CSE 574: INTRODUCTION TO MACHINE LEARNING

(FALL 2018)

Instructor: **Prof. S. N. Srihari**

University at Buffalo

PROJECT 4.0 REPORT

Submitted by-

ANIRUDDHA SINHA (UB Person No - 50289428)

Date: December 5, 2018

Aim

Play a game and teach an *agent* to navigate in a grid-world environment and reach a *goal,* in the shortest path.

In our modeled game –

- **Agent** is **Tom** (a cat) and the **goal** is **Jerry** (a mouse)
- Tom is supposed to find the shortest path to reach Jerry, when both their initial positions are known.
- We design a **Deep Reinforcement Learning Algorithm – DQN (Deep Q-Network)** for achieving these tasks.

Reinforcement Learning and Deep Q-Networks

Reinforcement learning is a concept in Machine Learning where an agent learns its future actions based on the results from the previous and present actions.

- It performs trial-and-error actions and learns from the feedback rewards from its environment received from these actions to take the future actions.
- The objective is to associate each state to its actions to maximize the final reward as the result of learning.
- The agent is supposed to learn a method of operation, a control policy that **maximizes the expected sum of rewards**.
- The future rewards are discounted exponentially using a discounting factor.

Reinforcement learning brings to use **Markov Decision Process** for receiving the feedback and taking future actions.

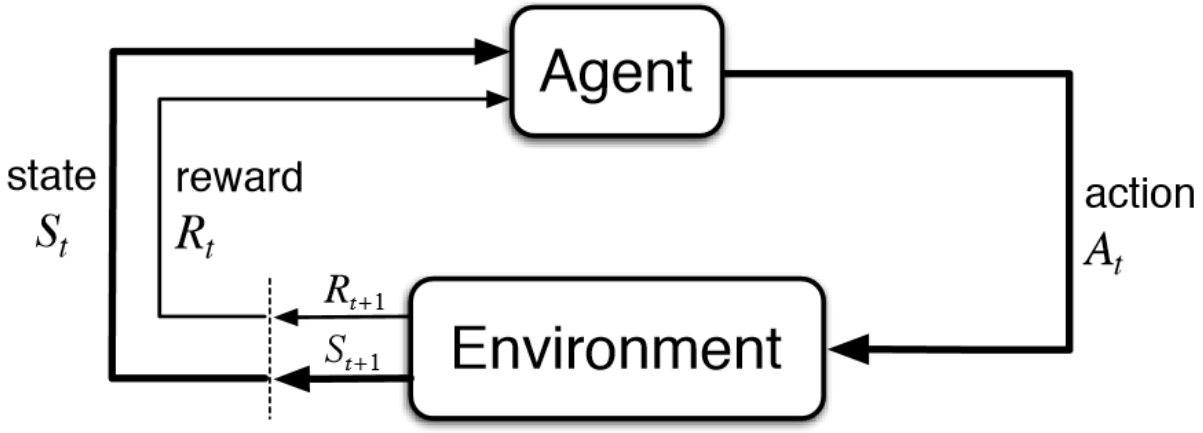


Fig. 1. The Markov Decision Process

How does it work?

First, let's understand the terminology and what they mean in this task –

- **Action (A):** all possible moves the agent can take. In this task, **there are 4 possible actions – UP, DOWN, LEFT, RIGHT**

- **State (S)**: The current position/state that the agent is in, returned by the environment. **We have a 5x5 grid environment** and the agent can have any of the 25 states from these, for this task.

- **Reward (R)**: The reward that is returned for a particular action immediately from the environment to evaluate the last action. In this task, we have 3 defined rewards –

    - +1: When the agent moves takes an action and moves to a state closer to the goal
    - 0: When the agent does not move at all
    - -1: When the agent takes an action and moves to a state away from the goal

- **Policy (Q)**: The strategy the agent employs to determine next action based on the current state. Here, we will use the concept of Q-learning.

- **Value (V)**: An expected long-term return with discount, as opposed to short-term reward.

After, having an idea of all these terms, let's dive into the ways in which **our game is designed**. Given the environment with the states, rewards and the agent –

- We start an episode, i.e. we take an initial state s_o and pass it to the agent.
- The agent passes an action a_o based on the state s_o back to the environment
- The environment reacts to the action, passes the next state s_1 along with the resulting reward, r_o for taking that action, back to the agent.
- The process continues until the agent reaches a terminal state, indicating the end of this episode.
- Repeat the same process for further episodes and train the agent based on all its feedbacks.

So, ***in Q-learning***, **we follow a value-based approach** and build a **memory table Q(s,a)** to store Q-values for all possible combinations of *s* and *a.* This means that we sample an action from the current state and calculate the reward and the new state *s'*. From the memory table, we determine the next action *a'* to take for the maximum *Q(s',a')*. So, the optimized value of Q(s,a) is achieved by the **Bellman equation** –

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a'),$$

Where, r(s,a) = reward for an action *a* in state *s*

γ = discounting factor

Need of a Deep Neural Network:

When the combinations of states and actions are too large, the memory and the computational power requirement for Q will be very high. Thus, we need a deep neural network – **DQN,** to approximate Q (s,a). Hence, the name **Deep Q-Learning.**

Our new aim is to generalize the approximation of the Q-value function rather than remembering the solutions.

Importance of Discounting Factor:

It's important to maintain a check on the future rewards. The discounting factor γ basically penalizes these rewards. $\gamma \in$ **[0, 1]**. We need this because –

- The future rewards do not always provide immediate best results
- Discounting eases mathematical convenience.
- Future rewards may/may not have higher uncertainty, and we need to check that.

