CSE 584: Homework- 2 Reinforcement Learning

Muni Bhavana konidala
mzk6126@psu.edu

ABSTRACT:

The codebase implements a reinforcement learning system using the Deep SARSA algorithm for an agent navigating a grid-based environment. It is all about teaching a computer agent to learn how to reach a goal in a small, simple grid world while avoiding obstacles. It is a 5x5 grid, where the agent can move up, down, left, or right. The agent's main objective is to reach a specific "goal" spot in the grid, which gives a reward. However, there are also some "obstacle" spots that give penalties if the agent steps on them.
The agent uses a learning method called **Deep SARSA** to understand which moves help it reach the goal. To do this, the agent relies on a neural network, to guess which actions will get the best results in different situations. With each move, the agent learns from rewards and penalties, which it uses to improve its guesses over time. To make sure it explores different paths, it sometimes chooses random moves (exploration), but over time it starts to rely more on what it has learned (exploitation).

The code has two main parts:

1. **The `DeepSARSAAgent`**: This part contains the agent's brain. It includes the neural network that helps the agent learn which moves to make, and it also decides when the agent should explore new moves versus sticking to what it has learned.
2. **The Env (Environment)**: This part sets up the grid and manages the goal, obstacles, and rewards. Every time the agent makes a move, the environment gives feedback like telling the agent where it is now, whether it hit an obstacle, and if it reached the goal. Key functions in this environment, like step() and get_state(), help the agent know its current state and what it might earn or lose from different moves.

