# CSE574 Intro to Machine Learning Project 4

**Utkarsh Behre**
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
ubehre@buffalo.edu

## Abstract

In this project, we were asked to build a reinforcement learning agent that can learn an optimal policy through Q-Learning which will allow it to reach the goal in a given 4x4 grid-world environment. We were asked to make a Q-Learning agent by writing its policy and Q-table updating code along with the code to train the agent on the given environment. After performing these tasks, herein lies my final report.

## 1 Introduction

In this project, I am going to be using a specific type of machine learning known as the reinforcement learning. For the project we were already provided with the environment setup code beforehand. Our task is to write code for a Q-Learning agent that makes use of, as the name suggests, Q-Learning. This type of machine learning is very different from the other types like supervised or unsupervised learning, where we train the system based on some given sample of data. Here, instead of having such sample of dataset, we instead have a state space on which our agent tries to find out best actions it can take for any given states by trying out every action for every state over time. Let's see what this means in more detail next.

### 1.1 Reinforcement Learning

The Reinforcement Learning is a machine learning process where, instead of having samples of a dataset, we typically have an environment. This environment is a defined state space, where one can take some predefined set of actions at any given time to change the state of the environment from one to another. We typically have a goal to achieve in the environment which can be done by taking a sequence of actions based on the states of the environment we are on. If we were trying to reach this goal by ourselves, then we maybe able to read the environment very clearly and tell the obvious optimal set of actions required to reach the goal. But we want a bot which we call an agent to be able to reach the goal for us.

What we must do is to provide a policy to the agent that it can use while trying to find a way to reach the goal. Now, there are many ways of defining how an agent can learn about any given environment. But we are only going to be focusing on the way called Q-Learning, using which we will try to train our agent and be able to guess the optimal way to reach the goal. Next, we'll look at the challenge that we were supposed to solve, then we'll go over the Q-Learning algorithm that we used to do this, and lastly, we'll go over the environment that we used to train our agent.

## 2 The Challenge

In any reinforcement learning task the challenge is to be able to make an agent that is capable of learning about any given environment by training so that ultimately it is able to perform actions like a human will to reach the goal. Usually if we try to solve a puzzle ourselves, we can do this easily if the puzzle is simple enough because of human intuition. However same is not true for an agent. The agent doesn't have intuition or a general sense on what to do next on any given state at all. It relies completely on what it has seen working before. So, the agent must try out all the possibilities before it can find out the way it can reach the goal.

But, even finding a way to reach the goal is not enough. It must find a way that is possibly the best way to do it as well. This is where the difficult part comes. So, we rely on a reward system for the agent, where simply speaking it tries to get the maximum reward possible while reach the goal. This part is particularly interesting as the agent can do better based on how we design it.

In order to avoid agent from choosing the same path on which it reached out the goal again and again also known as exploitation, we introduce exploration into the system. Even if an agent has found a way to reach the goal, we let it try out other ways for a while so that chances of it finding a better way increase. We do this by using a hyperparameter called epsilon. Based on the episode on which the training is, we use a random number generator and let the agent choose from a random direction instead of already better-known direction if our random number value is less than epsilon. Initially we set the **epsilon** value to be 1 and we degrade this value exponentially every episode, this way the agent tries out new ways a lot initially, but as the episodes go by, chances for the agent on taking a random action reduces. We use a **decay** value below 1 that we multiply to the current epsilon every episode to achieve this. The value of epsilon becomes very close to 0 if not 0 by the end of all the episodes. And as the agent is closer to the end of its episodes, it starts relying on the previously learnt actions more and at the end it gets the best result out of it if enough episodes were given to train it on.

When the action is not chosen randomly based on the epsilon value, then the agent makes use of the below policy to decide which is the best action a among all the actions A it can take next at any state $s_t$. What argmax does is basically picks the index of the array of actions of that state for which the Q value is maximum. Thing to notice here is that the Q table is a 3-dimensional table for our agent in the project. The state $s_t$ is a 2d point. For example (0,2) or (3,1). Representation given below is merely a simplification of the same.

$$\pi(s_t) = \underset{a \in A}{\operatorname{argmax}}(Q_\theta(s_t, a))$$

**Figure 1: Policy when not randomly deciding action based on epsilon value**

## 3 Q-Learning Algorithm

The agent learns about the environment using the Q-Learning algorithm. In Q-Learning algorithm the agent stores Q values on a Q-table, which is basically a table with chances of reaching to the goal for all possible actions for all possible states that the environment can be in. As one can imagine initially the agent will have no knowledge on which action with get it to the goal for which state. It can only learn these values by trying out all the possible actions for all the possible states. This part of the algorithm can also be called as exploration phase. Once it starts finding these Q values and is nearly done with the training then agent goes into exploitation phase where it makes use of the best-known actions and tries to reach the goal.

