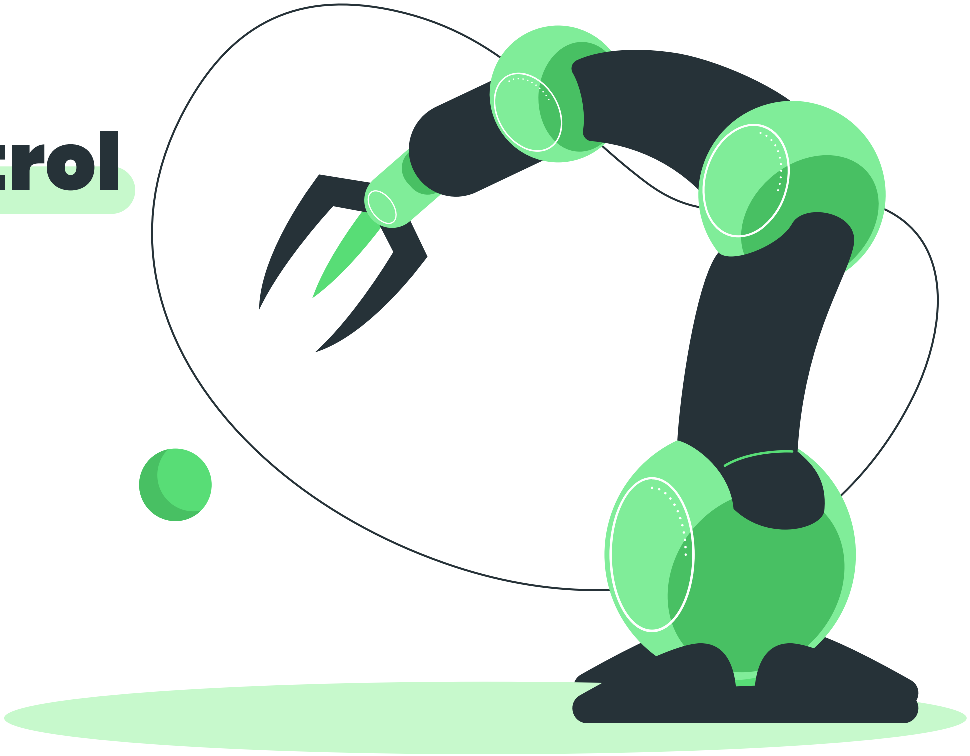
Learning Continuous Control using Inverse Reinforcement Learning



Shaz Nazar Karumarot

Table of Contents

1

Introduction And Objectives

Introduction

Challenge: Solve **complex** tasks using raw sensory input (high-dimensional, unprocessed).

Advancement: Deep Q-Network (**DQN**) solves problems with high-dimensional and **continuous observation spaces**.

Limitation: DQN can only handle low-dimensional and **discrete action spaces**.

- Many tasks, especially **physical control**, have **continuous** and high-dimensional action spaces.
- DQN relies on finding the best (maximum valued) action, which requires an **iterative optimization process in continuous domains** (not efficient).

Introduction

Partial Solution: **Discretizing** the **action space**.

- **Curse of dimensionality**: action space grows exponentially with degrees of freedom.
    - Example: 7-DOF arm (as in the human arm) with coarse discretization leads to massive action space ($3^7 = 2187$).
- Finer control requires **finer discretization**, further exploding the number of actions.
- Large action spaces are difficult to **explore** efficiently, hindering training.
- Naive discretization results in **loss of valuable information** about the action domain.

Objectives

Goal: Develop agents that can **learn complex continuous control tasks** through apprenticeship using inverse reinforcement learning (**IRL**).

Challenges:

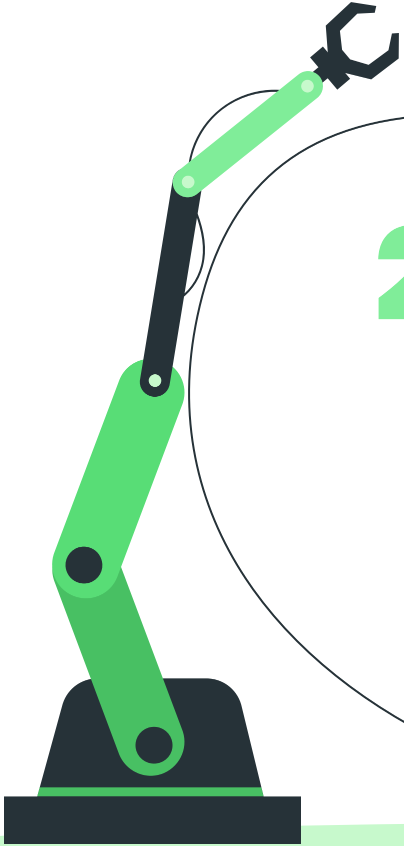
- Learning from **expert demonstrations**: Extract the underlying reward function from observed expert behavior in continuous control domains.

- **Continuous Action Space**: Design an agent that can effectively handle continuous and potentially high-dimensional action spaces for real-world control tasks.

Objectives

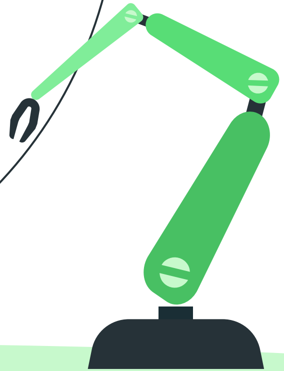
Expected Outcomes:

- Train agents that can achieve high performance on continuous control tasks by learning from expert demonstrations **rather than explicit reward functions** defined by us.
- Use suitable state of the art algorithms to train the apprentice agents.
- Implement an efficient **IRL algorithm** applicable to such control problems.
- Demonstrate the effectiveness of apprenticeship learning for training agents that achieve **performance close to the expert** (or even better).



2

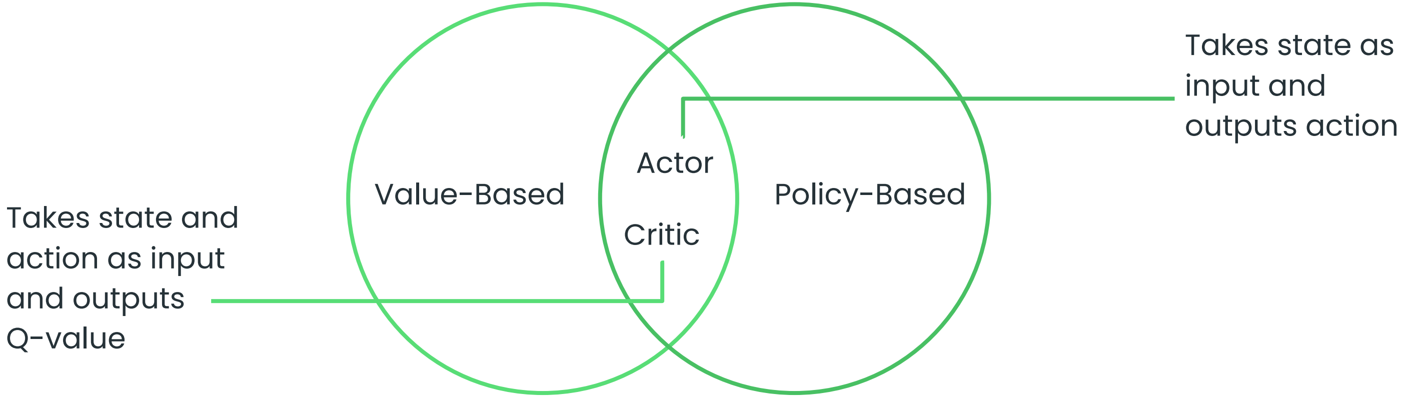
Reinforcement Learning Algorithms



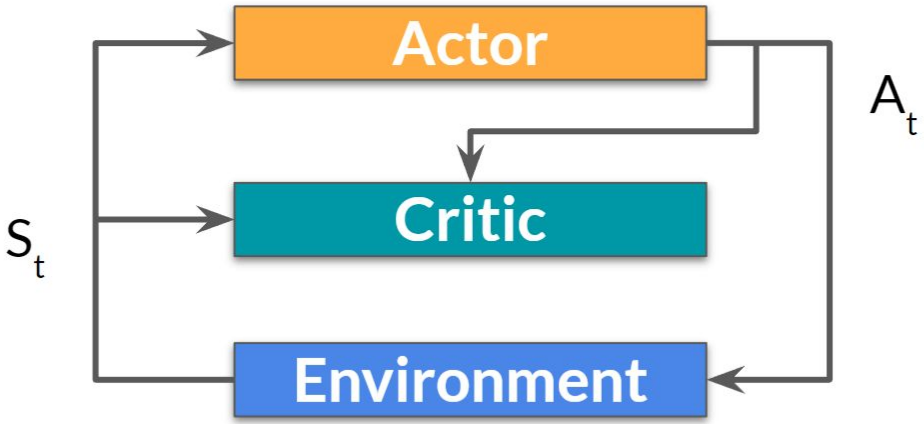
Deep Deterministic Policy Gradient (DDPG)

DDPG

- A model-free **off-policy** reinforcement learning algorithm for **learning continuous actions**.
- Concurrently learns a **Q-function** and a **policy**. It uses off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy.
- Incorporates an **actor-critic** approach based on **DPG**.

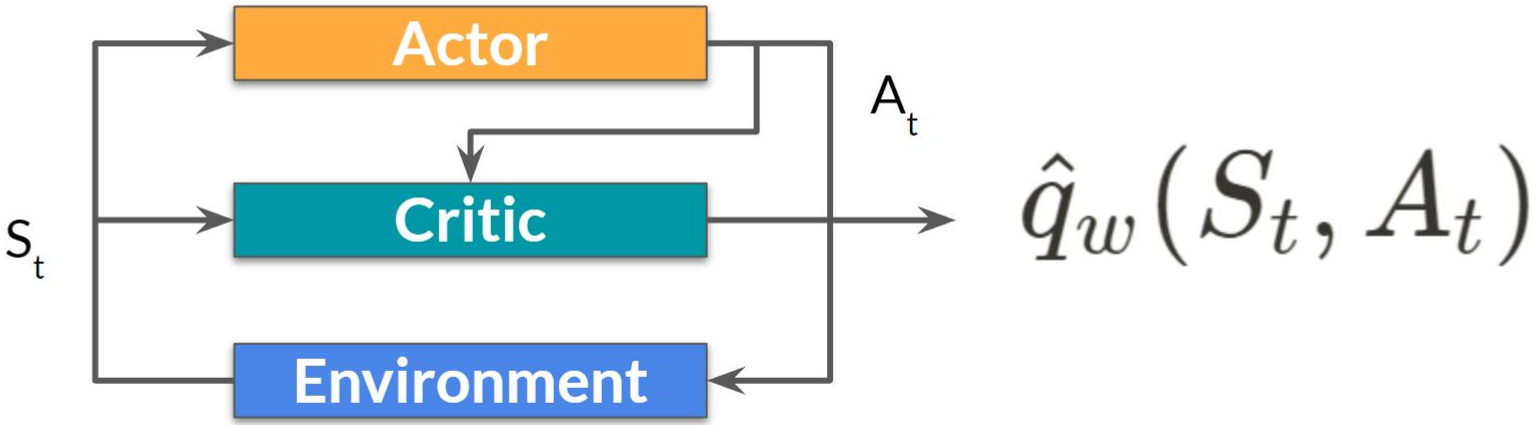


DDPG



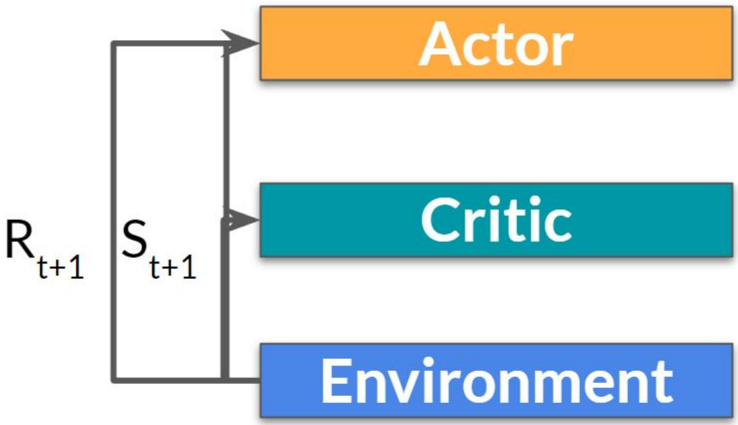
Source: Hugging Face Deep Reinforcement Learning Course - Hugging Face. (n.d.).

DDPG



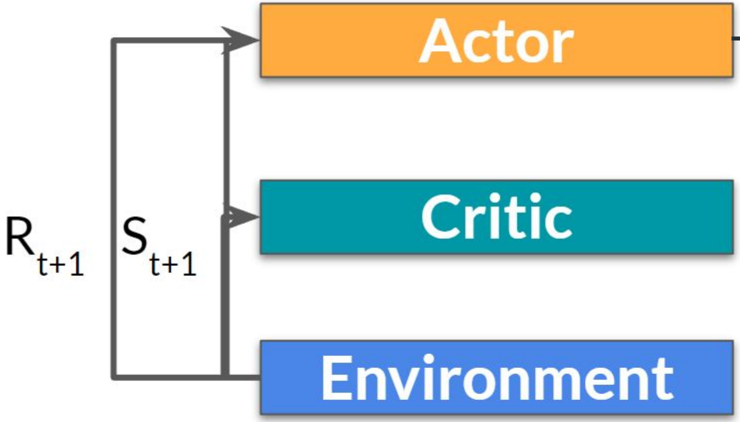
Source: Hugging Face Deep Reinforcement Learning Course - Hugging Face. (n.d.).

