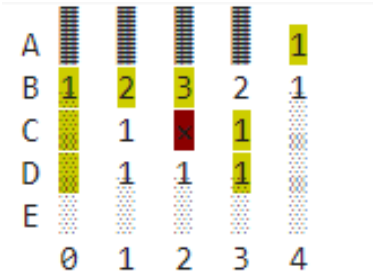
INM707 Deep Reinforcement Learning Coursework Report: Minesweeper

Nikita Ushakov and Julius Lord




```
Timestep: 13
State:
[[-2 -2 -2 -2  1]
 [ 1  2  3  2  1]
 [ 0  1 -1  1  0]
 [ 0  1  1  1  0]
 [ 0  0  0  0  0]]
Action: (2, 2)
Reward: -100
```

Figure 2: Our environment represented visually. A reward of -100 has been given as the agent selected the mine.

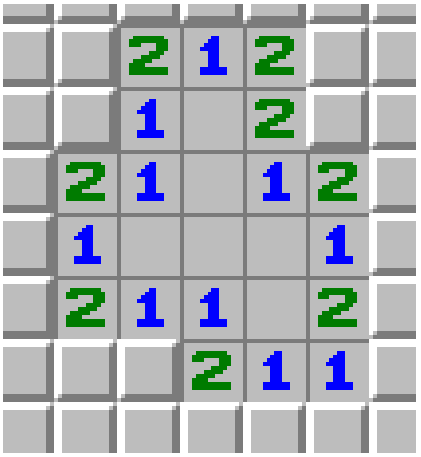


Figure 1: A section of grid from Minesweeper by Microsoft

1.1. Our domain and problem to be solved

In this report we will implement the Q-learning algorithm to solve a reinforcement learning task. The domain we will be working with is our version of the 1990 game Minesweeper. A two-dimensional grid environment with a single agent selecting squares in one of four states; Unknown, Empty, Numbered and Mined. The completion terminal state is reached when all unknown squares have been selected and revealed to be empty or numbered. Selecting a mined square is considered a failure and creates a terminal state, ending the episode. Numbered squares cannot be selected in the official version of Minesweeper but in our domain, the agent can. The agent is also capable of selecting already-selected Empty squares, though this has no impact on solving the puzzle and is discouraged with negative reinforcement.

The initial task's purpose is to train an agent to solve the puzzle, leaving only mined squares. The agent's actions (aka which square to select) are entirely random in the Brute Force Approach, so it acts as our 'control' to compare against. If our agent's average score is greater than that of randomized action, we can claim it is an improvement.

Our environment throughout is a 5x5 grid with 2 mines hidden within. After each episode Env.reset() is called, randomizing the environment between episodes. This makes every use of the model unique and returns a new mean reward each time.

Figure 2 represents the environment visually and the game operates as expected, numbered tiles representing the number of mines immediately adjacent.

1.2. Define a state transition function and the reward function

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha[r' + \gamma\, max_{a'} Q(s',a') - Q_{t-1}(s,a)]$$

$Q_t(s,a)$ is the new Q value

$Q_{t-1}(s,a)$ is the current Q value

α is the learning rate

r' is the reward function

γ is the discount factor

$max_{a'} Q(s',a')$ is the maximum expected future reward in the new state s'

State transition function: In a state s_t an agent performs the action a_t, leading to the state changing to s_{t+1} with a probability $P(s_{t+1}|s_t, a_t)$. In our domain the agent has only one action – to select a square. The agent starts by taking an action in the environment, creating a q-table filled with 0s. The agent transitions to the next state/observation by performing an action from the table. The state has multiple potential actions and the agent, unless told otherwise, will perform the action with the greatest reward. This transitions the agent to the new state/observation, storing the action, state, new state and reward in the agent's 'memory'. Normally an agent would be told to act randomly in the beginning of an episode, disregarding the q-table but still adding to its 'memory'. In a static environment this allows an agent to memorize its surroundings and gain an objective view on the optimal action to take.

Reward function: The reward function generates reward value *r* of an agent taking an action *a* in state *s* and is defined as $r' = r(s,a)$. After every transition, the agent receives a reward based on their action. For our domain, the rewards are:

Win (unknown squares containing empty or numbered fields have been selected, leaving 2 unknown squares containing mines) = 1000

Lose (selecting a mined square or reaching the step limit 100) = -100

Select an empty square (not mined) = 10

Select an already known empty square = -1

1.3. Set up the Q-Learning parameters (gamma, alpha) and policy

Learning Rate (α): Controls how fast learning occurs; How soon the new information garnered should replace the old in the Q matrix. It has a range between [0-1], 0 meaning values aren't updated and nothing is learned, 1 meaning the latest information is always recorded, regardless of accuracy.
Our learning rate was initially set at 0.8 as we wanted our agent to learn about its environment, though it wouldn't be very accurate. The environment is randomized each episode, which may explain why this theory didn't hold up experimentally.

Discount rate (γ): Controls the importance of future rewards; Whether the agent will focus on greater future returns or act for short-term benefit. It has a range between [0-1], 0 meaning no consideration for the future (myopic), 1 forcing the agent to focus on long-term rewards (far-sighted).
Our discount rate was initially set at 0.9 as we wanted a far-sighted agent, aiming for the 1000 reward for winning rather than the 10 for selecting an unknown square.

