CS7642 Project 2 Report: Lunar Lander

*github hash: fe9ccb68ff1a37861fa215ec2c2f0529cfb2bed4

Shide Qiu
dept. of Computer Science
Georgia Institute of Technology
sqiu74@gatech.edu

***Abstract*— This project report covered the fundamentals of Q learning and the detailed implementation of Deep Q-Network (DQN) found in Minh's paper [1]. The Lunar Lander v2 environment from OpenAI Gym was solved (achieve at least 200 points over 100 consecutive runs) in this report. The effect of hyperparameters of DQN algorithm were discussed.**

I. Introduction

In this project, an agent was trained to successfully land a lunar lander in "LunarLander-v2" environment from OpenAI Gym. The environment has 8-dimensional state space with 6 continuous state variables and 2 discrete ones which can be represented as $(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R)$. x, y are the horizontal and vertical position with $\dot{x}, \dot{y}$ representing speeds on these two directions. $\theta, \dot{\theta}$ are the angle and angular speed of the lander. leg_L, leg_R are binary values representing whether the lander's left or right leg is touching the ground. Four actions (do nothing, fire left engine, fire right engine, fire main engine) are available to the lander. The goal is to land at the landing pad which is always at coordinates (0,0). The base reward is between 100 and 140 points from the top of the screen to the landing pad. Lander moving away from landing pad is allowed with penalty. The episode terminates if the lander crashes or comes to the rest with additional -100 or +100 points, respectively. +10 points are given for each leg-ground contact. -0.3 point is incurred for firing main engine and -0.03 point for firing orientational engines. The problem is considered solved by achieving a score of 200 points on average over 100 consecutive episodes.

II. Algorithm

A. Q-learning

Q-learning, an off-policy learning algorithm, is one of the early breakthroughs in reinforcement learning which is defined as [2]: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)$. It learns the action-value function Q and directly approximates the optimal action-value q_*. This simplifies the analysis because it is independent of the policy being followed. In other words, it estimates the return for state-action pairs assuming a greedy policy is always followed despite the fact that it may not follow a greedy policy. A Q-value lookup table is used in Q-learning for value function representation. However, it has limitations for large MDP problem like "Lunarlander-v2". There are too many states and actions need to be stored in memory and it is too slow to learn the value of each state individually. Function approximation is introduced in next section to solve this issue.

B. Value Function Approximation

In function approximation, the approximate value function is not represented as a table but as a parameterized functional form with weight vector $w \in \mathbb{R}^d$. The approximate value of state s given weight vector w is written as $\hat{v}(s, w) \approx v_\pi(s)$. Typically, changing one weight changes the estimated value of many states. Thus, such generalization makes the learning more powerful for complex MDP problems [2]. According to Sutton's book, the objective of function approximation is to minimize the *Mean Squared Value Error*:

$$\overline{VE}(w) \doteq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2,$$

where $\mu(s)$ represents how much we care about the error in each state s; $v_\pi(s)$ and $\hat{v}(s, w)$ represents true value and approximate value at state s. However, we do not know the true value in real word problem. Thus, we substitute a target for $v_\pi(s)$.

C. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) methods are among the most widely used of all function approximation methods [2]. It minimizes the $\overline{VE}$ by adjusting the weight vector after each example in the direction that would reduce the error most:

$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S, w))\nabla_w \hat{v}(S, w),$$

where α is a positive step-size parameter.

D. Nonlinear Function Approximation

Artificial neural network (ANN) is widely used for nonlinear function approximation [2]. Here, a generic feedforward ANN with 8 input units (state), 4 output units (action), and 2 hidden layers is used in this report. The algorithm performance depends on the hidden layer dimensions and learning rate which are discussed in the experiment section. The activation function used for ANN is "relu" provided in Keras. The main advantage of ANN over linear function approximation is that it can automatically create appropriate features for our problem without relying on hand-crafted features.

However, reinforcement learning is unstable when a nonlinear function approximation (like ANN) is used due to the correlations in the sequence of observations and correlations between the action-values and the target values. This issue is solved with a DQN in Mnih's paper [1].

