Ding Ke

CID: 01328574

Reinforcement Learning (Second Half) Coursework

Question 1 (i) The mini-batch method yields the more stable training. It can be observed that the loss curve in **Figure 1(a)** for online learning has a much more zigzag shape, indicating more oscillation in the average loss values along the episodes, while for mini-batch learning in **Figure 1(b)** the curve has a loss decrease that is more steady and significantly less noisy, only a few small spikes appear towards the end of the 100 episodes. The instability of online learning stems from the fact that only single transition is used to train the DQN and the next transition tend to be "nearby" due to the way agent explores the environment, causing correlated updates and unstable data distribution. By keeping an experience replay buffer, mini-batch learning can randomly sample a number of transitions, which breaks the correlation and allows a more uniform data distribution to be trained on, thus more stable training.

Question 1 (ii) Mini-batch learning is more efficient. First by examining the loss curves, in **Figure 1(b)**, it is found that under 100 episodes, the loss for mini-batch learning can drop to a considerably lower level than that reached by online learning(10^{-6} compared with 10^{-4}), showing mini-batch learning's higher predictive accuracy under same training time. For theoretical explanations, online learning each step only samples one transition and discards it after training, to improve the accuracy(reduce loss) the agent must visit that particular state-action pair again to train on the same transition. Meanwhile mini-batch learning using a sample of transitions at each step can update the state-action space much more broadly, implying increased efficiency(less update needed to improve the predictive accuracy).

Question 2 (i) From **Figure 2(a)**, it can be observed that at the bottom right corner some grids are slightly "greener" than those in the top right corner. All the grids in the top right have Q values that all lead to the "up" action(according to greedy), which is obviously not reflective of the actual environment setting. While in some grids in the bottom right, the Q values for "left" action is becoming "greener", meaning more positive, which reflects the environment. The bottom right region predicts Q values more accurately because it is geometrically closer to the starting position, and given the agent is using random action selection for exploration, the chance of going into bottom-right is higher than entering top-right, causing bottom-right experiencing more exploration. This means state-action pairs in bottom-right can be more frequently visited and subsequently Q values updated more.

Question 2 (ii) It would not reach the goal. The agent will select the action with "greenest" Q values(largest) and this leads all the way up to wall, and moving to the wall will keep the agent static. Using only the immediate reward for the update

target, the agent is essentially short-sighted, due to the lack of $\gamma \max_a Q_{(S',a)}$ which is the term that encodes information about future(or next step) gains. As the default reward function used is distance reward, the short-sighted agent will only pick actions that reduce the distance to goal the most. Following this consecutive direct distance reducing action selection, from the starting point, it can only move upward and hits the wall again and again.

Question 3 (i) The key difference is the full Bellman update here ($R+\gamma \max_a Q_{(S',a)} - Q_{(S,A)}$) introduces an update target(an estimated return or gain for current step) that contains an estimation of the discounted future(next step) maximum Q value, using functional approximation of neural network itself. The agent can now learn to select actions that produce the maximum return(gain) which is a discounted sum of all future rewards. Instead of only running into wall, agent can choose to move to the right and go around wall, realising those actions produce the best long term rewards.

Question 3 (ii) The loss curve in **Figure 3(a)** shows a steady decline, while in **Figure 3(b)** with target network the loss shows 9 resurgence peaks after the initial drop in the 100 episodes, corresponding to the target network update which is set to be once every 10 episodes of training. The target network updates itself after Q network stabilises. Each time target network is updated its weights are copied from Q network which are many steps ahead. Before the update the DQN is able to reduce loss by changing Q network's weights via back-propagation. After a sudden change of weights in target network which is used to predict the update target in computing the loss, the DQN needs to re-learn some of Q net weights again to minimize loss, thus the loss temporarily resurges at each update. Overall when updates stabilise, the loss is reduced to a lower level along the training.

Question 4 (i) It's highly unlikely. However the agent could theoretically reach the goal when minibatch has not been filled and agent just randomly explore(assume random initial weights generation of DQN) and possibly land on the goal. After minibatch size filled, the transitions are randomly sampled, and each update changes weights across the state-action space, making the agent again reaching the goal with greedy policy very unlikely. Without exploration in training, agent only has knowledge from transitions initially stored in the experience replay buffer and can only exploits on those.

Question 4 (ii) As the goal state still has the 4 state-action pairs like other states, Q values of those state-actions will depend on how the surrounding states of goal state are being explored. As Q-net hasn't converged fully(meaning not all state-actions' Q values are optimal), it is highly likely the top-left hasn't been adequately explored in training, some Q values in top-left may remain still(as per initialised) and they may all lead to actions going to the left from the goal state.