The way agent learns about these values is by using a function called a Q-function. The function is used to update the values in the Q table for a state by seeing the best action found for all the possible actions it can take in the next state that it reached to by taking an action already and receiving the expected reward for it. It is given in the below figure 2.

There are a few parameters that can be used to control how these values are updated. The very first one being the **learning rate**, which as the name suggests controls how fast the values can be updated or in other words, how big of a jump the values can take in the next episode.

The next parameter is the **discount factor**, using which we can decide how much agent wants to rely on the next best action found for the next run. The higher the discount factor is the less sure an agent will be on relying the next best action it found. But eventually after taking this action many times the agent becomes sure about the best action it can take. Another important parameter is the number of **episodes** that we use to train the model. The more the episodes the more exploration can be done given the right values for the other parameters. Number of Episodes however completely depend on the complexity of the environment that we face.

Apart from these parameters, the values that updated rely heavily on the **reward** it may have received for an action. For example, a reward could have been a positive number or a negative number, which would drastically change the final assigned value to the Q table.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

**Figure 2: Q function used**

Another important part of the algorithm is how the reward is decided itself for an action. This is decided based on the policy used for the agent which was given in the figure 1 in the previous section. The environment itself will be giving us the appropriate rewards for an action which the agent will learn based on the parameters mentioned above.

## 4 The Environment

The environment given to us to train the agent is a basic deterministic 4 x 4 grid-world. The initial state for any run is given in the below figure 3 where the yellow square tells us the current position of the agent and the green square is the position where the agent must reach. The agent can take 1 step at a time in either of the 4 directions, down, up, right, and left from any point. So, there are 4 actions ( down, up, right, and left) in this environment for any given state space. When the agent takes an action, the yellow square moves in the direction based on the action. When it hits a wall after taking an action, its state doesn't change, and the yellow square remains in the same position. The goal is to take the least number of steps to reach the green square. A successful run ends when the yellow square reaches the green square. The agent is rewarded by +1 point if it has moved closer to the green square, but in any other case(moving farther from the goal or moving so that distance to the goal remains same) it is rewarded by -1 point. So, clearly an agent will be rewarded good if it gets closer to the goal. But if it does not than we punish the agent by giving it a negative reward so that it learns to not take that action again.

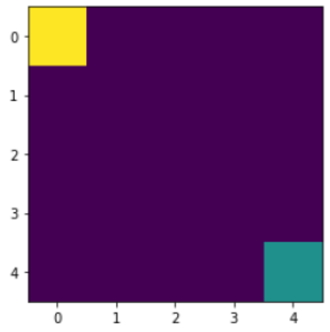


**Figure 3: The initial state of the given grid-world**

## 5 Experimentation

I tried to train the agent many times using different values for the hyperparameters mentioned in the above sections. In this section I'll explain how I got to those values of hyper parameters that can be seen in the results section later. At first I tried with epsilon's decay factor as 0.999 learning rate 0.1, gamma 0.9, and episodes 1000. For this set of hyperparameters, it seemed like exploration part was done way too much as values in the Q table were very much similar and on multiple runs the agent always ended up taking the path all the way down first and then all the way to the right. This path even though is an optimal path, but It would have been better if the agent would take right action before reaching the last row. So, it felt like the agent was over trained a lot in a way. So next, I just reduced the episodes to 100. For this the agent end up focusing on exploitation way too soon as some of the values in the Q table were even 0 after the training. Even though it was reaching the goal with these values, it couldn't be optimal as it didn't find all other paths that could have been better by chance.

Next I changed episodes to 500 and reduced gamma to 0.8 to allow more exploration done before deciding on a final path to exploit. The Q table values found for this was strange as few values were still 0 or close to it, but others seemed to have found its stopping point. So next I increased the learning rate to 0.5, 0.6, 0.7 and tried with all of them. The value with 0.6 seemed to be just right based on the Q values found at the end ( none being 0, either being positive or negative based on state and action).

So just to be sure I reduced the episodes to 300. This time the agent went on an almost perfect zig-zag path to the green square. But when looking at the Q-table, it still had many 0s in it. Then I tried more different episodes values like 800, 1200, , 1500, 2000, changing the decay value simultaneously so that it exponentially decays based on episodes. The decay values that had to be adjusted was mostly between .993 to .999. The run with the 2000 just felt like it didn't need that many episodes as the rewards were hitting 8 way sooner than the end. But at the same time, the Q values were almost all updated evenly being extreme positive or extreme negative which is exactly what we wanted. After trying episodes value 1500 many times, it seemed like this was just about right for the given grid-world. So, I ended up with final values of parameters mentioned next in the results section.

