Reinforcement Learning: Deep Q-Network

Tom Gotsman

Online Learning and Mini-Batch Learning: Figures 1 (a) and (b)

Question 1 (i): Inspect the variance of the loss in each graph. State which of these two meth- ods results in the most stable training, and explain the reason for this.

Online learning has a variance many orders of magnitude larger than the approach using an experience replay buffer (ERB), a biologically inspired mechanism. For online learning the loss of each step is a loss for a single transition, however for the ERB the loss is averaged over the mini-batch. Online learning results in subsequent training samples being highly correlated, meaning that the distribution of the data used to train the Q-network changes rapidly. This results in different parts of the Q-network being trained on and other parts which have not been trained on for a while to be overwritten. Using an ERB leads to more stable training because instead of sampling transitions in the order they were collected, we sample random mini-batches. This in turn maintains a more uniform data distribution across the environment, leading to the entire state-action space being updated all at once.

Question 1 (ii): State which method is more efficient at improving the Q-network's predictive accuracy per episode of interaction, and explain the reason for this.

One of the main limitations of online learning is that each transition is only trained on once and then discarded. However neural networks require many updates per transition, meaning that online learning requires the agent to revisit a state-action pair again and again to train on that same transition. The experience replay buffer solves this efficiency problem by creating a dataset containing all the data the agent has experienced so far. This means that the network can go back and re-visit some of the data and retrain on it without having to recollect the actual data. Overall this means that using the experience replay buffer is more efficient at improving the Q-network's predictive accuracy per episode of interaction.

Visualisation of Q-values and Policy: Figures 2 (a) and (b)

Question 2 (i): Inspect the Q-value visualisations for the top-right and bottom-right of the environment. Is it likely that one of these two regions will have more accurate Q-value predictions than the other region? Explain your reasoning.

Inspecting the Q-value visualisation for the bottom right of the environment, in figure 2(a), we see that the action with the highest Q-value is always the up action. This is expected as the goal is up from this position. On the other hand, the most favoured action for the top right of the environment is also up. These Q-values do not appear accurate as the goal is to the left for these states. As the agent visits the states near the top of the map less often, we mainly fill the experience replay buffer with transitions from states at the bottom of the map, which leads to mini-batches being sampled with transitions mainly from the bottom and so when the entire state action space is updated it is more likely to favour actions that are beneficial in the bottom. In essence the knowledge generalisation of function approximation leads to the actions at the bottom of the environment, which are more accurate and sampled more often, to be favoured for states at the top of the environment.

Question 2 (ii): Inspect the visualisation of the greedy policy. If the agent were to execute the greedy policy, state whether or not it would reach the goal, and explain why this is.

Visualising the greedy policy, the policy that always takes the action with the max Q-value, in figure 2(b), we clearly see that if the agent were to execute this policy it would not reach the goal. The loss is calculated based only on immediate rewards for the agent and it never accounts for future rewards. The reward is defined as $r = 1 - d$, where d = distance from the goal. When the agent moves right or left from a state in line with the goal vertically, while still below the barrier, the distance it calculates will be further from the goal than it was before and so the action to move up will always have the largest Q-value in these states. When the agent is at the state in line vertically with the goal, which is nearest to the goal below the barrier, it will reach a local minimum as the action to move up will keep the agent in the same position and moving either right or left will lead to a lower reward meaning that the action with the largest Q value will be up and we get stuck here.

The Bellman Equation and Target Network: Figures 3 (a) and (b)

Question 3 (i): Now that you have introduced the Bellman equation, explain what the agent is now able to learn, that it was not able to learn before.

The function which computes the loss for the Q-network has been changed such that instead of computing the Q-network's error compared to the reward, now it computes the error compared to the expected discounted sum of future rewards. The Bellman equation: $Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$, finds the optimal solution of a complex problem by breaking it down into simpler, recursive sub-problems and finding their optimal solutions. The Q-network is used to predict the Q-values for the next state in each transition, and it then finds the maximum Q-value in this state, across all actions. Introducing the Bellman equation allows our agent to escape local minima and therefore no longer have the problem which I postulated in Q2(ii).

Question 3 (ii): Inspect the shape of the loss curves. Explain why the shape of the loss curves is different in the two graphs.

