CS 7642 Reinforcement Learning and Decision Making (Project 2): Demonstration of Generalized Learning Scheme for Continuous State-space Task using Deep Q-Network with LunarLander-v2 Game

Qisen Cheng

Computer Science Department, Georgia Institute of Technology,
Atlanta, Georgia, USA 30318, qcheng35@gatech.edu

Abstract. This work, as part of the course requirement of CS7642, demonstrates the theoretical setup, and edge-cutting deep Q-network (DQN) method as a generalized learning solution for continuous state-space task, by playing the LunerLander-v2 game. The core generalization problem in tackling continuous state-space task is the approximation of the value function (or equivalently the quality function (Q function)). As an edge-cutting method, DQN has drawn wide attention since it first stably solved the Q function approximation problem with deep neural network, which is named deep Q-network (DQN) [1]. Two major strategies, random experience replay and fixed target, help the DQN algorithm achieves good stability while leveraging the flexibility of neural network in function representation. The LunerLander-v2 game well abstracts the continuous state-space tasks, so that serves as a good setup for demonstration of DQN method. In this report, a (python) implementation of the DQN algorithm is introduced to solve the LunarLander-v2 game by achieving average score over 200. The stabilization strategies and parameter tuning are also demonstrated.

Keywords: CS7642, generalized reinforcement learning, deep learning continuous state-space task, deep Q-network, stabilization, LunarLander-v2

1 Introduction

1.1 Deep Q-network (DQN)

Learning tasks with continuous state space and discrete action space is a good abstraction of real-life problems, in which we (or agents) are generally given limited choices in each step of a complex environment. One of the major problems in solving continuous state-space tasks is the value function approximation, or equivalently the quality function (Q function) approximation. Traditional methods base the approximation either on interpolation (i.e. Averager), or on fixed function representation (i.e. linear function). To better solve the problem, DQN leverages the advanced flexibility of neural network in function representation to achieve a much more accurate model. The main theoretical setup of DQN method can be demonstrated using following equations [1].

$$Q(s,a) = f_{DQN}(x_s) \tag{1}$$

$$Q(s,a) \to r + \gamma \cdot \max_a Q(s',a) \tag{2}$$

$$loss = \frac{\sum_D [Q(s,a) - Q(s',a)]^2}{N_D}, \boldsymbol{D}: \{(x_s, a, r, x_{s'})\} \tag{3}$$

where Q is the Q function at certain state-action pair (s, a); $\{x_s, a, r, x_{s'}\}$ is one sample transition which contains observations at current and next state, the action taken, and the reward given. The training of Q network is in analogy to bellman equation. The next-step discounted (γ) prediction is regarded as a target for adjusting the current Q network. The updating of neural network is done in batch mode, which is according to a randomly sampled batch of transitions from all the observed data. This batch-updating is known as *random experience replay* [1]. The current Q network is adjusted to minimize the loss defined as the mean-square-error between current values and next-step predictions of the selected mini-batch of samples. By iteratively updating the network weights, the Q network will get close to the true Q function. w is the weight vector in the prediction function of observation vector acquired at certain state. The updating rate (or learning rate) is determined by the optimizer parameter α.

One issue DQN might suffer is about the learning stability. Although the DQN gives more flexibility in adjusting the representation of Q function, it can be in severe overfitting, and ultimately embeds large instability in the learned model. Several tricks can be used to solve this problem in different extent. The most effective one introduced by [1] is called *fixed target*, which is essentially to regulate the updating rate of the target – the next-step prediction of Q in each step. This strategy de-correlates the updating Q and its target so that a more stably-paced optimization could be done for each step. Other stabilization improvements could be obtained from tweaking the loss function (i.e. Huber loss function) or tuning the general parameters in reinforcement learning algorithms (i.e. learning rate, exploration-greedy epsilon)

1.2 LunarLander-v2 Game

LunerLander-v2 game is one of the games available in platform OpenAI Gym [2]. It features in continuous state-space and discrete action space setup. The game can be described as following (Fig. 1). Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100-140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

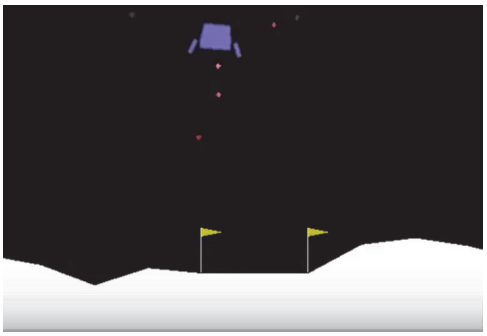


Fig. 1: Screen-shot of animation of lunarLander-v2 game [2].

