COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Learning to Play Pong using DQN and Policy Gradient Learning

Group Members

Shuyang Wang	sw3232
Lujia Wang	lw2772
Xiaorong Yuan	xy2347
Peilu Zhang	pz2233

Table of Contents

Abstract

1. Project Goals

Our project used the OpenAI Gym Pong environment and tried to maximize our scores in the Atari 2600 game Pong. In our project, we receive an image frame (a 210x160x3 byte array) as input to the network, and implement a deep reinforcement learner to decide to keep the paddle still or move it up or down. After every single choice the game simulator executes the action and gives us a feedback under the three scenarios: a +1 reward if the ball went past the opponent, a -1 reward if we missed the ball, or 0 otherwise. Our goal is to move the paddle so that we get as many rewards as we could.

In order to accomplish above goals, we applied two common methods to play the Pong game. The first is Deep Q Network (DQN), which we have learned in class, and the second is Policy Gradient (PG), which is also an efficient Deep Reinforcement Learning tool owning some advantages over the DQN.

2. Project Results

The two models both attained positive rewards after training for several days. Under the *Pong-v0* environment, the DQN model achieved an average score of 0 after about 3,000 episodes of training, which cost 2 days on *Google Colab*. And the trained Policy Gradient agent received an average reward of +6 after 2 days of training on a Macbook Pro until 20,000 episodes. Under the *PongDeterministic-v4* environment, the result seems to be more optimistic. The DQN agent reached +20 score after 1,600 episodes of training in 9.7 hours, while the Policy Gradient one achieved the same average score in 3,100 episodes during 3.5 hours of training. The basic project goal of winning our opponent in the game was accomplished by our models. But the paths and efficiency the two models reaching this goal are different. Details of training and more discussions are illustrated in the following parts of this report.

Models

1. Preprocess of Observations

Before inputting the images in our models, we convert the observation, a 210*160*3 colored image into an 80 * 80 or 84 * 84 black and white image by several steps shown in Fig 1.1. We use different input dimension in order to be strictly comply the different paper's instructions. Generally speaking, DQN papers use 84 * 84 as input and PG use 80 * 80 as input. The original image was cropped to get only the playground with paddles and the ball. Next the image was resampled by a factor of 2 and turned to

grayscale. At last, it was turned to black and white image with less dimensions than the original one.

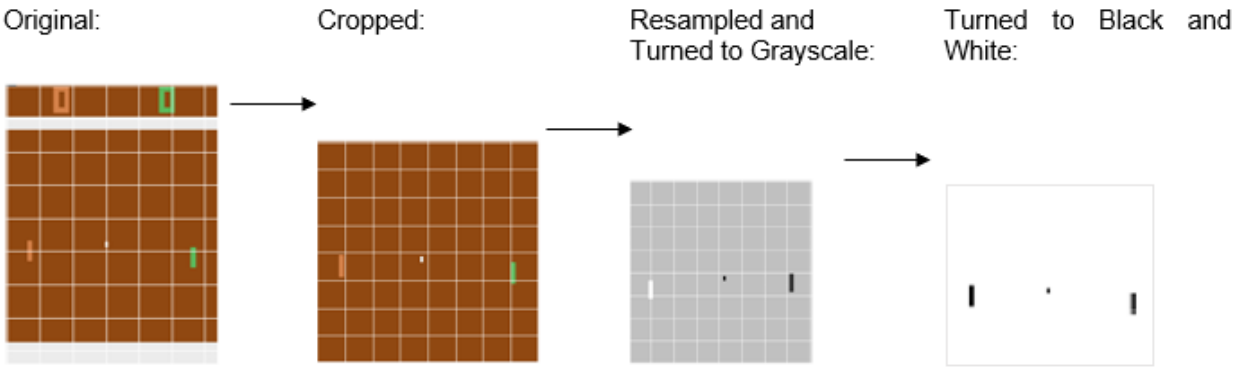

Fig 1.1 Preprocess of Image from the frame of the Game

2. DQN

2.1 Introduction

We learnt DQN from class and understood that it worked well with the RAM version of the *CartPole* game. So, we wondered if we can solve the *Pong* from pixels with just simple modification (we can't btw). The RAM *CartPole* game might be too easy for a DQN model, since it only came with small number of inputs. Meanwhile, a game from pixels will give us huge number of inputs, which might be a more suitable problem for DQN.

Thus, the sensible first step would be to change the original fully connected model (Cunningham, 2018) to a more complex multiple CNN layers model. We used preprocessing steps described above, and we stacked four frames together as an input. Apart from the classic DQN model from the DeepMind paper (Mnih, 2013) , we also tried advanced model including different Dueling DQNs and Double DQN, we trained and tested those models on the random frame skipping game *Pong-v0* and a deterministic frame skipping game *PongDeterministic-v4,* and last but not least, we compared DQN with Policy Gradient learning.

We used *Colab* with GPU for all the DQN results below. We define 'solved the game' as getting total rewards of more than 19 almost every game.

2.2 Implementations and Results

2.2.1 First step: Simple modification to class code

Leaving all the rest the same, we just changed the model to a standard DQN model that is same as described in the DeepMind paper (Mnih, 2013), the structure of Neural Network is shown below in Fig 2.2.1.

> *The input to the neural network consists is an 84 × 84 × 4 image produced by φ. The first hidden layer convolves 16 8 × 8 filters with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 32 4 × 4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action.*

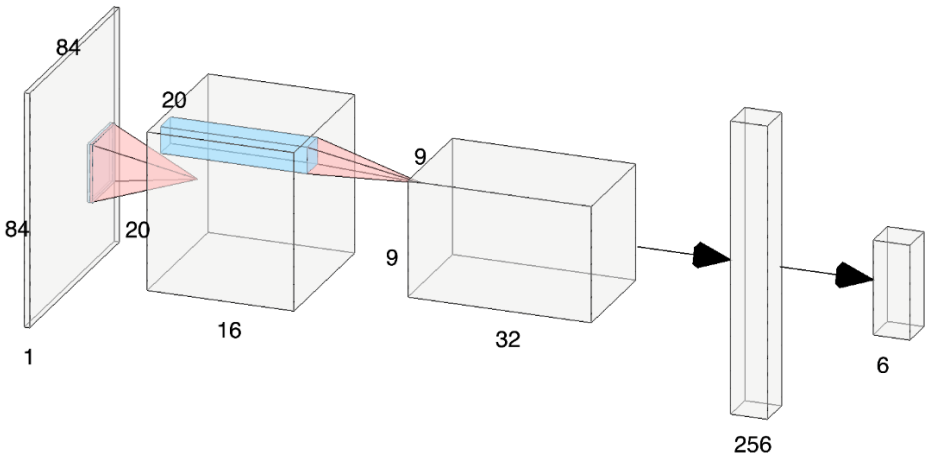

Fig 2.2.1 DQN (Mnih, 2013)

However, this model did not work well with the *Pong-v0* environment. As we can see from the training rewards in Fig 2.2.2, the DQN seems to get stuck at reward around -13.