Learning policy (π): For our policy, we are aiming for maximum reward over minimum time. To this end we started with an ε-greedy policy.
ε-greedy policy is based upon initial exploration which gradually becomes 'greedier'; performing the action that will net the greatest reward. ε is the probability of moving about the environment randomly, in our case selecting a random square. 1 - ε is the probability to act greedily and perform the action with the maximum reward value. ε has a range between [0-1] and, given the probabilities, we can see that $\varepsilon = 1$ would lead to entirely random choices whilst ε = 0 would make for an always greedy policy. Ideally, we'd like an explorative policy at the beginning followed by greedier actions later, once every option has been explored. The ε is thus multiplied by a decay value < 1 so ε decreases after every episode, eventually leading to an (almost) fully greedy policy, except we have maximum and minimum ε limits of 1.0 and 0.01 respectively.

If our environment were static (only had one layout) then ε-greedy policy would work perfectly, starting with a high ε so the agent explores the blank slate of squares. Since there are only two mined squares, once those had been found the agent could be told to select every other square and optimally only two fail episodes would be needed. Our environment is randomized every episode so this doesn't hold up, the agent cannot simply memorize where the mines lie but must learn the system to decipher their locations.
Normally we'd expect to have epsilon close to 1 with a high decay rate as then the agent explores the environment randomly, gathering information before becoming greedy. We began at ε = 0.9 and decay rate = 0.01 to keep ε high.

1.4. Run the Q-Learning algorithm and represent its performance

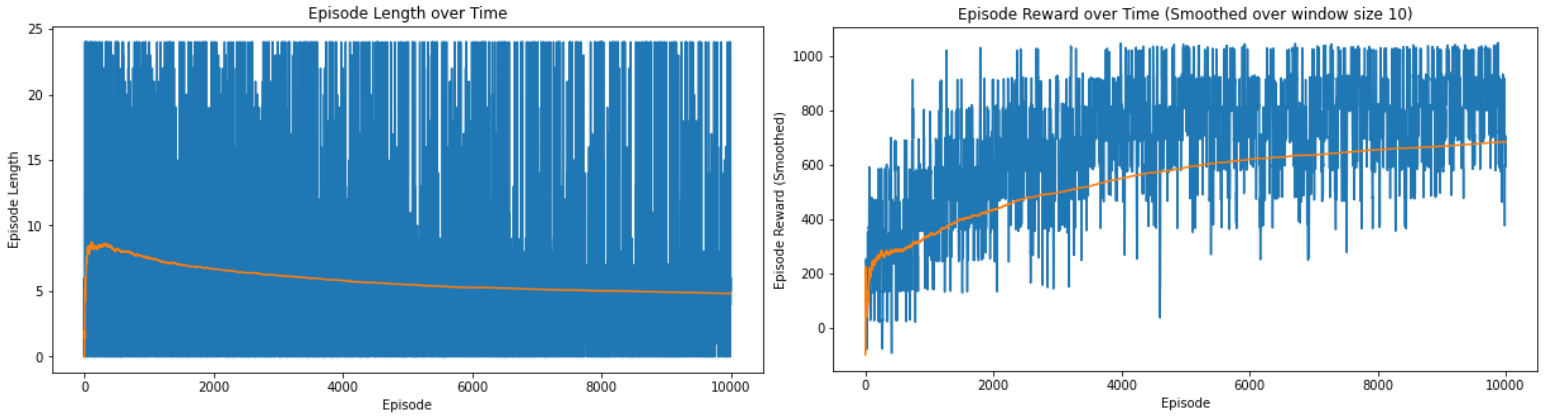


Figure 3 (above): The base setup; 5x5 grid with 2 mines,
α = 0.8, γ = 0.9, ε = 0.9, decay rate = 0.01

