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
- $\boldsymbol{R:S\times A\rightarrow[R_{min},R_{max}]\subset R}$ – The bounded reward or cost function that gives the agent reinforcement
- $\boldsymbol{P:S\times A\rightarrow\Delta(S)}$ – Transition probabilities of the agent moving to the next state
- $\boldsymbol{\gamma}$ – 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 $\boldsymbol{\pi^*}$ gives us the value function $\boldsymbol{V^*(s)=\max_{a\in A}[R(s,a)+\gamma\sum_{s'\in S}P(s,a)(s')V_*(s')]}$

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 V_{t+1} by leveraging our previous estimate V_t.

$$V_{t+1}(s) := \max_{a \in A}[R(s,a) + \gamma \sum_{s' \in S} P(s,a)(s') V_t(s')]$$

This is typically referred to as the *Bellman backup*. It can be shown that, starting from an initial estimate V_0, $\lim_{t \to \infty} V_t = V^*$.

This gives us the value iteration algorithm:

1. Initialize $V \equiv 0$
2. Loop until convergence:
    - For every $s \in S$:
      $V(s) \leftarrow \max_{a \in A}[R(s,a) + \gamma \sum_{s' \in S} P(s,a)(s') V(s')]$
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 π_t 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 $1-\epsilon$ select action $a=\text{argmax}_{a \in A}\pi(s)$" [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 $Q\pi(s,a)=V\pi(s)+\alpha[r+\gamma V\pi(s')-V\pi(s)]$ 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])

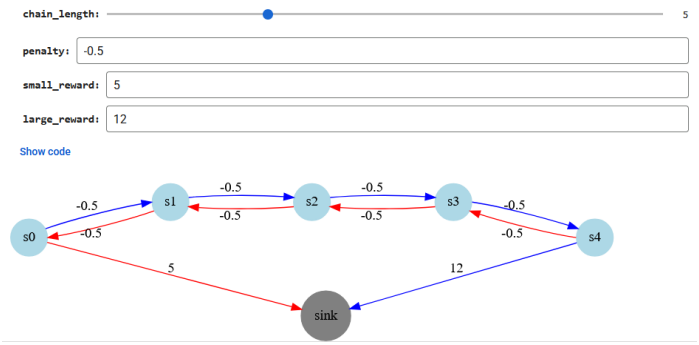

Figure 1: Chain MDP dynamics

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.

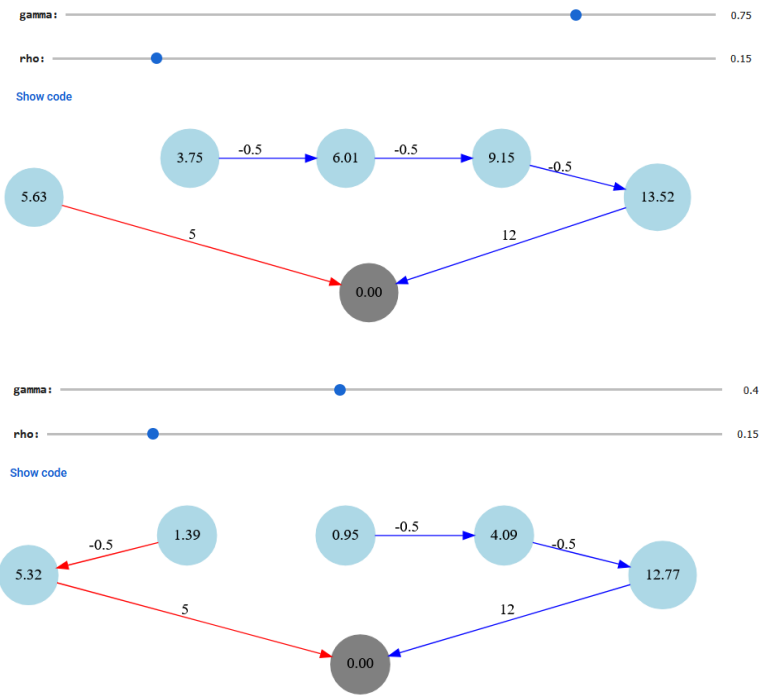

Figure 2: Illustration of different discount factors (0.75 vs. 0.4) on the MDP

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.

