CS181 Final Report

Jinxi Xiao, Yuecheng Xu, Shangli Zhou, Peijun Xu, Zimu Yang
ShanghaiTech University

Abstract

Lunar Lander *is a simple and popular game, what you need to do is to control the agent to move from the top of the screen to the target location. However, due to the uncertainty of the environment, the agent may lose control and crash. Based on this situation, we decide to train an agent to eventually pass the game, with knowledge learnt in AI class.* [1]

1. Analysis

"This environment is a classic rocket trajectory optimization problem." [1] In fact, we do think that this game is an **online** trajectory optimization problem, since the agent must make some decisions explicitly, which differs from the problem of classic SLAM. Also notice that what the agent receives along the way are *states* and *rewards*. The former indicates the current situations of the agent, while the later stands for how well the agent performs. Details about action and state space of Gym [2] are shown in Figure 1

Action Space	Discrete(4)
Observation Shape	(8,)
Observation High	[1.5 1.5 5. 5. 3.14 5. 1. 1.]
Observation Low	[-1.5 -1.5 -5. -5. -3.14 -5. -0. -0.]

Figure 1. Action, State Space

Then the procedure can be summarized as follows: the agent receives some observations, and make an action based on these observations as well as experiences. Thus, the key point is to gain useful experiences as many as possible.

Then the problem becomes gaining experiences, more specific, useful experiences. As a result, one solution taught in class are obvious: **reinforcement learning**, which interacts with the environment and update its knowledge of the known space. In reinforcement learning, we store all Q-values, and in another view, we can also store these information in another form: probabilities. Recall that we have learnt Bayesian Networks in class, which is a model for joint distributions. From this perspective, we can also use Bayesian networks to represent the whole action and state space. Thus, what we need to do is to learn these Q-values or probabilities, which can be regarded as online learning.

From another point of view, in real life, we do need to gain experiences by ourselves, but sometimes we directly learn from good examples. In this certain case, in each state, if we already know what good actions are, then why not directly learn from those? In fact, we would regard this method as offline learning.

In the following sections, we will explain the details of some implemented methods.

2. Reinforcement Learning

2.1. How to represent Q-Values?

In RL, one of the most important things is to record all the Q-values. Take the Gym environment as example, the observation space could be regarded as continuous, as a result, we could not directly build a table to store all possible values. Thus the method of "enumeration" does not work. Then it reminds me of the knowledge learnt in Computer Vision: to use a Signed Distance Function(SDF) to represent the surface of a 3D object. In this context, we can also use some kind of functions to map each state to its corresponding $Q(s, a)$ with all possible actions a. A very natural idea would be MLP, for it is capable of fitting any continuous functions.

2.2. When to terminate training?

We will record the game scores of each epoch, if the average score of last 20 epochs is larger than 200, we may think of this agent as passing the game.

2.3. Deep Q-Learning

From section 2.1 we have set the training model to be a MLP, to be more specific, we use a 3-layer MLP with the following channel changes:

$$8 \to 256 \to 128 \to 4$$

Then the task would be to find a suitable training loss to supervise the process. Recall that the update function for

[1] Codes for this project are in https://github.com/xiaojxkevin/lunarlander. And we have provided more visual results on that repo

Q-values is

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')] \tag{1}$$

To make the network θ converge, what we need to do is to constrain the gap as small as possible

$$||\theta(s,a) - (R(s,a,s') + \max_{a'} \theta(s',a'))||_2^2 \tag{2}$$

Also, it is not recommended to back propagate one single sample, normally we back propagate a mini-batch. To be more specific, we create a *memory buffer* to store received training samples: *(state, action, reward, next_state, if_terminated)* with a maximum container length. And we will randomly collect samples from the *memory buffer* to compute losses and back propagate.

