**Assignment 5 TD3**

**Task 1: Environment Setup (Dynamic Goals and Constraints)**

class DynamicEnvironment:

    def \_\_init\_\_(self, size=(10, 10), max\_velocity=1.0, goal\_update\_interval=20):

        self.size = size

        self.max\_velocity = max\_velocity

        self.goal\_update\_interval = goal\_update\_interval

        self.step\_count = 0

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

        plt.ion()

        self.reset()

    def reset(self):

        """Resets the environment to its initial state."""

        self.agent\_position = np.array([np.random.uniform(0, self.size[0]), np.random.uniform(0, self.size[1])])

        self.goal\_position = self.\_generate\_new\_goal()

        self.step\_count = 0

        return self.agent\_position, self.goal\_position

    def \_generate\_new\_goal(self):

        """Generates a new random goal position within the environment boundaries."""

        return np.array([np.random.uniform(0, self.size[0]), np.random.uniform(0, self.size[1])])

    def step(self, action):

        """Takes an action to update the environment state.

        Args:

            action (np.array): A 2D action vector representing velocity (dx, dy).

        Returns:

            tuple: (agent\_position, goal\_position, reward, done)

        """

        reward = -1  # Step penalty to discourage inefficiency

        # Check for exceeding maximum velocity

        if np.any(np.abs(action) > self.max\_velocity):

            reward -= 10  # Penalty for exceeding velocity constraint

            action = np.clip(action, -self.max\_velocity, self.max\_velocity)

        # Apply velocity constraints and update agent position

        self.agent\_position += action

        # Check for boundary violations

        if np.any(self.agent\_position < 0) or np.any(self.agent\_position > self.size):

            reward -= 5  # Penalty for moving out of bounds

            self.agent\_position = np.clip(self.agent\_position, 0, [self.size[0], self.size[1]])

        # Check for goal update

        self.step\_count += 1

        if self.step\_count % self.goal\_update\_interval == 0:

            self.goal\_position = self.\_generate\_new\_goal()

        # Compute reward for proximity to goal

        distance\_to\_goal = np.linalg.norm(self.agent\_position - self.goal\_position)

        if distance\_to\_goal < 0.5:

            reward += 10  # Reward for reaching the goal

            done = True

        else:

            done = False

        return self.agent\_position, self.goal\_position, reward, done

    def render(self):

        """Renders the current state of the environment as a dynamic frame."""

        self.ax.clear()  # Clear the axes to update the frame

        self.ax.set\_xlim(0, self.size[0])

        self.ax.set\_ylim(0, self.size[1])

        self.ax.plot(self.agent\_position[0], self.agent\_position[1], 'bo', label='Agent')

        self.ax.plot(self.goal\_position[0], self.goal\_position[1], 'ro', label='Goal')

        self.ax.legend()

        self.ax.set\_title("Environment State")

        plt.pause(0.1)

    def close(self):

        """Closes the rendering window."""

        plt.ioff()  # Turn off interactive mode

        plt.show()

**Task 2: TD3 Algorithm Implementation**

class ReplayBuffer:

    def \_\_init\_\_(self, max\_size, input\_shape, n\_actions):

        """

        Initialize the replay buffer.

        Args:

        - max\_size (int): Maximum number of transitions the buffer can store.

        - input\_shape (tuple): Shape of the state space (e.g., (state\_dim,)).

        - n\_actions (int): Number of actions in the action space.

        Attributes:

        - max\_size: Maximum buffer size.

        - mem\_cntr: Counter to keep track of the number of transitions added.

        - state\_memory: Array to store states.

        - new\_state\_memory: Array to store the next states.

        - action\_memory: Array to store actions taken.

        - reward\_memory: Array to store rewards received.

        - terminal\_memory: Array to store terminal flags (whether the episode ended).

        """

        self.max\_size = max\_size

        self.mem\_cntr = 0  # Initialize the memory counter to 0

        # Initialize memory arrays

        self.state\_memory = np.zeros((self.max\_size, input\_shape))  # Stores states

        self.new\_state\_memory = np.zeros((self.max\_size, input\_shape))  # Stores next states

        self.action\_memory = np.zeros((self.max\_size, n\_actions))  # Stores actions

        self.reward\_memory = np.zeros(self.max\_size)  # Stores rewards

        self.terminal\_memory = np.zeros(self.max\_size, dtype=np.float32)  # Stores done flags (1 - done)

    def store\_transition(self, state, action, reward, state\_, done):

        """

        Store a single transition in the replay buffer.

        Args:

        - state (array): Current state.

        - action (array): Action taken in the current state.

        - reward (float): Reward received after taking the action.

        - state\_ (array): Next state after taking the action.

        - done (bool): Whether the episode ended after this transition.

