Decision Making in Dense Traffic using DQN

Yahsiu Hsieh, Yuhao Liu, and Ramtin Hosseini

***Abstract*— The deep reinforcement learning community has made several independent improvements to the DQN algorithm. However, it is unclear which of these extensions work best in the autonomous driving problem. This paper examines four extensions to the DQN algorithm and empirically studies their performances in two different real-world scenarios. Our experiments show that the Dueling DQN provides the best final performance under these two scenarios.**

I. Introduction

The many recent successes in scaling reinforcement learning (RL) to complex sequential decision-making problems were kick-started by the Deep Q-Networks algorithm (DQN; Mnih et al. 2013, 2015) [1]. Its combination of Q-learning (van Hasselt 2010) [2] with convolutional neural networks and experience replay enabled it to learn, from raw pixels, how to play many Atari games at a human-level performance. Since then, many extensions have been proposed that enhance their speed or stability.

Double DQN (DDQN; van Hasselt, Guez, and Silver 2016) [3] addresses an overestimation bias of Q-learning, by decoupling selection and evaluation of the bootstrap action. Prioritized experience replay (Schaul et al. 2015) [4] improves data efficiency, by replaying more often transitions from which there is more to learn. The dueling network architecture (Wang et al. 2016) [5] helps to generalize across actions by separately representing state values and action advantages.

Each of these algorithms enables substantial performance improvements in isolation. Since they do so by addressing radically different issues. In this paper, we propose to study an agent that aims to solve a real-world autonomous driving problem using all the aforementioned models. We show how these methods perform in different tasks, and that they are indeed able to solve real-world autonomous driving problems.

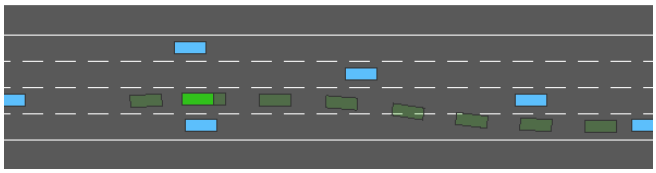
II. Background

In this section, we first describe the problem using Markov Decision Process and then briefly describe several previous works that our method is built upon.

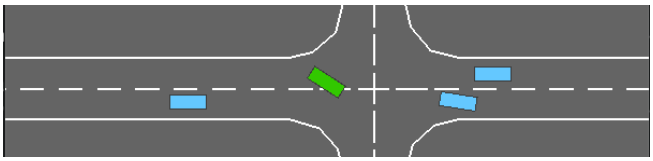
A. Problem Formulation

For decision making and planning, a model can be described using the Markov Decision Process (MDP). In our scenarios, a vehicle needs to predict the best policy within several high-level actions such as "speed up" or "brake" to safely pass through dense traffic. There are several crucial elements for an MDP M: states $s \in S$, actions $a \in A$, a reward function $R(s, a)$, and a state transition matrix $T(s'|s, a)$ that represents the probability of the next state given the current status. A policy $\pi(a|s)$ is defined as an action distribution based on states. For an MDP, the objective is to look for an optimal policy that could maximize the total reward.

Y. Hsieh, Y. Liu, and R. Hosseini are with Department of Electrical and Computer Engineering, University of San Diego, La Jolla, CA 92093 USA. {y4hsieh, yul133, rhossein} @ucsd.edu



(a) Highway



(b) Intersection

Fig. 1: The task is to navigate in dense traffic with reasonable speed while avoiding collisions with neighbouring vehicles. Fig. above shows the highway and intersection scenario in our experiments.

B. Q-Learning

Q-learning is a model-free reinforcement learning algorithm to learn the quality of actions telling an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards. For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy based on a calculated Q-value. Q-value is the reward obtained from the current state plus the maximum Q-value from the next state.

$$Q(s_t, a_t) = R_t + \gamma max Q(s_{t+1}, a') \tag{1}$$

Here s is the state and a is the action and $Q(s, a)$ is a value of the Q table cell and R is the reward and γ (between zero and one.) is the discount factor.

C. Deep Q-learning

The only difference between Q-learning and Deep Q-learning is the agent's brain. The agent's brain in Q-learning

is the Q table, but in Deep Q-learning, the agent's brain is a deep neural network. The input of the neural network will be the state or the observation and the number of output neurons would be the number of the actions that an agent can take. For training the neural network the targets would be the Q-values of each of the actions and the input would be the state that the agent is in.

