DA6400_MENACE_DA24S018

February 14, 2025

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

```
[6]: 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}
```

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

State Definition

```
[8]: 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:
```

```python
                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])
```

```python
        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

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

3

```python
        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

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

        def update_values(self, *args):
          pass

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

```python
[11]: 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
```

```python
[12]: 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)
```

```python
[13]: 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
      if (reward == 1):
        self.state_action_value[item.hash_value][item.action] += 1 # update
↪state_action with appropriate term.
      elif (reward == -1):
        self.state_action_value[item.hash_value][item.action] -= 1 # 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()])
```

```python
        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

```python
[14]: 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

```python
[15]: env = Env()

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

```python
# player2 = RandomPolicy(symbol=-1)
```

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

```python
[17]:  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)
```

```
-------------
| o |   | x |
-------------
|   | x | x |
-------------
|   |   | o |
-------------
Input your position: a
winner: player1
Player with symbol 1 updates the following history with 1 stone
-------------
|   |   | x |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
-------------
| o |   | x |
-------------
|   |   | x |
-------------
|   |   |   |
-------------
-------------
| o |   | x |
-------------
|   | x | x |
-------------
|   |   | o |
-------------
-------------
```

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

```
[18]: 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 |
-------------
| x | o |   |
-------------
|   | x | o |
-------------
Input your position: z
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 |
-------------
| x | o |   |
-------------
|   | x | o |
-------------
-------------
| x | o | x |
-------------
| x | o | x |
-------------
| o | x | o |
-------------
```

```python
[1]: from google.colab import drive
     drive.mount('/content/drive')
```

Mounted at /content/drive

```python
[19]: !jupyter nbconvert --to pdf /content/drive/MyDrive/DA6400_MENACE_DA24S018.ipynb
```

```
[NbConvertApp] Converting notebook
/content/drive/MyDrive/DA6400_MENACE_DA24S018.ipynb to pdf
[NbConvertApp] Writing 57100 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 56574 bytes to
/content/drive/MyDrive/DA6400_MENACE_DA24S018.pdf
```

```
[ ]:
```