CS6700: Reinforcement Learning

Gautham Govind A, EE19B022

#Tutorial 5 - Options Intro

Please complete this tutorial to get an overview of options and an implementation of SMDP Q-Learning and Intra-Option Q-Learning.

References:

Recent Advances in Hierarchical Reinforcement Learning is a strong recommendation for topics in HRL that was covered in class. Watch Prof. Ravi's lectures on moodle or nptel for further understanding the core concepts. Contact the TAs for further resources if needed.

For compatibility with the given template, we use an older version of gym environment:

```
pip install gym==0.15.3
```

```
Looking in indexes: https://pypi.org/simple, https://us-
python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gym==0.15.3 in
/usr/local/lib/python3.9/dist-packages (0.15.3)
Requirement already satisfied: pyglet<=1.3.2,>=1.2.0 in
/usr/local/lib/python3.9/dist-packages (from gym==0.15.3) (1.3.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.9/dist-
packages (from gym==0.15.3) (1.10.1)
Requirement already satisfied: numpy>=1.10.4 in
/usr/local/lib/python3.9/dist-packages (from gym==0.15.3) (1.22.4)
Requirement already satisfied: cloudpickle~=1.2.0 in
/usr/local/lib/python3.9/dist-packages (from gym==0.15.3) (1.2.2)
Requirement already satisfied: six in /usr/local/lib/python3.9/dist-
packages (from gym==0.15.3) (1.16.0)
Requirement already satisfied: future in
/usr/local/lib/python3.9/dist-packages (from pyglet<=1.3.2,>=1.2.0-
>gym==0.15.3) (0.18.3)
```

```
'''
A bunch of imports, you don't have to worry about these
'''

import numpy as np
import random
import gym
#from gym.wrappers import Monitor
import glob
import io
import matplotlib.pyplot as plt
from IPython.display import HTML
```

```
'''
The environment used here is extremely similar to the openai gym ones.
At first glance it might look slightly different.
The usual commands we use for our experiments are added to this cell
to aid you
work using this environment.
'''

#Setting up the environment
from gym.envs.toy_text.cliffwalking import CliffWalkingEnv
env = CliffWalkingEnv()

env.reset()

#Current State
print(env.s)

# 4x12 grid = 48 states
print ("Number of states:", env.nS)

# Primitive Actions
action = ["up", "right", "down", "left"]
#correspond to [0,1,2,3] that's actually passed to the environment

# either go left, up, down or right
print ("Number of actions that an agent can take:", env.nA)

# Example Transitions
rnd_action = random.randint(0, 3)
print ("Action taken:", action[rnd_action])
next_state, reward, is_terminal, t_prob = env.step(rnd_action)
print ("Transition probability:", t_prob)
print ("Next state:", next_state)
print ("Reward recieved:", reward)
print ("Terminal state:", is_terminal)
env.render()
```

```
36
Number of states: 48
Number of actions that an agent can take: 4
Action taken: up
Transition probability: {'prob': 1.0}
Next state: 24
Reward recieved: -1
Terminal state: False
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
```

Options

We custom define very simple options here. They might not be the logical options for this settings deliberately chosen to visualise the Q Table better.

We slightly modify the termination condition for Close option. This is because if the agent attempts to do this option from the start state (state number 36), int(state/12) = 3, and therefore the termination condition is never reached, forcing the agent to enter an endelss loop . We circumvent this issue by modifying the condition from (int(state/12) == 2) to (int(state/12) >= 2), thereby breaking loop at the starting state as well.

```
# We are defining two more options here
# Option 1 ["Away"] - > Away from Cliff (ie keep going up)
# Option 2 ["Close"] - > Close to Cliff (ie keep going down)

def Away(env,state):

    optdone = False
    optact = 0

    if (int(state/12) == 0):
        optdone = True

    return [optact,optdone]


def Close(env,state):

    optdone = False
    optact = 2

    # We slightly mpodify the termination condition for Close option
to break loop at start state
    if (int(state/12) >= 2):
        optdone = True

    return [optact,optdone]


print('''
Now the new action space will contain
Primitive Actions: ["up", "right", "down", "left"]
Options: ["Away","Close"]
Total Actions :["up", "right", "down", "left", "Away", "Close"]
Corresponding to [0,1,2,3,4,5]
''')
```

```
Now the new action space will contain
Primitive Actions: ["up", "right", "down", "left"]
```

```
Options: ["Away","Close"]
Total Actions :["up", "right", "down", "left", "Away", "Close"]
Corresponding to [0,1,2,3,4,5]
```

