SHENGNAN CHEN

COMPARING DEEP REINFORCEMENT LEARNING METHODS FOR ENGINEERING APPLICATIONS

Master of Science thesis

Examiner: Prof. Sanaz Mostaghim
Examiner and topic approved by the
Faculty Council of the Faculty of
Computer Science
on 25th August 2018

ABSTRACT

SHENGNAN CHEN: Comparing Deep Reinforcement Learning Methods for Engineering Applications
Otto-von-Guericke University
Master of Science thesis, 120 pages
25th August 2018
Master's Degree Program in Computer Science
Major: Digital Engineering
Examiner: Prof. Sanaz Mostaghim
Tutor: Msc. Gabriel Campero Durand
Keywords: Reinforcement Learning, Deep Learning, Deep Reinforcement Learning, Engineering applications, Artificial Intelligence, OpenAI gym, DQN, DDPG, A3C, PPO, benchmark

In recent years the field of Reinforcement Learning has come across a series of breakthroughs. In combining with Deep Learning practitioners have proposed to generate various Deep Reinforcement Learning solutions. Nowadays, researchers apply such methods to different kinds of applications. However, the DRL methods proposed in this recent period are still under development, and researchers in the field face the practical challenge of determining which methods are applicable to their use cases, and what specific design and runtime characteristics of the methods demand consideration for optimal applications. In this Thesis, we study the importance and requirements of DRL in engineering applications. For this we implement three Deep Reinforcement Learning methods(Deep Q Network, Deep Deterministic Policy Gradient, Distributed Proximal Policy Optimization), training and testing on three different environments(CartPole, PlaneBall, CirTurtleBot). The reference implementation are organized in CSN-RL in Github. We capture features from the Deep RL models we evaluate and generalize from our study to propose how comparison across models should be done, providing researchers with the information they need about methods. Our results can give suggestions of ranking the three methods according to specific applications for researchers.

ACKNOWLEDGEMENT

By submitting this thesis, my long term association with Otto von Guericke University will come to an end.

First and foremost, I am grateful to my advisor M.Sc. Gabriel Campero Durand for his guidance, patience and constant encouragement without which this may not have been possible.

I would like to thank Prof. Dr. Sanaz Mostaghim for giving me the opportunity to write my Masters thesis at her chair.

I would like to thank Dr.-Ing. Akos Csiszar in Stuttgart University for providing me this topic.

It has been a privilege for me to work in field of reinforcement learning.

I would like to thank my family and friends, who supported me in completing my studies and in writing my thesis.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

A3C	Asynchronous Advanced Actor Critic
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q Network
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network

α	learning rate
γ	discount rate
ϵ	exploration rate
R/r	rewards
$\Pi(S)/\pi(S)$	policy output
$V(S)$	value function
$A(S)$	advantage function

1. INTRODUCTION

In this chapter, we will present the brief definition, the problem statement, the motivation behind the thesis, describe goals and outline its organization.

1.1 Brief Definition

Since the concept of "industry 4.0" has been proposed, "How to build a 'smart factory'?" became the first question to consider. Without a doubt, more and more researchers focus on the field of artificial intelligence(AI). Developing machines and software agents with the ability of self learning without previous or preprocessed data to determine the ideal behavior has become one of the primary goals of the field of AI. There is a kind of framework that can generalize those sort of problems onto it, it called Reinforcement Learning(RL). Reinforcement learning (RL) methods provide a means for solving optimal control problems when accurate models are unavailable. Reinforcement learning (RL) enables an agent to autonomously discover an optimal behavior through trial-and-error interactions with its environment[36]. This can save a lot of programming time, and can avoid hard-engineering works that are needed to control every single action of a *controller*, we can also call it *agent*. Thus we can think of reinforcement learning as a trial-and-error learning done by the machine: the agent learns how to act from what receives (reward) as a consequence of its actions, and it can select actions based on its past experience (exploiting actions) and also by choosing new ones (exploring actions). The best trade-off between exploitation and exploration lets the agent understand how to achieve correctly the behavior required by the task of a specific problem[74]. A good exploration strategy is very important for a good policy selection.

Although RL had some successes in the past, previous approaches lacked scalablity and were inherently limited to fairly low-dimensional problems. These limitations exist because RL algorithms share the same complexity issues as other algorithms[2]. RL generally can be sorted as an optimal control problem. When Bellman (1957) explored optimal control in discrete high-dimensional spaces, he faced an exponential

explosion of states and actions for which he coined the term Curse of Dimensionality. The reinforcement learning community fights a long time dealing with dimensionality using computational abstractions from adaptive discretizations to function approximation approaches. Since the arise of Deep Learning. A new function approximation named deep neural network came out. The fusion of deep learning represents the new born of reinforcement learning, the Deep Reinforcement Learning(DRL) field appeared. In recent years, many successful DRL algorithms have been proposed. Deep Q Network(DQN) is the first DRL algorithm able to overbear in several challenging tasks with the same architecture proposed by [51], which have the trait to learn a successful behavior by observing only the raw images of the game it plays and receiving a reward signal builds up by the scores gained at each time-step. Simply say, DQN uses Q-learning as the policy updating strategy, Convolutional Neural Network as the function approximation, experience reply mechanism as the way to break data correlation. According to different RL policy estimating methods, combining with deep neural networks(DNNs) and inspiring from DQN, plentiful DRL methods are proposed. Deep Deterministic Policy Gradient(DDPG) is one of the DRL method, which combining deterministic actor-critic method with DNNs, using break correlation mechanism from DQN. Asynchronous Advanced Actor Critic(A3C) and Proximal Policy Optimization(PPO) are also two popular DRL methods nowadays, we will discuss them in detail in chapter 2.

Reinforcement Learning has a wide range of applications, [44] listed various applications of RL: Games, Robotics, Natural language processing, Computer vision, Neural architecture design, Business management, Finance, Healthcare, Industry 4.0, Smart Grid, Intelligence transportation system, Computer systems. From designing state-of-the-art machine translation models to constructing new optimization functions, DRL has already been used to approach all manner of machine learning tasks. As deep learning has been adopted across many branches of machine learning, it seems likely that in the future, DRL will be an important component in constructing general AI systems[2]. Although DRL is experimented and successful firstly in Game fields, people work without stopping at applying DRL in other application fields. Up-to-date, the applications of DRL are getting more and more widely, especially in engineering field. Table 3.1 shows current RL engineering applications. As engineering is such a large field including various categories, like mechanical, chemical, electrical, etc. Each category could have large number of applications. We sort RL in engineering applications into three function aspects: Optimize, Control, Monitor and Maintain. RL in engineering applications is still facing challenges, such like the physical world uncertainty, simulated environment issues, suitable RL method selection. We will discuss more in chapter 3.

1.2 Problem Statement

DRL algorithms have already been applied to a wide range of problems. Up-to-date, the Deep RL methods are plump during continuous researching. Various Deep RL methods have been proposed and open source implementations are becoming publicly available every day. For traditional reinforcement learning, value function based, policy search and actor-critic are general three approach domains. Each domain has various methods, when combining with deep neural network, there will be a bunch of Deep RL methods. Researchers in the field face the practical challenge of determining which methods are applicable to their use cases, and what specific design and runtime characteristics of the methods demand consideration. Some existed researches do DRL methods selection according to the discrete or continuous action space and low or high dimensional state space. No doubt, this can be a piece of benchmark design. However, only use these factors for benchmarking is unilaterally. Also, for real-world engineering applications, the physical environments are sometimes complex and usually need to simulate the environments. The way to design the simulated environments could be sundry. For example, "Cart-Pole" system in OpenAI Gym is designed as two discrete actions and four dimensional state space. If we apply this system in our suitable engineering applications like balancing the wheel chair like vehicles, we can design the action space as one dimensional continuous action(pushing force between -2 to 2). If we use the benchmark above to decide the DRL method, it makes no sense.