Average reward per episode = **683.8058**

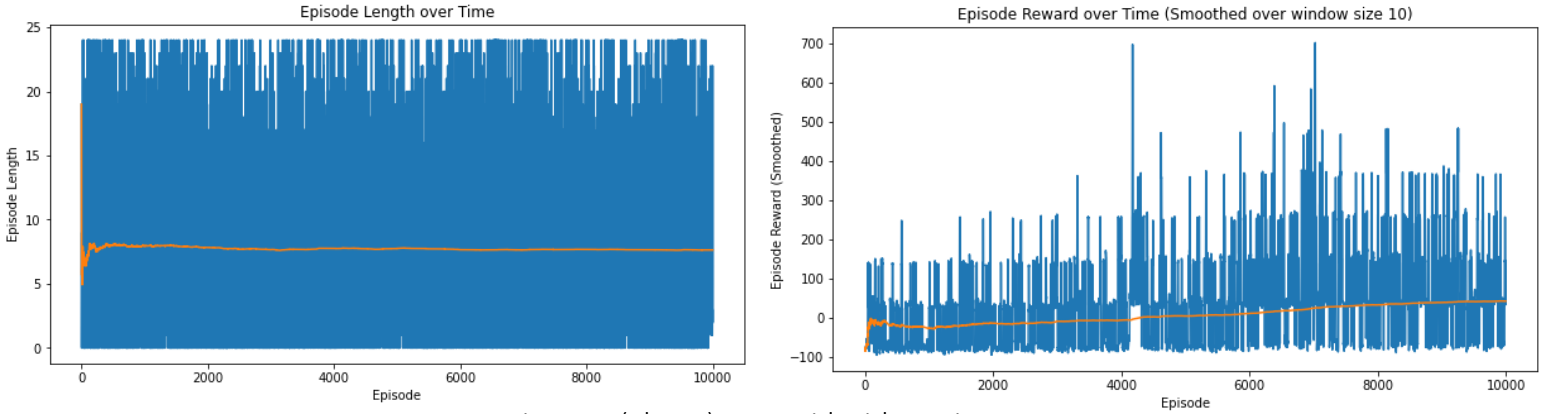


Figure 4 (above): 5x5 grid with 3 mines,
α = 0.8, γ = 0.9, ε = 0.9, decay rate = 0.01

Average reward per episode = **40.8306**

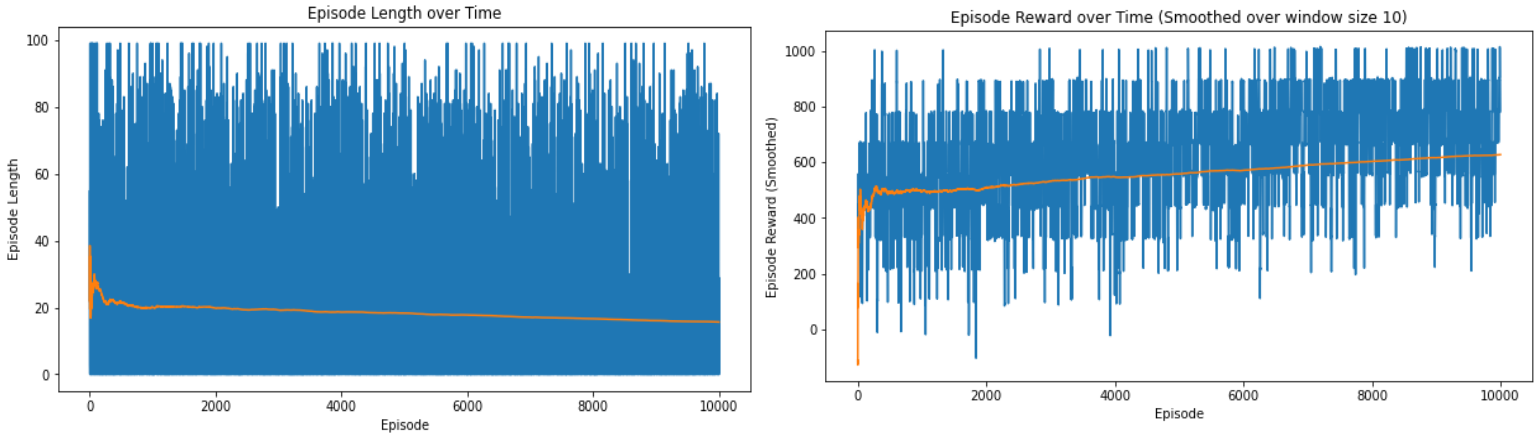


Figure 5 (above): 10x10 grid with 2 mines,
α = 0.8, γ = 0.9, ε = 0.9, decay rate = 0.01

Average reward per episode = **626.8592**

1.5. Repeat the experiment with different parameter values, and policies

Test	Discount factor (ɣ)	Learning rate (α)	Epsilon (Ɛ)	Decay rate	Mean reward (1)	Mean reward (2)	Mean reward (3)	Mean Average Reward
Brute Force	n/a	n/a	n/a	n/a	174.7271	173.5273	169.6895	172.648
One	0.9	0.8	0.9	0.01	679.235	670.4727	683.9675	677.8917
Two	0.1	0.8	0.9	0.01	673.6954	672.5191	679.0042	675.0729
Three	0.9	0.1	0.9	0.01	776.3056	772.5946	773.0166	773.9723
Four	0.9	0.8	0.1	0.01	685.4045	687.2155	683.5556	685.3919
Five	0.9	0.8	0.9	0.95	677.5872	677.4946	674.3462	676.476
Six	0.9	0.8	0.3	0.95	685.1028	677.5832	670.8997	677.8619
Seven	0.9	0.1	0.1	0.95	730.7615	747.5047	713.4891	730.5851
Eight	0.9	0.1	0.9	0.95	723.888	751.4459	761.2141	745.516
Nine	0.9	0.3	0.9	0.01	715.6377	698.3622	705.0134	706.3378
Ten	0.9	0.1	0.1	0.01	785.5868	768.2491	769.2784	774.3714
Eleven	1	0.8	0.9	0.01	701.4599	679.7813	684.9038	688.715
Twelve	1	0.1	0.9	0.01	760.6917	767.9717	767.1095	765.2576
Thirteen	1	0.1	0.1	0.01	737.9257	751.8424	751.1113	746.9598
Fourteen	1	0.1	0.1	0.95	750.9894	739.6516	766.5567	752.3992
Fifteen	1	0.01	0.9	0.95	780.2828	791.1231	783.9181	785.108
Sixteen	1	0.01	0.9	0.5	749.6398	753.1436	775.5365	759.44
Seventeen	1	0.01	1	0.99	778.8503	785.4045	786.7253	783.66
Eighteen	1	0.01	0.7	0.99	793.2769	749.206	775.4532	772.6454
Nineteen	1	0.01	0.8	0.95	771.6116	748.418	768.3945	762.808

Figure 6: Table containing all experimental values and mean rewards. We repeated each experiment three times and took an average of the results due to the random nature of the environment.

