Deep Reinforcement Learning on Car Racing Game

Jiahao Zhang, Minghe Ren, Weixuan Jiang, Zhonghao Guo
{ jiahaozh, sawyermh, jwx0728, gzh1994 }@bu.edu

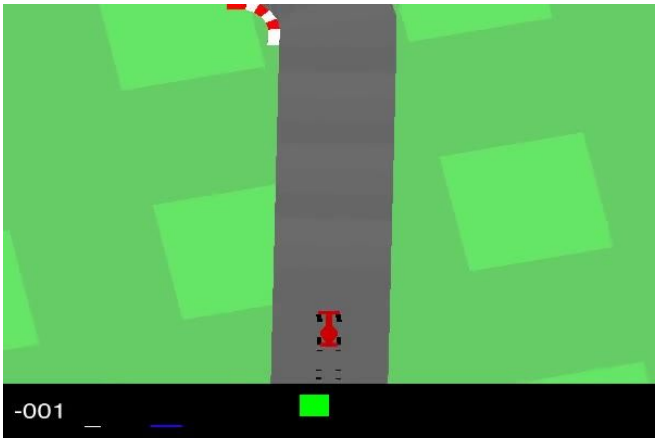


Figure 1. Car Racing Game powered by OpenAI Gym

1. Task

Reinforcement learning (RL) is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and expectation. Unlike supervised learning in which feedback provided to the agent is ground truth set of actions for performing a task, reinforcement learning adopts reward and punishment as criteria for positive and negative behaviors. Besides, it is also unlike unsupervised learning in terms of goals. Unsupervised learning is to find similarities and differences between data samples while in reinforcement learning, success is measured by reward and the goal is to find a suitable action that would maximize the total cumulative reward of the agent. The basic framework is shown as follows.

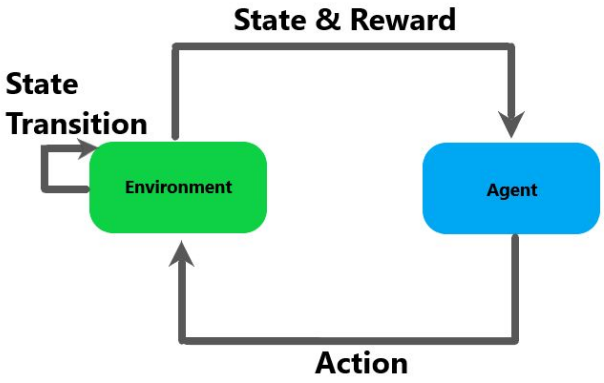


Figure 2. Basic Reinforcement Learning framework

The aim of our team was to explore three different DQN reinforcement learning networks. First, we learned and implemented Deep Q-network (DQN). Then, to improve Q value estimation method, we implemented Double Deep Q-network (DDQN), after that, we managed to improve improve network structure, so we implemented Dueling Deep Q-network (Dueling DQN). In experimental part, we trained each model for more than 10 hours and recorded and plotted average reward and average Q value of each model as experimental results. Finally, we made comparison among these results and it showed that DDQN performed best; followed by DQN while Dueling network had the worst performance.

2. Related Work

2.1 Deep Q network

The most similar prior work to our approach is neural fitted Q-learning(NFQ). Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method published in 2005 by M Riedmiller[6]. NFQ optimises the sequence of loss functions $L_i(\theta_i) = E_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2]$, where $y_i = E_{s'\sim\varepsilon}[r + \gamma max_{a'} Q(s',a';\theta_{i-1}|s,a]$, using the RPROP algorithm to update the parameters of Q-network. NFQ has also been successfully applied to simple real-world control tasks using purely visual input, by first using deep autoencoders to learn a low dimensional representation of the task, and then applying NFQ to this representation.

2.2 Q value Overestimation

The overestimate issue of Q-learning was first found in 1993 by Thrun and Schwartz. They showed that if the action values contain random errors uniformly distributed in an interval $[-\varepsilon, \varepsilon]$, then each Q target is overestimated up to $\gamma \in (\frac{m-1}{m+1})$. In addition to this, Van Hasselt also noted that noise in the environment can lead to overestimations even when using tabular representation [3]. Generally speaking, the traditional Q-learning or DQN Q value update equation, $y_t = r + \gamma max_a Q(S_{t+1};\theta_t)$, where the evaluation and selection of an action are using same weights, may

cause an over optimistic Q values. This can be solved by using Double DQN network introduced in Google Deepmind's paper [3].

