INF8953DE Assignment 2

Emile Dimas

TOTAL POINTS

48.5 / 52

QUESTION 1

Q1. Monte Carlo Methods 18 pts

1.1 Q1.1 2 / 2

✓ **- 0 pts Correct**

1.2 Q1.2 7 / 7

✓ **- 0 pts Correct**

1.3 Q1.3 8 / 9

✓ **- 1 pts Mention the bias-variance tradeoff**

-1: Mention bias-variance tradeoff

QUESTION 2

Q2. Prediction: Unifying MC methods and TD Learning 22 pts

2.1 Q2.1 4.5 / 5

✓ **- 0.5 pts Value estimates of terminal states fail to converge to -100.**

2.2 Q2.2 3 / 3

✓ **- 0 pts Correct**

2.3 Q2.3 4 / 5

✓ **- 1 pts Explanation: general intuition that when n=1, n-step TD updates are identical to TD(0)**

2.4 Q2.4 2 / 3

✓ **- 1 pts Plots**

Did you make sure to set n=100? The plot looks eerily similar to n=1 which shouldn't be the case.

2.5 Q2.5 6 / 6

✓ **- 0 pts Correct**

QUESTION 3

Q3. Temporal Difference Control Methods 12 pts

3.1 Q3.1 3 / 3

✓ **- 0 pts Correct**

3.2 Q3.2 3 / 3

✓ **- 0 pts Correct**

3.3 Q3.3 3 / 3

✓ **- 0 pts Correct**

3.4 Q3.4 3 / 3

✓ **- 0 pts Correct**

RL_assignment_2_1

October 26, 2021

[8]:
```
import numpy as np
from tqdm import trange
import matplotlib.pyplot as plt
```

[9]:
```
!git clone https://github.com/micklethepickle/modified-frozen-lake.git
```

```
Cloning into 'modified-frozen-lake'…
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 12 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), done.
```

[10]:
```
%cd /content/modified-frozen-lake
```

```
/content/modified-frozen-lake
```

[38]:
```
import frozen_lake
import numpy as np
env = frozen_lake.FrozenLakeEnv(map_name="4x4", slip_rate=0.1)
```

0.1 Question 1: Create a function generate episode

[39]:
```
def generate_episode(policy, env, render=False):

  state = env.reset()
  states = [state]
  actions = []
  rewards = []
  done = False
  while not(done):
    curr_state = env.s
    policy_ = policy[curr_state, :]
    action = np.random.choice(env.action_space.n, 1, p=policy_ )[0]
    next_state, reward, done, extra = env.step(action)


    actions.append(action)
    states.append(next_state)
```

```
    rewards.append(reward)

    if render:
      env.render()
  return states, actions, rewards
```

0.2 Question 2: First-visit Mionte Carlo control algorithm

[40]:
```
def find_first_occurence(states, actions, tup):
  tuple_sa = [(states[i], actions[i]) for i in range(len(actions))]
  for idx,i in enumerate(tuple_sa):
    if i == tup:
      return idx
```

[49]:
```
gamma = 0.99
def FirstVisitMC(epsilon=0.05, episode=2000):
  policy = np.ones((env.observation_space.n, env.action_space.n))/env.
 ↪action_space.n
  # Q = np.zeros((env.observation_space.n, env.action_space.n))
  Q = np.random.rand(env.observation_space.n,env.action_space.n)
  r = []
  returns = []
  # create an empty list for returns
  for state in range(env.observation_space.n):
    returns.append([])
    for action in range( env.action_space.n):
      returns[state].append([])

  # loop through the episodes
  for e in trange(episode):
    # generate one episode
    states, actions, rewards = generate_episode(policy, env, render=False)

    r.append(sum(rewards))  # return per episode for the plots

    # create the tuples (S_t, A_t)
    tuples = []
    for k in range(0, len(actions)):
      st = states[k]
      at = actions[k]
      tuples.append((st,at))

    for state in range(env.observation_space.n):
      for action in range(env.action_space.n):
        if (state, action) in tuples:
          idx = find_first_occurence(states, actions, (state, action))
          Gs = [r*gamma**i for i,r in enumerate(rewards[idx:])]
```

1.1 Q1.1 2 / 2

✓ - 0 pts Correct

```
    rewards.append(reward)

    if render:
      env.render()
  return states, actions, rewards
```

0.2 Question 2: First-visit Mionte Carlo control algorithm

[40]:
```
def find_first_occurence(states, actions, tup):
  tuple_sa = [(states[i], actions[i]) for i in range(len(actions))]
  for idx,i in enumerate(tuple_sa):
    if i == tup:
      return idx
```

[49]:
```
gamma = 0.99
def FirstVisitMC(epsilon=0.05, episode=2000):
  policy = np.ones((env.observation_space.n, env.action_space.n))/env.
 ↪action_space.n
  # Q = np.zeros((env.observation_space.n, env.action_space.n))
  Q = np.random.rand(env.observation_space.n,env.action_space.n)
  r = []
  returns = []
  # create an empty list for returns
  for state in range(env.observation_space.n):
    returns.append([])
    for action in range( env.action_space.n):
      returns[state].append([])

  # loop through the episodes
  for e in trange(episode):
    # generate one episode
    states, actions, rewards = generate_episode(policy, env, render=False)

    r.append(sum(rewards))  # return per episode for the plots

    # create the tuples (S_t, A_t)
    tuples = []
    for k in range(0, len(actions)):
      st = states[k]
      at = actions[k]
      tuples.append((st,at))

    for state in range(env.observation_space.n):
      for action in range(env.action_space.n):
        if (state, action) in tuples:
          idx = find_first_occurence(states, actions, (state, action))
          Gs = [r*gamma**i for i,r in enumerate(rewards[idx:])]
```

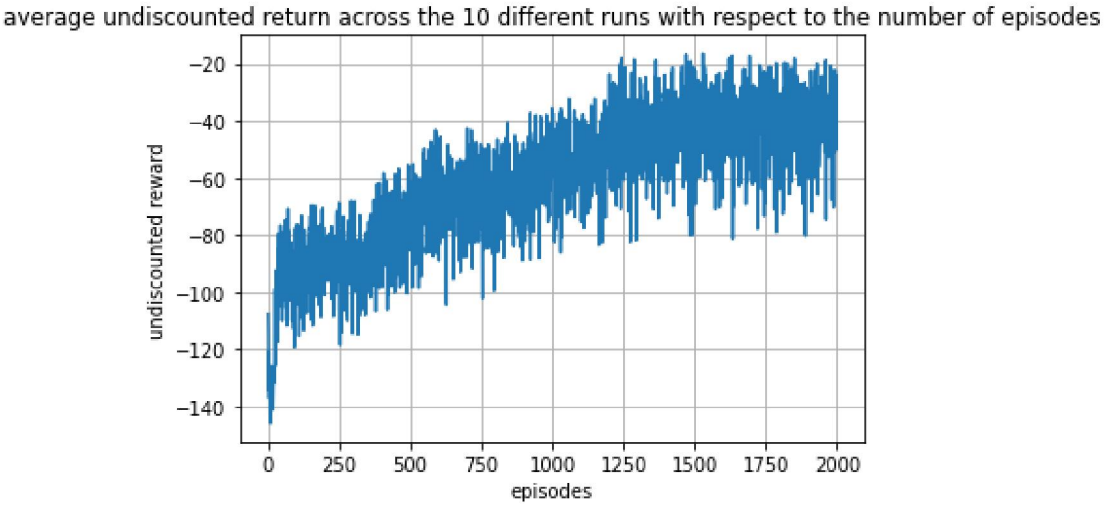
```
            returns[state][action].append(sum(Gs))

            Q[state,action] = np.mean(returns[state][action])
            a_star = np.argmax(Q[state,:])
            mask = np.ones(env.action_space.n)
            mask[a_star] = 0
            policy[state,:] = mask*epsilon/env.action_space.n␣
 ↪+(1-mask)*(1-epsilon + epsilon/env.action_space.n)

  return policy,r
```

[51]:
```
Returns = []
policies = []
for _ in range(10):
  policy,r = FirstVisitMC(epsilon=0.05, episode=2000)
  Returns.append(r)
  policies.append(policy)
```

```
100%|      | 2000/2000 [00:02<00:00, 836.73it/s]
100%|      | 2000/2000 [00:02<00:00, 782.37it/s]
100%|      | 2000/2000 [00:02<00:00, 848.22it/s]
100%|      | 2000/2000 [00:05<00:00, 340.68it/s]
100%|      | 2000/2000 [00:05<00:00, 341.67it/s]
100%|      | 2000/2000 [00:02<00:00, 870.77it/s]
100%|      | 2000/2000 [00:06<00:00, 311.30it/s]
100%|      | 2000/2000 [00:09<00:00, 209.43it/s]
100%|      | 2000/2000 [00:03<00:00, 569.44it/s]
100%|      | 2000/2000 [00:03<00:00, 558.44it/s]
```

[52]:
```
# a) Plot the average undiscounted return across the 10 different runs with␣
 ↪respect to
# the number of episodes (x-axis is the 2000 episodes, y-axis is the return for␣
 ↪each
# episode).
plt.figure()
plt.grid()
y = np.mean(np.array(Returns), axis=0)
x = np.arange(len(y))
plt.plot(x,y)
plt.xlabel('episodes')
plt.ylabel('undiscounted reward')
plt.title('average undiscounted return across the 10 different runs with␣
 ↪respect to the number of episodes')
```

[52]:
```
Text(0.5, 1.0, 'average undiscounted return across the 10 different runs with
respect to the number of episodes')
```



[ ]:
```
# b) Visualize an episode, by generating and rendeingr an episode using one of␣
 ↪the 10
# learned approximate  * to verify its optimality.
chosen_policy = np.random.randint(0,10,1)[0]
policy_chosen = policies[chosen_policy]
generate_episode(policy_chosen, env, render=True)
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
```

```
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
```

[ ]:
```
([0, 1, 2, 6, 10, 9, 13, 14, 15, 15],
 [2, 2, 1, 1, 0, 1, 2, 2, 0],
 [-1, -1, -1, -1, -1, -1, -1, -1, 0])
```

[ ]:
```
# the average undiscounted return for the last 100 episodes of the 10 runs␣
 ↪(average over the 100 episodes, over 10 runs)?
a = np.mean(np.array(Returns)[:,-100:],axis=0)
#plt.plot(np.arange(len(a)),a)
print('The average over the last 100 episodes for the 10 runs is: {}'.format(np.
 ↪mean(a)))
```

```
The average over the last 100 episodes for the 10 runs is: -63.246
```

0.2.1 Why might this be smaller than you'd expect?

Using Control Methods, we should converge to the optimal policy. We can see that the average over the last 100 episodes of training are smaller than what we expect from an optimal policy. This is due to the fact that First visit MC control is an on policy method and using an epsilon-greedy policy. On top of that there is also the stochasticity of the environment that also has an effect on the total reward.

