1.1

It would be easier for the agent to learn from a state as an image, compared to an array. This is because every possible state transition must be mapped onto the array, which would exponentially increase the required space and time. Instead, it is easier to map a state transition by a series of frames, which would limit each state to several frames of 2d-arrays. After pre-processing these frames (such as scaling), the number of states become severely reduced.

1.2

Deep Q-networks receive the state as an input and returns Q-value actions. With this, we can find which Q-value was the greatest and choose the corresponding action. Specifically, in the dqn implementation, the input variables were `env, num_frames, batch_size, gamma, replay_buffer`.

`Env` represents our environment. By taking an action on it, it would change to another state and return a given reward. `num_frames` is the maximum number of frames we allow our model to train. `batch_size` is how many elements (i.e. state transitions) we sample from our replay buffer to update the model's weights. Note that the entire batch must be inputted before the weights are updated, meaning the weights aren't being adjusted in between. `gamma` is the discount rate, which determines the importance of future rewards. Having a lower discount rate would make the agent put less value on future rewards, and vice-versa. Finally, `replay_buffer` is a buffer that allows our network to learn from games without introducing a lot of correlation. Because the network is sampling randomly from the buffer, the agent won't learn any "bad trends" from a sequential series of states. The output variable, `x`, is size `batch_size`, and each sample contains the Q-values for each action.

