Machine Learning Project 4

Dhanashree Solanke
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
ddsolank@buffalo.edu

Abstract

We have modelled Tom and Jerry game where the task is that Tom catches Jerry in a grid based environment. We used Reinforcement Learning, more specifically DQN, as an approach to solve this.

1 Reinforcement Learning

In Reinforcement Learning, the agent learns how to act in a environment by performing actions and seeing the results. It is based on reward hypothesis. Correct actions will earn the agent positive points and wrong actions will earn negative points. The goal can be described as maximizing the cumulative reward in each action the agent will perform.

2 Q Learning

The central idea in Q-Learning is to recognize or learn the optimal action in every state visited by the system via trial and error. The agent chooses an action, obtains feedback for that action, and uses the feedback to update its memory. In its memory, the agent keeps a so-called Q-factor for every state-action pair. When the feedback for selecting an action in a state is positive, the associated Q-factor's value is increased, while if the feedback is negative, the value is decreased. The feedback consists of the immediate reward plus the value of the next state. The immediate reward is denoted by r(i, a, j), where i is the current state, a the action chosen in the current state, and j the next state. The value of any state is given by the maximum Q-factor in that state. Thus, if there are two actions in each state, the value of a state is the maximum of the two Q-factors for that state.

$$\text{feedback} = r(i, a, j) + \lambda \, \text{maxbQ}(j, b),$$

where λ is the discount factor, which discounts the values of future states. Usually, $\lambda = 1/(1 + R)$ where R is the rate of discounting.

3 Model

We have implemented a reinforcement learning algorithm - DQN, that will train the agent to play a game.

3.1 Parts Implemented

- **Deep Q Learning (3 Layer Neural Network)**
  When we have a very large number of state-action pairs, it is not feasible to store very Q-factor separately. Then, it makes sense to store the Q-factors for a given action within one neural network. When a Q-factor is needed, it is fetched from its neural network. When a Q-factor is to be updated, the new Q-factor is used to
  update the neural network itself. For any given action, Q(i, a) is a function of i, the state.

  In our implementation, we have built a three-layer neural network with two hidden layers.
  Activation function for the first and second hidden layers is 'relu'. Activation function for the final layer is 'linear'. Input dimensions for the first hidden layer equals to the size of your observation space. Number of hidden nodes is 128. Number of the output same as the size of the action space.

```
### START CODE HERE ### (≈ 3 lines of code)

model = Sequential()
model.add(Dense(128, input_dim=state_dim, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dense(action_dim, activation='linear'))


### END CODE HERE ###
```

- **Exponential Decay Formula for Epsilon**
  We specify an exploration rate “epsilon,” which we set to 1 in the beginning. This is the rate of steps that we’ll do randomly. In the beginning, this rate must be at its highest value, because we don’t know anything about the values in Q-table. This means we need to do a lot of exploration, by randomly choosing our actions. We generate a random number. If this number > epsilon, then we will do “exploitation”. Else, we’ll do exploration. The idea is that we must have a big epsilon at the beginning of the training of the Q-function. Then, reduce it progressively as the agent becomes more confident at estimating Q-values.

```
### START CODE HERE ### (≈ 1 line of code)
temp = math.exp(-(self.lamb)*abs(self.steps))
self.epsilon = self.min_epsilon +( self.max_epsilon-self.min_epsilon)*temp
### END CODE HERE ###
```

- **Q- function**
  The Q-function uses the Bellman equation and takes two inputs: state (s) and action (a).

$$\underbrace{\boxed{Q^{\pi}(s_t, a_t)}}_{\text{Q-Values for the state given a particular state}} = \underbrace{\boxed{\underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots}}_{\text{Expected discounted cumulative reward}}\underbrace{\boxed{s_t, a_t]}}_{\text{Given the state and action}}$$