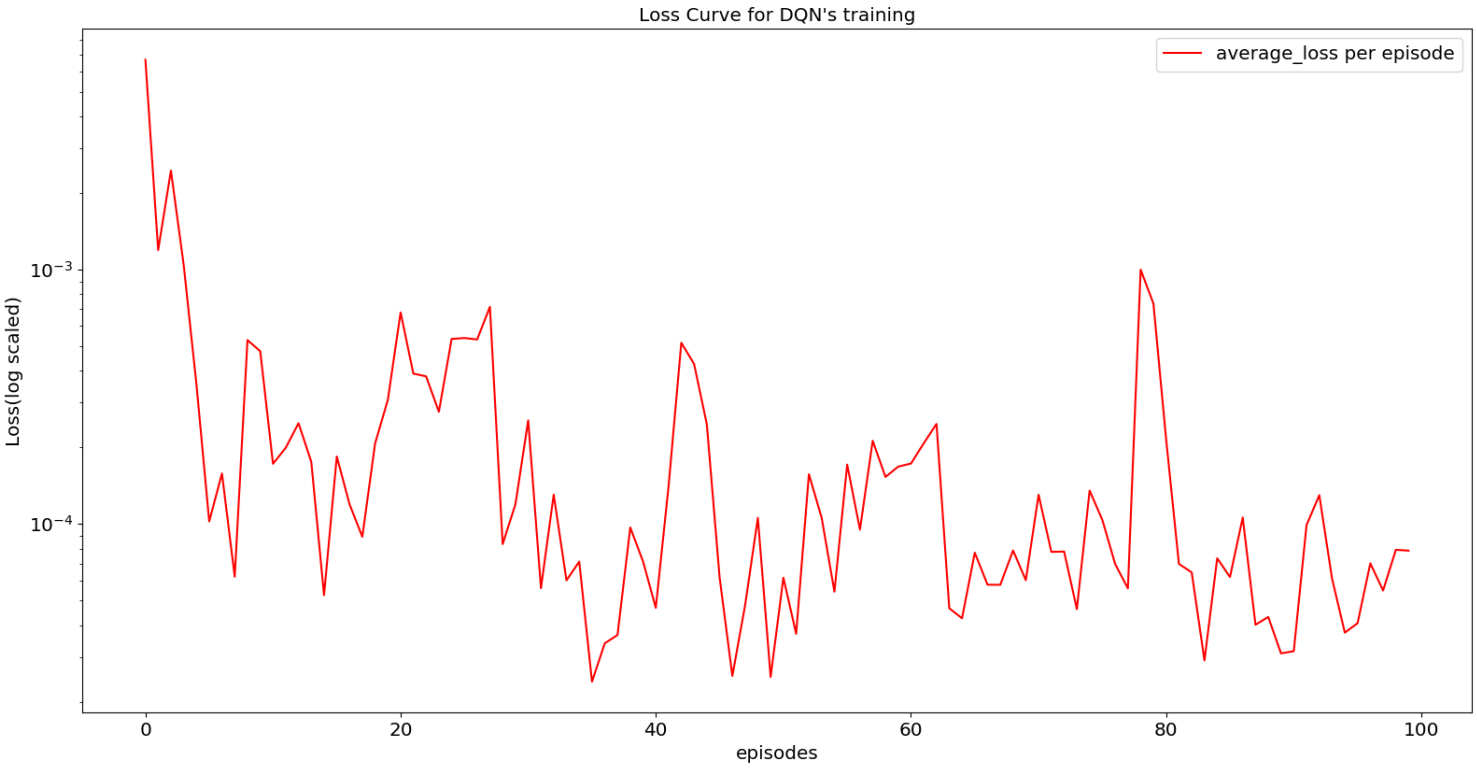
Loss Curve for DQN's training
average_loss per episode
Loss(log scaled)
10^{-3}
10^{-4}
0
20
40
60
80
100
episodes

Figure 1 (a)