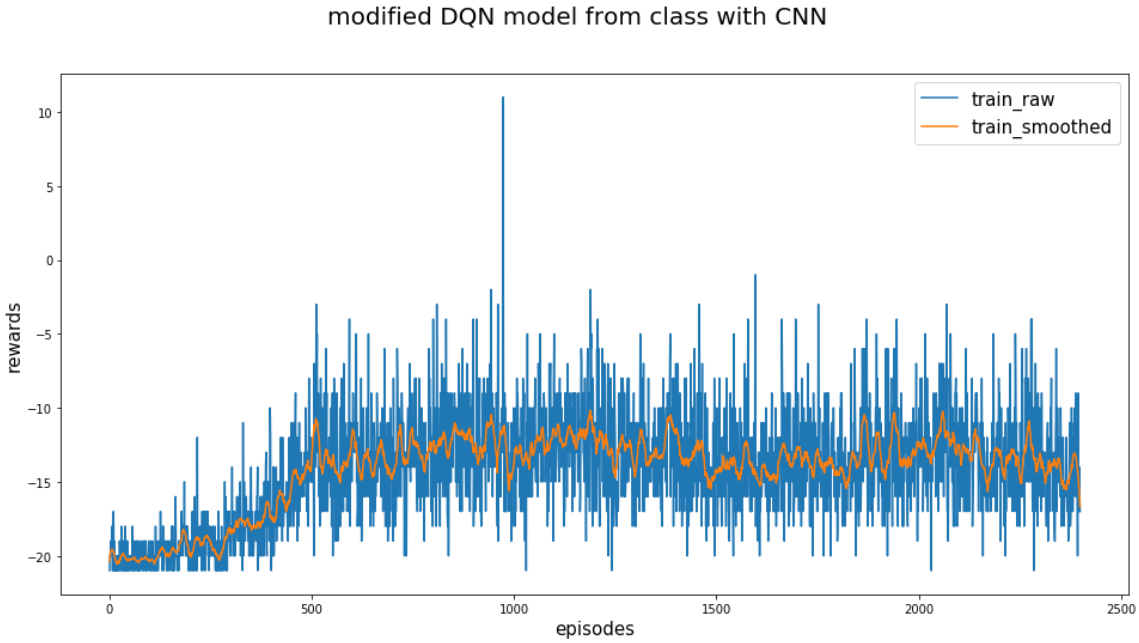

Fig 2.2.2 The Training Rewards of (Mnih, 2013) *-Pong v0*

There are a few possible reasons causing the model to underperform, but basically the reason is that we failed to do exactly what that paper asked for. First, we should do frame skipping with k = 4, however, the game *Pong-v0* use random frame skipping by a number sampled from {2, 3, 4}, thus we will use *PongDeterministic-v4* later for the required deterministic frame skipping (OpenAI, 2018). Second, if we chose to store Sarsa data with [s,a,r,s'] with processed frame like the lecture code, we can only store about 50,000 sets of data, there will not be enough space to store 1 million frames with the *Colab* memory limit, thus the only way to do this is to store frame by frame and make sure they are *int* not *float*. Finally, this model requires extensive computational power, since all we got is *Colab*, we need something faster than standard model, thus we use Dueling and Double DQN.

2.2.2 Second Step: Dueling DQN

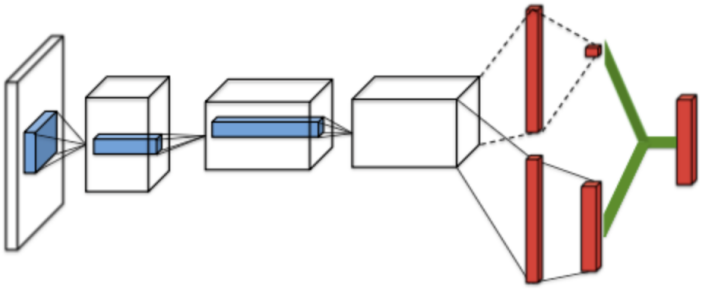

Fig 2.2.3 Neural Network Structure of Dueling DQN

The idea of Dueling DQN, described in this paper (Wang, 2015), is to separate last fully connected layer into two layers with same length. One of the layers will calculate state value $\mathrm{V}(s)$ while the other one will calculate action value $\mathrm{A}(s,a)$ and then we combine them to calculate Q value as following equation. Its structure is shown in Fig 2.2.3.

$$Q(s,\ a) = V(s) + \left(A(s,\ a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,\ a')\right)$$

I don't think I can explain the intuition of this model better than the author, so here is quote from the original paper.

> *Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. In the experiments, we demonstrate that the dueling architecture can more quickly identify the correct action during policy evaluation as redundant or similar actions are added to the learning problem.*

Also notice that the Neural Network used here is different with that in Part 1.2.1. We plotted the structure in *Fig 2.2.4.*

> *The first convolutional layer has 32 8x8 filters with stride 4, the second 64 4x4 filters with stride 2, and the third and final convolutional layer consists 64 3x3 filters with stride 1.*

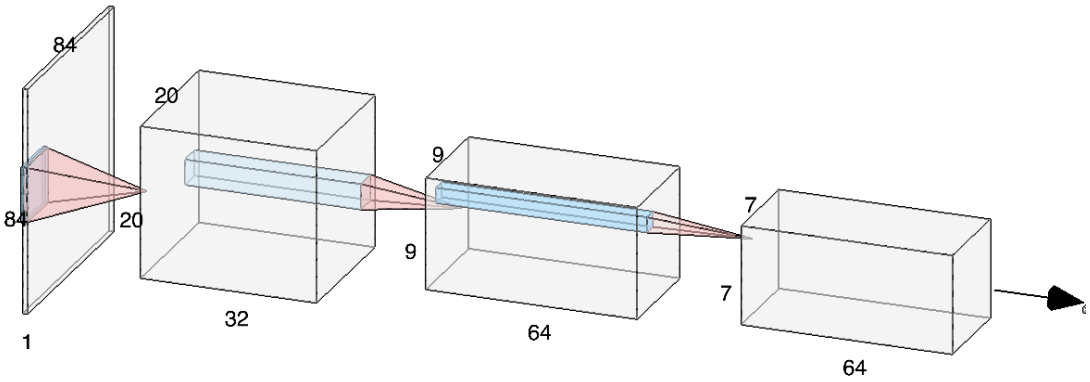

Fig 2.2.4 DQN (Wang, 2015) CNN part

With the following results plot (Fig 2.2.5), we could consider that this model 'solved' the game *PongDeterministic-v4* after about 1,500 episodes. Notice that we updated the Neural Network every four Atari steps, also notice that we tested every 50 episodes.

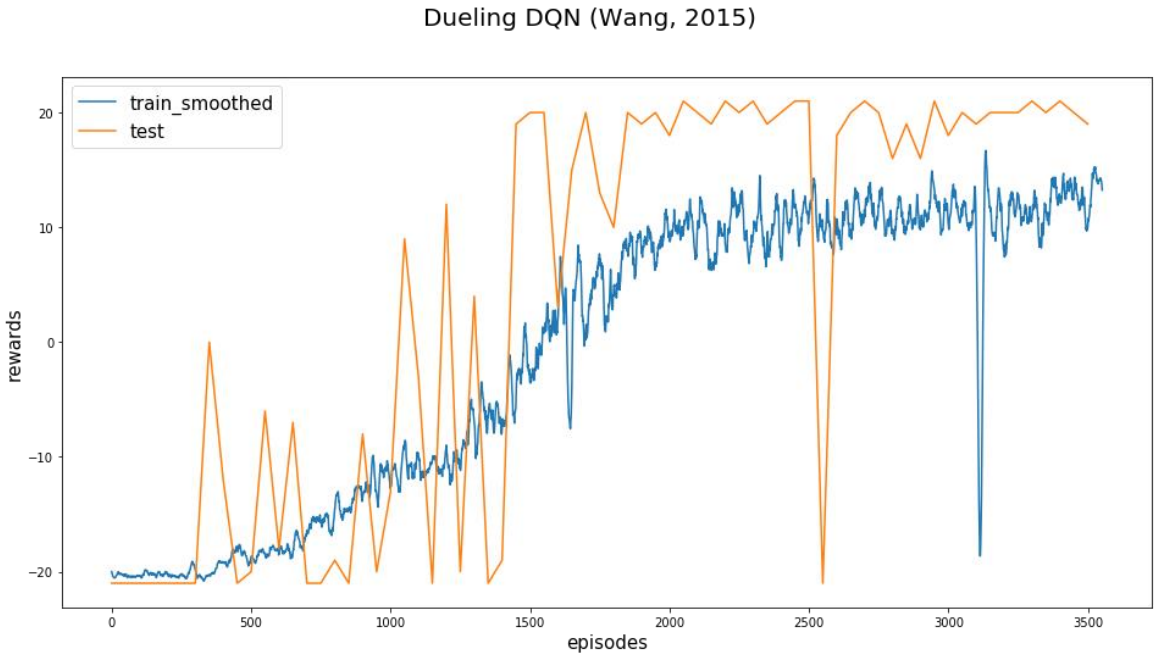

Fig 2.2.5 Train and Test Rewards of Dueling DQN

We used two fully separate connected layer of 512 neurons each here. One of them outputted one value and the other one outputted number of actions. We used initializer mentioned in this paper (He, 2015).

2.2.3 Another technique: Double DQN

Another technique that is rather popular in the world of DQN is Double DQN (Hasselt, 2016). The Q-value calculated during updating is from a Neural Network, which means it is an estimation, which means the error can add up during multiple training steps, eventually results a very biased Q.

Thus, we use two Neural Networks instead, we use main network to find best action, then we ask target network for Q-value regarding this action, this method can eliminate error efficiently. And after certain number of steps (e.g. 10000 frames), we copy the main network to target network to update it.

Here is the difference in Bellman equation of standard method and double method (fg91, 2018):

$$Q_{\text{target}}(s, a) = r + \gamma \max Q(s', a'; \theta_{\text{target}}) \quad \text{Normal DQN} \quad (1)$$

$$\text{to} \quad Q_{\text{target}}(s, a) = r + \gamma Q(s', a' = \operatorname{argmax} Q(s', a'; \theta_{\text{main}}); \theta_{\text{target}}) \quad \text{Double DQN} \quad (2)$$

We can see that this graph (Fig 2.2.6) is quite similar to the previous one, we didn't use dueling structure here, the last layer is a fully connected of 1024 neurons, which then output Q-value in dimension of action space.