Grey: Value stayed the same
Green: Value increased
Red: Value decreased

Experiment Fifteen

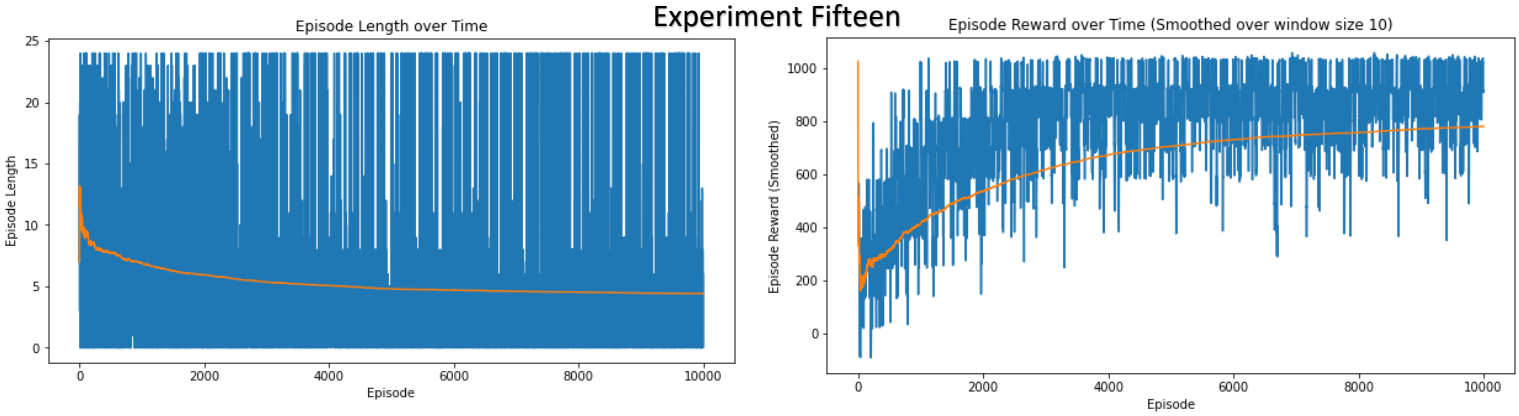


Figure 7 (above): The greatest mean score,
γ = 0.9, α = 0.01, ε = 0.9, decay rate = 0.01

Average mean reward per episode = **785.108**

Experiment 'Brute Force'

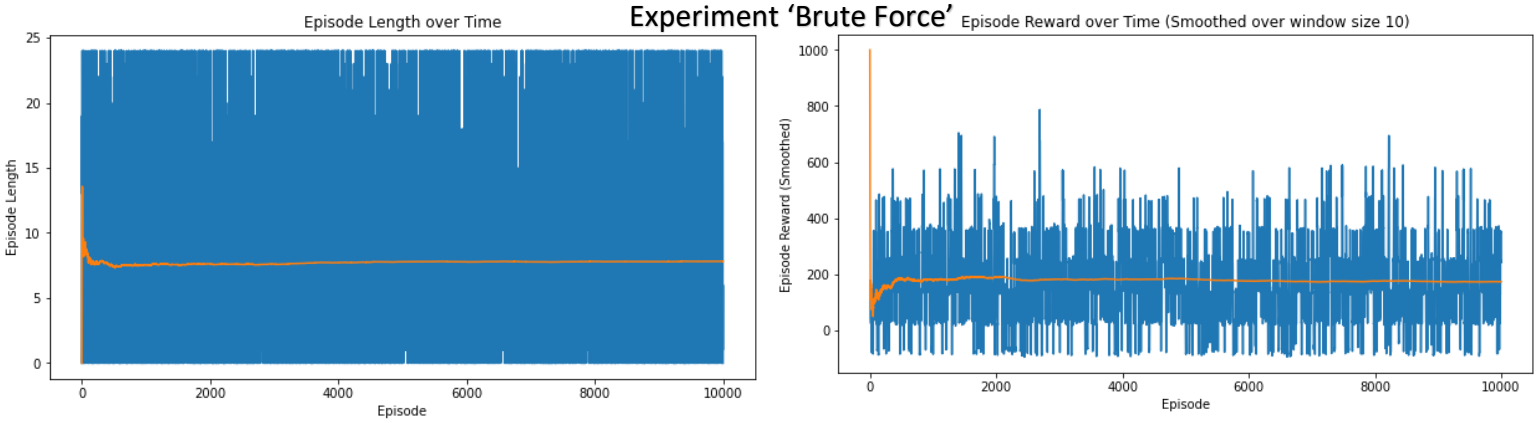


Figure 8 (above): Using completely random actions. No parameters.