Standard benchmarks for Deep RL methods need to be generated. Along with this recent progress, the Arcade Learning Environment(ALE) [7] has become a popular benchmark for evaluating algorithms designed for tasks with high-dimensional state inputs and discrete actions. Other benchmarks[17, 21, 76] are also proposed regarding different aspects and requirements.

The lack of a standardized and challenging testbed for Deep RL methods makes it difficult to quantify scientific progress. Systematic evaluation and comparison will not only further our understanding of the strengths of existing algorithms, but also reveal their limitations and suggest directions for future research.

1.3 motivation

As we said before, current state-of-art benchmarks were proposed regarding specific fields and aspects. [7] is currently popular benchmark for different RL methods in Atari game environments. However, these algorithms do not always generalize

straightforwardly to tasks with continuous actions. [18] contains tasks with relatively low-dimensional actions. [17] benchmark contains a wider range of tasks with high-dimensional continuous state and action spaces. However, the up-to date DRL methods are not evaluated and compared in these papers. We will discuss more benchmarks in chapter 6.

As the new successful Deep RL methods being proposed, like Asynchronous Advanced Actor Critic(A3C) and Proximal Policy Optimization(PPO), they are not contained in the previous benchmarks. The performance of these new algorithms need to be evaluated. As aforementioned described, for engineering applications, the prototypical of the simulated environments could be plenty designed. It is necessary to benchmark according to specific engineering application prototype rather than the environment features themselves.

1.4 implementation and goal

- **Criteria and experimental comparison**
  Determining the important comparison criteria regarding to state-of-art researches. Do experiments comparisons with three different Deep RL methods(DQN, DDPG, DPPO) which currently prevalent using three representative environments.

- **Outline of limitations and generalization**
  Using our tested environments to capture features from the Deep RL models we evaluate. Generalizing from our study to propose how this comparison should be done in a benchmarking tool, providing researchers with the information they need about methods.

1.5 research methodology

In this paper, we compare three DRL methods and evaluate them according to two research questions. Following is the methodology we use.

- **DRL methods understanding**
  Learn and understand the ideas behind the DQN, DDPG, PPO methods we compare in this paper.

- **Engineering applications analysis**
  Analyze and summarize the characteristics of engineering applications.

- **Training preparation**
  Preparing and programming the DRL agents and the test environments.
- **Obtaining history data**
  Getting the information data like steps, rewards, duration, during training phase.
- **Evaluation**
  Comparing the performance of three methods and evaluating them regarding to the evaluation factors.

This paper is structured as follows.

- Chapter 1: Introduction about general statement.
- Chapter 2: State-of-Art of Reinforcement Learning and Deep Reinforcement Learning.
- Chapters 3: Describing Deep Reinforcement Learning for Engineering Applications.
- Chapter 4: Proposing Research Questions, Environment and Implement study.
- Chapter 5: Experiment results and evaluation.
- Chapter 6: Discussing related work about exist comparisons and benchmarks.
- Chapter 7: Discussing conclusion and further work.

2. BACKGROUND: REINFORCEMENT LEARNING AND DEEP REINFORCEMENT LEARNING

In this chapter, we present an overview of the theoretical background about Reinforcement learning and Deep reinforcement learning. We will discuss the general theory of reinforcement learning with focus on Q-learning and Policy Gradient. Then discussing two deep neural networks: CNN, RNN and the hyper-parameters of Deep learning. After that, we lead out Deep reinforcement learning theory. More than that, four state-of-the-art DRL methods will be presented.

2.1 Reinforcement Learning

Reinforcement learning is proposed to solve problems by creating an agent that must learn behavior through trial-and-error interactions with a dynamic and unknown environment. In [31], it sorted solving RL problems into two main strategies: to search in the space of behaviors in order to find one that performs well in the environment; to use statistical techniques and dynamic programming methods to estimate the utility of taking actions in states of the world. Since the arise and requirement of artificial intelligence, people take advantages from the second strategy, more and more research is focused on it. In this paper, we also talk about the second option.

Although reinforcement learning is an area of machine learning fields, it has difference from normal machine learning. It does not depend on preprocessed data, it derives knowledge from its own experience. It focuses on performance, which involves finding a balance between exploration and exploitation. Also, it bases on the real-world environment scenarios. Reinforcement learning methods almost follow the RL model. There is the basic RL model in figure 2.1. Agent takes a certain action according to the internal action chosen strategy, basing on the previous state, then interacts with the environment to observe current state and relevant rewards. This process is called a transition.

Reinforcement learning problems are modeled as Markov Decision Processes(MDPs).

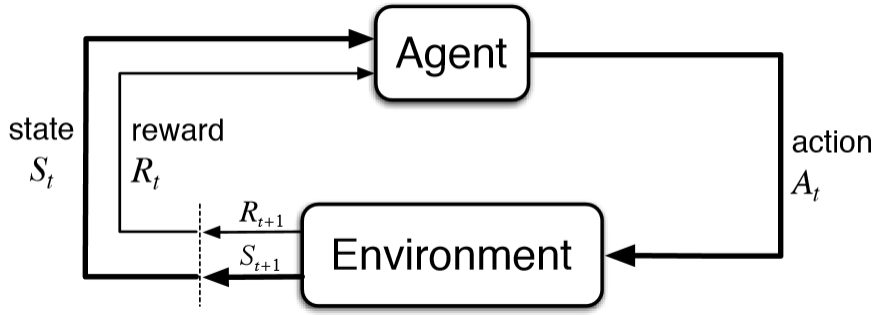


Figure 2.1 *The basic reinforcement learning model*

Source: https://adeshpande3.github.io/Deep-Learning-Research-Review-Week-2-Reinforcement-Learning

MDPs comprises:

- a set of agent states S and a set of actions A.
- a transition probability function T: $S \times A \in [0, 1]$, which maps the transition to probability. $T(s, a, s')$ represents the probability of making a transition taking action a from state s to state s'.
- an immediate reward function R: $S \times A \in R$, the amount of reward (or punishment) the environment will give for a state transition. $R(s, s')$ represents the immediate rewards after transition from s to s' with action a.

The premise of MDPs is Markov assumption, which explaining as the probability of the next state depends only on the current state and the taken action, but not on preceding states and actions. From the start of the transition to the end is called a episode. One episode of MDPs forms as a sequence: $< s_0, a_0, r_1, s_1 >, < s_1, a_1, r_2, s_2 >, \cdots, < s_{n-1}, a_{n-1}, r_n, s_n >$. In MDPs, if both the transition probabilities and reward function are known, reinforcement learning problem can be seen as an optimal control problem[66]. Actually, Both RL and optimal control solve the problem of finding an optimal policy.

Before starting to get optimal policies, we need to define what our model optimality is. More precisely, in RL, it is not enough to only consider the immediate reward of the current state, the far-reaching rewards should also be considered. But how can we define a reward model? [31] defined three models of optimal behavior.

- finite-horizon model: $\mathbf{E}(\sum_{t=0}^{h} r_t)$

- infinite-horizon discounted model: $\mathbf{E}(\sum_{t=0}^{\infty} \gamma^t r_t)$, $0 < \gamma < 1$
- average-reward model: $\lim_{h\to\infty} \mathbf{E}(\frac{1}{h} \sum_{t=0}^{h} r_t)$

The chosen of these models depends on the characteristic and requirements of the application. In this paper, our formulas are based on the infinite-horizon discounted model.

After determining one appropriate optimal behavior model, now we can start thinking about algorithms for learning to get optimal policies. According to the summarize of [37], reinforcement learning algorithms can be sorted into two classes.

Value function based RL algorithms
The value function[31] can be represented as reward function($V_\pi(s)$), which can be defined as:

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s')$$

The optimal value function($V_\pi^*(s)$) selects the maximum value among all $V_\pi(s)$ at state s and can be defined as:

$$V_\pi^*(s) = \max_a \left(R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi^*(s') \right)$$

The optimal policy($\pi^*(s)$) would be:

$$\pi^*(s) \ = \arg\max_a \left(R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi^*(s') \right)$$

Many of the reinforcement learning literature has focused on solving the optimization problem using value function. It can be split mainly into Dynamic programming based methods, Monte Carlo methods, Temporal Difference methods.