2.3 Advantage Updating Algorithm

In 1994, Baird proposed a problem that inherent in the definition of Q values, especially in continuous case and tiny step gap case, just like car racing game[5]. An important consideration is the relationship between Q values for the same state, and between Q values for the same action. In a typical reinforcement learning problem with continues states and actions, it is frequently the case that performing one wrong action in a long sequence of optimal actions will have little effect on the total reinforcement. In such case, whether taking action u_1 or u_2, $Q(x,u_1)$ and $Q(x,u_2)$ will have relatively close values. The Q function contains no information about the policy. Therefore, Q-learning would be expected to learn slowly and it is incapable of learning in continuous time.

One way to solve this is advantage updating. The advantage updating algorithm is a reinforcement learning algorithm in which two types of information are stored. For each state x, the value V(x) is stored, representing the total discounted return expected when performing optimal actions. The advantage A(x,u) represents the degree to which the expected total discounted reinforcement is increased by performing action u. As is proved by Baird, this avoids the representation problem for Q learning, where the Q function differs greatly between states but differs little between actions in the same state. And that is why we try to explore dueling DQN in our project.

3. Approach

3.1 DQN

Deep Q-learning Network is the combination of Q-learning algorithm and Neural Network. In the Q-learning, the main idea is that we can use $Q(s,a) = r + \gamma max_{a'} Q(s',a')$, the Bellman Equation to iteratively approximate the Q-function. The Q-function can be implemented as a table(Q-table) with states as rows and actions as columns. However, in the game environment, Q-table cannot hold a huge states space, because if we apply the same pre-processing on the game screen - take the last four screen shots as an example, adjust their size to 84×84 and adjust their grayscale to 256-level grayscale - we get 256^84x84x4 ≈10^67970 possible game states.

This is where the deep learning comes into play. We can use neural networks to represent our Q functions, and take the state and actions as input and the corresponding Q values as output. Or we can use a unique game screen as input and output a Q value for every possible action. This approach has its own advantages: if we want to perform a Q update or select the action that corresponds to the highest Q value, we can get the Q value of all possible actions immediately by simply passing the network forward.

Our neural network architecture is listed below:

Conv1 = 7*7	Filter = 8*8	stride=3
max_pooling1	Filter = 2*2	stride=2
Conv2 = 3*3	Filter = 16*16	stride=2
max_pooling2	Filter = 2*2	stride=2
fc1	node = 256	
fc2	node=4	

Table 1. Deep Q Network

The input to our network is four 96×96 grayscale game screens. The output of this network is the Q value of each possible action (we have 4 actions in car racing game). The Q value can be any real value, which makes it a regression task that can be optimized using simple squared error losses.

$$L = \frac{1}{2}[r + max_{a'} Q(s',a') - Q(s,a)]^2$$

Also, we use Experience Reply strategy to help Q-function converge faster. The problem comes from the difference between states in every four frames. A example scene is when the car is running on a straight road, the next state won't change a lot since the car may still run in the straight road after four frames. So, our model can learn few things when we use continuous states. So, during the game, all experiences < s, a, r, s' > are stored in the Experience Replay memory. When training the network, random microbatch data from the playback memory is used instead of using the most recent transform. This breaks the similarity of subsequent training samples, otherwise it may make the network develop to a local minimum. Experience Replay also simplifies the tuning and testing of the algorithm by making the training task more similar to the usual supervised learning.

3.2 Double DQN

We implemented the Double DQN network introduced by Google Deepmind [3]. The new Q value estimation update equation introduced as follow:

DQN Q update :

$$Y^{Q}_{t} = R_t + \gamma max_{a'} Q(S_{t+1}, a';\ \theta_t)$$

Double DQN Q update:

$$Y^{Double\ Q}_{t} = R_t + \gamma Q(S_{t+1}, argmax_{a'} Q(S_{t+1}, a'; \theta_t); \theta'_t)$$

Instead of using same weights to evaluate and select an action. We introduce another Network with different weights the evaluate the action selected by the original DQN network, where we choose an action which maximize the Q value in next state. This means that, we are still using greedy policy to selection an action meanwhile using another weights θ'_t to fairly evaluate the value of this policy. We update the second set of weights symmetrically by switching the roles of θ and θ'. By using this approach, the issue of overestimation can be averted. Some examples can be shown in Figure 3 and we also confirm that this method is working from the results of our implementation.

