CSE 574: Introduction to Machine Learning (Fall 2018)
Instructor: Sargur N. Srihari

Project 4: Tom and Jerry in Reinforcement Learning

December 5, 2018

Report By:
Siddheswar Chandrasekhar

Objective

This project combines Reinforcement Learning and Deep Learning. The task is to teach the agent to navigate in the grid-world environment.
We have modeled Tom and Jerry cartoon, where Tom, a cat, is chasing Jerry, a mouse. In our modeled game the task for Tom (an agent) is to find the shortest path to Jerry (a goal), given that the initial positions of Tom and Jerry are changing on every reset of the game.

To solve the problem, we would apply Deep Reinforcement Learning algorithm - DQN (Deep Q-Network), that was one of the first breakthrough successes in applying Deep Learning to Reinforcement Learning.

Plan of Action

1. Build a three-layer neural network
2. Implement exponential decay formula for epsilon
3. Implement Q-function

Reinforcement Learning

Reinforcement learning is a "less-supervised" sort of supervised learning. Generally speaking, our task will be to teach an agent "what to do" in a particular environment. That environment could be as concrete as a real or virtual world—like a robot in a room, or a character in a video game—or as abstract as a sentence parser reading some text. In a fully supervised regime, the agent would be told, through some sort of labels, what the right decision was at every step of its path. In a reinforcement learning regime, the agent is not told what the right decision is; instead, it only knows when something good or bad happens (positive or negative reward, respectively). It's the job of the reinforcement learning algorithm, then, to translate that perceived reward into the right decisions, given the other facts that it can perceive about its environment. [1]

Q – Learning

In Q – Learning, we'll be keeping track of Q-values, which are utilities associated with taking an action a from a given state '*s*'. The Q-value of the pair *(s,a)* is simply denoted *Q(s,a)*.

Then the algorithm is as follows:

1. Initialize $Q(s,a) = 0$ and $freq(s,a) = 0$, $\forall(s,a)$, and initialize the current state *s*.

2. Choose the best action a from the current state, according to the exploration function (using values from *freq(s,a)* as n).

3. Execute action *a*, and observe the reward *r*, as well as the next state, *s'*.

4. Update the Q-value for state s and action a according to the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + maxa'\ Q(s',a') - Q(s,a)]$$

5. Update current state $(s \leftarrow s')$.

6. Repeat from 2. until *s* is terminal.

Agent

Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward. We want our agent to decrease the number of random action, as it goes, so we introduce an exponential-decay epsilon, that eventually will allow our agent to explore the environment.

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda|S|},$$

Q – Table

The Q – Table can be thought of as a lookup table where we calculate the maximum expected future rewards for action at each state. This table would then guide us to make the best (maximum rewarding) action at each state.

Each Q – table score will be the maximum expected future reward that the agent will get if it takes that action at that state. This is an iterative process as we need to update the Q – table at each iteration / episode.

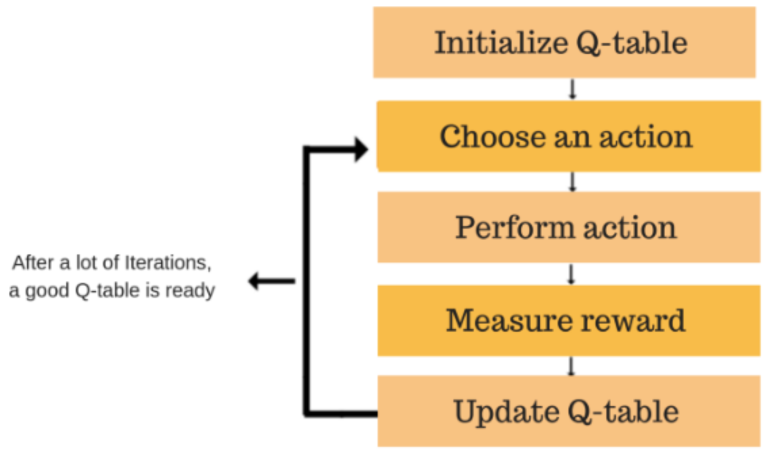

Fig 1 – Q-Table Psuedo

Our Q – Table is defined by:

$$Q_t = \begin{cases} r_t, & \text{if episode terminates at step } t+1 \\ r_t + \gamma max_a Q(s_t, a_t; \Theta), & \text{otherwise} \end{cases}$$

Neural Network

A neural network has a simple architecture that consists of an input layer, a black box which is a set of hidden layers, and an output layer. All the neurons in the network, connect with every neuron in its adjacent layer forming a mesh-like architecture.
The figure below is a good depiction of a simple neural network with three input neurons, one hidden layer of two neurons, and a single output neuron.

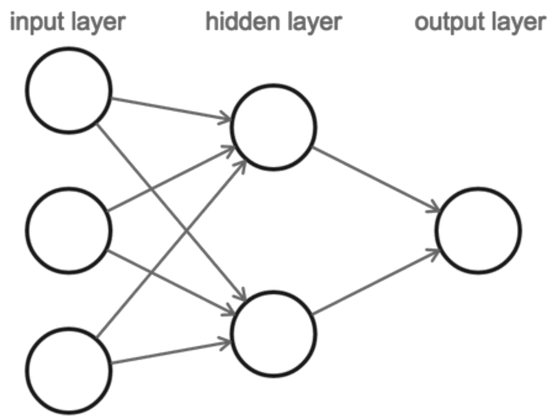

Fig 2 – Neural Network Architecture

We use ReLU (Rectified Linear Unit) as our Activation function for neural network which has the form:

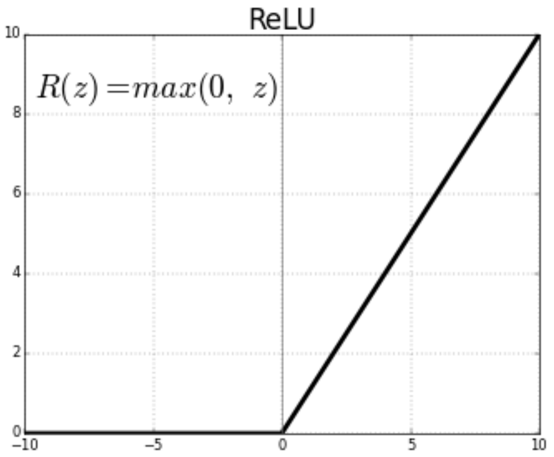

Fig 3 – ReLU Activation Function

Our neural network has the structure:

