me19b190-tutorial-2

February 10, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from typing import NamedTuple
from google.colab import output
```

[2]:
```
SEED = 0

BOARD_COL = 3
BOARD_ROW = 3
BOARD_SIZE = BOARD_COL * BOARD_ROW

"""
Game board and actions are: {q, w, e, a, s, d, z, x, c}

q | w | e
--|---|--
a | s | d
--|---|--
z | x | c
"""
ACTIONS_KEY_MAP = {'q': 0, 'w': 1, 'e': 2,
                   'a': 3, 's': 4, 'd': 5,
                   'z': 6, 'x': 7, 'c': 8}
```

[3]:
```
np.random.seed(SEED)
```

State Defination

[4]:
```
def print_state(board, clear_output=False):
  if clear_output:
    output.clear()
  for i in range(BOARD_ROW):
    print('-------------')
    out = '| '
    for j in range(BOARD_COL):
      if board[i, j] == 1:
          token = 'x'
      elif board[i, j] == -1:
```

```
          token = 'o'
      else:
          token = ' '  # empty position
      out += token + ' | '
    print(out)
  print('-------------')


class State:
  def __init__(self, symbol):
    # the board is represented by an n * n array,
    #  1 represents the player who moves first,
    # -1 represents another player
    #  0 represents an empty position
    self.board = np.zeros((BOARD_ROW, BOARD_COL))
    self.symbol = symbol
    self.winner = 0
    self.end = None

  @property
  def hash_value(self):
    hash = 0
    for x in np.nditer(self.board):
      hash = 3*hash + x + 1    # unique hash
    return hash

  def next(self, action: str):
    id = ACTIONS_KEY_MAP[action]
    i, j = id // BOARD_COL, id % BOARD_COL
    return self.next_by_pos(i, j)

  def next_by_pos(self, i: int, j: int):
    assert self.board[i, j] == 0
    new_state = State(-self.symbol)      # another player turn
    new_state.board = np.copy(self.board)
    new_state.board[i, j] = self.symbol  # current player choose to play at (i, j) pos
    return new_state

  @property
  def possible_actions(self):
    rev_action_map = {id: key for key, id in ACTIONS_KEY_MAP.items()}
    actions = []
    for i in range(BOARD_ROW):
      for j in range(BOARD_COL):
        if self.board[i, j] == 0:
          actions.append(rev_action_map[BOARD_COL*i+j])
```

```
    return actions

  def is_end(self):
    if self.end is not None:
      return self.end

    check = []
    # check row
    for i in range(BOARD_ROW):
      check.append(sum(self.board[i, :]))

    # check col
    for i in range(BOARD_COL):
      check.append(sum(self.board[:, i]))

    # check diagonal
    diagonal = 0; reverse_diagonal = 0
    for i in range(BOARD_ROW):
      diagonal += self.board[i, i]
      reverse_diagonal += self.board[BOARD_ROW-i-1, i]
    check.append(diagonal)
    check.append(reverse_diagonal)

    for x in check:
      if x == 3:
        self.end = True
        self.winner = 1   # player 1 wins
        return self.end
      elif x == -3:
        self.end = True
        self.winner = 2   # player 2 wins
        return self.end

    for x in np.nditer(self.board):
      if x == 0:          # play available
        self.end = False
        return self.end

    self.winner = 0       # draw
    self.end = True
    return self.end
```

Environment

[5]:
```
class Env:
  def __init__(self):
    self.all_states = self.get_all_states()
```

