19 Policy Function Iteration

Lab Objective: *Learn how iterative methods can be used to solve dynamic optimization problems. Implement value iteration and policy iteration in a pseudo-infinite setting where finite value iteration is intractable.*

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 to solve the dynamic cake-eating problem: infinite value function iteration, often just known as value iteration (VI), and infinite policy function iteration, also called policy iteration (PI).

Infinite Value Iteration

Recall that in finite dynamic optimization we are trying to optimize problems of the form

$$\begin{aligned} &\max_{c_t} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to } &\sum_{t=0}^{T} c_t = W_{max} \\ &c_t \geq 0. \end{aligned}$$

However, there are certain problems where T is unknown or extremely large. When T is large, β must be close to 1, otherwise β^t will rapidly approach zero. For example, in 100 time steps, $(0.9)^{100} \approx 2.65\text{e-}5$, whereas $\beta = 0.999$ does not get that low until around $t = 10500$. Thus even if T is large, we are usually constrained by our choice of β, because eventually $\beta^t \to 0$. In the previous lab, we defined the *value function* $V(W)$ to be the maximum utility that comes from having W cake. By Bellman's equation we express the value function as follows.

$$V(w_i) = \max_{c \in [0, w_i]} u(c) + \beta V(w_i - c) \tag{19.1}$$

It is common to make the substitution $w_i' = w_i - c$ in (19.1) to obtain the following.

$$V(w_i) = \max_{w_i' \in [0, w_i]} u(w_i - w_i') + \beta V(w_i'). \tag{19.2}$$

Note that (19.1) requires finding c, how much cake to consume in each period. The variable c is commonly described as the *action* to take when we have w_i cake. On the other hand, (19.2) requires finding w_i', or how much cake to choose to save for later. The variable w_i' is often described as the *state* we will be in at the next period.

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:

$$[T(f)](w) = \max_{w' \in [0,w]} u(w - w') + \beta f(w'). \tag{19.3}$$

It can be shown that (19.2) is the "fixed point" of our operator T. A result of contraction mappings is that there exists a unique solution to (19.3).

A powerful property of contraction mappings is that the fixed point can be found by applying the function repeatedly to some initial point in our domain. That is, if $f : X \to X$ is our contraction mapping, with fixed point $p \in X$, we can find p by randomly choosing $x \in X$ and repeatedly applying the function f to our point. This can be expressed as

$$f^M(x) = f \circ f \cdots \circ f(x) = p$$

for some M. Here M is dependent on both the initial guess x and the discount factor of the contraction mapping, which correspond to V_0 and β in dynamic optimization.

In the case of dynamic optimization, this implies that we can converge to the true value function $V^*(w)$ by using the following equation:

$$V_{k+1}(w_i) = [T(V_k)](w_i) = \max_{w' \in [0,w_i]} u(w_i - w') + \beta V_k(w') \;\forall\, w_i, \tag{19.4}$$

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

1. Discretize the space into N equal sized pieces: $\mathbf{w} = [w_0, w_1, \ldots, w_N]$, where $w_0 = 0$, $w_N = W_{max}$. w_i is the value when there are i pieces of cake remaining.
2. Choose an initial vector to represent V_0: $[V_0(w_0), V_0(w_1), \ldots, V_0(w_N)]$ A common initial choice for V_0 is $V_0(w_i) = u(w_i)$.
3. For $k = 1, 2, \ldots,$ `max_iter`:
   (a) For each $w_i \in \mathbf{w}$: Calculate $V_{k+1}(w_i) = \max_{w' \in \mathbf{w} \,|\, w' \leq w_i} u(w_i - w') + \beta V_k(w')$.
   (b) If $\|V_{k+1} - V_k\| < \varepsilon$, terminate early.

Most iterative algorithms have a `max_iter` parameter that will terminate the algorithm after `max_iter` iterations regardless of whether or not it has converged. This is because even though we have guaranteed convergence, we might have a convergence rate that is too slow to be useful. However, generally this algorithm will converge much faster than computing the true value function using dynamic programming as in Finite Value Iteration.

ACHTUNG!

Here w' represents how much cake to leave for the next period, **not** a derivative of w. Also note that $w' \leq w_i$, because we can never have more cake at a later period; cake can only be consumed, not created.

