**Assignment 4 DDPG**

**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: DDPG 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 Critic(nn.Module):

    def \_\_init\_\_(self, beta, input\_dims, fc1\_dims, fc2\_dims, n\_actions, name, chkpt\_dir = 'tmp/ddpg'):

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

        self.input\_dims = input\_dims

        self.fc1\_dims = fc1\_dims

        self.fc2\_dims = fc2\_dims

        self.n\_actions = n\_actions

        self.model\_name = name

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

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

        f1 = 1 / np.sqrt(self.fc1.weight.data.size()[0])

        torch.nn.init.uniform\_(self.fc1.weight.data, -f1, f1)

        torch.nn.init.uniform\_(self.fc1.bias.data, -f1, f1)

        self.bn1 = nn.LayerNorm(self.fc1\_dims)

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

        f2 = 1 / np.sqrt(self.fc2.weight.data.size()[0])

        torch.nn.init.uniform\_(self.fc2.weight.data, -f2, f2)

        torch.nn.init.uniform\_(self.fc2.bias.data, -f2)

        self.bn2 = nn.LayerNorm(self.fc2\_dims)

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

        f3 = 0.003

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

        torch.nn.init.uniform\_(self.q.weight.data, -f3, f3)

        torch.nn.init.uniform\_(self.q.bias.data, -f3, f3)

        self.optimizer = optim.Adam(self.parameters(), lr=beta, weight\_decay=0.01)

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

        self.to(self.device)

    def forward(self, state, action):

        state = state.to(self.device) # Ensure input is on the same device as the model

        action = action.to(self.device)

        state\_value = self.fc1(state)

        state\_value = self.bn1(state\_value)

        state\_value = F.relu(state\_value)

        state\_value = self.fc2(state\_value)

        state\_value = self.bn2(state\_value)

        action\_value = F.relu(self.action\_value(action))

        state\_action\_value = F.relu(T.add(state\_value, action\_value))

        state\_action\_value = self.q(state\_action\_value)

        return state\_action\_value

    def save\_checkpoint(self):

        print("....Saving Checkpoint....")

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

    def load\_checkpoint(self):

        print("....Loading Checkpoint....")

        checkpoint = T.load(self.checkpoint\_file, map\_location=T.device('cpu'))  # Load the checkpoint

        self.load\_state\_dict(checkpoint)  # Update the model with the loaded state\_dict

# Actor Network

class Actor(nn.Module):

    def \_\_init\_\_(self, alpha, input\_dims, fc1\_dims, fc2\_dims, n\_actions, name, chkpt\_dir = 'tmp/ddpg'):

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

        self.input\_dims = input\_dims

        self.fc1\_dims = fc1\_dims

        self.fc2\_dims = fc2\_dims

        self.n\_actions = n\_actions

        self.model\_name = name

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

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

        f1 = 1 / np.sqrt(self.fc1.weight.data.size()[0])

        T.nn.init.uniform\_(self.fc1.weight.data, -f1, f1)

        T.nn.init.uniform\_(self.fc1.bias.data, -f1, f1)

        self.bn1 = nn.LayerNorm(self.fc1\_dims)

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

        f2 = 1 / np.sqrt(self.fc2.weight.data.size()[0])

        T.nn.init.uniform\_(self.fc2.weight.data, -f2, f2)

        T.nn.init.uniform\_(self.fc2.bias.data, -f2, f2)

        self.bn2 = nn.LayerNorm(self.fc2\_dims)

        f3 = 0.003

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

        T.nn.init.uniform\_(self.mu.weight.data, -f3, f3)

        T.nn.init.uniform\_(self.mu.bias.data, -f3, f3)

        self.optimizer = optim.Adam(self.parameters(), lr=alpha, weight\_decay=0.01)

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

        self.to(self.device)

    def forward(self, state):

        state = state.to(self.device)

        x = self.fc1(state)

        x = self.bn1(x)

        x = F.relu(x)

        x = self.fc2(x)

        x = self.bn2(x)

        x = T.tanh(self.mu(x))

        return x

    def save\_checkpoint(self):

        print("....Saving Checkpoint....")

        os.makedirs(os.path.dirname(self.checkpoint\_file), exist\_ok=True)

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

    def load\_checkpoint(self):

        print("....Loading Checkpoint....")

