By: Harry Lording (717030990012) & Andrew Cain (717030990013)

HW#: 3

October 31, 2019

**Part 1. Reinforcement Learning in Maze Environment**

Dyna-Q Learning

My Dyna-Q agent used actual experience and hallucinations to update its Q Table values. While my agent was able to consistently find the optimal policy in maze1 it struggled to always find the optimal policy in maze2. While my Dyna-Q seemed to function as expected, I couldn’t formulate a good combination of parameter tuning and exploration-exploitation strategy to consistently find the treasure in maze2, after the exit had been found.

1.1

At its simplest my agent traversed the map, updating its action state pair q-values with one step look ahead. I one problem that regarded the implementation of Dyna-Q that was separate Exploration-Exploitation Strategy issues that impacted the performance of the agent.

The problem I encountered in implementing a working Dyna-Q was only having a single state for each position on the map, in which information of whether treasure had been found or not did not determine whether the agent was in a different state. This presented problems, in particular when the agent found the treasure, the agent would oscillate back and forth as if the treasure still on the map. This would occur until the q-values had decreased enough through q-table updates to the point where exiting was more desirable. While this worked, it was problematic as the values to the treasure decayed into the next episodes. This was simply solved by doubling the number of states on the board, in which every position would have two states, one for whether the treasure had been found and vice versa. After making this alteration, once the treasure was found, the agent would not consider travelling back to where the treasure use to be as a desirable action as that position was considered a different state to when the player didn’t have the treasure.

1.2

The aspect the impacted the performance of my agent the most was the exploration strategy.

The first problem I encountered was the degree to which my agent should hallucinate. At first, I assumed that the more hallucination of q-table updates the better. It was faster than actual experience and could propagate q-values that I had found across the entire board. However, with this strategy if I found the exit before the treasure early on in training the agent would learn the path to the exit too quickly. If the treasure wasn’t found shortly after the agent would be too enticed to go towards the exit. Even when exploration strategies hadn’t decayed, the agent still in general went to the exit. As a result, I toned down the number of hallucinations per episode, so that q values still propagated to other states but more slowly over time.

For my exploration strategy I decided to use an exploration function in combination with random exploration. The agent would receive a bonus, k, if it hadn’t seen the state action pair before when it updated the q table value. The bonus then would propagate back to other states through hallucination, encouraging the agent to explore those areas further in future episodes. After each time the action had been under taken, the bonus it received would be divided by the times it had done that action, thus decaying this bonus. While this is what I expected in theory the end result differed significantly.

Because the bonus was propagated in the q table, the bonuses remained in the q values over the course of the episode. Additionally, the bonuses from previous states would compound with each step the agent took, and as a result, states that took more steps to get to had higher values. Because of this the agent tended to drift down the left hand side of maze1, as it could circle around in the open area, compounding the bonus reward until it had reached the bottom left hand corner where it would get stuck and have no incentive to move. The agent would then leech bonus off its own state to increase its q-value. As a result of these numerous issues I decided to not allocate the counter bonus in the q value but rather give actions a bonus when the agent was deciding which action to choose. While this was more successful than the previous strategy it had its own problems.

The problem with giving not updating the q values with a bonus was that the effect of propagating value back from states that had not been seen often couldn’t be done. The agent could only choose the actions in its current state that hadn’t been chosen many times, but the agent didn’t know which actions would lead to states in hadn’t seen many times.

****

**Maze1 performance and convergence**

****

**Maze2 performance and convergence**

**Part 2. Reinforcement Learning on Atari Game**

\*co-authored by Andrew Cain (717030990013)