DQN- Algorithm:

Initialize replay memory to capacity N

Initialize the environment (reset)

For episode $= 1, M$ **do** (Begin a loop of interactions between the agent and environment)

Initialize the first state $s_0 = s$

For $t = 1, T$ **do**

With probability ϵ select a random action a_t, otherwise select $a_t = argmax_a Q(s, a; \Theta)$

Execute action a_t and observe reward r_t and next state s_{t+1}

A tuple $< s_t, a_t, r_t, s_{t+1} >$ has to be stored in memory

Sample random minibatch of observations (s_t, a_t, r_t, s_{t+1}) from memory

Calculate Q-value

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

Train a neural network on a sampled batched from the memory

End **For**

End **For**

Exploration and Exploitation:

In Reinforcement Learning, there are two important concepts – *exploration* and *exploitation* that dictate how the model should pattern its learning process.

- **Exploitation:** The agent learns to take actions that come from the current best version of the learned methods. Here, the agent puts to use all its previous observations and trains them in a deep neural network to find the best actions for the following states.
  It seeks for the actions that it knows will achieve a high reward.

- **Exploration:** Keep training on new and more actions to obtain more data. Basically, keep moving in the environment and keep exploring.

Both these policies can be used and switched to each other using a **decay function** – ***epsilon decay function.*** The function is given as –

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

where ϵ_{min}, $\epsilon_{max} \in [0,1]$

λ - hyperparameter for epsilon

$|S|$ - total number of steps

The agent first takes actions based on ϵ - *the exploration rate*, where it explores and tries all actions before it sees some patterns of best rewards and Q-values. When not deciding the action randomly, the agent will always go for the highest reward, which we want to reduce. With the epsilon-decay function, the randomly taken actions will be reduced, and more exploration will be introduced.

Experience Replay

We train all previous observations and experiences from the Markov Decision Process implementing the experience replay. It tells the agent to not only learn from the recent observations but from most previous observations. We try to sample some previous observations randomly from what the agent has learnt. Next, we simply pass them through the **Brain** of the agent to process and train on them.

Neural Networks (The BRAIN):

The Brain of our agent is designed using a Deep Neural Network in Keras. The features of this NN have been discussed in the implementation part, however, it looks similar to the one shown in Fig.2.

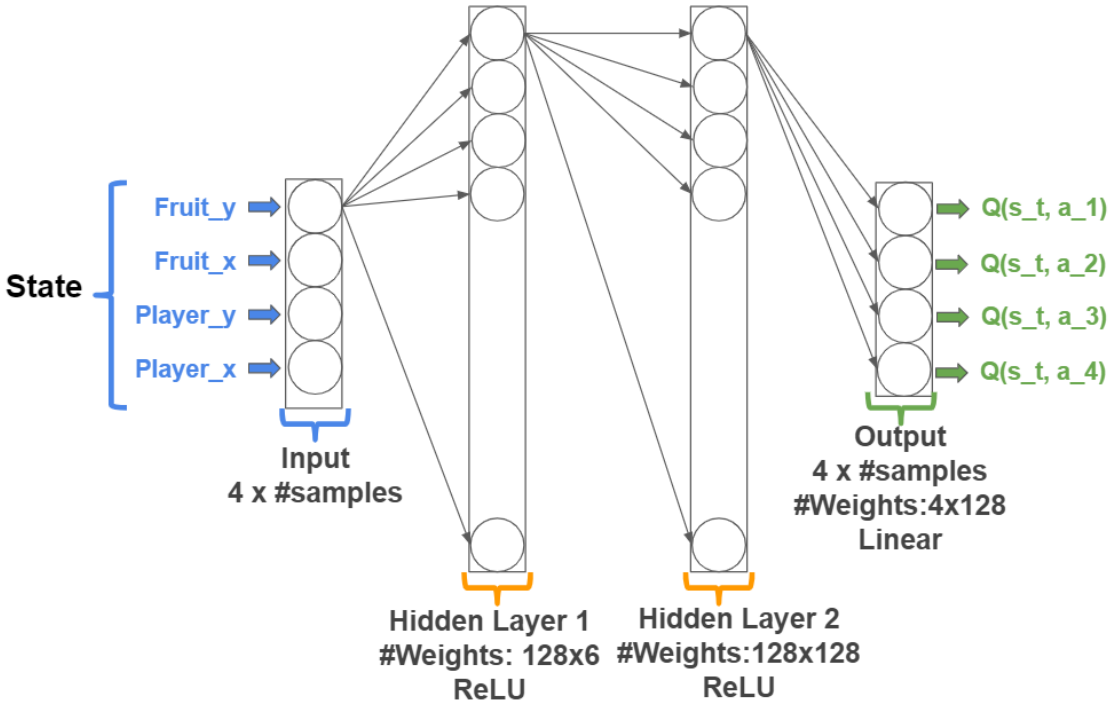


Fig.2 Deep Q – Network for this task

We have –

- A **Sequential model** in Keras.
- The model has an input layer and one or more hidden layers.
- The input layer takes the batch of input samples and trains them using weights and **activation functions**. The input layer takes an `input_size` or `input_dim` that determines the size of each sample.
- The output layer gives the final trained outputs from the input and has a size `output_size`.
- The model runs for a certain number of iterations, called `epochs`.

Implementation

❖ We start off by importing all the necessary libraries for the code.

❖ Next, **Creating the Environment:**

- Define the grid (5 x 5) and store images of Tom, Jerry and both being together for the grid.
- Define the methods for ways of movement by Tom – UP, DOWN, LEFT and RIGHT.
- No movement will be done if the grid ends in the direction of movement.

- Check for the current state after the last movement.
- Get the rewards for each movement/action. +1 for moving towards the goal, 0 for not moving at all and -1 for moving away from the goal.
- Check if the agent has reached the goal or if the number of episodes are over.
- Render the movements of Tom on the grid.
- Reset the environment after completion of the episodes or at the terminal state.