E. DQN Algorithm

Two key ideas are used to address the instability mentioned above. First, experience replay [3] decorrelates the trajectories by randomizing over the data. The agent's experience at each time-step t are stored in the replay memory D with a buffer size. Then samples (batches) of experience are drawn uniformly at random from the replay memory for learning. Another advantage of experience replay is that each step of experience is used in many weight updates which offers better data efficiency. Second, we keep a separate target network which is only updated periodically (every 4 steps in this report) to reduce the correlations with the target.

The Q-learning update at iteration i uses the following loss function:

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

where γ is the discount factor, w_i is the parameter of the Q-network at iteration i and $w_i{}^-$ is the network parameters used to compute the target at iteration i. $w_i{}^-$ is only updated with the Q-network parameter every C steps (4 steps in this work) and are held fixed between individual updates. Moreover, the soft target update for target Q-network is used here rather than directly copying the weights [4]. The update rule is :

$$w^{\hat{Q}} = \tau w^{Q} + (1 - \tau) w^{\hat{Q}},$$

where τ is the soft target update parameter. The target values are constrained to change slowly and thus greatly improves the stability of learning.

```
DQN Algorithm
Input: "LunarLander-v2" gym environment
Init replay memory D to capacity N
Init action-value function $Q$ with random weights $w$
Init target action-value function $\hat{Q}$ with weights $w^- = w$
Init $\varepsilon \leftarrow 1$
for episode = 1, train_episodes do
  reset game
  for step = 1, max_steps do
    Select action $a_t$ with $\varepsilon$-greedy policy
    Execute the action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1}, done_t)$ in D
    Sample a minibatch of transitions from D
    loop for each $(s_j, a_j, r_j, s_{j+1}, done_j)$ in minibatch
      if done then
        $y_j = r_j$
      else
        $y_j = r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; w^-)$
    Perform gradient descent step with respect to network parameter $w$
    Reset $w^{\hat{Q}} = \tau w^{Q} + (1 - \tau) w^{\hat{Q}}$ every C steps
  end
end
```

The DQN algorithm implemented in this work is shown above. The replay memory update rule is FIFO. The update rule for ε is : $\varepsilon = \varepsilon_{final} + (\varepsilon - \varepsilon_{final}) e^{-decay_{rate} * episode}$, where $\varepsilon_{final} = 0.01$. Under this update rule, ε will decay faster as the number of training episodes goes up.

III. Experiment and Results

The experiment is composed of two parts: training and testing. In the training section, the agent will run 2000 episodes or terminate early if it reaches an average score of 220 over 100 consecutive episodes. This termination criterion is more stringent than our test goal to make sure our agent can survive in the random test section. For each episode, the agent will take an action with ε-greedy policy. The environment will receive an action and return the reward and next state. We store the transition experience as $(s_t, a_t, r_t, s_{t+1}, done_t)$ in replay memory with maximum size of 100000. The replay memory updates according to the FIFO rule when it reaches its capacity. A batch of samples are sampled randomly from replay memory for Q-learning. For Q-learning update, we clone the Q-network with soft target update rule to obtain a target Q-network every C updates. The target Q-network is used for generating the Q-learning targets for the following C updates to Q-network.

The episode will terminate when it receives +100 points for successful landing or -100 points for crash. Moreover, it will automatically terminate when it reaches the max steps in each episode which is 1000 steps in this work. Once an episode is done, the ε is updated with the rule defined in last section. ε will not update when it reaches 0.01. In the test section, the trained model is directly used to predict Q-value for 100 episodes. The goal is to reach an average of 200 points in 100 runs. The optimal agent takes 50 minutes to train. The detailed is discussed in the following section.

Moreover, the effects of hyperparameters, including neural network learning rate, discount factor γ, ε decay rate, batch size, and hidden layer dimensions, are studied due to their high impact to the quality and efficiency of model training. All the figures for hyperparameters studies are using moving average over 100 episodes (the current episode and the 99 previous ones) of the reward. This is due to the high variance and overlay between different hyperparameters models make the figures not readable. We will not have the score of first 99 episodes (they do not have 99 previous episodes to average) in the figures but it does not influence our analysis.

