```python
import numpy as np
import matplotlib.pyplot as plt
from typing import NamedTuple
from google.colab import output
```

```python
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}
```

```python
np.random.seed(SEED)
```

State Definition

Tic-Tac-Toe game state with methods for updating the board, checking the game's status, and determining valid actions

```python
#Prints the current board state, Uses symbols 'x' for player 1, 'o' for player 2, ' ' for empty
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('-------------')
```

```python

# State Class : Represents the game state
# Stores:
# board → 3×3 NumPy array (1 = Player 1, -1 = Player 2, 0 = Empty)
# symbol → Current player’s turn
# winner → Stores the winner (1, 2, or 0 for draw)
# end → Boolean flag indicating if the game is over
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

  #to generates a unique hash for board states using a base-3 formula.
  @property
  def hash_value(self):
    hash = 0
    for x in np.nditer(self.board):
      hash = 3*hash + x + 1   # unique hash
    return hash

  #to takes an action key, Converts it to board coordinates, Calls next_by_pos(i, j).
  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)

#to generates a new game state after a move, asserts that the move is valid,swaps turns (self.symbol → -self.symbol).
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

#to returns available moves based on empty board positions
@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

#to check rows and columns for a sum of 3 or -3 (win condition), diagonals for a winning condition, if moves are availa
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

```
class Env:
  #to initializes the environment, calls get_all_states() to precompute all possible game states, sets curr_state to the
  def __init__(self):
    self.all_states = self.get_all_states()
    self.curr_state = State(symbol=1)

  #to precomputes all valid board states and stores them in all_states, uses a recursive function (explore_all_substates(
  # the dictionary all_states: Key is Unique "hash_value" of a board state and Value is Corresponding "State object".
  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

  # to resets the game to the initial state and returns the new starting State
  def reset(self):
    self.curr_state = State(symbol=1)
    return self.curr_state

  #to moves to the next state based on a given action, Checks if the move is valid (must be in possible_actions), Retriev
  #updates curr_state and returns the new state after the move and a reward (currently 0, can be modified for RL).
  # def step(self, action):
  #   assert action in self.curr_state.possible_actions, f"Invalid {action} for the current state \n{self.curr_state.prir
  #   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 step(self, action):
    assert action in self.curr_state.possible_actions, f"Invalid move {action}"
    next_state_hash = self.curr_state.next(action).hash_value
    next_state = self.all_states[next_state_hash]
    self.curr_state = next_state

    if self.curr_state.is_end():
        if self.curr_state.winner == 1:
            reward = 1   # Player 1 wins
        elif self.curr_state.winner == 2:
            reward = -1  # Player 2 wins
        else:
            reward = 0   # Draw
    else:
        reward = 0  # No immediate reward for non-terminal moves

    return self.curr_state, reward

  #to checks if the game has ended
  def is_end(self):
    return self.curr_state.is_end()

  #to returns 'player1', 'player2', or 'draw' based on curr_state.winner.
  @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

```

Right now, reward = 0 (neutral), Update the rewards +1 if player wins. -1 if player loses.

Policy

```
class BasePolicy:
  def reset(self):
    pass

  def update_values(self, *args):
    pass

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

```
# human player to interact with the Tic-Tac-Toe environment by manually choosing actions
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
```

```
#randomly selects a valid action for the given player
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)
```

```
class ActionPlayed(NamedTuple):
  hash_value: str
  action: str
  # reward: float  #reward received after action

```

```
class MenacePolicy(BasePolicy):
  def __init__(self, all_states, symbol, tau=5.0):
    self.all_states = all_states
    self.symbol = symbol
    self.tau = tau #tau is fixed, a decaying tau (annealing) can improve learning,this makes MENACE explore early, exploi

    # 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 = []

  #to initialize state-action values i.e. number of stones/beads
  def initialize(self):
    state_action_value = {}
    for hash_value, state in self.all_states.items():
      # initially all actions have 0 stones
      state_action_value[hash_value] = {action: 0 for action in state.possible_actions}
    return state_action_value

  #resets all state-action values to zero
  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)


  #to  adjusts stone counts based on game results
  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:

        # ensure minimum value is 1 (MENACE cannot have negative beads)
        self.state_action_value[item.hash_value][item.action] = max(1, self.state_action_value[item.hash_value][item.acti

    # clear history after updating values
    self.history = []

  def select_action(self, state):
    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
```

A MENACE (Matchbox Educable Noughts and Crosses Engine)-style learning agent for Tic-Tac-Toe:

It uses reinforcement learning principles by updating the number of "beads" (probability weights) in each matchbox (state-action pair) based on the game outcome. This policy learns by adjusting state-action values using a softmax-based selection.