❖ **Define the Brain:**

- Create a sequential model with **3 hidden layers.**
- Input size for first dense layer = `self.state_dim` (= 4 in this case)
- Size of **first two hidden layers = 128** (number of nodes)
- Size of last layer = `self.action_dim` (= 4 in this case, since there are 4 possible actions)
- **Activations** for the hidden layers: **`relu, relu, linear`**, respectively.
- **Optimizer =** `rmsprop, learning rate = 0.00025`
- **Loss =** `Mean Square Error`
- Compile the model using `mode.compile`
- Define the class-function to train the model using `model.fit` and, to predict the model using `model.predict`

The same has been implemented in the code as follows –

```
model.add(Dense(units=128, activation = 'relu', input_dim = input_size))

model.add(Dropout(0.1))

model.add(Dense(units=128, activation = 'relu'))

model.add(Dropout(0.2))

model.add(Dense(units=output_size, activation = 'linear'))

model.summary()
```

❖ **Develop the Agent's Memory:**

- Initialize a particular capacity for the agent's memory.
- Add all observations to the agent's memory.

❖ **Define Agent's behavior:**

- Initialize all parameters that will control the agent's behavior. These include – `state_dim, action_dim, memory_capacity, batch_size, gamma` (discounting factor), `lambda, max_epsilon` and `min_epsilon`

- Actions of the agent depend on epsilon, if the agent will explore or exploit. If the agent will **exploit** (act), we send the samples from previous observations for predictions of Q-values in the Brain.
- If the agent will **explore** (observe), take actions, keep moving around the grid and add the observations to the agent's memory. However, with more number of steps, agent can't always keep exploring. Use the following equation –

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

The same has been implemented in the code as –

```
self.epsilon = self.min_epsilon + (self.max_epsilon - self.min_epsilon) * np.exp(-self.lamb*np.abs(self.steps))
```

- So, decrease the size of epsilon to lead the agent towards exploitation over time.
- Calculate the best Q-values using **action-replay.** Seek random samples of previous observations from the behavior and update best Q-values.
- For this, we use the following conditions –

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

We implement this in the code as follows –

```
for i in range(batch_size):
      # Observation
      obs = batch[i]

      # State, Action, Reward, Next State
      st = obs[0]; act = obs[1]; rew = obs[2]; st_next = obs[3]

      # Estimated Q-Values for Observation
      t = q_vals[i]

      ### START CODE HERE ### (≈ 4 line of code)

      if st_next is None:
        t[act] = rew
      else:
        t[act] = rew + self.gamma*np.max(q_vals_next[i])

      ### END CODE HERE ###

     # Set training data
      x[i] = st
      y[i] = t

    # Train
    self.brain.train(x, y)
```

❖ **Finally run the game:**

- Initialize the grid as 5x5.
- Define all hyper-parameters and initializing parameters.
- Hyper-parameters include lambda, epsilon max/min, number of episodes, gamma
- Call all the above methods in sequence
- We check for the **maximum reward – mean rolling reward,** and try to achieve this value in the range of 6 to 8.
- Plot the necessary graphs.
- Also, save the movement of agent in the grid over all the number of episodes in the form of an animation in local storage, only for reference and fun.

Running the game and Test Cases:

We evaluate the efficiency of learning of our agent by running a few-cases and tuning the hyper-parameters.

Hyper-parameters:

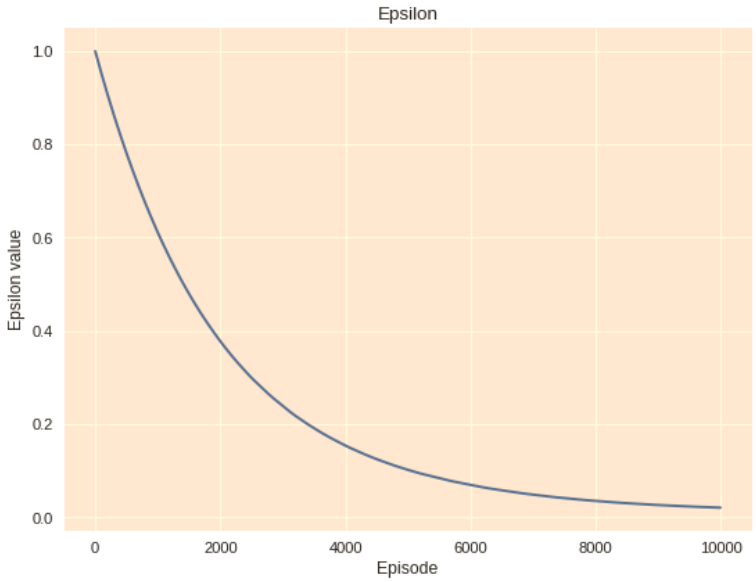
`Number of episodes, lambda,` discount factor – **`gamma,`** range of epsilon = $\epsilon_{max} - \epsilon_{min}$

*** We evaluate for the mean-rolling reward, which can be at max 8 for a 5x5 grid. We also see how it stabilizes, i.e. change in its magnitude is by only less than 0.1 and what was the time to stabilize it.