Average mean reward per episode = **172.648**

Experiment Nineteen

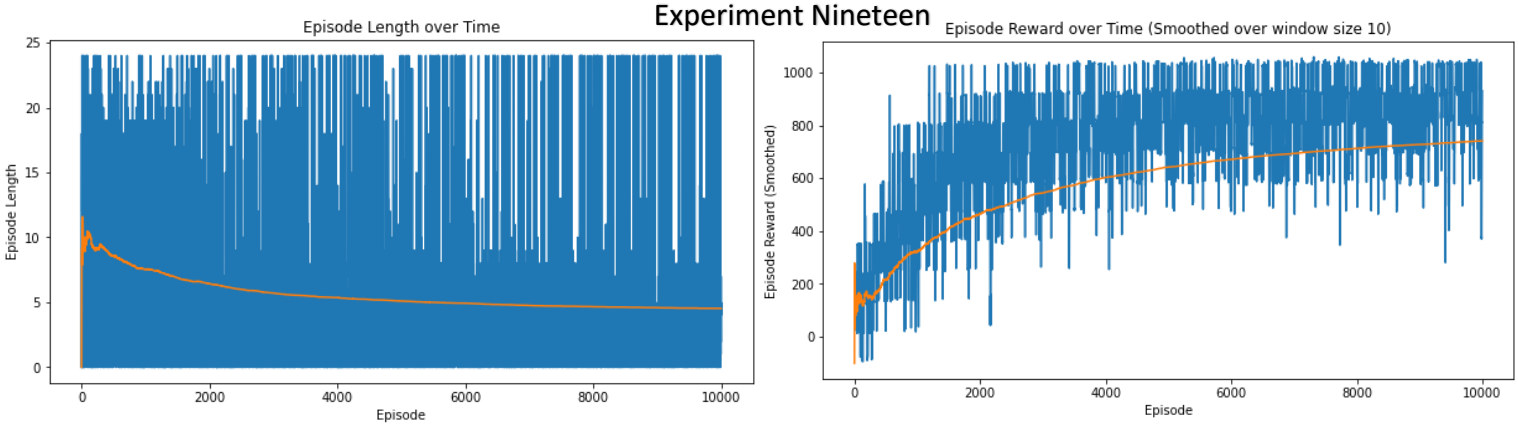


Figure 9 (above): The theoretical optimal parameters from previous experiments,
γ = 1.0, α = 0.01, ε = 0.8, decay rate = 0.95

Average mean reward per episode = **740.7387**

1.6. Analyze the results quantitatively and qualitatively

The orange line represents a simple moving average for easy comparison between episodes. We decided not to display the graph for every experiment as the results were visually very similar and the table (*figure 6*) displays the data clearly.
Between *figures 3 and 4* we observe the effect of adding an additional mine to our environment. First, the mean episode reward decreases dramatically and, though the data isn't recorded here, experiments in adding more mines led to an even greater decrease in the mean reward. We took this as the limit of our model, though with >3 mines the agent still improved with the same curve as with other experiments, it was still losing more games than it won and was incapable of further improvement.
Looking at the Episode Length over Time graphs, the shape of the average line stays the same for experiments with the same environment settings (grid size, number of mines) though interestingly the average goes from ≈5-7 episode length (steps) for 5x5 grid and 2 mines, to ≈20 episode length for 10x10 with 2 mines (*Figure 5*). This is to be expected, given a larger environment without additional obstacles, the agent spends longer exploring.
For the experiments with altered parameters, the general shape of the average line is that of an inverse exponential; rapidly increasing at the beginning then slowly flattening. It is worth noting that *figure 3, 7 and 9* end with the average line on a positive gradient, meaning over a greater number of episodes the agent still has room to improve. *Figure 8* acts as our control experiment and, fortunately for us, our average reward is higher for our model than for the 'Brute Force' randomizer. Looking at *Figure 8* it can be noted how quickly both average lines seem to 'flatten' early on (<1000 episodes). It makes sense for the randomizer to flatten, as it has no ability to get better or worse, but we did not expect it to reach its equilibrium in so few episodes.

From *Figure 6*, we will go through each experiment to explain what we changed, why and what was learned from each.

- Experiment *One*: The base stats, to act as a comparison for further experiments.
- Experiment *Two*: Decreasing the discount factor, led to a slight decrease in average reward but given the random factor in the results, we considered it to be negligible difference.
- Experiment *Three*: Decreasing the learning rate, led to a huge increase (+100) in average reward. This means that information was not recorded often but when it was, it was recorded accurately. We were not expecting this reaction, but the difference was large enough to discount randomization and say this was a positive change.
- Experiment *Four*: Decreasing epsilon, led to an increase in average reward (+10). Whilst this was an increase in reward, it is relatively small and given the range of values randomization adds to the average reward, as well as evidence later for increasing epsilon to increase reward, we did not put much stock in this result.
- Experiment *Five*: Increasing decay rate, led to negligible change in average reward. Almost identical reward to the base stats (*One*), given that decreasing epsilon led to an increase in average reward, we were surprised how little effect increasing the decay rate had, thinking that it would lead to a smaller epsilon sooner in the program and therefore increase the reward.
- Experiment *Six*: Decreasing epsilon less (to 0.3 rather than 0.1 in *Four*) and keeping the decay rate from *Five*, led to negligible change in average reward. Decreasing epsilon had led to slight increase in reward in *Four* but in this case, it seemed to have no effect.
- Experiment *Seven*: Combining *Three* and *Four*'s changes, decreasing learning rate and epsilon, led to an increase in average reward (+60). Looking at these results in the table, we can see there is a range of 30 between the three values taken, a sign the randomization is playing a larger part. The increase in reward is still significant, we already know decreasing the learning rate leads to increased reward, but this average is 45 points lower than *Three*, so we must question why decreasing epsilon and increasing decay rate would lead to this.
- Experiment *Eight*: Same parameters as *Seven* but with increased epsilon, led to an increase in average reward (+15). This appears to be an improvement from *Seven* however we must note that the average of *Eight* [745.516] is less than the 2nd result from *Seven* [747.5047]. The random factor is too present to say for sure whether changing a parameter directly caused a decrease in reward or if the agent was having a bad go at it for one of the three results. We try and reduce the magnitude of the random factor by taking averages over many episodes but the uncertainty remains.

