

RL CA3 : Research Assignment

Autonomous Robot Navigation in a Virtual Environment using Deep Reinforcement Learning with TD3

Sanskar Jadhav (21070126074)

Sahil Nagaralu (21070126076)

Roshan Yadav (21070126130)



Problem Statement – Fully Autonomous Goal-driven Exploration



Our robot must strategically select waypoints based solely on sensor (LiDAR) data, maximizing its chances of reaching the global goal without prior knowledge.



The robot motion policy should be agnostic to mapped data which is essential for uncertain environments.



The integration of DRL motion policy with global navigation strategy to effectively overcome local optimum challenges in unknown environments.

Our Robot - **Pioneer P3DX**



A 3D simulation environment showing a robot (a small red and black vehicle) on a light gray tiled floor. The environment contains several obstacles: a large gray rectangular block on the left, a gray cylindrical pillar in the center, and a small yellow cube on the right. A large, semi-transparent blue mesh structure covers a significant portion of the floor, representing a goal area or a region of interest. The text "MDP formulation" is overlaid in the center of the image.

MDP formulation

States

- Current Position of the Robot in the environment along x and y axes
- Position of the Goal that the robot must navigate towards
- LiDAR Sensor Data at that time with a length equal to environment dimension
- Previous Odometry Data used for calculating changes in position
- Orientation and Altitude
- Laser_readings + previous_actions + polar_goal_coordinates = 1D vector with 24 elements

Actions (Continuous)

- A tuple containing (linear velocity v , angular velocity ω), $v_{max} = 0.5m/s$ and $\omega_{max} = 1 rad/s$

Rewards

- 100 if goal reached, -100 if collision detected and immediate reward if not any of them

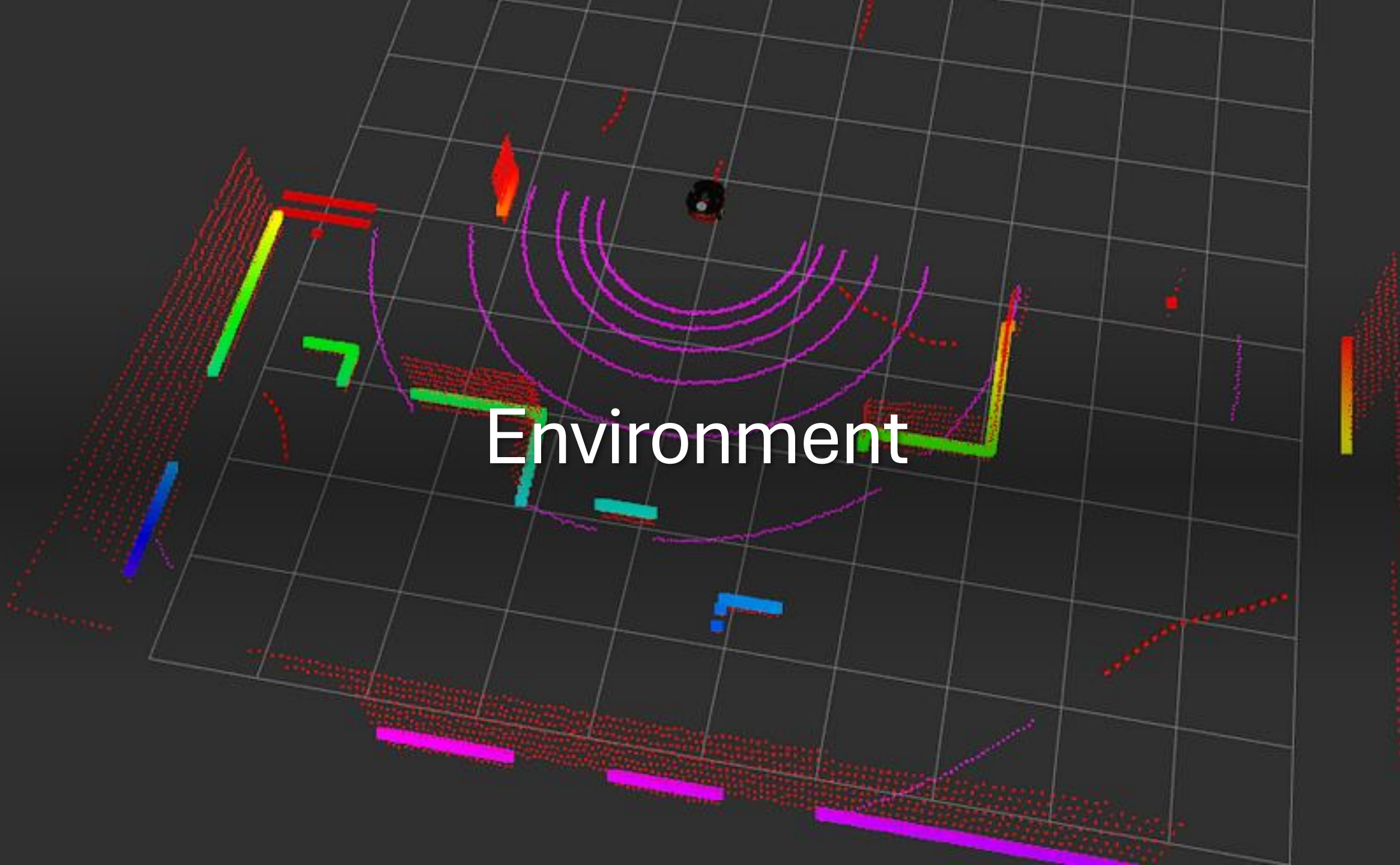
Discount Factor

- 0.99999

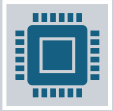
Model

- Model Free (TD3 Network in a Gazebo ROS Environment)

