CSE474: Project4
Reinforcement Learning

Kevin Yiu-Wah Cheung
UB Number: 50148100
Email: kcheung8@buffalo.edu

December 2018

Contents

1 Introduction

The project combines reinforcement learning and deep learning. Your 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 deterministic. 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.

There are two main parts of the project:

1. Coding part - build neural network in Keras for the deep learning part and write code in Python for the reinforcement learning part.
2. Writing part - answer questions.

This project does not require GPU and is supported by Windows/Linux/MacOS, though you are welcome to use Google Colab (Jupyter notebook environment that requires no setup and allows to use GPU for free). The grading will be based on the mean reward.

1.1 Coding Tasks

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

1.2 Questions

1. 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.
2. Calculate Q-value for the given states and provide all the calculation steps. [20 points] 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.

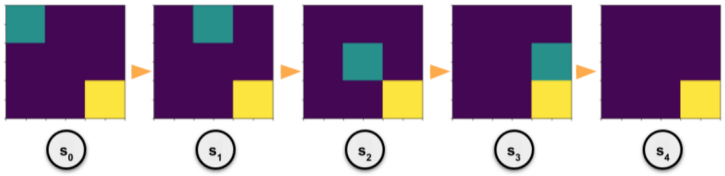


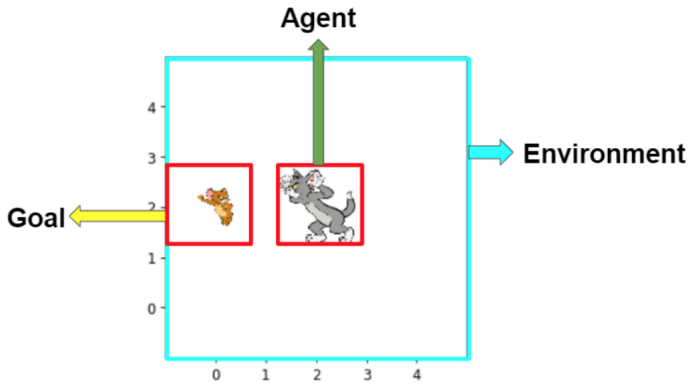
Figure 1: One of the possible optimal actions sequence

In Figure 1, 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). Your task is to fill out the Q-Table for the above states, where $\gamma = 0.99$. Hint: Start calculating Q-function from the last state.

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

1.3 Environment Description

The environment is designed as a grid-world 5x5:

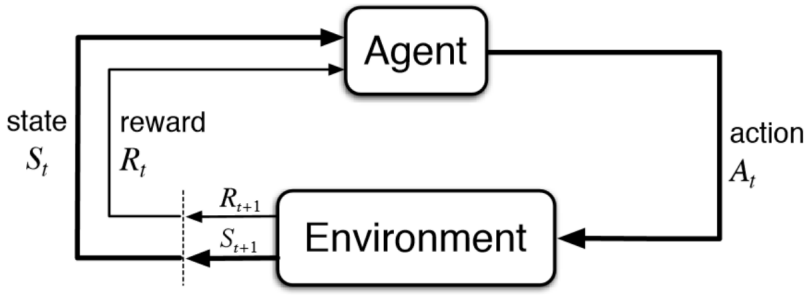


- States - 25 possible states (0, 0), (0, 1), (0, 2), ... ,(4, 3), (4, 4)
- Actions - left, right, up, down
- The goal (yellow square) and the agent (green square) are dynamically changing the initial position on every reset.

1.4 Reinforcement Learning

Reinforcement learning is a direction in Machine Learning where an agent learn how to behave in a environment by performing actions and seeing the results. In reinforecement learning, an agent learns from trial-and-error feedback rewards from its environment, and results in a policy that maps states to actions to maximize the long-term total reward as a delayed supervision signal. Reinforcement learning combining with the neural networks has made great progress recently, including playing Atari games and beating world champions at the game of Go. It is also widely used in robotics.

1.5 Markov Decision Process



Basic reinforcement is modeled as a Markov decision process a 5-tuple (S, A, P_a, R_a, γ), where:

- S is a finite set of states

- A is a finite set of actions
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time t + 1
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s' due to action a
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards

In application, we typically have an environment, which handles state and reward, and an agent, which decides which action to take given a particular state. An environment starts with some initial state s_0, which is passed to the agent. The agent passes an action a_0, based on the state s0, back to the environment. The environment reacts to the action, then passes the next state, s1, along with the resulting reward for taking the action, r_0, back to the agent. This process continues until we reach a terminal state, indicating the end of an episode, at which point the process may start over again.

In reinforcement learning, we attempt to choose actions which yields the best results given some predefined criteria. This involves learning a mapping from state to action which attempts to maximize discounted accumulative reward. In deep reinforcement learning, a neural network is used approximate this mapping. Currently, there are two frequently used approaches to doing this: off-policy learning and on-policy learning.

1.6 Discounting factor (γ)

The discounting factor ($\gamma \in [0, 1]$) penalize the rewards in the future. Reward at time k worth only γ^{k-1} Motivation:

- The future rewards may have higher uncertainty (stock market)
- The future rewards do not provide immediate benefits (as human beings, we might prefer to have fun today rather than 5 years later)
- Discounting provides mathematical convenience
- It is sometimes possible to use undiscounted Markov reward processes (e.g. $\gamma = 1$), if all sequences terminate

Main Definitions

Deterministic policy (π) is a function that maps states to actions:

$$\pi(s) = a \tag{1}$$

Return (G_t):

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

Value function ($V_\pi(s)$):

$$V_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \tag{3}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right], \text{for all } s \in S \tag{4}$$

Action-value function ($Q_\pi(s, a)$) - how good to take an action at a particular state:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{5}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \tag{6}$$

Objective is to find such a policy, that returns max Q-value.

$$\pi^*(s) = argmax_a Q(s, a) \tag{7}$$

1.7 Experience Replay

Experience replay will help us to handle three main things:

- Avoid forgetting previous experiences
- Reduce correlations between experiences
- Increases learning speed with mini-batches

The main idea behind the experience replay is that by storing an agents experiences, and then randomly drawing batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences we draw random, we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. Each of these experiences are stored as a tuple of `(state,action,reward,next state)`. The Experience Replay buffer stores a fixed number of recent memories (memory capacity), and as new ones come in, old ones are removed. When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them.

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**

---

2 3-Layers Neural Network

```
#Input Layer
model.add(Dense(4))
model.add(Activation('linear'))

#First Hidden Layer
model.add(Dense(128))
model.add(Activation('relu'))

#Second Hidden Layer
model.add(Dense(128))
model.add(Activation('relu'))

#Output Layer
model.add(Dense(4))
model.add(Activation('linear'))
```

- The model's structure is: LINEAR → RELU → LINEAR → RELU → LINEAR
- Activation function for the first and second hidden layers is 'relu'
- Activation function for the output layer is 'linear' (that will return real values)
- Input dimensions for the first hidden layer equals to the size of your observation space (state_size)
- Number of hidden nodes is 128 for both hidden layers
- Number of the output should be the same as the size of the action space (action_size)

In our 5x5 grids, it is relatively easy to calculate Q table, which is including each state. But in real world application, the grids could be infinitely large and it is not feasible to calculate the all Q values. Hence, neural network takes the memory in as a vector and predict those states, which cannot be explicitly calculated.

3 Epsilon Implementation

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

Exponential-decay formula for epsilon: $\epsilon = \epsilon_{min} + (\epsilon_{max} + \epsilon_{min}) * e^{-\lambda|S|}$
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.

4 Q-Function Implementation

```
### START CODE HERE ### (≈ 4 line of code)
'''Draft'''
if s_ is None:
  t[act] = rew
else:
  t[act] = rew + self.gamma * np.amax(q_vals_next[i])
### END CODE HERE ###
```

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

The Q-Function will return values for each action in that particular states. The agent will take action based on the q-value.

5 Question 1

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 there is no obstacle in the maze, the agent will keep going at the same path as previous successful episode.
If there are some obstacles between the starting point and ending point, the agent will only focus on current Q values, instead of the overall Q values. In some scenarios, it will lead to not-optimal solution.
This leads to an exploitation over exploration. In reinforcement learning, we need to balance exploitation vs exploration. Though this method will give us the best solution and performance all the time, it won't give the agent the ability to "adapt to change".

Suggestions:

- Set a threshold for epsilon so that it can only reach that minimum epsilon value. Epsilon is a value to determine exploration or exploitation.
- Introduce a new "competition". Other words, introduce another agent or simply introduce an obstacle. This way, it'll force the agent to perform new actions, based on what rewards it will receive in the new environment.

6 Question 2

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

Q_Table				
	Actions			
State	**Up**	**Down**	**Left**	**Right**
0	3.90099501	3.940399	3.9009501	3.940399
1	2.940399	2.9701	2.90099501	2.9701
2	1.940399	1.99	1.940399	1.99
3	0.9701	1	0.9701	0.99
4	0	0	0	0

$Q(S_3, down) = 1 + 0 = 1$

$Q(S_3, up) = -1 + 0.99 * \max Q(S_{13}, a)$

$\max Q(S_{13}, a) = Q(S_{13}, down) = 1 + 0.99 * \max Q(S_{23}, a)$

$= 1 + 0.99 * 1 = 1.99$

Hence, $Q(S_3, up) = -1 + 0.99 * 1.99 = 0.9701$

$Q(S_3, right) = 0 + 0.99 * \max Q(S_{23}, a)$

$\max Q(S_{23}, a) = Q(S_{23}, down) = 1$

Hence, $Q(S_3, right) = 0 + 0.99 = 0.99$

$Q(S_3, left) = -1 + 0.99 * \max Q(S_{32}, a)$

$\max Q(S_{32}, a) = Q(S_{32}, down) = Q(S_{32}, right)$

$Q(S_{22}, down) = 1 + 0.99 * \max Q(S_{32}, a)$

$Q(S_{22}, right) = 1 + 0.99 * \max Q(S_{23}, a) = 1 + 0.99 * 1 = 1.99$

$\max Q(S_{22}, a) = Q(S_{23}, down) = 1$

Hence, $Q(S_3, left) = -1 + 0.99 * 1.99 = 0.9701$

$Q(S_2, right) = 1 + 0.99 * \max Q(S_{23}, a)$

$\max Q(S_{23}, a) = Q(S_{23}, down) = 1$

Hence, $Q(S_2, right) = 1 + 0.99(1) = 1.99$

$Q(S_2, down) = 1 + 0.99 * \max Q(S_{32}, a)$

$\max Q(S_{31}, a) = Q(S_{32}, right) = 1 + 0.99(0) = 1$

Hence $Q(S_2, down) = 1 + 0.99 * (1) = 1.99$

$Q(S_2, up) = -1 + 0.99 * \max Q(S_{12}, a)$

$\max Q(S_{12}, a) = Q(S_{12}, down)$ or $Q(S_{12}, right)$

$Q(S_{12}, down) = 1 + 0.99 * \max Q(S_{22}, a)$

$\max Q(S_{22}, a) = Q(S_{22}, down)$ or $Q(S_{22}, right) = 1.99$.

$Q(S_{12}, down) = 1 + 0.99 * (1.99) = 2.9701$

Hence, $Q(S_2, up) = -1 + 0.99 * 2.9701 = 1.940399$

$Q(S_2, \text{left}) = -1 + 0.99 * \max Q(S_{21}, a)$
$\max Q(S_{21}, a) = Q(S_{21}, \text{down})$ or $Q(S_{21}, \text{right})$
$Q(S_{21}, \text{right}) = 1 + 0.99 * \max Q(S_{22}, a)$
$\max Q(S_{22}, a) = Q(S_{22}, \text{right})$ or $Q(S_{22}, \text{down})$
since $Q(S_{22}, \text{right}) = Q(S_{22}, \text{down}) = 1.99$
$Q(S_{21}, \text{right}) = 1 + 0.99 * 1.99 = 2.9701$
Hence $Q(S_2, \text{left}) = -1 + 0.99 * 2.9701 = 1.940399$

$Q(S_1, \text{down}) = 1 + 0.99 * \max Q(S_{22}, a)$
$\max Q(S_{22}, a) = Q(S_{22}, \text{right}) = Q(S_{22}, \text{down}) = 1.99$
Hence, $Q(S_1, \text{down}) = 1 + 0.99 * 1.99 = 2.9701$

$Q(S_1, \text{up}) = 0.99 * \max Q(S_1, a)$
$\max Q(S_1, a) = Q(S_{12}, \text{down})$ or $Q(S_{12}, \text{right}) = 2.9701$
$Q(S_1, \text{up}) = 0.99 * 2.9701 = 2.940399$

$Q(S_1, \text{right}) = 1 + 0.99 * \max Q(S_{13}, a)$
$\max Q(S_{13}, a) = Q(S_{13}, \text{down}) = 1.99$
Hence, $Q(S_1, \text{right}) = 1 + 0.99 * 1.99$
$= 2.9701$

$Q(S_1, \text{left}) = -1 + 0.99 * \max Q(S_{11}, a)$
$\max Q(S_{11}, a) = Q(S_{11}, \text{down})$ or $Q(S_{11}, \text{right})$
$Q(S_{11}, \text{right}) = 1 + 0.99 * \max Q(S_{12}, a) = 1 + 0.99 * 2.9701 = 3.940399$
$Q(S_1, \text{left}) = -1 + 0.99 * 3.940399 = 2.900099501$

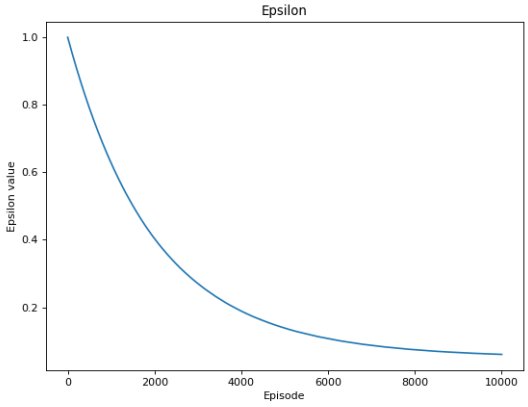
$Q(S_0, \text{right}) = 1 + 0.99 * \max Q(S_{12}, a)$
$\max Q(S_{12}, a) = 2.9701$
Hence, $Q(S_0, \text{right}) = 1 + 0.99 * 2.9701$
$= 3.940399$

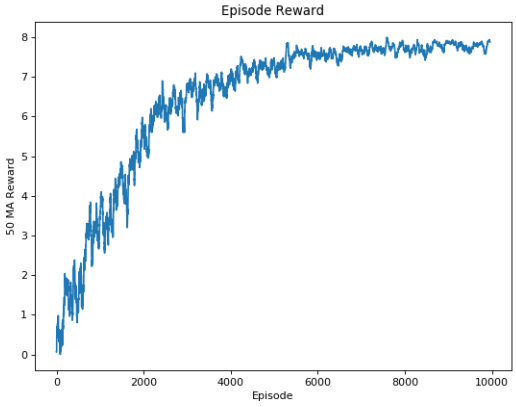
$Q(S_0, \text{down}) = 1 + 0.99 * \max Q(S_{21}, a)$
$\max Q(S_{21}, a) = Q(S_{21}, \text{down})$ or $Q(S_{21}, \text{right}) = 2.9701$
Hence, $Q(S_0, \text{down}) = 1 + 0.99 * 2.9701$
$= 3.940399$

$Q(S_1, \text{up}) = 0.99 * \max Q(S_1, a)$
$= 0.99 * 3.940399$
$= 3.900099501$

$Q(S_1, \text{left}) = 0.99 * \max Q(S_1, a)$
$= 0.99 * 3.940399$
$= 3.900099501$

7 Summary





The above two graphs show the performance of the reinforcement learning. After 10000 episodes, reward is 8 and episode reward rolling mean is 6.39. For improving the reinforcement learning, we could increase number of hidden layer and hidden nodes.