- Experiment *Nine*: Slightly increasing Learning rate and dramatically decreasing decay rate, led to a decrease in average reward (-40). At this point it seems clear that the optimal value for learning rate is low, though we cannot discount the effect of decreasing the decay rate as, when changing two parameters for an experiment, it is difficult to tell what is responsible for the change in results.
- Experiment *Ten*: Decreasing learning rate and epsilon (*Seven* with decreased decay rate or *Three* with decreased epsilon), led to an increase in average reward equal to *Three*. The fact the results for *Ten* and *Three* are so similar might suggest that the epsilon value has limited effect on the experiment, or at least that a tiny decay rate causes the epsilon value to have limited effect.
- Experiment *Eleven*: Increasing discount factor, learning rate and epsilon (*One* with discount = 1.0), led to an increase in average reward (+10). We reverted all parameters back to base and increased the discount rate to its maximum value, forcing the agent to focus on future rewards. It could be argued the increase was due to luck with the randomization but given that all three results were greater than *One*, we counted it as evidence enough of a positive change and maintained it from this experiment onwards.
- Experiment *Twelve*: Decreasing learning rate, led to an increase in average reward. Despite the only change from *Three* being an increase in discount rate, the average reward is about 10 points less. This could suggest the increased discount rate led to the decrease, though it had just been shown in *Eleven* to have increased the reward.
- Experiment *Thirteen*: Decreasing epsilon, led to a decrease in average reward (-20). Given epsilon was the only changed parameter, it seems reasonable to say it is directly responsible for the decrease in reward however previous experiments (*Four*) have returned contrary information. More experimentation is required for a conclusive answer.
- Experiment *Fourteen*: Increasing decay rate (with low epsilon), led to a slight increase in average reward. The increase in reward is small enough to be considered heavily affected by randomization and theoretically increasing decay rate whilst keeping a small epsilon should not do much. Compared to *Seven* the only difference is the slight increase in discount factor for *Fourteen* though this experiment has 20 more reward points, adding to the evidence that increasing discount factor is a positive change.
- Experiment *Fifteen*: Increasing epsilon and decreasing learning rate, led to an increase in average reward to the highest average recorded. Decreasing the learning rate leads to an increase in average reward, so we decreased it to its minimum. The effect of increasing epsilon is unknown, it is possible the increase was due to it but given decreasing the learning rate has led to significant reward increase before (*Three, Seven, Ten, Twelve*) it seems more likely to be due to that.
- Experiment *Sixteen*: Decreasing decay rate to half, led to a decrease in average reward. Given it was the only parameter altered and the change was great enough value (-25) to be considered due to the parameter change rather than random chance.
- Experiment *Seventeen*: Increasing epsilon and decay rate to their maximum values, led to an increase in average score. The average was close to *Fifteen*'s, suggesting increasing epsilon and decay rate is a positive change. A high epsilon leads to random actions and a high decay rate leads to reaching a low epsilon (and greedy action) in fewer episodes. Our environment is randomized each episode but it still seems to be beneficial to our agent to be allowed to explore the environment before making active choices on actions.
- Experiment *Eighteen*: Decreasing epsilon, led to a decrease (-10) in average reward. This decrease is low enough to be considered heavily affected by the random factor, especially as the highest recorded mean reward [793.2769] came from this experiment. The range in recorded mean values is large, between (1) and (2) is a 40-point gap for instance. This makes it difficult to tell whether decreasing epsilon had significant effect on the average reward.
- Experiment *Nineteen*: Increasing epsilon and decreasing decay rate, led to (-10) decrease in average reward. Like *Eighteen*, this decrease in average reward could easily be attributed to randomization rather than the small changes in epsilon and decay rate. There is a greater difference between the first and second recorded means for this experiment [771.6116 and 748.418] than between the average reward for *Eighteen* [772.6454] and *Nineteen* [762.808].