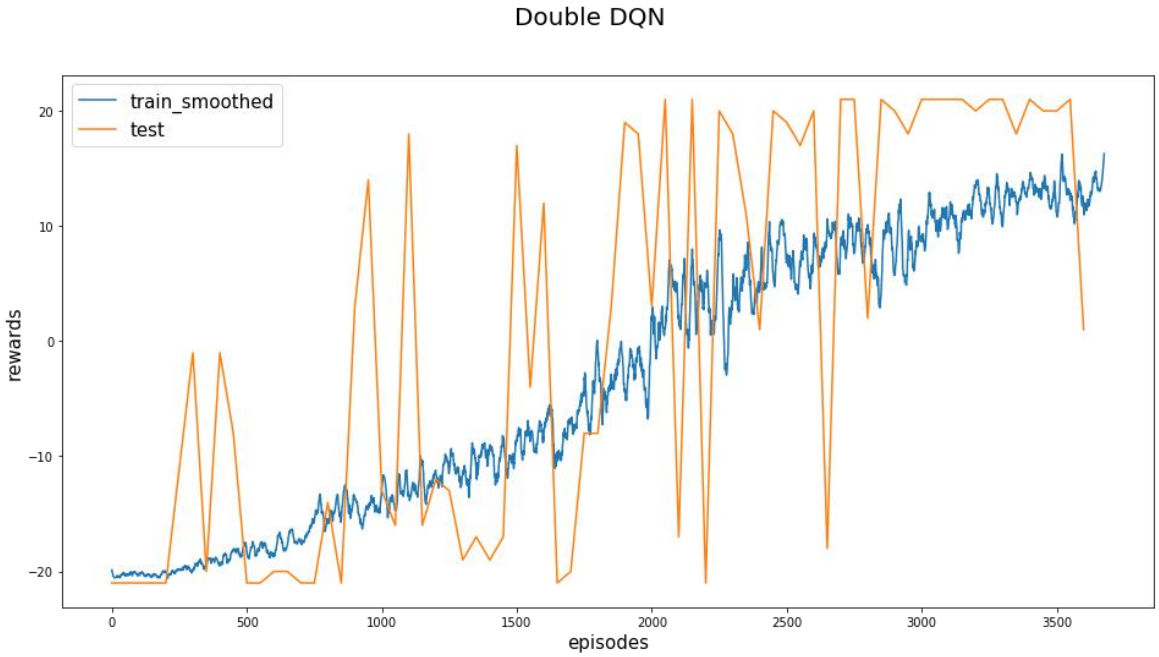

Fig 2.2.6 Train and Test Rewards of Double DQN

2.2.4 Combining together: D&D (Double & Dueling) (Wang, 2015)

Now we combined Double and Dueling network together. Here we have pseudo-code form this paper(Mnih, 2015), which concluded very well what we should do in the code.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

The results seemed to be similar with previous ones. However, the variance of test rewards is smaller than previous, meaning the trained agent found a better strategy to win this game with higher expected rewards.

DD DQN(Wang, 2015)

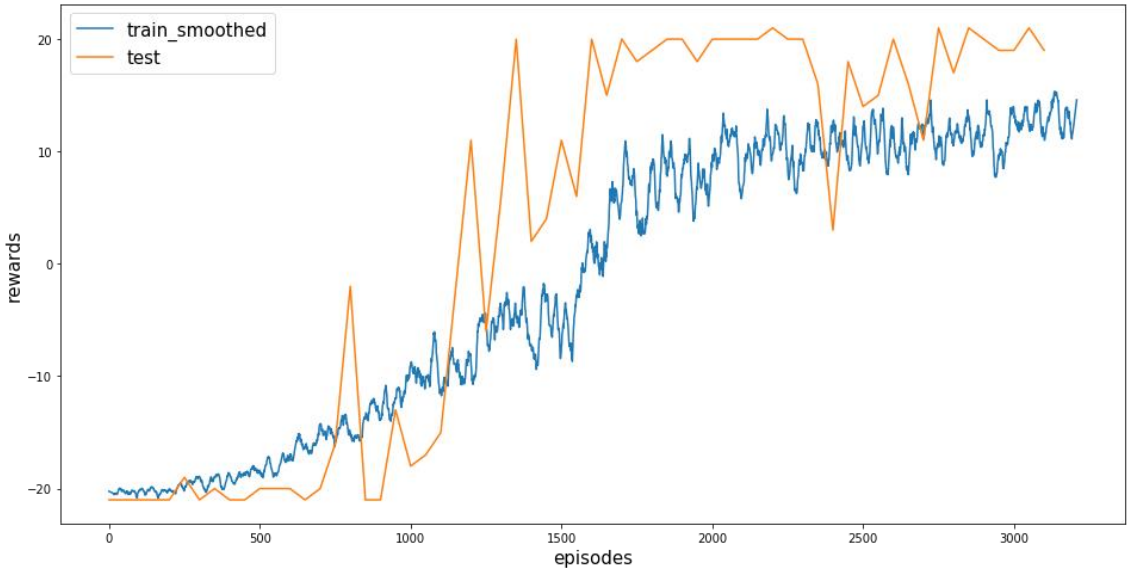

Fig 2.2.7 Train and Test Rewards of D&D DQN (Wang, 2015)

2.2.5 Yet another model: D&D (awjuliani, 2017)

It looks like those methods above solved the game, however, they are still slow, with *Colab* with GPU environment, it took about 30 hours train 3500 episodes. And *Colab* only allow a continuous session of maximum 12 hours. We will use another Neural Network structure here(awjuliani, 2017), which turned out work unbelievably well with the game Pong.

