Project 1 - Reinforcement learning

August Jonasson

February 12, 2025

Part 1 - Dynamic programming

The first part considers a cake production machine in a pastry factory that can be in any one of three states: *pristine*, *worn* or *broken*. There are two possible actions in each state - the agent decides to either *continue* producing cakes or to *repair* the machine. If the machine is already broken it is replaced by a new one with a punishing reward. The MDP dynamics of the system are completely known to us, the rewards for each action in each state are clearly defined, and the tasks ahead consist of retrieving the optimal state-value function v_π through exact solution under a predefined policy, and also iteratively using value/policy iteration.

More precisely, the state and action spaces are defined as

$$\begin{aligned}\mathcal{S} &= \{\text{pristine, worn, broken}\}, \quad \text{and}\\ \mathcal{A} &= \{\text{continue, repair}\}.\end{aligned}$$

The transition probabilities (dynamics) are given as

$$p(s'|s,a) = \begin{cases} \text{If } a = \text{continue:} & \begin{pmatrix} 1-\theta & \theta & 0 \\ 0 & 1-\theta & \theta \\ 1 & 0 & 0 \end{pmatrix}, \\ \text{If } a = \text{repair:} & \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \end{cases} \tag{1}$$

and the reward function is defined as

$$r(s,a) = \begin{cases} \text{If } a = \text{continue:} & \begin{pmatrix} C \\ C/2 \\ -Q \end{pmatrix}, \\ \text{If } a = \text{repair:} & \begin{pmatrix} -R \\ -R \\ -Q \end{pmatrix}. \end{cases} \tag{2}$$

Furthermore, we assume the parameter settings $\theta = 0.2$, $R = 5$, $C = 4$ and $Q = 10$.

Task 1

In this first task we are to retrieve the exact solution to $v_\pi(s)$ under the deterministic policy $\pi(s) =$ continue for all $s \in \mathcal{S}$. Since there are three states this will involve solving a system of three linear equations. Using equation (4.4) from the course book (Sutton 2018, p. 74), we have that

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

where $0 \leq \gamma < 1$ is the discount factor.

Since our policy is deterministic we can drop the first summation such that we are left with

$$v_\pi(s) = \sum_{s',r} p(s', r|s, \text{continue})(r + \gamma v_\pi(s')), \quad \forall s,$$

but now, distributing the transition probability, then distributing the summations over the two terms, and finally using equations (3.5) and (3.4) from the course book (p. 49) we can rewrite this equation to

$$v_\pi(s) = r(s, a) + \gamma \sum_{s'} v_\pi(s')p(s'|s, a),$$

which is now in terms of the dynamics and rewards from definitions (1) and (2). Plugging these into the equation we get

$$\begin{cases} v_\pi(\text{pristine}) & = C + \gamma\big[v_\pi(\text{pristine})(1-\theta) + v_\pi(\text{worn})\theta\big], \\ v_\pi(\text{worn}) & = C/2 + \gamma\big[v_\pi(\text{worn})(1-\theta) + v_\pi(\text{broken})\theta\big] \quad \text{and} \\ v_\pi(\text{broken}) & = -Q + \gamma v_\pi(\text{pristine}). \end{cases}$$

as our system of three linear equations. Using the assumed parameter settings and moving all of the unknowns over to the left-hand-side, we get

$$\begin{cases} (1 - 0.8\gamma)v_\pi(\text{pristine}) - 0.2\gamma v_\pi(\text{worn}) & = 4, \\ (1 - 0.8\gamma)v_\pi(\text{worn}) - 0.2\gamma v_\pi(\text{broken}) & = 2 \quad \text{and} \\ v_\pi(\text{broken}) - \gamma v_\pi(\text{pristine}) & = -10, \end{cases}$$

which is most easily solved by using your favorite solver of systems of linear equations (we used WolframAlpha) under the condition of $0 \leq \gamma < 1$.

