USING REINFORCEMENT LEARNING TO TRAIN AN AGENT TO PLAY FROZEN LAKE

1. Domain and Task

In this paper, we will use various reinforcement learning techniques to solve the Frozen Lake environment from the OpenAI Gym library. We will deep dive into using the Q-Learning algorithm to solve the environment, and then test the performance of two other reinforcement learning methods: Deep Q-Learning and SARSA.

In the Frozen Lake environment, we want our agent to reach the frisbee without falling into any of the holes in the lake. The environment consists of a 4x4 grid in which the holes are spread out in order to make it more challenging for the agent to reach the frisbee (Figure 1). The agent can move up, down, left, or right to navigate its way in the environment to find the frisbee located at State 16.

Orange: Starting point, Light blue: Frozen lake (safe), Dark Blue: Hole (lose), Green: Frisbee (win)

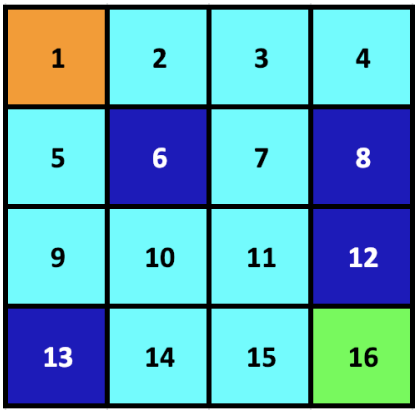

Figure 1 - Frozen Lake environment

In the default setup of the Frozen Lake environment in OpenAI Gym, the 'isSlippery' parameter is set to 'True'. This means that the agent only has a ⅓ chance of moving in the direction it chooses to, since the agent might 'slip' on the ice and move into an adjacent state instead. We have changed this parameter to 'False' so that our agent has a 100% chance of moving in its intended direction. This will allow us to more clearly see when the agent learns the optimal path to reach the frisbee without the agent 'slipping' into an adjacent state.

1.1 Defining the state transition function and reward function

In the Frozen Lake environment, our agent starts at State 1, as shown in Figure 1. In each state, the agent has a choice of 4 actions it can take: move 1 square up, down, left, or right. If the agent takes an action that would move it off the game board, the agent will not move and instead stay in the current state it is in. For example, if the agent is in State 1, and chooses to go left as its action, the agent will end up in State 1 again.

The goal of the agent is to reach State 16 in the shortest amount of steps possible. The shortest amount of steps to reach the goal state is 6 steps. There are 3 different paths for the agent to achieve this, shown by following the orange path in Figures 3, 4, and 5. On its path to get to the goal state, the agent has to avoid the 'holes' in the frozen lake. If the agent moves to a state that is a hole, the game ends. These holes are represented by a dark blue colour on the game board in Figures 1, 3, 4, and 5 (States 6, 8, 12, 13).

The transition possibilities from each state is shown in Figure 2 below. The states that have empty brackets as its transition possibility represents that the game will end if the agent reaches that state either because the agent has fallen into a whole and lost the game (States 6, 8 , 12, 13) or has found the frisbee and won the game (State 16).

Green: Goal state Red: Fall in hole

1 → {1, 2, 5}	7 → {3, 6, 8, 11}	12 → {}
2 → {1, 2, 3, 6}	8 → {}	13 → {}
3 → {2, 3, 4, 7}	9 → {5, 9, 10, 13}	14 → {10, 13, 14, 15}
4 → {3, 4, 8}	10 → {6, 9, 11, 14}	15 → {11, 14, 15, 16}
5 → {1, 5, 6, 9}	11 → {7, 10, 12, 15}	16 → {}
6 → {}		

