Learning Inverse Kinematics using Reinforcement Learning

Sahasrajit Anantharamakrishnan[*1], **Kenechi Franklin Dukor**[*2], **Anuj Shrivatsav Srikanth**[*1]

[1]College of Engineering, [2]Khoury College of Computer Science at Northeastern University - Boston, MA 02115,
{anantharamakrishn.sa, dukor.k, srikanth.anu}@northeastern.edu,

Abstract

In this paper, we present a Reinforcement Learning (RL) approach to the problem of Inverse Kinematics (IK), which involves controlling the end-effector of a 7 Degrees-of-Freedom (DoF) robotic arm to enable it to reach a target position. Our approach uses a policy gradient approach with non-linear function approximators like neural networks. Three major algorithms were investigated in this research (DDPG, SAC and TD3 algorithms). In addition, we see the effect of different reward structures in learning the optimal policy for our chosen domain. We demonstrate the effectiveness of our approach through experiments on a simulated robotic arm, showing that our method is able to learn a policy that can successfully control the arm to reach the target position.

1 Introduction

Robot Arm Control is a fundamental problem in robotics, which involves finding the joint angles that result in a desired end-effector pose. This is a challenging problem, especially for robots with complex kinematic structures, such as redundant or highly articulated robots. Traditional approaches to inverse kinematics often require complex mathematical equations, which can be difficult to derive and implement. In addition, these methods can be sensitive to errors and may not be able to handle certain scenarios, such as when the end-effector pose is outside the robot's workspace or at a singularity.

In recent years, there has been increasing interest in using deep reinforcement learning (RL) for robot control. These methods have the potential to overcome some of the limitations of traditional approaches, by enabling robots to learn inverse kinematics in a more efficient and effective manner which essentially involves allowing robots to learn from trial and error, by receiving feedback on the quality of their actions. This enables robots to learn complex motion tasks in a more efficient and effective manner.

In this paper, we present a method for learning inverse kinematics using reinforcement learning. Our approach uses a deep neural network to learn the mapping from the end-effector pose to joint angles. We evaluate the performance of the proposed method on a simulated robot arm and compare it with stable baselines. The results show that the proposed method is able to learn accurate inverse kinematics, even in challenging scenarios where the end-effector pose is outside the robot's workspace or at a singularity. These results demonstrate the potential of reinforcement learning for solving inverse kinematics in robotics.

One challenge in using RL for robot control is that it requires a large amount of data and computational resources. This can be particularly challenging for articulated robots, which have complex kinematic structures and are therefore difficult to control. In this paper, we present a method for controlling articulated robots using deep RL.

2 Background

A general Reinforcement Learning problem involves the interactions between the agent and the environment in a sequential way to learn the optimal behavior of executing a certain task by maximizing the cumulative reward an agent earns. The agent takes a step (by executing an action) in the environment and incurs a reward for taking that action and based on the rewards it receives, it continues taking actions that will help achieve the goal in an optimal way.

Thus, Reinforcement Learning is a type of learning that maps for each state in the environment the right action to take in order to optimally execute a task.
The key components of any RL problem are:

- *Environment*: the physical world with which the agent interacts and can perceive.
- *Agent*: The object of interest that interacts with the environment.
- *Reward*: Scalar valued feedback given by the environment.
- *Policy*: Mapping between states and actions.
- *Returns*: The cumulative sum of rewards that the agent receives from the environment from the start till the terminal condition is achieved.

From Fig 1, an agent takes an action at a given state in the environment, and based on the action taken at that state, the environment returns a reward and a next state and based on the next observed state, the agent then again takes an action and this loop continues until the termination is reached. To

[*]These authors contributed equally.

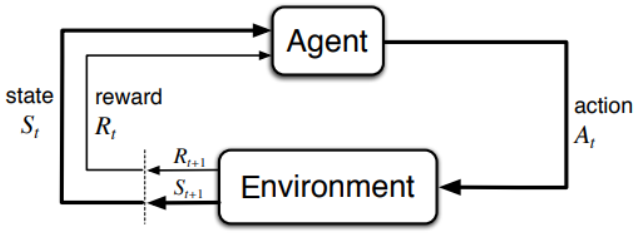

Figure 1: Agent-Environment Interaction in RL.

find the optimal policy we have to visit all the sets of accessible states that the agent can traverse in the environment and all the sets of possible actions that the agent can execute in those states. Thus, we fall into a confusion between which actions to choose and about the dilemma of exploration vs. exploitation, that is whether to take the action that gave the best reward at a particular state or explore all the set of actions in that state that maximizes the return rather than the immediate reward.