Game Board

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

  #to yields players in an alternating order (Player-1 to Player-2 to Repeat)
  def alternate(self):
    while True:
      yield self.player1
      yield self.player2

  def train(self, epochs=1_00_000):
    game_results = {'player1': 0, 'player2': 0, 'draw': 0}
    player1_reward_map = {'player1': 1, 'player2': -1, 'draw': 0}

    for _ in range(epochs):
      result = self.play()
      game_results[result] += 1

      # 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)

    print(f"Training complete! Results over {epochs} games:")
    print(game_results)

  #to runs a single game until the end state is reached
  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

```
env = Env()

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

#training MENACE (player1) through self-play, it learns by playing against another MENACE agent
game = Game(env, player1, player2)
game.train(epochs=1_00_000)

```

```
Training complete! Results over 100000 games:
{'player1': 95766, 'player2': 2113, 'draw': 2121}
```

There is an issue with the MENACE training process due to an overflow in the exponential calculation. The values used in the softmax function are growing too large, causing invalid probability distributions (NaN values).

I will adjust the update mechanism to keep the values stable.

```python
#testing MENACE against a human after training, we set up a game where player1 (trained MENACE) faces a human (HumanPolic

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)

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

# q | w | e
# --|---|--
# a | s | d
# --|---|--
# z | x | c
# """


```

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

```python
#testing MENACE against a human after training, we set up a game where player1 (trained MENACE) faces a human (HumanPolic
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)

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

# q | w | e
# --|---|--
# a | s | d
# --|---|--
# z | x | c
# """


```

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

```python
#testing MENACE against a human after training, we set up a game where player1 (trained MENACE) faces a human (HumanPolic
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)

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

# q | w | e
# --|---|--
# a | s | d
# --|---|--
# z | x | c
# """


```

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

modify the MENACE model to use k = 3

```python
class MenacePolicy(BasePolicy):
  def __init__(self, all_states, symbol, tau=5.0):
    self.all_states = all_states
    self.symbol = symbol
    self.tau = tau #tau is fixed, a decaying tau (annealing) can improve learning,this makes MENACE explore early, exploi

    # 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 = []

  #to initialize state-action values i.e. number of stones/beads
  def initialize(self):
      state_action_value = {}
      for hash_value, state in self.all_states.items():
          # initially, each action starts with k = 3 stones (instead of 0)
          state_action_value[hash_value] = {action: 3 for action in state.possible_actions}
      return state_action_value


  #resets all state-action values to zero
  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)


  #to  adjusts stone counts based on game results
  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:

        # ensure minimum value is 1 (MENACE cannot have negative beads)
        self.state_action_value[item.hash_value][item.action] = max(1, self.state_action_value[item.hash_value][item.acti

    # clear history after updating values
    self.history = []

  def select_action(self, state):
    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
```

```python
env = Env()

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


#training MENACE (player1) through self-play, it learns by playing against another MENACE agent
game = Game(env, player1, player2)
game.train(epochs=1_00_000)
```

```
Training complete! Results over 100000 games:
{'player1': 95962, 'player2': 1358, 'draw': 2680}
```

```python
#testing MENACE against a human after training, we set up a game where player1 (trained MENACE) faces a human (HumanPolic
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)

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

# q | w | e
# --|---|--
# a | s | d
# --|---|--
# z | x | c
# """
```

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

Currently, MENACE always follows tau=5.0, if we change tau **it** can encourage early exploration, later exploitation

```
def train(self, epochs=100_000):
    for epoch in range(epochs):
      self.player1.tau = max(0.1, 5.0 * np.exp(-0.0001 * epoch))  # reduce tau
      self.player2.tau = self.player1.tau
      result = self.play()

```

```
env = Env()

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


#training MENACE (player1) through self-play, it learns by playing against another MENACE agent
game = Game(env, player1, player2)
game.train(epochs=1_00_000)
```

```
Training complete! Results over 100000 games:
{'player1': 94874, 'player2': 1568, 'draw': 3558}
```

```
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)

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

# q | w | e
# --|---|--
# a | s | d
# --|---|--
# z | x | c
# """


```

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

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

Initial Number of Stones, At the beginning, every possible action for each state starts with 0 stones. Updating Stones After a Game

When the policy wins,Each action taken during the game is rewarded with +1 stone.

When the policy loses,Each action taken during the game is penalized with -1 stone, but the minimum number of stones is 1 (i.e., MENACE cannot have negative beads). When the game ends in a draw,No changes are made to the stone count.