1 MDP Basics and Dynamic Programming Methods

Topics in Machine Learning (CS975) — **IIIT Hyderabad**
Instructor: Dr. Naresh Manwani — Due: **4 September 2019 11:59pm, on Moodle**

Please use the moodle discussion thread for this assignment to communicate any queries or doubts.

In this assignment, you will have hands-on experience with creating toy environments and map them to the Markov Decision Process (MDP) based model discussed in lectures. Using dynamic programming and bootstrapping, you will implement agents which learn to navigate the environment to maximize return.

Grading will be out of **100 points + 50 bonus points**. Bonus entries are indicated inline with the question and are optional.

1.1 Problem Set

1. **GridWorld [20 points]** This is a guided walkthrough to build a widely used RL Environment and train agents. Try to stick to the boilerplate code in Python interspersed within this question.

```
env = DiscreteEnvironment(...)
agent = DiscreteAgent(env)

start_state = env.reset()
is_terminated = False

while not is_terminated:
    action = agent.get_action(state)
    next_state, reward, is_terminated = env.step(action)
    # ...
    state = next_state
```

With the above in mind, you will go on to define the required components (Environment, Agents) completing the below boilerplate.

(a) **Simulating the environment [5 points]** Create your own version of `GridWorld` of size 8×8 with randomly chosen start, end and the rewards being deterministic. Try to model doing as much as justice to the MDP Tuple (Keep explicit S, A, R, P and use these to `.step(action)`. This will help you to reuse this code further below.) Your `GridWorld` may adhere to the following structure.

```
from abc import ABC, abstractmethod

class DiscreteEnvironment(ABC):
    @abstractmethod
    def step(self, action): pass

    @abstractmethod
    def reset(self): pass
```

(b) **Train the agent: Policy and Value Iteration [10 points]** Define agent classes possibly adhering to the following structure which takes in an environment. Use the `.update()` function to make an iteration step in Policy and Value Iteration.

```
from abc import ABC, abstractmethod

class DiscreteAgent(ABC):
    @abstractmethod
    def __init__(self, env): pass
    @abstractmethod
    def update(self): pass
    @abstractmethod
    def get_action(self, state): pass
```

Using the above, you can define the training loop as follows to terminate either based on number of iterations or the difference threshold.

```
env = DiscreteEnvironment(...)
agent = DiscreteAgent(env)

sweep_no, is_converged = 0, False
max_sweeps = 100

# Constrain Iterations
while sweep_no < max_sweeps:
    sweep_no, is_converged = agent.update()

# Constrain based on a threshold parameter
while not is_converged:
    sweep_no, is_converged = agent.update()
```

Create two agents which provides a policy using dynamic programming methods and bootstrapping (`DPAgent: PolicyIteration, ValueIteration`). For `PolicyIteration` break down your agent's update function into `evaluate_policy` and `update_policy` functions. Also, create a `ConfusedAgent`, which randomly picks an action available from a given state (No need to train this one).

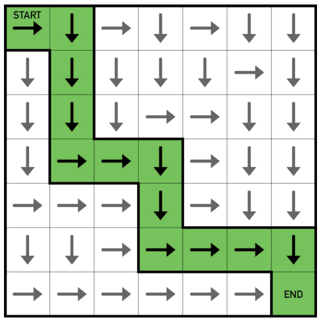
```
class PolicyIteration(DiscreteAgent):
    def evaluate_policy(self): pass
    def update_policy(self): pass
```

```
class ValueIteration(DiscreteAgent):
    pass
```

(c) **Test-run and visualizing learning [5 Points]** Use the skeleton code given for the interaction between Agent and Environment at the start.

(i) Compare the agents using the return obtained v/s training iterations. Indicate mean, min and max return over `N` runs in the plot using confidence intervals (matplotlib example). Are the learned agents better than `ConfusedAgent` (on average? always?)?

(ii) **[+10 Bonus Points]** Plot a matrix indicating the value function using colors (matplotlib: matshow) and the policy using arrows (matplotlib: arrow) as in the figure given below for `DPAgents`.



(iii) **[+10 Bonus Points]** Compare differences in paths obtained by setting $\gamma = 0, 0.1, 0.5, 0.75, 1$ while learning. How does γ affect the final path (or the policy learnt)?

2. **Jack's Car Rental [40 points]** *(Sutton and Barto Example 4.4 - Policy Iteration)*

