

General Sir John Kotelawala Defense University

Department of Electrical, Electronics & Telecommunication Engineering

Machine Learning

ET 4103

Assignment – 04

Reinforcement Learning

Index No : D/ENG/22/0120/ET

Name : M. A. E. Wijesuriya

Intake : 39

Submission Date : 20/06/2025

**Q1. Utilize the given Jupyter notebook[1] for Reinforcement Learning. Comment on the code and the output of the program, explaining utilized Machine Learning concepts where necessary**

The following code is a python program that demonstrates Reinforcement Learning. It explains the basic concepts utilized in Reinforcement Learning, and provides code snippets that can be run in the Jupyter notebook provided to witness these concepts put into practice in real time.

The basic principle of Reinforcement Learning is explained: An agent interacts with the environment, which responds with a reward or cost, which enables the agent to learn behavior that leads to greater rewards and lower costs. The problem is characterized as a Markov Decision Process, which has:

* **S** ­– A finite set of states that the agent can inhabit
* **A** – A finite set of actions that the agent can take in each state
* ***R*:S×A→[*Rmin*,*Rmax*]⊂R –** The bounded reward or cost function that gives the agent reinforcement
* ***P*:S×A→Δ(S) –** Transition probabilities of the agent moving to the next state
* ***γ –*** A discount factor. The closer to one it is, the less it encourages the agent to reach rewards as quickly as possible.

The **Policy** of an agent is the strategy or set of rules it utilizes in order to decide what actions it should take at each state in the environment. The optimal policy ***π\**** gives us the value function ***V*\*(*s*) = max*a*∈A [*R*(*s*,*a*)+*γ*∑*s*′∈S*P*(*s*,*a*)(*s*′)*V*∗(*s*′)]**

There are numerous ways we can calculate *π*\* and *V*\*, based on whether the reward/cost functions and transition probabilities are known.

The following code sets up the frameworks and libraries needed to demonstrate these concepts

# @title Installs and imports (run me first!)

!pip install pyglet~=1.3.2

!apt install -y graphviz

!pip install flax

!pip install graphviz

!pip install pyvirtualdisplay

!apt-get install python-opengl -y

!apt install xvfb -y

!pip install 'gym[atari]'

!pip install -U dopamine-rl

# Importing Libraries

import flax

from graphviz import Digraph

import jax

import jax.numpy as jnp

import matplotlib.pyplot as plt

import numpy as onp

from IPython.display import HTML

from pprint import pprint

import logging

from pyvirtualdisplay import Display

logging.getLogger("pyvirtualdisplay").setLevel(logging.ERROR)

# Configures the Virtual Display

display = Display(visible=0, size=(1024, 768))

display.start()

import os

os.environ["DISPLAY"] = ":" + str(display.display)

**Case 1: Known Environment**

These problems are commonly known as planning problems, as the transition and reward dynamics are known. This can be done in two ways, value iteration or policy iteration.

**Value Iteration**

“In Value iteration we are continuously updating an estimate *Vt*+1 by leveraging our previous estimate *Vt*.

***Vt*+1(*s*):=max*a*∈A[*R*(*s*,*a*)+*γ*∑*s*′∈S*P*(*s*,*a*)(*s*′)*Vt*(*s*′)]**

This is typically referred to as the *Bellman backup*. It can be shown that, starting from an initial estimate *V*0, lim*t*→∞*Vt*=*V*\*.

This gives us the value iteration algorithm:

1. Initialize *V*≡0
2. Loop until convergence:
   * For every *s*∈S:  
     *V*(*s*)←max*a*∈A[*R*(*s*,*a*)+*γ*∑*s*′∈S*P*(*s*,*a*)(*s*′)*V*(*s*′)]
3. Return *V ”*

*(extract taken from Jupyter Notebook text found at [1])*

def value\_iteration(P, R, gamma, tolerance=1e-3):

  """Find V\* using value iteration.

  Args:

    P: numpy array defining transition dynamics. Shape: |S| x |A| x |S|.

    R: numpy array defining rewards. Shape: |S| x |A|.

    gamma: float, discount factor.

    tolerance: float, tolerance level for computation.

  Returns:

    V\*: numpy array of shape ns.

    Q\*: numpy array of shape ns x na.