DDPG



Source: Hugging Face Deep Reinforcement Learning Course - Hugging Face. (n.d.).

DDPG



$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s,a))\hat{q}_w(s,a)$$

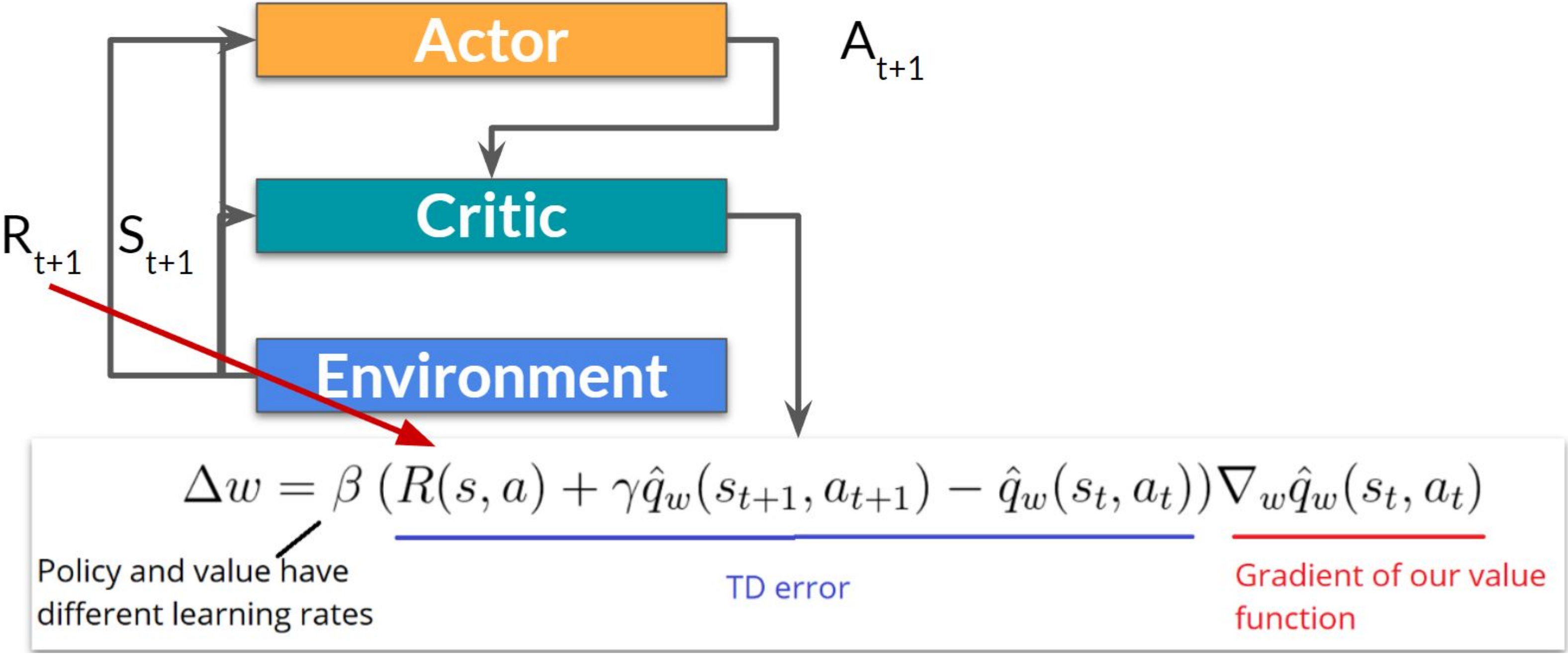
Change in policy parameters (weights)

Action value estimate

Gradient Ascent

Source: Hugging Face Deep Reinforcement Learning Course - Hugging Face. (n.d.).

DDPG



Source: Hugging Face Deep Reinforcement Learning Course - Hugging Face. (n.d.).

DDPG

Stochastic Policy

State → π(a,s) → Probability

$$a_t = \max_a Q^*(\phi(s_t), a; \theta)$$

Deterministic Policy

State → π(s) → Action A

$$a_t = \mu(s_t|\theta^\mu)$$

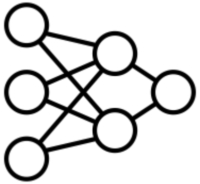
To **explore** more states, we add noise N (off-policy):

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$$

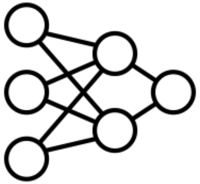
DDPG

- Similar to Deep Q-Network (DQN), DDPG employs a **replay buffer** to store transitions (state, action, reward, next state) collected during exploration.
- The buffer allows learning from a diverse set of **uncorrelated transitions**, even when using mini batches for updates.
- Directly applying Q-learning with neural networks can be unstable because the network used to calculate the target value is also being updated.
- DDPG addresses this by introducing **separate target networks** for the actor and critic (Q0 and μ0) that are used to calculate target values.
- The target networks are **time-delayed copies** of their original networks that slowly track the learned networks using a **soft update** rule - Polyak Averaging.

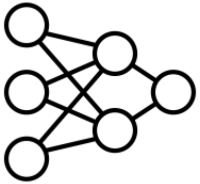
DDPG



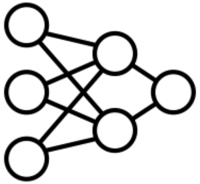
Actor



Critic 1



Target Actor



Target Critic 1

DDPG

Algorithm 1 DDPG algorithm

Randomly initialize $\boxed{\text{critic network } Q(s, a|\theta^Q)}$ and $\boxed{\text{actor } \mu(s|\theta^\mu)}$ with weights θ^Q and θ^μ.
Initialize $\boxed{\text{target network } Q' \text{ and } \mu' \text{ with weights } \theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu}$
Initialize $\boxed{\text{replay buffer } R}$
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select $\boxed{\text{action } a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t}$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\boxed{\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}}$$

    Update the target networks:

$$\boxed{\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \end{aligned}}$$

  end for
end for

Target Networks to do off-policy updates.

Action selected by deterministic actor and noise is added for exploration

Experience Replay

Policy Gradient

Slow updates using soft update (Polyak Averaging), increases stability

T << 1

Source: Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, & Daan Wierstra. (2019). Continuous control with deep reinforcement learning.

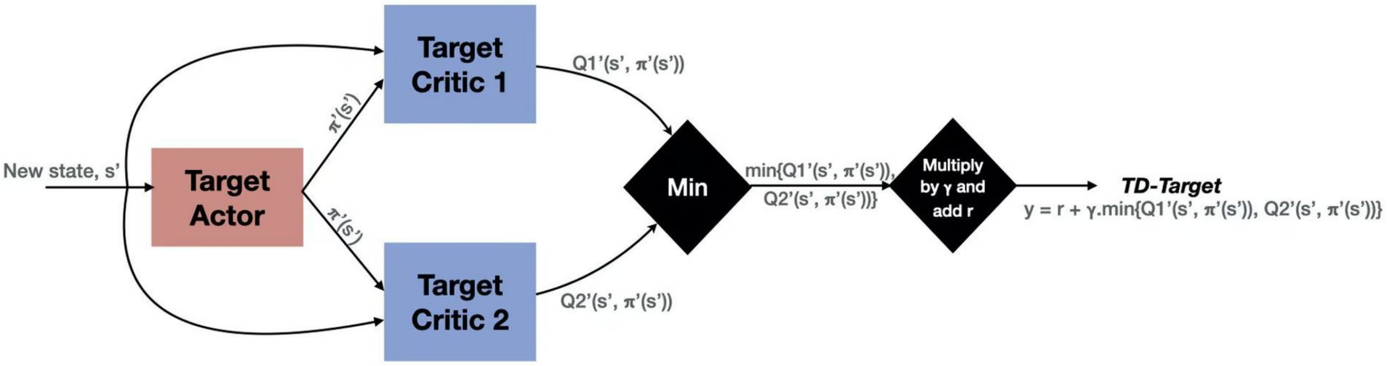
Twin Delayed Deep Deterministic Policy Gradient (TD3)

TD3

- Neural networks used for Q-value approximation can introduce noise, leading to **overestimated Q-values**, especially when taking the max.
    - **Slows** down learning
    - **Suboptimal policy** due to selecting bad actions based on biased Q-values.
- TD3 focuses on Actor-Critic settings and addresses the root cause: variance/noise in Q-values and employs three key techniques to deal with it:
    - **Clipped Double Q-Learning**: Modifies target calculation to reduce bias.
    - **Delayed Policy and Target Updates**: Stabilizes learning by updating critics more frequently than the actor and target networks.
    - **Target Policy Smoothing**: Regularizes Q-values by adding noise to target policy actions.

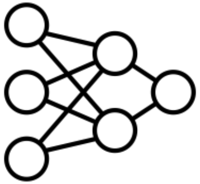
Clipped Double Q Learning

- Standard Double Q-Learning assumes **independent Q-value updates** for unbiased estimates, but DDPG's replay buffer does not guarantee this.
- TD3 addresses this with "clipped" Double Q-Learning:
    - Takes the **minimum** of two Q-value estimates (Q1 and Q2) for target calculation.
- **Two critics** (Q1 & Q2) with target networks (Q1' & Q2') are used for efficiency.
- Only **one actor** (π) is optimized against Q1 to reduce computation.

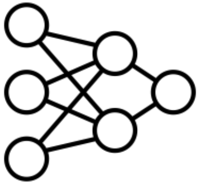


Source: Twin delayed DDPG (TD3): Theory. (n.d.).