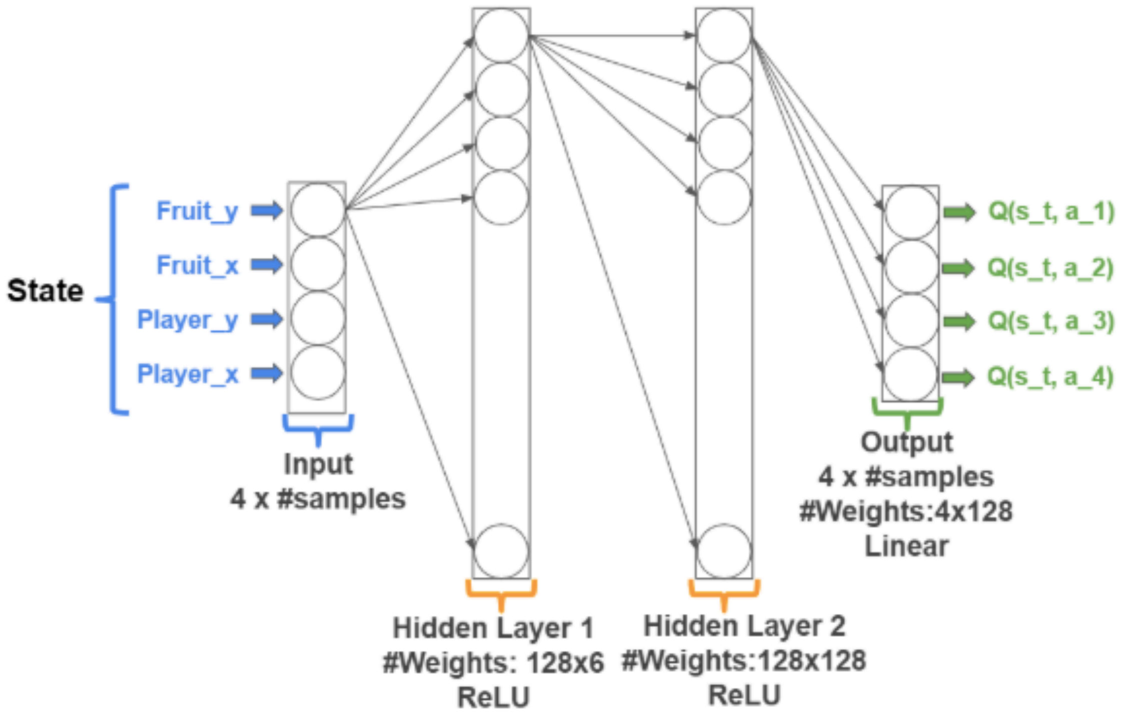

Fig 4 – Our Neural Network Model

As can be seen, our Deep Q-Network takes a stack of four-tuple as an input. It is passed through two hidden layers, and output a vector of Q-values for each action possible in the given state.

Questions

Q) What is the role of Neural Network, Epsilon and Q-Function in training the agent?

Answer:

The Neural Network is responsible for helping the agent make learned decisions during its exploitation phase. In the exploration phase, the Neural Network takes the state, action and reward as input and in exploitation phase predicts the action with the maximum rewards.

Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward. We want our agent to decrease the number of random action, as it goes, so we introduce an exponential-decay epsilon, that eventually will allow our agent to explore the environment.

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda|S|},$$

The Q-Function is the formula we use to fill the Q-Table. Each Q – table score will be the maximum expected future reward that the agent will get if it takes that action at that state.

Q) Can these be improved and how will it influence the training of the agent?

Answer:

The Neural Network model could possibly be improved to make predictions faster and better. The model we use in this project is a three-layered Neural Network with 128 hidden neurons in each hidden layer activated using the Rectified Linear Unit (ReLU) activation function.

Observations

With the nature of our environment set as deterministic, and with a batch size of 64, we have the following observations:

