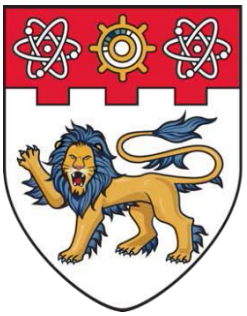

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CONTINUOUS CONTROL FOR ROBOT BASED ON DEEP REINFORCEMENT LEARNING

ZHANG SHANSI

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

2019

CONTINUOUS CONTROL FOR ROBOT BASED ON DEEP REINFORCEMENT LEARNING

ZHANG SHANSI

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Master of Engineering

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

2019/5/14	zhang shansi
Date	ZHANG SHANSI

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

14 May 2019

Date

HU GUOQIANG

Authorship Attribution Statement

This thesis contains material from 0 paper(s) published in the following peer-reviewed journal(s) where I was the first and/or corresponding author.

2019/5/14 ..... Date

zhang shansi ..... ZHANG SHANSI

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Prof. Hu Guoqiang, for his invaluable guidance and continuous support throughout my MEng study. His encouragement helped me to go through the difficult time during my MEng period. His perpetual energy and enthusiasm in research extremely motivated me in my study. He is also very patient and always willing to help me whenever I have questions.

I am also very grateful to my fellow lab mates for their kindly help and instructions in my research and they have made my MEng journey more meaningful and enjoyable.

Last but not least, I would like to express my sincere appreciation to my parents, for their constant love, encouragement and support since they are always my solidest backup.

Contents

Abstract

One of the main targets of artificial intelligence is to solve the complex control problems which have high-dimensional observation spaces. Recently, the combination of deep learning and reinforcement learning has made remarkable progress, including the high-level performance in the video and board games, 3D navigations and robotic control. In this thesis, deep reinforcement learning algorithms are studied to perform some robotic tasks with continuous action spaces.

Firstly, we use deep deterministic policy gradient (DDPG) and hindsight experience replay (HER) with a simple binary reward to achieve a multi-goal reinforcement learning task, making the redundant manipulator learn the policy of reaching any given position. Then we use DDPG with a shaped reward to train the redundant manipulator to complete the same task. By referring to the idea of HER, we propose a *future* and *random* strategy to obtain some additional goals combined with the shaped reward to generate some new transitions, which can help to improve the sample efficiency. After that, we use DDPG with prioritized experience replay to realize the trajectory tracking task of a SCARA robot and a mobile robot. Two training strategies, random referenced state initialization and early termination, are introduced to enable the robots to learn effectively from the referenced trajectories.

Secondly, we focus on the distributed deep reinforcement learning. We use asynchronous advantage actor-critic (A3C) and synchronous advantage actor-critic (A2C) algorithms, both of which have multiple workers to collect the transitions and compute the gradients, to train the redundant manipulator to complete the multi-goal task. We propose a new reward function to optimize the reaching path of the end-effector. The performances of agents trained by different algorithms and reward functions are compared. Next, we propose a

distributed framework of DDPG, where the synchronous workers generate transitions and compute gradients for the global network and the collecting workers only produce transitions for the shared replay memory with different policies and exploration noises. We use this proposed distributed DDPG with prioritized experience replay to train the SCARA robot and mobile robot to track the same trajectories, which presents a faster learning speed and smaller tracking errors compared with the single-worker DDPG.

Next, we study on the proximal policy optimization algorithm (PPO) with generalized advantage estimation (GAE). We propose a distributed framework of PPO by running multiple workers to collect transitions for the global network at the same time. Then we use this distributed PPO with GAE to train the redundant manipulator to achieve the multi-goal task and make comparison with the previous methods. After that, we use distributed PPO with GAE and the improved training strategies to train the mobile robot to track the trajectories. In order to improve the training and sample efficiency, a two-stage training strategy which consists of the supervised pre-training and fine-training by distributed PPO is proposed. This two-stage training strategy can also obtain a better tracking performance. Then we introduce LSTM to represent the actor and critic, and use buffers to store the cell state and hidden state of LSTM used for the initialization of each episode to solve the problem of inaccurate initial LSTM states. By introducing LSTM, the tracking performance of mobile robot can be improved compared with distributed PPO with fully-connected networks.

Finally, we utilize deep reinforcement learning to train a autonomous vehicle to learn the driving behaviors. Deep reinforcement learning provides an end-to-end method for the autonomous driving by directly mapping the high-dimensional raw sensory input to the control command output. We design a reward function which encourages the vehicle to drive along the road smoothly and overtake other vehicles. We adopt a two-stage training strategy which consists of the imitation learning stage and deep reinforcement learning stage. The imitation

learning stage could help to solve the exploration and sample efficiency problem of reinforcement learning. We use DDPG and the improved algorithm, TD3 to train the autonomous vehicle in the second training stage, respectively. We find that TD3 could improve the driving performance of autonoumous vehicle.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Artificial intelligence as the most advanced and cutting-edge technology has become the core driving force of new industrial transformation. One of the main goals of artificial intelligence is to solve the complicated control problems. Reinforcement learning as an important method of machine learning can be used to solve a variety of control problems. Recently, the integration of deep learning and reinforcement learning has made notable progress in many complex tasks. Deep reinforcement learning began to be paid more attention to since AlphaGo defeated human professional player in the game of GO. The extraordinary ability of AlphaGo is attributed to combining deep learning with reinforcement learning, because deep neural netwroks could provide the rich representations which enable the reinforcement learning algorithms to perform effectively. Deep reinforcement learning has achieved high-level performance in the board games like Go and chess, video games like Atari, 3D navigations, robotics control and even some very complicated games such as Dota and StarCraft.

The first author of AlphaGo has ever mentioned that he agreeed deep reinforcement learning is the key to solve the problem of general artificial intelligence.

Almost any task solution can be regarded as a mapping from the formal coding input to the decision distribution output, and the nonlinear deep neural network is a very good representation learning tool. Learning ability can be mainly divided into deductive method and inductive method. Reinforcement learning is like a deductive method based on the rewards, since the policy can be produced by giving the external environment and the corresponding rewards. Deep learning is like the inductive method based on the experiences and memory by giving the input and label and then learning the representation through the neural network. Self-Reflective capability is the key for the agent to produce self-consciousness and be independent of humans. Self-Reflective capability can be obtained from reinforcement learning to some extent by trial and error to strengthen self decision-making ability. Even if this self-reflection is limited by its existing world, as long as the environment and rewards are well-defined, the self-reflection is equivalent to promote its learning ability in the given environment.

The nature of deep reinforcement learning more accords with the requirement of general artificial intelligence. Making the robot perceive, reason and make decision like human is a hot topic in the artificial intelligence field. Traditional control methods usually need the exact dynamic model of the controlled system to design a precise controller which satisfies the Lyapunov stability. However, sometimes it is difficult to get the exact dynamic model of the system, so those model-based control methods may not achieve good performance in the real application. Moreover, the controller design process is very complicated, which requires a lot of mathematical analysis. Model-free deep reinforcement learning could provide a promising method for the robot to learn various complex tasks. 'Model-free' means it is not necessary to know the dynamic model of the system. The policy of the agent is represented by the neural network which only takes the observation from environment as input. The agent learns the policy from the interaction with the environment by maximizing the long-term rewards corresponding to the objective of specific task. The policy is constantly

updated by the experiences of the agent with the interaction process and gradually approaches to the optimal one. Therefore, the policy which is equivalent to the controller doesn't incorporate any model information. Model-free deep reinforcement learning is usually used as the high-level control method, which enables the robot to achieve the end-to-end learning by directly mapping the perceptual input to the action output. However, it still needs to be followed by the low-level controllers which output the final torques applied to the motors in the practical systems.

Deep reinforcement learning mainly focuses on the high-level control. The combination of this high-level decision making and low-level control can be applied to solve many complex robotic tasks that are difficult for the traditional control methods. The robot can gradually learn to make correct decisions after perceiving the environment during the process of exploring the environment. The policy of the robot is updated according to its own experiences and the feedback from environment, rather than the labeled data as supervised learning where the correct actions are already known, so the robot seems to possess the similar intelligence with human. Therefore, deep reinforcement learning plays an important role in the development of robotics and artificial intelligence.

1.2 Objectives

In this thesis, we hope to utilize model-free deep reinforcement learning algorithms to train the robots to learn some specific tasks. Firstly, we consider a multi-goal task for a redundant manipulator to make its end-effector reach any given position. Deep reinforcement learning frameworks are designed for the manipulator to learn the policy to gradually approach to any goal position directly from the local observation. Next, we focus on the trajectory tracking task of a SCARA robot and a mobile robot. Model-free deep reinforcement learning are employed to perform the high-level control for the robots based

on their kinematics without designing the complex controllers incorporating the information of the exact dynamic models. Finally, we hope to employ deep reinforcement learning to train a autonoumos vehicle to learn the driving policy end-to-end, which directly maps the high-dimensional raw sensory input to the control command output.

We mainly do some exploratory works related to the model-free deep reinforcement learning to make the high-level decision end-to-end for the different robotic tasks. This method can also be extended to solve other more complicated robotic tasks which are difficult to be dealt with by the traditional control methods.

1.3 Major contribution of the thesis

This thesis mainly studies on the deep reinforcement learning algorithms of continuous control to solve some robotic tasks.

The first robotic task is a multi-goal task which requires a redundant manipulator to reach any given position from a random initial position during each episode. Firstly, we employ DDPG with hindsight experience replay using a simple binary reward to train the manipulator. We compare the final performances of different strategies to choose the additional goals and find all the strategies could achieve nearly 80% success rate. Then, we adopt DDPG with a shaped reward to train the manipulator, but it can only get less than 10% success rate. By referring to the idea of hindsight experience replay, we propose a $future$ and *random* strategy combined with the shaped reward to produce some additional transitions. By introducing these strategies, the success rate of the manipulator increases substantially. We find the *random* strategy could achieve a better performance than the $future$ strategy due to the improved diversity of the additional goals. These two strategies can also help to promote the sample efficiency since some transitions can be generated artificially. Next, we use A3C and A2C to train the redundant manipulator respectively. In or-

der to optimize the reaching path of the end-effector, we propose a new reward function to encourage the end-effector to move approximately along the ideal direction. From the simulation results, we can see that this new reward function can significantly optimize the reaching path and also increase the success rate. Moreover, we find that A2C can obtain better performance than A3C, which may be because of the avoidance of gradient delay. We also use distributed PPO with the two reward functions to train the redundant manipulator. The simulation results show that distributed PPO with the path optimization reward can achieve the highest success rate by comparing with other methods.

Then we focus on the robotic trajectory tracking task. Firstly, we use DDPG with priority replay memory to train a SCARA robot and a mobile robot to track some given trajectories. In order to make the robots learn effectively, we adopt two training strategies, random referenced state initialization and early termination. In this way, the robots can overcome the exploration problem and learn to track the given trajectories by using a normalized exponential reward function successfully. However, there is still space to make some improvements. Then we propose a distributed framework of DDPG which has multiple synchronous workers to compute gradients for the global network and multiple collecting workers to explore the environment with different policies and exploration noises. All the workers interact with the environment to collect transitions for the shared replay memory. This framework can avoid the problem of gradient delay and the update of network can exploit more transitions with relatively high priorities. It can also promote the data throughput and increase the diversity of transitions in the replay memory to achieve better exploration to the environment. We use this proposed distributed framework of DDPG to train the SCARA robot and mobile robot again. The training results show that the agents can learn faster and get smaller tracking errors than the single-worker DDPG. Next, we investigate whether PPO could solve the tracking problem for the mobile robot. We firsly use distributed PPO and GAE with the improved training strategies to train the mobile robot, but it needs much more training

episodes to get relatively good results. Hence, we propose a two-stage training strategy which comprises of supervised pre-training and fine-training by distributed PPO. Pre-training can help the agent learn some useful experiences quickly, after which the agent doesn't need to learn the policy by reinforcement learning from scratch. So it can decrease the number of interaction steps of the agent to promote the sample and training efficiency. Then we introduce LSTM to represent the actor and critic. The global network utilizes the sequential transitions collected by all the workers to update its parameters. Aiming at the problem of inaccurate initial LSTM state, we use buffers to store the cell state and hidden state of LSTM for the initialization of each episode. The states in the buffers can also be updated during the training process. By comparing with distributed PPO with fully-connected networks, we find distributed PPO with LSTM could achieve a better performance in terms of the tracking errors.

Finally, we train a autonomous vehicle to learn the driving behaviors by deep reinforcement learning. Deep reinforcement learning could provide an end-to-end method which only takes the raw sensor data from the environment as input and directly outputs the maneuvering commands. The sensor information includes the pose and velocity of vehicle, the distances from the track edges and the distances from other vehicles. We design a reward function that could encourage the vehicle to drive along the track smoothly and overtake other vehicles. We use DDPG and its improved algorithm, TD3 to train the autonomous vehicle, respectively. Aiming to solve the exploration and sample efficiency problem of reinforcement learning, we adopt a two-stage training strategy. The vehicle is firstly trained to imitate the expert behaviors and then trained by deep reinforcement learning. We test the trained agents on different roads, including the training road and three more complicated new roads. Our autonomous vehicle can still drive smoothly on those more complex unseen roads, which demonstrates the strong generalization ability. By comparing the agents trained by DDPG and TD3 in the second stage, we find TD3 can improve the driving performance mainly by increasing the speed of autonomous vehicle.

1.4 Organization of this thesis

In chapter 2, we firstly introduce Markov decision process, which is the prerequisite of reinforcement learning. Then we give a brief introduction on the two dynamic programming methods, policy iteration and value iteration, and two value-based reinforcement learning algorithms, SARSA and Q-learning. Next, we give a preliminary introduction on the two deep reinforcement learning algorithms, Deep Q-Network and policy gradient.

In chapter 3, we introduce the deep deterministic policy gradient (DDPG) and two improvements for the replay memory, prioritized experience replay and hindsight experience replay. Firstly we use DDPG and hindsight experience replay with a simple binary reward to train a redundant manipulator to complete a multi-goal task. Then we propose a $future$ and $random$ strategy to get some additional goals combined with a shaped reward to generate some new transitions, which could promote the sample efficiency and enables the redundant manipulator to complete the multi-goal task successfully. After that, we use DDPG with prioritized experience replay to train a SCARA robot and mobile robot to track some given trajectories. Two training strategies, random referenced state initialization and early termination, are adopted to enable the robot to learn effectively from the referenced trajectories.

In chapter 4, we focus on the distributed deep reinforcement learning. Firstly, we introduce asynchronous advantage actor-critic (A3C) and synchronous advantage actor-critic (A2C) algorithms. Then we use these two algorithms to train the redundant manipulator to complete the multi-goal task. We propose a new reward function to optimize the reaching path of the end-effector. The performances of different algorithms with two reward functions are compared. Next, we propose a distributed framework of DDPG which has multiple synchronous workers and collecting workers. Then, distributed DDPG is used to train the SCARA robot and mobile robot again and their tracking performances are compared with the single-worker DDPG.

In chapter 5, we firstly introduce the proximal policy optimization (PPO) and generalized advantage estimation algorithm (GAE). We also propose a distributed framework of PPO. Then we use distributed PPO with GAE to train the redundant manipulator to complete the multi-goal task. After that, we use distributed PPO with the improved training strategies to train the mobile robot to track the given trajectories. In order to promote the training efficiency, we adopt a two-stage training strategy consisting of supervised pre-training and fine-training by distributed PPO. Next we introduce LSTM to represent the actor and critic, and use buffers to store the cell state and hidden state of LSTM which are used for the initialization of each episode to mitigate the problem of inaccurate inital LSTM states. The tracking performances of different methods are compared.

In chapter 6, deep reinforcement learning is employed to train a autonomous vehicle to learn the driving behaviors end-to-end. We design a reward function which could make our vehicle drive smoothly along the roads and overtake other vehicles. We also propose a two-stage training strategy consisting of pre-training by imitation learning and fine-training by deep reinforcement learning. After pre-training, we firstly use DDPG to further train the autonomous vehicle. Then we test the agent after the two-stage training on different roads, including some more complex unseen roads. Next we introduce an improved algorithm, TD3 to train the autonomous vehicle in the second stage. The driving performances of agents trained by DDPG and TD3 are compared.

Chapter 7 gives the conclusions and briefly discusses the future work.

Chapter 2

Preliminaries

Firstly, we review some classical deep reinforcement learning algorithms, improved algorithms and some successful applications of deep reinforcement learning. Then we will explain the principles of some basic reinforcement learing algorithms, which are the foundation of the following chapters.

2.1 Literature review

2.1.1 Classical deep reinforcement learning algorithms

The first notable application by using deep reinforcement learning (DRL) to learn the control policy was through deep Q-network (DQN), which directly receives the high-dimensional sensory input [1, 2]. The model is represented by the convolutional neural network (CNN) and outputs the value estimations of all the possible actions. The testing results on the classical Atari 2600 games demonstrated that DQN agent outperformed all the previous algorithms and could achieve human-level performance across some of the games.

DQN adopts the optimal value of the next time step when computing the target

value. However, it utilizes the same network to choose action and calculate target value, which may result in the value overestimation. Aiming to the overestimation problem, Google DeepMind proposed the Double DQN algorithm [3], which uses a behavior network to choose the optimal action and a target network to calculate the target value. This algorithm could yield more accurate value estimation and led to a better performance on the Atari games.

Another improvement of DQN is the Dueling DQN [4], which introduces two separate estimations: state value function and state-action advantage function, both of which share a common CNN to learn the features from images. It transforms one output to two outputs and the summation of the two outputs is the state-action value. This dueling architecture can learn the state value without learning the effect of each action on the state. Each part decomposed from the state-action value has a practical meaning and much valuable information can be excavated from it.

Google DeepMind combined six extensions of DQN to develop a merged model called Rainbow [5]. which includes double Q-learning, dueling network, prioritized replay memory [6] to give different weights to different transitions, multi-step learning, distributional network [7] to represent the probability of value in the form of histogram, noisy network [8] with parametric noise added to the weights to assist efficient exploration. The experiments showed these combinations could achieve a state-of-the-art performance on the Atari games.

DQN and its variants can only be applied to solve the problems with discrete action space, but many complex tasks in the real world have continuous action space. Hence, reinforcement learning algorithm applicable to the continuous control should be developed. Google DeepMind proposed a deterministic policy gradient algorithm (DPG) [9] with continuous actions, which calculates the expected gradient of action-value function rather than the stochastic policy gradient. Based on the DPG algorithm, Google DeepMind further proposed a deep deterministic policy gradient algorithm (DDPG) [10], which adopts an actor-

critic structure and introduces two important improvements of DQN, replay memory and target network. DDPG can learn the policies directly from the raw pixels in many physics tasks.

DDPG is an off-policy DRL algorithm, and a typical representative of on-policy reinforcement learning is the asynchronous advantage actor-critic (A3C) [11], which adopts multiple parallel workers to collect the transitions and compute the gradients. The gradients from each worker are sent to the global network asynchronously. A3C can solve both the discrete and continuous control problems successfully.

Researchers from U.C.Berkeley developed a trust region policy optimization algorithm (TRPO) [12], which is effective to optimize large nonlinear policies represented by the neural network and guarantees monotonic improvement of the policy. It demonstrated a robust performance on an extensive variety of tasks. The computation of TRPO is complicated, so its computing speed is slow compared with other policy gradient algorithms. Thus, OpenAI proposed a proximal policy optimization algorithm (PPO) [13], which not only has the advantages of TRPO but is also much easier to implement. PPO enables multiple updates per mini-batch sample, so it has a better sample efficiency. The experiments showed that the performance of PPO exceeded other policy gradient methods on a series of benchmark tasks. Google DeepMind also used PPO with a slightly different objective function to train the simulated bodies in the rich environments [14]. The agents can learn some locomotion behaviours without explicit reward-based guidance. Both the PPO objective function of OpenAI and Google DeepMind have the same core idea to limit the update amplitude of the policy.

2.1.2 Improvements of deep reinforcement learning algorithms

Besides the above-mentioned classical deep reinforcement learning algorithms, many improvements on the classical algorithms have also been studied. Researchers from Berkeley put forward generalized advantage estimation (GAE) [15] which can make a good balance between the variance and bias for the policy gradient estimates by adjusting an exponentially-weighted hyperparameter λ. GAE combining with the trust region optimization yielded good empirical results on some high-dimensional continuous control tasks.

Google DeepMind proposed a $Retrace(\lambda)$ algorithm [16], which is feasible to use the samples generated from any policy and efficient when taking good advantage of the samples collected from the near current policy. Moreover, it has low variance and can automatically adjust the length of return to the 'off-policy' degree of any data.

Google DeepMind also developed a deep reinforcement learning agent which adopts an actor-critic structure with experience replay [17], abbreviated as ACER. This algorithm introduces the following innovations: truncated importance sampling with bias correction, stochastic dueling network architecture and a new TRPO method. It was proved to be stable and sample efficient since it exhibited a good performance on some challenging environments.

Researchers from University of Toronto introduced trust region optimization by using Kronecker-factored trust region [18], which is a scalable trust region natural gradient approach to optimize the actor and critic. It was tested on both the discrete and continuous tasks and achieved a better performance than the previous state-of-the-art on-policy DRL methods.

Researchers from University of Cambridge studied an approach for DRL to merge the on-policy and off-policy update [19]. They introduced a parameterized family which interpolates between the on-policy and off-policy algorithms

for policy gradient methods. They also showed that some recent DRL methods could be regarded as the special cases of this general family. The experimental results demonstrated this interpolated gradients could promote the sample efficiency and improve the stability of agent.

Google DeepMind adopted a distributional perspective to develop a new algorithm that uses Bellman equation to learn the approximate value distributions [7]. The empirical results revealed the importance of approximate value distributions in deep reinforcement learning.

Improvement on the proximal policy optimization was also studied. Xiaomi applied point probability distance [20] to prevent too large update of the policy, which is also conducive to the more exploration and stronger stability than KL divergence. It is an alternative of PPO but could get a better performance than PPO in Atari games and Mujoco experiments.

Based on the *Retrace*, Google DeepMind further proposed a new deep reinforcement learning agent architecture, called *Reactor* [21]. It includes the following contributions: a new policy evaluation algorithm, distributional *Retrace*, which introduces multi-step off-policy updates to the distributional deep reinforcement learning; β-leave-one-out policy gradient algorithm which makes a better trade-off between the bias and variance by exploiting the action value as baseline; a prioritized replay for sequences that utilizes temporal locality of the adjacent observations to realize more efficient replay priority. It was proved that these contributions could help to improve the sample efficiency and agent performance.

Another improvement is for DDPG whose common failure is the overestimation of Q-function, which could lead to the policy breaking since it may exploit the errors of Q-function. Researchers from McGill University put forward Twin Delayed DDPG (TD3) [22] which can address this issue by introducing three critical tricks: learning two Q functions rather than one and using the smaller value to calculate the target value; updating the actor less frequently than the critic; adding noise to the target action so that the policy has less possibility to

utilize Q-function errors after smoothing out the Q-function over actions.

There are also some works aiming to solve the exploration problem of deep reinforcement learning. Google DeepMind proposed NoisyNet with parametric noise introduced to its weights [8]. The parameters of the noise can also be learned along with the network weights by gradient descent. The experiments showed that higher scores in Atari games could be yielded with NoisyNet for exploration. OpenAI employed demonstrations to overcome the exploration problem [23], which was built based on DDPG and Hindsight experience replay. It is simple to implement and could speed up the training of agent by only collecting a small set of demonstrations. Another improved algorithm published roughly concurrently with TD3 by researchers from Berkeley is the Soft Actor Critic (SAC) [24], which introduces the entropy regularization. The SAC policy is trained to seek a balance between the expected return and entropy which can measure the policy uncertainty. Increasing entropy contributes to more exploration, which can speed up the subsequent learning. The policy can also avoid converging to a bad local optimum prematurely.

In addition to the exploration problem, sparse reward is another challenge faced by deep reinforcement learning. Some works are conducted to solve the problem of sparse reward. OpenAI proposed hindsight experience replay [25] which can be combined with DQN and DDPG to enable the agent to learn even from sparse or binary reward. It was applied to the manipulation tasks and achieved a much better performance than the same algorithm with a shaped reward. This technology will be introduced in detail in section 3.3. Google DeepMind proposed a scheduled auxiliary control method [26] by assigning the agent a set of auxiliary tasks which are learned by DRL concurrently. The key idea of this approach is the learned scheduling and execution of the auxiliary tasks allow the agent to explore the environment more efficiently.

Curiosity-Driven Learning is another promising method to solve the exploration and sparse reward problem. Researchers from Berkeley developed an exploration

strategy by maximizing the agent's surprise for its experiences [27], which is known as the intrinsic reward. The environment model needs to be learned and the intrinsic reward is derived from the KL divergence between the real model and learned model. Researchers from Berkely further proposed the curiosity-driven exploration [28]. The curiosity is formulated as the error of the agent's prediction after performing an action in the feature space which is learned by the self-supervised inverse dynamics model. This method can scale to the high-dimensional state space, such as image. OpenAI presented large-scale purely curiosity-driven learning without any hand-designed extrinsic reward on a set of benchmark environments [29]. The experimental results presented a good performance and a high consistency between the intrinsic curiosity reward and manually designed extrinsic reward. Google Brain improved the curiosity method by introducing a episodic memory to determine the intrinsic reward [30]. The current observation needs to be compared with those in the memory by evaluating how many steps it may take to get to the current observation from the observations in the memory. It can avoid to cause some unpredictable consequences for the agent. The agent trained by this approach could learn faster than the previous curiosity methods in some 3D environments.

Nowadays, most deep reinforcement learning agents are trained by distributed architectures in order to increase the sampling rate and decrease the training time. Google DeepMind proposed the first massively distributed framework of deep reinforcement learning [31], which creates parallel actors to collect data and parallel learners to learn from the stored experiences. They adopted this architecture to implement DQN on the Atari games. The experimental results showed that it outperformed the non-distributed DQN and also reduced the wall-time. Google DeepMind also adopted the distributional perspective on DDPG with distributed framework [32], as well as some improvements, such as N-step return and prioritized experience replay, which presented a state-of-the-art performance on some continuous control tasks. Google DeepMind further proposed a distributed framework for the off-policy deep reinforcement learn-

ing [33], which decouples acting from learning: the actors interact with the environment to collect data for the shared replay memory and the learner samples mini-batch of transitions from the replay memory based on their priorities. This architecture could substantially improve the performance in terms of the wall-clock training time.

Although deep reinforcement learning can exceed human performance in some tasks, a brand new agent needs to be trained for each new task. This means each agent can only complete one task which it is trained on and can't generalize to other tasks. Hence, some researchers study if a single agent can be trained to solve multiple tasks. Google DeepMind proposed an adaptive normalization method for the target which is used in the learning update [34]. It provides the possibilty to train a single system which can solve multiple different tasks with varied magnitude of reward. In order to realize a large collection of tasks by using one single agent, Google DeepMind developed Importance Weighted Actor-Learner Architecture (IMPALA) [35], a distributed agent that can be trained both in single machine and thousands of machines. It adopts an off-policy correction method which is called $Vtrace$ and can achieve stable learning with high throughput. The experiments on DMLab-30 and Atari-57 presented positive transfer among different tasks by using IMPALA. However, the single agent may focus more on the tasks with more salient reward signals. So Matteo Hessel et al. proposed a multi-task reinforcement learning agent with PopArt [36] which can adapt the contribution of every task to the learning update automatically. PopArt normalization enables the update to be invariant to the scale and sparsity of rewards. It is the first single agent that can surpass human-level performance on the multiple tasks.

2.1.3 Applications of deep reinforcement learning

We have introduced most of the popular deep reinforcement learning algorithms and their improvements in subsections 2.1.1 and 2.1.2. In this subsection, we will

introduce some applications of deep reinforcement learning in different fields.

The first shocking achievement obtained by deep reinforcement learning is AlphaGo which defeated human Go champion by 4 to 1. AlphaGo used a value network and a policy network combined with Monte Carlo tree search [37]. The deep neural networks were trained through the supervised learning from human expert experiences and reinforcement learning from self-play, so AlphaGo required human expert knowledge as the supervised training data. After that, Google DeepMind further developed AlphaGo Zero [38] which was trained by deep reinforcement learning without any human experience and guidance but only the game rules. AlphaGo Zero adopted a neural network to predict its action and the winner of the game, which was the improvement of Monte Carlo tree search method. AlphaGo Zero learned the policy from zero by self-play and could achieve a score of 100 to 0 against previous AlphaGo.

Deep reinforcement learning has obtained much success in playing board and video games, however, its most promising application field is robotics. It has already been applied to solve many robotic tasks successfully, including manipulation, robot navigation and autonomous driving.

Firstly, I will introduce some works that apply deep reinforcement learning to solve the manipulation tasks. Ivaylo Popov et al. [39] used DDPG with two extensions, multiple minibtch updates per environment step and distributed version to allocate data collection and network training to multiple robots, to successfully train a dexterous manipulator to grasp an object and then stack it on the other object precisely. Shixiang Gu et al. [40] used asynchronous Normalized Advantage Functions (NAF) with multiple collecting threads and one training thread to train the robot to learn some complex 3D manipualtion tasks in the simulation environment. This methods could also train the real robot to master the door opening skill without any prior demonstration. Josh Tobin et al. [41] proposed a domain randomization method which randomly renders the images from simulation to approximate to the images of the real

world. It is possible to train a real manipulator to detect different objects with enough accuracy and robustness to the environment variations by only using the images from simulation. Therefore, it can be a powerful technique for transferring the policy trained by DRL in the simulation to the real-world robots. Aravind Rajeswaran et al. [42] used natural policy gradient with a small number of demonstrations from human to train the dexterous multi-fingered hands to complete some tasks, including object relocation, tool using, door opening and in-hand manipulation. The use of demonstration could reduce the sample complexity and lead to the nature actions. Ashvin Nair et al. [43] proposed a method which combines the unsupervised representation learning, variational autoencoder (VAE) and off-policy goal-conditioned DRL. The purpose of VAE is to learn the structured representation of raw images, which can also be used to sample the synthetic goals for the off-policy RL to improve the sample efficiency. This method enables the real robot to effectively learn the policies from raw image observations. OpenAI [44] adopted Proximal Policy Optimiztion (PPO) with an asymmetric Actor-Critic approach to train a physical dexterous hand to learn the vision-based object reorientation. The policy can transfer to the real-world robot by domain randomization even if it is entirely trained in the simulation. This method doesn't depend on any demonstration from human but can present some human behaviours in manipulation naturally.

There are also many works related to the autonomous robot navigation via deep reinforcement learning. Lei Tai et al. [45] applied asynchronous DDPG method to the mobile robot mapless navigation. It was trained through an end-to-end method by only taking the sparse range findings as input without any manually feature extraction and prior demonstration. The trained agent could also successfully transfer to the unseen simulated and real environments. Yuke Zhu et al. [46] applied DRL to the mobile robot visual navigation of indoor scenes. The policy has relation with the current state and the goal, which can promote the generalized ability. They also proposed a framework which can provide abundant high-quality 3D scenes. It is also end-to-end trainable and can generalize

to the real robot scenes just with a little extra fine-tuning. Tingxiang Fan et al. [47] demonstrated a decentralized navigation and collision avoidance policy for the multi-robot system. The policy was trained by PPO with multi-scenario and multi-stage training framework over a large number of mobile robots in the complicated environments. A hybrid control strategy was developed to improve the robustness and effectiveness of the policy. The generalization ability of the learned policy could be verified in the unseen virtual scenarios and the real human crowd. Xi Chen et al. [48] utilized PPO to train a wheel-legged robot to acquire navigation skills. They proposed a novel training strategy which uses some simple navigation skills to learn complex navigation skills. They also adopted domain randomization to promote the diversity of training samples. Piotr Mirowski et al. [49] proposed an end-to-end DRL framework addressed by A3C to make the agent learn to navigate in the complex environments. The performance could be greatly improved by introducing two auxiliary tasks, inferring the depth information from RGB observation and loop closure detection to evaluate whether the current location has been visited previously. This method could learn the navigation skills from the raw image input in the complex 3D maze and approach human-level performance.

