FROZEN LAKE GAME USING REINFORCEMENT LEARNING

Aishwarya Virigineni[1], Maturi Tanuj[1], Apoorva Mani[1], Amudha J [2]
[1]*Department of Computer Science and Engineering, Amrita School of Engineering, Bengaluru, Amrita Vishwa Vidyapeetham, India*
[2]*Department of Computer Science and Engineering, Amrita School of Engineering, Bengaluru, Amrita Vishwa Vidyapeetham, India*
aishwaryavirigineni@gmail.com[1], *charantanujma63@gmail.com*[1], *apoorvaa.mani@gmail.com*[1], *j_amudha@bl.amrita.edu* [2]

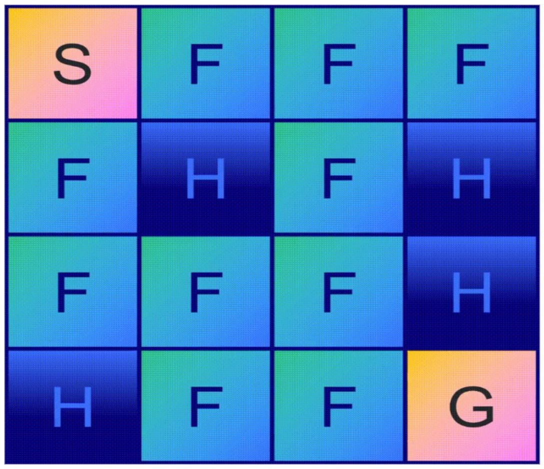
Abstract - **Reinforcement learning's (RL) primary aim is to learn an agent how to work in the best optimal way to maximise long-term reward. Q-learning is a learning algorithm that combines reinforcement learning with artificial intelligence. Reinforcement learning does not require external supervision since it interacts with the outside world via its own sensors. It uses continuous learning to map the status of the external input environment to output action and maximises the reward value of that action. To enable the submersible to adapt itself to surroundings on its own, it may modify its route autonomously through self-learning. To complete the adjustment of fuzzification strategy in an unknown environment, this work proposes to add Q-learning mechanism into reinforcement learning. We have implemented a Frozen Lake game with the Q-learning using the reinforcement learning methods.**

Keywords: Q-Learning, reinforcement learning, reward, agent

I. INTRODUCTION

Reinforcement learning is quite popular In the gaming world, The agent's primary goal is to maximize rewards while minimizing penalties by behaving appropriately in each scenario. Reinforcement learning is a machine learning approach in which the agent learns and goes through a series of stages, interacts with the environment, and ultimately maximizes the reward value. Machine learning may be utilized in the game industry to improve scores in a short amount of time. We can easily solve problems with Reinforcement Learning since the agent can learn from its mistakes and deliver the best solution.

The Q-Learning algorithm is a policy reinforcement learning method that is model free. In reinforcement learning, an action is chosen depending on the value of its state, which is updated using some sort of rule. An agent selects the activity that yields the greatest return from its surroundings. Depending on the surroundings, the reward signal may be positive or negative. In this article, two alternative variations of the Q-learning [2] approach are studied as part of the research of reinforcement learning in real-time. First, a traditional Q learning approach is discussed, in which a Q-value determines whether things have become better or worse than predicted because of a prior state action decision. To assess a previously selected action, a temporal difference (TD) error term delta (δ) is computed. The present condition is then used to calculate an approximated Q-value. The greatest Q-values are used to produce agent actions. Agent acts with a smaller TD error are more likely to be chosen. In the context of overestimation, the second type of Q-learning approach is defined. We offer a preliminary set in this latest version of the Q-learning approach. This article contributes by introducing an Approximate Q-learning approach for optimising an agent's performance. The following is the structure of this document. Section II contains a quick introduction to Reinforcement Learning. In part III, the basic notions from the rough set are briefly described. The traditional Q-learning reinforcement strategy is discussed in Section II



The Frozen Lake is a basic setting made of tiles in which the AI must navigate from an initial tile to a goal in this project. Tiles might be a safe frozen lake or a pit that will keep you stranded indefinitely. The AI, or agent, can take four different actions: LEFT, DOWN, RIGHT, or UP. To attain the goal in the fewest number of actions, the agent must learn to avoid holes. The environment is always configured the same manner by default.