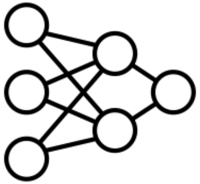
TD3 Neural Networks



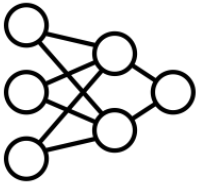
Actor



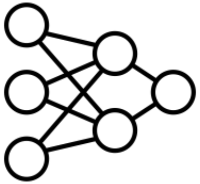
Critic 1



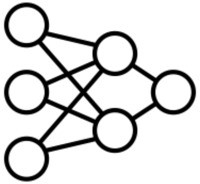
Critic 2



Target Actor



Target Critic 1

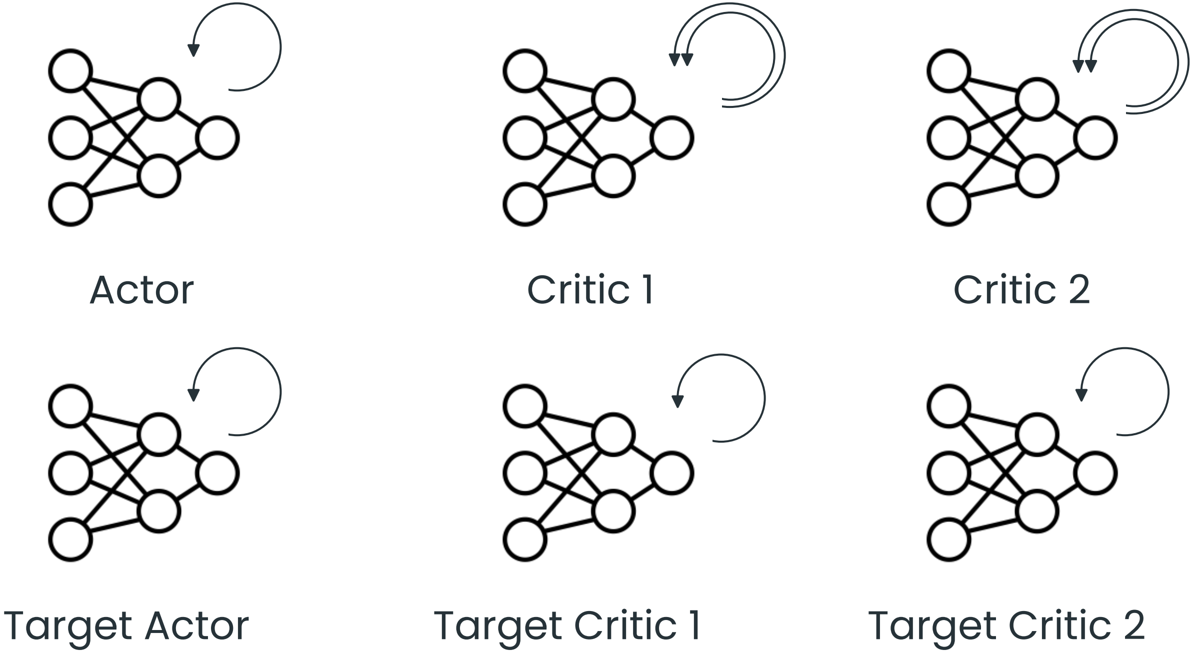


Target Critic 2

Delayed Policy and Target Updates

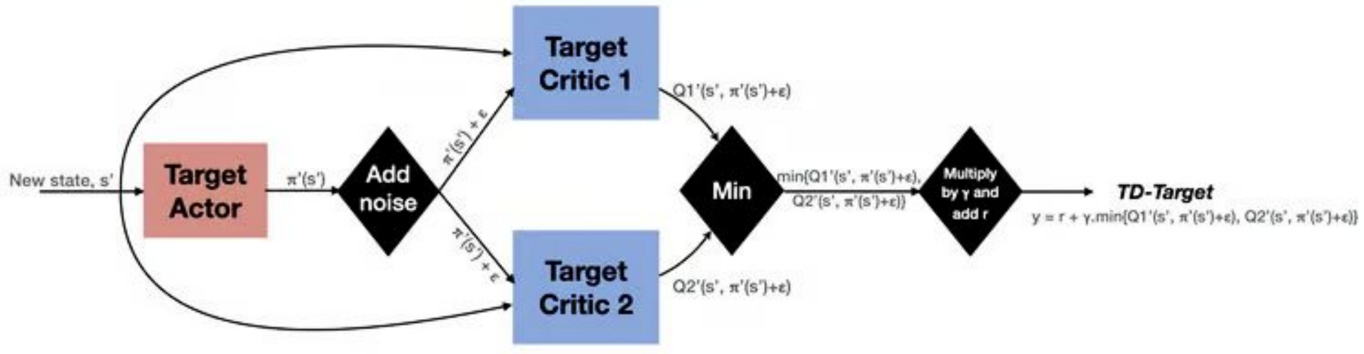
- The interaction between actor and critic can also cause **instability**.
  - Inaccurate value estimates from the critic can lead to a poor policy.
  - A poor policy can further worsen the value estimates.
- Solution: **Update critic networks more frequently** than the actor and target networks (e.g., every step vs. every other step).
  - This allows critic Q-values to converge, reducing value error.
  - More stable Q-values lead to better policy updates by the actor.
- Also uses soft update of target networks using Polyak Averaging.

Delayed Policy and Target Updates



Target Policy Smoothing

- In continuous action spaces, **nearby actions** should have **similar Q-values** for a given state.
  - Deterministic policies can **overfit to specific actions** otherwise, leading to brittle policies.
- Add small **Gaussian noise** with a low standard deviation to the target policy's actions.
  - This ensures similar Q-values for actions close together in the continuous space.
- The noise and perturbed action are **clipped** which ensures it only affects a small region around the action.
- **Action clipping** keeps the perturbed action within valid action values.



Source: Twin delayed DDPG (TD3): Theory. (n.d.).

TD3

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ

Two Critic networks and One actor network

Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$

Three target networks

Initialize replay buffer $\mathcal{B}$

for $t = 1$ **to** T **do**

Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'

Add a small zero-mean Gaussian noise to induce Stochastic Behaviour

Store transition tuple (s, a, r, s') in $\mathcal{B}$

Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$

$\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$

Target Policy Smoothing

$y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$

Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$

Clipped Double Q-Learning

if t mod d **then**

Update ϕ by the deterministic policy gradient:

$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

Delayed updates of the actor and target networks

Update target networks:

$\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$

$\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

Slow updates for target networks using soft update (Polyak Averaging)

T << 1

end if

end for

Source: Scott Fujimoto, Herke van Hoof, & David Meger (2018). Addressing Function Approximation Error in Actor-Critic Methods. CoRR, abs/1802.09477.

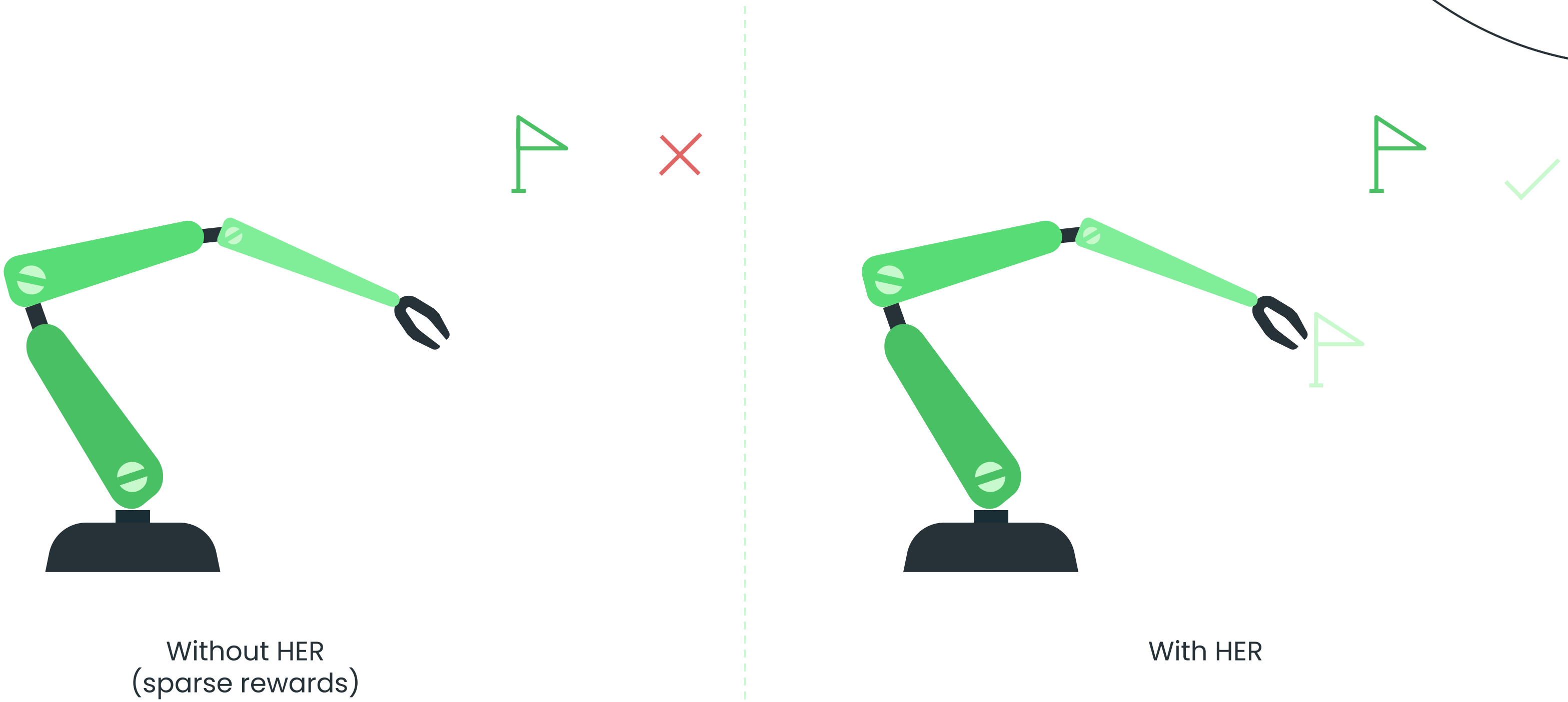
Hindsight Experience Replay (HER)

HER

Hindsight Experience Replay expands learning, especially in sparse reward environments, by creating "imagined" **successful experiences from failures**.

- Key Idea: Reframe failures as successes for different goals. Imagine the **achieved state** (S') was the **intended goal** all along.
- Real Experience: Agent interacts with the environment aiming for goal **G**, but ends up in state **S'**.
- Real Experience Storage: Stores the **actual experience** (states, actions, rewards) in the replay buffer. (S, a_k, r_k, S')
- Imagined Success: Creates a new experience where S' becomes the **achieved goal**.
- Imagined Experience Storage: Stores the imagined experience with **positive reward** in the replay buffer. (S, a_k, R+, S') (R+ is a positive reward for reaching the imagined goal)

HER



Without HER
(sparse rewards)

With HER

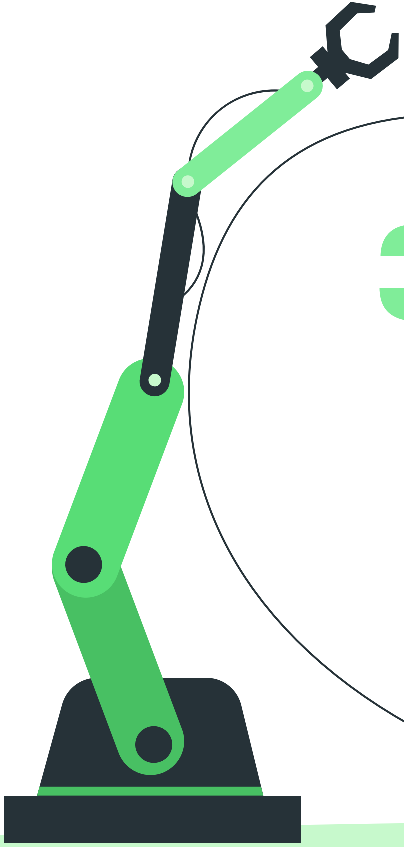
HER

Benefits:

- **Learning from Failures**: Even during bad policy stages, the agent has positive experiences to learn from.
- **Improved Generalization**: By learning from diverse (real & imagined) experiences, the agent generalizes better to unseen states similar to the "imagined goals."
- **Sample Efficiency**: Learning from hindsight improves efficiency by extracting valuable insights from diverse interactions.

Learning:

- Initially, the agent might reach random states which become the initial "imagined goals" for learning.
- The agent's policy improves and reaches states **closer to the actual goals**.
- Learning from both real and imagined successes helps the agent achieve the desired goals eventually.



3

Inverse Reinforcement Learning Algorithm

Preliminaries

- We consider an **MDP \ R** - (S, A, T, γ, D)
- State Features (φ) and True Reward Function (R*):
  - **φ(s):** Represents a **vector of features** associated with each state, providing additional information beyond the basic state identity.
  - **R*(s) = w* · φ(s):** Defines a **"true" reward function** based on the **feature vector** and a **weight vector (w*)**.

- **Feature Expectations (μ(π))**: The expected discounted accumulated feature vector for a policy π (captures the long-term effects of a policy on state features).

$$\mu(\pi) = E[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi] \in \mathbb{R}^k.$$

Preliminaries

- Learning from Expert Demonstrations:
  - We assume access to **demonstrations** generated by an expert policy ($\boldsymbol{\pi_E}$).
  - We can estimate the expert's feature expectations (μ_E) from observed monte carlo **trajectories**.
  - The empirical estimate for $\boldsymbol{\mu_E} = \mu(\pi_E)$ based on a set of *m* observed expert trajectories is given by:

$$m \text{ trajectories } \{s_0^{(i)}, s_1^{(i)}, \ldots\}_{i=1}^{m}$$

$$\hat{\mu}_E = \underbrace{\boxed{\frac{1}{m}\sum_{i=1}^{m}}}_{\text{Mean over all demonstrations}} \underbrace{\boxed{\sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)})}}_{\text{Discount-factor weighted sum of feature vectors}}$$

Algorithm: Max-Margin

Problem: Find a policy π̃ that induces feature expectations μ(π̃) close to μE.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \le 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \le \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Initialize a random policy, compute it's feature expectations

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin

Problem: Find a policy π̃ that induces feature expectations μ(π̃) close to μE.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \le 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \le \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Inverse Reinforcement Learning step:

Optimize w, using a Quadratic Programming Solver or SVM, to maximize the margin t between the expert and the best policy found thus far.

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin

Problem: Find a policy π̃ that induces feature expectations μ(π̃) close to μE.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \leq 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \leq \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Terminate within the threshold margin, i.e., when margin ≤ ϵ

(Returns a set of policies - pick one with best performance)

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin

Problem: Find a policy $\tilde{\pi}$ that induces feature expectations $\mu(\tilde{\pi})$ close to μE.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \leq 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \leq \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Retrain using new weights and rewards (reward = $\mathbf{w}^{\mathbf{T}} \cdot \boldsymbol{\varphi}$) to obtain new policy.

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin

Problem: Find a policy π̃ that induces feature expectations μ(π̃) close to μE.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \le 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \le \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Compute feature expectation of the new policy

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin

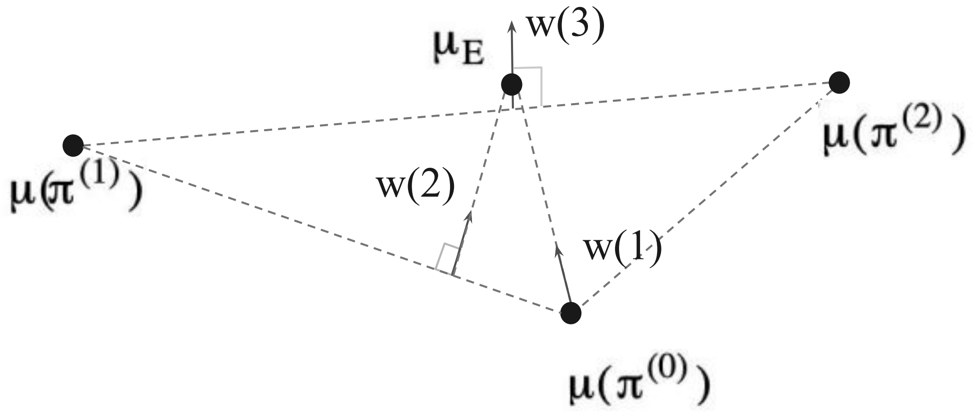
Problem: Find a policy $\tilde{\pi}$ that induces feature expectations $\mu(\tilde{\pi})$ close to μ_E.