Autonomous vehicle is a hot research field recently. Deep reinforcement learning is regarded as a promising technique to learn the driving policy, so some works related to the autonomous driving using deep reinforcement learning are conducted. Ahmad El Sallab et al. [50] used an end-to-end method via DQN to learn the driving behaviors, which received the raw sensor data and outputs the driving actions. They used LSTM to deal with partially observable scenarios. They also integrated the attention model to extract the relevant information from the sensor data to reduce the computation. This framework was tested on the car racing simulator, TORCS. The simulation results presented successful learning of the driving behaviours. Xinlei Pan et al. [51] proposed a virtual-to-real DRL framework based on A3C for the autonomous driving. They used a realistic translation network to convert virtual images in the simulation to the

approximate realistic images based on their common scene segmentation. So the reinforcement learning agent trained by those synthesized realistic images could adapt to the driving environments in the real world. Maximilian Jaritz et al. [52] used A3C to train the driving policy end-to-end by only using RGB image from camera as input. They proposed a new reward function and training strategy leading to fast convergence and robust performance. The generalization ability was also verified in the unseen environments. Sahand Sharifzadeh et al. [53] adopted a DQN-based inverse reinforcement learning to acquire the rewards for the autonomous driving scenario. After some training episodes, the agent could generate collision-free motions and lane change behaviours like human driver. Meha Kaushik et al. [54] used DDPG to train the autonomous vehicle to learn the overtaking behaviors. The agent was trained by a method similar to the curriculum learning, where the agent firstly learned to drive on the road and then learned to overtake other vehicles. This method enabled the vehicle to learn the overtaking maneuvers successfully. Yilun Chen [55] adopted a Deep Recurrent Deterministic Policy Gradient method with a hierarchical action space to train the autonomous vehicle to learn the lane change behaviors. He also introduced the attention mechanism, including spatial attention, which detected the most important region of an image, and temporal attention, which assigned different weights to the last few frames. In this way, the vehicle could get better lane change behaviors.

Deep reinforcement learning can also solve the problems in other fields. Runsheng Yu et al. [56] applied DDPG to solve the trajectory tracking problem for the autonomous underwater vehicle (AUV). They combined optimal control with deep reinforcement learning to get a better tracking performance than traditional PID control. Hui Wu et al. [57] used DDPG to solve the depth control problem of AUV whose exact dynamic model is difficult to obtain. They also introduced prioritized experience replay to enhance the control performance. The simulation results demonstrated that model-free reinforcement learning method could get better performance than the model-based controllers, such as LQR

and MPC.

Xue Bin Peng et al. [58] developed a data-driven deep reinforcement learning framework based on PPO to train the simulated characters to learn the contorl policies. The learned policies are robust and can generate natural motions which are almost indistinguishable from the referenced motions. This method enabled the simulated characters to learn a variety of skills and execute tasks while imitating the referenced motions. However, it still has some limitations, so they added some improvements [59] to the previous work. They introduced a motion reconstruction method which could improve the quality of the referenced motions. Furthermore, they proposed the adaptive state initialization, where the initial state distribution was updated dynamically to contribute to the long-horizon performance of reproducing the desired motions. This framework could make the agents learn a much larger variety of skills and motions with higher fidelity.

2.2 Basic reinforcement learning algorithms

In the classical reinforcement learning, the agent needs to interact with the environment. At every time step, the environment has one state and agent needs to obtain the observation of the current state. Then the agent takes an action based on the observation and its policy. The action is performed to the environment and change the agent's state. Then the agent can get a new observation and a reward from the changed environment. This reward could either be positive or negative. Afterwards the agent will take a new action according to the new observation. This process is demonstrated in Fig. 2.1.

Next, we will introduce the foundation of reinforcement learning and some basic algorithms, including policy iteration, value iteration, Q-learning and policy gradient.

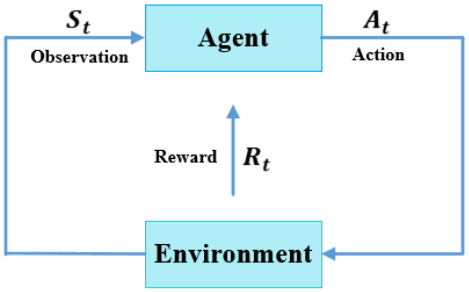

Figure 2.1: Interaction process of reinforcement learning

2.2.1 Markov decision process

Markov decision process (MDP) [60] is the foundation of reinforcement learning theoretical derivation, through which the interaction process of reinforcement learning can be represented by the form of probability. MDP can be expressed as a tuple (S, A, P, r, γ), where S and A are the finite set of states and actions respectively, P is the state transition probability, such that $P(s'|s, a)$ represents the probability of observing state s' after executing action a at state s, r is the received reward from the environment after performing an action, and $\gamma \in [0, 1]$ is the discount factor [60]. Markov process has the property that the next state only has relation with the current state and has no relation with the earlier states. Reinforcement learning can only be applied to solve the problems conforming to the Markov property.

The action choice of agent is determined by the policy that maps the state $s \in S$ to a probability distribution over action $\pi(A|s)$. Reinforcement learing aims to find a policy which can maximize the long-term returns, $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, which is the accumulated future rewards from the current state s_t. The future rewards need to be multiplied by the discount factor to reduce their influence.

However, we can't exactly know the future states when executing an action. So we need to consider all the possible situations and get the expectation of the long-term returns based on the state transition probability. Value function can represent the expectation of the long-term returns, which can be divided into two categories:

(1) state value function $v_\pi(s)$: the expectation of the long-term returns from the current state s under the policy π.

(2) state-action value function $Q_\pi(s, a)$: the expectation of the long-term returns after executing an action a at the current state s, then following the policy π.

They can be expressed as the recursive forms, shown in Eq. 2.1 and Eq. 2.2, which are called as Bellman Equation [60].

$$v_\pi(s_t) = \sum_{a_t} \pi(a_t|s_t) \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t)[r_t + \gamma v_\pi(s_{t+1})]. \tag{2.1}$$

$$Q_\pi(s_t, a_t) = \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \sum_{a_{t+1}} \pi(a_{t+1}|s_{t+1})[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})]. \tag{2.2}$$

2.2.2 Policy iteration

Policy iteration [60] is a basic method to obtain the optimal policy in the dynamic programming. It is initialized from a random policy and value, and alternately perform policy evaluation and policy improvement until the convergence of policy.

In the policy evaluation, the value of each state under the current policy is calculated. The values of all the states are obtained by using the iteration method shown in Eq. 2.3 [60].

$$v_\pi^k(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[r + \gamma v_\pi^{k-1}(s')], \tag{2.3}$$

where k is the iteration times. When $v_\pi(s_t)$ converges, we can get the values of all the states under the current policy π.

Next, the policy needs to be improved according to these state values. For each state, the state-action value is calculated by executing all the possible actions [60]:

$$Q_\pi(s,a) = \sum_{s'} P(s'|s,a)[r + \gamma v_\pi(s')], \tag{2.4}$$

and then the policy is improved by choosing the action which has the maximum state-action value [60]:

$$\pi'(s) = \arg\max_a Q(s,a). \tag{2.5}$$

For the new policy, the values of all the states need to be updated by Eq. 2.3. After that, the policy is updated again by Eq. 2.4. This process is repeated until the policy converges.

2.2.3 Value iteration

Policy iteration spends most of the computational time on the policy evaluation. Especially for some complex problems, this procedure is really time comsuming. In order to promote the computational efficiency, value iteration [60] is introduecd.

For each state, we calculate the state value after executing every possible action and choose the maximum value as the state value of this state, as expressed in Eq. 2.6 [60]. This process is repeated until all the state values converge.

$$v^k(s) = \max_a \sum_{s'} P(s'|s,a)[r + \gamma v^{k-1}(s')], \tag{2.6}$$

where k is the iteration times.

After all the state values converge, an action which has the maximum state-

action value is chosen for any state [60]:

$$\pi(s) = \arg\max_{a} \sum_{s'} P(s'|s,a)[r + \gamma v(s')]. \tag{2.7}$$

Therefore, value iteration selects the action according to the state value, while policy iteration selects the action according to the current policy.

2.2.4 SARSA and Q-learning

Policy iteration and value iteration require the state transition probability of the environment, so they can only solve the problem whose medel is already known. However, in many practical problems, we can't get the environment model. Thus, model-free methods are proposed to solve these problems.

SARSA and Q-learning are two model-free Temporal Difference (TD) methods which use the current reward and value evaluation of the next state to compute the state-action value.

The update of the state-action value of SARSA [60] is expressed in Eq. 2.8.

$$Q(s,a) \leftarrow Q(s,a) + \eta[r + \gamma Q(s',a') - Q(s,a)], \tag{2.8}$$

where action a is taken at the current state s on the basis of $\epsilon - greedy$ policy and r is the reward observed from the environment; at next state s', action a' is chosen according to the same $\epsilon - greedy$ policy; the difference between the target value and evaluated value calculated by the TD method is utilized to update the Q-value of the current state-action pair; η is the learning rate.

$\epsilon - greedy$ policy aims to solve the exploration and exploitation problem in reinforcement learning. The agent has ϵ probability to produce a random action to encourage exploration and $1 - \epsilon$ probability to exploit the current optimal policy. With the training process, the value of ϵ will decrease gradually to seek

a good trade-off between the exploration and exploitation.

The update of the state-action value of Q-learning [60] is expressed in Eq. 2.9.

$$Q(s,a) \leftarrow Q(s,a) + \eta[r + \gamma \max_{a'} Q(s',a') - Q(s,a)], \tag{2.9}$$

where action a is taken at the current state s on the basis of $\epsilon - greedy$ policy; action a' that could get the maximum Q-value is chosen at the next state s' to calculate the target Q-value.

Both of these two methods update the Q-value of one state-action pair in the table after observing a transition. SARSA uses the real action of the next state to evaluate the value, while Q-learning uses the action which has the maximum value but it may not actually execute this action at the next state. Therefore, SARSA is an on-policy method and Q-learning is an off-policy method.

2.2.5 Deep Q-Network

SARSA and Q-learning use a table to store the value of every state-action pair, so they can only solve the problems with finite state and action space. For the probelms which have finite action space but infinite state space, it is suitable to use the neural network to estimate the value of performing each action under a specific state. This approach is known as Deep Q-Network (DQN) [1], extended from the Q-learning.

DQN introduces two important improvements: replay memory and target network [1]. Replay memory is used to store the transitions produced by the agent when exploring the environment. A mini-batch of transitions are sampled randomly from the replay memory at each network update. So replay memory can disrupt the correlation of transitions during update and promote the data efficiency. Target network with the same architecture as the behavior networks is used to calculate the target value. The parameters of target network are up-

dated very slowly, which can reduce the fluctuation problem and enhance the learning stability.

The objective function of DQN is shown in Eq. 2.10 [1].

$$L(\theta) = \sum_{i=1}^{N} (r_i + \gamma \max_{a'} Q(s_{i+1}, a'|\theta') - Q(s_i, a_i|\theta))^2, \tag{2.10}$$

where θ' represents the parameters of the target network and θ represents the parameters of the bahavior network.

The main procedures of DQN are as follows:

(1) Select an action a_t according to the $\epsilon - greedy$ policy at the current state s_t.

(2) Execute the action a_t to the environment, and obtain the next state s_{t+1} and reward r_t.

(3) Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay memory.

(4) Use gradient descent to update θ based on the mini-batch samples from the replay memory.

2.2.6 Policy gradient

DQN is a reinforcement learning algorithm by estimating the value function, which aims to find the action with the optimal value. However, DQN can only be applied to solve the problems with discrete action space. If the action is continuous, other reinforcement learning algorithms applicable to the continuous action space should be used. Policy gradient is a foundamental method which directly calculate the update direction of the policy.

The objective of reinforcement learning is to find the policy which can maximize

the expectation of the long-term returns:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\boldsymbol{\tau}\sim\pi(\boldsymbol{\tau})}[R(\boldsymbol{\tau})], \tag{2.11}$$

where $\boldsymbol{\tau}$ is a trajectory obtained from the interaction with the environment by using the policy π; $R(\boldsymbol{\tau})$ represents the overall return of the trajectory $\boldsymbol{\tau}$.

We represent the expectation of the trajectory return J as a function of policy parameter θ:

$$J(\theta) = \mathbb{E}_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})}[R(\boldsymbol{\tau})] = \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \pi_\theta(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau}. \tag{2.12}$$

The gradient of $J(\theta)$ is expressed as follows:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \nabla_\theta \pi_\theta(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau} = \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \pi_\theta(\boldsymbol{\tau}) \nabla_\theta \log \pi_\theta(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau} \\ &= \mathbb{E}_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})}[\nabla_\theta \log \pi_\theta(\boldsymbol{\tau}) R(\boldsymbol{\tau})], \end{aligned} \tag{2.13}$$

where

$$\begin{aligned} \nabla_\theta \log \pi_\theta(\boldsymbol{\tau}) &= \nabla_\theta \log \Big[P(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t) \Big] \\ &= \nabla_\theta \Big[\log P(s_0) + \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{T} \log P(s_{t+1}|s_t, a_t) \Big] \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \end{aligned}$$

Then the policy gradient [61] is expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \big(\sum_{t=0}^{T} r_t \big) \Big]. \tag{2.14}$$

Since the action a_t can only affect the returns after time t, only the rewards

after time t should be included. So Eq. 2.14 is modified as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})}\Big[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\big(\sum_{t'=t}^{T} r_{t'}\big)\Big]. \tag{2.15}$$

Chapter 3

Continuous control for robot based on Deep Deterministic Policy Gradient

In chapter 2, we have introduced some basic reinforcement learning algorithms, including the value-based method, like DQN for the discrete action space, and policy gradient method for the continuous action space. Policy gradient outputs the probability distribution over the action and the policy can only be updated after the whole episode. Hence, Google DeepMind proposed an improved policy gradient method, which is known as deep deterministic policy gradient (DDPG) [10] applicable to the continuous control tasks with infinite state and action space. DDPG uses actor-critic structure and outputs a deterministic action rather than the probability distribution of the action. The actor network produces a specific action based on the current state while the critic network evaluates the Q-values of the state-action pairs. DDPG also adopts the two improvements of DQN, target networks and replay memory, to improve the stability and convergence of the network.

DDPG needs a replay memory to store the previous experiences. Some work has

been done to improve the structure of the replay memory. Google DeepMind developed prioritized experience replay [6] to replay the significant transitions more frequently to guarantee more effective learning of the agent. OpenAI proposed the hindsight experience replay [25] to solve the sparse reward problem and avoid the complicated reward engineering. We also use these two methods in our simulation.

In this chapter, we firstly train a redundant manipulator to reach any given position by using DDPG and hindsight experience replay with different strategies, from which we can see that the agent can effectively learn by only receiving a simple binary reward. Then we train the redundant manipulator by using DDPG with a shaped reward. We propose a *future* and *random* strategy to relabel some additional goals combined with the shaped reward to artificially produce some new transitions, which can improve the sample efficiency. The simulation results show that the *random* strategy with the shaped reward could achieve the highest success rate. Next, we use DDPG with priority replay memory to train a SCARA robot and a mobile robot to track some trajectories. Two training strategies, random referenced state initialization and early termination are introduced to solve the exploration problem and make the robots learn effectively from the referenced trajectories.

3.1 Deep Deterministic Policy Gradient

The objective of reinforcement learning is to learn a policy that can maximize the expectation of the accumulated future rewards from the start distribution [60]:

$$J = \mathbb{E}_{s_n \sim \epsilon, r_n, a_n \sim \pi}[R_1], \tag{3.1}$$

where state s_n is observed from the environment ϵ, action a_n is sampled from the policy π, and R_1 is the accumulated future rewards from the initial state s_1.

The action-value function is expressed as follows:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{n>t}\sim\epsilon, r_{n\geq t}, a_{n>t}\sim\pi}[R_t|s_t, a_t],$$
$$R_t = \sum_{n=t}^{T} \gamma^{n-t} r(s_n, a_n),$$

which describes the expected value of the accumulated future rewards after taking an action a_t at state s_t and then following the policy π; where r is the instant reward at current time step and γ is the discounted factor.

The function can be written in a recursive format that is the Bellman equation [60]:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim\epsilon, r_t}\big[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1}\sim\pi}[Q^{\pi}(s_{t+1}, a_{t+1})]\big]. \tag{3.2}$$

For the deterministic policy, Eq. 3.2 can be rewritten as [10]:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim\epsilon, r_t}\big[r(s_t, a_t) + \gamma Q^{\mu}\big(s_{t+1}, \mu(s_{t+1})\big)\big], \tag{3.3}$$

where μ is the deterministic policy function.

The action-value and policy is estimated by the deep neural network which has the ability to approach any complex and nonlinear function. Actor and critic are the two neural networks used to choose the action and estimate the action-value, respectively. The parameters of the actor and critic are updated during the training process. The update objective of critic aims to make accurate estimations for the Q-values by minimizing the mean square of TD errors which are the differences between the evaluated Q-values and the target Q-values:

$$L(\theta^{Q}) = \mathbb{E}_{s_t\sim s^{\vartheta}, a_t\sim\vartheta, r_t}\Big[\big(Q(s_t, a_t|\theta^{Q}) - y_t\big)^2\Big], \tag{3.4}$$

where θ^{Q} is the parameters of the critic, including the weights and biases; ϑ is a stochastic policy that may be different from the current policy; s^{ϑ} is the state

distribution under the policy ϑ; $y_t = r(s_t, a_t) + \gamma Q\big(s_{t+1}, \mu(s_{t+1})\big)$ is the target Q-value.

Usually, learning the action-value function directly by using the neural network can't guarantee the convergence and may tend to be unstable. In order to exploit the deep neural networks as the function approximators effectively, a replay memory and seperate target networks need to be introduced as DQN.

As we know, most machine learning algorithms have the assumption that the training data are independent and identically distributed. However, in the reinforcement learning, the samples which are used to update the networks are produced by exploring sequentially in the environment. Thus, the above assumption doesn't hold any more since the samples have correlation with each other. Hence, a replay memory is introduced to solve this problem. The replay memory can be regarded as a cache with a specific capacity. The transitions (s_t, a_t, r_t, s_{t+1}) generated by the agent's exploration to the environment are stored in the replay memory. If the replay memory is full, old transitions will be removed to enable the new transitions to be stored. During training, a mini-batch of transitions sampled from the replay memory are fed into the neural networks. This random sampling can reduce the correlations among the transitions to improve the learning stability. Moreover, the transitions in the replay memory can be used for multiple times, which helps to promote the data efficiency. Due to the replay memory, DDPG is an off-policy reinforcement learning algorithm since the transitions utilized to update the networks may not be produced by the current policy.

If the target Q-value y_t is calculated by the same critic network, the critic may be difficult to converge. So a target critic $Q'(s, a|\theta^{Q'})$ and a target actor $\mu'(s|\theta^{\mu'})$ are introduced to calculate the target Q-value. The target networks have the same architectures with the evaluated networks and their parameters are updated by a soft method: $\theta' \leftarrow \varepsilon\theta + (1-\varepsilon)\theta'$ with $\varepsilon \ll 1$. This means the parameters of the target networks change very slowly, which can greatly

improve the learning stability. The soft update of target networks can make the reinforcement learning closer to the supervised learning which has a fixed label for each sample and thus has robust solutions. So the introduction of Q' and μ' can make y_t stable, which aims to train the critic network without divergence. Then the target Q-value can be expressed as [10]:

$$y_t = r(s_t, a_t) + \gamma Q'\big(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}\big), \tag{3.5}$$

where $(s_t, a_t, r(s_t, a_t), s_{t+1})$ is a transition from the replay memory.

The action chosen at state s_{t+1} is determined by the target actor μ'. The Q-value of the state s_{t+1} and action $\mu'(s_{t+1})$ is calculated by the target critic Q'. The overall loss of the critic is obtained by determining the expected losses of the transitions sampled from the replay memory as shown in Eq. 3.4.

The update objective of the actor is to maximize the expectation of accumulated reward J or Q, whose gradient can be derived by using the chain rule [10]:

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim s^\vartheta}\big[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\big] \\ &= \mathbb{E}_{s_t \sim s^\vartheta}\big[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}\big], \end{aligned} \tag{3.6}$$

where s_t are sampled from the replay memory. The gradients are computed according to the average Q-value of the sampled states from the replay memory and the corresponding actions are obtained from the current policy.

Exploration is a major challenge of reinforcement learning with continuous action space.The exploration of DDPG can be considered separately from the leaning process. So some noises generated from a specific process can be added to the policy μ in order to encourage the exploration. Then the policy function can be written as:

$$\mu_n(s_t) = \mu(s_t|\theta^\mu) + \omega_t, \tag{3.7}$$

where ω is a noise process.

The noise process can be the Ornstein-Uhlenbeck process, which is a kind of temporally correlated noise aiming to realize good exploration in the physical environment with momentum. Its generation fomula is:

$$dx_t = \sigma_1(\eta - x_t) + \sigma_2 W_t, \tag{3.8}$$

where x_t is the value we want to generate at time t; η is the expectation of the random variable; W_t can be a random function; σ_1 and σ_2 are two parameters of the random process; dx_t is the change of value and the true sampled value equals to the value of the last time step plus the change.

Figure 3.1 shows the framework of DDPG. A target actor and a target critic are used to calculate the target Q-value. The critic which can evaluate the performance of the actor is updated by minimizing the mean square of TD errors. The policy gradient to update the actor is calculated with respect to the Q-function.

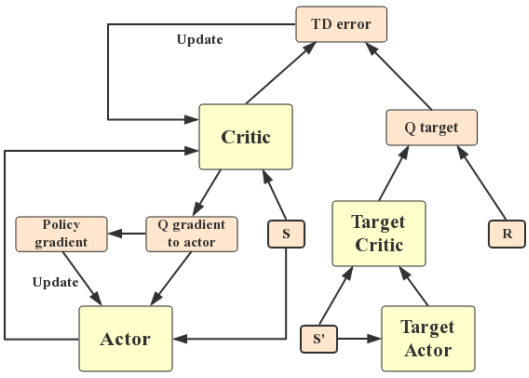

Figure 3.1: Framework of DDPG

The pseudocode of DDPG is presented in Algorithm 1 [10], where the expectation of actor gradient and critic gradient are approximated by the mean of the sampled transitions.

The optimization algorithm we use for the update of network parameters is

Algorithm 1 DDPG

Initialize:
Critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with parameters θ^Q and θ^μ randomly
Target network Q' and μ' with parameters $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Replay memory R with capacity C
for *episode* $= 1, \ldots, M$ **do**
  Obtain the initial state s_0
  for $t = 1, \ldots, T$ **do**
    Get an action $a_t = \mu(s_t|\theta^\mu) + \omega_t$ based on the current policy and exploration noise
    Perform the action a_t to the environment
    Get the reward r_t and the next state s_{t+1} from the environment
    Store the transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a mini-batch of N transitions (s_n, a_n, r_n, s_{n+1}) $(n = 1, \ldots, N)$ randomly from R
    Calculate the target Q-value for each transition according to Eq. 3.5
    Calculate the gradient of critic loss, g^Q with respect to θ^Q and update the critic network:
    $g^Q = \nabla_{\theta^Q} L(\theta^Q) \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_{\theta^Q} \left[\left(Q(s_n, a_n|\theta^Q) - y_n \right)^2 \right]$
    Calculate the gradient of actor loss, g^μ with respect to θ^μ and update the actor network:
    $g^\mu = \nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{n=1}^{N} \left[\nabla_a Q(s, a|\theta^Q)|_{s=s_n, a=\mu(s_n|\theta^\mu)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_n} \right]$
    Update the target networks by using the soft replace:
    $\theta^{Q'} \leftarrow \varepsilon\theta^Q + (1 - \varepsilon)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \varepsilon\theta^\mu + (1 - \varepsilon)\theta^{\mu'}$ $(\varepsilon \ll 1)$
  end for
end for

Adam (Adaptive moment estimation) [62]. This method is computationally efficient and has little memory requirements. It can adjust the learning step adaptively for the different parameters. The concrete procedures are as follows [62]:

$$g_t = \nabla_\theta L_t(\theta_{t-1})$$

$$m_t = c_1 m_{t-1} + (1 - c_1) g_t$$

$$n_t = c_2 n_{t-1} + (1 - c_2) g_t^2$$

$$m'_t = \frac{m_t}{1 - c_1^t}$$

$$n'_t = \frac{n_t}{1 - c_2^t}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{m'_t}{\sqrt{n'_t + \epsilon}}.$$

3.2 Priority replay memory

In the last section, we have introduced the replay memory used to store the transitions produced by the agent's exploration to the environment. The replay momory can reduce the correlation of samples and improve the data efficiency since the agent could reuse the previous transitions. The transitions are sampled uniformly from the replay memory at each update, so all the transitions have the same possibility to be sampled without considering their significance. However, the difficulty of learning each transition is different, which leads to the different improvement after learning different transitions. Some transitions are easy to learn and learning from these easy transitions may not get much benefit. While some transitions are relatively difficult to learn, so the model may have obvious promotion after learning from these difficult transitions. If every transition is treated equally, the agent may waste much time on those easy transitions, which results in the inefficient learning.

Priority replay memory [6] could solve the above-mentioned problem effectively. Each transition has a weight according to its performance based on the current parameters. The sampling probability of each transition has a relation with its weight. The transition with worse performance has a higher weight and can be sampled with a higher probability, and vice versa.

The idea of the priority replay memory is to increase the sampling probability of more significant transitions. The significance of a transition is usually measured by its TD error. The higher the TD error, the higher probability the transition will be sampled with [6].

The TD error is a reasonable measurement of the significance, but it may change as the network parameters are updated during the training process. Hence, the

following situation may occur: the TD error of a transition is high when it is firstly stored in the replay memory, and its TD error may decrease as the network parameters are updated constantly, but the transition still has a high probability to be sampled. A reasonable approach for this problem is to recalculate the TD errors of the transitions which have been sampled from the the replay memory, but it may need a lot of extra computation.

Since the measurement method is not totally reliable, we need to introduce extra improvement. We hope that the transitions with relatively higher TD errors should have higher probability to be sampled and the transitions with low TD errors should also have some probability to be sampled. Hence, the sampling probability of each transition is calculated by [6]:

$$P(i) = \frac{Pr_i^\alpha}{\sum_k Pr_k^\alpha}, \tag{3.9}$$

where Pr_i is the TD error of each transition; α is a hyperparameter that can adjust the importance of TD error when sampling the transitions; when $\alpha = 1$, the transitions are sampled directly according to their TD error; when $\alpha < 1$, the influence of TD error is weaken and transitions with small TD errors could have larger probability to be sampled; when $\alpha = 0$, it becomes uniform sampling as the case without considering the priorities.

For the ordinary replay memory, each transition has the same probability to be sampled. The priority replay changes the probability distribution, which may introduce bias when estimating the expectation of gradients. In order to make the estimation unbias, we introduce an importance sampling weight to each sampled transition.

$$\begin{aligned}
\mathbb{E}_{i\sim RM}[\nabla J] &= \mathbb{E}_{i\sim PRM}\Big[\frac{P_{RM}(i)}{P_{PRM}(i)}\nabla J\Big] \\
&= \mathbb{E}_{i\sim PRM}\Big[\frac{\frac{1}{N}}{P_{PRM}(i)}\nabla J\Big] \\
&= \mathbb{E}_{i\sim PRM}\Big[\frac{1}{N\cdot P_{PRM}(i)}\nabla J\Big],
\end{aligned}$$

where $P_{RM}(i)$ is the sampling probability of a transition from the replay memory and $P_{PRM}(i)$ is the sampling probability from the priority replay memory; N is the total number of transitions in the replay memory. By multiplying the weight $\frac{1}{N \cdot P_{PRM}(i)}$ to the gradients, the update will become unbias.

If we just use this weight, it has no difference with the ordinary replay memory, so we should introduce another hyperparameter β to adjust the influence of the prioritized experience replay. Then the weight becomes [6]:

$$w_i = \Big(\frac{1}{N \cdot P_{PRM}(i)}\Big)^{\beta}, \tag{3.10}$$

when $\beta = 1$, the update is equivalent to the ordinary replay memory; when $\beta < 1$, prioritized experience replay may have some impact. Furthermore, we usually normalize the weight by $\frac{1}{\max_i w_i}$, so the maximum value of the nomalized weight is 1, which could confine the weights in a reasonable range and avoid large updates of parameters.

Many theoretical derivation is based on the ordinary replay memory. The priority replay memory mainly aims to speed up the training, but the convergence can't be guaranteed. Hence, we hope the priority replay memory could become the ordinary replay memory eventually. At the beginning of training, β can be assigned to a number less than 1. With the increase of training iteration, β should become larger and finally approximate to 1. In this way, we could promote the training speed without influencing the convergence of the networks.

The main procedures of priority replay memory are summarized as follows.

(1) Store the transition in the replay memory with its priority measured by the TD error.

(2) Sample transitions based on the probabilities calculated by Eq. 3.9.

(3) Multiply the importance sampling weight w_i to the gradient of each sampled transition when updating the networks.

(4) Let β gradually approximate to 1 with the training process.

However, if we directly calculate the sampling probabilities of the transitions, a lot of computation is needed since the capacity of the replay memory may be very large. In order to efficiently sample the transitions from the distribution, we could use a 'sum-tree' structure to store the transition priorities. The structure of 'sum-tree' is shown in Fig. 3.2. The transition priorities are stored in the leaf nodes. The internal nodes store the intermediate sum of the priorities and the parent node contains the sum of all the transition priorities p_{total}. When the priority of a transition is updated, all the corresponding internal nodes and parent node will be updated. When a mini-batch of size m needs to be sampled, the range $[0, p_{total}]$ will be equally divided into m interval. Next, a number is uniformly sampled from each interval, and then it starts from the parent node and compares with the following internal nodes until getting to a leaf node which is the transition we will sample. In this way, we could update the priorities and sample the transitions more effectively.

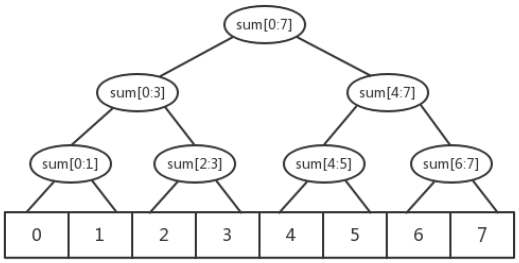

Figure 3.2: Structure of sumtree

3.3 Hindsight experience replay

We have known that the replay memory can reduce the correlation among the samples and promote the data efficiency to enable more effective learning of the agent. In this section, we will introduce another momory structure, hindsight

experience replay (HER) [25] which mainly aims to deal with the sparse rewards in the reinforcement learning.

Sparse reward problem is a major challenge of reinforcement learning. In order to make the agent learn effectively from the environment, we usually need to design the reward function accurately. However, sometimes designing a proper reward function is difficult, which results in a bad learning effect. Hindsight experience replay is a novel technique proposed by OpenAI to solve the problem of sparse rewards. HER enables the agent to learn efficiently even from the sparse and binary rewards and avoides the sophisticated reward engineering. It could be combined with any off-policy reinforcement learning, such as DQN and DDPG.

The motivation of hindsight experience replay is similar to the curriculum learning which is a kind of incremental learning strategy. Sometimes, the real goal is very difficult to realize, but we could decompose the real goal into some small goals, or wo could reduce the difficulty of the tasks. Even if we can't complete the difficult tasks directly, we could firstly complete the relatively easier tasks and gradually we will know how to achieve those difficult goals.

The main idea of hindsight experience replay is to replace the real goal with the achieved goals for each transition, which is equivalent to reduce the task difficulty. When the agent gets to the state s_T after a series of actions, it doesn't obtain any positive reward if the real goal is g. Even though the agent can't get to the goal state g, it can reach another state s_T. This experience also deserves to be stored in the replay memory, becuase it indicates how to achieve state s_T and the next goal may be exactly s_T. One obvious advantage of human compared with the trained agent is that human can still get some meaningful experiences even if they can't get any explicit feedback. By using hindsight experience replay, the agent could behave more like human and learn the policy more intelligently.

We hope to train the agents which could achieve different goals, so we use the

similar method with the universal value function approximator [63]. The actor and critic network take the state $s \in S$ and the goal $g \in G$ as input. A state-goal pair is sampled from a specific distribution at the beginning of each episode and the goal remains fixed during each episode. The reward received by the agent at each time step is $r_t = r_g(s_t, a_t)$, which is related to the goal. It has been proved that the policy trained by this method could generalize to the other unseen state-goal pairs.

It is assumed that given any state s, a goal corresponding to this state can be found, which can be described by a mapping $m : S \to G$. We should note that the goal may just specify some features of the state, so the mapping may be not always an identity, but has some other formats, eg. $m((x, y)) = x$.

