Reinforcement Learning – Function Approximation

Qing Zhou

Apr, 2020

1 Introduction

The motivation of this project is to understand deep Q-learning with learning stability, by experimenting with OpenAI's game Mountain Car and Breakout. For more complicated games like these, we cannot iterate through all possible actions at each state and estimate the reward, this is computationally infeasible. Instead we take an action at a given state, based on the experiences from previous states. And this is where the deep Q-learning comes in.

1.1 Deep Q-learning

Deep Q-learning is basically the combination of deep learning and reinforcement learning. It is a method of automated discovery of feature to approximate an objective function. To understand how deep Q-learning network functions we should first understand how it relates to our games. For a given state of the game, what the Q network does it taking a stack of frames as inputs, and fed it into a sequence of convolutional networks. Then all possible actions are evaluated as Q-values, with the highest value representing the best action at this state. This sounds like deep learning, but the real difference here is how we define the loss function. To understand the loss function defined in the Deep Q network we have to first understand Bellman's principle of optimality and function approximation.

1.2 Principle of Optimality and Function Approximation

Training our network to play games is basically a dynamic optimisation problem, where the actions are evaluated step-by-step, recursively. The principle of optimality states that "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."[]. Moreover, this optimality does not necessarily means it is the best one for the current state, yet it means to select actions to maximize the "cumulative future reward". The deep convolutional network used to approximate the Q-value function is[1]

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+1}^2 + \cdots | s_t = s, a_t = a, \pi] \tag{1}$$

which tells us how good it is for the agent to take action (a) in a state (s) with a policy π. The expectation takes into account the randomness in future actions according to the policy, as well as the randomness of the returned state from the environment, a mathematical expression of "cumulative future reward". Hence the loss function can be expressed as[1]

$$Loss = (r + \gamma \max_{a'} Q(s', s'; \theta_i') - Q(s, a; \theta_i))^2 \tag{2}$$

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ_i' are the network parameters used to compute the target at iteration i.

1.3 Deadly Triad

It is known that deep Q-learning can be unstable, even diverge. This instability has three main causes: function approximation, bootstrapping, and off-policy learning. Function approximation may have inaccuracies in the attributions of values to states. Bootstrapping, may have introduced biases in the initial values, and even spill to other states. The off-policy learning behaves differently from the target policy, hence the improvements on both are not in sync, resulting in a diverging behaviour[2].

2 Mountain Car

The corresponding python file for this section is 'MC_DQN4.py','analysis.py','model.py','summary.py'.

Now that we have understood the deep Q-network on the theoretical aspects, it is time to get in practice. We start with the easier game Mountain Car. After trials of error, we found that using a one-layer model is sufficient to solve this problem. The memory size was to be 100,000, and the target network was set to be in sync with the policy network every 10 episodes. The networks were allowed to run for 200 episodes, after which they seem to diverge and took longer in each episode to find a solution. Fig. 1 present the results for the action in the last 6 episodes. The action meshgrid

