Chris Fenton
CNC - AI
AI Final Exam

1. For each of the update rules, write the corresponding algorithm, e.g. TD(0), Sarsa,

a) $V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$ **TD(0)**

b) $Q(S_t,A_t) = Q(S_t,A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_t,a) - Q(S_t,A_t))$ **Q-Learning**

Note: I think $\boldsymbol{\gamma \max_a Q(S_t,a)}$ should be $\boldsymbol{\gamma \max_a Q(S_{t+1},a)}$

c) $Q(S_t,A_t) = Q(S_t,A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t))$ **Sarsa**

2. In learning, a larger step size α usually means:

a) more rapidly approaching the final performance level
b) greater error before reaching the final performance level
c) less residual error a the final performance level
d) reduced risk of divergence.
e) both a) and d)

The step size (or α) adjusts for how much you value recent information. With an α of 0, you throw away the error (or new information). With an α of 1, you give new information full value. By increasing the step size, you will converge to the optimal value faster; however, increasing the step size increases the risk that you will diverge from the optimal value.

I think using an analogy of what happens when you average average values is helpful. If you don't weight the averages, you won't get the true average by averaging two average values alone. $AVG_1(2,2,2,2,2,2,2,2,2,2) = 2$, $AVG_2(5,5) = 5$, $AVG_{bad}(2,5) = 3.5$, $AVG_{good}(2,2,2,2,2,2,2,2,2,2,5,5) = 2.5$. By averaging the two averages, you overestimate the true average.

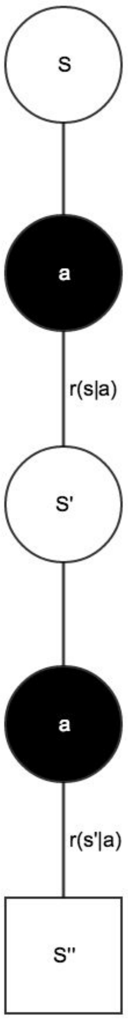
When doing iterative updates, as you get closer to convergence, the learning rate is having a smaller effect on the total estimate; however, if the learning rate is too large, you might overweight new information.

3. Write the equation defining $v_\pi(s)$ in terms of the subsequent rewards R_{t+1}, R_{t+2}, ... that would follow if the MDP were in state s at time t. If you write it in terms of G_t, define your return notation in terms of the underlying rewards.

With a discount rate of 1:
$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots R_T$, where R_T = the reward at the terminal state. Ie, the sum of of all the rewards in the chain.

The value of a policy given a state, $v_\pi(s)$, is the expected value of G_t given that state.

4. Draw the backup diagram for the dynamic programming algorithm (full backup) for v_π. Use labels s, a, r, and s'.



5. Write the Bellman equation for v_π in terms of p(s',r | s,a).

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} \pi(a|s) p(s',r|s,a)[r + \gamma v_k(s')]$$

6. Answer exercise 1.4, in terms of account off-policy learning.

Suppose learning updates occurred after all moves, including exploratory moves. If the step-size parameter is appropriately reduced over time (but not the tendency to explore), then the state values would converge to a set of probabilities. What are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins?

In q-learning, you build a mapping of state/action rewards. It is the policy that returns a set of probabilities for the actions (epsilon-greedy for example). In q-learning, if you don't learn from exploratory moves ($\alpha = 0$), you won't update the state/action rewards. In epsilon-greedy, if you don't explore ($\varepsilon = 0$), you will always choose the same action given a state. Learning from exploratory moves will result in more wins.

7. What is the difference between first-visit and every-visit Monte Carlo?

First-visit MC gives the average returns the first time a state is visited in an episode; whereas, every-visit MC gives the average returns for every time a state is visited in an episode.

8. For the 4x4 gridworld in Example 4.1, there are 14 non-terminal states and 2 terminal states. The reward is -1 for a transition from a non-terminal state. The only actions are NEWS. Assuming the equi-probable policy, for dynamic programming,

a) what is the value of q_π(11, down)?

0

b) write the equation for the value of q_π(7, down)?

$$q_\pi = \sum_{7,S} 0.25[-1 + \gamma v_\pi(11)].$$

9. Suppose you have been in a country where the disease kuru is found. The proportion of the general population with this disease is 0.000007. If you don't have the disease, the probability of a false positive for the test is 0.00035. The probability of a false negative is is very low (less than 10-10). If you are part of the general population and test positive, what is the likelihood that you have kuru?