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

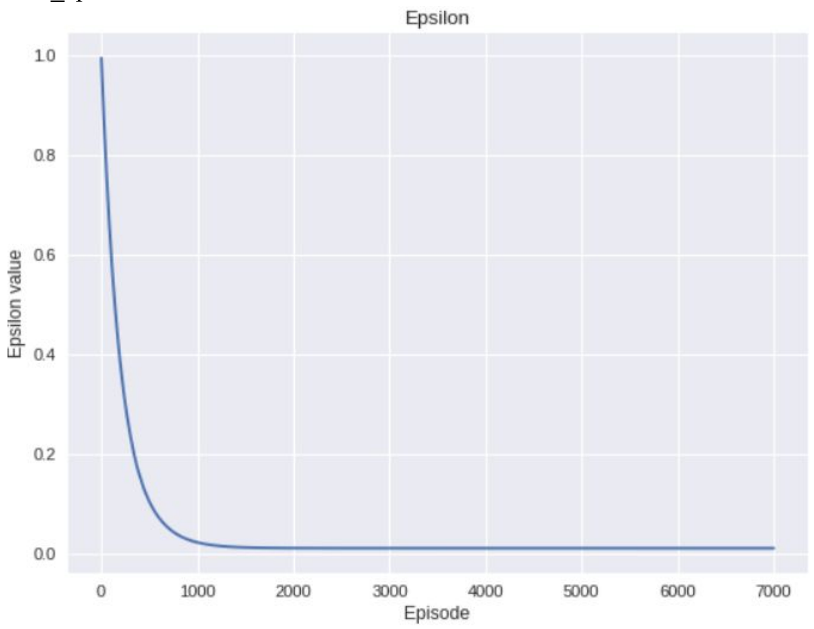
Using the above function, we get the values of Q for the Q- table.Initially, all the values in the Q-table are zeros. There is an iterative process of updating the values. As we start to explore the environment, the Q-function gives us better and better approximations by continuously updating the Q-values in the table.

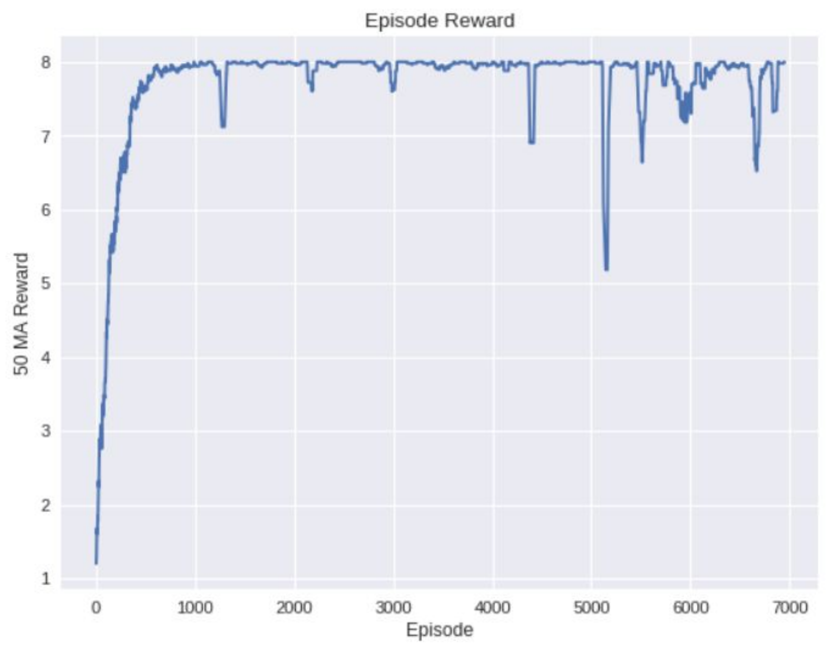
```
### START CODE HERE ### (≈ 4 line of code)
if st_next is None :
  t[act] = rew
else:
  t[act] = rew + 0.99*np.amax(q_vals_next[i])

### END CODE HERE ###
```

Using above implementations, the agent was able to find the path to Jerry in 1900 episodes with a last episode reward of 8 and mean of 5.4. The training time can be improved with different minimum and maximum epsilon values.s