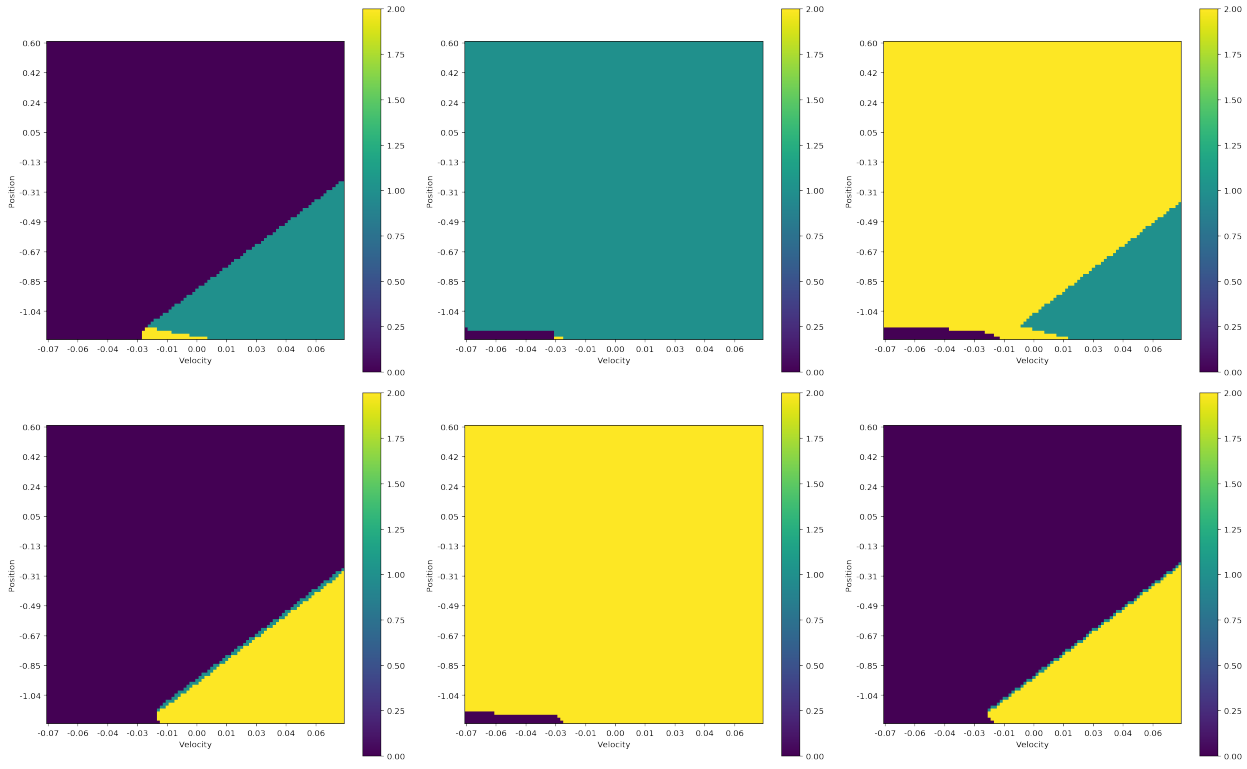

Figure 1: The last 6 episodes for the actions. The x-axis is velocity, ranging from -0.07 to 0.07; the y-axis is position, ranging from -0.6 to 1.2. The possible actions are $[0, 1, 2]$, correspond to "push left","no push" and "push right".

was stored for each episode, at different positions and velocities. Ideally, we expect the network to

take an action to "push left" when the car is on the left part of the slope, and "push right" if it is on the right part of the slope, so it accelerate itself and ultimately arrives at the checkpoint. So the ideal case for the action meshgrid would be a clear cut at $v = 0$, with the left side filled with zeros("push left"), and the right side filled with 2s ("push right"). Yet there is a physical obstacle at the left most of the slope, whenever the car reaches that point, it bounces back. Hence the network learnt to avoid this, by sometimes "rest" at the slope bottom (see top middle panel). When this happens it takes the car normally four times as long to reach the checkpoint. It can also be seen, the network tends to oscillate between the best solution (see bottom left and bottom right panel) and the sub-optimum solution (see top left and top right panels), and sometimes get "stuck" at the local minimum (see middle panels). We have experimented with more complicated networks, with increased number of layers and more neurons per layer, and we have also experimented running the network for a longer time (10000 episodes), but for a longer time the network does not seem to be stablised, yet more "get-stuck" episodes arose, leading to an increasing total loss. The loss function can be seen in Fig. 2. Hence in this Q-learning network, the loss function can not be a good measure

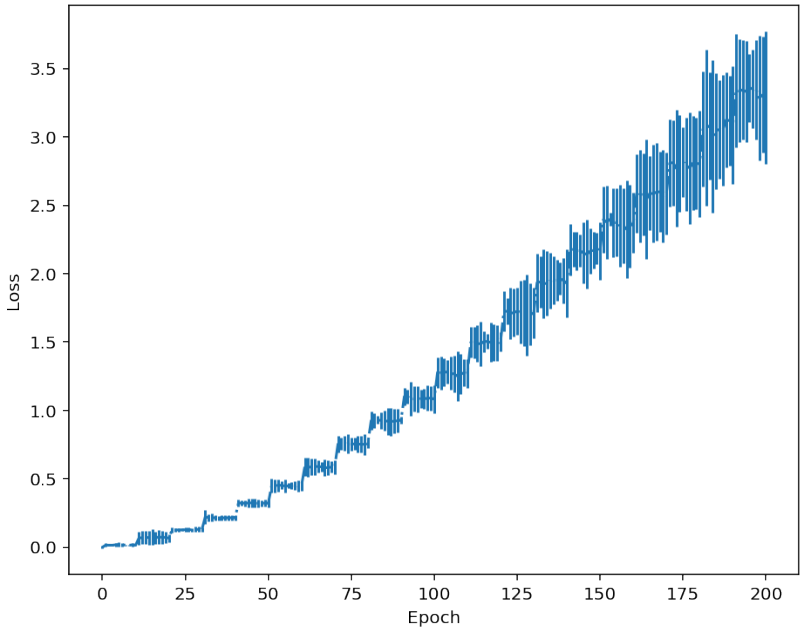

Figure 2: Loss function for training with the Mountain Car game, for a total duration of 200 episodes. Each data point is calculated as the mean loss per episode, and the error bars are the corresponidng standard deviation.

