19 Policy Function Iteration

Lab Objective: *Iterative methods can be powerful ways to solve dynamic optimization problems without computing the exact solution. Often we can iterate very quickly to the true solution, or at least within some ε error of the solution. These methods are significantly faster than computing the exact solution using dynamic programming. We demonstrate two iterative methods, value iteration (VI) and policy iteration (PI), and use them to solve a deterministic Markov decision process.*

Dynamic Optimization

Many dynamic optimization problems take the form of a *Markov decision process*. A Markov decision process is similar to that of a Markov chain, but rather than determining state movement using only probabilities, state movement is determined based on probabilities, actions, and rewards. They are formulated as follows.

$\mathbb{T}$ is a set of discrete time periods. In this lab, $\mathbb{T} = 0, 1, \dots, L$, where L is the final time period. S is the set of possible states. The set of allowable actions for each state s is A_s. $s_{t+1} = g(s_t, a_t)$ is a transition function that determines the state s_{t+1} at time $t+1$ based on the previous state s_t and action a_t. The reward $u(s_t, a_t, s_{t+1})$ is the reward for taking action a while in state s at time t and the next state being state s_{t+1}.

The time discount factor $\beta \in [0, 1]$ determines how much the reward function decreases in value with time. Looking at the cake eating problem described in the previous lab, the cake goes stale with each passing day. In other words, each day you do not finish the cake, the reward of eating a piece decreases. β accounts for this decrease in value.

Let $N_{s,a}$ be the set of all possible next states when taking action a in state s. $p(s_t, a_t, s_{t+1})$ is the probability of taking action a at time t while in state s and arriving at state $s_{t+1} \in N_{s,a}$. A deterministic Markov process has $p(s_t, a_t, s_{t+1}) = 1 \ \forall s, a$. This means that $N_{s,a}$ has one element $\forall s, a$. A stochastic Markov process has $p(s_t, a_t, s_{t+1}) \leq 1$, which implies that there can be multiple possible next states for taking a given action in a given state. As a result, $N_{s,a}$ has multiple elements for each s, a.

The dynamic optimization problem is

$$\max_{\mathbf{a}} \sum_{t=0}^{L} \beta^t u(s_t, a_t) \tag{19.1}$$

$$\text{subject to } s_{t+1} = g(s_t, a_t) \ \forall t. \tag{19.2}$$

The cake eating problem follows this format where S consists of the possible amounts of remaining cake ($\frac{i}{W}$), c_t is the amount of cake we can eat, and the amount of cake remaining $s_{t+1} = g(s_t, a_t)$ is $w_t - c_t$, where w_t is the amount of cake we have left and c_t is the amount of cake we eat at time t. This is an example of a deterministic Markov process.

For this lab, we define a dictionary P to represent the decision process. This dictionary contains all of the information about the states, actions, probabilities, and rewards. Each dictionary key is a state-action combination and each dictionary value is a list of tuples.

$$P[s][a] = [(p(s, a, \bar{s}), \bar{s}, u(s, a, \bar{s}), is\_terminal), ...]$$

Note the slight notation change from (s_t, a_t, s_{t+1}) to $(s, a, \bar{s})$. In the dictionary, s is the current state, a is the action, $\bar{s} \in N_{s,a}$ is the next state if action a is taken, and $is\_terminal$ indicates if $\bar{s}$ is a stopping point. In addition, $p(s, a, \bar{s})$ is the probability of taking action a while in state s, and $u(s, a, \bar{s})$ is the reward for taking action a while in state s.

Moving on a Grid

Now consider an $N \times N$ grid. Assume that a robot moves around the grid, one space at a time, until it reaches the lower right hand corner and stops. Each square is a state, $S = \{0, 1, \ldots, N^2 - 1\}$, and the set of actions is $\{Left, Down, Right, Up\}$. For this lab, $Left = 0$, $Down = 1$, $Right = 2$, and $Up = 3$. If you take the action $a = 1$, then you move $Down$ on the grid.

Let $N = 2$ and label the squares as displayed below. In this example, we define the reward to be -1 if the robot moves into 2, -1 if the robot moves into 0 from 1, and 1 when it reaches the end, 3. We define the reward function to be $u(s, a, \bar{s}) = u(\bar{s})$. Since this is a deterministic model, $p(s, a, \bar{s}) = p(\bar{s}) = 1$ for all possible s, a.