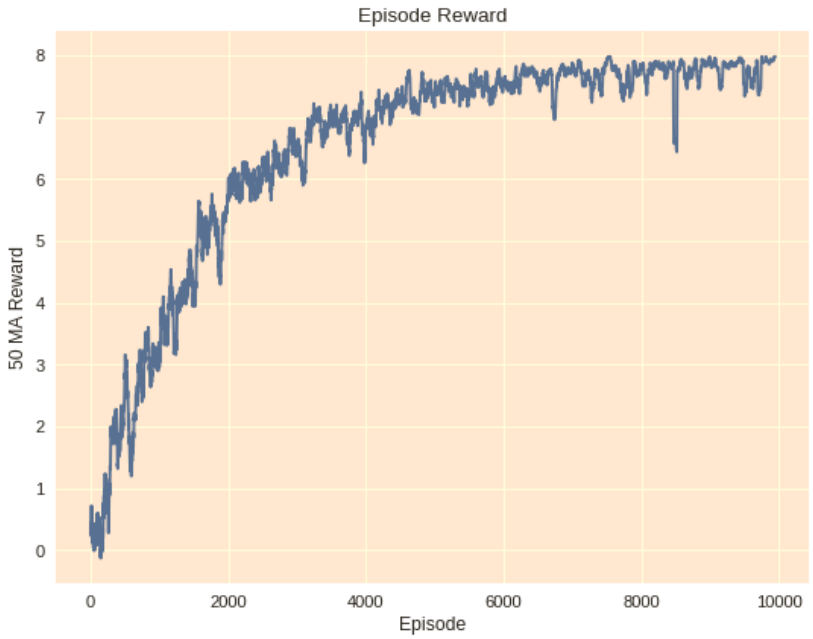
Case 1.

Hyper-parameter	**Value**
Number of Episodes	10000
Epsilon Range	0.05 – 1
Lambda	0.00005
Gamma	0.99
Last Epsilon Value	0.1
Mean – Rolling Reward	**6.4386**
Episodes for convergence:	**5500**

Time Elapsed (seconds)	**787.75**



(a)

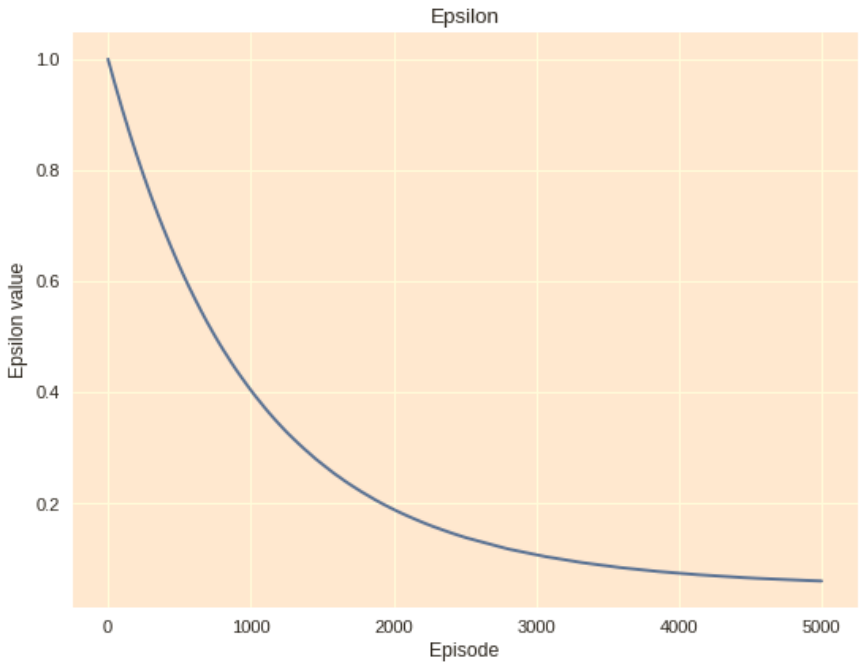


(b)

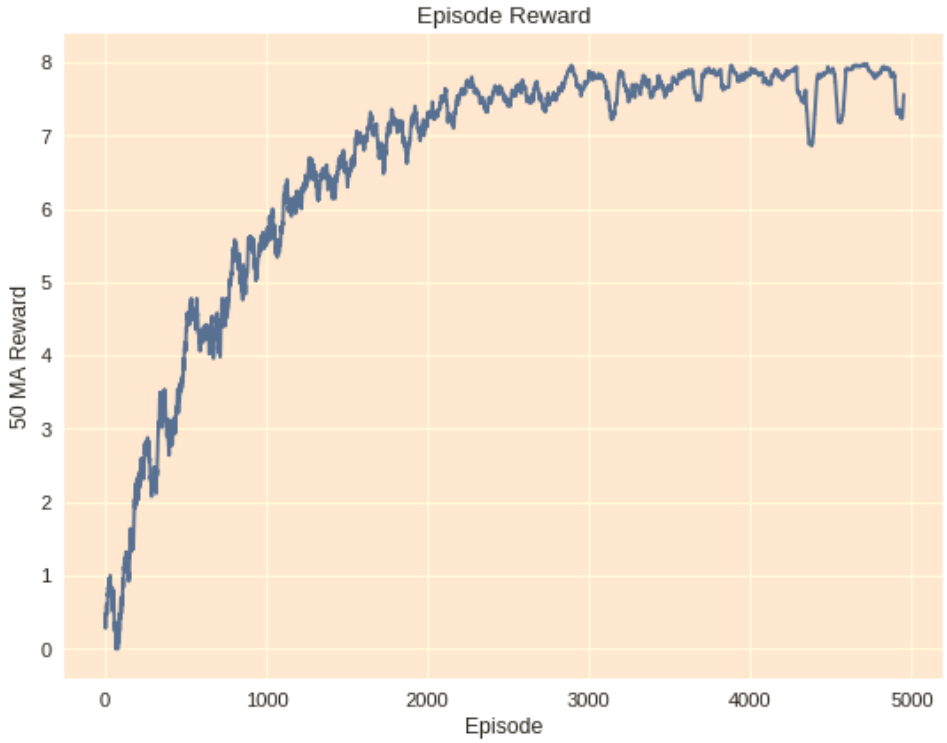
Fig.3. (a) Graph of number of number of episodes vs epsilon values for Case1
(b) Graph of number of episodes vs rewards for Case 1

Case 2.