It is very common to design such a reward: the agent will get a negative one reward as long as the goal is not achieved or a zero reward when the goal is achieved at each time step. It can be represented as follows:

$$r_g(s_t, a_t) = -[f_g(s_{t+1}) = 0], \tag{3.11}$$

where $f_g(s)$ is a function that indicates whether the goal is realized. If the agent gets to the goal state, $f_g(s)$ is equal to 1, or it is equal to 0. However this reward can't work well in practice since it is sparse and not informative. Next, we will introduce the detail of HER which can train the agent successfully only by using the sparse reward in Eq. 3.11.

Firstly, the agent explores the environment and gets a series of transitions. All the transitions are temporarily stored in a replay buffer. It should be noted that all the states of these transitions include the real goal. After an episode, each transition with the real goal in the replay buffer is sent to the replay memory and then it will also be stored in the replay memory for multiple times with other additional goals. The additional goals for replay could be the function of the final state of the trajectory $m(s_T)$, or the function of the following states in the same trajectory. We need to recompute the reward for each transition

based on the additional goals. With this modification, some transitions could have the reward different from -1, which makes the learning much simpler and more effective. This approach enables the agent to learn how to achieve any goal which may never be observed during the interaction process.

The pseudocode of hindsight experience replay is shown in Algorithm 2 [25].

Algorithm 2 Hindsight experience replay

Initialize:
    An off policy reinforcement learning algorithm $\mathbb{A}$
    A strategy $\mathbb{S}$ for choosing additional goals for replay
    A replay memory R
for *episode* $= 1, \ldots, M$ **do**
    Generate a initial state s_0 and a goal g randomly
    for $t = 1, \ldots, T$ **do**
        Get an action $a_t \leftarrow \pi(s_t||g)$ based on the current policy from $\mathbb{A}$
        Implement the action a_t and then obtain the reward r_t and the next state s_{t+1}
        Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in an replay buffer temporarily
    end for
    for $t = 1, \ldots, T$ **do**
        Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in R
        Get some additional goals G for replay by using strategy $\mathbb{S}$
        for $g' \in G$ **do**
            Recompute the reward r'_t based on the goal g'
            Store the transition $(s_t||g', a_t, r'_t, s_{t+1}||g')$ in R
        end for
    end for
end for
for $i = 1, \ldots, N$ **do**
    Sample a mini-batch of transitions from R randomly
    Update network parameters of $\mathbb{A}$ by using the sampled mini-batch
end for

3.4 Multi-goal reinforcement learning

In this section, we focus on the multi-goal reinforcement learning task, which means the goal in each training episode is different. Even though we only

care about one specific goal, training on the mutiple goals could get better performance than training on the single goal and also enables the agent to achieve any goal.

The simulation environment is a redundant manipulator with seven degree of freedom (Fig. 3.3). The common space manipulator has six degree of freedom to move along and rotate around the three axes. The redundant manipulator has another extra degree of freedom, so it can arrive at a certain position in the three-dimensional space with infinite number of configurations. The training objective is to enable the end-effector of the redundant manipulator to reach any given goal position within its work envelope from a random initial position during each episode. Since the redundant joint makes the manipulator have enough flexibility to reach any position in its work envelope and avoid the singular positions, we can indicate any position within the work envelope as the goal position.

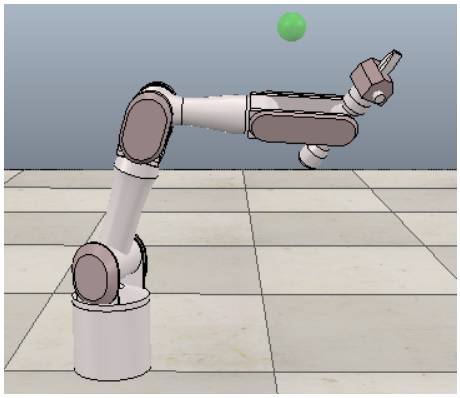

Figure 3.3: Redundant manipulator

Deep reinforcement learning is employed to make the high-level decision for the redundant manipulator to move to the goal position. Firstly, we train the redundant manipulator by using DDPG with HER and then we train it again by using DDPG with the shaped reward. We propose the $future$ and $random$ strategy to generate some additional transitions by referring to the idea of HER. The performance comparisons of different methods are presented.

3.4.1 Architecture of the actor and critic

The input of the actor is the observation vector s and the outputs of the actor are the changes of the end-effector position in the space relative to the current positions, $\Delta \boldsymbol{p} = [\Delta x, \Delta y, \Delta z]$. The changes of positions need to be converted to the absolute positions of the end-effector, $\boldsymbol{p}$, and then the inverse kinematics of the manipulator is applied to get the joint positions. We can utilize the robot simulator V-REP to solve the inverse kinematics of the redundant manipulator by using Pseudo Inverse method and damped least squares method (DLS).

We only consider the high-level control by deep reinforcement learning to decide on the target position of the end-effector at each time step. The policy function can be expressed as:

$$\Delta \boldsymbol{p} = f(\boldsymbol{s}). \tag{3.12}$$

For DDPG with priority replay memory and without any additional goals, the observation is designed as:

$$\boldsymbol{s} = [x, y, z, dx_g, dy_g, dz_g],$$

where $[x, y, z]$ is the end-effector position; $[dx_g, dy_g, dz_g]$ is the difference between the goal position and the real position of the end-effector in each dimension.

For DDPG with HER and additional goals combined with the shaped reward, the observation needs to include the goal position. Thus, the observation is designed as:

$$\boldsymbol{s} = [x, y, z, dx_g, dy_g, dz_g, x_g, y_g, z_g],$$

where $[x_g, y_g, z_g]$ is the given goal position of an episode.

The architecture of the actor and critic for the redundant manipulator is shown

in Fig. 3.4. For the actor, we use three-hidden-layer fully connected (FC) neural network and each hidden layer has 100 neurons. All the hidden layers adopt *relu* activation function and the output layer adopts *tanh* activation function. $\boldsymbol{C}$ is a scaled vector to constrain the range of the position change.

For the critic, we also use three-hidden-layer FC network and each hidden layer has 100 neurons. The hidden layers adopt *relu* activation function and the output layer adopts linear activation function.

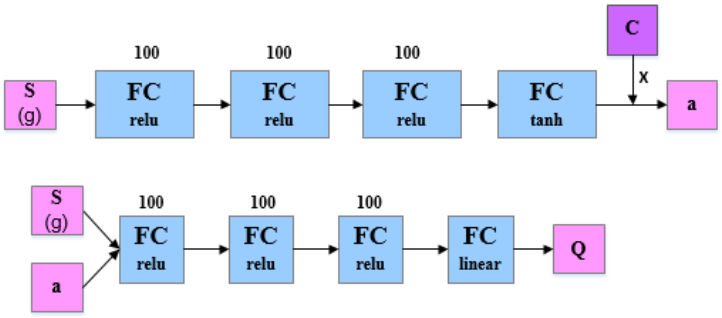

Figure 3.4: Architecture of actor and critic

3.4.2 Training results of redundant manipulator

In this part, we will show the training results of DDPG with the sparse reward and DDPG with the shaped reward respectively for the redundant manipulator.

At the beginning of each episode, a random goal position within the reachable region of the end-effector is chosen and the end-effector position is also randomly given with any configuration. If the end-effector reaches the goal position within a small distance threshold, it completes the task and the episode ends. If the end-effector moves beyond its feasible region or it doesn't get to the goal position within the maximum steps in an episode, it fails to finish the task and the episode ends. Then it will continue to be trained for the next episode with a different goal.

Training results of DDPG with hindsight experience replay

Firstly, we train the redundant manipulator by using DDPG with HER introduced in section 3.3. We don't need to design the reward function accurately since HER is powerful to deal with the problems with the sparse rewards. Thus, we only use a binary reward: if the end-effector arrives at the goal position, it will get a zero reward, or it will get a negative one reward. At the beginning of each episode, a random goal position is chosen and the manipulator is also initialized to a random position.

We train the redundant manipulator by using DDPG with HER for 200 epochs. Each epoch has 50 cycles and each cycle includes 16 episodes to run the policy and collect the transitions. After running a cycle with 16 episodes, the network parameters will be updated for 40 times. Since the optimization of networks is only performed after running a certain number of episodes rather than at each time step during an episode, it is reasonable to sample more transitions fed into the networks at each update. Hence, we use a replay memory which can contain 10^6 transitions and sample a mini-batch of transitions with size 128 at each optimization step.

In the section 3.3, we have mentioned that different strategies can be used to choose the additional goals for replay. Here, we employ two strategies, $final$ and $future$. $final$ means the final state of an episode is treated as the goal of the transitions in this episode. $future$ means k random states observed after the transition in the same trajectory are chosen as the additional goals for this transition. If the end-effector moves out of the feasible region, the final state of this episode can not be chosen as the additional goal since it is not a expected state.

Firstly, we train the redundant manipulator by using HER with the strategy $final$, so each transition will be stored in the replay memory twice with the real goal and achieved goal (i.e. the final state of the episode). Then, the redundant

manipulator is trained again by using the $future$ strategy with different k. If $k = 2$, two subsequent states are chosen to be the additional goals for the transition and each transition will be stored in the replay memory for three times with different goals. If $k = 4$, four subsequent states are chosen and each transition will be stored in the replay memory for five times with different goals.

We record the success rate of the manipulator in each epoch. The comparison of success rate of DDPG with HER by using different strategies is shown in Fig. 3.5, where we can see that all the strategies could achieve approximately 80% success rate after some epochs by the multi-goal training method. The $final$ strategy could make the agent learn faster compared with $future$ strategy. The $future$ strategy with $k = 2$ also has a faster learning speed than $future$ strategy with $k = 4$. It may be explained as: increasing the value of k could decrease the percentage of the normal transitions in the replay memory, which may degrade the performance.

Hence, we can conclude that DDPG with HER could achieve good performance in the manipulator multi-goal task even if we just use a simple binary reward, which proves that HER can effectively solve the probelm of sparse rewards.

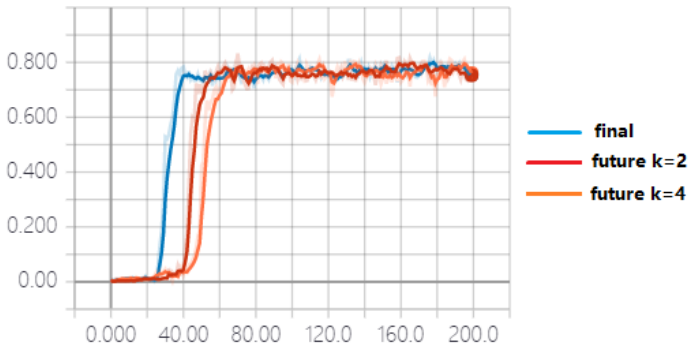

Figure 3.5: Success rate of different strategies

Training results of DDPG with the shaped reward

Next, we train the redundant manipulator by using DDPG with a shaped reward. A reward function which enables the agent to learn effectively from the

environment needs to be designed. The reward function is expressed in Eq. 3.13:

$$r(s_t, a_t) = \lambda \|g - p_t\|^k - \|g - p_{t+1}\|^k, \tag{3.13}$$

where g is the random goal position; p_{t+1} is the position of end-effector after executing action a_t at position p_t; $\lambda \in \{0, 1\}$ and $k \in \{1, 2\}$ are two hyperparameters.

We choose $\lambda = 0$ and $k = 1$, then the reward function becomes:

$$r(s_t, a_t) = -\|g - p_{t+1}\| \tag{3.14}$$

Hence, the reward of every step is determined by the negative distance between the real end-effector position and the goal position after executing an action. If the end-effector moves out of the range, it will receive a negative reward. If the end-effector reaches the position goal, it will receive a positive reward. We also normalize the reward of each time step by a scaling factor to make the agent learn more effectively.

The $future$ and $final$ strategy is used to generate some additional goals in HER, so we could obtain some new transitions by replacing the real goal. It is also an effective method to improve the sample efficiency since we could convert a single transition (s, a, s') into many other transitions by generating the additional goals. We have the curiosity if the similar idea with HER could combine with the shaped reward. Thus, we firstly adopt the $future$ strategy to choose two future states of the current state during an episode as the additional goals. Then we need to recompute the reward according to the next state of the transition and the additional goals by Eq. 3.14. Therefore, for each original transition, we could relabel another two new transitions with different goals. However, this $future$ strategy is limited to sampling the goals from a specific trajectory, which may restrict the diversity of the goals that we can relabel a given transition with. Hence, we propose a *random* strategy which

chooses two random goals within the feasible region as the additional goals. The reward of each transition is recomputed by the same approach as $future$ strategy. The *random* strategy can help the agent to learn how to realize any other goal in addition to the given goal in one episode. By artifically generating some transitions, the agent only need to collect less data, which can reduce the interaction steps with the environment to improve the sample efficiency.

Firstly, we train the redundant manipulator by using DDPG with priority replay memory and without introducing any additional goals. Then we train the manipulator by using the $future$ and *random* strategy combined with the shaped reward, respectively. The training process consists of 100 epochs, each of which contains 50 cycle, and each cycle has 16 episodes. After one cycle, the actor and critic will be updated for 40 times by sampling a mini-batch of transitions with size 128 at each update. We also set the capacity of the replay memory as 10^6.

The average episode rewards of one cycle and success rates of every 5 cycle of these three methods are respectively shown in Fig. 3.6 and Fig. 3.7. We can see that the *random* strategy could achieve 80% success rate for average and the $future$ strategy could achieve 60% success rate for average. However, DDPG with priority replay memory without any additional goals could only get less than 10% success rate. Therefore, we can see that $future$ strategy combined with the shaped reward gets a worse performance than the sparse reward, which may be becasue the shaped reward penalizing inappropriate behaviours could hinder the exploration of the agent. DDPG with the *random* strategy could achieve the highest success rate, which indicates that the agent can learn more effectively by introducing artificially generated transitions with more diverse additional goals.

We save the models trained by different methods and test them for 100 episodes to observe if the agents can reach any given position from a random initial position. The success times of DDPG with both the sparse reward and shaped reward are summarized in Table 3.1, which demonstrates that shaped reward

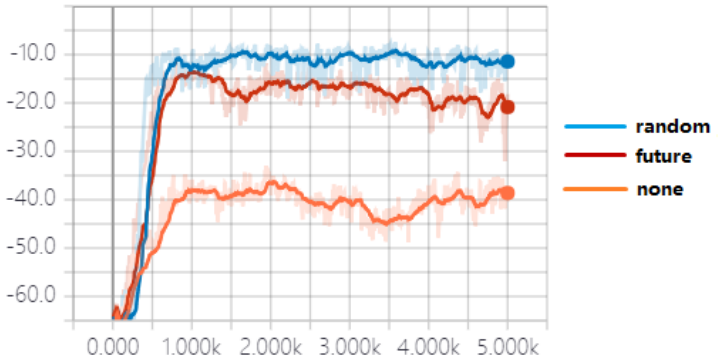

Figure 3.6: Average episode rewards of one cycle

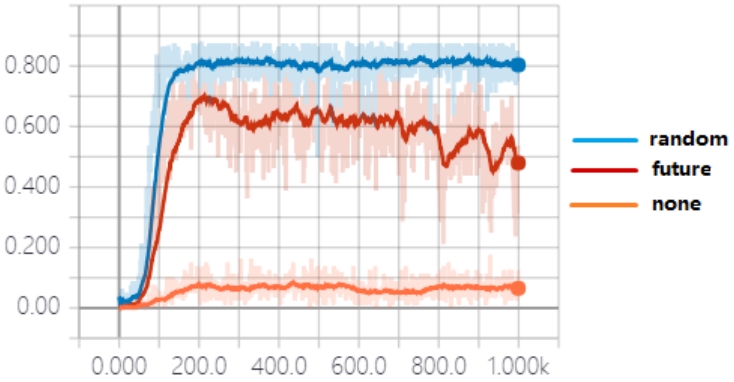

Figure 3.7: Success rates of every 5 cycle

with *random* strategy could achieve the highest success rate.

Table 3.1: Testing results of DDPG

Methods	Success times
Sparse reward with $final$ strategy	79
Sparse reward with $future(k = 2)$ strategy	78
Sparse reward with $future(k = 4)$ strategy	81
Shaped reward with *random* strategy	87
Shaped reward with $future$ strategy	56
Shaped reward without additional goals	9

The hyperparameters for the multi-goal task of redundant manipulator are shown in Table 3.2.

This simulation shows that the manipulator can learn a successful high-level policy to reach any given goal position by the model-free deep reinforcement learning with multi-goal training method. Different methods are tried in order to compare their performance and find the best strategy which can achieve the

Table 3.2: Hyperparameters of DDPG for redundant manipulator

Hyperparameters	Value
Learning rate of actor α_μ	0.001
Learning rate of critic α_Q	0.001
Soft replacement coefficient ε	0.05
Reward discounted factor γ	0.9
Memory capacity C	10^6
Batch size b_n	128
Priority exponent α	0.6
Initial importance sampling weight exponent β_0	0.4
Incremental rate of β	0.001

highest success rate. Therefore, the redundant manipulator task can demonstrate the effectiveness of model-free deep reinforcement learning to the high-level decision-making.

3.5 Tracking control of robots via DDPG

Deep reinforcement learning is mainly applied to make the high-level decision for the robotic tasks. In this section, we focus on the trajectory tracking task of the robots. DDPG with priority replay memory is used to train a SCARA robot and a mobile robot to learn the high-level control policy for the tracking task based on the kinematics. The policy trained by deep reinforcement learning can directly map the observation input to the action output.

3.5.1 Problem Formulation

Firstly, we define a general robotic trajectory tracking problem. The robotic system can be described as:

$$x_{k+1} = F(x_k, u_k), \quad k = 0, 1, \cdots$$

where x_k is the robot state and u_k is the system input; F is a mapping characterizing the system dynamics.

If a reference trajectory, $x_{d,k}$ is given, we need to design a deep reinforcement learning framework to make the position error $e_k \triangleq x_{d,k} - x_k$ approach to zero with the training process.

Next we introduce the models of two robotic systems: SCARA robot and mobile robot.

SCARA robot

SCARA robot (Fig. 3.8) is a widely used manipulator suitable for the plane positioning and vertical assembly. It has three joints, two revolute joints and one prismatic joint, so there are three joint variables $q = [\theta_1\ \theta_2\ d_3]$.

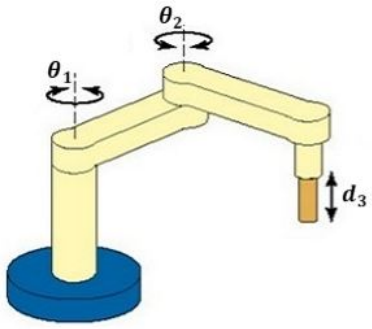

Figure 3.8: SCARA robot

A homogeneous transformation matrix, which is known as the arm matrix, can be constructed to map the tool coordinate to the base coordinate:

$$T_{base}^{tool} = T_0^1 T_1^2 T_2^3 = \begin{bmatrix} \cos(\theta_1 - \theta_2) & \sin(\theta_1 - \theta_2) & 0 & l_1\cos(\theta_1) + l_2\cos(\theta_1 - \theta_2) \\ \sin(\theta_1 - \theta_2) & -\cos(\theta_1 - \theta_2) & 0 & l_1\sin(\theta_1) + l_2\sin(\theta_1 - \theta_2) \\ 0 & 0 & -1 & D - d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where l_1 and l_2 are the lengths of the two links; D is the vertical distance between

the base coordinate frame and the first joint.

$$p(q) = \begin{bmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2) \\ D - d_3 \end{bmatrix}, \tag{3.15}$$

which specifies the end-effector position measured in the base coordinate frame.

The objective is to design a DDPG based learning framework such that the end-effector of the SCARA robot could move along the given trajectory with $e_{x,k} \approx 0, e_{y,k} \approx 0, e_{z,k} \approx 0$.

Mobile robot

Consider a mobile robot (Fig. 3.9) with three wheels, two of which are the driving fixed standard wheels located at the back of the chassis and one is the front caster wheel which can make the mobile robot keep balance and doesnt exert any motion constraint to the mobile robot. Two coordinate frames can be used to describe the motion of the mobile robot. One is the global coordinate frame (X, Y) which is fixed in the world and the other is the local coordinate frame (X_l, Y_l) which is fixed on the mobile robot. The angle between the two coordinate frames is denoted as θ.

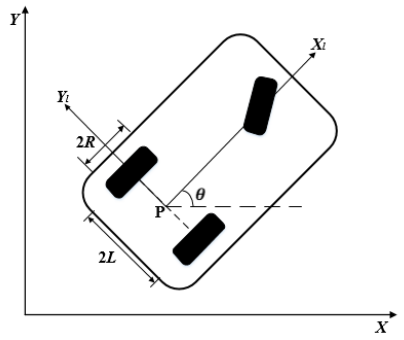

Figure 3.9: Mobile robot

To specify the position of the mobile robot, a point P is chosen on the robot

chassis as its position referenced point. Point P is positioned by the coordinate (x, y) in the global frame. To describe the motion of the mobile robot in terms of the component motions, it is necessary to map the motion along the axes of the gloabl frame to the motion along the axes of the local frame. This mapping is expressed as:

$$\begin{bmatrix} \dot{x}_l \\ \dot{y}_l \\ \dot{\theta}_l \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}. \tag{3.16}$$

By considering the sliding constraint of the fixed standard wheel, $\dot{y}_l = 0$, which means the wheel can't slide orthogonal to the wheel plane, then we can get:

$$-\dot{x}\sin\theta + \dot{y}\cos\theta = 0. \tag{3.17}$$

Denoting the forward velocity $\dot{x}_l$ of the mobile robot as v and the rotation speed $\dot{\theta}$ as w, the kinematics of the mobile robot becomes:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix}. \tag{3.18}$$

The forward velocity and rotation speed of the mobile robot have the following relationships with the linear velocities of the two wheels:

$$v = \frac{1}{2}(v_l + v_r) \qquad w = \frac{v_l - v_r}{2L}.$$

If we consider the linear and angular velocity of the two fixed standard wheels, we can get: $v_l = \omega_l R$ and $v_r = \omega_r R$. Then we can express the angular velocities of the two fixed standard wheels in terms of the forward velocity and rotation

speed of the mobile robot:

$$\begin{aligned} \omega_l &= \frac{v + wL}{R} \\ \omega_r &= \frac{v - wL}{R}. \end{aligned} \tag{3.19}$$

By considering the acceleration of the mobile robot in its local coordinate frame, the dynamic model can be expressed as [64]:

$$\begin{bmatrix} \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} \frac{1}{Rm} & \frac{1}{Rm} \\ \frac{L}{RI} & \frac{-L}{RI} \end{bmatrix} \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}, \tag{3.20}$$

where m and I are the mass and inertia of the mobile robot, respectively; R is the radius of the two fixed standard wheels; L is the half of the distance between the two fixed standard wheels; $\tau = [\tau_1 \ \tau_2 \ ]^T$ is the input torque vector from the motors exerted to the two fixed standard wheels;

The objective is to train the mobile robot to track the given trajectory with $e_{x,k} \approx 0, e_{y,k} \approx 0, e_{\theta,k} \approx 0$.

3.5.2 Network architecture and training strategies

The architecture of the actor and critic for both the SCARA robot and mobile robot is shown in Fig. 3.10. We use FC network with two hidden layers for the actor. The first hidden layer has 300 neurons and the second hidden layer has 100 neurons. All the hidden layers adopt *relu* activation function and the output layer adopts *sigmoid* activation function multiplied by a scaled vector $\boldsymbol{C}$. We also uses two-hidden-layer FC network for the critic with the same number of neurons as the actor.

In order to train the robots to track the given trajectories successfully, we introduce two strategies during the training process, random referenced state initialization and early termination.

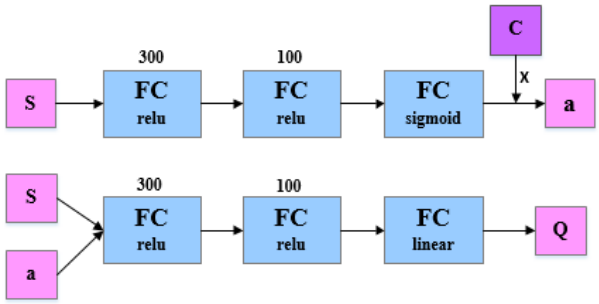

Figure 3.10: Architecture of actor and critic

It is very common to choose a fixed initial state for the agent at the beginning of each episode. For the trajectory tracking task, a simple way is to initialize the robot at the starting point of the trajectory. In this way, the agent have to learn the policy in a sequential approach, firstly learning the actions of the early states, and then learning the actions of the subsequent states. However, if the agent doesn't learn how to reach the early expected states, it has little possibility to arrive at the later expected states, so the training may have little effect on the policy improvement. Moreover, the fixed initial state can result in the exploration problem which is a large challenge of reinforcement learning. The agent can receive a high reward only when it visits an expected state. Then the agent can learn that the state is favorable if a high reward is received after visiting this state. However, it is difficult for the agent to explore to the favorable states by only learning from the previous undesirable experiences. These disadvantages can be mitigated by using the method of random referenced state initialization. It is easy to get any referenced state for the robotic trajectory tracking task since the reference state at any time step can be obtained from the given position and velocity curve. At the beginning of each episode, one time step is sampled within the designated simulation time steps and the corresponding position and velocity can be obtained from the referenced motion curves. Hence, the robot doesn't always begin to learn the policy from the starting state anymore, instead, it could start at any favorable state during an episode. With this method, the agent can immediately encounter the expected states at the early stage of training, even though it doesn't master the policy to

reach the expected states. Therefore, it can alleviate the exploration problem of reinforcement learning by reducing the demand for the agent to discover the promising states by its own.

The training process consists of a certain number of episodes. Usually, each episode ends either after the maximum time steps, or when the specific termination condition has been triggered. For the robotic trajectory tracking task, the training of an episode may terminate when it passes through the last time step. However, in some initial episodes, the agent may take some unexpected actions that could cause large tracking errors. Even if the robot continues to be trained during this episode, it is still difficult for the robot to approach to the referenced trajectory, instead, it may further deviate away from the referenced trajectory. If this phenomenon happens, the data collected in this episode may include many unexpected experiences. Hence, in the early stage of training, the agent may collect many bad experiences and the neural networks will put a lot of effort to learn these unexpected experiences, which could impede the subsequent learning of the agent. By introducing the early termination, the agent could cease to be trained in advance during an episode in case of collecting many undesirable experiences. In our task, we use a counter to record the times of unexpected states of the robot. When the distance beween the real position of the robot and referenced position exceeds a designated value at each time step, the value of counter increases 1. If the value of counter is greater than the upper limit prescribed by us, early termination will be triggered and the episode will end in advance. Once the early termination is triggered, the agent will receive zero reward in the rest of this episode. Early termination could help to discourage the undesirable behaviors and make the agent collect more useful data that contributes to the effective learning.

3.5.3 Trajectory tracking of SCARA robot

In this subsection, we train the SACRA robot described in subsection 3.5.1 to enable its end-effector to move along a given trajectory by using DDPG with priority replay memory.

Design of observation and reward function

The input of the actor is the observation vector $\boldsymbol{s}$ and the output of the actor are the changes of joint variables $[\Delta\theta_1 \ \Delta\theta_2 \ \Delta d_3]$. The observation $\boldsymbol{s}$ at each time step is designed as:

$$\boldsymbol{s} = [\sin(\theta_1), \cos(\theta_1), \sin(\theta_2), \cos(\theta_2), x, y, z, e_x, e_y, e_z], \tag{3.21}$$

where the trigonometric functions of θ_1 and θ_2 are used since the angles are periodic; $[x, y, z]$ denotes the end-effector position; $[e_x, e_y, e_z]$ represents the position errors of the end-effector. We also need to scale some elements of the observation vector to make all of them have the similar range.

After getting the changes of all the joint positions, the new positions of the joints can be obtained and the end-effector will move to a new position. In the real implementation, low-level controllers are needed to connect with the joints. The positions obtained from the reinforcement learning policy are treated as the target positions for the PID controllers at all the joints. The outputs of the PID controllers are the final torques exerted to the joints. The overall control diagram is shown in Fig. 3.11, where the high-level control is implemented by deep reinforcement learning and low-level control can be implemented by PID controllers which also don't require the dynamic model of the system. If the reinforcement learning policy directly operates on the torques, it is difficult to learn a successful control policy due to the complexity. The use of low-level controllers can abstract away the low-level control details from the reinforcement learning policy, which can increase the learning speed and improve the

performance for the robotic control tasks.

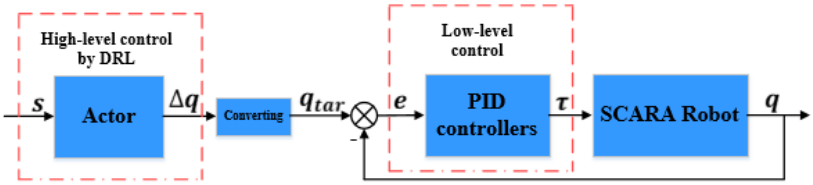

Figure 3.11: Overall control diagram of SCARA robot

We design a normalized reward function which is an exponential function of the position errors of the end-effector as expressed in Eq. 3.22. It can be seen that the reward is in the range of $0 \sim 1$. If there is no error in the position, the reward can get to the maximum value 1.

$$r = exp\big(-2(e_x^2 + e_y^2 + e_z^2)\big). \tag{3.22}$$

Training results of SCARA robot

We firstly choose a set of referenced positions for the three joints and then we can get the referenced trajectory of the end-effector. At the beginning of each episode, a referenced position for each joint is chosen and the SCARA robot is trained from this referenced position.

We train the SCARA robot for 1000 episodes by using DDPG with priority replay memory. The capacity of the replay memory is 50000. Usually, the networks begin to be trained after the replay memory is full. In order to start training as soon as possible, we set that the training starts when 5000 transitions have been stored in the replay memory. After training, we save the trained parameters of the neural networks to some files. During the testing period, the saved parameters are restored from the files, and the SCARA robot moves from $t = 0$ by using the saved parameters of the actor.

Fig. 3.12 shows the change of episode reward. It can be seen that the reward goes up gradually as the training episode increases, which proves the policy of

the agent is improved gradually with the training process.

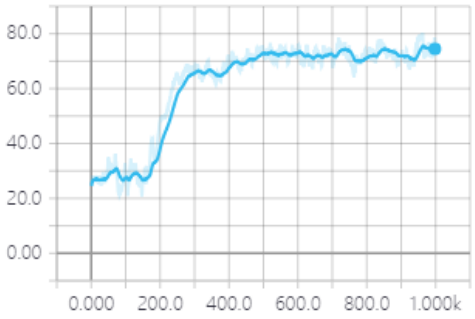

Figure 3.12: Episode rewards of SCARA robot

The trajectory tracking result of the SCARA robot is presented in Fig. 3.13. We observe the real x, y and z position of the end-effector versus time and compare them with the referenced positions (Fig. 3.14 $\sim$ Fig. 3.16). We can see that the real positions only has a little deviation from the referenced ones in three dimensions.

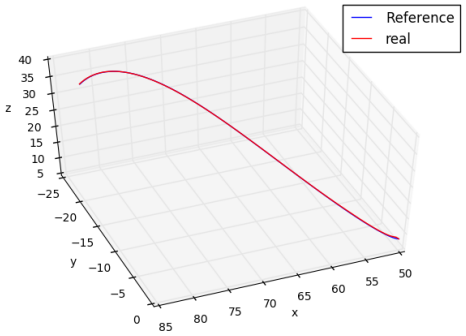

Figure 3.13: Referenced path and real path

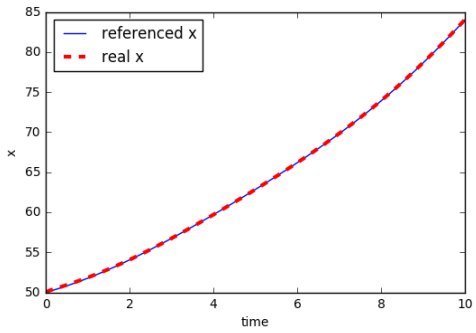

Figure 3.14: Referenced and real x position

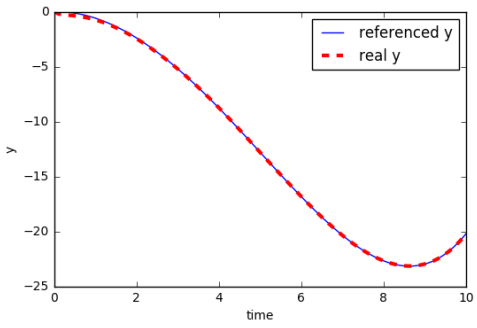

Figure 3.15: Referenced and real y position

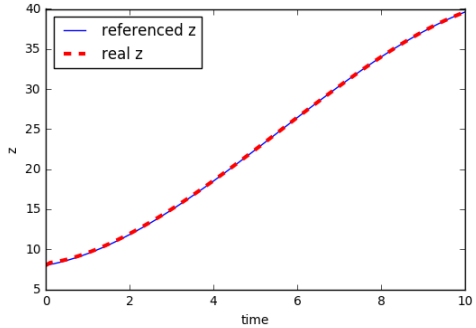

Figure 3.16: Referenced and real z position

Fig. 3.17 presents the tracking errors of the end-effector in three dimensions. All the tracking errors just have small fluctuations around zero, so the SCARA

robot can achieve the trajectory tracking task by using DDPG with priority replay memory.