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:\|w\|_2 \leq 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.
3. If $t^{(i)} \leq \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

Repeat to get new policies until termination

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Max-Margin



First three iterations of the max-margin algorithm

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Projection-Method

Problem: Find a policy $\tilde{\pi}$ that induces feature expectations $\mu(\tilde{\pi})$ close to μ_E.

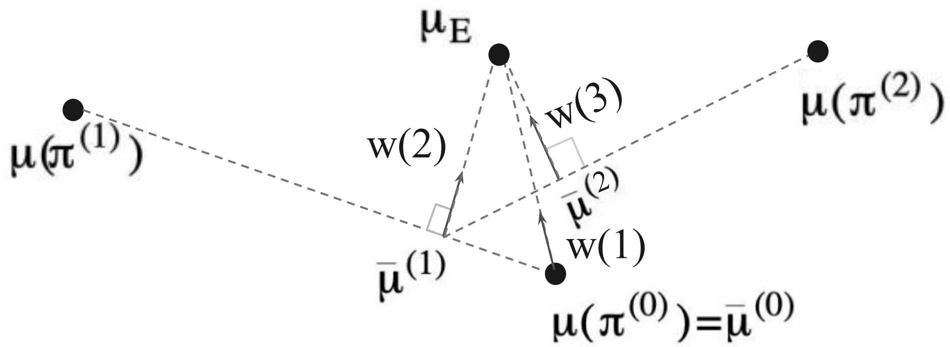
1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. ~~Compute $t^{(i)} = \max_{w:\|w\|_2 \le 1} \min_{j \in \{0..(i-1)\}} w^T(\mu_E - \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum.~~
3. If $t^{(i)} \le \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

- Set $\bar{\mu}^{(i-1)} = \bar{\mu}^{(i-2)} + \frac{(\mu^{(i-1)} - \bar{\mu}^{(i-2)})^T(\mu_E - \bar{\mu}^{(i-2)})}{(\mu^{(i-1)} - \bar{\mu}^{(i-2)})^T(\mu^{(i-1)} - \bar{\mu}^{(i-2)})}(\mu^{(i-1)} - \bar{\mu}^{(i-2)})$ (This computes the orthogonal projection of μ_E onto the line through $\bar{\mu}^{(i-2)}$ and $\mu^{(i-1)}$.)
- Set $w^{(i)} = \mu_E - \bar{\mu}^{(i-1)}$
- Set $t^{(i)} = \|\mu_E - \bar{\mu}^{(i-1)}\|_2$

Eliminates need for QP Solver

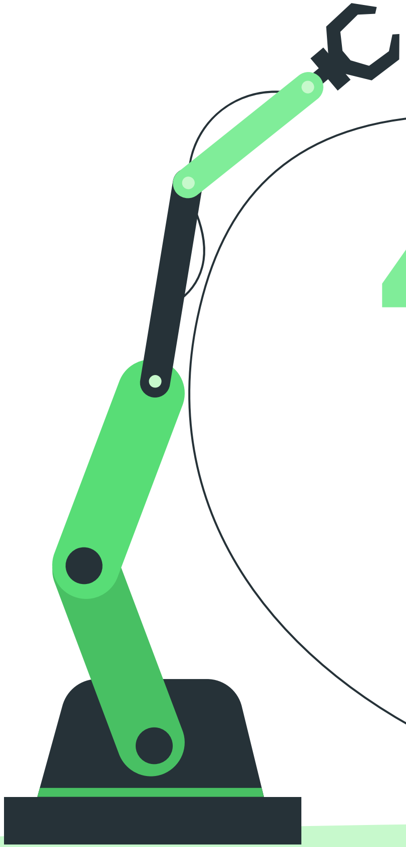
Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5

Algorithm: Projection-Method



First three iterations of the projection algorithm

Source: Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. ICML '04: Proceedings of the twenty-first international conference on Machine learning (p./pp. 1), New York, NY, USA: ACM. ISBN: 1-58113-828-5



4

Experiments and Results

Environment

- **Panda-gym** - a toolkit for training robots using Reinforcement Learning (RL).
- **Simulates** the 7-DOF Panda robot arm with a parallel gripper.
- Observation Space:
  - Gripper **position & speed (6D)**
  - Gripper opening (if applicable) (1D)
- Action Space:
  - **Gripper movement (3D: x, y, z)**
  - Gripper opening/closing (1D) (optional)

Source

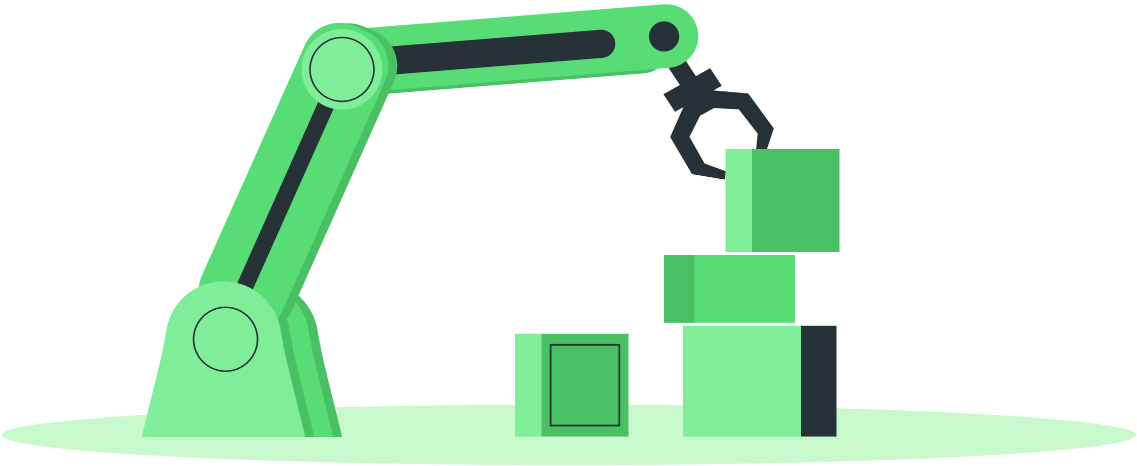
Panda Gym: Quentin Gallouédec, Nicolas Cazin, Emmanuel Dellandréa, & Liming Chen. (2021).
panda-gym: Open-source goal-conditioned environments for robotic learning.

Environment

- Simulation:
  - Runs at 25 Hz (20 simulation steps per agent action)
  - Episodes last 2 seconds
- Reward Functions:
  - **Sparse Reward** (default): **0** for successful task completion within a tolerance (5cm) and **-1** otherwise.
- Benefits:
  - Enables training complex tasks in **simulation** before real-world deployment.
  - **Open-Source** and Flexible: Freely available and allows defining new tasks and robots.

Task: PandaReach-v3

A **target position** must be reached with the gripper. This target position is **randomly generated** in a volume of **30 cm × 30 cm × 30 cm**.



DDPG (with HER)

Experimental Setup

- Environment: **PandaReach-v3**
- Algorithm: Deep Deterministic Policy Gradient (**DDPG**)
- Hyperparameters:
  - alpha=0.001
  - beta=0.002
  - gamma=0.99
  - tau=0.05
  - batch_size=256
  - replay_size=10**6
  - noise_factor=0.1
  - exploration_period=200
  - n_episodes=500

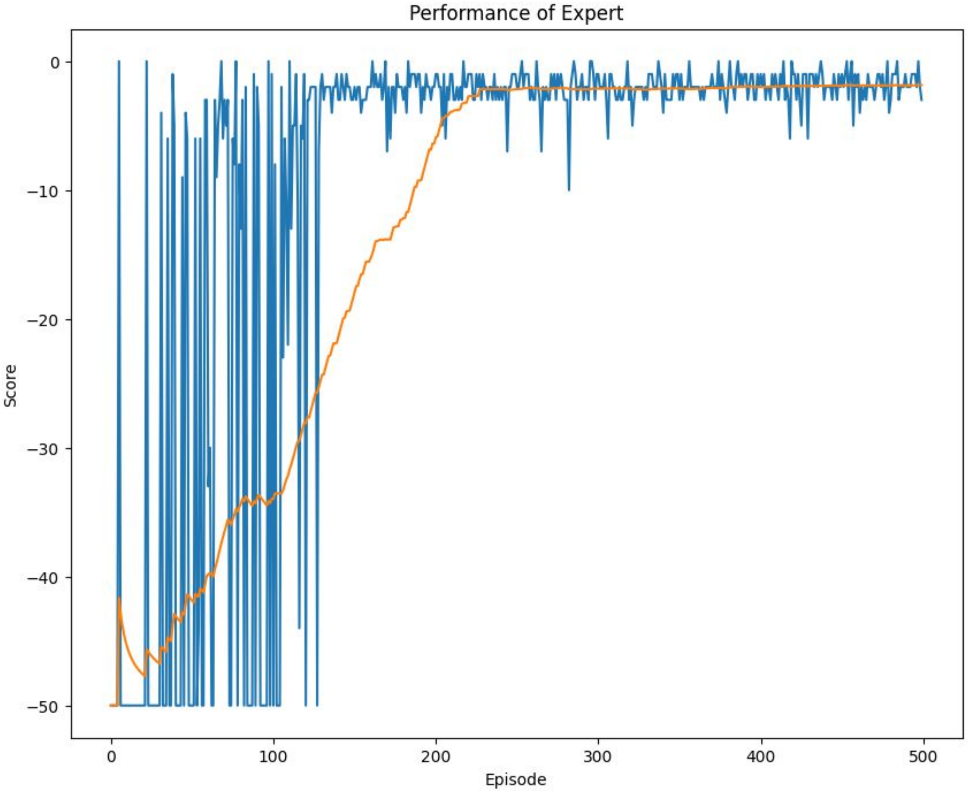
Code: https://github.com/Shaz-5/manipulator-inverse-rl

DDPG (with HER)

```python
# Actor and Critic Networks
class Actor(nn.Module):
    def __init__(self, state_shape, num_actions, name, checkpoints_dir="../Data/"):
        super(Actor, self).__init__()
        if not os.path.exists(checkpoints_dir):
            os.makedirs(checkpoints_dir)
        self.checkpoints_file = os.path.join(checkpoints_dir, name + ".pth")

        self.hidden1 = nn.Linear(in_features=state_shape, out_features=512)
        self.hidden2 = nn.Linear(in_features=512, out_features=256)
        self.hidden3 = nn.Linear(in_features=256, out_features=256)
        self.action_output = nn.Linear(in_features=256, out_features=num_actions)

    def forward(self, state):
        x = torch.relu(self.hidden1(state))
        x = torch.relu(self.hidden2(x))
        x = torch.relu(self.hidden3(x))
        action = torch.tanh(self.action_output(x))
        return action

class Critic(nn.Module):
    def __init__(self, state_action_shape, name, checkpoints_dir="../Data/"):
        super(Critic, self).__init__()
        if not os.path.exists(checkpoints_dir):
            os.makedirs(checkpoints_dir)
        self.checkpoints_file = os.path.join(checkpoints_dir, name + ".pth")

        self.hidden1 = nn.Linear(in_features=state_action_shape, out_features=512)
        self.hidden2 = nn.Linear(in_features=512, out_features=256)
        self.hidden3 = nn.Linear(in_features=256, out_features=256)
        self.q_value = nn.Linear(in_features=256, out_features=1)

    def forward(self, state, action):
        x = torch.cat([state, action], dim=1)
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = torch.relu(self.hidden3(x))
        q_value = self.q_value(x)
        return q_value
```

Code: https://github.com/Shaz-5/manipulator-inverse-rl

DDPG (with HER)

