**IEEE I06 ASSIGNMENT 02 -PPO**

**PPO: A Practical Assignment**

**Assignment Tasks**

**Task 1: Environment Setup**

**Python Code:**

class Environment():

    # Initializing the class with initial position and grid

    def \_\_init\_\_(self):

        self.pos = (0, 0)

        self.grid = np.zeros((10, 10), dtype=int)

        self.obs = set()

        self.fig, self.ax = plt.subplots(figsize=(6, 6))

        self.cmap = ListedColormap(["red", "white", "blue", "green", "yellow"])

        self.plot = None

        # Initialize the plot

        self.ax.set\_aspect('equal', adjustable='box')

        self.ax.invert\_yaxis()

        self.ax.xaxis.tick\_top()

        self.ax.set\_xticks(np.arange(0.5, 10, 1))

        self.ax.set\_xticklabels(range(10))

        self.ax.set\_yticks(np.arange(0.5, 10, 1))

        self.ax.set\_yticklabels(range(10))

        self.ax.set\_title("CurrentPosition(Yellow), Start(Blue), Obstacles(Red), End(Green)", y=-0.1)

    # Function to reset the grid and add obstacles

    def reset(self):

        self.grid = np.zeros((10, 10), dtype=int)

        self.pos = (0, 0)

        self.obs = set()

        # Add predefined obstacles

        self.obs.add((3,2))

        self.obs.add((2,2))

        self.obs.add((2,4))

        self.obs.add((1,2))

        self.obs.add((1,1))

        self.obs.add((5,4))

        self.obs.add((4,4))

        self.obs.add((4,6))

        self.obs.add((3,5))

        self.obs.add((3,3))

        self.obs.add((7,6))

        self.obs.add((6,6))

        self.obs.add((6,8))

        self.obs.add((5,6))

        self.obs.add((4,5))

        for obs in self.obs:

            self.grid[obs] = -1

        self.grid[0, 0] = 1  # Start position

        self.grid[9, 9] = 2  # End position

    # Function to move the agent given an action

    def step(self, action):

        y, x = self.pos

        if action == 0 and y > 0:

            newpos = (y - 1, x)  # Move up

        elif action == 1 and y < 9:

            newpos = (y + 1, x)  # Move down

        elif action == 2 and x > 0:

            newpos = (y, x - 1)  # Move left

        elif action == 3 and x < 9:

            newpos = (y, x + 1)  # Move right

        else:

            newpos = (y, x)  # Invalid move

        if self.grid[newpos] == -1:  # Obstacle

            return self.pos, -15, False

        self.pos = newpos

        reward = -1

        terminated = False

        if self.pos == (9,9):  # Reached the goal

            reward = 20

            terminated = True

        return self.pos, reward, terminated

    # Render the grid world

    def render(self):

        # Update the grid with the current position

        grid = self.grid.copy()

        grid[self.pos] = 4  # 4 corresponds to Yellow in cmap

        if self.plot is None:

            # First-time plot initialization

            self.plot = self.ax.pcolormesh(grid, cmap=self.cmap, edgecolors='k', linewidth=0.5, shading='auto')

        else:

            # Update the plot data

            self.plot.set\_array(grid.ravel())

        # Redraw the plot

        plt.draw()

        plt.pause(0.1)  # Pause for a short duration to allow updates

**Task 2: Implement PPO**

**Python code:**

**Memory:**

class BufferMemory:

    def \_\_init\_\_(self, batch\_size):

        self.states = []

        self.probs = []

        self.vals = []

        self.actions = []

        self.rewards = []

        self.dones = []

        self.batch\_size = batch\_size

    # Generate mini-batches for training

    def generate\_batches(self):

        n\_states = len(self.states)

        batch\_start = np.arange(0, n\_states, self.batch\_size)

        indices = np.arange(n\_states, dtype=np.int64)

        np.random.shuffle(indices)

        batches = [indices[i:i+self.batch\_size] for i in batch\_start]

        return np.array(self.states),\

                np.array(self.actions),\

                np.array(self.probs),\

                np.array(self.vals),\

                np.array(self.rewards),\

                np.array(self.dones),\

                batches

    # Store transition in memory

    def store\_memory(self, state, action, probs, vals, reward, done):

        self.states.append(state)

        self.actions.append(action)

        self.probs.append(probs)

        self.vals.append(vals)

        self.rewards.append(reward)

        self.dones.append(done)