A. Optimal Agent

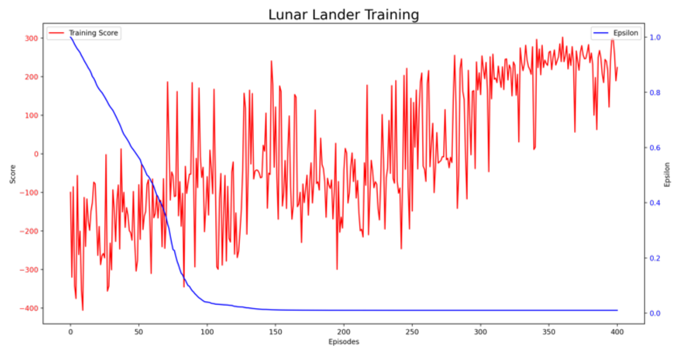

Fig. 1. Score and ε over Lunar Lander training episodes under DQN

As shown in Fig. 1, the optimal agent reaches an average score of 220 over 100 consecutive episodes under 400 training episodes. The ε decays to minimum (0.01) at around 150 episodes. The optimal parameters used are shown in Table 1. The score gets more stable (less variation) near 400 episodes. The reason could be the soft target update implemented in the algorithm. However, there are still cases where score is below 100 after 300 episodes training. It is because the "LunarLander-v2" environment is complicated and its states are continuous which means our model cannot learn all possible states within 400 episodes. There must be some states that the model never encounter before and perform not well on it. To solve this, set a more stringent termination criterion and train more episodes may help. Moreover, our current termination criterion only cares about the average score over last 100 episodes. It does not consider the score of each single episode. We could add the termination criterion "stop if there is no score under 50 points over last 100 episodes" to improve the performance. However, this will require more computational power.

Table 1

Replay memory size N = 100000
Neural network hidden size = 64×64
ANN learning rate: 0.0005
Soft target update: $\tau = 0.03$
Discount factor: $\gamma = 0.999$
Initial ε: $\varepsilon_{initial} = 1$
Final ε: $\varepsilon_{final} = 0.01$
ε decay rate: 0.0001
Batch size: 32
Q target update frequency: C = 4

The test result is shown in Fig. 2. The agent reaches an average of 235.50 points in 100 consecutive runs. However, it has similar issues discussed above. Even if it can perform well on average over 100 episodes, there are still possible cases where score is negative. The possible solutions like modification of termination criterion could be studied in the future work.

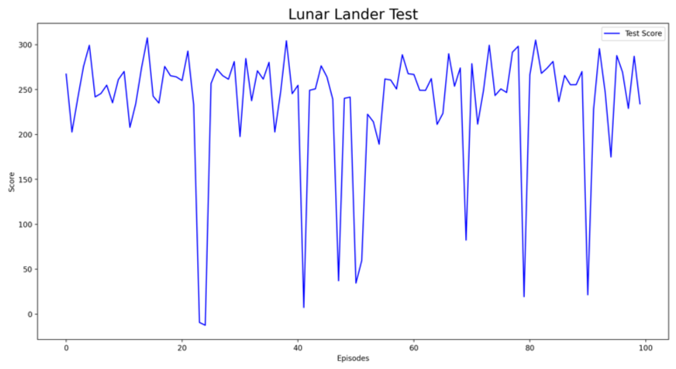

Fig. 2. Testing optimal agent on 100 consecutive episodes

B. *Learning Rate*

Learning rate, used in Adam optimization of neural network, is a key hyperparameter for DQN. It decides how much weights in each pass get updated to the direction that minimize the current error the most. Lower learning rate guarantee the convergence to global optimum while it takes longer computation time. Higher learning rate reduce the computation time by updating more aggressively but gradient descent can overshoot the minimum and it may fail to converge or even diverge. Thus, it is important to find the balance between the training speed and the convergence to global optimum.