  """

  assert P.shape[0] == P.shape[2]

  assert P.shape[0] == R.shape[0]

  assert P.shape[1] == R.shape[1]

  ns = P.shape[0]

  na = P.shape[1]

  V = onp.zeros(ns)

  Q = onp.zeros((ns, na))

  error = tolerance \* 2

  while error > tolerance:

    # This is the Bellman backup (onp.einsum FTW!).

    Q = R + gamma \* onp.einsum('sat,t->sa', P, V)

    new\_V = onp.max(Q, axis=1)

    error = onp.max(onp.abs(V - new\_V))

    V = onp.copy(new\_V)

  return V, Q

**Policy Iteration**

Policy Iteration allows us to iterate over *πt* and stop once the policy is no longer changing. We calculate Q and V for each policy, and improve the policy by targeting maximum Q values.

def policy\_iteration(P, R, gamma):

  """Find V\* using policy iteration.

  Args:

    P: numpy array defining transition dynamics. Shape: |S| x |A| x |S|.

    R: numpy array defining rewards. Shape: |S| x |A|.

    gamma: float, discount factor.

  Returns:

    V\*: numpy array of shape ns.

    Q\*: numpy array of shape ns x na.

  """

  assert P.shape[0] == P.shape[2]

  assert P.shape[0] == R.shape[0]

  assert P.shape[1] == R.shape[1]

  ns = P.shape[0]

  na = P.shape[1]

  V = onp.zeros(ns)

  Q = onp.zeros((ns, na))

  pi = onp.zeros((ns, na))

  for s in range(ns):

    pi[s, onp.random.choice(na)] = 1.

  policy\_stable = False

  while not policy\_stable:

    old\_pi = onp.copy(pi)

    # Extract V from Q using pi.

    V = [Q[s, onp.argmax(pi[s])] for s in range(ns)]

    Q = R + gamma \* onp.einsum('sat,t->sa', P, V)

    pi = onp.zeros((ns, na))

    for s in range(ns):

      pi[s, onp.argmax(Q[s])] = 1.

    policy\_stable = onp.array\_equal(pi, old\_pi)

  V = [Q[s, onp.argmax(pi[s])] for s in range(ns)]

  Q = R + gamma \* onp.einsum('sat,t->sa', P, V)

  V = [Q[s, onp.argmax(pi[s])] for s in range(ns)]

  return V, Q

**Case 2 : Unknown Environment**

When we do not have access to P and R, that is the transition probabilities and reward/cost functions, the agent has to form its behavioural policy by interacting with the environment. This happens in 3 steps:

1. The agent selects an action from its policy in state s
2. The environment responds with a new state and reward
3. This information is used by the agent to update its policy

There are two ways of doing this: **Model-based** methods, and **Model-free** methods.

* Model-based methods: This approach tries to create approximate models for P and R from the experience it gains from the environment and then solves for V\*,Q\*, and *π*\* using value or policy iteration.
* Model-free methods: This approach directly approximates for V\*,Q\*, and *π*\* based on feedback received from the environment.

**Exploration vs. Exploitation dilemma:** In the context of Reinforcement Learning, an Exploitative policy refers to one that prioritizes choosing the action that directly obtains the greatest reward obtainable in the current state. Exploration refers to the process of the agent selecting a sub-optimal action in the present state in order to potentially lead to greater rewards in the future. A purely Exploitative policy therefore, misses out on potential larger rewards in the future, while a purely Explorative policy causes the agent to constantly act in a random fashion, which prevents it from maximizing on rewards. Therefore, balancing these two policies is important.

The following code snippet illustrates a common exploration method known as *ϵ*-greedy exploration.

“At state *s*, given a policy *π*, the rule for this exploration policy is simply:

* With probability *ϵ* select an action randomly
* With probability 1−*ϵ* select action *a*=argmax*a*∈A*π*(*s*)” [1]

def epsilon\_greedy(s, pi, epsilon):

  """A simple implementation of epsilon-greedy exploration.

  Args:

    s: int, the agent's current state.

    pi: numpy array of shape [num\_states, num\_actions] encoding the agent's

      policy.

    epsilon: float, the epsilon value for epsilon-greedy exploration.

  Returns:

    An integer representing the action choice.

  """

  na = pi.shape[1]

  p = onp.random.rand()

  if onp.random.rand() < epsilon:

    return onp.random.choice(na)

  return onp.random.choice(na, p=pi[s])

**Monte-Carlo Approach:** This approach picks a random starting state s and a random policy *π* and generates a trajectory from the initial state to the final state. This accumulates the returns for each possible action at each possible state, and sets Q to be the average of all these returns. The policy is then updated by maximizing Q for each state.