of the stability of the system. Insdead we have developed another measure of "stability", by taking the "relative loss", which is given by the ratio of the error in loss and the mean loss, so

$$Loss_{rel} = \frac{Loss_{std}}{Loss_{mean}} \tag{3}$$

the relative loss function can be seen in Fig. 3. Compared with Fig. 2, it can be seen even though the error in loss and loss itself is increasing, the relative loss tends to stabilise, decreasing from ~ 10 to $\sim 10^{-1}$ for the whole course of the training.

3 Breakout

The corresponding python file for this section is 'DQN-learn.py','model.py','summary.py','ana.py'

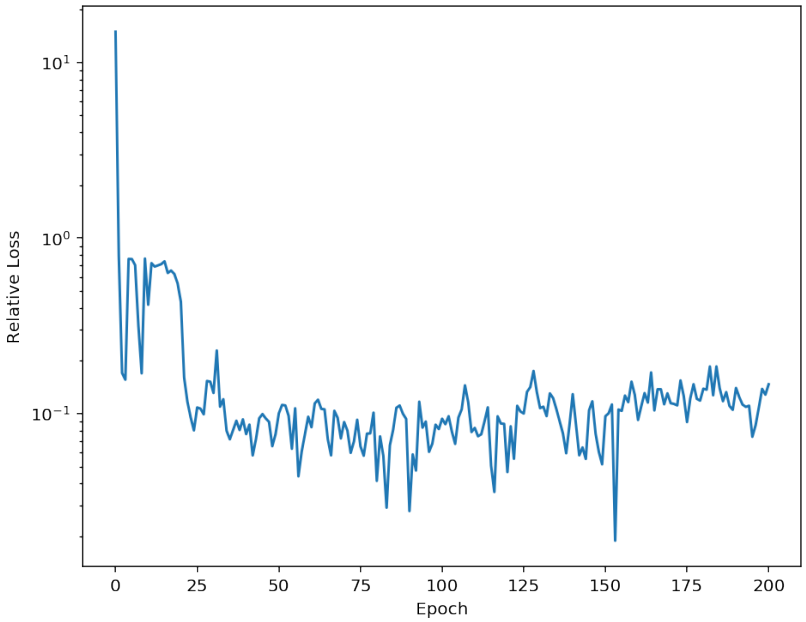

Figure 3: Relative loss function, which is given by the standard deviation of loss in each episode, divided by the average of loss in each episode.

For the game Breakout, we constructed a four-layer convolutional network , with a two layer MLP as the model. The memory size was set to be 10000, and the network was trained for a total duration of 10000 episodes. The policy was set to be updated with the target network every 10 iterations. The reward function and loss function(s) can be found in Fig. 4 and Fig. 5.

