TOM & JERRY GAME IMPLEMENTATION USING REINFORCEMENT LEARNING

INTRODUCTION

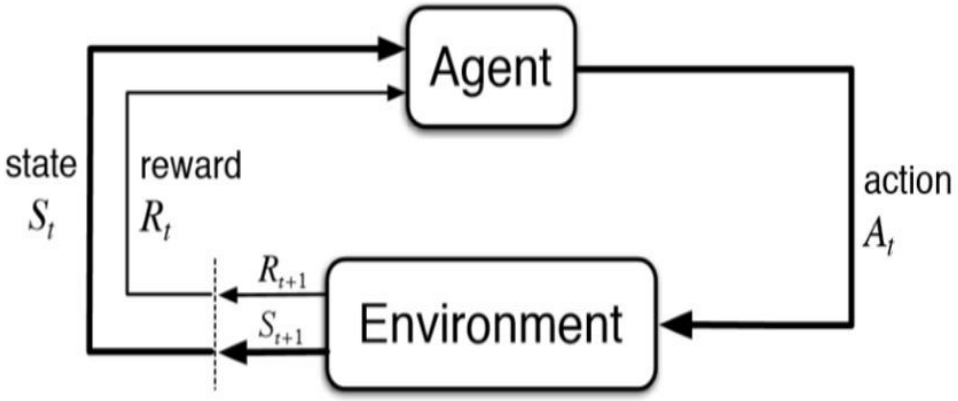
- The objective of the project is to learn and implement the concepts of reinforcement learning techniques using the simulation of a simple game.
- Since deep learning methods are also required, a Deep Q Network which combines deep learning and reinforcement learning concepts, is used to achieve the path to the objective with maximum gain

GAME WORLD

- The game world consists of the following components:
  A grid environment in which the agent and goal are present.
  An “agent” which solves the problem. Here, it is Tom the cat.
  A goal, to which a path needs to be found. Here, the goal is Jerry the mouse.
- The goal of the agent is to iteratively learn the best path to the goal.

REINFORCEMENT LEARNING

- It is a branch of machine learning where an “agent” in an “environment” learns to achieve a “goal state” by performing various actions and learning from the results of the results of these actions.
- A policy is present for the agent which is essentially “state-action” map which informs the agent which action to take for a given state.
- The agent learns this policy.



MARKOV DECISION PROCESS

- The Markov decision process states that provided we have the current state, the future state is independent of the previous states
- For Reinforcement learning, Markov decision process is modeled using a tuple containing 5 elements.
- Important terminologies:
- S: set of states which the agent can be in.
- A: set of actions which the agent can perform.
- P: is probability of ending up in a particular state, if an action is performed while in the current state.
- R: Reward for a given state and action.
- γ: A discount is introduced into the model. This value lies between 0 and 1. The purpose of this factor is to discount the long-term reward associated with a particular state and action.

DEEP Q LEARNING

- Deep Q learning introduces the concept of “experience replay”
- This allows us to retain the results of past learnings.
- A fixed number of results to be kept is decided upon.
- As the system trains more, newer results replace the older ones in memory.
- Random batches from these results are taken as learning sample data and then given to the network for training.
- This allows for more reliable training as the system experiences a variety of training samples.

EXPLORATION AND EXPLOITATION

- Exploitation refers to taking that action which promises to give the most reward (best decision from perspective of current state).
- This equates to maximum reward.
- Whereas Exploration refers to performing more actions and gathering information to make the overall best choice for the long-term.
- This is done by taking many random actions, observing the results and then learning from them.
- Here, the reinforcement learning model employs the Exponential Decay for Epsilon.
- This means that according to the rate specified, initially epsilon is at a high rate which results in more random actions.
- As the model learns, the value of epsilon gradually decreases, and random action is decreased. The model predicts from Q values.