Overall we ought to go with our highest average score (*Fifteen*) for our optimal values:

Discount factor (γ) = 1.0, Learning rate (α) = 0.01, Epsilon (ε) = 0.9, Decay rate = 0.95

Although the largest individual measurement came from *Eighteen*:

Discount factor (ɣ) = 1.0, Learning rate (α) = 0.01, Epsilon (ε) = 0.7, Decay rate = 0.99

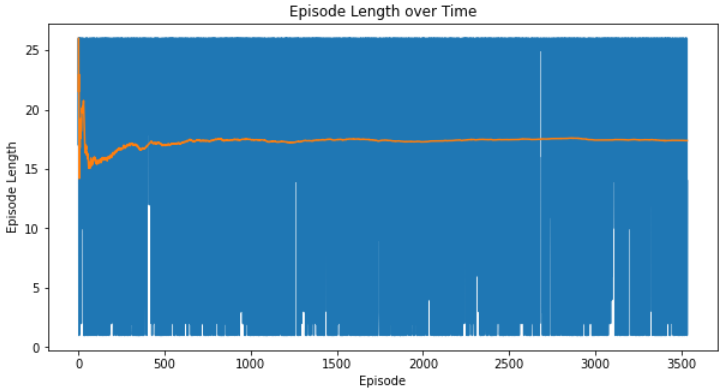
Advanced:

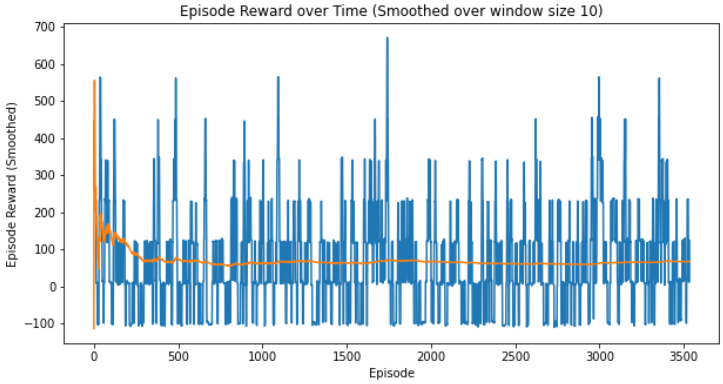
Implement DQN with two improvements. Motivate your choice and your expectations for choosing a particular improvement. Apply it in an environment that justifies the use of Deep Reinforcement Learning.

DQN;

Q-learning has serious limitations, the obvious being that the agent's 'brain' is the q-table. We stored our q-values as a dictionary structure to allow use of the state as a string. This won't be possible with DQN as the 'brain' is a neural network; DQN makes an approximation of the Q(s,a) function, to predict the reward the agent will receive performing a certain action in a certain state. The input to the neural network is the state/observation, and the number of output neurons is equal to the number of potential actions the agent could take.

We have kept our Minesweeper environment for this section as we believed our agent was limited by Q-learning and the idea of predicting future rewards for potential actions would theoretically work well with a puzzle deduction game.



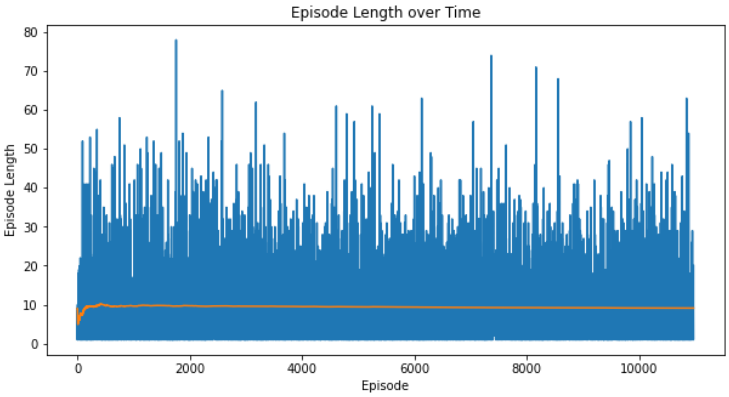


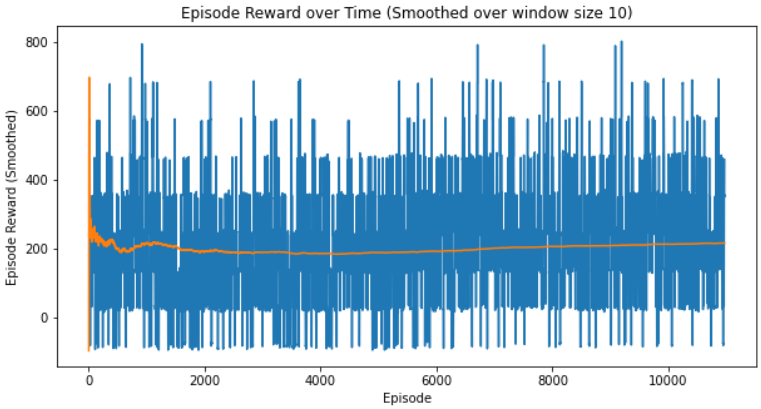
Analysis

The results surprised us. We had expected DQN to function more efficiently and, whilst the Episode Length has similar shape to the Q-learning results, the average step count is higher. This combined with an average score close to 100 is a significant decrease. Even compared to the Brute Force method, the score is lower, suggesting the DQN method is worse than random chance. It should be noted however that only 3500 episodes were performed for DQN, for a better comparison with Q-learning we should keep variables the same.