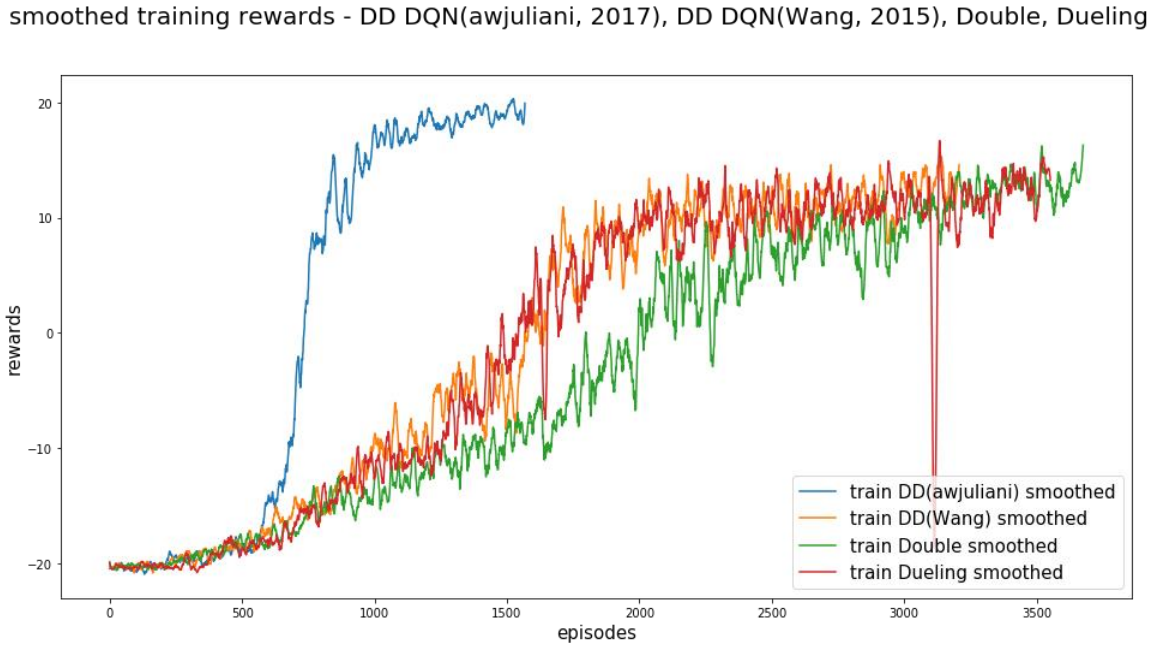

Fig 2.2.8 Comparisons of Four Models' Training Rewards

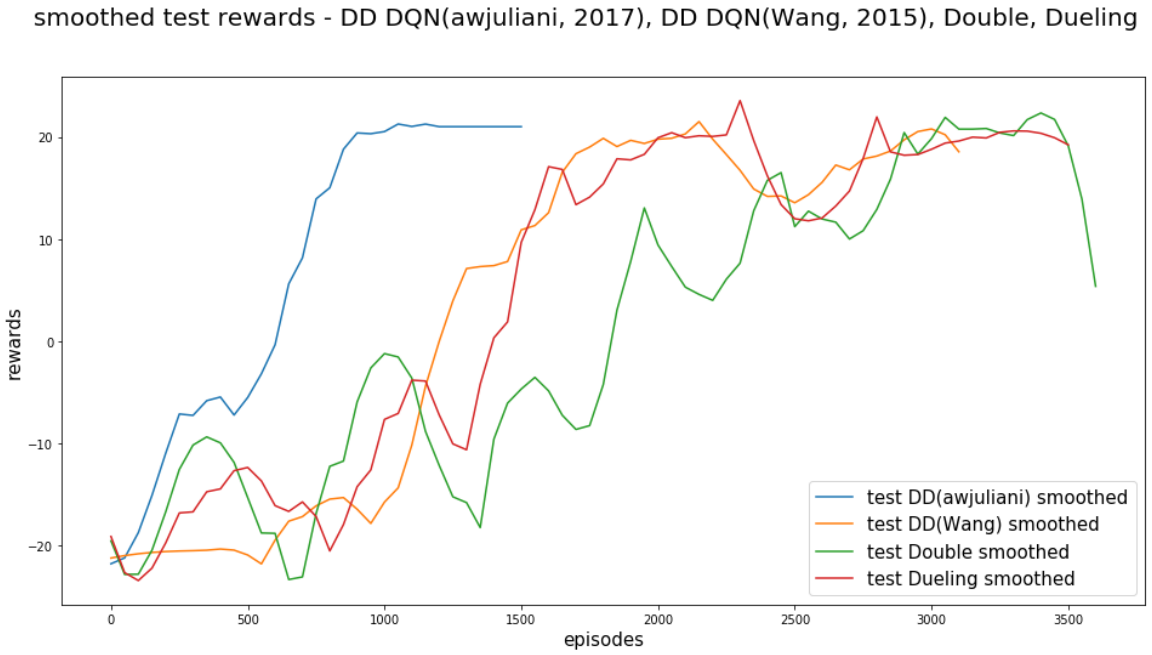

Fig 2.2.9 Comparisons of Four Models' Testing Rewards

Noticing in our previous model, we used two fully connected layers take *conv3* as input and output action value and state value(Wang, 2015). Now, the only difference in this model(awjuliani, 2017) is that it uses a fourth convolutional layer *conv4* with full filter which output 1024 numbers separated into two parts instead of two fully-connected. Next, it uses first half 512 outputs to calculate state value and later half 512 outputs to calculate action value.

It turned out we can solve this problem within one *Colab* session (it feels great to finish our task without being kicked out by *Google Colab* at some point). This model converged at least two times faster than the other model, and the training procedure is smoother.

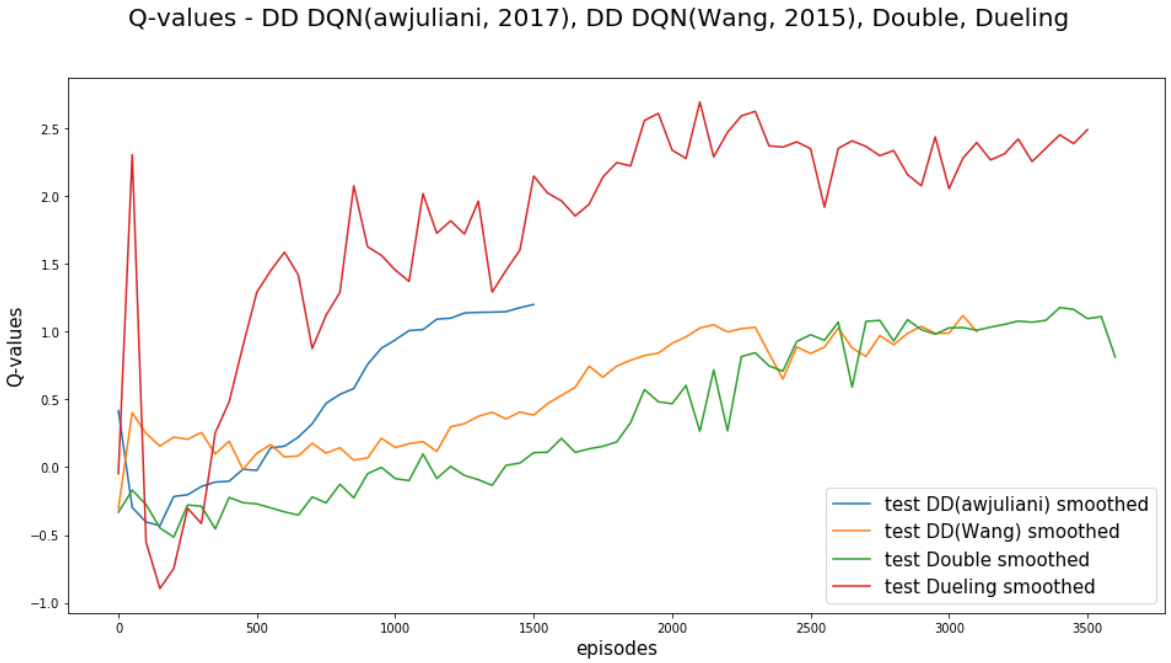

Fig 2.2.10 Comparisons of Four Models' Testing Q-values

We can get some more insights with the max Q-value plot (Fig 2.2.10). Firstly, three models with Double DQN has similar final Q-values, while the red line without Double DQN oscillates a lot, and producing higher Q-values might be caused by bias. We can see that (awjuliani, 2017) model has the smoothest line among four models, which also explains why it cost least time to run.

2.3 A Glimpse

2.3.1 Parameters

- Frame skipping

The DeepMind paper uses deterministic frame skipping for reasons. theoretically, it is *k* times faster to train a *k* frames skipping game than a no frame skipping game.

Compared with random frames skipping game (*Pong-v0*), we found that it is much faster and easier to train a deterministic frames-skipping game. As the following graph shows, if we use the same model (awjuliani, 2017), *Pong-v0* will take much longer to train.

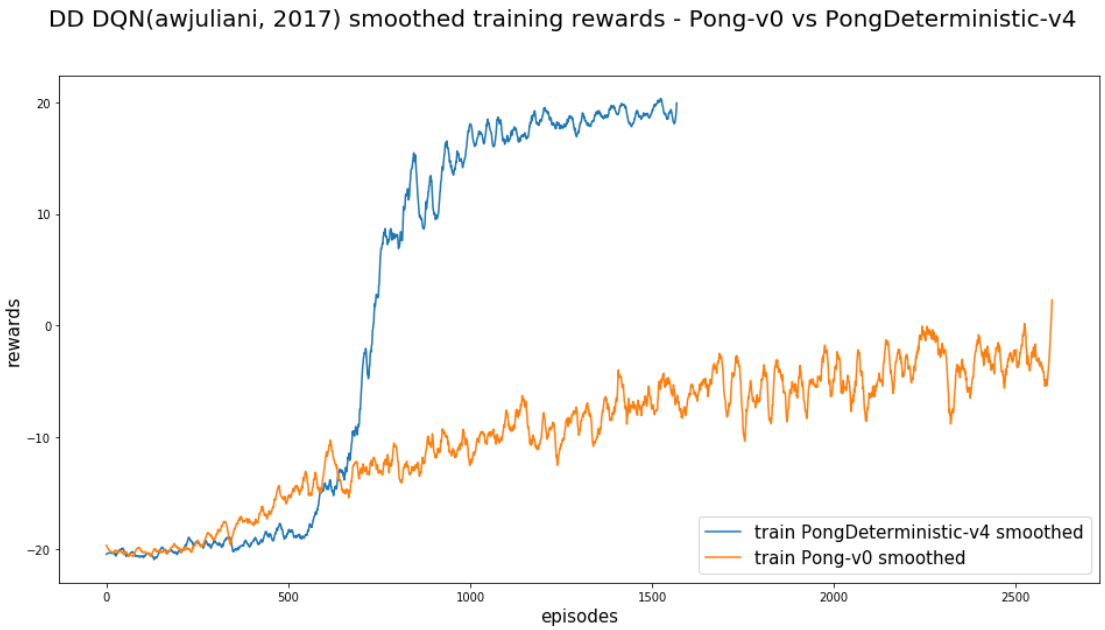

Fig 2.3.1 Comparisons of Training Rewards in Pong-v0 and PongDeterministic-v4

- Terminal signal

Pong returns match terminal signal to be True only after one player reaches +21 points. However, if every game is independent, it should be more reasonable for the terminal signal to true after every point is earned. We tried to compare between two different ways of recording terminal flags. It turned out terminal per game is better than terminal per point. Indeed, it even ended up that the location of the ball of previous point has correlation with the kick-off of next game, and thus this result is reasonable.

