Deep Q-Learning on Lunar Lander Game

Technical Report · May 2019

CITATIONS
0

READS
1,154

1 author:

Xinli Yu
Temple University
6 PUBLICATIONS **12** CITATIONS

SEE PROFILE

Deep Q-Learning on Lunar Lander Game

Xinli Yu
xyu350@gatech.edu

ABSTRACT

The main objective of reinforcement learning (RL) is to enable an agent to act optimally to maximize the cumulative long-term reward. Q-learning is a model free RL algorithm, which iteratively learns a long-term reward function "Q" given the current state and action. The Deep Q-Learning Network (DQN) facilitates the Q-learning by modeling the Q-function as a neural network.

This project implements and experiments such DQN models on the OpenAI Gym's LunarLander-v2 environment, using a two-layer feed-forward network with a technique named "experience replay". Extensive experiments are done to determine the neural network size and tune various hyper-parameters including learning rate α, reward discount factor γ and exploration-exploitation trade-off ϵ. Major findings include 1) the Lunar Lander favors a large hidden layer but not a deeper network; 2) a near-one reward discount is necessary for the model to consider final successful landing. Finally, our best model can stably achieve 280+ mean reward for a trial of 100 landing episodes. The code can be found at https://github.com/XinliYu/RL-Projects.

1. PROBLEM DESCRIPTION

Our problem is based on the OpenAI Gym's LunarLander-v2 environment. Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents on a variety of tasks from walking to playing games like Pong, Pinball or Car Racing. In particular, this project focuses on the LunarLander-v2 environment, and the goal is to teach the agent "lunar lander" to successfully land on randomly generated surfaces on the "moon". The following formally defines the state space, action space and the reward for this RL problem, according to [1].

State space. The state space for LunarLaner-v2 is an eight-dimensional vector

$$(x, y, v_x, v_y, \theta, v_\theta, \text{left\_leg}, \text{right\_leg}) \quad (1)$$

providing information of the agent's position (x, y) in space, horizontal and vertical velocity (v_x, v_y), orientation in space θ, angular velocity v_θ, and two 0-1 flags left_leg, right_leg indicating whether the left foot/right foot is in contact with the ground.

Action space. The action space has four actions available: doing nothing, firing the left orientation engine, firing the main engine, and firing the right orientation engine.

Reward. The environment always has a landing pad for the lander at coordinates (0,0). Fuel is infinite. The total reward for moving from the top of the screen to the landing pad ranges from 100 to 140 points varying on the lander placement on the pad. If the lander moves away from the landing pad, then it is penalized the amount of reward that would be gained by moving towards the pad. An episode finishes if the lander crashes or comes to rest (the environment will return a 0-1 flag indicating if the episode finishes), receiving an extra -100 or +100 points respectively. Each leg ground contact is worth +10 points. Firing the main engine incurs a -0.3 point penalty for each occurrence.

> **Remarks**. Based on the description, the maximum reward for one episode should be around 260. However, in our experiments, 310+ reward is observed.

Goal: The problem is considered solved when achieving a score of 200 points or higher on average over 100 consecutive landing attempts.

> **Difficulty**: The LunarLander-v2 changes the shape of the landing surface every time. The lander must learn to land in all situations.

2. Q-LEARNING ALGORITHM

Q-Learning is a *model-free RL* algorithm that does not require an explicit definition of a Markov decision process. It trains an agent to learn an optimal policy from a dynamic environment, and the learned optimal policy tells the agent what action to take at each state.

Given an environment, let S denote the state space and A denote the action space. In Q-learning, the expected total long-term reward given a state s and an action a is predicted by the *Q-function* $Q(s,a): S \times A \to \mathbb{R}$, where the use of letter "Q" may be interpreted as the "quality" of the action a in the state s. The agent should take the optimal action $\pi(s)$ for a state s such that the expected long-term reward is maximized, i.e.,

$$\pi(s) = \text{argmax}_a \, Q(s,a),$$

where π denotes the optimal policy, an $S \to A$ map from states to actions.

Q-learning applies a modified form of Bellman equation to learn the optimal policy. If we only consider 1-step transition $\langle s_t, a_t, r_t, s_{t+1} \rangle$ where r is the immediate reward of the current step, and s_{t+1} is the next state, then we have:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a) \quad (2)$$

where $\gamma \in [0,1]$ is the discount factor specifying how far ahead in time the algorithm should look. This means the optimal long-term reward $Q(s_t, a_t)$ for the current state s_t and action a_t, is the immediate reward plus a discounted optimal long-term reward for the next state.

