EE526 Deep Machine Learning Final Project

Xiaoshi Guo*

December 17, 2019

1 Introduction

Reinforcement learning Sutton et al. (1998) is an area of machine learning concerned with how software agents should take actions in an environment in order to maximize cumulative reward. It involves an agent, a set of states (S), and a set of actions (A) per state. By performing an action ($a \in A$), the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (R). The goal of the agent is to maximize its total future reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward, $Q(s, a)$. This potential reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state, which can be represented as

$$Q(s, a) = E[G_t | S_t = s, A_t = a] \tag{1}$$

Here, $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, and γ is the discount factor with the value between 0 and 1 and has the effect of valuing rewards received earlier higher than those received later.

To get the value of $Q(s, a)$, Q-learning Watkins and Dayan (1992) is proposed. It applies Bellman equation to learn and update the Q value. The essence of Q-learning is to independently adopt the action which can provide the best Q value:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma max'_a Q(s', a') - Q(s, a)] \tag{2}$$

However, taking the maximum overestimated values as such is implicitly taking the estimate of the maximum value. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping — learning estimates from estimates — such overestimation can be problematic.

To avoid such problem, double Q-learning, developed based on Q-learning Algorithm, is introduced. The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite

*xsguo@iastate.edu

estimator Hasselt (2010), and can thus avoid maximization bias by disentangling our updates from biased estimates. The detailed double Q-learning will be described in Section 2.

When the number of states is small, double Q-learning at its simplest stores data in tables. Nevertheless, when the number of states is very large, this approach fails. Therefore, people combine double Q-learning with function approximation. This makes it possible to apply the algorithm to larger problems, even when the state space is continuous. One solution is to use an (adapted) artificial neural network as a function approximator Van Hasselt et al. (2016). Function approximation may speed up learning in finite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

The remainder of this report is organized as follows: Section 2 illustrates the double deep Q network and how I use it in the project with the pseudo-code. Section 3 presents the performance of this double Q network. Section 4 concludes the paper with a summary of findings and proposes possible improvements.

2 Methodology

In this section, we will introduce double deep Q-learning (double DQN), which is the combination of the double Q-learning and the function approximation of multi-layer neural network.

Double Q-learning is proposed by Hasselt (2010). This algorithm overcomes the issue of maximization bias of Q by having two value functions Q. These two value functions, Q_1 and Q_2, are learned by assigning experiences randomly to update the other one. More specifically, the update consists of finding the action a^* that maximizes Q_1 (e.g. $Q_1(s', a^*) = maxQ_1(s', a)$) in the next state, then use a^* to get the value of $Q_2(s', a^*)$ in order to update $Q_1(s, a)$.

Deep Q-learning (DQN) is a multi-layered neural network that for a given state s outputs a vector of action values $Q(s, ; \theta)$, where θ are the parameters of the network. There are two important ingredients of the DQN algorithm. One is the use of a target network with parameter of θ', which is the same as the primary network except that its parameters are copied every τ steps from the primary network. The other is the experience replay, which is to gather and store samples in a replay buffer with current policy and then randomly sample batches of experiences from the replay buffer. With experience replay, we avoid significant oscillations or divergence in our model — problems that can arise from correlated data.

Double DQN integrates the idea of Double Q-learning and DQN. Also, to make the model more accurate, we can conduct some exploration with ϵ-greedy policy. This policy allows us to take a random action with probability of ϵ and take the greedy action with probability of $1 - \epsilon$. The pseudo-code for the double DQN algorithm is modified based on Van Hasselt et al. (2016) as below.

Algorithm 1 Double Deep Q-Network

Initialize replay buffer D, $\tau < 1$, batch size B, ϵ_{min}, ϵ_{max}, λ, γ

Initialize primary network Q_θ and target network $Q_{\theta'}$

for each episode **do**

  for each environment step **do**

    Observe state s_t and select action a_t using ϵ-greedy policy

    Execute a_t, then observe s_{t+1} and reward $r_t = R(s_t, a_t)$

    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer D

    Update ϵ: $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda * step}$

  end for

  for each update step **do**

    Sample (s_t, a_t, r_t, s_{t+1}) with the batch size of B from buffer D

    Compute target Q value: $Q^*(s_t, a_t) \approx r_t + \gamma Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$

    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t)^2$

    Update target network parameters: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$

  end for

end for

In this algorithm, the designed primary network has two hidden layers. The size of the first fully connected layer is 256, and the size of the second fully connected layer is 512. Activation function of each layer is relu, and minimum square error (MSE) is applied as the loss function. The number of parameters in each layer is shown below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_222 (Dense)            (None, 256)               1792
_________________________________________________________________
dense_223 (Dense)            (None, 512)               131584
_________________________________________________________________
dense_224 (Dense)            (None, 3)                 1539
=================================================================
Total params: 134,915
Trainable params: 134,915
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: Structure of the primary network in double DQN

3 Performance of the Algorithm

All the following results are conducted in Python. The environment of reinforcement learning is Acrobot-v1, which is taken from OpenAI Gym package. In the Acrobot environment, the agent is given rewards for swinging a double-jointed pendulum up from a stationary position. The agent can actuate the second joint by returning one of three actions, corresponding to left, right, or no torque. The agent is given a six dimensional vector describing the environments angles and velocities. The episode ends when the end of the second pole is more than the length of a pole above the base. For each timestep that the agent does not reach this state, it is given $a - 1$ reward. The neural network is built using Tensorflow and keras.