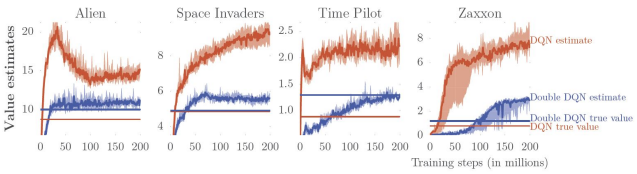


Figure 3. The estimated Q value when using DQN and Double DQN (using figure from [3])

3.3 Dueling DQN

The idea of maintaining separate value and advantage function goes back to Baird(1993). The dueling architecture represents both the value $V(s)$ and advantage $A(s,a)$ with a single model whose output combines the two to produce a state-action value $Q(s,a)$.

For an agent behaving according to a stochastic policy π, the values of state-action pair (s,a) and the state s are defined as follows:

$$Q^{\pi}(s,a) = E[R_t | s_t = s, a_t = a, \pi]$$
$$V^{\pi}(s) = E_{a \sim \pi(s)}[Q^{\pi}(s,a)]$$

The optimal Q function satisfies the Bellman equation, as is introduced in 3.1.

$$Q^{*}(s,a) = E_{s'}[r + \gamma max_{a'} Q^{*}(s',a') | s, a]$$

Here we define another important quantity, which is called the *advantage function*, relating the value and Q functions:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Note that $E_{a \sim \pi(s)}[A^{\pi}(s,a)] = 0$. Moving $V^{\pi}(s)$ to the left, we get a new Q value here,

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a)$$

Moreover, for a deterministic policy, $a^{*} = argmax_{a' \in A} Q(s,a')$, it follows that $Q(s,a^{*}) = V(s)$ and hence $A(s,a^{*}) = 0$. The value function V measures how good a particular state s is. The Q function measures the value of choosing a particular action when in this state. The advantage function A subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.

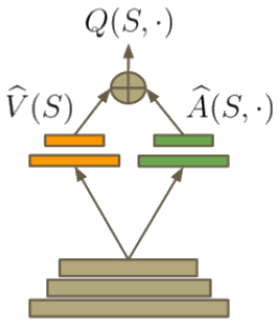


Figure 4. Dueling Network

In this project, we built structure called dueling network to implement this algorithm (Figure 4). Like the Q network shown in 3.1, the lower layers of the dueling network are convolutional as in the original DQNs (Mnih et al., 2015). However, instead of following the convolutional layers with a single sequence of fully connected layers, we instead use two streams of fc layers. The streams are constructed such that they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q function. As in (Mhin et al., 2015), the output of the network is a set of Q values, one for each actions.

Let's deeper look at the network in figure 4. The one stream of fc layers output a state scalar $V(s;\theta,\beta)$, and the other stream output an |A|-dimensional vector $A(s,a;\theta,\alpha)$. Here, θ denotes the parameters of convolutional layers. α and β are parameters of fc layers. Using the definition of *advantage function* above, we define the Q value as follows:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

However, this equation is lack of identifiability. We cannot recover V and A from this equation uniquely. For example, we add a constant to V and subtract it from A gives the same Q. To avoid it, we force the *advantage function* to have zero advantage at the chosen action. Thus, we let the last module of the network implement the forward mapping

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - max_{a' \in |A|} A(s,a';\theta,\alpha))$$

An alternative model replaces the max operator with an average:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) +$$

$$\left(A(s,a;\theta,\alpha) + \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

This loses the original semantics of V and A because they are now off-target by a constant, but it increases the stability of the optimization, because the advantage only need to change as fast as the mean, instead of any change to the optimal advantage function. Also, subtracting the mean helps with identifiability, it doesn't change the rank of A values, preserving any greedy or ε- greedy policy based on Q values.