> **Remarks**. For the lunar-lander problem, there is 120 points for a good landing, a large portion of the reward we can earn. To prioritize this final success, we expect a good γ to be near 1.

In addition, we can use a learning rate α, which enables a moving averaging between old and new values of Q. This allows the learning of Q to converge smoothly, even if our environment is noisy. The update formula with α is then

$$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a_{t+1})) \quad (3)$$

The equation (3) defines an iterative process. When both states and actions are discrete and finite, a straightforward method to solve (3) is to model Q as a matrix and then iteratively runs (3) until convergence. This is called the tabular Q-Learning method.

Difficulty. Such method is difficult to deal with large and continuous state space. Our lunar lander problem has infinite continuous state space like velocity or angle. We can of course apply discretization techniques, but the performance will be hurt.

Solution. Fortunately, Q-function in the Q-learning is a conceptually general function and can be implemented as any proper model. Therefore, we can implement it as a neural network, a popular machine learning approach successful in a wide spectrum of tasks.

3. THE DEEP-Q LEARNING

The *Deep Q-learning* is an extension of the Q-Learning algorithm by modeling the Q-function $Q(s, a)$ as a (deep) neural network [1-2]. The discussions in the section hold for the general framework of deep Q-learning. Our concrete implementation of the neural network is in the next section.

In this approach, the Q-function is a complicated composition of a variety of parameterized functions, taking the input s, a and making predictions of long-term utility. A loss function L is designed to measure how well Q makes the prediction. Finally, the parameters of Q are trained by calculating gradients of L and applying optimization. The objective of Q-learning is to make the iterative process such like (2) converge, and then one choice of *loss function* is the "squared difference",

$$L = \left[Q(s_t, a_t) - \left(r_t + \gamma \max_a Q(s_{t+1}, a)\right)\right]^2 \quad (4)$$

where "$\max_a Q(s_{t+1}, a)$" is evaluated using the current predictions, and "$r_t + \gamma \max_a Q(s_{t+1}, a)$" is like a target value for $Q(s_t, a_t)$ in a classic regression problem .

Single forward pass design. Again, let S denote the state space, A denote the action space, and $|A|$ be the number of actions in the action space. If our neural network strictly follows the definition of Q-function as an $S \times A \to \mathbb{R}$ function, then we need $|A|$ forward passes in order to find the optimal a for maximization $\max_a Q(s_{t+1}, a)$.

Following [1-2], we instead design a neural network $\mathcal{N}(s): S \to \mathbb{R}^{|A|}$. $\mathcal{N}$ works like a score function, producing a $|A|$-dimensional real-valued vector given a state s, and each score in this vector is the score for the corresponding action. If we denote $\mathcal{N}(s)[a]$ as the score in $\mathcal{N}(s)$ corresponding to action a, then $Q(s, a)$ is implicitly modelled as

$$Q(s, a) = \mathcal{N}(s)[a] \quad (5)$$

The benefit of such $\mathcal{N}$ is that it generates all scores in one forward pass, and then $\max_a Q(s_{t+1}, a) = \max \mathcal{N}(s)$. We will implement $\mathcal{N}$ as a feed-forward network in the next section.

Experience replay. The above design is observed to have serious stability issues, as shown later in Fig 2, if we always train (4) using the latest state transition. A technique called *experience replay* described in [3] is adopted by [1-2] to improve learning stability. It simply uses a list called *replay memory* to preserve a fixed number of recent historical state transitions (e.g. 10000 historical transitions, may come from multiple episodes). Every time we randomly sample a number of state transitions from the replay memory, pass them as a batch into the neural network for training.

Remarks. This also improves training efficiency in practice. Training a model using batch data is much faster than one-by-one training when using a modern deep learning tool like Pytorch or Tensorflow.

Exploration-exploitation trade-off. Q-learning needs to consider the exploration–exploitation tradeoff like other reinforcement learning algorithms. One way for this trade-off is to look at the term "$\max_a Q(s_{t+1}, a)$" in (2) and (4). On one hand we need to explore the environment to get a more complete picture of transition and outcomes, which means we could choose some random action rather than the optimal action; on the other hand, we should always execute the current optimal action in order to effectively train the agent.