When we use a target $\hat{Q}$-network, figure 3(b), the loss function we define: $(r(s,a) + \gamma max_a \hat{Q}(s',a) - Q(s,a))^2$, has both the Q-network and the $\hat{Q}$-network as inputs. When the Q-network is training it is optimising the weights by minimising the cost function through implementing gradient descent. We optimise this loss function with respect to the the Q-network and the $\hat{Q}$-network, which is unchanging for 10 episodes. This means that when we update the $\hat{Q}$-network, after 10 episodes, we suddenly get a large spike as the parameters we have been optimising have abruptly changed and the network must adjust and re-begin the optimisation process.

Exploration vs Exploitation: Figures 4 (a) and (b)

Question 4 (i): If was always 0.0, is it possible that the agent could ever reach the goal? Explain your answer.

The ϵ-greedy algorithm makes use of the exploration-exploitation trade-off. Q-learning can identify an optimal action-selection policy for any given finite MDP, given infinite exploration time, which we assume here, and a partly-random policy. This was proven in F.S. Melo's paper "Convergence of Q-learning: a simple proof". If we always set ϵ=0 then we are at the deterministic limit, therefore we have no partly-random policy. In this case unless by complete chance the network weights are initialised to the global minimum, the agent should never reach the goal.

Question 4 (ii): After a period of training, but before the Q-network has converged, imagine that the agent executes the greedy policy, which results in the agent reaching the goal. But after reaching the goal, the agent continuous on in a straight line, and hits the wall on the left of the environment. Why might this happen?

As the network has not yet converged, the Q-values in the states to the left of the goal are not yet very accurate, as they have not been explored enough. This is turn can mean that when the agent executes the greedy policy even if the agent reaches the goal, the states to the left of the goal still have actions with the max Q value not correctly pointing towards the goal. When the agent reaches these states, if the Q values have not converged, the agent could follow the action with the max Q value in each state at the time and continue on in a straight line into the wall.

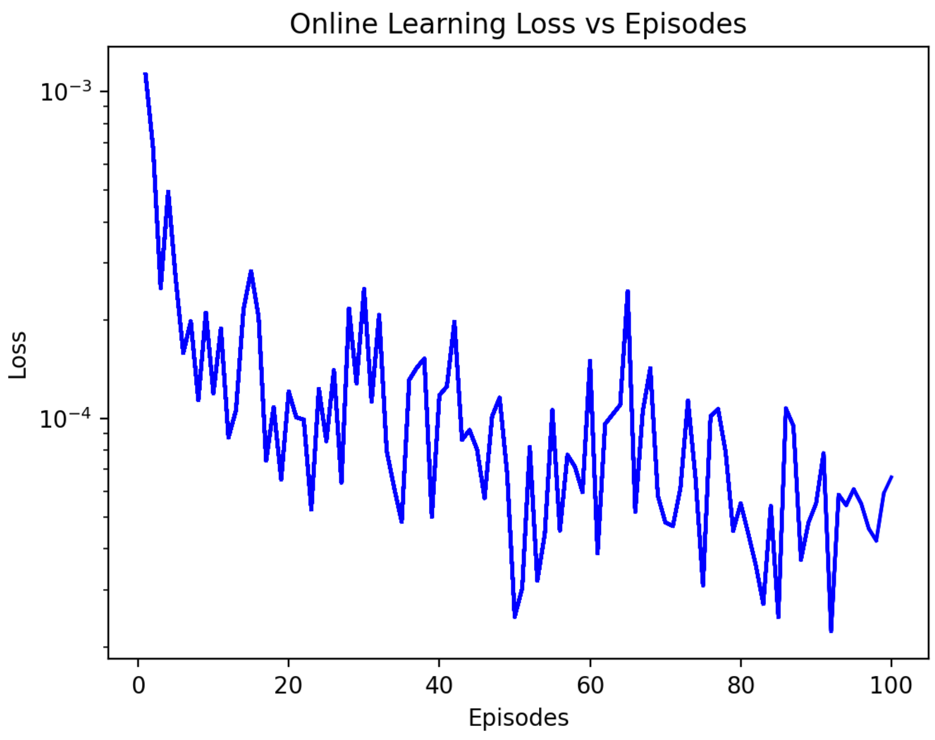

Figure 1 (a)