0	1
2	3

A_s is the set of actions that keep the robot on the grid. If the robot is in the top left hand corner, the only allowed actions are $Down$ and $Right$ so $A_0 = \{1, 2\}$. The transition function $g(s, a) = \bar{s}$ can be explicitly defined for each s, a where $\bar{s}$ is the new state after moving.

All of this information is encapsulated in P. We define $P[s][a]$ for all states and actions, even if they are not possible. This simplifies coding the algorithm but is not necessary.

```
P[0][0] = [(0, 0, 0, False)]    P[2][0] = [(0, 2, -1, False)]
P[0][1] = [(1, 2, -1, False)]   P[2][1] = [(0, 2, -1, False)]
P[0][2] = [(1, 1, 0, False)]    P[2][2] = [(1, 3, 1, True)]
P[0][3] = [(0, 0, 0, False)]    P[2][3] = [(1, 0, 0, False)]
P[1][0] = [(1, 0, -1, False)]   P[3][0] = [(0, 0, 0, True)]
P[1][1] = [(1, 3, 1, True)]     P[3][1] = [(0, 0, 0, True)]
P[1][2] = [(0, 0, 0, False)]    P[3][2] = [(0, 0, 0, True)]
P[1][3] = [(0, 0, 0, False)]    P[3][3] = [(0, 0, 1, True)]
```

For the sake of clarity, we will do a quick example using the above dictionary. We first assume that we start in state 0 corresponding to the 0 in the above grid. Next, we move *Down* the grid to state 2. This corresponds to taking action 1. To get the correct values from the dictionary, we look at $P[s][a]$ or in this case $P[0][1] = [(1, 2, -1, False)]$. So, when we move *Down* from square 0 to square 2, $p(\bar{s}) = 1$, $u(\bar{s}) = -1$, and $\bar{s} = 2$. As a final note, when the action is not possible $p(\bar{s}) = 0$, as shown in the dictionary above.

We define the *value function* $V(s)$ to be the maximum possible reward of starting in state s. Then using Bellman's optimality equation,

$$V(s) = \max_{a \in A_s} \{\Sigma_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V(\bar{s}))\}. \tag{19.3}$$

The summation occurs when it is a stochastic Markov process. For example, if the robot is in the top left corner and we want it to move right, we could have the probability of the robot actually moving right as .5. In this case, $P[0][2] = [(.5, 1, 0, False), (.5, 2, -1, False)]$. This type of process will occur later in the lab.

Value Iteration

In the previous lab, we used dynamic programming to solve for the value function. This was a recursive method where we calculated all possible values for each state and time period. *Value iteration* is another algorithm that solves the value function by taking an initial value function and calculating a new value function iteratively. Since we are not calculating all possible values, it is typically faster than dynamic programming.

Convergence of Value Iteration

A function f that is a contraction mapping has a *fixed point* p such that $f(p) = p$. Blackwell's contraction theorem can be used to show that Bellman's equation is a "fixed point" (it actually acts more like a fixed function in this case) for an operator $T : L^\infty(X; \mathbb{R}) \to L^\infty(X; \mathbb{R})$ where $L^\infty(X; \mathbb{R})$ is the set of all bounded functions:

$$T[f](s) = \max_{a \in A_s} \{\Sigma_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta f(\bar{s}))\} \tag{19.4}$$

It can be shown that 19.1 is the fixed "point" of our operator T. A result of contraction mappings is that there exists a unique solution to 19.4.

$$V_{k+1}(s_i) = T[V_k](s_i) = \max_{a \in A_s} \{\Sigma_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V_k(\bar{s}))\} \tag{19.5}$$

where an initial guess for $V_0(s)$ is used. As $k \to \infty$, it is guaranteed that $(V_k(s)) \to V^*(s)$. Because of the contraction mapping, if $V_{k+1}(s) = V_k(s)\ \forall\, s$, we have found the true value function, $V^*(s)$.

