t at time t.

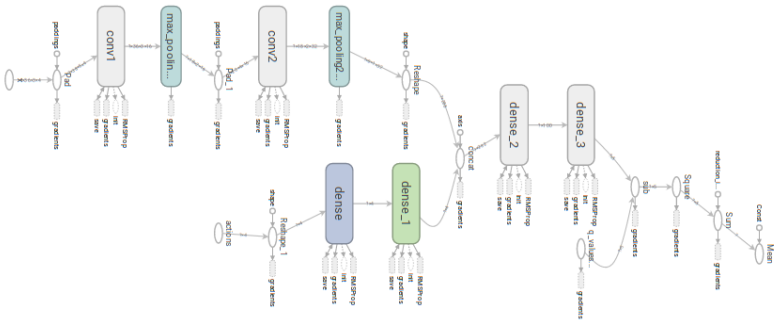
Based on these ideas, we build a deep neural agent that performs well in this setting. This paper has shown that recurrent blocks in neural agents can handle partially observable environments[1]. We implement and explore Action specific Recurrent Deep Q networks(ADRQN) and compare them with Action specific Deep Q networks(ADQN) believing that information over time will enable the agent to make good decisions.

3 Methodologies Explored

Within the context of Reinforcement learning, Deep Q networks(DQN) [4] and Deep recurrent Q networks(DRQN) have proved to be successful larger and complex implementations of Q learning algorithms as value iteration methods. There is established literature on this family of methods. Our goal is to propose a model-free deep RL approach that incorporates the influence of the performed action through time. We go beyond and look at Action specific Q networks inspired by these works.

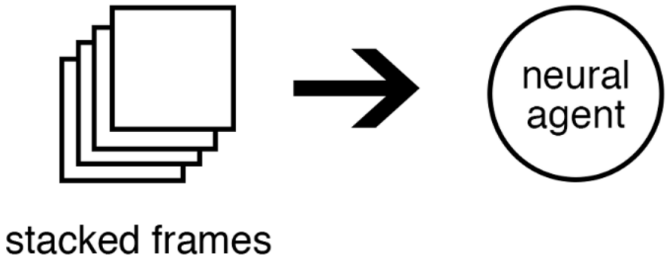
ADQN [4] attempts to couple actions and observations as the input to the Q network. Thus the model is able remember past actions. DQN or DRQN only consider the convolutional features of the history of observations instead of explicitly incorporating the actions. However, the action performed is crucial for belief estimation in POMDPs.[6] argue why the past history of actions is important while giving a formal mathematical definition of the POMDP setting and formulations that estimate this belief.[5] implements ADQN to the same problem we try to tackle, But a key factor in solving such problems is the capacity of an agent to integrate temporal observations. The intuition being: if

information at a single moment isn't enough to make a good decision, then varying information over time probably is. DQN (or ADQN) suggests this temporal integration by using an external frame buffer which keeps the last few frames of the environment in memory and feeds this to the neural network. This involves serious memory issues as the environment gets complicated and the number of frames increase. An alternative is to move the temporal integration into the agent itself by utilizing the RNN network, which is what ADRQN attempts at.