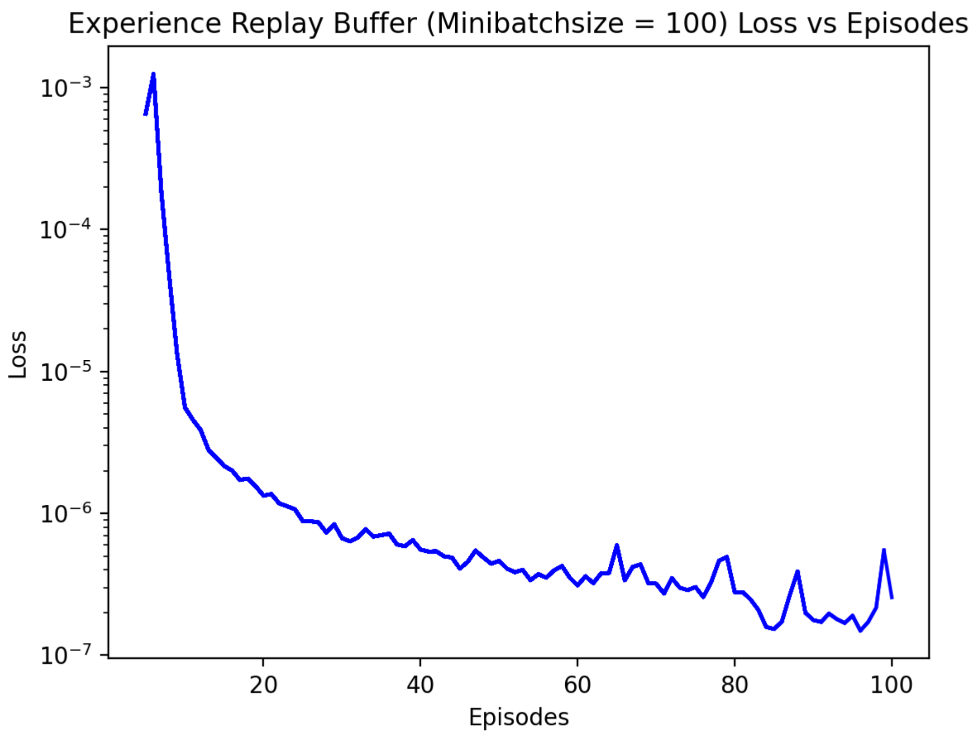

Figure 1 (b)

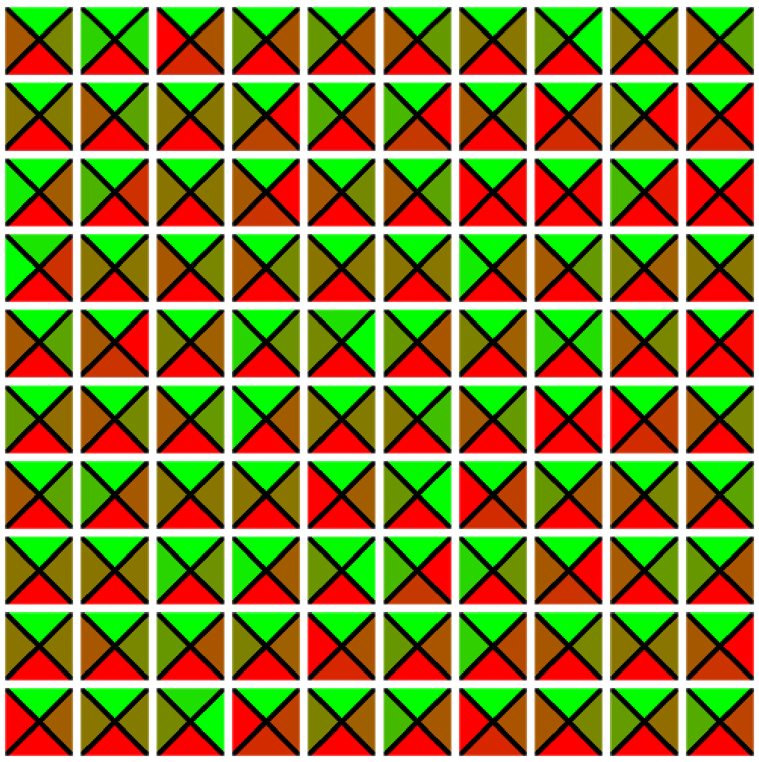

Figure 2 (a) Visualising the Q-values to show what the agent has learned after 100 episodes