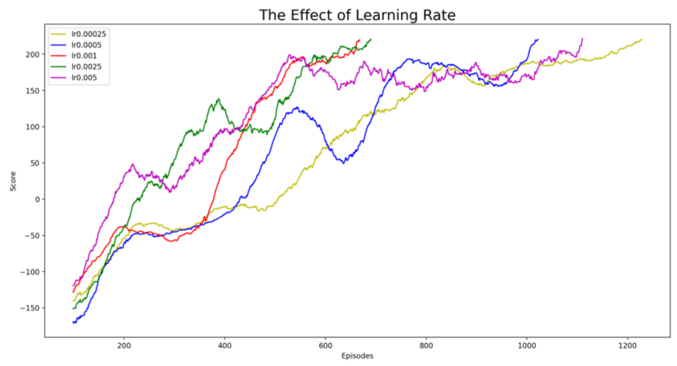

Fig. 3. The effect of learning rate on training speed and convergence

Here, we use the same experiment setup with optimal parameter (except $\gamma = 0.99$) and adjust the learning rate to assess its influence on the convergence. 5 learning rates (0.00025, 0.0005, 0.001, 0.0025, 0.005) are studied and the result is shown in Fig. 3. All agents converge to the target under 1300 episodes. As expected, the lowest learning rate 0.00025 takes the longest time to converge. As learning rate increases, it takes fewer episodes to converge due to more aggressive update. The optimal learning rate is around 0.001-0.0025 as shown in the figure which only take around 650 training episodes to converge. When the learning rate gets larger than 0.0025 like 0.005, it takes more time to converge due to possible divergence.

C. *Gamma*

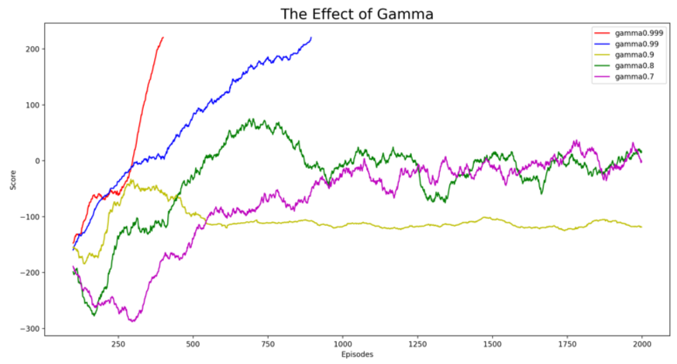

Fig. 4. The effect of gamma on training speed and convergence

Discount factor $0 \leq \gamma \leq 1$ determines the present value of future reward. γ close to 0 leads to "myopic" evaluation which is only concerned with maximizing immediate rewards. γ close to 1 leads to "far-sighted" evaluation which takes future rewards into account more strongly.

To study the effect of γ, the same experiment setup with optimal parameters is used and 5 different γ (0.7, 0.8, 0.9, 0.99, 0.999) values are trained as shown in Fig. 4. Small γ (0.7, 0.8, 0.9) fail to converge under 2000 training episodes because it does not consider enough future rewards. γ =0.999 is the parameter used in the optimal agent discussed before which converge within 400 training episodes.

D. The Effect of ε Decay Rate

The ε decay rate is the key of ε greedy policy which determines the action selection in DQN algorithm. It is used to balance the exploration and exploitation. The ε update rule is introduced in Section II. The higher decay rate reduces ε faster to 0.01 which means the agent switch to exploitation from exploration faster. However, it may fail to converge due to not enough exploration. The lower decay rate agent explore more states before exploitation, and it will take longer time to converge.

The same experiment setup with optimal parameters (except γ=0.99) is used. As shown in Fig. 5 and Fig. 6, the ε is not decayed as expected with its decay rate. I double check the code and it turns out there is a typo in ε update rule. The current episode number is supposed to put into the update rule $\varepsilon = \varepsilon_{final} + (\varepsilon - \varepsilon_{final})e^{-decay_{rate}*episode}$. However, the step number in the episode is put instead by accident. Since I do not have enough time to run the model again with correct update rule, I leave the original figures here. More importantly, even if we don't get the update rule work as expected, it still offers us some insights because we do have 4 agents with different ε decay rate as shown in Fig. 6. The agent with lower ε decay rate (yellow and magenta lines) takes longer time to converge. On the other hand, the agent with highest ε decay rate (red line) converge slower than blue line because it does not explore enough states before exploitation.