Figure 2 - Transition possibilities for each state

Orange: Optimal path to reach goal state

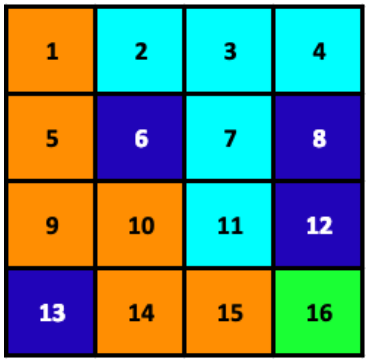

Figure 3 - Optimal path 1 of 3

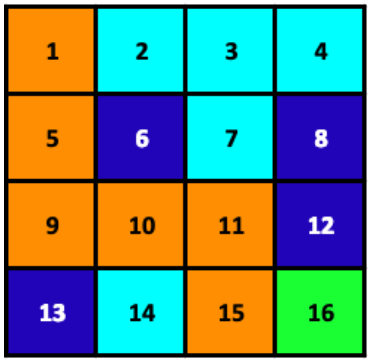

Figure 4 - Optimal path 2 of 3

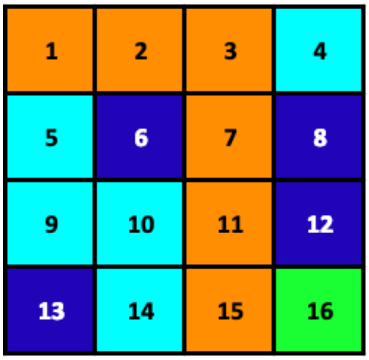

Figure 5 - Optimal path 3 of 3

The reward function for our environment is fairly simple. The only reward in the environment is a reward of +1 when the agent moves immediately from its current state into the goal state. Since the only way to move to the goal state (State 16) is from State 15, the +1 reward is set when the agent takes an action to move from State 15 to State 16. Moving around the environment has no positive or negative reward. In addition, there is no negative reward if the agent falls into a hole and ends the episode. The agent should learn on its own of the negative consequences of moving into a state with a hole. The reward function is represented in the R-matrix in Figure 6.

Action / State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1		0			0											
2	0		0			0										
3			0	0			0									
4				0				0								
5	0					0			0							
6																
7			0			0		0			0					
8																
9					0					0			0			
10						0			0		0			0		
11							0			0		0			0	
12																
13																
14										0			0		0	
15											0			0		1
16																

Figure 6 - R-matrix

1.3 Choosing a policy

Our agent will use an epsilon (ε) greedy policy in order to explore the environment. We must choose a starting ε value so the agent can decide when to explore versus when to exploit the environment. Exploring is when the agent randomly chooses an action to see what the resulting reward is. This is how our agent will learn about the environment. Exploiting the environment is when the agent uses the knowledge it has gained from exploration in order to choose the action that will maximize all rewards.

In the beginning, since our agent will have no knowledge about the environment, we want to prioritize exploration. Therefore, we want to set a high ε value (close to 1). We will initialize our ε value at 0.9. For each subsequent action that the agent takes, the agent will gain more knowledge and will eventually come to a point where it will be beneficial for the agent to start exploiting the environment. We can set an ε-decay rate to a value between 0 and 1, which we will multiply our ε value by after each step so our ε value will gradually become smaller and smaller. We will initialize our ε-decay rate at 0.9999. In order for the agent to choose between exploring and exploiting the environment, we generate a random number between 0 and 1. If the random number is less than 0.9, the agent will explore, and if it is greater or equal to 0.9, then it will exploit. After each step, the rate of exploration will decrease slightly (ε * 0.9999), giving the agent a better chance to start exploiting as it gains knowledge.

1.4 Set parameter values for Q-learning

The Q-learning parameter values we will use are shown below. ε-value and ε-decay rate have been explained in section 1.3. and γ and α will be explained in section 1.5.

γ (discount rate): 0.8
α (learning rate): 0.01
ε-value (exploration rate): 0.9
ε-decay rate (exploration decay rate): 0.9999

1.5 Show how Q-learning is updated in an episode

In this section, we will walk through how Q-learning is updated at each step in an episode. The Q-learning formula is below:

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha[r_{t+1} + \gamma maxQ(s_{t+1}, a_{t+1}) - Q_{old}(s_t, a_t)]$$

Q(s, a) - location in Q-matrix
s - state
a - action
t - time

α - learning rate (alpha)
r - reward
γ - discount rate (gamma)

For the first step in the episode, we will walk through both the detailed explanation as well as the mathematical steps. For the rest of the steps in the episode, we will follow the explanation at step one to update the Q-matrix until we reach the end of the episode.

1. Choose a random number between 0 and 1. If the number is greater than the current epsilon value, then exploit, otherwise explore.
   a. Exploit - agent chooses the next action based on the highest Q-matrix value (if all actions have the same value, then the action is chosen at random).

b. Explore - agent chooses next action at random

random(0,1) → 0.43 → **Explore** → Agent randomly moves down to state 5.

2. After the action is taken, update the Q-matrix using the Q-learning formula.
   a. Q-learning takes into consideration immediate reward and future reward at time t+1 in order to update the Q-matrix. For future reward, we take the maximum reward out of the rewards from all the potential actions we can take at time t+1. Since we want immediate reward to be more highly weighted than future reward, we multiply the future reward by the discount rate to lower its value. After adding up the immediate and discounted future rewards, we multiply that value by the learning rate. The learning rate slows down the agent's knowledge intake by lowering the reward value that we will update our Q-matrix with. This is especially important in more complex environments so the agent does not learn too quickly and make assumptions about the environment that may not be fully accurate. The mathematical steps to update the Q-matrix for the first step in the episode are shown below:

Q(1, 5) = Q(1, 5) + 0.01[r(1, 5) + 0.8 * max[Q(5, 5), Q(5, 9), Q(5, 6), Q(5, 1)] - Q(1, 5)]
Q(1, 5) = 0 + 0.01[0 + 0.8 * max(0, 0, 0, 0) - 0]
Q(1, 5) = 0

3. After the Q-matrix is updated, the final step is to update ε (exploration rate) using the ε-decay rate, before we can move on to the next step in the episode.