Hyper-parameter	**Value**
Number of Episodes	5000
Epsilon Range	0.05 – 1
Lambda	0.0001
Gamma	0.99
Last Epsilon Value	0.07
Mean – Rolling Reward	**6.5929**
Episodes for convergence:	**3100**
Time Elapsed (seconds)	**421.71**



(a)

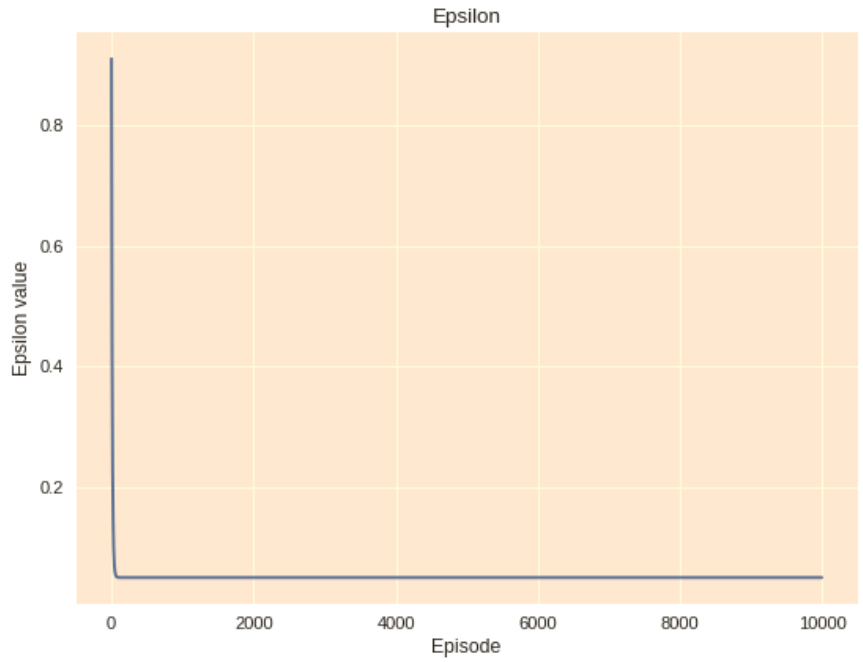


(b)

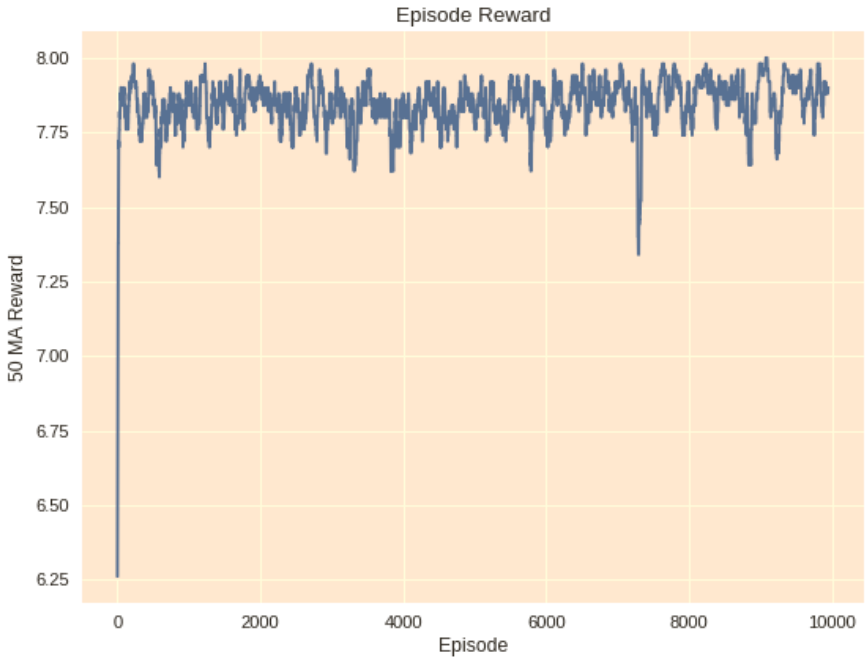
Fig. 4 (a) Graph of number of number of episodes vs epsilon values for Case2
(b) Graph of number of episodes vs rewards for Case 2

Case 3.

Hyper-parameter	**Value**
Number of Episodes	10000
Epsilon Range	0.05 – 1
Lambda	0.01
Gamma	0.5
Last Epsilon Value	0.05
Mean – Rolling Reward	**7.8370**
Episodes for convergence:	**4000**
Time Elapsed (seconds)	**807.23**



(a)

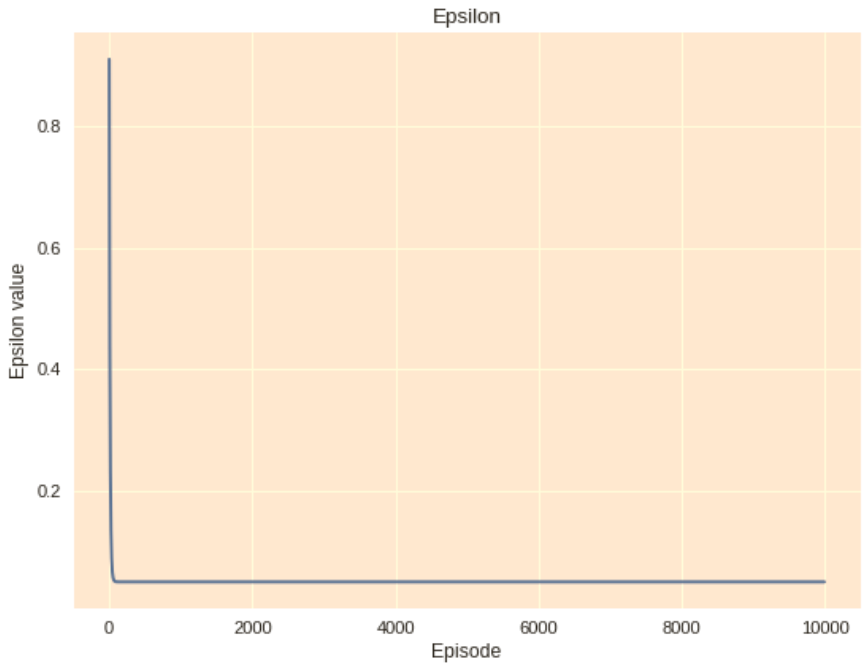


(b)