Policy search RL algorithms
We may broadly break down policy-search methods into black box and white box methods. Black box methods are general stochastic optimization algorithms using only the expected return of policies, estimated by sampling, and do not leverage any of the internal structure of the RL problem. White box methods take advantage of some of additional structure within the reinforcement learning domain, including, for instance, the (approximate) Markov structure of problems, developing approximate models, value-function estimates when available, or even simply the causal

ordering of actions and rewards. There are still discussions about the benefits of both black-box and white-box methods. As[19] described, white-box methods have advantage of leveraging more information, and the disadvantage which could be the advantage of black-box methods is that the performance gains are trade-off with additional assumptions that may be violate and less mature optimization algos with exception of models. The core of policy search methods is iterative updating the policy parameters θ, so that the expected return J will be increased. The optimize process can be formalized as following:

$$\theta_{i+1} = \theta_i + \Delta\theta_i$$

θ_i is a set of policy parameters which is parametrized of existing policies π, $\Delta\theta_i$ is the changes in the policy parameters.

model-free and model-based RL methods

Some papers sort RL methods into model-free and model-based methods. A problem can be called an RL problem is depended on the agent does not know all the elements of the MDP. Reinforcement learning is primarily concerned with how to obtain the optimal policy when MDPs model is not known in advance[31]. The agent must interact with its environment directly to obtain information which, can be processed to produce an optimal policy. At this point, there are two ways to proceed[52].

- Model-based: The agent attempts to sample and learn the probabilistic model and use it to determine the best actions it can take. In this flavor, the set of parameters that was vaguely referred to is the MDP model.
- Model-free: The agent doesnt bother with the MDP model and instead attempts to develop a control function that looks at the state and decides the best action to take. In that case, the parameters to be learned are the ones that define the control function.

One way to distinguish model-based and model-free methods is: whether the agent can make predictions about what the next state and reward will be before it takes each action after learning. If it can, then its a model-based RL algorithm. if it cannot, its a model-free algorithm. Both methods has their pros and cons. Model-free methods almost can be guaranteed to find optimal policies eventually and use very little computation time per experience. However, they make extremely inefficient use of the data during the trials and therefore often require a great deal of experience to achieve good performance. These model-based algorithms can overcome this problem, but agent only learn for the specific model, sometimes it is not suitable

some other model, and it also cost time to learn another model.

Figure 2.2 shows the category of reinforcement learning methods according to[37]. In following, we focus on Q-learning, which is a classical value function based method, belongs to Temporal Difference methods; and policy gradient, which belongs to Policy search algorithms.

Exploration-Exploitation
AI tries out actions it has never seen before at the start of the training (exploration). However, as weights are learned, the AI converges to a solution (a way of playing) and settles down with that solution (exploitation). If we choose an action that "ALWAYS" maximize the "Discounted Future Reward", you are acting greedy. This means that you are not exploring and you could miss some better actions. This is called exploration-exploitation problem. Here we discuss two action choosing approaches(exploration approaches) for discrete actions: $\epsilon - greedy$ policy and $Boltzmann$ policy.

- $\epsilon - greedy$ policy
  In this approach the agent chooses what it believes to be the optimal action most of the time, but occasionally acts randomly. This way the agent takes actions which it may not estimate to be ideal, but may provide new information to the agent. The ϵ in $\epsilon - greedy$ is an adjustable parameter which determines the probability of taking a random, rather than principled, action. Due to its simplicity and surprising power, this approach has become the common used technique for most recent reinforcement learning algorithms, including DQN and its variants. During the training, we usually do some adjustments. At the start of the training process the e value is often initialized to a large probability, to encourage exploration in the face of knowing little about the environment. The value is then annealed down to a small constant (often 0.1), as the agent is assumed to learn most of what it needs about the environment.

- $Boltzmann$ policy
  In exploration, we would ideally like to exploit all the information present in the estimated Q-values produced by our network. Boltzmann exploration does just this. Instead of always taking the optimal action, or taking a random action, this approach involves choosing an action with weighted probabilities. To accomplish this we use a softmax over the networks estimates of value for each action. In this case the action which the agent estimates to be optimal is most likely (but is not guaranteed) to be chosen. The biggest advantage over e-greedy is that information about likely value of the other actions can also

be taken into consideration. If there are 4 actions available to an agent, in e-greedy the 3 actions estimated to be non-optimal are all considered equally, but in Boltzmann exploration they are weighed by their relative value. This way the agent can ignore actions which it estimates to be largely sub-optimal and give more attention to potentially promising, but not necessarily ideal actions. In practice we utilize an additional temperature parameter (τ) which is annealed over time. This parameter controls the spread of the softmax distribution, such that all actions are considered equally at the start of training, and actions are sparsely distributed by the end of training. The following equation shows the Boltzmann softmax equation.

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)}$$

For policy gradient methods, there exists another exploration strategy. Because it is not easy to select a random action for continuous actions. They constructed the policy by adding noise sampled from a noise process N.

2.1.1 Q_learning

Q_learning is a value function based RL algorithm which learning an optimal policy, we also call it a Temporal Difference method. In 1989, Watkins proposed Q-learning algorithm which is typically easier to implement[15]. Nowadays, many RL algorithms are based on it, for example, Deep Q Network uses Q-learning as optimal policy learning combining with Neural Network as the function approximation. We will talk later in section 2.3.2. First of all, we need to understand the term "Temporal Difference". One way to estimate the value function is using difference between the old estimate and a new estimate of the value function, and the reward received in the current sample. One classical algorithm is called TD(0) algorithm proposed by Sutton in 1988. The update rule is:

$$V(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$$

After using TD(0) method to calculate the estimate of the value function, $\pi(s) = \arg\max_a V(s)$ is used to decide the optimal policy.

Q-learning method combine the 'estimate value function' part and 'define the optimal policy' part together. To understand Q-learning, we need a new notation $Q(s, a)$. $Q(s, a)$ represents the state-action value, the expected discounted value of