Double DQN;

One of the major issues with DQN is the overestimation of rewards, the Q-values are predicted higher than they are. Our first improvement made was Double DQN: This is using two identical neural networks at different stages, one acting as a normal DQN and the other doing the same but for the *last episode* from the first model. The Q-value is calculated using the second model as, in DQN, the Q-value is computed combining the maximum next-state Q-value and the current reward. This is an issue for high values, as each step the value increases further. Every output neuron's value gets higher until the difference between the output values themselves is high and, let's say for a state s_t, action a_{t1} has a greater value than a_{t2}, a_{t1} will be selected in state s_t every step. Even if a_{t2} is an objectively better action for the agent, it's difficult to retrain the neural network with only one model. So, we bring in a second, a copy of the main model from the last episode with, crucially, lower differences in values.





Analysis

These are better results from a scientific perspective, we have done 10000 episodes for this, like for Q-learning, so a direct comparison between the two holds more merit. The average episode is lower than DQN yet more than Q-learning. We aim for the least amount of moves for an optimal model, so this change is positive, and the average score is higher too at around 200. This is greater than the Brute Force method but still significantly less than even the base Q-learning model. We're unsure as to the reason for this, the policy is the same throughout and the rewards given for winning, losing etc are the same.

Duelling DQN;

The major difference between DQN and duelling is the structure of the model. It has the formula

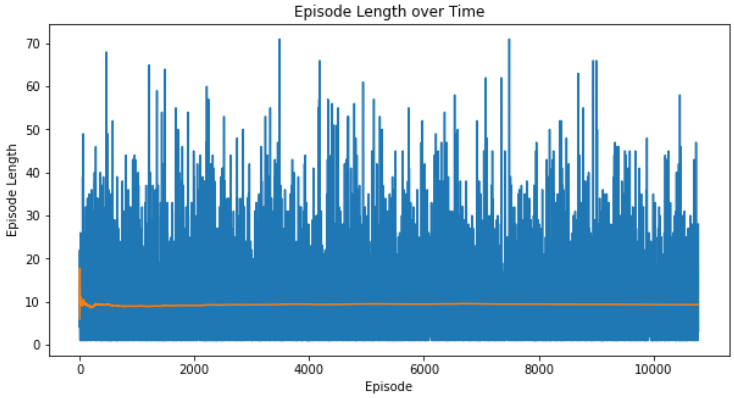
$$Q_t(s,a) = V(s) + A(s,a)$$

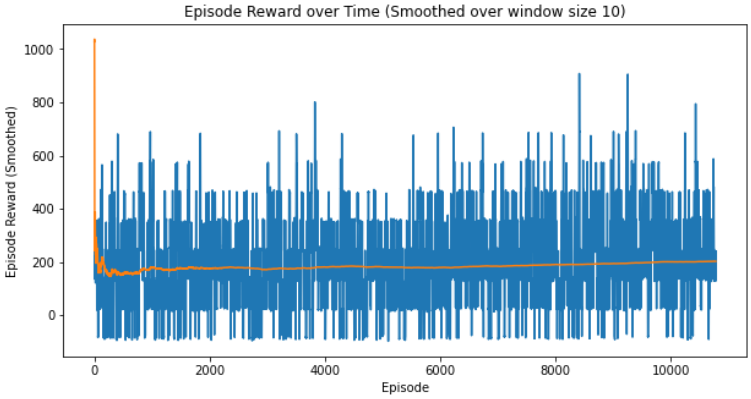
Where $V(s)$ is the value of state s and $A(s,a)$ is the advantage of performing action a whilst in state s. The value of state s is independent of the action, meaning the agent focuses on the benefit of being *in* a state rather than the actions it can take in that state. The duelling DQN proposes that the same neural network should split its last layer into two parts, one to estimate the state value $V(s)$ and the other to estimate the advantage function $A(s,a)$. The parts are combined into a single output which then estimates the Q-values. This isn't enough to train the neural network though, given $Q_t(s,a)$ we cannot find the values of V and A. Unless we force the maximum Q-value to be equal to $V(s)$, causing the maximum value of the advantage function to become 0 and all other values become negative. This gives us the value of $V(s)$ and from there we can calculate the advantages, solving the equation. There is another way: Instead of calculating the maximum value, we can replace it with the mean value, resulting in the equation

$$Q_t(s,a) = V(s) + A(s,a) - \frac{1}{n}\sum A(s,a')$$

Subtracting the mean of all the advantages from each advantage, we would get close to 0 and the Q-value would become the actual value of the state.

We're unsure of what to expect from this model, given the agent only has one action that can result in multiple different states depending on the episode/seed, the advantage to performing any action in any state should be the same.





Analysis

The results are very similar to Double DQN. Almost identical in average episode length and average score, though Double's average score is slightly higher. The only conclusion we can make is that there must be some issue with the code or method employed for Q-learning to beat DQN. It may have something to do with the agent having a potential action over the entire grid regardless of state. The issue is not a lack of complexity in the environment however, having researched other Minesweeper ai, they are generally daunting models beyond the capacity of Q-learning.

Contribution to team tasks:

We worked on the code together for several weeks, conducted experiments, and also wrote a report on the work done together.

Github link: https://github.com/qubka/QLearningProject