Conventional Research Project

COMP90055 (25 points)

Performance of Different Deep Reinforcement Learning Algorithms

Supervisors

Nir Lipovetzky: nir.lipovetzky@unimelb.edu.au
Timothy Miller: tmiller@unimelb.edu.au

Students

Bahriddin Abdiev
Student ID: 824110
babdiev@student.unimelb.edu.au

Lingtao Jiang
Student ID: 867583
lingtaoj@student.unimelb.edu.au

Zhenxiang Wang
Student ID: 879694
zhenxiangw@student.unimelb.edu.au

Performance of Different Deep Reinforcement Learning Algorithms

Bahriddin Abdiev
Student ID: 824110
babdiev@student.unimelb.edu.au

Lingtao Jiang
Student ID: 867583
lingtaoj@student.unimelb.edu.au

Zhenxiang Wang
Student ID: 879694
zhenxiangw@student.unimelb.edu.au

Abstract

Deep reinforcement learning without prior knowledge about environment achieves complex goals in a wide variety of games. This aspect of deep reinforcement learning algorithms created a new wave towards general AI algorithm. As a result, deep neural networks have used in multiple versions of Q-learning and policy-based algorithms developed over the last few years, and each of them has some advantages over the others. In this paper, we analyse some of these algorithms by mapping their behaviour on Lunar Lander game.

1 Introduction

Deep Reinforcement learning algorithms are supposed to solve infinite state-space problems effectively with little or prior knowledge about problem environment. Another significant advantage is that they independently “learn” by interacting and observing with an environment. Although different deep reinforcement learning algorithms seem to do the same job from a high overview, under the hood there are structural and technical dissimilarities. Thus, different algorithms exhibit various behaviours. Comparing, analysing and reasoning them help us gain more profound knowledge about the workings of advanced self-learning AI algorithms. This goal forms the primary objective of this paper. Specifically, following hypotheses are proposed:

1. Using modified versions of Deep Q-Network (DQN) such as Double DQN (DDQN), Prioritized Experience Replay (PER) and Dueling DQN can significantly improve scores and shorten agent training time compared with using DQN.
2. If we combine value-based reinforcement learning algorithm with policy-based ones on our AI agent, then it can get higher scores in less training time than using either algorithm alone.

Lunar Lander game from Open AI gym package (Brockman et al., 2016) is chosen as an environment to test AI algorithms. After implementing each algorithm, they are trained for 50000 episodes and played the game for 100 times. Results are provided in section 5. Learning pace, stability and game results are selected as experiment evaluators. Research showed that modified DQN algorithms significantly shortened training time than DQN. However, an Actor-Critic algorithm which combines Q-learning and policy-based ones does not show better result even though it learns comparatively faster than all other implemented algorithms.

2 Related Work

Watkins & Dayan (1992) proved reinforcement learning algorithm converges with probability 1 when all states and action-values are represented discreetly. After this, it became hot topic among AI researchers. However, using non-linear approximator functions like the neural network was argued to cause instability and, in most cases, divergence (Tsitsiklis & Roy, 1997, p. 674). Mnih et al. (2015) proved the opposite by using neural networks to approximately calculate Q-values of each pair of state and action. After that, different modifications of DQN were published by different researchers:

- Shaul et al. (2016) modified DQN by using advanced data structure to store samples in memory. This technique helped to pick samples with evenly distributed TD-error from memory and stabilise learning progress.
- Introduced by Hasselt et al. (2016), Double DQN was proved to obtain more advanced

performance by solving overestimation without an additional network.

- By improvising the structure of the neural network, Wang et al. (2015) presented an improvement of DQN called Dueling DQN. It separates the output into two streams so the Q-value is estimated based on the state and actions are estimated independently.

Another vital area of reinforcement learning is the policy-based algorithms. Sutton et al. (2000) explored a new policy gradient method. It represents the policy by its function approximator, which is independent of the value function. The policy is updated pertaining to the gradient of expected reward concerning the policy parameters.

Actor-Critic method is a combination of value-based and policy-based methods. Contrary to intuition, it appeared earlier than the value-only and policy-only methods. Witten (1977) proposed an adaptive controller for discrete-time Markov environments. This implementation is the embryonic form of the Actor-Critic algorithm. Barto et al. (1983) formally introduced the Actor-Critic architecture. After deep learning is widely applied, the Actor-Critic method combines the development dividends of the policy-based and the value-based method. Mnih et al. (2016) proposed the Asynchronous Actor-Critic method, which used asynchronous gradient descent for optimisation of deep neural network controllers.

3 Methods

DQN is chosen as a baseline algorithm. Here we

- Briefly introduce environment
- Explain each algorithm - how it was implemented
- Discuss why these algorithms are chosen.

All methods are evaluated using the same environment, the Lunar Lander game. Eight features are extracted to enable the agent to perceive the environment adequately. These features include:

- The horizontal distance to the goal position
- Vertical distance to the goal position
- Horizontal velocity
- Vertical velocity
- The angle between the lander and a plumb line
- Angular velocity
- Whether the left foot gets contact to the ground.
- Whether the right foot gets contact to the ground.

The action space in this environment contains four actions: using central power; using left power; using right power or doing nothing. Furthermore, the agent keeps recording a 'shaping' value at all states, which is calculated by negatively weighting the lander's distance to the goal position, velocity, angle and positively weighting the legs get contact to the ground. The reward of any actions comes from the difference between current shaping and the shaping value from the previous state. Also, the agent will receive lower reward from consuming more power, which is applied more strenuously to the central power than side power. When the game is over, a successful landing in goal position leads to a considerable positive reward while a crashing leads to a negative reward. In this case, the agent is expected to learn how to land in goal position with as less energy consumption as possible.

3.1 Deep Q-Network

Deep Reinforcement Learning is a kind of algorithm that combines deep learning with reinforcement learning to achieve an end to end learning from perception to action. In brief, just like humans, the deep neural network accepts input of perceptual information like images, then it directly outputs the best action as a result. Although the idea of combining deep learning with reinforcement learning was raised a few years ago, the real success was observed at the beginning of the Playing Atari with Deep Reinforcement Learning published by Mnih et al. in 2015.

DQN uses two neural networks: one as the main network, another, also known as target network, for predicting next state's highest Q-value. The latter is updated periodically by copying main network's weights and biases (in our implementation, after every 1000 steps). It is separated into different networks to reduce correlation with the target (Mnih et al., 2015). Another improvement was to take a random set of samples from the pool of stored samples, replay and improve the model to reduce correlations in the observation sequence and to smooth over changes in the data distribution (Mnih et al., 2015).

The Q-learning update at iteration *i* uses the following loss function (Mnih et al., 2015):

$$L_i(\Theta_i) = E_{(s,a,r,s')\sim U(D)}\left[(Q' - Q(s,a;\theta_i))^2\right] \quad (1)$$

$$Q' = r + \gamma max_{a'} Q(s', a'; \theta_i^-)$$

Here, γ is the discount factor, θ_i and θ_i^- are the parameters of the main and target Q-networks in iteration *i*, respectively. U(D) is a pool of stored samples. DQN converges by minimising this loss value gradually. In the implementation, Adam optimiser is used as it is proved to show shorter time to converge and keep global maximum stable after it reaches that point. It can be explained by Adam's adaptive learning rates (Kingma & Ba, 2017). We also updated main network's weight and biases after ten steps to make the learning process faster.

The input layer of the network is the game state (s) which is an 8-dimensional vector. Output layer cover's each action (a). In other words, output vector's coordinates define Q(s, a) values. Research showed that using one hidden layer with size 50 converges faster to achieve the goal and it did not meet overfitting issue. It uses rectified linear unit while output layer uses linear as an activation function.

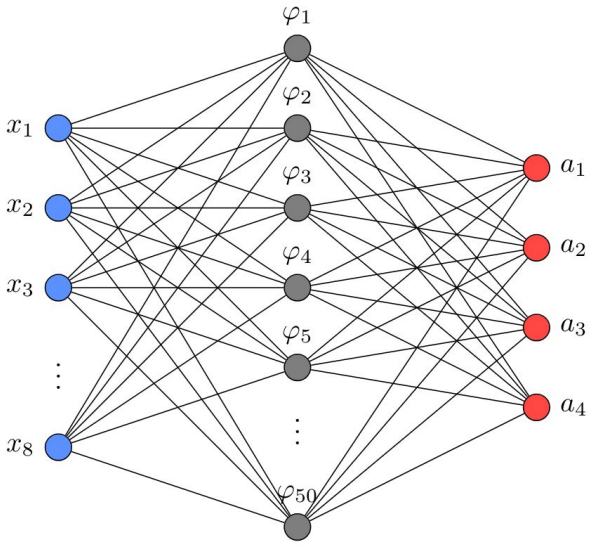

Figure 1. Structure of the DQN network.

We used the linear epsilon-greedy method to solve the exploration vs exploitation problem:

$$\varepsilon = \varepsilon_{max} - max(\frac{\varepsilon_{max} - \varepsilon_{min}}{A} \cdot i,\ 0) \qquad (2)$$

Here, $\varepsilon_{max} = 1$, $\varepsilon_{min} = 0.1$, $A = 10,000$ - annealing steps, i - current step.

For this and all other Q-learning based algorithms following tuning parameters are used:

Learning rate	0.0001
Pool size of samples	500000
Batch size	512
Gamma	0.99
Target update frequency (steps)	1000

Table 1. Tuning Parameters.

Since our environment's action space is discrete and narrow while its state space is infinitely large, DQN is expected to give outstanding results (and it did as depicted later).

3.2 Double DQN

Due to the inherent drawback of Q learning (Thrun et al., 1993), it is well justified that DQN can perform poorly in many stochastic environments. During the training of neural network, the agent continually observes state, action, reward and the following state. It is supposed to choose the action which maximises the Q value at any state. In this case, the agent can provide a policy based on the maximised expected cumulative discount reward. However, if the Q value is estimated by an approximator instead of a lookup table, which means it is not precise, the noise in the estimation is likely to introduce some overestimation.

If all Q values are overestimated with the same amplitude, that is, the overestimation is uniform, the problem seems to be trivial since the policy is based on greedy strategy. Nevertheless, Hasselt et al. (2016) claimed that the overestimation in Q learning is common but not predictable. When the overestimations are not equivalent, the agent might spend time exploring unbeneficial states. A solution to this issue is applying Double Q Learning (Hasselt, 2010) in DQN.

Double Q Learning uses two Q functions to choose the action and approximates the expected Q value of actions separately. Each of these functions is updated with the values from each other for the next state. Since they are updated to solve the same problem but with a different set of samples from the experience pool, they can be reckoned as unbiased. Instead of overestimating, the Double Q learning sometimes underestimates the expected Q value.