1.2 Q1.2 7 / 7

✓ - 0 pts Correct

1 Question 3: Importance Sampling

[ ]:
```
def plot_many(experiments, label=None, color=None):
  mean_exp = np.mean(experiments, axis=0)
  std_exp = np.std(experiments, axis=0)
  plt.plot(mean_exp, color=color, label=label)
  plt.fill_between(range(len(experiments[0])), mean_exp + std_exp,
  mean_exp - std_exp, color=color, alpha=0.1)
```

[ ]:
```
# off-policy MC prediction
gamma = 0.99
def WIS(target_policy, epsilon=0.05, episode=2000):
  Vs = []
  Q = np.random.rand(env.observation_space.n,env.action_space.n)
  Q = np.zeros((env.observation_space.n,env.action_space.n))
  C = np.zeros((env.observation_space.n,env.action_space.n))
  for e in trange(episode):
    b = np.ones((env.observation_space.n,env.action_space.n))/env.action_space.n
    states, actions, rewards = generate_episode(b, env, render=False)
    G = 0
    W = 1
    for i in range(len(actions)-1,-1,-1):
      s = states[i]
      a = actions[i]
      G = gamma*G + rewards[i]
      C[s,a] = C[s,a] + W
      Q[s,a] = Q[s,a] + (W/C[s,a]) *(G - Q[s,a])
      W = W *(target_policy[s,a]/b[s,a])

    Vs.append(np.max(Q,axis=1))

  return Q, Vs
```

[ ]:
```
def OIS(target_policy, epsilon=0.05, episode=2000):
  Vs = []
  Q = np.random.rand(env.observation_space.n,env.action_space.n)
  Q = np.zeros((env.observation_space.n,env.action_space.n))
  C = np.zeros((env.observation_space.n,env.action_space.n))
  for e in trange(episode):
    b = np.ones((env.observation_space.n,env.action_space.n))/env.action_space.n
    states, actions, rewards = generate_episode(b, env, render=False)
    G = 0
    W = 1
    for i in range(len(actions)-1,-1,-1):
      s = states[i]
      a = actions[i]
      G = gamma*G + rewards[i]
```

```
      C[s,a] = C[s,a] + 1
      Q[s,a] = Q[s,a] + (W/C[s,a]) *(G - Q[s,a])
      W = W *(target_policy[s,a]/b[s,a])
    Vs.append(np.max(Q,axis=1))

  return Q,Vs
```

[ ]:
```
idx = np.random.randint(0,10,1)[0]
target_policy = policies[idx]
print('The chosen policy is the {}th policy'.format(idx))
```

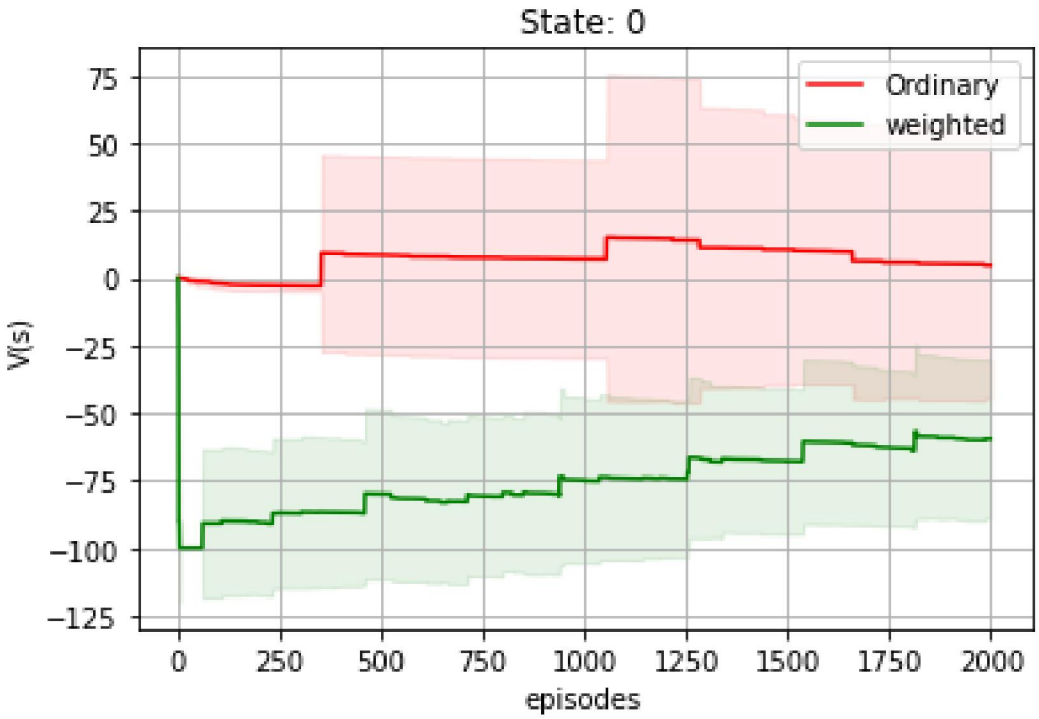
```
The chosen policy is the 6th policy
```

[ ]:
```
states_of_interest = [0, 3, 7, 14]

exp_OIS = {0:[],3:[], 7:[], 14:[]}

exp_WIS = {0:[],3:[], 7:[], 14:[]}

for _ in range(10):
  Q_OIS,V_OIS = OIS(target_policy, epsilon=0.05, episode=2000)
  Q_WIS, V_WIS = WIS(target_policy, epsilon=0.05, episode=2000)
  for s in states_of_interest:
    v_o = np.array(V_OIS)[:,s]
    v_w = np.array(V_WIS)[:,s]
    exp_OIS[s].append(v_o)
    exp_WIS[s].append(v_w)
```

```
100%|      | 2000/2000 [00:01<00:00, 1556.18it/s]
100%|      | 2000/2000 [00:01<00:00, 1616.54it/s]
100%|      | 2000/2000 [00:01<00:00, 1578.98it/s]
100%|      | 2000/2000 [00:01<00:00, 1516.95it/s]
100%|      | 2000/2000 [00:01<00:00, 1515.83it/s]
100%|      | 2000/2000 [00:01<00:00, 1551.52it/s]
100%|      | 2000/2000 [00:01<00:00, 1549.95it/s]
100%|      | 2000/2000 [00:01<00:00, 1514.90it/s]
100%|      | 2000/2000 [00:01<00:00, 1486.37it/s]
100%|      | 2000/2000 [00:01<00:00, 1444.89it/s]
100%|      | 2000/2000 [00:01<00:00, 1547.97it/s]
100%|      | 2000/2000 [00:01<00:00, 1545.79it/s]
100%|      | 2000/2000 [00:01<00:00, 1505.13it/s]
100%|      | 2000/2000 [00:01<00:00, 1540.22it/s]
100%|      | 2000/2000 [00:01<00:00, 1601.02it/s]
100%|      | 2000/2000 [00:01<00:00, 1472.95it/s]
100%|      | 2000/2000 [00:01<00:00, 1485.82it/s]
100%|      | 2000/2000 [00:01<00:00, 1578.75it/s]
100%|      | 2000/2000 [00:01<00:00, 1516.16it/s]
100%|      | 2000/2000 [00:01<00:00, 1499.82it/s]
```

[ ]:
```
s= states_of_interest[0]
plt.figure()
plt.title('State: {}'.format(s))
plt.xlabel('episodes')
plt.ylabel('V(s)')
plt.grid()
xp1 = np.stack(exp_OIS[s])
xp2 = np.stack(exp_WIS[s])
plot_many(xp1, label='Ordinary', color='r')
plot_many(xp2, label='weighted', color='g')
plt.legend()
```

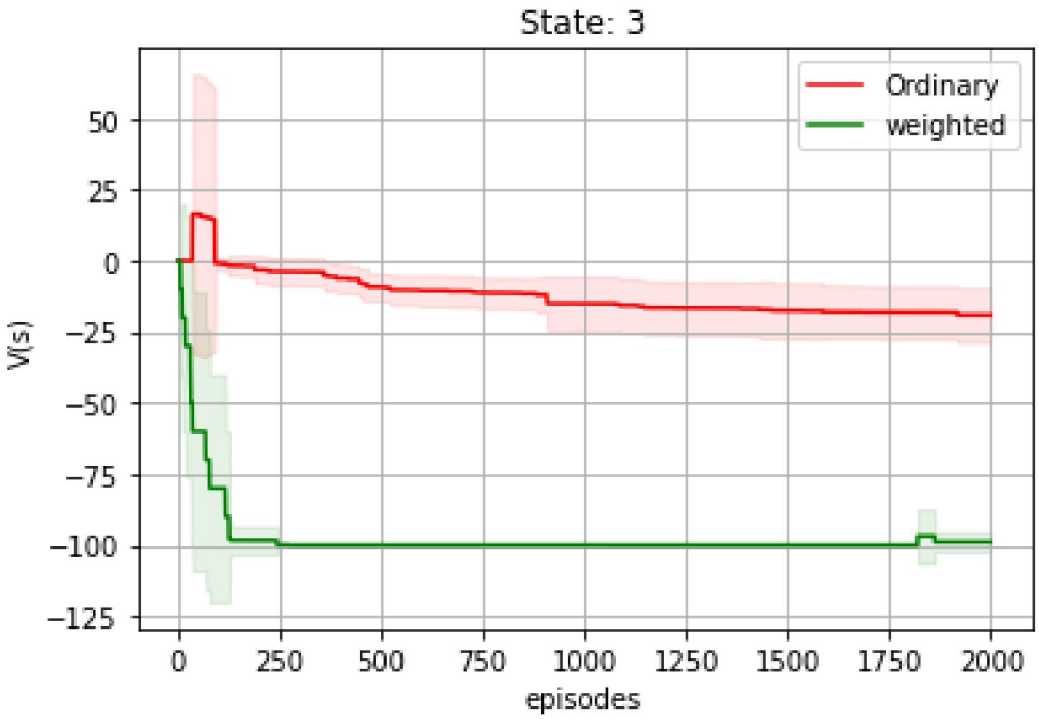
[ ]:
```
<matplotlib.legend.Legend at 0x7f7b88c67fd0>
```



[ ]:
```
s= states_of_interest[1]
plt.figure()
plt.title('State: {}'.format(s))
plt.xlabel('episodes')
plt.ylabel('V(s)')
plt.grid()
xp1 = np.stack(exp_OIS[s])
xp2 = np.stack(exp_WIS[s])
plot_many(xp1, label='Ordinary', color='r')
```