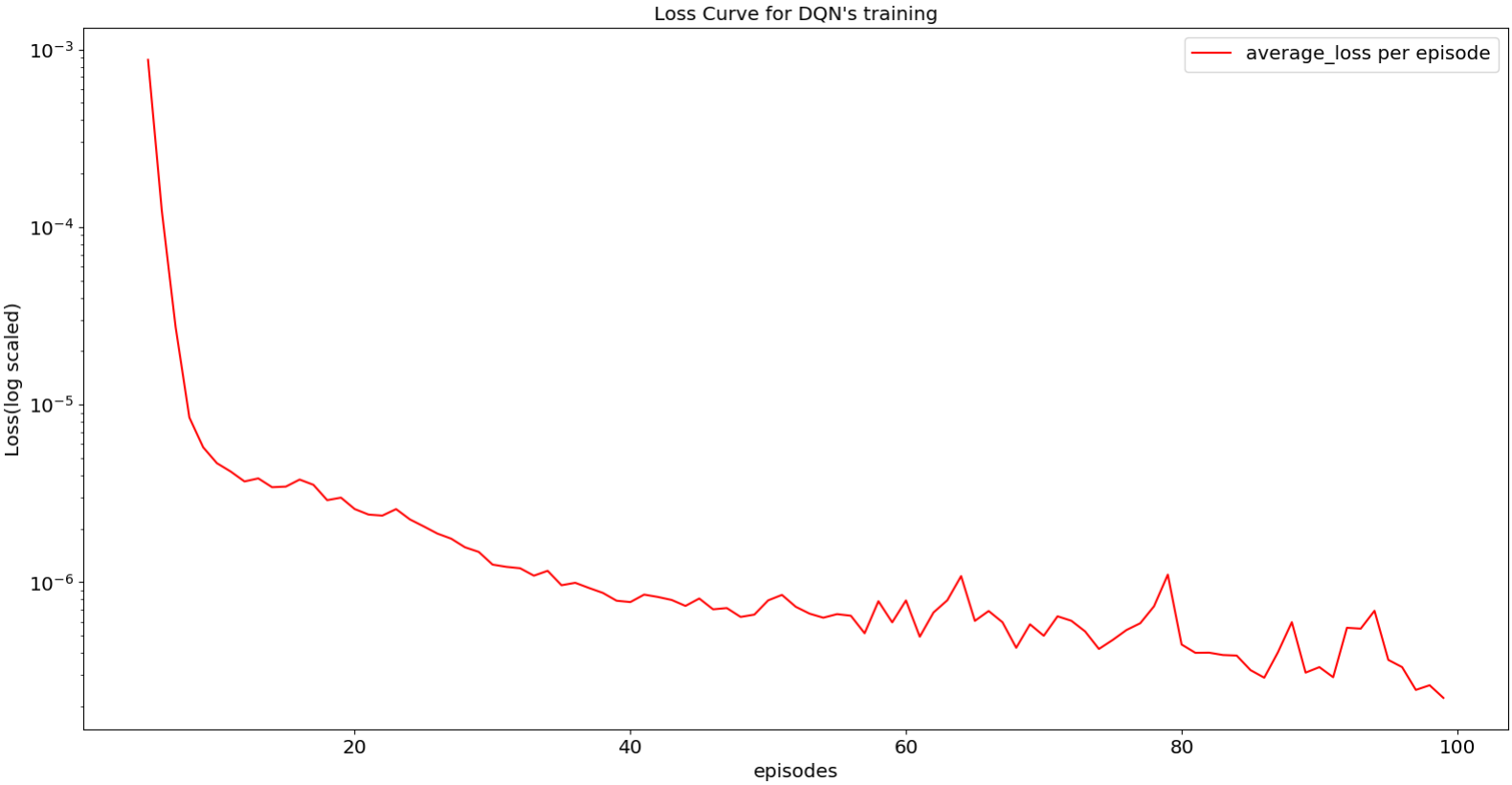
Loss Curve for DQN's training
average_loss per episode
Loss(log scaled)
episodes
10^{-3}
10^{-4}
10^{-5}
10^{-6}
20
40
60
80
100

Figure 1 (b)