The implementation of DQN and experimental success on LunarLander game are detailed in section 2. More discussions on parameter tuning and stability improvements are presented in section 3.

2 Implementation and Experimental Results

This section first details the implementation of DQN with ultimately selected parameters. The tuning process of these parameters are presented in following section 3. The experimental results on LunarLander-v2 game shows success of the implementation and performance tuning.

Implementation.

Code structure. In this work, the DQN is implemented in an object-oriented manner. The game setup is abstracted with a Game class, which controls the playing, rewarding, and finishing functions of the game. The playing function is encapsulated by an Agent class, which mainly controls the updating of actions by referring to past experience. The past experience is abstracted by another Experience class, which is initialized in agent class to specify the agent's past experience. The deep Q network is setup and adjusted in this class. A built-in main class is serving as an individual observer of the training process performed by Game-Agent-Experience trilogy. A separate experiment.py file is used to perform test on the learned model.

Deep Q network. The architecture of used neural network is based on sequential model, and has 4 layers in total. The first layer has 8 nodes to interface with the 8 dimensional input of observation at each state. The following two hidden layers are both of 64 nodes and rectified linear units activation function (ReLu). Two reasons are behind the choice of ReLu. First, ReLu is nonlinear in nature, and combinations of ReLu can approximate almost any function. Second, the horizontal line part (or zero–line part) could yield possible zero gradient, which indicates no responding to further variation in the input. This could help in speeding up the optimization process due to fewer active neurons or lighter network, and also prevent overfitting by outliers. The last layer outputs 4 dimensional Q(s, a) to represent Q for each discrete action.

Optimizer. RMSProp is selected due to the fact that it scales alpha for each parameter according to the history of gradients (previous steps). The updating is basically done by dividing current gradient in update rule by the sum of previous gradients. As a result, when the gradient is very large, the learning rate is reduced and vice-versa. This could help reduce the overfitting problem and converge to optimal estimation.

Random experience replay. A batch of 64 transitions are sampled randomly from the entire past experienced data. The 64 is selected semi-randomly by using the same number of neurons in each hidden layer of Q network. Updating rate of target. The targets are from a semi-separate Q network, which is regulated to be updated to be the latest trained Q network in a certain updating rate. The rate is chosen to be 600 with the consideration that each episode is finished in roughly 200 steps in average. So that this updating rate will allow the target to be static in roughly 3 episodes, which is assumed to be reasonable.

General parameters.

The major parameters selected are the learning rate (α = 0.00025), number of episodes for training (N = 1500), learning discount (γ = 0.99), and exploration greedy rate ε. The number of episodes for training is selected so that enough examples have been experienced for training, which is determined using several ad-hoc experiments. The large learning discount is selected to reflect the fact that the game is played in finite form for each episode. The exploration greedy ε is calculated in the following formula:

$$\varepsilon = \epsilon_{\min} + (\varepsilon_{max} - \epsilon_{\min}) \cdot \varepsilon_{decay} \tag{4}$$

In which $\epsilon_{\min}$=0.01 to ensure the learned Q network is dominating the selection of actions but remain a slight randomness for avoiding overfitting. ε_{max} is 1. ε_{decay} is 0.000025 to allow sufficient exploration in the beginning.

Experimental results.

The experimental results are shown in Fig. 2 and 3. In the training stage (Fig. 2), the model (deep Q network) is learned by experiencing 1500 episodes. It can be seen that the agent is starting to learn about "going down" at very beginning. Then it learned "slowing down for landing" around 250 episodes – reward per episode becomes -100 to -200. After a while, it learned "proper landing" around 380 episodes – reward becomes positive. Finally, it learned "proper landing within the assigned area", which makes the reward around 200 or larger afterwards.

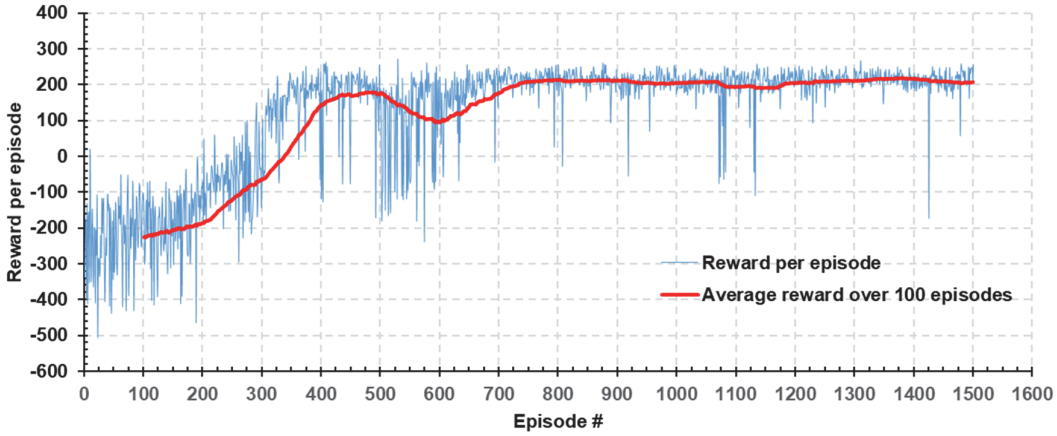


Fig. 2: Learning progress of DQN agent in terms of reward per episode.