DESCRIPTION OF THE WORKING OF THE MODEL

```
### START CODE HERE ### (≈ 3 lines of code)
model.add(Dense(output_dim=128, activation='relu', input_dim=self.state_dim))
model.add(Dense(output_dim=128, activation='relu'))
model.add(Dense(output_dim=self.action_dim, activation='linear'))
### END CODE HERE ###
```

```
  ### START CODE HERE ### (≈ 4 line of code)
  if s_ is None:
    # If the agent reaches terminal state the Q-Value is taken as reward
    t[act] = rew
  else:
    # if there is still a state to go to, we calculate reward with  discount
    t[act] = rew + self.gamma * np.amax(q_vals_next[i])
  ### END CODE HERE ###
```

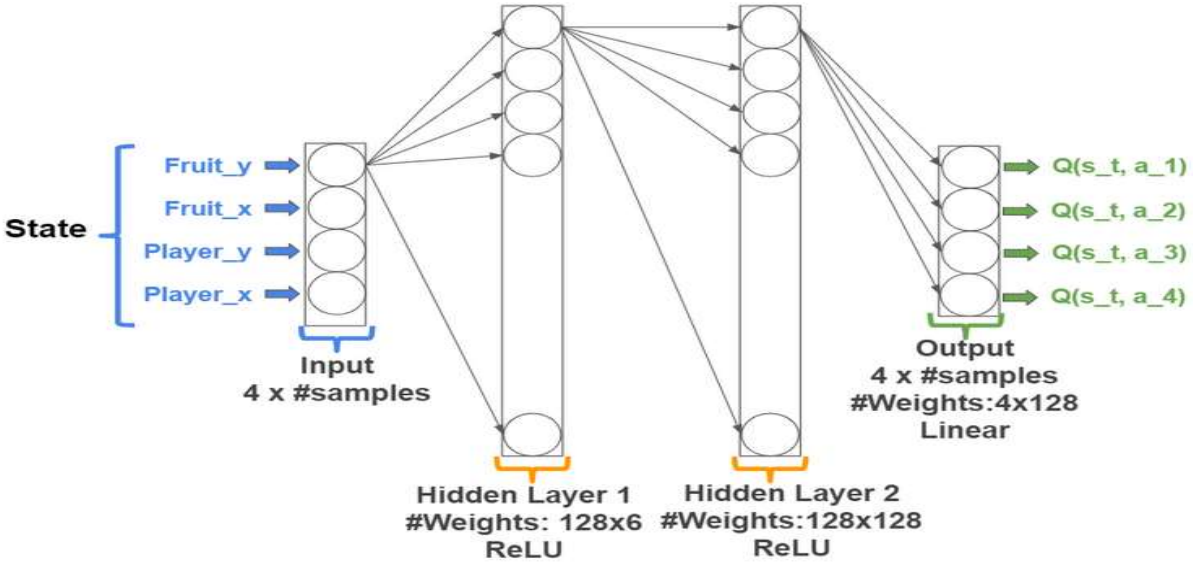
```
### START CODE HERE ### (≈ 1 line of code)
self.epsilon = self.min_epsilon + (self.max_epsilon - self.min_epsilon)\
* math.exp(-self.lamb * self.steps)
### END CODE HERE ###
```

Environment:

- A Class that consists of a grid world in which the cat and mouse are initialized
- A method to update the state based on the four directions is created.
- Here, fx and fy are the final coordinates of our agent whereas px and py are the coordinates of the player(agent).
- The distance to goal for previous state and distance to goal after moving to new state are calculated as old_d and new_d respectively.
- The difference is computed.
- A reward function is created which returns a reward of 1 if the player reaches the final/goal state.
- "_is_over" function checks whether the game has ended which occurs when time has ended.
- The "step" function keeps track of the reward collected for movement to a state. It also adds this reward to the already collected reward hence maintaining the total reward.
- The "reset" function resets the environment after each iteration

Random actions:

- This block executes random actions where the number of games/episodes can be specified. The player then executes random actions accordingly.
- Essentially, there should be a balance between exploration and exploitation where the agent should "explore" and perform random actions to see the results and learn rather than immediately seeking the actions with the maximum possible reward (immediate best actions)



Brain:

- This important block acts as the “reasoning” of the agent.
- Two important variables, the state_dim() which defines the state space and the action_dim() which defines the action space, are defined.
- The _createModel() method is the neural network part of the implementation.
- It creates a Deep Q Network which the model utilizes for training.
- The neural network consists of three layers. Two hidden layers (where weight updating takes place) and one output layer.
- The model fits the data in batches of size 64 using the train method.

Method:

- This code block is the Deep Q Network’s implementation of memory and “Experience Replay”.
- An initial capacity (buffer) is initialized to hold the experiences.
- As the buffer fills up and more new experiences are stored, the old ones are deleted.
- For the purpose of training, random samples from this are chosen.

Agent:

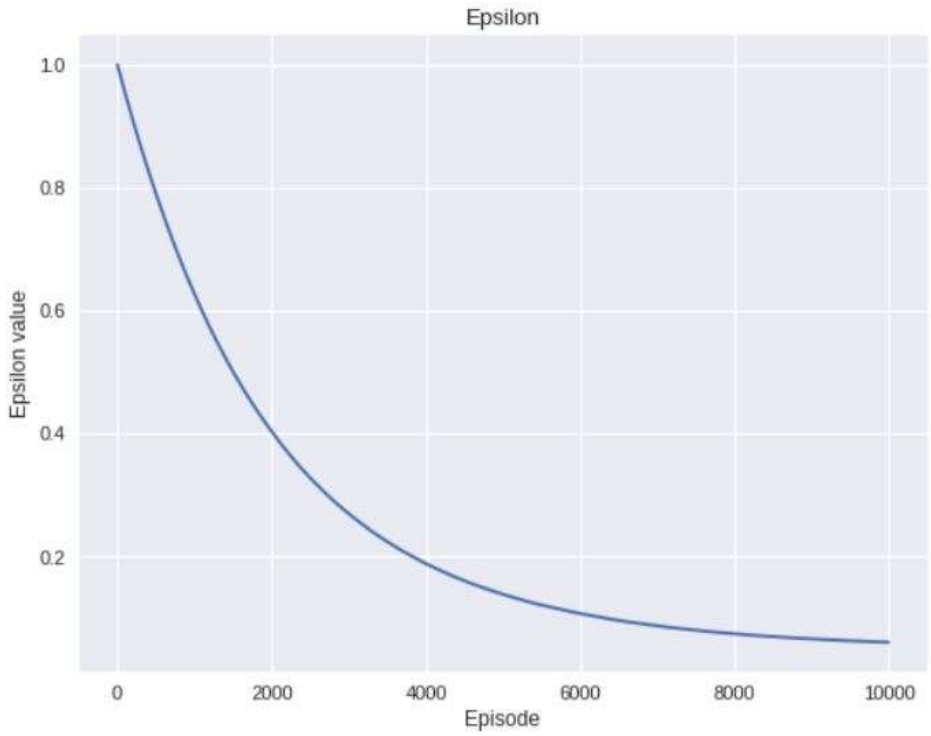
- This block of code defines the agent which moves in the grid world and e=reaches the target/goal.
- In the init method, we define various important parameters.

- The memory capacity for the model is defined and the value is set to 1000.
- The batch size which determines the size of each set of training samples.
- Epsilon defines the rate of exploration for the agent.
- This is nothing but the rate at which the agent will perform exploration by moving in random directions rather than the action which will maximize its reward.
- The value of Epsilon is set to a higher value in the beginning, hence resulting in a higher rate of random actions.
- Over time there is “decay”, i.e a decrease in exploration.
- The act method is an important method that defines the policy for the agent.
- Here, the decision about whether random action is better or whether training should occur through prediction of the q values which are defined earlier.
- The replay method assesses the sample observations from past observations as we are already storing all these experiences in memory.
- The experiences are stored in the form of state, action, reward, and next state.
- Training is then performed on these experiences.
- The Q values for the current and the next state are calculated by predicting the current and next state.
- The Q function is then calculated using by first checking for a terminal state. If it exists, then the q value is defined by the reward gathered.
- If there is a next state, then the reward along with the discount is calculated.