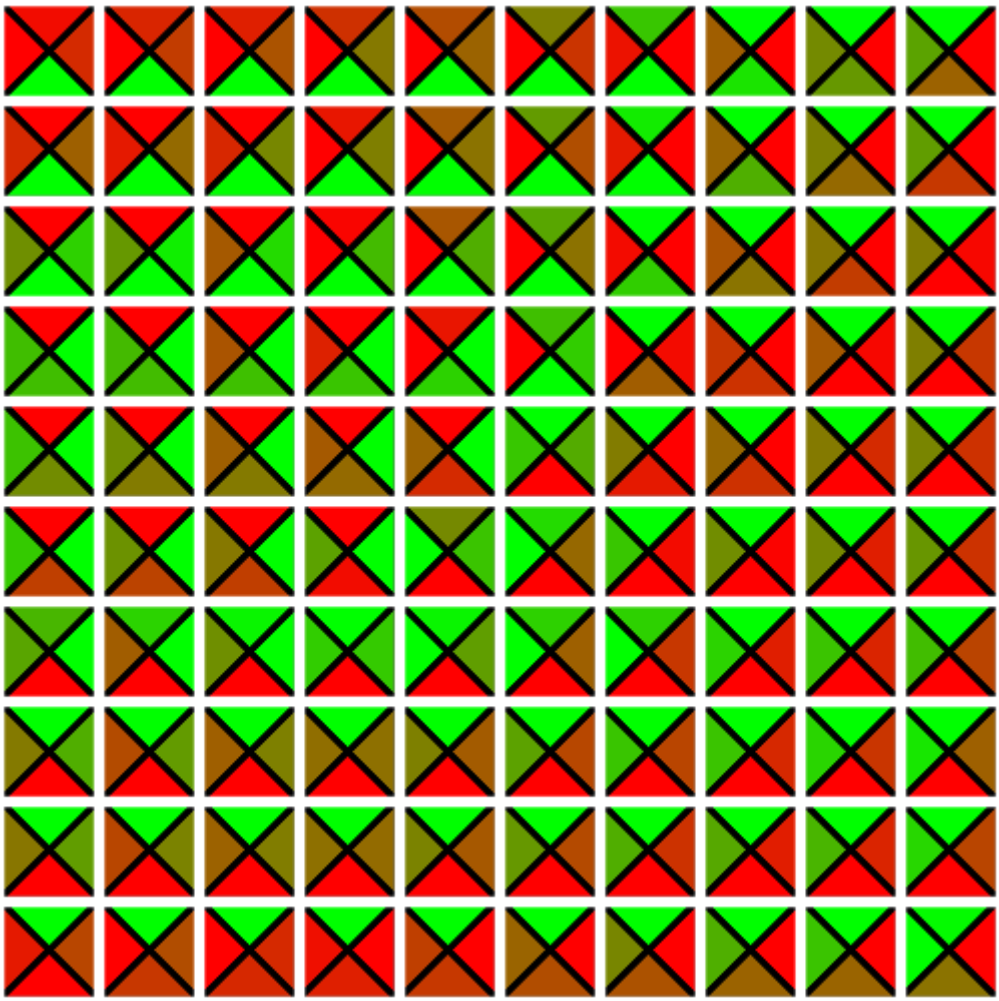

Figure 2 (a)

