CSE 574: Introduction to Machine Learning (Fall 2018) -Project 4

Sneha Muppala
UBID: snehamup

Person#
50288710

Objective

To implement deep reinforcement learning algorithm- DQN (Deep Q network and teach the agent to navigate in the grid world environment to learn the shortest path to the goal.

1 Building a 3- layer neural network using keras library

We have implemented Neural network structure with activation function for the first and second hidden layer as relu and activation function for the output layer as linear which will return real values. Number of hidden layers is 128 for first two hidden layers and number of output hidden nodes are same as the size of the action space.

1.1 Implementation

```
def _createModel(self):
  # Creates a Sequential Keras model
  # This acts as the Deep Q-Network (DQN)

  model = Sequential()

  ### START CODE HERE ### (≈ 3 lines of code)
  # 'Dense' is the basic form of a neural network layer
  # Input Layer with activation function relu and Hidden Layer with 128 nodes
  model.add(Dense(128, input_dim=self.state_dim, activation='relu'))
  #Second Hidden layer with 128 nodes
  model.add(Dense(128, activation='relu'))
  #Output layer with activation linear.
  #action_size=4
  model.add(Dense(self.action_dim, activation='linear'))


  ### END CODE HERE ###

  opt = RMSprop(lr=0.00025)
  model.compile(loss='mse', optimizer=opt)

  return model
```

- States - 25 possible states (0, 0), (0, 1), (0, 2), … ,(4, 3), (4, 4)
- Actions - left, right, up, down

1. Neural network understands and predict based on the environment data. The input and output data is fed with fit() method to the model.
2. Model will train on those data to approximate the output based on the input. The training process makes the neural network to predict the reward value from the state given.
3. After training the model successfully, the model can predict the output from unseen input. When you call predict () function on the model, the model will predict the reward of the current based on the data we trained.

2 Implementing exponential-decay formula for epsilon

The 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.

Exponential-decay formula for epsilon:

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

where $\epsilon min,\epsilon max\in[0,1]$
λ - hyperparameter for epsilon
$|S|$ - total number of steps

2.1 Implementation

```
def observe(self, sample):  # in (s, a, r, s_) format
  """The agent observes an event.
  We pass a sample (state, action, reward, next state) to be stored in memory.
  We then increment the step count and adjust epsilon accordingly.
  """
  self.memory.add(sample)

  # slowly decrease Epsilon based on our eperience
  self.steps += 1

  ### START CODE HERE ### (≈ 1 line of code)

  self.epsilon=self.min_epsilon+(self.max_epsilon-self.min_epsilon)* math.exp((-self.lamb)*abs(self.steps))
  #ϵ=ϵmin+(ϵmax−ϵmin)∗e−λ|S|

  ### END CODE HERE ###

  self.epsilons.append(self.epsilon)
```

3 Implementing Q- Function

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

A method that trains the neural net with experiences in the memory is called replay(). First, we sample some experiences from the memory and call them batch. To make the agent perform well in long-term, we need to take into account not only the immediate rewards but also the future rewards we are going to get. In order to do this, we are going to have a ‘discount rate’ or ‘gamma’. This way the agent will learn to maximize the discounted future reward based on the given state.

3.1 Implementation

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

1. We extract information from each batch size. If st_next is none, we make our target reward else we predict the future discounted reward. The agent is made to approximately map the current state to future discounted reward. We train our neural network with state and target.