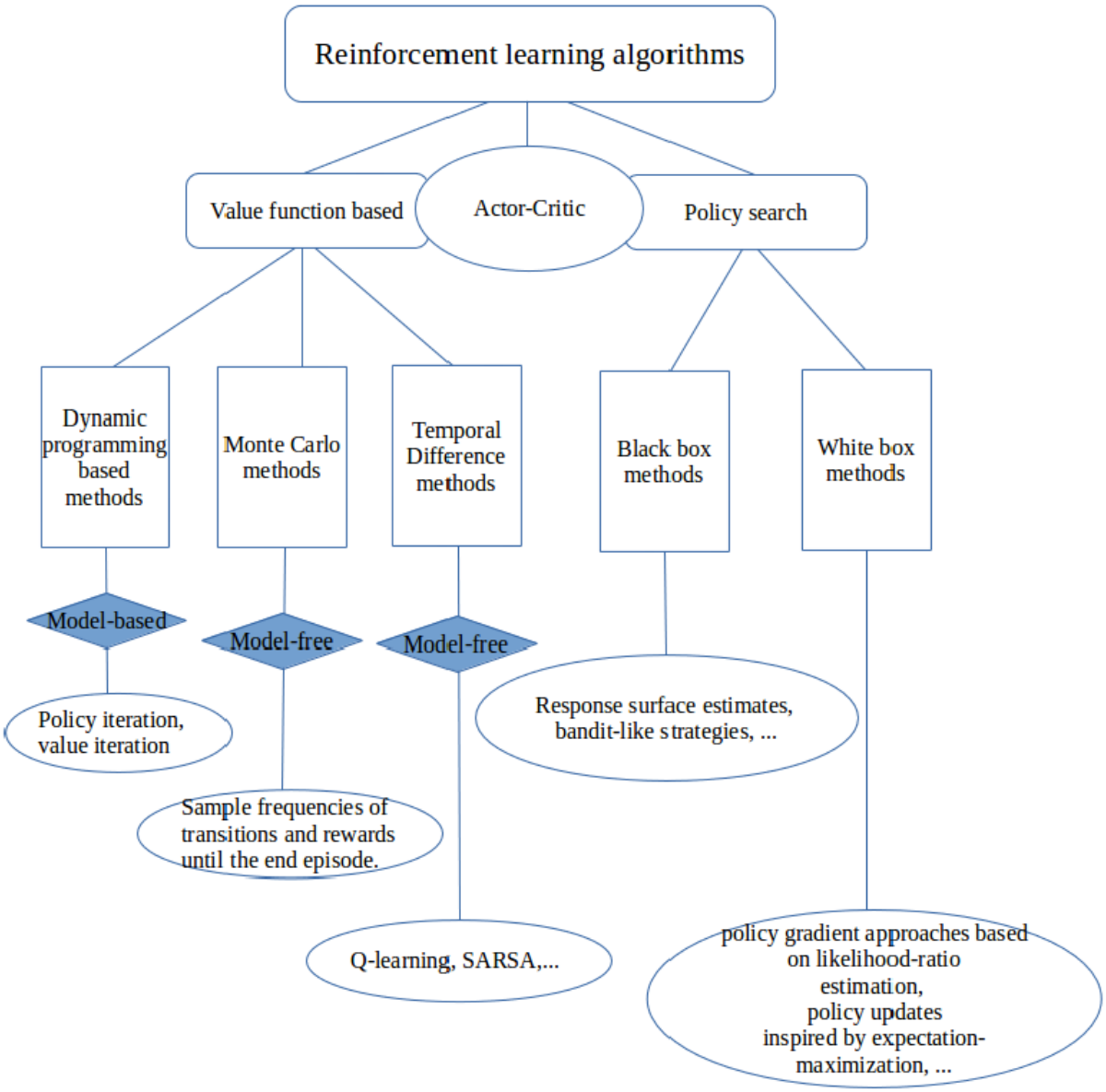


Figure 2.2 *Category of Reinforcement learning methods*

taking action a in state s. As we described before, $V(s)$ is the optimal policy, the value of take the best action in state s. Therefore, $V(s) = \max_a Q(s, a)$.

As $Q(s, a)$ is defined as the reinforcement signal at state s, taking a specific action a. We can estimate the Q value using TD(0) method. Since we define the optimal policy by choosing the action with maximum Q value in state s, we can modify the TD(0) method to be the Q-leaning method which combine estimate Q value and define the policy together. The Q-learning rule is:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

It is also called *bellman equation*. s represents the current state . a represents the action the agent takes from the current state. s' represents the state resulting from the action. a' represents the action the agent takes from the next state. r is the reward you get for taking the action, γ is the discount factor, α is the learning rate. So, the Q value for the state s taking the action a is the sum of the instant reward and the discounted future reward (value of the resulting state). The discount factor γ determines how much importance you want to give to future rewards. Say, you go to a state which is further away from the goal state, but from that state the chances of encountering a state with snakes is less, so, here the future reward is more even though the instant reward is less. α determines how much the agent from this estimate.

2.1.2 Policy Gradient

In addition to Value-function based methods which are represented by Q-learning, there is another category called Policy search methods. Inside policy search RL algorithms, Policy Gradient algorithms are the most popular among them. Policy Gradient algorithms belong to gradient-based approach which is a kind of white-box method. Computing changes in policy parameters $\Delta\theta_i$ is the most important part, several approaches are available now. The gradient-based approaches use gradient of the expected return J multiplies by learning rate α to compute the changes which represent as $\alpha\nabla_\theta J$. Therefore, the optimize process form can be written as:

$$\theta_{i+1} = \theta_i + \alpha\nabla_\theta J$$

There exist several methods to estimate the gradient $\nabla_\theta J$ [60], figure 2.3 shows different approaches to estimate the policy gradient. Generally, there are Regular Policy Gradient and Natural Policy Gradient Estimation. Regular Policy Gradient methods include Finite-different methods which is known as PEGASUS in reinforcement learning[54], Likelihood Ratio method[23] (or REINFORCE algorithms[82]). Policy gradient theorem/GPOMDP[75, 6] and Optimal Baselines strategy are the improved approaches based on Likelihood Ratio method. Natural Policy Gradient estimation was proposed by [32], based on this, [63] proposed the Episodic Natural Actor-Critic.

Here we discuss *policy gradient theorem/GPOMDP*. Since it based on likelihood ratio methods, we give a general idea about likelihood ratio methods. Likelihood ratio is known as:

$$\nabla_\theta P^\theta(\tau) = P^\theta(\tau)\nabla_\theta \log P^\theta(\tau)$$

where $P^\theta(\tau)$ is the episode(τ) distribution of a set of policy parameters(θ).

The expected return for a set of policy parameter(θ) J^θ can be written as:

$$J^\theta = \sum_\tau P^\theta(\tau)R(\tau)$$

where $R(\tau)$ is the total rewards in episode τ. $R(\tau) = \sum_{h=1}^{H} a_h r_h$, where a_h denote weighting factors according to step h, often set to $a_h = \gamma_h$ for discounted reinforcement learning (where γ is in [0, 1]) or $a_h = \frac{1}{H}$ for the average reward case.

Combine two equations above, the policy gradient can be estimated as following:

$$\nabla_\theta J^\theta = \sum_\tau \nabla_\theta P^\theta(\tau)R(\tau) = \sum_\tau P^\theta(\tau)\nabla_\theta \log P^\theta(\tau)R(\tau) = \mathbf{E}\{\nabla_\theta \log P^\theta(\tau)R(\tau)\}$$

If the episode τ is generated by a stochastic policy $\pi^\theta(s, a)$, we can directly express this equation:

$$P^\theta(\tau) = \sum_{h=1}^{H} \pi^\theta(s_h, a_h)$$

Therefore,

$$\nabla_\theta \log P^\theta(\tau) = \sum_{h=1}^{H} \nabla_\theta \log \pi^\theta(s_h, a_h)$$

$$\nabla_\theta J^\theta = \mathbf{E}\{(\sum_{h=1}^{H} \nabla_\theta \log \pi^\theta(s_h, a_h))R(\tau)\}$$

In practice, likelihood ratio method is often advisable to subtract a reference, also called baseline b, from the rewards of the episode $R(\tau)$:

$$\nabla_\theta J^\theta = \mathbf{E}\{(\sum_{h=1}^{H} \nabla_\theta \log \pi^\theta(s_h, a_h))(R(\tau) - b_h)\}$$

Despite the fast asymptotic convergence speed of the gradient estimate, the variance of the likelihood-ratio gradient estimator can be problematic in practice[61]. The policy gradient theorem improved likelihood ratio methods with $R(\tau)$. Before, we defined $R(\tau) = \sum_{h=1}^{H} a_h r_h$, which represents that the reward of episodeτ considers all steps($h \in [1, H]$). However if we consider the characters of RL, MDPs have the

precondition that future actions do not depend on past rewards (unless the policy has been changed). Depending on this, we can improve the likelihood ratio method and it can result in a significant reduction of the variance of the policy gradient estimate.

We redefine $R(\tau) = \sum_{h=k}^{H} a_k r_k$, which means that we only consider the future rewards. Therefore, the equation of likelihood ratio policy gradient can be written as:

$$\nabla_\theta J^\theta = \mathbf{E}\{(\sum_{h=1}^{H} \nabla_\theta \log \pi^\theta(s_h, a_h))(\sum_{k=h}^{H} a_k r_k - b_h)\}$$

we note that the term $R(\tau) = \sum_{h=k}^{H} a_k r_k$ in the policy gradient theorem is equivalent to a Monte-Carlo estimate of the value function $Q^\pi(s_h, a_h)$, so we can rewrite the equation as:

$$\nabla_\theta J^\theta = \mathbf{E}\{(\sum_{h=1}^{H} \nabla_\theta \log \pi^\theta(s_h, a_h))(Q^\pi(s_h, a_h) - b_h)\}$$

Actually, in this equation, it uses Q-value function to participate in updating the policy, we could also call it a Q actor-critic method.