Main:

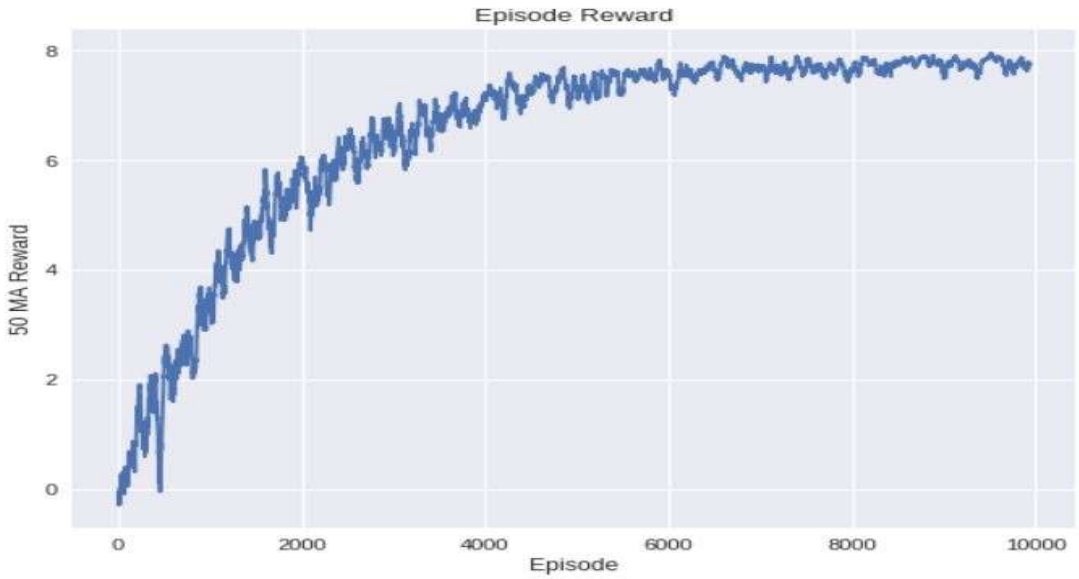
- Here, the final environment is set up and the game simulation is run.
- Various hyper-parameters such as the range for epsilon including the minimum and maximum values are set.
- Value of lambda, which is our trade-off value is set here.
- The number of episodes is also set here.
- This decides the number of times the agent trains. A value for this parameter must be chosen such that the agent is able to reach the goal state in minimum number of moves and find the path to goal quickly.

PERFORMANCE ANALYSIS OF THE MODEL.



Observation of exponential decay for Epsilon obtained from output

- Initially, Epsilon is set to nearly 1 (maximum value) and as the number of episodes for which the system trains increases, Epsilon steadily decreases
- This conforms to exploitation vs exploration trade-off, we can see that as training progresses the model makes fewer random actions and uses the q values to make predictions.
- In the graph below, we observe the rewards collected by agent for each episode of playing the game. In other words, the reward collected for each cycle of training.
- Initially, the agent does not collect a lot of reward due to the random nature of its actions.
- As the agent converges on the solution, the number of rewards becomes more as the agent chooses those actions which higher rewards only.



Reward vs number of episodes.

1.2: WRITING TASK

1.