We are finally left with the state-value function expressions

$$\begin{cases} v_\pi(\text{pristine}) & = \dfrac{10(\gamma^2 + 7\gamma - 10)}{\gamma^3 - 16\gamma^2 + 40\gamma - 25}, \\ v_\pi(\text{worn}) & = \dfrac{-10(6\gamma^2 - 9\gamma + 5)}{\gamma^3 - 16\gamma^2 + 40\gamma - 25} \quad \text{and} \\ v_\pi(\text{broken}) & = \dfrac{10(23\gamma^2 - 50\gamma + 25)}{\gamma^3 - 16\gamma^2 + 40\gamma - 25}. \end{cases}$$

We can loosely check that these seem reasonable by setting $\gamma = 0$, which corresponds to only caring about the immediate rewards. The value functions should thus return the rewards themselves exactly, i.e. $v_\pi(\text{pristine}) = C = 4$, $v_\pi(\text{worn}) = C/2 = 2$ and $v_\pi(\text{broken}) = -Q = -10$. This can easily be seen to be the case for our equations and we leave it at this.

Task 2

In this task we are to perform **one** step of value iteration for the cake machine problem. The update rule can be seen from equation (4.10) in the course book (p. 83) and is given as

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

where k denotes the k:th value iteration. Using the same rewriting as in the previous task, this can be expressed in the alternative way

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

which is now again in terms of the rewards and dynamics that we have been given.

Now, since the initial values v_0 can be set arbitrarily when performing value iteration, we make it easy for ourselves by setting these to zero, i.e. $v_0(\text{pristine}) = v_0(\text{worn}) = v_0(\text{broken}) = 0$. The summation thus evaluates to zero for all states when calculating v_1 and we only have to focus on maximizing $r(s, a)$, which simply corresponds to choosing whatever action has the highest reward in a given state. More precisely, we have

$$v_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

which leads to

$$v_1 = \begin{pmatrix} C \\ C/2 \\ -Q \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ -10 \end{pmatrix},$$

and we are done.

Task 3 (see code in appendix)

In this final task of Part 1 we are to implement value and policy iteration in code (see algorithms on pages 80 and 83), using different discount factors $\gamma = \{0.7, 0.8, 0.9\}$. For $\gamma = 0.9$ specifically, we are to present the optimal value function and optimal policy in a table. After this, we plot the learning curves for each of the γ-values and compare their speeds of convergence. Which one converges the fastest and why?

Using $\gamma = 0.9$ and a tolerance $\epsilon = 10^{-6}$, both methods converged to the same optimal value and policy (up to the fifth decimal place), which can be seen from the following table where we have rounded to two decimals:

fcts \ states	Pristine	Worn	Broken
Optimal value	26.27	18.64	13.64
Optimal policy	continue	repair	continue

It can be mentioned that in some of the runs, we got $\pi(\text{broken}) = \text{repair}$ as the optimal policy, with unchanged optimal value. This is an equivalent solution as the outcome is the same when the machine is broken, no matter what action is taken - the machine has to be replaced.

Now moving on to studying the rates of learning convergence for the different values on γ.