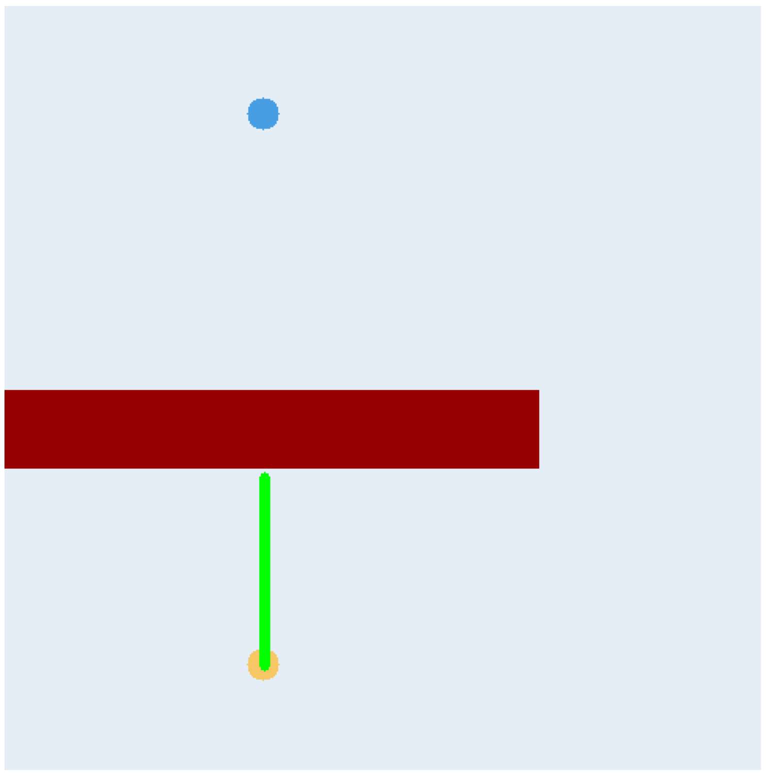

Figure 2 (b) Visualising the greedy policy to show what the agent has learned after 100 episodes

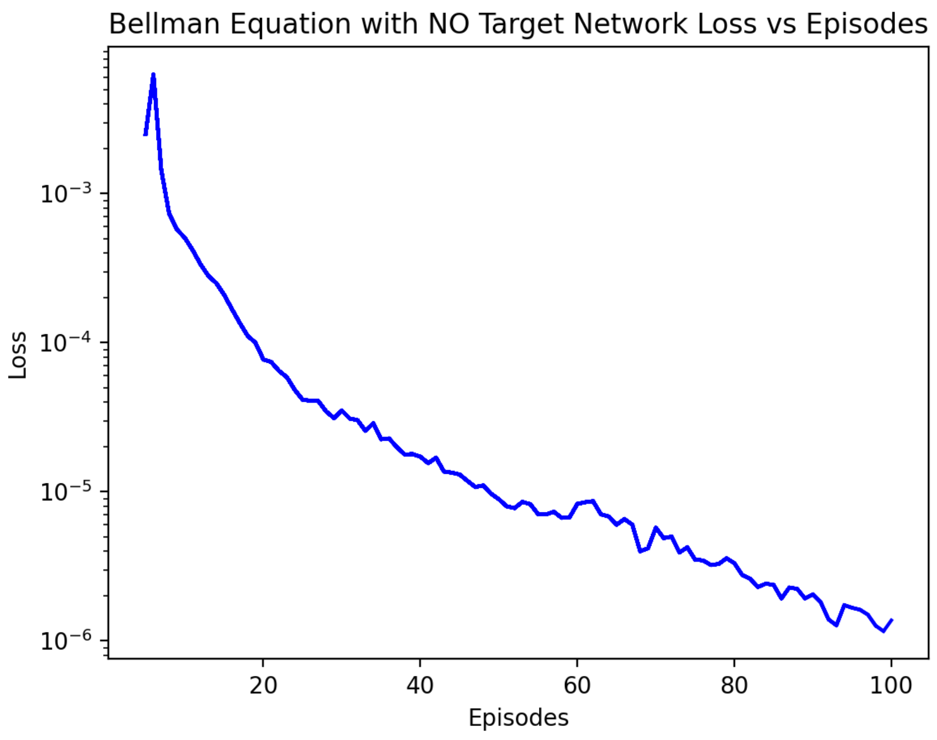

Figure 3 (a)

