Reinforcement Learning	April 03, 2019
Exercise 3	
Lecturer:Yishay Mansour	*TA: Lee Cohen*

Theory

Question 1

Let M be an MDP with an optimal Q-function, $Q^*_M(s,a)$ and an optimal policy π^*. Let M' be an MDP such that $Q^*_{M'}(s,a) = Q^*_M(s,a) + f(s)$.
Is π^* also an optimal policy for M'?

Question 2

Let M be an MDP with reward $R(s,a,s')$ and M' be an MDP with reward $R'(s,a,s')$.
Suppose that $R'(s,a,s') = R(s,a,s') + \Phi(s) - \gamma\Phi(s')$.
Are the optimal policies in M and M' identical?
Hint: Start with Bellman equation for $Q^*_M(s,a)$ and relate it to $Q^*_{M'}(s,a)$.

Question 3

Consider a DDP (Deterministic Decision Process) that is modeled by a directed graph, where the states are the vertices, and each action is associate with an edge with unknown reward function R and $R_{max} = 1$.

Consider the following algorithm to recover the DDP. (We assume that the DDP is strongly connected.) The algorithm works in iterations. We partition the states-action pairs to *known* and *unknown*. In iteration t we assign fictitious rewards R'_t. We assign reward 1 to each unknown state-action pair, i.e, $R'_t(s,a) = 1$ if (s,a) is unknown. We assign reward 0 to each known state-action pair, i.e, $R'_t(s,a) = 0$ if (s,a) is known. For unknown state-action pairs we define $f(s,a) = s$, i.e., we stay in state s for each state-action unknown pair. In iteration t, we compute an optimal policy π_t based on R'_t and run it until it discovers a new state-action pair. Once there are no any remaining unknown state-action pairs, we output the DDP we discovered.

1. Show that every iteration terminates. (Recall that the true DDP is strongly connected.)

2. What is the maximal time complexity of each iteration?

3. How many iterations does the algorithm performs? (until all the DDP is known, both transitions and rewards.)

4. Let $T_{exploitation}$ be the time in the DDP is fully known. Derive an upper bound for $T_{exploitation}$ as a function of $|S|$ and $|A|$.

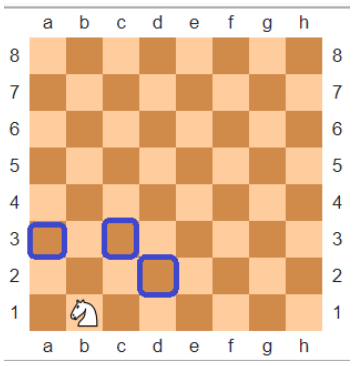
Question 4

Consider a chess board (8×8 grid) with a single knight making random moves, starting from $(b, 1)$. At every time step, the knight is equally likely to make any one of the legal moves. A legal move is to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally, assuming the such square exists on the board. In the example you can see all the legal moves from the starting point, but notice that there can be up to 8 legal moves.

1. What is the probability that the knight would return to the starting point after two time steps?

2. Is the corresponding Markov chain irreducible and/or periodic? (Recall that it is starting at $(b, 1)$.)

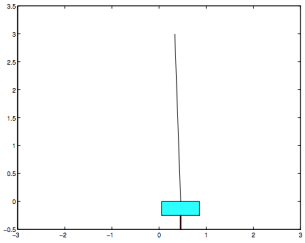
3. What is the mean recurrence time to a corner? (Hint: given that the knight is taking a really long walk on the board, what fraction of it's time will it spend on each square? It might help to think of the problem as an undirected graph with $n = 64$ states and $m = 336$ edges describing possible knight moves.)



Programming

Question 1: Off-Policy Model-Based [1]

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.



The problem we will consider is the pole-balancing problem, also known as cart-pole. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.

Attached is a simple Python simulator(cart_pole.py) for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it'd be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 1 to `NUM STATES`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no do-nothing action.) These are represented as actions 0 and 1 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards **observed**.

[1] This exercise is based on HW #4 of Standford CS 229 course

The files for this problem are cart_pole.py and control.py. Most of the the code has already been written for you, and you need to make changes only to control.py in the places specified. The files also contain the function `show_cart` to visualize the system state and code to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation

1. To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

   Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

   During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

   Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

   Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO LEARNING THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly. The code outline for this problem is already in control.py, and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

   Implement the reinforcement learning algorithm as specified, and run it. How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged?

2. Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Python starter code already includes the code to plot. Submit the plot.

Question 2: Q-Learning

Recall FrozenLake environment from previous HW #2. It consists of a 4×4 grid of blocks, each one either being the start block, the goal block, a safe frozen block, or a dangerous hole. The objective is to have an agent learn to navigate from the start to the goal without moving onto a hole. At any given time the agent can choose to move either up, down, left, or right.

The catch is that there is a wind which occasionally blows the agent onto a space they didn't choose. As such, perfect performance every time is impossible, but learning to avoid the holes and reach the goal are certainly still doable. The reward at every step is 0, except for entering the goal, which provides a reward of 1. Thus, we will need an algorithm that learns long-term expected rewards - Q-Learning.

You may have heard about DeepQ-Networks(DQN) which can play Atari Games (`https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf`). These are really just larger and more complex implementations of the Q-Learning algorithm you will implement here.

1. Implement tabular Q-Learning i.e. learn a table of values for every state (row) and action (column) possible in the environment. Implement the missing code parts in the attached `tabular_Q.py` (marked by `#TODO` comments). Report the "percent of successful episodes" and the final Q table.

2. Tables are great, but they don't really scale. While it is easy to have a 16×4 table for a simple grid world, the number of possible states in any real-world environment is nearly infinitely larger. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. The neural network will learn to map states which are represented by vector to Q-values.

   Implement a one-layer network which takes the state encoded in a one-hot vector (1x16), and produces a vector of 4 Q-values, one for each action. Use the attached `network_Q.py` and fill in the missing code segments (marked by `#TODO` comments). Note that in this case the method of updating is different. Instead of directly updating our table, with a network we will be using backpropagation and a loss function. The following depicts a single update:

   (a) $Q = f(s)$ and $a = argmax(Q)$, where $Q \in \mathbb{R}^4$

   (b) Play action a from state s: move to step $s' = P(s, a)$ with reward $r = R(s, a)$.

   (c) $Q' = f(s')$

   (d) $Q_{target}[i] = \begin{cases} Q[i] & \text{if}_i \neq a \\ r + \gamma * max(Q') & \text{else} \end{cases}$

   (e) Update network f with regard to the loss $||Q_{target} - Q||^2$

   Report the "percent of successful episodes". Is it better than the tabular method?

Submit your modified code, the requested outputs and plots.