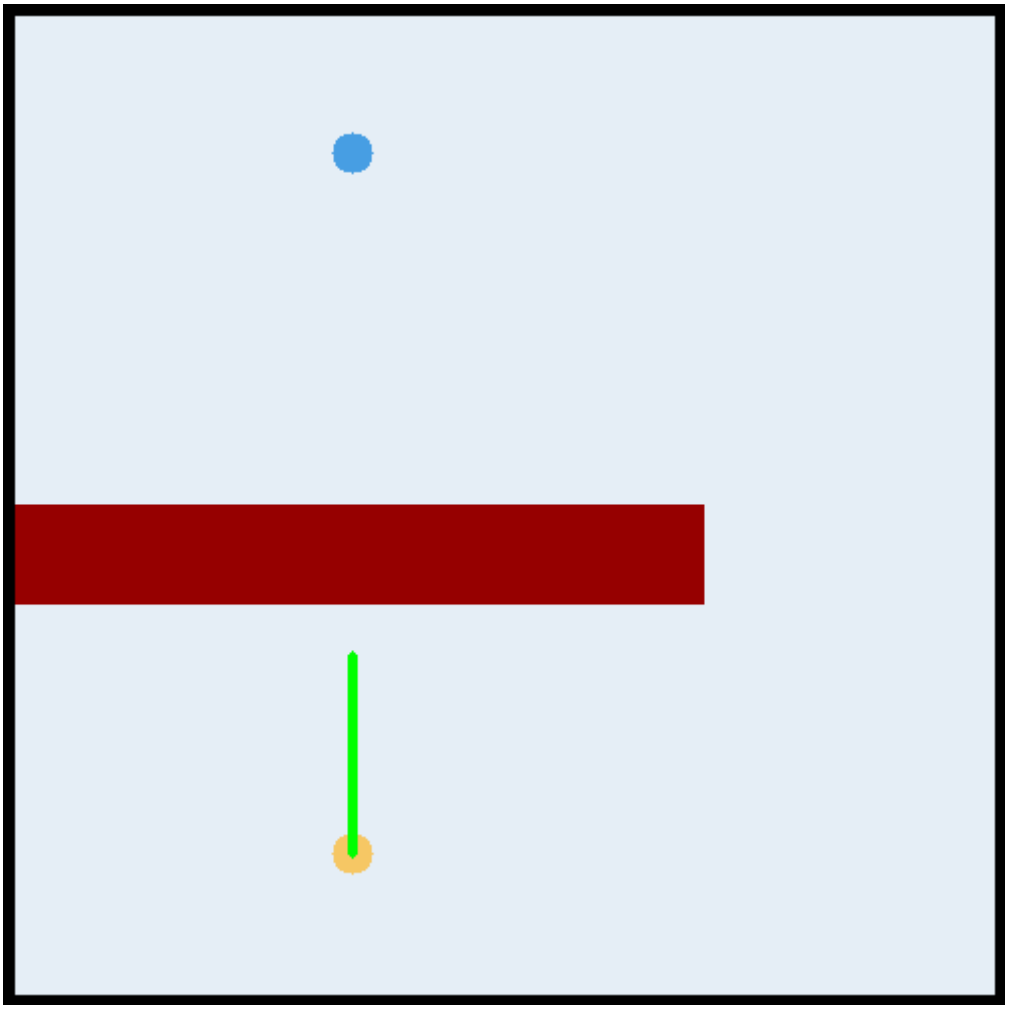

Figure 2 (b)

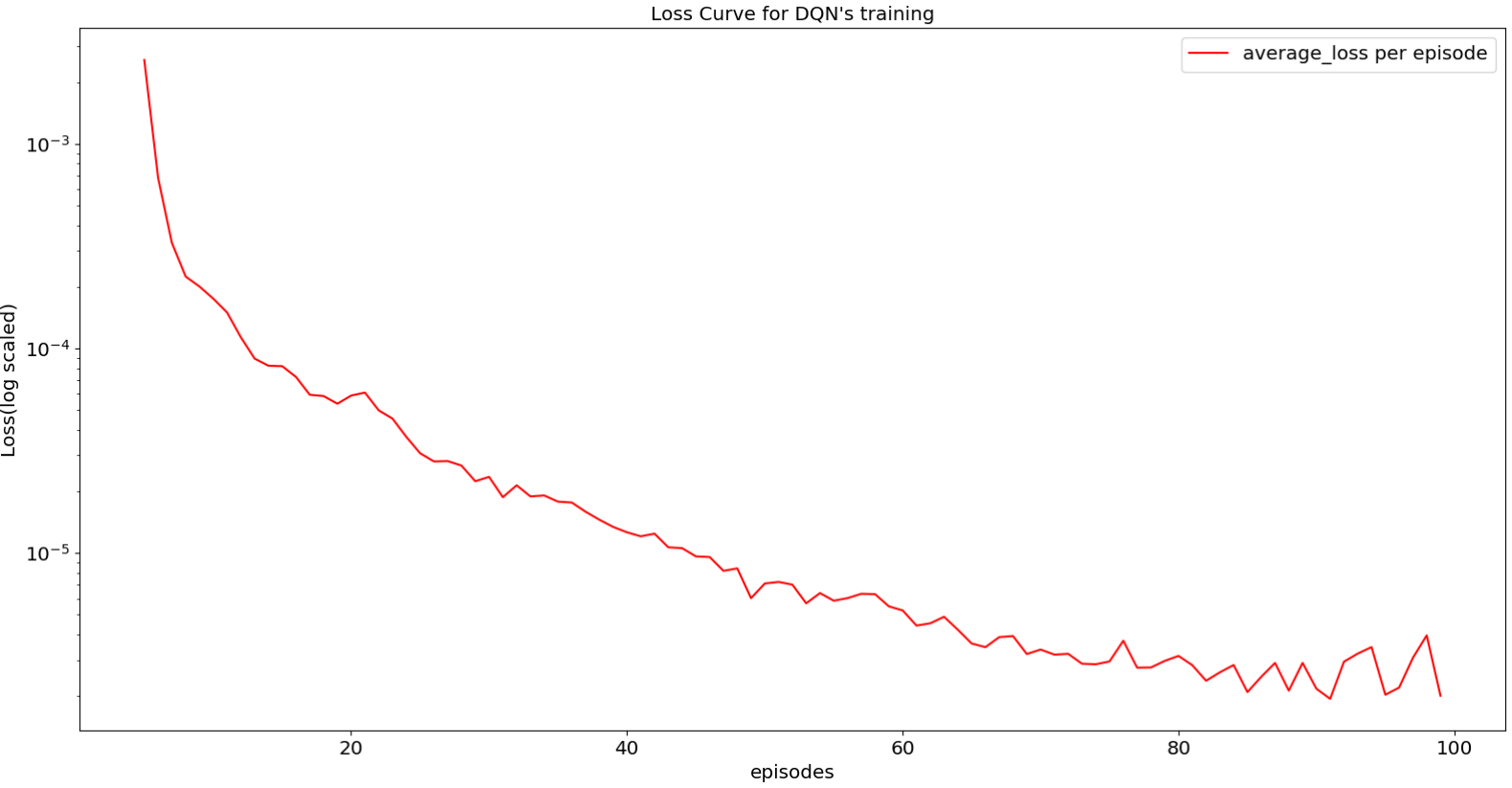
Loss Curve for DQN's training
average_loss per episode
Loss(log scaled)
10^{-3}
10^{-4}
10^{-5}
20
40
60
80
100
episodes