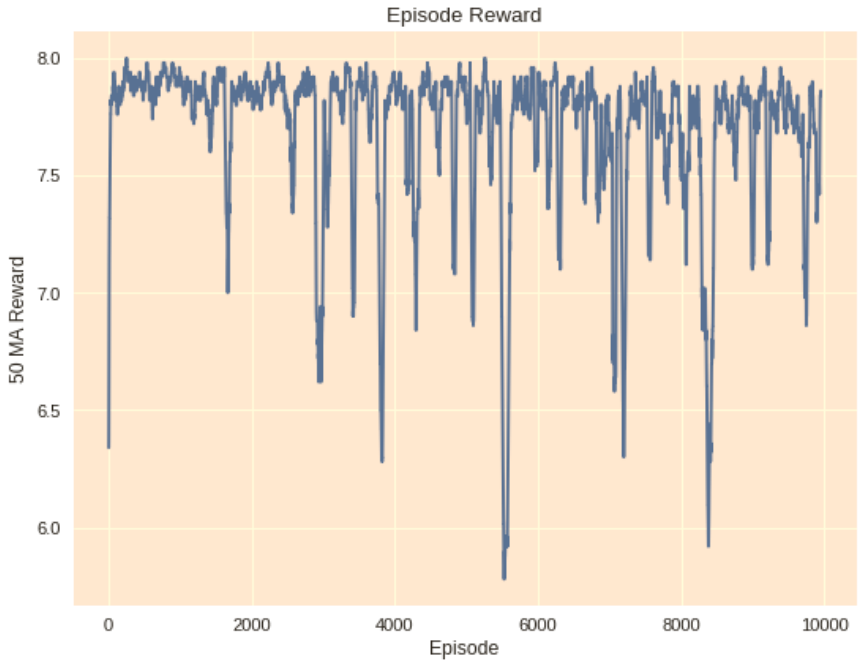
Fig. 5 (a) Graph of number of number of episodes vs epsilon values for Case 3
(b) Graph of number of episodes vs rewards for Case 3

Case 4.

Hyper-parameter	**Value**
Number of Episodes	10000
Epsilon Range	0.05 – 1
Lambda	0.01
Gamma	0.99
Last Epsilon Value	0.05
Mean – Rolling Reward	**7.6879**
Episodes for convergence:	**2800**
Time Elapsed (seconds)	**833.72**



(a)

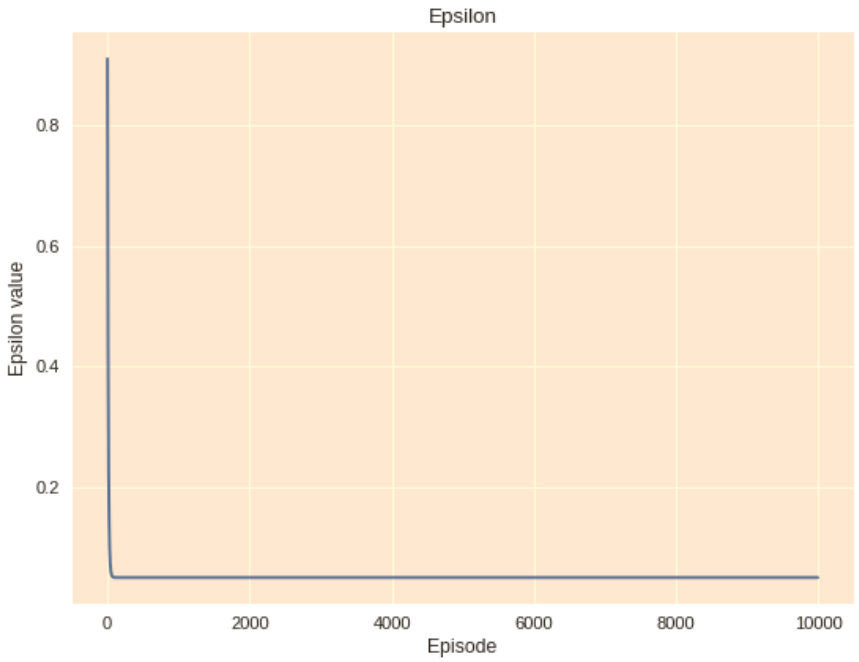


(b)