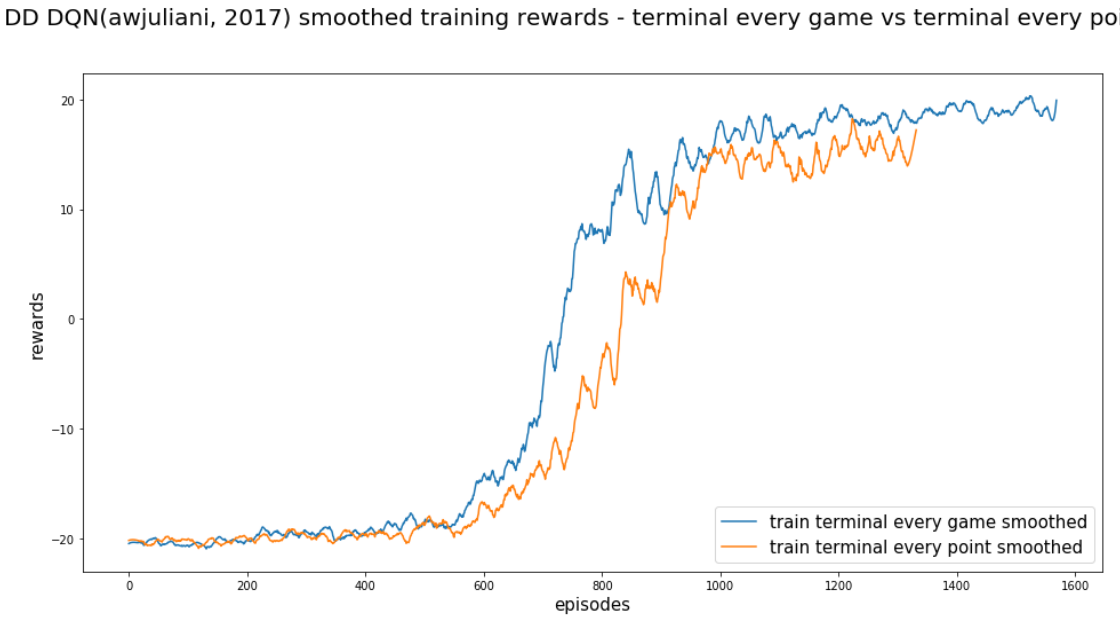

Fig 2.3.2 Comparisons of Training Rewards on Different Terminal Signals

- Action space

Most models we found online do not consider what the action space really means, but just plug the whole action space into the model (that's what we initially did). For the game Pong, there are 6 actions available. However, three of them are redundant. If we use the minimal action space, we could see that the training took less time to converge. For this game the difference is small, but for a more complex game it might save us a decent amount of time.

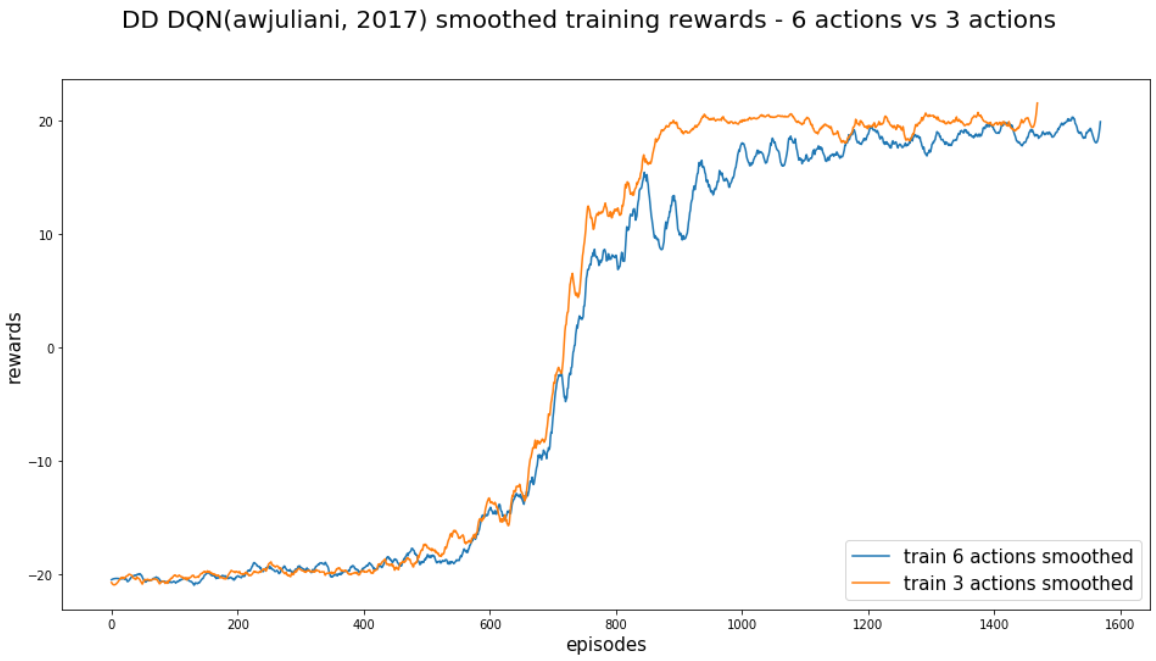

Fig 2.3.3 Comparisons of Training Rewards on Different Action Spaces

- Different epsilon schedule

People use all kinds of different epsilon schedule as this is one important factor which could affect training time dramatically. We tried both DeepMind (Mnih, 2013) and OpenAI Baseline(OpenAI, 2017) schedule. DeepMind's epsilon stops at 0.1 after 1 million frames, while OpenAI's epsilon continues to decrease to 0.01 till the maximum frame number the model can see. It seems like keeping epsilon at 0.1 wins. So, we should not decrease exploration further more after certain amount of low epsilon.

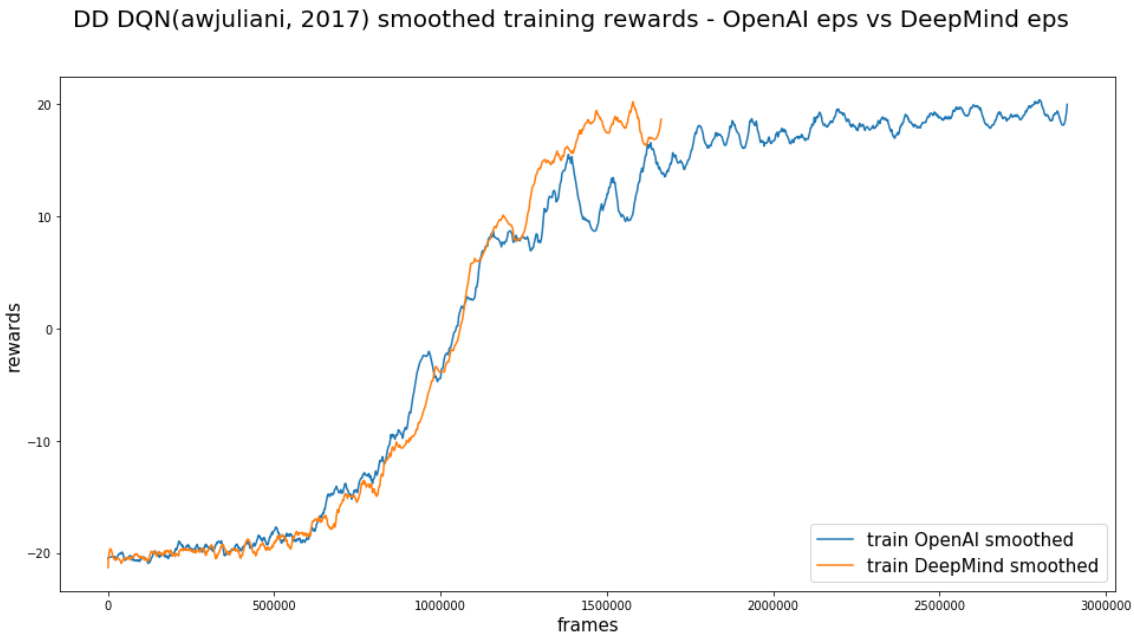

Fig 2.3.3 Comparisons of Training Rewards on Different Epsilon Schedule

2.3.2 Test on Pong without skipping frame

Since *PongDeterministic-v4* is a deterministic frame skipping version of standard Pong game, we cannot use trained model on *Pong-v0* which used random frame skipping. We can, however, use trained model to play the game *PongNoFrameskip-v4.* All we need to do is to read a frame every four frames and keep the action same between them. you can find GIFs of performance with the game *PongNoFrameskip-v4* in our README.md. It is about the same as *PongDeterministic-v4,* as expected.

In addition, it is worth noticing that all the converged models played the game in a very similar way, you can see that in the readme document.

3. Policy Gradient

3.1 Introduction

DQN can be inefficient if the state space is large, which leads to insufficient exploration of the whole state space. As a result, gradient-free approaches represented by DQN has a poor generalization ability. It motivated us to find a gradient-based approach such as policy gradient. This method establishes a policy network and approximate the policy by back-propagating errors to adjust the weights of the policy network. While DQN updates its table of value function entry by entry, the Policy Gradient updates its policy according to the gradient of expected reward with respect to the policy parameters. And it is an end-to-end method without constructing features. Attracted by those properties of Policy Gradient, we explored further in this method and decided to experiment with two deep ANN architectures, which are Feed Forward Neural Network with 1 and 2 hidden layers respectively.