    # Clear memory after training

    def clear\_memory(self):

        self.states = []

        self.probs = []

        self.actions = []

        self.rewards = []

        self.dones = []

        self.vals = []

**Actor Network:**

# Actor network for the PPO algorithm

class ActorNetwork(nn.Module):

    def \_\_init\_\_(self, n\_actions, input\_dims, alpha,

            fc1\_dims=256, fc2\_dims=256):

        super(ActorNetwork, self).\_\_init\_\_()

        # Neural network architecture

        self.actor = nn.Sequential(

                nn.Linear(input\_dims, fc1\_dims),

                nn.ReLU(),

                nn.Linear(fc1\_dims, fc2\_dims),

                nn.ReLU(),

                nn.Linear(fc2\_dims, n\_actions),

                nn.Softmax(dim=-1)

        )

        # Optimizer and device setup

        self.optimizer = T.optim.Adam(self.parameters(), lr=alpha)

        self.device = T.device('cuda:0' if T.cuda.is\_available() else 'cpu')

        self.to(self.device)

    # Forward pass

    def forward(self, state):

        dist = self.actor(state)

        dist = Categorical(dist)

        return dist

**Critic Network:**

# Critic network for the PPO algorithm

class CriticNetwork(nn.Module):

    def \_\_init\_\_(self, input\_dims, alpha, fc1\_dims=256, fc2\_dims=256):

        super(CriticNetwork, self).\_\_init\_\_()

        # Neural network architecture

        self.critic = nn.Sequential(

                nn.Linear(input\_dims, fc1\_dims),

                nn.ReLU(),

                nn.Linear(fc1\_dims, fc2\_dims),

                nn.ReLU(),

                nn.Linear(fc2\_dims, 1)

        )

        # Optimizer and device setup

        self.optimizer = T.optim.Adam(self.parameters(), lr=alpha)

        self.device = T.device('cuda:0' if T.cuda.is\_available() else 'cpu')

        self.to(self.device)

    # Forward pass

    def forward(self, state):

        value = self.critic(state)

        return value

**Agent:**

class Agent:

    def \_\_init\_\_(self, n\_actions, input\_dims, gamma=0.99, alpha=0.0003, gae\_lambda=0.95,

            policy\_clip=0.2, batch\_size=64, n\_epochs=10):

        self.gamma = gamma

        self.policy\_clip = policy\_clip

        self.n\_epochs = n\_epochs

        self.gae\_lambda = gae\_lambda

        # Initialize actor, critic, and memory

        self.actor = ActorNetwork(n\_actions, input\_dims, alpha)

        self.critic = CriticNetwork(input\_dims, alpha)

        self.memory = BufferMemory(batch\_size)

    # Convert grid position to one-hot input

    def converter(self, posi):

        x, y = posi

        inp = T.zeros(100)  # 10x10 grid = 100 states

        inp[10 \* x + y] = 1

        return inp

    # Store transition in memory

    def remember(self, state, action, probs, vals, reward, done):

        self.memory.store\_memory(state, action, probs, vals, reward, done)

    # Choose action using the actor network

    def choose\_action(self, observation):

        state = self.converter(observation).to(self.actor.device)

        dist = self.actor(state)

        value = self.critic(state)

        action = dist.sample()

        probs = (dist.log\_prob(action)).item()

        action = (action).item()

        value = (value).item()

        return action, probs, value

    # Train the agent

    def learn(self):

        for \_ in range(self.n\_epochs):

            state\_arr, action\_arr, old\_prob\_arr, vals\_arr,\

            reward\_arr, dones\_arr, batches = \

                    self.memory.generate\_batches()

            values = vals\_arr

            advantage = np.zeros(len(reward\_arr), dtype=np.float32)