        """

        # Find the index to store the transition (overwrites oldest transitions if full)

        index = self.mem\_cntr % self.max\_size

        # Store the transition components in their respective buffers

        self.state\_memory[index] = state

        self.new\_state\_memory[index] = state\_

        self.action\_memory[index] = action

        self.reward\_memory[index] = reward

        self.terminal\_memory[index] = 1 - done  # Store 1 if not done, 0 if done

        # Increment the memory counter

        self.mem\_cntr += 1

    def sample\_buffer(self, batch\_size):

        """

        Sample a batch of transitions from the replay buffer.

        Args:

        - batch\_size (int): Number of transitions to sample.

        Returns:

        - states (array): Batch of sampled states.

        - actions (array): Batch of sampled actions.

        - rewards (array): Batch of sampled rewards.

        - states\_ (array): Batch of sampled next states.

        - dones (array): Batch of sampled terminal flags (1 - done).

        """

        # Determine the maximum number of stored transitions

        max\_mem = min(self.mem\_cntr, self.max\_size)

        # Randomly sample `batch\_size` indices from the range [0, max\_mem)

        batch = np.random.choice(max\_mem, batch\_size, replace=False)

        # Retrieve the sampled transitions using the indices

        states = self.state\_memory[batch]

        states\_ = self.new\_state\_memory[batch]

        actions = self.action\_memory[batch]

        rewards = self.reward\_memory[batch]

        dones = self.terminal\_memory[batch]

        return states, actions, rewards, states\_, dones

# Critic Network

class CriticNetwork(nn.Module):

    def \_\_init\_\_(self, beta, input\_dims, fc1\_dims, fc2\_dims, n\_actions,

            name, chkpt\_dir='tmp/td3'):

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

        self.input\_dims = input\_dims

        self.fc1\_dims = fc1\_dims

        self.fc2\_dims = fc2\_dims

        self.n\_actions = n\_actions

        self.name = name

        self.checkpoint\_dir = chkpt\_dir

        self.checkpoint\_file = os.path.join(self.checkpoint\_dir, name+'\_td3')

        # I think this breaks if the env has a 2D state representation

        self.fc1 = nn.Linear(self.input\_dims + n\_actions, self.fc1\_dims)

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

        self.q1 = nn.Linear(self.fc2\_dims, 1)

        self.optimizer = optim.Adam(self.parameters(), lr=beta)

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

        self.to(self.device)

    def forward(self, state, action):

        q1\_action\_value = self.fc1(T.cat([state, action], dim=1))

        q1\_action\_value = F.relu(q1\_action\_value)

        q1\_action\_value = self.fc2(q1\_action\_value)

        q1\_action\_value = F.relu(q1\_action\_value)

        q1 = self.q1(q1\_action\_value)

        return q1

    def save\_checkpoint(self):

        print('... saving checkpoint ...')

        T.save(self.state\_dict(), self.checkpoint\_file)

    def load\_checkpoint(self):

        print('... loading checkpoint ...')

        self.load\_state\_dict(T.load(self.checkpoint\_file))

# Actor Network

class ActorNetwork(nn.Module):

    def \_\_init\_\_(self, alpha, input\_dims, fc1\_dims, fc2\_dims,

            n\_actions, name, chkpt\_dir='tmp/td3'):

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

        self.input\_dims = input\_dims

        self.fc1\_dims = fc1\_dims

        self.fc2\_dims = fc2\_dims

        self.n\_actions = n\_actions

        self.name = name

        self.checkpoint\_dir = chkpt\_dir

        self.checkpoint\_file = os.path.join(self.checkpoint\_dir, name+'\_td3')

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

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

        self.mu = nn.Linear(self.fc2\_dims, self.n\_actions)

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

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

        self.to(self.device)

    def forward(self, state):

        prob = self.fc1(state)

        prob = F.relu(prob)

        prob = self.fc2(prob)

        prob = F.relu(prob)

        prob = T.tanh(self.mu(prob)) # if action is > +/- 1 then multiply by max action

        return prob

    def save\_checkpoint(self):

        print('... saving checkpoint ...')