Environment



Method Description



Introduce a POI whenever a difference between two consecutive laser readings exceeds a predefined threshold suggesting robot to navigate through the perceived gap,



During navigation, the robot deletes any points of interest (POIs) from the memory that are found to be near obstacles, have already been visited, or cannot be reached after a certain number of steps.



TD3, an actor-critic type of network, contains an “actor” network to calculate an action to perform, and a “critic” network to estimate how good is this action.

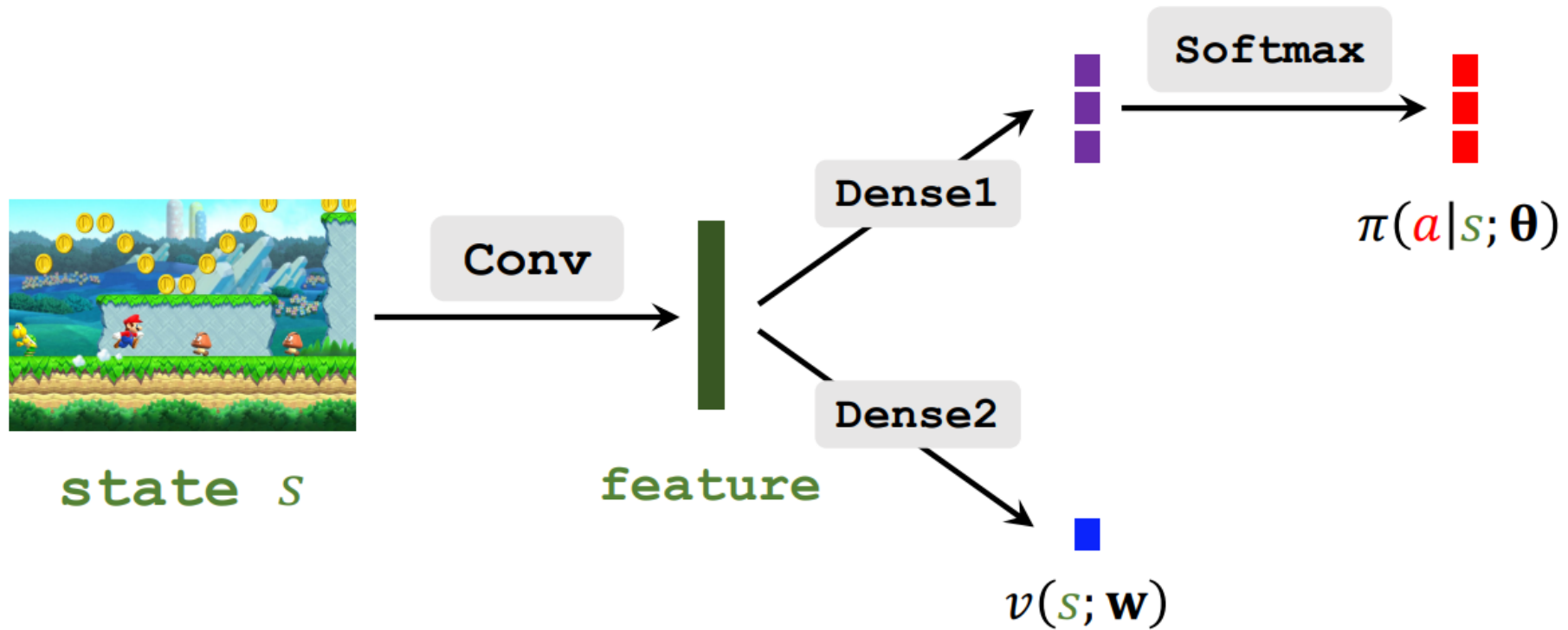


TD3 is an extension of DDPG which has a problem of overestimating Q-value dramatically which then leads to policy breaking because it exploits the errors in the Q-function.

Actor and Critic

- **Policy network (actor):** $\pi(a|s; \theta)$.
 - It is an approximation to the policy function, $\pi(a|s)$.
 - It controls the agent.
- **Value network (critic):** $v(s; w)$.
 - It is an approximation to the state-value function, $V_{\pi}(s)$.
 - It evaluates how good the state s is.