Task 1

Complete the code cell below

```
#Q-Table: (States x Actions) === (env.ns(48) x total actions(6))


# Q-Tables for SMDP and intra-option q-learning

q_values_SMDP = np.zeros((48,6))
q_values_IO = np.zeros((48,6))

# Update_Frequency tables for SMDP and intra-option q-learning

update_freq_SMDP = np.zeros((48,6))
update_freq_IO = np.zeros((48,6))

# Epsilon-greedy action selection function
def egreedy_policy(q_values, state, epsilon):
  if ( (np.random.rand() < epsilon) or (not
q_values[state, :].any()) ):
        return np.random.choice(np.arange(0, 6))
  else:
        return np.argmax(q_values[state, :])
```

Task 2

Below is an incomplete code cell with the flow of SMDP Q-Learning. Complete the cell and train the agent using SMDP Q-Learning algorithm. Keep the **final Q-table** and **Update Frequency** table handy (You'll need it in TODO 4)

```
#### SMDP Q-Learning

# Add parameters you might need here
gamma = 0.9
alpha = 0.1

# Iterate over 1000 episodes
for _ in range(1000):

  state = env.reset()
  done = False
```

```
  # Loop till environment is solved
  while not done:

    # Epsilon-greedy action selection
    action = egreedy_policy(q_values_SMDP, state, epsilon=0.1)

    # Primitive actions
    if action < 4:

      # Regular Q-Learning update
      next_state, reward, done, dummy = env.step(action)
      q_values_SMDP[state, action] +=  alpha*(reward +
gamma*np.max(q_values_SMDP[next_state, :4]) - q_values_SMDP[state,
action])
      update_freq_SMDP[state, action] += 1
      state = next_state

    # Away option
    if action == 4:

      # Initialize reward bar, start_state and optdone
      reward_bar = 0
      optdone = False
      start_state = state
      time_steps = 0

      # To account for the case where agent is already at the top
boundary, we add an environment step outside the loop
      # Otherwise, away option, which leads to an exit from the
gridworld will yield 0 reward, thereby making this favourbale
      # This will lead to undesirable behaviour and so we add this
block outside the loop
      optact,_ = Away(env,state)
      next_state, reward, done,_ = env.step(optact)
      time_steps += 1
      reward_bar +=  (gamma**(time_steps - 1))*reward
      state = next_state

      while (optdone == False):

        # Apply the option
        optact,optdone = Away(env,state)

        # If option is done, update and terminate
        # We multiply with gamma^time_steps to allow for correct
scaling
        if optdone:
          q_values_SMDP[start_state, action] += alpha*(reward_bar +
(gamma**time_steps)*np.max(q_values_SMDP[state, 4:]) -
```

```
q_values_SMDP[start_state, action])
          update_freq_SMDP[start_state, action] += 1
          break

        # Note that we also change the reward_bar update slightly
        # This is because we apply discounting to the later rewards
and not the initial rewards
        # We keep track of how late a reward is using the time_steps
variable
        next_state, reward, done,_ = env.step(optact)
        time_steps += 1
        reward_bar +=  (gamma**(time_steps - 1))*reward
        state = next_state

    # Close option
    # Except for using close option, everything is identical to the
code for away option
    if action == 5:

      #print("Close:")
      reward_bar = 0
      optdone = False
      start_state = state
      time_steps = 0

      optact,_ = Close(env,state)
      next_state, reward, done,_ = env.step(optact)
      time_steps += 1
      reward_bar +=  (gamma**(time_steps - 1))*reward
      state = next_state

      while (optdone == False):

        optact,optdone = Close(env,state)

        if optdone:
          q_values_SMDP[start_state, action] +=  alpha*(reward_bar +
(gamma**time_steps)*np.max(q_values_SMDP[state, 4:]) -
q_values_SMDP[start_state, action])
          update_freq_SMDP[start_state, action] += 1
          break

        next_state, reward, done,_ = env.step(optact)
        time_steps += 1
        reward_bar +=  (gamma**(time_steps - 1))*reward
        state = next_state
```

Task 3

Using the same options and the SMDP code, implement Intra Option Q-Learning (In the code cell below). You *might not* always have to search through options to find the options with similar policies, think about it. Keep the **final Q-table** and **Update Frequency** table handy (You'll need it in TODO 4)