2.1.3 Limitations of RL in practice

Value function approaches(e,g. Q-learning) theoretically require total coverage of state space and the correspond reinforce values of all the possible actions at each state. Thus, the computational complexity could be very high when dealing with high dimensional applications. And even small change of the local reinforce values may cause large change in policy. At the same time, finding an appropriate way to store the huge data becomes a significant problem.

In contrast to value function methods, policy search methods(e,g. Policy Gradient) consider the current policy and the next policy to the current one, then computing the changes in policy parameters. The computational complexity is far less then value function methods. More then that, these methods are also available to continuous features. However, due to above theory, the policy search approaches may cause local optimal, and even cannot reach global optimal.

A combination of value function and policy search approaches called *actor-critic structure*[5] was proposed to fusing both advantages. The "Actor" is known as

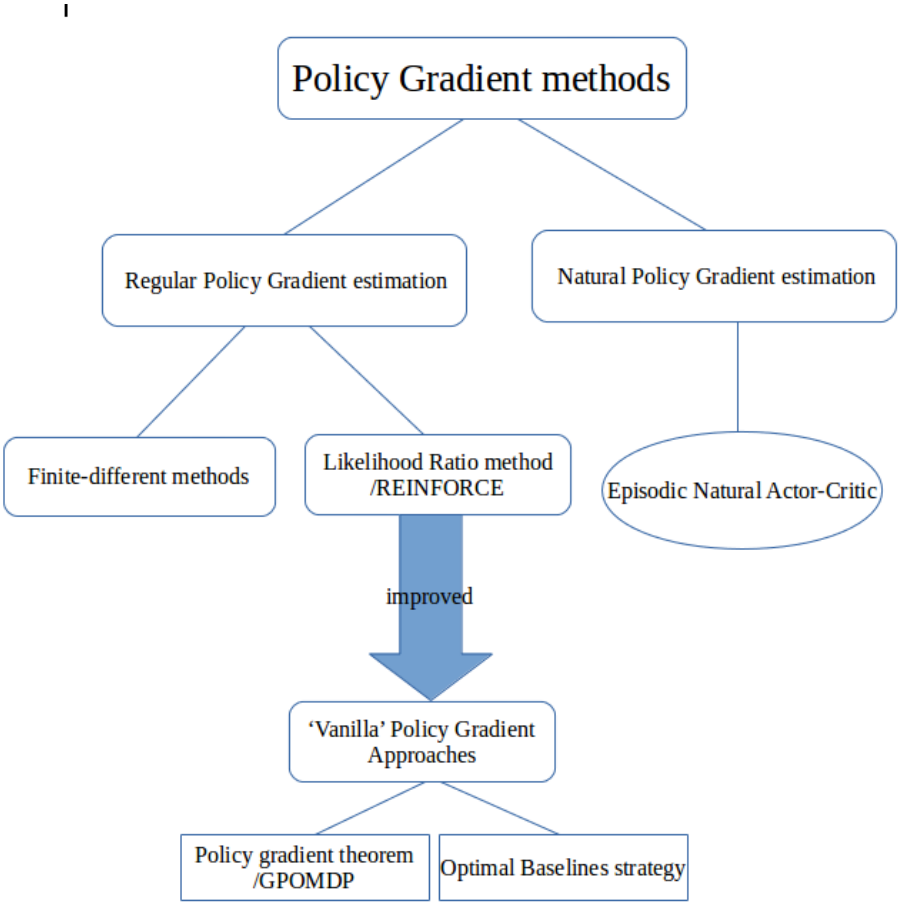


Figure 2.3 *Policy Gradient methods*

control policy, the "Critic" is known as value function. As figure 2.4 shows, the action selection is controlled by Actor, the Critic is used to transmit the values to Actor, so that deciding when the policy to be updated and preferring the chosen action.

Although there are several methods in RL fitting different kind of problems, these methods all share the same intractable complexity issues, for example memory complexity. Searching for a suitable and powerful *function approximation* becomes the imminent issue. The Function Approximation is a family of mathematical and statistical techniques used to represent a function of interest when it is computationally or information-theoretically intractable to represent the function exactly or explicitly. Typically, in reinforcement learning the function approximation is based on sample data collected during interaction with the environment[36]. Function approximation to date is investigated extensively, and since the fast development of deep learning, the powerful function approximation: deep neural network can solve these complexity issues. We will discuss in next section with focus on deep learning and artificial neural network.

As [46] described, there are two issues we must overcome in traditional RL. First is to reduce the learning time. Second is how to use RL methods when real-world applications don't follow Markov decision process. We will discuss techniques around these issues in Deep reinforcement learning section.

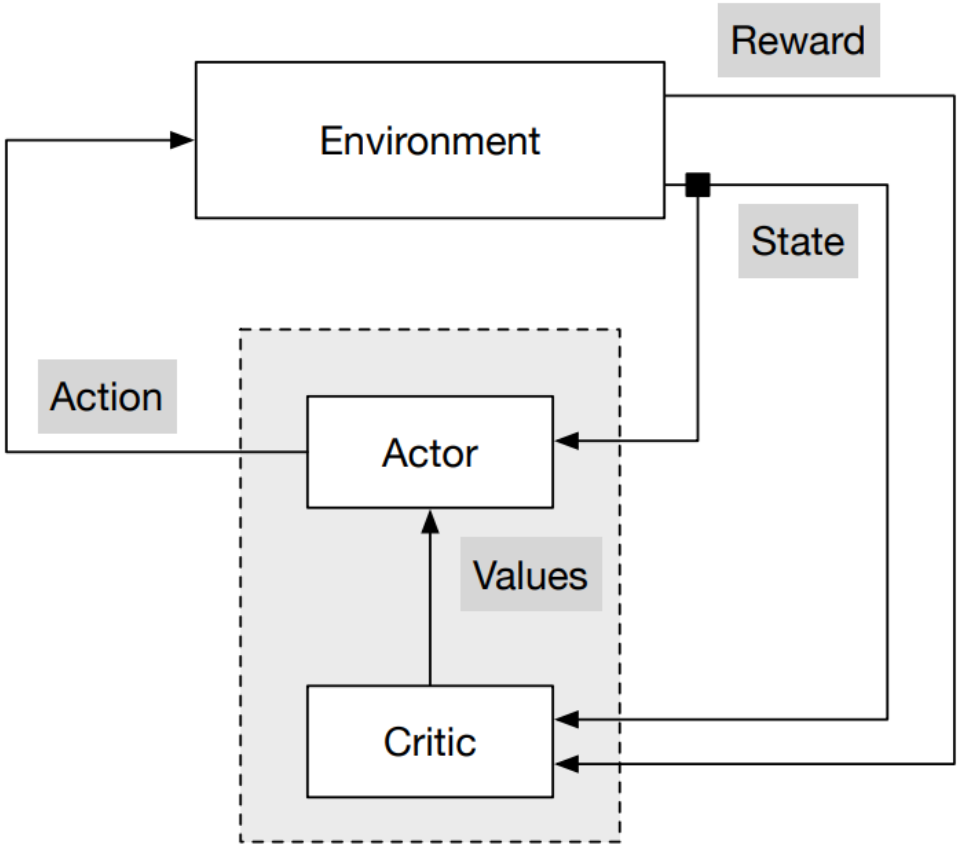


Figure 2.4 *Actor-Critic Architecture*

Source: http://mi.eng.cam.ac.uk/ mg436/LectureSlides/MLSALT7/L5.pdf

2.2 Deep Learning

Machine-learning systems are used to identify objects in images, transcribe speech into text, match news items, posts or products with users interests, and select relevant results of search. Increasingly, these applications make use of a class of techniques called *deep learning*[39]. Deep learning algorithms rely on deep neural network. Deep neural networks can automatically find compact low-dimensional representations (features) of high-dimensional data (e.g., images, text and audio)[3]. Deep learning has accelerated the progress in many other machine learning fields, like supervised learning, reinforcement learning.