ε_{new} = ε * ε-decay rate
ε = 0.9 * 0.9999 = **0.8999**

The rest of the Q-learning updates for the first episode are shown below:

STEP 2
Explore vs. Exploit
random(0,1) → 0.92 → **Exploit** → Since the Q-value for all available actions is 0, the agent randomly moves down to state 9.
Update Q-matrix
Q(5, 9) = Q(5, 9) + 0.01[r(5, 9) + 0.8 * max[Q(9, 9), Q(9, 13), Q(9, 10), Q(9, 5)] - Q(5, 9)]
Q(5, 9) = 0 + 0.01[0 + 0.8 * max(0, 0, 0, 0) - 0]
Q(5, 9) = 0
Update exploration rate ε
ε = 0.8999 * 0.9999 = **0.8998**

STEP 3
Explore vs. Exploit
random(0,1) → 0.07 → **Explore** → The agent randomly moves right to state 10.
Update Q-matrix
Q(9, 10) = Q(9, 10) + 0.01[r(9, 10) + 0.8 * max[Q(10, 9), Q(10, 14), Q(10, 11), Q(10, 6)] - Q(9, 10)]
Q(9, 10) = 0 + 0.01[0 + 0.8 * max(0, 0, 0, 0) - 0]
Q(9, 10) = 0
Update exploration rate ε
ε = 0.8998 * 0.9999 = **0.8997**

STEP 4
Explore vs. Exploit
random(0,1) → 0.61 → **Explore** → The agent randomly moves down to state 14.
Update Q-matrix
Q(10, 14) = Q(10, 14) + 0.01[r(10, 14) + 0.8 * max[Q(14, 13), Q(14, 14), Q(14, 15), Q(14, 10)] - Q(10, 14)]
Q(10, 14) = 0 + 0.01[0 + 0.8 * max(0, 0, 0, 0) - 0]
Q(10, 14) = 0
Update exploration rate ε
ε = 0.8997 * 0.9999 = **0.8996**

STEP 5
Explore vs. Exploit
random(0,1) → 0.72 → **Explore** → The agent randomly moves right to state 15.

Update Q-matrix

Q(14, 15) = Q(14, 15) + 0.1[r(14, 15) + 0.8 * max[Q(15, 14), Q(15, 15), Q(15, 16), Q(15, 11)] - Q(14, 15)]

Q(14, 15) = 0 + 0.01[0 + 0.8 * max(0, 0, 1, 0) - 0]

Q(14, 15) = 0.01 * (0.8 * 100)

Q(14, 15) = 0.008

Update exploration rate ε

ε = 0.8996 * 0.9999 = **0.8995**

STEP 6

Explore vs. Exploit

random(0,1) → 0.72 → **Explore** → The agent randomly moves right to state 16.

Update Q-matrix

Q(15, 16) = Q(15, 16) + 0.01[r(15, 16) + 0.8 * max[ ] - Q(15, 16)]

Q(15, 16) = 0 + 0.01[1 + 0.8 * max( ) - 0]

Q(15, 16) = 0.01 * (1)

Q(15, 16) = 0.010

Update exploration rate ε

ε = 0.8995 * 0.9999 = **0.8994**

FIRST EPISODE COMPLETE

We can see the updates in the Q-matrix after the first episode in Figure 7.

Action State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1		0			0											
2	0		0			0										
3			0	0			0									
4				0				0								
5	0					0			0							
6																
7			0			0		0			0					
8																
9					0					0			0			
10						0			0		0			0		
11							0			0		0			0	
12																
13																
14										0			0		.008	
15											0			0		.01
16																

Figure 7 - Q-matrix after first episode of training

1.6 Represent performance vs. episodes

After training our agent for 5000 episodes, the final Q-matrix can be seen in Figure 8. Our agent was able to learn the optimal path of 6 steps to reach the goal state. I have highlighted the optimal path that the agent learned in green in the Q-matrix. The optimal path that the agent learned is the path shown in Figure 3.

Action State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	.205	.090			.328											
2	.196	.007	.000			0										
3		.001	.000	.000			.005									
4			.000	.000				0								
5	.204				.255	0			.410							
6																
7			.000			0		0			.102					
8																
9					.232				.304	.512			0			
10						0			.290		.431			.640		
11							.009			.061		0			.764	
12																
13																
14										.401			0	.516	.800	
15											.407			.518	.653	1
16																

Figure 8 - Q-matrix after training

Figure 9 shows the steps per episode. At the beginning, we can see that the graph is pretty random, as a result of the agent exploring and gaining knowledge about the environment. We can see that at around 1700 episodes, the range in the number of steps per episode starts decreasing, and a line starts forming around 6 steps per episode, which is the number of steps in the optimal path. We can infer that at this point, the agent has learned the optimal path to the goal state, and the deviance from the path during training is because the agent is still exploring during some steps.