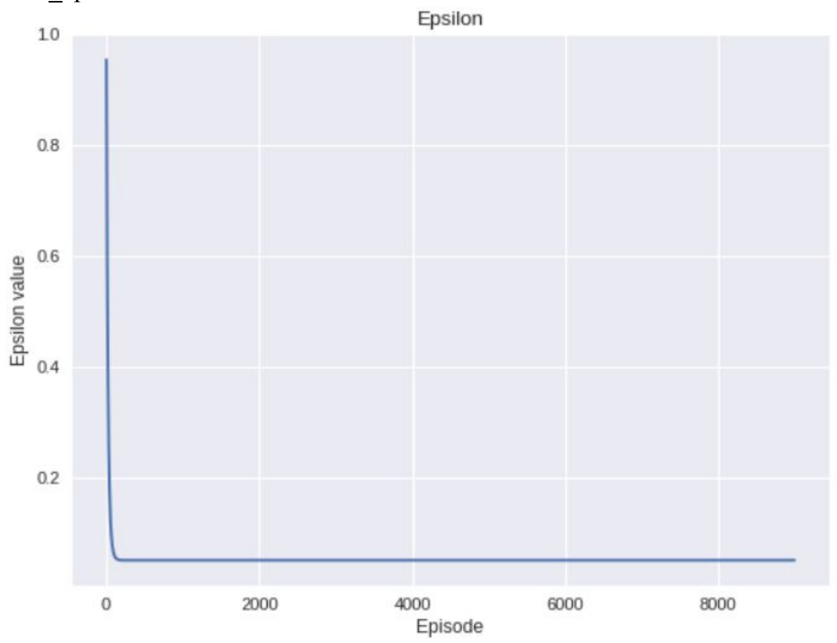
3.2 Tuning Hyperparameters

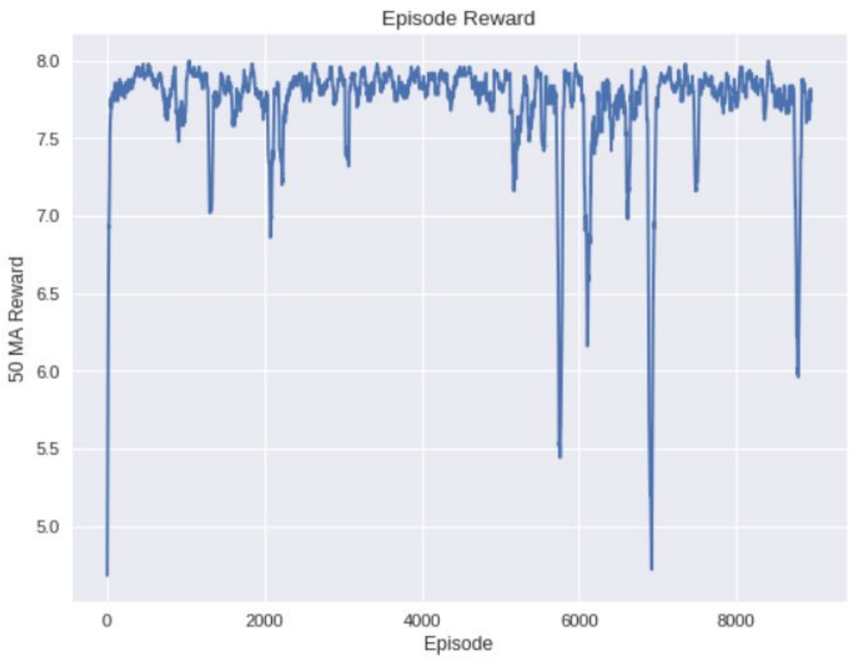
Case 1:
MAX_EPSILON = 1
MIN_EPSILON = 0.01
LAMBDA = 0.0005
num_episodes = 7000