Environment.py:

```
# Import necessary libraries
import time
import numpy as np
import tkinter as tk            # For creating GUI applications
from PIL import ImageTk, Image    # For handling and displaying images in the GUI

# Defined constants
```

```
PhotoImage = ImageTk.PhotoImage  # Alias for image handling in tkinter
UNIT = 50                        # Size of each grid unit in pixels
HEIGHT = 5                       # Height of the grid in units
WIDTH = 5                        # Width of the grid in units

# Set a random seed for reproducibility of random operations
np.random.seed(1)

# Define the environment class for the grid-based world
class Env(tk.Tk):
    def __init__(self):
        # Initialize tkinter framework
        super(Env, self).__init__()
        # Defined possible actions: up, down, left, right
        self.action_space = ['u', 'd', 'l', 'r']
        # Number of actions
        self.action_size = len(self.action_space)
        # Title of the window
        self.title('DeepSARSA')
        # Set window size based on grid dimensions
        self.geometry('{0}x{1}'.format(HEIGHT * UNIT, HEIGHT * UNIT))
        # Load images for the grid items
        self.shapes = self.load_images()
        # Create the grid canvas
        self.canvas = self._build_canvas()
        # Initialize step counter
        self.counter = 0
        # List to hold reward states

        self.rewards = []
        # List to hold goal states
        self.goal = []
        # Defined obstacles with negative rewards
        self.set_reward([0, 1], -1)
        self.set_reward([1, 2], -1)
        self.set_reward([2, 3], -1)
        # Defined goal state with positive reward
        self.set_reward([4, 4], 1)

    # Method to build the canvas with grid lines and set initial objects
    def _build_canvas(self):
        # Created a canvas with a white background of specified size
        canvas = tk.Canvas(self, bg='white', height=HEIGHT * UNIT, width=WIDTH * UNIT)
        # Created vertical grid lines
```

```
        for c in range(0, WIDTH * UNIT, UNIT):
            x0, y0, x1, y1 = c, 0, c, HEIGHT * UNIT
            canvas.create_line(x0, y0, x1, y1)
        # Created horizontal grid lines
        for r in range(0, HEIGHT * UNIT, UNIT):
            x0, y0, x1, y1 = 0, r, HEIGHT * UNIT, r
            canvas.create_line(x0, y0, x1, y1)

        self.rewards = []  # Clear rewards list
        self.goal = []     # Clear goal list
        # Placed the agent's rectangle image at the starting position
        x, y = UNIT/2, UNIT/2
        self.rectangle = canvas.create_image(x, y, image=self.shapes[0])

        # Pack the canvas to make it visible
        canvas.pack()
        return canvas

    # Method to load images for different objects on the grid
    def load_images(self):
        # Agent image
        rectangle = PhotoImage(Image.open("../img/rectangle.png").resize((30, 30)))
        # Obstacle image
        triangle = PhotoImage(Image.open("../img/triangle.png").resize((30, 30)))
        # Goal image
        circle = PhotoImage(Image.open("../img/circle.png").resize((30, 30)))
        return rectangle, triangle, circle

    # Method to reset rewards and obstacles on the canvas
    def reset_reward(self):
        # Removed all reward figures from the canvas
        for reward in self.rewards:
            self.canvas.delete(reward['figure'])

        self.rewards.clear()  # Cleared the rewards list
        self.goal.clear()     # Cleared the goal list
        # Re-set obstacles and goal
        self.set_reward([0, 1], -1)
        self.set_reward([1, 2], -1)
        self.set_reward([2, 3], -1)
        self.set_reward([4, 4], 1)

    # Method to set a reward at a specific state in the grid
    def set_reward(self, state, reward):
```

```python
    state = [int(state[0]), int(state[1])]  # Ensure state values are integers
    x, y = int(state[0]), int(state[1])      # Extract x and y coordinates
    temp = {}                               # Temporary dictionary to hold reward properties
    if reward > 0:                           # If reward, set as goal
        temp['reward'] = reward
        temp['figure'] = self.canvas.create_image((UNIT * x) + UNIT / 2,
                                                  (UNIT * y) + UNIT / 2,
                                                  image=self.shapes[2])
        self.goal.append(temp['figure'])      # Add to goal list