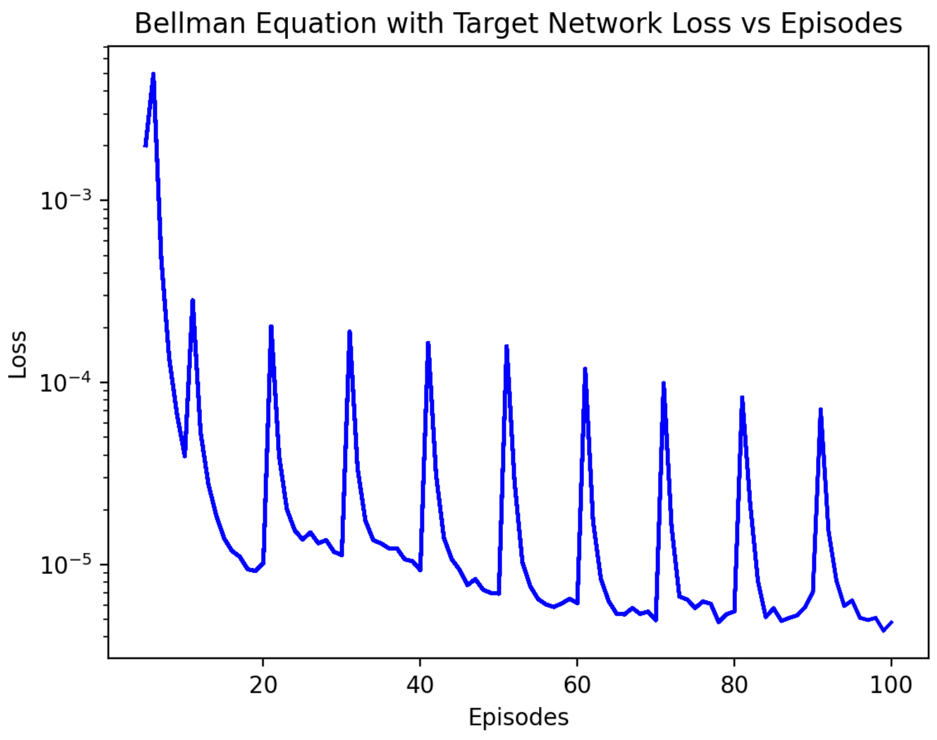

Figure 3 (b)