Figure 3 (a)

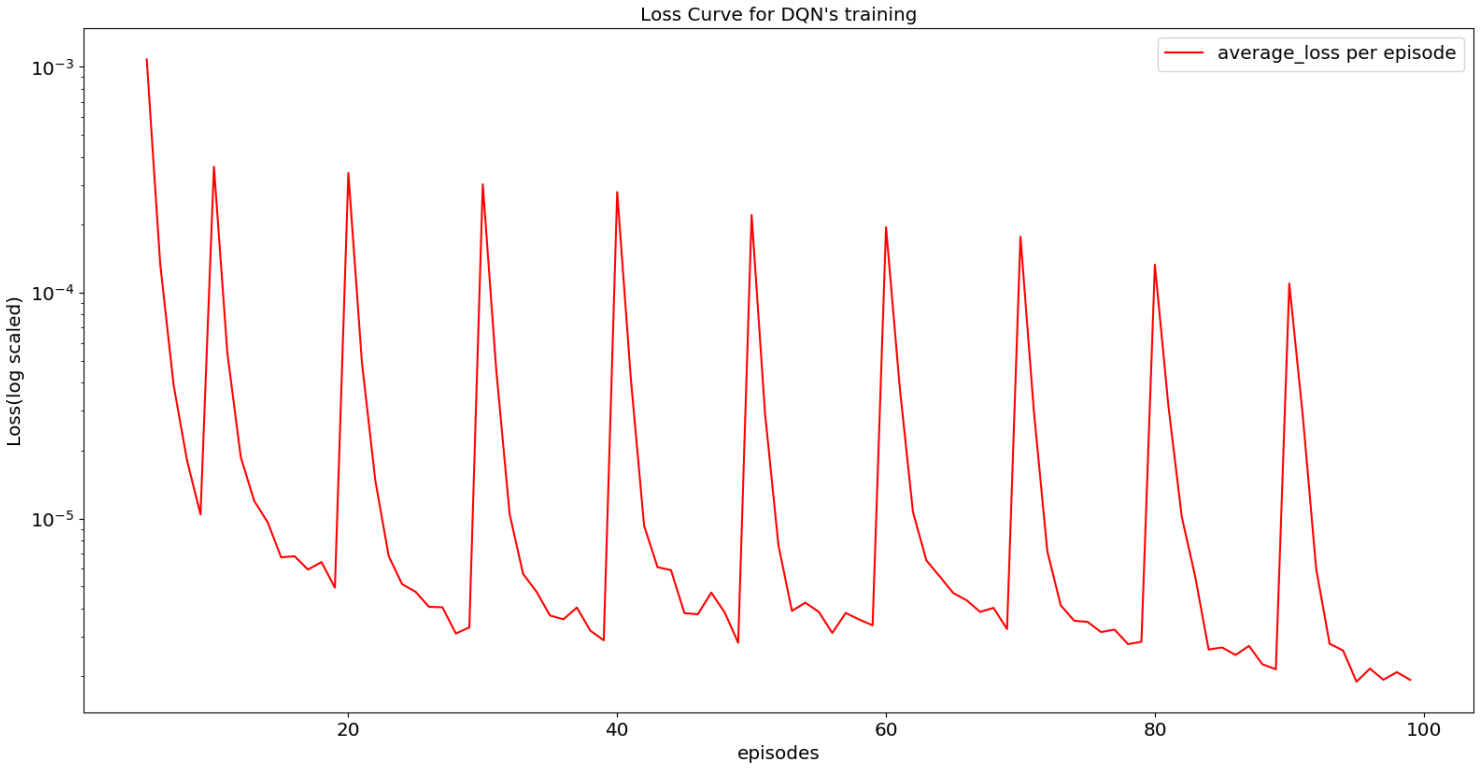
Loss Curve for DQN's training
average_loss per episode
10^{-3}
10^{-4}
10^{-5}
Loss(log scaled)
20
40
60
80
100
episodes

Figure 3 (b)