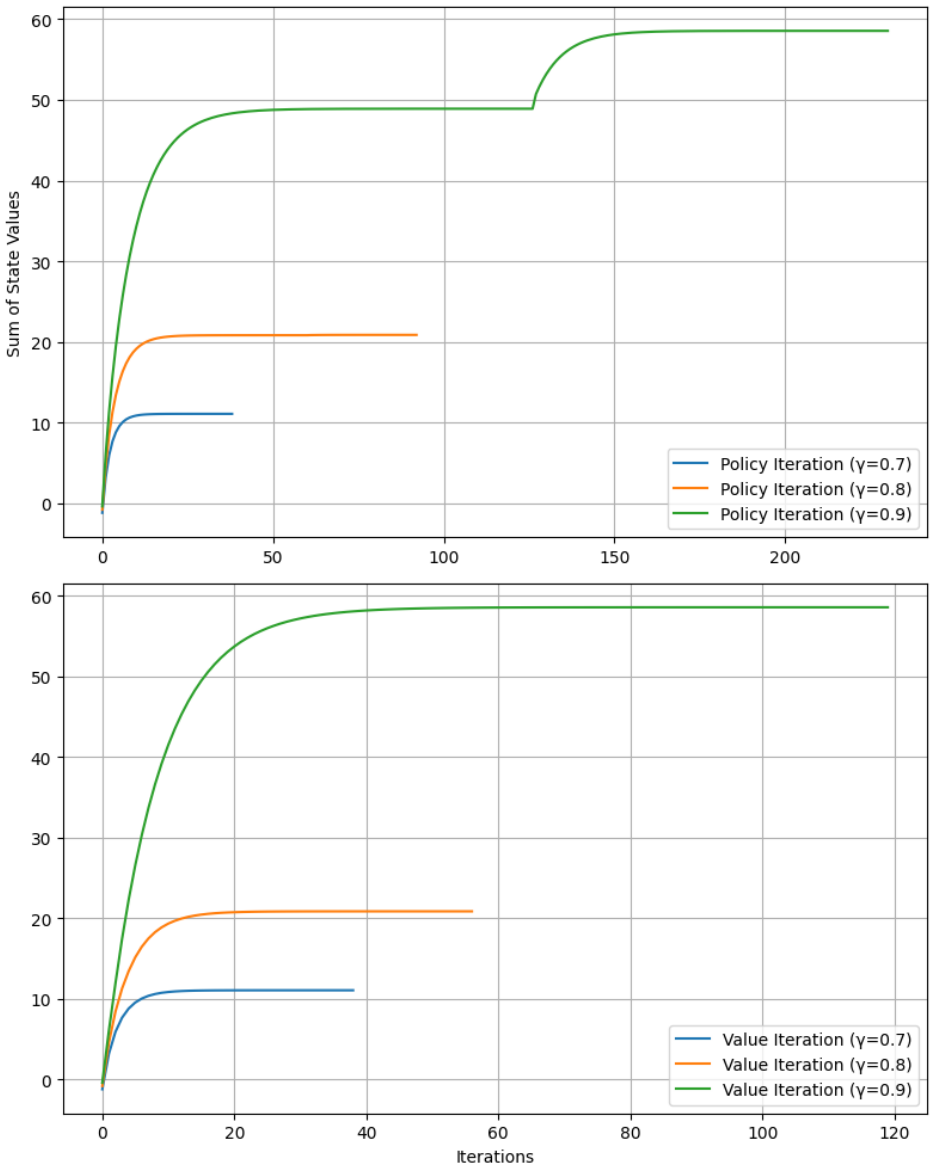

Figure 1: Policy and value iteration learning curves for different values on γ.

From Figure 1 it is evident that the value iteration algorithm converges to the optimal value in fewer iterations than the policy iteration. The difference seems to increase almost exponentially as γ grows linearly. It is not surprising that the value iteration is more efficient, as the policy iteration is very susceptible to wasting compute on evaluating state-values under non-optimal policies. This phenomenon can be seen from the policy iteration curve of $\gamma = 0.9$ - it has to "converge" twice in order to reach the optimal policy. Value iteration, on the other hand, combines evaluation and improvement in each step, thus wasting less resources.

Finally, we want to answer why higher values on the discount factor γ lead to slower convergence. Consider the return

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

(eq. (3.11), p. 57) where $T \to \infty$ as the task we are modeling is continuous, and R_k is the reward at time step k. For $0 \leq \gamma < 1$ this will obviously converge slower as γ grows closer to one. One could say that the complexity of the sum increases alongside γ. Since we are trying to maximize

$$v_\pi(s) := \mathbb{E}\left[G_t | S_t = s\right],$$

which is directly dependent on G_t, it is entirely expected that convergence is slower for larger γ. Practically speaking, a larger γ means looking further into the future, which intuitively should be a more complex task than considering, say, only the immediate rewards.