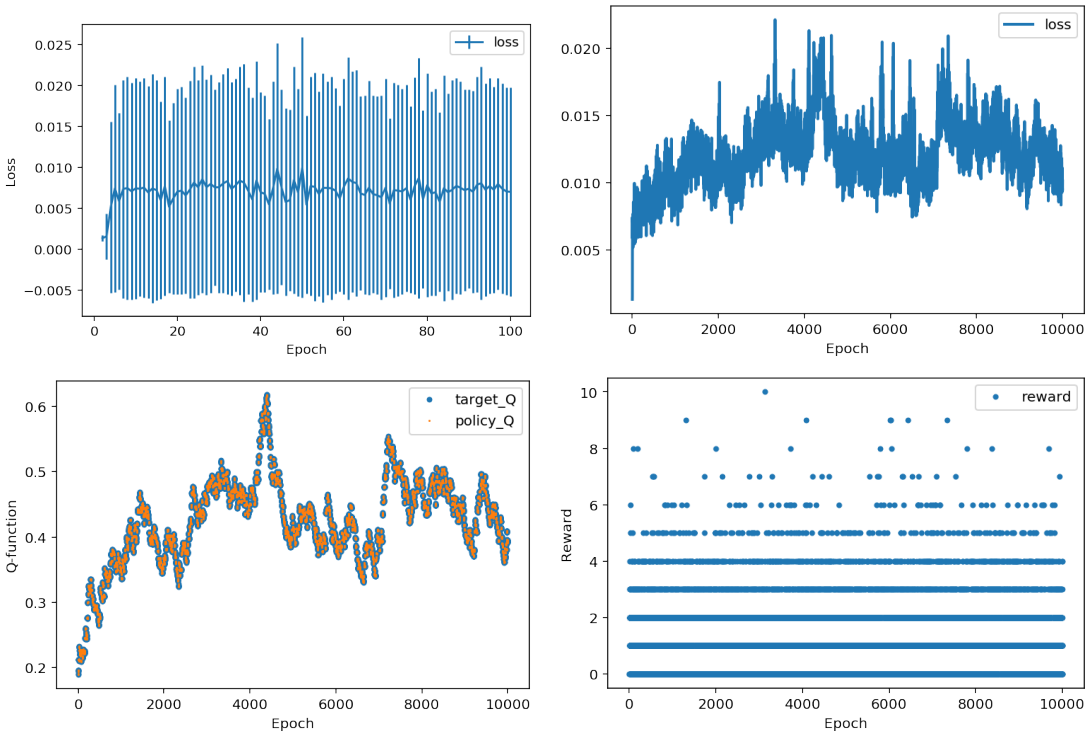

Figure 4: Top left: the loss function in the first 100 episodes, skipping the 0th episode. Top right: the loss function for the total duration of 10000 episodes. Bottom left: the target Q-function and policy Q-function. Bottom right: the reward distribution for the 10000 episodes.

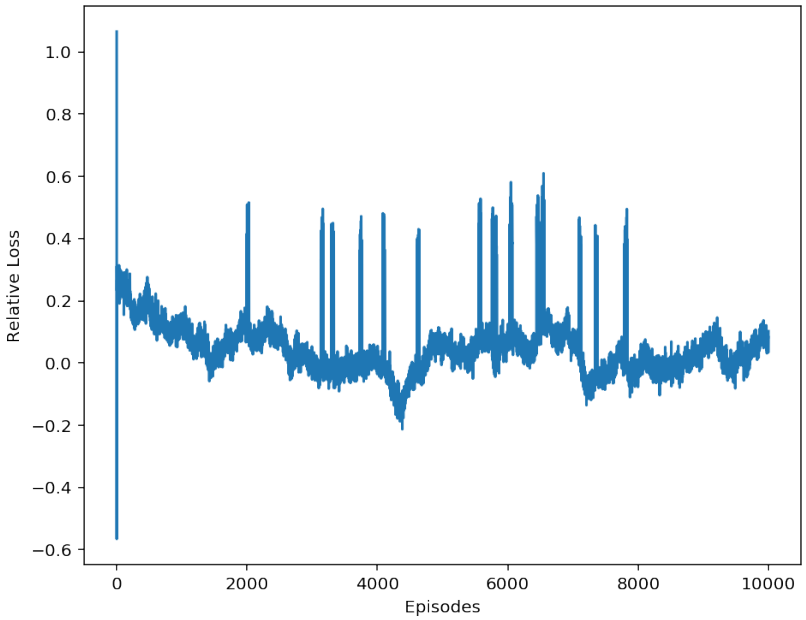

Figure 5: Relative loss function for the game Breakout, plotted on a logarithmic scale.

It can be seen, the loss is almost on the same scale, and within 10000 episodes the relative loss is fluctuating around 1.6 ($10^{0.2}$). The spikes seen in Fig. 5 corresponds to the peaks in the Q-function, each time the target network is updated with the policy network, there could be a peak forming. From the reward function it can be seen, within 10000 episodes the network training was not able to deliver a satisfactory result.

To improve the training result, the following aspects can be considered:

1. Hyperparameter tunning: learning rate, batch size, update frequency, an number of "do nothing" actions performed;[3]

2. Increase the replay memory size: the current memory size was set to be 10000, only 1/100 of the memory size of other successful cases;

Due to limited time we were unable to train the network to deliver a satisfactory result, albeit we have encountered the limitations of the Deep Q learning network, and found potential ways to improve our network for future experiments.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[2] Aske Plaat. Learning to play—reinforcement learning and games, 2020.

[3] Adrien Lucas Ecoffet, Beat Atari with Deep Reinforcement Learning, `https://becominghuman.ai/beat-atari-with-deep-reinforcement-learning-part-2-dqn-improvements-d3563f665a2c`