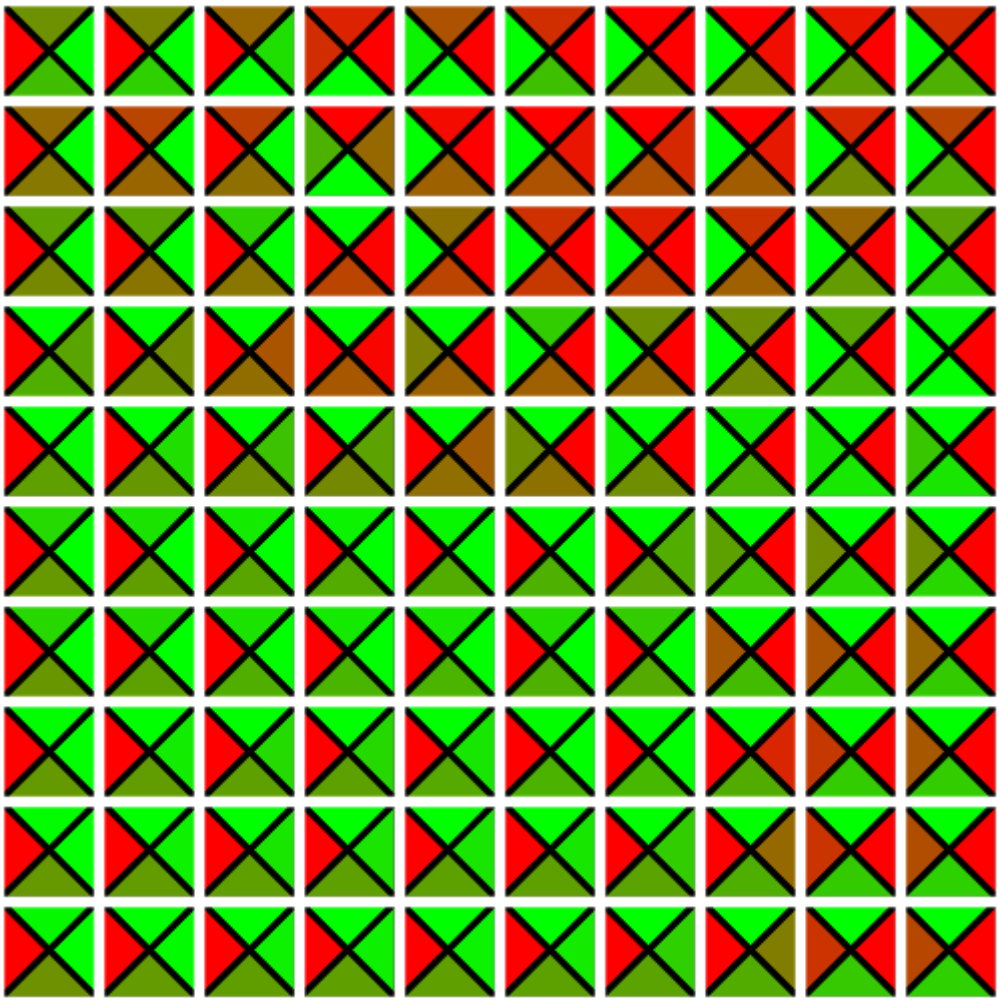

Figure 4 (a)