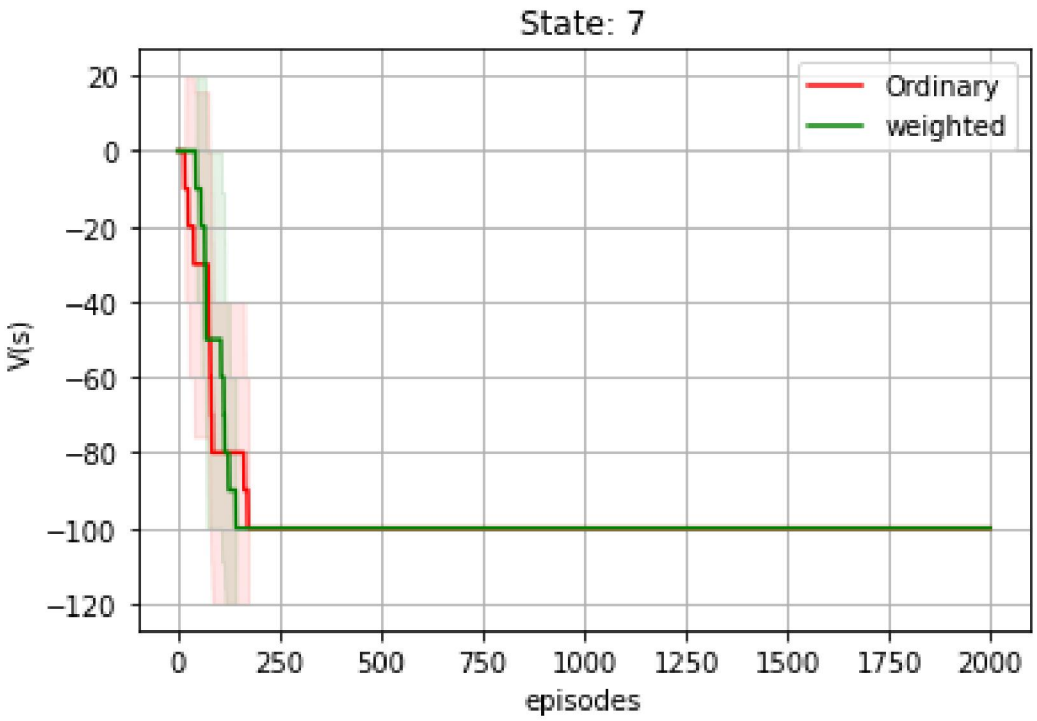
```
plot_many(xp2, label='weighted', color='g')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7f7b84ade350>



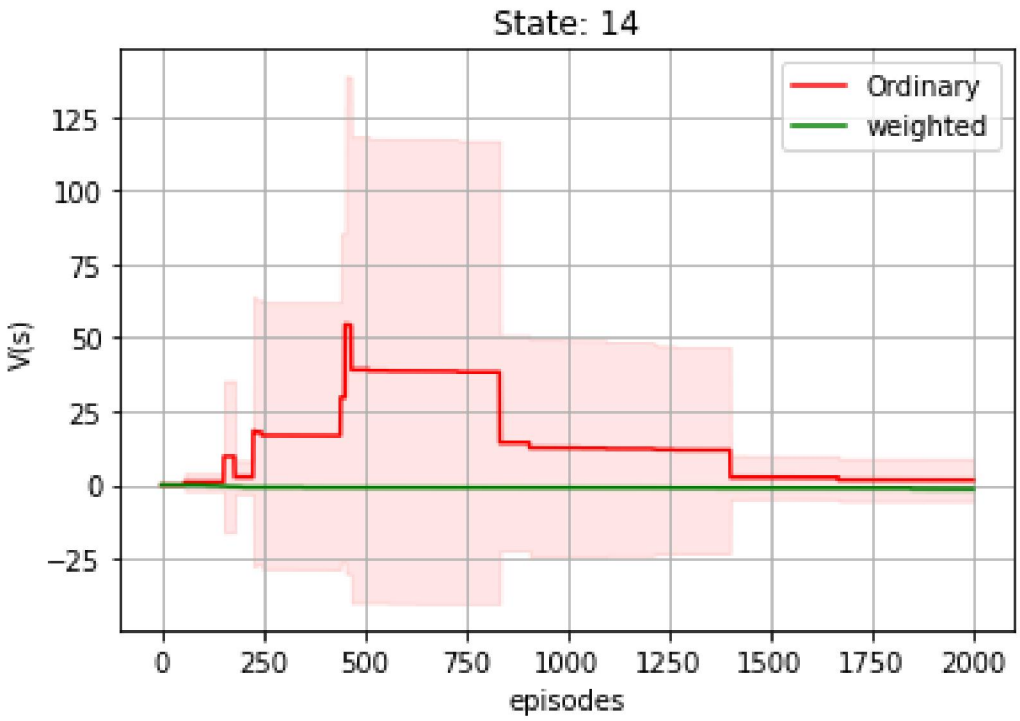
[ ]:
```
s= states_of_interest[2]
plt.figure()
plt.title('State: {}'.format(s))
plt.xlabel('episodes')
plt.ylabel('V(s)')
plt.grid()
xp1 = np.stack(exp_OIS[s])
xp2 = np.stack(exp_WIS[s])
plot_many(xp1, label='Ordinary', color='r')
plot_many(xp2, label='weighted', color='g')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7f7b84a02f90>



[ ]:
```
s= states_of_interest[3]
plt.figure()
plt.title('State: {}'.format(s))
plt.xlabel('episodes')
plt.ylabel('V(s)')
plt.grid()
xp1 = np.stack(exp_OIS[s])
xp2 = np.stack(exp_WIS[s])
plot_many(xp1, label='Ordinary', color='r')
plot_many(xp2, label='weighted', color='g')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f7b849df190>
```



Based on the theory, Ordinary Importance Sampling has more variance than Weighted Importance Sampling. This is because the weights in OIS are unbounded, therefore its variance isn't bounded either. The opposite is true for WIS. The weights are normalized, therefore bounded, therefore its variance is bounded.

This is confirmed by our experiences. in fact, looking at the plot we see that the standard deviation (shaded area around the lines) of OIS (red) is way larger than the one for WIS (green).

2 Exercise 2

[ ]:
```
env = frozen_lake.FrozenLakeEnv(map_name="4x4-easy", slip_rate=0)
```

[ ]:
```
random_policy =  np.ones((env.observation_space.n, env.action_space.n))/env.
↪action_space.n
```

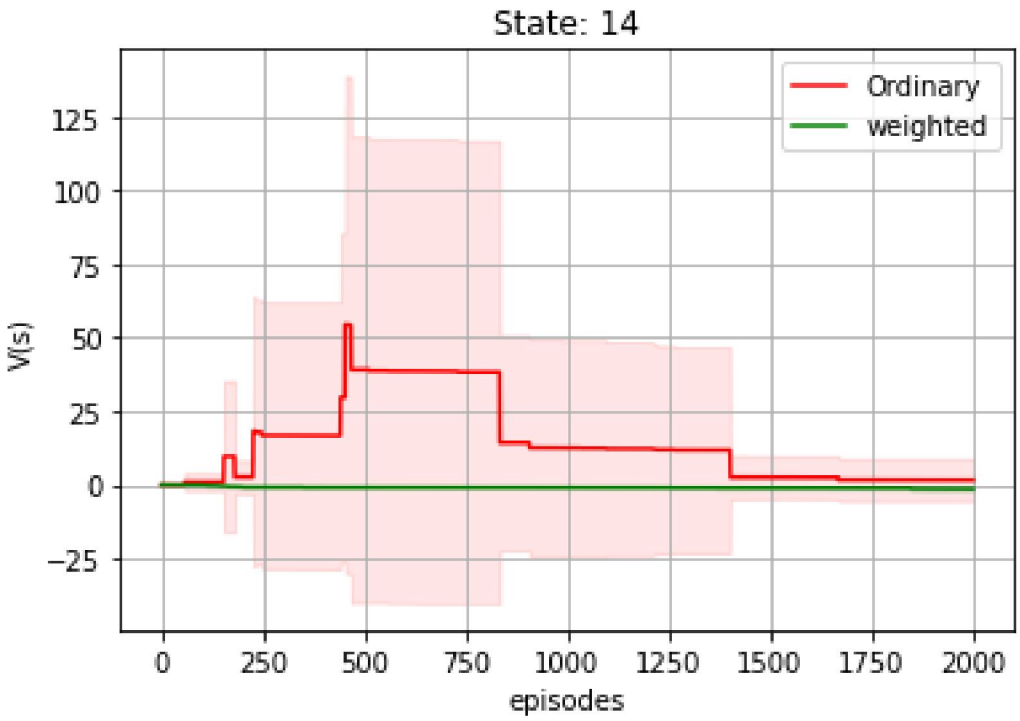
2.1 Question 1: Every Visit Monte-Carlo prediction

[ ]:
```
gamma = 0.99
def EveryVisitMC_prediction(policy, episode=10000):

  V = np.ones((env.observation_space.n,episode)) *1000
  r = []
  returns = []
  # create an empty list for returns
```

1.3 Q1.3 8 / 9

✓ **- 1 pts Mention the bias-variance tradeoff**

-1: Mention bias-variance tradeoff



Based on the theory, Ordinary Importance Sampling has more variance than Weighted Importance Sampling. This is because the weights in OIS are unbounded, therefore its variance isn't bounded either. The opposite is true for WIS. The weights are normalized, therefore bounded, therefore its variance is bounded.

This is confirmed by our experiences. in fact, looking at the plot we see that the standard deviation (shaded area around the lines) of OIS (red) is way larger than the one for WIS (green).