4. Environment

Existing datasets are not needed in our project, since the three networks are all based on Q-learning, which can learn exclusively from experience replay. We put all history of state, action, reward and next state and sampled from it. It was enough to train our agent.

We used Car-Racing-v0 from gym library as our basic game environment, which is easiest top-down racing environment to learn from data. The agent is the small red car. Each state is a screenshot taken every 4 frames consisting of 96×96 pixels. There are 4 actions: Forward, Brake, Turning Left and Turning Right. Reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track. Also, when the car goes out of playfield, which is displayed as green zone below, reward is -100. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. Each episode finishes when all tiles are visited or the car is far away from the playfield.

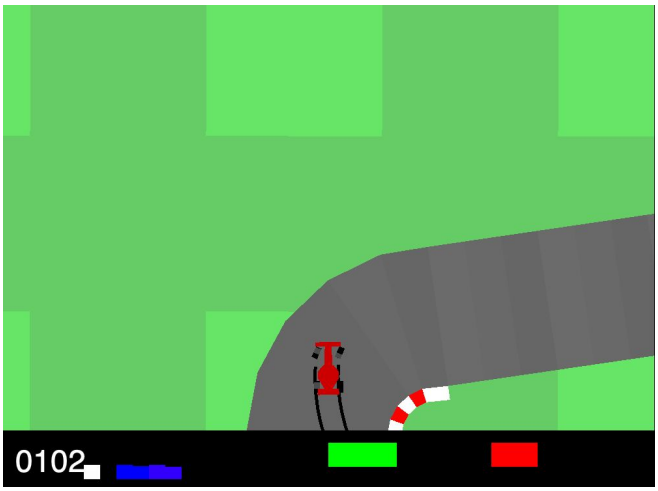


Figure 5. Reward and indicator

When the environment is started up, indicators shown at the bottom of the window and the state RGB buffer. From left to right: true speed (white), four ABS sensors (blue), steering wheel position (green), gyroscope (red).The number shown in the graph is reward score. At the same time, it also return to car's action to the terminal every 200 steps or when it goes outside of playfield or one episode finishes. The action map contains three parts. First value represents left or right. If left then -1 and if right then +1. Second value shows forward or backward. If forward then +1. And third value means braking. If braking then +0.8.

```
action ['-0.00', '+0.00', '+0.80']
step 1200 total_reward -68.44

action ['-1.00', '+0.00', '+0.00']
step 1400 total_reward -88.44

action ['-0.00', '+1.00', '+0.00']
step 1600 total_reward -108.44
```

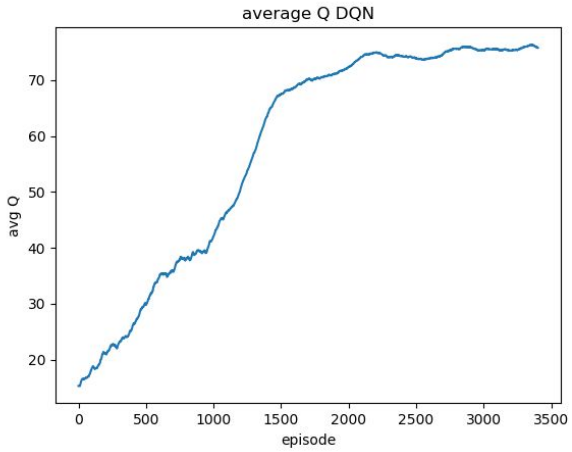
Figure 6. Action map

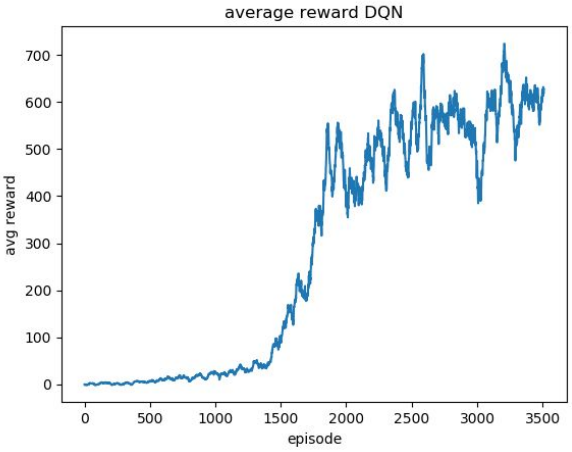
5. Results

For all these three reinforcement learning algorithms, we trained models over 12 hours on cpu and recorded the average Q value, average reward and loss value for each episode. We set the total training episode at 4000 for each 3 model. 5.1-5.3 give figures of the three respectively. 5.4 tests the performances based on 20 rounds score, making comparison between them and giving some explanations.