    elif reward < 0:                         # If penalty, set as obstacle
        temp['direction'] = -1               # Initial movement direction
        temp['reward'] = reward
        temp['figure'] = self.canvas.create_image((UNIT * x) + UNIT / 2,
                                                 (UNIT * y) + UNIT / 2,
                                                 image=self.shapes[1])

    temp['coords'] = self.canvas.coords(temp['figure'])  # Store coordinates
    temp['state'] = state                          # Store state
    self.rewards.append(temp)                          # Add to rewards list

# Method to check if current state contains a reward
def check_if_reward(self, state):
    check_list = {'if_goal': False}  # Initialized goal flag
    rewards = 0                   # Initialized reward accumulator

    # Checked each reward in the environment
    for reward in self.rewards:
        if reward['state'] == state:      # If the reward state matches given state
            rewards += reward['reward']   # Added the reward value
            if reward['reward'] == 1:      # Checked if it's the goal state
                check_list['if_goal'] = True

    check_list['rewards'] = rewards       # Updated check list with reward total
    return check_list                  # Returned check list with goal and reward info

# Convert coordinates to grid state
def coords_to_state(self, coords):
    x = int((coords[0] - UNIT / 2) / UNIT)  # Calculate x position in grid
    y = int((coords[1] - UNIT / 2) / UNIT)  # Calculate y position in grid
    return [x, y]

# Method to reset the environment for a new episode
def reset(self):
```

```python
        self.update()                    # Update the UI
        time.sleep(0.5)                  # Delay to visually reset the UI
        x, y = self.canvas.coords(self.rectangle)
        self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)  # Move agent to start
        self.reset_reward()              # Reset all rewards and goal
        return self.get_state()          # Return initial state

    # Method to take a step in the environment based on action
    def step(self, action):
        self.counter += 1                # Increment step counter
        self.render()                    # Render the environment

        # Move obstacles every alternate step
        if self.counter % 2 == 1:
            self.rewards = self.move_rewards()

        next_coords = self.move(self.rectangle, action)  # Move agent based on action
        # Check for reward at new position
        check = self.check_if_reward(self.coords_to_state(next_coords))
        done = check['if_goal']          # Check if goal is reached
        reward = check['rewards']        # Get reward at current state

        self.canvas.tag_raise(self.rectangle)  # Ensure agent is visible above other items
        s_ = self.get_state()            # Get new state
        return s_, reward, done          # Return new state, reward, and goal status

    # Method to get current state in the environment
    def get_state(self):
        location = self.coords_to_state(self.canvas.coords(self.rectangle))  # Get agent's position
        agent_x, agent_y = location[0], location[1]  # Extract agent coordinates

        states = []                      # Initialize list for state information
        for reward in self.rewards:
            reward_location = reward['state']  # Get reward location
            states.append(reward_location[0] - agent_x)  # Relative x-position
            states.append(reward_location[1] - agent_y)  # Relative y-position
            if reward['reward'] < 0:
                states.append(-1)        # Obstacle flag
                states.append(reward['direction'])  # Obstacle direction
            else:
                states.append(1)         # Goal flag
        return states                    # Return state vector

    # Method to move rewards (obstacles) within the grid
```

```python
    def move_rewards(self):
        new_rewards = []                    # Initialize list for new rewards
        for temp in self.rewards:
            if temp['reward'] == 1:         # If goal, skip moving
                new_rewards.append(temp)
                continue
            temp['coords'] = self.move_const(temp)  # Move obstacle
            temp['state'] = self.coords_to_state(temp['coords'])  # Update obstacle
```