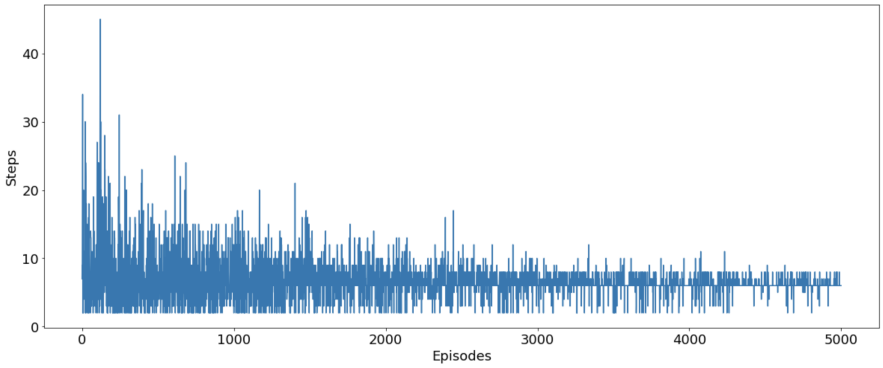

Figure 9 - Steps per episode

Figure 10 shows the average number of steps vs. average reward per 100 episodes. We can see that there is a steep increase in reward and decrease in the steps in the beginning as the agent starts learning, and starts to settle at around 1700 episodes, where the slight deviation comes from the agent's random actions during exploration.

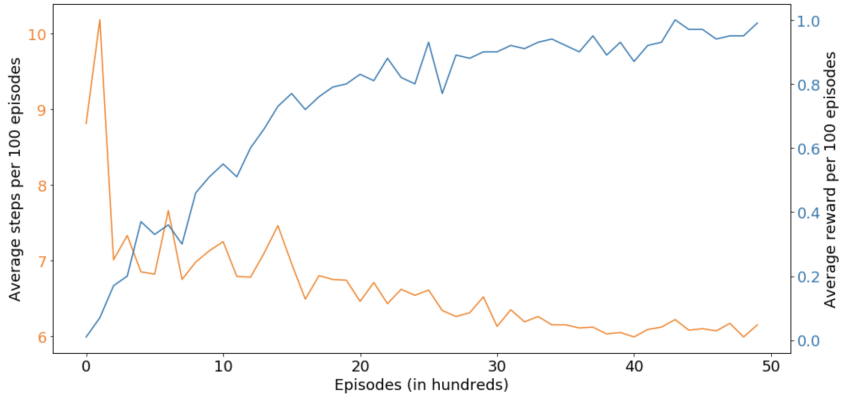

Figure 10 - Average steps vs. average reward per 100 episodes

1.6 Altered Q-learning parameter values

In this section, we will train our agent by altering one parameter value at a time while keeping the rest of the parameters at their default values (Learning rate α: 0.01, Discount rate γ: 0.8, Exploration rate ε: 0.9, Exploration decay rate: 0.9999). This will allow us to see how each parameter affects the learning process. Figure 11 shows our results. We will analyse the results in the following section.

Parameter Altered	Altered Value	Explore Count	Exploit Count	Converged?
ε (exploration rate)	0.99	9619	23,527	Yes
ε (exploration rate)	0.25	2492	485,386	No
α (learning rate)	0.9	8527	23,915	Yes
α (learning rate)	0.0001	8345	25,122	Yes
γ (discount rate)	0.99	8699	25,265	Yes
γ (discount rate)	0.1	8581	25,371	Yes

Figure 11 - Results from altering parameter values

2. Quantitative and qualitative analysis of Q-learning

For ε = 0.99

The agent converges to the minimum 6 steps needed to reach the goal state. However, it takes the agent considerably more episodes to learn than with our default parameters. This is because the higher exploration rate causes the agent to spend a lot more time exploring than it needs to in order to find the optimal path to the goal state.