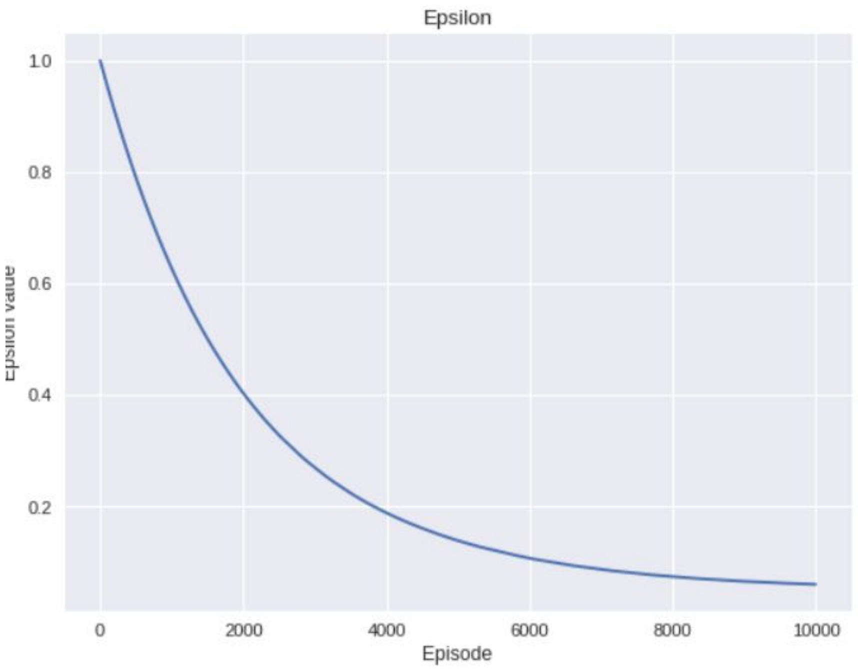


Case 2:
MAX_EPSILON = 1
MIN_EPSILON = 0.05
LAMBDA = 0.005
num_episodes = 9000

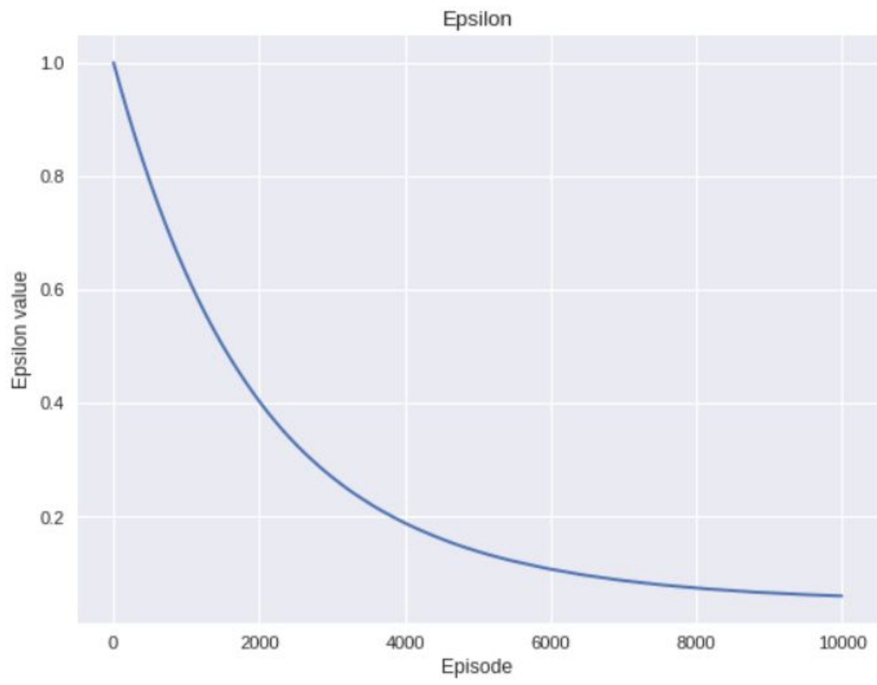




Case 3: Changing Gamma Values
gamma = 0.1



Gamma = 0.5



Observations:

- As we tend to increase the min epsilon the agent is performing exploitation more and tending to reach the goal in lower in number of episodes.
- Increased gamma value leads the agent to learn quickly with more exploration at first and then exploitation.