2.1 Markov Decision Process (MDP)

An MDP is a mathematical framework for modeling sequential decision-making in an environment with the goal of attaining a policy that maximizes some reward signal (Sutton and Barto 2018). The policy is defined by π: a mapping from states to a probability distribution over the actions $\pi : S \rightarrow P(A)$. This is exactly what the RL problem is also trying to achieve and hence in order to address the RL problem we have to first solve a Markov Decision Process (MDP). We model a Markov decision process with a state space S, action space $A = R^N$, an initial state distribution $P(S_1)$, transition dynamics $P(S_{t+1}|S_t, A_t)$, and reward function $R(S_t, A_t)$ and describe it using the four-tuple (S, A, T, R) configuration, where S is the set of states the agent may encounter, A is the set of actions available to the agent, T is the transition function T$(S'|S, A)$ describing the probability of entering a new state S' given that the agent is in state S and takes action A, and R is the reward function $R(S, A)$ mapping state-action pairs to rewards. Using the MDP framework, we can learn an estimation of an important parameter called value function $v_*(S)$ or $q_*(S, A)$, that is the expectation over the returns or the cumulative rewards obtained by the agent under the optimal policy. These estimates then help us select the best set of actions by greedily choosing actions that have the highest value.

2.2 Policy Gradient Methods

Policy gradient methods rely upon parameterizing the policy using a set of parameters and then performing gradient ascent to learn the parameters. This method helps tackle problems faced by traditional RL approaches like the lack of guarantees of a value function and the continuous state and action space. Policy gradient methods directly attempt to learn the policy instead of learning value functions and hence this helps directly find the optimal policy rather than checking for the value function and then finding a mapping between states and actions. In these methods, we train a non-linear network to learn the preferences for all the actions that can be taken from a given state and then based on the preferences returned by the network we select an action using a function like softmax. An extension of these methods is the Actor-Critic method, where a separate network called Critic utilizes the value function as a form of evaluation to evaluate the parameterized policy that the Actor-network learns.

2.3 Experience Replay

In order to train our networks we need to sample transition tuples from the MDP and use them to calculate the TD error and perform Stochastic Gradient Descent (SGD) to update our function parameters. The transitions that we observe are highly correlated with each other but for SGD the training data should be Independent and Identically Distributed (I.I.D). This violation can be resolved by storing the tuples in a replay buffer and during training randomly sampling tuples from this buffer ensuring independence between the tuples. In essence, we are generating our labeled dataset via experience and shuffling that data in how we sample from it.

3 Related work

Inverse Kinematics is a prominent area that finds application in robotic research. In times past the field has experienced various transitions in methods that work towards optimally moving the joints of a robot to reach the desired target. Most of these methods involved complex mathematical concepts that required lots of computational power.

A well-known method used to solve inverse kinematics problems is the **Jacobian Inverse IK Method** (Merlet 2000). This is an old and powerful method widely used in robotic research, however, the computational requirement needed to successfully use this method is expensive. The method used an iterative approach to find the joint configurations (T), compute the change in rotation (dO), and lastly, compute the Jacobian (J)

$$T = O + dO \cdot h \tag{1}$$

Where h is just a simulation step that can be tuned.

Another such method is **Forward and Backward reaching Inverse Kinematics (FABRIK)** (Aristidou and Lasenby 2011). The method locates the individual joint position through a point locating point on a line. It does so in fewer iterations and with lesser computational power when compared to the Jacobian method (Aristidou, Chrysanthou, and Lasenby 2016).

Research continues in this area and improvements are constantly being tested on different domain applications in robotics. One recent improvement and extension to the FABRIK algorithm is **Continuum Robot Reaching Inverse Kinematics (CRRIK)** (Wu et al. 2022) which was inspired by the physical process of pulling a rope with a fixed end, which is straightforward and obvious, avoiding complicated nonlinear operations. The CRRIK algorithm promises a high convergence rate and low computational cost, which are suitable for real-time applications.

One noticeable difference between the discussed approach to solving inverse kinematic problems in robotics is

the fact that computing inverse kinematics under conditions of stability and self-collision avoidance cannot be done efficiently in real-time.