The testing of learned model is shown in Fig 3. The average reward-per-episode score is evaluated over 500 episodes. This is equivalent to evaluate the reward-per-episode score over 100 episodes for 5 runs, which satisfies the requirement of the project.

Fig. 3: Testing of DQN agent with new episodes to evaluate the average reward per episode. The shown testing is equivalent to evaluate the reward-per-episode score over 100 episodes for 5 runs.

3 Parameter Tuning and Stability Improvements

The tuning of two major hyper-parameters (learning rate and exploration greedy rate) are showed in this section. The improvements of learning stability from using fixed target strategy and huber-loss function are also discussed.

Learning rate.

Learning rate is determined by incrementally sweeping in pre-defined ranges with other parameters being fixed. For simplicity, only the effect of 4 different learning rate is discussed to reflect the selection process. In Fig. 4, we can see the training progress corresponding to 4 different learning rates, respectively. The best learning rate (=0.00025) is selected because it allows the convergence of the updating and remains stable afterwards. If the learning rate is too large (=0.00075) or too small (=0.00005), the model will not converge to the optimal result. It becomes tricky when learning rate is around 0.0005. It can be seen that it could converge to the optimum and stabilized for some time, but it has larger variance than the 0.00025 case. In fact, the 0.0005 case diverges after convergence in other runs. In general, it is still not stable.

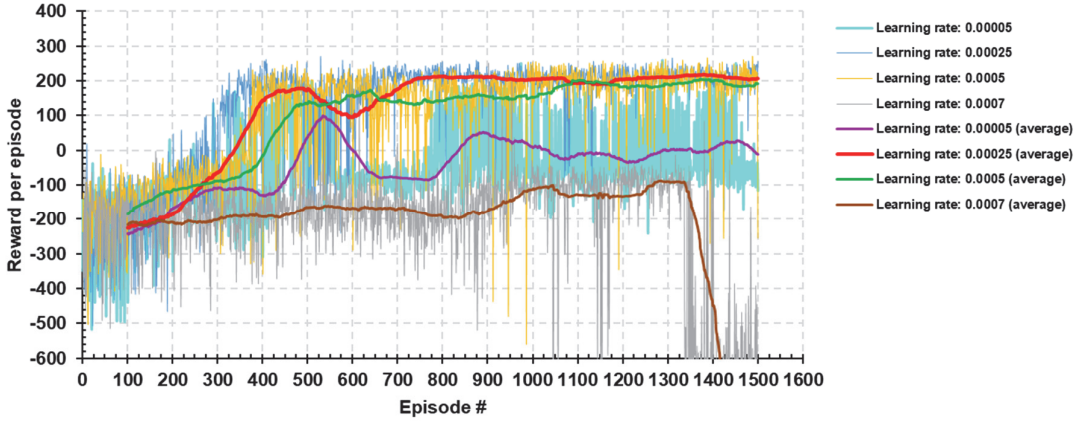


Fig. 4: Tuning of learning rate via a series of ad-hoc experiments.

Exploration greedy level.

Exploration greedy level is determined by equation 4 as presented in section 2. The most important parameter in determining the greedy level is the decay rate (ε_{decay}) of greedy level. The selection of the decay rate should ensure two things: 1) sufficient exploration before the agent actually learns the landing strategy; and 2) randomness is reduced to very low when the learning is almost done. The selection could be illustrated by the following figure. With decay rate of 0.000025, the exploration greedy level roughly decays inversely proportional to the learning progress.