Number of Episodes	Gamma	Lambda	Min epsilon
10000	0.5	0.00005	0.05

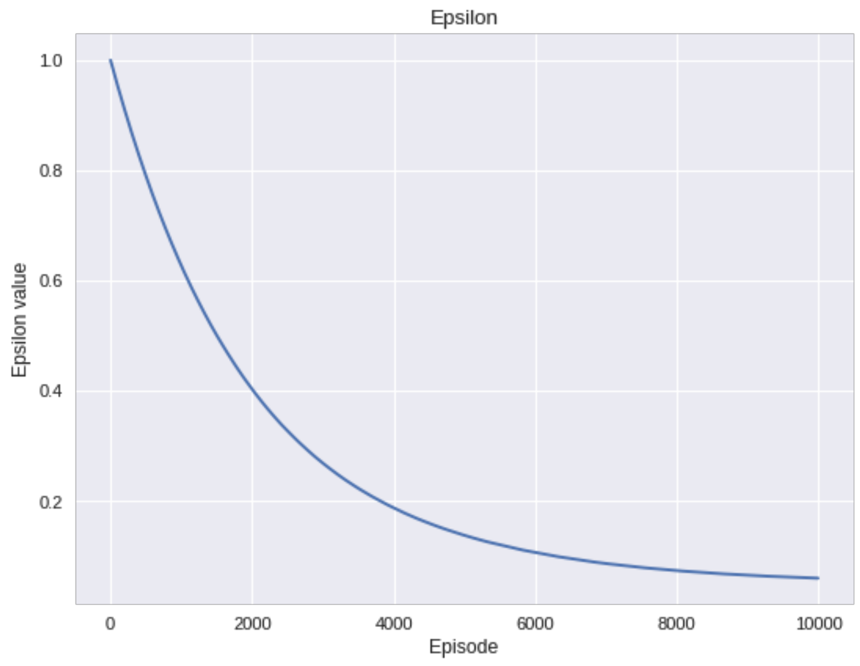

Fig 5 – Observation #1 Epsilon-decay graph

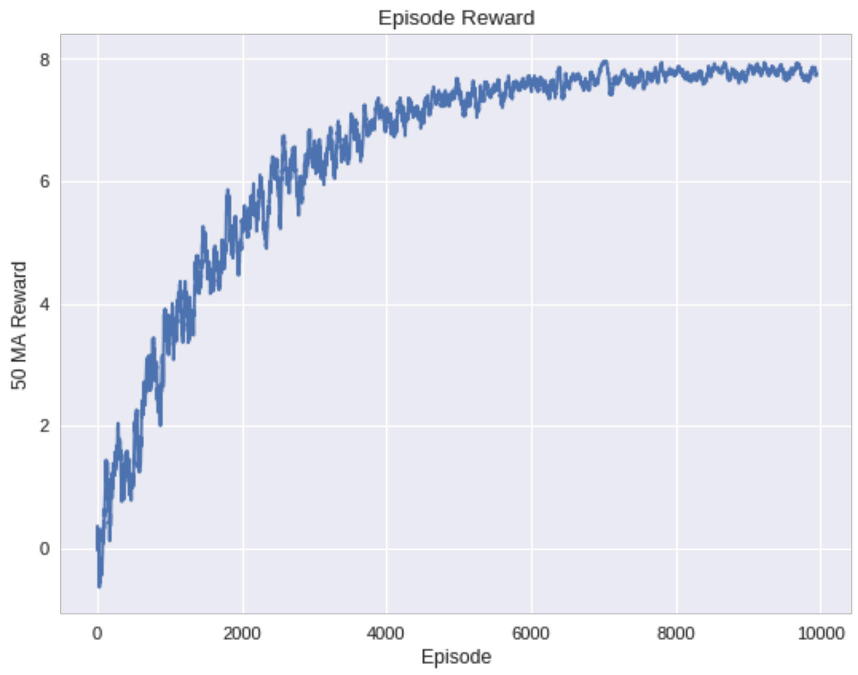

Fig 6 – Observation #1 Rewards graph

We observe decent results using these hyperparameters. A smooth epsilon decay and the agent gets enough time to explore the environment well enough. As can be seen, the agent was able to start making the most rewarding decisions after approximately 4,000 episodes.

Number of Episodes	Gamma	Lambda	Min epsilon
10000	0.5	0.0005	0.05

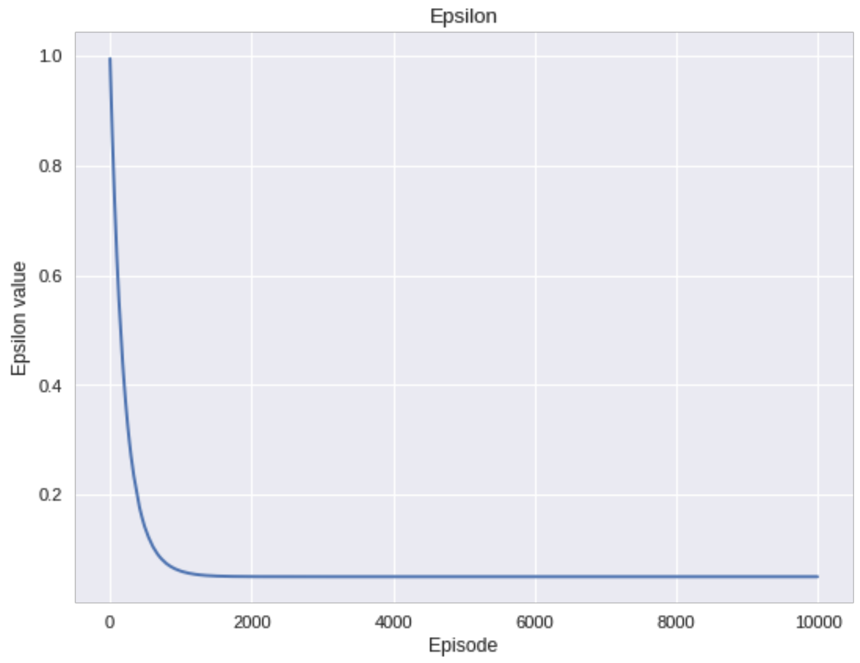

Fig 7 – Observation #2 Epsilon-decay graph

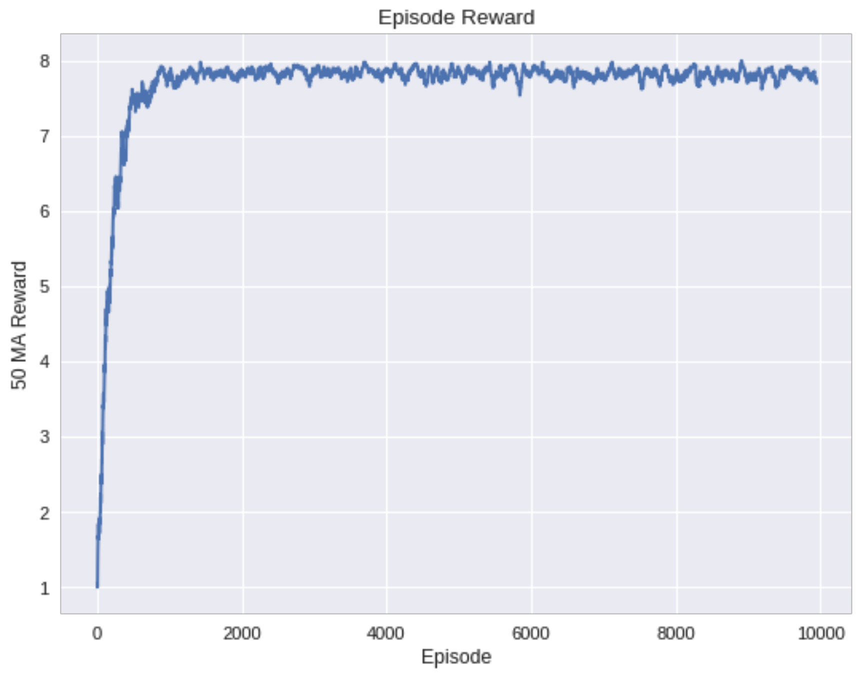

Fig 8 – Observation #2 Rewards graph

These set of hyperparameters fruit good results, and our agent starts making most rewarding decisions after approximately 500 episodes. But our agent is able to find only one path to the goal during exploration and uses only that path during exploitation after the first 1,000 episodes. Hence there is no more learning after the first 1,000 episodes (approximately) and running the game for 9,000 more episodes has no effect on our model