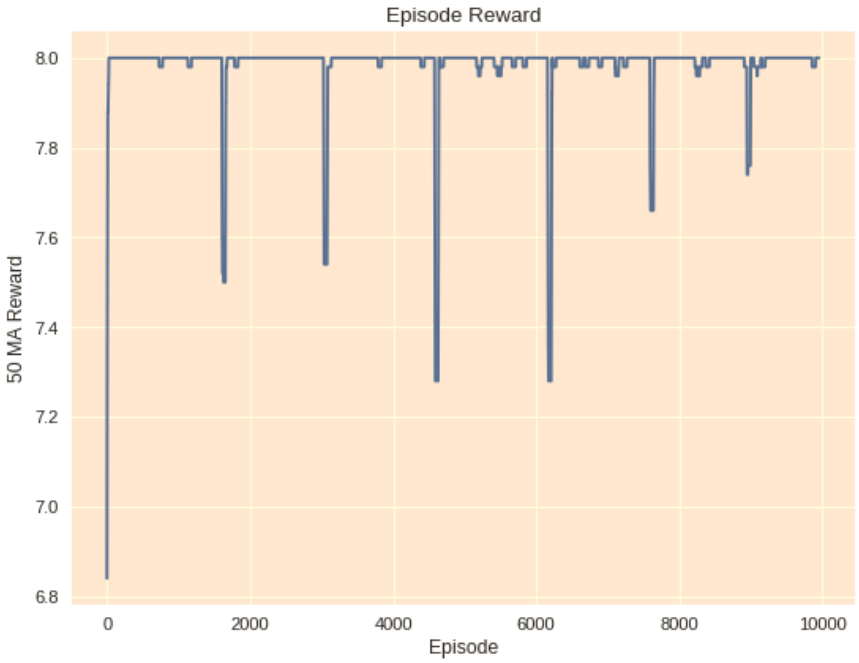
Fig. 6 (a) Graph of number of number of episodes vs epsilon values for Case 4
(b) Graph of number of episodes vs rewards for Case 4

Case 5.

Hyper-parameter	**Value**
Number of Episodes	10000
Epsilon Range	0.001 – 1
Lambda	0.01
Gamma	0.5
Last Epsilon Value	0.001
Mean – Rolling Reward	**7.9759**
Episodes for convergence:	**1500**
Time Elapsed (seconds)	**793.15**



(a)



(b)

Fig. 7 (a) Graph of number of number of episodes vs epsilon values for Case 5
(b) Graph of number of episodes vs rewards for Case 5

****** The best scenario was seen in Case 5, with lambda = 0.01, high epsilon range and number of episodes = 10000.**

Questions:

- **What parts have I implemented?**

As we can see in the report, for this task, we have implemented the 3-layer neural network using Keras, the exponential decay formula for epsilon and the Q-function.

- **What is their role in training the agent?**

The neural networks help in developing the prediction methods and processing policy of the agent that helps it in learning the results of the actions from the environment and in making future decisions for selecting the states that maximize the rewards.

The epsilon-decay function forces the agent to explore less over time and reduce randomization, rather exploit the observations made in the memory. This leads to reducing time in finding the best states and the actions that should lead to those states.

The Q-function is quintessential after the agent has started moving in the grid. Once, it has made sufficient amount of observations, it is pertinent to train it over its feedbacks or experiences and select the optimal states. This is done by the Q-function.

- **Can these snippets be improved and how it will influence the training of agent?**

The neural networks can be improved by increasing the number of hidden layers.

The epsilon-decay function can be replaced by other decay functions to improve the learning.

Since DQN depends on the learning model of Q-values, which is based on the Q-function, it is not possible to change the Q-function (the Bellman equation). Manipulating that would result in creating a different learning technique altogether.

- **How quickly were your agents able to learn?**

The agents was most quickly able to learn in the last case, Case 5. That was because we increase the epsilon decay function by increasing the epsilon range and increasing the value of lambda.

WRITING TASKS

Q1. What will happen to reinforcement learning if the agent always chooses the maximizes the Q-value? Suggest two ways to force the agent to explore.

We learn the Q-values from trial and error. First, we initialize the Q, perform an action and update by the reward. Initially, randomness plays an important role as the agent keeps exploring, but the algorithm finds the best or optimal Q-values for each state and action. It calculates the maximum future reward for a state and action, by taking into account the immediate reward and the maximum of all future rewards for the next state.

Now, the above nature of prediction shows that **calculation of Q-values or Q-matrix is greedy in nature**. This implies that reinforcement learning will always choose the action with the best value. However, **this can ignore any small-important action with very small probability and yet produce a very large reward**. ***Thus, reinforcement learning will take a lot of time for the agent to learn properly the optimal actions for achieving the goal.***

To avoid this, we keep the epsilon-decay function to introduce randomization and exploration time-to-time in the learning policy. The agent will perform a random move, without considering the optimal policy.

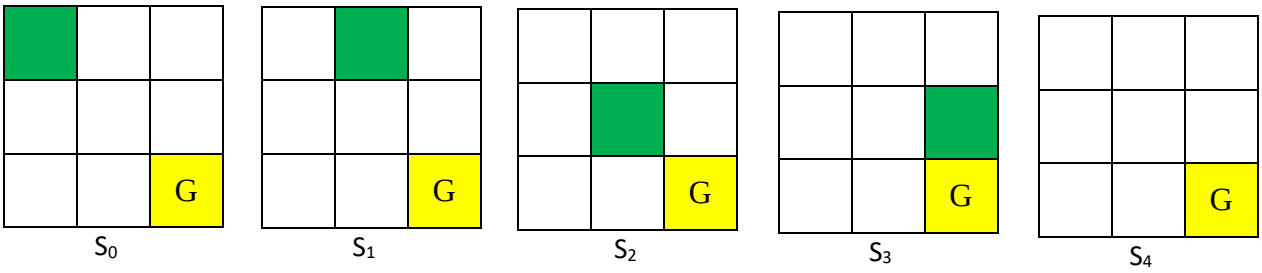
Forcing the agent to explore:

The agent can be forced towards exploration in many ways, however, two of them are –

- **Introducing randomization:** Take random actions every now and then such that all possible actions are covered.
- **Design a model-based approach:** Compute a selection of action based on what its expected reward will be and how certain is the model for achieving that reward.