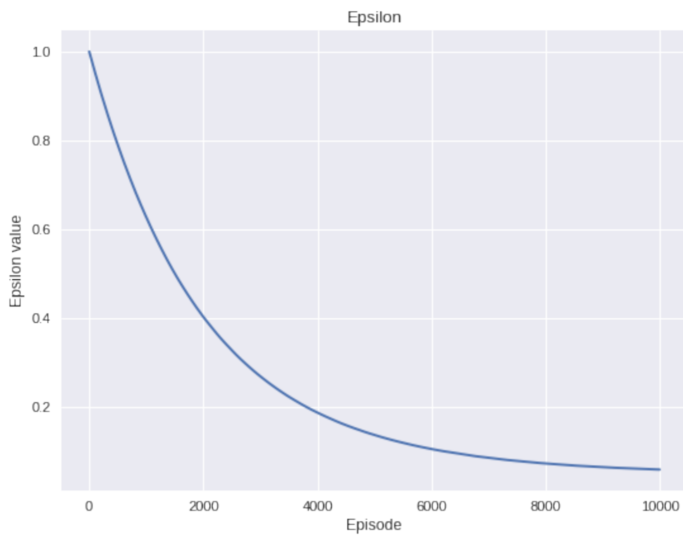
```
# Setting up training data
x = np.zeros((batch_size, self.state_dim))
y = np.zeros((batch_size, self.action_dim))
done=False
for i in range(batch_size):
    obs = batch[i]
    st = obs[0];
    act = obs[1];
    rew = obs[2];
    st_next = obs[3]
    t = q_vals[i]

  ### START CODE HERE ### (≈ 4 line of code)
  #if st_next is none make our target reward
    if st_next is None:
        t[act]=rew
    else:
      ## predict the future discounted reward
        t[act] = (rew + self.gamma *np.amax(q_vals_next[i]))