III. Methods

In this section, we present our designed methods for solving the problem, including Double DQN, Dueling DQN, and Prioritized Buffer. Fig. 2 shows the general idea of Dueling DQN.

A. Double Deep Q Networks

Double deep Q networks use two identical neural network models. One learns during the experience replay, just like DQN does, and the other one is a copy of the last episode of the first model. The Q-value is computed with this second model.

In DQN, Q-value is calculated with the reward added to the next state maximum Q-value. If every time the Q-value calculates a high number for a certain state, the value that is obtained from the output of the neural network for that specific state will become higher every time. Each output neuron value will get higher and higher until the difference between each output value is high. As a result, if let's say for state s action a is a higher value than action b, then action a will get chosen every time for state s. now consider if for some memory experience action b becomes the better action for state s. then since the neural network is trained in a way to give a much higher value for action a when given state s, it is difficult to train the network to learn that action b is the better action in some conditions.

Double deep Q networks solve this issue by using a second model that is a copy of the main model from the last episode and obviously since the difference between values of the second model is lower than the main model, we use this second model to attain the Q-value:

$$Q(s,a) = R_t + \gamma Q^c(s_{t+1}, argmax Q(s_{t+1}, a')) \quad (2)$$

B. Dueling Deep Q Networks

The difference in dueling deep Q networks is in the structure of the model. The model is created in a way to output the formula below:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{n}\Sigma_{a'} A(s,a') \quad (3)$$

Here, the $V(s)$ stands for the value of state s and A is the Advantage of doing action a while in state s. The value of a state is independent of action.

In previous settings, the agent might be in a state where each of the actions would give the same Q-value, so there is no good action in this state. Therefore, dueling DQN addresses this issue by dividing the Q-value by the value of a state and the Advantage that each action has. If every action has the same result, then the Advantage of each action will have the same value. Now, if we subtract the mean of all the Advantages from each advantage as in equation 3, we get zero (or close to zero) and Q-value would be the Value that the state has. So overtime the Q-value would not overshoot. The states that are independent of action would not have a high Q-value to train on.

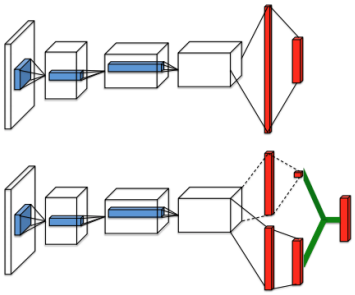


Fig. 2: A popular single stream Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation 3 to combine them. Both networks output Q-values for each action

C. Prioritized Buffer

Just like all standard algorithms for training a deep neural network, our proposed method makes use of an experience replay buffer to derive the Q function. It is the set D of recent steps taken previously. The major objective of experience replay is to stabilize the performance, breaking the ties between individual data and encourage the algorithms to reuse the helpful data. However, the batch chosen randomly might not be eligible. The importance of each data is not equal. Some are more important and some aren't. As a result, if we can recognize the importance of each collected experience. We can promote the one with larger importance and improve the result.

In our definition, the importance is a value that is proportional to the change between a Q-value from the next state and from the current state. Higher difference means higher importance. This difference is called Temporal Difference error (TD error). It is calculated as followed:

$$TD = \left|Q(s,a) - Q(s+1,a)\right| \quad (4)$$

After deriving the TD error, we can then calculate the probability of an data from the replay buffer as:

$$p_i = \frac{(TD_i + \epsilon)^\alpha}{\sum_{n=1}^{buffersize}(TD_n + \epsilon)^\alpha} \quad (5)$$

α is a random value between zero and one. which controls the portion of important experience. If $\alpha << 1$, the batch would get filled randomly. On the other hand, it chooses the most important experience. Note that ϵ is to prevent the error of division by zero.