As an example, let $V_0 = [0, 0, 0, 0]$ and $\beta = 1$, where each entry of V_0 represents the maximum value at that state, and $V_0(s) = V_0[s]$ if we are using the array or list form of the *value function*. We calculate $V_1(s)$ from the robot example above. For $V_1(0)$, we choose the `max` of the possible outcomes, states 1 or 2, after moving. Thus we use $P[0][2]$ for state 1 because moving from state 0 to state 1 requires going right, action 2.

$$
\begin{aligned}
V_1(0) &= \max_{a\in A_0}\{\Sigma_{\bar{s}\in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + V_0(\bar{s}))\} \\
&= \max\{p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2)))\} \\
&= \max\{1(0+0), 1(-1+0)\} \\
&= \max\{0, -1\} \\
&= 0 \\
V_1(1) &= \max\{p(0) * (u(0) + V_0(0)), p(3) * (u(3) + V_0(3))\} \\
&= \max\{1(-1+0), 1(1+0)\} \\
&= 1 \\
V_1(2) &= \max\{p(0) * (u(0) + V_0(0)), p(3) * (u(3) + V_0(3))\} \\
&= \max\{1(0+0), 1(1+0)\} \\
&= 1 \\
V_1(3) &= \max\{p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2))\} \\
&= \max\{1(0+0), 1(0+0)\} \\
&= 0
\end{aligned}
$$

This calculation gives $V_1 = [0, 1, 1, 0]$. Repeating the process yields $V_2 = [1, 1, 1, 0]$. Repeating a third time gives $V_3 = [1, 1, 1, 0]$, which is the same as V_2, so the process has converged. This means that the solution is $[1, 1, 1, 0]$. Thus, the total maximum reward the robot can achieve by starting on square i is the ith entry of the solution $[1, 1, 1, 0]$.

When implementing functions in this lab, instead of only looking at possible actions $a \in A_s$, we can consider all of the actions. This will not affect the results, because $p(\bar{s}) = 0$ when an action is not possible. This simplifies the coding significantly. For example, when calculating $V_{k+1}(s_i)$ consider the following lines of code.

```
sa_vector = np.zeros(nA)
for a in range(nA):
    for tuple_info in P[s][a]:
        # tuple_info is a tuple of (probability, next state, reward, done)
        p, s_, u, _ = tuple_info
        # sums up the possible end states and rewards with given action
        sa_vector[a] += (p * (u + beta * V_old[s_]))
#add the max value to the value function
V_new[s] = np.max(sa_vector)
```

Problem 1. Write a function called `value_iteration()` that will accept a dictionary P representing the decision process, the number of states, the number of actions, a discount factor $\beta \in (0, 1]$, the tolerance amount ε, and the maximum number of iterations `maxiter`. Perform value iteration until $\|V_{k+1} - V_k\| < \varepsilon$ or $k >$ `maxiter`. Return the final vector representing V^* and the number of iterations. Test your code on the example given above.

Calculating the Policy

While knowing the maximum expected value is helpful, it is usually more important to know the policy that generates the most value. Value Iteration tells the robot what reward he can expect, but not how to get it. The policy vector, $\mathbf{c}$, is found by using the policy function: $\pi : \mathbb{R} \to \mathbb{R}$. $\pi(s)$ is the action we should take while in state s to maximize reward. We can modify the Bellman equation using $V^*(s)$, which is the true value function we found in problem 1, to find π:

$$\pi(s) = \text{argmax}_{a \in A_s}\{\Sigma_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V^*(\bar{s}))\} \tag{19.6}$$

Using value iteration, we found $V^* = [1, 1, 1, 0]$ in the example above. We find $\pi(0)$ from the example above with $\beta = 1$ by looking at actions 1 and 2 (since actions 0 and 3 have probability 0).

$$\begin{aligned}
\pi(0) &= \text{argmax}_{1,2}\{p(2) * (u(2) + V^*(2)), p(1) * (u(1) + V^*(1))\} \\
&= \text{argmax}\{1 * (-1 + 1), 1 * (0 + 1)\} \\
&= \text{argmax}\{0, 1\} \\
&= 2
\end{aligned}$$

So when the robot is in state 0, he should take action 2, moving *Right*. This avoids the -1 penalty for moving *Down* into square 2. Similarly,

$$\begin{aligned}
\pi(1) &= \text{argmax}_{0,1}\{1 * (-1 + 1), 1 * (1 + 0)\} \\
&= \text{argmax}\{0, 1\} = 1 \\
\pi(2) &= \text{argmax}_{2,3}\{1 * (1 + 0), 1 * (0 + 1)\} \\
&= \text{argmax}\{1, 1\} = 2
\end{aligned}$$

Since 3 is terminal, it does not matter what $\pi(3)$ is. We set it to 0 for convenience. The policy corresponding to the optimal reward is $[2, 1, 2, 0]$. The robot should move to square 3 if possible, avoiding 2 because it has a negative reward.