  ### END CODE HERE ###

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


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

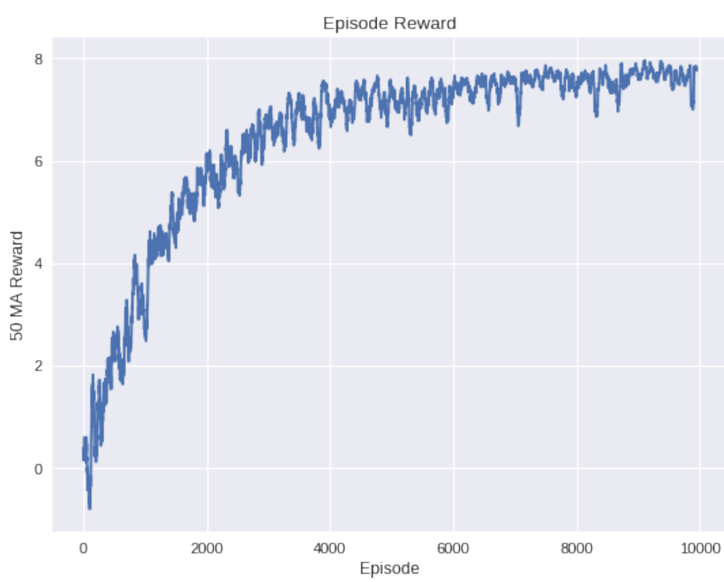
4 Hyper Parameters Tuning

These instructions apply to everyone, regardless of the formatter being used.

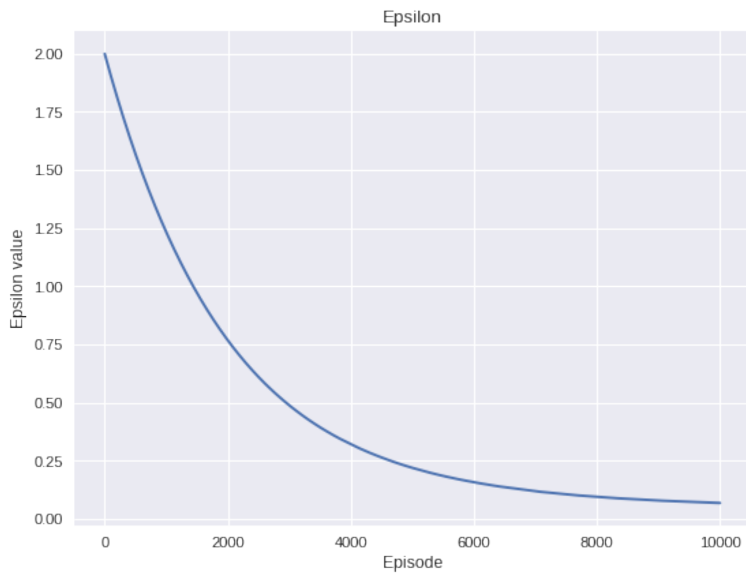
4.1 Epsilon

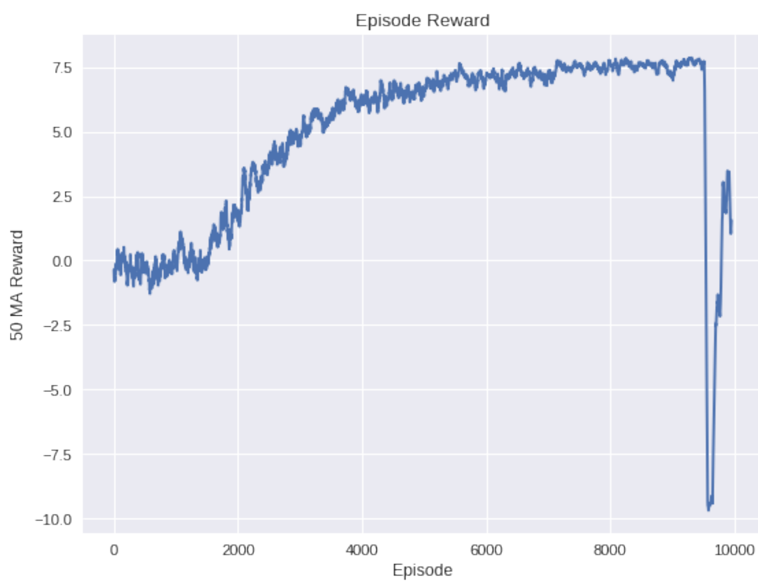
```
MAX_EPSILON = 1 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.00005      # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play