Let's discuss artificial neural network architecture. A deep neural network(DNN) consists multiple layers of nonlinear processing units (hidden layers). It performs feature extraction and transformation. As figure 2.5 shows, a deep neural network (DNNs) consists of three layers, input layer, hidden layers, output layer. In input layer, the neurons are generalized from *features* getting through sensors perceiving the environment. The hidden layers may include one or more layers, neurons on them are called *feature representations*. The output layer contains the outputs which we want, for example, the distribution of all possible actions. Each successive layer of DNN uses the output from the previous layer as input. All the neurons of the layers are fully activated through weighted connections. As the input layer neurons are known from environment, how could we calculate the hidden layers' neurons(feature representation)? Actually, the whole DNNs are mathematical functions. We use the input and the first hidden layers as an example. Notating the neurons in the input layer and the first hidden layer as $a^{(0)}$ and $a^{(1)}$, the *weights* of all connections between the two layers as W. Also we define a pre-determined number called bias b. The function of the first hidden layer would be:

$$a^{(1)} = Wa^{(0)} + b$$

However, in real-world applications, there couldn't be linear function above all the time, most of the time, it is nonlinear transformation. We need active function to make the function nonlinear, so that different applications can be satisfied. Thus,

$$a^{(1)} = AF\{Wa^{(0)} + b\}$$

Different active functions could be used to solve different problems, you can also create your own active function to fit your own issue. There are also several pre-defined AFs, such like 'relu', 'tanh'.

After computations flow forward from input to output, at output layer and each hidden layer, we can compute error derivatives backward, and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function[44]. This is the core of learning part, to find the right weights and biases.

Still, DNNs have a knotty issue to solve: Statistical Invariance or Translation Invariance. Imaging we have two images with exactly same kitty on them, the only difference is the kitty locates in different position. If we want the DNNs to train to recognize it is a cat on both images, the DNNs should give different weights. Same issue comes from text or sequences recognition. One way to solve it is Weight Sharing. For image, people built the Convolution Neural Network(CNN) structure,

for text and sequences, people built Recurrent Neural Network(RNN). Simply say, CNNs use shared parameters across space to extract pattens over image, RNNs do same thing across time instead of space.

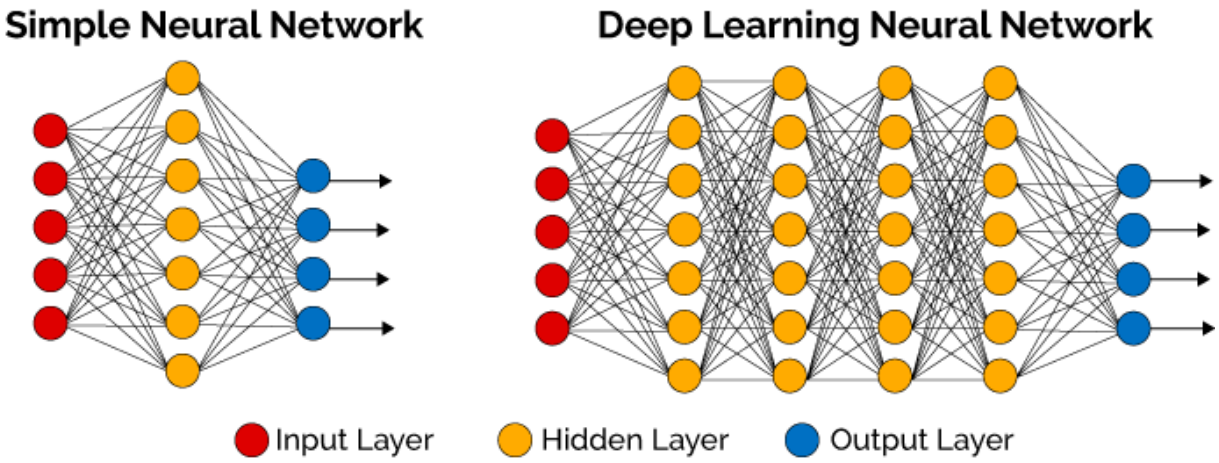


Figure 2.5 *Simple NNs and Deep NNs*

Source: https://www.quora.com/What-is-the-difference-between-Neural-Networks-and-Deep-Learning

2.2.1 Convolution Neural Network(CNN)

As we talked before, CNN is used to solve Translation Invariance issue, sharing their parameters across space. CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture[13]. In a CNN, neurons are arranged into layers, and in different layers the neurons specialize to be more sensitive to certain features. For example in the base layer the neurons react to abstract features like lines and edges, and then in higher layers neurons react to more specific features like eye, nose, handle or bottle.

CNN architectures mainly consist of three types of layers: Convolutional Layer, Pooling Layer, and Fully-Connected Layer. We will stack these layers to form different CNN architectures.

- **Convolutional layer:** We use a small size($m * m$) patch(filter) to scan over the whole image($w * h * d$) with stride s, each step of the patch go through a neural network to get output, also we call the procedure mathematically as convolution. Then combining these procedures, we get a new represented image with new width, height, depth($w' * h' * d'$). The whole "scan over" process is also called padding. There are two padding types: Same padding

and Valid padding. The difference is whether going pass the edge of input image or not.

- **Pooling Layer:** Pooling layers performs a down sampling operation (subsampling) and reduces the input dimensions. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting[13].There are many types of pooling layers: max pooling, average pooling.

- **Fully-Connected Layer:** Same like regular deep neural network layer: Neurons full connect to all activations in the previous layer.

There are some famous CNN structures with given games: "LeNet-5" in figure 2.6, "AlexNet" in figure 2.7. Other architctures are summarized in table 2.8.

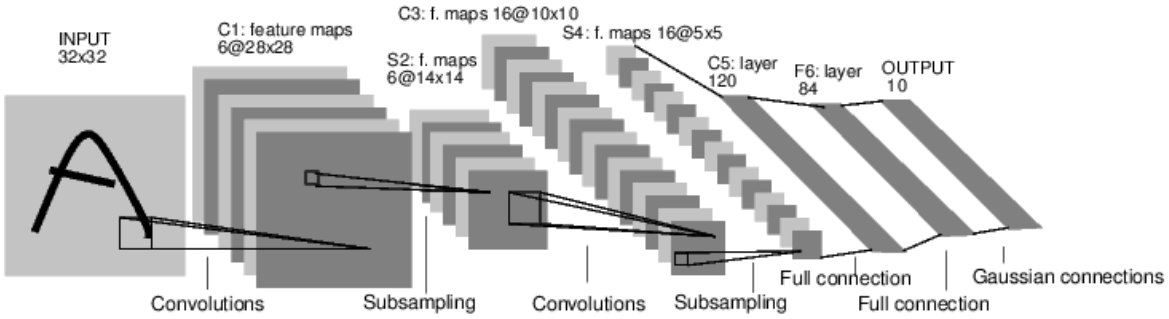


Figure 2.6 *Architecture of LeNet-5*

Source: from: [40]

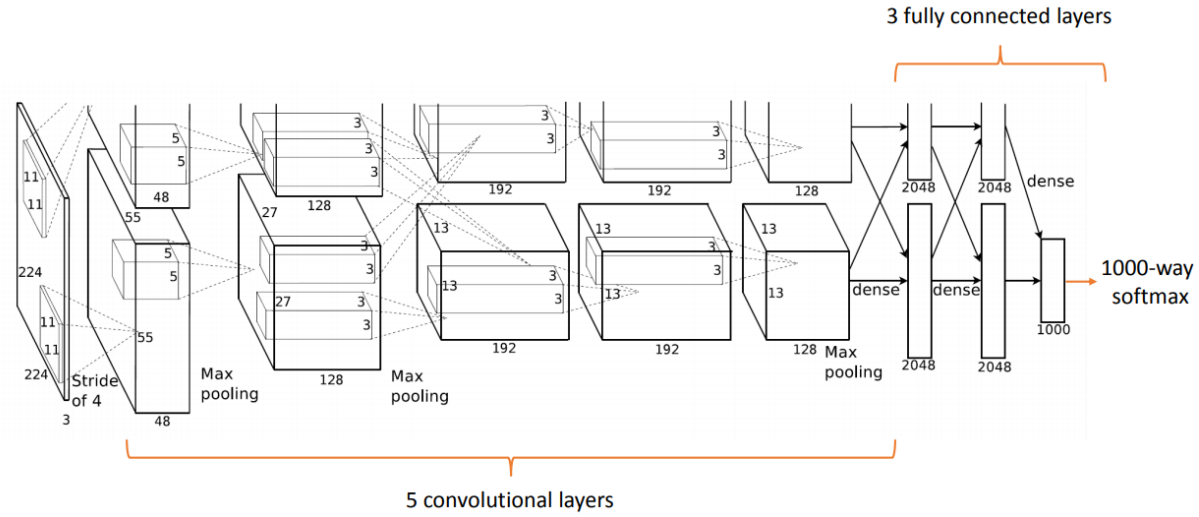


Figure 2.7 *Architecture of AlexNet*

Source: from: [38]