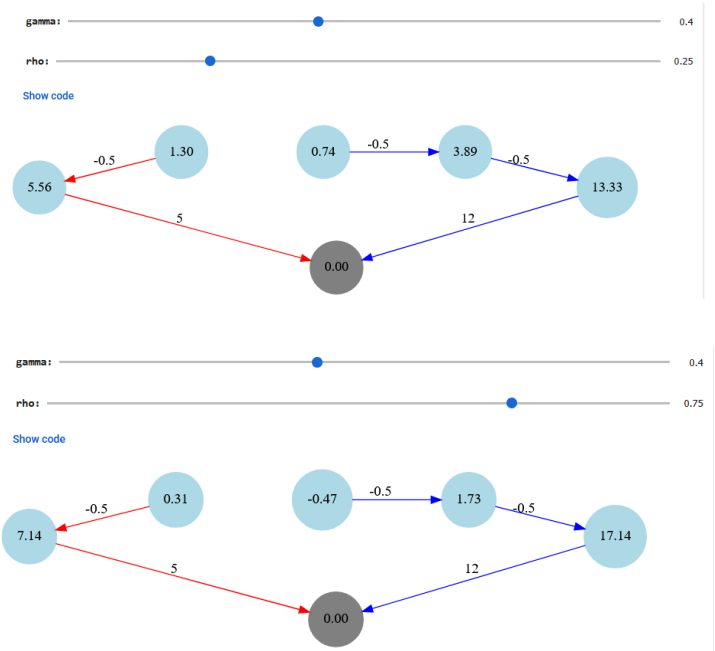

Figure 3: Illustration of different slipping factors on the MDP

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.

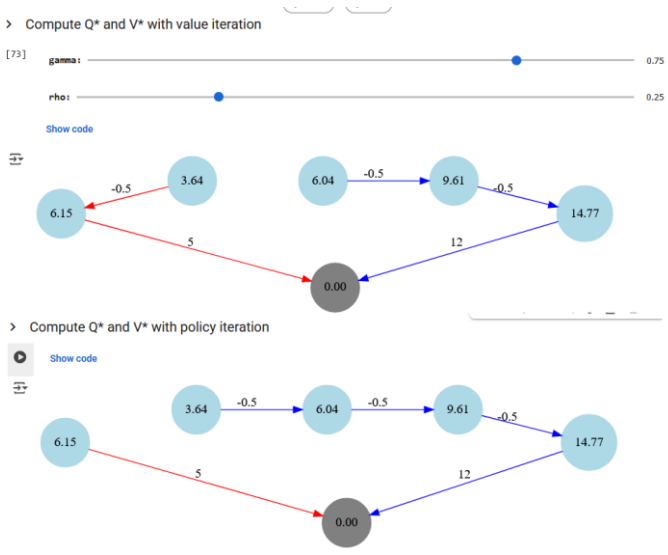

Figure 4: Value Iteration vs. Policy Iteration

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.