In recent times as reinforcement learning began to gain relevance in research, and methods have been proposed to solve this problem of inverse kinematic control. The reinforcement learning methods promise better learning for more complicated tasks with infinite observation space(Phaniteja et al. 2018).

4 Project description

We describe the problem of solving the inverse kinematics equation for a robot arm by controlling its position and making the robot arm move to any desired location in a cube of size $0.3 \times 0.3 \times 0.3$m. The robot arm should reach the desired position that is generated randomly at the start of each episode with a tolerance of ± 0.05m

4.1 MDP Formulation

- S - observation space that consists of the position of the arm in the environment along with its speed and the desired goal.

$$S = [x \quad y \quad z \quad v_x \quad v_y \quad v_z \quad x_g \quad y_g \quad z_g]$$

  The first six variables represent the position and velocity of the robot and the last three variables represent the goal position.

- A - set of control actions that control the position of the arm in three coordinates, one for each axis of movement x, y, and z.

$$A = (x, y, z) \; \forall \; x, y, z \in [-1, 1] \; cm.$$

- R - we have performed simple reward engineering and set a dense reward that calculates the negative of the distance between the present robot arm position and the desired position and assigns it to the reward function, and a reward of $+1$ on termination that is reaching within 0.05m within the target location.

$$R = \begin{cases} -\|X_{obs} - X_{goal}\|, & \text{if: } \|X_{obs} - X_{goal}\| \geq \delta \\ 0, & \text{else: } \|X_{obs} - X_{goal}\| \leq \delta \end{cases}$$

  Here, $X_{obs} = (x, y, z)$, and $X_{goal} = (x_g, y_g, z_g)$.

- T - The transition function is deterministic and the next state reached depends on the action taken.

We can't directly apply Q-learning to continuous action spaces because in continuous action spaces finding the greedy policy requires optimization at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action space (Lillicrap et al. 2015).

We thus have explored and implemented various actor-critic methods that work well on continuous action spaces and analyzed the performance of these algorithms and presented them in the results section 5.

4.2 Deep Deterministic Policy Gradients (DDPG)

DDPG is a model-free off-policy algorithm for learning continuous actions using the methods from Deep Q-Networks and Deep Policy Gradients that learn a Q-function and a policy to iterate over actions. It employs the use of off-policy data and the Bellman equation to learn the Q function which is in turn used to derive and learn the policy. It uses an Experience Replay (buffer) and slow-learning target networks from DQN.

The experience replay is used to train the network based on a sample of observations rather than just a single observation and as seen in the background section we need to randomly sample to ensure the I.I.D condition for gradient descent. Apart from an experience replay we use target networks and perform a soft update on these.

Soft Update Equations:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1 - \tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1 - \tau)\theta^{\mu'}$$

The purpose of target networks is to encourage off-policy learning and to update the targets toward the correct estimates. We periodically update these networks using the weights from the behavior actor and critic using a hyperparameter τ without directly copying the weights and instead synchronize them using the τ. We thus integrate the use of replay buffers and a target Actor-Critic network to stabilize learning, forming an off-policy variation of the Actor-Critic. Here, we have used the Actor network to directly output the value of the action rather than a preference for the actions. DDPG outputs a deterministic policy, that is if a particular state is passed into the Actor network over and over again it outputs the same action value. To address this problem of exploration vs exploitation we use a noise model called Ornstein Uhlenbeck noise and sample from this model to add exploration to our problem.

The input to our Actor-network is the current observation of the robot concatenated with the desired goal that is returned from the observation space of the simulator hence the state space is 9 dimensional, while its output is the action itself and thus the actor returns 3 values in the range from -1 to 1 corresponding to control in each axis. The input to the critic model is a concatenation of the state observation and the action that the actor model chooses based on the state, while its output gives the Q value for each action and state. The Critic loss is given by the minimum mean squared error between the targets and the Q-values and we try to minimize this error.

$$L = \frac{1}{N} \sum_{i} (y_i - Q(s_i, a_i|\theta^Q))^2$$

Actor loss is computed using the mean of the value given by the Critic network for the actions taken by the Actor-network. We seek to maximize this quantity and hence perform a gradient ascent to update the weights of the Actor-network as shown in equation 2.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i} \nabla_a (s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_s{}^i \tag{2}$$

4.3 Soft Actor-Critic (SAC)

Soft Actor-Critic, is an off-policy actor-critic algorithm that is based on a maximum entropy reinforcement learning framework. In the maximum entropy framework, the actor aims to maximize the expected long-term reward while also maximizing long-term entropy. In essence, for a task to be completed successfully, it should act as randomly as possible (Haarnoja et al. 2018).

$$J(\pi_\theta) = \sum_{t=0}^{T} \mathbb{E}_{\pi_\theta}[R(s_t, a_t)] + \alpha H(\pi(\cdot|s_t))]$$

From the equation above, we see that the standard maximum reward objective is augmented with an entropy maximization.