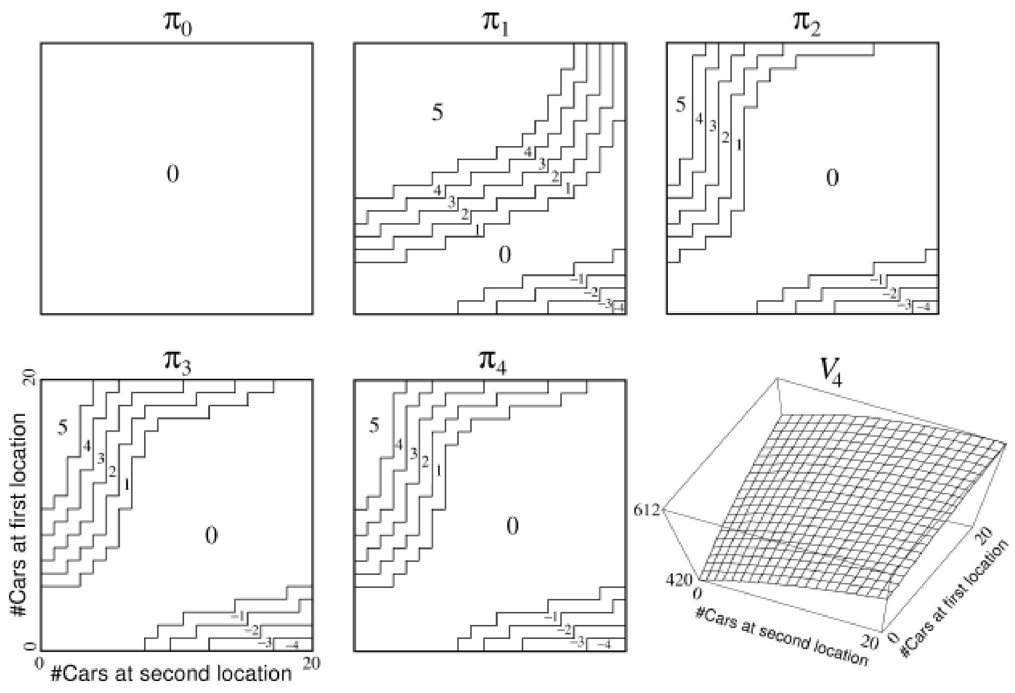


Figure 1: Jack's Car rental

Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is:

$$P(n) = \frac{\lambda^n}{n!}e^{-\lambda}$$

where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 1 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

(a) **[15 Points]** Describe the states, actions, rewards and transitions in detail. Using the notations used in your description, show the expressions for the Bellman Update forming the policy iteration.

(b) **[25 Points]** Solve for π using Policy Iteration. Reproduce the plots in Figure 1 for the problem.

(c) **[+20 Bonus Points]** One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of non-linearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. Solve the problem incorporating the new complications using Policy Iteration.

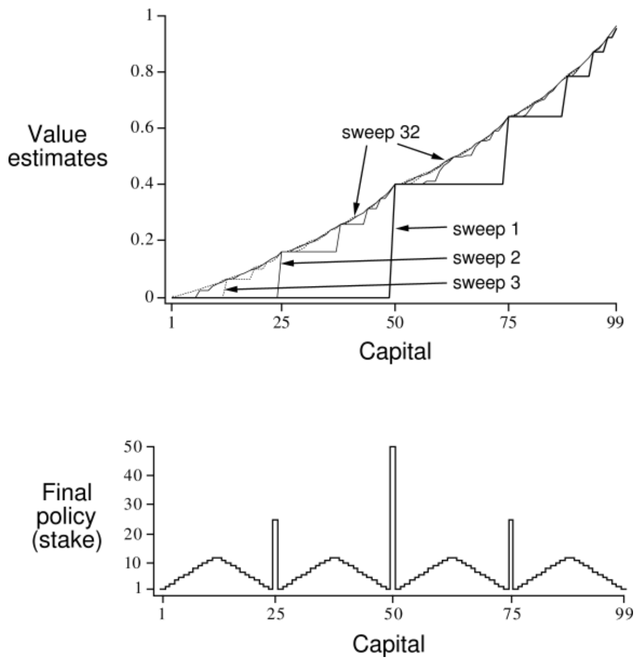


Figure 2: Gambler's Problem

3. **Gambler's Problem [40 points]** *(Sutton and Barto Exercise 4.8 - Value Iteration)* A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads,

he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital $s \in \{1, 2, \dots 99\}$, and the actions are stakes, $a \in \{0, 1, \dots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is $+1$. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p denote the probability of the coin coming up heads. If p is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 2 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p = 0.3$.

(a) **[15 Points]** Describe the states, actions, rewards and transitions in detail. Using the notations used in your description, show the expressions for the Bellman Update forming the value iteration.

(b) **[25 Points]** Solve the above for $p = 0.15$ and $p = 0.65$. You may find it convenient to introduce two dummy states corresponding to the termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Reproduce the plots in Figure 2 (Match the axes, not necessarily the plot).

(c) **[+10 Bonus points]** Are your results stable as $\theta \to 0$, where θ is the convergence threshold parameter in Value Iteration? Illustrate.

1.2 Submission Instructions

Your submissions are expected to be a `.tar.gz` [1] file containing the following:

- `main.ipynb` - A Jupyter notebook[2] with code and explanations written as a walkthrough for the code. Any proofs and derivations can also be embedded in the notebook using MathJax + Markdown.
- `main.pdf` generated from the above notebook.
- Auxilliary python scripts (plotting, helpers for loading etc, abstract classes which need not appear in the notebook [`DiscreteAgent,DiscreteEnvironemnt`]) which you import and reuse in the notebook.

The submissions are meant to be evaluated offline. In the event your code fails to reproduce in the TA's machine, you will be summoned for manual evaluation during TA hours. It is advised you tryout Google Colaboratory to ensure the code is reproducible.

Collaboration Policy You can indicate your collaborations and avail marks $2x/3$ if you score x as a team. Groups of maximum 2 members are allowed. Extra content in your report (notebook) on how you divided work **AND** `.git` history supporting the claims are required here.

Plagiarism Policy Plagiarism detection software is guaranteed to be run before any evaluation. Trying to beat any such software will make your code significantly unreadable and easy to prove malicious intent. In case of heavy plagiarism - all parties involved (giver, taker) will get a 0.

[1] (**not** .zip, .gz, .tgz, etc.)
[2] Make sure this is after a restart and run-all option, so a lot of size doesn't go up in notebook state.