Using Baye's theorem, we know that

$$P\left(A\right|B=\frac{\left(\left(P\left(B\right|A\right)P\left(A\right)\right)}{P\left(B\right)}$$

Here's what we know from the problem:

$$P(A)=0.000007, P(\neg A)=0.999993$$
$$P(B|\neg A)=0.00035$$
$$P(B|A)\approx 1$$
$$P(\neg B|A)\approx 0$$

To solve P(A|B) we need to find P(B):

$$P(B)=P(B|A)P(A)+P(B|\neg A)P(\neg A)$$

$$P(B)=1\times 0.000007+0.00035\times 0.999993$$
$$P(B)=0.000357$$

Now we can solve P(A|B):

$$P(A|B)=(1\times 0.000007)/0.000357$$
$$P(A|B)=0.019608$$

If you're a part of the general population and test positive, you still only have an almost 2% chance of actually having the disease.

10. Suppose you have a policy π, and its action-value function q_π. Suppose you define a deterministic policy π' by the equation:

$\pi'(s) = \text{argmax}_a\, q_\pi(s,a)$ for all s

What can you say about the relationship between π(s) and π'(s)?

The value of the new policy give state S is at least as good as the previous policy:

$$v_{\pi'}(s) \geq v_\pi(s)$$

11. Programming questions attached in Python files

a) For the gridworld in example 4.1, Figure 4.1 shows a *synchronous* iterative policy evaluation, although the text explains asynchronous. An *asynchronous* iterative policy evaluation would go through the states (in numerical order from 1 to 14) and update after each state based on the previous updates. Program the asynchronous version, and write the value for each state after 2000 iterations.

```python
def asyncPolicyEvaluation(gamma=0.9,iterations=2000):
    # ENV.P[s][a] : prob, next_state, reward, terminal?) tuple
    # Actions: up=0, right=1, down=2, left=3
    Q = defaultdict(lambda: np.zeros(ENV.action_space.n))
    # Q is the optimal action-value function, a dictionary mapping state ->
action values.
    A = [0,1,2,3] #Actions
    values = [0] * ENV.nS
    for i in range(iterations):
        state = ENV.reset() #get a random state
        action = np.random.choice(A, replace=False) #random action
        #Q(s,a) = EV[R(t+1) + γV(s')]
        for prob, next_state, reward, done in ENV.P[state][action]:
            Q[state][action] = reward + gamma * np.max(Q[next_state])
    # For each state, get the optimal action
    for state in Q:
        values[state] = np.average(Q[state])
    # return the value of each state after 'i' iterations
    return np.round(values, decimals=1)
```

0	-2.1	-2.9	-3.1
-2.1	-2.7	-2.7	-2.9
-2.9	-2.7	-2.7	-2.1
-3.1	-2.9	-2.1	0

b) Program gridworld using a Monte Carlo off-policy solution. (submit on ada). Write the values for the state-action pairs, write the equations, including importance sampling.

```python
def offPolicyMC(gamma=1,episodes=50000):
    def newGreedyPolicy(q):
        def policy(state):
            actions = np.zeros_like(Q[state], dtype=float)
            a = np.argmax(q[state])
            actions[a] = 1.0
            return actions
        return policy
    def newSoftPolicy(actions): #random policy
        actions = np.ones(actions, dtype=float) / actions
        def policy(obs):
            return actions
        return policy

    Q = defaultdict(lambda: np.zeros(ENV.action_space.n))
    C = defaultdict(lambda: np.zeros(ENV.action_space.n))
    PI = newGreedyPolicy(Q)
    softPolicy = newSoftPolicy(ENV.action_space.n)

    for ep in range(1, episodes + 1):
        # Generate an episode
        episode = []
        state = ENV.reset()
        for t in range(100):
            # Sample an action from our policy
            probs = softPolicy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, done, _ = ENV.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        G = 0.0
        W = 1.0

        # for t = T-1, T-2,...,0
        for t in range(len(episode))[::-1]:
            state, action, reward = episode[t]
            G = gamma * G + reward # G ← γG + Rt+1
            C[state][action] += W # C(St, At) ← C(St, At) + W
            # Incrementally update q(s,a) and improve target policy (PI)
            Q[state][action] += (W / C[state][action]) * (G - Q[state][action])
            # If At/= π(St) then ExitForLoop
            if action != np.argmax(PI(state)):
                break
            # Update weight
            W = W * 1.0 / softPolicy(state)[action]

    values = [0] * ENV.nS
    for state in Q:
        values[state] = np.average(Q[state])
    # return the value of each state after 'i' iterations
    return [np.round(values, decimals=1), Q]
```