Actor and Critic



Training of A2C

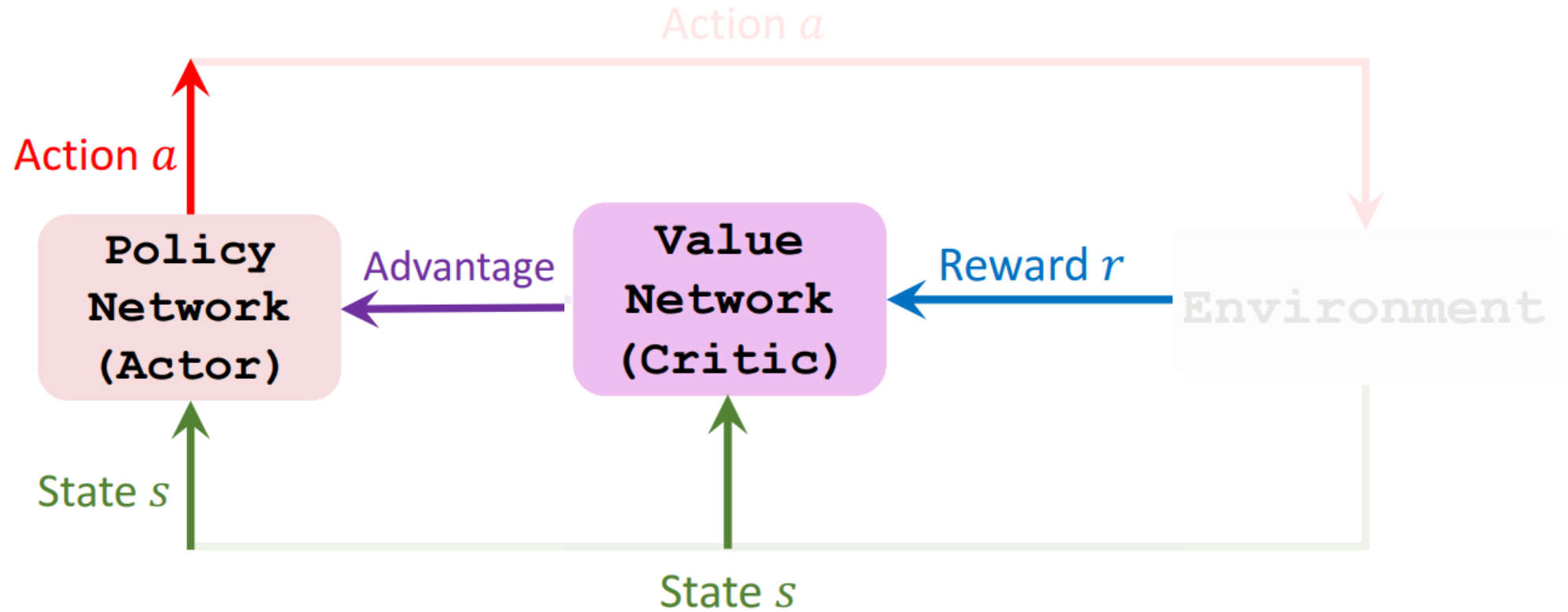
- Observe a transition (s_t, a_t, r_t, s_{t+1}) .
- TD target: $y_t = r_t + \gamma \cdot v(s_{t+1}; \mathbf{w})$.
- TD error: $\delta_t = v(s_t; \mathbf{w}) - y_t$.
- Update the policy network (actor) by:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \beta \cdot \delta_t \cdot \frac{\partial \ln \pi(a_t | s_t; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}.$$

- Update the value network (critic) by:

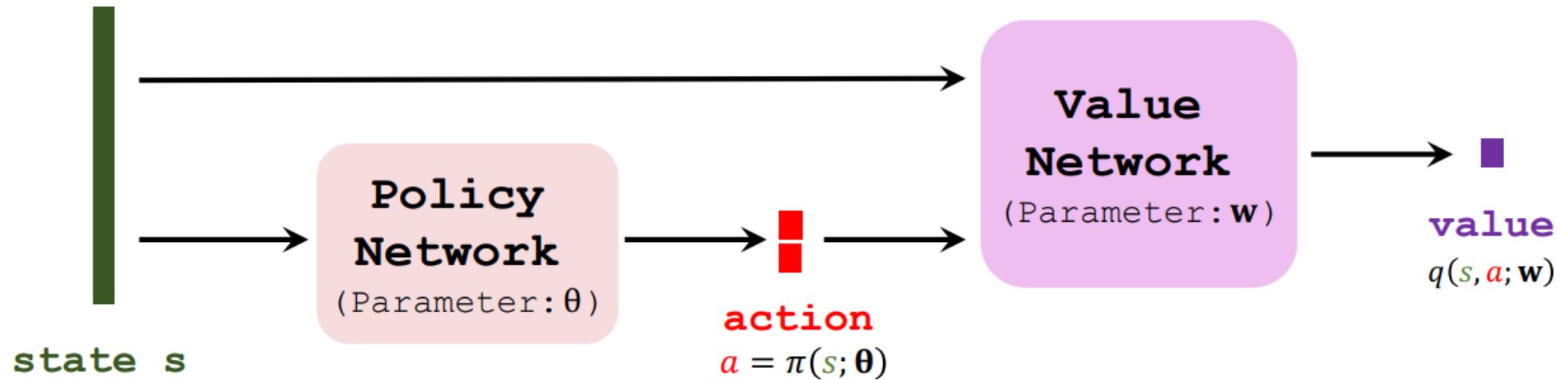
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \frac{\partial v(s_t; \mathbf{w})}{\partial \mathbf{w}}.$$

Advantage Actor-Critic (A2C)



Deterministic Actor-Critic

- Use a deterministic policy network (actor): $a = \pi(s; \theta)$.
- Use a value network (critic): $q(s, a; w)$.
- The critic outputs a scalar that evaluates *how good the action a is*.