```



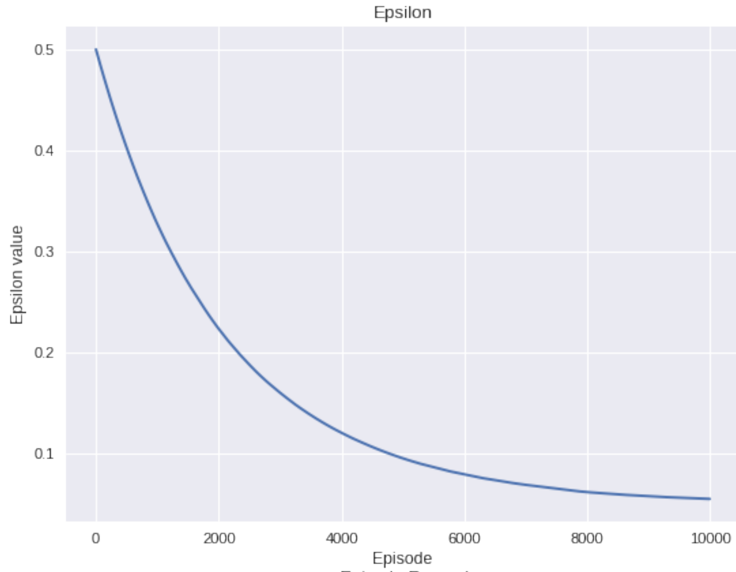


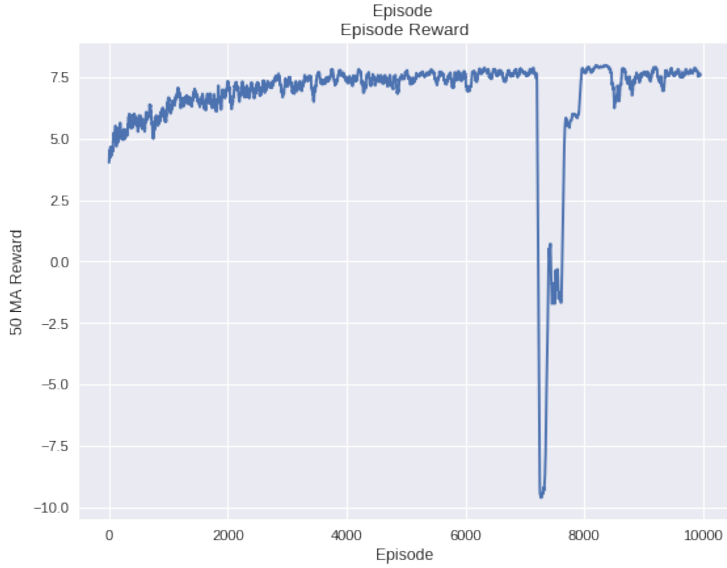
MAX_EPSILON = 2 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.00005 # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play



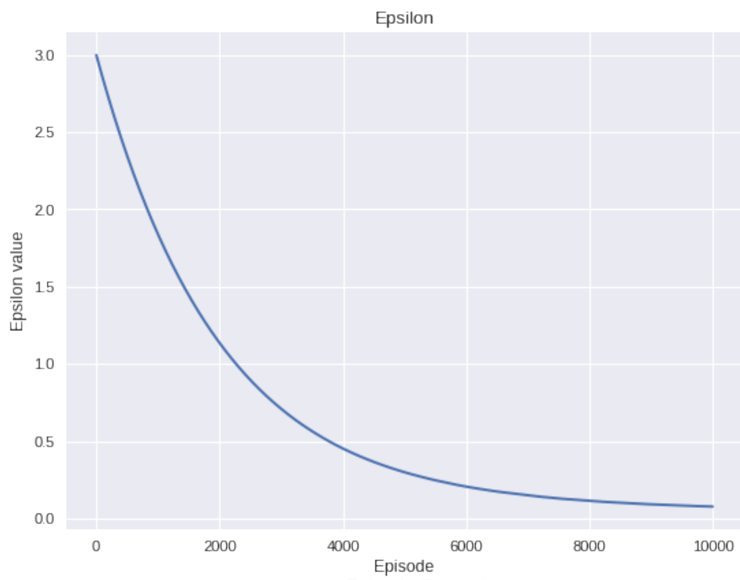


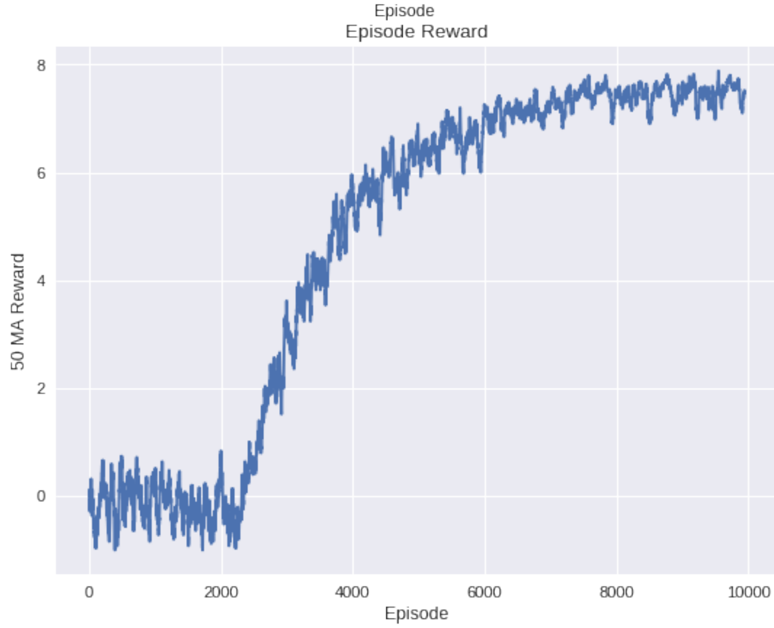
MAX_EPSILON = .5 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.00005 # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play





MAX_EPSILON = 3 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.00005 # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play