Problem 1. Write a function called `value_iteration()` that will accept a numpy array representing the initial vector V_0, a discount factor $\beta \in (0, 1)$, the number of states to split $\mathbf{w}$ into N, the amount of initial cake W_{max}, a utility function $u(x)$, the tolerance amount ε, and the maximum number of iterations `max_iter`. Perform value iteration until $\|V_{k+1} - V_k\| < \varepsilon$ or $k >$ `max_iter`. Return the final vector representing V^*.

It is useful for our function to accept V_0 as a parameter instead of calculating an initial guess inside our function, so that we can try different initial states for V_0.

Test your function with the parameters $N = 400$, $\beta = .995$, $u(x) = \sqrt{x}$, $W_{max} = 1$. Try different values for V_0 and see if you get the same value for $V^*(W_{max})$ (The value should be approximately 9.4988). How do different initial guesses for V_0 affect the number of iterations required for convergence?

The value function $V^*(w_i)$ that we found describes how much utility w_i will yield over time, thus $V^*(W_{max})$ is the optimal value for our problem:

$$\begin{aligned} V^*(W_{max}) &= \max_{c_t} \sum_{t=0}^{T} \beta^t u(c_t), \\ \text{subject to } &\sum_{t=0}^{T} c_t = W_{max}, \\ & c_t \geq 0. \end{aligned}$$

Although $V^*(W_{max})$ is the solution, it is usually more important to know which sequence of $(c_t)_{t=0}^{T}$ yields the solution. This sequence is known as the *policy vector* $\mathbf{c} = [c_0, c_1, \ldots, c_T]$ that corresponds to eating c_i cake at time $t = i$.

If this were a truly infinite problem, $\mathbf{c}$ would be impossible to calculate, there would be an infinite number of time periods. Fortunately, in the cake-eating problem, we will never have more than N time periods. This happens because it is never optimal to eat 0 pieces of cake in a single time period. The discount factor β means at least 1 piece must be eaten at each step. It is important to note that the length of $\mathbf{c}$ changes depending on what β is. When $\beta = 0$, for example, only the first time period matters because all the rest will yield zero utility. So when $\beta = 0$, $T = 0$ and the length of $\mathbf{c}$ is 1. As the value of β gets closer to 1, T increases as well.

The policy vector, $\mathbf{c}$, is found by using the policy function: $\pi : \mathbb{R} \to \mathbb{R}$ which has the constraint $0 \leq \pi(w_i) \leq w_i \ \forall\, i$. $\pi(w_i)$ is the amount of cake to save for the next period, given we started with w_i cake. Because $\pi(w_i)$ is the amount of cake saved until next period, we can use $V^*(W)$ and modify the Bellman equation to find π:

$$\pi(w_i) = \text{argmax}_{w_i' \in \mathbf{w} \,|\, w_i' \leq w_i} u(w_i - w_i') + \beta V^*(w_i') \ \forall\, i. \tag{19.5}$$

For our purposes, the policy function will be represented as a vector π. This is convenient because in practice it is infeasible to code a functional representation for π. π_i dictates how many pieces of

cake to save for the next period if we currently have i pieces of cake. For example, if $\pi(W_{max}) = w_4$, we would represent this by having $\pi[N] = 4$. Then we use $\mathbf{w}_4$ to get the actual value for w_4.

Once we have a vectorized representation of the policy function, π, we can use it to calculate the policy vector $\mathbf{c}$ using the relationship:

$$\begin{aligned} c_t &= w^{(t)} - \pi(w^{(t)})), \\ \text{where } w^{(t+1)} &= \pi(w^{(t)}), \\ \text{and } w^{(0)} &= W_{max}. \end{aligned}$$

ACHTUNG!

$w^{(t)}$ represents a time index, how much cake we have at time t, not to be confused with w_i, the numeric value of having i pieces.

Problem 2. Write a helper function called `extract_policy_vector()` that will accept an array of discretized values $\mathbf{w}$, and a vector representing a policy function π. Return the policy vector $\mathbf{c}$ that determines how much cake should be eaten at each time step.

Test your function with $\mathbf{w} = [0, .1, .2, \ldots, 1]$ and $\pi = [0, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5]$. The resulting policy vector should be $\mathbf{c} = [0.5, 0.2, 0.1, 0.1, 0.1]$.

Problem 3. Modify `value_iteration()` to return the true value function V_{k+1} and the corresponding policy vector $\mathbf{c}$.
(Hint: Use (19.5) to find the policy function and then call `extract_policy_vector()` inside of `value_iteration()`).