```
    self.curr_state = State(symbol=1)

  def get_all_states(self):
    all_states = {}  # is a dict with key as state_hash_value and value as State object.
    def explore_all_substates(state):
      for i in range(BOARD_ROW):
        for j in range(BOARD_COL):
          if state.board[i, j] == 0:
            next_state = state.next_by_pos(i, j)
            if next_state.hash_value not in all_states:
              all_states[next_state.hash_value] = next_state
              if not next_state.is_end():
                explore_all_substates(next_state)
    curr_state = State(symbol=1)
    all_states[curr_state.hash_value] = curr_state
    explore_all_substates(curr_state)
    return all_states

  def reset(self):
    self.curr_state = State(symbol=1)
    return self.curr_state

  def step(self, action):
    assert action in self.curr_state.possible_actions, f"Invalid {action} for the current state \n{self.curr_state.print_state()}"
    next_state_hash = self.curr_state.next(action).hash_value
    next_state = self.all_states[next_state_hash]
    self.curr_state = next_state
    reward = 0
    return self.curr_state, reward

  def is_end(self):
    return self.curr_state.is_end()

  @property
  def winner(self):
    result_id = self.curr_state.winner
    result = 'draw'
    if result_id == 1:
      result = 'player1'
    elif result_id == 2:
      result = 'player2'
    return result
```

Policy

[6]:
```python
class BasePolicy:
  def reset(self):
    pass

  def update_values(self, *args):
    pass

  def select_action(self, state):
    raise Exception('Not Implemented Error')
```

[7]:
```python
class HumanPolicy(BasePolicy):
  def __init__(self, symbol):
    self.symbol = symbol

  def select_action(self, state):
    assert state.symbol == self.symbol, f"Its not {self.symbol} symbol's turn"
    print_state(state.board, clear_output=True)
    key = input("Input your position: ")
    return key
```

[8]:
```python
class RandomPolicy(BasePolicy):
  def __init__(self, symbol):
    self.symbol = symbol

  def select_action(self, state):
    assert state.symbol == self.symbol, f"Its not {self.symbol} symbol's turn"
    return np.random.choice(state.possible_actions)
```

[9]:
```python
class ActionPlayed(NamedTuple):
  hash_value: str
  action: str


class MenacePolicy(BasePolicy):
  def __init__(self, all_states, symbol, tau=5.0):
    self.all_states = all_states
    self.symbol = symbol
    self.tau = tau

    # It store the number of stones for each action for each state
    self.state_action_value = self.initialize()
    # variable to store the history for updating the number of stones
    self.history = []

  def initialize(self):
    state_action_value = {}
    for hash_value, state in self.all_states.items():
```

```python
    # initially all actions have 0 stones
    state_action_value[hash_value] = {action: 0 for action in state.
↪possible_actions}
  return state_action_value

def reset(self):
  for action_value in self.state_action_value.values():
    for action in action_value.keys():
      action_value[action] = 0

def print_updates(self, reward):
  print(f'Player with symbol {self.symbol} updates the following history with␣
↪{reward} stone')
  for item in self.history:
    board = np.copy(self.all_states[item.hash_value].board)
    id = ACTIONS_KEY_MAP[item.action]
    i, j = id//BOARD_COL, id%BOARD_COL
    board[i, j] = self.symbol
    print_state(board)

def update_values(self, reward, show_update=False):
  # reward: if wins receive reward of 1 stone for the chosen action
  #          else -1 stone.
  # reward is either 1 or -1 depending upon if the player has won or lost the␣
↪game.

  if show_update:
    self.print_updates(reward)
  for item in self.history:

    # your code here
    self.state_action_value[item.hash_value][item.action] = self.
↪state_action_value[item.hash_value][item.action]+reward  # update␣
↪state_action with appropriate term.

  self.history = []

def select_action(self, state):  # Softmax action probability
  assert state.symbol == self.symbol, f"Its not {self.symbol} symbol's turn"
  action_value = self.state_action_value[state.hash_value]
  max_value = action_value[max(action_value, key=action_value.get)]
  exp_values = {action: np.exp((v-max_value) / self.tau) for action, v in␣
↪action_value.items()}
  normalizer = np.sum([v for v in exp_values.values()])
  prob = {action: v/normalizer for action, v in exp_values.items()}
  action = np.random.choice(list(prob.keys()), p=list(prob.values()))
  self.history.append(ActionPlayed(state.hash_value, action))
```