Updating Value Network by TD

- Transition: (s_t, a_t, r_t, s_{t+1}) .
- Value network makes prediction for time t :

$$q_t = q(s_t, a_t; \mathbf{w}).$$

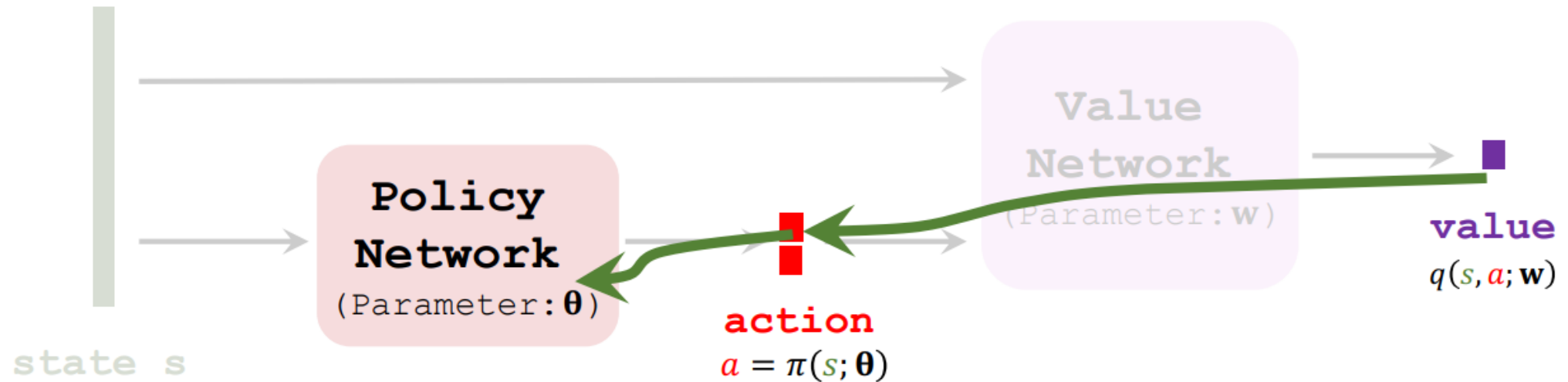
- Value network makes prediction for time $t + 1$:

$$q_{t+1} = q(s_{t+1}, a'_{t+1}; \mathbf{w}), \text{ where } a'_{t+1} = \pi(s_{t+1}; \boldsymbol{\theta}).$$

- TD error: $\delta_t = q_t - (r_t + \gamma \cdot q_{t+1})$.
- Update: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \frac{\partial q(s_t, a_t; \mathbf{w})}{\partial \mathbf{w}}$.

Updating Policy Network by DPG

- **Goal:** Increasing $q(s, a; \mathbf{w})$, where $a = \pi(s; \boldsymbol{\theta})$.
- DPG: $\mathbf{g} = \frac{\partial q(s, \pi(s; \boldsymbol{\theta}); \mathbf{w})}{\partial \boldsymbol{\theta}} = \frac{\partial a}{\partial \boldsymbol{\theta}} \cdot \frac{\partial q(s, a; \mathbf{w})}{\partial a}$.
- Gradient ascent: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta \cdot \mathbf{g}$.



Using target networks

- Value network makes a prediction for time t :

$$q_t = q(s_t, a_t; \mathbf{w}).$$

- Target networks make a prediction for time $t + 1$:

$$q_{t+1} = q(s_{t+1}, a'_{t+1}; \mathbf{w}^-), \text{ where } a'_{t+1} = \pi(s_{t+1}; \boldsymbol{\theta}^-).$$

Target value network

Target policy network

The same network structure, but different parameters.

Why Target Networks?



Bootstrapping



Target networks stabilize training by reducing variance.



They provide a consistent target despite the dynamic environment.



Target networks minimize temporal difference errors.



They separate target from learning networks to prevent feedback loops.