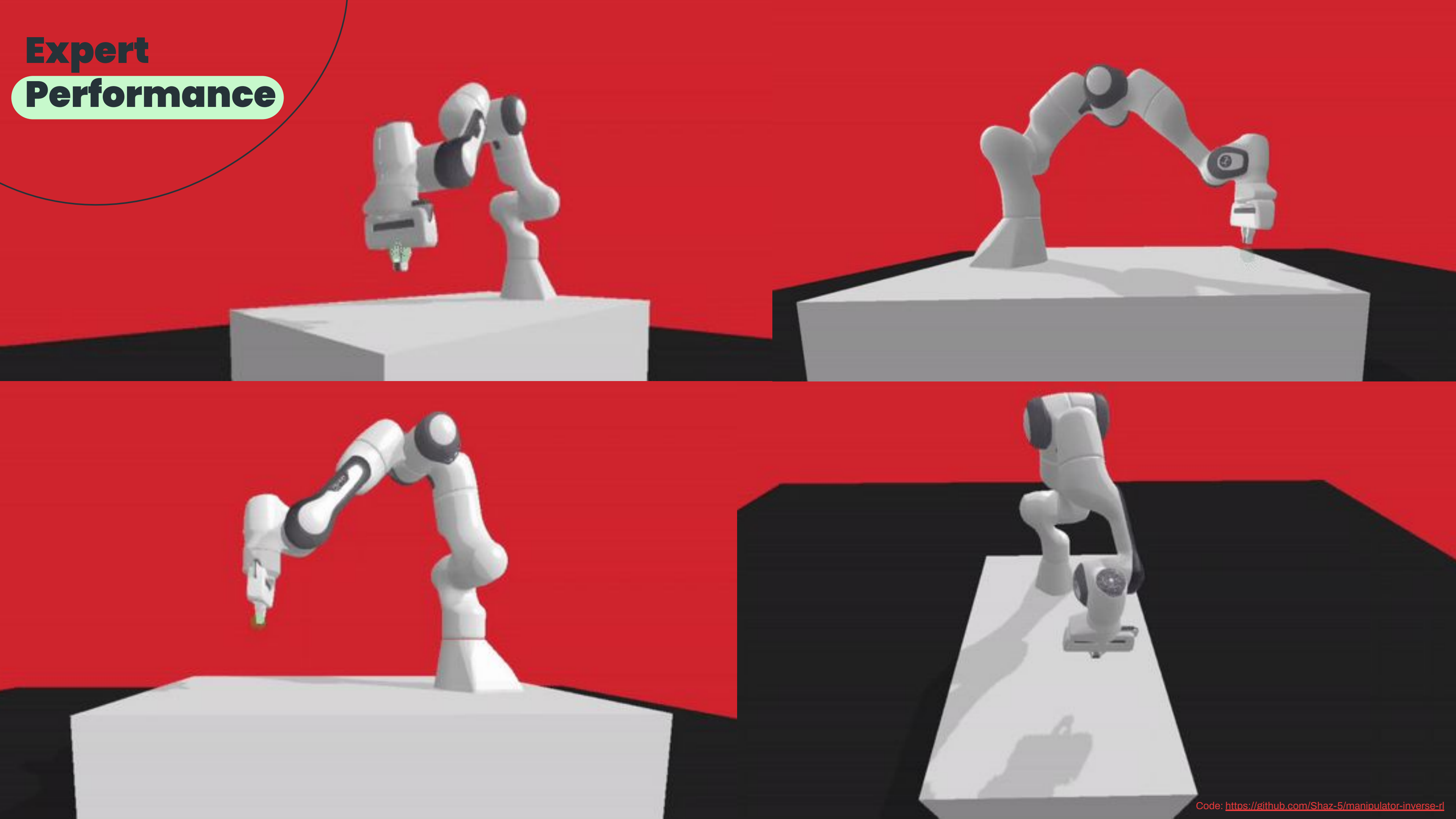


Average Reward (over 1000 episodes) = **-1.768**

Code: https://github.com/Shaz-5/manipulator-inverse-rl

Code: https://github.com/Shaz-5/manipulator-inverse-rl

Expert Performance

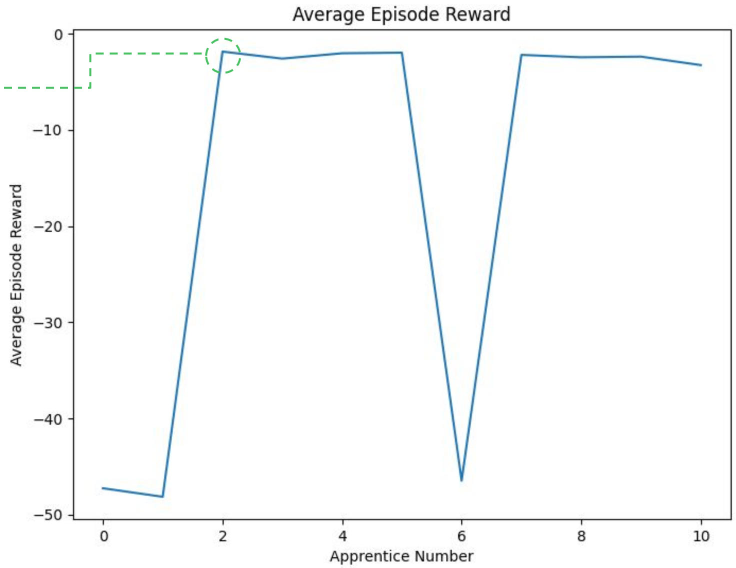


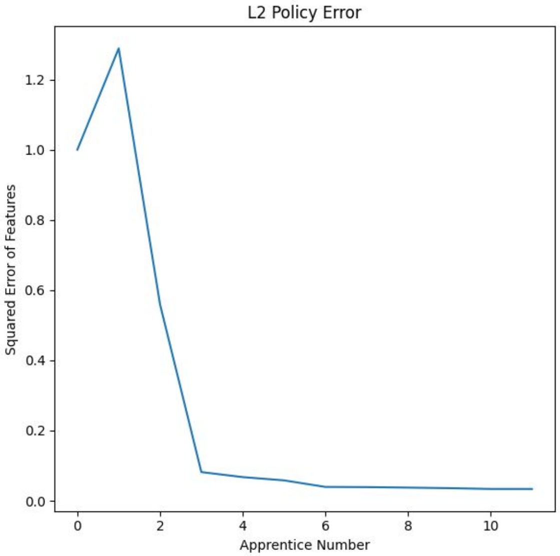
Code: https://github.com/Shaz-5/manipulator-inverse-rl

IRL Projection Algorithm

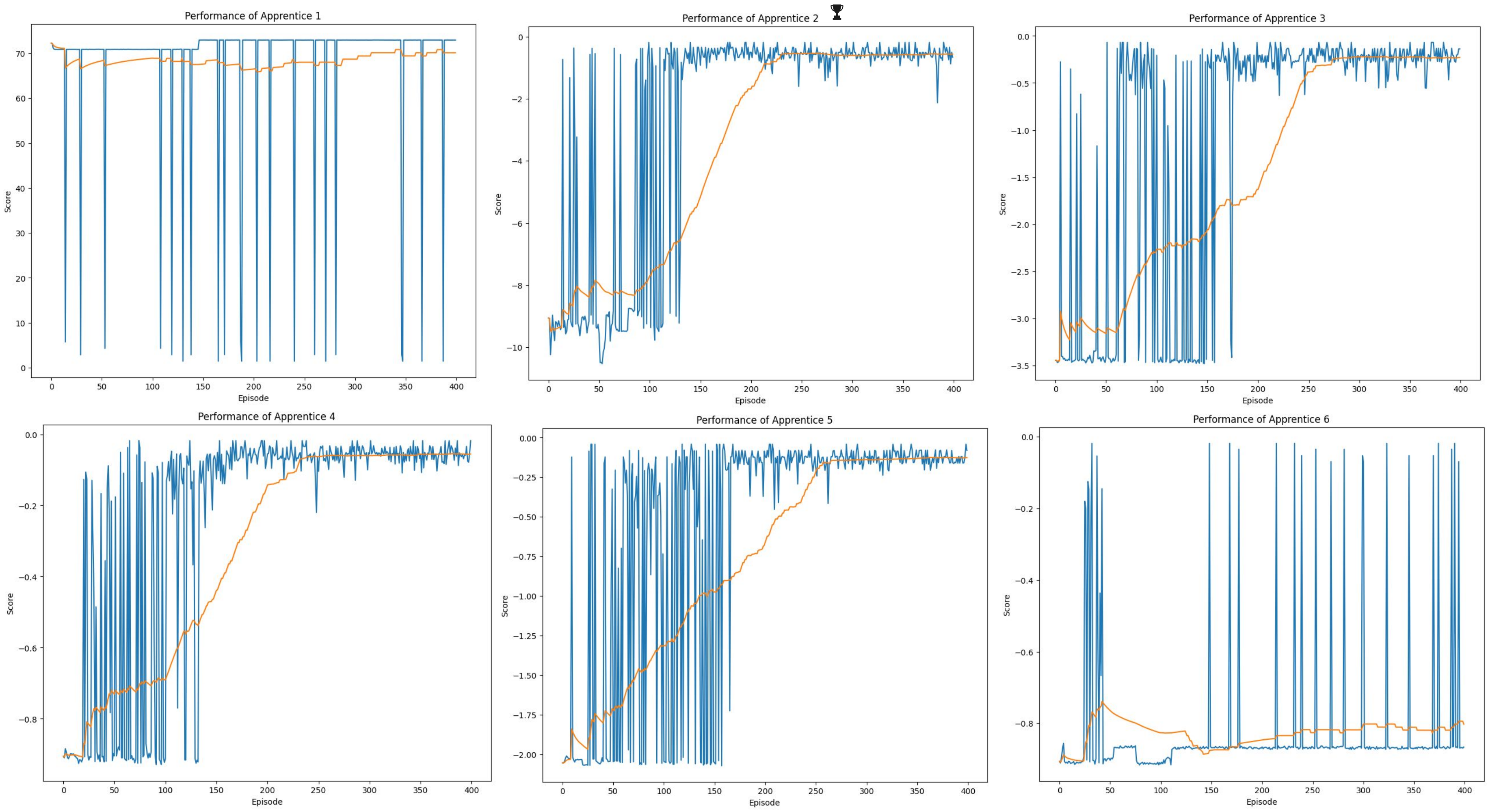
- Ten apprentice agents were trained using the **Projection Algorithm** using DDPG in the **RL step**.
- Feature expectation were calculated over **m = 500** monte carlo trajectories.
- Most agents have learned **optimal policies** albeit through **different inferred reward** functions.

Average Reward (over 500 episodes) = **-1.852**

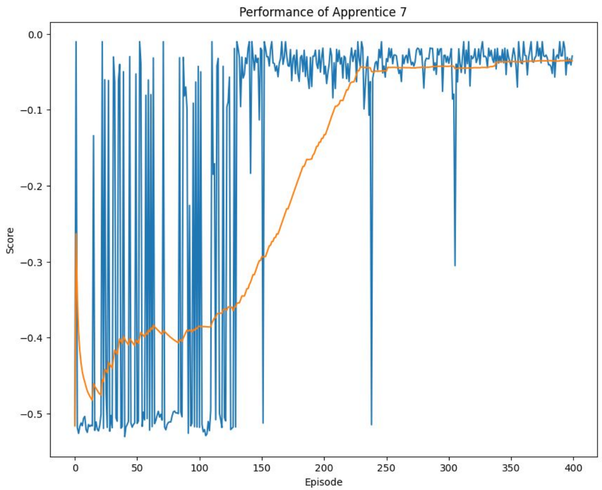


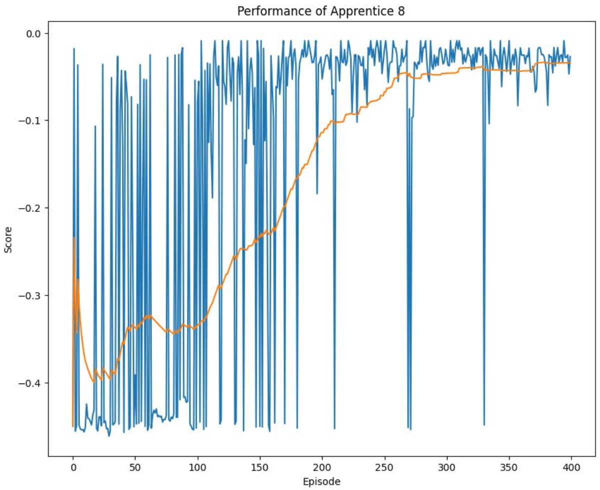


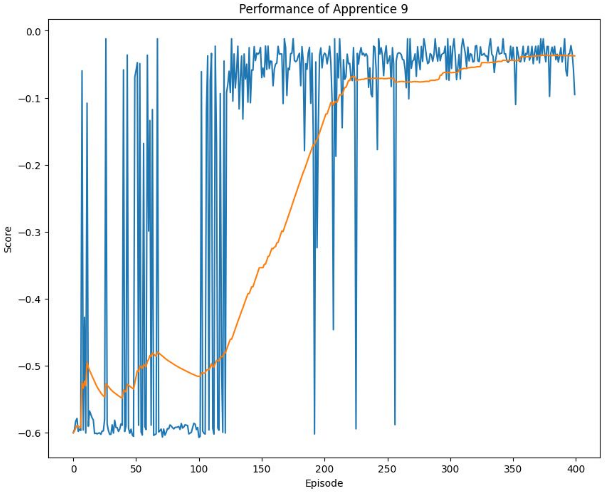
Code: https://github.com/Shaz-5/manipulator-inverse-rl