We use a mixed strategy. Define a trade-off parameter $\epsilon \in [0,1]$. The agent has probability ϵ to randomly choose an action for the loss (4), rather than going for the optimal action. In addition, we also dynamically change ϵ overtime.

- When training beings, the Q-function is not trained enough to make good prediction. In this case, going for the optimal action is not useful. Therefore, we should have larger ϵ at the early stage.
- As the training goes on, the Q-function gains more predictive power, and we should gradually have more trust in its predicted utilities. Therefore, ϵ should decrease overtime.

To achieve above heuristics, we use three parameters ϵ_{start}, ϵ_{min} and ϵ_{decay}, and in the range of $[0,1]$. Let ϵ_t be the value of ϵ for the tth state transition, then $\epsilon_1 = \epsilon_{\text{start}}$, and $\epsilon_t = \max\left(\epsilon_{\text{min}}, \epsilon_{\text{decay}}^{t-1}\epsilon_1\right)$; that is, ϵ_t is decreased by a factor of ϵ_{decay} after every state transition.

Learning rate. In deep Q-Learning, scheme similar to (3) is implemented by the optimizer, and our learning rate α is the learning rate of the optimizer.

Complete algortihm. Based on all previous discussion, our Q-learning algorithm have the following parameters: **1)** the discount factor γ; **2)** the learning rate α; **3)** the exploration-exploitation trade-off parameters $\epsilon_{\text{start}}, \epsilon_{\text{min}}, \epsilon_{\text{decay}}$; **4)** the

replay memory size, and the replay memory sample size for every state transition.

Algorithm 1: The deep Q-Learning algorithm for this project

Input : the environment, all parameters, and the maximum number of training episodes max_ep.
Output: A trained Q-function

Initialize parameters of Q, initialize replay memory;
$\epsilon \leftarrow \epsilon_{\text{start}}$, ep_idx $\leftarrow 0$;
for ep_idx < max_ep **do**
  reset environment;
  while the lander not crashed or landed **do**
    With probability ϵ, sample an action a from the sample space; with probability $1-\epsilon$, choose a that maximizes $Q(s,a)$;
    Given a, the environment generates a state transition s_t, r_t, s_{t+1} and a 0-1 flag indicating if the current episode ended;
    if the replay memory is full **then**
      remove the oldest transaction;
    end
    Add s_t, r_t, s_{t+1} to the replay memory;
    if the replay memory has enough state transitions to sample **then**
      Sample a batch of state transitions;
      Update parameters of Q through optimization with learning rate α, using $r_t + \gamma \max_a Q(s_{t+1}, a)$ as target values, and the loss function in (4);
    end
    if $\epsilon > \epsilon_{\min}$ **then**
      $\epsilon \leftarrow \epsilon \times \epsilon_{\text{decay}}$;
    end
  end
end

4. THE FEED FORWARD NETWORK

The previous section established the deep Q-learning framework. In this project, we try a *feed-forward network* (FFW) as the $\mathcal{N}$ in (5).

> **Remarks**. FFW is one simplest type of neural networks. We make this choice because this lunar-lander problem is relatively simpler in comparison to other problems the deep learning approach usually target in practice (e.g. computer vision, NLP, etc.). By Occam's razor, its simplicity matches the simplicity of FFW.

An FFW is a composition of multiple "layers" of linear or non-linear functions [4], and an FFW with H hidden layers can be written as the following, where the symbol "$\circ$" denotes a function composition,

$$\text{FFW} = f_{\text{input}} \circ (f_1 \circ a_1) \circ \dots \circ (f_H \circ a_H) \circ f_{\text{output}} \qquad (6)$$

Here the *input layer* f_{input} takes the input state vector (***in our problem*** the 8-dimensional vector as in (1)) and generates a hidden state vector of dimension n_1. A *hidden layer* $f_i \circ a_i, i = 1, \dots, H$_consists of a transformation f_i and an activation function a_i; every hidden layer takes a vector of dimension n_i as its input, and we say f_i has n_i *hidden units*. If f_i is a *linear function* or an *affine function* (linear plus bias) then we say f_i is a linear layer. For $i \le H-1$, the hidden layer f_i produces a vector of dimension n_{i+1} for the next hidden layer; for $i = H$ the hidden layer produces a vector for the *output layer* f_{output}, and f_{output} generates a final state vector (***in our case*** a 4-dimensional score vector for each candidate action). Usually $f_{\text{input}}, f_1, \dots, f_H, f_{\text{output}}$ have learnable parameters. Also note if f_i takes an n_{i} dimensional vector as its input, then we can say this hidden layer has n_i *hidden units*. After defining the network as in (6), we also need to define a *loss function* (***in our case*** (4)) and then use optimization to learn those learnable parameters.