5.1 DQN

Figure 7 here is the training result of DQN. The model goes into a stable state after 1500 episodes. The average Q value remains at 75 and the average reward is about 550. The loss reaches highest in first 1000 episode and drops after then.





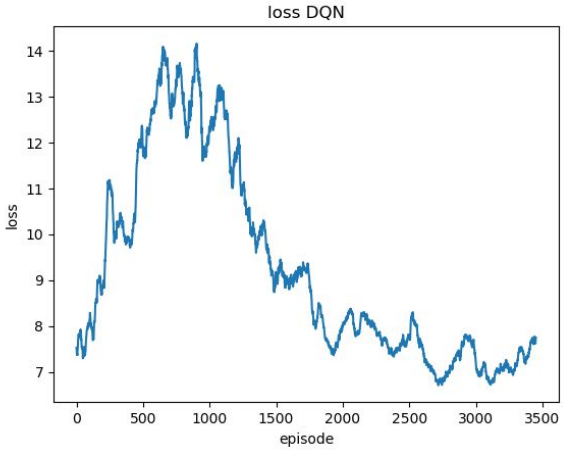
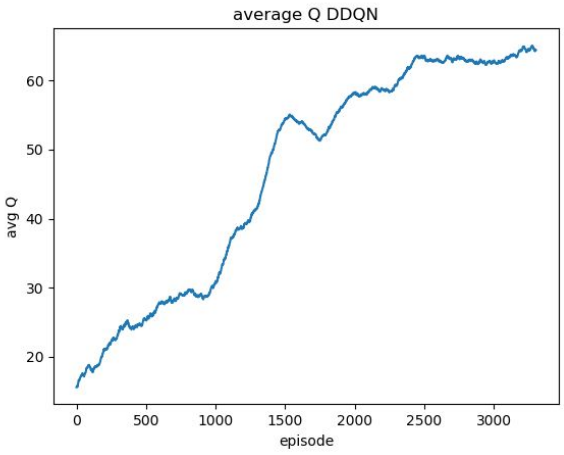
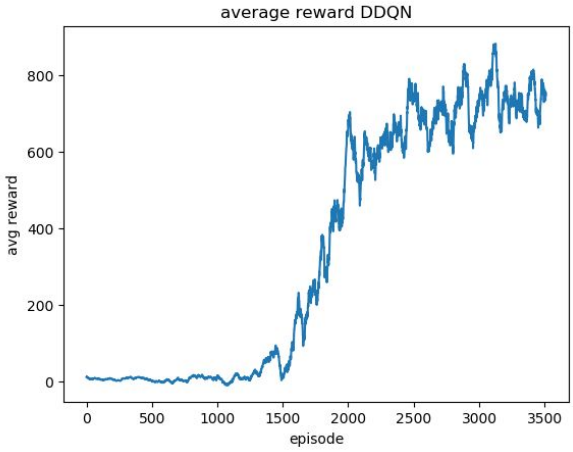


Figure 7. DQN result

5.2 DDQN

As shown in Figure 8, the average Q value and reward go higher as we train episodes goes. And the loss goes lower as training step goes.