        checkpoint = T.load(self.checkpoint\_file, map\_location=T.device('cpu'))  # Load the checkpoint

        self.load\_state\_dict(checkpoint)  # Update the model with the loaded state\_dict

class Agent():

  def \_\_init\_\_(self, alpha, beta, input\_dims, tau, env, gamma = 0.99, n\_actions = 2, max\_size = 1000000, layer1\_size = 400, layer2\_size = 300, batch\_size = 64):

    self.gamma = gamma

    self.tau = tau

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

    self.batch\_size = batch\_size

    self.actor = Actor(alpha, input\_dims, layer1\_size, layer2\_size, n\_actions = n\_actions, name = 'Actor')

    self.target\_actor = Actor(alpha, input\_dims, layer1\_size, layer2\_size, n\_actions = n\_actions, name = 'TargetActor')

    self.critic = Critic(beta, input\_dims, layer1\_size, layer2\_size, n\_actions = n\_actions, name = 'Critic')

    self.target\_critic = Critic(beta, input\_dims, layer1\_size, layer2\_size, n\_actions = n\_actions, name = 'TargetCritic')

    self.noise = OUActionNoise(mu = np.zeros(n\_actions))

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

  def choose\_action(self, observation):

    self.actor.eval()

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

    mu = self.actor.forward(observation)

    mu\_prime = mu + T.tensor(self.noise(), dtype = T.float).to(self.actor.device)

    self.actor.train()

    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.device)

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

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

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

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

    self.target\_actor.eval()

    self.target\_critic.eval()

    self.critic.eval()

    target\_actions = self.target\_actor.forward(new\_state)

    critic\_value\_ = self.target\_critic.forward(new\_state, target\_actions)

    critic\_value = self.critic.forward(state, action)

    target = []

    for j in range(self.batch\_size):

      target.append(reward[j] + self.gamma\*critic\_value\_[j]\*done[j])

    target = T.tensor(target).to(self.critic.device)

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

    self.critic.train()

    self.critic.optimizer.zero\_grad()

    critic\_loss = F.mse\_loss(target, critic\_value)

    critic\_loss.backward()

    self.critic.optimizer.step()

    self.critic.eval()

    self.actor.optimizer.zero\_grad()

    mu = self.actor.forward(state)

    self.actor.train()

    actor\_loss = -self.critic.forward(state, mu)

    actor\_loss = T.mean(actor\_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\_params = self.critic.named\_parameters()

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

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

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

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

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

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

    for name in critic\_state\_dict:

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

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

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

  def save\_models(self):

    self.actor.save\_checkpoint()

    self.target\_actor.save\_checkpoint()

    self.critic.save\_checkpoint()

    self.target\_critic.save\_checkpoint()

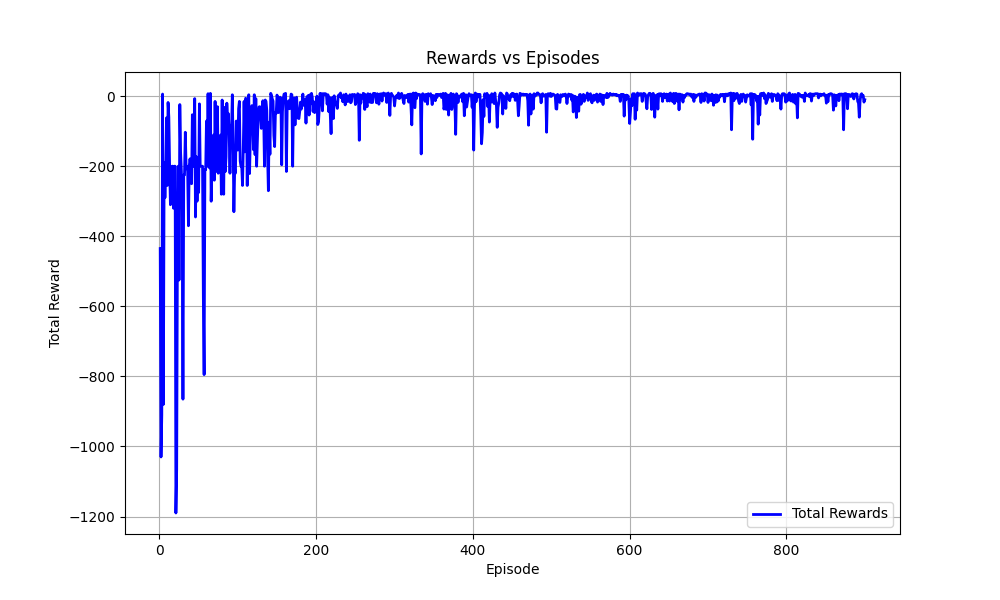
  def load\_models(self):

    self.actor.load\_checkpoint()

    self.target\_actor.load\_checkpoint()

    self.critic.load\_checkpoint()

    self.target\_critic.load\_checkpoint()

**Plot of Rewards Vs. Episodes**

**Task 3: Advanced Evaluation and Analysis**

The trained agent works well with dynamically changing goal positions.