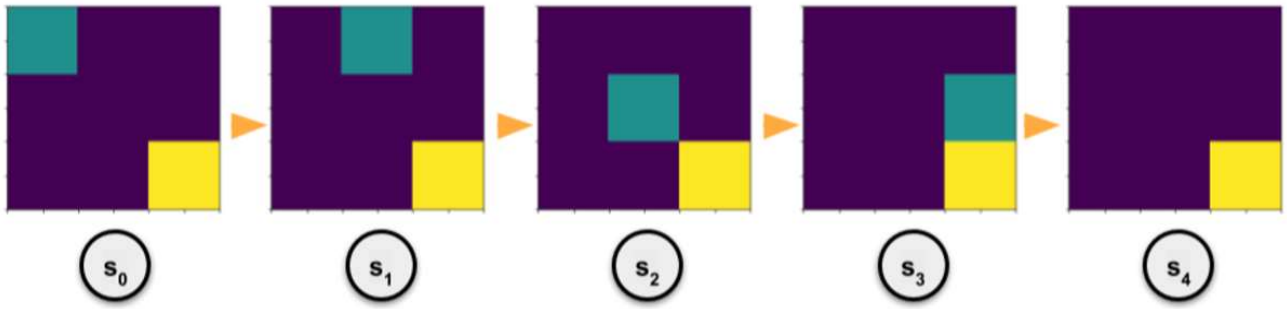
Always choosing greater Q value:

- When an agent always chooses an action that maximizes the Q value, since that is the priority, it might end up performing the actions which result in a path that has high Q value but is a sub-optimal path.
- Conversely, If the agent performs actions that result in an optimal path, that path may not produce the highest Q value.

2 ways to force the agent to learn more:

- To make the agent learn more, we must make it explore more. This is achieved using the Epsilon Greedy Algorithm.
- Here, we take a high value of Epsilon in the beginning so that agent is forced to explore more and hence observe and learn more.
- The other way to force an agent to learn more is to use the “Optimism in Uncertainty” rule. This means that greater the uncertainty associated with a particular action, i.e if it is not known whether action is good or not, the more important it is to explore that action.
- This is because that very action could yield the best result. Hence it becomes important to explore.

2:



For the given states mentioned in the question:

$Q(S_t, a_t) = r_t + \gamma * Q(S_{t+1}, a_{t+1})$

Gamma $\gamma = 0.99$

We fill the table according to the path mentioned, starting from the last state. (S_4)

This is the terminal state.

In S_4, all actions will have Q values of zero. this is because there is no further movement.

STATE	ACTIONS			
	UP	DOWN	LEFT	RIGHT
0	3.900	3.940	3.900	**3.940**
1	2.940	**2.970**	2.900	2.970
2	1.940	1.99	1.940	**1.99**
3	0.970	**1**	0.970	0.99
4	0	0	0	0

VALUES FOR PATH ARE SHOWN IN WORKINGS. SIMILARLY, Q-VALUES FOR OTHER ACTIONS CAN BE OBTAINED

BACK TRACING GIVEN OPTIMAL PATH

1) $Q(S_3, a_t = D) = 1 + 0.99 \times \max(S_4, \text{action})$

$= 1 + 0.99 \times 0$

$Q(S_3, a_t = D) = \boxed{1}$

2) $Q(S_3, a_t = R) = 0 + 0.99 * \max(S_3, \text{action})$

$= 0 + 0.99\,(1)$

$Q(S_3, a_t = R) = \boxed{0.99}$

3) $Q(S_2, a_t = R) = 1 + 0.99 * \max(S_3, \text{action})$

$= 1 + 0.99(1)$

$Q(S_2, a_t = R) = \boxed{1.99}$

4) $Q(S_1, a_t = U) = 0 + 0.99 * \max(S_1, \text{action})$

$= 0 + 0.99\,(2.970)$

$Q(S_1, a_t = U) = \boxed{2.9403}$

5) $Q(S_0, a_t = R) = 1 + 0.99 * \max(S_1, \text{action})$

$= 1 + 2.940$

$Q(S_0, a_t = R) = \boxed{3.940}$

s