2 Exercise 2

[ ]:
```
env = frozen_lake.FrozenLakeEnv(map_name="4x4-easy", slip_rate=0)
```

[ ]:
```
random_policy =  np.ones((env.observation_space.n, env.action_space.n))/env.
↪action_space.n
```

2.1 Question 1: Every Visit Monte-Carlo prediction

[ ]:
```
gamma = 0.99
def EveryVisitMC_prediction(policy, episode=10000):

  V = np.ones((env.observation_space.n,episode)) *1000
  r = []
  returns = []
  # create an empty list for returns
```

```
  for state in range(env.observation_space.n):
    returns.append([])


  # loop through the episodes
  for e in trange(episode):
    # generate one episode
    states, actions, rewards = generate_episode(policy, env, render=False)
    G = 0
    length = len(actions)
    # for i in range(length-1,0,-1):
    #   G = gamma*G + rewards[i]
    #   state = states[i]
    #   returns[state].append(G)
    #   V[state, e] = np.mean(returns[state])
    G = [r*gamma**i for i,r in enumerate(rewards)]
    G = np.cumsum(G[::-1])[::-1]

    for s, r in zip(states[:-1],G):
      returns[int(s)].append(r)

    for s in range(env.observation_space.n):
      V[int(s),e] = np.mean(returns[int(s)])
      if int(s) in [5,12]:
          V[int(s)] = -100
      if int(s) in [15]:
          V[int(s)] = 0

  return V

# NOT SURE WHAT"S THE V THEY ARE ASKING FOR
```

[ ]:
```
V = EveryVisitMC_prediction(random_policy, episode=10000)
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-
packages/numpy/core/fromnumeric.py:3373: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:170:
RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
100%|     | 10000/10000 [01:02<00:00, 160.26it/s]
```

[ ]:
```
finalV = V[:,-1] # NOT SURE WHAT"S THE V THEY ARE ASKING FOR
print(finalV.reshape((4,4)))
```

```
[[ -91.33404403  -87.25074096  -75.21065536  -66.51350099]
 [ -91.50395363 -100.          -70.05254678  -58.10830496]
 [ -87.73308048  -76.86310037  -58.06424875  -38.88733091]
```

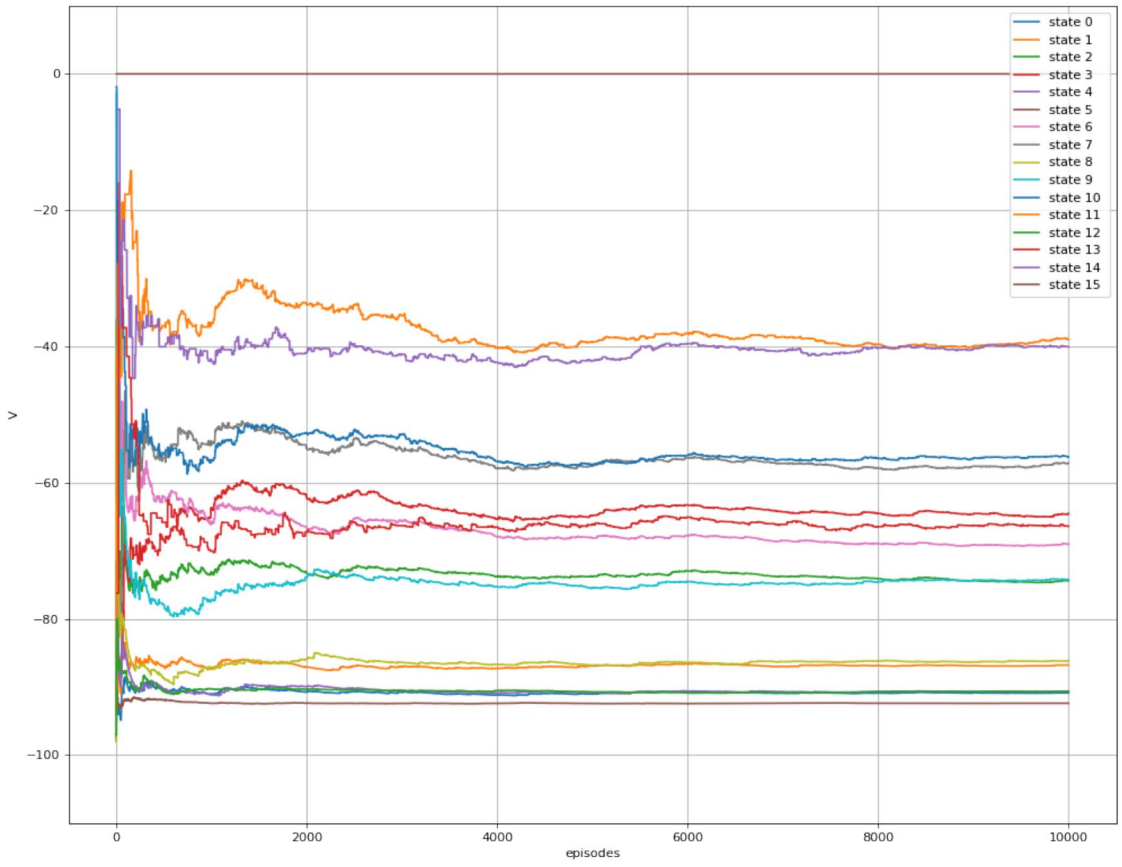
```
[-100.          -67.33617748  -40.55985425    0.          ]]
```

###Intuitively, do these values make sense?

These values do make sense. In fact, we can see that the further away we are from the goal, the lower the result is. Moreover, we can see that the values around the holes are lower than the values around the goal, which make complete sense.

[ ]:
```
plt.figure(figsize=(15, 12), dpi=80)
plt.grid()
# MAKE A BETTER GRAPH
x = np.arange(V.shape[-1])
for i in range(env.observation_space.n):
  plt.plot(x,V[i,:],'-', label='state %s ' % i)
plt.xlabel('episodes')
plt.ylabel('V')
plt.legend()

plt.ylim(-110, 10)
```

[ ]:
```
(-110.0, 10.0)
```



2.1 Q2.1 4.5 / 5

✓ **- 0.5 pts** Value estimates of terminal states fail to converge to -100.

2.2 Question 2: TD(0) prediction algorithm

[ ]:
```
def TD0(policy, alpha, episode=10000):
  # V = np.zeros((env.observation_space.n,episode))
  # V = np.random.randn(env.observation_space.n,episode)
  # V = np.random.randn(env.observation_space.n)
  V = np.zeros((env.observation_space.n,episode))
  V[-1,:] = 0
  for e in trange(episode):
    # if e>0:
    #   V[:,e] = V[:,e-1]
    states, actions, rewards = generate_episode(policy, env, render=False)

    for i, _ in enumerate(rewards):
        state = states[i]
        reward = rewards[i]
        next_state = states[i+1]
        V[state,e:] = V[state,e] + alpha*(reward + gamma*V[next_state,e] -␣
 ↪V[state,e])
        if state in [5,12,15]:
               V[state] =reward

  return V
```

[ ]:
```
V = TD0(random_policy, alpha=0.01,episode=10000)
```

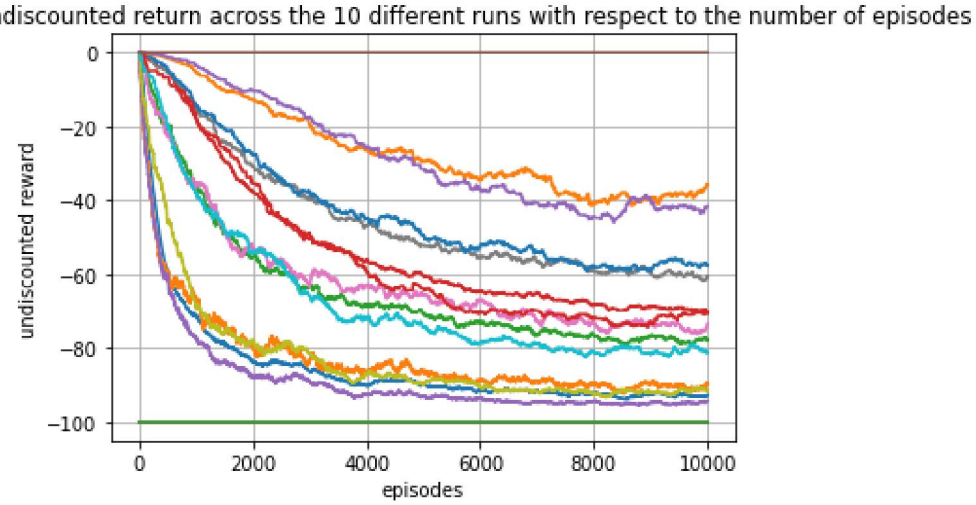
```
100%|      | 10000/10000 [00:07<00:00, 1392.58it/s]
```

[ ]:
```
finalV = V[:,-1]
print(finalV.reshape((4,4)))
```

```
[[ -92.6497231   -89.73083979  -77.99275867  -70.29349179]
 [ -94.25810391 -100.          -73.3213404   -60.65276251]
 [ -91.42332595  -81.13330378  -57.77114718  -35.87898507]
 [-100.          -70.63667999  -41.64331549    0.        ]]
```

[ ]:
```
plt.figure()
plt.grid()
# MAKE A BETTER GRAPH
x = np.arange(V.shape[-1])
for i in range(env.observation_space.n):
  plt.plot(x,V[i,:])
plt.xlabel('episodes')
plt.ylabel('undiscounted reward')
plt.title('average undiscounted return across the 10 different runs with␣
 ↪respect to the number of episodes')
```

[ ]:
```
Text(0.5, 1.0, 'average undiscounted return across the 10 different runs with
respect to the number of episodes')
```



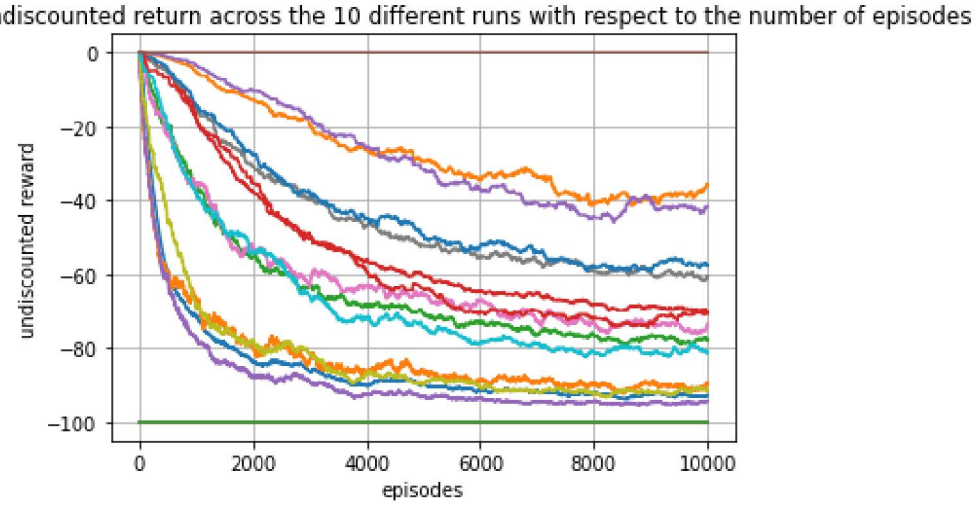
2.3 Question 3: n-step TD prediction algorithm (n=1)

[ ]:
```
def nStep_TD(policy, n, alpha, episode):
  Vs = np.empty((env.observation_space.n,episode))
  V = np.random.randn(env.observation_space.n)
  V[5] = -100
  V[12] =-100
  V[15] = 0
  #V = np.zeros(env.observation_space.n)
  for e in trange(episode):
    # print('_'*20)

    states, actions, rewards = generate_episode(policy, env, render=False)

    T = np.inf
    t=-1
    while True:

      t+=1

      if t<T:
        if t+1 == (len(actions)-1):
          T= t+1


      tau = t-n+1
```

2.2 Q2.2 3 / 3

✓ **- 0 pts** Correct

[ ]:
```
Text(0.5, 1.0, 'average undiscounted return across the 10 different runs with
respect to the number of episodes')
```