3.2 Related Work

We are inspired by the article of (Karpathy, 2016), who built a 2-layer fully connected policy network with 200 hidden layer units using RMSProp optimizer in Numpy. And his action space only includes two actions {Up, Down}, and updates the policy net every 10 episodes. His agent performed only slightly better than the opponent after running 200,000 episodes for three nights on a Mac. Although the model seems relatively interpretable and the result is acceptable, we still searched for a reduction in training time probably by using Tensorflow and a better score.

We got a hint from (Grimm, 2018), who implemented the Policy Gradient training in the CarPole-v0 and InvertedPendulum-v0 environment and transferred this method to the *Pong-v0* environment. He used Tensorflow to construct a fully connected hidden layer with 200 units and expanded the action space to include No Operation (NOOP) because of the built-in jitters of the environment. The score rose to positive 8, but the training process still took more than two days on an AWS GPU.

In order to further accelerate the training process, we learned two possible solutions. The first is a recent study of Asynchronous Advantage Actor-Critic (A3C) introduced by (Mnih, 2016). This approach speeds up the training process and improves results in three ways as suggested by its name. The term asynchronous means A3C employs different agents to explore the environment independently. Advantage refers to the use of a biased reward and actor-critic refers to the behavior of exploitation based on shared knowledge via the critical component. Besides, another method is skipping fixed number of frames. The original *Pong-v0* environment skipped frames randomly from {2, 3, 4}, which brings randomness and increases the difficulty of the game. We found a similar environment called *PongDeterministic-v4* with a fixed skipping number of 4. The latter environment may effectively cut training time to several hours.

After understanding previous related work, we decided to utilize fully connected neural network with 200 units in the hidden layer, using activation function ReLU and optimizer Adam. We built our model based on Tensorflow and trained the model both on Google Cloud and our laptops. Both environments of *Pong-v0* and *PongDeterministic-v4* are used for comparison.

3.3 Implementation

3.3.1 Models

The objective of Policy Gradients is to maximize the total future expected rewards $E[R_t]$, which can be stated as the following expression,

$$arg\ max_\theta E[\sum_{t=0}^{T-1} \gamma^t r_t]$$

where θ is the parameter of the policy, π is the policy and T is the length of the episodes. The goal of the Policy Gradient is to learn the value of θ. Taking the gradient of $E[R_t]$ respected to the parameters, we get

$$\nabla_\theta E[R_t] = E[R_t \nabla_\theta\ log\ P(a|\theta)]$$

The algorithm is known as REINFORCE (Sutton, 2000).

We tested two types of network architectures. First one is a three-layer neural network, consisting of an input layer of size 80*80, a hidden fully connected layer of size 200 and a SoftMax output of size 3. The input layer is formed by flattening the input frame into a vector of dimension 1600, and fully connecting it to the hidden layer. The three output nodes corresponding to an action space of [Up, Down, NOOP]. The second one is very much like the first one, adding one more hidden layer of size 200.

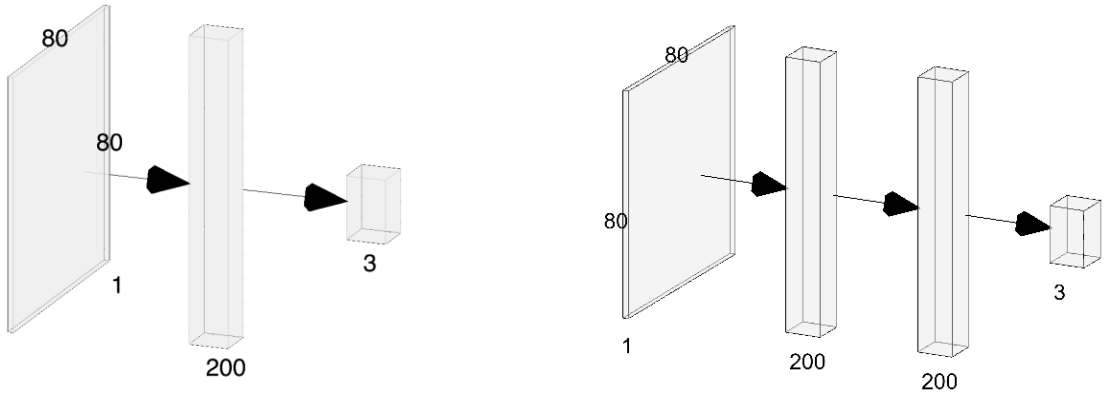

Fig 3.3.1 Architectures of Policy Gradient Network of Model-1 and Model-2

	Model-1	Model-2	Model-3
Input layer	80 * 80	80 * 80	80 * 80
Hidden layer	200	200	200
Output layer	3	3	3
Policy Network	6400*200*3	6400*200*200*3	6400*200*3
Learning Rate	0.001	0.001	0.001
Batch Size	1	1	10
Smoothing Factor	0.99	N/A	0.99
Discount Factor	0.99	0.99	0.99
Environment	*Pong-v0*	*Pong-v0*	*PongDeterministic-v4*

Table 3.3.1 Summary of Policy Gradient Network Parameters

The preprocessing step is the same with DQN model. Then the preprocessed image is transformed into a vector of length 6,400. Note that the input frame here is not the current frame, but the difference between current frame and the previous frame. In the training part, the advantages are calculated from the discounted reward of current episode with a discount factor $\gamma = 0.99$. The weights of the network are updated after every batch. Since the variance of episode reward over time is relatively large, it is difficult to see the trend from the noises, we applied a smoothing factor on the episode reward, which means the reported reward of current episode $r_{t,smoothed}$ is given by,

$$r_{t,smoothed} = 0.01 \times r_t + 0.99 \times r_{t-1,smoothed}$$

3.3.2 Results

The computation of the Model-1 was carried out on a MacBook Pro (early 2015). Because of the simplicity of the network architecture, it can be trained without a GPU. The model took more than 2 days to train until 22,000 episodes and received an average of reward of +6. Results shown in Fig 3.3.2.

Model-2 was trained on a policy gradient network with two hidden layers, each has 200 units. Interestingly, the model with one more hidden layer has a worse performance, both in running time and the rewards score. The agent learned relatively slow, and only received around an average reward of -15 until 18,000 episodes. We suspect that the extra hidden layer has negatively affected the convergence rate and increased the computation cost of taking the gradients of the discounted reward function as the neural network becomes more complex.

Model-3 was trained in *PongDeterministic-v4* which uses the fixed number of frame skips. The number of frame skips in *Pong-v0* is sampled from {2, 3, 4}. In other words,

PongDeterministic-v4 has less computational cost and makes it easier for the agent to learn the pattern of the ball movements. From the Fig 3.3.4, we can see that the agent reached the highest reward +21 until 4,000 episodes. Compared with previous two model, Model-3 has a relatively smaller variance of episode rewards and a better and faster convergence at the highest reward one can receive in an episode.