Values after 100,000 episodes with gamma=0.9:

0	-1	-1.4	-1.3
-1.3	-1.6	-1.2	-1.4
-1.5	-1.1	-1.4	-1.5
-1.2	-1.3	-1.2	0

Q:

```
S     [ U,     R,     D,     L ]
0:     0.0,  0,0,  0,0,  0,0
1:    -1.0, -1.0, -1.0, -1.0
2:    -1.5, -1.9, -1.0, -1.4
3:    -1.6, -1.0, -1.2, -1.5
4:    -1.0, -1.0, -1.3, -1.9
5:    -1.7, -1.7, -1.0, -1.9
6:    -1.5, -1.3, -1.0, -1.2
7:    -1.0, -1.3, -1.9, -1.3
8:    -1.9, -1.0, -1.8, -1.5
9:    -1.0, -1.2, -1.0, -1.3
10:   -1.5, -1.9, -1.0, -1.4
11:   -1.7, -1.9, -1.0, -1.4
12:   -1.3, -1.0, -1.5, -1.0
13:   -1.3, -1.3, -1.5, -1.0
14:   -1.0, -1.0, -1.4, -1.4
15:    0.0,  0,0,  0,0,  0,0
```

c) Program gridworld using a TD(0) off-policy solution. Write the values for the state-action pairs, write the equations.

```
def offPolicyTD0(gamma=1.0, alpha=0.5, n=10000):
    V = np.zeros(ENV.nS) #initialize V(s)
    Q = defaultdict(lambda: np.zeros(ENV.action_space.n))
    PI = newEpsilonGreedyPolicy(Q, ENV.action_space.n)

    # Repeat for each episode
    for ep in range(n):
        state = ENV.reset()
        # Repeat for each step of episode
        for step in itertools.count(): #infinite loop, will break on T
            # A ← action given by π for S
            action_probs = PI(state)
            action = np.random.choice(np.arange(len(action_probs)),
p=action_probs)

            # Take action A, observe R, S′
            next_state, reward, done, _ = ENV.step(action)

            # V (S) ← V (S) + α R + γV (S′ ) − V (S)
            temp_diff = (reward + gamma * V[next_state]) - V[state]
            V[state] = V[state] + (alpha * temp_diff)
            # Update q(s,a) to make policy fit the template
            Q[state][action] = V[state]

            #S ← S′
            state = next_state

            # until S is terminal
            if done:
                break
    return [np.round(V, decimals=1), Q]
```

```
V, Q = offPolicyTD0(n=20000)

V =

0.0 -1.0 -2.3 -3.1

-1.0 -2.1 -3.1 -2.0

-2.7 -3.5 -2.0 -1.0

-3.1 -2.0 -1.0 0.0

Q =

0: 0.0 0.0 0.0 0.0
1: -2.2 -2.3 -2.0 -1.0
2: -3.1 -3.2 -3.0 -2.3
3: -3.9 -3.7 -3.1 -3.4
4: -1.0 -2.0 -2.1 -2.2
5: -2.5 -3.0 -3.2 -2.1
6: -3.6 -3.1 -3.6 -3.5
7: -3.1 -2.5 -2.0 -3.3
8: -2.7 -3.4 -3.3 -3.3
9: -3.5 -3.5 -3.6 -3.4
10: -3.0 -2.0 -2.8 -3.1
11: -2.1 -1.5 -1.0 -2.2
12: -4.4 -3.1 -4.5 -3.8
13: -3.0 -2.0 -2.5 -3.0
14: -2.1 -1.0 -2.0 -2.0
15: 0.0 0.0 0.0 0.0
```