The following code snippet illustrates the Monte Carlo approach

def monte\_carlo(ns, na, step\_fn, gamma, start\_state, reset\_state, total\_episodes,

               max\_steps\_per\_iteration, epsilon, V):

  """A simple implementation of Q-learning.

  Args:

    ns: int, the number of states.

    na: int, the number of actions.

    step\_fn: a function that receives a state and action, and returns a float

      (reward) and next state. This represents the interaction with the

      environment.

    gamma: float, the discount factor.

    start\_state: int, index of starting state.

    reset\_state: int, index of state where environment resets back to start

      state, or None if there is no reset state.

    total\_episodes: int, total number of episodes.

    max\_steps\_per\_iteration: int, maximum number of steps per iteration.

    epsilon: float, exploration rate for epsilon-greedy exploration.

    V: numpy array, true V\* used for computing errors. Shape: [num\_states].

  Returns:

    V\_hat: numpy array, learned value function. Shape: [num\_states].

    Q\_hat: numpy array, learned Q function. Shape: [num\_states, num\_actions].

    max\_errors: list of floats, contains the error max\_s |V\*(s) - \hat{V}\*(s)|.

    avg\_errors: list of floats, contains the error avg\_s |V\*(s) - \hat{V}\*(s)|.

  """

  # Initialize policy randomly.

  pi\_hat = onp.zeros((ns, na))

  for s in range(ns):

    pi\_hat[s, onp.random.choice(na)] = 1.

  # Initialize Q randomly.

  Q\_hat = onp.zeros((ns, na))

  # Initialize the accumulated returns and number of updates.

  returns = onp.zeros((ns, na))

  counts = onp.zeros((ns, na))

  # Lists to keep track of training statistics.

  iteration\_returns = []

  max\_errors = []

  avg\_errors = []

  for episode in range(total\_episodes):

    # Each episode starts in the same start state.

    s = start\_state

    step = 0

    # Lists collected for each trajectory.

    states = []

    actions = []

    rewards = []

    # Generate a trajectory for a limited number of steps.

    while step < max\_steps\_per\_iteration:

      step += 1

      states.append(s)

      a = epsilon\_greedy(s, pi\_hat, epsilon)  # Pick action.

      actions.append(a)

      r, s2 = step\_fn(s, a)  # Take a step in the environment.

      rewards.append(r)

      if s2 == reset\_state:

        # If we've reached a reset state, the trajectory is over.

        break

      s = s2

    # Update the Q-values based on the rewards received by traversing the

    # trajectory in reverse order.

    G = 0  # Accumulated returns.

    step -= 1

    while step >= 0:

      G = gamma \* G + rewards[-1]

      rewards = rewards[:-1]

      s = states[-1]

      states = states[:-1]

      a = actions[-1]

      actions = actions[:-1]

      # We only update Q(s, a) for the first occurence of the pair in the

      # trajectory.

      update\_q = True

      for i in range(len(states)):

        if s == states[i] and a == actions[i]:

          update\_q = False

          break

      if update\_q:

        returns[s, a] += G

        counts[s, a] += 1

        Q\_hat[s, a] = returns[s, a] / counts[s, a]

        pi\_hat[s] = onp.zeros(na)

        pi\_hat[s, onp.argmax(Q\_hat[s])] = 1.

      step -= 1

    iteration\_returns.append(G)

    V\_hat = onp.max(Q\_hat, axis=1)

    max\_errors.append(onp.max(onp.abs(V - V\_hat)))

    avg\_errors.append(onp.mean(onp.abs(V - V\_hat)))

  return V\_hat, Q\_hat, iteration\_returns, max\_errors, avg\_errors

**Q-Learning :** This is a more dynamic method that is similar to Monte Carlo, but updates its estimate after each single step, rather than after the entire episode. It updates estimates using ***Qπ*(*s*,*a*)=*Vπ*(*s*)+*α*[*r*+*γVπ*(*s*′)−*Vπ*(*s*)]** where*α* is the step size, that determines how aggressively the estimates are updated based on environment feedback.

The following code illustrates Q-Learning.

def q\_learning(ns, na, step\_fn, gamma, start\_state, reset\_state, total\_episodes,

               max\_steps\_per\_iteration, epsilon, alpha, V):

  """A simple implementation of Q-learning.