III. CONCEPTS

A. Q – *Learning*

Every episode should have a reward or penalty for actions taken. If the highest-paying acts are known ahead of time, the agent can choose which ones to do. It will carry out a sequence of actions to maximise the reward yield. The entire reward is also known as the Q-value and can be expressed numerically as

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The above equation contains the following components:

- s: state
- a: action
- s': next state
- r (s, a): immediate reward plus the highest Q-value possible from the next possible state
- γ: discount factor (controls the contribution of rewards further in the future)

Q (s', a) in turn depends on Q (s'', a), which will have a coefficient of γ2. Therefore, the Q-value depends on Q-values of the future states, as displayed in this equation:

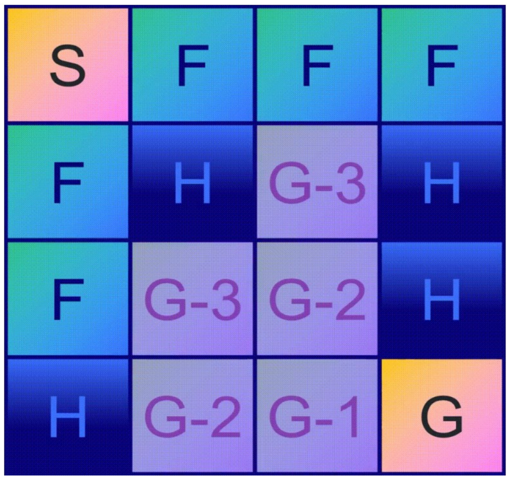
$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) + \ldots + \gamma^n Q(s''^{\ldots n},a)$$

The contribution of future values will be reduced or increased when the value is changed. The Q-values can be given any value using this recursive equation. The equations converge to an optimal strategy as the iterations continue and more experience is gained. This may be implemented as a Bellman equation-derived Q-value update.

We can manually try our formula to check if it looks correct: let's pretend our agent is in the state G-1 next to the goal G for the first time. We can update the value corresponding to the winning action in this state G-1 with:

$$Q_{new}(G-1,a_t) = Q(G-1,a_t) + r_t + mc$$

where Q(G-1, a_t) = 0 and $\max_a$Q(G, a) = 0 because the Q-table is empty, and r_t = 1 because we get the only reward in this environment.



We obtain Q{new}(G-1, a_t) = 1. The next time the agent is in a state next to this one (G-2), we update it too using the formula and get the same result:

In Frozen Lake, we want a high discount factor since there's only one possible reward at the very end of the game.
With the real Q-learning algorithm, the new value is calculated as follows:

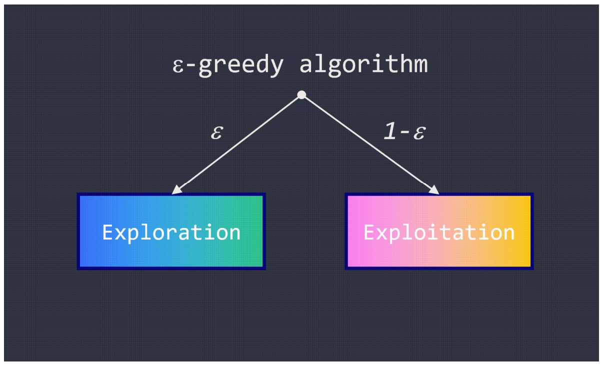
$$Q_{new}(s_t,a_t) = Q(s_t,a_t) + \alpha \cdot (r_t + \gamma \cdot max_a Q(s_{t+1},a$$

Once again, we can pretend that our agent is next to the goal G for the first time. We can update the state-action pair to win the game using our formula. As the number of states grows, the amount of memory required to save and update the Q-learning table grows. Exploring each state to generate such a Q-table would take an excessive amount of time.