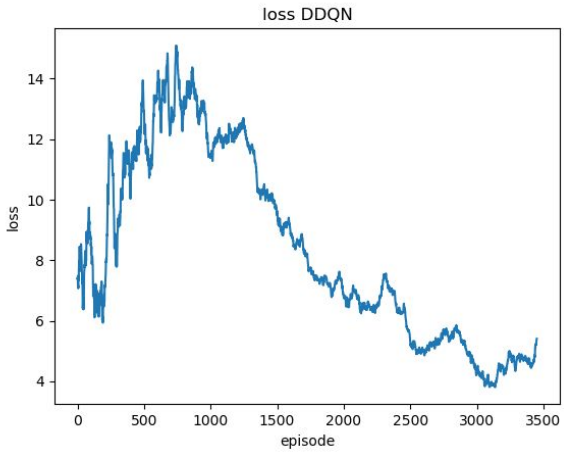
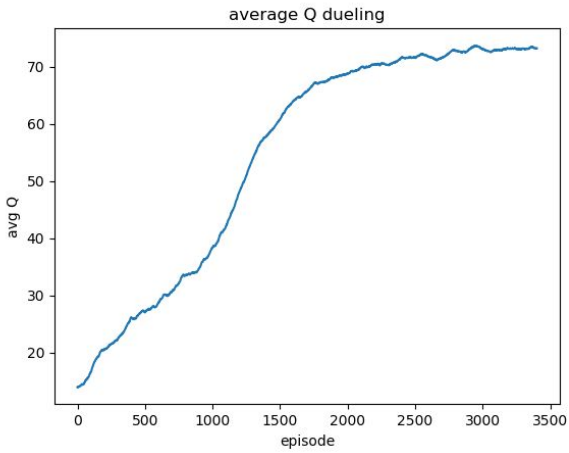
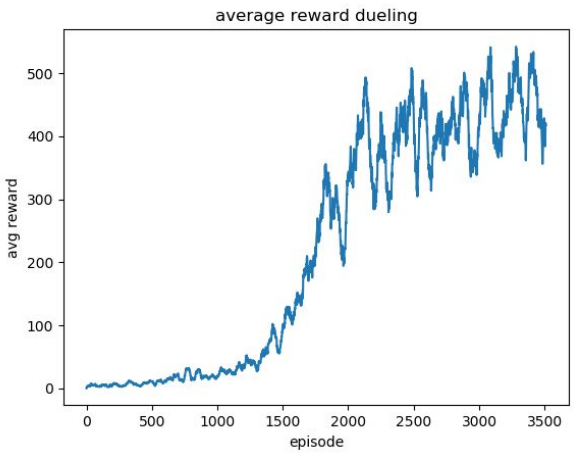


Figure 8. DDQN result

5.3 Dueling DQN

Figure 9 shows the training result of dueling network. As our expectation, the average Q value goes up along with episodes, while the average score of each round increases in jagged pattern. The loss climbs to the top at around 1800 episodes and then decreases. The dueling network does not perform as well as it should be. We will give some explanation in section 5.4.





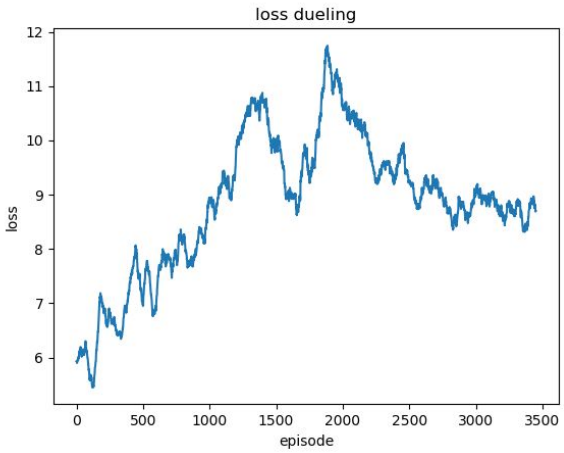


Figure 9. Dueling DQN result

5.4 Comparison

We recorded the reward scores of three DQN networks. It shows the average score of 20 episodes. DDQN achieved the highest score; DQN's score secondly ranks and Dueling DQN performs worst. Apart from that, they all worked much better than human did.