Updating policy and value networks

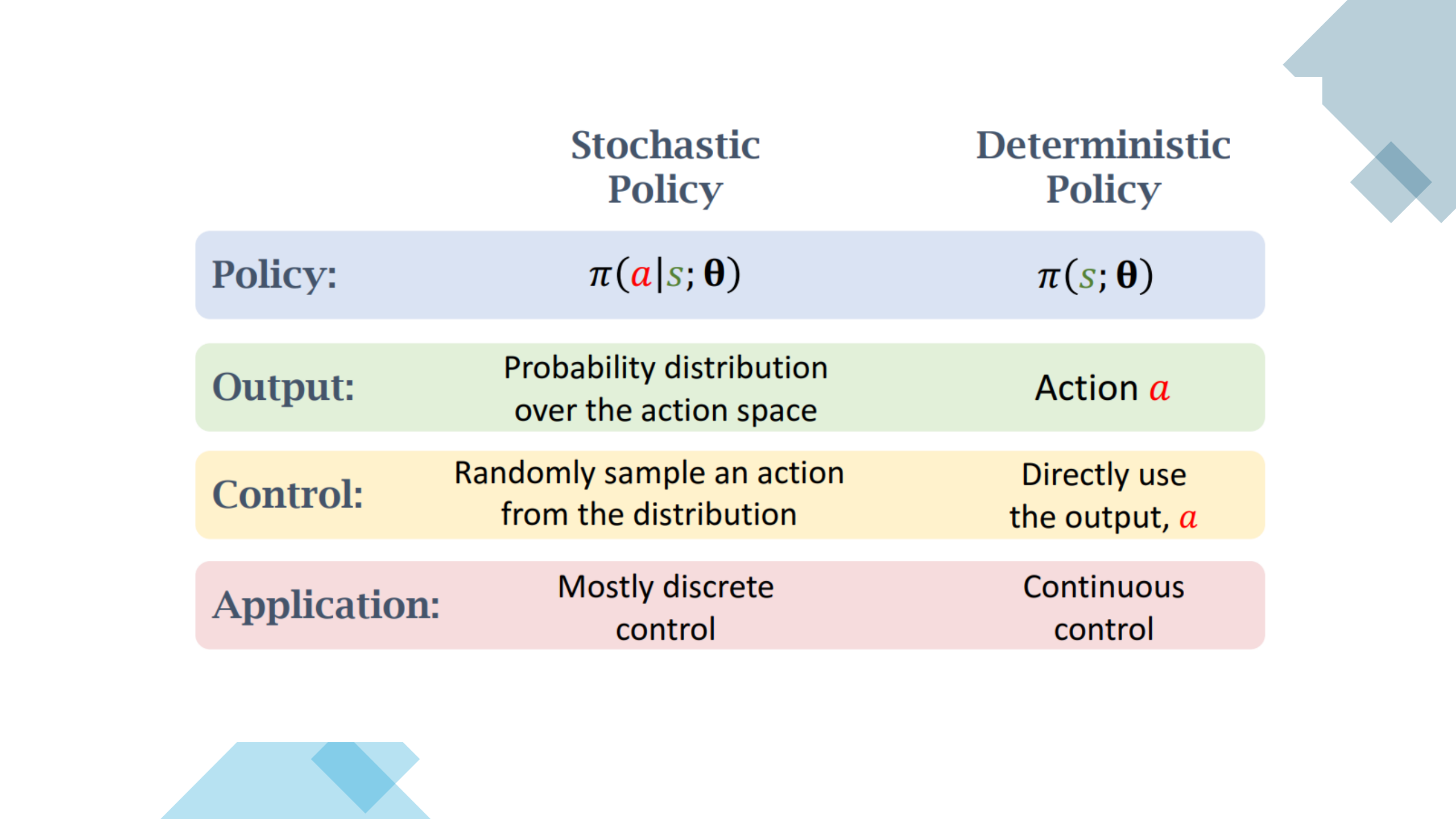
- Policy network makes a decision: $a = \pi(s; \theta)$.
- Update policy network by DPG: $\theta \leftarrow \theta + \beta \cdot \frac{\partial a}{\partial \theta} \cdot \frac{\partial q(s, a; \mathbf{w})}{\partial a}$.
- Value network computes $q_t = q(s, a; \mathbf{w})$.
- Target networks, $\pi(s; \theta^-)$ and $q(s, a; \mathbf{w}^-)$, compute q_{t+1} .
- TD error: $\delta_t = q_t - (r_t + \gamma \cdot q_{t+1})$.
- Update value network by TD: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \frac{\partial q(s, a; \mathbf{w})}{\partial \mathbf{w}}$.

Set a hyper-parameter $\tau \in (0, 1)$.

Update the target networks by weighted average

$$\mathbf{w}^- \leftarrow \tau \cdot \mathbf{w} + (1 - \tau) \cdot \mathbf{w}^-.$$

$$\theta^- \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta^-.$$



	Stochastic Policy	Deterministic Policy
Policy:	$\pi(a s; \theta)$	$\pi(s; \theta)$
Output:	Probability distribution over the action space	Action a
Control:	Randomly sample an action from the distribution	Directly use the output, a
Application:	Mostly discrete control	Continuous control

Need of TD3

A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.

TD3 in Rescue

Clipped Double-Q Learning. TD3 learns *two* Q-functions instead of one (hence “twin”) and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

“Delayed” Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Mathematics and Algorithm

- It's an off-policy algorithm.
- It concurrently learns two Q-functions by mean square Bellman error minimization.
- Clipped Double Q-Learning
 - Both Q-functions use a single target, calculated using whichever of the two functions gives a smaller target value.