## 6 Results

The agent was performing best keeping the balance between exploration and exploitation well for the below values of hyper parameters. It was able to reach the goal of green square in just 8 moves and was getting 8 reward points for it.

| **Hyper Parameters** | **Value** |
|---|---|
| Learning rate | 0.6 |
| Gamma/discount factor | 0.8 |
| Episodes | 1500 |
| Epsilon (initial value) | 1 |
| Epsilon decay value | 0.9963 |

**Table 1: Final values of hyper parameters that agent was trained on**

The final epsilon plot graph that I got is given in the figure 4, and the total rewards plot graph that is given in the figure 5 below. The final values of the Q-table are also shown in the figure 6.

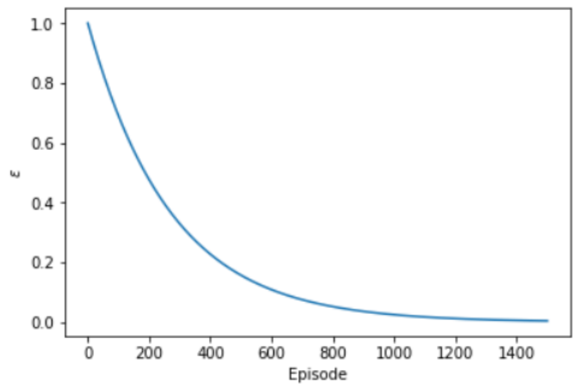


**Figure 4: Final Epsilon decay graph**

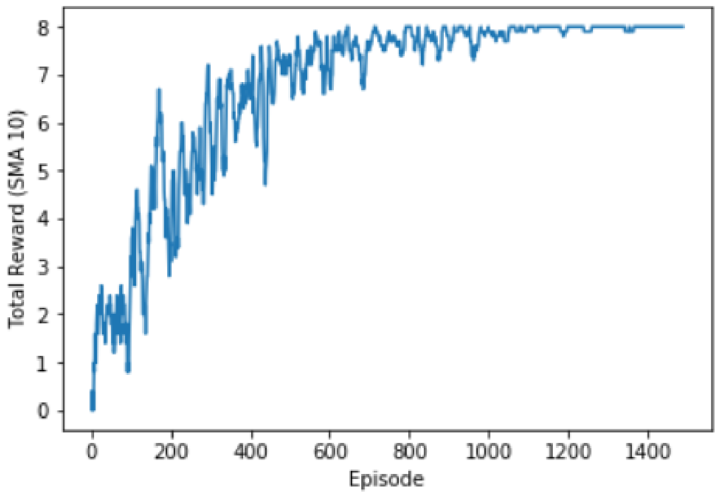
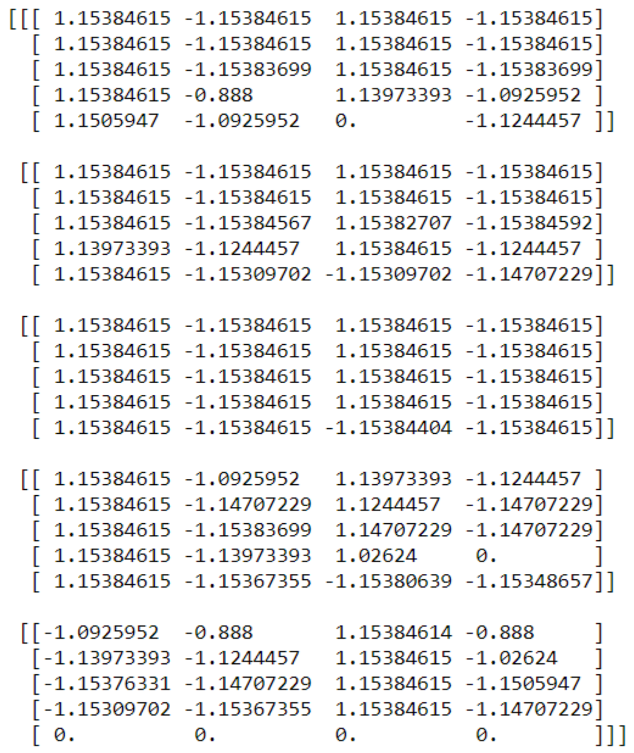