Together with entropy, SAC combines off-policy updates with a stable stochastic actor-critic formulation. We know that the actor-critic is a combination of policy-based and value-based approaches to the high variance experience when policy-based methods are used alone.

In the SAC algorithm, there are three networks. The state-value network (V) is parameterized by ψ; the policy function that parameterized by ϕ; and the soft Q function is parameterized by θ.

The state value function approximates the soft value, and the soft value function is trained to minimize the squared residual error.

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathfrak{D}} \Bigg[\frac{1}{2} \left(V_\psi(s_t) \; - \mathbb{E}_{a_t \sim \pi_\phi} \left[Q_\theta(s_t, a_t) \right. \right. \\ \left. \left. - \log \pi_\psi(a_t|s_t) \right] \right)^2 \Bigg]$$

The soft Q-function parameters are trained to minimize the soft Bellman residual (given below) and further optimized using stochastic gradient descent.

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathfrak{D}} \left[\frac{1}{2} \left(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t) \right)^2 \right]$$

And lastly, the policy parameters are learned by minimizing the expected KL divergence.

$$J_\pi(\phi) = \mathbb{E}_{(s_t) \sim \mathfrak{D}} \left[D_{KL} \left(\pi_\phi(\cdot|s_t) || \frac{e^{(Q_\theta(s_t, \cdot))}}{Z_\theta(s_t)} \right) \right]$$

SAC has proven to be an efficient algorithm for real-world robotic problems and one of the advantages presented by SAC is its ability to explore widely in an incentivized way strategically. The algorithm also captures multiple modes of near-optimal behavior. Also, the algorithm has an impressive learning speed over state-of-art methods that optimize the conventional RL objective function.

4.4 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Twin Delayed Deep Deterministic Policy Gradients (TD3) is an algorithm for learning control policies in reinforcement learning (RL) tasks. It is the successor to the Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. 2015). It is an actor-critic method that uses deep neural networks to learn policies and value functions. It is different from DDPG in these key features:

1. *Two (Twin) Critic Networks*: TD3 is similar to double Q-learning, in using two separate target value functions to estimate Q-value. This reduces bias, reduces overfitting, and improves stability in the learning process. Moreover, it takes a minimum over the Q-values generated from the two critic networks, this creates an underestimation of the Q-values which does not affect learning as much as compared to overestimation of the same.

$$y_1 = r + \gamma Q_{\theta_2'}(s', \pi_{\phi_1}(s'))$$
$$y_2 = r + \gamma Q_{\theta_1'}(s', \pi_{\phi_2}(s'))$$

2. *Delayed updates of the Actor Network*: In an Actor-critic method, when overestimation happens for a sub-par policy it snowballs and creates a terrible policy. This can be shifting the update phase of the two algorithms i.e. delaying the update of the actor policy as compared to the critic network.

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

3. *Noise Addition*: Small amount of clipped noise is added to the actor network which reduces the variance in the final policy and helps in exploitation. The equation (Fujimoto, van Hoof, and Meger 2018) below shows the noise added to the action and subsequent clipping of both noise and action.

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right),$$

Where, $\epsilon \sim \mathcal{N}(0, \sigma)$

5 Experiments[1]

Experiments were performed on a robot in navigation domains with continuous action spaces and coordinate state observations. We focus on exploring the performances of three major reinforcement learning algorithms and evaluate their performances against those recorded in (Gallouédec et al. 2021).

5.1 Domain

We have briefly described the domain we explored. The panda-gym environment, which uses the PyBullet physics engine, was utilized. Panda-gym, an open-source library integrated with OpenAI-gym, allows smooth experimentation of reinforcement learning algorithms.