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_i, \text{targ}}(s', a'(s')),$$

- and then both are learned by regressing to this target:

$$L(\phi_1, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right], \quad L(\phi_2, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right].$$

- The policy is learned just by maximizing Q_{ϕ_1}

$$\max_{\theta} \mathbb{E}_{s \sim \mathcal{D}} [Q_{\phi_1}(s, \mu_{\theta}(s))],$$

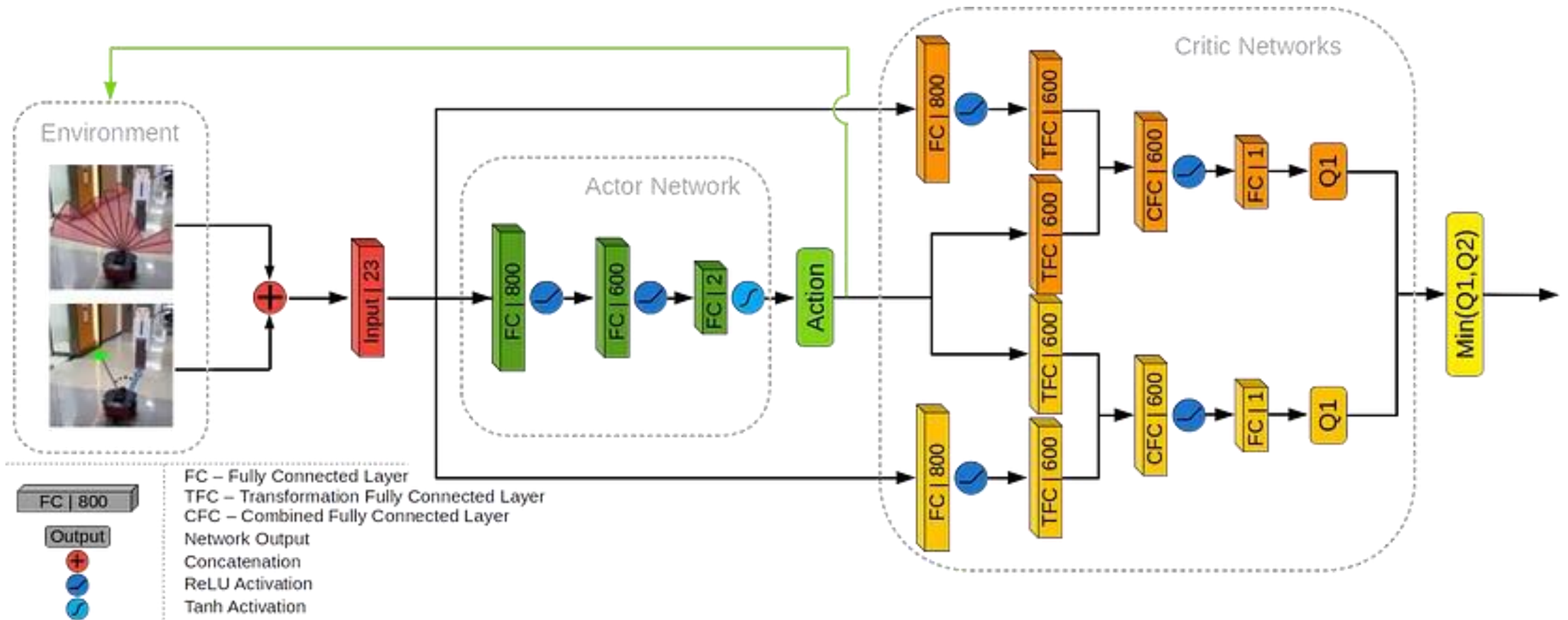
The policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

Where's the exploration?

- To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise.
- This is termed as **Target Policy Smoothing**



Architecture Used



```
class Actor(nn.Module):  
    def __init__(self, state_dim, action_dim):  
        super(Actor, self).__init__()  
  
        self.layer_1 = nn.Linear(state_dim, 800)  
        self.layer_2 = nn.Linear(800, 600)  
        self.layer_3 = nn.Linear(600, action_dim)  
        self.tanh = nn.Tanh()  
  
    def forward(self, s):  
        s = F.relu(self.layer_1(s))  
        s = F.relu(self.layer_2(s))  
        a = self.tanh(self.layer_3(s))  
        return a
```

```
class Critic(nn.Module):  
    def __init__(self, state_dim, action_dim):  
        super(Critic, self).__init__()  
  
        self.layer_1 = nn.Linear(state_dim, 800)  
        self.layer_2_s = nn.Linear(800, 600)  
        self.layer_2_a = nn.Linear(action_dim, 600)  
        self.layer_3 = nn.Linear(600, 1)  
  
        self.layer_4 = nn.Linear(state_dim, 800)  
        self.layer_5_s = nn.Linear(800, 600)  
        self.layer_5_a = nn.Linear(action_dim, 600)  
        self.layer_6 = nn.Linear(600, 1)  
  
    def forward(self, s, a):  
        s1 = F.relu(self.layer_1(s))  
        self.layer_2_s(s1)  
        self.layer_2_a(a)  
        s11 = torch.mm(s1, self.layer_2_s.weight.data.t())  
        s12 = torch.mm(a, self.layer_2_a.weight.data.t())  
        s1 = F.relu(s11 + s12 + self.layer_2_a.bias.data)  
        q1 = self.layer_3(s1)  
  
        s2 = F.relu(self.layer_4(s))  
        self.layer_5_s(s2)  
        self.layer_5_a(a)  
        s21 = torch.mm(s2, self.layer_5_s.weight.data.t())  
        s22 = torch.mm(a, self.layer_5_a.weight.data.t())  
        s2 = F.relu(s21 + s22 + self.layer_5_a.bias.data)  
        q2 = self.layer_6(s2)  
        return q1, q2
```

```

class TD3(object):
    def __init__(self, state_dim, action_dim, max_action):
        # Initialize the Actor network
        self.actor = Actor(state_dim, action_dim).to(device)
        self.actor_target = Actor(state_dim, action_dim).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters())

        # Initialize the Critic networks
        self.critic = Critic(state_dim, action_dim).to(device)
        self.critic_target = Critic(state_dim, action_dim).to(device)
        self.critic_target.load_state_dict(self.critic.state_dict())
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters())

        self.max_action = max_action
        self.writer = SummaryWriter()
        self.iter_count = 0