For our problem, we use an FFW with two linear hidden layers of units 256 and 128 by extensive experiments (see later Experiment 4). Let the activation functions be rectifiers $a_i(\mathbf{x}) = \text{relu}(\mathbf{x}) = \max(0, \mathbf{x})$ (element-wisely set negative values as zero), and let f_{output} be the identity function, then

$$\begin{aligned} &\mathcal{N}(s) \coloneqq \text{FFW}_{\text{lunar-lander}}(s) \\ &= \mathbf{A}_2 \text{relu}\big(\mathbf{A}_1 \text{relu}\big(\mathbf{A}_{\text{input}} s + \mathbf{b}_{\text{input}}\big) + \mathbf{b}_1\big) + \mathbf{b}_2 \end{aligned} \qquad (7)$$

where $\mathbf{A}_{\text{input}}$ is a 256×8 matrix, $\mathbf{A}_1$ is a 256×128 matrix, $\mathbf{A}_2$ is a 128×4 matrix, and $\mathbf{b}_{\text{input}}, \mathbf{b}_1, \mathbf{b}_2$ are 256/128/4 dimensional bias vectors.

5. EXPERIMENTS

Overview: We implemented the Algorithm 1 in Sec. 3 in Python 3 with a PyTorch based FFW. We first need to select a proper FFW to implement the Q-function, e.g. how many hidden layers, and how many units for each layer. The we will tune the hyperparameters to stretch the model performance as much as possible. ***Every training*** is using 2000 episodes. After a model is trained, it will be ***tested*** by ten trials of 100 episodes, i.e. 1000 test episodes in total.

> **Difficulty**. An exhaustive grid search for parameter tuning is infeasible because we have multiple hyperparameters in Algorithm 1, and training a neural network is time consuming.
>
> **Solution**. Use the heuristics in Sec 2 and 3, e.g. γ should be near 1, more exploration when training begins, etc. Also use a medium-sized network to preliminarily test some guesses of parameters without obvious heuristics, like learning rate α. After all, we choose our initial parameters as $\gamma = 0.99, \alpha = 10^{-4}$, and $\epsilon_{\text{start}} = 1.0$, $\epsilon_{\text{decay}} = 0.998, \epsilon_{\min} = 0$. We choose replay memory of size 65536, and a sample size of 32.
>
> During parameter tuning, each time we only vary one parameter with other parameters fixed to see the effect. The tuning results turn out very near our initial parameters. We discovered potential competitive or better options $\epsilon_{\text{start}} = 0.5$ and $\epsilon_{\text{decay}} = 0.99$.

Experiment 1: **network size selection & effectiveness of experience replay**. We tested seven 2-layer networks with 32/16, 64/32, 96/48, 128/64, 192/96, 256/128, 512/256 hidden units, and three 3-layer networks with 128/64/32, 256/128/64 and 512/256/128 hidden units. All layers have RELU as the activation excluding the last layer. We did 6 times of trainings of 2000 episodes (exception: 512/256 and 512/256/128 trained with extra 500 episodes), and ten 100-episodes tests for each trained model (i.e. 1000 episodes for each trained model). The parameters are those recommended in above overview. ***Our best models*** include a 256/128 model achieving mean test reward of 283, whose per-episode rewards in both training and test, and mean rewards in training are shown in Fig 1. A 512-256 network also achieves a mean test reward of 284. However, we ***decide*** to choose the 256/128 model for this project because it uses less

computation time. ***In addition***, we disable the experience replay of the best-performing 256/128 model and did another 6 times of train/test for comparison. Results are shown in Fig 2, which confirms the instability issue discussed in Ssec 3.