Number of Episodes	Gamma	Lambda	Min epsilon
10000	0.5	0.00001	0.01

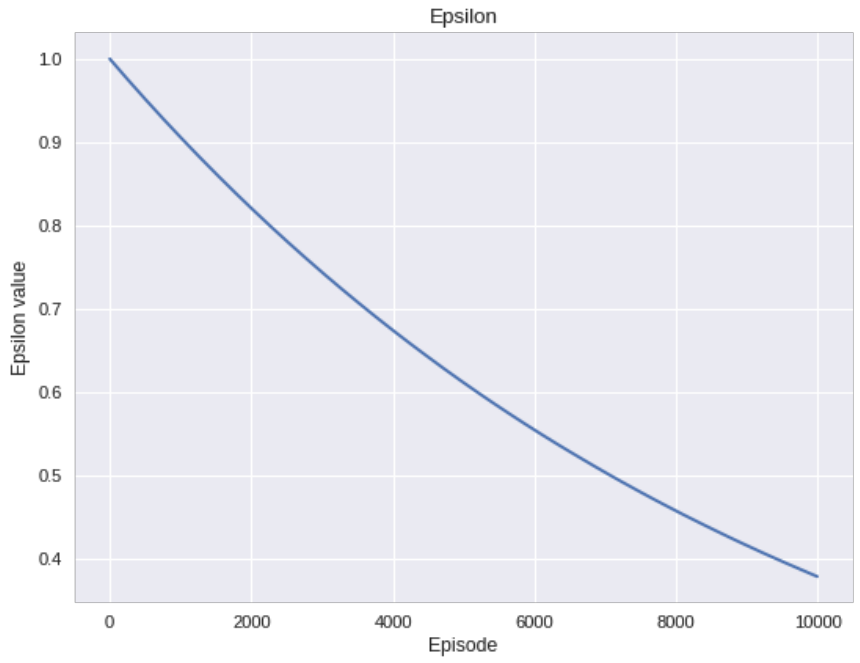

Fig 9 – Observation #3 Epsilon-decay graph

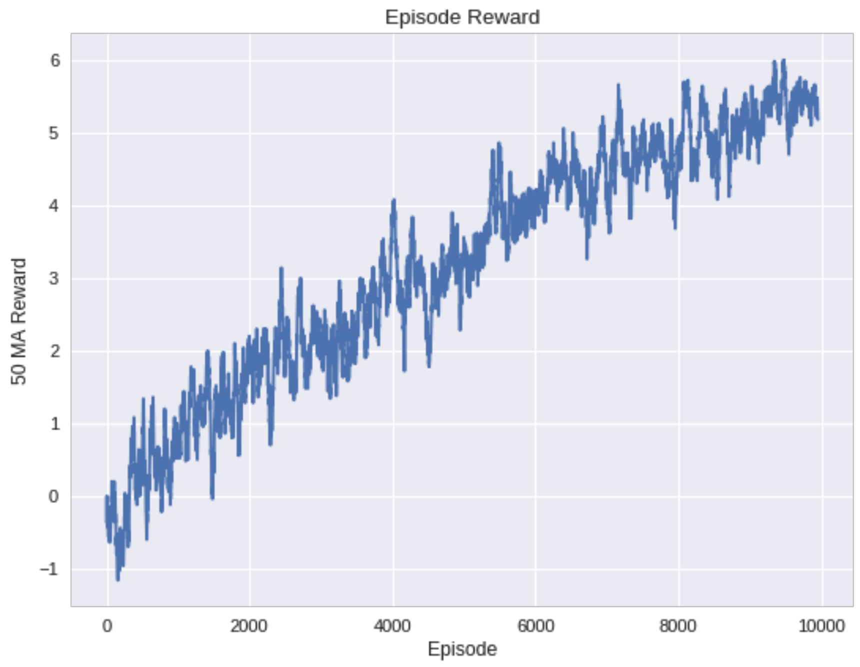

Fig 10 – Observation #3 Rewards graph

This is not a good set of hyperparameters. Our epsilon hasn't decayed enough and hence we get rewards mostly for the exploration stage and we are unable to test the performance as we haven't entered exploitation enough number of times.one solution is to increase the number of episodes, or to change the hyperparameters to make our agent learn (make epsilon decay) faster.