The proposed environments also allow fast learning on computers with limited computing capacity. The PyBullet physics engine allows the parallel simulation of several scenes. Thus, the environments are compatible with learning methods that use multiple CPU cores. Tests show that the environments are, on average 9.2% faster than their equivalents developed on MuJoCo (Gallouédec et al. 2021).

5.2 Baselines

We compare the results of our implementation of SAC, DDGP, and TD3 against the results obtained in the panda-gym paper (Gallouédec et al. 2021). In this paper, we will be considering the experiment on the PandaFetch domain, where SAC, DDPG, and TD3 algorithms were also explored.

[1] GitHub: https://github.com/Sahas-Ananth/RL-FinalProject

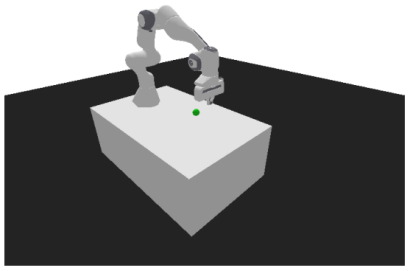

Figure 2: Panda-gym Enviroment for PandaReach

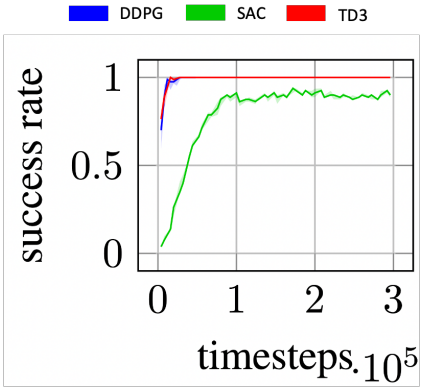

Figure 3: Extract from (Gallouédec et al. 2021) for PandaFetch domain with HER

5.3 Experimentation

Metrics Depending on the implementation, we record the success rate and/or the average scores obtained for 70,000 timesteps and compare the performance of the three algorithms (SAC, DDPG, TD3) in the same domain for both sparse and dense reward structures.

The report shows results for our implemented algorithm and stable-baseline implementation for the vanilla replay buffer and hindsight replay buffer. The plots for our result were averaged over 100 episodes.

Result - Sparse Rewards The agents' performance on stable baseline implementation of the algorithm is shown in Fig 4 for the hyperparameter recorded in Table 1. We observed a significant positive success rate of 98% to 100% for the three algorithms. However, we see a smoother learning curve of success rate and mean reward in TD3 and SAC when compared with DDPG. This is suspected to be a result of the better exploration features of SAC and TD3 that allows the model to find an optimal path faster. However, when this result is compared to the result from our baseline (Gallouédec et al. 2021) we see differences in the behavior of the SAC algorithm. In contrast to our result, the SAC algorithm (according to the paper) did not perform as well as TD3 and DDPG for the same hyperparameter setup. While it is still unclear why there is a difference in the results, we suspect the issue may emanate from domain versions and updates.

Gamma	0.95
Tau	0.005
Learning rate	1e-3
Batch size	2048
Total training timestep	70,000
Buffer size	100,000
Network Arch	3 hidden layer (512), 2 critic

Table 1: Hyperparameters for Stable-baselines with HER and Sparse Rewards.

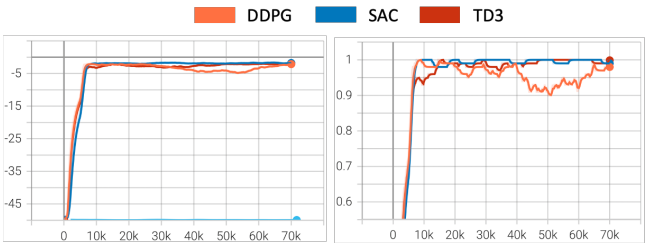

Figure 4: Results from Stable-baselines with HER and Sparse Rewards. **Left:** Mean Rewards **Right:** Success Rate

Metrics	**DDPG**	**SAC**	**TD3**
Actor loss	0.164	0.101	0.373
Critic Loss	0.00725	0.00476	0.0748
Mean Reward	-2.09	-1.74	-1.83
Success Rate	98%	99%	100%

Table 2: Results for Stable-baselines with HER and Sparse Rewards.

We went further to try applying our implementation of the algorithm (which does not have Hindsight Experience Replay) on the same sparse reward domain and observed an expected result where the agent could not learn. This agent cannot path efficiently to the goal since it receives a reward only when the goal is reached.