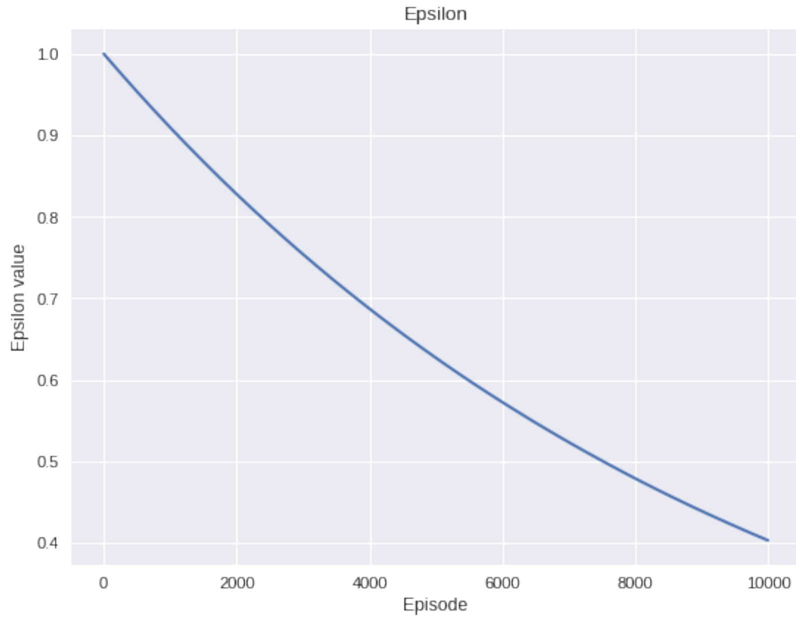
4.2 Lambda

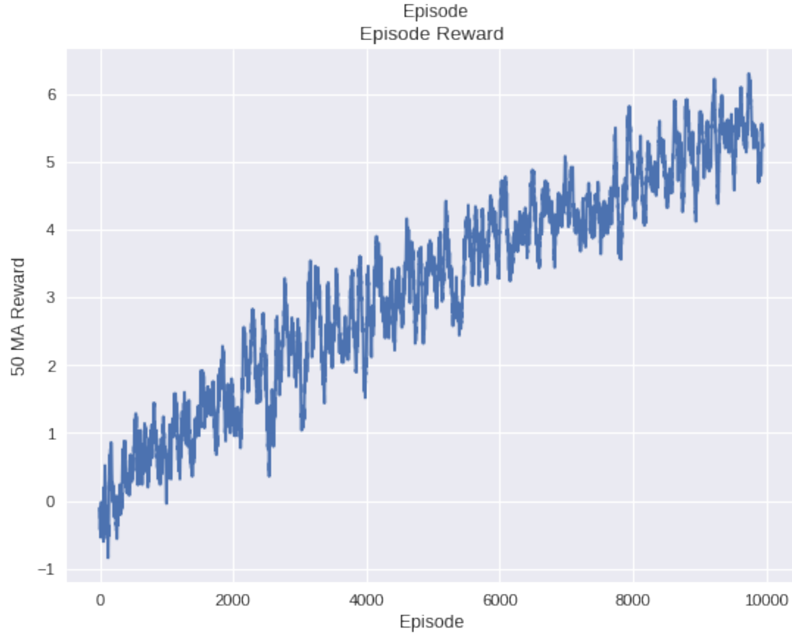
MAX_EPSILON = 1 # the rate in which an agent randomly decides its action

MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action

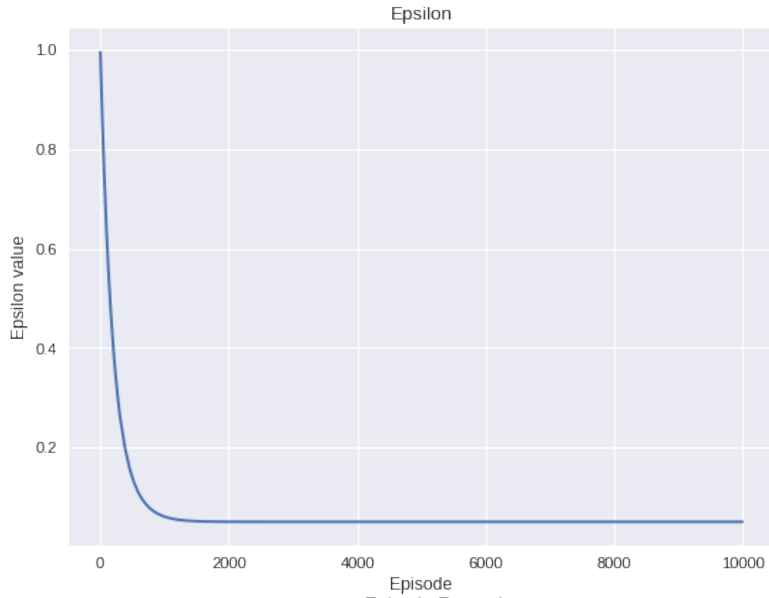
LAMBDA = 0.00001 # speed of decay for epsilon

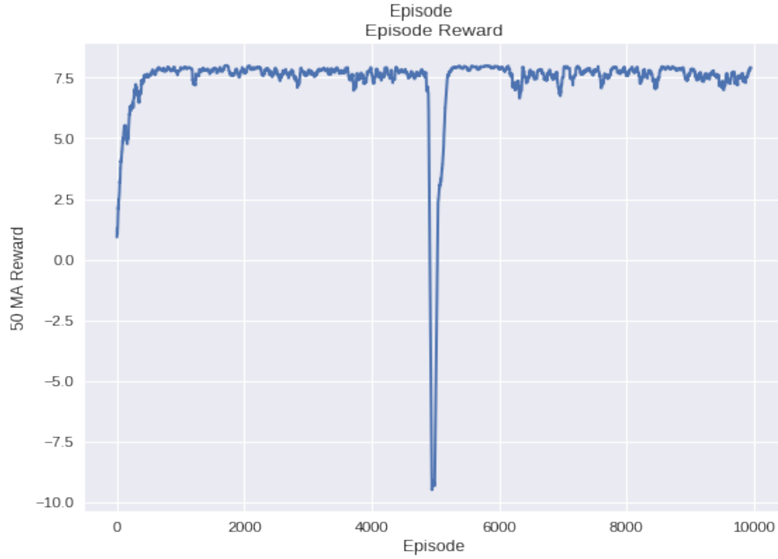
num_episodes = 10000 # number of games we want the agent to play





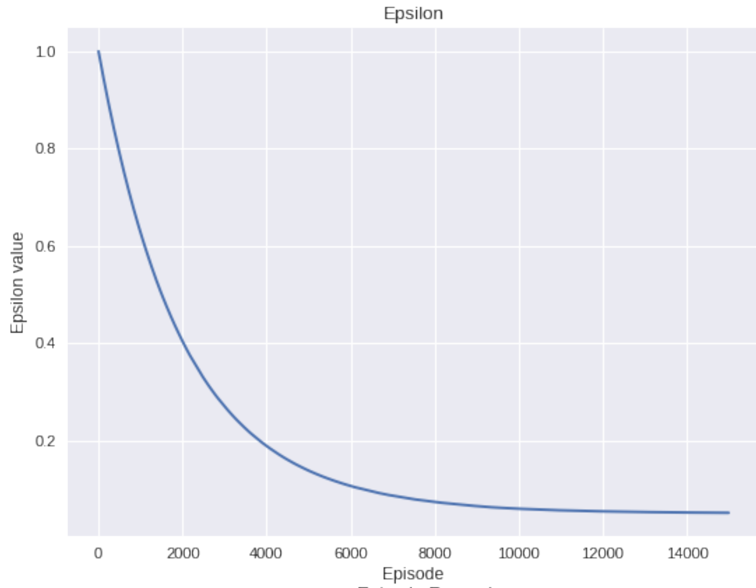