Number of Episodes	Gamma	Lambda	Min epsilon
20000	0.5	0.000025	0.01

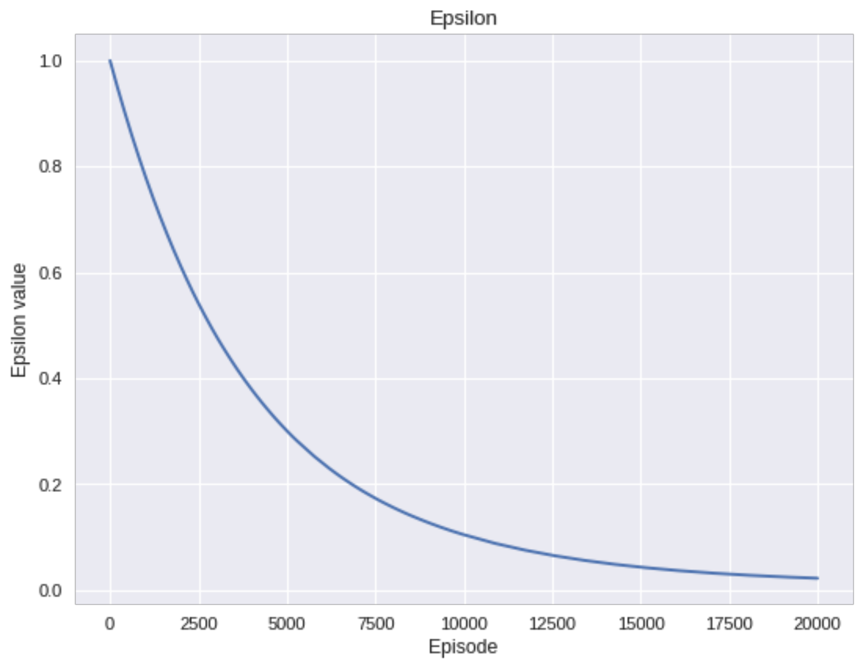

Fig 11 – Observation #4 Epsilon-decay graph

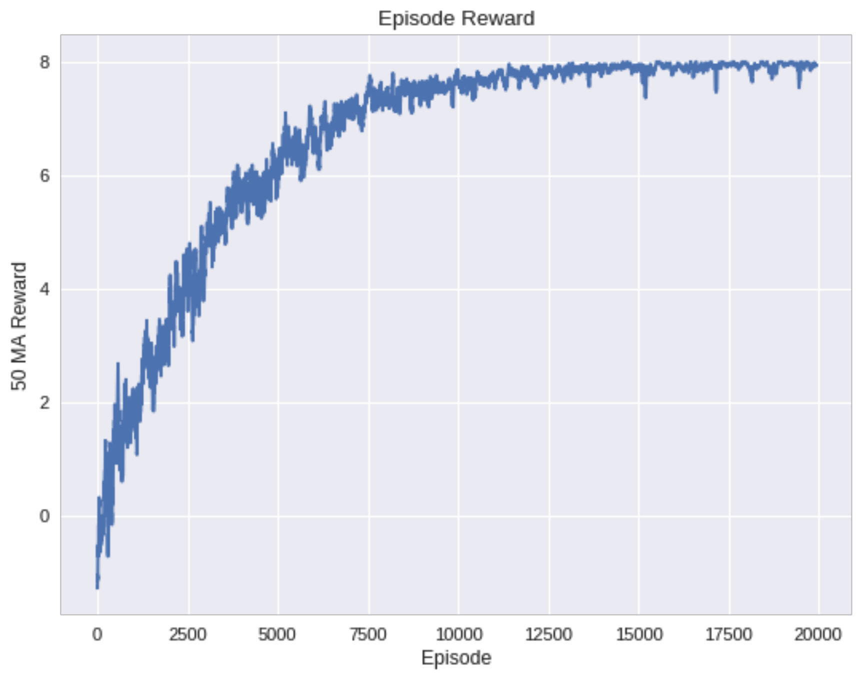

Fig 12 – Observation #4 Rewards graph

This is a pretty good model. Our epsilon decays smoothly and we get enough time to both explore and exploit sufficient number of times. We notice that after a point (approximately 10,000 episodes), we only get around 8 reward points, which means that the agent has found the optimal path/s. With these set of hyperparameters we also observe that unlike the previous observations, in this model, the agent has found multiple optimal paths to reach the goal. Though running the game for as high as 20,000 episodes is not ideal, here we can assure best learned decisions being made by the agent.

We now set the nature of our environment to non-deterministic (i.e. Tom and Jerry are placed randomly in the grid-world at every reset of the game). We get the following observations:

Number of Episodes	**Gamma**	**Lambda**	**Min epsilon**
5000	0.5	0.0005	0.01

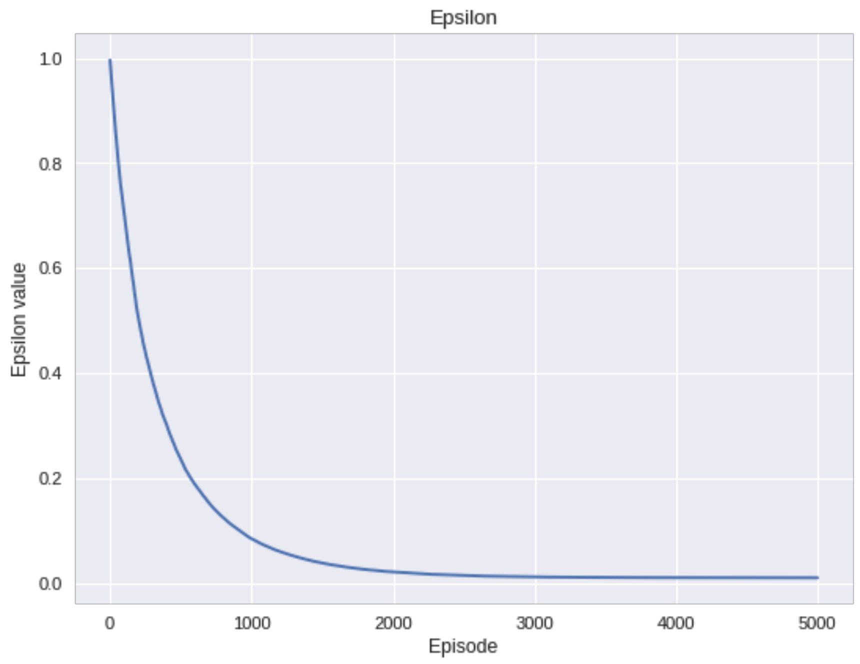

Fig 13 – Observation #5 Epsilon-decay graph

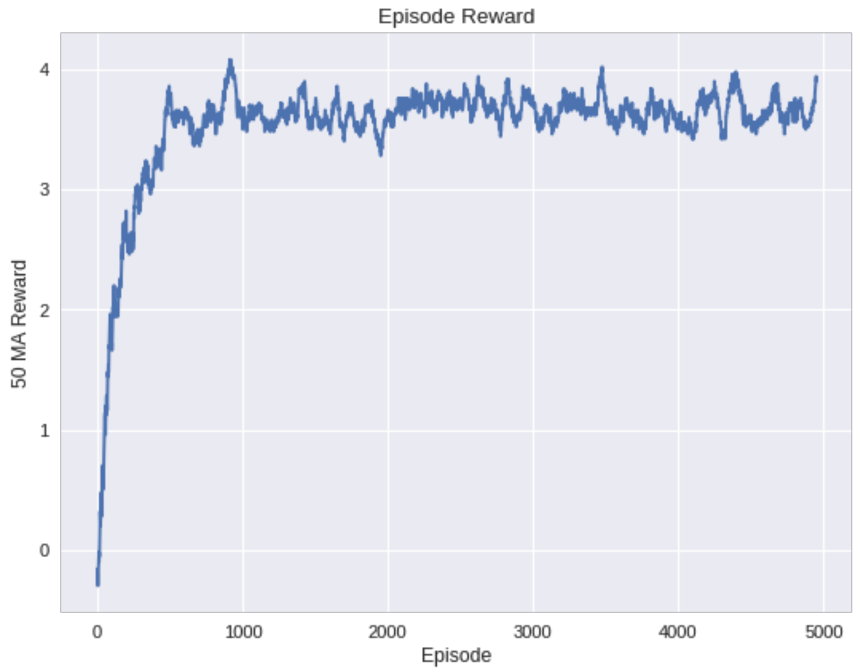

Fig 14 – Observation #5 Rewards graph

Seeing the rewards, we notice that Tom has still somehow caught Jerry, but we can deduce from the graphs that this isn't a learned decision by our agent.