This technique is a double-sided blade even though the importance gives us a direction to pick the previous experience wisely. It tends to choose the one with higher probability, which is potentially biased to certain experiences. In other words, the neural network might be over-fitting. To avoid this, we can define a final form for importance. It is described as followed:

$$Importance = (\frac{1}{p_i} * \frac{1}{buffersize})^{\beta} \tag{6}$$

In our implementation, β is not a fixed value. It starts from a value close to 0, gradually rises, and approaches 1 in the end. The dynamic follows the truth that the replay buffer is filled with memories having unknown importance in the beginning. At the end of the training, we can count on the importance to decide the selected batch. Finally, we can apply the importance to the loss function, which is calculated as:

$$loss = \frac{1}{k}\sum(y - y_{estimate})^2 * Importance \tag{7}$$

IV. Implementation Details

In this section, we provide the implementation details of our model, for additional details please refer to supplementary material. The proposed neural models were implemented in PyTorch. For the rest of the section, we will discuss the data generation, scene setup, different module architectures, and hyper-parameters that lead to our final results.

A. Data Collection

In this section, we will describe what simulator we choose to test our algorithms. In different scenarios, observation data will be different. As a result, we will also describe what information the algorithms take to process and generate policies.

1) Scene Setup

The simulator we choose is highway-env[1], which is a collection of environments for autonomous driving and tactical decision-making tasks. There are five environments in this simulator and we choose two of them to test our proposed methods.

The first scene is "highway". In this task, the ego-vehicle is driving on a 4-lane highway populated with other vehicles. The agent's objective is to complete a safe drive in which the ego-vehicle maintains a reasonable speed while avoiding collisions with neighboring vehicles. Driving on the right side of the road is also rewarded.

The other scene is "intersection", as shown in Fig. **??**. In this task, the objective to pass through an intersection with dense traffic. The number of the lane is set to 2 so that the agent can focus on intersection negotiation instead of changing the lane before or after passing the intersection.

[1]https://github.com/eleurent/highway-env

Env	Models	Reward	Efficacy
Highway			
	Dueling DQN	59.54 ± 6.04	92%
	Double DQN	59.26 ± 4.97	79%
	Double DQN w CNN	41.88 ± 7.36	67%
	Double DQN w PB	47.23 ± 10.56	73%
Intersection			
	Dueling DQN	1.56 ± 0.59	40%
	Double DQN	1.56 ± 0.59	77%
	Double DQN w CNN	1.44 ± 0.60	57%
	Double DQN w PB	1.59 ± 0.73	39%

TABLE I: The total mean training reward with standard deviations and mean success rates are presented in solving tasks in both highway and intersection setup.

2) Observation Data

In the highway scenario, the ego-vehicle can observe five vehicles around it. It can collect five types of information (features) from each of them: presence, position x, position y, velocity x, velocity y.

The presence refers to the status of whether the ego-vehicle observes this vehicle or not. If fewer than five vehicles are observed in a frame, the presence will be zero. As a result, we will get a 2D array with a size of $(5, 5)$.

As for the intersection scenario, the default configuration collects 15 vehicles with 7 observations to generate a policy. Besides the 5 features mentioned in the highway environment, this task needs additional 2 features: $sinh$ and $cosh$. Therefore, the final dimension of the observation will be a 2D array with a size of $(15, 7)$.

B. Models Architecture

Here we discuss how we design the model architecture for the proposed method. We implemented three types of architectures, which are Fully Connected Network, Convolutional Neural Network, and Attention-based Neural Network.

1) Fully Connected Network

The first architecture is a 3-layer DNN. The two layers have 512 and 256 nodes individually for the highway scene and 1024 and 512 for the intersection scene. The We use Rectified Linear Unit (ReLU) [6] for non-linearity. The reason that we use different settings for highway and intersection is that the input dimension, as well as output dimension, are different. Since the input dimension of the intersection scene is 105, it is apparent that a layer with higher nodes is better.

2) Convolutional Neural Network

The second architecture is a 5-layer CNN-based one, which consists of 2 convolutional layers and 3 fully connected layers. Each convolutional layer has a 3×3 kernel, a ReLU layer, and a max-pooling layer with a 2×2 kernel.