Policy Function Iteration

For infinite horizon dynamic programming problems, it can be shown that value function iteration converges relative to the discount factor, β. As β approaches 1, the number of iterations increases dramatically. As mentioned earlier β is usually close to 1, which means this algorithm can converge slowly. In Problem 1 you should have noticed that runtime was significantly longer to run for larger N or β closer to 1.

In value iteration we used an initial guess to the value function, V_0 and used (19.2) to iterate towards the true value function. Once we achieved a good enough approximation for V^*, we recovered the true policy function π. Unfortunately, as stated, this can have very slow convergence. 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.

That is, given a specific policy function $\pi_k(W)$, we can modify (19.2) by assuming that the

policy function is the optimal choice, that is:

$$V_k(w_i) = \max_{w_i' \in [0, w_i]} u(w_i - w_i') + \beta V_k(w_i') = u(w_i - \pi_k(w_i)) + \beta V_k(\pi_k(w_i)).$$

Because the value function is defined recursively, this implies:

$$V_k(W) = \sum_{t=0}^{\infty} \beta^t u(\pi_k^t(W) - \pi_k^{t+1}(W)), \tag{19.6}$$

where $\pi_k^t(W)$ means applying π_k t times, and $\pi_k^0(W) = W$. Recall that (19.6) will terminate in a finite number of steps (because we will eventually run out of cake to eat). Fortunately, in cake-eating, (19.6) can alternatively be calculated by using dynamic programming which defines the relationship as:

$$V_k(W) = u(\pi_k^t(W) - \pi_k^{t+1}(W)) + \beta V_k(\pi_k^{t+1}(W)). \tag{19.7}$$

This happens because $\pi_k^{t+1}(W) < W$, with the initial condition that $V_k(0) = 0$. Thus, given an initial guess for our policy function, π_0, we calculate the corresponding value function using (19.6), and then use (19.5) to improve our policy function. This process is known as policy iteration. The algorithm for policy function iteration can be summarized as follows:

1. Discretize the space into N equal sized pieces: $\mathbf{w} = [w_0, w_1, \ldots, w_N]$ where $w_0 = 0$, $w_N = W_{max}$.
2. Choose an initial vector to represent π_0: $[\pi_0(w_0), \pi_0(w_1), \ldots, \pi_0(w_N)]$ A common initial choice for π_0 is $\pi_0(w_i) = w_{i-1}$, meaning we save w_{i-1} pieces, and $\pi_0(w_0) = 0$.
3. For $k = 1, 2, \ldots,$ `max_iter`:
   (a) For each $w_i \in \mathbf{w}$ compute the value function $V_k(w_i)$ using (19.7).
   (b) For each $w_i \in \mathbf{w}$ calculate $\pi_{k+1}(w_i) = \text{argmax}_{W' \in \mathbf{w} \,|\, W' \leq w_i} u(w_i - W') + \beta V_k(W')$.
   (c) If $\|\pi_{k+1} - \pi_k\| < \varepsilon$, terminate early.

Problem 4. Write a function called `policy_iteration()` that will accept a numpy array representing the initial vector π_0, a discount factor $\beta \in (0, 1)$, the number of states to split $\mathbf{w}$ into N, the initial amount of initial cake W_{max}, a utility function $u(x)$, the convergence tolerance ε, and the maximum number of iterations `max_iter`. Perform Policy Iteration until $\|\pi_{k+1} - \pi_k\|_\infty < \varepsilon$ or $n >$ `max_iter`. Return the final vector representing V_k as well as the policy vector $\mathbf{c}$.

Test your policy iteration by calling `value_iteration()` and `policy_iteration()` using the same values for β, N, W_{max}, and $u(x)$. The value functions returned should be close to equal (use `np.allclose()`), and the policy vectors $\mathbf{c}$ should be identical.

Problem 5. Solve the cake eating problem with both value iteration and policy iteration for various values of β and compare how long each method takes. Use $N = 1000$ as the number of grid points for $\mathbf{w}$ and $\beta = [.95, .975, .99, .995]$.

It is important to use feasible initial guesses in each case in order to make the results

comparable. A good initial guess greatly affects the number of iterations required for convergence. Use $V_0(w_i) = u(w_i)$ for value iteration, and $\pi_0(w_i) = w_{i-1}$, with $\pi_0(w_0) = 0$ for policy iteration.
(Hint: set `max_iter` high enough for each method so that the functions actually converge; large values of β may require several hundred iterations for value iteration.)

Graph the results for each method with β on the x-axis and time on the y-axis. Compare your results to the following figure.