  Args:

    ns: int, the number of states.

    na: int, the number of actions.

    step\_fn: a function that receives a state and action, and returns a float

      (reward) and next state. This represents the interaction with the

      environment.

    gamma: float, the discount factor.

    start\_state: int, index of starting state.

    reset\_state: int, index of state where environment resets back to start

      state, or None if there is no reset state.

    total\_episodes: int, total number of episodes.

    max\_steps\_per\_iteration: int, maximum number of steps per iteration.

    epsilon: float, exploration rate.

    alpha: float, learning rate.

    V: numpy array, true V\* used for computing errors. Shape: [num\_states].

  Returns:

    V\_hat: numpy array, learned value function. Shape: [num\_states].

    Q\_hat: numpy array, learned Q function. Shape: [num\_states, num\_actions].

    max\_errors: list of floats, contains the error max\_s |V\*(s) - \hat{V}\*(s)|.

    avg\_errors: list of floats, contains the error avg\_s |V\*(s) - \hat{V}\*(s)|.

  """

  # Initialize policy randomly.

  pi\_hat = onp.zeros((ns, na))

  for s in range(ns):

    pi\_hat[s, onp.random.choice(na)] = 1.

  # Initialize Q to zeros.

  Q\_hat = onp.zeros((ns, na))

  # Lists collected for each trajectory.

  iteration\_returns = []

  max\_errors = []

  avg\_errors = []

  for episode in range(total\_episodes):

    # Each episode begins in the same start state.

    s = start\_state

    step = 0

    num\_episodes = 0

    steps\_in\_episode = 0

    cumulative\_return = 0.

    average\_episode\_returns = 0.

    # Interact with the environment for a maximum number of steps

    while step < max\_steps\_per\_iteration:

      a = epsilon\_greedy(s, pi\_hat, epsilon)  # Pick action.

      r, s2 = step\_fn(s, a)  # Take a step in the environment.

      delta = r + gamma \* max(Q\_hat[s2]) - Q\_hat[s, a]  # TD-error.

      Q\_hat[s, a] += alpha \* delta  # Q-learning update.

      cumulative\_return += gamma\*\*(steps\_in\_episode) \* r

      pi\_hat[s] = onp.zeros(na)

      pi\_hat[s, onp.argmax(Q\_hat[s])] = 1.

      s = s2

      steps\_in\_episode += 1

      if s2 == reset\_state:

        s = 0

        num\_episodes += 1

        steps\_in\_episode = 0

        average\_episode\_returns += cumulative\_return

        cumulative\_return = 0.

      step += 1

    average\_episode\_returns /= max(1, num\_episodes)

    iteration\_returns.append(average\_episode\_returns)

    V\_hat = onp.max(Q\_hat, axis=1)

    max\_errors.append(onp.max(onp.abs(V - V\_hat)))

    avg\_errors.append(onp.mean(onp.abs(V - V\_hat)))

  return V\_hat, Q\_hat, iteration\_returns, max\_errors, avg\_errors

**Chain MDP Example**

The Jupyter Notebook further provides us with an example of a Chain MDP that works as follows:

“

* The agent starts in the leftmost state in the chain and can either move "left" (red arrows below) into a sink state and receive a small reward, or move "right" (blue arrows below) into the next state and incur a penalty.
* At each intermediate state in the chain, moving left or right incurs the same penalty.
* In the rightmost state of the chain the agent can move "right" into the sink state and receive a large reward, or move "left" and incur a penalty.
* All transitions have a probability *ρ* of slipping and staying in the same state.

Depending on the values of the following parameters, the resulting values for *V*∗, *Q*∗, *π*∗, and how well the agent learns, will vary:

* Length of the chain
* Penalty and rewards
* Discount factor *γ*
* Slippage amount *ρ*
* Learning rate *α*
* Exploration rate *ϵ*
* Number of episodes to train, and maximum number of steps per episode.” [1]

(the code to create this demonstration has been omitted for sake of conciseness, but can be obtained at [1])