```

```

def get_action(self, state):
    # Function to get the action from the actor
    state = torch.Tensor(state.reshape(1, -1)).to(device)
    return self.actor(state).cpu().data.numpy().flatten()

def save(self, filename, directory):
    torch.save(self.actor.state_dict(), "%s/%s_actor.pth" % (directory, filename))
    torch.save(self.critic.state_dict(), "%s/%s_critic.pth" % (directory, filename))

def load(self, filename, directory):
    self.actor.load_state_dict(
        torch.load("%s/%s_actor.pth" % (directory, filename))
    )
    self.critic.load_state_dict(
        torch.load("%s/%s_critic.pth" % (directory, filename))
    )

```

```
def train(
    self,
    replay_buffer,
    iterations,
    batch_size=100,
    discount=0.99,
    tau=0.005,
    policy_noise=0.2,
    noise_clip=0.5,
    policy_freq=2,
):
    av_Q = 0
    max_Q = -inf
    av_loss = 0
```

Training

- replay_buffer – where to get samples for training from,
- iterations – how many trainings to run this function call,
- batch_size – how many samples to sample in each iteration,
- discount – discount factor in Bellman equation,
- tau – how much of base network parameters we will add to target networks in soft update,
- policy_noise – std. deviation of the noise to add to the action for exploration,
- noise_clip – min and max value to clip the noise with,
- policy_freq – how often to update the actor-network parameters

Training

```
for it in range(iterations):
    # sample a batch from the replay buffer
    (
        batch_states,
        batch_actions,
        batch_rewards,
        batch_dones,
        batch_next_states,
    ) = replay_buffer.sample_batch(batch_size)
    state = torch.Tensor(batch_states).to(device)
    next_state = torch.Tensor(batch_next_states).to(device)
    action = torch.Tensor(batch_actions).to(device)
    reward = torch.Tensor(batch_rewards).to(device)
    done = torch.Tensor(batch_dones).to(device)

    # Obtain the estimated action from the next state by using the actor-target
    next_action = self.actor_target(next_state)

    # Add noise to the action
    noise = torch.Tensor(batch_actions).data.normal_(0, policy_noise).to(device)
    noise = noise.clamp(-noise_clip, noise_clip)
    next_action = (next_action + noise).clamp(-self.max_action, self.max_action)
```

```
# Calculate the Q values from the critic-target network for the next state-action
target_Q1, target_Q2 = self.critic_target(next_state, next_action)

# Select the minimal Q value from the 2 calculated values
target_Q = torch.min(target_Q1, target_Q2)
av_Q += torch.mean(target_Q)
max_Q = max(max_Q, torch.max(target_Q))

# Calculate the final Q value from the target network parameters by using Bellman
target_Q = reward + ((1 - done) * discount * target_Q).detach()

# Get the Q values of the basis networks with the current parameters
current_Q1, current_Q2 = self.critic(state, action)

# Calculate the loss between the current Q value and the target Q value
loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)

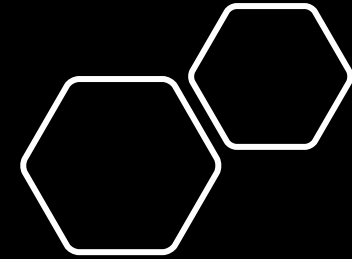
# Perform the gradient descent
self.critic_optimizer.zero_grad()
loss.backward()
self.critic_optimizer.step()
```

```
if it % policy_freq == 0:
    # Maximize the actor output value by performing gradient descent on negative Q
    # (essentially perform gradient ascent)
    actor_grad, _ = self.critic(state, self.actor(state))
    actor_grad = -actor_grad.mean()
    self.actor_optimizer.zero_grad()
    actor_grad.backward()
    self.actor_optimizer.step()

    # Use soft update to update the actor-target network parameters by
    # infusing small amount of current parameters
    for param, target_param in zip(
        self.actor.parameters(), self.actor_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )

    # Use soft update to update the critic-target network parameters by infusing
    # small amount of current parameters
    for param, target_param in zip(
        self.critic.parameters(), self.critic_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )

av_loss += loss
```



Results

The results obtained are after training the model for over 19 hours and 4500 episodes.