In our particular case, we only have two options: Away and Close. Away always generates an UP action, whereas Close always generates a DOWN action. It is therefore evident that these two options can never have the same action choice . Therefore, for this particular problem, it is sufficient to update the two options separately as they have no similar policies.

```
#### Intra Option Q-Learning

# Add parameters you might need here
gamma = 0.9
alpha = 0.1

# Iterate over 1000 episodes
for _ in range(1000):

  state = env.reset()
  done = False


  while not done:

    # Epsilon-greedy action selection
    action = egreedy_policy(q_values_IO, state, epsilon=0.1)

    # Primitive action selection; identical to SMDP
    if action < 4:

      next_state, reward, done, _ = env.step(action)
      q_values_IO[state, action] +=  alpha*(reward +
gamma*np.max(q_values_IO[next_state, :4]) - q_values_IO[state,
action])
      update_freq_IO[state, action] += 1
      state = next_state

    # Away option
    if action == 4:

      # Initializing variables
      optdone = False
      start_state = state
```

```
      # As with SMDP case, we add an environment step outside the loop


      optact,_ = Away(env,state)
      next_state, reward, done,_ = env.step(optact)
      state = next_state

      while (optdone == False):

        optact,optdone = Away(env,state)

        # If option is done, use the termination update condition and
break
        if optdone:
          q_values_IO[start_state, action] += alpha*(reward +
(gamma)*np.max(q_values_IO[state, :4]) - q_values_IO[start_state,
action])
          update_freq_IO[start_state, action] += 1
          break

        # For other cases, use the following update on a loop
        q_values_IO[start_state, action] += alpha*(reward +
(gamma)*np.max(q_values_IO[state, action]) - q_values_IO[start_state,
action])
        update_freq_IO[start_state, action] += 1

        # Note also that in this case reward_bar and time_steps are
unnecessary as we update on every step

        next_state, reward, done,_ = env.step(optact)
        start_state = state
        state = next_state

    # Close option
    # Again identical to away except for the fact that close option is
used
    if action == 5:

      optdone = False
      start_state = state

      optact,_ = Close(env,state)
      next_state, reward, done,_ = env.step(optact)
      state = next_state

      while (optdone == False):

        optact,optdone = Close(env,state)
```

```
        if optdone:
          q_values_IO[start_state, action] += alpha*(reward +
(gamma)*np.max(q_values_IO[state, :4]) - q_values_IO[start_state,
action])
          update_freq_IO[start_state, action] += 1
          break

        q_values_IO[start_state, action] += alpha*(reward +
(gamma)*np.max(q_values_IO[state, action]) - q_values_IO[start_state,
action])
        update_freq_IO[start_state, action] += 1

        next_state, reward, done,_ = env.step(optact)
        start_state = state
        state = next_state
```

Task 4

Compare the two Q-Tables and Update Frequencies and provide comments.

We first take argmax over all actions/ options for each state for SMDP and intra option q-learning. This gives us an idea of the policy that will be learnt.

In the following visualization, the print statement is structured such that each action corresponds to the optimal action at that particular grid cell, i.e., the action at grid position $(3, 0)$ would correspond to the action at the start state.

```
print("Q-Table for SMDP Q-Learning:")
print(np.argmax(q_values_SMDP, axis = 1).reshape(4, 12))
```

```
Q-Table for SMDP Q-Learning:
[[2 3 0 4 2 1 1 1 1 1 2 5]
 [1 1 1 2 1 1 4 1 2 2 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 0 0]]
```

```
print("Q-Table for Intra-Option Q-Learning:")
print(np.argmax(q_values_IO, axis = 1).reshape(4, 12))
```

```
Q-Table for Intra-Option Q-Learning:
[[1 5 1 1 2 1 3 1 4 5 4 2]
 [1 1 1 1 1 2 1 1 1 1 5 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 0 0]]
```

We make the following observations from the Q-value table:

- As expected, both methods learn to select the correct action near the start state. We can see that last two rows are identical for both methods.