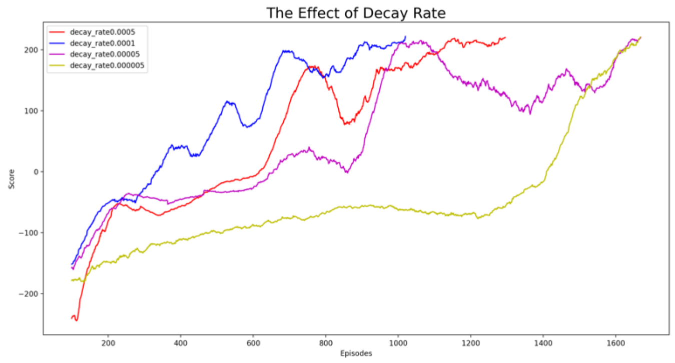

Fig. 5. The effect of ε decay rate on training speed and convergence

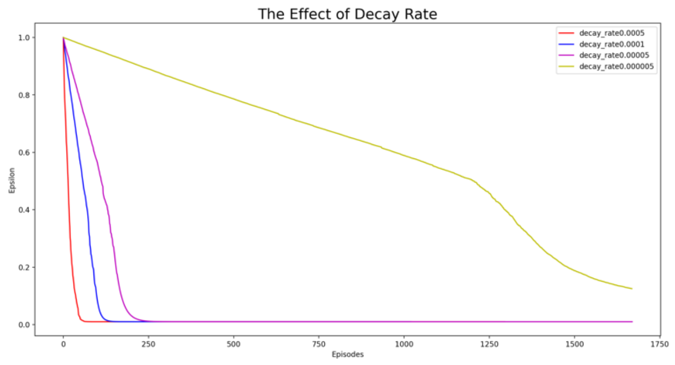

Fig. 6. ε over training episodes with different decay rate

E. The Effect of Batch Size

The batch size determines how many transitions are sampled from replay memory every C steps. Larger batch size will improve the training efficiency significantly while taking more computation time. Smaller batch size will run faster but it may not be enough to remove the correlations in the observation sequences.

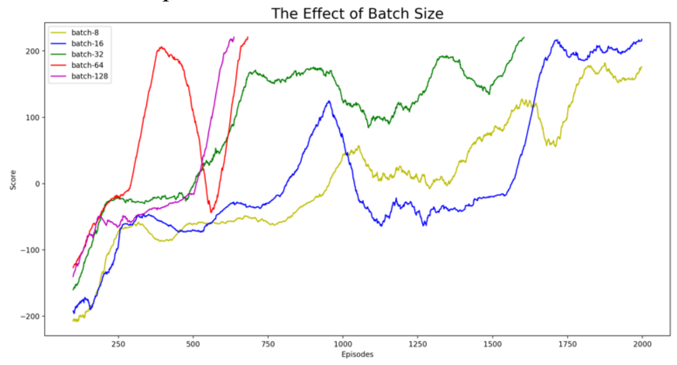

Fig. 7. The effect of batch size on training speed and convergence

5 different batch sizes (8, 16, 32, 64, 128) are studied here. The same experiment setup with optimal parameters (except γ=0.99) is used. As shown in Fig. 7, smaller batch size (8, 16) fail to converge. As batch size increases, it converges to target with fewer training episodes. However, it takes more time to train each episode with larger batch size. For example, it takes 1.75 hours to run 600 episodes with batch size of 32 while 3.75 hours to run 600 episodes with batch size of 128.