This result validates the usefulness of HER buffers in sparse reward domains. Other methods, such as reward shaping, could have been considered; however, there is a domain knowledge requirement constraint.

Result - Dense Rewards The agents' performance on stable baseline implementation of the algorithm is shown in Figure 4 for the same hyperparameter recorded in table 1.

We observed an excellent success rate of 100% for the three algorithms. We also see that a better actor and critic loss was achieved compared to the sparse reward domain. This is a result of the benefits of the dense reward structure and the application of HER, which further improves performance in learning.

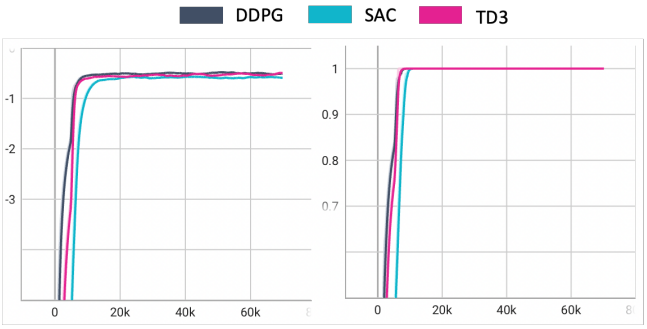

Figure 5: Results from Stable-baselines with HER and Dense Rewards. **left:** Mean Rewards **right:** Success Rate

Metrics	**DDPG**	**SAC**	**TD3**
Actor loss	0.00861	0.182	0.213
Critic Loss	3.17e-06	6.51e-06	0.000113
Mean Reward	-0.51	-0.593	-0.496
Success Rate	100%	100%	100%

Table 3: Results for Stable-baselines with HER and Dense Rewards.

However, when our implementation of the algorithms (without HER) was applied to the same domain with dense rewards, a poor result was recorded, as you will see in Figure 6. This result was obtained after different permutations of hyperparameters, which led to the final parameters we have recorded in Table 4.

Gamma	0.99
Tau	0.005
Learning rate	1e-3
Batch size	100
Episodes	2000
Buffer size	128
Network Arch	2 hidden layers (256), 2 critics

Table 4: Hyperparameters for our implementation of the algorithms without HER (Dense rewards).

To confirm that this behaviour is valid. We trained another agent on the same domain with stable baseline implementation of the three algorithms, but without HER buffer and we experienced a similarly poor result for the three algorithms.

6 Conclusion

This paper has successfully investigated the performance of policy gradient algorithms (DDPG, SAC, TD3) on inverse kinematics problems. More specifically, we have established the usefulness of hindsight experience replay in learning the optimal policy in the investigated domain. This is a significant add-on to the different algorithms since they all produced better results when HER was applied. We have also seen the limitations of the DDPG algorithm with respect

Figure 6: Average score result from our implementation of the algorithms without HER (dense rewards domain)

to sufficient and efficient exploration and the improvements seen in the SAC and TD3 algorithms. While there were recorded successes in the PandaReach domain, difficulties were still experienced in learning the optimal policy in more complex domains such as PandaPush and PandaPickandPlace. Such domains still require complex hyperparameter tuning and computing power.

References

Aristidou, A.; Chrysanthou, Y.; and Lasenby, J. 2016. Extending FABRIK with model constraints. *Computer Animation and Virtual Worlds*, 27: 35–57.

Aristidou, A.; and Lasenby, J. 2011. FABRIK: A fast, iterative solver for the Inverse Kinematics problem. *Graphical Models*, 73(5): 243–260.

Fujimoto, S.; van Hoof, H.; and Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods.

Gallouédec, Q.; Cazin, N.; Dellandréa, E.; and Chen, L. 2021. panda-gym: Open-source goal-conditioned environments for robotic learning.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning.

Merlet, J.-P. 2000. Jacobian and inverse kinematics. 65–89.

Phaniteja, S.; Dewangan, P.; Guhan, P.; Sarkar, A.; and Krishna, K. M. 2018. A Deep Reinforcement Learning Approach for Dynamically Stable Inverse Kinematics of Humanoid Robots.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Wu, H.; Yu, J.; Pan, J.; Li, G.; and Pei, X. 2022. CRRIK: A Fast Heuristic Algorithm for the Inverse Kinematics of Continuum Robot. *Journal of Intelligent &amp Robotic Systems*, 105(3).