B Epsilon-Greedy algorithm

If the agent only takes random actions, the training is pointless since it doesn't use the Q-table. So, we want to change this parameter over time: at the beginning of the training, we want to explore the environment as much as possible. But exploration becomes less and less interesting, as the agent already knows every possible state-action pairs. This parameter represents the amount of randomness in the action selection.

This technique is commonly called the epsilon-greedy algorithm, where epsilon is our parameter. It is a simple but extremely efficient method to find a good trade-off. Every time the agent must take an action, it has a probability ε of choosing a random one, and a probability 1-ε of choosing the one with the highest value. We can decrease the value of epsilon at the end of each episode by a fixed amount (linear decay) or based on the current value of epsilon (exponential decay).



The loss function is then optimised, and the neural network weights are updated using gradient descent. While optimising Li(i), the neural network parameters from the previous iteration, i1, are kept unchanged. Since the next action is decided based on the greedy policy, Q-learning is an off-policy method.

IV. MODEL ARCHITECTURE

A Q-Learning

- Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state by iteratively improving the behavior of the learning agent using Q-values.
- Q-learning seeks to learn a policy that maximizes the total reward.

B Epsilon-Greedy Algorithm

The Epsilon-Greedy Algorithm makes use of the exploration-exploitation tradeoff by

- instructing the computer to explore (i.e. choose a random option with probability epsilon)
- and exploit (i.e. choose the option which so far seems to be the best) the remainder of the time.

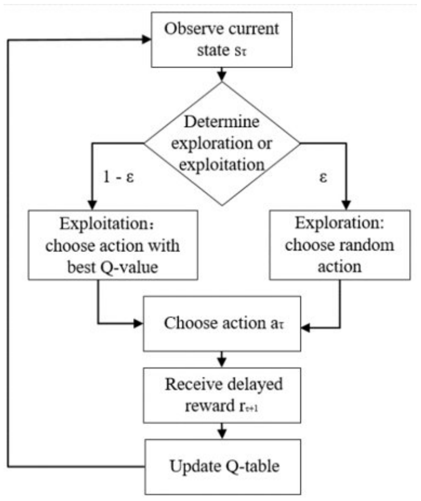

Observe current state s_t
Determine exploration or exploitation
1 - ε
ε
Exploitation: choose action with best Q-value
Exploration: choose random action
Choose action a_t
Receive delayed reward r_{t+1}
Update Q-table

Fig 1: Q Learning Archhitecture

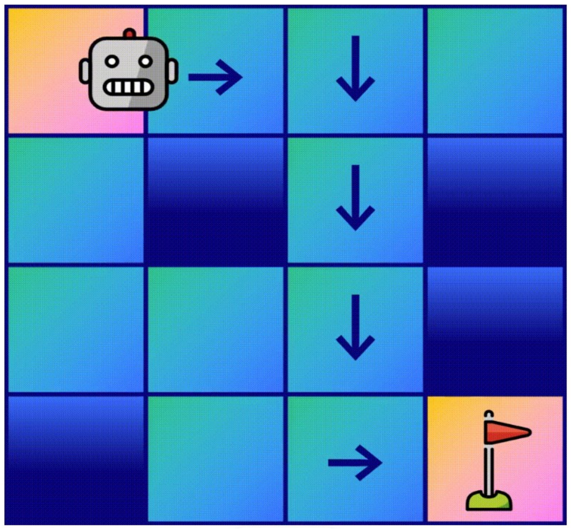
C RL FORMULATION

Goal: The agent must learn to avoid holes in order to reach the goal in a minimal number of actions.

Environment: A 4x4 grid composed of tiles

State Space: In Frozen Lake, there are 16 tiles, which means our agent can be found in 16 different positions.
S is the starting state
F is the frozen states
H is the hole states
G is the goal state

Action Space: For each state, there are 4 possible moves: LEFT, DOWN, RIGHT, and UP.



Reward: The agent gets a reward of +1 when it reaches the goal tile, and, at all other times, there is a 0 reward.

Policy: The epsilon is picked at random, and the playing philosophy is e greedy.