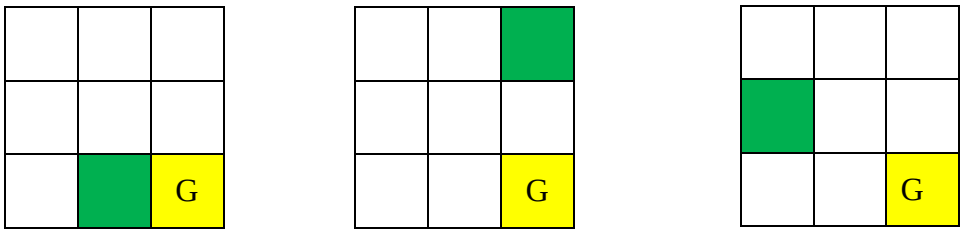
Q2.

We have the following states given to us –



S_0 S_1 S_2 S_3 S_4

G represents the goal. However, while calculating the Q-table, there will be few $\mathbf{S_{t+1}}$ states that are not yet defined in the above states. Hence, let's define few more intermediate states as follows:



S_5 S_6 S_7

Now, we calculate the Q-table based on the Bellman equation.

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

Here, rewards = 0 for no movement, 1 for movement towards the goal, -1 for movement away from the goal.

$\boldsymbol{\gamma = 0.99}$

Thus, let's calculate the Q-values for each state now. We start from the last state, i.e., S_4.

Since, no further movement is possible from S_4, all values = 0.

STATE, ACTION	CALCULATION	VALUE
S_4, UP	0 + γ *max (S_4, a) = 0 + 0.99 x 0	0
S_4, DOWN	0 + γ *max (S_4, a) = 0 + 0.99 x 0	0
S_4, LEFT	0 + γ *max (S_4, a) = 0 + 0.99 x 0	0

S_4, RIGHT	$0 + \gamma$ *max (S_4, a) = 0 + 0.99 x 0	0

State S_3

S_3, UP	$-1 + \gamma$ *max (S_6, a) = -1 + 0.99 x 1.99	0.97
S_3, DOWN	$+1 + \gamma$ *max (S_4, a) = +1 + 0.99 x 0	1
S_3, LEFT	$-1 + \gamma$ *max (S_2, a) = -1 + 0.99 x 1.99	0.97
S_3, RIGHT	$0 + \gamma$ *max (S_3, a) = 0 + 0.99 x 0	0.99

State S_2

S_2, UP	$-1 + \gamma$ *max (S_1, a) = -1 + 0.99 x 2.97	1.94
S_2, DOWN	$+1 + \gamma$ *max (S_5, a) = +1 + 0.99 x 1	1.99
S_2, LEFT	$-1 + \gamma$ *max (S_7, a) = -1 + 0.99 x 2.97	1.94
S_2, RIGHT	$+1 + \gamma$ *max (S_3, a) = +1 + 0.99 x 1	1.99

State S_1

S_1, UP	$0 + \gamma$ *max (S_1, a) = 0 + 0.99 x 2.97	2.94
S_1, DOWN	$+1 + \gamma$ *max (S_2, a) = +1 + 0.99 x 1.99	2.97
S_1, LEFT	$-1 + \gamma$ *max (S_0, a) = -1 + 0.99 x 2.97	2.90
S_1, RIGHT	$+1 + \gamma$ *max (S_6, a) = +1 + 0.99 x 1.99	2.97

State S_0

S_0, UP	$0 + \gamma$ *max (S_0, a) = 0 + 0.99 x 3.94	3.90
S_0, DOWN	$+1 + \gamma$ *max (S_7, a) = +1 + 0.99 x 2.97	3.94
S_0, LEFT	$0 + \gamma$ *max (S_0, a) = 0 + 0.99 x 3.94	3.90
S_0, RIGHT	$+1 + \gamma$ *max (S_1, a) = +1 + 0.99 x 2.97	3.94

In order to suffice our calculation, we calculated the values of the intermediate states, S_5, S6, S_7 as –

State S_5

S_5, UP	-1 + γ *max (S_2, a) = -1 + 0.99 x 1.99	0.97
S_5, DOWN	0 + γ *max (S_5, a) = 0 + 0.99 x 1	0.99
S_5, LEFT	-	-
S_5, RIGHT	+1 + γ *max (S_4, a) = +1 + 0.99 x 0	1

State S_6

S_6, UP	0 + γ *max (S_6, a) = 0 + 0.99 x 1.99	1.97
S_6, DOWN	+1 + γ *max (S_3, a) = +1 + 0.99 x 1	1.99
S_6, LEFT	-1 + γ *max (S_1, a) = -1 + 0.99 x 2.97	1.94
S_6, RIGHT	0 + γ *max (S_6, a) = 0 + 0.99 x 2.97	1.97

State S_7

S_7, UP	-1 + γ *max (S_0, a) = -1 + 0.99 x 3.94	2.90
S_7, DOWN	-	-
S_7, LEFT	0 + γ *max (S_7, a) = 0 + 0.99 x 2.97	2.94
S_7, RIGHT	+1 + γ *max (S_2, a) = +1 + 0.99 x 1.99	2.97

[The – values in the tables above do not contribute to the calculation of the Q-table for the current givent states and are dependent on other new states, and hence have been left blank.]

Thus, the final Q-table has been updated as below –

STATE	UP	DOWN	LEFT	RIGHT
0	3.90	3.94	3.90	3.94
1	2.94	2.97	2.90	2.97
2	1.94	1.99	1.94	1.99
3	0.97	1	0.97	0.99
4	0	0	0	0

References

- https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4
- https://sergioskar.github.io/Deep_Q_Learning/
- https://skymind.ai/wiki/deep-reinforcement-learning
- https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8
- http://rail.eecs.berkeley.edu/deeprlcourse/
- https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288