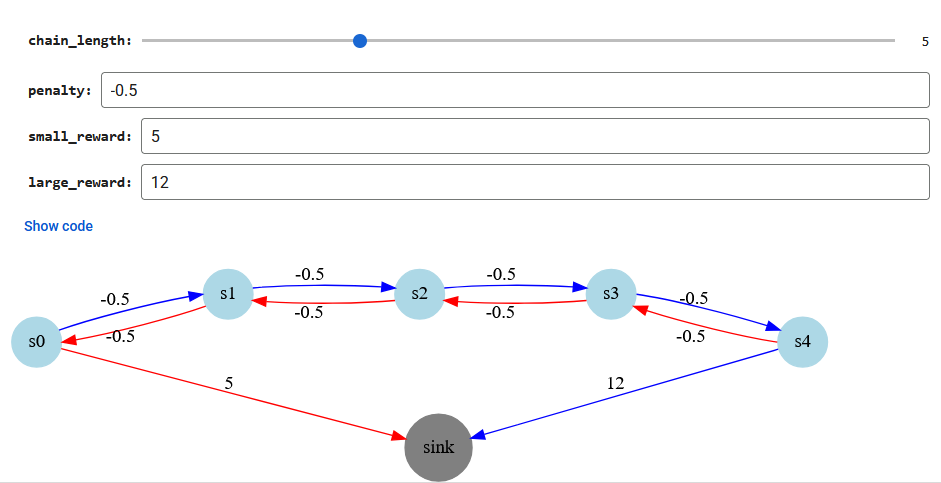
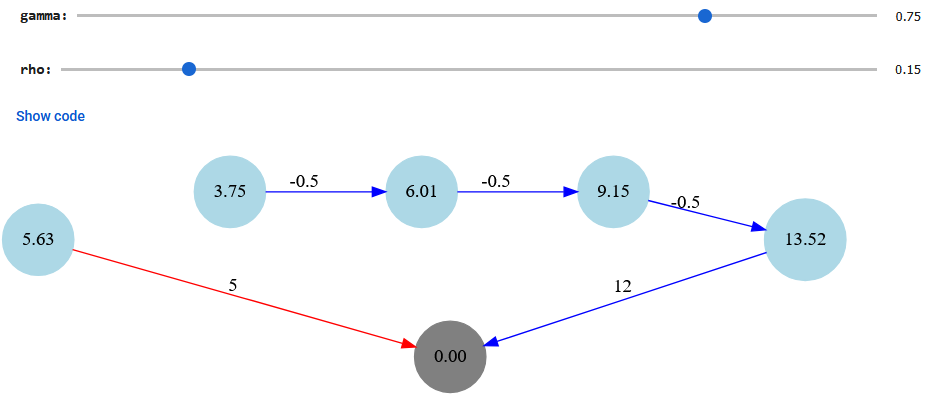


Figure 1: Chain MDP dynamics

This figure shows the chain MDP graph, with states depicted as blue circles (s1,s2…), actions as arrows with rewards on them, and the final state, or sink.



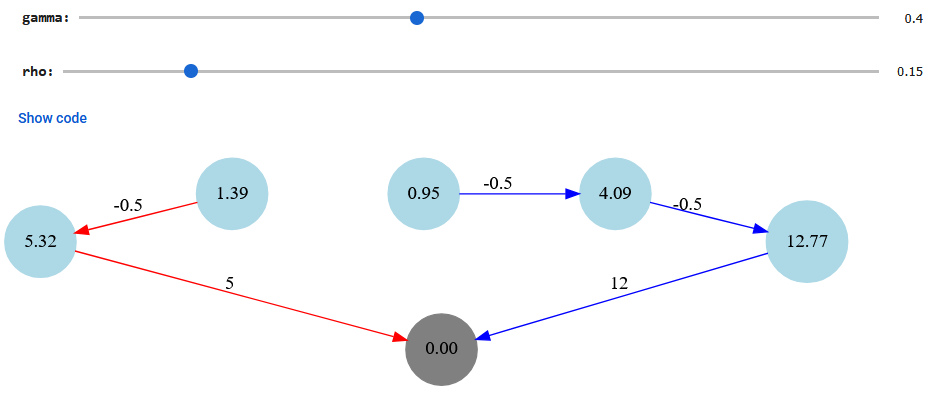
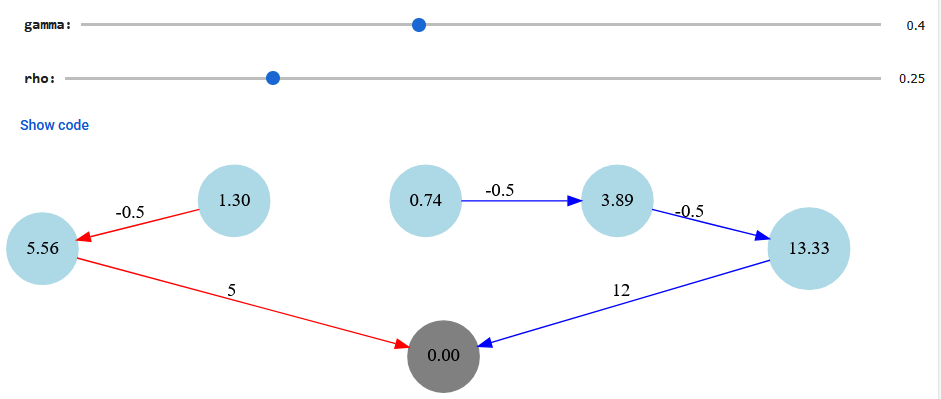