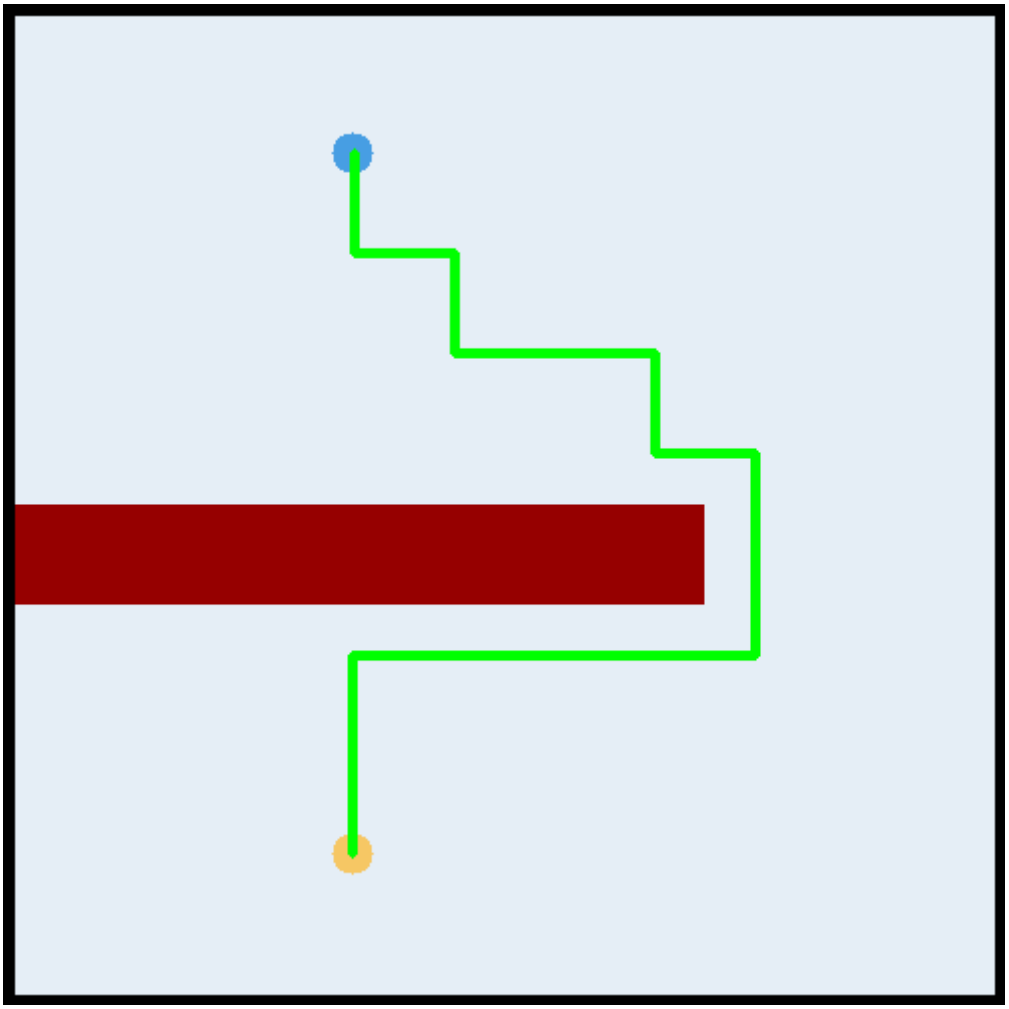

Figure 4 (b)

Description of Implementation for Part 2

At *line 213*, the `Network` class is defined and it basically shares the same structure as the tutorial implementation, with the same 2-hidden-layer configuration and even the same layer sizes(neuron count per hidden layer) and same activations per layer. The input data dimension is by default 2 and the output data dimension is dependent on the action space design which will be covered in the later sections.

At *line 233*, the `DQN` class is defined. Target network along with experience replay(*line 304*) are used to improve stability as danger of instability and divergence arises when combining a "deadly triad" of functional approximation, bootstrapping and off-policy training. High γ is used to encourage long-term rewards. At *line 265* in the `_calculate_loss` , transitions in terms of minibatches are converted to tensors and reshaped for input to the network forward pass, and the Bellman update target is estimated by the target network for computing loss. At *line 293*, the `predict_single` uses the trained DQN to make prediction on a given state, return its Q values as numpy array, note the `detach` method is used to allow numpy conversion of a tensor with require_grad on. The actual training takes place inside `Agent` class as training takes place every step, from *line 166* the transition data is collected and fed to the buffer which has max size of 5000 and minibatch size of 100. Then it checks whether transitions are enough to make a minibatch(enabled by *line 320*), if true then a minibatch can be randomly sampled from the buffer(enabled by *line 313*), for the DQN to be trained on. *line 180* enables the target network update for every 250 steps taken.

In the `Agent`, first, the action space of 4("right","up","left","down") is specified in the mapping dict at *line 27*. This is a class constant created for dual purposes: defining action space and allowing faster discrete to continuous conversion(using dict constant instead of function). The reward function(*line 27*) giving non-negative rewards is specifically designed, taking advantage of the fact that goal is always on the right of starting point. For left-half the rewards always drives agent to the right, passing the mid-line agent is more precisely driven by distance rewards which are guaranteed to have larger values than the left-half, effectively enticing agent moving to the right-half. Epsilon greedy is used for action selection in training(*line 188*). Every episode epsilon is decayed(*line 71* and episode number recorded. Every 10 episodes of training the agent will adopt the greedy policy(*line 84*) and do a complete greedy episode without training(*line 135*), to check if the greedy policy in 100 steps can reach goal already, if true then the training flag is set False(*line 152* and the training stops for good. Within those greedy check episodes, the last distance to goal from the previous greedy check is stored(as allowed by Ed on piazza) and compared with the current one, if current last distance deteriorates or remains the same, increases the epsilon back up to allow more exploration. The check greedy frequency is also doubled 5 minutes into training(*line 144*). Finally in the last minute if greedy check still could not reach target, then to avoid sudden drop in distance to goal(caused by instability in update especially for the starting region), the training can be terminated if a decent distance value is still achieved(*line 157*).