```
MAX_EPSILON = 1 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.0005      # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play
```

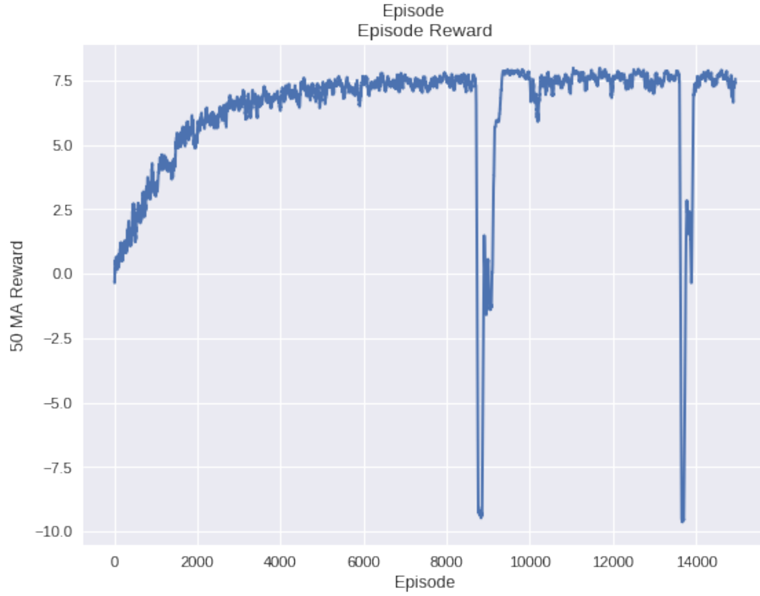




4.2 Number of episodes