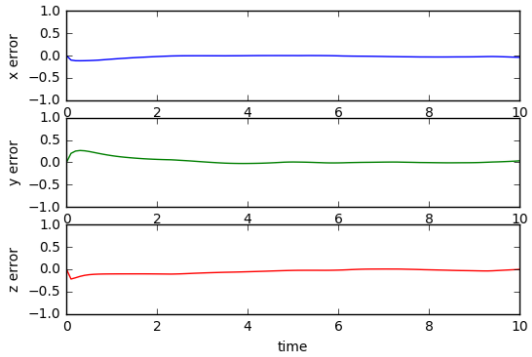

Figure 3.17: Tracking errors in three dimensions

3.5.4 Trajectory tracking of mobile robot

In this subsection, we use DDPG with priority replay memory to train the mobile robot described in subsection 3.5.1 to track the given trajectories.

Design of observation and reward function

The input of the actor is the observation vector $\boldsymbol{s}$ and the output of the actor are the forward velocity v and rotation speed w of mobile robot measured in its local coordinate frame.

The observation $\boldsymbol{s}$ is designed as:

$$\boldsymbol{s} = [x, y, \sin(\theta), \cos(\theta), \dot{x}, \dot{y}, \dot{\theta}, e_x, e_y, e_{\sin(\theta)}, e_{\cos(\theta)}, e_{\dot{x}}, e_{\dot{y}}, e_{\dot{\theta}}],$$

where $[x, y, \sin(\theta), \cos(\theta)]$ represents the real position and orientation of the mobile robot in the global frame and the trigonometric function of the orientation is used since the angle is periodic; $[\dot{x}, \dot{y}, \dot{\theta}]$ denotes the real linear velocities and rotation speed of mobile robot in the global frame; $[e_x, e_y, e_{\sin(\theta)}, e_{\cos(\theta)}]$ denotes

the errors of the positions and orientation; $[e_{\dot{x}}, e_{\dot{y}}, e_{\dot{\theta}}]$ denotes the errors of the velocities and rotation speed.

The low-level control of the mobile robot should be based on the angular velocities of the two driving wheels which can be derived from Eq. 3.19 in the real application. In our simulation, after getting the forward velocity and rotation speed of the mobile robot in the local frame, Eq. 3.18 is used to obtain its velocities and rotation speed in the global frame. Then the positions and orientation of the mobile robot can be updated as follows if the time interval dt is small enough.

$$
\begin{aligned}
x &\leftarrow x + \dot{x} \cdot dt \\
y &\leftarrow y + \dot{y} \cdot dt \\
\theta &\leftarrow \theta + \dot{\theta} \cdot dt.
\end{aligned}
$$

The training objective is to make the mobile robot arrive at the given position with proper orientation at each time step, while its velocities should also satisfy the referenced values. Hence, the reward function is designed as an exponential function of the position and velocity errors as expressed in Eq. 3.23.

$$
\begin{aligned}
r_p &= exp\Big(- a_p\big(e_x^2 + e_y^2 + (e_{\sin\theta})^2 + (e_{\cos\theta})^2\big)\Big) \\
r_v &= exp\big(- a_v(e_{\dot{x}}^2 + e_{\dot{y}}^2 + e_{\dot{\theta}}^2)\big) \\
r &= \omega_p r_p + \omega_v r_v.
\end{aligned}
\tag{3.23}
$$

The total reward consists of two terms, r_p related to the position tracking error and r_v related to the velocity tracking error. ω_p and ω_v are the weights of the two terms. a_p and a_v are the coefficients of the exponential functions. We choose $\omega_p = 0.4$, $\omega_v = 0.6$, $a_p = 1$, $a_v = 2$. The actor outputs the velocities which can influence the accuracy of the positions, so velocity term should have larger weight and coefficient. The reward is still in the range of $0 \sim 1$. If there is no

error in both the positions and velocities, the reward can get to the maximum value 1.

Training result of mobile robot

Firstly, we define a specific curve $y_r = 2\sin(0.2x_r)$ for the mobile robot to track. Both x position and y position are the function of time t, and the parametric equation about time t of the given curve is:

$$\begin{aligned} x_r &= t \\ y_r &= 2\sin(0.2t). \end{aligned} \tag{3.24}$$

The corresponding velocity functions with respect to the time t are $\dot{x_r} = 1$ and $\dot{y_r} = 0.4\cos(0.2t)$. The given orientation of the mobile robot can be derived from the given velocity $\dot{x_r}$ and $\dot{y_r}$:

$$\theta_r = \arctan(\frac{dy_r}{dx_r}) = \arctan(\frac{\frac{dy_r}{dt}}{\frac{dx_r}{dt}}) = \arctan(\frac{\dot{y}_r}{\dot{x}_r}) = \arctan\big(0.4\cos(0.2t)\big), \tag{3.25}$$

where the range of the referenced orientation is between $-\frac{\pi}{2} \sim \frac{\pi}{2}$.

We still use the random referenced state initialization and early termination to train the mobile robot for 1000 episodes. Fig. 3.18 shows the change of episode reward and Fig. 3.19 presents the loss change of the actor which is the negative mean of Q-value of the state-action pairs. Therefore, the Q-value tends to increase, which conforms to the training objective.

The tracking results of the mobile robot are presented in Fig. 3.20 $\sim$ Fig. 3.23, including the x, y position and orientation. Fig. 3.24 presents the tracking errors of the mobile robot in three dimensions. We can see some obvious deviations from the referenced positions and orientation.

Next, we train the mobile robot to track a more complicated trajectory. The

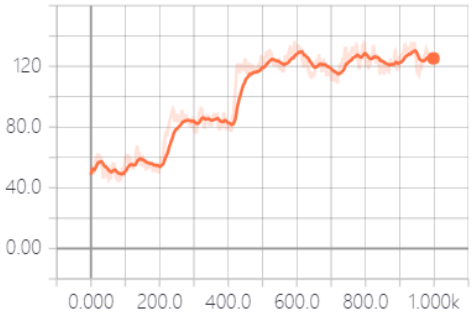

Figure 3.18: Episode rewards

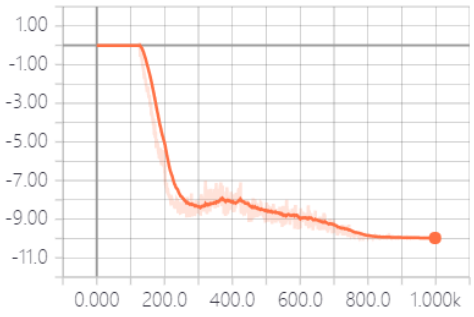

Figure 3.19: Actor loss

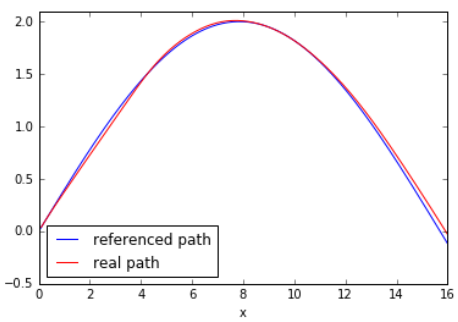

Figure 3.20: Referenced path and real path

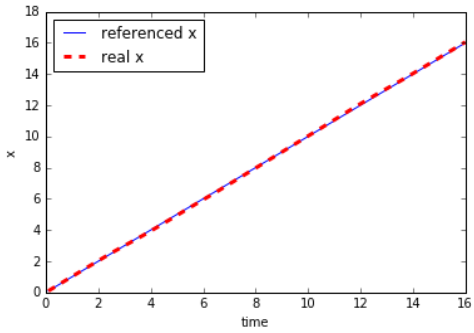

Figure 3.21: Referenced and real x position

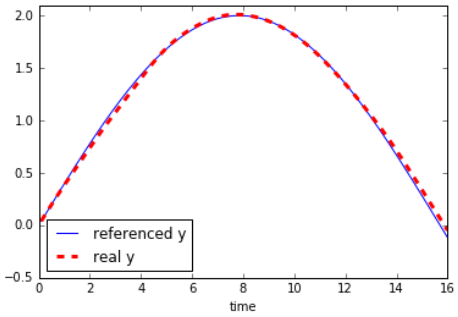

Figure 3.22: Referenced and real y position

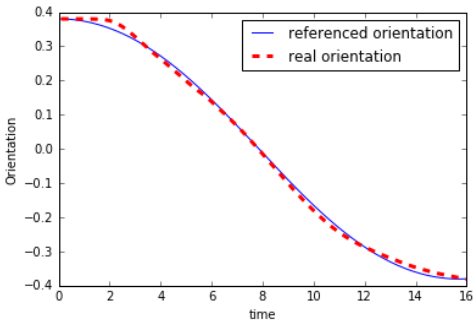

Figure 3.23: Referenced and real orientation

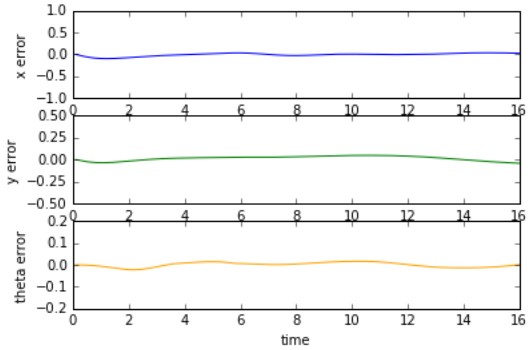

Figure 3.24: Tracking errors of position and orientation

parametric function about time t of the curve is:

$$\begin{aligned} x_r &= \sin(\frac{\pi}{8}t) \\ y_r &= -2\cos(\frac{\pi}{8}t). \end{aligned} \tag{3.26}$$

so the curve function is $x^2 + \frac{y^2}{4} = 1$, which is a ellipse. The velocity functions of x and y direction are $\dot{x_r} = \frac{\pi}{8}\cos(\frac{\pi}{8}t)$ and $\dot{y_r} = \frac{\pi}{4}\sin(\frac{\pi}{8}t)$, respectively. The given orientation of the mobile robot can be derived from $\dot{x_r}$ and $\dot{y_r}$.

$$\theta_r = \begin{cases} \arctan(\frac{\dot{y_r}}{\dot{x_r}}), & \dot{x_r} > 0, \dot{y_r} \geq 0 \\ \arctan(\frac{\dot{y_r}}{\dot{x_r}}) + 2\pi & \dot{x_r} > 0, \dot{y_r} < 0 \\ \arctan(\frac{\dot{y_r}}{\dot{x_r}}) + \pi & \dot{x_r} < 0 \\ \frac{\pi}{2} & \dot{x_r} = 0, \dot{y_r} \geq 0 \\ \frac{3\pi}{2} & \dot{x_r} = 0, \dot{y_r} < 0 \end{cases}, \tag{3.27}$$

where θ_r is between $0 \sim 2\pi$.

The reward function and hyperparameters are the same as the previous one except the scaling factors of the actions. The change of episode reward and actor loss are respectively presented in Fig.3.25 and Fig.3.26.

The tracking results and tracking errors of the mobile robot are shown in Fig.3.27 $\sim$ Fig.3.31. We can see that the mobile robot can track the given trajectory roughly, so it has space to make some improvements.

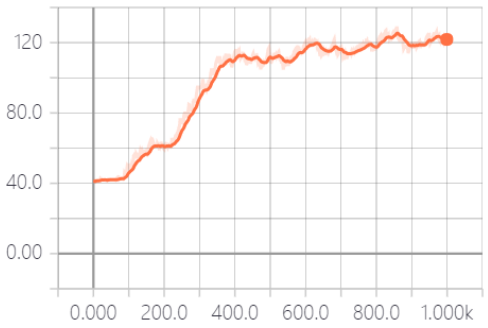

Figure 3.25: Episode rewards

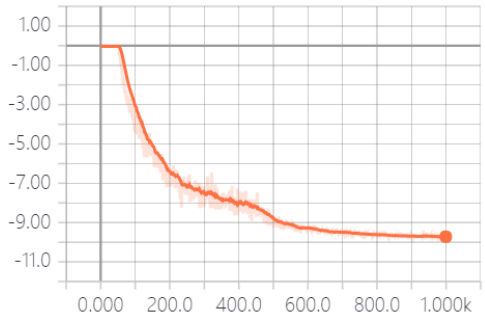

Figure 3.26: Actor loss

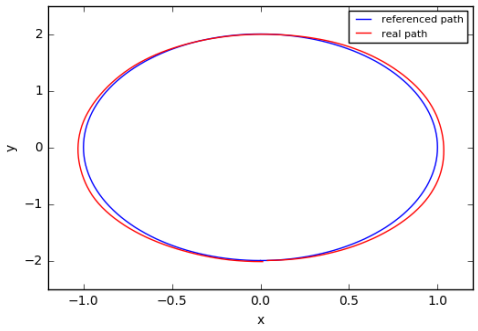

Figure 3.27: Referenced path and real path

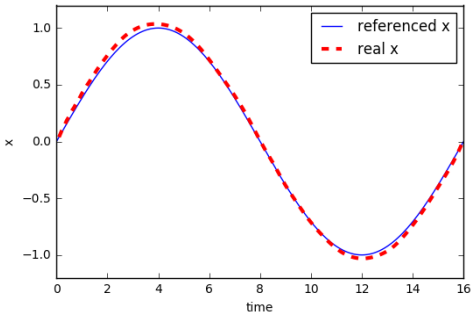

Figure 3.28: Referenced and real x position

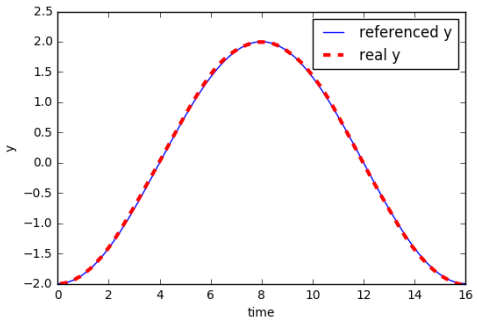

Figure 3.29: Referenced and real y position

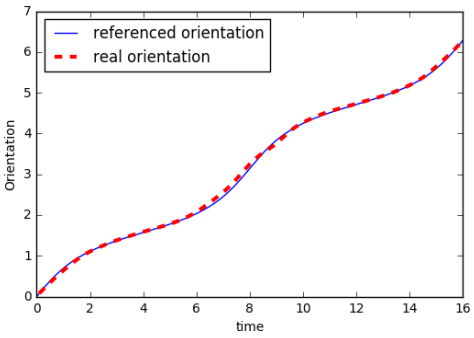

Figure 3.30: Referenced and real orientation

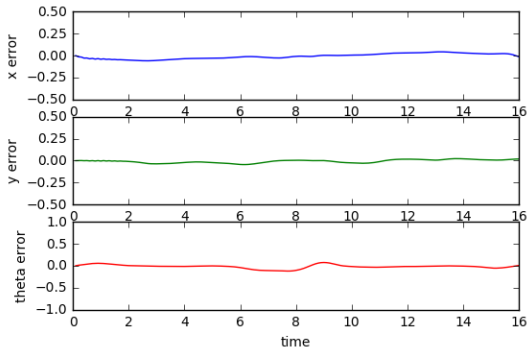

Figure 3.31: Tracking errors of position and orientation

The hyperparameters of DDPG with priority replay memory for the robotic tracking control are shown in Table 3.3.

Although many traditional control methods could achieve good tracking performance in the simulation, most of these control algorithms need the exact dynamic model of the controlled system. The dynamic models of the manipulator and mobile robot are very comlpex and it is difficult to obtain their exact dynamic models, so those model-based control algorithms may not have good performance in the real application. Another promising method to solve the

Table 3.3: Hyperparameters of DDPG for mobile robot

Hyperparameters	Value
Learning rate of actor α_μ	0.0001
Learning rate of critic α_Q	0.0001
Soft replacement coefficient ε	0.01
Reward discounted factor γ	0.9
Memory capacity C	50000
Batch size b_n	32
Priority exponent α	0.6
Initial importance sampling weight exponent β_0	0.4
Incremental rate of β	0.001

complex control problems is to introduce model-free deep reinforcement learning as the high-level controller. The robots can learn the controllers from the interaction with the environment, so the controllers are updated constantly by the experiences of the robots to maximize the long-term rewards without considering any model information. In our simulations, we just consider the kinematics of the robots when using deep reinforcement learning, which aims to reduce the complexity of the high-level controllers to improve the learning speed and performance. The reinforcement learning controllers just provide the target positions or velocities of the robots, so low-level controllers are also required to produce the final torques exerted to the motors. PID controller is most suitable for the low-level control since it can be model-free and has enough robustness to apply to the practical system. Therefore, model-free deep reinforcement learning focuses on the high-level decision making. Trajectory tracking is just a relatively simple robotic task. The combination of the high-level decision making by deep reinforcement learning and low-level control can also be applied to solve other more complicated robotic tasks which are difficult for the traditional control methods.

Chapter 4

Continuous control for robot based on distributed deep reinforcement learning

Distributed stochastic gradient descent has been widely used in the supervised learning since it can help to accelerate the training of deep neural network. It adopts parallel workers, each of which computes the gradients by using its own mini-batch data sampled from the whole training set. The gradients computed by the workers need to be sent to the parameter server to update the global parameters. The parameters can be updated either in an asynchronous or synchronous way.

Deep reinforcement learning can also adopt this distributed framework to update the parameters of networks. Google DeepMind proposed a asynchronous variant of actor-critic [11], which is known as the asynchronous advantage actor-critic (A3C), trained on a single multi-core CPU rather than GPU. A3C can be successfully applied to many continuous and discrete control tasks.

A3C has the problem of gradient delay, which may influence the convergence of neural network. Hence, we introduce a synchronous version of the advantage

actor-critic, abbreviated as A2C, to eliminate the gradient delay problem. We propose a new reward function for the redundant manipulator aiming to optimize the reaching path of the end-effector. We use both A3C and A2C with two different reward functions to train the redundant manipulator to complete the multi-goal task. The simulation results show that the new reward function could optimize the reaching path and A2C could achieve better performance than A3C. Next we propose a distributed framework of DDPG, where synchronous workers compute the gradients for the global network and collecting workers only produce the transitions to the shared replay memory with different policies and exploration noises. Then, we use distributed DDPG to train the SCARA robot and mobile robot to track the same trajectories. The simulation results demonstrate that distributed DDPG enables the agent to learn faster and achieves smaller tracking errors compared with single-worker DDPG.

4.1 Advantage Actor-Critic

In chapter 2, we have introduced policy gradient, which aims to adjust the probabilities of actions according to their accumulated rewards. However, in some reinforcement learning problems, the rewards from environment are always positive, which cause that the accumulated rewards are always positive regardless of the policy. That is to say, we may increase the probobilities of all the actions but just reduce the incement amplitude for the actions with bad performance. Therefore, we can't decrease the probabilities of bad actions by using this approach. One simple method to solve this problem is to subtract a baseline from the long-term return [11]. Traditional policy gradient method updates the network parameters in the direction of $\nabla_\theta \log \pi_\theta(a_t|s_t)R_t$, where R_t is the long term return from time t. By introducing the baseline b_t, the policy gradient becomes $\nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b_t)$. In this way, the coefficient of policy gradient can be positive if the action is evaluated as a good one by comparing with the baseline and negative if the action is evaluated as a bad one. At the

same time, the absolute value of coefficient is decreased, which can enhance the stability of algorithm. Moreover, adding this baseline will not bias the original computation, because we can derive:

$$\begin{aligned}\mathbb{E}[\nabla_\theta \log \pi_\theta(\tau) b] &= \int_{\tau\sim\pi_\theta(\tau)} \big(\pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) b\big) d\tau = \int_{\tau\sim\pi_\theta(\tau)} \big(\nabla_\theta \pi_\theta(\tau) b\big) d\tau \\ &= b\int_{\tau\sim\pi_\theta(\tau)} (\nabla_\theta \pi_\theta(\tau)) d\tau = b\nabla_\theta \int_{\tau\sim\pi_\theta(\tau)} \pi_\theta(\tau) d\tau = b\nabla_\theta 1 = 0.\end{aligned} \tag{4.1}$$

After adding a baseline, the policy gradient is represented as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\Big[\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\big(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) - b_{i,t'}\big)\Big]. \tag{4.2}$$

The above equation uses the trajectory reward as the value, which is unbiased. However, in the real training process, we can't complete a lot of interactions since the training time is limited. Therefore, we can only get finite trajectories, which can not represent the true expectation of all the trajectories well. Each sequence obtained from one interaction has some difference with others, resulting in the difference of corresponding rewards. So inadequate interactions could introduce relatively large variance for the trajectory rewards.

An important problem of machine learning is the balance between the bias and variance. As mentioned above, policy gradient has relatively large variance. In order to stabilize the model, we can reduce the variance by sacrificing the bias to some extent. An effective method is to use the Actor-Critic algorithm which uses an independent neural network to evaluate the long-term return of the trajectory instead of using its real returns. Hence, two neural networks, actor and critic, are involved. The update of critic aims to minimize the TD-Error: $r(s_t, a_t) + V(s_{t+1}) - V(s_t)$, where $V(s_t)$ is the value at state s_t under a specific policy. This method also enables one-step update because we can estimate the future rewards at each state to compute the policy gradient instead of going

through the whole episode to collect the rewards of all the time steps.

The coefficient of the policy gradient can use TD error, which can also be regarded as the advantage estimation of performing action a_t at state s_t. $V(s_t)$ is the estimation of baseline which can be treated as the average value at state s_t by considering all the possible actions. $r(s_t, a_t) + V(s_{t+1})$ is the action value of state s_t after taking action a_t. So the advantage can measure an action a_t under the state s_t by comparing with the average value of state s_t. Although this method could decease the variance and improve the model stability, the bias is a little large. In order to make a better balance between the bias and variance, we use multiple-step returns to estimate the advantage and critic loss by collecting sequential data, which can also speed up the training of critic. The advantage function is written as [11]:

$$A(s_t, a_t) = \sum_{i=0}^{T_n - t - 1} \gamma^i r_{t+i} + \gamma^n V(s_{T_n}) - V(s_t). \tag{4.3}$$

In order to encourage the exploration, the entropy of the policy is introduced to the objective function of the actor. Entropy can measure the uncertainty of probability distribution, so we hope the entropy should be as large as possible to avoid approaching a deterministic policy. The resulting policy gradient is as follows [11]:

$$\nabla_{\theta_\pi} J(\theta_\pi) \approx \frac{1}{N} \sum_{t=1}^{N} \Big[\nabla_{\theta_\pi} \log \pi(a_t | s_t; \theta_\pi) A(s_t, a_t) + \beta \nabla_{\theta_\pi} H\big(\pi(s_t; \theta_\pi)\big) \Big], \tag{4.4}$$

where H is the entropy and β is a hyperparameter that adjusts the strength of entropy regularization.

The update of critic is to minimize the difference between the evaluated value $V(s_t)$ and target value $R_t = \sum_{i=0}^{T_n - t - 1} \gamma^i r_{t+i} + \gamma^n V(s_{T_n})$. Thus, the loss of critic

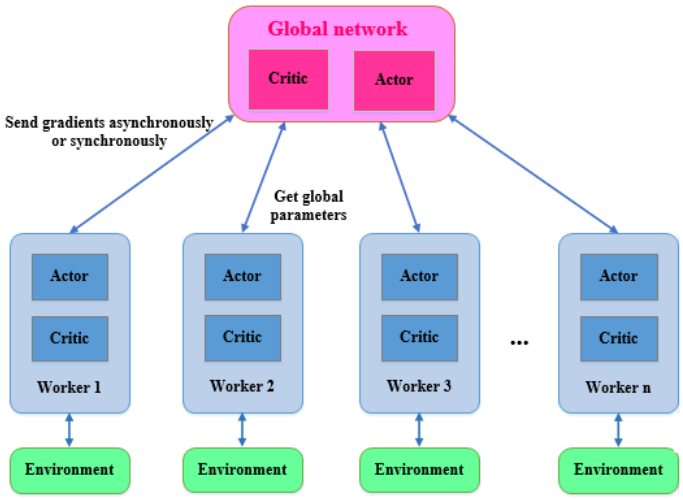

Figure 4.1: A3C and A2C framework

network is represented as follows:

$$L(\theta_v) \approx \frac{1}{N} \sum_{t=1}^{N} \left(R_t - V(s_t; \theta_v)\right)^2. \tag{4.5}$$

4.1.1 Asynchronous Advantage Actor-Critic

Firstly, we introduce A3C [11], a popular deep reinforcement learning algorithm, which can achieve good performance in many tasks. Fig. 4.1 shows the framework of A3C and A2C. The only difference between A3C and A2C is the update method of global network.

A3C adopts asynchronous stochastic gradient descent to update the neural networks. It has multiple workers which run in the different threads. Each worker explores its own environment and computes the gradient based on the collected transitions. The gradients of each worker are sent to the global networks which has the same network architectures with the workers, asynchronously. The global network applies the gradients from the workers to update its parameters. After that, the worker will obtain the parameters of global network and continue to explore the environment by using the new policy. Workers may have different parameters because the global network is constantly updated by the gradients

of different workers. Furthermore, using gradients from different workers can reduce the correlation of samples since the collected transitions of one worker is sequential.

The pseudocode of one worker of A3C used in our simulation is shown in Algorithm 3. There is a local buffer for each worker to store the transitions generated from the interaction with the environment. When the local buffer has a certain number of transitions, the target value of each state will be computed according to the critic and a series of rewards. Then the advantage of each state-action pair is calculated by Eq. 4.3. With the advanteges, the gradients of actor and critic are derived by averaging the gradients of the sequential transitions in the local buffer.

4.1.2 Synchronous Advantage Actor-Critic

For A3C, the parameters of the global networks are updated by the different workers asynchronously, so it has the problem of gradient delay [65]. When a worker sends its gradient to the global network, the global parameters may have been updated by other workers, so this gradient becomes delayed with respect to the global network. Gradient delay may slow down the convergence speed of network or cause some unexpected results.

In order to avoid the gradient delay problem, we introduce a synchronous version of advantage actor-critic, A2C. A2C also has multiple workers to explore the environment and compute the gradients. The difference is that all the workers of A2C have the same parameters with the global network. Each worker needs to calculate the gradients by using the transitions it collects and then sends the gradients to the global network. After the global network receives the gradients of all the workers, it uses the average of these gradients to update its parameters. Then the updated parameters are synchronized to all the workers and the workers continue to collect transitions with the new policy. Therefore, the

Algorithm 3 One worker of A3C

Initialize:
Critic $V(s;\theta_v)$ and actor $\pi(s;\theta_\pi)$ of global network
Critic $V(s;\theta'_v)$ and actor $\pi(s;\theta'_\pi)$ of each worker with
$\theta'_v \leftarrow \theta_v$ and $\theta'_\pi \leftarrow \theta_\pi$
for $episode = 1, \ldots, M$ **do**
  Get the initial state s_0
  for $t = 1, \ldots, T$ **do**
    Sample an action a_t from the policy $\pi(s_t;\theta'_\pi)$
    Preform the action a_t and then obtain the reward r_t and the next state s_{t+1} from the environment
    Store s_t, a_t and r_t to the local buffer
    if local buffer is full or terminate state **then**
      $$R = \begin{cases} 0, & if\ terminate\ state \\ V(s_{t+1};\theta'_v) & if\ not\ terminate\ state \end{cases}$$
      for $r \in$ local buffer R_r $[r_t, r_{t-1}, r_{t-2}, \ldots]$ **do**
        $R \leftarrow r + \gamma R$
        Store R to the local buffer R_{target}
      end for
      Compute the actor gradient and critic gradient by using the data in the local buffer:
      $g_\pi = \frac{1}{N}\sum_{t=1}^{N}\Big[\nabla_{\theta'_\pi}\log\pi(a_t|s_t;\theta'_\pi)\big(R_t - V(s_t)\big) + \beta\nabla_{\theta'_\pi}H\big(\pi(s_t;\theta'_\pi)\big)\Big]$
      $g_v = \frac{1}{N}\sum_{t=1}^{N}\nabla_{\theta'_v}\Big[\big(R_t - V(s_t;\theta'_v)\big)^2\Big]$
      Update the global parameters θ_π and θ_v by using g_π and g_v asynchronously
      Synchronize worker parameters from the global network: $\theta'_v \leftarrow \theta_v$ and $\theta'_\pi \leftarrow \theta_\pi$
      Clean out the local buffer
    end if
    $s_t \leftarrow s_{t+1}$
  end for
end for

workers always explore the environments with the latest parameters obtained from the global network. The gradients of the workers are computed with respect to the global parameters, so A2C doesn't have the problem of gradient delay, which can guarantee the convergence of networks.

Usually, A3C has a higher training efficiency than A2C because the worker of A3C can apply its computed gradient to the global parameters immediately.

However, the workers of A2C need to wait for each other and will not continue until all the workers send their gradients to the global network. Hence, A3C has no barrier when updating the global parameters, which guarantees a higher training speed.

A3C and A2C are the on-policy reinforcement learning algorithms since they use the current policy to produce the transitions and compute the gradients. They don't need a replay memory with large capacity to store the transitions of a long period of time. Instead, they only need the local buffers with small capacity for the workers to store the interactive data with the environment temporarily. After collecting enough transitions, the worker will calculate the gradients of the actor and critic loss with respect to its own parameters and then send them to the global network. Then the local buffers are cleaned out and new transitions will be added. So on-policy reinforcement learning can help to save the memory but can't ensure the data efficiency.

4.1.3 Training results of redundant manipulator

In chapter 3, we have used DDPG with HER and the shaped reward to train a redundant manipulator to make its end-effector reach any given goal position. In this section, we will use A3C and A2C to trian the redundant manipulator to complete the same task.

The input of the actor is the observation and the outputs of the actor are the means of position changes $\Delta p = [\Delta x, \Delta y, \Delta z]$ rather than the deterministic changes as DDPG. The standard deviations are added separately as the hyper-parameters. The actions are sampled from the normal distributions with the means output from the actor and the standard deviations. This approach could encourage more exploration than the deterministic policy since it could have some possibilities to sample other actions besides the mean actions.

The elements of the observation may have different magnitudes, so it may be

difficult for the neural network to learn effectively. One method to solve this problem is to scale the features by some different constants in order to make them in the similar ranges. Here, we use a more effective approach, Batch Normalization, to solve this problem and also speed up the training of neural network.

Traditional neural networks usually only normalize the input features to make each feature have zero mean and one variance, which can help to reduce the difference among the samples. The learning nature of neural network is to learn the data distribution. If the training data and testing data have different distributions, the generalization ability of neural network can be largely degraded. On the other hand, if the distributions of mini-batch samples are different during the training process by using stochastic gradient descent, neural network must learn to adapt to different distributions at each iteration, which can considerably slow down the training speed. The training of neural network is a complicated process. As long as the parameters of former layers change slightly, the input distributions of latter layers could have large alterations due to the accumulation. So the latter layers need to constantly adapt to the parameter updates of the former layers, which may cause the decrease of learning speed. Moreover, when the changed parameters of former layers cause the outputs of latter layers falling into the saturated zone of the activation function, it could make the learning stop prematurely, which is known as the gradient vanishing problem. The phenomenon of changing data distribution in hidden layers is called the internal covariate shift. Batch Normalization [66] is effective to solve the above-mentioned problem and can also make the learning less affected by the parameter initialization. The concrete procedure of Batch Normalization is shown in Fig. 4.2.