2.2.2 Recurrent Neural Network(RNN)

As discussed before, RNN shares parameters(weights) to process text over time. In another words, RNN deals with sequential data. In a traditional neural network we assume that all inputs and outputs are independent of each other. However, for many tasks, if we want to predict the output, it's better to know the previous inputs. Fro example, we want to translate the sentence, we better know the whole input sentence and the order of the sequence.

A typical RNN structure shows in figure 2.9. By unfolding, we can consider that each element of the sequence can be unrolled into one layer of neural networks. x_t is the input at time step t; s_t is the hidden state at time step t, calculated by $s_t = AF(Ux_t + Ws_{t-1})$. The first hidden states$_0$ is typically initialized to all zeroes; o_t is the output at time step$t$.

During the learning process, we do backpropagation to update the weights W, RNN is a "memory" neural network, we need to backpropagate the derivative through time, all the way to the beginning or to some afford point. All the derivatives will multiply the same weight W. If W is bigger than 1, mathematically we know, to the beginning, the updated weight will be infinities(Gradient exploding). Otherwise, if W is smaller than 1, the beginning weight will be 0(Gradient vanishing). To fix Gradient exploding, we can use *Gradient clipping*[57] to limit a maximum bound to prevent. To deal with Gradient vanishing, we combine a control system with RNN, which called Long Short-Term Memory(LSTM)[27]. LSTMs dont have a

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Figure 2.8 *Summary table of famous CNN architectures*

Source: https://medium.com/@siddharthdas_32104/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

fundamentally different architecture from RNNs, but they use a different function to compute the hidden state.

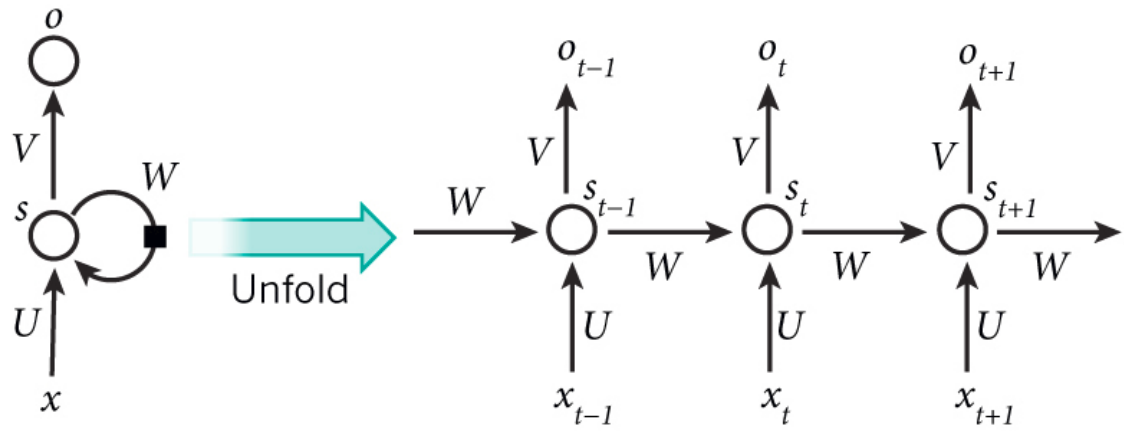


Figure 2.9 *Typical RNN structure*

Source: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

2.2.3 hyperparameters

Deep neural network structure could be plentiful, building a suitable structure is necessary for solving problems. In order to build a deep neural network, first thing to consider is hyperparameters. Here we list the considerable hyperparameters in this paper.

Ordinary Neural Networks

- number of layers
- number of hidden-layers' neurons
- initial weights
- choose of active functions
- choose of loss functions
- optimization types

CNNs specialization

- number of convolutional layers
- number of FC layers

- patch/filter size
- padding types
- pooling types
- layers' connection types

RNNs specialization

- suitable structure related to specific problem
- hidden state initialization
- design of LSTM

2.3 Deep Reinforcement Learning

2.3.1 Overview

As we discussed in section 2.1.3, value function methods and policy search methods have their own pros and cons, they have different application domains. However, RL methods share the same complexity issues. When dealing with high-dimensional or continuous action domain problems, RL suffers from feature representation. Therefore, the learning time of RL is slow and techniques for speeding up the learning process must be devised. As the development of Deep neural network which belongs to Deep learning domain,a new arising field *Deep Reinforcement Learning* showed up to solving RL in high dimensional domain. The most important property of deep learning is that deep neural networks can automatically find compact low-dimensional representations of high-dimensional data, so that DRL break the "Curse of dimension".

Let's take Q-learning as the example. Q-learning algorithm stores state-action pairs in a table, a dictionary or a similar kind of data structure. The fact is that there are many scenarios where tables don't scale nicely. Let's take "Pacman". If we implement it as a graphics-based game, the state would be the raw pixel data. In a tabular method, if the pixel data changes by just a single pixel, we have to store that as a completely separate entry in the table. Obviously that's silly and wasteful. What we need is some way to generalize and pattern match between states and actions. We need our algorithm to say "the value of these kind of states

is X" rather than "the value of this exact, super specific state is X." Due to this, people replaced tabular with deep neural networks, combining with Q-learning policy update method, a new Deep Reinforcement learning method appeared. It is called *Deep Q Network*. Since Q-learning is a value function based method, it inherits the pros and cons from value function methods.

We could also combine deep neural networks with policy search methods and with Actor-Critic method which approximating value function and direct policy. In following we discuss four state-of-the-art algorithms, two deep policy search methods: Deep Deterministic Policy Gradient and Proximal Policy Optimization, also an asynchronous deep actor critic method called Asynchronous Advanced Actor Critic. These methods are currently most popular and effective algorithms, proposed by DeepMind and OpenAI. In this paper, these four methods are used for experiments. We will present the theoretical background in detail as well as their advantages and disadvantages.

2.3.2 Deep Q Network

Deep Q Network was first proposed by [51], it presents the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. More precisely, DQN in paper[51] used the images showed on the Atari emulator as input, using convolution neural network to process image data. Q-learning algorithm was used to make decision, with stochastic gradient descent to update the weights. Since deep learning handle only with independent data samples, *experience replay* mechanism was used to break correlations. Generally, DQN algorithm replaces the tabular representation for Q-value function with deep neural network(Figure 2.10).

Function approximation
Basically we get the Q function by experience, using an iterative process called *bellman equation* which is introduced in section 2.1.1. Because in reality the action-value function(Q-value) is estimated separately for each sequence without any generalization, the basic approach to converge to the optimal Q-value is not practical. We use a deep neural network with weight θ as the function approximation to estimate the Q-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. So the network is trained by minimizing the loss function $L(\theta_t)$ at time step t. The loss function in this DQN case is the difference between Q-target and Q-predict.

$$L(\theta_t) = \mathbf{E}_{s,a}[(Q_{target} - Q_{predict})^2]$$

$$Q_{target} = r + \gamma \max_{a'} Q(s', a'; \theta_t); Q_{predict} = Q(s, a; \theta_t) \tag{2.1}$$

Then we use stochastic gradient descent to optimize the loss function.

