

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.

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