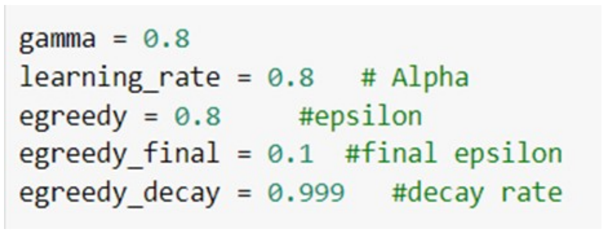
ε-Greedy Policy: A greedy policy means the Agent constantly performs the action that is believed to yield the highest expected reward.

D. Reinforcement Learning Parameters

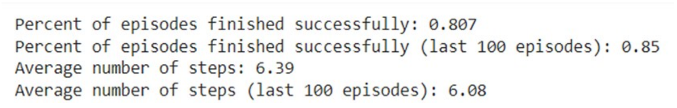
There are several parameters involved in reinforcement learning. Mainly four parameters are experimented with:

- epsilon: probability of choosing a random option and not going with the greedy choice
- epsilon -decay: factor by which epsilon decreases and provides a balance between exploration and exploitation
- α: learning rate, which controls how rapidly the model is adapted to the problem
- γ: discount factor that quantifies how much importance is given for future rewards

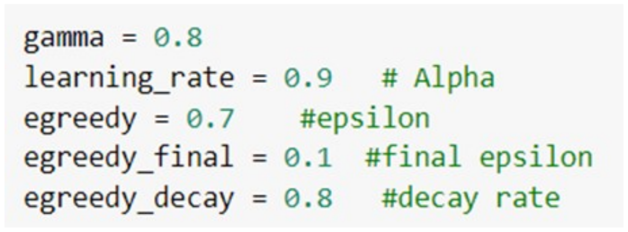
VI. Hyper Parameter Tuning



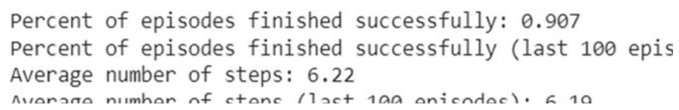
```
gamma = 0.8
learning_rate = 0.8   # Alpha
egreedy = 0.8     #epsilon
egreedy_final = 0.1  #final epsilon
egreedy_decay = 0.999   #decay rate
```



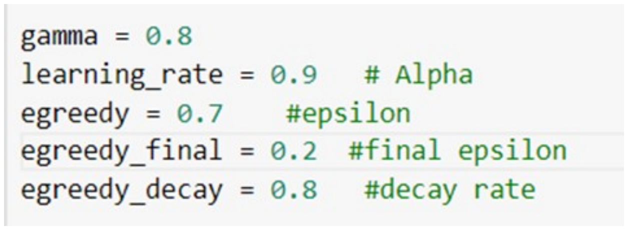
```
Percent of episodes finished successfully: 0.807
Percent of episodes finished successfully (last 100 episodes): 0.85
Average number of steps: 6.39
Average number of steps (last 100 episodes): 6.08
```



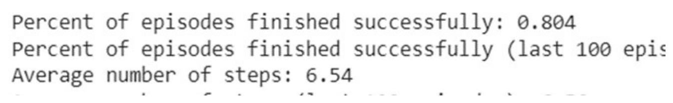
```
gamma = 0.8
learning_rate = 0.9   # Alpha
egreedy = 0.7     #epsilon
egreedy_final = 0.1  #final epsilon
egreedy_decay = 0.8   #decay rate
```



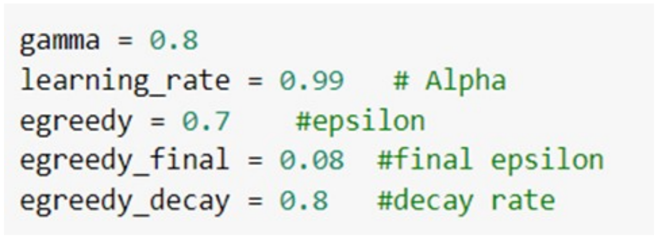
```
Percent of episodes finished successfully: 0.907
Percent of episodes finished successfully (last 100 epis
Average number of steps: 6.22
Average number of steps (last 100 episodes): 6.19
```