        T.save(self.state\_dict(), self.checkpoint\_file)

    def load\_checkpoint(self):

        print('... loading checkpoint ...')

        self.load\_state\_dict(T.load(self.checkpoint\_file))

class Agent():

    def \_\_init\_\_(self, alpha, beta, input\_dims, tau, env,

            gamma=0.99, update\_actor\_interval=2, warmup=1000,

            n\_actions=2, max\_size=1000000, layer1\_size=400,

            layer2\_size=300, batch\_size=100, noise=0.1):

        self.gamma = gamma

        self.tau = tau

        self.max\_action = 1

        self.min\_action = 0

        self.memory = ReplayBuffer(max\_size, input\_dims, n\_actions)

        self.batch\_size = batch\_size

        self.learn\_step\_cntr = 0

        self.time\_step = 0

        self.warmup = warmup

        self.n\_actions = n\_actions

        self.update\_actor\_iter = update\_actor\_interval

        self.actor = ActorNetwork(alpha, input\_dims, layer1\_size,

                                  layer2\_size, n\_actions=n\_actions,

                                  name='actor')

        self.critic\_1 = CriticNetwork(beta, input\_dims, layer1\_size,

                                      layer2\_size, n\_actions=n\_actions,

                                      name='critic\_1')

        self.critic\_2 = CriticNetwork(beta, input\_dims, layer1\_size,

                                      layer2\_size, n\_actions=n\_actions,

                                      name='critic\_2')

        self.target\_actor = ActorNetwork(alpha, input\_dims, layer1\_size,

                                         layer2\_size, n\_actions=n\_actions,

                                         name='target\_actor')

        self.target\_critic\_1 = CriticNetwork(beta, input\_dims, layer1\_size,

                                         layer2\_size, n\_actions=n\_actions,

                                         name='target\_critic\_1')

        self.target\_critic\_2 = CriticNetwork(beta, input\_dims, layer1\_size,

                                         layer2\_size, n\_actions=n\_actions,

                                         name='target\_critic\_2')

        self.noise = noise

        self.update\_network\_parameters(tau=1)

    def choose\_action(self, observation):

        if self.time\_step < self.warmup:

            mu = T.tensor(np.random.normal(scale=self.noise, size=(self.n\_actions,)))

        else:

            state = T.tensor(observation, dtype=T.float).to(self.actor.device)

            mu = self.actor.forward(state).to(self.actor.device)

        mu\_prime = mu + T.tensor(np.random.normal(scale=self.noise),

                dtype=T.float).to(self.actor.device)

        mu\_prime = T.clamp(mu\_prime, self.min\_action, self.max\_action)

        self.time\_step += 1

        return mu\_prime.cpu().detach().numpy()

    def remember(self, state, action, reward, new\_state, done):

        self.memory.store\_transition(state, action, reward, new\_state, done)

    def learn(self):

        if self.memory.mem\_cntr < self.batch\_size:

            return

        state, action, reward, new\_state, done = \

                self.memory.sample\_buffer(self.batch\_size)

        reward = T.tensor(reward, dtype=T.float).to(self.critic\_1.device)

        done = T.tensor(done).to(self.critic\_1.device)

        state\_ = T.tensor(new\_state, dtype=T.float).to(self.critic\_1.device)

        state = T.tensor(state, dtype=T.float).to(self.critic\_1.device)

        action = T.tensor(action, dtype=T.float).to(self.critic\_1.device)

        target\_actions = self.target\_actor.forward(state\_)

        target\_actions = target\_actions + \

                T.clamp(T.tensor(np.random.normal(scale=0.2)), -0.5, 0.5)

        # might break if elements of min and max are not all equal

        target\_actions = T.clamp(target\_actions, self.min\_action, self.max\_action)

        q1\_ = self.target\_critic\_1.forward(state\_, target\_actions)

        q2\_ = self.target\_critic\_2.forward(state\_, target\_actions)

        q1 = self.critic\_1.forward(state, action)

        q2 = self.critic\_2.forward(state, action)

        q1\_[done] = 0.0

        q2\_[done] = 0.0

        q1\_ = q1\_.view(-1)

        q2\_ = q2\_.view(-1)

        critic\_value\_ = T.min(q1\_, q2\_)

        target = reward + self.gamma\*critic\_value\_

        target = target.view(self.batch\_size, 1)

        self.critic\_1.optimizer.zero\_grad()

        self.critic\_2.optimizer.zero\_grad()

        q1\_loss = F.mse\_loss(target, q1)

        q2\_loss = F.mse\_loss(target, q2)

        critic\_loss = q1\_loss + q2\_loss

        critic\_loss.backward()

        self.critic\_1.optimizer.step()

        self.critic\_2.optimizer.step()

        self.learn\_step\_cntr += 1

        if self.learn\_step\_cntr % self.update\_actor\_iter != 0:

            return

        self.actor.optimizer.zero\_grad()

        actor\_q1\_loss = self.critic\_1.forward(state, self.actor.forward(state))

        actor\_loss = -T.mean(actor\_q1\_loss)

        actor\_loss.backward()

        self.actor.optimizer.step()

        self.update\_network\_parameters()

    def update\_network\_parameters(self, tau=None):

        if tau is None:

            tau = self.tau

        actor\_params = self.actor.named\_parameters()