In sum, the algorithm is shown in Algorithm1, and training scores are shown in Figure 2

Algorithm 1 Deep Q-Learning

```
Initialize the environment and memory buffer
while Training not terminated do
    Initialize state s
    while Game not end do
        a ← f(s)
        s', r ← step(a)
        s ← s'
        Add the sample to memory buffer
        if len(memory buffer) ≥ batch size then
            randomly select a batch of samples
            compute loss and update parameters
        end if
    end while
end while
```

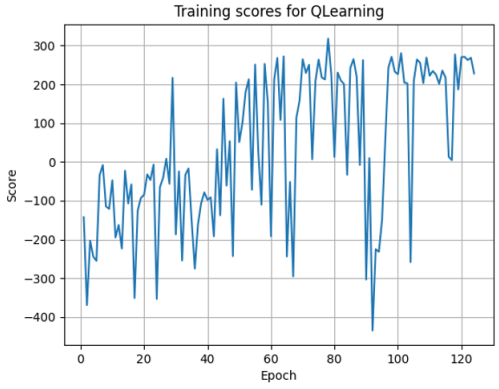

Figure 2. The training scores for Deel Q-Learning

2.4. Q-Learning With Self-designed Game

To assess the generalization capabilities of our algorithm, we explored another variant of the "Lunar Lander" game [3], which is significantly more challenging than the Gym's LunarLander-v2. Even human players find it difficult to achieve high scores in this game. If our algorithm can achieve high scores in this game as well, it indicates strong generalization capabilities.

This game has a five-action space, including left rotation, right rotation, burn gas (accelerate), stop burning gas (stop acceleration), and empty. The state space consists of seven elements: spaceship position (2), velocity (2), rotation angle, remaining fuel, and acceleration mode. The objective remains the same as the original game - safely land the spaceship at the specified location.

Several challenges arise for reinforcement learning algorithms in this game:

1. Due to the large state space, storing Q-values in a tabular format is infeasible. Thus, we must use a neural network to approximate the Q-value function.

2. During the early stages of training, reaching the specified location is challenging, resulting in mostly negative feedback. This makes it difficult for the algorithm to learn the correct strategy.

3. Learning the relationship between "burn gas" and "stop burning gas" operations proves difficult, leading to the algorithm associating only negative feedback with burning gas and avoiding this action during subsequent training.

4. The algorithm struggles to learn the relationship between spaceship angle and acceleration mode, preventing proper control of the spaceship's angle for acceleration and deceleration.

We employed the DQN algorithm, defining the game's state as spaceship position, velocity, angle, remaining fuel, and acceleration mode. These states are input into a neural network, which outputs probabilities for the five actions.

Given the game's extreme difficulty, we simplified it:

1. Lowering the initial height of the spaceship and positioning it closer horizontally to the target point reduces the state space, making training easier.

2. Transitioning from randomly generated complex maps to fixed, simpler maps stabilizes the training environment.

To encourage the algorithm to quickly learn the correct strategy, we introduced additional rewards:

1. Smaller spaceship velocities yield larger rewards.

2. Smaller spaceship angles result in larger rewards.

3. More variations in spaceship acceleration modes lead to larger rewards.

4. More remaining fuel results in larger rewards.

5. Closer proximity to the target point yields larger rewards.

6. Lower spaceship altitudes result in larger rewards.

The maximum number of steps in the game was set to 10,000. If the spaceship fails to land successfully, runs out of fuel, or exits the screen within this limit, the game is considered a failure.

While these adjustments made the game easier, the difficulty remained substantial. Our algorithm eventually converged to a local optimal strategy after 2000 epochs, adjusting the angle for free fall, but failed to find the desired global optimal strategy. Training scores are shown in Figure 3

These experiments demonstrate the good generalization capabilities of our algorithm. However, due to the game's extreme difficulty, there is still substantial room for improvement. Strategies for enhancement may include expanding the action space, incorporating fully connected layers for the state space, enabling the model to predict its own state, and establishing connections between actions and states. This approach would allow the algorithm to learn more complex strategies.

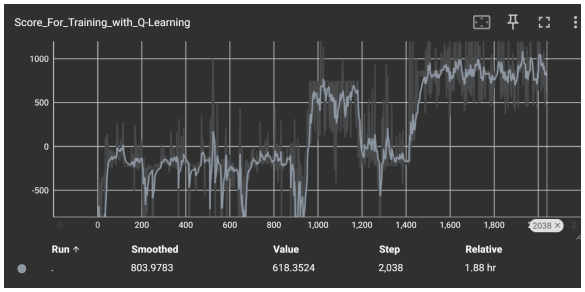

Figure 3. Training scores for self-design game with DQN. It converge to a local optimal strategy after more than 2000 epochs of training

