Shayan Mandegarian
914100795
ECS 170 Program 3

1 Problem Representation

1. I think it would be easier for the agent to learn from the hardware ram, for a couple of reasons. Firstly, I believe if it were to learn from the game display images, it would then have to interpret the image in some way, likely searching for the white spots in the images corresponding to the ball and the paddle, then transforming that information into an array of some sort. Thus, if we simply use the array provided by the ram, we can cut out that unnecessary step. Secondly, if choosing to learn from the ram rather than display would allow the computer to not have to render the display in the first place, it could make execution faster, and allow the agent to learn in a more time efficient manner. However, this is under the assumption that the way the game stores it's information in ram is in an easily accessible way, like an array. Thus, if a game has overly complicated memory storage that makes it hard to get the information required for training, it could end up being easier to use image processing on the game's display.

2. In Q-Learning, if there is a very large amount of state/action pairs, storing each q-value of each pair would be infeasible. Instead, a neural network can be used to approximate the q-value of a given state, which will then be able to make better and better approximations based on the rewards it gets for its actions. The inputs mainly include the current state, which is calculated based on either the game display or ram, a given epsilon value which is used to determine whether or not to do a random move, and "self" which contains many different things, like the "action space." From those inputs, it will return an action output, which is used to actually play the game. In this case, the input is not directly from the game's display output, but consists of multiple different possible actions to choose from, and the current "state" of the game. It will pair up the state with the possible actions it can choose from to create the aforementioned state/action pairs which it will assign Q-values to.

3. One problem that can arise with Q-learning is it may "overfit" to a certain pattern of actions, and if something changes about the game, like an opponent behaving differently, it may not be able to win anymore. Thus, some degree of randomness can help with exploration. Exploitation and exploration are two main goals of Q-learning, so while not having an element of randomness can maximize exploitation, it would miss out on crucial exploration. So, finding a balance between the two by tweaking the epsilon value is important.

2 Making the Q-Learner Learn

From my understanding of this loss function, it is the square of the "model output", or q-value, of a specific action in the current state, minus the q-value of the current state. As time goes on, this value will approach zero, and the closer it is to zero, the better trained the model is.

Therefore, it can be used as a way to determine how well the model is learning. As stated in the hint, Θ corresponds to the model parameters and y corresponds to the model output, while Q() corresponds to the q-value function, which will give the Q value of a given state/action pair, where s,a refers to said pair. When the subscripts are added, Θ_i and y_i refer to the model parameters and output of the i-th iteration respectively.

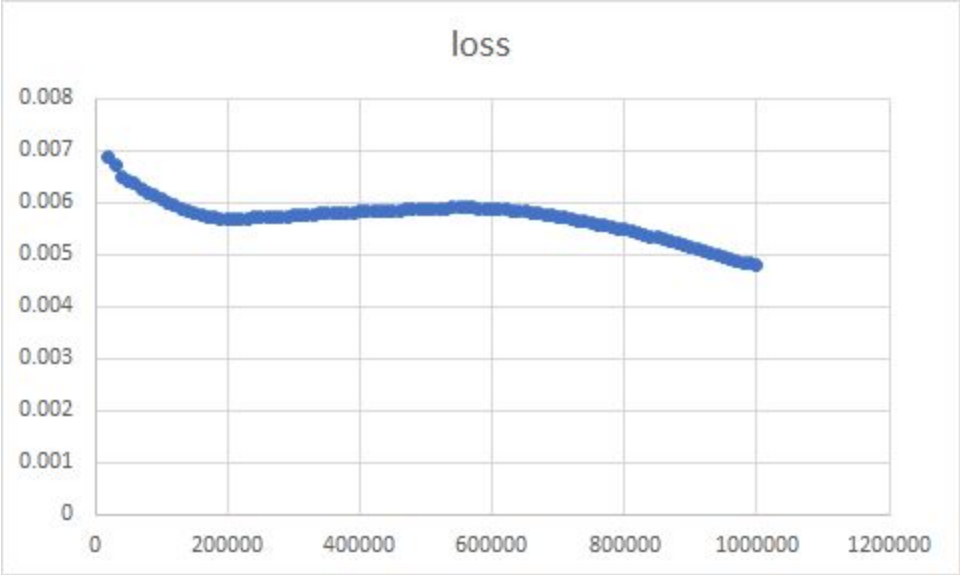
In terms of the actual arguments being passed to the "compute_td_loss" function, those being model, target_model, batch_size, gamma, and replay_buffer, I will explain each one next. Model is the current environment that is being trained in, and it is used to find the Q-value of the state and next_state variables that will be explained later. The target_model is similar, but rather than storing the current training environment, it stores whatever environment was last saved as the target_model within the run_dqn_pong.py file. Batch_size and replay_buffer go hand in hand, as batch_size simply controls the size of the random sample fetched from replay_buffer, which returns tensors for the "state", "action", "reward", "next_state", and "done" variables, which are ultimately used to actually calculated the given loss function. Gamma is simply the provided gamma value from run_dqn_pong.py, which is used to control whether the Q-learner tries to get short term greedy rewards, or long term rewards that might ultimately be higher. In my case, I experimented with a few different gamma values, but found that the original .99 value to be the best performer.