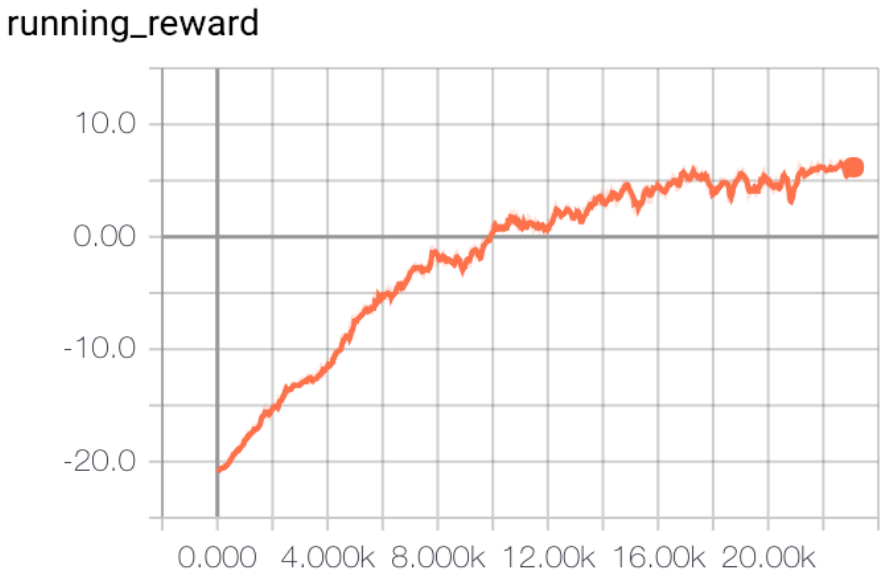

Fig 3.3.2 Training Rewards of Policy Gradient Network Model-1

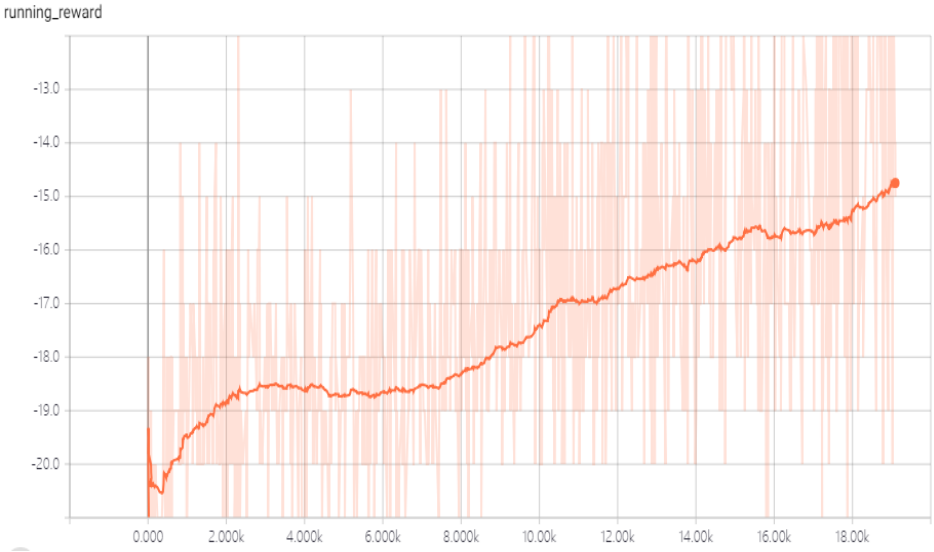

Fig 3.3.3 Training Rewards of Policy Gradient Network Model-2

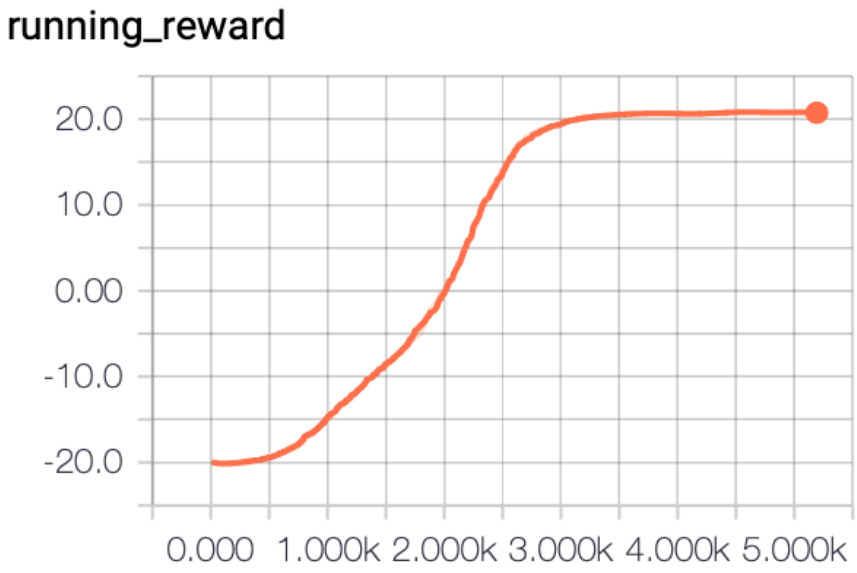

Fig 3.3.4 Training Rewards of Policy Gradient Network Model-3

Discussion

1. Comparison of the Two Algorithm

From our result of rewards after same number of episodes as Fig 4.1.1 and Fig 4.1.2, we can see that Policy Gradient has a faster convergence rate than DQN. But we may also notice that Policy Gradient tends to converge to a local optimal. In addition, Policy Gradient is able to learn stochastic policies since it outputs a probability distribution, while DQN can only learn deterministic policy. This contract may explain when training in the *Pong-v0* environment, DQN performs much worse than Policy Gradient while performs better in the *PongDeterministic-v4* environment which reduces much uncertainty (shown in Fig 4.1.3 and Fig 4.1.4).

However, Policy Gradient has a high *variance* in estimating the gradient of the expected future rewards even after it converges, while DQN shows a more stable performance. And Policy Gradient has a risk of converging to the local maximum. In contrast, DQN always tries to reach the global maximum.

Generally speaking, Policy Gradient is believed to have a wider application range than DQN, which could be the instances where the Q-function is too complex and the action space is continuous. Two algorithms have different advantages and drawbacks under different game environments. The goal of the project, trade-off between efficiency and performance, hardware set up and game environment should be considered when deciding which algorithm to use.

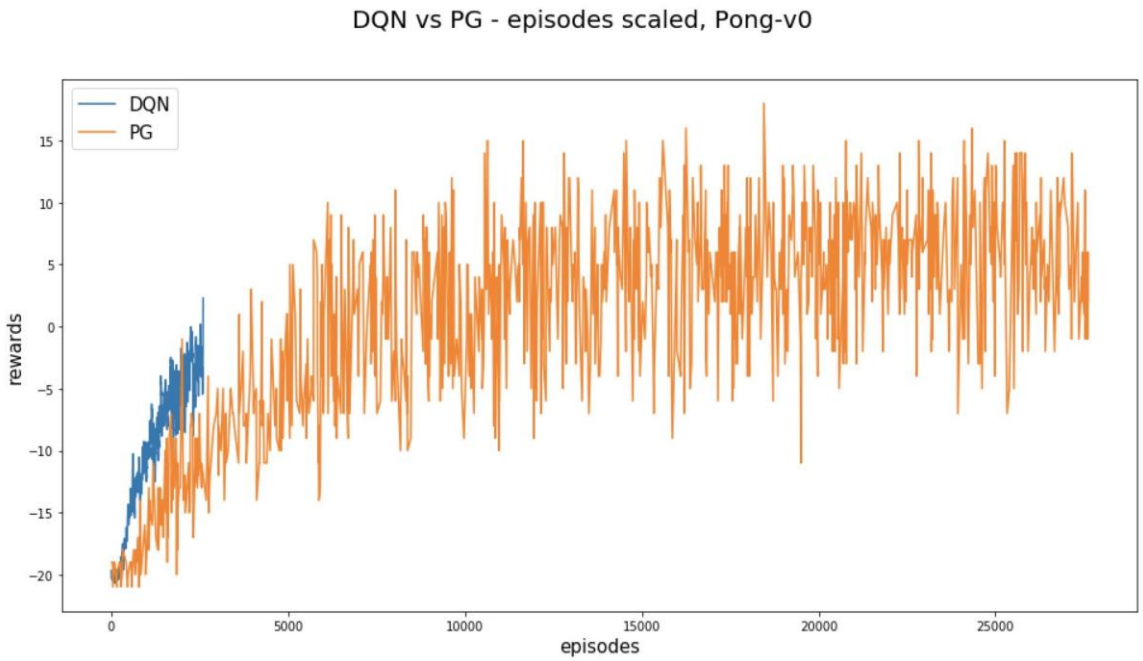

Fig 4.1.1 Performances of DQN and Policy Gradient Agents in Pong-v0 Environment compared on episode number

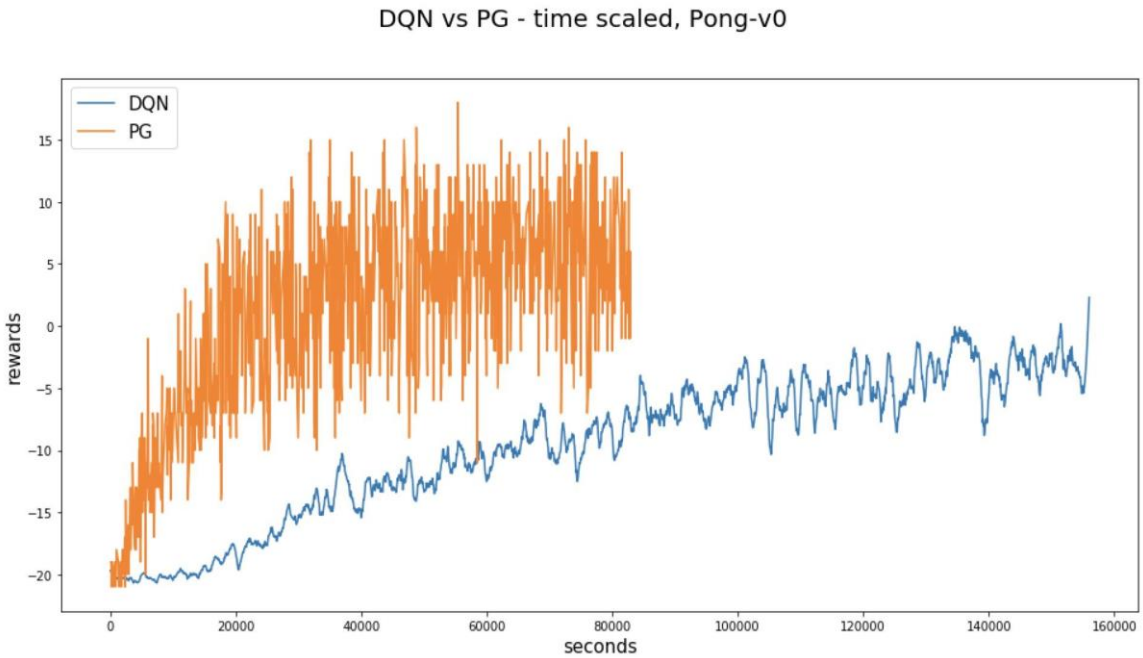

Fig 4.1.2 Performances of DQN and Policy Gradient Agents in Pong-v0 Environment compared on relative time

