```
import gym
import numpy as np
import random
from stable_baselines3 import PPO
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
import torch
import torch.nn as nn

episode_durations = [] #only used for plotting how long the games are
episode_rewards = [] #only used for finding the final rewards of the game
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display
plt.ion()

class BSGameEnv(gym.Env):

    def plot_rewards(show_result=False): #plots the rewards
        plt.figure(1)
        rewards = torch.tensor(episode_rewards, dtype=torch.float)
        if show_result:
            plt.title('Result')
        else:
            plt.clf()
            plt.title('Training...')
        plt.xlabel('Episode')
        plt.ylabel('Episodic Reward')
        plt.plot(rewards.numpy())
        # Take 100 episode averages and plot them too
        if len(rewards) >= 5:
            means = rewards.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(means.numpy())

        plt.pause(0.001)  # pause a bit so that plots are updated
        if is_ipython:
            if not show_result:
                display.display(plt.gcf())
```

```
                display.clear_output(wait=True)
            else:
                display.display(plt.gcf())

    def plot_durations(show_result=False): #plots how long the games are
        plt.figure(2)
        durations_t = torch.tensor(episode_durations, dtype=torch.float)
        if show_result:
            plt.title('Result')
        else:
            plt.clf()
            plt.title('Training...')
        plt.xlabel('Episode')
        plt.ylabel('Duration')
        plt.plot(durations_t.numpy())
        # Take 100 episode averages and plot them too
        if len(durations_t) >= 5:
            means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(means.numpy())

        plt.pause(0.001)  # pause a bit so that plots are updated
        if is_ipython:
            if not show_result:
                display.display(plt.gcf())
                display.clear_output(wait=True)
            else:
                display.display(plt.gcf())

    def addCards(self, player, gamecards): #adds two card piles together
in order, so [1,2,3]+[2,3,4] = [3,5,7]
        newState = [a + b for a, b in zip(player, gamecards)] #adds the
two pairs of cards together

        return newState

    def findTheStateOfTheGame(self, playerCards):
        print("player cards", playerCards)
        stateOfTheGame = [0] * 13  # makes a blank list with 0's
        for card in playerCards:
```

```
            stateOfTheGame[card] += 1  # adds a value for the cards (0-12
anlso 0 -12 )
        return stateOfTheGame
    def resetDuplicate(self):
        self.cardsInTheGame = []
        self.pastPutdown = [1,0,0] #just sets the past put down to nothing
        self.numberOfCards = random.randint(30, 45) #finds a random amount
of cards to put down
        self.cards_in_game = [2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5,
5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9,
                    10,10,10,10,11,11,11,11,12,12,12,12,0,0,0,0,1,1,1,1]

        self.cardsInTheGame = random.sample(self.cards_in_game,
self.numberOfCards) #samples the numbers 30-45
        #print("cards in the game:", self.cardsInTheGame)
        if len(self.cardsInTheGame) % 2: #so that there is an even number
of cards
            self.cardsInTheGame.pop()
        #print(len(self.cardsInTheGame)) #prints how many cards in the
game



        self.stateOfPlayer1 = random.sample(self.cardsInTheGame,
len(self.cardsInTheGame) // 2) #finds player 1 state by sampling half the
cards

        self.stateOfPlayer1 =
self.findTheStateOfTheGame(self.stateOfPlayer1) #reorganizes teh list
        #print("player 1:", self.stateOfPlayer1)
        #could do more than 13 cards but idk
        self.cardsInTheGame =
self.findTheStateOfTheGame(self.cardsInTheGame) #reorganizes the total
list
        self.stateOfPlayer2 = [abs(a - b) for a, b in
zip(self.cardsInTheGame, self.stateOfPlayer1)] #subtracts player 1 from
the total list
        #print("player 2:", self.stateOfPlayer2)

        self.stateOfTheGame = np.zeros(13, dtype=np.int32) #makes the
original state of the game 0's, nothing has been put down yet
```

```
        self.number_for_player1 = self.numberOfCards // 2 #assigns the
initial numbers so the agent knos how much it has
        self.number_for_player2 = self.numberOfCards // 2


        self.action_space = gym.spaces.MultiDiscrete((3, 13, 4, 3, 14, 4,
3, 13, 4, 3, 13, 4)) #assigns the action space, 3(bs, nothing, action)
13(ace, 2, 3 etc.. ) 4[1,2,3,4 cards to put down]
        #the agent will see how much it has, how much player 2 has, how
much is in the stack, and the current cars it's on
        self.fixednumber_for_player1 = [self.number_for_player1]  #
Initialize as a list of 13 zeros

        self.observation_space = gym.spaces.Dict({
            'player1_count': gym.spaces.Box(low=0,
high=self.numberOfCards, shape=(13,), dtype=np.int32),
            'player1_cards': gym.spaces.Box(low=0,
high=self.numberOfCards, shape=(1,), dtype=np.int32),
            'player2_count': gym.spaces.Box(low=0,
high=self.numberOfCards, shape=(1,), dtype=np.int32),
            'stack_count': gym.spaces.Box(low=0, high=self.numberOfCards,
shape=(1,), dtype=np.int32),
            'current_card': gym.spaces.Box(low=0, high=13, shape=(1,),
dtype=np.int32) #the position of the current card