Number of Episodes	**Gamma**	**Lambda**	**Min epsilon**
5000	0.5	0.00001	0.01

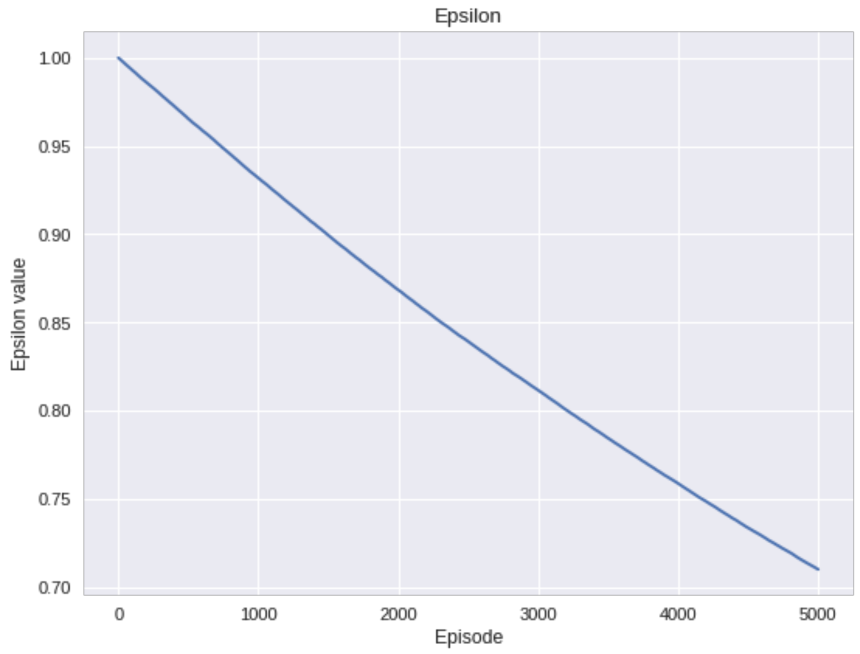

Fig 15 – Observation #6 Epsilon-decay graph

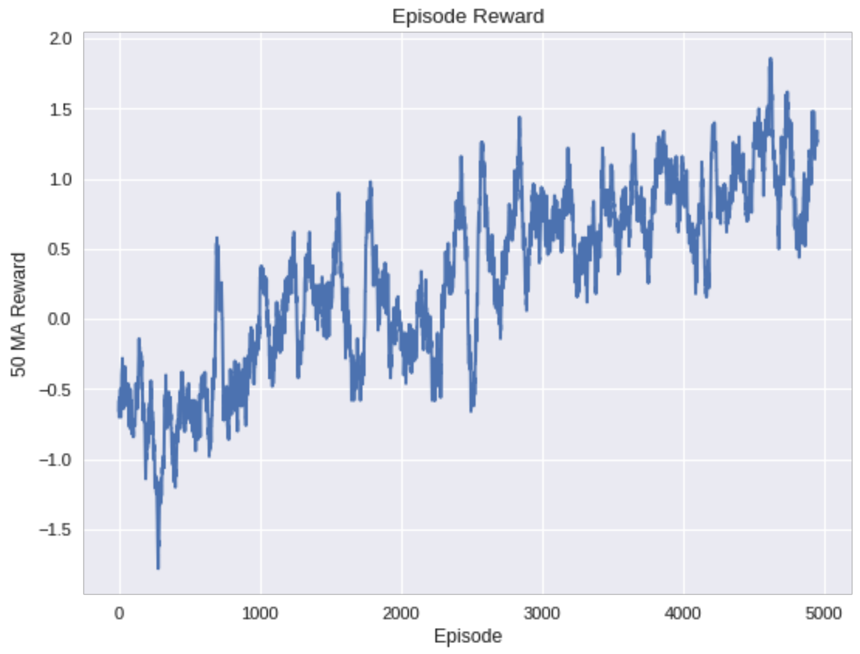

Fig 16 – Observation #6 Rewards graph

This is clearly not a good set of hyperparameters. We don't have a smooth epsilon decay and our epsilon hasn't decayed below 0.7 in the given number of episodes. Hence our agent hasn't exploited enough. We can solve this in multiple ways. One way is to increase the number of episodes, which is not a very ideal approach, the other is to tweak the other hyperparameters such that our agent is able to learn and start exploiting quicker.