```
MAX_EPSILON = 1 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.0005      # speed of decay for epsilon
num_episodes = 15000 # number of games we want the agent to play
```





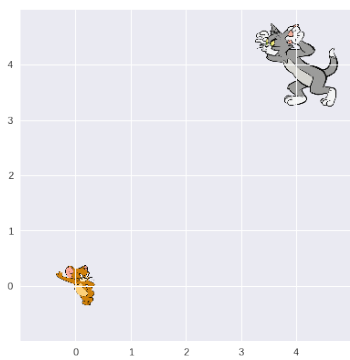
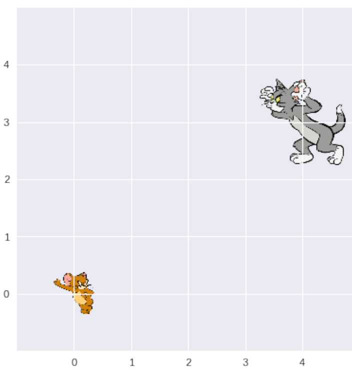
5 Observations

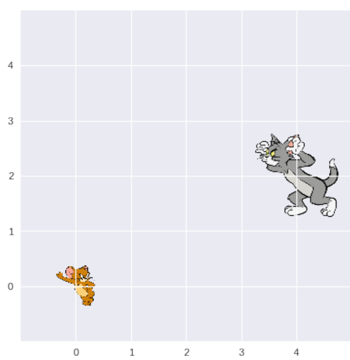
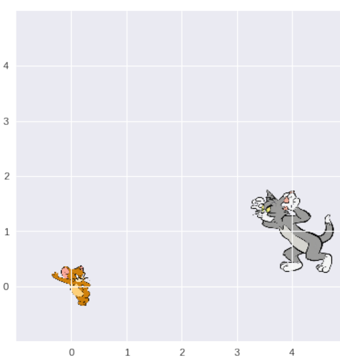
1. When we keep other hyperparameters constant, and change MAX_EPSILON=2, the rate in which an agent randomly decides its action. we can see that mean reward drops drastically at episodes 10000.
2. Similarly, when we change the MAX_EPSILON values to .5, 3 we observe the sudden drop of mean reward at certain episodes.
3. Epsilon and episode perform good, as we can in the graph.
4. When we tune the lambda value to 0.0005 and 0.00001, we can see the changes in the graph. Epsilon decrease in very few episodes.
5. When number of episodes are increased, we can notice the sudden drop in mean rewards of the model.
6. After hyper tuning the model, the model performs well with reward of 6 at episode 10000.
7. The rate in which an agent randomly decides its action is set to 1 and minimum

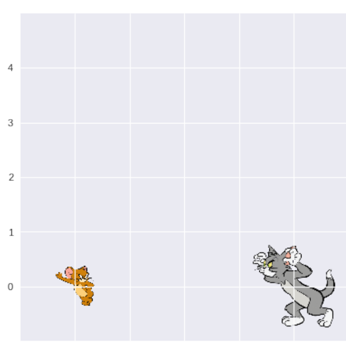
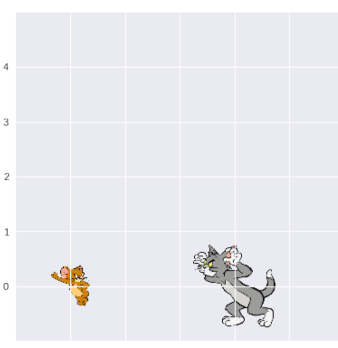
rate in which an agent randomly decided its action as 0.05 and speed of decay for epsilon as 0.00005 and number of episodes that is number of games we want the agent to play as 10000.

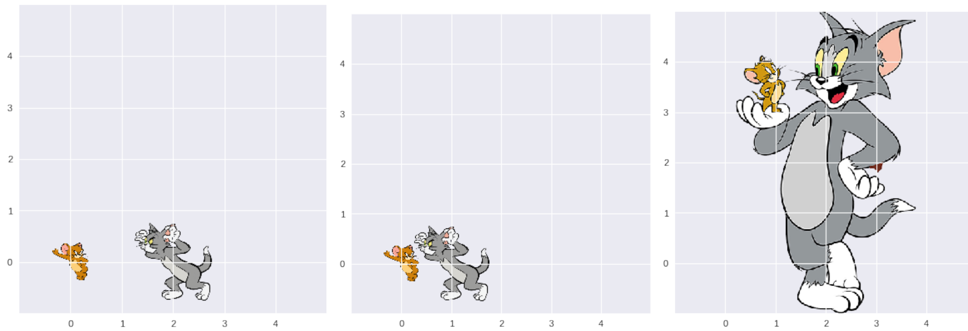
7 Results:

In the below results we can see the actions of tom to reach jerry in one of the optimal way.