2.3 Question 3: n-step TD prediction algorithm (n=1)

[ ]:
```
def nStep_TD(policy, n, alpha, episode):
  Vs = np.empty((env.observation_space.n,episode))
  V = np.random.randn(env.observation_space.n)
  V[5] = -100
  V[12] =-100
  V[15] = 0
  #V = np.zeros(env.observation_space.n)
  for e in trange(episode):
    # print('_'*20)

    states, actions, rewards = generate_episode(policy, env, render=False)

    T = np.inf
    t=-1
    while True:

      t+=1

      if t<T:
        if t+1 == (len(actions)-1):
          T= t+1


      tau = t-n+1
```

```
      if tau >= 0:
        start_idx = tau+1

        end_idx = int(np.minimum(tau+n, T) + 1)
        G = sum([x*gamma**(i-tau-1) for i,x in zip(range(start_idx,end_idx),␣
↪rewards[start_idx:end_idx])])
        state_tau = states[tau]

        if (tau+n) < T:
          state_tau_n = states[tau+n]
          G += (gamma**n)*V[state_tau_n]
        V[state_tau] += alpha*(G-V[state_tau])
        if state_tau in [5,12]:
          V[state_tau] = -100
        if state_tau in [15]:
          V[state_tau] = 0
      if tau == (T-1) :
        repeat = episode - e
        # print(V)
        Vs[:,e:] = np.repeat(V[:,np.newaxis],repeat,axis=1)

        break

  return Vs
```

[ ]:
```
Vs = nStep_TD(random_policy, n=1, alpha=0.01, episode=10000)
```

```
100%|      | 10000/10000 [00:12<00:00, 815.10it/s]
```

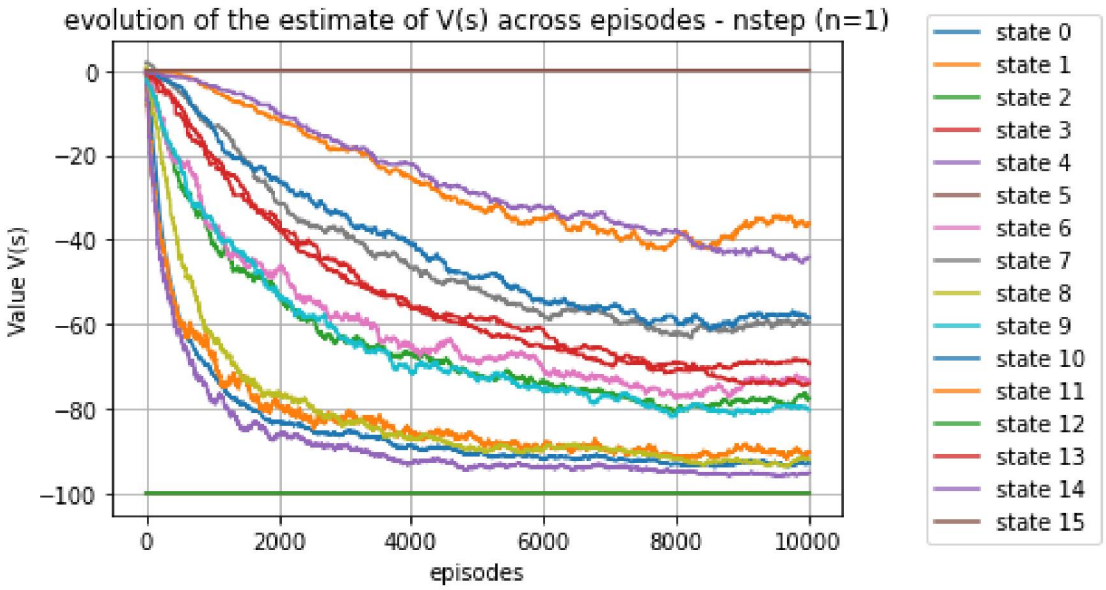
[ ]:
```
finalV = Vs[:,-1]
print(finalV.reshape((4,4)))
```

```
[[ -92.8504921   -90.2362797   -77.51890922  -69.40597099]
 [ -95.13410677 -100.          -73.53101076  -59.73314855]
 [ -92.05233415  -80.31191286  -58.32674287  -36.08307717]
 [-100.          -74.03570042  -44.17710822    0.        ]]
```

[ ]:
```
x = np.arange(10000)
plt.figure()
plt.grid()

for i in range(env.observation_space.n):
  plt.plot(x,Vs[i,:], label='state %s ' % i)

plt.legend(loc='center left', bbox_to_anchor=(1.1, 0.5))
plt.xlabel('episodes')
plt.ylabel('Value V(s)')
```

```
plt.title('evolution of the estimate of V(s) across episodes - nstep (n=1)')
```

[ ]:
```
Text(0.5, 1.0, 'evolution of the estimate of V(s) across episodes - nstep
(n=1)')
```



2.3.1 Compare

If we compare this plot with the 2 previous we got, we can conclude many things. We can see that the nstep TD (n=1) and TD(0) are very similar whereas EveryVisit MC is different. The latter is very noisy at the beginning and then stabilise very fast (after 2000 episodes). The nstep methods takes more time to converge. the 2 nsteps are very similar and it's hard to find out differences. Finally, we can almost say that the values converges toward similar values for all methods

2.4 Question 4: n-step TD prediction algorithm (n=100)

[ ]:
```
V = nStep_TD(random_policy, n=100, alpha=0.01, episode=10000)
```

```
100%|      | 10000/10000 [00:12<00:00, 797.14it/s]
```

[ ]:
```
finalV = V[:,-1]
print(finalV.reshape((4,4)))
```

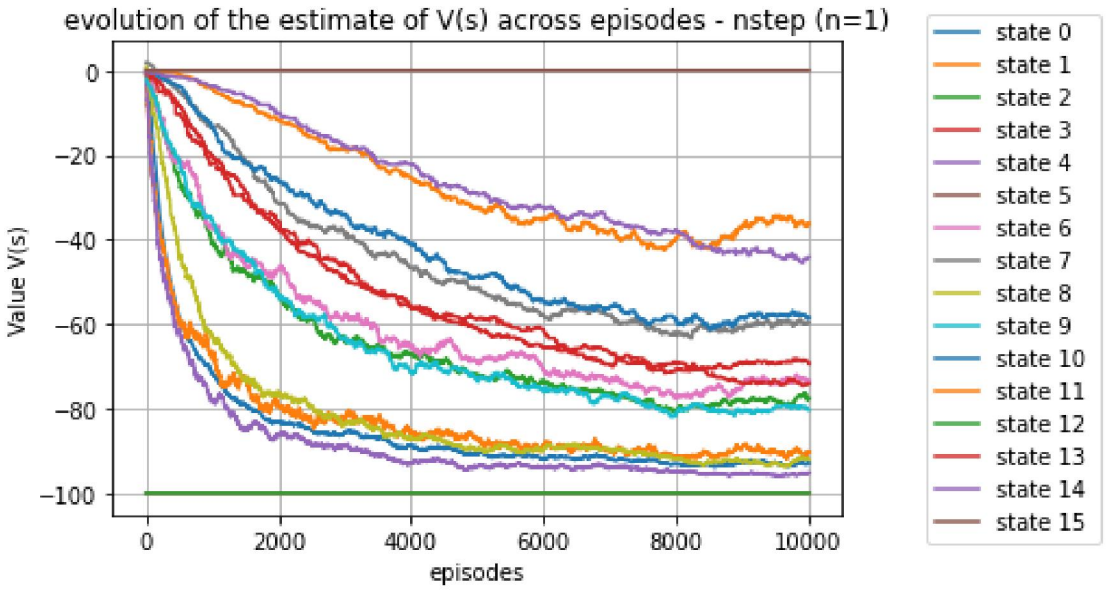
```
[[ -93.52845313  -89.2330557   -80.49866268  -77.41248583]
 [ -92.70089739 -100.          -75.95005941  -70.41327744]
 [ -91.00127033  -87.37323646  -65.3288898   -47.91430008]
 [-100.          -82.92261963  -53.31402022    0.        ]]
```

2.3 Q2.3 4 / 5

✓ **- 1 pts** **Explanation: general intuition that when n=1, n-step TD updates are identical to TD(0)**

```
plt.title('evolution of the estimate of V(s) across episodes - nstep (n=1)')
```

[ ]:
```
Text(0.5, 1.0, 'evolution of the estimate of V(s) across episodes - nstep
(n=1)')
```



2.3.1 Compare

If we compare this plot with the 2 previous we got, we can conclude many things. We can see that the nstep TD (n=1) and TD(0) are very similar whereas EveryVisit MC is different. The latter is very noisy at the beginning and then stabilise very fast (after 2000 episodes). The nstep methods takes more time to converge. the 2 nsteps are very similar and it's hard to find out differences. Finally, we can almost say that the values converges toward similar values for all methods

2.4 Question 4: n-step TD prediction algorithm (n=100)

[ ]:
```
V = nStep_TD(random_policy, n=100, alpha=0.01, episode=10000)
```

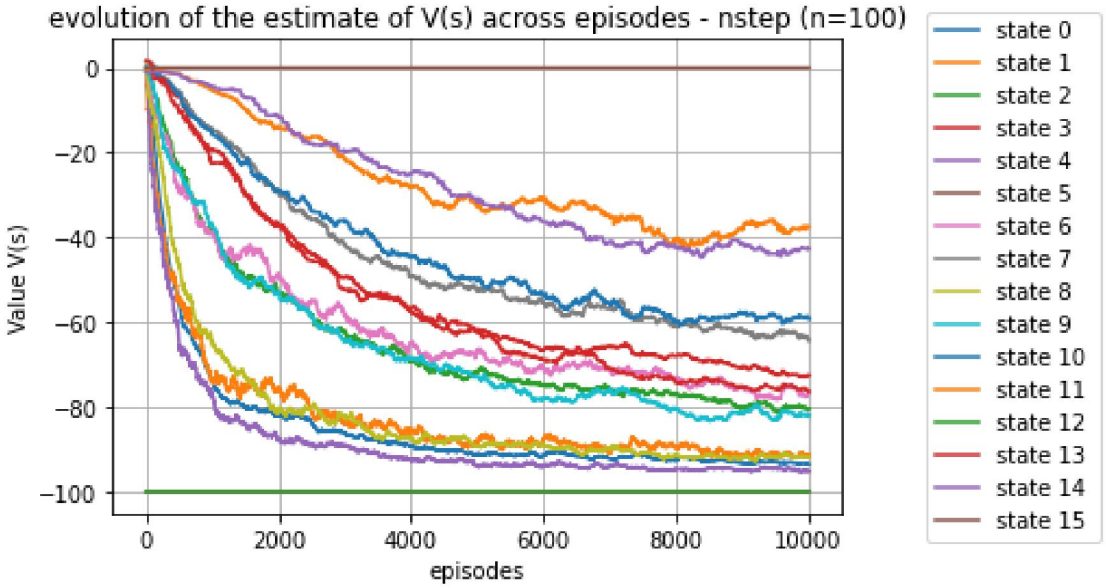
```
100%|      | 10000/10000 [00:12<00:00, 797.14it/s]
```

[ ]:
```
finalV = V[:,-1]
print(finalV.reshape((4,4)))
```

```
[[ -93.52845313  -89.2330557   -80.49866268  -77.41248583]
 [ -92.70089739 -100.          -75.95005941  -70.41327744]
 [ -91.00127033  -87.37323646  -65.3288898   -47.91430008]
 [-100.          -82.92261963  -53.31402022    0.        ]]
```

```
[ ]: x = np.arange(10000)
     plt.figure()
     plt.grid()


     for i in range(env.observation_space.n):
       plt.plot(x,Vs[i,:], label='state %s ' % i)

     plt.legend(loc='center left', bbox_to_anchor=(1.1, 0.5))
     plt.xlabel('episodes')
     plt.ylabel('Value V(s)')
     plt.title('evolution of the estimate of V(s) across episodes - nstep (n=100)')
```

```
[ ]: Text(0.5, 1.0, 'evolution of the estimate of V(s) across episodes - nstep
     (n=100)')
```



2.4.1 Compare

We can see that the 2 plots are different. However, nstep (n=100) converges faster than the other methods (which makes it closer to MC) However the learning curves are different and this is due to the step size as we will see in the next question

2.5 Question 5: Modify your n-step TD algorithm such that when n = 100, it becomes equivalent to Every visit Monte Carlo prediction.