            # Calculate advantages

            for t in range(len(reward\_arr)-1):

                discount = .99

                a\_t = 0

                for k in range(t, len(reward\_arr)-1):

                    a\_t += discount\*(reward\_arr[k] + self.gamma\*values[k+1]\*\

                            (1-int(dones\_arr[k])) - values[k])

                    discount \*= self.gamma\*self.gae\_lambda

                advantage[t] = a\_t

            advantage = T.tensor(advantage).to(self.actor.device)

            values = T.tensor(values).to(self.actor.device)

            for batch in batches:

                states = T.tensor(np.array([self.converter(state) for state in state\_arr[batch]]), dtype=T.float).to(self.actor.device)

                old\_probs = T.tensor(old\_prob\_arr[batch]).to(self.actor.device)

                actions = T.tensor(action\_arr[batch]).to(self.actor.device)

                dist = self.actor(states)

                critic\_value = self.critic(states)

                critic\_value = T.squeeze(critic\_value)

                new\_probs = dist.log\_prob(actions)

                prob\_ratio = new\_probs.exp() / old\_probs.exp()

                weighted\_probs = advantage[batch] \* prob\_ratio

                weighted\_clipped\_probs = T.clamp(prob\_ratio, 1-self.policy\_clip,

                        1+self.policy\_clip)\*advantage[batch]

                actor\_loss = -T.min(weighted\_probs, weighted\_clipped\_probs).mean()

                returns = advantage[batch] + values[batch]

                critic\_loss = (returns-critic\_value)\*\*2

                critic\_loss = critic\_loss.mean()

                total\_loss = actor\_loss + 0.5\*critic\_loss

                self.actor.optimizer.zero\_grad()

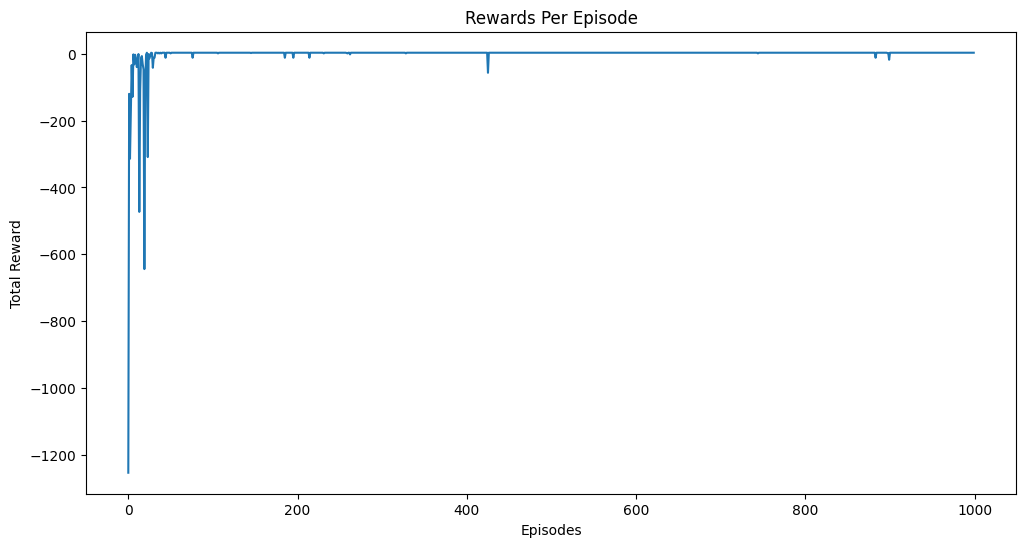
                self.critic.optimizer.zero\_grad()

                total\_loss.backward()

                self.actor.optimizer.step()

                self.critic.optimizer.step()

        self.memory.clear\_memory()



**The above plots show the total rewards in each episode**

**Task 3: Analyze Performance**

After training the model for a few different obstacle placements(10) we can see that the model is successful in learning the optimal policy

The agent was able to reach the goal in all 10 runs, giving us a success rate of 100%(but it takes a lot of time to train the agent when the obstacle placement is more complex)

Many different values of hyperparameters were tried to give us the existing values.

Mainly, the learning rate was reduced to improve stability.

**Visualization:**

Video uploaded to drive

**Some Important Observations:**

* The agent showed greatly increased efficiency when the learning rate of the optimizer was changed from 0.0003 to 0.001
* Using a learning rate scheduler might be necessary as the model needs high learning rate at the start to learn the policy but might need to fine tune it later
* Model takes a lot of time to train due to the intensity of the computations of the PPO algorithm