The value of parameters are:

ϵ_{max}	0.99
ϵ_{min}	0.01
λ	0.0005
γ	0.95
τ	0.08
D	1000000
B	32

The performance of this setting is shown in Figure 2 and 3 in terms of cumulative score and average loss, respectively. From Figure 2, we can find that the cumulative score jumps from -499 to -100 very quickly, however it oscillates greatly around -100 as we increase the number of episodes. Eventually, double DQN algorithm has the consistent score of -140 to -80 per episode. Also, I notice that that the average loss ranges from 0.070 to 0.130 eventually. The total training time is for 1000 episodes is roughly 75.739 seconds.

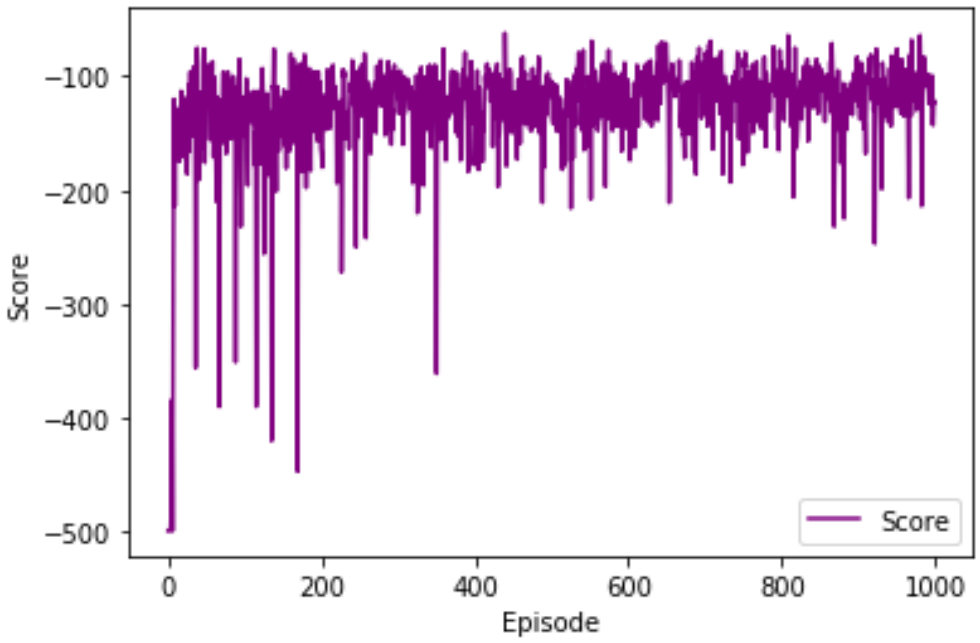

Figure 2: Double DQN performance in terms of cumulative score

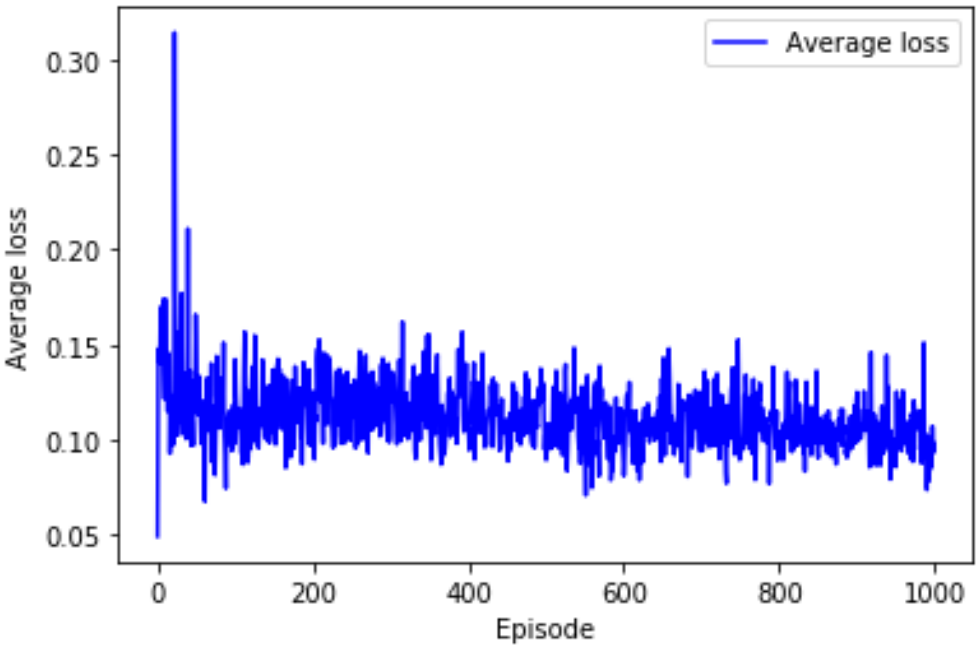

Figure 3: Double DQN performance in terms of average loss

4 Conclusions and Possible Improvements

In this report, the function approximation using neural network is applied to the double Q-learning, which can not only efficiently deal with large number of states in the reinforcement learning environment but also avoid the issue of overestimation brought by naive Q-learning. We test this algorithm on the environment of Acrobot-v1 in Python Gym package, and find that the final score by using this algorithm jumps very fast at the beginning, however it will oscillate around -100 afterwards.

One possible improvement could be increasing the number of neurons and complexity of the neural network, which can increase the accuracy of the function approximation according to the theorem that neural network can approximate the function as long as the number of neurons is enough. Also, by conducting literature review, I find Fujimoto et al. (2018) proposes clipped double Q-learning, which takes the minimum of the next-state action values produced by two networks to compute the update target. This new approach can lead to safer policy updates with stable learning targets.

References

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*.

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

Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 2. MIT press Cambridge.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.