C. Hyperparameters

For all of our scenarios, we trained our model for 4000 epochs. We did update using Adagrad with a learning rate of 0.001 and a batch size of 64. The discounted factor γ is set to 0.99. For an experienced replay buffer, the maximal buffer size is 1000 and the initial size is 100. Also, We define a $\beta := 1 - \beta_{origin}$, where β_{origin} is the one that we mentioned

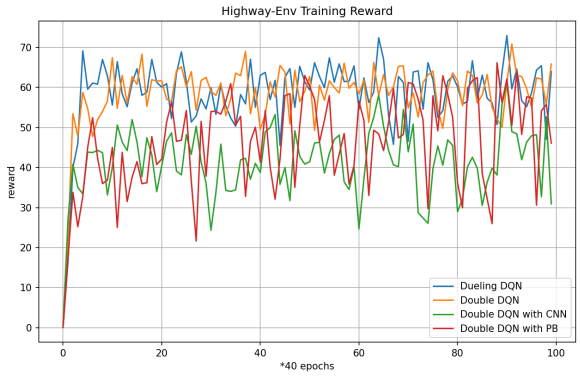


Fig. 3: The line chart shows the training reward v.s. epochs for all models in highway environment.

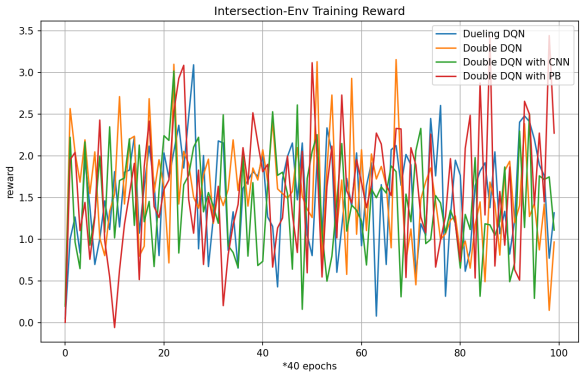


Fig. 5: The line chart shows the training reward v.s. epochs for all models in intersection environment.

Fig. 4: The box plot shows the mean and standard deviation of the training reward for all models in highway environment.

Fig. 6: The box plot shows the mean and standard deviation of the training reward for all models in intersection environment.

in the previous section, which starts from 0 and approaches 1 at the end. As for proposed methods Double and Dueling DQN, they all need a parameter that controls the update frequency between two networks. We define this parameter $learncounter = 10$.

V. Results

In this section, we present and compare the performance of our models in both highway and intersection environments.

Table I presents the total training reward with their standard deviations and the success rates of all the methods in solving both highway and intersection environment. Fig. 3 and Fig. 5 compares algorithms in line chart. Fig. 4 and Fig. 6 using the box plots of mean training reward in solving the task. We validate that Dueling DQN and Double DQN outperform others in these two environments.

From Fig. 4 and Fig. 6, we can observe some features from this task. Theoretically, Dueling DQN and Double DQN with Prioritized Buffer would have the best performance. From this experiment, Dueling DQN performs almost as well as expected. It has best distribution of training reward and performs great on the evaluation. But we notice that Dueling DQN met it's Waterloo while solving the intersection problem. From our observations, Dueling DQN is more conservative than other algorithms because it tends to pick the same action unless there is a dramatic change. As a result, it is more likely to hit a car while passing the intersection.

On the other hand, Double DQN with prioritized buffer doesn't play as we expect. It has the biggest oscillation on a training reward. We argue that in these scenarios, the idea of importance does not help the algorithm get a more eligible batch than the original version. On the other hand, Fig. 6 and Fig. 7 shows that Double DQN with PB works better than Double DQN in terms of training reward. It can reach a higher reward than the original one cannot. Note that the variance of Double DQN with PB is also higher.

VI. Conclusions and Future Work

In this paper, we present a revised version of the deep Q network that utilizes several extra techniques to tackle the dense traffic problem. Two scenarios are tested and the results are presented. From the previous section, we can compare the difference between each method and discuss the possible reasons. In our future studies, we plan to apply our method to the rest of the scenarios that the simulator has, and investigate the performance on each task.

REFERENCES

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.

[2] B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in IEEE Access, vol. 7, pp. 133653-133667, 2019, doi: 10.1109/ACCESS.2019.2941229.

[3] van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double Q-learning. In Proc. of AAAI, 2094–2100.

[4] Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. In Proc. of ICLR.

[5] Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; and de Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In Proceedings of The 33rd International Conference on Machine Learning, 1995–2003.

[6] Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted Boltzmann machines. In ICML, pp. 807–814, 2010.