**Figure 5: Final Total Rewards vs Episodes graph**



```
[[[ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15383699  1.15384615 -1.15383699]
  [ 1.15384615 -0.888       1.13973393 -1.0925952 ]
  [ 1.1505947  -1.0925952   0.         -1.1244457 ]]

 [[ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384567  1.15382707 -1.15384592]
  [ 1.13973393 -1.1244457   1.15384615 -1.1244457 ]
  [ 1.15384615 -1.15309702 -1.15309702 -1.14707229]]

 [[ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615  1.15384615 -1.15384615]
  [ 1.15384615 -1.15384615 -1.15384404 -1.15384615]]

 [[ 1.15384615 -1.0925952   1.13973393 -1.1244457 ]
  [ 1.15384615 -1.14707229  1.1244457  -1.14707229]
  [ 1.15384615 -1.15383699  1.14707229 -1.14707229]
  [ 1.15384615 -1.13973393  1.02624     0.        ]
  [ 1.15384615 -1.15367355 -1.15380639 -1.15348657]]

 [[-1.0925952  -0.888       1.15384614 -0.888     ]
  [-1.13973393 -1.1244457   1.15384615 -1.02624   ]
  [-1.15376331 -1.14707229  1.15384615 -1.1505947 ]
  [-1.15309702 -1.15367355  1.15384615 -1.14707229]
  [ 0.          0.          0.          0.        ]]]
```

**Figure 6: Final values in the Q table**

The Final run done by the agent is shown in the below figure 7.

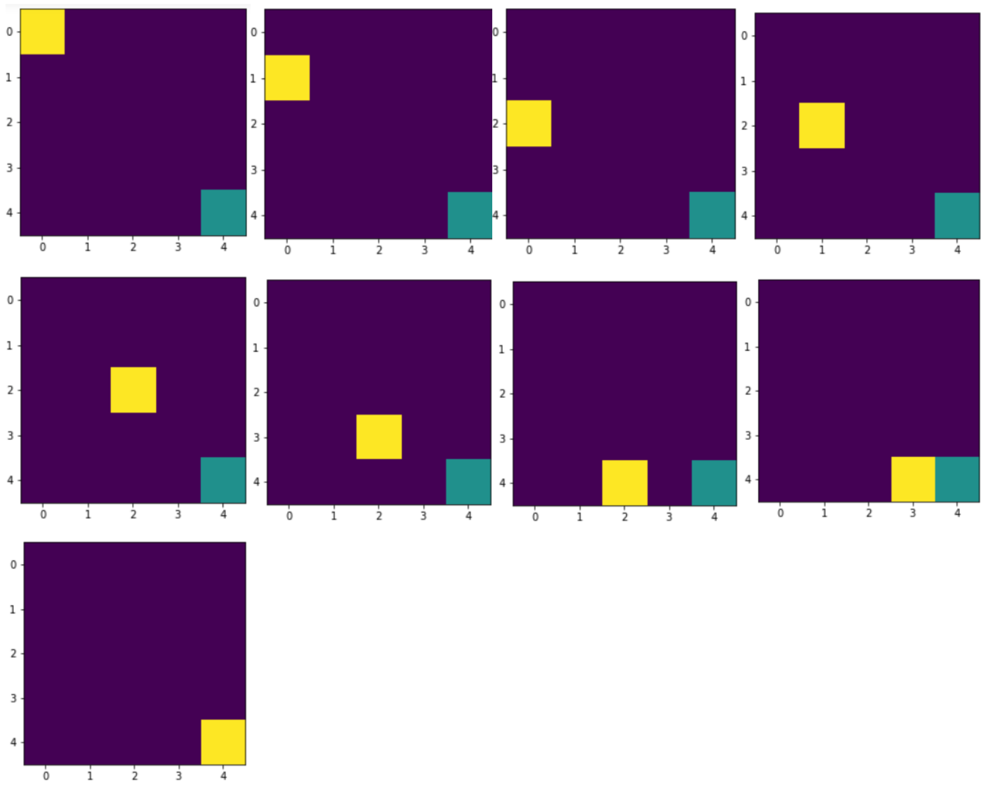


**Figure 7: Final run done by the agent to reach the goal of green square.**

## 7 Conclusion

I was able to make a Q-Learning agent with my defined policy and a q table updating function that was able to reach the goal in the least number of steps possible once it was trained. The rewards graph was a graph going upwards as episodes went on as expected. The final run done by the agent was indeed one of the best runs it could do for the environment.

### Acknowledgments

I am grateful to Professor Sargur Srihari for helping me understand the concepts of reinforcement learning, Q-Learning, and how environments work for which agents are to be trained.

### References

[1] https://www.youtube.com/watch?v=yMk_XtIEzH8

[2] https://www.youtube.com/watch?v=Gq1Azv_B4-4

[3] https://www.youtube.com/watch?v=CBTbifYx6a8

[4] http://gym.openai.com/docs/

[5] https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2

[6] https://nips.cc/Conferences/2018/PaperInformation/StyleFiles