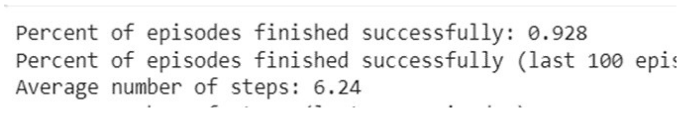
```
gamma = 0.8
learning_rate = 0.9   # Alpha
egreedy = 0.7     #epsilon
egreedy_final = 0.2  #final epsilon
egreedy_decay = 0.8   #decay rate
```



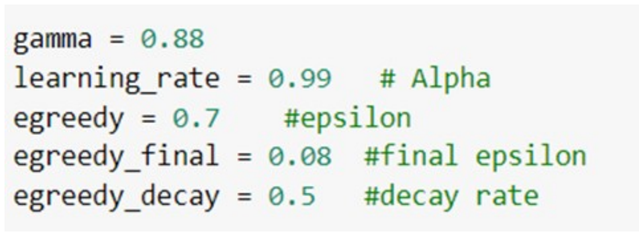
```
Percent of episodes finished successfully: 0.804
Percent of episodes finished successfully (last 100 epis
Average number of steps: 6.54
[illegible]
```



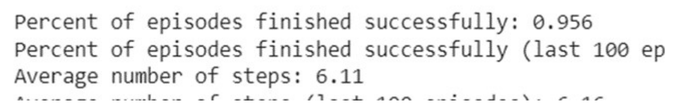
```
gamma = 0.8
learning_rate = 0.99   # Alpha
egreedy = 0.7     #epsilon
egreedy_final = 0.08  #final epsilon
egreedy_decay = 0.8   #decay rate
```



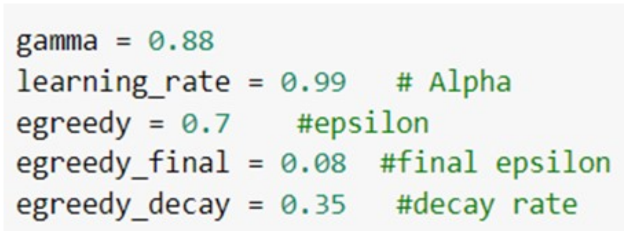
```
Percent of episodes finished successfully: 0.928
Percent of episodes finished successfully (last 100 epis
Average number of steps: 6.24
[illegible]
```



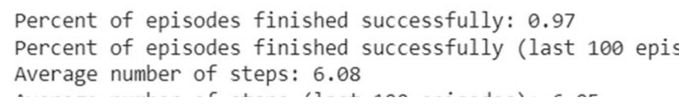
```
gamma = 0.88
learning_rate = 0.99   # Alpha
egreedy = 0.7     #epsilon
egreedy_final = 0.08  #final epsilon
egreedy_decay = 0.5   #decay rate
```



```
Percent of episodes finished successfully: 0.956
Percent of episodes finished successfully (last 100 ep
Average number of steps: 6.11
Average number of steps (last 100 episodes): 6.16
```



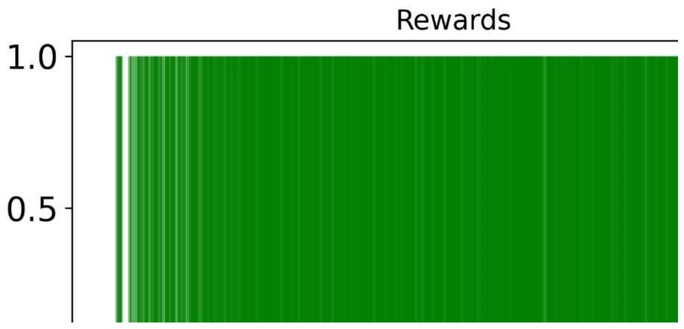
```
gamma = 0.88
learning_rate = 0.99   # Alpha
egreedy = 0.7     #epsilon
egreedy_final = 0.08  #final epsilon
egreedy_decay = 0.35   #decay rate
```



```
Percent of episodes finished successfully: 0.97
Percent of episodes finished successfully (last 100 epis
Average number of steps: 6.08
Average number of steps (last 100 episodes): 6.05
```