Number of Episodes	Gamma	Lambda	Min epsilon
20000	0.5	0.000025	0.01

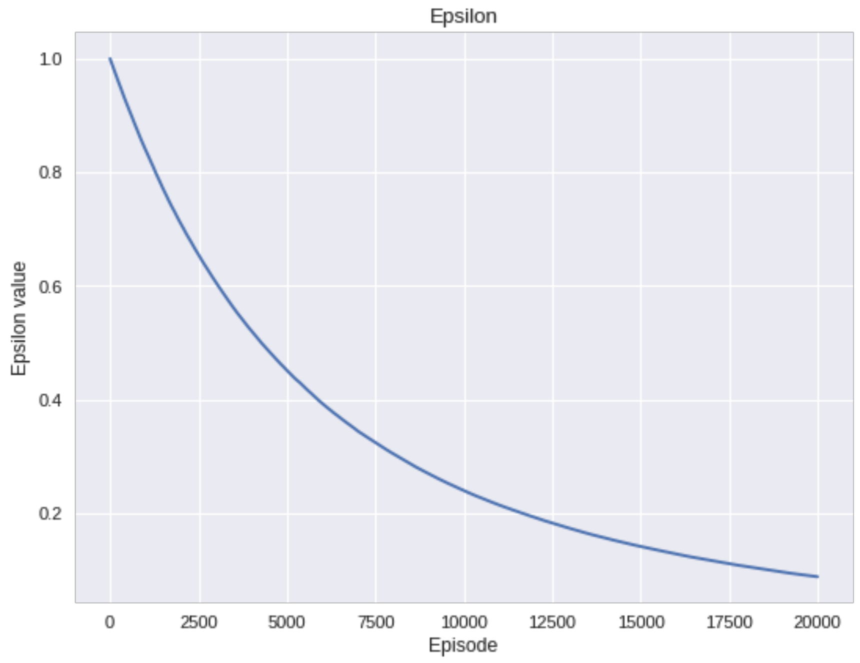

Fig 17 – Observation #7 Epsilon-decay graph

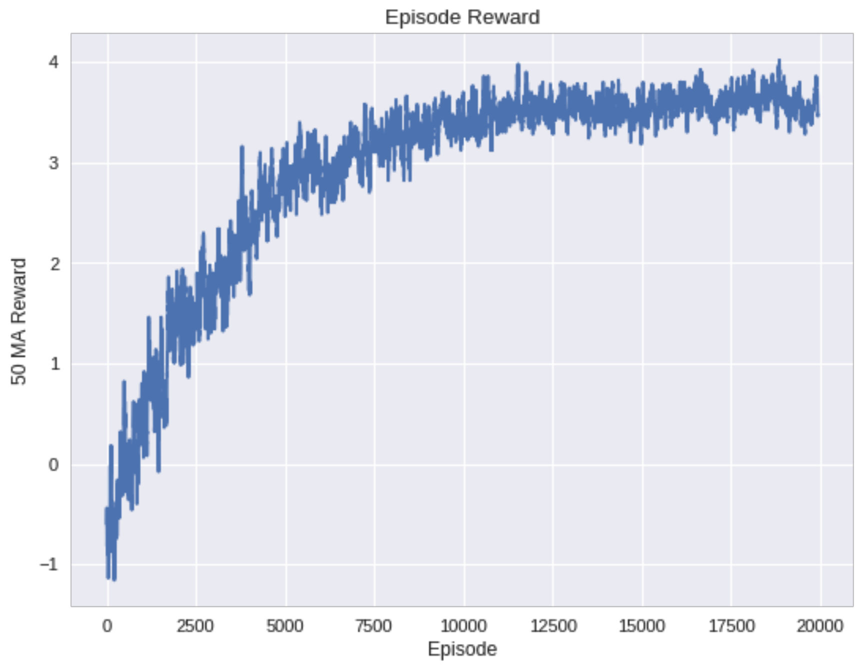

Fig 18 – Observation #7 Rewards graph

This is a pretty good model. Our epsilon decays smoothly and we get enough time to both explore and exploit sufficient number of times. We notice from the rewards graph, that the agent has found the optimal path/s to the goal. We see that after a certain number of episodes (approximately 10,000), Tom is able to catch Jerry regardless of where Jerry spawns in the environment. Hence, we can say that our agent is able to make learned decisions to reach the goal state.

Inferences

From the above observations, we can make the following infers:

- A higher 'lambda' λ value would mean that the epsilon ε would decay faster.
  This would imply that that agent would explore for less amount of time and start exploiting more often soon. This could mean that if the agent hasn't already found an optimal (or the most optimal) path to reach the goal state, then it never would and only try to reach the goal by the path it has found, which may or may not be optimal enough, because the agent hasn't explored the environment enough.

- We know that the number of episodes we run our game for, has no effect on the performance of our agent model, provided that the game is run for enough number of episodes needed for the agent to explore the world enough to learn about the optimal solutions to reach the goal state. This can be found by the rewards our agent receives. Beyond a point, the agent only gets that many rewards as it takes only one path to reach the goal and hence running the game for a greater number of episodes would not make any difference on the performance of the agent as the agent is said to already be in the exploitation stage and is just performing what it has learned in the exploration stage.

- A larger 'gamma' γ value would, as the formula suggests, imply that the agent puts more weightage to the discounted rewards of the next state. This could have negative effects on our model as a significantly high γ value (say 0.99) would result in the agent receiving significantly high (not the maximum though) rewards for actually moving away from the goal state. Though this would happen only if the environment isn't explored well enough, but a bad set of hyperparameters, one being a significantly high γ, would result in the agent moving only in a small set of paths, and in cases these paths may not even lead the agent to the goal state.

- We observe that our agent, Tom, is able to reach the goal state, i.e. catch Jerry, even if we change the nature of our environment to be non-deterministic (i.e. the positions of Tom and Jerry in the environment are random at every reset of the game).
  However, we would need to allow our agent to explore the world for a longer time.
  We can do this in various ways:
  1. Decrease λ : Would result in ε being decyed slower.
  2. Increase ε_{min} : Would force the agent to 'explore' more often
  3. Increase ε_{max} : Would force the agent to only 'explore' for the initial few episodes of the game (as the agent can only 'exploit' if the ε value is below 1)

Questions

Q) Explain what happens in reinforcement learning if the agent always choses the action that maximizes the Q-value. Suggest two ways to force the agent to explore.

Answer

In Reinforcement Learning, at every state the agent uses the concept of discounted rewards to predict the best possible move to make using the Q-function which has the form:

$$Q_t = \begin{cases} r_t, & \text{if episode terminates at step } t+1 \\ r_t + \gamma max_a Q(s_t, a_t; \Theta), & \text{otherwise} \end{cases}$$

However, these rewards are highly dependent on the extent to which the agent has 'explored' the environment, as only after sufficient explorations are the Q-table values (rewards for actions performed in each state) generated. Initially, the Q-table is filled with zeros. Hence, if the agent hasn't explored the environment enough, the Q-table would still have considerably large number of zeros.

The hand-off between whether the agent should perform 'exploration' or 'exploitation' is handled by epsilon which is exponentially decayed after each episode and has the form:

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda |S|},$$

Hence, we can say that the rate at which epsilon decays, is handled by the hyperparameter lambda. Higher lambda value would result in quick decay of epsilon which would result in the environment being explored less by the agent, and lower lambda value would result in slower decay of epsilon which would result in the environment being explored more by the agent. ^

As can be understood, maximize Q-value means our agent would try to get as high rewards as possible at each state. In other words, our agent would try to exploit whenever possible. As explained above, this would happen only if we have a high lambda value.

^ Here by more and less exploration of the environment we mean that the agent would tend to explore for larger/fewer number of episodes.

Therefore, we can say that if our agent always choses the action that maximizes the Q-value, then provided that we have reached our goal state at least once during exploration, the agent would tend to try and reach the goal state using the same path every time instead of trying and exploring the environment to find better (possibly shorter and more rewarding) paths that hasn't been explored yet.

This problem is a rather questionable behavior by the agent and something that is not acceptable by a 'good' model. Two possible ways this problem can be avoided in a model are:

1. Decrease λ (lambda):

   If we decrease lambda, our agent would tend to explore the environment more (i.e. for larger number of episodes) and hence the chances of our Q-table of being filled entirely increases considerably which would result in our agent trying to get to the goal using multiple optimal paths.