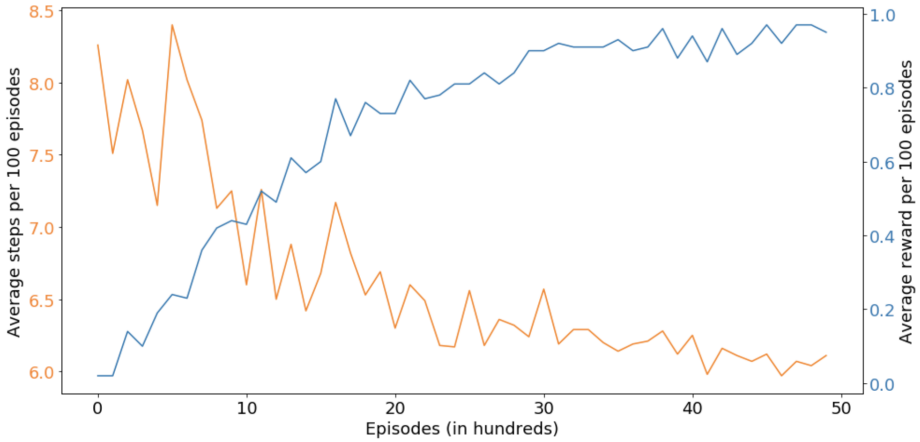

Figure 12 - Steps vs. episodes for ε = 0.99

For ε = 0.25

The agent does not find the optimal path to reach the goal state. Even with the decaying at a very slow rate, the starting ε of 0.25 is too low for the agent to explore the environment enough to gain the required knowledge to find the optimal path. This leads to many of the moves the agent makes being random. With the game capped at a maximum of 100 steps per episode, the agent does not get close enough to the goal state within 100 steps enough to gain the benefits of future rewards provided in the Q-learning algorithm. The agent uses up almost all of the maximum 500,000 steps its can take during the 5000 episode training process (5000 episodes * max. 100 steps per episode) since it cannot find the path to the goal state.

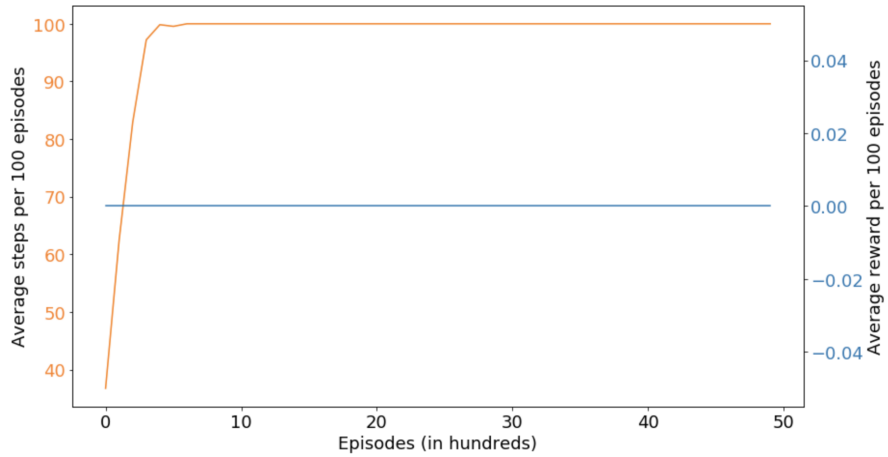

Figure 13 - Steps vs. episodes for ε = 0.25

Learning rate α: 0.9

The agent converges to the minimum 6 steps needed to reach the goal state. The agent takes a very similar path to learn as the default agent in terms of increasing rewards and decreasing number of steps per episode during training. This is because this is a relatively small and simple environment, with no obstacles on the way to the optimal path, such as negative rewards or randomness in the environment (the agent does not always reach the next state it intends to based on its chosen action).

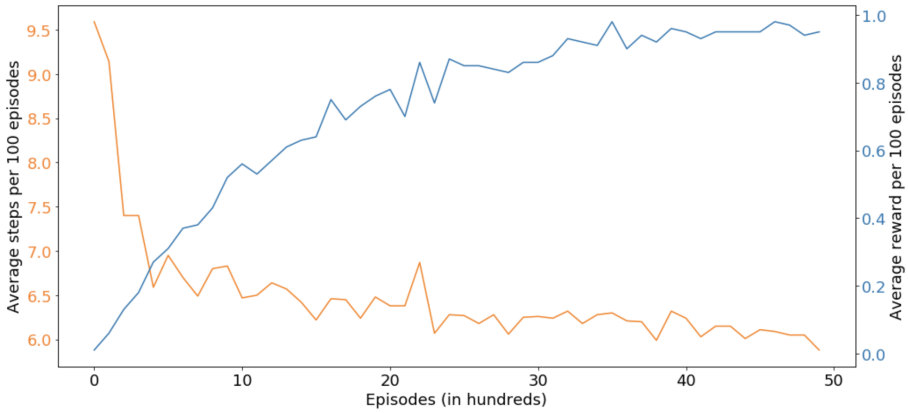

Figure 14 - Steps vs. episodes for α: 0.9

Learning rate α: 0.0001

The agent converges to the minimum 6 steps needed to reach the goal state. In this case, the Q-matrix did not end up with the goal state with a value of 1, as it did with the other cases that converged to the optimal path. For a more complex environment, the agent would need more training episodes to find the optimal path with a very low learning rate.