```
    return action
```

Game Board

[10]:
```
class Game:
  def __init__(self, env, player1, player2):
    self.env = env
    self.player1 = player1
    self.player2 = player2
    self.show_updates = False

  def alternate(self):
    while True:
      yield self.player1
      yield self.player2

  def train(self, epochs=1_00_000):
    game_results = []
    player1_reward_map = {'player1': 1, 'player2': -1, 'draw': 0}
    for _ in range(epochs):
      result = self.play()

      # if player1 wins add 1 stone for the action chosen
      player1_reward = player1_reward_map[result]
      player2_reward = -player1_reward   # if player2 wins add 1 stone

      self.player1.update_values(player1_reward)
      self.player2.update_values(player2_reward)

  def play(self):
    alternate = self.alternate()
    state = self.env.reset()
    while not self.env.is_end():
      player = next(alternate)
      action = player.select_action(state)
      state, _ = self.env.step(action)
    result = self.env.winner
    return result
```

Experiment

[11]:
```
env = Env()

player1 = MenacePolicy(env.all_states, symbol=1)
player2 = MenacePolicy(env.all_states, symbol=-1)
# player2 = RandomPolicy(symbol=-1)
```

[12]:
```python
game = Game(env, player1, player2)
game.train(epochs=1_00_000)
```

[14]:
```python
game_with_human_player = Game(env, player1, HumanPolicy(symbol=-1))

game_with_human_player.play()

result = env.winner
print(f"winner: {result}")

player1_reward_map = {'player1': 1, 'player2': -1, 'draw': 0}
player1.update_values(player1_reward_map[result], show_update=True)
```

```
-------------
| x | o | x |
-------------
| o | o | x |
-------------
|   | x |   |
-------------
Input your position: c
winner: draw
Player with symbol 1 updates the following history with 0 stone
-------------
|   |   | x |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
-------------
| x |   | x |
-------------
|   | o |   |
-------------
|   |   |   |
-------------
-------------
| x | o | x |
-------------
|   | o |   |
-------------
|   | x |   |
-------------
-------------
| x | o | x |
-------------
```

```
| o | o | x |
-------------
|   | x |   |
-------------
-------------
| x | o | x |
-------------
| o | o | x |
-------------
| x | x | o |
-------------
```

[13]:

```

```

orial-3-value-and-policy-iteration

February 10, 2023

[17]:
```python
import numpy as np
from enum import Enum
import copy
```

Consider a standard grid world, where only 4 (up, down, left, right) actions are allowed and the agent deterministically moves accordingly, represented as below. Here yellow is the start state and white is the goal state.

Say, we define our MDP as: - S: 121 (11 x 11) cells - A: 4 actions (up, down, left, right) - P: Deterministic transition probability - R: -1 at every step - gamma: 0.9

Our goal is to find an optimal policy (shown in right).

[18]:
```python
# Above grid is defined as below:
#    - 0 denotes an navigable tile
#    - 1 denotes an obstruction/wall
#    - 2 denotes the start state
#    - 3 denotes an goal state

# Note: Here the upper left corner is defined as (0, 0)
#       and lower right corner as (m-1, n-1)

# Optimal Path: RIGHT RIGHT UP UP LEFT LEFT UP UP UP UP UP UP LEFT LEFT DOWN␣
 ↪DOWN LEFT LEFT


GRID_WORLD = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
    [1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1],
    [1, 3, 0, 0, 1, 0, 1, 0, 1, 0, 1],
    [1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1],
    [1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
```

```
    [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
])
```

0.0.1 Actions

[19]:
```
class Actions(Enum):
  UP    = (0, (-1, 0))  # index = 0, (xaxis_move = -1 and yaxis_move = 0)
  DOWN  = (1, (1, 0))   # index = 1, (xaxis_move = 1 and yaxis_move = 0)
  LEFT  = (2, (0, -1))  # index = 2, (xaxis_move = 0 and yaxis_move = -1)
  RIGHT = (3, (0, 1))   # index = 3, (xaxis_move = 0 and yaxis_move = -1)

  def get_action_dir(self):
    _, direction = self.value
    return direction

  @property
  def index(self):
    indx, _ = self.value
    return indx

  @classmethod
  def from_index(cls, index):
    action_index_map = {a.index: a for a in cls}
    return action_index_map[index]
```

[20]:
```
# How to use Action enum
for a in Actions:
  print(f"name: {a.name}, action_id: {a.index}, direction_to_move: {a.
 ↪get_action_dir()}")

print("\n------------------------------------\n")

# find action enum from index 0
a = Actions.from_index(0)
print(f"0 index action is: {a.name}")
```

```
name: UP, action_id: 0, direction_to_move: (-1, 0)
name: DOWN, action_id: 1, direction_to_move: (1, 0)
name: LEFT, action_id: 2, direction_to_move: (0, -1)
name: RIGHT, action_id: 3, direction_to_move: (0, 1)

------------------------------------

0 index action is: UP
```

0.0.2 Policy

[21]:
```
class BasePolicy:
  def update(self, *args):
    pass

  def select_action(self, state_id: int) -> int:
    raise NotImplemented


class DeterministicPolicy(BasePolicy):
  def __init__(self, actions: np.ndarray):
    # actions: its a 1d array (|S| size) which contains action for each state
    self.actions = actions

  def update(self, state_id, action_id):
    assert state_id < len(self.actions), f"Invalid state_id {state_id}"
    assert action_id < len(Actions), f"Invalid action_id {action_id}"
    self.actions[state_id] = action_id

  def select_action(self, state_id: int) -> int:
    assert state_id < len(self.actions), f"Invalid state_id {state_id}"
    return self.actions[state_id]
```

0.0.3 Environment

[22]:
```
class Environment:
  def __init__(self, grid):
    self.grid = grid
    m, n = grid.shape
    self.num_states = m*n

  def xy_to_posid(self, x: int, y: int):
    _, n = self.grid.shape
    return x*n + y

  def posid_to_xy(self, posid: int):
    _, n = self.grid.shape
    return (posid // n, posid % n)

  def isvalid_move(self, x: int, y: int):
    m, n = self.grid.shape
    return (x >= 0) and (y >= 0) and (x < m) and (y < n) and (self.grid[x, y] !
↪= 1)

  def find_start_xy(self) -> int:
    m, n = self.grid.shape
```

```
    for x in range(m):
      for y in range(n):
        if self.grid[x, y] == 2:
          return (x, y)
    raise Exception("Start position not found.")

  def find_path(self, policy: BasePolicy) -> str:
    max_steps = 50
    steps = 0

    P, R = self.get_transition_prob_and_expected_reward()
    num_actions, num_states = R.shape
    all_possible_state_posids = np.arange(num_states)

    path = ""
    curr_x, curr_y = self.find_start_xy()
    while (self.grid[curr_x, curr_y] != 3) and (steps < max_steps):
      curr_posid = self.xy_to_posid(curr_x, curr_y)
      action_id = policy.select_action(curr_posid)
      next_posid = np.random.choice(
          all_possible_state_posids, p=P[action_id, curr_posid])
      action = Actions.from_index(action_id)
      path += f" {action.name}"
      curr_x, curr_y = self.posid_to_xy(next_posid)
      steps += 1
    return path

  def get_transition_prob_and_expected_reward(self):  # P(s_next | s, a), R(s,␣
↪a)
    m, n = self.grid.shape
    num_states = m*n
    num_actions = len(Actions)
    P = np.zeros((num_actions, num_states, num_states))
    R = np.zeros((num_actions, num_states))
    for a in Actions:
      for x in range(m):
        for y in range(n):
          xmove_dir, ymove_dir = a.get_action_dir()
          xnew, ynew = x + xmove_dir, y + ymove_dir  # find the new co-ordinate␣
↪after the action a

          posid = self.xy_to_posid(x, y)
          new_posid = self.xy_to_posid(xnew, ynew)


          if self.grid[x, y] == 3:
            # the current state is a goal state
```

```
            P[a.index, posid, posid] = 1
            R[a.index, posid] = 0
          elif (self.grid[x, y] == 1) or (not self.isvalid_move(xnew, ynew)):
            # the current state is a block state or the next state is invalid
            P[a.index, posid, posid] = 1
            R[a.index, posid] = -1
          else:
            # action a is valid and goes to a new position
            P[a.index, posid, new_posid] = 1
            R[a.index, posid] = -1
    return P, R
```

0.0.4 Policy Iteration

[23]:
```
def policy_evaluation(P: np.ndarray, R: np.ndarray, gamma: float,
                      policy: BasePolicy, theta: float,
                      init_V: np.ndarray=None):
  num_actions, num_states = R.shape

  # Please try different starting point for V you will find it will always
  # converge to the same V_pi value.
  if init_V is None:
    init_V = np.zeros(num_states)
  V = copy.deepcopy(init_V)

  delta = 100.0
  while delta > theta:
    delta = 0.0
    for state_id in range(num_states):
      action_id = policy.select_action(state_id)
      v_old = V[state_id]
      # Following equation is a different way of writing the same equation␣
↪given in the slide.
      # Note here R is an expected reward term.
      V[state_id] = R[action_id, state_id] + gamma * np.dot(P[action_id,␣
↪state_id], V)
      delta = max(delta, abs(V[state_id] - v_old))
  return V


def policy_improvement(P: np.ndarray, R: np.ndarray, gamma: float,
                       policy: BasePolicy, V: np.ndarray):
  num_actions, num_states = R.shape
  policy_stable = True
  for state_id in range(num_states):
    old_action_id = policy.select_action(state_id)
```

```
    # your code here
    new_action = old_action_id
    V_old = V[state_id]
    for action in Actions:
      action_id = action.index
      q_new = R[action_id, state_id] + gamma * np.dot(P[action_id, state_id], V)
      if q_new > V_old:
        new_action = action_id

    new_action_id = new_action # update new_action_id based on the value␣
 ↪function.

    policy.update(state_id, new_action_id)
    if old_action_id != new_action_id:
      policy_stable = False
  return policy_stable


def policy_iteration(P: np.ndarray, R: np.ndarray, gamma: float,
                     theta: float=1e-3, init_policy: BasePolicy = None):
  num_actions, num_states = R.shape

  # Please try exploring different policies you will find it will always
  # converge to the same optimal policy for valid states.
  if init_policy is None:
    # Say initial policy = all up actions.
    init_policy = DeterministicPolicy(actions=np.zeros(num_states, dtype=int))

  # creating a copy of a initial policy
  policy = copy.deepcopy(init_policy)
  policy_stable = False
  while not policy_stable:
    V = policy_evaluation(P, R, gamma, policy, theta)
    policy_stable = policy_improvement(P, R, gamma, policy, V)
  return policy, V
```

0.0.5 Value Iteration

[24]:
```
# Directly find the optimal value function
def get_optimal_value(P: np.ndarray, R: np.ndarray, gamma: float,
                      theta: float, init_V: np.ndarray=None):
  num_actions, num_states = R.shape

  # Please try different starting point for V you will find it will always
  # converge to the same V_star value.
  if init_V is None:
    init_V = np.zeros(num_states)
```

```python
  V = copy.deepcopy(init_V)

  delta = 100.0
  while delta > theta:
    delta = 0.0
    for state_id in range(num_states):
      v_old = V[state_id]
      q_sa = np.zeros(num_actions)
      for a in Actions:
        q_sa[a.index] = R[a.index, state_id] + gamma * np.dot(P[a.index,␣
↪state_id], V)
      V[state_id] = np.max(q_sa)
      delta = max(delta, abs(V[state_id] - v_old))
  return V


def value_iteration(P: np.ndarray, R: np.ndarray, gamma: float,
                    theta: float=1e-3, init_V: np.ndarray=None):
  V_star = get_optimal_value(P, R, gamma, theta, init_V)

  num_actions, num_states = R.shape
  policy = DeterministicPolicy(actions=np.zeros(num_states, dtype=int))
  for state_id in range(num_states):
    # Your code here

    best_action = policy.select_action(state_id)
    state_value_after_best_action = -np.inf
    n = P.shape[-1]
    n = int(n**0.5)
    temp_grid = np.empty((n,n))
    for action in Actions:
      x_curr, y_curr = env.posid_to_xy(state_id)
      x_dir,y_dir = action.get_action_dir()
      x_new, y_new = (x_curr+x_dir) , (y_curr+y_dir)
      if env.isvalid_move(x_new,y_new):
        state_transitioned = env.xy_to_posid(x_new,y_new)
        V_transition = V_star[state_transitioned]
        if V_transition > state_value_after_best_action:   #greedy with respect␣
↪to optimal value function
          best_action = action.index
          state_value_after_best_action = V_transition

    action_id = best_action # update the action_id based on V_star

    policy.update(state_id, action_id)

  return policy, V_star
```

0.0.6 Experiments

[25]:
```
def is_same_optimal_value(V1, V2, diff_theta=1e-3):
  diff = np.abs(V1 - V2)
  return np.all(diff < diff_theta)
```

[26]:
```
seed = 0
np.random.seed(seed)

gamma = 0.9
theta = 1e-5
```

[27]:
```
env = Environment(GRID_WORLD)
P, R = env.get_transition_prob_and_expected_reward()
```

Exp 1: Using Policy iteration algorithm find the optimal path from start to goal position

[28]:
```
# # Start with random choice of init_policy.
# One such choice could be: init_policy = np.ones(env.num_states, dtype=int)
# Start with your own choice of init_policy
init_policy = DeterministicPolicy(actions=np.ones(env.num_states, dtype=int))

pitr_policy, pitr_V_star = policy_iteration(P, R, gamma, theta=theta,␣
 ↪init_policy=init_policy)
pitr_path = env.find_path(pitr_policy)
print(pitr_path)
```

```
RIGHT RIGHT UP UP LEFT LEFT UP UP UP UP UP UP LEFT LEFT DOWN DOWN LEFT LEFT
```

Exp 2: Using value iteration algorithm find the optimal path from start to goal position

[29]:
```
vitr_policy, vitr_V_star = value_iteration(P, R, gamma, theta=theta)
vitr_path = env.find_path(vitr_policy)
print(vitr_path)
```

```
RIGHT RIGHT UP UP LEFT LEFT UP UP UP UP UP UP LEFT LEFT DOWN DOWN LEFT LEFT
```

Exp 3: Compare the optimal value function of policy iteration and value iteration algorithm

[30]:
```
is_same_optimal_value(pitr_V_star, vitr_V_star)
```

[30]:
```
True
```

Exp 4: Using initial guess for V as random values, find the optimal value function using policy evaluation and compare it with the optimal value function

[31]:
```python
# Start with random choice of init_V.
# One such choice could be: init_V = np.random.randn(env.num_states)
# Another choice could be: init_V = 10*np.ones(env.num_states)
# Start with your own choice of init_V
init_V = 10*np.ones(env.num_states) # your choice

V_star = policy_evaluation(P, R, gamma, pitr_policy, theta, init_V)
is_same_optimal_value(pitr_V_star, V_star)
```

[31]: True

Exp 5: Using initial guess for V as random values, find the optimal value function using get_optimal_value and compare it with the optimal value function

[32]:
```python
# Start with random choice.
# One such choice could be: init_V = np.random.randn(env.num_states)
# Another choice could be: init_V = 10*np.ones(env.num_states)
# Start with your own choice of init_V
init_V = 100*np.ones(env.num_states)

V_star = get_optimal_value(P, R, gamma, theta, init_V)
is_same_optimal_value(vitr_V_star, V_star)
```

[32]: True

[32]:
```python

```

Exp Optional: Try changing the grid by adding multiple paths to the goal state and check if our policy__iteration or value__iteration algorithm is able to find optimal path. Redo the above experiments.

- 1 way to add another path would be GRID_WORLD[4, 1] = 0

[32]:
```python

```