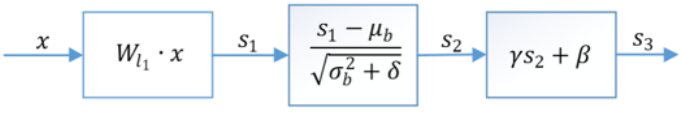

Figure 4.2: Batch Normalization

Fig. 4.2 only presents the Batch Normalization procedure of the first hidden layer, and the procedures for the following layers are the same as it. Firstly, the normalized mini-batch input x passes through a linear operation by multiplying the weights W_{l_1}. The biases can be canceled from operation since it can be offset by subtracting the mean. Instead of passing through the activation function directly, the output of linear operation also needs to subtract the mean and divide the standard deviation of the mini-batch data. However, if the data are normalized to the distribution with zero mean and one standard deviation, it may destroy the learned features of this layer. To solve this problem, two parameters γ and β are introduced to scale and shift the normalized values. So the layer is likely to recover to the original input if $\gamma = \sigma_b$ and $\beta = \mu_b$. Furthermore, β and γ are the learning parameters, so their values can be updated with the training of network. After that, the output goes through the activation function and then is sent to the next layer. Here, $\mu_b = \frac{1}{m}\sum_{i=1}^{m} W_{l_1} x_i$ and $\sigma_b^2 = \frac{1}{m}\sum_{i=1}^{m}(W_{l_1} x_i - \mu_b)^2$, where m is the mini-batch size. δ is a small positive number to guarantee the denominator doesn't equal to zero.

It should be noted that the above method can only be used in the training process and can't be used in the testing period since there is only one sample for every testing. If we still use this method, the mean should be the sample itself and thus $s_1 - \mu_b$ equals to 0. In the testing period, the ideal solution is to compute μ and σ^2 using the whole training set after finishing training. However, it is not applicable since the training set may be very large. Another approximated method is to use the moving averages of the means and standard deviations calculated during the whole training process, which guarantees that the means and standard deviations approaching the end of the training have larger weights. In the reinforcement learning, the exploration process which chooses an action at each step should use the moving average of means and standard deviations to normalize the input data of each layer since only one state is input to the actor.

The architecture of actor and critic is shown in Fig. 4.3. For the actor, We use a

two-hidden-layer FC network with Batch Normalization. Each hidden layer has 200 neurons. $\boldsymbol{C}$ is the scaled vector multiplied to the mean action $\boldsymbol{\mu}$. The real action is sampled from the normal distribution with the mean μ and standard deviation σ. The input of critic is the observation $\boldsymbol{s}$ and the output of critic is the evaluated value of observation, $\boldsymbol{V}$. We also use two-hidden-layer FC network for the critic and each hidden layer has 200 neurons.

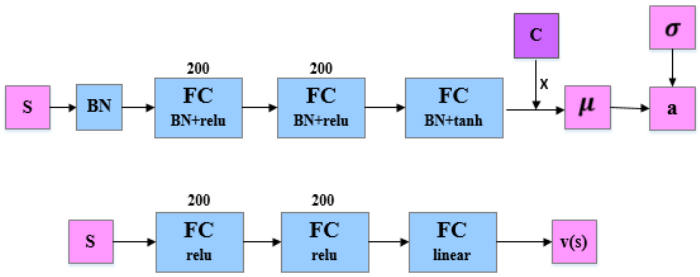

Figure 4.3: Architecture of actor and critic

We only apply Batch Normalization to the actor and don't use it to the critic. Because the output of actor uses tanh activation function which has the saturated zone while the output of critic uses the linear activation function without saturated zone.

The reward funciton for the redundant manipulator is as follows:

$$r(s_t, a_t) = \|g - p_t\| - \|g - p_{t+1}\|, \tag{4.6}$$

which measures the change of distance between the end-effector and the goal position. If the distance becomes smaller, it will get a positive reward, or it will get a negative reward. Moreover, if the end-effector moves out of the feasible region, it will get a negative reward. The manipulator can get a positive reward as long as it moves closer to the goal position. However, we don't consider the concrete path of the end-effector from the initial position to the target position, since this reward function only encourages the end-effector to approach to the goal position regardless of its reaching path.

Here, we propose a new reward function which could help to optimize the reach-

ing path of the end-effector. The ideal path is the linear path connecting the starting position and the target position, which is the shortest path. We hope to design a reward function that could encourage the end-effector to move along a relatively shorter path. The basic idea of this reward function is to constrain the moving direction of the end-effector. The moving direction should approximately point to the target position, so the reward received by the agent at each time step depends on the angle between the actual moving direction of the end-effector and the target direction, ϕ, which is calculated as follows:

$$\begin{aligned}
\overrightarrow{D}_{tar} &= [x_g - x_t, y_g - y_t, z_g - z_t] \\
\overrightarrow{D}_{act} &= [x_{t+1} - x_t, y_{t+1} - y_t, z_{t+1} - z_t] \\
\phi &= \arccos\Big(\frac{\overrightarrow{D}_{tar}}{|\overrightarrow{D}_{tar}|} \cdot \frac{\overrightarrow{D}_{act}}{|\overrightarrow{D}_{act}|}\Big),
\end{aligned}$$

where $[x_g, y_g, z_g]$ is the target position; $[x_t, y_t, z_t]$ and $[x_{t+1}, y_{t+1}, z_{t+1}]$ are the end-effector position at time t and $t+1$, respectively; $\overrightarrow{D}_{tar}$ is the target direction; $\overrightarrow{D}_{act}$ is the actual moving direction; the inner product of their unit vectors is the cosine of ϕ.

We set that if ϕ equals to 0, the reward is 2; if ϕ equals to $\frac{\pi}{6}$, the reward is 0; if ϕ equals to π, the reward is -2. The rewards corresponding to other angles are obtained by using the linear interpolation. Therefore, the reward function is expressed as follows:

$$r = \begin{cases} -\frac{12}{\pi} \cdot \phi + 2, & 0 \leq \phi \leq \frac{\pi}{6} \\ -\frac{12}{5\pi} \cdot \phi + 0.4, & \frac{\pi}{6} < \phi \leq \pi \end{cases} \tag{4.7}$$

In this way, we only give a positive reward when ϕ is in the range of $0 \sim 30^\circ$ and negative reward when ϕ is between $30^\circ \sim 180^\circ$.

Firstly, We apply A3C to train the redundant manipulator for 2000 episodes by using the two reward functions. Four workers are created to explore the environment and compute the gradients for the global network. Figure 4.4 shows

the overall 2000 episode rewards by using the ordinary reward function, which integrate the episode rewards of all the workers. Each curve with a different color in Fig. 4.5 represents the episode rewards of one worker. It can be seen that each worker is trained for about 500 episodes under the total 2000 episodes. Figure 4.6 and Fig. 4.7 show the overall episode rewards and episode rewards of different workers respectively by using the path optimization reward.

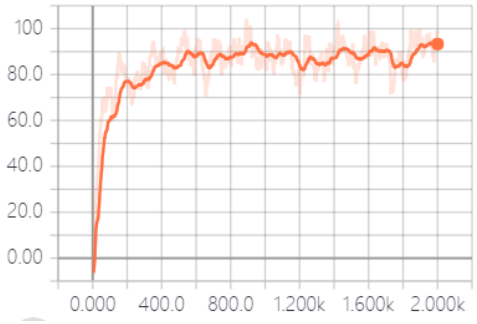

Figure 4.4: Overall episode rewards

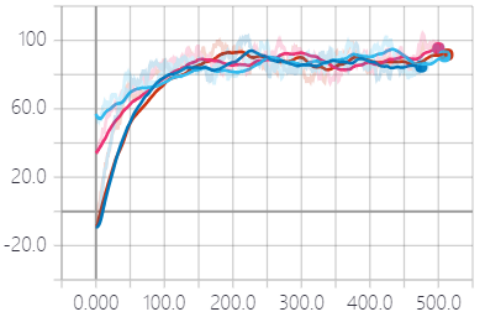

Figure 4.5: Episode rewards of different workers

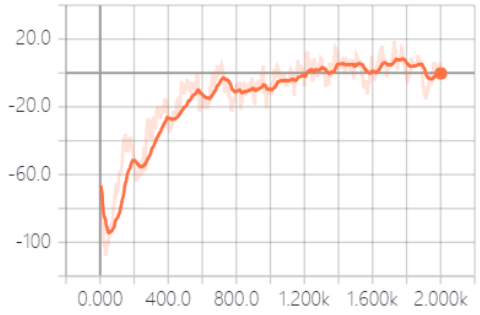

Figure 4.6: Overall episode rewards

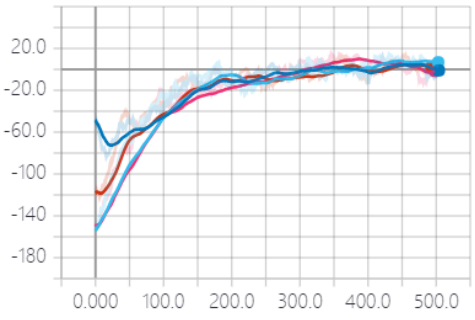

Figure 4.7: Episode rewards of different workers

The success rate of A3C for every 100 episodes of three random seeds by using the two reward functions are recorded in Fig. 4.8. We can see that A3C with ordinary reward function can achieve a average success rate of more than 70% and A3C with path optimization reward can get to more than 80% average success rate. We also evaluate the reaching path by measuring the difference between the actual moving distance of the end-effector and the linear distance between the starting position and target position, which is called the extra distance. We compute the average extra distance of 100 episodes of the three random seeds by using the two reward functions, which is depicted in Fig. 4.9. We can see that the average extra distance of the path optimization reward is smaller than

the ordinary reward, which means the path of end-effector trained by path optimization reward is closer to the ideal path. With the training process, the extra distances of both reward functions gradually decrease while the extra distance of the path optimization reward has relatively larger decreasing speed.

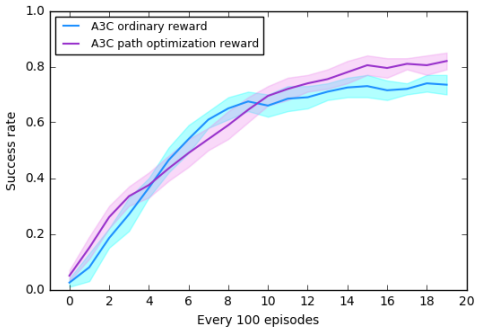

Figure 4.8: Success rate

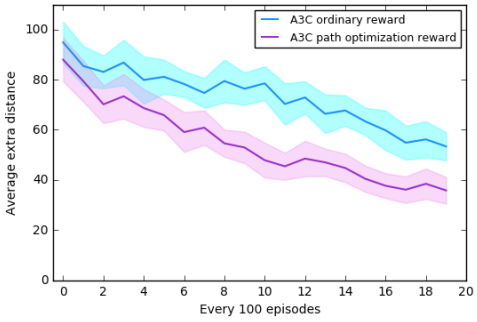

Figure 4.9: Average extra distance

Next, we use A2C to train the redundant manipulator to complete the same task. After all the four workers collect 8 transitions and calculate the gradients, the global network averages these gradients to update its parameters. The episode rewards of the two reward functions are presented in Fig. 4.10 $\sim$ Fig. 4.13 respectively.

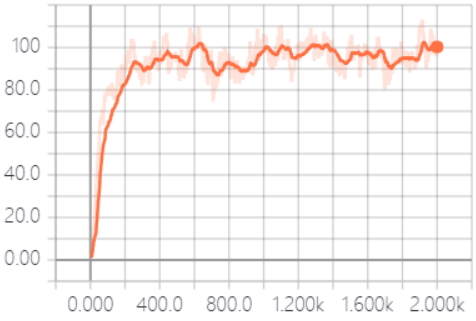

Figure 4.10: Overall episode rewards

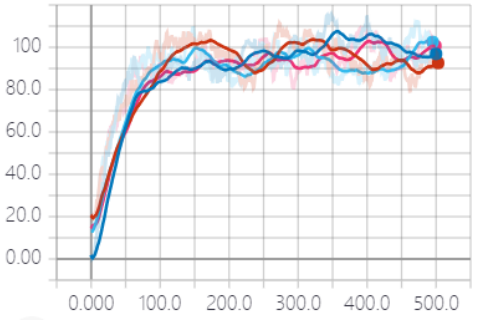

Figure 4.11: Episode rewards of different workers

The success rate of A2C is presented in Fig. 4.14. A2C with the ordinary reward could reach nearly 80% average success rate and A2C with the path optimization reward could achieve more than 85% average success rate. The extra distance by using the path optimization reward is obviously smaller than the ordinary reward as shown in Fig. 4.15.

We use the saved model with the best results to evaluate the performance of each

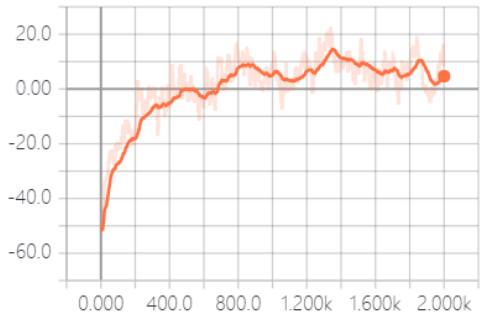

Figure 4.12: Overall episode rewards

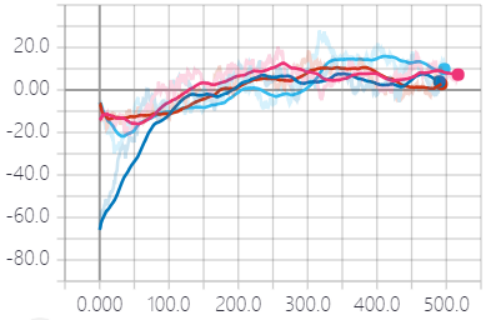

Figure 4.13: Episode rewards of different workers

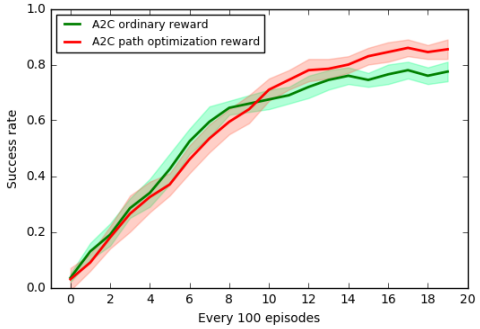

Figure 4.14: Success rate

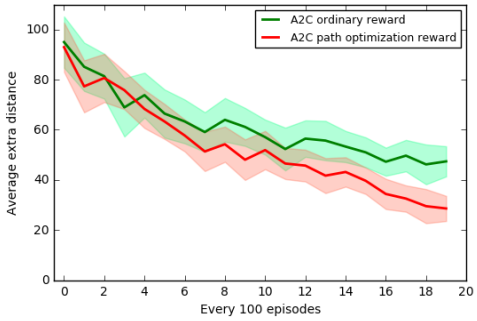

Figure 4.15: Average extra distance

method. The agents are tested to reach any given position from a random initial position for 100 episodes. The testing results are shown in Table 4.1. We can see that the path optimization reward can not only optimize the reaching path but also promote the success rate. Moreover, A2C can get a better performance than A3C, which may be because that the gradient delay problem of A3C influences the convergence of networks and degrades the performance of agent. We also note that the training time of A3C and A2C is much shorter than DDPG, so they can speed up the training of agent by runing multiple workers at the same time.

Table 4.1: Testing results of A3C and A2C

Methods	Success times
A3C with ordinary reward	76
A3C with path optimization reward	84
A2C with ordinary reward	79
A2C with path optimization reward	88

The hyperparameters of A3C and A2C for the redundant manipulator are shown in Table 4.2.

Table 4.2: Hyperparameters of A3C for manipulator

Hyperparameters	Value
Learning rate of actor α_π	0.0001
Learning rate of critic α_v	0.0002
Policy entropy coefficient β	0.01
Reward discounted factor γ	0.9
Standard deviation σ	[0.8,0.8,0.1]

4.2 Distributed Deep Deterministic Policy Gradient

A3C and A2C are the on-policy reinforcement learning algorithms since their gradient calculations are based on the current policy. The agent explores the environment by using the current policy and collects the sequential transitions which are used to compute the gradients. After the parameters are updated, these transitions are discarded and new transitions will be collected. Hence, on-policy reinforcement learning needs more time to collect transitions which can only be used once. Off-policy reinforcement learning usually has a replay memory to store the transitions of the past period of time, so the gradient calculation can use the transitions produced by the previous policies. Therefore, off-policy reinforcement learning can reuse the transitions and achieve higher data efficiency.

In chapter 3, we have introduced an off-policy reinforcement learning algorithm, DDPG. In this section, we propose a distributed framework of DDPG to improve the performance.

4.2.1 Distributed framework of DDPG

The distributed framework of DDPG is shown in Fig. 4.16. In order to collect the data faster, we create multiple workers, which run on the multi-processes, to explore the environment at the same time. Each worker has its own instance

of environment to generate the transitions. The workers can be divided into the synchronous workers and the collecting workers. The synchronous workers need to calculate the gradients for the global network while the collecting workers only explore the environment to collect the transitions. Each worker has an actor to choose the action according to its current policy. Each synchronous worker also has a target critic and a target actor used to calculate the target Q-value of each transition. There is a shared replay memory that stores the tranistions from all the workers. When the worker sends one transition to the replay memory, it also computes the priority of this transition based on its copy of critic.

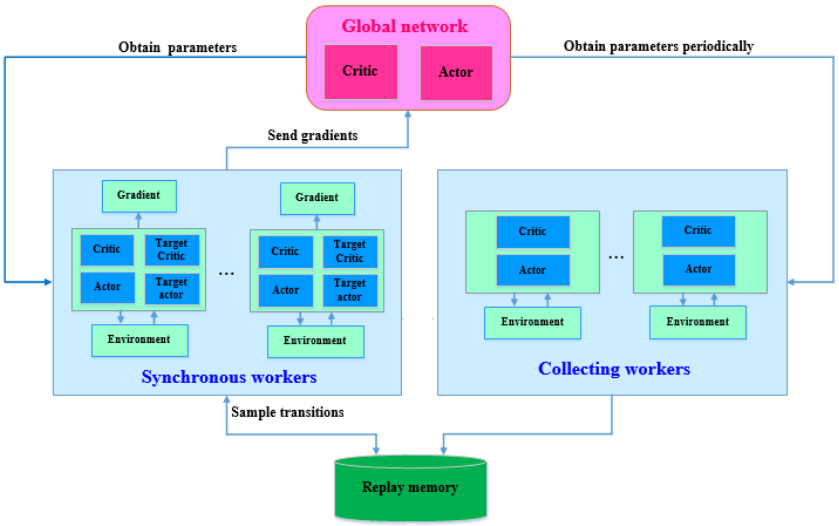

Figure 4.16: Framework of distributed deep deterministic policy gradient

In the learning process, each synchronous worker samples a mini-batch of transitions from the shared replay memory according to the priorities, and then it needs to calculate the gradients of the actor and critic loss based on the sampled transitions. The computation procedures are the same as DDPG. Instead of updating the parameters of the synchronous workers directly, the gradients are sent to the global network to update the global parameters. The global network only has an actor and a critic since it doesn't need the target networks to calculate the losses by itself. The global network needs to calculate the average of the gradients from all the synchronous workers and apply this average gradients to update its parameters. Then the updated parameters of the global

network are synchronized to all the synchronous workers. So the synchronous workers always explore the environment with the latest parameters of the global network. After the update of synchronous workers, the priorities of the transitions sampled by each synchronous worker also need to be updated according to the new network parameters. The collecting workers only explore the environment to generate the transitions without sampling the transitions from the shared replay memory to compute gradients. Furthermore, the collecting workers just obtain the latest parameters from the global network after some time steps rather than every time step. Therefore, the collecting workers may explore the environment by using different policies. The pseudocodes of the global network, one synchronous worker and one collecting worker of distributed DDPG are shown in Algorithm 4, Algorithm 5 and Algorithm 6, respectively.

Algorithm 4 Global network of distributed DDPG

Initialize:
Critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with parameters θ^Q and θ^μ randomly
for $episode = 1, \ldots, M$ **do**
  for $t = 1, \ldots, T$ **do**
    Collect the critic gradients g_i^Q $(i = 1, 2, \ldots, m)$ with respect to θ^Q of all the synchronous workers
    Calculate the average of the critic gradients from all the synchronous workers, g^Q
    Use the average critic gradient g^Q to update the parameters of the global critic, θ^Q
    Collect the actor gradients g_i^μ $(i = 1, 2, \ldots, m)$ with respect to θ^μ of all the synchronous workers
    Calculate the average of the actor gradients from all the synchronous workers, g^μ
    Use the average actor gradient g^μ to update the parameters of the global actor, θ^μ
  end for
end for

All the workers of distributed DDPG explore the environment in parallel and store their transitions in the shared replay memory, so the speed of collecting data can be faster. We can also increase the capacity of the repay memory to store more transitions, so the dataset we construct can reflect the information of the environment better. The collecting workers could explore the environment

Algorithm 5 Synchronous worker of distributed DDPG

Initialize:
Critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with parameters θ^Q and θ^μ synced from global networks
Target critic $Q(s, a|\theta^{Q'})$ and target actor $\mu(s|\theta^{\mu'})$ with parameters $\theta^{Q'}$ and $\theta^{\mu'}$
A shared replay memory R with capacity C
for $episode = 1, \ldots, M$ **do**
  Get the initial state s_0
  for $t = 1, \ldots, T$ **do**
    Choose an action $a_t = \mu(s_t|\theta^\mu) + \omega_t$ based on the current policy and exploration noise
    Perform the action a_t to its environment instance
    Obtain the reward r_t and the next state s_{t+1} from the environment
    Store the transition (s_t, a_t, r_t, s_{t+1}) in R
    compute the priority of transition based on its network parameters
    Sample N prioritized mini-batch transitions (s_n, a_n, r_n, s_{n+1}) $(n = 1, \ldots, N)$ from R
    Update the target networks by using the soft replace:
    $\theta^{Q'} \leftarrow \varepsilon\theta^Q + (1-\varepsilon)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \varepsilon\theta^\mu + (1-\varepsilon)\theta^{\mu'}$ $(\varepsilon \ll 1)$
    Compute the target Q-value for each transition according to Eq.3.5
    Compute the gradient of critic loss g_i^Q with respect to θ^Q:
    $g_i^Q = \nabla_{\theta^Q} L(\theta^Q) \approx \frac{1}{N}\sum_{n=1}^{N} \nabla_{\theta^Q}\left[\left(Q(s_n, a_n|\theta^Q) - y_n\right)^2\right]$
    Send the gradient g_i^Q to the global critic network
    Compute the gradient of actor loss g_i^μ with respect to θ^μ:
    $g_i^\mu = \nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{n=1}^{N}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_n, a=\mu(s_n|\theta^\mu)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_n}\right]$
    Send the gradient g_i^μ to the global actor network
    Wait until the global networks finish update
    Obtain the latest network parameters θ^Q and θ^μ from the global networks
    Update the priorities of the sampled transitions
  end for
end for

with different policies and exploration noises, which can enhance the diversity of the transitions in the replay memory. Using multiple workers with varying policies helps to achieve the better exploration to the environment and discover more expected actions. It should be noted that the collecting workers are separate from the global network since they don't need to wait for the update of global network. Therefore, they can explore the environments all the time during the training process. This separation can help to promote the data throughput.

Algorithm 6 Collecting worker of distributed DDPG

Initialize:
    Critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ randomly
    A shared replay memory R with capacity C
for *episode* $= 1, \ldots, M$ **do**
    Get the initial state s_0
    for $t = 1, \ldots, T$ **do**
        Choose an action $a_t = \mu(s_t|\theta^\mu) + \omega_t$ based on the current policy and exploration noise
        Implement the action a_t to its environment instance
        Obtain the reward r_t and the next state s_{t+1} from the environment
        Store the transition (s_t, a_t, r_t, s_{t+1}) in R
        Compute the priority of transition based on its network parameters
        Obtain the latest network parameters θ^Q and θ^μ from the global networks after a certain number of steps
    end for
end for

Since all the synchronous workers generate the transitions with the latest parameters, it could also guarantee the enough ratio of the transitions produced by the latest policy in the replay momery. The shared replay memory ensures that the transitions with high priorities discovered by any worker could benefit the learning of the global network. The update of global parameters doesn't only depend on one worker since it needs the gradients from all the synchronous workers, which means the global network can exploit more transitions with relatively high priorities at each update step. It can also help to increase the gradient computation speed by allocating to the different workers. With this method, the agent can learn faster and more effectively.

4.2.2 Tracking control of robots via distributed DDPG

In this section, we will use distributed DDPG to train the SCARA robot and mobile robot to track the given trajectories which are the same as those used in DDPG for comparison.

Tracking results of SCARA robot

The hyperparameters of distributed DDPG are the same as DDPG except that the capacity of shared replay memory are extended to 200000 due to the faster collecting speed. We use multi-process to run the multiple workers at the same time. A 8-core CPU is used to train the agent, so we could create 8 workers totally, four of which are the synchronous workers and four are the collecting workers. Each synchronous worker still samples a mini-batch of 32 transitions from the shared replay memory to compute the gradients. Therefore, 128 transitions are sampled by all the four synchronous workers to update the global networks for one time. The collecting workers just obtain the latest parameters from the global network every 5 time steps.

We still train the SCARA robot for 1000 episodes. The episode rewards of the synchronous workers are shown in Fig. 4.17 since only synchronous workers have the latest parameters of the global network. Fig. 4.18 presents the comparison of episode rewards between the single-worker DDPG and one synchronous worker of the distributed DDPG. It can be seen that the agent trained by distributed DDPG can learn a little faster than DDPG.

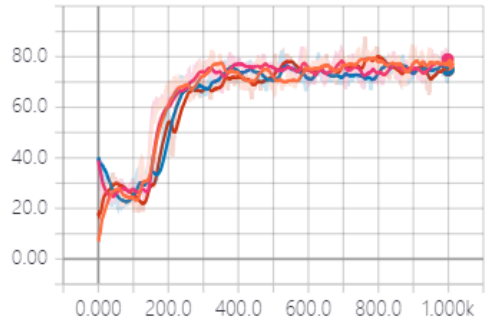

Figure 4.17: Episode rewards of synchronous workers

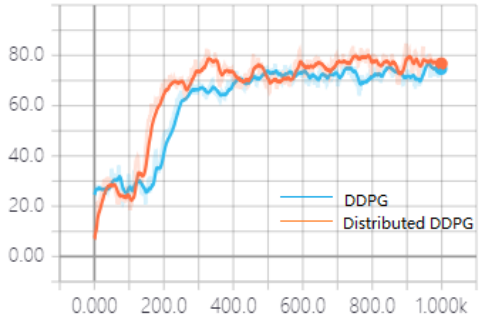

Figure 4.18: Comparison of episode rewards

Fig. 4.19 $\sim$ Fig. 4.23 presents the tracking results and tracking errors of the SCARA robot. We compare the tracking errors with DDPG as shown in Fig. 4.24. We also compute the sum of absolute value of the tracking errors of all the time steps in three dimensions and the results are recorded in Table 4.3. So SCARA robot trained by distributed DDPG could complete the

tracking task with smaller errors in three dimensions than the single-worker DDPG.

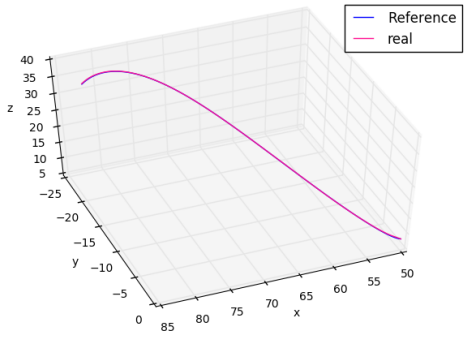

Figure 4.19: Referenced path and real path

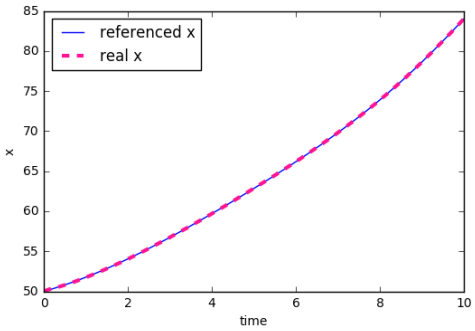

Figure 4.20: Referenced and real x position

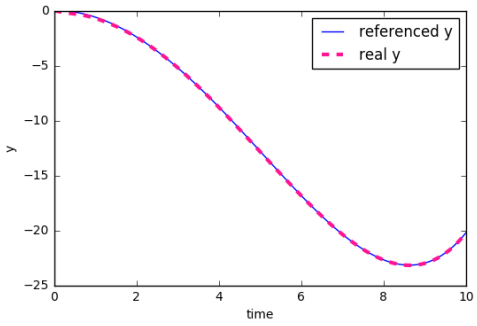

Figure 4.21: Referenced and real y position

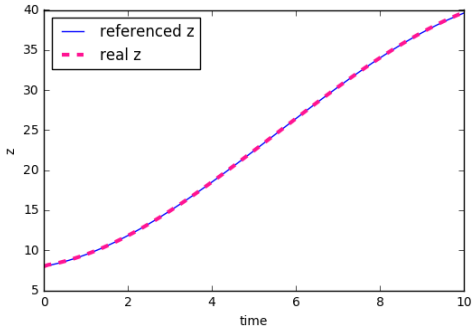

Figure 4.22: Referenced and real z position

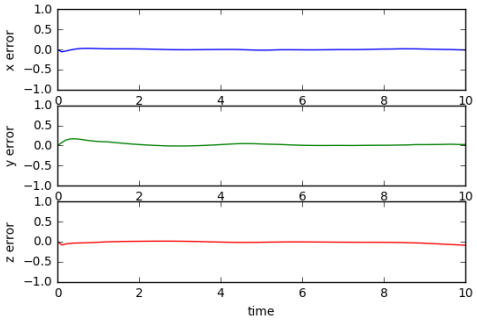

Figure 4.23: Tracking errors in three dimensions

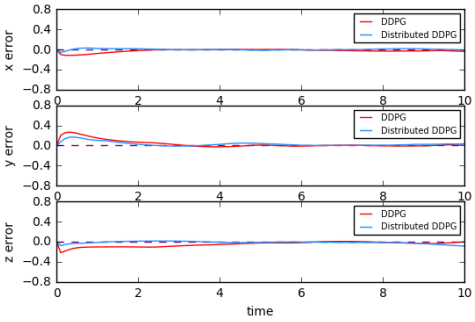

Figure 4.24: Comparison of tracking errors

Table 4.3: Tracking errors of SCARA robot in three dimensions

Methods	x error	y error	z error
DDPG	2.585	4.210	5.389
Distributed DDPG	0.988	3.944	2.081

Tracking results of mobile robot

We still train the mobile robot to follow the two trajectories given in chapter 3 to further prove the effectiveness of the distributed DDPG. Fig. 4.25 and Fig. 4.26 presents the episode rewards of the synchronous workers and the reward comparison between the single-worker DDPG and distributed DDPG. We can see more obviously that the agent trained by the distributed DDPG could learn faster than the single-worker DDPG since its episode reward goes up more rapidly.

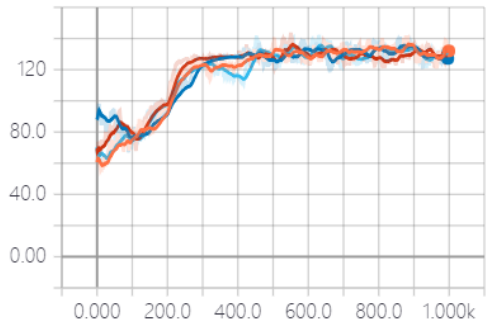

Figure 4.25: Episode rewards of synchronous workers

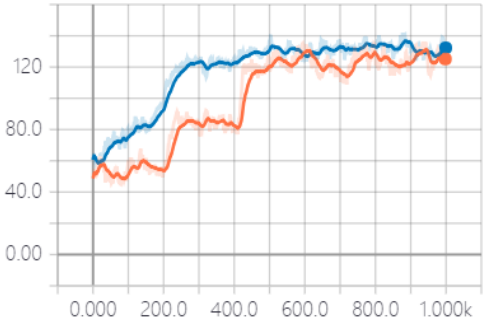

Figure 4.26: Comparison of episode rewards

The tracking results and error comparison of the first trajectory are shown in Fig. 4.27 $\sim$ Fig. 4.32. Table 4.4 gives the summation of absolute value of tracking errors of all the time steps in three dimensions.

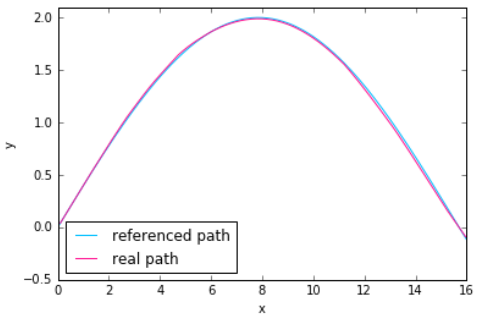

Figure 4.27: Referenced path and real path

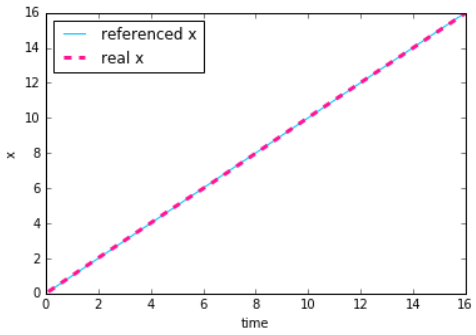

Figure 4.28: Referenced and real x position

Similarly, the training results of the second trajectory of the mobile robot, including the episode rewards and tracking results, are presented in Fig. 4.33 $\sim$

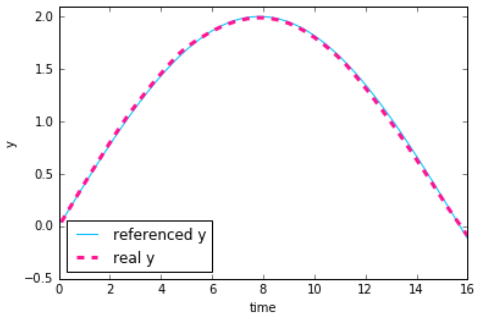

Figure 4.29: Referenced and real y position

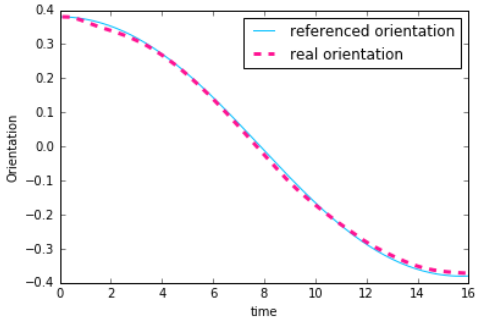

Figure 4.30: Referenced and real orientation

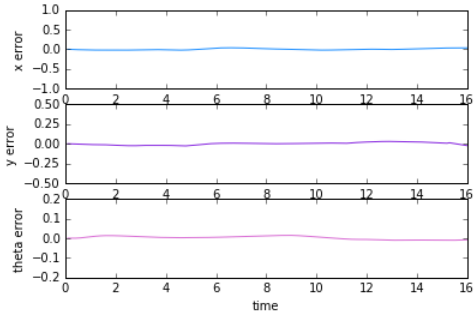

Figure 4.31: Tracking errors

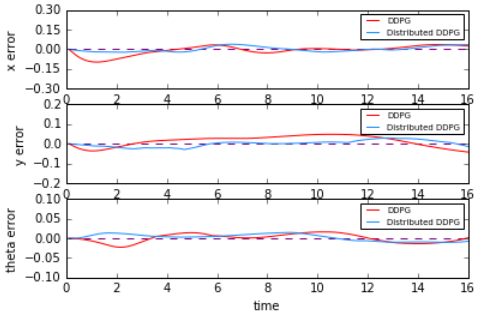

Figure 4.32: Comparison of errors

Table 4.4: Tracking errors of first trajectory in three dimensions

Methods	x error	y error	orientation error
DDPG	6.578	3.495	1.471
Distributed DDPG	2.592	2.915	1.242

Fig. 4.40. The summation of absolute value of tracking errors in three dimensions are presented in Table 4.5. From the simulation results of the two trajectories, we can see that the mobile robot trained by distributed DDPG could track the trajectories with smaller tracking errors. Therefore, by introducing our proposed distributed framework of DDPG, the robot can learn the tracking task faster and achieve smaller tacking errors compared with single-worker DDPG.

Table 4.5: Tracking errors of second trajectory in three dimensions

Methods	x error	y error	orientation error
DDPG	4.267	2.533	5.265
Distributed DDPG	2.837	2.919	3.802

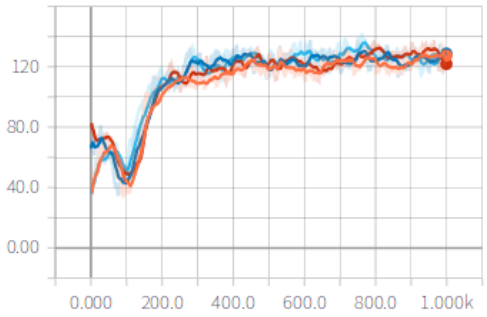

Figure 4.33: Episode rewards of synchronous workers

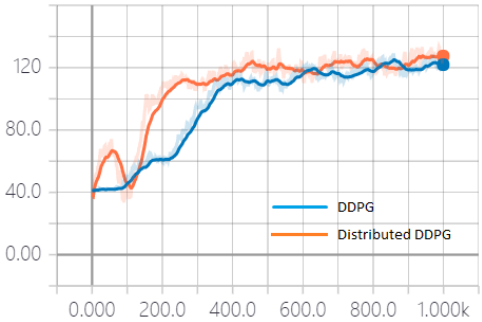

Figure 4.34: Comparison of episode rewards

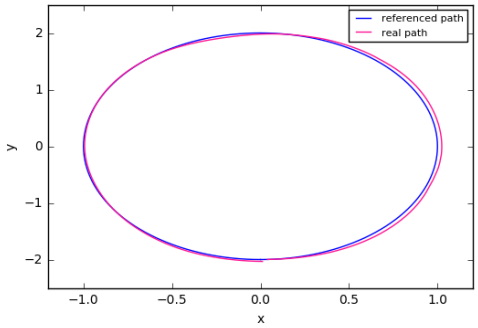

Figure 4.35: Referenced path and real path

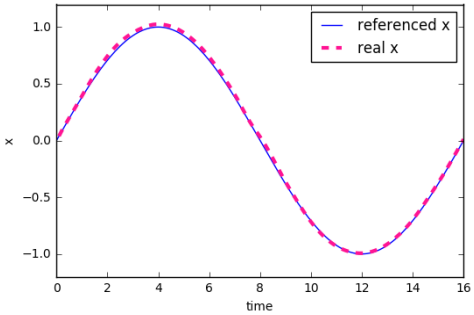

Figure 4.36: Referenced and real x position

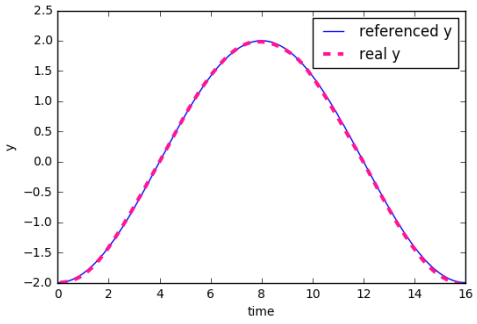

Figure 4.37: Referenced and real y position

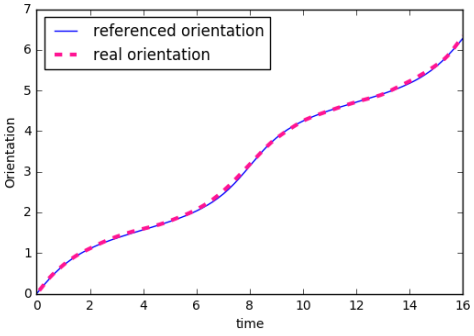

Figure 4.38: Referenced and real orientation

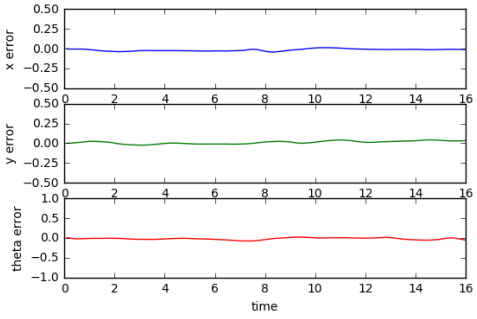

Figure 4.39: Tracking errors

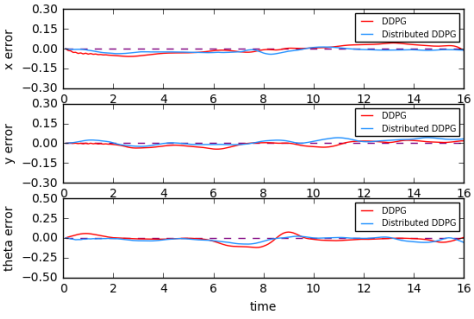

Figure 4.40: Comparison of errors

Chapter 5

Continuous control for robot based on Proximal Policy Optimization

Standard policy gradient method that we have introduced in the previous chapter can only perform one parameter update per data sampling. Hence, OpenAI proposed a new policy gradient method, proximal policy optimization (PPO) [13]that enables multiple updates per mini-batch. PPO has some advantages of trust region policy optimization (TRPO), and is also much simpler to implement. DeepMind employed PPO but with a slightly different objective function to train the agents to learn some locomotion behaviours in rich environments [14].

In this chapter, we adopt a distributed framework of PPO with generalized advantage estimation (GAE) to train the robots. Firstly, we use distributed PPO with GAE to train the redundant manipulator and make comparision with the previous methods. Then we use distributed PPO with GAE to train the mobile robot to track the given trajectories. In order to enhance the training and sample efficiency, we propose a two-stage training strategy consisting of supervised

pre-training and fine-training by distributed PPO. In this way, the mobile robot can be trained for fewer episodes but still get good performance. Then we introduce the long short-term memory (LSTM), to represent the actor and critic. During training, the cell state and hidden state of LSTM are stored in the buffers used for the initialization of each episode to alleviate the problem of inaccurate intial LSTM state. The simulation results demonstrate that distributed PPO with LSTM can improve the tracking performance.

5.1 Proximal Policy Optimization

As stated previously, the gradient of expected return is estimated by using the transitions generated from the current policy in the on-policy reinforcement learning. The transitions can only be used once, after which a new batch of transitions needs to be collected based on the updated policy to estimate the gradients. Therefore, the data efficiency of the on-policy reinforcement learning is low, which can be ameliorated by introducing the importance sampling that we have already applied in the priority replay memory (section 3.2).

Considering a function $f(x)$ whose argument x conforms to the distribution $p(x)$, the expectation of $f(x)$ can be estimated by the mean of $f(x^i)$:

$$\mathbb{E}_{x\sim p(x)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x^i),$$

where x^i is sampled from $p(x)$.

If we can't sample x^i from $p(x)$ and can only sample x^i from another distribution $q(x)$, the expectation is expressed as:

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \int p(x)f(x)dx = \int \frac{p(x)}{q(x)}q(x)f(x)dx = \mathbb{E}_{x\sim q(x)}\Big[\frac{p(x)}{q(x)}f(x)\Big],$$

where $\frac{p(x)}{q(x)}$ is the importance weight to guarantee unbias. Furthermore, in order

to make an accurate estimation, the sampling data from $q(x)$ must be enough.

If we consider the variance $Var_{x\sim p(x)}[f(x)]$ and $Var_{x\sim q(x)}\big[\frac{p(x)}{q(x)}f(x)\big]$:

$$Var_{x\sim p(x)}[f(x)] = \mathbb{E}_{x\sim p(x)}[f(x)^2] - \big(\mathbb{E}_{x\sim p(x)}[f(x)]\big)^2.$$

$$\begin{aligned} Var_{x\sim q(x)}\Big[\frac{p(x)}{q(x)}f(x)\Big] &= \mathbb{E}_{x\sim q(x)}\Big[\big(\frac{p(x)}{q(x)}f(x)\big)^2\Big] - \Big(\mathbb{E}_{x\sim q(x)}\big[\frac{p(x)}{q(x)}f(x)\big]\Big)^2 \\ &= \mathbb{E}_{x\sim p(x)}\Big[f(x)^2\frac{p(x)}{q(x)}\Big] - \big(\mathbb{E}_{x\sim p(x)}[f(x)]\big)^2, \end{aligned}$$

so their variances are not the same. Only if $\frac{p(x)}{q(x)}$ approximates to 1, their variances may not be very different.

By using the importance sampling, we can transform the on-policy to the off-policy:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}\big[A(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\big] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}\Big[\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta'}(s_t,a_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\Big] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}\frac{p_\theta(s_t)}{p_{\theta'}(s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\Big] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\Big], \end{aligned}$$

where $\frac{p_\theta(s_t)}{p_{\theta'}(s_t)}$ can be regarded as 1 since the emergence of a state is not dependent on the policy; π_θ is the policy that needs to be updated; $\pi_{\theta'}$ is the policy which generates the data used to update the policy π_θ.

Since we know $\nabla_x f(x) = f(x)\nabla_x \log f(x)$, we can derive:

$$\begin{aligned} \nabla_\theta\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\Big] &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\Big] \\ &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\big(\nabla_\theta \log \pi_\theta(a_t|s_t) + \\ &\quad \nabla_\theta \log A^{\theta'}(s_t,a_t) - \nabla_\theta \log \pi_{\theta'}(a_t|s_t)\big) \end{aligned}$$

$$= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t),$$

So the corresponding objective function whose gradient is the importance sampling gradient is:

$$J(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}} \big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t) \big]. \tag{5.1}$$

In this way, the same batch of data generated from the policy $\pi_{\theta'}$ can be utilized for multiple times to update the policy π_θ, so the data efficiency can be improved by transforming the on-policy to the off-policy.

If $\pi_{\theta'}$ is treated as the old policy which is expressed as $\pi_{\theta_{old}}$ and π_θ is the current policy to be updated, we hope the probability distributions of the new policy and old policy should not have large difference to guarantee the stability of optimization. Trust region policy optimization (TRPO) could provide an effective approach to this problem, which is defined as follows [12]:

$$\begin{aligned} &\max \quad \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}} \Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^{\theta_{old}}(s_t, a_t) \Big], \\ &s.t. \quad \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}} \big[KL\big(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\big) \big] \leqslant \delta, \end{aligned} \tag{5.2}$$

where KL divergence is used to measure the difference between the new policy and old policy based on the transitions from the old policy. Smaller value of KL divergence means less difference between the new policy and old policy. Hence, TRPO can not only maximize the objective function but also confine the amplitude of policy update to guarantee monotonic improvement of the policy and decrease the fluctuation.

TRPO is solved by using the conjugate gradient method which needs a lot of computation, resulting in the low operation speed. In addition, the conjugate gradient method is difficult to implement since it is much more complicated than the gradient descent.

In order to utilize stochastic gradient descent to update the policy, the constraint in Eq. 5.2 should be transformed to a penalty in the objective function. Thus, the unconstrained optimization problem is: [13]:

$$\max \quad \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A^{\theta_{old}}(s_t,a_t) - \beta KL\big(\pi_{\theta_{old}}(\cdot|s_t),\pi_\theta(\cdot|s_t)\big)\Big], \quad (5.3)$$

where β is a hyperparameter called adaptive KL penalty coefficient to adjust the KL constraint.

β can be adjusted adaptively according to the expectation of KL divergence of the sampled transitions, represented as D [13]:

$$D = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\Big[KL\big(\pi_{\theta_{old}}(\cdot|s_t),\pi_\theta(\cdot|s_t)\big)\Big]. \quad (5.4)$$

A threshold D_{thre} is defined to compare with D and then adjust β:

if $D < D_{thre}/1.5$, $\beta \leftarrow \beta/2$, which is equivalent to attenuate KL constraint;

if $D > 1.5D_{thre}$, $\beta \leftarrow 2\beta$, which is equivalent to enhance KL constraint;

The updated β will work in the following optimization steps.

PPO aims to confine the update amplitude of policy, so the probability ratio of new policy and old policy should approximate to 1. Hence, another objective function of PPO [13] is proposed:

$$L(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{old}}}\Big[\min\big(r_t(\theta)A_t, clip(r_t(\theta),1-\epsilon,1+\epsilon)A_t\big)\Big], \quad (5.5)$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio of the current policy to the old policy; $r_t(\theta)$ is clipped to $1-\epsilon \sim 1+\epsilon$ to remove the change of policy away from 1, which can ensure that the update of policy doesn't have large fluctuation. Then, we choose the minimum of the unclipped term and clipped term so that the final objective function is a lower bound of the unclipped objective. The update of policy is to maximize this objecctive function.

During the update of policy, the starting point is at $r = 1$ where the new policy is the same as the old policy. From Eq. 5.5, we can get that when the advantage is positive, r is clipped at $1 + \epsilon$ and when the advantage is negative, r is clipped at $1 - \epsilon$. Therefore, this objective function can constrain the update amplitude of policy.

5.2 Generalized Advantage Estimation

In the section 5.1,we focus on the improvement of policy update. In this section, we will introduce Generalized Advantage Estimation (GAE) [15] which aims to improve the critic update.

In the policy gradient, if we use the on-policy method to interact with the environment and evaluate the long-term return $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ through the returns of all the time steps, the algorithm may have large fluctuation due to the large variance of gradient. If we adopt the Actor-Critic method which adopts a critic to evaluate the state value, the variance could be reduced but it may increase bias because of the inaccurate model. Therefore, the critical issue is how to make a good balance between the bias and variance.

The objective function of PPO contains the advantage $A(s_t, a_t)$, which can be expressed as:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t). \tag{5.6}$$

The advantage can be estimated by the TD error $\delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$, which is an unbias estimation of $A^{\pi}(s_t, a_t)$:

$$\begin{aligned} \mathbb{E}_{s_{t+1}}[\delta_t] &= \mathbb{E}_{s_{t+1}}[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)] \\ &= \mathbb{E}_{s_{t+1}}[Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)] = A^{\pi}(s_t, a_t) \end{aligned} \tag{5.7}$$

Next, we consider the n-step ($n = 1, 2, 3, \cdots$) advantage estimation, which is represented as $A_t^{(n)}$.

$$
\begin{aligned}
A_t^{(1)} &= \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \\
A_t^{(2)} &= \delta_t + \gamma\delta_{t+1} = r_t + \gamma V(s_{t+1}) - V(s_t) + \gamma\big(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1})\big) \\
&= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t).
\end{aligned}
$$

Then the n-step advantage estimation is expressed as [15]:

$$
A_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k \delta_{t+k} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) - V(s_t). \tag{5.8}
$$

If $n \to \infty$, $\gamma^\infty V(s_{t+\infty}) \to 0$. Then we can get:

$$
A_t^{(\infty)} = \sum_{k=0}^{\infty} \gamma^k \delta_{t+k} = \sum_{k=0}^{\infty} \gamma^k r_{t+k} - V(s_t), \tag{5.9}
$$

which becomes the Monte Carlo method since the return of each step is utilized to estimate the target value.

With the increase of step numbers, the bias of the estimation gradually decreases while the variance gradually increases. If we could consider all these estimations at the same time and calculate their weighted average, we can obtain a better trade-off between the bias and variance, which is the idea of GAE computed as follows [15]:

$$
\begin{aligned}
A_t^{GAE} &= (1-\lambda)(A_t^{(1)} + \lambda A_t^{(2)} + \lambda^2 A_t^{(3)} + \ldots) && (5.10) \\
&= (1-\lambda)\big(\delta_t + \lambda(\delta_t + \gamma\delta_{t+1}) + \lambda^2(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}) + \ldots\big) \\
&= (1-\lambda)\big((1 + \lambda + \lambda^2 + \ldots)\delta_t + (\lambda + \lambda^2 + \lambda^3 + \ldots)\gamma\delta_{t+1} \\
&\quad + (\lambda^2 + \lambda^3 + \lambda^4 + \ldots)\gamma^2\delta_{t+2} + \ldots\big) && (5.11) \\
&= (1-\lambda)\Big(\frac{1}{1-\lambda}\delta_t + \frac{\lambda}{1-\lambda}\gamma\delta_{t+1} + \frac{\lambda^2}{1-\lambda}\gamma^2\delta_{t+2} + \ldots\Big)
\end{aligned}
$$

$$= \delta_t + (\lambda\gamma)\delta_{t+1} + (\lambda\gamma)^2\delta_{t+2} + \dots$$
$$= \sum_{k=0}^{\infty}(\lambda\gamma)^k\delta_{t+k},$$

where another hyperparameter λ is introduced.

When $\lambda = 0$, $A_t^{GAE} = \delta_t = A_t^{(1)}$, it is reduced to the TD error, which has a low variance but high bias;

When $\lambda = 1$, $A_t^{GAE} = \sum_{k=0}^{\infty}\gamma^k\delta_{t+k} = \sum_{k=0}^{\infty}\gamma^k r_{t+k} - V(s_t) = A_t^{(\infty)}$, it is equivalent to the Monte Carlo method, which has a low bias but high variance.

Therefore, we could adjust the value of λ to make a better balance between the bias and variance. Then the advatage A_t in the objective function of PPO could be replaced by A_t^{GAE}.

In the real application, we can only use a truncated version of GAE since the interactive sequences are finite. If we collect a sequential transitions with T time steps, the GAE is reduced to:

$$A_t^{GAE} = \delta_t + (\lambda\gamma)\delta_{t+1} + (\lambda\gamma)^2\delta_{t+2} + \dots + (\lambda\gamma)^{T-t-1}\delta_{T-1}. \tag{5.12}$$

5.3 Distributed framework of PPO

In chapter 4, we have introcuced some distributed frameworks of deep reinforcement learning, which can achieve better performance in various control tasks. In this section, we propose a distributed framework of PPO, which is shown in Fig. 5.1.

Multiple workers are created and run on the different threads. Each worker has an instance of the environment and an actor to choose the action according to the current policy and state. The worker explores its own environment and collects a sequence of transitions which are stored in its local replay buffer.

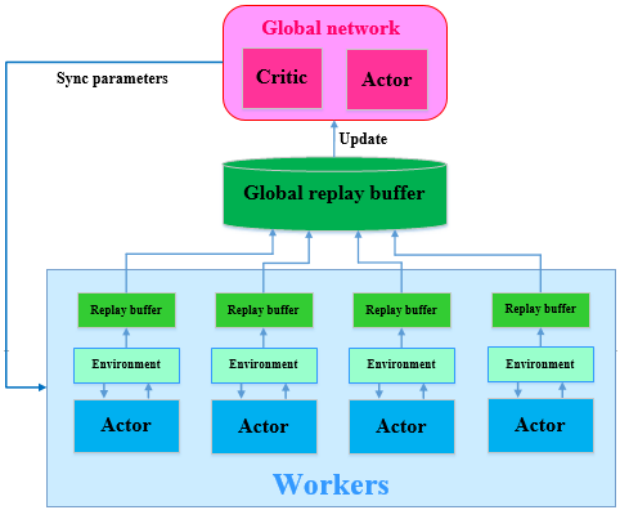

Figure 5.1: Distributed framework of PPO

After the worker collects N transitions, it sends these transitions to the global replay buffer and stops interacting with the environment. If we have K workers, the global replay buffer will have NK transitions at each iteration. Then the global network will use all the transitions or sample a mini-batch of transitions $B \leq NK$ from the global replay buffer to construct the objective function expressed in Eq. 5.3 or 5.5 and compute the gradients to update the global parameters for some times. The updated parameters of global network will be synchronized to all the workers, after which the workers continue to interact with the environment by using the latest policy to collect transitions for the next iteration. This distributed framework is different from the ones in chapter 4 because the workers only collect transitions for the global network without computing the gradients.

The pseudocodes of the worker and global network of distributed PPO are shown in Algorithm 7 and Algorithm 8 respectively. We use the mean of sampled transitions to approximate the expectation. The actor and critic are updated for several times by using the transitions from the global replay buffer at each collection. The target value R_t is required when calculating the critic loss. R_t is also estimated by the weighted sum of n-step return with hyperparameter λ:

$$R_t^{(1)} = r_t + \gamma V(s_{t+1}),$$

$$R_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}),$$
$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}).$$

Then the estimated target value R_t can be expressed as follows:

$$R_t(\lambda) = (1-\lambda)(R_t^{(1)} + \lambda R_t^{(2)} + \lambda^2 R_t^{(3)} + \ldots) = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}. \quad (5.13)$$

The truncated version of R_t is:

$$R_t(\lambda) = (1-\lambda)\sum_{n=1}^{T-t} \lambda^{n-1} R_t^{(n)}, \quad (5.14)$$

since $R_t^{(T-t)} = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} + \gamma^{T-t} V(s_T)$, where s_T is the final state of the sequence.

Algorithm 7 Worker of distributed PPO

Initialize:
    Global replay buffer R
for *episode* $= 1, \ldots, M$ **do**
    Get the initial state s_0
    for $t = 1, \ldots, T$ **do**
        Choose an action a_t from $\pi_\theta(\cdot|s_t)$ based on the current state s_t and policy of global network
        Implement the action a_t to its environment instance
        Get the reward r_t and the next state s_{t+1} from the environment
        Store s_t, a_t and r_t in its local replay buffer
        if N transitions are collected *or* episode ends **then**
            Compute the generalized advantages of all the transitions $A_1^{GAE}, A_2^{GAE}, \ldots, A_N^{GAE}$ by using Eq. 5.12
            Send the transitions in the local replay buffer and their estimated advantages to the global replay buffer R
            Wait until the global networks complete the parameter update
            Clear the local replay buffer
        end if
    end for
end for

Algorithm 8 Global network of distributed PPO

Initialize:
  Critic $V(s;\theta_v)$ and actor $\pi(s;\theta_\pi)$
while episode < maximum episode **do**
  Wait until the transitions from all the workers are stored in the global replay buffer R
  Assign θ_{old} with the current policy parameters: $\pi_{\theta_{old}} \leftarrow \pi_\theta$
  for $i = 1, 2, \ldots, m$ **do**
    Sample a mini-batch of transitions with size B from R randomly
    Construct the clip objective function by using the sampled transitions:
    $L_{clip} = \frac{1}{B}\sum_{t=1}^{B}\left[\min\left(\frac{\pi_{\theta_\pi}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, clip(\frac{\pi_{\theta_\pi}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)A_t\right)\right]$
    Compute the gradient $\nabla_{\theta_\pi} L_{clip}$ and update the parameters of actor
  end for
  for $j = 1, 2, \ldots, n$ **do**
    Sample a mini-batch of transitions with size B from R randomly
    Construct the critic loss by using the sampled transitions:
    $L_{critic} = \frac{1}{B}\sum_{t=1}^{B}\left(R_t - V_{\theta_v}(s_t)\right)^2$
    Compute the gradient $\nabla_{\theta_v} L_{critic}$ and update the parameters of critic
  end for
  Start the worker threads
end while

5.4 Training results of redundant manipulator

In this section, we use distributed PPO with GAE to train the redundant manipulator to achieve the multi-goal task. The architecture of actor and critic is same as the one used in A3C and A2C (Fig. 4.3).

We also create four workers running on the four threads and train the redundant manipulator for 2000 episodes in total by using the ordinary reward and path optimization reward respectively. Each worker collects 8 transitions and sends these transitions to the global network. Then the global network will update its parameters by using the 32 transitions from all the workers for 5 times. Figure 5.2 $\sim$ Fig. 5.5 show the the overall episode rewards and episode rewards of different workers using the two reward functions, respectively.

The success rate of every 100 episodes of the two reward functions are recorded in Fig. 5.6. After training for 2000 episodes, the redundant manipulator trained

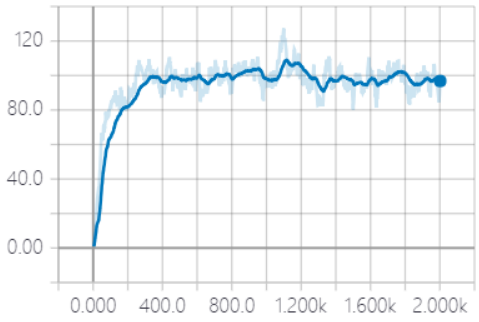

Figure 5.2: Overall episode rewards

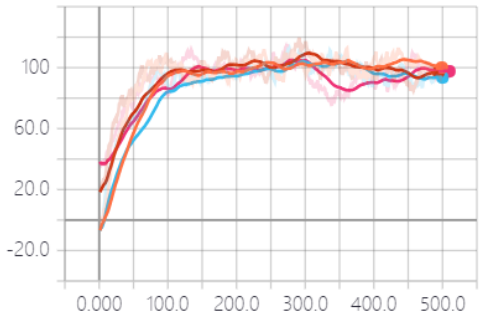

Figure 5.3: Episode rewards of workers

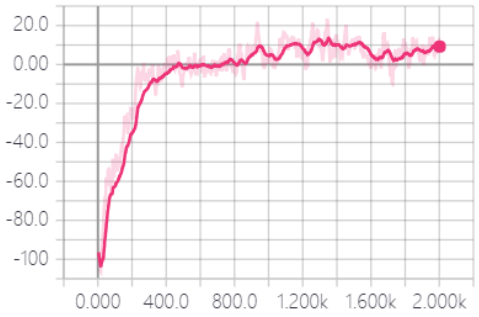

Figure 5.4: Overall episode rewards

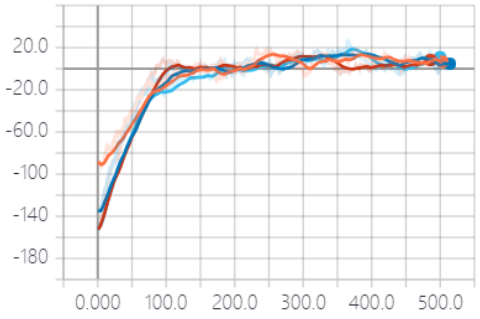

Figure 5.5: Episode rewards of workers

by the ordinary reward could complete the task with a success rate of nearly 80%, and the one trained by the path optimization reward could realize nearly 90% success rate. The extra distance of the two reward functions are depicted in Fig. 5.7, which again prove that the path optimization reward could optimize the reaching path of the end-effector.

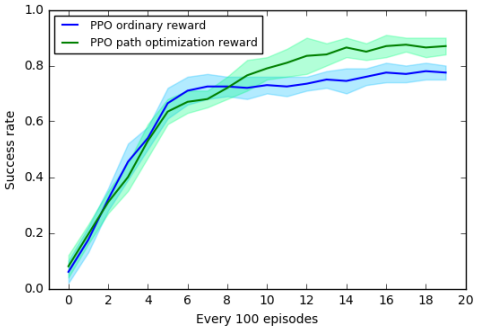

Figure 5.6: Success rate

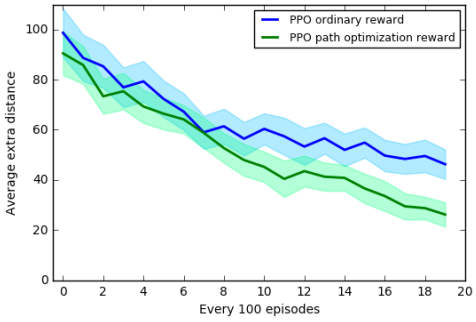

Figure 5.7: Average extra distance

We still test the agent trained by distributed PPO for 100 episode and the testing results are recorded in Table 5.1, from which we can see that the agent trained by distributed PPO could get a little better performance than A2C in terms of the success rate.

Table 5.1: Testing results of distributed PPO for the redundant manipulator

Methods	Success times
Distributed PPO with ordinary reward	82
Distributed PPO with path optimization reward	91

The hyperparameters of distributed PPO for the redundant manipulator are shown in Table 5.2.

Table 5.2: Hyperparameters of Distributed PPO for manipulator

Hyperparameters	Value
Learning rate of actor α_π	0.0001
Learning rate of critic α_v	0.0005
Clip parameter ϵ	0.2
Reward discounted factor γ	0.9
GAE parameter λ	0.95
Standard deviation σ	[0.8,0.8,0.1]

5.5 Tracking control of mobile robot based on distributed PPO

In this section, distributed PPO is employed to train the mobile robot to track the trajectories. The input of the actor is the observation vector which incorporates the position and velocity information of the mobile robot. The outputs of the actor are the means of the forward velocity and rotation speed of the mobile robot measured in its local frame. The stanadrd deviations are the hyperparameters to encourage the exploration of agent. The architecture of the actor and critic is presented in Fig. 5.8. All the hidden layers of actor and critic have 256 neurons.

We add some extra improvements to the training strategies, random referenced state initialization and early termination, introduced in subsection 3.5.2, in order to achieve better tracking performance. We adopt the piecewise random referenced state initialization, which means the robot is initialized to the differ-

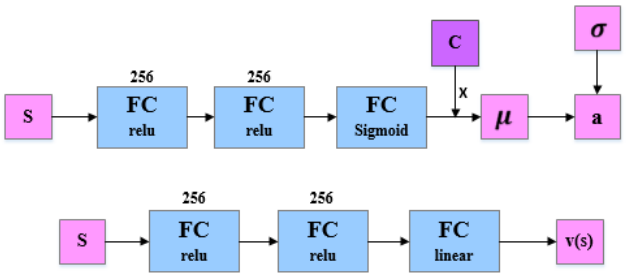

Figure 5.8: Architecture of actor and critic

ent parts of the renferenced trajectory during different training stages, since we hope the robot could start to move from the former part of trajectory in the later training stage to collect more transitions of previous time steps and further update the policy to fit the whole trajectory. For the early termination, we reduce the upper limit of the counter with the training process in order to collect fewer unexpected states and actions, since the agent should learn more from the expected transitions to fine tune the policy at the later stage of training.

We create four workers running on the four threads to collect transitions for the global network. The mobile robot is trained for totally 3000 episodes. Because the workers firstly need to collect enough transitions and then the global network samples a mini-batch of transitions to update its parameters for multiple times, it is not a one-step update as DDPG. During the training process, each worker explores the environment and collects the transitions with the latest policy of the global network. After the worker collects 16 tranistions, it sends these transitions to the global replay buffer and then stops runing. When all the workers finish collecting the transitions and sending them to the global replay buffer, the global network will randomly sample a mini-batch of 32 transitions from the 64 transitions collected by all the four workers at each update. The global parameters are updated for 8 times by sampling from the 64 transitions, after which these transitions will be discarded. And then the workers continue to explore the environment to collect new transitions with the updated policy. This distributed framework can increase the collecting speed by running multiple workers at the same time, and the random sampling can reduce the correlation of transitions during update to ensure the convergence of networks.

During $0 \sim 2000$ episode, the mobile robot is initialized at any referenced state of the whole trajectory and the maximum value of the early termination counter is 20. During $2000 \sim 2500$ episode, the mobile robot is initialized at the referenced state from $\frac{1}{4} \sim \frac{1}{2}$ of the trajectory and the maximum value of the early termination counter is 15. During $2500 \sim 3000$ episode, the mobile robot is initialized at the referenced state from the first quarter of the trajectory and maximum value of the early termination counter is 10.

For the first trajectory, the overall episode rewards and episode rewards of different workers are presented in Fig. 5.9 and Fig. 5.10.

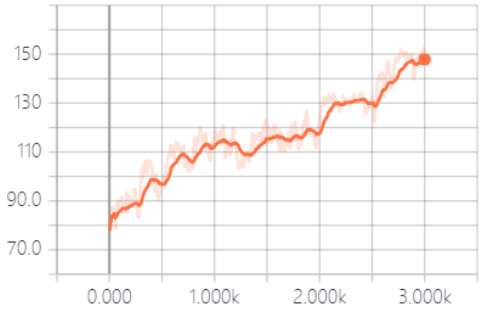

Figure 5.9: Overall episode rewards

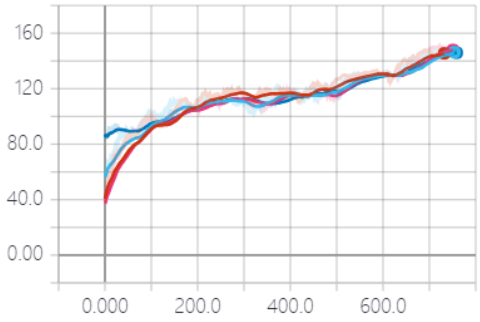

Figure 5.10: Episode rewards of workers

After training for 3000 episodes, we save the parameters of the global network. In the testing stage, only the means of actions restored from the saved model are used without the standard deviation. Here, we only present the tracking errors in three dimensions as shown in in Fig. 5.11.

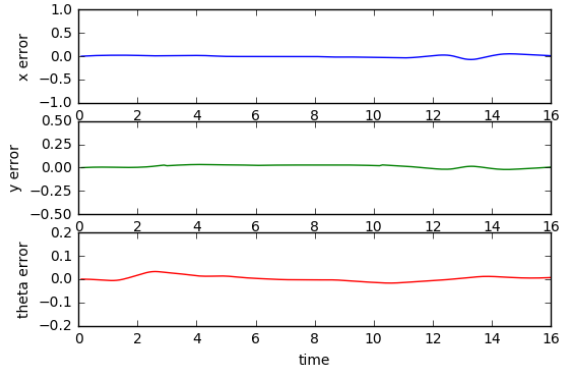

Figure 5.11: Tracking errors

For the second trajectory, the overall episode rewards and episode rewards of

different workers are presented in Fig. 5.12 and Fig. 5.13. The tracking errors are shown in Fig. 5.14.

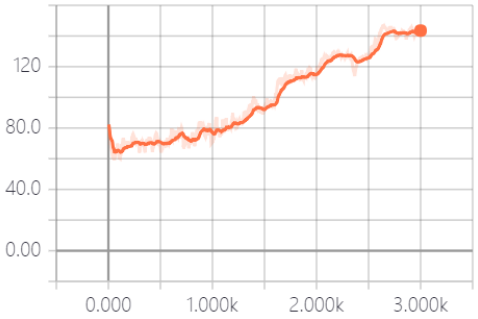

Figure 5.12: Overall episode rewards

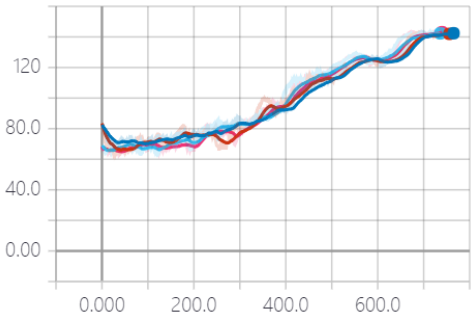

Figure 5.13: Episode rewards of workers

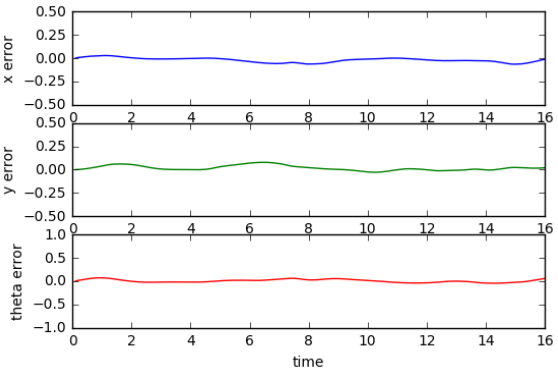

Figure 5.14: Tracking errors

From the tracking results of the two trajectories, we can see that distributed PPO could enable the mobile robot to track the trajectories with small errors, but the workers may need much more episodes in total to collect the transitions in order to realize better performance compared with DDPG.

The hyperparameters of distributed PPO for the mobile robot are shown in Table 5.3.

Table 5.3: Hyperparameters of distributed PPO for mobile robot

Hyperparameters	Value
Learning rate of actor α_π	0.0001
Learning rate of critic α_v	0.0005
Clip parameter ϵ	0.2
Reward discounted factor γ	0.9
GAE parameter λ	0.95
Standard deviation vector σ	[0.1,0.01]

5.6 Two-stage training strategy

In the last section, we have used distributed PPO to train the mobile robot to track the given trajectories. However, it needs 3000 training episodes and exploits a lot of trasitions in order to get a good performance. Aiming to promote the training and data efficiency, we introduce a two-stage training strategy for the mobile robot, which consists of the supervised pre-training and fine-training by distributed PPO. The fine-training by using deep reinforcement learning based on the pre-trained policy can further improve the policy according to the actual task objective.

The training process is comprised of two stages. Firstly, supervised learning is used to pre-train the actor. It is known that each training data of the supervised learning needs a label. The input of the actor is the observation and the output is the corresponding action. If we want to use the supervised learning, we need to know the target action for each observation. However, it may be difficult to calculate the exact action that could make the agent get to the referenced state at next time step, especially for some complex control tasks. Thus, we adopt an action search method which randomly generates some actions within the prescribed range. Each action is assumed to be performed at the current state and the next state is estimated by using the known kinematic relationship. After getting the next state for each action, we can calculate its reward by comparing with the referenced state. Then we choose the action which has the maximum reward as the target action for the current state. We need to search sufficient number of actions so that we could find a good enough action that has high reward. After collecting a certain number of training data, the actor will be trained by these data using stochastic gradient descent. After that, we still use distributed PPO to further train the agent with the initial policy obtained by the pre-training process. The pre-training process can help the agent to learn some useful experiences quickly, so the agent doesn't need to be trained from scratch by distributed PPO.

The pre-training process consists of 200 episodes. We still use the random referenced state initialization and early termination during the pre-training process. At each visited state, 200 random actions are generated. Every action is hypothetically performed to the environment and the next state is estimated through the kinematics. Then, the action which has the maximum reward is chosen. The real action taken by the agent is obtained from the current policy and the reward from this real interaction needs to be compared with the selected maximum reward from the action search. If the maximum reward is greater than the real reward, the corresponding action will be treated as the target action and the state-action pair will be collected as one training data, which guarantees the target action is better than the action from the current policy. The same process will be repeated in the following visited states of the agent. Therefore, for each visited state, a target action can be obtained. After collecting 100 training data, stochastic gradient descent with a mini-batch size of 50 is conducted to update the parameters of the actor. These 100 training data will be gone through for 5 times and the training data need to be shuffled before each epoch. Then these training data will be discarded and new data will be collected by using the same method. The learning rate we use for the gradient descent optimizer is 0.1.

The critic also needs to be updated with the imporvement of policy. After collecting some sequential transitions, the target value of every state is calculated by using Eq. 5.14. So for each visited state, we could get its target value and this state-value pair is stored in a replay buffer. After updating the policy, the update of critic is followed by sampling a mini-batch of 32 pairs randomly from the replay buffer for 5 times. Then the replay buffer is cleaned out and new data will be collected based on the new policy.

The pre-training of the actor and critic is only for the global network, which means the global network should have its own worker to explore the environment in the main thread while the other workers in the child threads don't run during the pre-training process. After pre-training, workers in the child threads start to explore their environments and collect transitions for the global network. Then

the actor and critic network are further updated by using distributed PPO. The simplified pre-training pseudocode is shown in Algorithm 9.

Algorithm 9 Pre-training for the actor and critic

Initialize:
    Critic $V(s;\theta_v)$ and actor $\pi(s;\theta_\pi)$
for $episode = 1, \ldots, M$ **do**
    Get the initial state s_0
    for $t = 1, \ldots, T$ **do**
        Obtain an action a_t from the current policy $\pi_\theta(\cdot|s_t)$
        Perform the action a_t and then get the reward r_t and the next state s_{t+1}
        Randomly generate sufficient number of actions within the given range
        Perform every action to the state s_t and then evaluate the next state s_{t+1} and reward r_t
        Choose the action a'_t that obtains the maximum reward r_{max}
        if $r_{max} > r_t$ **then**
            Treat a'_t as the target action at the state s_t
            Store (s_t, a'_t) to the dataset D
        end if
        Calculate the target value R_t of the state s_t by using Eq. 5.14
        Store (s_t, R_t) to the replay R
        if the size of dataset D gets to the prescribed value **then**
            Update the actor by sampling a mini-batch data from D for multiple times
            Update the critic by sampling a mini-batch data from R for multiple times
            Clean out dataset D and replay R
        end if
    end for
end for

Firstly, we use this two-stage training strategy to train the mobile robot to track the first trajectory. The episode rewards of pre-training for the actor are depicted in Fig. 5.15, where we can see that the reward goes up quickly during the supervised training period. After 200 episodes, the reward can increase to a relatively high value.

Fig. 5.16 and Fig. 5.17 shows the overall episode rewards and episode rewards of different workers trained by distributed PPO. The mobile robot is only trained for 2000 episodes after the supervised training. The training strategy is similar,

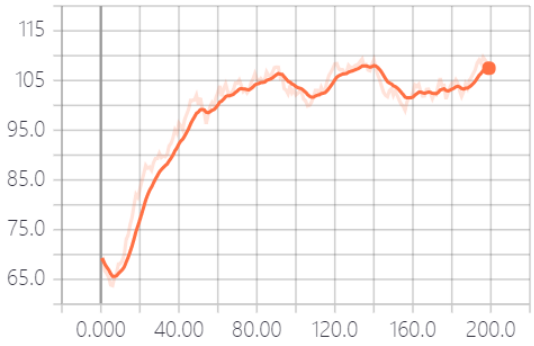

Figure 5.15: Episode rewards of pre-training for actor

with 1000 episodes starting from any referenced state and 1000 episodes starting from $\frac{1}{4} \sim \frac{1}{2}$ and first quarter of the referenced trajectory. We can see that the initial reward after pre-training is much higher than the previous one without pre-training, so the training episodes by distributed PPO can be reduced, which can help to promote the sample and training efficiency. The tracking errors of the mobile robot in three dimensions are presented in Fig. 5.18.

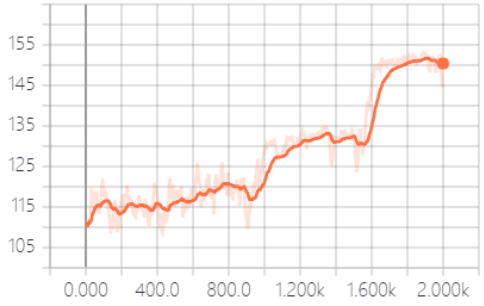

Figure 5.16: Overall episode rewards

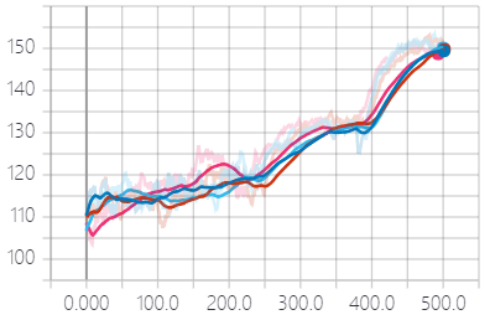

Figure 5.17: Episode rewards of workers

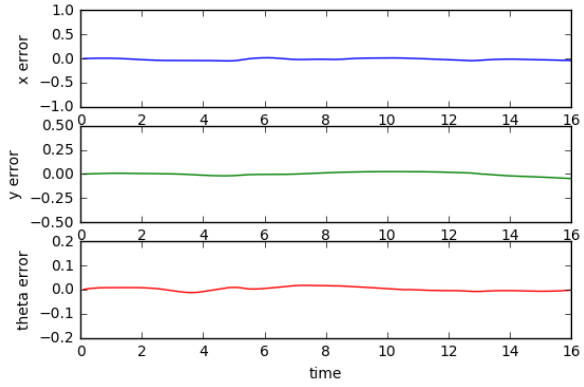

Figure 5.18: Tracking errors

Next, we train the mobile robot to track the second trajectory. The episode

rewards of pre-training for the actor are shown in Fig. 5.19. The episode rewards trained by distributed PPO are presented in Fig. 5.20 and Fig. 5.21. The tracking errors in three dimensions are shown in Fig. 5.22. From these figures, we can get the same conclusion as the first one.

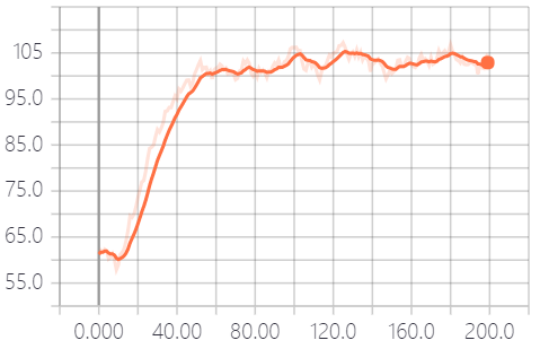

Figure 5.19: Episode rewards of pre-training for actor

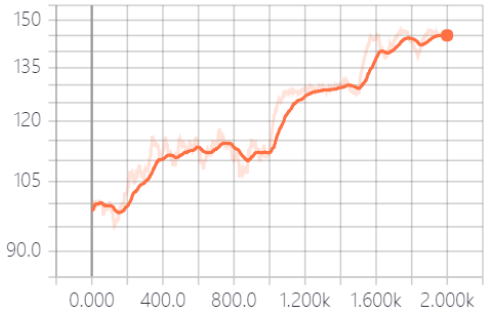

Figure 5.20: Overall episode rewards

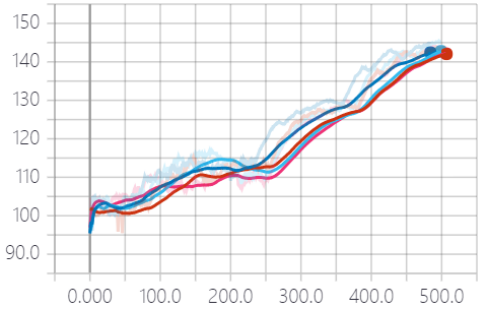

Figure 5.21: Episode rewards of workers

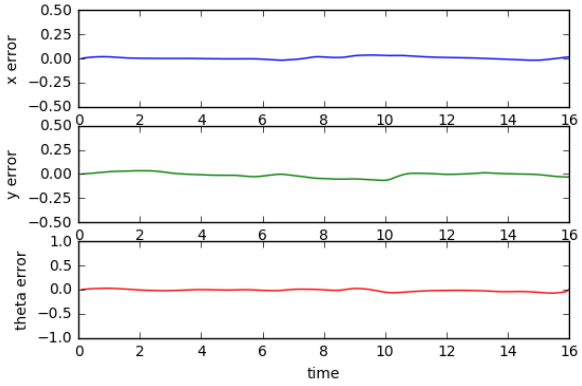

Figure 5.22: Tracking errors

We compare the tracking errors between the distributed PPO and the two-stage training strategy(Table 5.4 and Table 5.5). The final results of the two-stage training strategy are a little better than the distributed PPO.

Table 5.4: Tracking errors of first trajectory in three dimensions

Methods	x error	y error	orientation error
Distributed PPO	2.998	2.961	1.495
Two-stage training	3.276	2.293	1.147

Table 5.5: Tracking errors of second trajectory in three dimensions

Methods	x error	y error	orientation error
Distributed PPO	3.906	3.683	4.451
Two-stage training	1.857	3.213	3.643

By introducing the supervised pre-training for the mobile robot, we can reduce the training episodes by distributed PPO and still get good tracking performance. Therefore, the two-stage training strategy could help to decrease the number of interaction steps of the agent to improve the sample and training efficiency.

5.7 Distributed PPO with LSTM

In the previous sections, we just use fully connected neural network to represent the actor and critic. In this section, we will introduce the recurrent neural network (RNN) to the actor and critic for the mobile robot tracking task.

Traditional RNN can be influenced by the short-term memory. If the sequence is long enough, RNN may have difficulty to transmit the information from the early time step to the subsequent time step. Moreover, during the backpropagation through time process, RNN has the problem of gradient vanishing, resulting in the early stop of learning. Therefore, in our simulation, we utilize the most popular improved RNN, Long Short-term Memory (LSTM) [67] which could solve the problem of short-term memory and gradient vanishing. Many works [11, 21, 35, 68] of deep reinforcement learning use LSTM as the representation of the actor and critic, especially for some tasks with only partially-observable states.

The structure of one LSTM cell is shown in Fig. 5.23. The cell has three gates: forget gate, input gate and output gate. The internal states of LSTM cell comprises of the cell state c_t and the hidden state h_t. The forget gate determines how much information of the cell state should be discarded. It takes x_t and the hidden state of last time step h_{t-1} as the input and outputs the value between $0 \sim 1$. 1 denotes complete reservation while 0 denotes complete discard for the cell state information. The forget gate performs the following calculation [69]:

$$\boldsymbol{f}_t = sigmoid(\boldsymbol{W}_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f).$$

The input gate decides on how much new information will be added to the cell state. The computation procedures are as follows [69]:

$$\boldsymbol{i}_t = sigmoid(\boldsymbol{W}_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i),$$

$$\hat{\boldsymbol{c}}_t = tanh(\boldsymbol{W}_c \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c),$$

where $\hat{\boldsymbol{c}}_t$ is the candidate cell state.

The new cell state is calculated as follows [69]:

$$\boldsymbol{c}_t = \boldsymbol{f}_t * \boldsymbol{c}_{t-1} + \boldsymbol{i}_t * \hat{\boldsymbol{c}}_t.$$

The output gate determines the next hidden state. The computation procedures are as follows [69]:

$$\boldsymbol{o}_t = sigmoid(\boldsymbol{W}_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o),$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t * tanh(\boldsymbol{c}_t).$$

Then the new cell state c_t and hidden state h_t will be transmitted to the next time step.

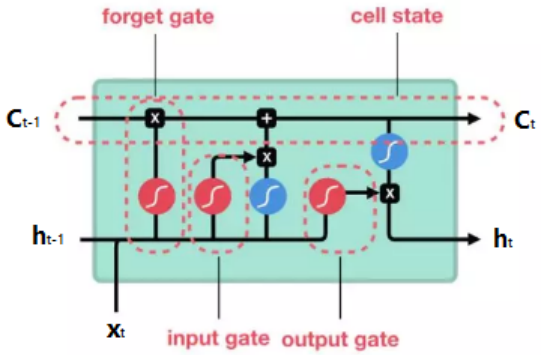

Figure 5.23: Structure of LSTM cell

The cell state of LSTM could effectively reduce the influence of the short-term memory. Theoretically, the cell state can carry the relevant information during the whole sequence process. As the cell state is transmitted over time steps, the information in the cell state can be added or deleted by the gate mechanism. These gates are represented by different neural networks and they can learn what information should be reserved or discarded during the training process.

The architecture of the actor and critic is presented in Fig. 5.24. For both the actor and critic, the observation is firstly processed by a FC layer and then sent to the LSTM. The output of LSTM is further processed by a FC layer, after which we can get the action distribution and value at the specific observation. The first FC layer of the actor and critic has 256 neurons. The size of LSTM is also 256, which means the dimension of the cell state and hidden state is 256.

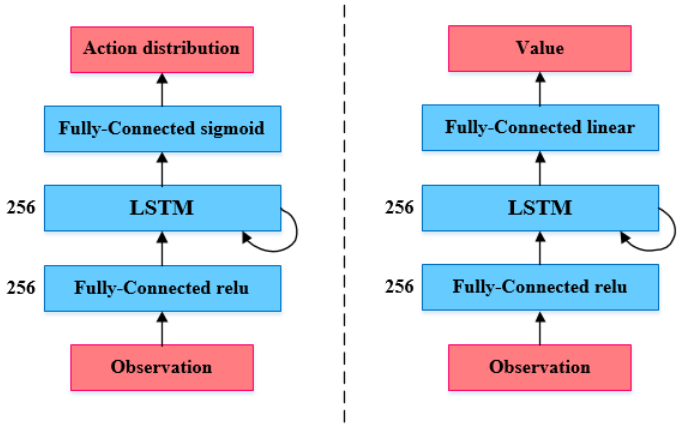

Figure 5.24: Architecture of actor and critic

Four workers are created to run on the four threads of CPU. Each worker explores its own environment to collect the transitions. When the worker utilizes the parameters of the global actor and critic to choose the action and calculate the value, it needs to keep the cell state and hidden state of LSTM, which will be input to the LSTM at the next time step. Therefore, the action and value depend on both the current state and the cell state and hidden state of LSTM at last time step. Each worker needs to collect a series of sequential transitions. We set the number of transitions as 10. However, if the episode ends or early termination is triggered, the number of sequential transitions can be less than 10. The sequential transitions collected by all the workers regarded as a mini-batch are sent to the global network. One series of sequential transitions from one worker is treated as a time sequence input to the global actor and critic to calculate the loss and update the network parameters.

For each sequence, we also need to know the intial cell state and hidden state of LSTM. Some previous works of deep reinforcement learning using LSTM set the intial cell state and hidden state as 0. Zero intial state may allow the decorrelated sampling of the sequences, which contributes to the robust optimization of the neural networks, but it may cause the inaccurate state output of LSTM at the first few time steps. Thus, LSTM may have some difficulties to recover the meaningful predictions from an inaccurate intial state. Aiming at this problem, we store the cell state and hidden state of both the actor and critic at each time step obtained from the interaction process in the buffers for each worker. The stored states can be updated during the training process. Since we still use the random referenced state initialization, we need to search the corresponding cell state and hidden state from the buffers at the starting point of each episode. If the first state of a sequence is not the starting point of an episode, the initial LSTM state of that sequence is the LSTM output state of the last time sequence. In this way, the initial LSTM state of the starting point at each episode is not always zero, which could mitigate the problem of inaccurate LSTM intial state. After collecting a mini-batch of sequences with their intial LSTM states,

the global network will be updated for 8 times with the sequence of each worker used twice.

The mobile robot is trained for 3000 episodes by using distributed PPO with LSTM. The initial position of the mobile robot is randomly chosen from the referenced trajectory at the former 2000 episodes. In the following 400 episodes, it is intialized at $\frac{1}{4} \sim \frac{1}{2}$ of the referenced trajectory and in the next 400 episodes, it starts from the first quarter of the referenced trajectory. In the last 200 episodes, it starts from $t = 0$ and the initial LSTM state at $t = 0$ is zero. During the testing stage, the mobilt robot is also initialized at $t = 0$.

For the first trajectory, Fig. 5.25 and Fig. 5.26 show the change of episode reward. The tracking results are presented in Fig. 5.27 $\sim$ Fig. 5.30 and the tracking errors of three dimensions are shown in Fig. 5.31.

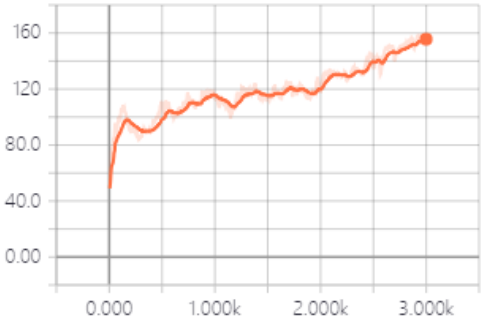

Figure 5.25: Overall episode rewards

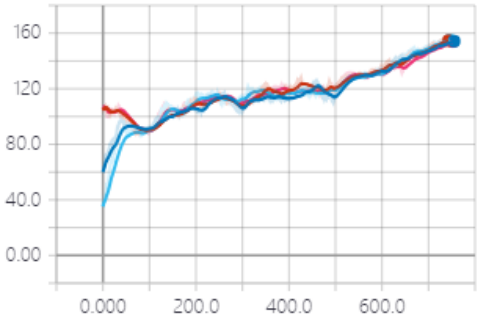

Figure 5.26: Episode rewards of workers

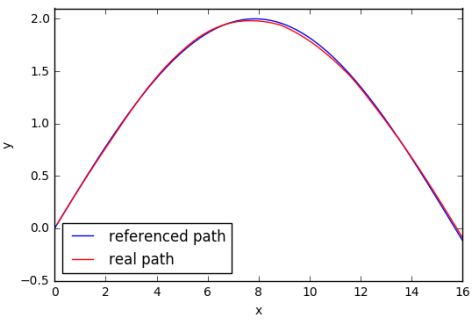

Figure 5.27: Referenced path and real path

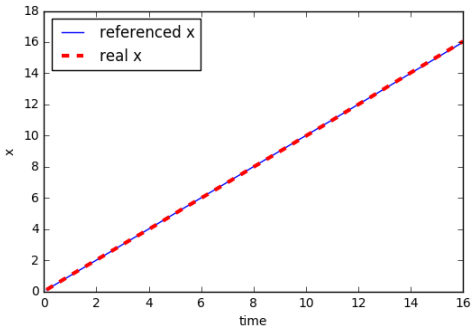

Figure 5.28: Referenced and real x position

For the second trajectory, the change of episode reward is presented in Fig. 5.32 and Fig. 5.33. The tracking results and errors are shown in Fig. 5.34 $\sim$ Fig. 5.38.

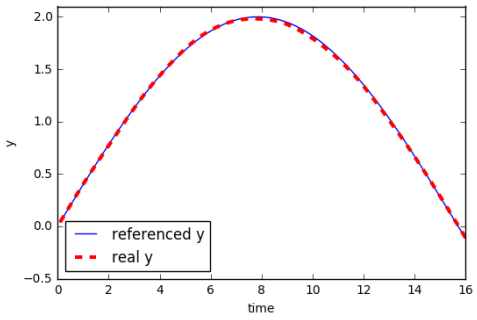

Figure 5.29: Referenced and real y position

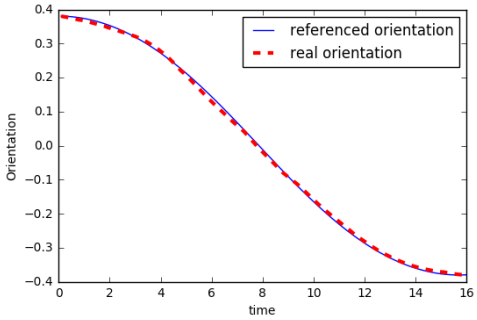

Figure 5.30: Referenced and real orientation

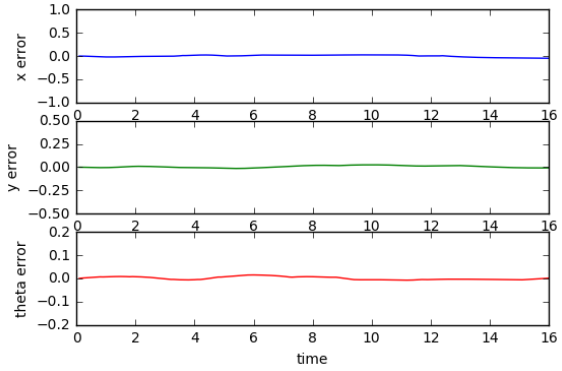

Figure 5.31: Tracking errors

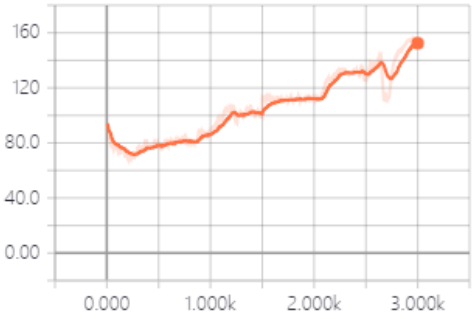

Figure 5.32: Overall episode rewards

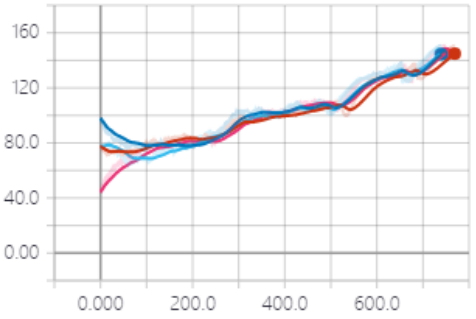

Figure 5.33: Episode rewards of workers

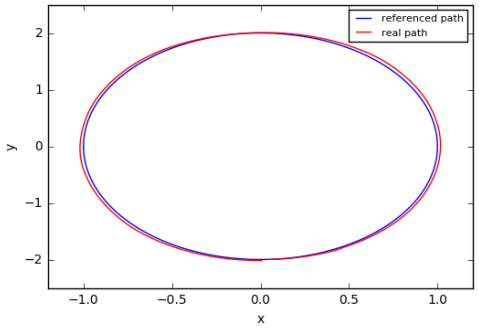

Figure 5.34: Referenced path and real path

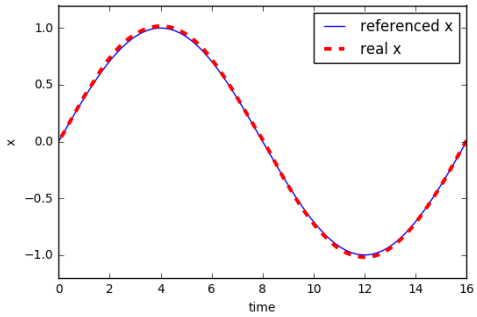

Figure 5.35: Referenced and real x position

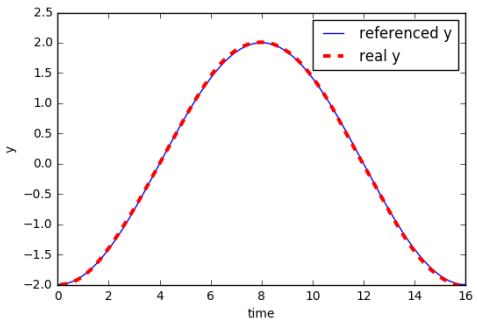

Figure 5.36: Referenced and real y position

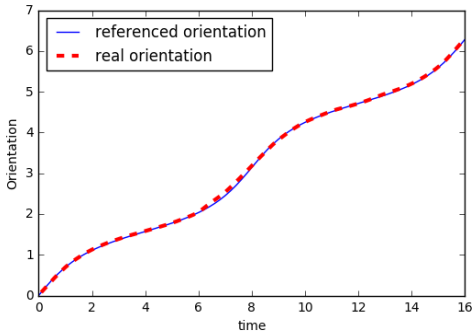

Figure 5.37: Referenced and real orientation

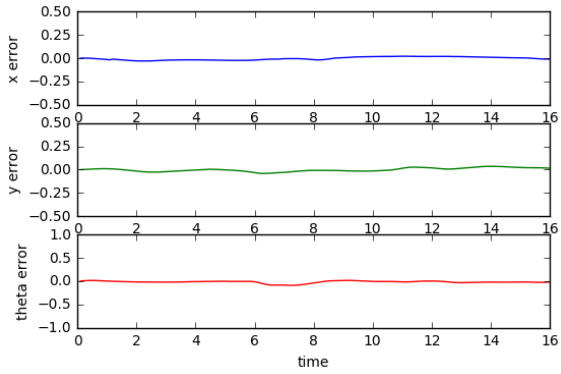

Figure 5.38: Tracking errors

We compare the tracking errors between the distributed PPO with FC networks and LSTM (Table 5.6 and Table 5.7). We can see that the tracking performance can be enhanced by introducing LSTM to the actor and critic. LSTM is powerful to process the time sequence and the transitions produced by the reinforcement learning agent are sequencial, so LSTM could explore the temporal dependencies of the sequencial states to make better decision. Furthermore, it can also provide a better representation for the actor and critic.

Table 5.6: Tracking errors of first trajectory in three dimensions

Methods	x error	y error	orientation error
Distributed PPO	2.998	2.961	1.495
Distributed PPO with LSTM	3.002	1.717	0.942

Table 5.7: Tracking errors of second trajectory in three dimensions

Methods	x error	y error	orientation error
Distributed PPO	3.906	3.683	4.451
Distributed PPO with LSTM	2.374	2.636	3.360

Chapter 6

Continuous control for the autonomous vehicle based on deep reinforcement learning

Autonomous driving has become a very popular research field in recent years since it has the potential to reduce the traffic accident and relief the traffic congestion, and it is also energy-efficient and environmentally friendly.

Autonomous driving is a complicated system comprising of several subsystems: perception subsystem, prediction subsystem and decision-making subsystem. Perception subsystem fuses the data of multiple sensors such as camera, Lidar, GPS and IMU to understand the environment. Prediction subsystem make predictions for the surroundings in a short period. Decision-making subsystem integrates all the information from the perception and prediction subsystem to plan an exact trajectory which the vehicle should follow in the coming future.

The most common decision-making architecture for the autonomous driving is shown in Fig. 6.1 [55, 70]. The highest level is the route planner which plans a route to the destination for the autonomous vehicle through the road network. The second level is the behavior planner that plans the local driving behaviors

while proceeding to the destination and obeying the traffic rules. The third level is the motion planner which produces a continuous trajectory according to the strategy from the behavior planner. The last level is the control planner which generates the execution commonds such as acceleration, brake and steering based on the trajectory from the motion planner.

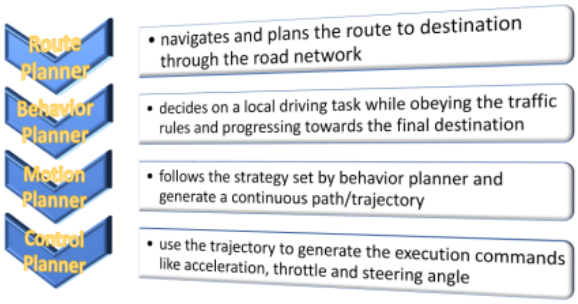

Figure 6.1: Decision-making architecture

The above decision-making architecture is complex since each level requires some specific algorithms. With the development of artifical intelligence, end-to-end methods has already been used for the autonomous driving. In this chapter, we will use an end-to-end method to make the autonomous vehicle learn the driving behaviors directly from the raw sensor data through deep reinforcement learning. We use DDPG and improved DDPG, which is called TD3, to train the autonomous vehicle and make comparison between their performance.

6.1 Learn the driving behaviors using DDPG

Firstly we use DDPG introduced in chapter 3 to train the autonomous vehicle to drive along different roads without being out of the track and colliding with other vehicles as well as try to overtake other vehicles. We train the autonomous vehicle in the open racing car simulator (TORCS) [71] which can provide a vivid 3D visualization, a complicated physics engine, and accurate vehicle dynamics which takes into consideration the traction, aerodynamics, fuel consumption and so on.

6.1.1 Architecture of the actor and critic

The autonomous vehicle can obtain the environment information from a number of sensors that the simulator provides, including its own current state, tracks and opponents. We just choose the following information from the sensors as the observation:

(1) θ: angle between the longitudinal axis of our vehicle and the track axis, which is in the range $[-\pi, +\pi]$.

(2) *tracks*: vector of readings from the 19 range finder sensors; each sensor returns the distance between our vehicle and the track edges, which is in the range $0 \sim 200m$.

(3) d: distance between our vehicle and the track axis which is normolized to $-1 \sim +1$; 0 means on the axis; -1 means on the right edge of track; $+1$ means on the left edge of track; $d > 1$ or $d < -1$ means out of the track.

(4) v_x: speed of our vehicle along its longitudinal axis.

(5) v_y: speed of our vehicle along its transverse axis.

(6) v_z: speed of our vehicle along its vertical axis.

(7) *opponents*: vector of readings from the 36 opponent sensors; each sensor covers a span of 10° and returns the distance with the closest opponent, which is in the range $0 \sim 200m$.

The total dimension of observation is 60, so it is high-dimensional input.

The outputs of the actor are the steering, acceleration and brake command which connect to the actuators: steering wheel, gas pedal and brake pedal, respectively. The steering command is in the range $[-1, +1]$. -1 denotes fully turning right and $+1$ denotes fully turning left. The acceleration command is in

the range $[0, 1]$. 0 denotes no gas and 1 denotes full gas. The brake command is in the range $[0, 1]$. 0 denotes no brake and 1 denotes full brake.

The architecture of the actor and critic is shown in Fig. 6.2. Both the actor and critic have two hidden layers with 300 neurons. For the critic, actions are firstly concatenated with the output of the first hidden layer and then input to the second hidden layer. We also use Batch Normalization to each layer of the actor to avoid the actions falling into the saturated zone of the activation function.

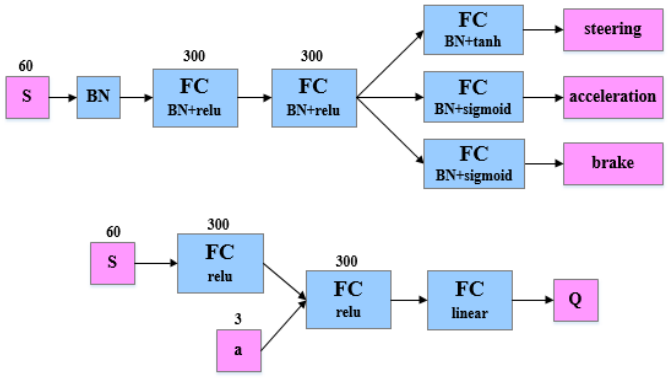

Figure 6.2: Architecture of actor and critic

6.1.2 Reward function and training strategy

We hope to design a reward function which can encourage the autonomous vehicle to drive along the track axis with as fast speed as possible and overtake other vehicles in the track. Hence, the reward function consists of two terms as expressed in Eq. 6.1.The first term aims to train the autonomous vehicle to drive along the track smoothly and the second term is to encourage our vehicle to overtake other vehicles.

$$r_{total} = r_1 + r_2, \tag{6.1}$$

$$r_1 = v_x \cos\theta - v_x|\sin\theta| - v_x|d|,$$

$$r_2 = 2(n - rank),$$

where n is the total number of vehicles on the track; $rank$ is the ranking of our autonomous vehicle among all the vehicles, which can be obtained from TORCS. r_1 try to maximize the velocity of our vehicle along the track axis and minimize the velocity along the perpendicular direction of the track axis. r_1 also encourages our vehicle to drive in the middle of the track by minimizing the absolute value of d. $|d|$ is multiplied by v_x since large velocity can make the vehicle deviate from the track axis easily. r_2 can encourage our vehicle to overtake other vihicles since higher ranking could get larger reward. There are totally 10 vehicles on the road.

Moreover, if our vehicle collide with other vehicles or the wall, it will get a reward of -10. The collision can be detected by the vehicle state *demage*. If the current value of *demage* is larger than the previous value, it means the vehicle has collided with something. If the vehicle is out of the track ($\min(tracks) < 0$), it will also get a reward of -10. We use a counter to record the times of being out of the track for our vehicle. When the counter value is greater than 100, the episode will terminate early. If the vehicle drives backwards ($\cos\theta < 0$), the agent will get a reward of -10 and the current episode will end in advance. In addition, if the velocity of our vehicle along the track axis is too small, the episode will also end early.

Aiming to the exploration and sample efficiency issue of deep reinforcement learning, we adopt a two-stage training strategy similar to section 5.6, which consists of pre-training by imitation learning and fine-training by DDPG. Imitation learning can also be regared as the supervised learning, which utilizes the expert data to update the policy. For each visited state of our vehicle, we can obtain its target action according to the driving example provided by TORCS. After collecting 50 training data, the actor will be updated for multiple times by using these data. Then new training data will be collected with the interaction process to update the actor for the next iteration. The critic is also updated by the same method as DDPG during the pre-training precess. After imitation learning stage, the agent is further trained through DDPG to enhance the

generalization and robustness of the policy. By introducing imitation learning, the agent can directly learn from the desirable actions, so it doesn't need to explore to the expected states by itself, which helps to alleviate the exploration problem of deep reinforcement learning. Moreover, the agent is not trained from scratch by deep reinforcement learning, which could significantly reduce the sample complexity. Fig. 6.3 and Fig. 6.4 show the profile and GUI of the training road, respectively.

Figure 6.3: Training road profile

Figure 6.4: Training road GUI

In the reinforcement learning training stage, we add some noises to the actions in order to further increase the explortion. We use the Ornstein-Uhlenbeck process (Eq.3.8) which has the mean-reverting property to produce noises. The hyper-parameters of OU process we use for the three actions are shown in Tabel 6.1.

Table 6.1: Hyperparameters of OU process

Action	η	σ_1	σ_2
steering	0.8	0	0.2
acceleration	1	0.5	0.1
brake	1	-0.1	0.05

The average rewards of every 20 steps for the two stages are shown in Fig. 6.5. During the imitation learning stage, the average reward goes up quickly in the first 2000 steps. In the following 2000 steps, the average reward doesn't have obvious increase. Then DDPG is employed to train the actor and critic which are initialized with the pretrained weights from the imitation learning stage. The average reward of reinforcement learning stage goes up relatively slowly,

but it can further improve the driving policy via the reward mechanism by interacting with the environment.

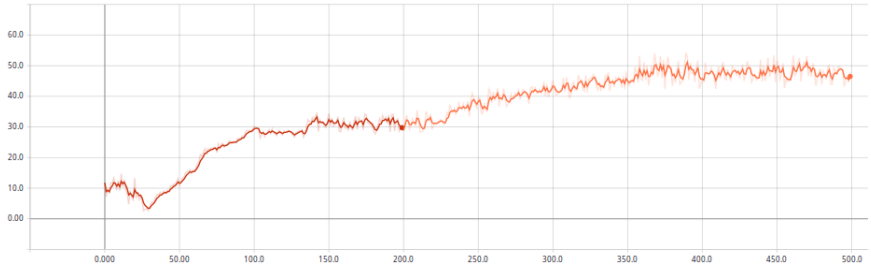

Figure 6.5: Average reward of every 20 steps

6.1.3 Performance evaluation

After the two-stage training, we save the final model and use it to test the autonomous vehicle in the different roads. In addition to the training road, we also test the vehicle in three unseen roads which are shown in Fig. 6.6. These testing roads are more tortuous or narrower than the training road in order to evaluate the generalization ability of the agent better. Their corresponding GUIs are displayed in Fig. 6.7.

Figure 6.6: Three testing roads

Figure 6.7: GUIs of three testing roads

We test the autonomous vehicle on each road for 20 laps and record the times of sucess and failure. The failure mode includes the collision with other vehicles and being out of the track. We also evaluate its overtaking performance by calculating the average overtaking number of 20 laps for each road. The testing results and the road width are presented in Table 6.2. We can see that our vehicle can still drive smoothly on most of the laps of the more complex unseen testing roads, which demonstrates enough generalization ability of the agent. Because sometimes the velocities of other vehicles are very fast, it may be difficult for our vehicle to overtake other vehicles, which results in the unsatisfactory overtaking performance.

Table 6.2: Testing results of DDPG for autonomous vehicle

Road	width(m)	success	collision	out of track	average overtaking
Training road	12	18	2	0	2.65
Testing road 1	10	16	2	2	2.55
Testing road 2	12	17	2	1	2.6
Testing road 3	12	18	1	1	2.85

6.2 Learn the driving behaviors using Twin Delayed DDPG

Even if DDPG could achieve good performance on some tasks, sometimes it may fail because of the overestimation of Q-value, which can lead to the sub-optimal policies. In this section, we firstly introduce some improvements on DDPG. Then we use this improved algorithm which is called TD3 to train the autonomous vehicle and make comparison with the performance of DDPG.

6.2.1 Twin Delayed DDPG

Twin Delayed DDPG (TD3) [22] is an improved version of DDPG proposed by the researchers from McGill University. It maily aims to deal with the over-

estimation bias of the value function. Overestimation is a obvious problem in the discrete action space which has the explicit maxmizing operation. While it can be proved that the value estimate of DDPG is also an overestimation under some assumptions. Therefore, TD3 adopts a clipped double Q-learning to reduce the overestimation problem.

TD3 has two critics Q_1 and Q_2. Each critic has a target critic. It uses the smaller value of the two target critics to calculate the target value y_t [22].

$$y_t = r_t + \gamma(1-d)\min_{i=1,2} Q_i'(s_{t+1}, \mu'(s_{t+1})), \tag{6.2}$$

where Q_1' and Q_2' are the target critics of Q_1 and Q_2 respectively; d indicates if it is the last state of the episode; when s_{t+1} is the last state, $d = 1$; when s_{t+1} is not the last state, $d = 0$.

Both Q_1 and Q_2 use y_t as the target value to update their parameters. The actor can be updated only by maximizing Q_1.

The next improvement of TD3 is the target policy smoothing. A clipped noise is added to every dimension of the target action. The target action should also be clipped to the allowable action range. Thus, the target action is expressed as follows:

$$a'(s_{t+1}) = clip(\mu'(s_{t+1}) + \epsilon, a_l, a_h), \tag{6.3}$$

where $\epsilon \sim clip(N(0, \sigma), -c, c)$ is the noise; a_l and a_h are the lower bound and upper bound of action.

Target policy smoothing can deal with the failure situation of DDPG: if the critic has inaccurate sharp peaks on some actions, the policy may utilize these peaks and produce impropor actions. This problem could be mitigated by smoothing out the Q-function over similar actions, which is the idea of the target policy smoothing.

Moreover, TD3 adopts the delayed policy update. Because the policy is updated according to the Q-value, it is necessary to firstly minimize the estimated error of the critic before updating the policy. Therefore, the policy shoule only be updated after a certain number of updates to the critic. This less frequent update for the policy can utilize the critic value with lower variance, which may lead to the policy with higher quality.

6.2.2 Performance evaluation

We add the normal distribution noises which have mean of 0 and standard deviation 0.2 to the target actions $\mu'(s_{t+1})$ and the noises are clipped to $-0.5 \sim 0.5$. The target actions with noises should also be clipped to the range of real actions. In addition, the actor is only updated once after two updates of critic.

During the imitation learning stage, the actor is updated by the expert experiences and the two critics Q_1 and Q_2 are updated by the transitions collected from the environment. Then TD3 is used to further train the policy in the second stage. Fig. 6.8 presents the average reward comparison of the second training stage between TD3 and DDPG. We can see that TD3 can get relatively higher reward than DDPG.

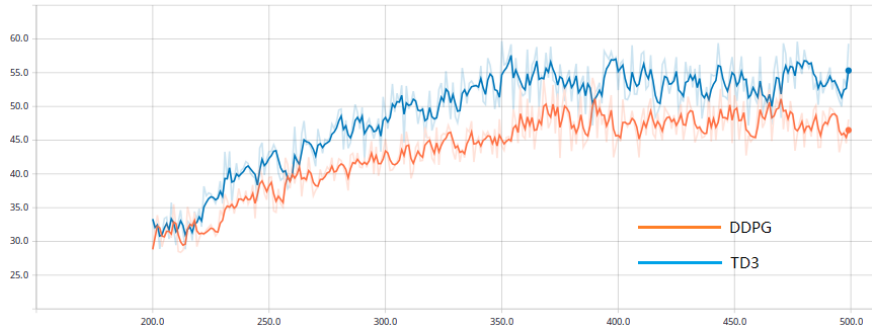

Figure 6.8: Comparison of rewards

Table 6.3 records the testing results of TD3 for the autonomous vehicle in the same roads as DDPG. Fig. 6.9 presents the comparison of success times between TD3 and DDPG. We find that TD3 can generally achieve a little higher success

times than DDPG. We also record the speed of our vehicle during the testing period and the speed comparison between DDPG and TD3 on one testing road is shown in Fig. 6.10, from which we can see that the speed of vehicle trained by TD3 is obviously higher than DDPG. Therefore, in our experiment, TD3 can achieve better performance than DDPG for the autonomous driving.

Table 6.3: Testing results of TD3 for autonomous vehicle

Road	width(m)	success	collision	out of track	average overtaking
Training road	12	18	2	0	2.6
Testing road 1	10	17	2	1	2.65
Testing road 2	12	18	1	1	2.9
Testing road 3	12	17	2	1	2.7

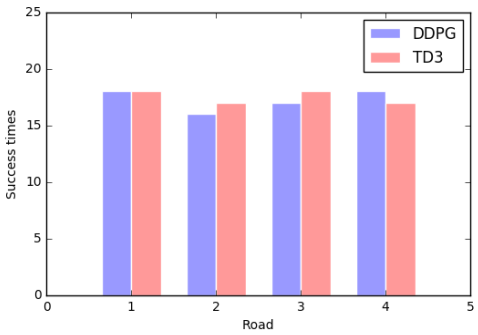

Figure 6.9: Success times

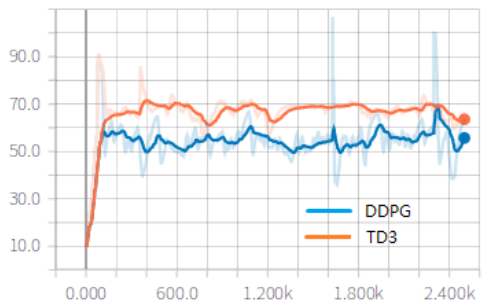

Figure 6.10: Speed comparison

By introducing the improvements of TD3, the autonomous vehicle can be trained to master better driving behaviors. The trained agent has strong generalization ability since it also has good driving performance in the more complex unseen environments.

Chapter 7

Conclusions and future work

This thesis employs model-free deep reinforcement learning algorithms to enable the robots to complete some tasks, including the multi-goal task, tracking task and autonomous driving task.

The multi-goal task is to train a redundant manipulator to reach any given position from a random initial position. The goal position of each episode is different during the training process. Firstly, we use DDPG combined with HER and different strategies of choosing the additional goals to train the redundant manipulator. Even if we only use a simple binary reward, it can get approximately 80% success rate with all the strategies. Then we use DDPG with a shaped reward to trian the redundant manipulator. The manipulator can't complete the task by DDPG with priority replay memory without additional goals. By referring to the idea of HER, we propose a *random* and $future$ strategy combined with the shaped reward to generate additional transitions. The simulation results demonstrate the *random* strategy can achieve the best performance with more than 80% success rate. Therefore, the agent can learn more effectively by artificially generating some additional transitions with more diverse goals. Next, we use A3C and A2C to train the redundant manipulator to complete the multi-goal task. By considering the reaching path of the end-

effector, we propose a new reward function which aims to constrain the moving direction of the end-effector to be closer to the ideal direction. The performance of different algorithms and reward functions are compared, which shows that A2C could get a better performance than A3C due to the avoidance of gradient delay and the new reward function can not only optimize the reaching path but also increase the success rate. Distributed PPO with GAE is also employed to train the redundant manipulator with the two different reward functions. The distributed framework which has multiple workers to produce the transitions for the global network can increase the sample collection speed and the random sampling can reduce the correlations among the transitions when updating the global network. GAE is introduced when calculating the advantage of the objective function of the actor to make a good balance between the variance and bias. Table 7.1 summarizes the best testing results of all the algorithms, which presents that distributed PPO could achieve the highest success rate.

Table 7.1: Success times of redundant manipulator over 100 episodes by different algorithms

Algorithms	Success times
DDPG	87
A3C	84
A2C	88
Distributed PPO	91

The tracking task is to train the robots to track the given trajectories. Firstly, we use DDPG with priority replay memory to train a SCARA robot and a mobile robot to track the trajectories. Two training strategies, random referenced state initialization and early termination, are introduced to make the robots learn effectively from the referenced trajectories. After training for enough episodes, the SCARA robot and mobile robot can learn to move along the referenced trajectories with small errors in three dimensions. Then we propose a distributed framework of DDPG, which has synchronous workers to generate samples and compute gradients for the global network and the collecting workers to produce transitions with different policies and exploration noises for the shared replay memory. This distributed framework can promote the data throughput and increase the diversity of the transitions in the replay memory. We use this dis-

tributed DDPG to train the SCARA robot and mobile robot to track the same trajectories. The simulation results show that the agent trained by distributed DDPG could learn faster and achieve smaller tracking errors than the single-worker DDPG. We also use distributed PPO with GAE to train the mobile robot to track the trajectories with the improved training strategies. In order to enhance the training and sample efficiency, we propose a two-stage training strategy which consists of the supervised pre-training and fine-training by distributed PPO. The supervised pre-training enables the agent to learn some useful experiences quickly and therefore reduce the training episodes by distributed PPO. The final results of the two-stage training is also a little better than distributed PPO. After that, we introduce LSTM to represent the actor and critic. Some buffers are adopted to store the cell state and hidden state of LSTM used for the initialization of each episode to alleviate the problem of inaccurate initial LSTM states. The simulation results demonstrate that the tracking performance can be improved by introducing LSTM to the actor and critic. Table 7.2 and 7.3 summarize the tracking errors of the mobile robot by different methods, which shows that the tracking performance of distributed DDPG is a little better than distributed PPO while distributed PPO with LSTM could achieve the best tracking performance.

Table 7.2: Tracking errors of mobile robot for the first trajectory by different algorithms

Algorithms	x error	y error	orientation error
DDPG	6.578	3.495	1.471
Distributed DDPG	2.592	2.915	1.242
Distributed PPO	2.998	2.961	1.495
Distributed PPO with LSTM	3.002	1.717	0.942

Table 7.3: Tracking errors of mobile robot for the second trajectory by different algorithms

Algorithms	x error	y error	orientation error
DDPG	4.267	2.533	5.265
Distributed DDPG	2.837	2.919	3.802
Distributed PPO	3.906	3.683	4.451
Distributed PPO with LSTM	2.374	2.636	3.360

Finally, we utilize deep reinforcement learning to train the autonomous vehicle to learn the driving behaviors end-to-end. The agnet only takes the raw

sensor data, including the pose and velocity of the vehicle, distances from the track edges, and distances from the other vehicles, as the input and directly outputs the steering, acceleration and brake commond. A reward function is designed to encourage the autonomous vehicle to drive along the roads smoothly and overtake other vehicles. Moreover, we propose a two-stage training strategy including the pre-taining by imitation learning and fine-training by deep reinforcement learning. After the imitation pre-training stage, the autonomous vehicle is further trained by using DDPG and TD3 in the second training stage, respectively. We find that the vehicle trained by TD3 can obtain larger speed and a little higher success rate than DDPG in the testing.

The agents of model-free deep reinforcement learning usually need massive amounts of interactive data to learn a specific task and the agent trained in one environment may not generalize to other unseen environments. Meta-learning, which means learning to learn, is a significant research direction to solve the fast learning problem of agent. The goal of meta-learning is to train an agent on a variety of tasks, such that it can solve the new tasks by only using a small number of training samples. In the future work, we can adopt the idea of meta-learning to train the reinforcement learning agents. The trained meta-agent should have good generalization ability to adapt to multiple different tasks quickly. Then the agent doesn't need to collect a large amount of interactive transitions from the environment for each specific task, which can greatly improve the sample efficiency.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," in *Nature*, vol. 518, pp. 529–533, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

[4] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," *arXiv preprint arXiv:1511.06581v3*, 2015.

[5] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *International Conference on Learning Representations*, 2016.

[7] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," in *International Conference on Machine Learning*, 2017.

[8] J. Menick and M. Hessel, "Noisy Networks for Exploration," in *International Conference on Learning Representations*, 2018.

[9] D. Silver, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *International Conference on Machine Learning*, pp. 387–395, 2014.

[10] 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," in *International Conference on Learning Representations*, 2016.

[11] V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *International Conference on Machine Learning*, 2016.

[12] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust Region Policy Optimization," in *International Conference on Machine Learning*, 2015.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[14] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," *arXiv preprint arXiv:1707.02286*, 2017.

[15] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," in *International Conference on Learning Representations*, 2016.

[16] A. Harutyunyan, T. Stepleton, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," in *30th Conference on Neural Information Processing Systems*, pp. 1054–1062, 2016.

[17] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample Efficient Actor-Critic with Experience Replay," in *International Conference on Learning Representations*, 2016.

[18] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," in *31st Conference on Neural Information Processing Systems*, pp. 5279–5288, 2017.

[19] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, B. Schölkopf, and S. Levine, "Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning," in *31st Conference on Neural Information Processing Systems*, pp. 3846–3855, 2017.

[20] X. Chu, "Policy Optimization With Penalized Point Probability Distance: An Alternative To Proximal Policy Optimization," *arXiv preprint arXiv:1807.00442*, 2018.

[21] A. G. Wil, l Dabney, M. G. Azar, B. Piot, M. Bellemare, and Remi, "The Reactor: A fast and sample-efficient Actor-Critic agent for Reinforcement Learning," in *International Conference on Learning Representations*, 2018.

[22] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[23] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming Exploration in Reinforcement Learning with Demonstrations," in *IEEE International Conference on Robotics and Automation*, pp. 6292–6299, 2018.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv preprint arXiv:1801.01290*, 2018.

[25] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight Experience Replay," in *31st Conference on Neural Information Processing Systems*, pp. 5048–5058, 2017.

[26] P. Gallien, B. Nicolas, S. Robineau, M. P. Lebot, and R. Brissot, "Learning by Playing Solving Sparse Reward Tasks from Scratch Martin," *arXiv preprint arXiv:1802.10567*, 2018.

[27] J. Achiam and S. Sastry, "Surprise-Based Intrinsic Motivation for Deep Reinforcement Learning," *arXiv preprint arXiv:1703.01732*, 2017.

[28] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven Exploration by Self-supervised Prediction," in *International Conference on Machine Learning*, 2017.

[29] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros, "Large-Scale Study of Curiosity-Driven Learning," *arXiv preprint arXiv:1808.04355*, 2018.

[30] N. Savinov, A. Raichuk, R. Marinier, D. Vincent, M. Pollefeys, T. Lillicrap, and S. Gelly, "Episodic Curiosity through Reachability," *arXiv preprint arXiv:1810.02274*, 2018.

[31] A. Nair, P. Srinivasan, and S. Blackwell, "Massively parallel methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2015.

[32] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributed Distributional Deterministic Policy Gradients," *arXiv preprint arXiv:1804.08617*, 2018.

[33] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed Prioritized Experience Replay," in *International Conference on Learning Representations*, 2018.

[34] H. P. van Hasselt, A. Guez, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," in *30th Conference on Neural Information Processing Systems*, pp. 4287–4295, 2016.

[35] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," *arXiv preprint arXiv:1802.01561*, 2018.

[36] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt, "Multi-task Deep Reinforcement Learning with PopArt," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[37] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[38] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[39] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation," *arXiv preprint arXiv:1704.03073*, 2017.

[40] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *IEEE International Conference on Robotics and Automation*, pp. 3389–3396, 2017.

[41] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 2017.

[42] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations," *arXiv preprint arXiv:1709.10087*, 2017.

[43] A. Nair, V. Pong, M. Dalal, S. Bahl, S. Lin, and S. Levine, "Visual Reinforcement Learning with Imagined Goals," *arXiv preprint arXiv:1807.04742*, 2018.

[44] OpenAI:, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning Dexterous In-Hand Manipulation," *arXiv preprint arXiv:1808.00177*, 2018.

[45] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 31–36, 2017.

[46] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *IEEE International Conference on Robotics and Automation*, pp. 3357–3364, 2017.

[47] T. Fan, P. Long, W. Liu, and J. Pan, "Fully Distributed Multi-Robot Collision Avoidance via Deep Reinforcement Learning for Safe and Efficient Navigation in Complex Scenarios," *arXiv preprint arXiv:1808.03841*, 2018.

[48] X. Chen, A. Ghadirzadeh, J. Folkesson, and P. Jensfelt, "Deep Reinforcement Learning to Acquire Navigation Skills for Wheel-Legged Robots in Complex Environments," *arXiv preprint arXiv:1804.10500*, 2018.

[49] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, "Learning to Navigate in Complex Environments," in *International Conference on Learning Representations*, 2017.

[50] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep Reinforcement Learning framework for Autonomous Driving," *IS&T Electronic Imaging, Autonomous Vehicles and Machines*, pp. 70–76, 2017.

[51] X. Pan, Y. You, Z. Wang, and C. Lu, "Virtual to Real Reinforcement Learning for Autonomous Driving," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2017.

[52] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, "End-to-End Race Driving with Deep Reinforcement Learning," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2070–2075, 2018.

[53] S. Sharifzadeh, I. Chiotellis, R. Triebel, and D. Cremers, "Learning to Drive using Inverse Reinforcement Learning and Deep Q-Networks," in *Conference on Neural Information Processing Systems*, 2016.

[54] M. Kaushik, V. Prasad, K. M. Krishna, and B. Ravindran, "Overtaking Maneuvers in Simulated Highway Driving using Deep Reinforcement Learning," in *IEEE Intelligent Vehicles Symposium, Proceedings*, 2018.

[55] Y. Chen, *Learning-based Lane Following and Changing Behaviors for Autonomous Vehicle*. PhD thesis, Carnegie Mellon University, 2018.

[56] R. Yu, Z. Shi, C. Huang, T. Li, and Q. Ma, "Deep reinforcement learning based optimal trajectory tracking control of autonomous underwater vehicle," in *Proceedings of the 36th Chinese Control Conference*, pp. 4958–4965, 2017.

[57] H. Wu, S. Song, K. You, and C. Wu, "Depth Control of Model-Free AUVs via Reinforcement Learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2018.

[58] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne, "DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills," *arXiv preprint arXiv:1804.02717*, 2018.

[59] X. B. Peng, A. Kanazawa, J. Malik, P. Abbeel, and S. Levine, "SFV: Reinforcement Learning of Physical Skills from Videos," *arXiv preprint arXiv:1810.03599*, 2018.

[60] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. London, England: The MIT Press, 2018.

[61] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, pp. 1057–1063, 1999.

[62] O. Konur, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION," *International Conference on Learning Representations*, 2015.

[63] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal Value Function Approximators," in *International Conference on Machine Learning*, pp. 1312–1320, 2015.

[64] L. Xin, Q. Wang, J. She, and Y. Li, "Robust adaptive tracking control of wheeled mobile robot," *Robotics and Autonomous Systems*, vol. 78, pp. 36–48, 2016.

[65] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous Stochastic Gradient Descent with Delay Compensation," in *Proceedings of the 34 th International Conference on Machine Learning*, 2017.

[66] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[67] S. Hochreiter and J. . Urgen Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[68] M. Hausknecht and P. Stone, "Deep Recurrent Q-Learning for Partially Observable MDPs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 29–37, 2015.

[69] A. Graves, "Generating Sequences With Recurrent Neural Networks," *arXiv preprint arXiv:1308.0850*, 2014.

[70] B. Paden, M. Cap, S. Z. Yong, D. Yershov, and E. Frazzoli, "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[71] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Simulated Car Racing Championship: Competition Software Manual," *arXiv preprint arXiv:1304.1672*, 2013.