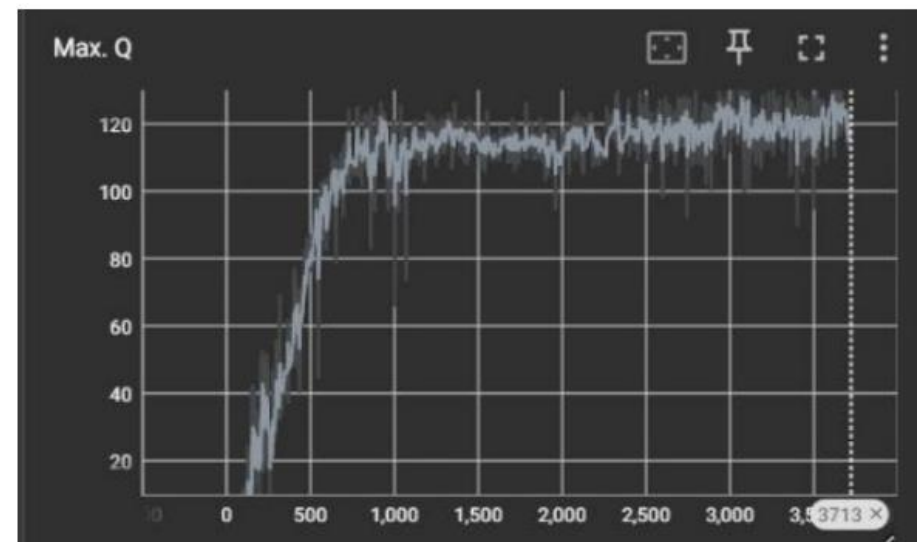
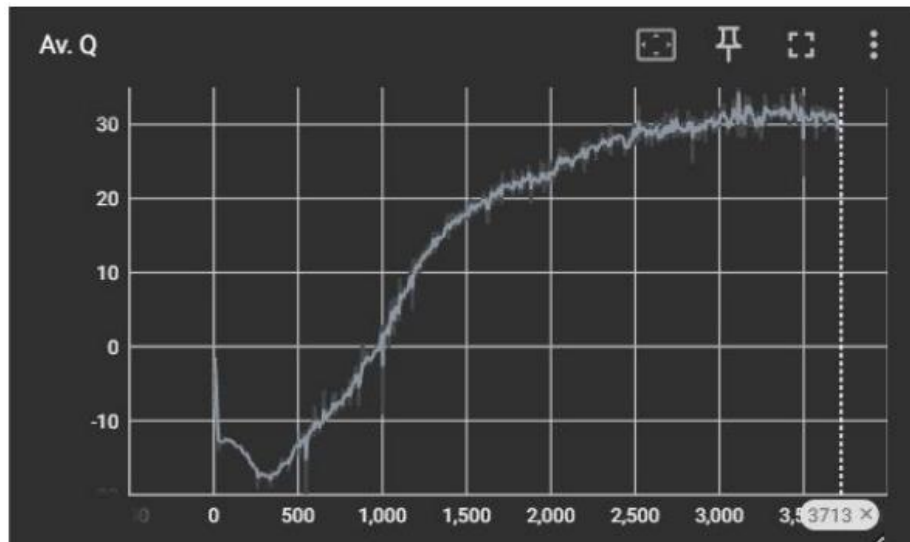
- The score of each epoch was calculated by getting the average score of 10 episodes.
- After each iteration, the Q values between critic-target network for the next state-action pair are calculated and the average and maximum Q values are recorded.
- Using Bellman equation, the target Q value is calculated

Epoch	Score	Epoch	Score	Epoch	Score	Epoch	Score	Epoch	Score	Epoch	Score	Epoch	Score
1	-91.3	8	7.6	15	32.5	22	84.4	29	61.8	36	55.8	43	47.6
2	-62.6	9	-9.4	16	48.9	23	39.1	30	100.8	37	100.7	44	101.2
3	-67.5	10	-51.5	17	-3.4	24	52.9	31	42.3	38	70.0	45	89.6
4	-89.2	11	-47.4	18	17.2	25	78.0	32	80.9	39	25.3	46	79.8
5	4.5	12	-26.4	19	43.2	26	75.5	33	71.2	40	100.1	47	100.0
6	-26.5	13	-27.2	20	52.6	27	79.5	34	99.3	41	38.1	48	73.7
7	-53.5	14	-108.9	21	49.6	28	32.2	35	66.5	42	35.9	49	80.7

Results

The results obtained are after training the model for over 19 hours and 4500 episodes.

- The score of each epoch was calculated by getting the average score of 10 episodes.
- After each iteration, the Q values between critic-target network for the next state-action pair are calculated and the average and maximum Q values are recorded.
- Using Bellman equation, the target Q value is calculated



Summary

Fully autonomous exploration system driven by Deep Reinforcement Learning (DRL) eliminates the need for direct human supervision.

Successful integration of reactive local and global navigation strategies enhances system performance.

Goal-Driven Autonomous Exploration (GDAE) system achieves results resembling those obtained by path planners in familiar environments.

Future developments aim to enhance system versatility by incorporating robot dynamics and Long Short-Term Memory (LSTM) architecture.