2.5. Advantage Actor-Critic

2.5.1 Design

Method introduced in 2.3 can be viewed as a policy iteration method. And now we want to not only evaluate the policy, but also the cumulative rewards. Then we find a new method, which is named as "Actor-Critic" [4] [5]. In this very method, there are two constraints. The *actor* tries to learn the policy, whose loss functions is

$$||\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_{\phi}^{\pi}(s_t)||_1 \tag{3}$$

while the *critic* tries to estimate the expected cumulative reward, whose loss function would be

$$\sum_{t=1}^{T} -\log \pi_\theta(a_t|s_t)((\sum_{t'=t}^{T} \gamma^{t'-t} r(s_t, a_t)) - \hat{V}_{\phi}^{\pi}(s_t)) \tag{4}$$

As for the network design, we choose a shared network, which is shown in Figure 4. And to be more specific, we use a encoder to map 8 observations into a 128 dim vector, and the action_layer maps the 128 dim vector to a 4 dim action vector; while the value_layer maps the 128 dim vector to a scalar [6].

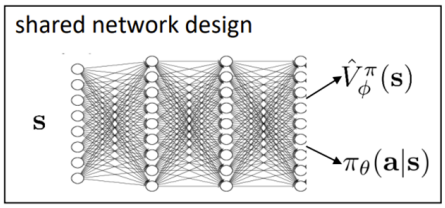

Figure 4. Network Design For A2C

Notice that our loss function requires a full time series, thus we will compute loss and back propagate at the end of every epoch. In sum, our algorithm is similar to algorithm 1, only when and how to compute loss are different. In addition, training scores are shown in Figure 5.

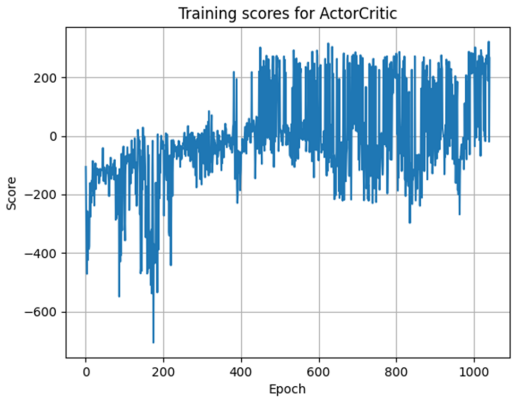

Figure 5. Training scores for A2C

From the Training curve we have found that this model is unstable: after 800 epochs it still gets scores less than 0; and it converges slowly. Based on situation, we think that

it might be the case that we always sample from the action-distribution, and there's no explicit stage of exploration or exploitation. As a result, we decide to add this part into the network. To be more specific, we take advantage of the sum of entropy of all the returned probabilities, which is calculated as

$$\sum_{t=1}^{T}\sum_{i=1}^{4} -P(a_i|s_t)\log P(a_i|s_t) \tag{5}$$

2.5.2 Try Exploration At the Beginning

Since we have calculated the entropy, then at the beginning phase, like the first 100 epochs, if the score of the game is less than 0, we will add the reciprocal of the entropy with a weighted scalar w to the loss. Since we want to constraint the entropy to be large at the beginning so that the agent can explore more. And due to the optimization method is gradient decent, we use the reciprocal of the entropy.

However, the results are even more unstable, I run this method two times and record all training scores, which is shown in Figure 6. We can clearly see that the model can converge within 500 epochs, and it can also converge even slower than the origin version. We think that it might be the case of the reciprocal, since it is not linear, which is hard to optimize.

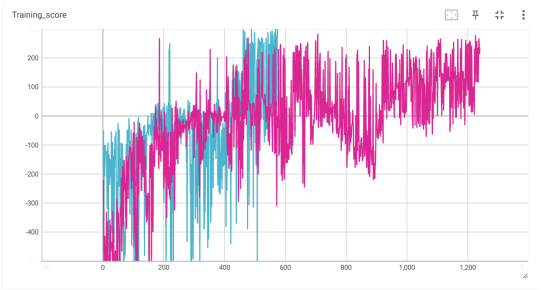

Figure 6. Training scores for A2C with Exploration

2.5.3 Try Exploitation Later in the Game

After 300 epochs, if the score of one game is larger than 0, we will add the entropy with a weight w to the loss. This constraint the probabilities estimated by the network to be sharp, which means sticking to the current best action. However, we have encountered a similar problem in exploration, which is that the whole system gets unstable with a high variance. Two training processes are shown in Figure 7.