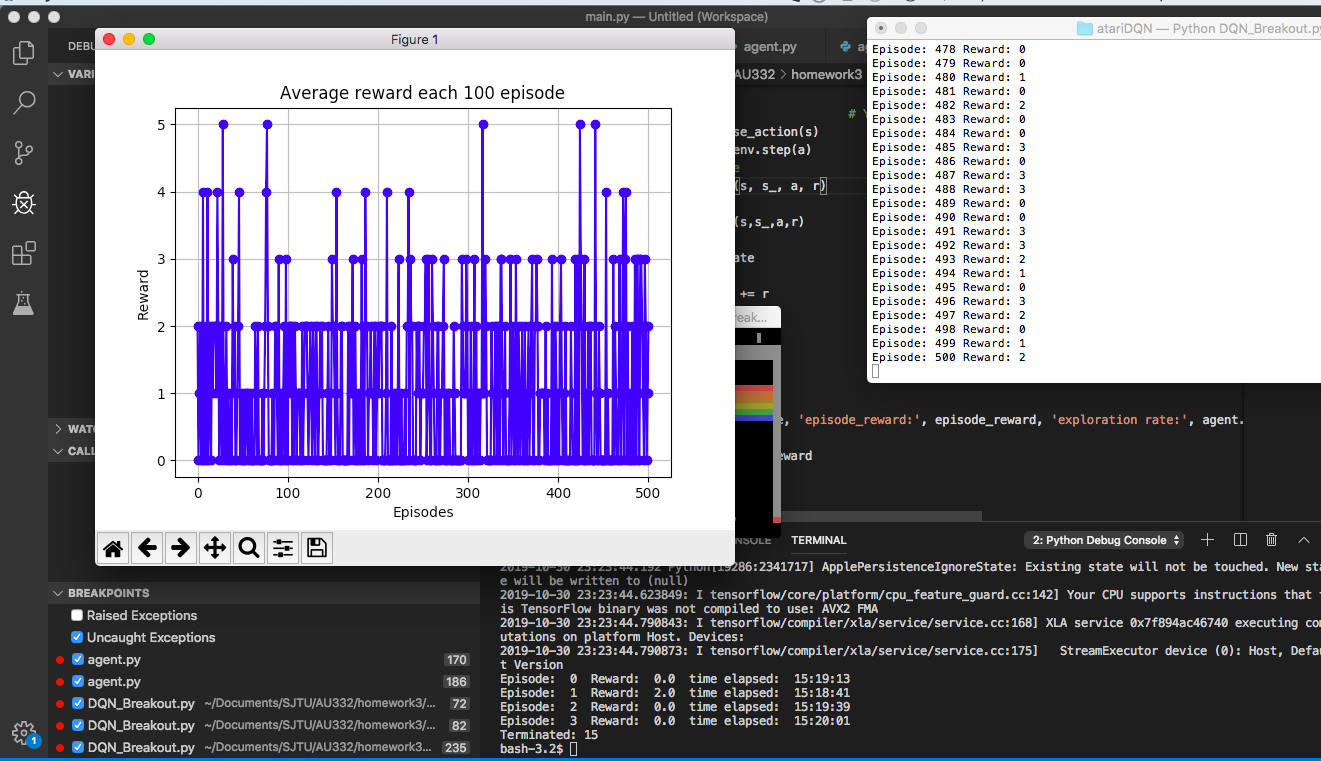
**Reinforcement Learning**

Overall this exercise presented many problems. Initially we thought that the actor critic architecture was necessary to complete this task. This led to many complications such as how to implement gradient ascent, with particular confusion on how to find the derivative of the log of p(a|s,theta), and the advantage function. It was also considered whether to use a model consisting of two neural networks, each for the actor and the critic, or to use a single neural network that shared the parameters for both agents. In the end we realised that a simpler implementation would yield better results, given the limited time period to complete the assignment.

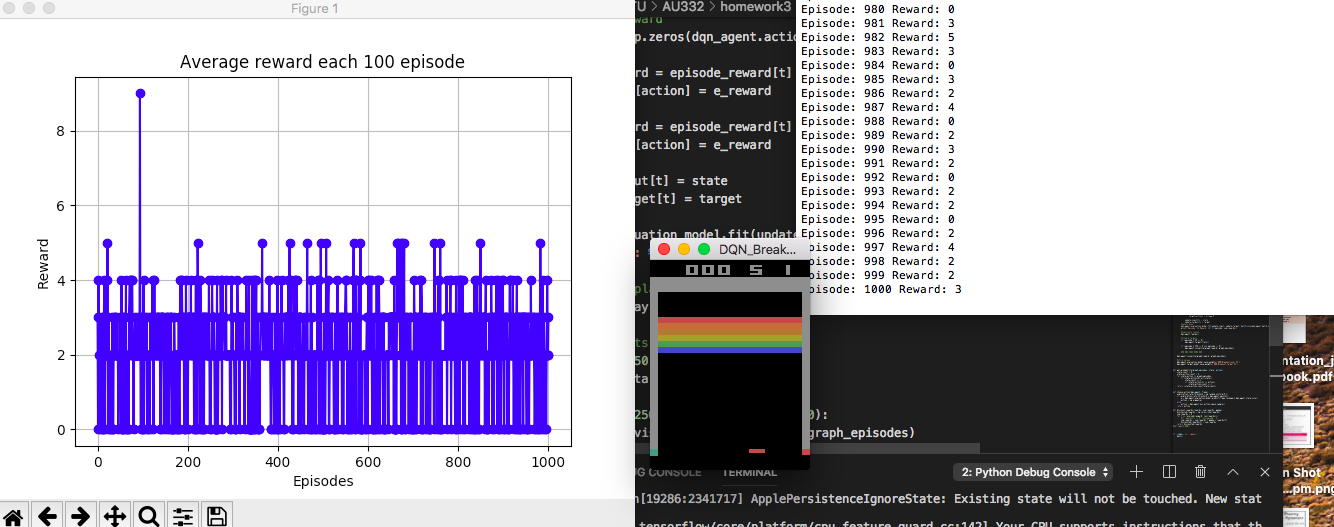
Our final code uses the default neural network structure given for the assignment. The network consisted of a 4 layer structure, the first 3 layers having 128\*2 neurons using relu activation function and the last output layer using linear activation. Modifications to the number of layers, number of hidden units per layer, activation functions, optimizers, learning rate, batch size and epochs were tried however none leaded to materially better results in the rate of training.

While the neural network did show signs of improvement, no tuning to the network seemed to improve the rate of training significantly. Higher batch sizes were tested in an attempt to increase the rate of convergence however this learning seemed to become slow. Higher number of epochs was also tested to try to fit the neural network faster to the Atari game however no signs of improvements were noticed. Alterations to the learning rate were also tested that would be optimized for RMSprop however no material improvements were noticed. Though the network did improve, the efficiency of training didn’t seem to change after tuning.

Below is the results from training. Weights were saved so training could be done over time and not in one execution of the program.



Above: Training at first over the 500 iterations, averaging a score of 2 with occasional scores of 4 ~ 5.

****

Above: Training over the next 1000 iterations, averaging a score of 3 with higher chances of scoring 4 ~ 5.

A screenshot of a cell phone

Description automatically generated

Above: Training after another 500 iterations, averaging a score of 3 with higher chances of scoring 4 ~ 5 and now sometimes scores reaching 9.