```
MAX_EPSILON = 1 # the rate in which an agent randomly decides its action
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its action
LAMBDA = 0.00005      # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play
```

```
----------
Episode 9400
Time Elapsed: 758.38s
Epsilon 0.06281045948610747
Last Episode Reward: 7
Episode Reward Rolling Mean: 6.30792387915278
----------
Episode 9500
Time Elapsed: 765.93s
Epsilon 0.06227128505373458
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.322731624295288
----------
Episode 9600
Time Elapsed: 773.52s
Epsilon 0.061755391478611615
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.337648668561204
----------
Episode 9700
Time Elapsed: 781.10s
Epsilon 0.06125949738602101
Last Episode Reward: 6
Episode Reward Rolling Mean: 6.351317571086345
----------
Episode 9800
Time Elapsed: 788.63s
Epsilon 0.06078775812838438
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.3641892588392945
----------
Episode 9900
Time Elapsed: 796.14s
Epsilon 0.06033888453458969
Last Episode Reward: 6
Episode Reward Rolling Mean: 6.377614529129681
----------
```

6 Writing tasks

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

If agent always chooses the action that maximizes the Q- value, the agent might be stuck performing non optimal actions. If choosing the maximum Q- value doesn't have the optimal solution, exploring the non-maximum Q -value may lead to find better solution.

Exploitation: In any given time step, the agent has a choice, it can pick the action with the highest Q value for the state it is in is called exploitation. It maximizes the reward based on what agent already knows.

Exploration: Picking the action randomly is called exploration. It discovers better action selections and improves the knowledge about the environment.

Two ways to force agent to explore:

ε- greedy strategy: ε- greedy strategy is one of the algorithm trading off exploration and exploitation. This strategy selects the greedy action one that maximizes Q[s, a], but epsilon we can select between the range 0 <=epsilon<=1 or we can we the other formula like

Exponential-decay formula for epsilon:

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

where

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

λ - hyperparameter for epsilon

$|S|$ - total number of steps

We can make the decision at each step of the way. We can either choose to take a random action or choose the agent's trusted action. This algorithm more or less mimics greedy algorithm as it generally exploits the best available option, but every once in a while the Epsilon-Greedy algorithm explores the other available options.

Optimism in the face of uncertainty: Idea is to prioritize actions with uncertain outcomes. More the uncertainty and greater the expected value gives better outcome. Initializing the Q function to the values that gives space to exploration is one the method. In case the Q values are initialized to high values, the unexplored parts (uncertainty) will seem have better outcome, the exploration is done through greedy search.

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

We update the Q- table by calculating the values using given formula.

	U(up)	**D(down)**	**L(left)**	**R(right)**
S0	**3.90**	**3.94**	**3.9006**	**3.94**
S1	**2.94**	**2.97**	**2.9006**	**2.97**
S2	**1.94**	**1.99**	**1.94**	**1.99**
S3	**0.97**	**1**	**0.97**	**0.99**
S4	**0**	**0**	**0**	**0**

$Q(s_t, a_t) = r_t + \gamma * \max_a Q(s_{t+1}, a)$

1 when it moves closer to the goal , -1 when it moves away from the goal ,0 when it does not move at all

We start calculating Q values from last that is S4 by assigning 0 value, as the goal is at S4.

$Q(s_4, U)= Q(s_4, D)= Q(s_4, L)= Q(s_4, R)=0$

$Q(s_3, D)=1+0.99(\max(s4))=1+0.99*0=1$

$Q(s_2, R)=1+0.99(\max(s3))=1+0.99(1)=1.99$

$Q(s_1, D)=1+0.99(\max(s2))=1+0.99(1.99)=2.97$

$Q(s_0, R)=1+0.99(\max(s1))=1+0.99(2.97)=3.94$

$Q(s_3, U)=-1+0.99(\max(s2))=-1+0.99(1.99)=0.97$

$Q(s_2, U)=-1+0.99(\max(s1))=-1+0.99(2.97)=1.94$

$Q(s_1, U)=-1+0.99(\max(s1))=0+0.99(2.97)=2.94$

$Q(s_0, U)=0+0.99(\max(s0))=0+0.99(3.94)=3.9006$

$Q(s_2, D)=1+0.99(\max(s3))=1+0.99(1)=1.99$

$Q(s_0, D)=1+0.99(\max(s1))=1+0.99(2.97)=3.94$

$Q(s_3, L)=-1+0.99(\max(s2))=-1+0.99(1.99)=0.97$

$Q(s_2, L)=1+0.99(\max(s1))=1+0.99(2.97)=1.94$

$Q(s_1, L)$=-1+0.99(max(s0))=-1+0.99(s0)=2.9006

$Q(s_0, L)$=0+0.99(max(s0))=0+9.99(3.94)=3.9006

$Q(s_3, R)$=0+0.99(max(s3))=0+0.99(1)=0.99

$Q(s_1, R)$=1+0.99(max(s2))=1+0.99(1.99)=2.97

References

https://keon.io/deep-q-learning/

http://www.aispace.org/

https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe