Playing Atari games with Deep Reinforcement Learning

Team Members:
Qinyuan Sun (ECE major, 3rd year PhD)
Zhanpeng Zeng (CS major, Senior)
Jingyi Zhao (Chemistry major, CBE major, CS minor, Senior)

Abstract

We evaluated different deep reinforcement learning(RL) algorithm to play Atari games. The algorithms we studied include deep Q network (DQN), policy gradient and asynchronous method. We used OpenAI Gym to test our agents, and the agents are evaluated in 4 Atari games: Breakout, Pong, Space Invaders, Alien.

Introduction

Reinforcement learning algorithm used to leverage hand-crafted features. However, it is hard to design and task specific. Deep convolution network enable reinforcement learning to encode the environment without aid of human and perform effectively. The objective is to design a general learning agent that performs well in different Atari game environments using deep convolution feature representation.

We evaluated three types of algorithm: value-based approach, policy-based approach and asynchronous approach. Value-based approach attempts to approximate the action values on each state and select action according to the action values. The most representative algorithm is Q learning. An alternative method to find a good policy is to search directly in the policy space using gradient of certain objectives, such as discounted total rewards. Asynchronous approach built upon policy-based approach and performs asynchronous training in a distributed setting. The state of art is asynchronous advantage actor-critic (A3C).

We implemented deep Q learning, policy gradient and asynchronous advantage actor-critic algorithms. Our learning agents can explore the environment and learn good policy for the selected Atari games.

Problem Definition and Algorithms

Task

We were trying to explore three different types of agents and see how well they perform in different game environments. Thus, the input and output and network we used are different in different agents. However, the input, output, and networks we used are provided in each subsection of result section.

Algorithm

The algorithm we evaluated are DQN, policy gradient and A3C.

We covered DQN in class. The DQN algorithm uses replay memory and convolutional network to approximate the action value. For each episode, the agent makes initial observation, then repeat the following: take an action, and observe new state

and receive reward, then store the transition to replay memory, then randomly sample a minibatch of transition to train the model.

The A3C algorithm use multiple asynchronous workers to explore the same global environment. For each worker, the algorithm use estimated advantage as the objective function to search the policy space using root mean square backpropagation. The network build two fully connected neural network upon the convolution layer representation. One is to estimate value of the state, the other is used to determine the policy.

The policy gradient algorithm uses neural network to estimate the probability of choosing certain action given a state. The previous two algorithm use temporal difference learning method to train the agents, but policy gradient uses Monte-Carlo learning method. Basically, in each time, the policy gradient agent observes 10 consecutive episodes, then it estimates the how well an action using the actual discounted reward agent gains through out the episode, and then update the network.

Experimental Evaluation

Methodology

OpenAI Gym is a reinforcement learning testbed that provides a simple interface between agent and environment. In the game environment, the observation is an RGB image of the screen. Each action is repeatedly performed for a duration of k frames. After initializing a specific game environment, the agent will receive an initial observation. The agent chooses an action from a valid action space. OpenAI Gym then provides the next state, the reward gained from this transition, and binary value states whether the episode is completed. The achieved normalized reward is our evaluation metric. We are interested in how effective the agent can learn to play the game, which is represented by maximum normalized rewards. In addition, we want to compare training speed. However, since each algorithm ran on different platform, it is hard to compare fairly. DQN ran on both local GPU machine and google cloud 8 CPU VM. Policy gradient and A3C ran on google cloud 8 CPU VM.

We also performed multiple controlled experiments in DQN beside running DQN in different games. We tried different independent variables: the Q value update rules, different activation functions, and different batch sizes. We evaluate how the DQN agent's learning is affected by these variables.

Results

DQN

After we implement DQN agent, we test this agent in the 4 games: Breakout, Space Invader, Pong and Alien.

DQN network configuration (Figure is at the end of this report):

Input Layer	80 x 80 x 4
Convolutional Layer with ReLU	16 filters of size 8 x 8 x 4 with stride 4
Convolutional Layer with ReLU	32 filters of size 4 x 4 x 16 with stride 2
Fully Connected Layer with ReLU	256 hidden units

Output Layer	Number of actions

The parameters that we used is the following:

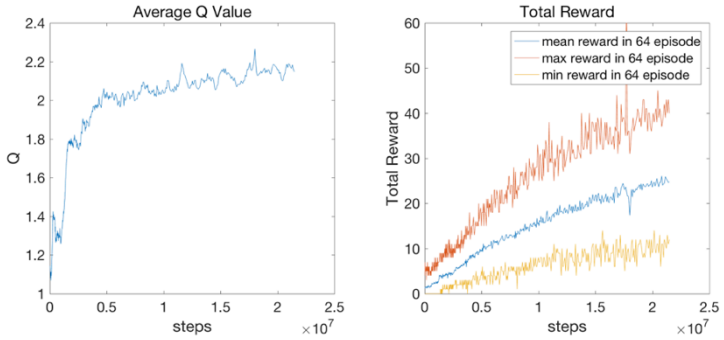
State representation	Last 4 observations
Optimizer	Adam with Learning Rate 10e-6
Batch size	32 transitions
Experience Replay Memory Size	Last 0.3 or 0.4 million transitions
Epsilon Greedy Policy	Epsilon decreases from 1 to 0.1 in first million steps stay at 0.1 after the first million steps
Normalized Reward	+1 for positive reward 0 for no reward -1 for negative reward

The x axis of the following plots is the total number of steps the agent experiences.

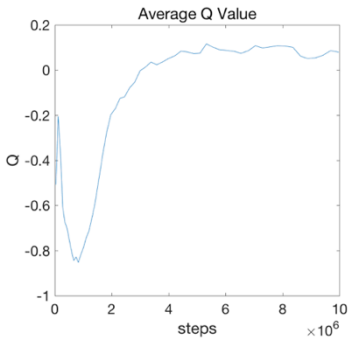
For the left plots, to measure the average Q value, we initially sample a set of 1000 states before training using random walk. Then after each episode complete, we compute the average estimated Q value of these 1000 states. DeepMind's paper, Playing Atari with Deep Reinforcement Learning, suggests that this metric is more stable in measuring performance than directly measuring the total reward agent gains in one episode. The average Q metric is smoother as expected, but after a while of training, the average Q value stabilizes or converges while the total reward can be still increasing.

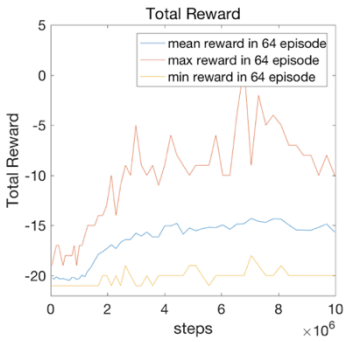
For the right plots, we plot the total reward the agent gains in one episode. Since this metric is not stable, we smooth this reward by taking the mean of 64 consecutive episodes. And to get more information about these 64 episodes, we also plot the maximal total reward and minimal reward in these 64 episodes.

Breakout:

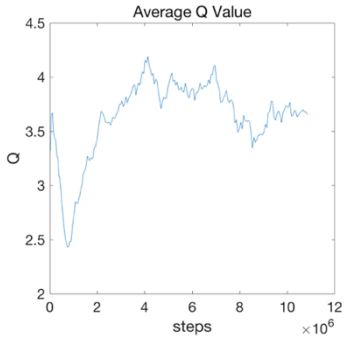


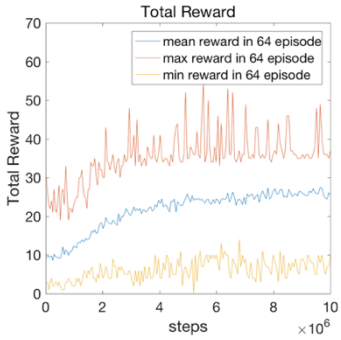
Pong:



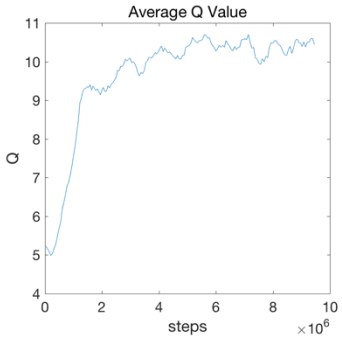


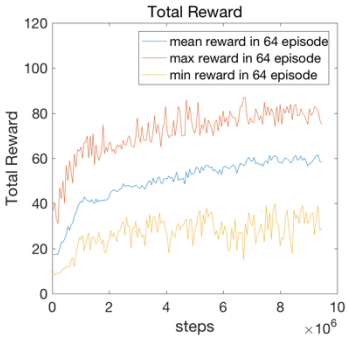
Space Invaders:





Alien:



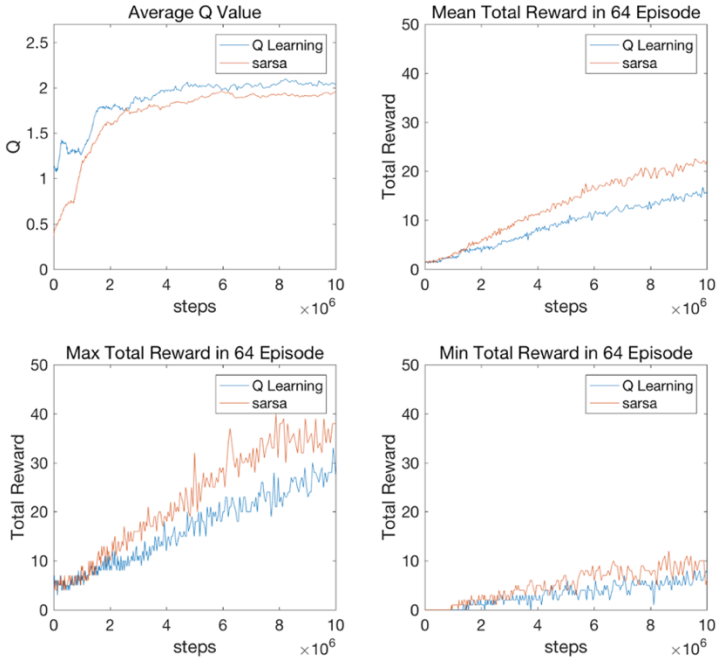


Discussion

The results are not expected. In Breakout, the average Q value seems converge although the total reward is still increasing. The average Q value metric does not seem to be accurate in measuring performance. Since the 1000 states is sampled though random walker, it's likely these state sample bias toward the beginning of games since random walker may not be able to explore the game environment enough. One explanation for the convergence of this Q value is that the network may approximate the Q value for the states in the beginning of games very well, but the approximation for the states far from the beginning may not be good, so the performance continue increase while average Q value does not change much. This may suggest a new method to sample transition batch by using different probability distribution.

The learning results are also not expected. The performance fluctuates a lot during the training, and the variance is large. Although we used the mostly same parameters as the parameters states in DeepMind's paper, the mean total reward is smaller than the result in DeepMind's paper. However, the difference in learning curves of difference environments is expected. The four environments have different characteristics, so we expected to see the difference.

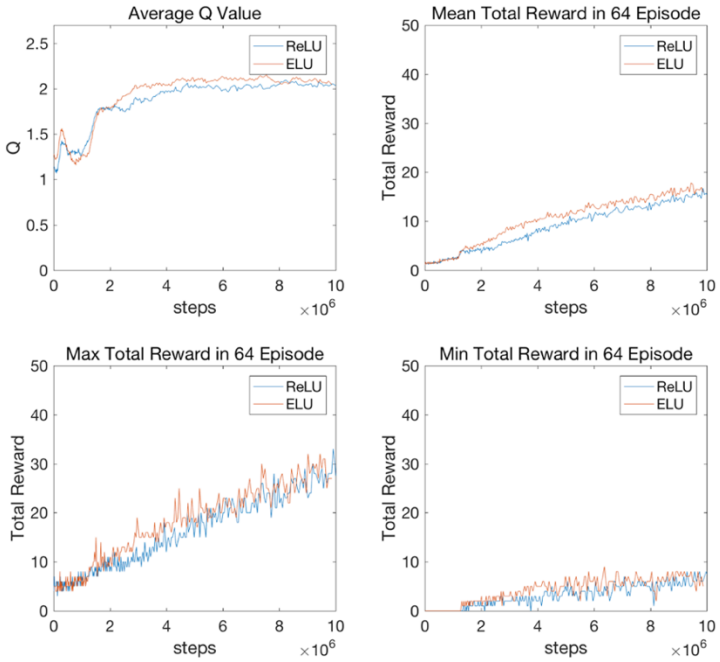
Comparing Q learning and Sarsa:



Discussion

Using Sarsa to update the Q value is suggested by Professor Shavlik. We run the experiment in Breakout, the performance improvement is quite significant, Sarsa DQN beats the performance of standard DQN in all mean, maximal, and minimal total rewards. And the result in average Q value is expected. The average Q value for Sarsa DQN is a bit lower than standard DQN. The reason is that in Q value update rules, Sarsa DQN uses $Q(S_t, a_t) = R + \gamma E[Q(s_{t+1}, a_{t+1}; \theta)]$, but standard DQN uses $Q(S_t, a_t) = R + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)$. Standard DQN uses maximal Q of next state, so the average Q value is larger than Sarsa DQN. However, because Sarsa DQN use the expected Q value in learning, we expected to see that Sarsa DQN have more stable performance, but the learning curve also has a lot of fluctuations, and the variance is also large.

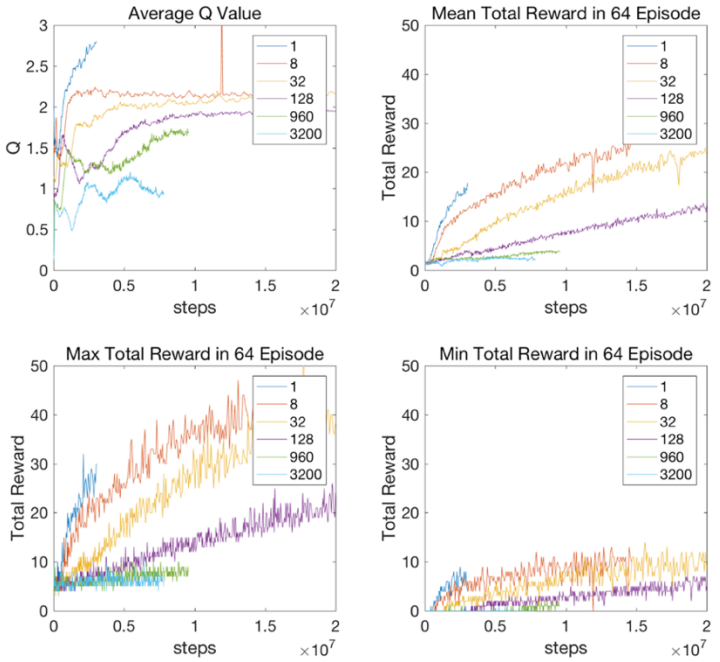
Comparing ReLU and ELU activation functions:



Discussion

We also try to replace ReLU activation function with ELU activation function. The experiment also run in Breakout. Using ELU is supposed to provide a better convergence property, but we did not observe any significant performance improvement.

Comparing different batch sizes:



Discussion

In this experiment, we tried different batch size. With smaller batch size, the overhead of the data transfer between GPU and CPU is large, but with larger batch size, the model is likely to overfit the data. Thus, we want to see what is the optimal batch size to train the agent without losing learning performance. The experiment also run in Breakout. In addition, each step corresponds to 32 instance being feed to training. For example, for batch size of 1, in each step, the network performs 32 times stochastic gradient descend. And for batch size of 8 and 32, the network performs 4 and 1 times minibatch training respectively. And for batch size of 128, 960, and 3200, the network performs a training in every 4, 30, and 100 steps respectively.

The result is quite interesting and straightforward. The smaller batch size is better in all learning performance. However, note that the stochastic gradient descend has the shortest learning curve. The reason is that it takes a very long time to training. Since the batch size of 1, 8, and 128 run on the same machine with the same total training time, we can compare they in another dimension. Although the batch size of 1 has the best learning performance in terms of the number of steps, the batch size of 8 has the best learning performance in terms of total training time. The larger batch size can iterate through much more training instance than smaller batch size in the fixed amount of time, so there is a tradeoff.

Asynchronous Advantage Actor-Critic

We implemented A3C algorithm and we tested it on 3 games: Breakout, Space Invader and Pong.

The A3C network configuration is the following:

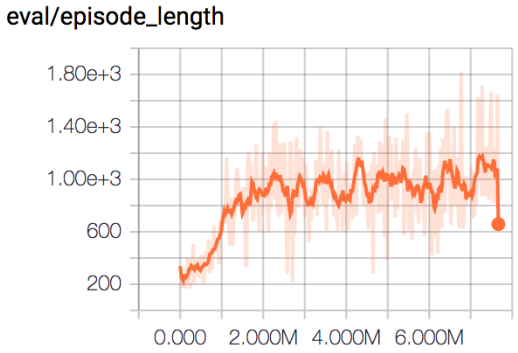
Input Layer	80 x 80 x 4
Convolutional Layer with ReLU	32 filters of size 42 x 42
Convolutional Layer with ReLU	64 filters of size 21 x 21
Convolutional Layer with ReLU	64 filters of size 10 x 10
Fully Connected Layer with ReLU	512 hidden units
Output Layer	Number of actions for policy net, one for value net

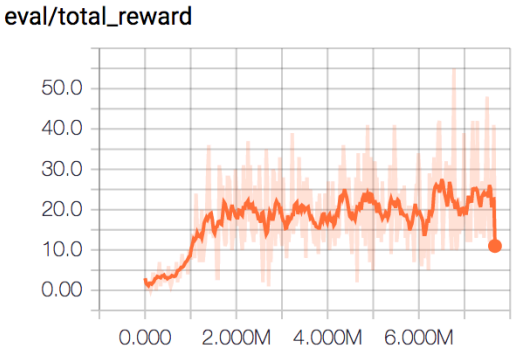
The parameters that we used is the following:

State representation	Last 4 observations
Optimizer	RMSprop with Learning Rate 10e-6
Normalized Reward	+1 for positive reward 0 for no reward -1 for negative reward

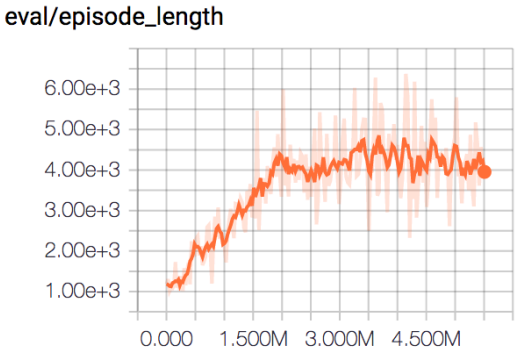
The normalized reward for each game is reported in the following figures. The x-axis is number of steps. We can see that after 2 million steps, the algorithm reaches a performance bottleneck and the total normalized rewards just fluctuates. The total reward the agents can achieve is on par with the DQN. The training time it takes to reach the same level of total reward of DQN is two folds less.