        })

        self.stepsDone = 0 #sets the steps done to 0
        self.reward = 0 #makes the reward 0
        self.pastBSTrue = False #makes a global variable to check if bs
has been put down before

        return

    def __init__(self):
        super(BSGameEnv, self).__init__()
        self.resetDuplicate()

    def bs(self, player_state, current_card, past_put_down):
        bs_true = False  # Assume BS is true until proven otherwise
```

```
        index_value = (current_card - 1) % 13 #card to be looked at
        for i in range(len(past_put_down) - 1): #looks at what was put
down
            if past_put_down[i] %3 == 2: #if it's an action card
                if past_put_down[i % 3 + 1] != index_value:  # If any
put-down does not match the current card index
                    bs_true = True  # Set BS to false
                    break  # Exit the loop since BS is already false
        player_state = self.addCards(player_state, self.stateOfTheGame)

        if bs_true:

            return True, player_state
        else:
            return False, player_state

    def step(self, action):
        #print("state of player 1:", self.stateOfPlayer1)
        #print("state of player 2:", self.stateOfPlayer2)
        self.stepsDone += 1
        BS_DECISION = False
        #bs_call = False
        bs_call, action_string = self.translate_actions_to_string(action)
#finds if the action has a "bs" and also gets a LIST(not string)
        #print(bs_call, "1. bs call", "action string:", action_string)
        if (bs_call ):
            number_cards_put_down = 0 #no cards are put down since the
agent said "bs"
            addCards = sum(self.stateOfTheGame) #find the total amount of
cards
            #print("past put down!", self.pastPutdown) #should be scripted
            #print("card it's on!", (self.stepsDone-2)%13)
            #print("player 1 state!", self.stateOfPlayer1)
            BS_DECISION, playerState = self.bs(self.stateOfPlayer1,
(self.stepsDone -1) % 13, self.pastPutdown) #finds out who told the truth
            #print("2. is it a bs?", BS_DECISION, "the card it's on: ",
(self.stepsDone-2)%13, "past put down", self.pastPutdown ) #subs 1 bc
everything
            #print('game state', self.stateOfTheGame)
            #print('expected game state', playerState)
```

```
            if not(BS_DECISION): #if it was nottt bs
                #print("3. the agent takes the cards :(( ")
                self.reward -=0.03
                self.stateOfPlayer1 = playerState #the agent takes the
cards
                self.number_for_player1 += addCards #the agent get's all
the cards
                self.stateOfTheGame = [0,0,0,0,0,0,0,0,0,0,0,0,0] #the
game is set to 0s
                order_card = np.zeros(13, dtype=np.int32)
                order_card[self.stepsDone % 13] = 1  # Set the current
card position to 1
                order_card = [self.stepsDone %13 -1]
                observation = {
                    'player1_count': self.stateOfPlayer1.copy(),
                    'player1_cards': [self.number_for_player1].copy(),
                    'player2_count': [self.number_for_player2].copy(),
                    'stack_count': [len(self.stateOfTheGame)].copy(),
                    'current_card': order_card.copy()  # Use an array to
represent the current card
                }

                done = (self.number_for_player1 <= 0) or
(self.number_for_player2 <= 0) or self.stepsDone > 1000
                reward = self.get_reward(done)
                self.stateOfTheGame = [0,0,0,0,0,0,0,0,0,0,0,0,0]
                return observation, reward, done, {} #restarts, since obvv
the agent messed up
            else:
                #print("3. the scripted player takes the cards :)) ")
                self.reward +=0.03
                self.stateOfPlayer2 = self.addCards(self.stateOfPlayer2,
self.stateOfTheGame) #player 2 takes the cards
                self.number_for_player2 += addCards #player 2 gets more
cards
                self.stateOfTheGame = [0,0,0,0,0,0,0,0,0,0,0,0,0]

        else:  # if there is no BS
            actual_action_string = []
            number_cards_put_down = 0
```

```
            for i in range(len(action_string)-2):  # Loop through the
action string with step size 3
                if i%3 == 2:  # If action is to put down cards
                    type_of_card = action_string[i + 1] #everything
shifted back
                    number_of_card = action_string[i + 2]

                    if self.stateOfPlayer1[type_of_card-1] >=
number_of_card:  # Check if player has enough cards
                        self.stateOfPlayer1[type_of_card-1] -=
number_of_card #if it does subtract
                        self.stateOfTheGame[type_of_card-1] +=
number_of_card #add to the card stack
                        #print(self.stateOfTheGame) #print the new state
                        self.number_for_player1 -= number_of_card
#subtracts from player 1's cards

                        number_cards_put_down += number_of_card #records
how many cards were put down
                        if(number_cards_put_down>4):
                            break
                        actual_action_string.append(2) #ritten out that it
says what was done previouslyyy
                        actual_action_string.append(type_of_card)
                        actual_action_string.append(number_of_card)

        self.stepsDone +=1 #the agent has taken it's turn!!