2.5.4 Conclusion

Based on the poor performance of exploration and exploitation, we have one assumption to explain this phenomenon.

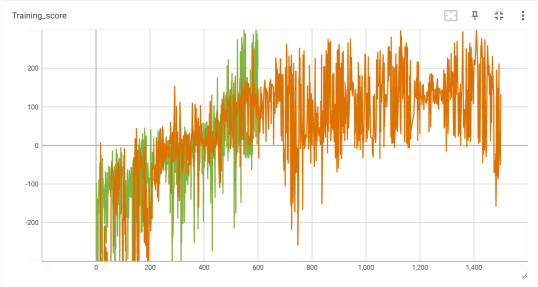

Figure 7. Training scores for A2C with Exploitation. The orange one does not converge, for the maximum epoch is set to be 1500

It is that forcing entropy to be large or small way greatly affect the decision-making process of A2C. These constraints force the model to try certain actions. On some cases, these actions are extremely useful, then the model tends to learn quicker; on the other hand, some actions are bad, which lead the model to a sub-optimal point and results in a slower convergence or even no convergence.

3. Bayesian Online Learning

3.1. Bayesian Network

In this project, we similarly hope to use probabilistic modeling to find the optimal landing strategy for the spaceship. First, we tried to fit it through a simple bayes network. We treated the flight altitude in three cases ($Height > 1.0$, $0.5 < Height < 1.0$ and $Height < 0.5$) and provided CPT for each case as shown in Table 1. Given the action of the previous step and a known flight altitude, we hope to find the optimal action for each step by searching the CPT to find the action with max probability for the next step. Therefore, we needed to get the correct CPT of each altitude condition for action prediction and since this task is an online learning category, we needed to update the initial CPT value by searching and experiencing new samples to get the accurate probability model.

$P(A\_n\|height, A\_p)$ previous action / next action	$A_{1,h}$	$A_{2,h}$	$A_{3,h}$	$A_{4,h}$
$A_{1,h}$	$P(A_{1,h}\|A_{1,h})$	$P(A_{2,h}\|A_{1,h})$	$P(A_{3,h}\|A_{1,h})$	$P(A_{4,h}\|A_{1,h})$
$A_{2,h}$	$P(A_{1,h}\|A_{2,h})$	$P(A_{2,h}\|A_{2,h})$	$P(A_{3,h}\|A_{2,h})$	$P(A_{4,h}\|A_{2,h})$
$A_{3,h}$	$P(A_{1,h}\|A_{3,h})$	$P(A_{2,h}\|A_{3,h})$	$P(A_{3,h}\|A_{3,h})$	$P(A_{4,h}\|A_{3,h})$
$A_{4,h}$	$P(A_{1,h}\|A_{4,h})$	$P(A_{2,h}\|A_{4,h})$	$P(A_{3,h}\|A_{4,h})$	$P(A_{4,h}\|A_{4,h})$

Table 1. CPT given Height and previous action

The whole updating algorithm is shown in Algorithm2, where we simulated the way of updating through samples in reinforcement learning. We set different learning rates at different flight altitudes (Because when the altitude is low enough, it shows that the ship needs to prepare for landing. We need a strong weight to make the probability of an action that leads to successful landing extremely large, as a way to avoid choosing other actions that could lead to a

crash and make the game fail. But we also needed to consider that the rewards returned in the low-altitude state always has extreme large or small values (winning or crashed) so that extreme high weight may also cause the probability increase or decrease too much to become negative or much larger than 1. We needed to find a balance.) to update the probabilities of the selected action and its opposite action by using the reward received after taking the selected action based on the action in previous step.

Algorithm 2 Bayesian Network Update

Initialize the environment and memory buffer
Initialize state $CPT \quad lr \quad A \rightarrow$ action
for $Epoch = 1, 2, ..., N$ **do**
  Initialize $A^0 = 0$
  for $t = 1, 2, ..., MAXITER$ **do**
    judge height $h = i$
    $A^{t+1} = \underset{j \in \{1,2,3,4\}}{\arg\max} \ P_{j|A^t,h}$
    $reward, terminated = update(A^{t+1})$
    $P_{A^{t+1}|A^t,h} = P_{A^{t+1}|A^t,h} + lr_h \times reward$
    $P_{A^{t+1}_{opposite}|A^t,h} = P_{A^{t+1}|A^t,h} - lr_h \times reward$
    if Game Terminated **then**
      break
    end if
  end for
  $CPT^{epoch+1} = \frac{0.95 \times CPT^{epoch} + 0.05 \times CPT^{epoch+1}}{Z}$
  Add the sample to memory buffer
  if $Epoch \bmod 1000 = 0$ **then**
    UPdate Learning rate
  end if
end for

However, after testing, the above algorithm didn't show desirable results (shown in Fig 8), with less than 50 successful landings out of 100,000 epochs. After analyzing, we believed that since the presence of gravity is considered in that game, we needed to consider the effect of gravitational acceleration, and if we only considered a fast descent when the flight altitude was very high, then in low flight altitude phase, it may not be possible to reduce the speed below the value for collision avoidance even if we keep taking deceleration action. Based on the above considerations, we thought that we should not only rely on the reward of the next action to update the CPT, but should also consider the whole action series, from the start to the end, so as to ensure the accuracy of the results. However, because there are so many action combinations that we couldn't list them all, we would like to incorporate the Bayesian parameters into the neural network and output the probability directly through the neural network.

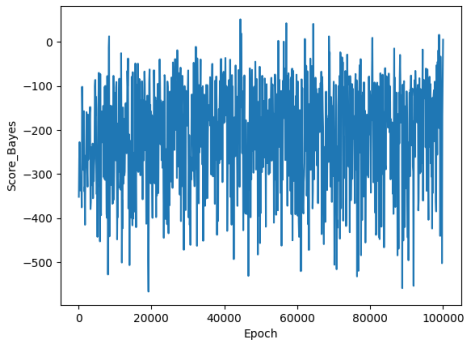

Figure 8. Result of BN

3.2. Bayesian with Neural Network

The whole network architecture is shown in Fig 9. We used a three-layer fully connected layers (a ReLU active layer between each two full-connected layers) and a Sigmoid active layer to output an initial probability and then do normalization by Softmax function to get the final probability. Our input is the action value and 8 observation values obtained after taking the action, and the output is the probability of each of the 4 action options that can be taken in the next step, i.e. the input is a 9-dimensional vector and the output is a 4-dimensional vector. Our loss function is related to the rewards, which are defined as shown in Fig 10. Since action selections in low altitude phase is more important considering they directly influence the final landing, we use discounted reward (Fig 11) to optimize the network so that the weight of the rewards that are advanced in the whole process decreases exponentially as the game progresses.

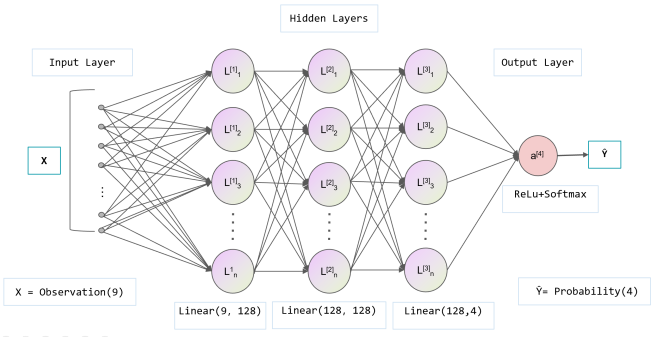

Figure 9. Network Structure

$$Loss = \text{Expected Optimal Reward} - \text{Total Discount Reward}$$

Figure 10. Loss

Calculate discount reward

input Reward ListProb List

Initialize $total = 0, discount = 0.9$

1: **for** $t = 1, 2, ..., ITERs$ do

3: $temp = temp + Reward[t] \times Prob[t]$

5: $total = temp + discount * total$

6: **end for**

Return $total$

Figure 11. Discounted Reward

By combining with neural network did make the score have a certain degree of improvement for some epochs (shown in Fig 12). However, its success rate was still very low. From our analysis, we found that because the game space is continuous, the number of combinations between variables is extremely large, and we couldn't rewind time to get more samples when updating. Therefore, the number of updates can't match the huge conditional space which make it extremely difficult to get the optimal solution through Online Learning. The probabilistic relationship and optimization in Bayesian online Learning needs further discussion.