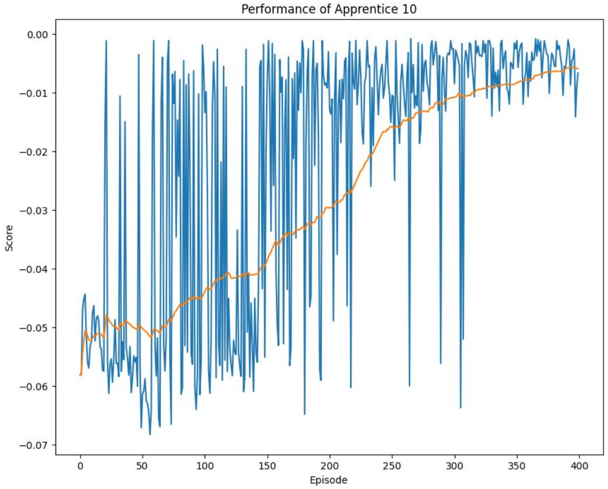


Code: https://github.com/Shaz-5/manipulator-inverse-rl

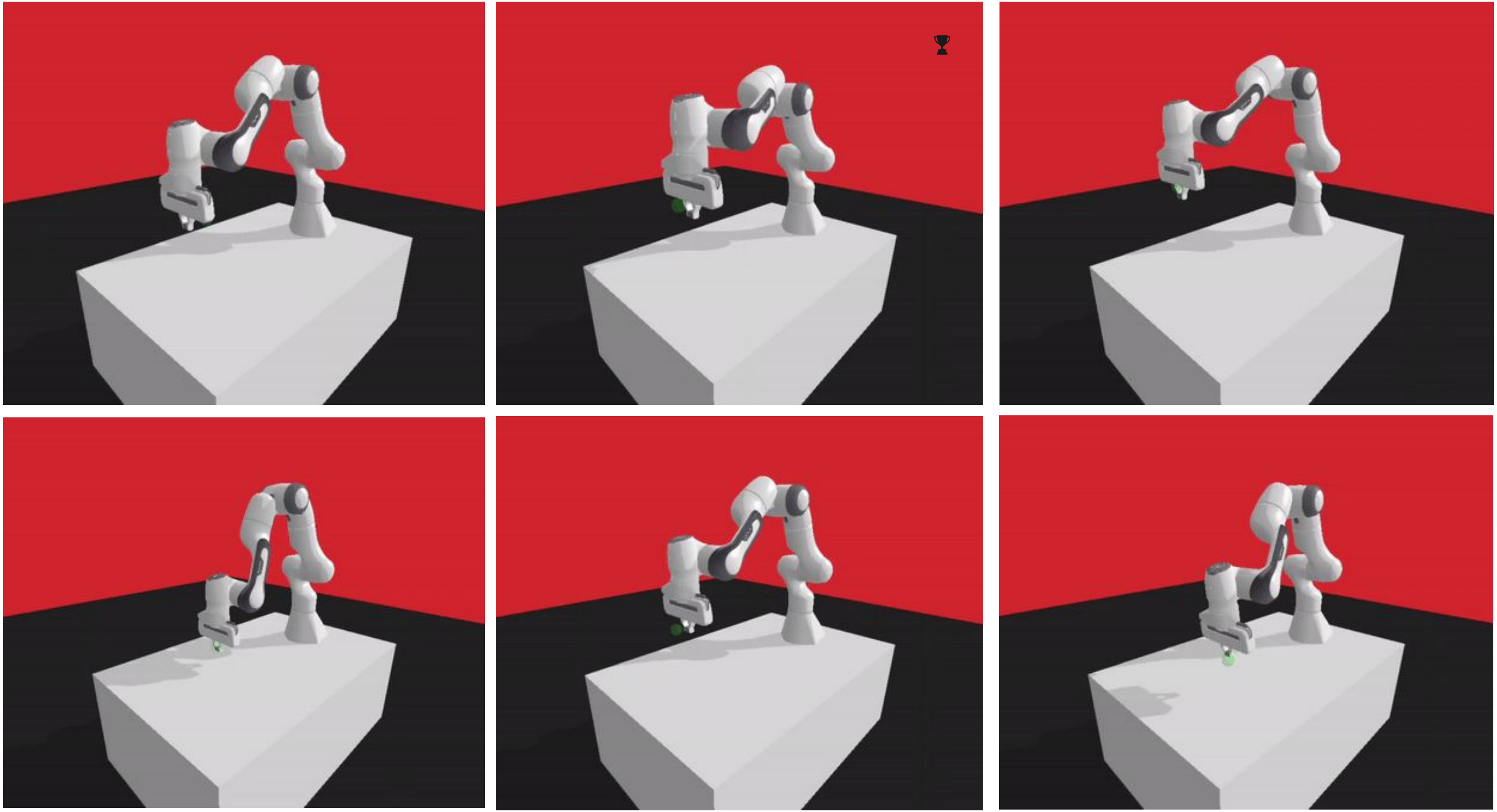




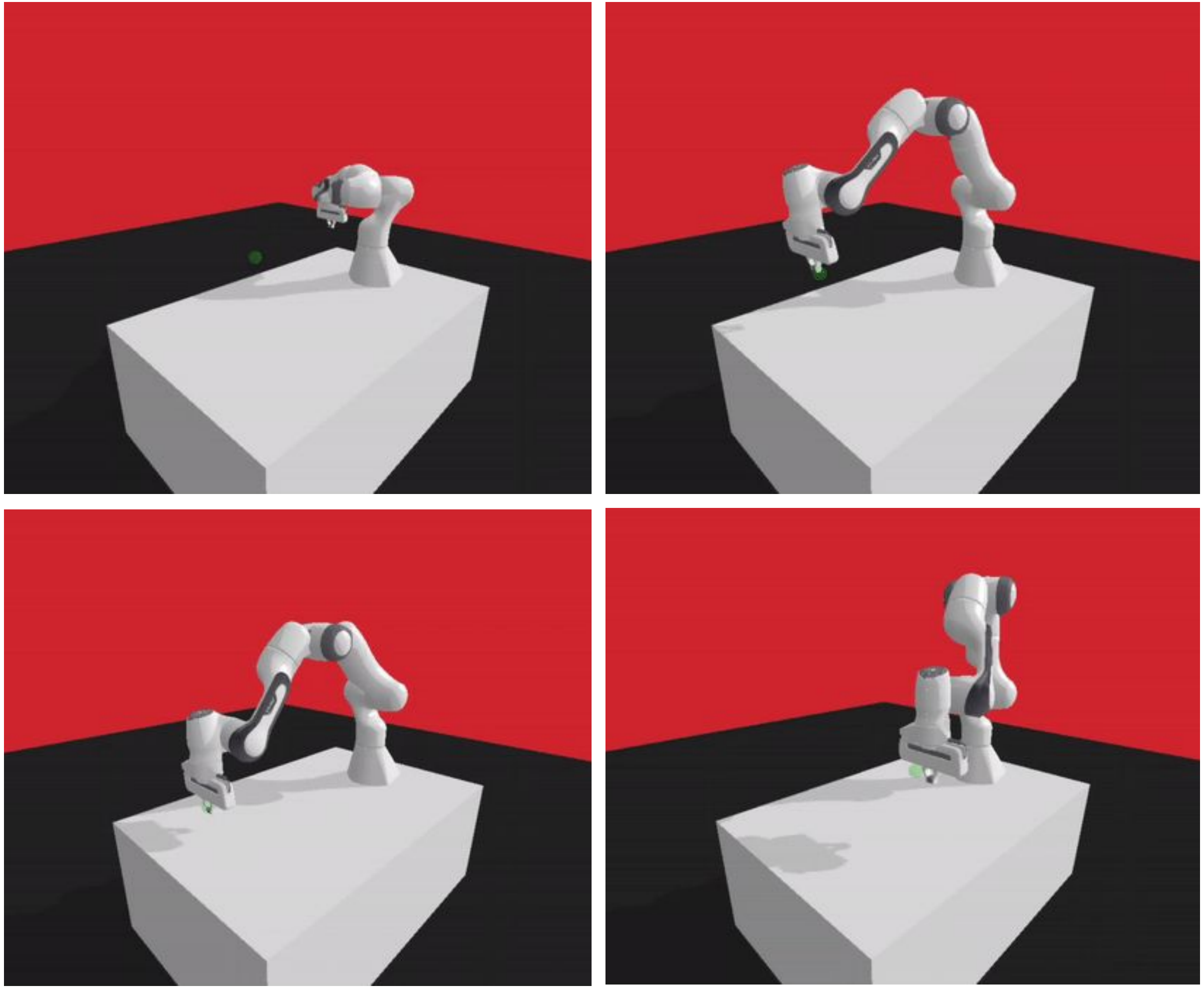




Code: https://github.com/Shaz-5/manipulator-inverse-rl



Code: https://github.com/Shaz-5/manipulator-inverse-rl



Code: https://github.com/Shaz-5/manipulator-inverse-rl

TD3 (with HER)

Experimental Setup

- Environment: **PandaReach-v3**
- Algorithm: Twin Delayed Deep Deterministic Policy Gradient (**TD3**)
- Hyperparameters:
  - alpha=0.001
  - beta=0.002
  - gamma=0.99
  - tau=0.05
  - batch_size=256
  - replay_size=10**6
  - noise_factor=0.1
  - exploration_period=300
  - n_episodes=500

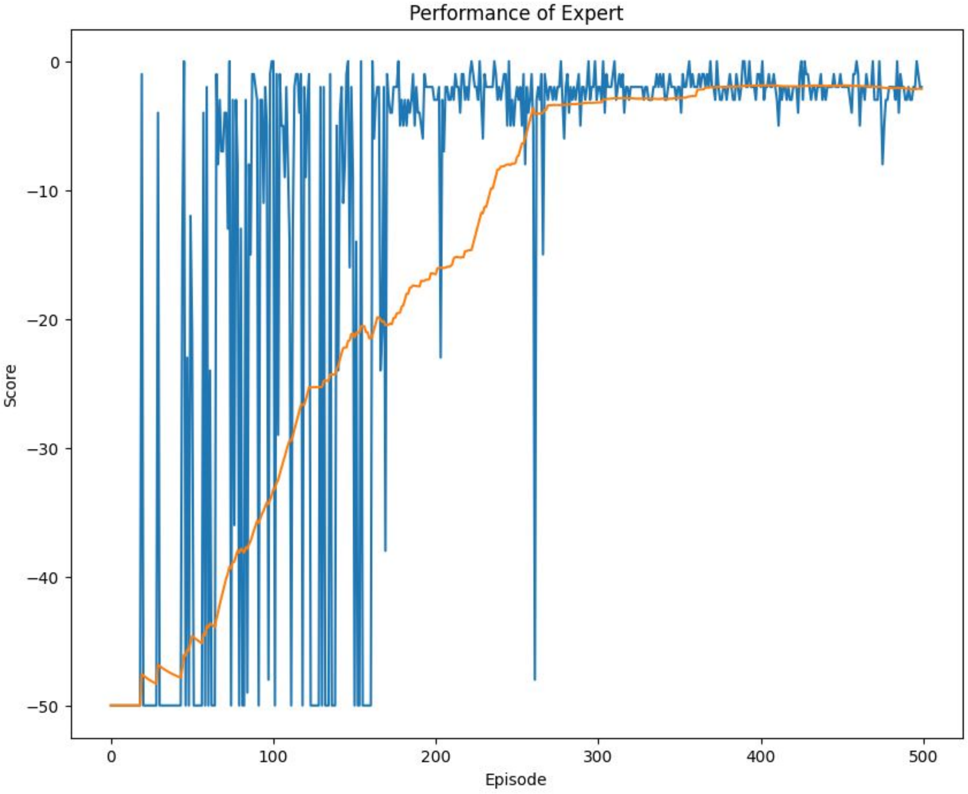
Code: https://github.com/Shaz-5/manipulator-inverse-rl

TD3 (with HER)

```python
# Actor and Critic Networks
class Actor(nn.Module):
    def __init__(self, state_shape, num_actions, name, checkpoints_dir="../Data/"):
        super(Actor, self).__init__()
        if not os.path.exists(checkpoints_dir):
            os.makedirs(checkpoints_dir)
        self.checkpoints_file = os.path.join(checkpoints_dir, name + ".pth")

        self.hidden1 = nn.Linear(in_features=state_shape, out_features=512)
        self.hidden2 = nn.Linear(in_features=512, out_features=256)
        self.hidden3 = nn.Linear(in_features=256, out_features=256)
        self.action_output = nn.Linear(in_features=256, out_features=num_actions)

    def forward(self, state):
        x = torch.relu(self.hidden1(state))
        x = torch.relu(self.hidden2(x))
        x = torch.relu(self.hidden3(x))
        action = torch.tanh(self.action_output(x))
        return action

class Critic(nn.Module):
    def __init__(self, state_action_shape, name, checkpoints_dir="../Data/"):
        super(Critic, self).__init__()
        if not os.path.exists(checkpoints_dir):
            os.makedirs(checkpoints_dir)
        self.checkpoints_file = os.path.join(checkpoints_dir, name + ".pth")

        self.hidden1 = nn.Linear(in_features=state_action_shape, out_features=512)
        self.hidden2 = nn.Linear(in_features=512, out_features=256)
        self.hidden3 = nn.Linear(in_features=256, out_features=256)
        self.q_value = nn.Linear(in_features=256, out_features=1)

    def forward(self, state, action):
        x = torch.cat([state, action], dim=1)
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = torch.relu(self.hidden3(x))
        q_value = self.q_value(x)
        return q_value
```