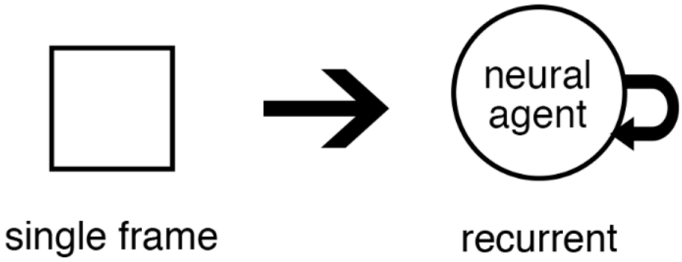


Graph of the ADQN as visualized by Tensorboard

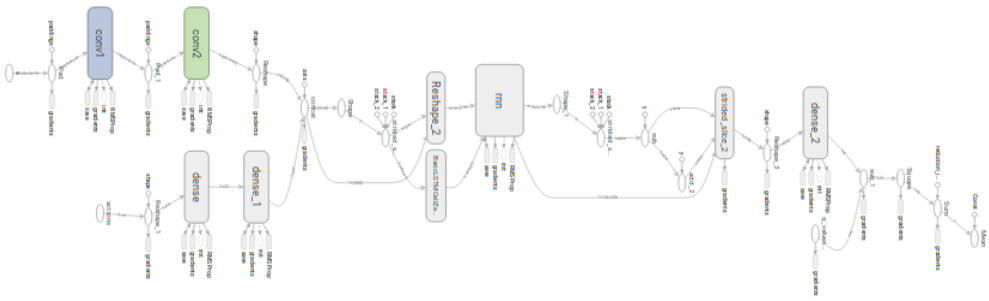
By utilizing a recurrent block in our network, we can pass the agent single frames of the environment, and the network will be able to change its output depending on the temporal pattern of observations it receives. It does this by maintaining a hidden state that it computes at every time-step. The recurrent block can feed the hidden state back into itself, thus acting as an augmentation which tells the network what has come before.[6]



Input state space that DQN proposes to deal with partial observability.



A general recurrent setting framework.[3]



The graph of the ADRQN as visualized by Tensorboard

(i) We implement ADRQN by adding an LSTM recurrent cell to the ADQN framework. The input to the LSTM layer is the concatenation of output of the action layer and output of the observation state layer. The output of the LSTM is then fed into the dense output layers.

(ii) We change the way the experience buffer stores memories. We make it store entire episodes, and from these episodes we draw random batches of fixed trace length. This way we retain random sampling and also make sure we have traces of experience that follow form one another.

But implementing and deploying RL algorithms on these problems poses other challenges as well. Hyper parameter configurations need to be tuned for optimal model selection. [2] try to achieve this through large scale open competitions demonstrating the need and difficulty of this task. Other challenges involve engineering rewards and effective exploration. RL is based on a trial-and-error process and in complex tasks, effective exploration remains a challenge. We briefly describe about the hyper-parameters and rewards used to model our agent without tuning these or engineering rewards specific for our task. Taking into account the limited time available for this project, we build upon previous works and competitions on similar tasks for these values. We are more interested in effective exploration strategies used, in context of our problem. To study the impact of action selection strategies as effective exploration paradigms, we describe and implement them in detail and try to get a comparative analysis.

3.1 Hyper Parameters Used

The hyper parameters for the ADRQN implementation are:

Table 1: Hyperparameters

activation function	tanh
number of episodes	2000
Grid cells to the front of the car	20
Grid cells to the back of the car	5
Grid cells to the sides of the car	1
batch size	4
trace length	8
number of convolutional layers for observation state layer	2
number of LSTM units	256
Final dense layer for output = num of actions	5
temperature for boltzmann exploration	0.5
dropout probability	0.5
learning rate	0.001

3.2 Reward Modeling

The objective here is to maximize the average speed of the agent over a number of episodes while using a stable strategy. In our framework, a stable strategy maintains that the agent not change its lane too often as it is impractical in an urban scenario. We impose a penalty of -0.00001 in case of lane switches. We define the maximum speed that the agent can achieve to be 110km/hr and a high

reward of 30 on achieving this goal. The discount factor is 0.98 and the rewards associated with each action are:

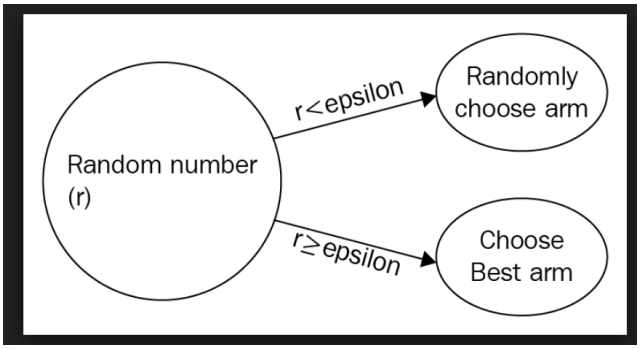
Left $\rightarrow$ 0.01, Right $\rightarrow$ 0.01, Accelerate $\rightarrow$ 0.30, Decelerate $\rightarrow$ 0.20, Maintain $\rightarrow$ 0.50

3.3 Exploration Strategies

The process of obtaining representative training data in Reinforcement Learning is called exploration. The following techniques were used by us in the training process:

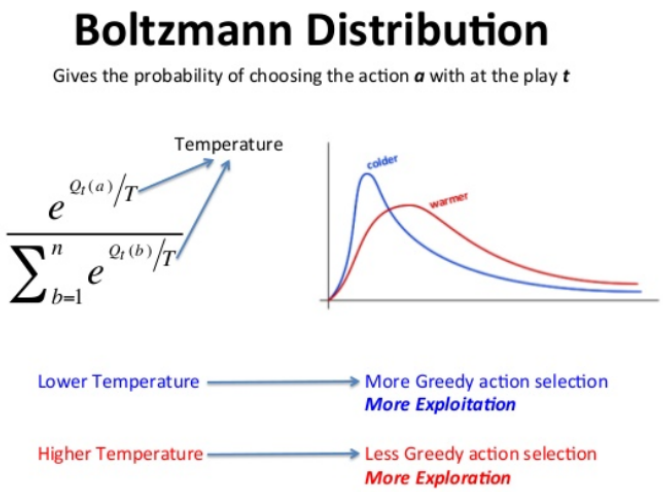
(i) A simple combination of the greedy and random approaches yields the ϵ-greedy exploration strategy. Consider the case of the multi-arm bandit problem. The two goals here would be to try a few coins and determine which yields the best results and then exploit the gained knowledge to earn as much as possible. If a fixed set of experiments are conducted and the data of winning from each machine is logged, the ϵ greedy approach involves selecting the machine with the current average highest payout with probability (1-ϵ)+(ϵ/k), where k is the number of machines and you select the machines that dont have the current highest payout with the probability ϵ/k. The ϵ in ϵ-greedy is an adjustable parameter which determines the probability of taking a random, rather than principled, action. Due to its ease of use,this approach has become the defacto technique for most recent reinforcement learning algorithms, including DQN and its variants.

Our implementation - we start with a high epsilon value to encourage random exploration and then decay it linearly over the number of iterations performed assuming agent learns what it needs to learn about the environment as iterations increase.



ϵ-Greedy Method [3]

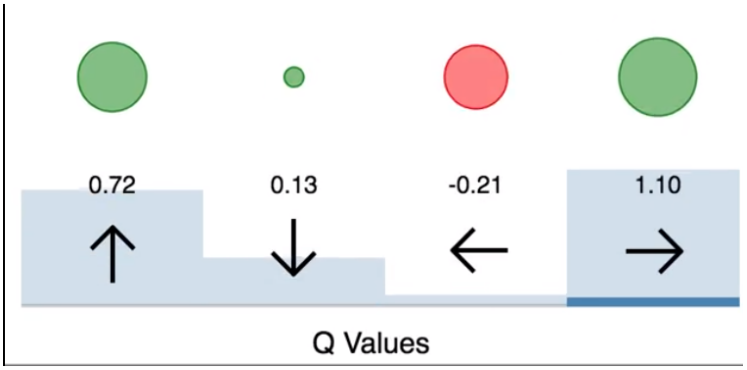
(ii) During exploration, the agent would ideally like to exploit all the information present in the estimated Q-values produced by our Deep network. This approach is followed by Boltzmann Exploration. The key here is choosing the action based on weighted probabilities rather than any element of randomness.In machine learning, a Softmax layer is used, to map the non-normalized output to a probability distribution over predicted output classes. This way, the optimal action is most likely to be chosen though not guaranteed to be chosen. The biggest advantage over e-greedy is that unlike epsilon-greedy where all non-optimal actions are estimated equally, in Boltzmann exploration they are weighed by their relative value. This probability based action selection makes sure that the agent ignores actions which it estimates to be largely sub-optimal and gives more attention to those actions with higher probabilities, but not necessarily ideal. We implement this approach similarly to that stated in Arthur Juliani's Blog. [3]



Boltzmann Exploration [3]

(iii) A Bayesian neural network(BNN) is a neural network with a prior distribution on its weights. In a reinforcement learning setting, the distribution over weight values allows us to obtain distributions over actions as well. The variance of the probability distribution of weight values provides us an idea of the agent's uncertainty about each of it's actions. Due to the sheer volume of weights in most practical networks, it is impractical to maintain a distribution over all weights. Instead we can utilize dropout to simulate a probabilistic network. Dropout is a regularization technique for reducing over-fitting in neural networks by preventing complex co-adaptations on training data [1]. By repeatedly sampling from a network with dropout, we can obtain a measure of uncertainty for each action. When taking a single sample from a network with Dropout, we are doing something that approximates sampling from a BNN.

Our Implementation - we add a variational dropout wrapper in the LSTM layer of our network with the dropout probability of 0.5, as variational dropouts have shown to work well for recurrent networks. The action selection strategy is still ϵ - greedy here. We also tried adding dropouts before and after the recurrent layer but it did not seem to provide any performance boost.