Figure 2: Illustration of different discount factors (0.75 vs. 0.4) on the MDP

From this graph, we can see the value estimates within each state. This shows us the effect of different discount factors on the MDP. When gamma is higher (0.75), the model is more farsighted, and accepts more immediate penalties (-0.5) for the bigger reward at the end (12). Therefore right hand side states have higher value estimations.

When gamma is lower (0.4), the model is more shortsighted, and focused on immediate rewards. This leads to lower value estimations in the right-hand side states.



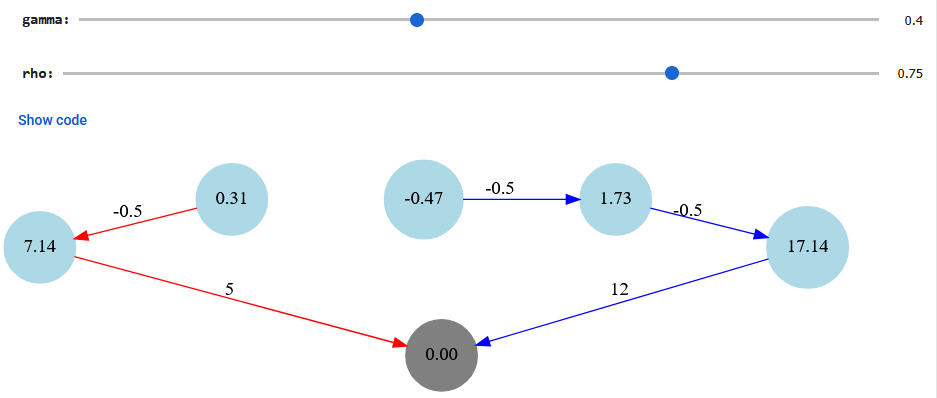


Figure 3: Illustration of different slipping factors on the MDP

As the slippage factor increases, the actions of the agent become less reliable (higher chance of slipping). This leads to the estimated value of states near the sink going up.

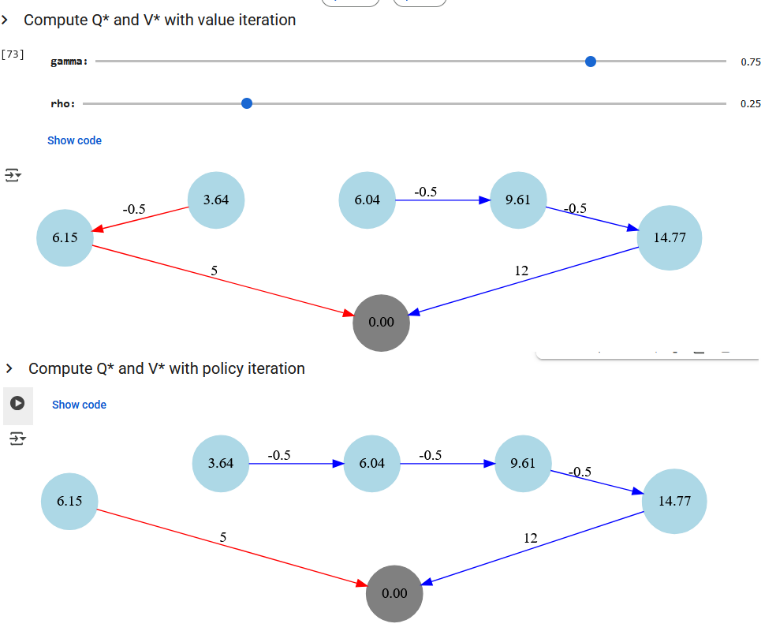


Figure 4: Value Iteration vs. Policy Iteration

As visible from the chart above, the value estimates obtained through both Value Iteration and Policy Iteration are the same, although the paths differ very slightly.

ns = P.shape[0]

na = P.shape[1]

def step\_fn(s, a):

  """Receives a state and action, returns reward and next state."""

  return R[s, a], onp.random.choice(ns, p=P[s,a])

This step function is used to run a Monte Carlo approach on the MDP