Code: https://github.com/Shaz-5/manipulator-inverse-rl

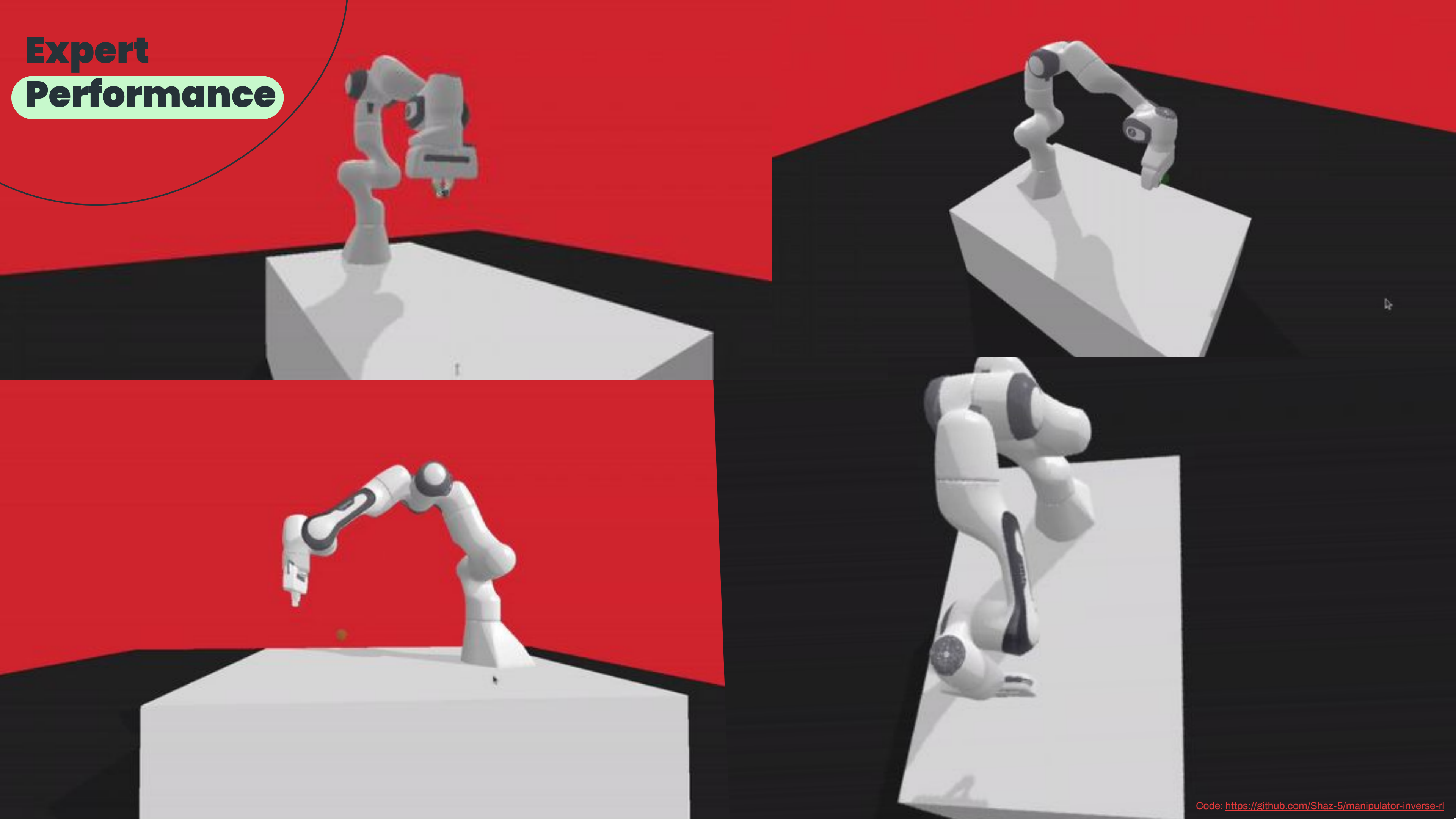
TD3 (with HER)



Average Reward (over 1000 episodes) = **-1.932**

Code: https://github.com/Shaz-5/manipulator-inverse-rl

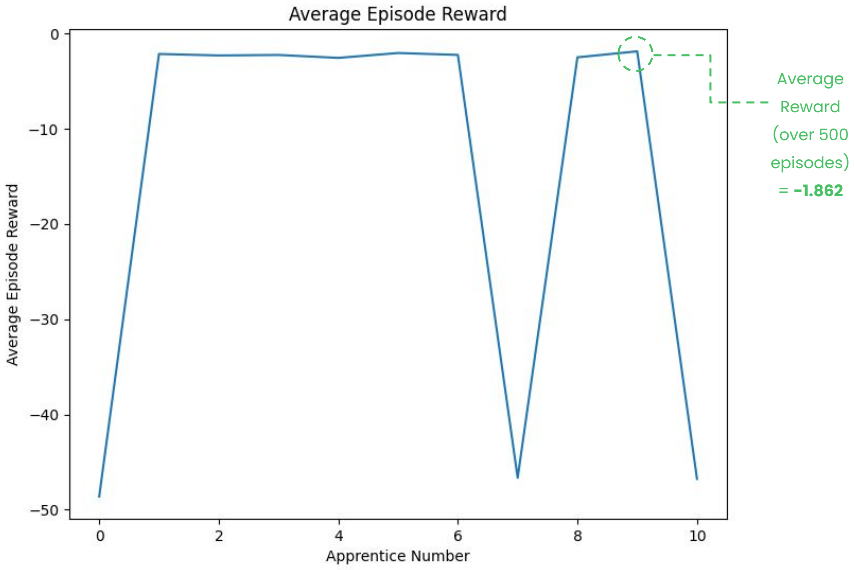
Expert Performance

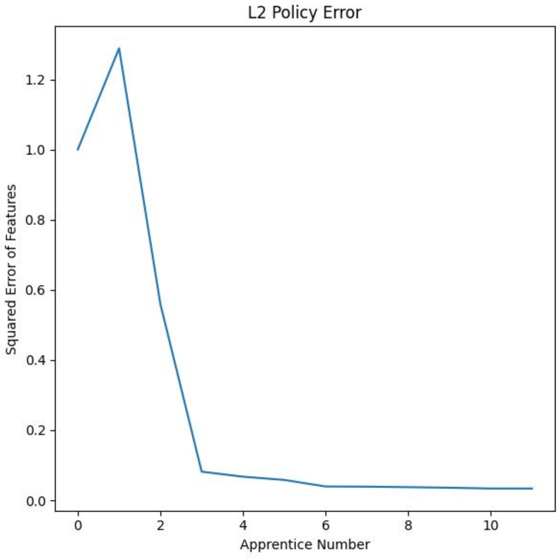


Code: https://github.com/Shaz-5/manipulator-inverse-rl

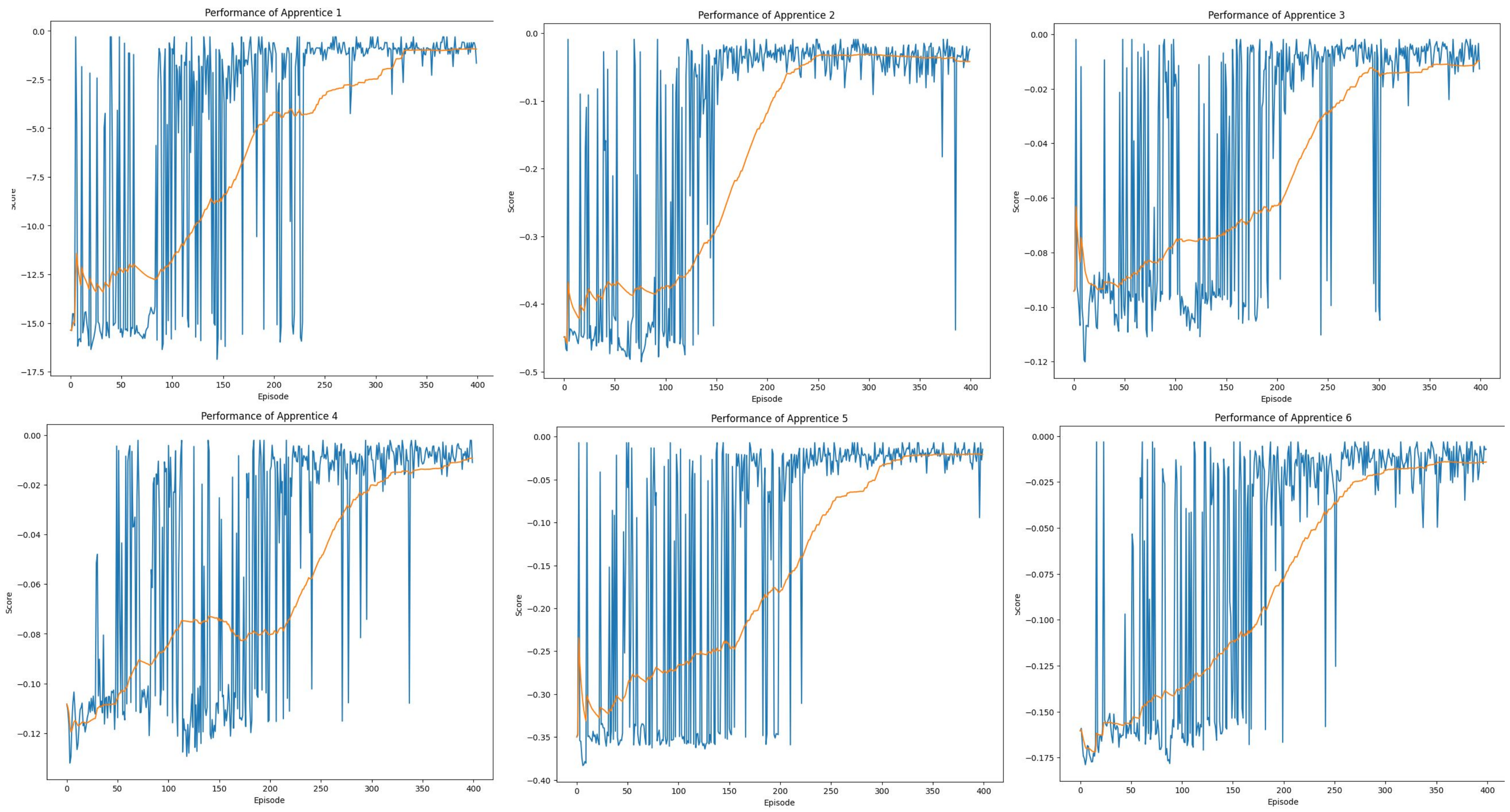
IRL Projection Algorithm

- Ten apprentice agents were trained using the **Projection Algorithm** using DDPG in the **RL step**.
- Feature expectation were calculated over **m = 500** monte carlo trajectories.
- Not only did most agents learn optimal policies, but one even **surpassed** the expert's performance.

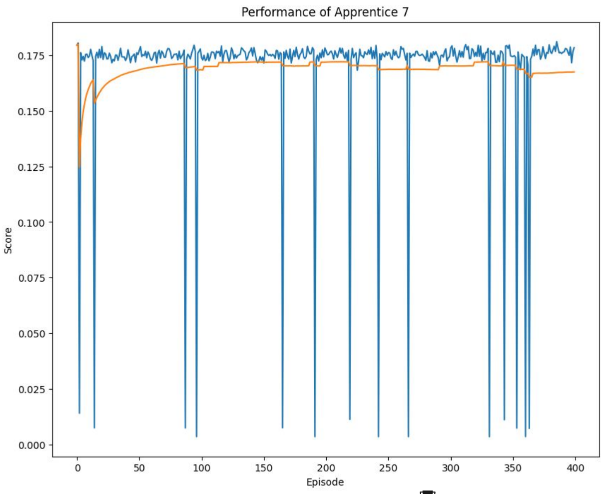


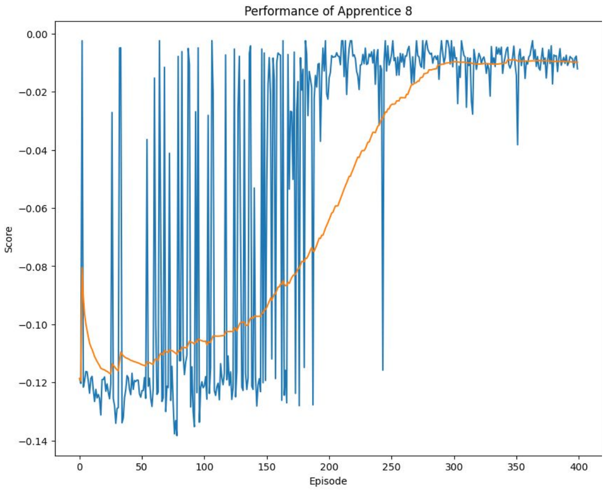


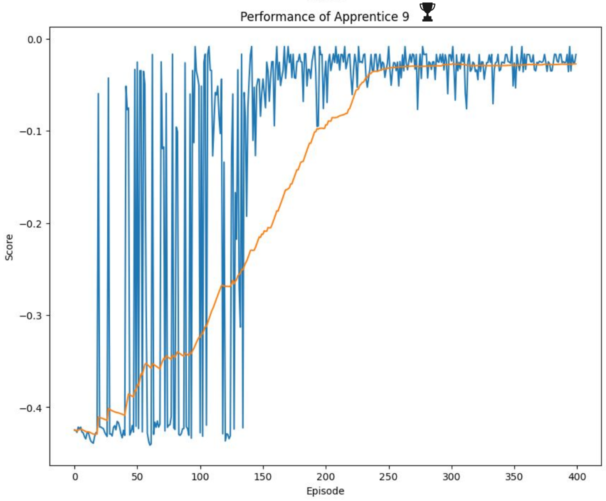
Code: https://github.com/Shaz-5/manipulator-inverse-rl