        BS_DECISION = False

        #SCRIPTED AGENT MOVE!! THE AI AGENT HAS JUST MOVED AND NOW IT"S
THE SCRIPTED ONE"S TURN!!
        bs_random = random.randint(1, 5)
        # if (number_cards_put_down >4 ): #if it's greater than 4 then
automatically bs !
        #     self.reward -=0.3 #negative reward for automatically getting
a bs :((
        #     bs_random = 3

        if (bs_call == True) or (number_cards_put_down ==0):
            bs_random = 2 #can't do bs if the past card had bs also
```

```
        if bs_random == 3:
            BS_DECISION, stateOfPlayer = self.bs(self.stateOfPlayer2,
(self.stepsDone - 1) % 13, actual_action_string) #figures out if the agent
lied
            #print("2. is it a bs?", BS_DECISION, "the card it's on: ",
(self.stepsDone-2)%13, "past put down", actual_action_string ) #what the
agent actually put down
            addCards = sum(self.stateOfTheGame)
            if BS_DECISION:
                #print("the agent lied and takes the cards!! :((")
                self.reward -=0.03 #if the agent lied :((
                self.stateOfPlayer1 = self.addCards(self.stateOfPlayer1,
self.stateOfTheGame) #player 1 takes the cards
                self.number_for_player1 += addCards
                self.stateOfTheGame = np.zeros(13, dtype=np.int32)
            else: #the agent told the truth
                #print("the scripted agent and takes the cards!! :))")
                self.reward +=0.03
                self.stateOfPlayer2 = stateOfPlayer
                self.number_for_player2 += addCards
                self.stateOfTheGame = np.zeros(13, dtype=np.int32)

            self.pastPutdown = [0,0,0] #0 = bs, and 0 cards and 0 amount

        else: #if there is no bs
            positions_of_i = (self.stepsDone-1) % 13
            number_of_cards = self.stateOfPlayer2[positions_of_i] #finds
out if the scripted player has cards in the desired spot (like it's 2's to
put down)
            randomNumber = 0
            if number_of_cards > 0:
                    randomNumber = random.randint(1, number_of_cards)
#just puts down a random amount of cards possible
            else:
                number_of_cards = 0 #sets it to 0 and finds a spot that
does nottt have 0 cards

                while number_of_cards<=0: #hile something has 0 cards
                    positions_of_i = (positions_of_i +1)% 13
```

```
                    number_of_cards = self.stateOfPlayer2[positions_of_i]
#finds how many cards are at that position

                if number_of_cards >0: #once it breaks the past loop
                    randomNumber = random.randint(1, number_of_cards)
                else:
                    randomNumber = 0
            self.stateOfPlayer2[positions_of_i]-= randomNumber #subtracts
the cards
            self.stateOfTheGame[positions_of_i] += randomNumber
            self.number_for_player2 -= randomNumber
            self.pastPutdown = [2, positions_of_i, randomNumber] #writes
down the move it took
        order_card = np.zeros(13, dtype=np.int32)
        order_card = [self.stepsDone % 13-1]  # Set the current card
position to 1
        observation = {
            'player1_count': self.stateOfPlayer1.copy(),
            'player1_cards': [self.number_for_player1].copy(),
            'player2_count': [self.number_for_player2].copy(),
            'stack_count': [len(self.stateOfTheGame)].copy(),
            'current_card': order_card.copy()  # Use an array to represent
the current card
        }

        done = (self.number_for_player1 <= 0 ) or (self.number_for_player2
<= 0 ) or self.stepsDone > 1000
        if done:
            print("number for player 1:", self.number_for_player1)
            print("number for player 2:", self.number_for_player2)
            print("steps done:", self.stepsDone)
        reward = self.get_reward(done)
        return observation, reward, done, {}

    def get_reward(self, done):
        if done:
            print("player 1 state:", self.stateOfPlayer1)
            print("player 2 state:", self.stateOfPlayer2)
            print("state of the game:", self.stateOfTheGame)
```

```
            if self.number_for_player1 <= 0:
                self.reward += 5
            elif self.number_for_player2 <= 0:
                self.reward-= 5
            else:
                if (self.number_for_player1+3)<self.number_for_player2:
                    self.reward += 1.0
                else:
                    self.reward-= 1.0
            episode_durations.append(self.stepsDone)
            episode_rewards.append(self.reward)

            print("player 1 number", self.number_for_player1, "player 2
number", self.number_for_player2)
            return self.reward
        else:
            return 0

    def reset(self): #same thing as the initialize method!!
        self.resetDuplicate()

        order_card = np.zeros(13, dtype=np.int32)
        order_card[self.stepsDone % 13] = 1  # Set the current card
position to 1
        order_card = [self.stepsDone%13 -1]
        observation = {
            'player1_count': self.stateOfPlayer1.copy(),
            'player1_cards': [self.number_for_player1].copy(),
            'player2_count': [self.number_for_player2].copy(),
            'stack_count': [len(self.stateOfTheGame)].copy(),
            'current_card': order_card.copy()  # Use an array to represent
the current card
        }

        return observation

    def render(self, mode='human'): #doesn't matter just prints if wanted
        print("Number of cards for player 1:", self.number_for_player1)
        print("Number of cards for player 2:", self.number_for_player2)
```

```
    def close(self): #sorta needed for the structure
        pass

    def translate_actions_to_string(self, action):
        bs_call = False
        order  = [0,1,2,3,4,5,6,7,8,9,10,11,12]
        action_string = []
        positions_of_bs = self.pastBSTrue #if there was a bs previously
one can't be put down now

        if (positions_of_bs == True): #the agent can't put down bs
            #print("no bs available :((")
            bs_call = False
            for i in range(len(action)):
                if i % 3 == 0: #if it's a multiple of 3
                    if action[i] == 0: #0 means it's bs
                        #action_string.append([0])
                        i+=3 #excludes the bs if it's impossible to put
down bs
                        #action_string = "bs"
                        break
                    elif action[i] == 1: #1 means no action
                        action_string.append(1)
                    elif action[i] == 2: #2 means to take an action
                        action_string.append(2)
                elif i % 3 == 1:
                    action_string.append( action[i]) # appends the number
                elif i % 3 == 2:
                    action_string.append( action[i]) #appends the number
of cardds
        else: #does the same thing except doesn't exclude bs
            for i in range(len(action)):
                if i % 3 == 0:
                    if action[i] == 0:
                        bs_call = True #shows bs call as true
                        action_string.append(0)

                        #break
                    elif action[i] == 1:
                        action_string.append(1)
```

```
                    elif action[i] == 2:
                        action_string.append(2)
                elif i % 3 == 1:
                    action_string.append( action[i])
                elif i % 3 == 2:
                    action_string.append( action[i])
        return  bs_call, action_string
# Training loop

env = BSGameEnv()
model = PPO('MultiInputPolicy', env, verbose=1)
model.load("Final_Test5")

model.learn(total_timesteps=int(2e6))
model.save("Final_Test7")  # Saves the final policy

# loop that actually plays the game
for i in range(1):
    obs = env.reset()
    term = False
    score = 0
    ep_len_mean = []
    while not term:
        action, _ = model.predict(obs)
        obs, rew, term, _ = env.step(action)
        #print("action taken:", action)
        score += rew
        if term:a
            ep_len_mean.append(env.stepsDone)
            print("IT IS DONE!!!!")
env.plot_durations()
env.plot_rewards()
plt.ioff()
plt.show() #shows the average rewards and durations
env.close()
```