As aforementioned, there are two neural networks in the implementation of DQN: main and target networks. It indicates the expected Q value can be estimated without other networks being added. The update of Q value in Double DQN is mostly similar to

DQN, but the target network is utilised to estimate the Q value of actions chosen by the main network.

$$L_i(\Theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \cdot Q_{target} - Q(s,a;\theta_i)\right)^2\right] \quad (3)$$
$$Q_{target} = Q(s', argmax_{a'}Q(s',a';\theta_i);\theta_i^-)$$

Compared with equation (1), it is evident that Double DQN updates Q target using action which maximises the Q value in the main network. For comparability, the remaining part of neural networks is the same as DQN, including the utilisation of experience pool and periodically updates.

3.3 Prioritized Experience Replay

Intuitively, if a humans play a game, they learn more from unexpected observations by replaying this scenario in their brain over and over until it is absorbed. Schaul et al. (2016) introduced a new version of DQN, namely Prioritized Experience Replay (PER) where particular data structure used to imitate this process. All other parts of the algorithm are same with DQN.

For each sample it's TD-error will be calculated:

$$error = \left|r + \gamma max_{a'}Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right| \quad (4)$$

Then this error is converted to the priority value:

$$p = (error + \varepsilon)^{\alpha} \quad (5)$$

Where ε is the positive small number. It is utilised to guarantee to make priority value non-zero; α is also positive number in [0, 1] to narrow the range of priority values.

Proportional prioritisation is used to get samples from memory because it showed better performance in our environment. Proportional prioritisation technique's "sum-tree" structure is very similar to priority queues:

1) Each leaf keeps a pair of the sample, and it's priority value.
2) Parent node's priority value is the sum of it's both left and right children's priorities.

If these 2 steps are repeated, starting from leaves up to the root, the root will get a summation of all priorities. If random number *r* is picked between 0 and summed priority value, it will go further to the left node if $r < p_{left}$, otherwise $(r - p_{left})$ value goes further to the right node. This step is repeated until it gets a leaf. It has been shown in previous research (Shaul et al., 2016), this way, samples with higher priorities are more likely to be chosen. However, proportional prioritization picks random number *r* from each equally divided *n* segments (n - batch size) and gets the leaf's sample from sum-tree. This way, it always gets samples with different priorities. This trick solves overestimation issue but sometimes may underestimate.

3.4 Dueling DQN

Presented by Wang et al. (2015), Dueling DQN adopts an unconventional architecture which separates the neural network into two estimators. As shown in Figure 2, Dueling DQN does not use a fully connected layer to produce a single stream output like DQN. Instead, two estimator layers are introduced: one outputs a value based on the state and the second outputs advantage values for each action individually. Subsequently, the state value and actions advantage values are sum up to an overall Q value.

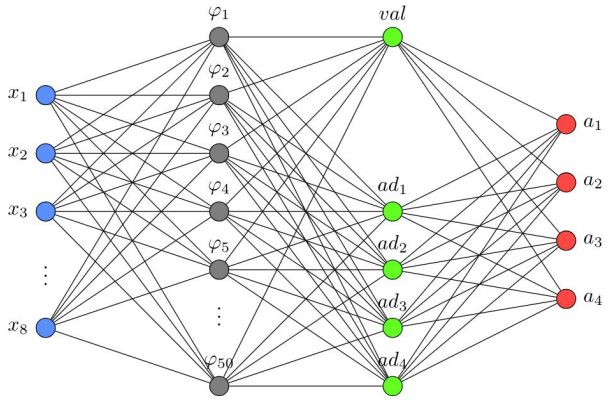

Figure 2. Structure of the Dueling DQN network.

In the following equation, V indicates the value of the state s, while A is the advantage of action a in state s. α and β refer to the parameters of the two fully-connected estimator layers.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \quad (6)$$

As the state value is scalar, it must be transformed into a vector with the length of the action space in the neural network. As a result, there is no way to resolve the state value or advantage value through the Q value. One solution is to subtract the state value and advantage value from the same constant which is the maximum advantage value. Thus, for the action a that gets the maximum Q value in state s, its Q value is equal to the state value of s.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - max_{a'\in|\tilde{A}|}(A(s,a';\theta,\alpha)) \quad (7)$$

Similarly, another alternative is to replace the maximum value in equation (7) with the mean average value. It will lead the state value to deviate from a constant value, namely, its real semantics will be lost. Yet, this is proved to make the optimizer more stable. Hence, this alternative is applied in the implementation of Dueling DQN.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'}(A(s,a';\theta,\alpha)) \quad (8)$$

The reason why this structure is adopted is that there are many cases the choice of actions does not significantly affect the next state. In these cases, the significance of calculating Q values of actions is not as high as the one of calculating the state value. The advantage of Dueling DQN is that it can separate action values from state values and quickly learn the state values. Furthermore, we also combine Double DQN and Dueling DQN to evaluate whether these two improvements can be operated coefficiently.

3.5 Policy Gradient

DQN guides the policy generation through learning value functions, including Q value functions:

$$V_\theta(s) \approx V^\pi(s) \quad (9)$$

$$Q_\theta(s,a) \approx Q^\pi(s,a) \quad (10)$$

$V_\theta(s)$ is the value function while $Q_\theta(s,a)$ is the Q value function. The policy can be generated from the value function using ε- greedy search method. In contrast, Policy Gradient will directly parameterize the policy itself and the parameterized policy will no longer be a probability set but a function (Sutton, et al., 2000):

$$\pi_\theta(s,a) = P[a|s,\theta] \quad (11)$$

In the above equation, the policy function is represented as a parameterized policy function $\pi_\theta(s, a)$. The policy function determines the probability of taking any possible action under given state and specific parameters. When using policy to generate actions, the action sampling is based on this probability distribution. The parameters in the policy function determine the shape of the probability distribution.