In ordere to get tn-step TD to be equivalent to Every Visit MC prediction, we need to have the Value of a state equal to the average return from that state. Therefore we want to divide the total

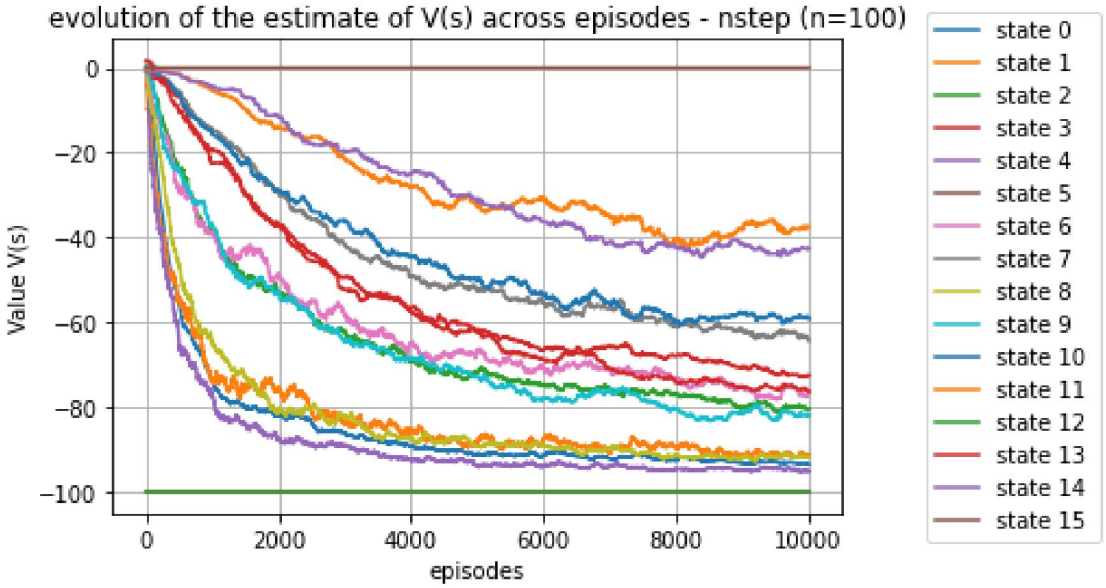
2.4 Q2.4 2 / 3

✓ **- 1 pts** **Plots**

Did you make sure to set n=100? The plot looks eerily similar to n=1 which shouldn't be the case.

```
[ ]: x = np.arange(10000)
     plt.figure()
     plt.grid()


     for i in range(env.observation_space.n):
       plt.plot(x,Vs[i,:], label='state %s ' % i)

     plt.legend(loc='center left', bbox_to_anchor=(1.1, 0.5))
     plt.xlabel('episodes')
     plt.ylabel('Value V(s)')
     plt.title('evolution of the estimate of V(s) across episodes - nstep (n=100)')
```

```
[ ]: Text(0.5, 1.0, 'evolution of the estimate of V(s) across episodes - nstep
     (n=100)')
```



2.4.1 Compare

We can see that the 2 plots are different. However, nstep (n=100) converges faster than the other methods (which makes it closer to MC) However the learning curves are different and this is due to the step size as we will see in the next question

2.5 Question 5: Modify your n-step TD algorithm such that when n = 100, it becomes equivalent to Every visit Monte Carlo prediction.

In ordere to get tn-step TD to be equivalent to Every Visit MC prediction, we need to have the Value of a state equal to the average return from that state. Therefore we want to divide the total

return by the number of visits in that state. We therefore use the learning rate as an incremental step size which is equal to the inverse of the number of times we have visited this state.

$$V(S_\tau) := V(S_\tau) + \alpha * (G - V(S_\tau))$$

$$\alpha = 1/N_\tau$$

We therefore get:

$$V(S_\tau) := V(S_\tau) + \frac{1}{N_\tau} * (G - V(S_\tau))$$

with G being the similar discounted return as MC method

[ ]:
```
gamm = 0.99
def Modified_nStep_TD(policy, n, episode):
  Vs = np.empty((env.observation_space.n,episode))
  V = np.random.randn(env.observation_space.n)
  V[5] = -100
  V[12] =-100
  V[15] = 0
  steps = np.zeros((env.observation_space.n))
  for e in trange(episode):
    # print('_'*20)

    states, actions, rewards = generate_episode(policy, env, render=False)

    T = np.inf
    t=-1
    while True:

      t+=1

      if t<T:
        if t+1 == (len(actions)-1):
          T= t+1


      tau = t-n+1

      if tau >= 0:
        start_idx = tau+1

        end_idx = int(np.minimum(tau+n, T) + 1)
        G = sum([x*gamma**(i-tau-1) for i,x in zip(range(start_idx,end_idx),␣
↪rewards[start_idx:end_idx])])
        state_tau = states[tau]
        steps[state_tau] += 1
```

```
        if (tau+n) < T:
          state_tau_n = states[tau+n]
          G += (gamma**n)*V[state_tau_n]
        V[state_tau] += (1/steps[state_tau])*(G-V[state_tau])
        if state_tau in [5,12]:
          V[state_tau] = -100
        if state_tau in [15]:
          V[state_tau] = 0
      if tau == (T-1) :
        repeat = episode - e
        # print(V)
        Vs[:,e:] = np.repeat(V[:,np.newaxis],repeat,axis=1)

        break

  return Vs
```

[ ]:
```
Vs = Modified_nStep_TD(random_policy, n=100, episode=10000)
```

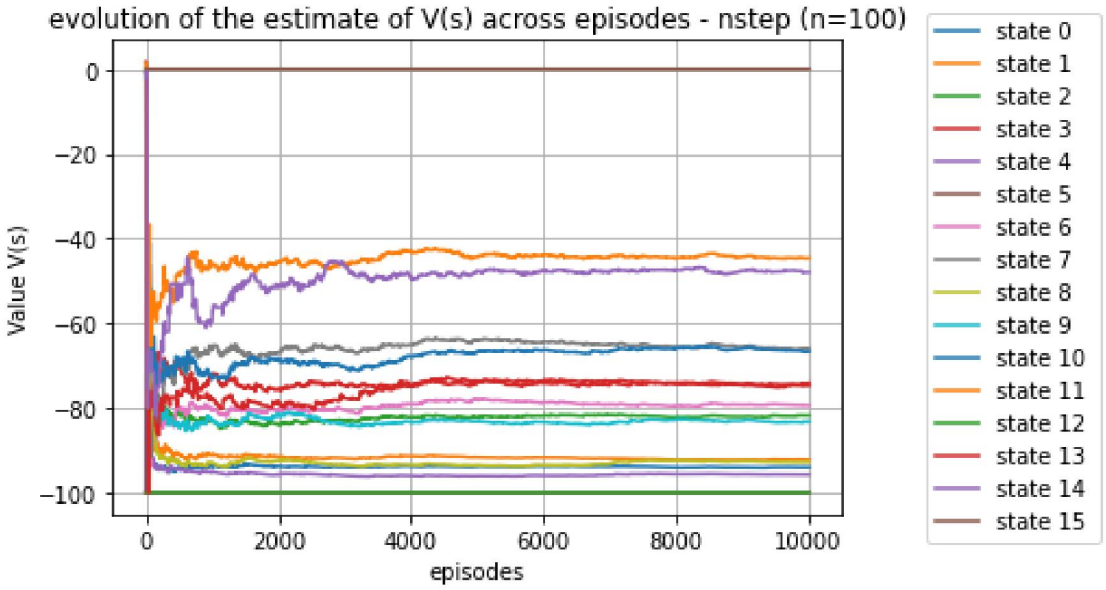
```
100%|      | 10000/10000 [00:12<00:00, 776.45it/s]
```

[ ]:
```
finalV = Vs[:,-1]
print(finalV.reshape((4,4)))
```

```
[[ -93.97834322  -92.20419424  -81.77352372  -74.08714673]
 [ -95.74920641 -100.          -79.36430985  -65.9436671 ]
 [ -92.78941345  -83.21463435  -66.61230493  -44.58663654]
 [-100.          -74.69936989  -47.96032703    0.        ]]
```

[ ]:
```
x = np.arange(10000)
plt.figure()
plt.grid()


for i in range(env.observation_space.n):
  #plt.figure()
  plt.plot(x,Vs[i,:],'-', label='state %s ' % i)

plt.legend(loc='center left', bbox_to_anchor=(1.1, 0.5))
plt.xlabel('episodes')
plt.ylabel('Value V(s)')
plt.title('evolution of the estimate of V(s) across episodes - nstep (n=100)')
```

[ ]:
```
Text(0.5, 1.0, 'evolution of the estimate of V(s) across episodes - nstep
(n=100)')
```



The plot do indeed matches the plot of the Every Visit MC prediction algorithm obtained for Q2.1. This is because we modified the learning rate such as the 2 methods are equivalent