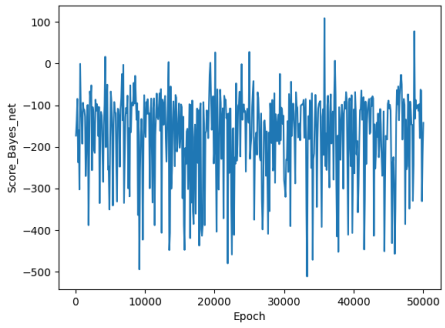

Figure 12. Result of Bayesian with Neural Network

4. Offline Learning

4.1. Sampling

Since the final results obtained through the online learning described above all have a large variance, we would like to exploit the better results produced in the online learning as a way to obtain a more stable optimizer by Offline Learning. The first step we need to do was to sample the data we wanted in the model obtained from Online Learning as training data and ground truth. A single data point is related to one step, consisting of the action A_t we took, observation after we took action A_t and the next action A_{t+1} we predicted. We took the game result $score >= 200$ as the standard, sampled a total of 100 rounds of games, including a total of about 30,000 steps. The structure of each data point is shown in the Fig 13 and Fig 14.

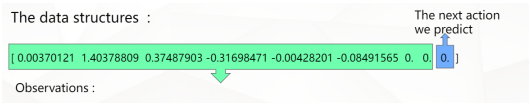

Figure 13. data point for MLP

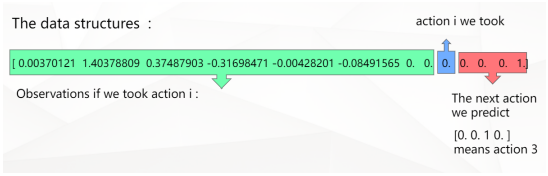

Figure 14. data point for BNN

4.2. Basic MLP

We first chose to use the basic MLP for supervised learning to help us obtain better models, and our network architecture is shown in Fig 15. It consists of a 4-layer fully-connected network (a ReLU active layer between each two full-connected layers), with the inputs being the values of observations (8-dimensional) and the final output being a 4-dimensional vector representing the probability of each of the 4 action options that can be taken in the next step. We chose to use cross entropy as our loss function. After 1500 iterations with a dataset of 30,000 data points, we tested our model for 100 epochs and obtained the following results: Fig 16

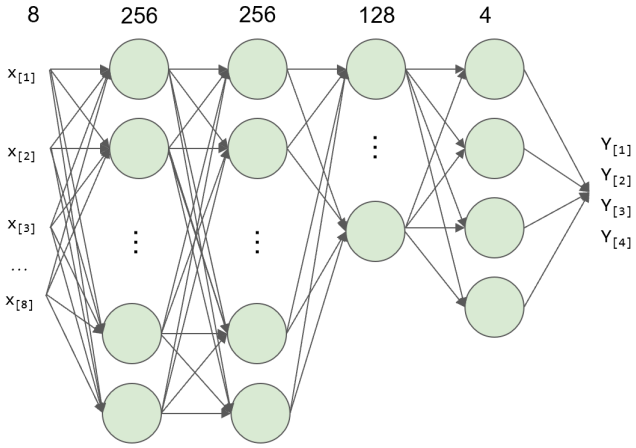

Figure 15. Network Structure

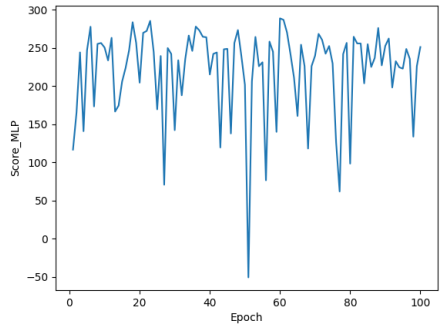

Figure 16. Test from Basic MLP

4.3. BNN

We found that since the game environment is initialized by random seed and the final landing place was also determined by random seed, we wanted to have more flexibility in the network parameters, so we chose to use Bayesian Neural Network to obtain the better prediction model. The main difference between Bayesian Neural Network and ordinary Neural Network is that network parameters in BNN are not fixed, but are generated by different probability distributions. Thus, the optimization object is also changed from the direct parameters in the network to the probability distributions that generates them. The basic principle of Bayesian Neural Network is that we hope to use a simple distribution (we assume a Gaussian distribution) $P(w|\mathcal{N})$ to fit the posterior distribution of the parameters given data points $P(w|\mathcal{D})$. We use KL divergence to describe distance of the two items above so that our target function is:

$$F(D,\mathcal{N}) = \arg\min_{\mathcal{N}} KL[q(w|\mathcal{N})||P(w|\mathcal{D})] \tag{6}$$

$$= \arg\min_{\mathcal{N}} \int q(w|\mathcal{N}) \log \frac{q(w|\mathcal{N})P(\mathcal{D})}{P(w)P(\mathcal{D}|w)} dw \tag{7}$$

$$= \arg\min_{\mathcal{N}} \int q(w|\mathcal{N}) \log \frac{q(w|\mathcal{N})}{P(w)P(\mathcal{D}|w)} dw \tag{8}$$

$$= \arg\min_{\mathcal{N}} KL[q(w|\mathcal{N})||P(w)] - E_q[\log P(\mathcal{D}|w)] \tag{9}$$

$$= \mathcal{L}_1 - \mathcal{L}_2 \tag{10}$$

In this task, we optimized by replacing $\mathcal{L}_2$ with MSE Loss between prediction action and ground truth because MSE Loss performances better in our model than other loss function such as cross entropy. We referenced the BNN model in Github Repository [7] and the input is a 9-dimensional vector consisting of the action we took and the observation after we took the action. The output vector is also a 4-dimentional vector representing the probability of each of the 4 action options that can be taken in the next step. We also trained our model for 1500 epochs with 30000 data points. We tested our model for 100 epochs and the result is shown in Fig 17.

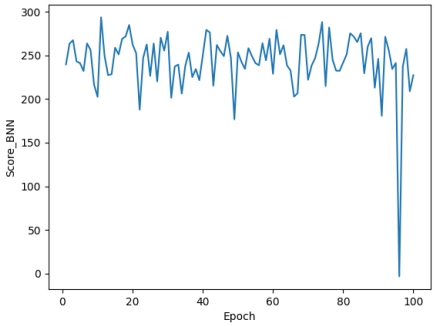

Figure 17. Test from BNN

5. Test Results

We show test results of all methods in Table 2

Test on 100 Epochs on Gym				
	mean	std	max	min
DeepQL	225.14	62.88	299.08	-91.01
A2C	90.56	50.90	196.76	-89.97
Off_MLP	220.98	60.04	288.84	-50.62
Off_BNN	239.16	39.08	289.34	-3.77

Table 2. Test Results for all methods

6. Conclusions

As a game with long history, Lunar Lander and many other similar optimization problems have some standard solutions. Methods of AI are a small part solution. Nevertheless, we have learnt a lot when doing this project, it is great to make use of some learnt methods working successfully on a simple game. Among all the methods, we can see that offline methods achieve a general high scores, which stress the importance of good samples. While online learning works not that well but is able to work when there's no sufficient data, which is crucial in many cases. What's more, it is really kind that TAs provide a nice web that stores a number of games that could be implemented with AI methods, which inspire us to keep digging into this field.

7. Resources that we use

1. **Gym**: used as the environment for development

2. **Numpy**: to load data and do some simple matrix operations
3. **Pytorch**: used for deep learning
4. **Pillow**: used to import and export images
5. **matplotlib**: used to plot images
6. **argparse**: used to add options
7. **tensorboardX**: keep logs when training
8. **collections**: use deque as the memory buffer
9. **os**: Employed for file saving and loading (images, checkpoints, etc.).
10. **time**: utilized to obtain the current time for creating file paths.
11. **random**: generating pseudo-random numbers.
12. **math**: use to calculate fundamental number such as sin, cos,π
13. **pgzrun**: use to test and visualize our game, which helps the following algorithm design

References

[1] "gym." https://www.gymlibrary.dev/environments/box2d/lunar_lander/.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[3] "Lunar lander game." https://aayala4.github.io/Lunar-Lander-Python/.

[4] "a2c-blog." https://blog.csdn.net/qq_33302004/article/details/115530428.

[5] "a2c-slide." https://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_5_actor_critic_pdf.pdf.

[6] "a2c reference." https://github.com/nikhilbarhate99/Actor-Critic-PyTorch.

[7] "Bayesian-inverse-reinforcement-learning." https://github.com/reedsogabe/Bayesian-Inverse-Reinforcement-Learning.