The purpose of parameterisation is to solve large-scale problems. In this study, it is impossible to strictly separate each state independently and point out that a particular state should perform a specific action. Therefore, Policy Gradient does parameterisation and use a limited number of parameters to approximate the actual functions reasonably.

Policy Gradient uses the parameterised policy function to get a better policy by adjusting these parameters. The actions that follow this policy will be rewarded more. The specific mechanism is to design an objective function and use the gradient ascent algorithm to optimise the parameters to maximise the reward. The objective function is $J(\theta) = E_{\pi_\theta}[r]$, which represents the discounted reward that policy can get (Sutton et al., 2000). Instead, we can calculate its gradient to update the parameter step by step. Using Tensorflow package, the equivalent work is done by designing a loss function and applying an optimiser to minimise the loss. The details of the loss function and optimiser will be discussed as follow.

Reward Guided Loss Function

The loss function contains two parts: cross entropy and discounted and normalised rewards. These two parts are merely multiplied by the loss function. The design choice is discussed as follows. Firstly, Cross-entropy indicates the distance between output distribution of the current model, and what the distribution should be. It is widely used when the output is a probability distribution, therefore, suits this case. Besides, future rewards are discounted by a discounting factor, and the reward is normalised by subtracting mean then dividing by standard deviation. This processing helps to reduce variance. Finally, cross entropy and discounted and normalised rewards are multiplied to produce a reward guided loss function, so the parameters of the network can be updated in a way that encourages actions with high rewards and discourages actions with low rewards.

Optimization

After defining the loss function, Adam Optimizer is used to train the model. The Adam optimisation algorithm is an improvement to stochastic gradient descent which helps accelerate the training.

Policy Gradient Network

A policy gradient network is built to approximate the policy, which has two hidden layers and one output layer. Each hidden layer has ten nodes and uses rectified linear unit as an activation function. The output layer has four nodes and applies softmax as the activation function to output the probability distribution of four actions. The input of the policy

gradient network is a vector of eight dimensions representing the state *s* as shown in Figure 3.

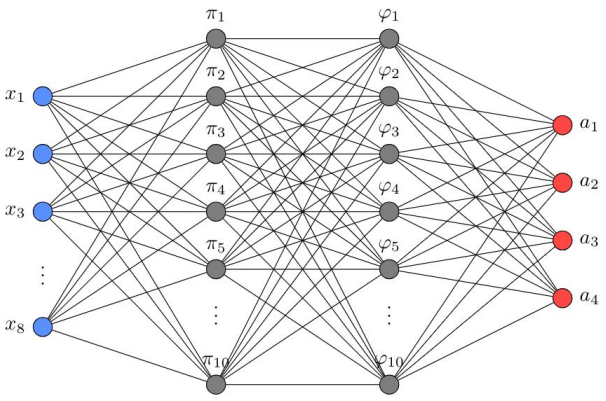

Figure 3. Policy Network Architecture.

Algorithm:

1. Obtain the observation (features) from the game.

2. Feedforward the observation (features) through the policy network to predict the probability distribution of each action, then sample from the distribution to generate an action.

3. Take the action and receive the next observation and the reward of this action from the game.

4. Store the transition for training, which contains the observation, the reward and the action taken.

5. Repeat 1-4 until this episode is done. Then train the neural network using the defined loss function by learning from the stored transitions for this episode.

6. Clean the transition buffer, start the next game and repeat the steps above until it reaches the defined number of episodes.

3.6 Asynchronous Actor-Critic

Actor-Critic

The variability of the state value estimated by policy gradient method is relatively large, that is, the variance is high. If an algorithm can estimate the state value relatively accurately and use it to guide policy updating, it will have better learning effect. This technique is the main idea of Actor-Critic policy gradient. Actor-Critic uses Critic to estimate the value of action in formula (10).

Actor-Critic based policy gradient learning contains two parts:

1. Critic: the parameterized action-value function $Q_w(s,a)$.

2. Actor: Update of the policy function parameter according to the value obtained by Critic part.

In this way, the Actor-Critic algorithm follows an approximate policy gradient:

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(s,a)Q_w(s,a)\right] \quad (12)$$

Jan Peters, Sethu Vijayakumar and Stefan Schaal (2008) have improved the original Actor-Critic algorithm by defining an advantage function shown below, and we have adopted the same method.

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \quad (13)$$

The practical meaning of the advantage function is when the agent takes action to leave the state s, how much more value it can get.

Now, the gradient of the objective function can be written as:

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(s,a)\, A_w(s,a)\right] \quad (14)$$

The advantage function can significantly reduce the variance of state value, so the Critic part of the algorithm estimates the advantage function instead of only estimating the value function of the action.

In our Asynchronous Actor Critic architecture, each Actor-Critic network will estimate both a value function $V^{\pi_\theta}(s)$, and a policy $\pi(s)$, which will be the output of two separate fully-connected layers sitting at the top of the network. We then use the TD error δ^{π_θ} to calculate the policy gradient.

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(s,a)\delta^{\pi_\theta}\right] \quad (15)$$

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s) \quad (16)$$

δ^{π_θ} is proved to be an unbiased estimation of the advantage functions. After discuss the theory of Actor-Critic, the next section will talk about how to implement the asynchronism.

Asynchronism

The idea of Asynchronous Actor-Critic is to create multiple worker agents in parallel environments. Each of these agents has their own set of network parameters and interacts with its copy of the environment at the same time while updating the parameters in a global network shared by all of them. The agents do not interfere with each other. The update of the parameters in the global network is subject to the discontinuity of local network updating, so the update correlation is reduced, and the convergence is improved.

The implementation of the Asynchronous Actor-Critic algorithm in this game is to synchronously train multiple worker agents with the same Actor-Critic

network structure in multiple threads. In the next section, we will explain the detail of Actor-Critic network structure.

Actor-Critic Network

Each Actor-Critic network consists of an input layer to process the input data, followed by a recurrent neural network layer with 64 cells to process temporal dependencies. The output of the recurrent neural network layer will be used for two purposes. One is to feed it into a fully-connected layer, which has 200 nodes and use the rectified linear unit as the active function, then transfer it into a value output layer. Another is to feed it into a fully-connected layer, which has 300 nodes and use the rectified linear unit as the active function, then transfer it into the final a policy output layer.

To update the network, we calculate the value loss, policy loss and the entropy of the policy. The entropy refers to the spread of action probabilities. If the actions generated by the output policy have similar probabilities, the entropy will be high, while if a single action has a substantial probability, the entropy will be low. The entropy is used to encourage the model to be conservative regarding its sureness of the correct actions.

Each worker will use this loss to obtain gradients concerning its network parameters and then uses the gradients to update the parameters in the global network. The Algorithm of Asynchronous Actor-Critic is shown below.

Algorithm:

Create a set of worker agent. Each has their network and environment.

For each worker:

1. Worker gets the latest parameters from the global network and resets its own network parameters.

2. Worker interacts with its environment and store transitions (state, action, reward).

3. If an episode of one worker is done, that worker calculates the determine discounted return, advantage and entropy of the policy, then calculates value loss and policy loss.

4. Worker gets gradients from losses.

5. Worker updates global network with gradients.

6. Repeats 1-5 until reach the predefined maximum global episodes.

DQN and all modified versions of it: Double DQN, Duelling DQN, Prioritized Experience Replay algorithms perfectly fit to assure the first hypothesis. Policy Gradient and Actor-critic algorithms enable us to submit the second hypothesis.

4 Experimental Evaluation

To evaluate algorithms' performance, we should test them in the actual environment, in this case, Lunar Lander game. The primary goal of the experiments is to testify if our hypotheses are true or false. More specifically we designed experiments:

- To check how fast they converge. In other words, how fast AI agents learn.
- To test how well they play the game.

Accordingly, 2 experiments were designed.

1. Each algorithm is trained in 50000 game episodes. For each episode total reward is stored as a list to check if modified versions of DQN converges faster than DQN itself.

Independent variables of the experiment:

Reinforcement learning algorithms are used to train the agent. Each algorithm is trained only once as it is a very time-consuming process.

Dependent variables of the experiment and how to measure:

- Evaluate when it gets global optimum reward first time and stays there by slightly fluctuating around this value. In Lunar Lander game maximum average reward value is about 200[1].

Constants:

- All agents are trained with the same amount of game episodes: 50,000 times.
- All agents are trained with the same environmental rules and the same scoring conditions.
- Game states are fully observable for all agents.
- All agents are trained and compared using the same computer resources.

2. After training finished, each algorithm's best state is chosen to load neural network model and played the same number of games to compare and

[1] Game leaderboard: https://github.com/openai/gym/wiki/Leaderboard#lunarlander-v2

to assure if Q-learning based and policy based combined algorithm, namely Actor-Critic gets higher scores than other algorithms.

Independent variables of the experiment:

Reinforcement learning algorithms used to train the agent. Each algorithm plays 100 games.

Dependent variable:

How good each agent plays the game. Followings are used to measure:

- Show each algorithm's 100 results in sorted order on the same plot to provide a pictorial representation of all algorithms' performance
- Calculate average, minimum and maximum reward values of 100 games for each AI agent
- Calculate standard deviation to evaluate its stability

Constants:

- All agents are trained and compared using the same computer resources.
- All agents used best training model which is taken from experiment 1.

Experiments are run using following technical resources and software packages:

- CPU: 2.5 GHz x 4
- Hardware: 128GB
- RAM: 8GB
- Linux Ubuntu 16.04 (64-bit)
- Python v.3.5.5
- Python packages:
  - Tensorflow v.1.8.0
  - Keras v.2.1.5
  - OpenAI gym v.0.10.5
  - Numpy v.1.7.1
  - Matplotlib v.2.2.2
  - Scipy 0.19.1

Training took 48 hours in the worst case for each algorithm.

5 Results

5.1 Experiment 1

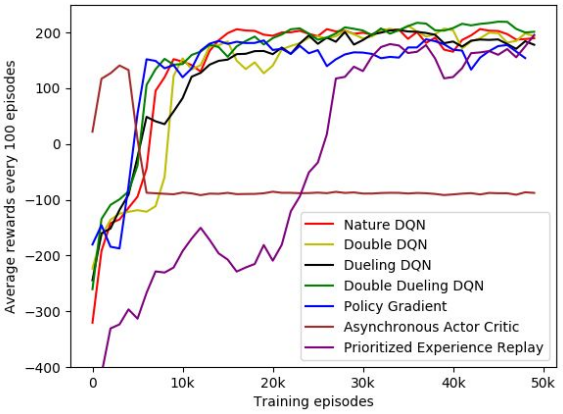

Figure 4. Reward of each algorithm through training.

DQN improved it's performance steadily and achieved global optimum after about 15000 episodes and maintained this performance throughout the rest of the experiment.

Unexpectedly, although Double DQN resolved the overestimating problem of DQN, it did not converge significantly faster than DQN. Clearly, Figure 4 demonstrates that it converged after almost 30,000 training episodes, which was much slower than DQN. Also, its performance is not very stable after it learnt how to achieve the goal. When the model was applied on the environment, we noticed that the agent implemented using Double DQN and Dueling DQN try to keep a balance and maintain a low velocity in most states. As reward is derived from shaping value which penalizes the lander's velocity and angle, it is meaningful for the agents to learn such a pattern. However, the agent from DQN behaved very erratically. It tends to take no control of velocity in most states until it is very close to the ground. Then, it uses the main power positively to slow down. As a result, it averagely consumes less power than Double DQN, which enables it to offer a better average reward. The cause of the dissimilar behavior between these two models is potentially the overestimation. Since Double DQN is more sensitive to negative reward, it quickly learned to avoid performing 'doing nothing' to be free of the negative rewards caused by high velocity. Nevertheless, DQN, which was likely affected by overestimating, did not learn such a pattern. But it even help the agent to show a better reward during the training.

As for the Dueling DQN, it showed an approximately equal performance as DQN's. It indicates that the action does considerably influence the next state, as the usage of power will directly change the lander's velocity and angle. However, the combination of Double DQN and Dueling DQN perform the best among all the variants of DQN. As shown in Figure 4, the reward of it increased the fastest in the first 10,000 training episodes. Moreover, after 50,000 training episodes, there is a noteworthy considerable gap between its performance and others'.

Prioritized Experience Replay converged much slower than other algorithms. Only after 25000 episodes, it became smart enough to get 0 reward, and after 30000 episodes it managed to get an average high score. The use of proportional prioritisation can explain it: the large capacity of the memory for samples will include useless samples meaning even if algorithm gets new samples with high priority, it still takes into account useless samples and causes underestimation. Entirely cleaning useless samples from memory takes time. One improvement would be to increase the number of episodes or lower memory size.

Policy gradient converged faster than any value-based algorithms. The reasons are as follows:

Firstly, policy-based learning improves only a little bit at the time, but it always improves in the proper direction; however, in some value-based methods like DQN, value functions sometimes may continue to oscillate around the optimal value function in the later stage and converge slower than expected. Secondly, policy gradient can learn some stochastic policies, but the value-based method is usually not able to learn stochastic policies since they tends to generate a deterministic policy. When states aliasing occurs, the stochastic policy will be better than the deterministic policy. Because the latter can only choose the same action when an agent is in any aliased states, which will cause the agent to fall into a sub-optimal strategy all the time (for greedy policy) or for a long time (for ε - greedy policy).

However, after converging, the performance of policy gradient could not keep stable as value-based method did. It proves that this algorithm has a high variance. This behaviour can also be observed in the second experiment.

The performance of Asynchronous Actor-Critic increased fastest among all the tested algorithms. It indicates that using the state value to guide policy updating will have better learning effect. Also, multiple parallel actors can help exploration. Using different exploration strategies on different threads can reduce the time correlation of empirical data. This approach does not require the experience replay in DQN to play a role in stabilising the learning process, which means that the learning process can be on-policy. However, after about 2500 episodes, its performance suddenly went down. The output actions all become "nan", so the agent behaves randomly and crashes all the time. The possible reason is that when using unbounded activation functions (e.g. Relu), the softmax function can saturate, which may lead to nan gradients when paired with categorical cross entropy cost. Due to the time constraint, this issue has been proposed to be considered in the future works.

5.2 Experiment 2

In this experiment after training is complete, each agent played 100 episodes. Their reward results are sorted and plotted (Figures 5 and 6). In these figures, perfect algorithm's dots should be on the same reward value, and it should be as high as possible. Thus, the algorithm with flatter regression line and with higher y-value can be measured as better. Statistical figures of each algorithm are also shown in table 2 for more precision. Due to the issue mentioned above, we did not show the result of Asynchronous Actor-Critic.

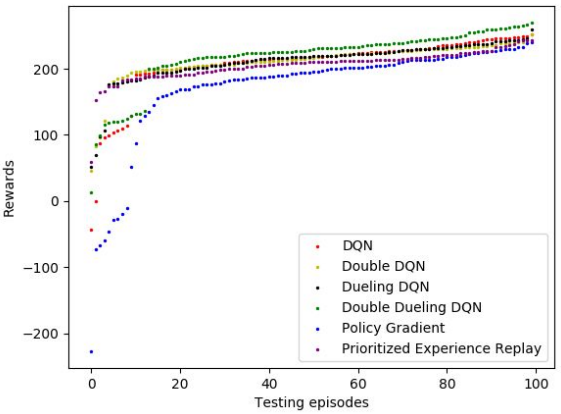

Figure 5. Sorted reward values of each algorithm

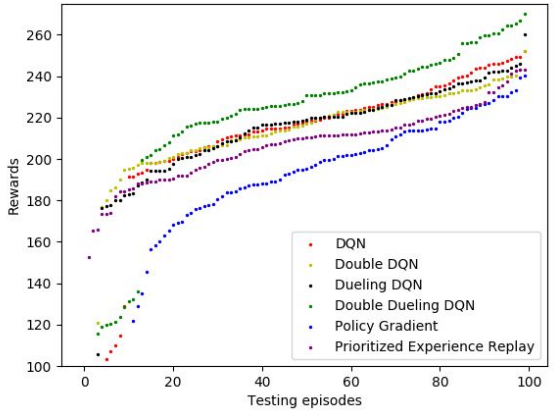

Figure 6. Sorted reward values of each algorithm (focusing on area where only few dots are missed)

Method	Min Reward	Max Reward	Average Reward	Standard Dev.
DQN	-42.57	251.85	206.91	47.73
Double DQN	46.49	252.16	211.63	30.04
Prioritized Experienc e Replay	58.85	243.27	205.67	22.91
Dueling DQN	51.01	260.11	211.43	32.48
Double Dueling DQN	13.74	270.22	217.77	45.52
Policy Gradient	-227.32	240.29	171.23	81.08

Table 2. Statistical details of the reward values.

It is shown that the average reward of both Double DQN and Dueling DQN slightly exceeds that of DQN. Also, their performance is more stable than DQN, as depicted by the smaller standard deviation. As mentioned previously, the agent of DQN learned a pattern to land with less energy consumption due to overestimation. Indeed, in the best cases, it helps the agent to get higher rewards than Double DQN, or Dueling DQN can get. The negative rewards and relatively high standard deviation indicate it suffered from a risk of cashing or landing outside the goal position as well.

In addition, by adopting Double Q Learning algorithm and Dueling neural network structure, Double Dueling DQN dramatically overwhelmed all other methods in almost all evaluation metrics. Compared to DQN, it increased the average reward by almost 5% with similar standard deviation. The positive minimum reward also proves that it is relatively more stable than DQN.

Among all algorithms, Prioritized Experience Replay showed lowest standard deviation and highest minimum reward although it's average reward were not very great. It means even though it learned slower than others as depicted in figure 4, it always guarantees successfully landing and its performance more satisfiable (figures 5 and 6). Training it in even more episodes might show better performance.

Policy Gradient had the similar maximum reward as the Prioritized Experience Replay, but it showed the lowest minimum reward and highest standard deviation. It means that the performance of Policy Gradient was volatile. In Policy Gradient, the action was chosen by sampling from the probability distributions even after training. Therefore, it could not guarantee that the agent could always choose the action with high probabilities.

For all algorithms analysing not only reward values but also storing the number of steps in each episode during the training could be an excellent material to learn algorithms' behaviour.

6 Conclusion

In this research, we analysed Q-learning based and policy-based deep reinforcement learning algorithms to test how fast they learn an environment and how well they perform after we train them in a comparatively short amount of time with lower computing resources. We explained how each algorithm implemented and why this approach was chosen in the methods section. Then in experimental evaluation section, we described goals and the design of our experiments. In the next section, we thoroughly analysed and critically explained results of each experiment.

Our results demonstrate that modified versions of DQN converged faster than DQN itself in the tested Lunar Lander environment. So the first hypothesis was correct. This observation proves two facts:

1) Even though all algorithms are published recently one after another, deep reinforcement learning field is growing gradually and they capable of performing well in most environments.

2) Later modifications of Reinforcement Learning are improving learning speed.

Actor-critic converged faster than any other algorithm taken alone, but it did not show the best result. So only first part of the second hypothesis was correct.

References

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5), 834-846.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W. (2016). *OpenAI Gym*. Retrieved from https://arxiv.org/pdf/1606.01540.pdf

Hasselt, H. V. (2010). Double Q-learning. In *Advances in Neural Information Processing Systems* (pp. 2613-2621).

Hasselt, H. V., Guez, A., & Silver, D. (2016, February). Deep Reinforcement Learning with Double Q-Learning. In *AAAI* (Vol. 16, pp. 2094-2100).

Janisch, J. (2016, November 17). Let's make a DQN: double learning and prioritized experience replay [Blog post]. Retrieved from https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

Kingma, D. P., Ba, J. L. (2017). *Adam: a Method for Stochastic Optimization*. Retrieved from https://arxiv.org/pdf/1412.6980.pdf

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). *Human-level control through deep reinforcement learning*. Retrieved from https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning* (pp. 1928-1937).

Schaul, T., Quan, J., Antonoglou, I., Silver, D. (2016). *Prioritized Experience Replay*. Retrieved from https://arxiv.org/pdf/1511.05952.pdf

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems* (pp. 1057-1063).

Tsitsiklis, J. N., Roy B. V. (1997). An Analysis of Temporal-Difference Learning with Function Approximation. IEEE Transactions on Automatic Control, vol. 42, no. 5, pp. 674-690. Retrieved from http://www.mit.edu/~jnt/Papers/J063-97-bvr-td.pdf

Thrun, S., & Schwartz, A. (1993, December). Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*.

Watkins, C. J. C. H., Dayan, P. 1992. 'Q-Learning'. *Machine learning*, *8* (3-4), pp. 279-292.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and control*, *34*(4), 286-295.

Appendix

A Individual Report – Bahriddin Abdiev

Hereby I listed all of my contributions to the project.

1. I implemented DQN and Prioritized Experience Replay algorithms.
2. I wrote sections 2-5 of research paper for algorithms related the algorithms I implemented.
3. I wrote common sections of research document like abstract, Introduction and conclusion.
4. I was responsible to format and standardize current document.

I believe other team members did great job. Lingtao Jiang designed excellent presentation. He was the one who created the best performed agents using Double DQN, Dueling DQN and combination of them. He also created amazing figures which describes our experiment results. Zhenxiang Wang designed project proposal and hypothesis. He also implemented Policy Gradient and Actor-Critic algorithms. His technical skills and knowledge about neural networks were very helpful to design and implement research. I am very thankful to them for being great team players. I learned from them a lot.

Here is my thoughts about the project:

- I learned working in a team means not only dividing responsibilities between members and keep working as an individual but rather to collaborate and develop as a whole system. At first, it seems like very slow process because we are different people with different characters. But if members are really keen to achieve common goal they will adapt to work together better and faster. Then facing and solving the problem as a team becomes easier than individual work. This experience was priceless.
- I learned more about deep reinforcement learning algorithms, obviously. During the process I also learned part of Machine Learning, Calculus, Linear Algebra and Statistics. I also figured out implementing those algorithms were not easy task as it seemed. Huge amount of time spent on tuning parameters. Also, AI algorithm implementations are more prone to bugs than normal codes because issue might be on statistical calculations or in coding. AI frameworks like Tensorflow and Keras knowledge is another thing that I gained during this experience.
- Doing research in computer science was challenging and enjoyable process at the same time. It is challenging because even designing the project proposal and hypothesis requires lot of research. To confirm correctness of hypothesis is also not always the case. Limited time is another challenging part of it. Besides, if you complete it successfully no matter what, I can't express this feeling with words. It means to be better you and contributing something to the human knowledge even if it is very tiny addition.
- Best aspect of my performance was in suggesting new ideas and improvements from the start until the very end of the project. Furthermore, I did my best to make this project report as it looks ready to publish in a conference.
- I found out I should work on improving my time management and academic writing skills.

B Individual Report – Zhenxiang Wang

My contribution in this project includes but not limited:

1. I proposed two hypotheses and designed two experiments for my team to test these hypotheses.
2. I implemented Policy gradient and Asynchronous Actor-Critic algorithms
3. I wrote section 2-5 of research paper of algorithms stated in part 1.

My teammates all did excellent job. Bahriddin implemented DQN and Prioritized Experience Replay algorithms. He also did a lot of work on the report. He wrote common sections of research document like abstract, Introduction and conclusion and formated the report to a standard style. Lingtao implemented Double DQN, Dueling DQN and the combination of them. He also created a great slides for presentation and visualized the experiment data for us to make analysis. It was my luck to work with them. They helped me a lot.

Here is my thoughts about the project:

- Teamwork gave us more chance to try different algorithms and gained more knowledge by learning from each other. If I work individually, I may not be able to go deep into so many

complex algorithms in limited time. I learned that in team work, each one need to be conscientious, responsible, and willing to help others.

- I learned a lot of knowledge about deep reinforcement learning. Because I need to combine value-based method and policy-based method, I gained a in-depth understanding of both methods. In addition, I learned how to use deep learning tools such as Tensorflow to build the model, tune the parameter, train the agent, visualize the performance and accelerate the training.
- I also learned how to do research in computer science. I learned how to make a reasonable hypothesis, how to set up baseline, how to design experiments to test the correctness of hypotheses, and how to control variables in experiments. Additionally, I learned that if you want to gain the state-of-art knowledge, the best way is to read as many related papers as possible. Another good way is to contact the author of the paper directly. They are kind to answer our questions.
- The best aspect of my performance in my team is to understand the value-based method and policy-based method then combine them together to test the second hypothesis. I am also happy that I helped my teammates to understand the basic idea and the mathematical knowledge of my algorithm so that we can compare our algorithms to make analysis.
- I regret that due to time limit, I had not solved the sudden deterioration of the performance of Asynchronous Actor Critic so that we can hand in a completed result. I use Tensorflow to implement these algorithms. It is very convenient, and many functions can be called directly, but this also makes me lack understanding of the underlying implementations of some functions, and it will be difficult to overcome the problems involved in these functions. In the future, I will do more in-depth research to explore the reasons behind the issue and learn more deep learning techniques to solve this problem.

C Individual Report – Lingtao Jiang

In this research project, I researched different deep reinforcement learning methods and developed an understanding of them. I implemented an environment program which adapts Lunar Lander game from OpenAI gym package and then generalized it so the models implemented by other team members can be trained on a unbiased environment. Among all of the methods, I was also responsible to implement several variants of DQN including Double DQN, Dueling DQN and the combination of them. Moreover, I critically analyzed the experimental results of all the models I implemented. By comparing their performance with DQN, I offered the ideas about how the models performed and why the result is this so. Also, I created the slides for presentation helped the team to make a good preparation for it.

Meanwhile, other team members also perform excently on this project. Zhenxiang proposed two hypotheses and designed two experiments. In order to test the second hypotheses, he also implemented Policy gradient and Asynchronous Actor-Critic algorithms. As for Bahriddin, he implemented the baseline DQN agent and one improvement using Prioritized Experience Replay. He also made a great effort on formatting the report.

Through this project, I gained valuable experience, including:

- A team consists of different individuals and each member has his own communication styles and different understandings about the task. It is really important to hear other's idea and provide critical feedback in group working. In such a way, everyone can understand the thoughts of each other, which can helps the team cooperated efficiently.
- The project provides a opportunity for me to build a solid understanding of several state-of-art techniques in deep reinforcement learning. Also, it is my first trying to adapt neural network in practice. I gained not only basic knowledge about neural network but also great experience about relevant skills such as parameters tuning and structure designing.
- In this project, I learnt the process to define questions by researching in related domain. As I am a student of coursework project, there is very

few opportunities for me to do researching like this project. What's more, it helps me to cultivate a habit to quickly acquire knowledge by researching.

- I showed an passion and initiatives for the project as I provided some feasible ideas such as combining Double DQN and Dueling DQN, which was proved perform the best among all methods. In addition, I helped the team to make a good preparation for project presentation
- As the the project process later than I expected, I need to improve on my time arrangement. Besides, I think I should keep the team updated about the process of everyone's task, to enable the team to have a better view of the entire task.