Deep_sarsa_agent.py:

```python
# Import necessary libraries
import copy
import pylab
import random
import numpy as np
from environment import Env # Importing custom environment class (GridWorld)
from keras.layers import Dense       # For creating neural network layers
from keras.optimizers import Adam    # Optimizer for training the neural network
from keras.models import Sequential  # Model type for a linear stack of layers

# Defined total number of epochs for training the agent
EPISODES = 1000

# Defined the DeepSARSA Agent class, using neural network as Q-function approximator
class DeepSARSAgent:
    def __init__(self):
        # Flag to determine if the model should load pre-trained weights
        self.load_model = False

        # Define action space: possible actions the agent can take
        self.action_space = [0, 1, 2, 3, 4]  # e.g., up, down, left, right, stay

        # Get the size of the action and state spaces
        self.action_size = len(self.action_space)
        self.state_size = 15  # Example state size for a 15-dimensional state

        # Hyperparameters for the SARSA algorithm
        self.discount_factor = 0.99  # Future reward discount
        self.learning_rate = 0.001   # Learning rate for the neural network
```

```
        # Epsilon-greedy parameters for exploration-exploitation trade-off
        self.epsilon = 1.0          # Starting exploration rate
        self.epsilon_decay = .9999  # Decay rate for epsilon
        self.epsilon_min = 0.01     # Minimum value for epsilon to maintain some exploration

        # Initialize the Q-function approximator (neural network)
        self.model = self.build_model()

        # Load a pre-trained model if the flag is set
        if self.load_model:
            self.epsilon = 0.05  # Reduced exploration rate for pre-trained model
            self.model.load_weights('./save_model/deep_sarsa_trained.h5')

    # Method to build the neural network for approximating Q-values
    def build_model(self):
        model = Sequential()  # Sequential model allows a linear stack of layers
        model.add(Dense(30, input_dim=self.state_size, activation='relu'))  # Input layer
        model.add(Dense(30, activation='relu'))  # Hidden layer with ReLU activation
        # Output layer with one node per action
        model.add(Dense(self.action_size, activation='linear'))
        model.summary()  # Display the model architecture
        # Compile model with mean squared error loss and Adam optimizer
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model

    # Method to get an action using epsilon-greedy policy
    def get_action(self, state):
        if np.random.rand() <= self.epsilon:
            # Explore: randomly select an action
            return random.randrange(self.action_size)
        else:
            # Exploit: predict Q-values and choose the action with the highest value
            state = np.float32(state)  # Ensure state is in the correct format
            q_values = self.model.predict(state)  # Predict Q-values for all actions
            return np.argmax(q_values[0])  # Select action with the maximum Q-value

    # Method to train the model using SARSA algorithm
    def train_model(self, state, action, reward, next_state, next_action, done):
        # Decay epsilon value after each training step
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        # Convert states to proper format
        state = np.float32(state)
```

```
        next_state = np.float32(next_state)

        # Predict current Q-value for the given state and action
        target = self.model.predict(state)[0]

        # SARSA update: calculate the target Q-value
        if done:
            # If the epoch ends, the target is simply the reward
            target[action] = reward
        else:
            # Calculate the SARSA target using reward and next state-action pair
            target[action] = (reward + self.discount_factor *
                              self.model.predict(next_state)[0][next_action])

        # Reshape target for model input and output dimensions
        target = np.reshape(target, [1, 5])

        # Train the model on the state and target Q-value
        self.model.fit(state, target, epochs=1, verbose=0)

# Main code block for training and running the DeepSARSA agent
if __name__ == "__main__":
    env = Env()  # Initialize the environment
    agent = DeepSARSAgent()  # Initialize the DeepSARSA agent

    # Variables to track progress
    global_step = 0
    scores, episodes = [], []

    # Training loop over epochs
    for e in range(EPISODES):
        done = False  # Reset done flag for each epoch
        score = 0     # Initialize score
        state = env.reset()  # Reset the environment and get initial state
        state = np.reshape(state, [1, 15])  # Reshape for the neural network

        # Loop to take steps in the environment until the epochs ends
        while not done:
            # Increment global step counter
            global_step += 1

            # Choose an action based on current state
            action = agent.get_action(state)
```

```
        # Take action, receive reward, and observe next state
        next_state, reward, done = env.step(action)
        next_state = np.reshape(next_state, [1, 15])  # Reshape for neural network input

        # Choose the next action in the next state (SARSA's characteristic)
        next_action = agent.get_action(next_state)

        # Train the model with the transition (SARSA update)
        agent.train_model(state, action, reward, next_state, next_action, done)

        # Update state and accumulate the score
        state = next_state
        score += reward

        # Copy state to avoid modifying original references
        state = copy.deepcopy(next_state)

        # If epochs ends, log score and epsilon
        if done:
            scores.append(score)      # Record the score for the epochs
            episodes.append(e)        # Record the epochs number
            pylab.plot(episodes, scores, 'b')  # Plot score over epochs
            pylab.savefig("./save_graph/deep_sarsa_.png")  # Save plot to file
            print("episode:", e, "  score:", score, "global_step",
                  global_step, "  epsilon:", agent.epsilon)

    # Save model weights every 100 epochs
    if e % 100 == 0:
        agent.model.save_weights("./save_model/deep_sarsa.h5")
```

Core Section:

train_model():

The train_model() function is responsible for updating the agent's Q-values using the SARSA (State-Action-Reward-State-Action) algorithm. This function applies a reward received from the environment to adjust the Q-values so that the agent can learn the best actions to take in different states.

get_action():

The get_action() function helps the agent decide which action to take using an epsilon-greedy policy. This approach ensures the agent explores new actions but gradually prefers the best-known actions as it learns.

These two functions are at the heart of the Deep SARSA learning process and all the comments are added to this section of the above code.

Core Section:

step() Function:

The step function is responsible for taking in an action from the agent, updating the agent's position in the environment, handling rewards, and determining if the episode (or game) is over.

get_state() Function:

The get_state() function provides a representation of the current state, which includes the agent’s location and relative positions of any rewards (obstacles or the goal) on the grid. This state information helps the agent understand its surroundings and make informed decisions.

These core functions of environment.py step() and get_state(), define the environment's interaction with the agent