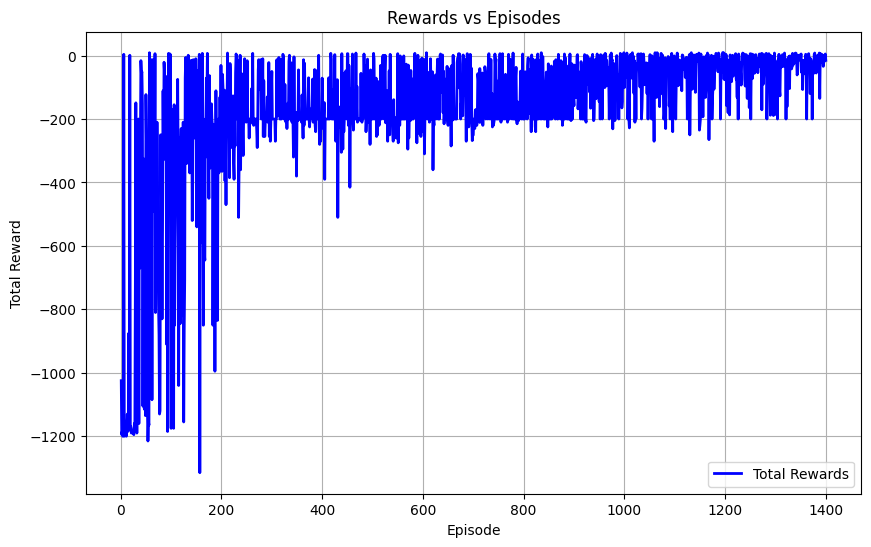
But, it accumulates a lot of penalty when the max velocity constraints are modified from (1,1) to (0.5,0.5). This happens as the agent was trained on a different velocity constraint.

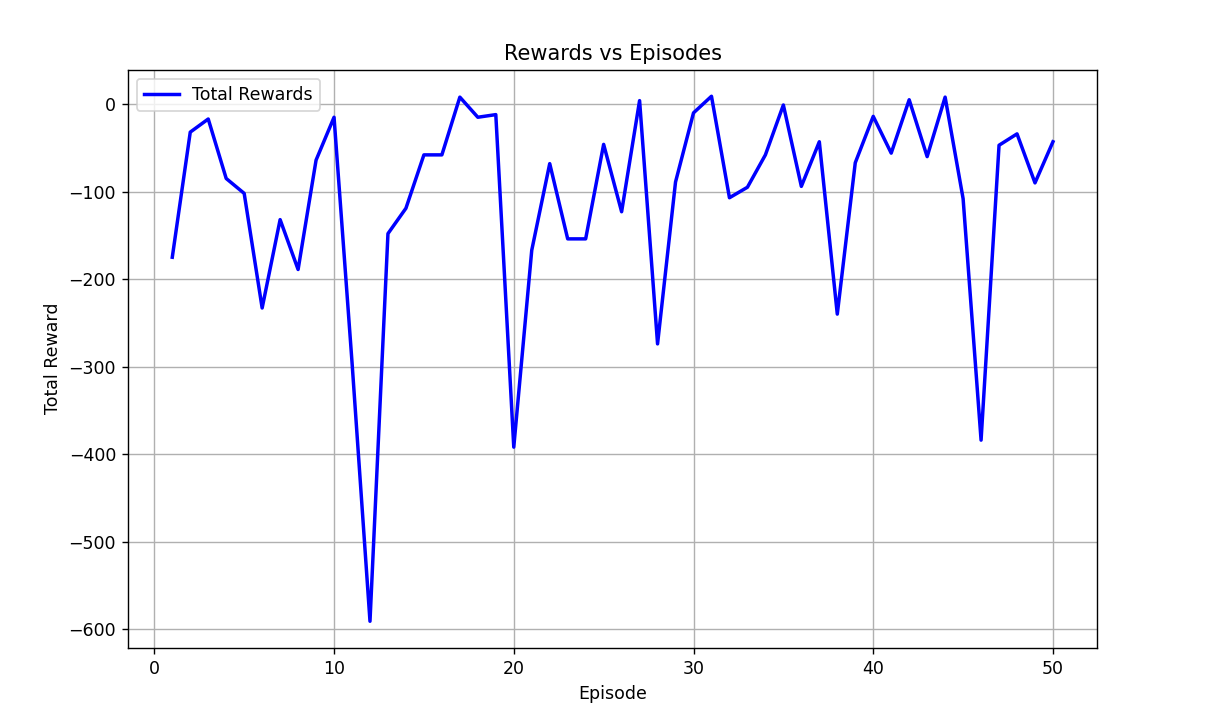
The agent has a success rate of 99.2%(calculated for 500 episodes),   
The average steps taken by the agent are 13.

Visualization uploaded to google drive…



**Trying to implement an obstacle:**



****

This model was trained with 1 obstacle that changes positions dynamically every 20 steps. It is observed that the agent is able to navigate while avoiding the obstacle but the training time was greatly increased(by a factor of 5x). It is also observed that the agent would oscillate near the goal accumulating negative reward for a few steps before actually reaching the goal(I was unable to find the reason.)