Fig 1. **Best result**. Illustrates the reward overtime for our best performing FFW-Based Q-Learning of 256/128 hidden unites. ***Grey line***: per-episode rewards; ***red line***: 100-episode mean training rewards; ***blue line*** (top axis): per-episode reward during one trial. The ***oscillation*** of per-episode training reward (grey line) is normal and caused by the random sample and the stochastic optimization. ***Overall***, the 100-episode mean training reward steadily grows, and the test reward consistently stays above 200.

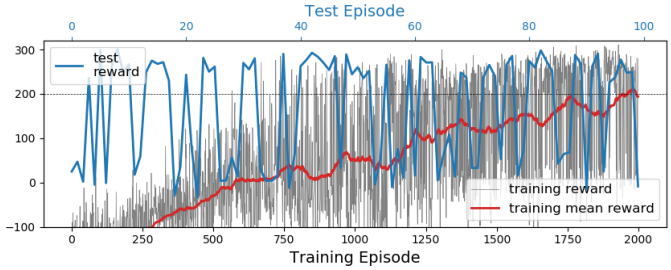

Fig 2. **No experience replay**. The same model configuration as Fig 1 with experience replay disabled: model is always trained by the last 32 state transitions rather than sample 32 state transitions from the replay memory. ***Instability*** is obvious in both training and test. Performance is also much worse than Fig 1.

To have a more comprehensive comparison, we recorded the mean rewards of the last 100 episodes in training, and the mean rewards of all 1000 test episodes for each trained model. The mean rewards for both training and test for all networks are illustrated by box plots in Fig 3. ***The advantage of larger hidden layers is clear***, but it is a ***surprise*** that a deeper 3-layer network performs worse and more unstable (higher mean-reward variance) than their corresponding 2-layer networks. A 512/256 generally network performs at the same level as a 256/128 network but need more computing time. This figure provides ***strong empirical support*** for our choice of the 256/128 network for this project.

Experiment 2: **The discount factor γ tuning** is shown in Fig 4. We trained and tested twice for each $\gamma = 0.87, 0.93, 0.96, 0.99, 0.995$ and $, 0.999$ and choose the better result for comparison. As explained before, since we have an ultimate goal of successful landing, intuitively ***γ should be large enough*** for the Q-learning to consider cumulative reward for a long future. We see $\gamma = 0.99, 0.995$ have good performance, but $\gamma \leq 0.96$ is not working. However, an extreme value like $\gamma = 0.999$ also does not work, meaning it is still necessary for the model to consider immediate reward.

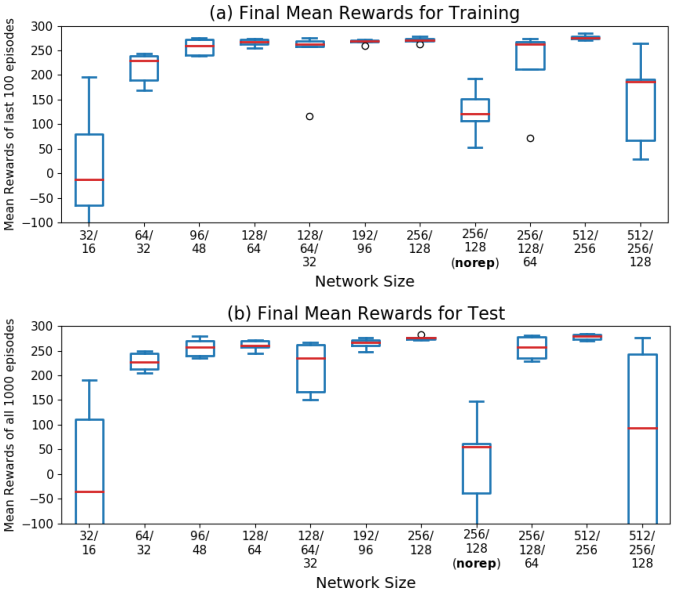

Fig 3. **Network size selection**. Each training has 2000 episodes and each test is ten times of 100 episodes. Repeat training/test six times for each size configuration. Record the mean rewards for the last 100 episodes in training and the mean reward of all test episodes. Plot these mean rewards as box plots showing the median, the quartiles, the max/min and outliers (circles in the plot) of these mean rewards. One extra experiment of 256/128 model without experience replay is included, and its inferior performance is again obvious, consistent with Fig 2.