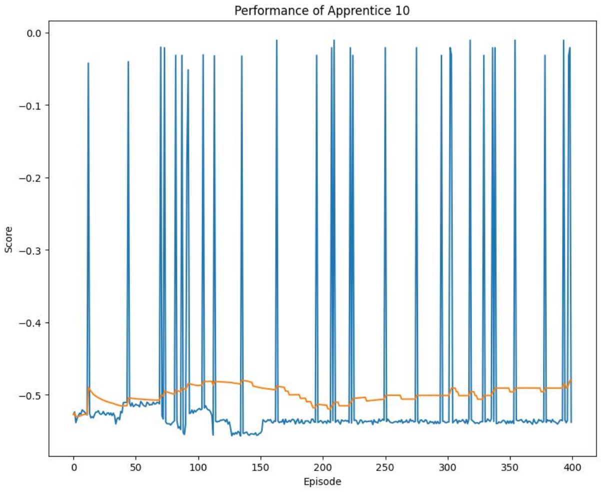


Code: https://github.com/Shaz-5/manipulator-inverse-rl

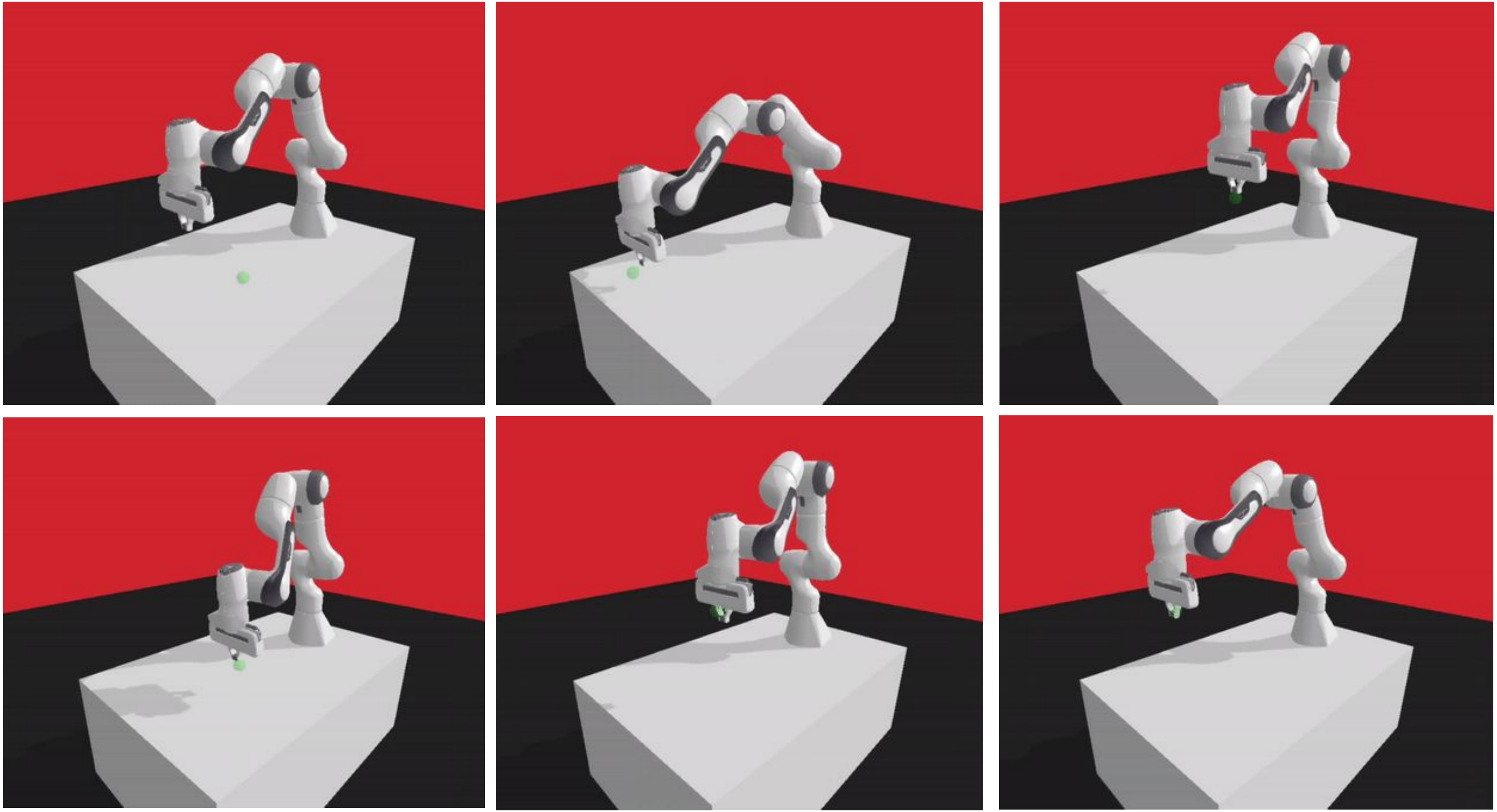




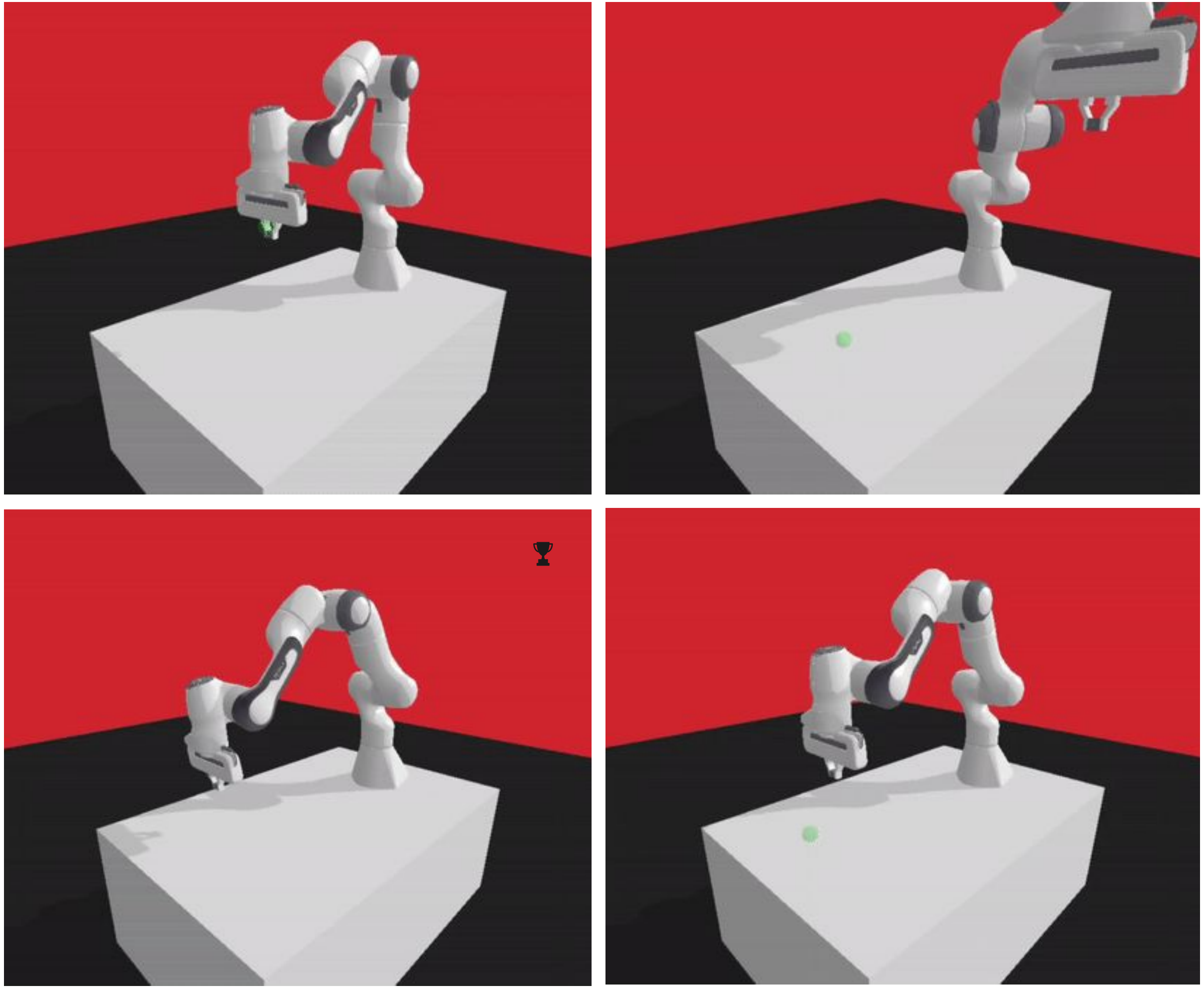




Code: https://github.com/Shaz-5/manipulator-inverse-rl



Code: https://github.com/Shaz-5/manipulator-inverse-rl



Code: https://github.com/Shaz-5/manipulator-inverse-rl

Conclusion

This project successfully investigated Reinforcement Learning (RL) for robot arm control and Inverse Reinforcement Learning (IRL) for apprenticeship learning.

Key Findings:

- RL for Robot Arm Control:
  - DDPG and TD3 effectively learned to control the Panda robot arm in the PandaReach-v3 environment.
- IRL for Apprenticeship Learning:
  - Both DDPG and TD3 were used in the apprenticeship learning IRL framework.
  - The trained apprentice agents achieved performances close to the expert in the PandaReach-v3 task.
  - Notably, one apprentice agent even surpassed the expert's performance.

Conclusion

Implications:

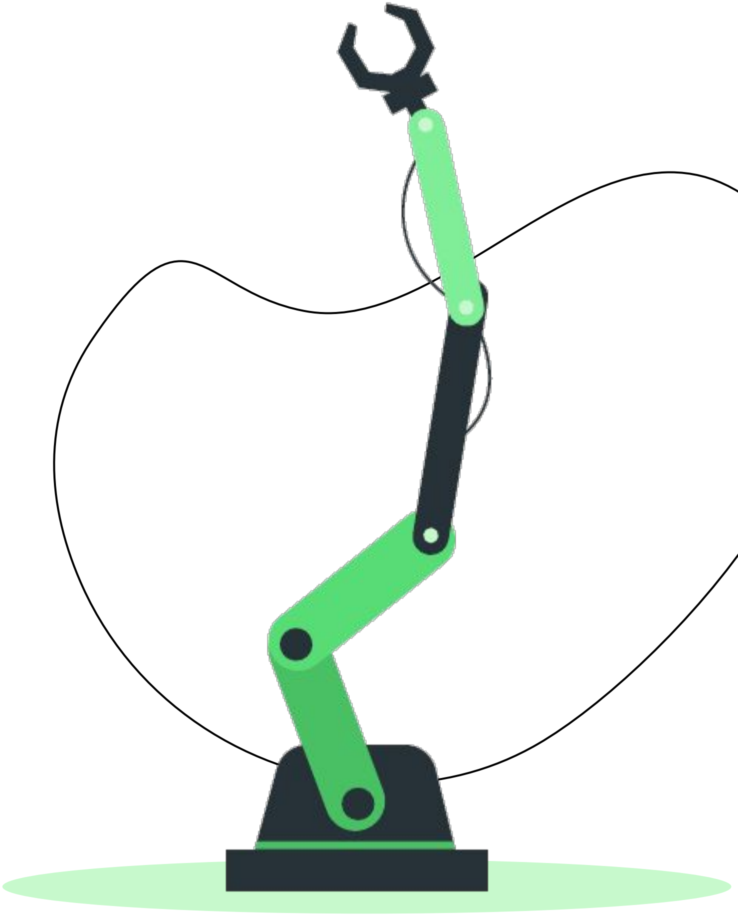
- These results demonstrate the potential of RL and IRL for continuous control tasks.
- IRL with apprenticeship learning offers a promising approach for training robots by leveraging expert demonstrations without explicitly defining reward functions.

Future Work:

- Explore more complex manipulation tasks with panda-gym.
- Experiment with IRL for other control tasks like obstacle avoidance.
- Experiment with transferring learned skills to real robots.

References:

- Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, & Daan Wierstra. (2015). Continuous control with deep reinforcement learning.
- Scott Fujimoto, Herke van Hoof, & David Meger (2018). Addressing Function Approximation Error in Actor-Critic Methods. CoRR, abs/1802.09477.
- Quentin Gallouédec, Nicolas Cazin, Emmanuel Dellandréa, & Liming Chen. (2021). panda-gym: Open-source goal-conditioned environments for robotic learning.
- Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, & Wojciech Zaremba. (2017). Hindsight Experience Replay.
- Abbeel, P. & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning.
- Marekar, A. (2022, June 12). How DDPG (deep deterministic policy gradient) algorithms works in reinforcement learning?.
- DDPG-Algorithm.pdf. (n.d.). http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Spring18/ujjawal/DDPG-Algorithm.pdf
- Twin delayed DDPG (TD3): Theory. (n.d.-a).
- Rivlin, O. (2020, July 28). Reinforcement learning with hindsight experience replay.



Thank You!