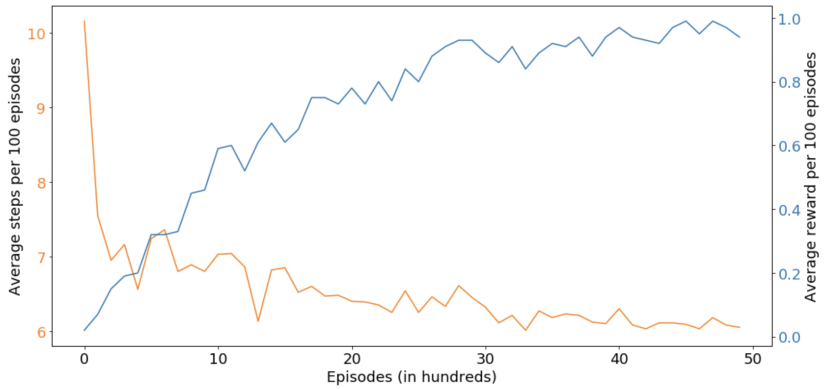

Figure 15 - Steps vs. episodes for α: 0.0001

Discount rate γ: 0.99

The agent converges to the minimum 6 steps needed to reach the goal state. The agent takes a very similar path to learn as the default agent in terms of increasing rewards and decreasing number of steps per episode during training.

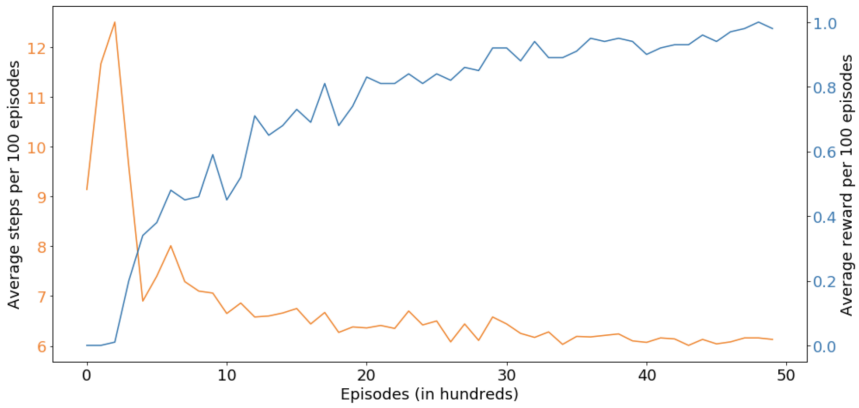

Figure 16 - Steps vs. episodes for γ: 0.99

Discount rate γ: 0.1

The agent converges to the minimum 6 steps needed to reach the goal state. The values of the Q-table were very low compared to the default agent. This is because the only reward in the environment is at the end (goal state) and the low discount rate makes the value of future rewards very low.

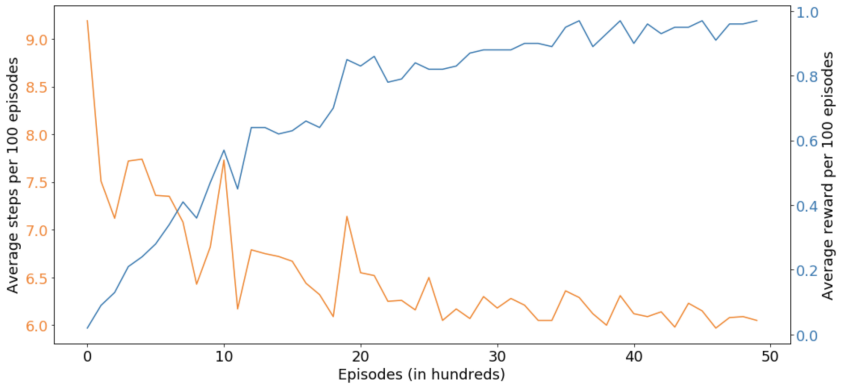

Figure 17 - Steps vs. episodes for γ: 0.1

The agents we trained without lowering the exploration rate learned fairly quickly and did not the full 5000 episodes to converge. In this instance, we could lower the ε-value and the number of episodes in order for the agent to accurately train in a shorter amount of time. But if we lower the ε-value by too much, the agent will not have enough knowledge to converge, as seen when we reduced the ε-value to 0.25. In more complex environments, the ε-value will need to be kept high in order for the agent to have the opportunity to adequately explore the environment. The observations about ε-value and number of episodes are important because increasing these values will increase the opportunity that your agent has to learn about its environment, but can significantly increase training time and resources, which is a big factor when training on large and complex environments.

In Q-learning, our results show that the discount rate has a significant impact on learning. Figure 17 shows that convergence took longer and learning was unstable for approximately the first 2000 episodes of training when we lowered the discount rate to 0.1. This is because updating Q-values largely depends on future rewards at time t+1, but those reward values have been discounted by 90% so they do not have as large of an impact during Q-value updates. In more complex environments where a reward at time t+1 can be misleading, having a lower discount rate can be beneficial to training, up to a certain point, where it may be too low and may hurt the training process.

In our experiments, changing the learning rate did not have much of an impact on learning. Our results show that we could use a high learning rate for simpler environments. As environments get more complex, we should lower the learning rate so that the Q-value is incremented slowly instead of updating it at each step using the full error value from the Q-learning algorithm.

Overall, Q-learning has worked well to train our agent to find the optimal path. The Q-learning algorithm has many similarities to the logic that humans, animals, and organisations use when learning about their environments. This shows us that many aspects of Reinforcement Learning and Machine Learning overall can be inspired by psychological and neuroscientific models, and we should continue to look at those areas of study for inspiration (see section 3.1 and 3.2).

3. Other reinforcement learning techniques

3.1 Deep Q-Learning

An alternative way to train our agent is by using a Deep Q Network (DQN). A DQN uses a Neural Network to predict Q-values. We use embedding and reshape layers in our model with the appropriate input and output dimensions (16, 4) to represent the states and actions in our environment. Our DQN agent utilizes ‘Memory’ to store it’s experiences in order to learn from them by training from random samples in memory. We will use 500 ‘warm-up steps’ in order to get some experiences in memory at the beginning of training. We will also use a target model, which is a copy of the Neural Network. Our first network has a constantly changing target, which makes training the agent more difficult. Our target model has a fixed target (the goal state) and is only updated periodically. For our policy, we will be using the epsilon greedy policy, as described in section 1.3. Our ε-value will be 0.1. Even though this is low, our agent will still learn because it has enough steps stored in memory to learn from because of the 500 warm-up steps it will take.

We trained the agent for 10,000 steps (approx. 1400 episodes). The results are shown in Figure 18. We can see that the DQN agent took a lot fewer episodes to converge (approx. 200 episodes) compared to using the Q-learning technique proposed in Section 1.

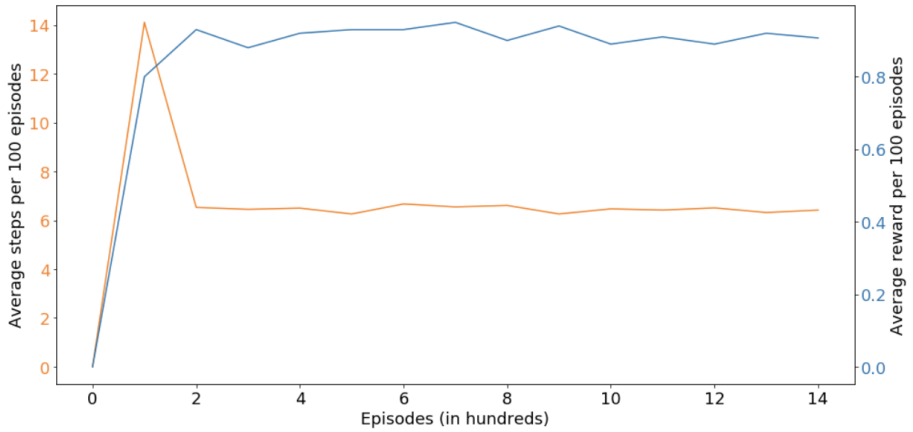

Figure 18 - Steps vs. episodes for Deep Q-learning

3.2 SARSA

SARSA (state action reward state action), is another method to train our agent. SARSA gets its name from the way it updates Q-values in the Q-matrix. The SARSA formula for updating Q-values is below:

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q_{old}(s_t, a_t)]$$

We can see that there are similarities to the Q-learning formula, but SARSA differs in how it obtains future rewards. Whereas Q-learning uses the maximum Q-value at state t+1 with actions t+1, SARSA does not use the maximum value. Instead, SARSA uses the policy it is following for a second time to choose action t+1 at state t+1. We will train an agent with the SARSA method, using the same parameters and policy used in our Q-learning method in section 1. The results can be seen in Figure 19.

The agent was able to learn the optimal path, but took slightly longer to converge. This can be due to the fact that SARSA uses the epsilon greedy policy again instead of using the maximum future Q-value to update the Q-values, so initially when epsilon is high, the future value is mostly chosen at random. This can lead to the agent needing more episodes to learn the optimal path.

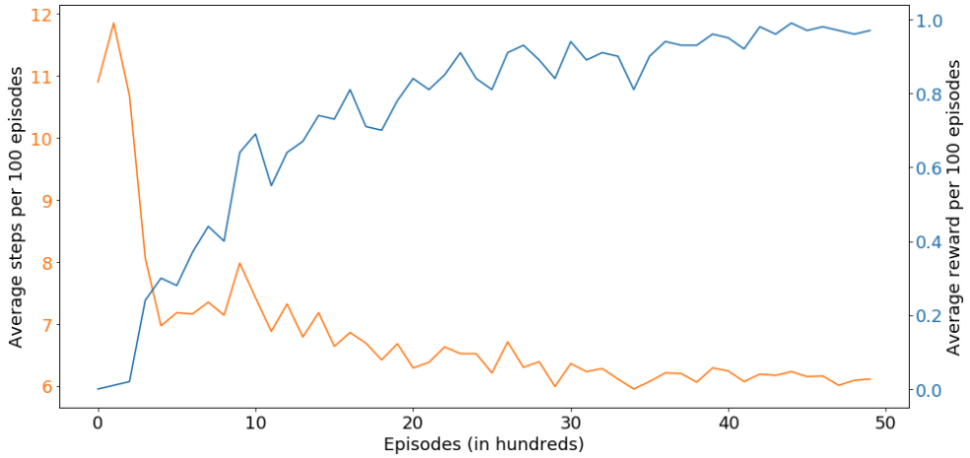

Figure 19 - Steps vs. episodes for SARSA

4. Reinforcement learning and error correction models

4.1 Q-learning and error correction models

In psychology, error correction refers to the feedback given when an action is taken by an organism [1]. The error is the difference between the intended result of the action and what is actually achieved [1]. The organism received the error feedback, and responds in a way that intends to reduce the error - in other words - move closer to the intended result. Niv (2009) lays out this out in an example in which conditional stimuli, a light and a tone, are given, along with an affective stimulus such as food or a tail pinch [2]. An organism can learn the outcomes that the conditional stimuli lead to in the environment.

This principle is the same principle that our agent used to train and extract knowledge from the environment using the Q-learning algorithm. In the Q-learning process, our agent explores and is given error feedback. The error is then used in the Q-learning algorithm to update the Q-value, which will help the agent reduce the error in the future. This can be seen more clearly using the error correction learning rule that is presented by Rescorla and Wagner. They present the error correction learning rule as $V_{new}(CS_i) = V_{old}(CS_i) + \alpha[\lambda_{US} - \sum_i V_{old}(CS_i)]$, where $V(CS_i)$ is being updated by the current value of $V(CS_i)$ plus the learning rate multiplied by the error [2]. We can see that the error correction learning rule discounts the value of future rewards. This is because organisms prefer immediate rewards over future rewards. For example, if we refer back to the situation presented by Niv in the above paragraph, an animal would prefer to get food immediately after being exposed to the relevant conditional stimuli, rather than get the food at a later time period. We can also see that mathematically, the error correction learning rule follows the same knowledge update path as the Q-learning algorithm, as shown in section 1.5.

4.2 Deep Q Networks and error correction models

In this section, we will focus on how error correction models relate to advanced reinforcement learning architectures. We will focus on Deep Q Networks and SARSA methods, since we reviewed them in section 3.

Campion et al. ran an error correction experiment in which they concluded that goal setting is a dynamic process and goals, state, and environmental feedback should be considered when monitoring performance and should be used to adjust goals, behaviours, and strategies [3]. This conclusion is very similar to the purpose of the main Neural Network model in DQNs. In DQNs, the main model takes in random samples from memory and the goal is dynamic, based on the input state. This allows the DQN agent to learn about its environment, often times with less training episodes than the simple Q-learning method. However, the agent should also keep the end target goal in mind while it is learning. This is where the target model comes in. The target model is used as a stable model in which the target goal does not change. This logic can also apply in other settings, where organisms should keep the big picture in mind while learning about their environment in order to extract the best knowledge.

As with Q-learning, SARSA is also related to psychological error correction models using the error correction rule, since the idea that updating values is calculated using the error, which in turn is found by discounting future rewards. The difference lies in the way future rewards are identified, with SARSA agents reusing the policy they are following for a second time to identify future rewards.

5. Scope and Conclusion

In this paper, we detailed how the Q-learning algorithm works and how it can be applied to a game environment. We altered Q-learning parameters to analyse the impact of certain parameters on an agent's ability to learn about its environment. We also tested two other reinforcement learning methods: Deep Q-learning and SARSA. We saw

that using DQNs can significantly speed up learning times, and would be well suited for more complex environments. We analysed the connection between psychological error correction models and Q-learning.

We can build on our analysis in this paper by testing other reinforcement learning methods, such as Monte Carlo or Q-learning with Neural Networks without using memory and a target network. We could also change the environment to see how these models translate to other environments. In addition, we can analyse how to alter the methods used when we are using environments in which we need to train on the pixels in the game, such as Mario.

References

[1]Busby, J. (1999). Problems in error correction, learning and knowledge of performance in design organizations. *IIE Transactions*, 31(1), pp.49-59.
[2]Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3), pp.139-154.
[3]Campion, M. and Lord, R. (1982). A control systems conceptualization of the goal-setting and changing process. *Organizational Behavior and Human Performance*, 30(2), pp.265-287.
[4]https://gist.githubusercontent.com/adesgautam/4821e9f18dd486f7af40a329078db0b7/raw/469f73b31908a65d564efc47e7883eeca0be17ac/frozen-lake_SARSA.py
[5]https://gist.githubusercontent.com/yashpatel5400/049fe6f4372b16bab5d3dab36854f262/raw/b24f420646c226096b59a60888f699272f7b440f/mountaincar.py
[6]Software Agents Moodle resources, MSc Data Science, City, University of London