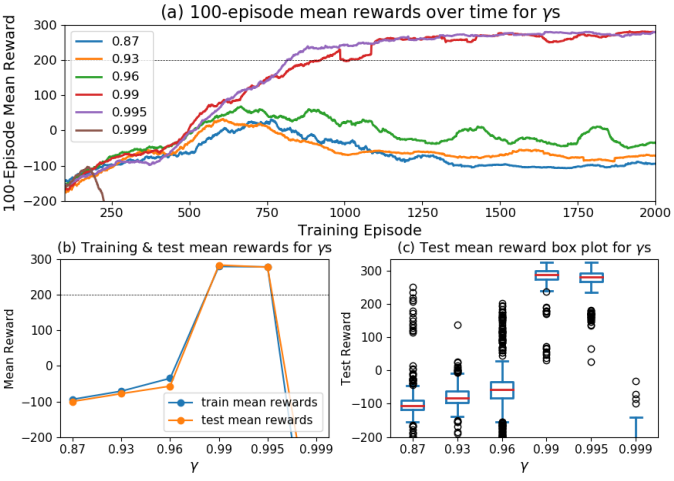

Fig 4. **Parameter tuning for γ**. (a) 100-episode mean reward trend over time; (b) train/test mean reward for different γs; (c) box plot for test rewards for each γ. Based on the results, ***$\gamma = 0.99$ is best***, and $\gamma \in (0.99, 0.995)$ should be good.

Experiment 3: **Tuning the exploration-exploitation trade-off parameters** $\epsilon_{\text{start}}, \epsilon_{\text{min}}, \epsilon_{\text{decay}}$, introduced by the dynamic strategy of adjusting the trade-off in Sec 3. We ***discover*** the model, for this particular lunar-lander environment, is less sensitive to the trade-off than expected if trained by sufficiently many episodes. This is probably due to that LunarLander-v2 is not a complicated environment.

Nonetheless, the experiments also indicate certain level of exploration can make a faster learning.

ϵ_{start} is the initial trade-off as well as the maximum probability the model may explore a non-optimal action. We find from Fig 5 that even $\epsilon_{\text{start}} = 0.01$ leads to a good performance after 2000 episodes; however, the learning is slower and less stable. $\epsilon_{\text{start}} = 0.5$ seems to be a good option, fattest learning with less outliers, as shown in Fig 5 (a)(c), even though $\epsilon_{\text{start}} = 1.0$ has best mean reward.

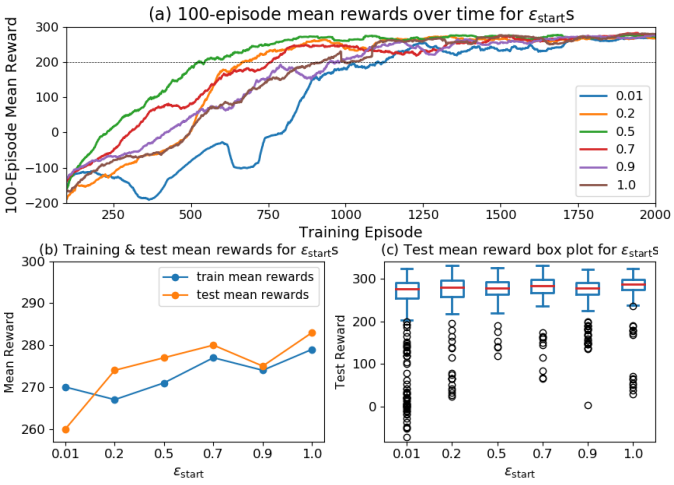

Fig 5. Parameter tuning for ϵ_{start}.

The situation for ϵ_{decay} is similar. $\epsilon_{\text{decay}} = 0.9$ will effectively disable the exploration after 100 episodes, and the learning is slower and less stable. A higher ϵ_{decay} means less decrease of ϵ overtime and thus encourages exploration, resulting in a faster learning. Based on the results, $\epsilon_{\text{decay}} = 0.99$ could be a better choice than the initial guess of 0.998, faster training with less test outliers.