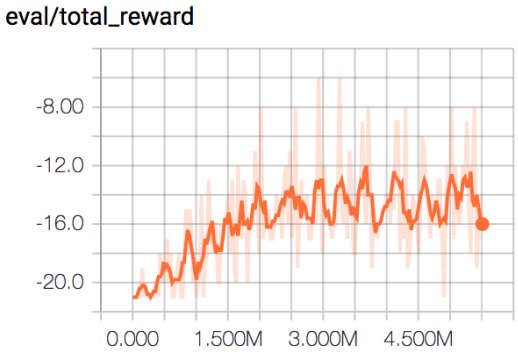
Breakout



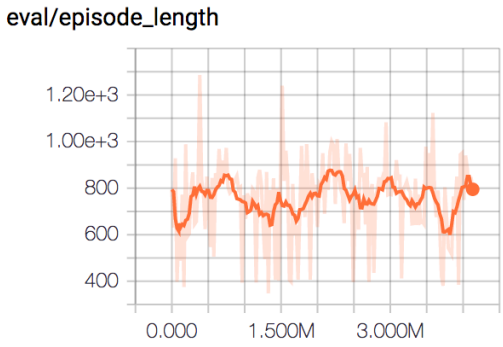


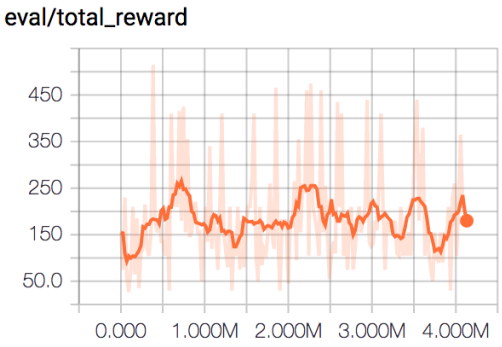
Pong



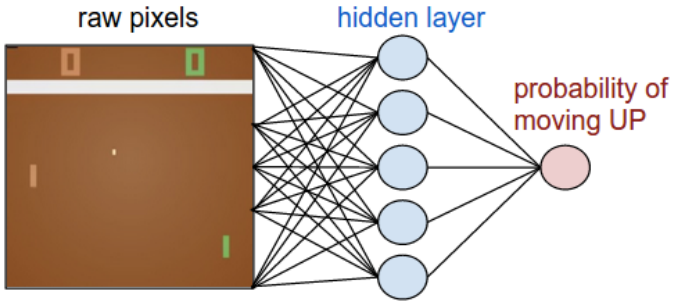


Space Invader





Policy Agent



The raw image was downsized from 210x160x3 into 80x80x1 pixels, and aligned into a input layer with 6400 neurons. We connect this input layer to a 200 neurons' hidden layer, and eventually the output. The output gives the probability of moving the pad up to win the game based on numbers of episode being trained. W1 and W2 are initialized randomly. Each Game has approximately 500 frames (500 pictures).

Learning:

Gradient per action: $\frac{\partial L_i}{\partial f_j} = y_{ij} - \sigma(f_j)$, where L_i is the value of result and f_j is the function that gives the activation of final layer. y_{ij} is supposed be the true label of the action. Since we are feedforwarding, we are treating every action as the "rewarding" and assign y_{ij} as 1. At the end of each episode, the gradient is computed. If an action leads to winning, this action would be more favored.

At the end of each episode, gradients, rewards, and observation of 10 games will be extracted. Actions taken in the beginning of the games are discounted in weight.

Backpropagation method:

RMSProp:

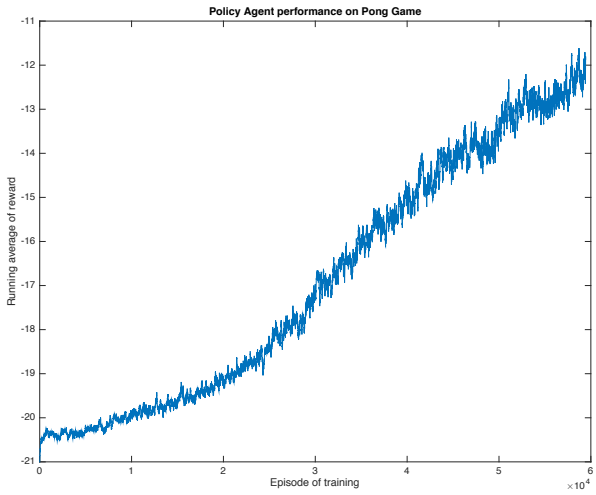
$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1\, g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}\, g_t$$

Where $E[g^2]_t$ is the running average at time step t, and g_t is the gradient at current time step.

Parameters:

Specified Parameters	Value
Updating period (batch size)	10
gamma	0.99
number of hidden units	200
input dimensions	80x80
learning rate	0.0001

Performance:



The results could be better if run longer.

Individual Contributions

Qiyuan Sun: A3C
Zhanpeng Zeng: DQN
Jingyi Zhao: Policy Gradient

Conclusion

We explored three types of algorithm: value-based approach, policy-based approach, and asynchronous approach. We tried that DQN and A3C in different game environments. The RL agents can learn how to play games in all game environments, but the learning performance in different games are different, which suggests that there may be some properties of the games that the agents cannot learn very well. Thus, in out view, although the agents are general enough to handle different games, these agents are not general enough to learn different rules in different environments.

Future Work

Zhanpeng Zeng: I will continue exploring the topic of reinforcement learning. Particularly, I am interested in model-based reinforcement learning, which try to learn the transitions between states. The Q learning approach needs a lot of episodes for agents to perform well in environments. I am more interested in how to use the episode information more effectively. Model-based approach requires fewer steps to train the model, so I may look into this approach.

Qinyuan Sun: all the agents we implemented use normalized rewards. OpenAI gym provides non-normalized rewards. I am wondering how will this affect the training process. Furthermore, OpenAI provides next consecutive frames as input observation. We want to explore whether using further backward observation can improve gameplay learned by the

agent. This is different from random sample from experience replay.

References

DQN:

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

A3C:

Mnih, V., Badia, A., Mirza, M., Graves, A., Lilicrap, T., Harley, T., Silver, D., Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783.*

Policy Gradient:

Sutton, McAllester, et al., 1999, R.S. Sutton, D.A. McAllester, S.P. Singh, Y. Mansour, *Policy gradient methods for reinforcement learning with function approximation*, Advances in neural information processing systems (NIPS), vol. 12 (1999), pp. 1057–1063

Andrej. *Deep reinforcement learning: Pong from pixels*.(2016)

Network Configuration

DQN: