The Application of AI Algorithms in Robotic Arms

Yixiao Chen, Ruopeng Huang, Katharine Huang, Qianlai Yang

Northeastern University, Boston, MA 02115

December 12, 2023

Abstract

This paper explores the application of Artificial Intelligence (AI) algorithms, specifically the Deep Deterministic Policy Gradient (DDPG) [1] algorithm, in controlling robotic arms for target-grabbing simulations. We construct a custom simulation environment to mimic real-world scenarios for robotic arms, crucial for training and testing AI algorithms. Our approach involves developing a specialized RL model with unique hyper parameters and neural network architectures. The actor network determines arm actions, and the critic network evaluates these actions. The training process includes reaching target positions with increasing complexity, demonstrating the model's learning and adaptation. Results show reduced steps and increased rewards, indicating successful learning. The final model exhibits the robotic arm's capability to reach arbitrary positions in a simulated environment, a significant step toward real-world applications. Future prospects involve transitioning to a 3D simulation environment, refining the reward system, and enhancing feature engineering. This research contributes to robotics by showcasing AI algorithms' potential in improving robotic arm functionality and efficiency.

$https://github.com/Bobchenyx/CS5100_Group11_Project$

Keywords— Reinforcement Learning, Robotic Arms, Deep Deterministic Policy Gradient (DDPG), 2D Simulation, Neural Networks, Actor-Critic Model, Python, Visualization

Introduction

The integration of Artificial Intelligence (AI) into robotics has revolutionized the field, offering unprecedented advancements in automation, precision, and adaptability. Among the various applications of AI in robotics, robotic arms represent a particularly intriguing area of study due to their wide range of applications in industries such as manufacturing, healthcare, and service sectors. This paper delves into the application of AI algorithms, specifically Reinforcement Learning (RL), in the control and operation of robotic arms.

The advent of sophisticated AI algorithms has opened new frontiers in the capabilities of robotic arms. These algorithms enable robotic arms to perform complex tasks with a level of precision and autonomy that was previously unattainable. Among these algorithms, Reinforcement Learning stands out due to its ability to learn optimal actions through trial and error interactions with the environment. This learning paradigm is particularly well-suited to robotic arms, which often operate in dynamic and unpredictable environments.

Our research focuses on the Deep Deterministic Policy Gradient (DDPG) algorithm, a cutting-edge RL technique known for its efficiency in handling high-dimensional, continuous action spaces – a common characteristic in robotic arm applications. The DDPG algorithm, which combines the strengths of Deep Q-Networks (DQN) [2] and Actor-Critic methods, provides a robust framework for training robotic arms in simulation environments.

To facilitate our research, we have developed a custom simulation environment, termed ArmEnv. This environment is tailored to mimic real-world scenarios, providing a realistic and controlled setting for training and evaluating the performance of the AI-driven robotic arm. The ArmEnv plays a crucial role in our study, allowing us to simulate various tasks and challenges that the robotic arm must learn to overcome.

This paper presents a comprehensive study of the application of the DDPG algorithm in training a robotic arm within the ArmEnv. We detail the architecture of our RL model, the training process, and the results obtained. Furthermore, we discuss the prospects of extending our research to more complex scenarios, including transitioning to a 3D simulation environment and enhancing the reward and feature engineering aspects of the model.

Through this research, we aim to contribute to the growing body of knowledge in AI-driven robotics, showcasing the potential of RL algorithms in enhancing the functionality and efficiency of robotic arms. Our findings have significant implications for the future of automated systems and the role of AI in advancing the capabilities of robotic technology.

1 Problem Statement and Related Works

1.1 Modeling Process

In order to be able to apply AI algorithms, on our robotic arm, first we need to mathematically model the robotic arm. The modeling process is very important because it directly determines the difficulty stability and extensibility of the algorithm implementation. Next we'll describe how we modeled it.

1. Create a coordinate system

First we create a coordinate system. Then we can calculate the position of the gripper based on the length and angle of the robotic arms.

2. Set action space

We use the angle of the robotic arm as the action space. The arms have rotating joints between them. So we can calculate the angle of each arm relative to the coordinate system.

3. Set reward policy

When the gripper can't reach the target, we use the distance between the gripper and the target to calculate the reward value. When the robotics can reach the target, we use the reaching times to calculate the reward value.

1.2 Model Schematic

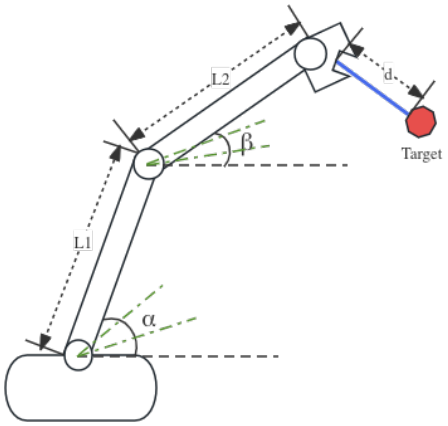


Figure 1: Arm Model

The schematic shows an example of a two-section robotic arm.
Where:
- L_1, L_2 are the lengths of each arm.
- α, β are the angles of each arm.
- d is the distance between gripper and target.

Calculate position of gripper:

$$\begin{aligned}(x_0, y_0) &= (0, 0)\\ (x_1, y_1) &= (L_1 \cos\alpha, L_1 \sin\alpha)\\ (x_{gripper}, y_{gripper}) &= (L_1 \cos\alpha + L_2 \cos\beta, L_1 \sin\alpha + L_2 \sin\beta)\end{aligned}$$

We set the base of robotics arm as origin point. Then we can use length and angle of each arms to calculate the position of gripper.

1.3 Model Algorithm

After building the model, we started to research what algorithm to choose to realize the robotic arm grasping the target.

1. DFS/BFS

Initially we wanted to use the traditional path search algorithm such as DFS/BFS to realize the movement of the robotic arm. DFS/BFS are very common algorithms applied in path planning, which can applied our model. Each robotics arms takes action to explore a feasible path to reach the target.

At first, the arm movement unit was set to 5 degrees, but in the end, it always returned no solution because it is difficult for the gripper to reach the arbitrary position of the target based on the limited action space. Of course, we can set a very small action unit, but it will take more computation time and more memory. At the same time, the search algorithm cannot solve the problem in real time.

2. Q-Learning

Q-learning [3] is a value iteration-based reinforcement learning algorithm suitable for problems with discrete states and actions. In path planning, states can represent the position and orientation of a robotic arm, while actions represent chosen paths.

If we set arm movement unit too small, the model will keep a large Q table, which will cost a large amount of training time and memory space.

3. DDPG

We eventually abandoned the use of traditional search algorithms and Q-learning to implement our model. So we want to find a better way.

After our research, we finally chose to use the DDPG algorithm, which is an Actor-Critic algorithm designed for continuous action spaces. It can be applied to learn continuous action policies for robotic arm path planning.

2 Method Applied

2.1 DDPG Introduction

DDPG (Deep Deterministic Policy Gradient) is an reinforcement learning algorithm that combines deep learning, policy gradient methods, and value function estimation. Specifically, DDPG consists of two main components: the Actor Network and the Critic Network.

1. Deep Q Network (DQN):

DQN extends Q-learning with deep learning by approximating the Q-value function using deep neural networks. It can handle problems with continuous state and action spaces, making it more versatile in robotic arm path planning.

2. Actor-Critic Model:

The Actor Network is responsible for learning and outputting the policy, i.e., given the current state, it outputs the corresponding action. The goal of this network is to maximize the cumulative reward by directly learning the optimal policy. The Actor Network typically employs a deep neural network structure, often utilizing fully connected layers.

The Critic Network estimates the Q-value for a given state-action pair, representing the long-term cumulative reward of taking a specific action in the current state. This helps the Actor Network make more accurate action selections by guiding the learning process to maximize the estimated Q-value. The Critic Network also adopts a deep neural network structure.

In DDPG, the Actor and Critic Networks are two separate neural networks, but they work together to facilitate path planning. The Actor Network updates its parameters using policy gradient methods, while the Critic Network learns to estimate Q-values.

DDPG is primarily used for problems with continuous action spaces, and thus, the Actor Network outputs values for continuous actions. The algorithm employs experience replay to optimize learning effectiveness and reduce the correlation between samples.

In summary, DDPG combines policy gradient methods and value function estimation within the Actor-Critic framework, making it suitable for solving reinforcement learning problems with continuous action spaces, including path planning for robotic arms.

2.2 Model Structure

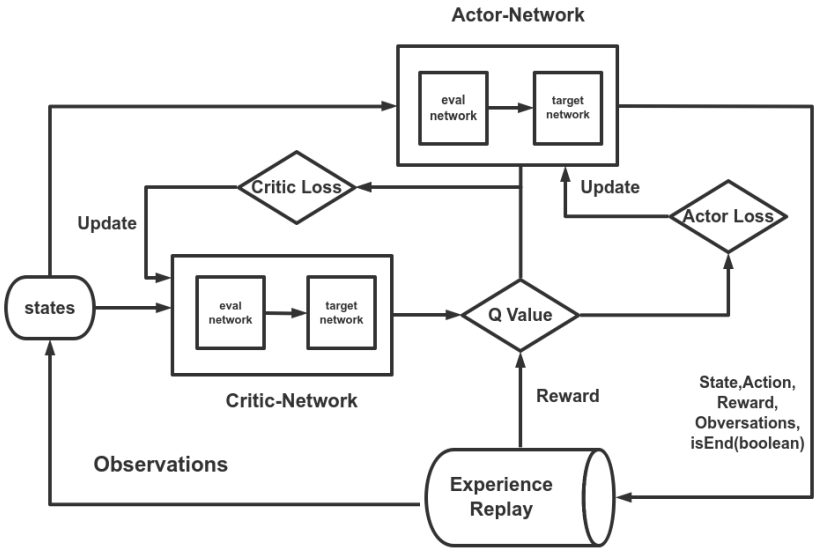


Figure 2: DDPG model structure

In our project, we use the DDPG as the model to train our model. As we reminded in the preview part, the DDPG is based on the DQN and Actor-Critic Network. The structure of the DDPG is show in the figure. As an Actor-Critic Model, we have two networks, the Actor network, and the Critic network.

As same as DQN, in both of Actor-Network and Critic-Network, they contain eval network and target network. We use the eval netwrok to train the data.

For the Network, we will get the state as input to the Actor-network and the Actor-network will output an action. For this action, we will get the observations and reward, also it will check whether this state is the end state or not. Pass the state, observations, action, reward, and a boolean value to show if this observation is the end state or not to the experience replay collection. Then we will update the observation as a new state.

For the Critic network, it will get the state-action pair as input and output a probability distribution. The Q value is updated by rewards and distribution by:

$$y_j = \begin{cases} R_j & isEnd_j istrue \\ R_j + \gamma Q'(\phi(S'_j), \pi_{\theta'}(\phi(S'_j)), w) & isEnd_j isfalse \end{cases}$$

After the update. we will use the gradient backpropagation with updated Q-value to update the Critic-network and the Actor-network parameter.

For the Actor-network, we use

$$J(\theta) = -\frac{1}{m}\sum_{j=1}^{m} Q(s_i, a_i, \theta)$$

to get the loss. Through the backpropagation, we could update θ.

For the Critic-network, we use

$$L(w) = \frac{1}{m}\sum_{j=1}^{m} Q(\phi(S_j), A_j, w))^2$$

to get the loss and use the backpropagation to update.

For the target network, we use the soft update to update the Actor network and Critic network.

In the final, if the S' is the end state, we just stop the iteration. Otherwise, we will keep do the preview step until we find it.

3 Experiments and Results

3.1 Environment Setup

Similar with OpenAI gym environment we've come across during lectures and assignments. Our project also requires and simulation environment that helps realize all the steps and actions, returning the states and rewards per attempt.

The only difference is that for a robotic arm simulation in this project, we don't have such conception as transaction probability or potential states, each state and action pair should be leading to a specific and concrete state with a consistent reward.

To realize what discussed above, a custom environment is required. We have conducted a **ArmEnv** Class in Python that help us realized these goals. The class have certain self attributes like **arm info** and **goal info** to help us keep track of the current status by storing the angles of each arm joint and the coordinate of the goal position, which could further be further transformed into states and rewards info. It also have functionalities like **step** and **reset**, that help simulate the process of taking the action an retrieving the environment information.

Also, handling works with coordinates and angles, visualization process is crucial. Not only for demoing purposes, but also an indispensable process while building environment and verifying all the outcomes. So, similarly with the visualization as render() function in OpenAI gym, we also implemented our render() method with a stand along visualization class using functionalities provided by package **pyglet**. The Viewer class plot the positions of each arm joint by calculation from each joint angle info getting from our env class and get the start and end position of each joint by applying trigonometric functions.

3.2 Training Loop

To Implement an RL algorithm, we need to understand how the outer loop are constructed. As shown below in the pseudo code, we need certain constrains as MAX_EP and MAX_STEPS for each iteration.

During each EP, the ArmEnv would be reset to a random state (Meaning both the arm position and goal state would be randomized). After starting each EP, the agent would be allowed to take no more than the MAX_STEPS to try to reach the goal.

In this process, each action-state pair would be used to fill/update the memory bank of our RL model. The date are stored for offline training purposes. Different from common neural network deep learning models, this RL model gather its own data instead of using public dataset as it needs to learn from the current, unique environment. Whenever the memory bank is full, we trigger the training process, and also we would reorder the index of the current data so that we could replaced the one that have been used in our database when a new data came.

The rest is just picking the best option by the current given state, and keep retrieving info from the env for further decision purposes which wouldn't be any different between training and evaluation stages.

Algorithm 1: Training Algorithm Pseudo code

Data: ArmEnv Class, RL model Class, MAX_EP, MAX_STEPS
Result: Trained Reinforcement Learning Model

```
for i ← 1 to MAX_EP do
    s ← env.reset();
    for j ← 1 to MAX_STEPS do
        a ← choose_action(s) by RL model;
        s_, r, done ← env.step(a);
        store_transition(s, a, r, s_) to RL model;
        if memory_full then
            learn() in RL model;
        end
        s ← s_;
        if done then
            break;
        end
    end
end
```

3.3 RL Implementation

As mentioned above, we have implemented an RL learning class applying DDPG algorithm that we have learnt during our research. An algorithm that applied DQN and Actor-Critic idea. Therefore, we would need several hyper parameters while construction our model.

LR_A, LR_C : Learning rate for our actor and critic unit.

GAMMA : The reward discount when calculating Q value and reward related info.

TAU : Soft replacement rate for updating the target network parameters in each unit.

MEMORY_CAPACITY : The max number of action-state pair stored in the memory bank.

BATCH_SIZE : The number of data to fetch from the memory bank each time when calling train().

After defining above parameters, we also need 2 pair of neural network for our actor and critic unit. Here below are the diagram for both our Actor and Critic unit. Note that the eval and target network mentioned above that exist in each unit are adopting the same network structure, so there are just 2 diagram intotal.

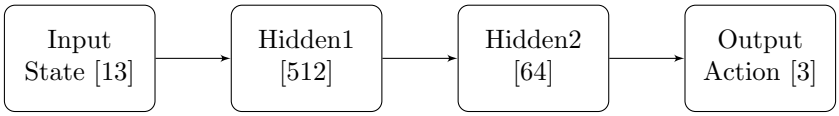


Figure 3: Actor Network

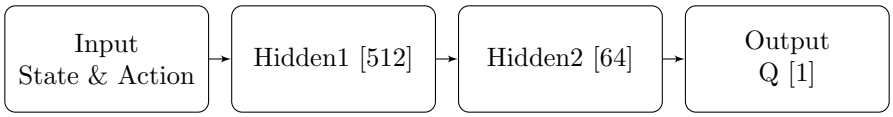


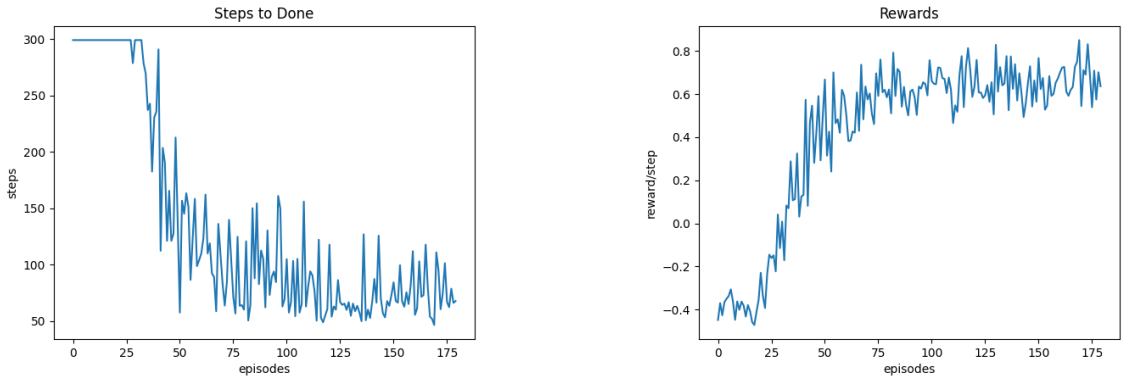
Figure 4: Critic Network

For our Actor unit, we take the number of state dimension as input and have a simple feed forward network with a number of action dimension as the output. The Critic network is a bit different, since the shape of action and state are not the same, we are doing a transform process to map each input from their own number of dimension into a total number of 512 neurons as the first layer, therefore we are able to take both action and state as input and output only one value as the estimate Q value for this action-state pair.

4 Discussion and Conclusion.

4.1 Result and Evaluation

During implementing and training our RL model, we have chosen to observe the number of steps and as well as the reward per steps gained during each EP to check if our agent indeed learning from the environment. 300 on top is where our max number of steps that could be take per iteration, 50 which the steps are converging to is the number of steps that we require the agent to stay on the target after reaching it before we call it done. After all, the agent can't be allowed to just take an action that could swing pass the target point since in a real world circumstances, we actually need certain time for further grabbing.



(a) numSteps

(b) rewards

Figure 5: model analysis

In order to realize our final goal of having the agent being able to reach any arbitrary location, we first start with constructing a robot arm with 2 moving joint and learning to reach a fixed goal. Later after that we have complicate the problem to 3 joints which involves more state parameters/features, and eventually we have randomize our goal state position during each reset to train the model to be more robust.

Result shown as what we've present during class time. We are able to realize training the agent to reach certain mouse position from a random state.

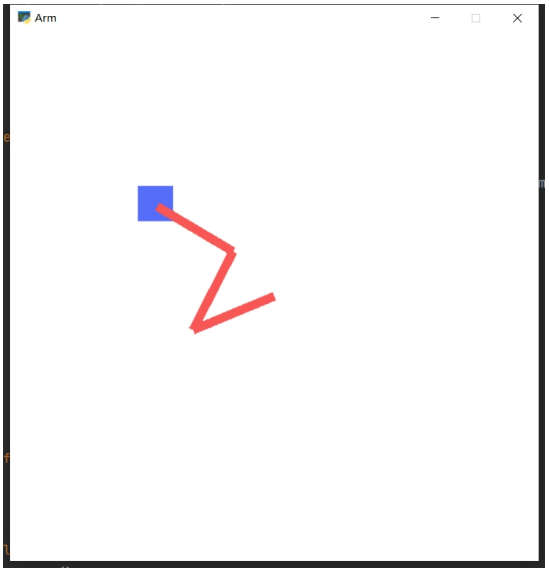


Figure 6: Arm Demo

4.2 Prospect

Certain prospects are proposed during defense on class.

First, We have considered enhancing the arm simulation to a 3D environment, which would seems much more fancy. However, the process of moving from 2D-3D has more to do with math, coordinate transformation than AI algorithms which we believe shouldn't be the main purpose of this project.

Second, reward engineering. Classmates have proposed to consider the weight of each joint when calculating rewards. It has been one of the initial thoughts we have in mind when we proposing this project. Further more, the angle of the last arm joint with the target could also be a measurement since a higher vertical angle would conceptually lead to a higher probability of success in actual gripping.

Last but not the least, feature engineering. We have discovered during implementation that the quality of features actually infect the result of training. Initially we used angles which similar to actions as our state output, turns out it's not as good as the performance of using coordinate locations and x/y distances as our state input to our learning network. Which we believe should also be highly considered when doing future work and researches.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2016.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] C. J. Watkins, "Learning from delayed rewards," *Ph.D. thesis, University of Cambridge*, 1989.