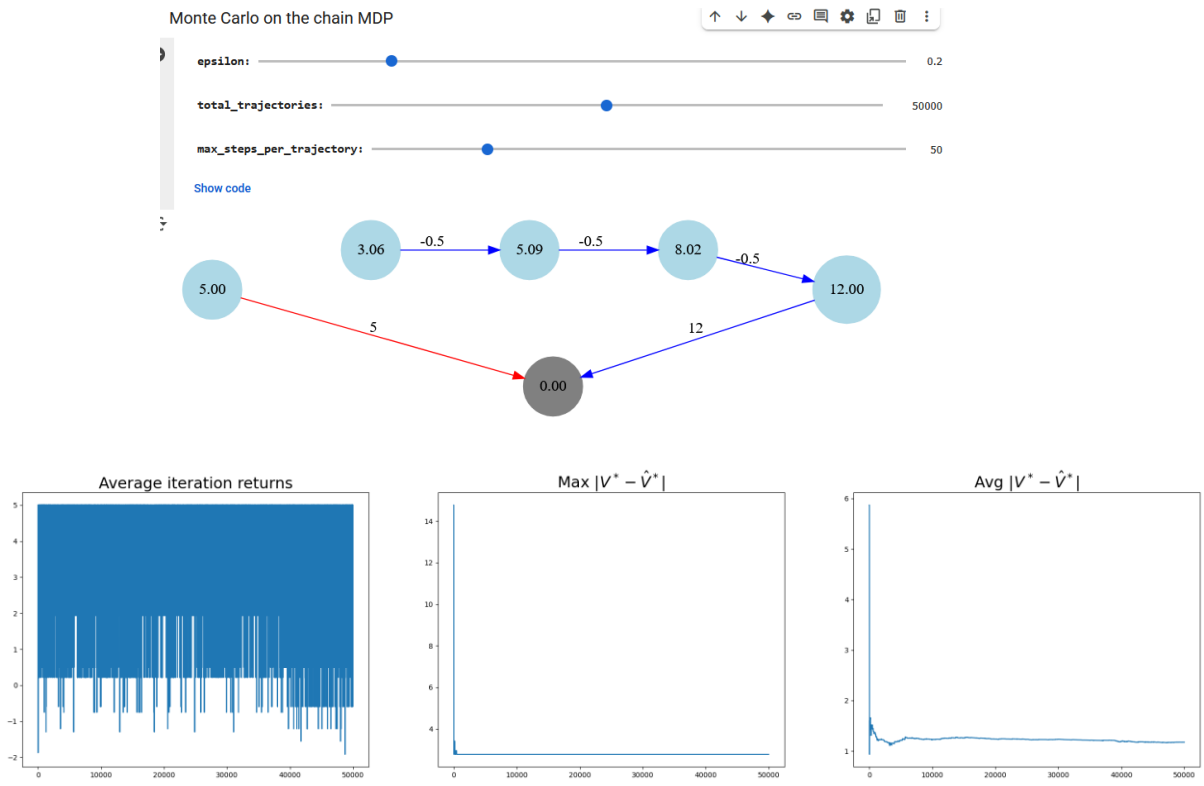

Figure 5: The Monte Carlo approach run on the MDP

This shows a Monte Carlo approach run with an epsilon value of 0.2 and total trajectory of 50000

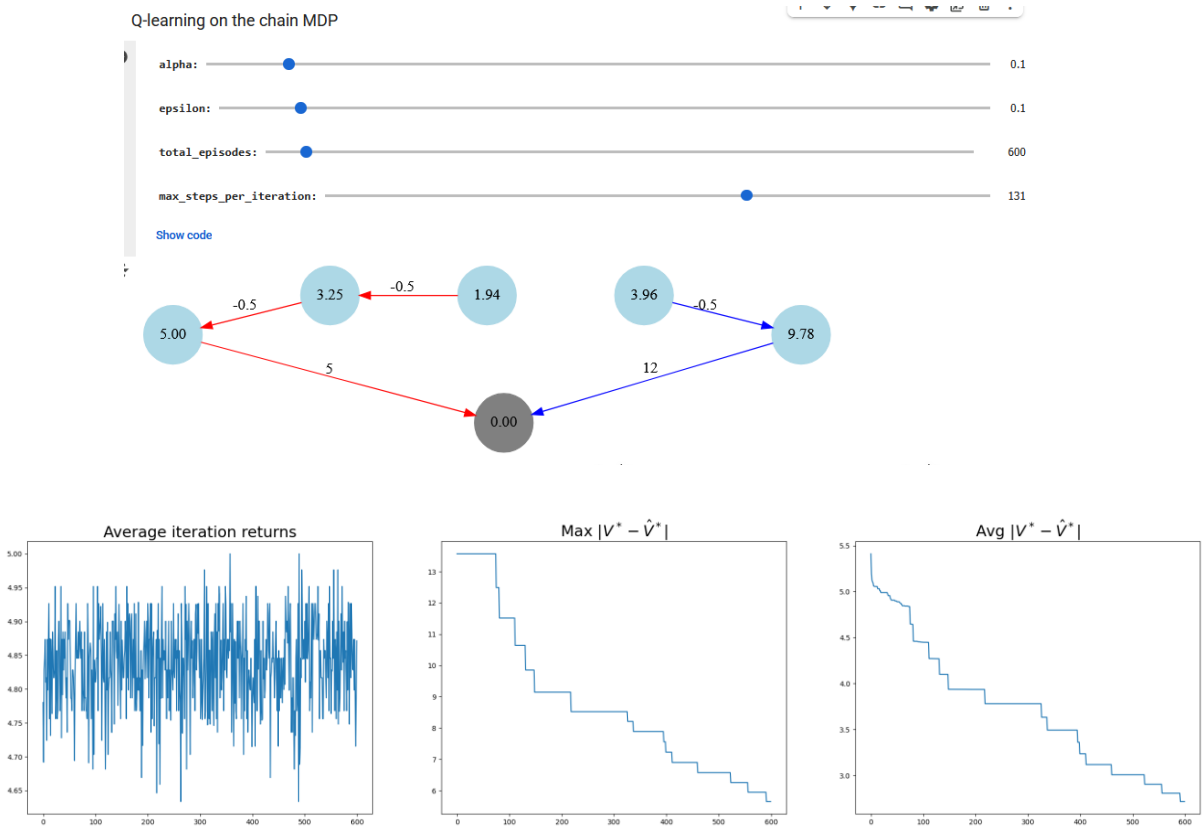

Figure 6: Q-Learning on the MDP

This shows a Q-Learning approach being performed on the chain MDP with alpha and epsilon of 0.1, total episodes 600, and 131 max steps per iteration.

The notebook ends with a short note on Deep Reinforcement Learning, which is RL Algorithms combined with Neural Networks. A CartPole demonstration is provided as an example.

From the outputs of these programs and the codes provided, we can gain an understanding of Reinforcement Learning and the concepts that drive it. We can also understand how to create these models for practical scenarios, and the different approaches (known vs. unknown environment, value vs. policy iterations) that can be applied based on the situation. We can observe the effect different parameters have on the models, and how they learn from the environment.

Code Source:

[1]https://colab.research.google.com/drive/1HG6MPM5GnDHmBaLsnRy9HObTmYq7g6Vk