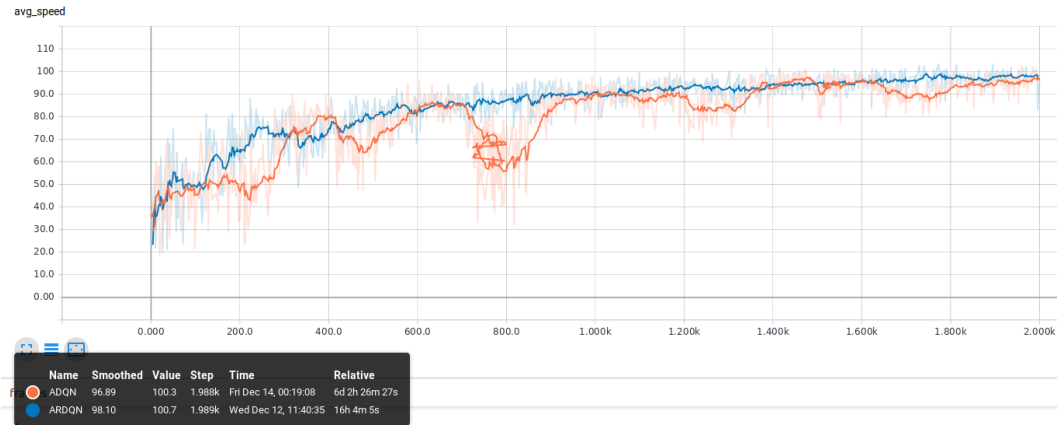


Light Blue-Probability of choosing an action; Dark Blue-Action Chosen; Each change in value corresponds to a new sampling from a BNN using dropout [3]

4 Evaluation of Methods

We trained the ADQN reference model, ADRQN with ϵ-greedy exploration, ADRQN with boltzmann exploration and the two variations of dropout exploration for 2000 episodes each. We use the average speeds over the episode of our agent as the primary performance criteria. The higher the average speed the better the agent is able to perform. Other measures of performance are convergence rate and stability of the agent.

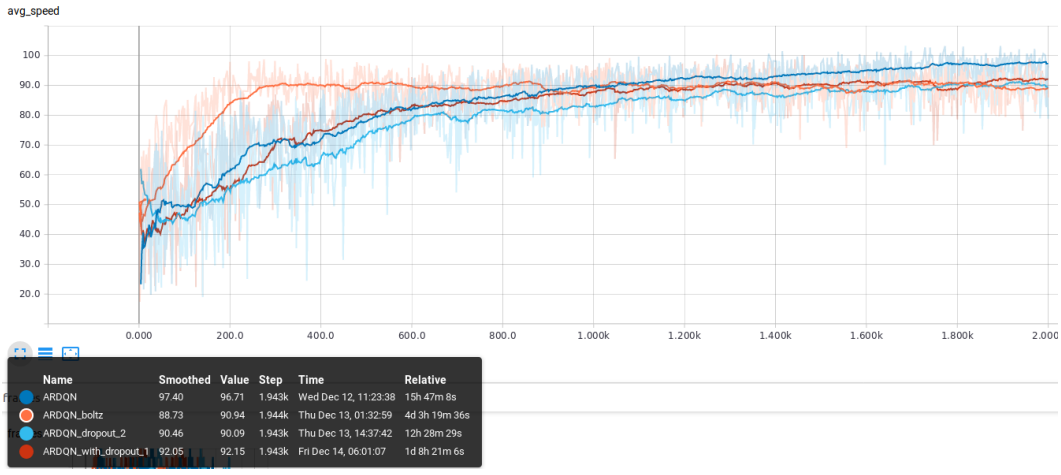
4.1 ADQN vs ADRQN



Average speeds comparision between ADQN and ADRQN networks,
ADQN-orange,ADRQN-blue

Doing a comparitive study between ADRQN and ADQN with ϵ-greedy exploration policies. We can see that ADRQN clearly performs better than the ADQN with respect to model stability and convergence of average speed to the optimal performance. The average speed at the end of 2000 episodes for the ADRQN is 98 and ADQN is 95. There is a clear improvement in the model performance with a recurrent layer added to the ADQN network. This agrees with our hypothesis of integrating temporal observations to predict q values and approximating the q value function as an important criteria for boosting model performance. The model parameters haven't been fine tuned yet. An ideal comparison would involve finding the optimal hyperparameter configurations and then comparing the role recurrent network plays for the given problem. Anyways, our implementations provide with a good start towards this study and help form an intuitive bias why recurrent networks work well in these partially observable settings.