While not given to the loss function as arguments directly, the "state", "action", "reward", "next_state", and "done" variables mentioned earlier are very important. The simplest one is done, which determines if the game is finished or not, and if it is, the q-value should be zero since the next game doesn't affect the current one.

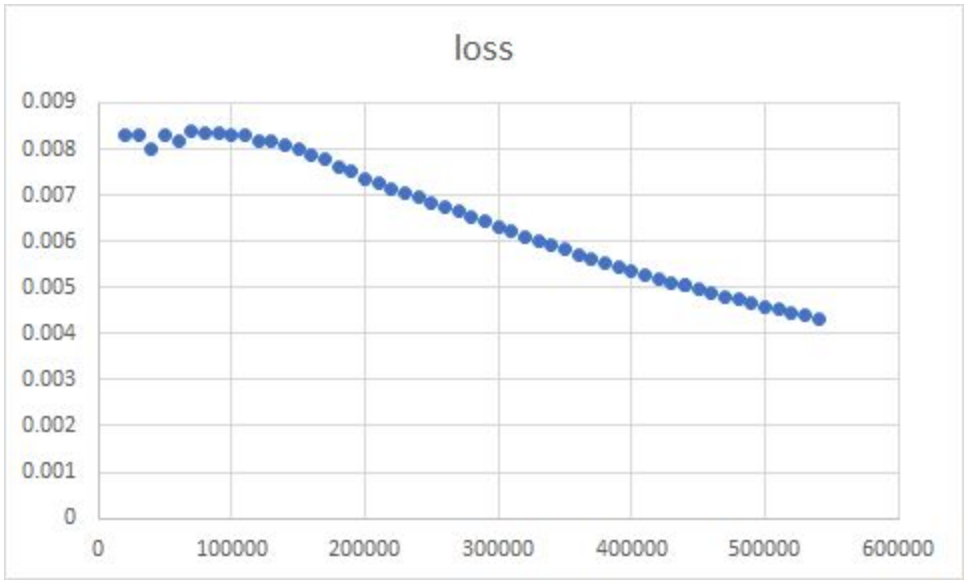
4 Learning to Play Pong

The first few times I tried training my model, I noticed that it would start around -20 average reward, go up to around -18, then go back down to -20 after around 120k frames, at which point I would Ctrl-C and try making changes since I assumed it wasn't working. One of the best changes I made to prevent it going back down in reward averages was to change the way I was saving the model. Rather than simply saving it every 5k frames like I had it at first, I made it so it would save the maximum average reward for the last 10k frames, (the same value it prints in the console) and whenever a new maximum average was achieved, only then would it save the model. This ended up working very well and after 1mil frames it achieved an average reward of 19.5, and when running test_dqn_pong.py it won every game but one. However, after running another training session on that same model, I found the rewards were increasing much faster than the first one. For instance, on the first session, it had achieved an average reward of -14.3 by frame 180k, but during the second session it achieved an average reward of 7.3 in the same number of frames, whereas it took 640k frames for the first session to achieve a 7.2 average. Admittedly, I did not implement having run_dqn_pong.py save the loss and reward data into memory to graph them, as I was more focused on getting the other functionalities. So, I took the data from the console, converted it into .csv files, and used Excel to generate the following plots for both training sessions.

Initial Training Session Graphs:



Second Training Session Graphs:
For the second session, the loss decreased and the reward increased at steeper angles, and required less frames to achieve similar values as the first session, so I only needed to run it for about 500k frames, here are the plots:


loss
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
100000
200000
300000
400000
500000
600000


reward
25
20
15
10
5
0
-5
-10
-15
-20
-25
0
100000
200000
300000
400000
500000
600000