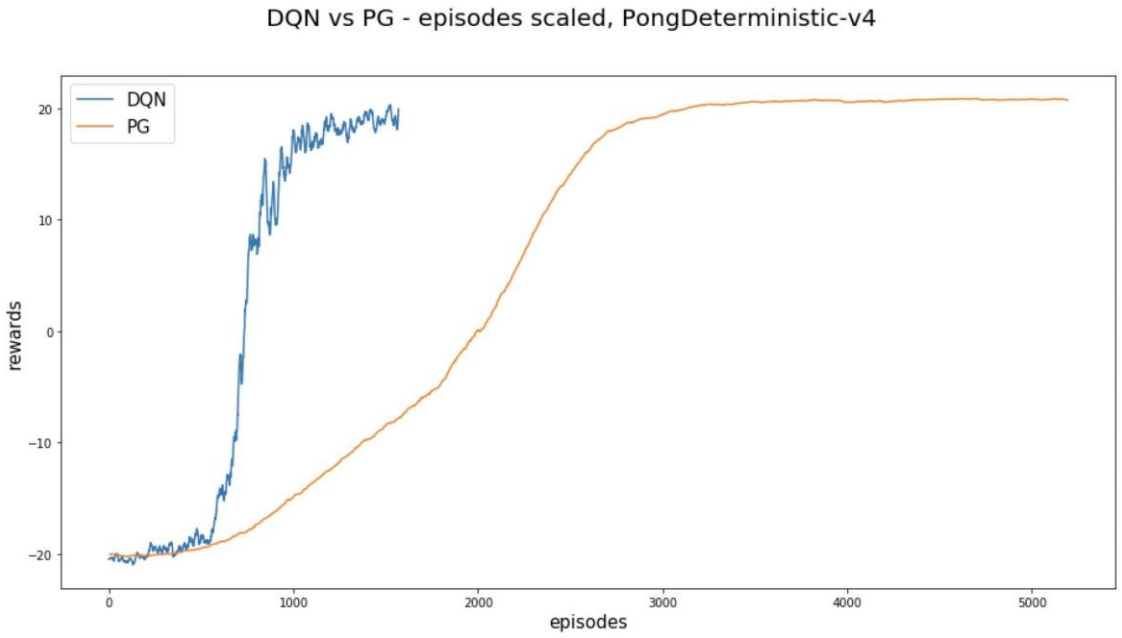

Fig 4.1.3 Performances of DQN and Policy Gradient Agents in Pong-v4 Environment compared on episode number

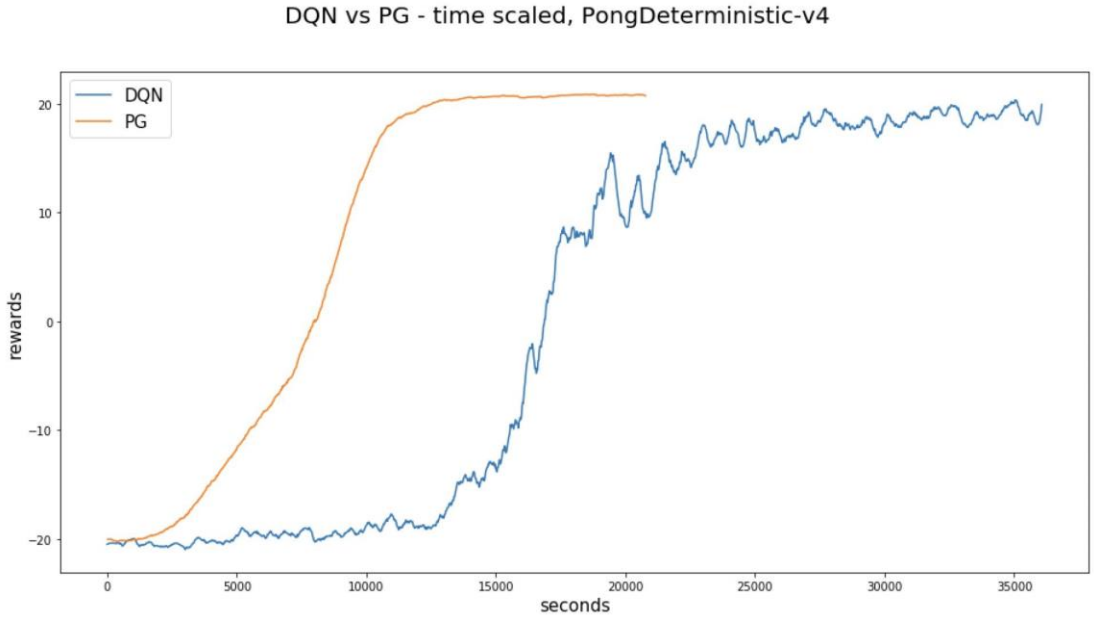

Fig 4.1.4 Performances of DQN and Policy Gradient Agents in Pong-v4 Environment compared on time

2. Potential Winning Patterns

Another interesting finding is that in the *PongDeterministic-v4* environment setting, both DQN and Policy Gradient achieve +21 rewards in one episode at the end of training and

both agents formed a similar pattern of winning rewards, which can be shown in the GIF in the README file.

If we look closely, we can find out that the agent always hit the ball so that it bounces back from floor and then score through computer's upper left side, which makes sure that next ball will kick off to the upper left side, and computer will hit the ball back to the very similar place every time. Agents we trained might find the correlation between adjacent games, and using the same pattern every game. It might be the reason why Pong requires lower amount of training time (i.e. easier) compared with other Atari games relatively. This sort of strategy might fail if the agent is playing with a real person rather than computer, unfortunately OpenAI does not allowed AI play with human being or another AI yet so we can't tell.

Further Exploration

1. DQN

DQN can handle more difficult games, however it might require more computational power. As far as we understand, *Colab* won't be enough for a more complex game.

2. Policy Gradient

Three methods can be employed to improve our current model given more time. The first one is to introduce baseline using the Advantage Actor Critic (A2C) model to cut variance. The second one is employing more agents using the A3C model to reduce training time. The last one could be to construct a more complex policy network by utilizing Convolutional Neural Network and experiment with different parameters.

Works Cited

awjuliani. (2017). *Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond*¶ . Retrieved from https://github.com/awjuliani/DeepRL-Agents/blob/master/Double-Dueling-DQN.ipynb

Cunningham, J. (2018). *Code from lecture 6*. Retrieved from http://stat.columbia.edu/~cunningham/teaching/scratch_lec06.ipynb

fg91. (2018). *Deep-Q-Learning.* Retrieved from https://github.com/fg91/Deep-Q-Learning/blob/master/DQN.ipynb

Grimm, H. (2018). *Week 4 - Policy Gradients on Atari Pong and Mujoco.* Retrieved from https://hollygrimm.com/rl_pg

Hasselt, H. v. (2016). Retrieved from http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847

He, K. (2015). *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.* Retrieved from https://www.cv-foundation.org/openaccess/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html

Karpathy, A. (2016). *Deep Reinforcement Learning: Pong from Pixels.* Retrieved from http://karpathy.github.io/2016/05/31/rl/

Mnih, V. (2013). *Playing Atari with Deep Reinforcement Learning*. Retrieved from https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

Mnih, V. (2015). *Human-level control through deep reinforcement learning.* Retrieved from https://www.nature.com/articles/nature14236/

Mnih, V. (2016). *Asynchronous Methods for Deep Reinforcement Learning .* Retrieved from https://arxiv.org/abs/1602.01783

OpenAI. (2017). Retrieved from https://blog.openai.com/openai-baselines-dqn/

OpenAI. (2018). *Game init code (Frame Skipping info)*. Retrieved from https://github.com/openai/gym/blob/master/gym/envs/__init__.py

Sutton, R. S. (2000). *Policy gradient methods for reinforcement learning with function approximation.* Retrieved from http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf

Wang, Z. (2015). *Dueling Network Architectures for Deep Reinforcement Learning* . Retrieved from https://arxiv.org/abs/1511.06581