F. The Effect of Hidden Layer

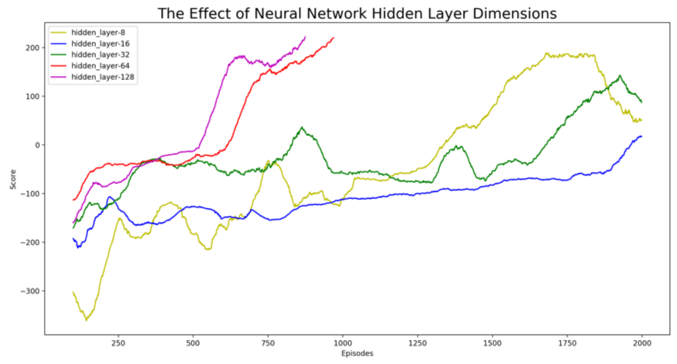

Fig. 8. The effect of hidden layer dimensions on training speed and convergence

Hidden layer dimensions is critical for the performance of ANN. Higher dimensions may cause overfitting while lower dimensions may not be enough to represent all the appropriate features. Thus, the balance between overfitting and convergence to global optimum is essential.

We use two hidden layers with 5 different layer dimensions (8, 16, 32, 64, 128) in this work to study how it impacts the training speed and convergence. The same experiment setup with optimal parameters (except γ=0.99) is used. As shown in Fig. 8, agent with fewer nodes (8, 16, 32) cannot converge to the target under 2000 training episodes due to underfitting. The agent with higher hidden layer dimensions (64, 128) converges to target under 1000 training episodes. We do not see the overfitting condition here. However, we may see it happened with 256 or 512 hidden layer dimensions. Moreover, with the higher hidden layer dimension, it demands more computational power.

IV. Conclusions

In this project, a DQN algorithm with soft target update is implemented to train the agent for "LunarLander-v2" environment. The impact of 5 different hyperparameters is discussed and the optimal agent model is found. The agent can complete the project goal which is reaching an average of 200 points over 100 consecutive episodes.

A. *Pitfalls*

The computation power is the main limit for tuning hyperparameters. It takes me 2-6 hours to train one model with 2000 training episodes which makes it impossible to study all possible hyperparameter combinations.

The "LunarLander-v2" environment is complicated and requires long time for me to understand if my algorithm works or not. A good solution is to use the same algorithm on the easier environment like "Cartpole" which will reduce training time significantly. Apply the algorithm to our project environment after confirming it works with the easier environment.

The soft target update parameters (0.0005, 0.001, 0.003, 0.01, 0.025) are studied. However, it does not give any promising results. τ-0.0005 and τ-0.025 converge at around 1100 training episodes while τ-0.01 does not converge over 2000 training episodes. The reason for this is not quite clear and more study is needed to understand how the soft target update parameter will impact the training efficiency and convergence. The Jupiter notebook for τ analysis is included in the github.

There are so many parameters which makes it easy to mess up with their names. For example, I messed up with the ε decay update rule and did not get expected decay rate. Luckily, the result is still be able to help us understand the effect of ε decay rate on the training speed and convergence.

B. *Future Work*

First, the correct ε decay rate update should be implemented, and the model need to be trained and discussed again.

The optimal agent model found can still be improved. We select it as the best agent model because it requires the fewest training episodes to reach an average of 220 points over last 100 consecutive training episodes in all training models. However, this model is the optimal model with best γ. All the other parameters for $\gamma = 0.999$ model is not studied here due to high computation cost. Also, there might be other hyperparameter combinations not studied here represent better performance. If possible, I would like to invest more time on training all possible hyperparameter combinations.

There are some hyperparameters not discussed in this project like number of neural network hidden layers. Such hyperparameter is important to the efficiency of training model. I would like to study more on these hyperparameters later.

Moreover, the neural network activation function is not discussed in detail in this project. ReLU provided with Keras is used as an activation function here. More activation function like Softmax and LeakyReLU could be worth to study in the future.

Last but not least, FIFO is used as the update rule for replay memory when it reaches its maximum capacity. Some more complicated update rule may offer better result. For example, each transition in the replay memory could be assigned a weight vector determining its importance. When the replay memory reaches its limit, just drop the most unimportant transition experience first.

Overall, even if the project is successfully solved in this report, there are still many relevant studies need to work on in the future.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529-533.

[2] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[3] 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*.

[4] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.