4.2 Exploration Policy Comparisons



Average speeds comparision between different exploration policies with ADRQN,
linear epsilon - greedy-dark blue, Boltzmann - orange,
Dropout - light blue, Variational Dropout - red

Next we compare ϵ-greedy exploration with a linear decay, Boltzmann exploration and variations with Dropouts as exploration policies. Looking at the average speeds at the end of 2000 episodes, ϵ-greedy performs has the best average speed of 98, starts converging to the optimal value while learning a stable model. The Boltzmann approach has a really fast convergence to the optimum, at about 300 training episodes and is quite stable while it fares worse off in terms of the maximum

average speed it achieves. Using variational dropouts on the recurrent layer though improves the performance over Boltzmann approach but loses its property to converge much faster. Using dropouts on the dense layers in the network gives the worst performance both in terms of achieving an optimal average speed and convergence rate. For a fair comparison we assume that the hyper parameters and the model parameters used are optimal and then see how different exploration policies lead to differing agent behaviours under the same set of parameters for an architecture used.

Table 2: Avg speed comparisions

Network Architecture	Avg speed(km/hr)
ADQN	95
ARDQN with linear epsilon greedy	98
ARDQN with Boltzmann	88
ARDQN with variational dropout	92
ARDQN with dropouts	90

5 Conclusions

We started with the aim of exploring machine learning techniques for optimizing traffic flow in partially observable environments. We explored ADQNs and ADRQNs as possible methods of implementing Reinforcement Learning in this setting. We were able to clearly discern, that ADRQNs perform better in partially observable environiments than ADQNs as demonstrated by P .Zhu *et al*.. [6]. We were further able to test action exploration policies for the setting and observed that they perform worse in terms of optimal speed but may lead to a faster convergence for our choice of hyperparameters. We believe that these exploration policies are dependent on hyperparameter tuning and that is an essential task to guarantee performance.

There is also a body of work in effective exploration strategies that deals with the nature of reward signals or uses deep exploration in similar scenarios rather than just dealing with action selection strategies. As a future extension we would like to look into these policies and compare their performances in the context of our problem. We would also like to derive a mathematical basis for these action selection strategies explaining if perform well or not in this regard.

6 Description of Individual Effort

- All three members of the group were involved in meeting up with GSIs for their guidance over the approach. The final report was also a combined effort from all three. The individual efforts are as detailed below:

- Vihang – Conducted survey for the environments that we can use for the project ranging from Deep Traffic to the GitHub repository we ended up using. Chose the perfect hyperparameters to be used resulting in avoidance of local minima. Was mainly responsible for developing the ADRQN architecture being used in the project. Detailed out the results the code gave and why they made sense.

- Ananya – Conducted literature review for baseline paper to be implemented given the environment. Tuned the environment to be used from the one on GitHub repository to the one customized for our use case. This was a particularly tedious process since it involved going through around 3000 lines of code to ensure a seamless integration from available open source code to our adaptation of the same.

- Vijay – Did the coding in TensorFlow. Given the environment developed by Ananya and the architecture by Vihang, he developed the code for the Deep Neural networks and trained it on GPU. His modular code helped to ensure we could, despite the limited time available, simulate and train both networks and three different exploration strategies. Simulated results for the three exploration strategies we ended up using.

P.S.–Our fourth team member opted out of the project work since she decided to audit the course. Hence the absence of her mention in this report.

References

[1] Reinforcement learning. URL https://en.wikipedia.org/wiki/Reinforcement_learning.

[2] L. Fridman, B. Jenik, and J. Terwilliger. Deep traffic: Driving fast through dense traffic with deep reinforcement learning.

[3] A. Juliani. Simple reinforcement learning with tensorflow part 6: Partial observability and deep recurrent q-networks. https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-6-partial-observability-and-deep-recurrent

[4] D. S. A. G. Volodymyr Mnih, Koray Kavukcuoglu. Playing atari with deep reinforcement learning.

[5] H. S. Yan. Reinforcement learning for self-driving cars. URL https://github.com/songyanho/Reinforcement-Learning-for-Self-Driving-Cars.

[6] P. Zhu, X. Li, P. Poupart, and G. Miao. On improving deep reinforcement learning for pomdps.