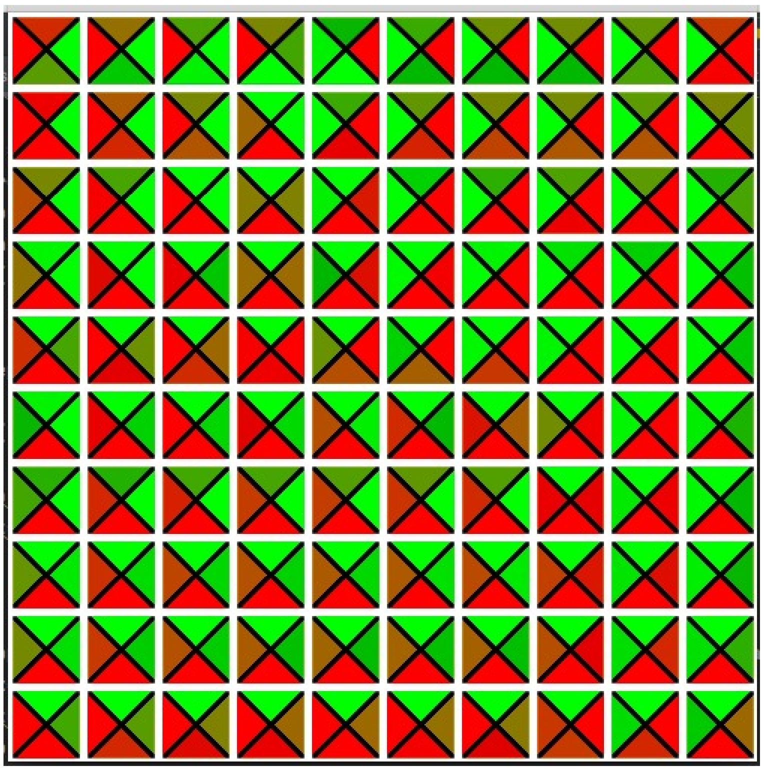

Figure 4 (a) Visualising Q-values after agent trained with ϵ-greedy policy

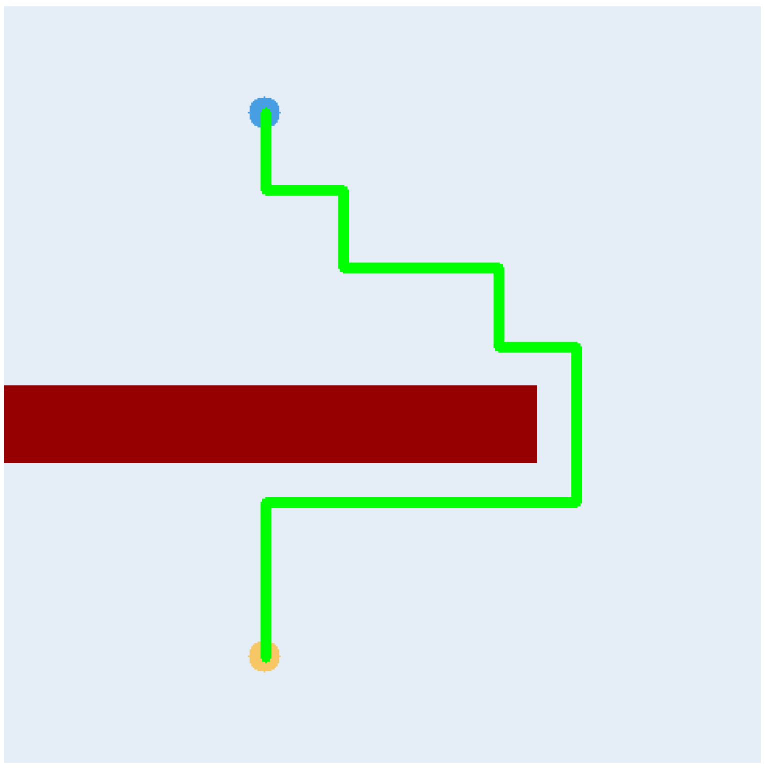

Figure 4 (b) Visualising the greedy policy after agent trained with ϵ-greedy policy

Description of Implementation for Part 2

I created a deep Q-network [line 48] using the DQN class [line 291], where I created a Q-network [line 296], and a target network [line 299], updated every 5 episodes [line 195] to prevent over-generalisation of the Q-values. I used the Network class [line 264] to create these two networks, which uses 4 hidden layers [line 271-275] to ensure that the agent is able to learn more complex patterns, to more effectively navigate the maze. The train_q_network function [line 311], implements gradient descent using the _calculate_loss function [line 337], which uses a minibatch to calculate the predicted Q value for each state [line 350-353]. In addition I used double Q-Learning [line 356-361] to prevent overestimation of the Q-values in the system. I followed by implementing the Bellman equation [line 363-369] into the loss function, setting a fixed γ=0.95 [line 363] to ensure the agent values future rewards highly.

On line 50 I created a replay buffer using the ReplayBuffer class [line 376], with a maximum size of 20,000 [line 380]. It contains an add_transition function [line 382-383] and a sample_minibatch function [line 385]. I set a minibatch size of 1849 [line 52] to maintain a more uniform data distribution across the environment on updates, and it ensures we train on each data point many times. The has_finished_episode function within the agent class [line 62] is True when an episode ends.

For the get_next_action function [line 75], if the agent is in its first 4 episodes it implements the _choose_next_action function [line 227], with an episode length of 1000 [line 30]. In this function an ϵ-greedy policy is implemented with an ϵ=0.8 [line 42]. When the agent is between the 5^{th} to 9^{th} episode then I implemented ϵ-decay [line 82-88] with a decay rate of 0.9999 every step. This leads to an $\epsilon \approx$0.5 by the end of the 9^{th} episode. At the 10^{th} episode the strategy changes. Whilst the time is less than 420 secs the agent conducts episodes of 420 steps [line 92]. For 2 episodes the agent conducts the greedy policy, using the function get_greedy_action defined on line 201, for the first 80 steps of the episode [line 96-100] and then conducts ϵ-greedy [line 102-106] for the remainder of the episode with ϵ=0.8. For the 3^{rd} episode I conduct a full greedy policy of episode length 100. After 420 secs the agent runs shorter episodes of 280 steps [line 117], with the same methodology as above, 2 episodes of greedy and ϵ-greedy and then 1 fully greedy.

The beginning of the set_next_state_and_distance function [line 156-169] ends the training using the end_the_search function [line 218] if certain requirements are met. It checks firstly whether the number of steps taken to reach the end_parameter, which is a distance from the goal, is less than 100, secondly that the distance_to_goal input to the function is less than the inputted end_parameter distance and thirdly that the last policy run was a full greedy policy, which we turn to true when a full greedy policy is run [line 137]. As time passes the radius from the goal which must be reached to end the training increases [line 156-169]. Once the flag to end the training is raised the training on lines 176-196 stops. Before this flag is raised a reward of $0.4 \times ((1 - distancetogoal)^2)$ [line 178] is used and transitions are created which are added to the buffer [line 185]. Once the buffer reaches the minibatch size [line 190] the training is allowed to begin by training the Q-network on the minibatch [line 192], using the DQN class function train_q_network.