        critic\_1\_params = self.critic\_1.named\_parameters()

        critic\_2\_params = self.critic\_2.named\_parameters()

        target\_actor\_params = self.target\_actor.named\_parameters()

        target\_critic\_1\_params = self.target\_critic\_1.named\_parameters()

        target\_critic\_2\_params = self.target\_critic\_2.named\_parameters()

        critic\_1\_state\_dict = dict(critic\_1\_params)

        critic\_2\_state\_dict = dict(critic\_2\_params)

        actor\_state\_dict = dict(actor\_params)

        target\_actor\_state\_dict = dict(target\_actor\_params)

        target\_critic\_1\_state\_dict = dict(target\_critic\_1\_params)

        target\_critic\_2\_state\_dict = dict(target\_critic\_2\_params)

        for name in critic\_1\_state\_dict:

            critic\_1\_state\_dict[name] = tau\*critic\_1\_state\_dict[name].clone() + \

                    (1-tau)\*target\_critic\_1\_state\_dict[name].clone()

        for name in critic\_2\_state\_dict:

            critic\_2\_state\_dict[name] = tau\*critic\_2\_state\_dict[name].clone() + \

                    (1-tau)\*target\_critic\_2\_state\_dict[name].clone()

        for name in actor\_state\_dict:

            actor\_state\_dict[name] = tau\*actor\_state\_dict[name].clone() + \

                    (1-tau)\*target\_actor\_state\_dict[name].clone()

        self.target\_critic\_1.load\_state\_dict(critic\_1\_state\_dict)

        self.target\_critic\_2.load\_state\_dict(critic\_2\_state\_dict)

        self.target\_actor.load\_state\_dict(actor\_state\_dict)

    def save\_models(self):

        self.actor.save\_checkpoint()

        self.target\_actor.save\_checkpoint()

        self.critic\_1.save\_checkpoint()

        self.critic\_2.save\_checkpoint()

        self.target\_critic\_1.save\_checkpoint()

        self.target\_critic\_2.save\_checkpoint()

    def load\_models(self):

        self.actor.load\_checkpoint()

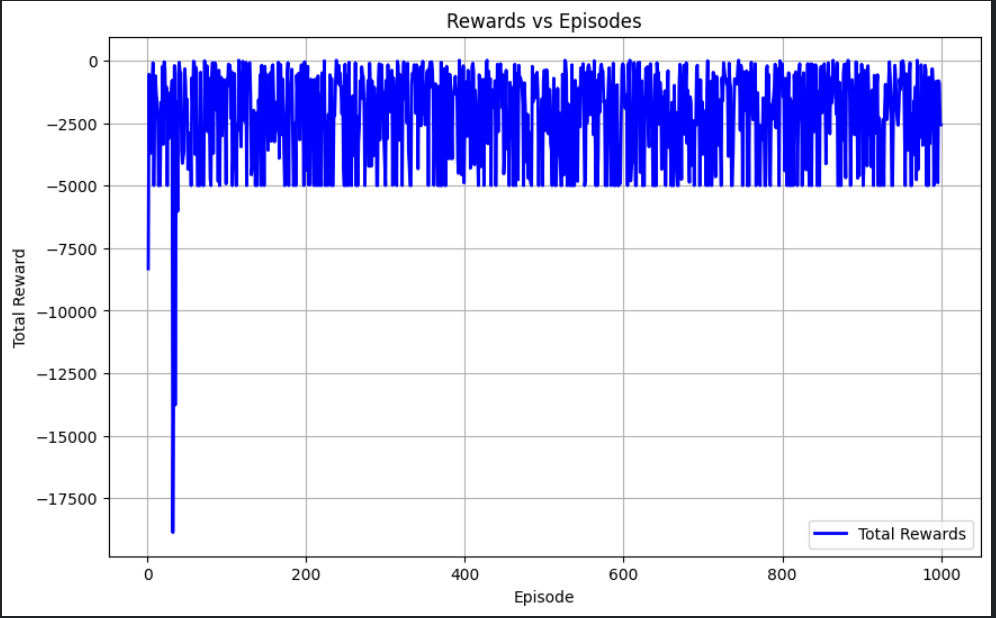
        self.target\_actor.load\_checkpoint()

        self.critic\_1.load\_checkpoint()

        self.critic\_2.load\_checkpoint()

        self.target\_critic\_1.load\_checkpoint()

        self.target\_critic\_2.load\_checkpoint()

**Plot of Rewards Vs. Episodes**

**Task 3: Advanced Evaluation and Analysis**

The trained agent failed to learn an optimal policy even after 9 hrs of training.

It learned not to bounce into any walls but it keeps oscillating near the goal instead of touching it.

The agent has a 47% success rate.