Part 2 - Monte Carlo methods

In this part, we consider Monte Carlo methods for an agent located in a 4x4 gridworld where the start is in the bottom left corner (0,0) and the goal is in the top right (3,3). This can be modeled as an MDP with state and action spaces

$$\begin{aligned}\mathcal{S} &= \{(0,0),\dots,(3,3)\} \quad \text{and}\\ \mathcal{A} &= \{\text{left},\text{up},\text{right},\text{down}\},\end{aligned}$$

and reward function

$$r(s,a) = \begin{cases} 10 & \text{if } s = \text{“Goal reached”},\\ -100 & \text{if } s = \text{“Wall hit”},\\ -1 & \text{if } s = \text{“Reaching non-goal state”}.\end{cases}$$

Task 1

In this task, we are to compute the return G_t for each time step t in a given sequence

$$\begin{aligned} s_0 &= (0,0), \quad a_0 = \text{right}, \quad r_1 = -1,\\ s_1 &= (0,1), \quad a_1 = \text{right}, \quad r_2 = -1,\\ s_2 &= (0,2), \quad a_2 = \text{right}, \quad r_3 = -1,\\ s_3 &= (0,3), \quad a_3 = \text{up}, \quad r_4 = -1,\\ s_4 &= (1,3), \quad a_4 = \text{up}, \quad r_5 = -1,\\ s_5 &= (2,3), \quad a_5 = \text{up}, \quad r_6 = -1,\\ s_6 &= (3,3), \quad a_6 = \text{up}, \quad r_7 = 10,\\ s_7 &= (3,3). \end{aligned} \tag{3}$$

Starting at $G_T = G_7 = 0$ and using equation (3.9) from the course book (p. 55)

$$G_t = R_{t+1} + \gamma G_{t+1},$$

we work our way backwards towards the beginning of the sequence. We then get the returns

$$\begin{aligned} G_6 &= 10,\\ G_5 &= -1 + 10\gamma,\\ G_4 &= -1 - \gamma + 10\gamma^2,\\ &\vdots\\ G_0 &= -1 - \gamma - \gamma^2 - \gamma^3 - \gamma^4 - \gamma^5 + 10\gamma^6, \end{aligned}$$

and we are done.

Task 2

Using the sequence (3), we are to perform one step of MC prediction, and present the estimated value for each state (that was visited) in a 4x4 table.

In order to solve this task we use the first-visit algorithm in the course book (p. 92). Now, since all states that appear in the sequence (episode) only appear once, each visit to a state will be the first and only visit to that state. We can thus attribute the returns calculated in the previous task to one-episode estimations of the state-values directly, which can be seen from Table 1.

nan	nan	nan	G_6
nan	nan	nan	G_5
nan	nan	nan	G_4
G_0	G_1	G_2	G_3

Table 1: One-episode estimate of state values of the 4x4 gridworld using first-visit Monte Carlo prediction. Bottom left cell is the initial state (0,0) and top right is the terminal state (3,3).

This concludes this task.

Task 3

We are finally to implement the given gridworld in code, using discount factor $\gamma = 0.9$. Unfortunately, we were unable to get this code to work in time, and as such, we leave this answer blank.