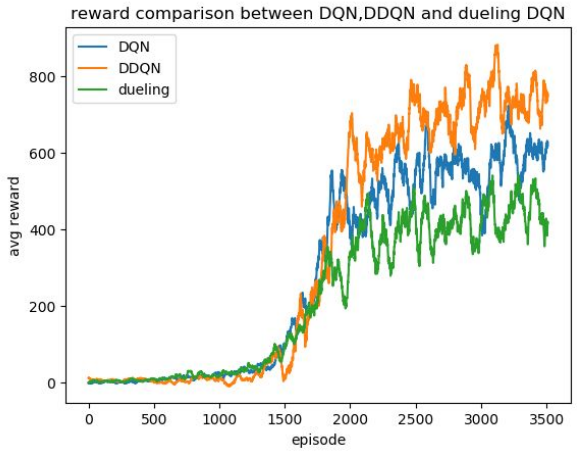
	DQN	DDQN	Dueling DQN	Human
Score	755	784.95	737.35	216.35

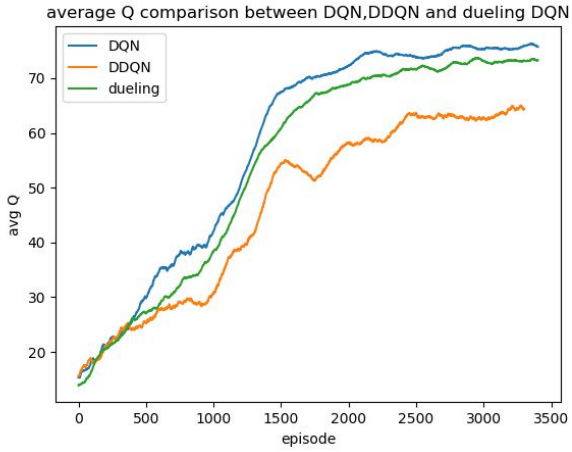
Table 2. Comparison of test score

We also plotted average reward, average Q value and loss of three networks in the graphs. As shown below, In the reward graph, at the beginning, as the number of episode becomes larger, all three lines are going up. After convergence, they tends to fluctuate over the average value. The average of DDQN reward is highest while that of Dueling DQN is lowest.

The average Q value plots displays different trend where DDQN's Q value is lower than DQN's Q value. DDQN' Q value ranks in the middle position.

As for loss plots, the trend is rising before decreasing.





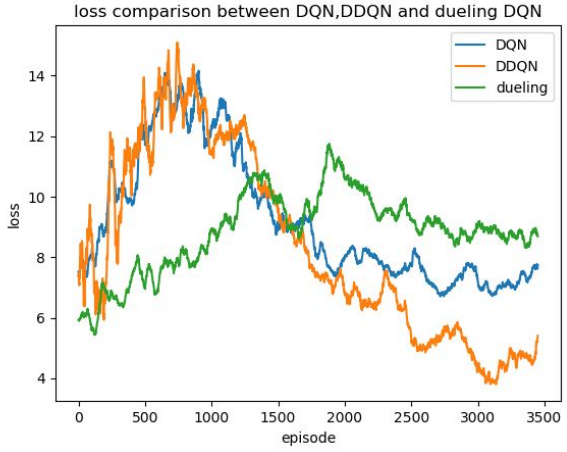


Figure 10. Comparison between three models

There are two reasons behind the bad performance of Dueling DQN. One is that the view of the car is narrow,

so every action value estimation is significant in every state. Another is that the number of nodes in fully-connected layer in dueling network is smaller. Thus, features extracted is not sufficient.

6. Detailed Roles

Task	File names	Who
● Wrote interface code connecting game environment and network; ● Wrote Task and Environment part of final report	data/plot.py DQN/dqn/experience_history.py data/score.xlsx	Jiahao Zhang
● Wrote DDQN code; ● Wrote DDQN part of final report	Double DQN/car_runner_main.py Double DQN/agent.py	Minghe Ren
● Wrote DQN code; ● Wrote DQN part of final report	DQN/car_runner_main.py DQN/agent.py	Weixuan Jiang
● Wrote Dueling DQN code; ● Write Dueling DQn part of final report	Dueling DQN/car_runner_main.py Dueling DQN/agent.py	Zhonghao Guo

7. Code repository

https://github.com/guozhonghao1994/Deep_Reinforcement_Learning_on_Car_Racing_Game

References

1) Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
2) Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540), p.529.
3) Van Hasselt, H., Guez, A. and Silver, D., 2016, February. Deep Reinforcement Learning with Double Q-Learning. In *AAAI* (Vol. 2, p. 5).
4) Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M. and De Freitas, N., 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.
5) Baird, L.C., 1994, June. Reinforcement learning in continuous time: Advantage updating. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on* (Vol. 4, pp. 2448-2453). IEEE.
6) 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.