2. Increase ε_{min} (minimum epsilon):

   The exponential decay formula for epsilon has two variables (and hyperparameters), minimum epsilon and maximum epsilon.
   Maximum epsilon is the starting value of epsilon i.e. the value from which epsilon starts decaying and minimum epsilon is the value below which epsilon doesn't decay.

   Since epsilon decides whether our agent would 'explore' or 'exploit' the environment, a larger value for minimum epsilon would force our agent to 'explore' the environment more often.

   As in the case of decreasing lambda, this would increase the chances of our Q-table of being filled entirely, which would then result in our agent trying to get to the goal using multiple optimal paths.

3. Increase ε_{max} (maximum epsilon):

   Just like increasing the ε_{min}, the ε_{max} can also be increased to avoid this problem.
   Since our agent could only possibly exploit after the ε decays below 1, setting a ε_{max} above 1, would force our agent to only explore in the initial few episodes (until the ε decays to below 1).

Q) Calculate Q-value for the given states and provide all the calculation steps.

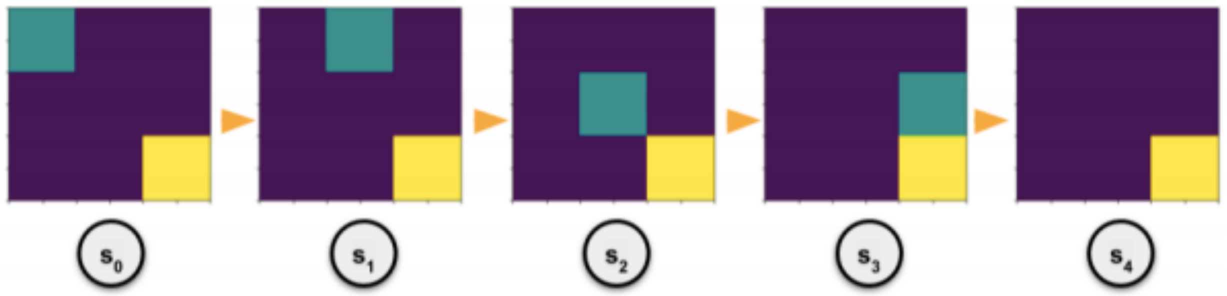

Fig 19 – Question #2 State Model

Consider a deterministic environment which is a 3x3 grid, where one space of the grid is occupied by the agent (green square) and another is occupied by a goal (yellow square). The agent's action space consists of 4 actions: UP, DOWN, LEFT, and RIGHT. The goal is to have the agent move onto the space that the goal is occupying in as little moves as possible. The episode terminates as soon as the agent reaches the goal.
Initially, the agent is set to be in the upper-left corner and the goal is in the lower-right corner. The agent receives a reward of:

1 when it moves closer to the goal
-1 when it moves away from the goal
0 when it does not move at all (e.g., tries to move into an edge)

Consider the following possible optimal episode and their resulting states, that reach the goal in the smallest number of steps. In the example below, s4 is a terminal state.

The agent takes the following sequence of actions: RIGHT → DOWN → RIGHT → DOWN.
It is an optimal path for the agent to take to reach the goal (although this is not the only possible optimal path). Our task is to fill out the Q-Table for the above states, where $\gamma = 0.99$.

Answer

We use the following formula to calculate our Q-Table values:

$$Q_t = \begin{cases} r_t, & \text{if episode terminates at step } t+1 \\ r_t + \gamma max_a Q(s_t, a_t; \Theta), & \text{otherwise} \end{cases}$$

Our Q-value Calculations are as follows:

State 3

- Q (S3, Down) = r (S3, Down)
  = 1

- Q (S3, Up) = r (S3, Up) + γ max{ Q (?) }
  = -1 + 0.99*0
  = -1

- Q (S3, Left) = r (S3, Left) + γ max{ Q (S2) }
  = -1 + 0.99*(?)
  = ?

- Q (S3, Right) = r (S3, Right) + γ max{ Q (S3) }
  = 0 + 0.99*1
  0.99

State 2

- Q (S2, Left) = r (S2, Left) + γ max{ Q (?) }
  = -1 + 0.99*0
  = -1

- Q (S2, Down) = r (S2, Down) + γ max{ Q (?) }
  = 1 + 0.99*0
  = 1

- Q (S2, Right) = r (S2, Right) + γ max{ Q (S3) }
  = 1 + 0.99*1
  = 1.99

- Q (S2, Up) = r (S2, Up) + γ max{ Q (S1) }
  = -1 + 0.99*(?)
  = ?

State 1

- Q (S1, Right) = r (S1, Right) + γ max{ Q (?) }
  = 1 + 0.99*0
  = 1

- Q (S1, Down) = r (S1, Down) + γ max{ Q (S2) }
  = 1 + 0.99*1.99
  = 2.97

- Q (S1, Left) = r (S1, Left) + γ max{ Q (S0) }
  = -1 + 0.99*(?)
  = ?

- Q (S1, Up) = r (S1, Up) + γ max{ Q (S1) }
  = 0 + 0.99*2.97
  = 2.94

State 0

- Q (S0, Down) = r (S0, Down) + γ max{ Q (?) }
  = 1 + 0.99*0
  = 1

- Q (S0, Right) = r (S0, Right) + γ max{ Q (S1) }
  = 1 + 0.99*2.9
  = 3.94

- Q (S0, Left) = r (S0, Left) + γ max{ Q (S0) }
  = 0 + 0.99*3.94
  = 3.9

- Q (S0, Up) = r (S0, Up) + γ max{ Q (S0) }
  = 0 + 0.99*3.94
  = 3.9

We can now calculate the Q-values of the ones we skipped (left as ‘?’) above

- Q (S3, Left) = r (S3, Left) + γ max{ Q (S2) }
  = -1 + 0.99*1.99
  = 0.97

- Q (S2, Up) = r (S2, Up) + γ max{ Q (S1) }
  = -1 + 0.99*2.97
  = 1.94

- Q (S1, Left) = r (S1, Left) + γ max{ Q (S0) }
  = -1 + 0.99*3.94
  = 2.9

After all the calculations, our Q-Table looks as follows:

State / Action	**Up**	**Down**	**Left**	**Right**
S_0	3.9	1	3.9	3.94
S_1	2.94	2.97	2.9	1
S_2	1.94	1	-1	1.99
S_3	-1	1	0.97	0.99
S_4	0	0	0	0

References

1. Reinforcement Learning
   http://web.cse.ohio-state.edu/~stiff.4/cse3521/reinforcement-learning.html