> **Note**
>
> Note that π gives the optimal action a to take at each state s. It does not give a sequence of actions to take in order to maximize the policy.

Problem 2. Write a function called `extract_policy()` that will accept a dictionary P representing the decision process, the number of states, the number of actions, an array representing the value function, and a discount factor $\beta \in (0, 1]$, defaulting to 1. Return the policy vector corresponding to V^*. Test your code on the example with $\beta = 1$.

Policy Iteration

For dynamic programming problems, it can be shown that value function iteration converges relative to the discount factor β. As $\beta \to 1$, the number of iterations increases dramatically. As mentioned earlier β is usually close to 1, which means this algorithm can converge slowly. In value iteration, we used an initial guess for the value function, V_0 and used (19.1) to iterate towards the true value function. Once we achieved a good enough approximation for V^*, we recovered the true policy function π^*. Instead of iterating on our value function, we can instead make an initial guess for the policy function, π_0, and use this to iterate toward the true policy function. We do so by taking advantage of the definition of the value function, where we assume that our policy function yields the most optimal result. This is policy iteration.

That is, given a specific policy function π_k, we can modify (19.1) by assuming that the policy function is the optimal choice. This process, called *policy evaluation*, evaluates the value function for a given policy.

$$V_{k+1}(s) = \max_{a \in [A_s]} \{\Sigma_{\bar{s} \in N_{s,a}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s}))\} = \Sigma_{\bar{s} \in N_{s,\pi(s)}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})) \quad (19.7)$$

The last equality occurs because in state s, the robot should choose the action that maximizes reward, which is $\pi(s)$ by definition. Similarly to *value iteration*, *policy iteration* iterates until the desired tolerance is reached. This ensures that we get the value function that optimizes the reward for each starting state.

Problem 3. Write a function called `compute_policy_v()` that accepts a dictionary P representing the decision process, the number of states, the number of actions, an array representing a policy, a discount factor $\beta \in (0, 1]$, and a tolerance amount ε. Use the *policy evaluation* process described above to return the value function corresponding to the policy.

Test your code on the policy vector generated from `extract_policy()` for the example. The result should be the same value function array from `value_iteration()`.

Now that we have the value function for our policy, we can take the value function and find a better policy. This is called *policy improvement*. This step is the same method used in value iteration to find the policy. In other words, this step uses the *extract_policy()* method from 2 with the newly computed value function.

Policy function iteration starts with an intial π_0 and iterates using *policy evalutaion* and *policy improvement* successively until the desired tolerance is reached. The algorithm for policy function iteration, using two of the functions that you previously implemented, can be summarized as follows:

Algorithm 19.1 Policy Iteration

procedure POLICY ITERATION FUNCTION(P, nS, nA, β, tol, maxiter)
  $\pi_0 \leftarrow [\pi_0(w_0), \pi_0(w_1), \ldots, \pi_0(w_N)]$ ▷ Common choice is $\pi_0(w_i) = w_{i-1}$ with $\pi_0(0) = 0$
  for $k = 0, 1, \ldots,$ `maxiter` **do** ▷ Iterate only `maxiter` times at most
    v_{k+1} = compute_policy_v(π_k) ▷ Policy evaluation using compute_policy_v
    π_{k+1} = extract_policy(v_{k+1}) ▷ Policy improvement using extract_policy
    if $||\pi_{k+1} - \pi_k|| < \varepsilon$ **then**
      `break` ▷ Stop iterating if the policy doesn't change enough
  return V_{k+1}, π_{k+1}

Problem 4. Write a function called `policy_iteration()` that will accept a dictionary P representing the decision process, the number of states, the number of actions, a discount factor $\beta \in (0, 1]$, the tolerance amount ε, and the maximum number of iterations `maxiter`. Perform policy iteration until $\|\pi_{k+1} - \pi_k\| < \varepsilon$ or $k >$ `maxiter`. Return the final vector representing V_k, the optimal policy π_k, and the number of iterations. Test your code on the example given above and compare your answers to the results from problems 1 and 2.
(Hint: This is just the *Policy Iteration* algorithm, except you also return the number of iterations)