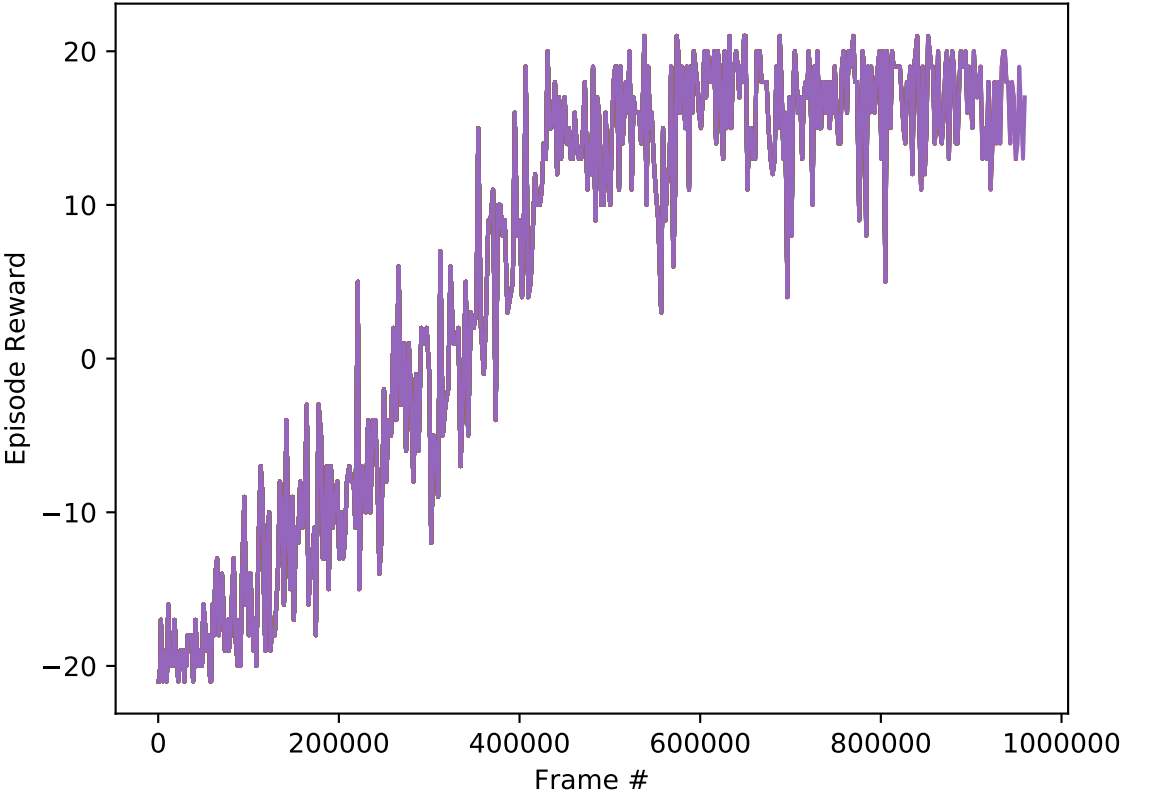
1.3

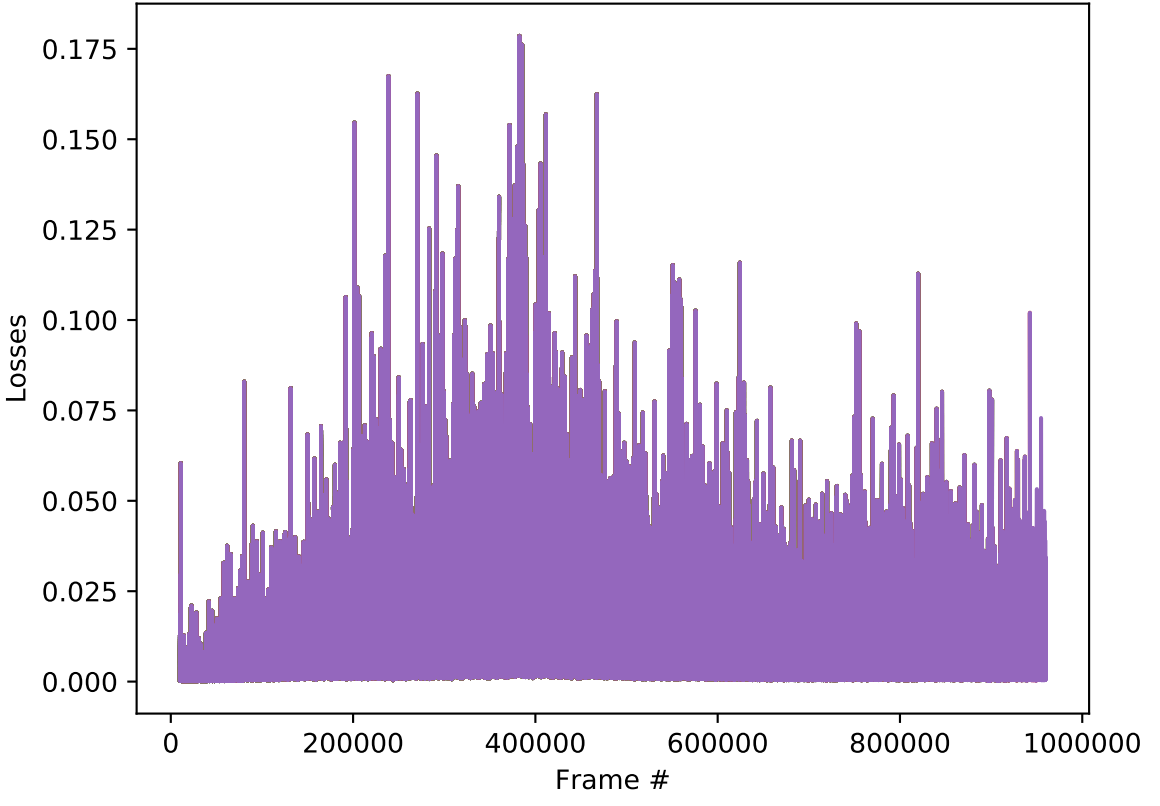
The purpose of those lines is because we want to begin training by exploring. The reason the agent do this is because if it only were to exploit (i.e. epsilon = 0), then it would avoid making risky, but potentially long-term rewarding actions. It would only take short-sighted actions that would lead to minimal rewards. `if random.random() > epsilon:` tells the agent to exploit, while the else condition tells the agent to explore (given by the `randrange(self.env.action_space.n))`. `epsilon` decreases exponentially after a certain number of frames, at which the agent will essentially only exploit. Technically the q learner does tell us what action to take, but it knows nothing about the best action to take during the initial stages of training. That is why it's important to "ignore" the Q-val actions given at the initial training stages.

2.1

y_i is the model output, or the Q value for the state, action pair using model (i.e. the Q value that model says it is for the action). It is obtained by `reward + gamma * max(Q(s', a'))`. On the other hand, `Q` is the target output, or the Q value that the update rule says it should be using target_model. `s` and `a` are the state and action taken at the `ith` iteration. This loss function helps our model learn because it is much more efficient to use neural networks to compute Q-values compared to regular Q-learning. A game could be represented by a million states, which would clearly be too large for a regular Q-table. Instead, we use Deep Q-learning to

estimate the Q-value. Because this is an estimate, we want to minimize the difference between the model's output and the true Q-value. Because Q-values are recursively defined by themselves, we can simply solve the current state in terms of the next state, which is `y - Q(s, a)`. Using this loss value, we can backpropogate the error and adjust the weights accordingly. By calculating the loss gradient, we know how to adjust the weights. After enough training, the weights would be adjusted enough such that the Q-network can accurately estimate any Q-value (for a given state and action).


20
10
0
−10
−20
Episode Reward
0
200000
400000
600000
800000
1000000
Frame #


0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
Losses
0
200000
400000
600000
800000
1000000
Frame #