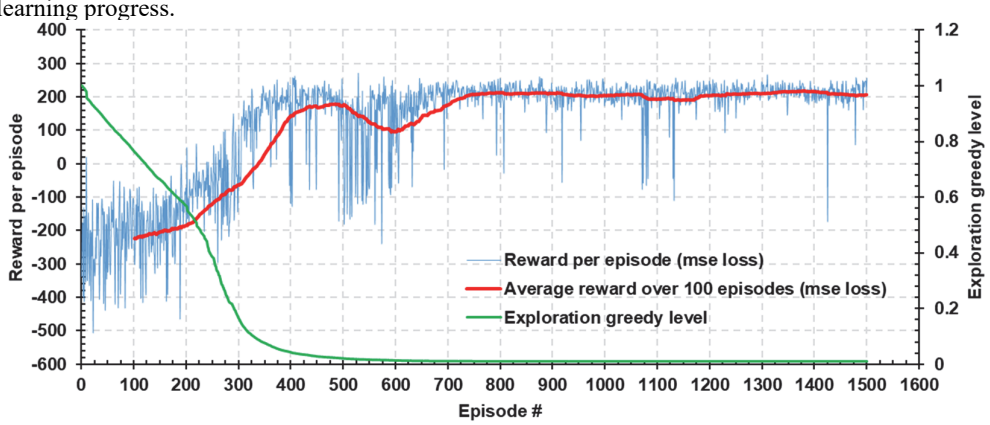
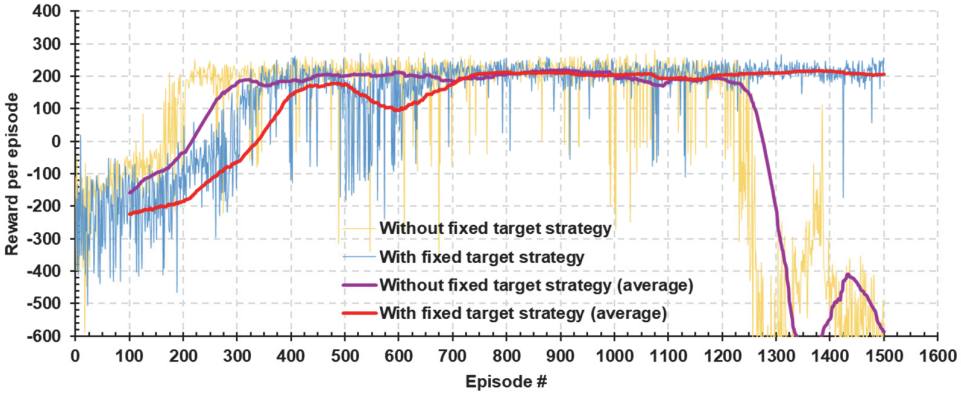


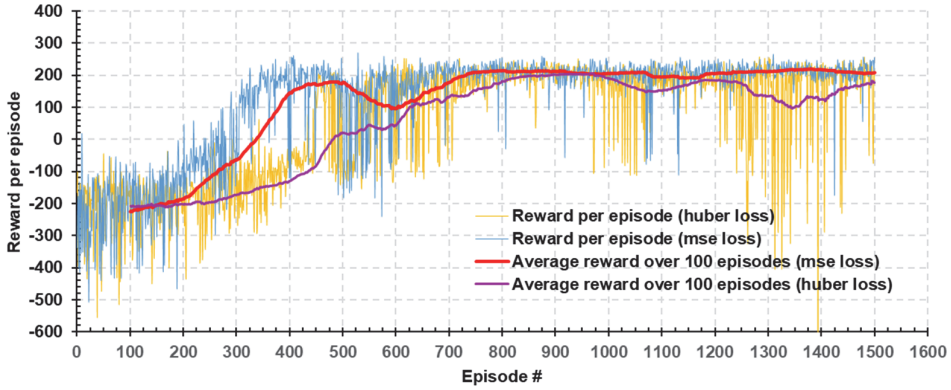
Fig. 5: Determination of decay rate of exploration greedy level. The idea is that exploration greedy level should be roughly inversely proportional to the learning progress indicated by reward.

Stability improvements.

Two strategies are introduced in [1] to improve the learning stability. Fig. 6 shows the effect on stability brought by 1) using fixed target strategy (FT), and 2) using huber loss instead of MSE loss. From Fig.6 (a), FT turns out to be pretty effective in stabilizing the learning process, which makes sense since the algorithm is now converging in a slower pace. However, from Fig. 6(b), it can be seen that the Huber loss does not help with the stability very much. In this case, I believe the drawback comes from the fact the Huber loss essentially introduces bias in estimation compared to MSE loss, which gives the MLE estimation. Thus, MSE loss is finally selected.



(a)



(b)

Fig. 6: Strategies for learning stability improvement. (a) Comparison of learing stability between algorithm with fixed target strategy and without fixed target strategy. (b) Comparison of learning stability between algorithm with MSE loss and with huber loss function.

References

1. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.
2. Brockman, Greg, et al. "OpenAI gym." arXiv preprint arXiv:1606.01540 (2016).