The Frozen Lake Problem

For the rest of the lab, we will be using OpenAi Gym environments. They can be installed using conda or pip.

```
$ pip install gym[all]
```

In the Frozen Lake problem, you and your friends tossed a frisbee onto a mostly frozen lake. The lake is divided into an $N \times N$ grid where the top left hand corner is the start, the bottom right hand corner is the end, and the other squares are either frozen or holes. To retrieve the frisbee, you must successfully navigate around the melted ice without falling. The possible actions are left, right, up, and down. Since ice is slippery, you won't always move in the intended direction. Hence, this is a stochastic Markov process, i.e. $p(s_t, a_t, s_{t+1}) < 1$. If you fall, your reward is 0. If you succeed, your reward is 1. There are two scenarios: $N = 4$ and $N = 8$.

S			
	H		H
			H
H			F

Figure 19.1: Diagram of the 4x4 scenario. The green S represents the starting point and the green F represents the frisbee. Red squares marked H are holes. Blue squares are pieces of the frozen lake.

This problem can be found in two environments in OpenAi Gym. To run the 4×4 scenario, use `env_name='FrozenLake-v0'`. For the 8×8 scenario, use `env_name='FrozenLake8x8-v0'`.

Using Gym

To use gym, we import it and create an environment based on the built-in gym environment. The FrozenLake environment has 3 important attributes, P, nS, and nA. P is similar to the dictionaries we used in the previous problems. As previously mentioned, $p(s_t, a_t, s_{t+1}) < 1$, which means the set $N_{s,a}$ has more than one value. If you did not create your earlier functions to account for that, it will not work as intended on this dictionary. We will use the enivronment's generated P instead of creating our own dictionary of states and actions.

We can calculate the optimal policy with value iteration or policy iteration using these 3 attributes. Since the ice is slippery, this policy will not always result in a reward of 1.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name  = 'FrozenLake-v0'
env = gym.make(env_name).env
# Find number of states and actions
number_of_states = env.nS
number_of_actions = env.nA
# Get the dictionary with all the states and actions
dictionary_P = env.P
```

Problem 5. Write a function that runs `value_iteration` and `policy_iteration` on FrozenLake. It should accept a boolean `basic_case` defaulting to `True` and an integer M defaulting to 1000 that indicates how many times to run the simulation. If `basic_case` is `True`, run the 4x4 scenario. If not, run the 8x8 scenario. Calculate the value function and policy for the environment using both value iteration and policy iteration. Return the policy generated by value iteration and the policy and value function generated by policy iteration. Set the mean total rewards of both policies to 0 and return them as well.

The gym environments have built-in functions that allow us to simulate each step of the environment. Before running a scenario in gym, always put it in the starting state by calling `env.reset()`. To simulate moving, call `env.step`.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name  = 'FrozenLake-v0'
env = gym.make(env_name).env
# Put environment in starting state
obs = env.reset()
# Take a step in the optimal direction and update variables
obs, reward, done, _ = env.step(int(policy[obs]))
```

The step function returns four values: observation, reward, done, info. The observation is an environment-specific object representing the observation of the environment. For FrozenLake, this is the current state. When we step, or take an action, we get a new observation, or state, as well as the reward for taking that action. If we fall into a hole or reach the frisbee, the simulation is over so we are done. When we are done, the boolean `done` is `True`. The info value is a dictionary of diagnostic information. It will not be used in this lab.

Problem 6. Write a function `run_simulation()` that takes in the environment `env`, a policy `policy`, a boolean `render`, and a discount factor β. Calculate the total reward for the policy for one simulation using `env.reset()` and `env.step()`. If `render` is *True*, render the environment using `env.render(mode = 'human')`. Stop the simulation when *done* is *True*. Make sure not to use `env.close()` within this function. Use `env.reset`, because it is much faster than opening and closing the environment every time you run the simulation. You should close the environment at the end of your `frozen_lake()` function.
(Hint: When calculating reward, use β^k as shown in 19.1.)

Next, modify `frozen_lake()` to call `run_simulation()` for both the value iteration policy and the policy iteration policy M times. Finally, make sure that `frozen_lake()` returns the updated values of the mean total reward for both policies.
(Hint: Even though you run the simlutaion M times, you should only calculate the policies once, because each policy depends on the dictionary P, which does not change.)