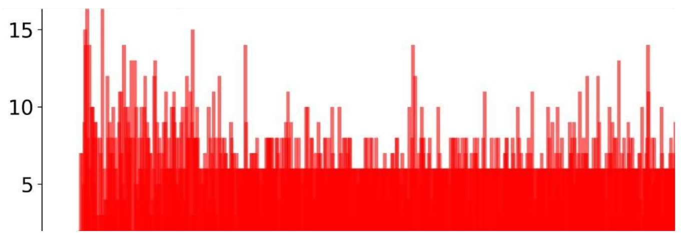
VII. Results

Various Hyperparameter values were tested as shown in the



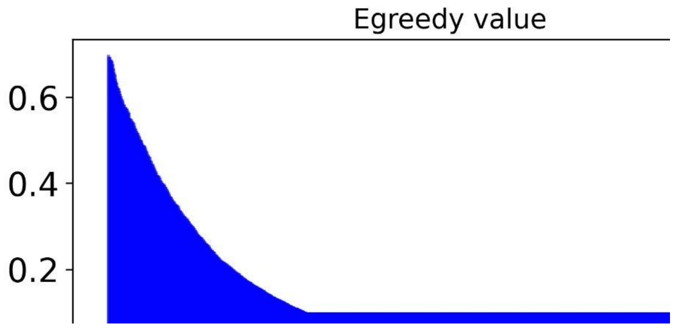
figures and the most optimal values were used for the algorithm. Given below are the graph results.

The first graph shows the plot of rewards vs Episodes. Here the white lines represent the episodes where the agent encountered a hole and thereby didn't reach the goal.



Steps taken per episode

The second Graph is plot of the steps taken by the agent vs the episodes. As it can be seen from the figure, the agent takes a lot of steps in the initial episode, i.e.; Before learning and it gradually decreases the number of steps taken with the increase in number of episodes as it learns.



epsilon value is reducing with time

The Third graph represents the learning rate (epsilon value) vs the episodes. The Value of the epsilon is high in the starting and gradually decreases due to the epsilon decay parameter. Overall, the implemented model has performed satisfactorily.

VII. Conclusion

Q-learning is a simple yet powerful algorithm at the core of reinforcement learning. In this article, we discovered how to interact with the gym environment to make decisions and move our agent. We introduced the concept of a Q-table, in which rows represent states, columns represent actions, and cells represent the value of an action in a certain state. To solve the sparse reward problem, we rebuilt the Q-learning update formula in an experiment. We established a comprehensive training and evaluation approach that was 100 percent successful in resolving the Frozen Lake situation.

We used the well-known epsilon-greedy algorithm to strike a balance between the search of new state-action pairings and the exploitation of the most successful.

The Frozen Lake is a basic environment, but others can have so many states and activities that storing the Q-table in memory becomes impractical. This is especially true in situations when occurrences aren't discrete but rather continuous (like Super Mario Bros. or Minecraft). When this problem arises, one common solution is to train a deep neural network to approximate the Q-table. Because neural networks are not very stable, this method adds numerous levels of complexity. However, I'll discuss it in a separate tutorial with various approaches for stabilising them.

References

[1] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction . MIT press, 2018.

[2] Meng Xu; Haobin Shi; Yao Wang ;Play games using Reinforcement Learning and Artificial Neural Networks with Experience Replay

[3] Reinforcement Learning in Games Istv´an Szita

[4] D. Pandey and P. Pandey, "Approximate Q-Learning: An Introduction," 2010 Second International Conference on Machine Learning and Computing, 2010, pp. 317-320, doi: 10.1109/ICMLC.2010.38.

[5] C. Peters, T. Stewart, R. West, and B. Esfandiari, “Dynamic action selection in openai using spiking neural networks,” 2019. [Online]. Available:https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS19/paper \/view/18312/17429

[6] Yunfeng Xin, Soham Gadgil, Chengzhe Xu Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning

[7] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” The International Journal of Robotics Research , vol. 32, no. 11, pp. 1238–1274, 2013.

[8] L.-J. Lin, “Reinforcement learning for robots using neural networks,” Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, p. 529, 2015.