Python code - Part 1

```python
# %%
import numpy as np
from matplotlib import pyplot as plt

# %%
# MDP components
states = ["pristine", "worn", "broken"]
actions = ["continue", "repair"]
gamma = 0.9  # Discount factor
theta = 1e-6  # tolerance

# Transition probabilities p(s' | s, a)
P = {
    "pristine": {"continue": {"pristine": 0.8, "worn": 0.2, "broken": 0},
                 "repair": {"pristine": 1, "worn": 0, "broken": 0}},

    "worn": {"continue": {"pristine": 0, "worn": 0.8, "broken": 0.2},
             "repair": {"pristine": 1, "worn": 0, "broken": 0}},

    "broken": {"continue": {"pristine": 1, "worn": 0, "broken": 0},
               "repair": {"pristine": 1, "worn": 0, "broken": 0}}
}

# Rewards r(s, a)
r = {
    "pristine": {"continue": 4, "repair": -5},
    "worn": {"continue": 2, "repair": -5},
    "broken": {"continue": -10, "repair": -10}
}

# Policy and value function initializations (arbitrary)
policy = {s: "continue" for s in states}
V = {s: 0.0 for s in states}


# %%
def policy_evaluation(policy, V, P, r, theta, gamma):
    """Performs policy evaluation until value function converges."""
    iterations = 0
    history = []
    while True:
        delta = 0
        for s in states:
            v = V[s]  # Store old value
            a = policy[s]  # Policy dictates action
            V[s] = sum(P[s][a][s_next] * (r[s][a] + gamma * V[s_next]) for s_next in
    states)
            delta = max(delta, abs(v - V[s]))
        history.append(sum(V.values()))  # Track value function sum over time
        iterations += 1
        if delta < theta:
            break
    return iterations, history

def policy_improvement(policy, V, P, r, gamma):
    """Performs policy improvement by selecting greedy actions."""
```

```
    policy_stable = True
    for s in states:
        old_action = policy[s]
        action_values = {a: sum(P[s][a][s_next] * (r[s][a] + gamma * V[s_next]) for s_next in states) for a in actions}
        best_action = max(action_values, key=action_values.get)
        policy[s] = best_action
        if old_action != best_action:
            policy_stable = False
    return policy_stable

def policy_iteration(P, r, gamma):
    """Runs the policy iteration algorithm."""
    policy = {s: "continue" for s in states}  # Initial policy
    V = {s: 0.0 for s in states}  # Initialize state-value function
    iteration_count = 0
    history = []
    while True:
        iteration_count += 1
        eval_iterations, eval_history = policy_evaluation(policy, V, P, r, theta, gamma)
        history.extend(eval_history)
        if policy_improvement(policy, V, P, r, gamma):
            break  # Stop if policy is stable
    return policy, V, iteration_count, history

def value_iteration(P, r, gamma):
    """Runs the value iteration algorithm."""
    V = {s: 0.0 for s in states}  # Reset value function
    iteration_count = 0
    history = []
    while True:
        delta = 0
        for s in states:
            v = V[s]
            V[s] = max(sum(P[s][a][s_next] * (r[s][a] + gamma * V[s_next]) for s_next in states) for a in actions)
            delta = max(delta, abs(v - V[s]))
        history.append(sum(V.values()))  # Track value function sum over time
        iteration_count += 1
        if delta < theta:
            break
    policy = {s: max(actions, key=lambda a: sum(P[s][a][s_next] * (r[s][a] + gamma * V[s_next]) for s_next in states)) for s in states}
    return policy, V, iteration_count, history


# %%
# Run policy iteration
optimal_policy, optimal_values, iteration_count, history_pi = policy_iteration(P, r, gamma)
print("Optimal Policy (Policy iteration):", optimal_policy)
print("Optimal State Values (Policy iteration):", optimal_values)

# Run value iteration
optimal_policy_vi, optimal_values_vi, iteration_count_vi, history_vi = value_iteration(P, r, gamma)
print("\nOptimal Policy (Value Iteration):", optimal_policy_vi)
```

```
print("Optimal State Values (Value Iteration):", optimal_values_vi)

# %%
gamma_values = [0.7, 0.8, 0.9]
fig, axes = plt.subplots(2, 1, figsize=(8, 10))

# Policy Iteration Plot
for gamma in gamma_values:
    _, _, _, history_pi = policy_iteration(P, r, gamma)
    axes[0].plot(history_pi, label=f'Policy Iteration (  ={gamma})')
axes[0].set_ylabel("Sum of State Values")
axes[0].legend(loc = 'lower right')
axes[0].grid()

# Value Iteration Plot
for gamma in gamma_values:
    _, _, _, history_vi = value_iteration(P, r, gamma)
    axes[1].plot(history_vi, label=f'Value Iteration (  ={gamma})')
axes[1].set_xlabel("Iterations")
axes[1].legend()
axes[1].grid()

plt.tight_layout()
plt.show()

# %%
```

References

Sutton, Richard S (2018). “Reinforcement learning: An introduction”. In: *A Bradford Book*.