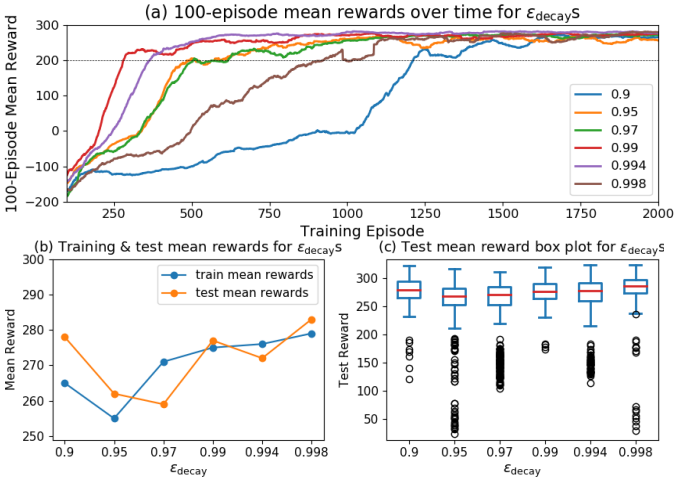

Fig 6. Parameter tuning for ϵ_{decay}.

ϵ_{min} determines how much chance to explore at a later stage of the training, when the model is growing stronger in evaluation. Intuitively, ϵ_{min} should not be large because, as randomness at a later stage of training makes less sense. Results in Fig 7 confirms this heuristic, with $\epsilon_{\text{min}} = 0$ clearly has the best performance.

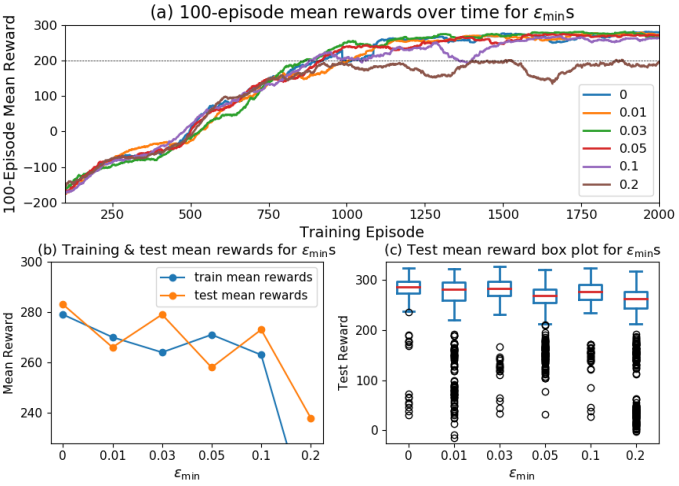

Fig 7. Parameter tuning for ϵ_{end}.

Experiment 4: **Tuning the learning rate** α. Our models is sensitive to the learning rate, with a small working range $[5 \times 10^{-4}, 5 \times 10^{-5}]$.

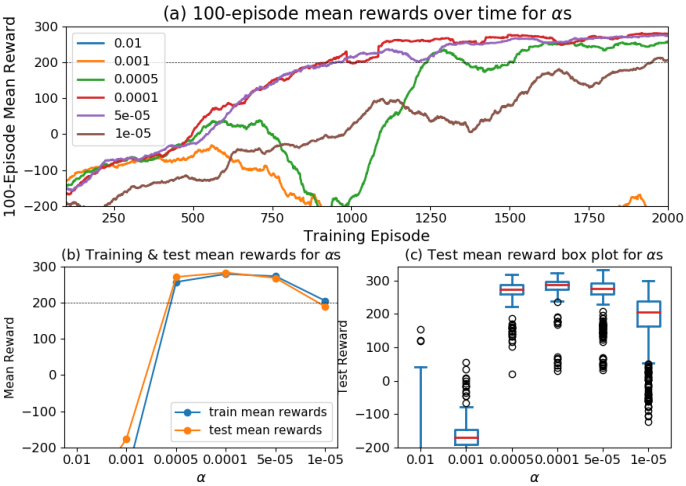

Fig 8. Parameter tuning for learning rate α.

Future work on replay memory & sample size tuning. We experimented other options of replay memory size and replay sample size, and found the memory size should not be too short (no much difference from not using replay memory), or too high (like sampling the whole history, introducing much noise). Sample size does not have strong effect as long as it is not too small, because more samples is just like more training. Due to time and space limit, we are unable to present the full results, and this can be a future work.

REFERENCES

1. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518 (2015): 529.
2. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *rXiv:1312.5602* (2013).
3. Lin, Long-Ji. *Reinforcement learning for robots using neural networks*. No. CMU-CS-93-103.d, 1993.
4. Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.