- Being greedy with respect to the q-function from the start state leads to the optimal policy for both methods.
- Both methods deviate from the optimal policy slightly in the first two rows. This could be because the start state is in the last row and hence these rows may not be explore enough. Note however that this does not lead to suboptimal results as we are only interested in a fixed start state.

```
print(update_freq_SMDP)
```

```
[[ 129.  215.  184.  129.  129.   61.]
 [ 122.  235.  175.   92.  123.   60.]
 [ 114.  235.  169.   85.  114.   57.]
 [ 106.  231.  157.   78.  105.   54.]
 [  98.  222.  147.   71.   97.   51.]
 [  89.  209.  137.   65.   89.   46.]
 [  80.  198.  124.   58.   79.   42.]
 [  70.  182.  111.   51.   70.   37.]
 [  61.  154.  100.   44.   61.   34.]
 [  52.  129.   90.   37.   51.   28.]
 [  42.   91.   80.   31.   42.   26.]
 [  32.   32.   76.   25.   32.   48.]
 [  97.  211.   84.  124.   95.   81.]
 [  89.  227.   88.   88.   90.   81.]
 [  83.  233.   85.   79.   82.   77.]
 [  76.  233.   81.   73.   75.   72.]
 [  69.  217.   76.   68.   69.   67.]
 [  62.  205.   72.   60.   60.   60.]
 [  55.  190.   68.   53.   53.   55.]
 [  48.  175.   66.   45.   46.   47.]
 [  40.  151.   63.   40.   40.   41.]
 [  34.  130.   63.   32.   33.   33.]
 [  24.   97.   66.   24.   26.   31.]
 [  18.   21.   74.   18.   18.   74.]
 [ 296. 1255.   94.  143.   88.  101.]
 [ 198. 1139.   26.   94.   83.   36.]
 [ 170. 1072.   23.   92.   68.   18.]
 [ 152. 1012.   21.   85.   63.   27.]
 [ 126.  957.   25.   81.   57.   22.]
 [ 112.  934.   15.   62.   55.   16.]
 [ 100.  897.   18.   59.   54.   26.]
 [  86.  871.   18.   55.   45.   19.]
 [  68.  868.   13.   47.   33.   18.]
 [  60.  864.   16.   38.   32.   15.]
 [  40.  886.   15.   33.   24.   20.]
 [  29.   26.  661.   31.   22.  339.]
 [1514.   32.  160.  155.   88.  150.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
```

```
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]]
```

```
print(update_freq_IO)
```

```
[[ 128.  214.  135.  128.  127.  105.]
 [ 120.  229.  125.   91.  119.  108.]
 [ 112.  233.  118.   82.  111.  105.]
 [ 104.  224.  111.   75.  103.   99.]
 [  95.  221.  102.   68.   94.   92.]
 [  86.  202.   93.   66.   85.   87.]
 [  77.  181.   83.   55.   76.   77.]
 [  67.  159.   75.   49.   67.   68.]
 [  58.  136.   67.   42.   57.   62.]
 [  50.  108.   60.   36.   49.   54.]
 [  40.   72.   53.   30.   39.   51.]
 [  31.   31.   52.   26.   31.   51.]
 [ 120.  203.   95.  125.  243.  122.]
 [ 114.  230.  106.   90.  143.  125.]
 [ 106.  232.  105.   80.  128.  120.]
 [  96.  228.  100.   73.  112.  115.]
 [  87.  220.   94.   66.  106.  105.]
 [  77.  206.   87.   60.   95.   98.]
 [  68.  189.   82.   52.   77.   87.]
 [  58.  175.   75.   46.   70.   82.]
 [  48.  148.   69.   40.   64.   76.]
 [  38.  121.   65.   31.   54.   69.]
 [  30.   83.   64.   25.   42.   65.]
 [  19.   23.   81.   19.   38.   83.]
 [ 221. 1360.  107.  147.  232.  100.]
 [ 157. 1231.   35.   95.  135.   34.]
 [ 129. 1149.   33.   96.  116.   24.]
 [ 117. 1082.   26.   91.   99.   25.]
 [  98. 1027.   24.   76.   98.   23.]
 [  90.  983.   22.   65.   81.   31.]
 [  78.  951.   27.   60.   65.   24.]
 [  72.  930.   25.   53.   62.   18.]
 [  56.  900.   26.   52.   54.   24.]
 [  51.  892.   18.   37.   40.   25.]
 [  34.  920.   21.   29.   31.   16.]
 [  26.   34.  660.   30.   28.  340.]
 [1510.   37.  156.  158.  198.  164.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.]
```

```
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]
[   0.    0.    0.    0.    0.    0.]]
```

We make the following observations from the update_frequency table:

- The last 11 rows are not updated for either methods. This is to be expected because these correspond to the cliff and the terminal states. Since these are not valid states (as per gym documentation), it makes sense that updates do not happen.
- The primitive actions have similar update frequency for both methods. This is to be expected as both have the same code for updating q-value of primitive actions.
- We see that the update frequency is much higher for options in intra-option q-learning as compared to SMDP. This is because in intra-option q-learning, we update the q-value on every step of the option, whereas in SMDP we only make one update once the option terminates. Of course this is in agreement with the general theme of intra-option q-learning as it was originally designed keeping in mind the low update-frequency of SMDP q-learning.