3 Exercise 3

3.1 Question 1

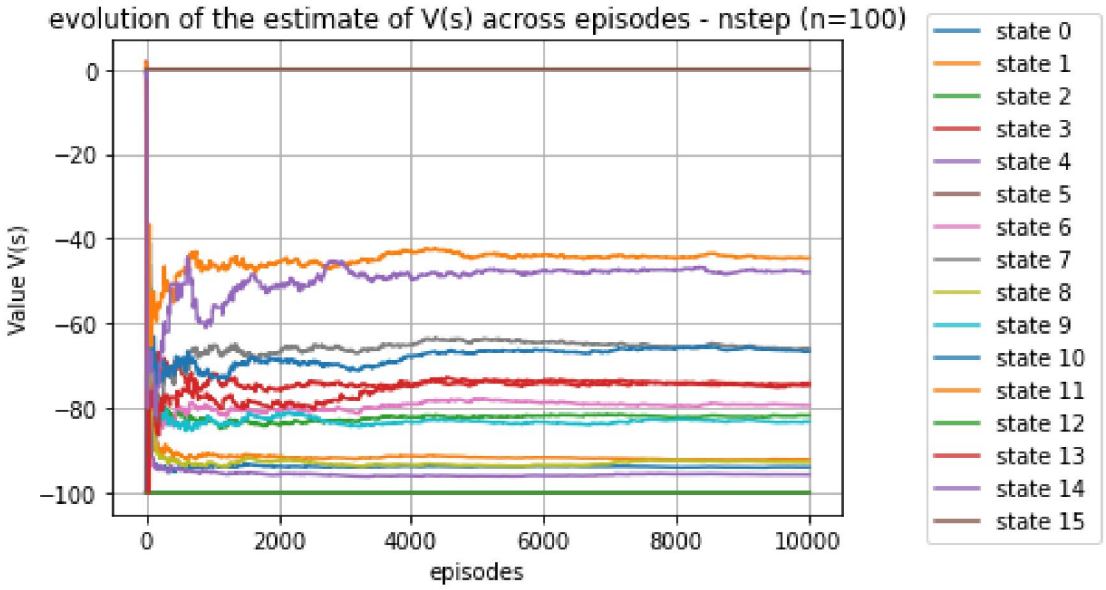
[ ]:
```
env = frozen_lake.FrozenLakeEnv(map_name="4x4", slip_rate=0.1)
gamma = 0.99
```

[ ]:
```
def choose_action(Q, epsilon):
  p = np.random.random()
  if p < epsilon:
    action = np.random.choice(env.action_space.n)
  else:
    action = np.argmax(Q)
  return action
```

[ ]:
```
def Sarsa(episode=20000, alpha=0.1, epsilon=0.01, gamma=0.99):
  Q = np.zeros((env.observation_space.n, env.action_space.n))

  undiscounted_reward = []
  for e in trange(episode):

    curr_state = env.reset()

    done = False
```

2.5 Q2.5 6 / 6

✓ - 0 pts Correct



The plot do indeed matches the plot of the Every Visit MC prediction algorithm obtained for Q2.1. This is because we modified the learning rate such as the 2 methods are equivalent

3 Exercise 3

3.1 Question 1

[ ]:
```
env = frozen_lake.FrozenLakeEnv(map_name="4x4", slip_rate=0.1)
gamma = 0.99
```

[ ]:
```
def choose_action(Q, epsilon):
  p = np.random.random()
  if p < epsilon:
    action = np.random.choice(env.action_space.n)
  else:
    action = np.argmax(Q)
  return action
```

[ ]:
```
def Sarsa(episode=20000, alpha=0.1, epsilon=0.01, gamma=0.99):
  Q = np.zeros((env.observation_space.n, env.action_space.n))

  undiscounted_reward = []
  for e in trange(episode):

    curr_state = env.reset()

    done = False
```

```
    r = 0
    while not(done):
      action = choose_action(Q[curr_state,:], epsilon)
      next_state, reward, done, extra = env.step(action)
      r += reward
      next_action = choose_action(Q[next_state,:], epsilon)
      if not(done):
        Q[curr_state,action] += alpha*(reward␣
↪+gamma*(Q[next_state,next_action]) - Q[curr_state,action])
      if done:
        Q[curr_state,action] += alpha*(reward - Q[curr_state,action])
        break

      action = next_action
      curr_state = next_state
    undiscounted_reward.append(r)



  return Q, undiscounted_reward
```

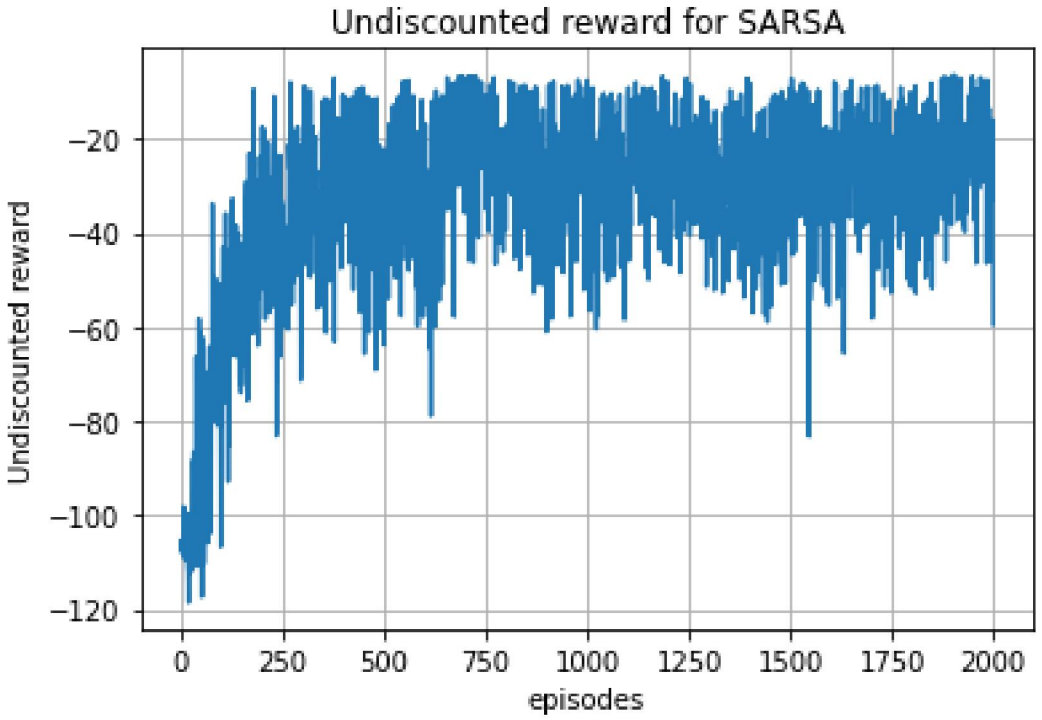
[ ]:
```
policies = []
Returns = []
for _ in range(10):
  seed = np.random.randint(np.random.randint(1000000))
  np.random.seed(seed)
  env.seed(seed)
  Q, r = Sarsa(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99)
  policies.append(Q)
  Returns.append(r)
```

```
100%|      | 2000/2000 [00:00<00:00, 3285.30it/s]
100%|      | 2000/2000 [00:00<00:00, 3384.46it/s]
100%|      | 2000/2000 [00:00<00:00, 3037.93it/s]
100%|      | 2000/2000 [00:00<00:00, 3146.41it/s]
100%|      | 2000/2000 [00:00<00:00, 3201.39it/s]
100%|      | 2000/2000 [00:00<00:00, 3566.74it/s]
100%|      | 2000/2000 [00:00<00:00, 2984.77it/s]
100%|      | 2000/2000 [00:00<00:00, 3004.10it/s]
100%|      | 2000/2000 [00:00<00:00, 3267.11it/s]
100%|      | 2000/2000 [00:00<00:00, 3427.18it/s]
```

[ ]:
```
plt.figure()
plt.grid()
y = np.mean(np.array(Returns),axis=0)
x = np.arange(len(y))
plt.plot(x,y)
```

```
plt.xlabel('episodes')
plt.ylabel('Undiscounted reward')
plt.title('Undiscounted reward for SARSA')
```

[ ]: Text(0.5, 1.0, 'Undiscounted reward for SARSA')



[ ]:
```
k = np.random.choice(len(policies))
chosen_policy = policies[k]
curr_state = env.reset()
env.render()
epsilon = 0.00
action = choose_action(Q[curr_state,:], epsilon)
done = False
i = 0
tot_reward = 0
while not(done):
  i += 1
  next_state, reward, done, extra = env.step(action)
  tot_reward += reward
  env.render()
  action = choose_action(Q[next_state,:], epsilon)
```

```
print('Using the {}th policy, the episode lasted {} timesteps and got a total␣
 ↪reward of {}'.format(k+1,i+1, tot_reward ))
```

```
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
Using the 6th policy, the episode lasted 8 timesteps and got a total reward of
-6
```

3.1 Q3.1 3 / 3

✓ - 0 pts Correct

3.2 Question 2

[ ]:
```
def compute_policy(Q, epsilon):
  policy = np.empty_like(Q)
  for i in range(policy.shape[0]):
    a_star = np.argmax(Q[i,:])
    mask = np.ones(policy.shape[-1])
    mask[a_star] = 0
    policy[i,:] = mask*epsilon/policy.shape[-1] +(1-mask)*(1-epsilon + epsilon/
↪policy.shape[-1])
  return policy
```

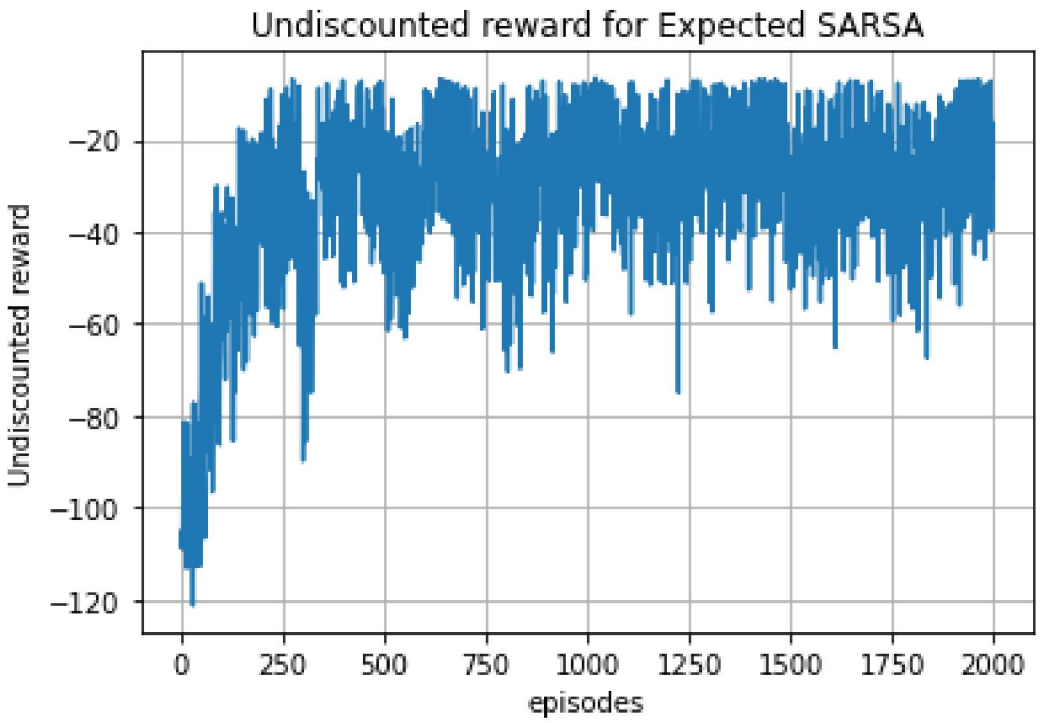
[ ]:
```
def ExpectedSarsa(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99):
  Q = np.zeros((env.observation_space.n, env.action_space.n))
  undiscounted_reward = []
  for e in trange(episode):

    curr_state = env.reset()

    done = False
    r = 0
    while not(done):
      action = choose_action(Q[curr_state,:], epsilon)
      next_state, reward, done, extra = env.step(action)
      r += reward

      policy = compute_policy(Q, epsilon)
      expectation = np.sum(policy[next_state,:]*Q[next_state,:])


      if done ==True:
        Q[curr_state,action] += alpha*(reward - Q[curr_state,action])
      else:
        Q[curr_state,action] += alpha*(reward +gamma*expectation -␣
↪Q[curr_state,action])



      curr_state = next_state
      if done==True:
        break
    undiscounted_reward.append(r)



  return Q, undiscounted_reward
```