4 Writing Task

1. If the agent always choose to maximize the Q-value there will be more of exploitation of the environment than exploration. We'll also exploit nearest rewards rather than getting a huge reward by exploring all the environment states and rewards.
   We can force the agent to explore more using some techniques rather than using the maximum reward value to decide the move. Below are two examples of such techniques

   Counter Based Exploration (Basic Directed Exploration)
   In basic directed exploration, a history of learning process is maintained. This retained knowledge is used to guide the exploration process.
   In Counter-based exploration, the number of visits to each state s is maintained. Actions are evaluated using a combination of the exploitation value and an exploration term. One such combination is given by the following formula in which the exploration term is the quotient of the counter value for the current state and the expected counter value for the state that results from taking an action.

$$eval_c(a) = \alpha \cdot f(a) + \frac{c(s)}{E[c \mid s,a]}$$

$$E[c \mid s,a] = \sum_s P_{s \to s'}(a) \cdot c(s')$$

   Other possible formulations use the difference between the counter value for the current state and the expected counter value for the state that results from taking an action. The method can be more effective if we take into account when the state occured(decay) and what was the error in calculating the next step at that time.

Semi-Uniform Distributed Exploration (Undirected Exploration)
Undirected exploration does actions that are generated based on some random distribution and does not take into consideration the learning process itself.

In this method actions are generated with a probability distribution that is based on the utility estimates that are currently available to the agent. One such approach selects the action having the highest current utility estimate with some predefined probability Pbest. Each of the other actions is selected with probability 1 – Pbest regardless of its currently utility estimate.

$$P(a) = \begin{cases} P_{best} + \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, if } a \text{ maximizes } f(a) \\ \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, otherwise} \end{cases}$$

The Pbest parameter facilitates a continuous blend of exploration and exploitation that ranges from purely random exploration (Pbest = 0) to pure exploitation (Pbest = 1)

2. Q - Table

State	**UP**	**LEFT**	**DOWN**	**RIGHT**
0	**3.9006**	**3.9006**	**3.9403**	**3.9403**
1	**2.9403**	**2.9006**	**2.9701**	**2.9701**
2	**1.9403**	**1.9403**	**1.99**	**1.99**
3	**0.9701**	**0.9701**	**1**	**0.99**
4	**0**	**0**	**0**	**0**

Steps for S4:

Since the agent has already reached the goal S4(up,down,left,right) =(0,0,0,0)

Steps for S3:

(S3, up) = -1 + (0.99)(1.99) = 0.9701

(S3, down) = 1 + (0.99)(0) = 1

(S3, left)= -1 + (0.99)(1.99) =0.9701

(S3, right) = 0 + (0.99)(1) = 0.99

Steps for S2:

(S2, up) = -1 + (0.99)(2.97) =1.9403

(S2, down) = 1 + (0.99)(1) = 1.99

(S2, left) = -1 + (0.99)(2.97) =1.9403

(S2, right) = 1 + (0.99)(1) = 1.99

Steps for S1:

(S1, up) = 0 + (0.99)(2.97) =2.9403

(S1, down) = 1 + (0.99)(1.99) = 2.9701

(S1, left) = -1 + (0.99)(3.94) =2.9006

(S1, right) = 1 + (0.99)(1.99) = 2.9701

Steps for S0:

(S0, up) = 0 + (0.99)(3.94) = 3.9006

(S0, down) = 1 + (0.99)(2.97) = 3.9403

(S0, left) = 0+(0.99)(3.94) =3.9006

(S0, right) = 1+(0.99)(2.97) = 3.9403

5 Conclusion

Thus, we have implemented a DQN to model Tom and Jerry game in a grid based environment.