[ ]:
```
policies = []
Returns = []
for _ in range(10):
  seed = np.random.randint(np.random.randint(1000000))
  np.random.seed(seed)
  env.seed(seed)
  Q, r = ExpectedSarsa(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99)
  policies.append(Q)
  Returns.append(r)
```

```
100%|      | 2000/2000 [00:06<00:00, 296.00it/s]
100%|      | 2000/2000 [00:06<00:00, 300.33it/s]
100%|      | 2000/2000 [00:06<00:00, 325.05it/s]
100%|      | 2000/2000 [00:06<00:00, 321.90it/s]
100%|      | 2000/2000 [00:06<00:00, 291.17it/s]
100%|      | 2000/2000 [00:07<00:00, 268.20it/s]
100%|      | 2000/2000 [00:07<00:00, 269.51it/s]
100%|      | 2000/2000 [00:05<00:00, 334.43it/s]
100%|      | 2000/2000 [00:05<00:00, 333.81it/s]
100%|      | 2000/2000 [00:07<00:00, 277.89it/s]
```

[ ]:
```
plt.figure()
plt.grid()
y = np.mean(np.array(Returns),axis=0)
x = np.arange(len(y))
plt.plot(x,y)
plt.xlabel('episodes')
plt.ylabel('Undiscounted reward')
plt.title('Undiscounted reward for Expected SARSA')
```

[ ]:
```
Text(0.5, 1.0, 'Undiscounted reward for Expected SARSA')
```



[ ]:
```
k = np.random.choice(len(policies))
chosen_policy = policies[k]
curr_state = env.reset()
env.render()
epsilon = 0.00
action = choose_action(Q[curr_state,:], epsilon)
done = False
i = 0
tot_reward = 0
while not(done):
  i += 1
  next_state, reward, done, extra = env.step(action)
  tot_reward += reward
  env.render()
  action = choose_action(Q[next_state,:], epsilon)

print('Using the {}th policy, the episode lasted {} timesteps and got a total␣
 ↪reward of {}'.format(k+1,i+1, tot_reward ))
```

```
SFFF
FHFH
FFFH
HFFG
```

```
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
Using the 3th policy, the episode lasted 8 timesteps and got a total reward of
-6
```

3.3 Question 3

[ ]:
```python
def QLearning(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99):
  Q = np.zeros((env.observation_space.n, env.action_space.n))
  undiscounted_reward = []
  for e in trange(episode):

    curr_state = env.reset()
```

3.2 Q3.2 3 / 3

✓ - 0 pts Correct

```
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
Using the 3th policy, the episode lasted 8 timesteps and got a total reward of
-6
```

3.3 Question 3

[ ]:
```python
def QLearning(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99):
  Q = np.zeros((env.observation_space.n, env.action_space.n))
  undiscounted_reward = []
  for e in trange(episode):

    curr_state = env.reset()
```

```
    done = False
    r = 0
    while not(done):
      action = choose_action(Q[curr_state,:], epsilon)
      next_state, reward, done, extra = env.step(action)
      r += reward

      maximum = np.max(Q[next_state,:])
      if done ==True:
        Q[curr_state,action] += alpha*(reward - Q[curr_state,action])
      else:
        Q[curr_state,action] += alpha*(reward +gamma*maximum -␣
↪Q[curr_state,action])
      #action = next_action
      curr_state = next_state
      if done==True:
        break

    undiscounted_reward.append(r)



  return Q, undiscounted_reward
```

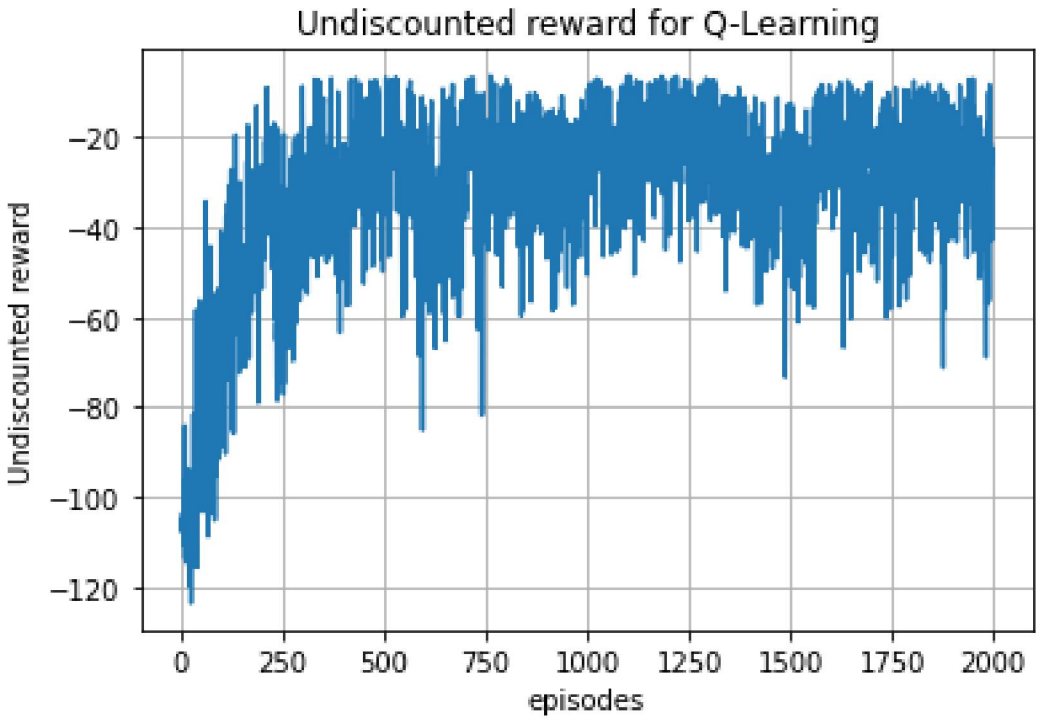
[ ]:
```
policies = []
Returns = []
for _ in range(10):
  seed = np.random.randint(np.random.randint(1000000))
  np.random.seed(seed)
  Q, r = QLearning(episode=2000, alpha=0.1, epsilon=0.01, gamma=0.99)
  policies.append(Q)
  Returns.append(r)
```

```
100%|      | 2000/2000 [00:01<00:00, 1648.56it/s]
100%|      | 2000/2000 [00:00<00:00, 2685.46it/s]
100%|      | 2000/2000 [00:00<00:00, 2380.81it/s]
100%|      | 2000/2000 [00:00<00:00, 2226.64it/s]
100%|      | 2000/2000 [00:00<00:00, 2318.51it/s]
100%|      | 2000/2000 [00:00<00:00, 2587.97it/s]
100%|      | 2000/2000 [00:00<00:00, 2558.46it/s]
100%|      | 2000/2000 [00:00<00:00, 2381.44it/s]
100%|      | 2000/2000 [00:00<00:00, 2010.72it/s]
100%|      | 2000/2000 [00:00<00:00, 2324.32it/s]
```

[ ]:
```
plt.figure()
plt.grid()
y = np.mean(np.array(Returns),axis=0)
```

```
x = np.arange(len(y))
plt.plot(x,y)
plt.xlabel('episodes')
plt.ylabel('Undiscounted reward')
plt.title('Undiscounted reward for Q-Learning')
```

[ ]:
```
Text(0.5, 1.0, 'Undiscounted reward for Q-Learning')
```



[ ]:
```
k = np.random.choice(len(policies))
chosen_policy = policies[k]
curr_state = env.reset()
env.render()
epsilon = 0.01
action = choose_action(Q[curr_state,:], epsilon)
done = False
i = 0
tot_reward = 0
while not(done):
  i += 1
  next_state, reward, done, extra = env.step(action)
  tot_reward += reward
  env.render()
  action = choose_action(Q[next_state,:], epsilon)
```

```
print('Using the {}th policy, the episode lasted {} timesteps and got a total␣
 ↪reward of {}'.format(k+1,i, tot_reward ))
```

```
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
```

```
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
Using the 10th policy, the episode lasted 11 timesteps and got a total reward of
-10
```

3.4 Question 4

If we compare the plots obtained in Q1 vs the ones we just got, we can see that the results from MC and TD methods differ. In fact, we realize that the MC methods converge slower than the TD methods. Theoretically, TD methods are biased because of teh bootsrapping process. However, in practice we can see that they are better evn though the bias can be bigger in some algorithms like in Q-learning(slightly lower than Sarsa). MC methods do not have a bias but suffer fromhigh variance, therefore we need more sample to do the learning, explaining why it converges in a slower manner

[ ]:

3.3 Q3.3 3 / 3

✓ - 0 pts Correct

```
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
Using the 10th policy, the episode lasted 11 timesteps and got a total reward of
-10
```

3.4 Question 4

If we compare the plots obtained in Q1 vs the ones we just got, we can see that the results from MC and TD methods differ. In fact, we realize that the MC methods converge slower than the TD methods. Theoretically, TD methods are biased because of teh bootsrapping process. However, in practice we can see that they are better evn though the bias can be bigger in some algorithms like in Q-learning(slightly lower than Sarsa). MC methods do not have a bias but suffer fromhigh variance, therefore we need more sample to do the learning, explaining why it converges in a slower manner

[ ]:

3.4 Q3.4 3 / 3

✓ - 0 pts Correct