Experience Replay

Reinforcement learning with value function based methods must overcome two issues when combining with deep learning. Fist, deep learning assumes data samples to be independent, however, the training data of reinforcement learning are collected by the sequence correlated states which leaded out by actions chosen. Second, the collected data distributions of RL are non-stationary because RL keep learning new behaviors. But for deep learning, we need a stationary data distribution.

Experience replay mechanism was used to DQN, it was first proposed by LJ Lin(1993)[46]. It aims to break correlations between data samples, also it can smooth the training data distribution. During RL playing, the transitions $T(s, a, r, s')$ are stored in the *experience buffer*, after enough number of these transitions, we randomly sample a *mini-batch* sized data from the experience buffer, and handle them to the network for training. Necessarily, The buffer size must much larger than the mini-batch size. This is how this mechanism works. Therefore, two hyperparameters could be controlled during DRL method designing and evaluation, the buffer size and mini-batch size.

Fixed Q-target

Another break correlation mechanism called *Fixed Q-target*. We produce two neural network structure for DQN, with the same structure but difference parameters(weights). In equation 2.2, we compute Q-target using current weights, and Q-predict is get from Q-network with newest weights. In this mechanism, we fixed the NN with k time-step-old weights θ_{t-k}, which is used for calculating Q-target. Then periodically update fixed weights in the NN.

$$Q_{target} = r + \gamma \max_{a'} Q(s', a'; \theta_{t-k}); Q_{predict} = Q(s, a; \theta_t) \tag{2.2}$$

Since the compute of Q-target use the old parameters and Q-predict use the current parameters, this can also break data correlation efficiently. Moreover, since policy changes rapidly with slight changes to Q-values, the policy may oscillate. This mechanism can also avoid oscillations. 2.9 shows the complete pseudocode of Deep Q Network with experience reply which be produced by [51].

Deep Q Network algorithm represents value function by deep Q-network with weights θ. It is a model-free, off-policy strategy. It inherits the characteristics of value function based RL methods. Besides, DQN is a flexible method, the structure of Q

network could be the ordinary neural network, or be the convolutional neural network if directly using an image as input, or be the recurrent neural network if the input are ordered text sequences. Moreover, the hyperparameters of NNs, e.g, the layers, neurons, could also be adjusted flexibly. Recently, an advanced DQN algorithm called Doub8888le DQN was proposed by [78]. In Double DQN, the online network predicts the actions while the target network is used to estimate the Q value, which efficiently reduced the overestimation problem. The Q_{target} in Double DQN is:

$$Q_{target} = r + \gamma \max_{a'} Q(s', a'_{\theta'_t}; \theta_t) = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta'_t); \theta_t)$$

Generally, Double DQN reduces the overestimation by decomposing the max operation in the target into action selection and action evaluation. We compared both DQN algorithms, the evaluation showed in chapter 5.

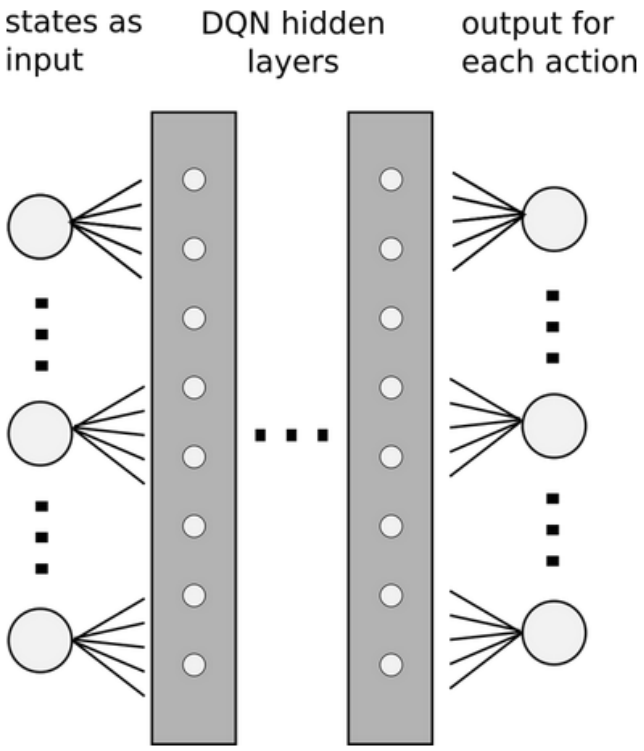


Figure 2.10 *Deep Q-learning structure*

Source: https://morvanzhou.github.io/

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2.11 *Deep Q-learning with Experience Replay*

Source: from [51]

2.3.3 Deep Deterministic Policy Gradient

Since the rise of deep neural network function approximations for learning value or action-value function, deep deterministic policy gradient method was proposed by [45]. It used an actor-critic approach based on the DPG algorithm[73], combined with *experience replay* and *fixed Q-target* techniques which inspired by DQN to use such function approximators in a stable and robust way. In this algorithm, a recent advantage in deep learning called *batch normalization*[30] is also adapted. The problem of exploration in off-policy algorithms like DDPG can be addressed in a very easy way and independently from the learning algorithm. Exploration policy is then constructed by adding noise sampled from a noise process N to the actor policy.

Deterministic Policy Gradient with neural networks

Deterministic Policy Gradient[73] based on Actor-Critic methods which we have discussed in section 2.1.3. For the Actor part, it replaced the stochastic policy $\pi_\theta(s)$ with a deterministic target policy $\mu_\theta(s)$by mapping states to a specific action. For the Critic part, Q-value function is estimated by Q-learning. Neural networks are used as the function approximators, there are two NN structures for Actor and Critic, we call them *Actor Network* and *Critic Network*. We denote θ^μ for the weights of the Actor neural network, θ^Q for the weights of the Critic neural network. The Critic

is updated by minimizing the loss function:

$$L(\theta^Q) = \mathbf{E}[(Q_{target} - Q_{predict})^2]; where, Q_{target} = r + \gamma Q(s', \mu(s'; \theta^\mu); \theta^Q), Q_{predict} = Q(s, a; \theta^Q)$$

The Actor is updated by maximizing the expected return J^{θ^μ}, using sampled policy gradient:

$$\nabla_{\theta^\mu} J^{\theta^\mu} \approx \mathbf{E}[\nabla_{\theta^\mu} Q(s, \mu(s; \theta^\mu); \theta^\mu)] = \mathbf{E}[\nabla_{\mu(s)} Q(s, \mu(s); \theta^Q) \nabla_{\theta^\mu} \mu(s; \theta^\mu)]$$

Innovations from DQN

As we mentioned in Deep Q Network, experience reply technique is used for breaking correlations of training data. It works by sampling a random mini-batch of the transitions stored in the buffer. As the policy changes rapidly with slight changes to Q-values, another technique called "fixed Q-target" is used for solving this issue. It not only can break correlations, but also can avoid oscillations. Differently from DQN, where the target network were updated every k steps, the parameters of the target networks are updated in DDPG case at every time step, following the "soft" update:

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

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

Therefore, Q-target can be rewritten as:

$$Q_{target} = r + \gamma Q(s', \mu_{target}(s'; \theta^{\mu'}); \theta^{Q'})$$

For the DDPG structure, there are totally four neural networks, both Actor Net and Critic Net have two neural networks with same structures, different weights. The whole pseudocode shows in figure 2.12.

Batch Normalization

Additionally, a robust strategy called batch normalization[30] is adopted to scale the range of input vector observations in order to make the network capable of finding hyper-parameters which generalize across environments with different scales of state values. This method normalizes each dimension across the samples in a mini-batch to have unit mean and variance. In deep neural network each layer thus receives whitened input. Batch normalization reduces the dependence of gradients on the scale of the parameters or of their initial value, and makes it possible to use saturating nonlinearities by preventing the network from getting stuck in the saturated modes.

Algorithm 1 DDPG algorithm

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

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

    Update the target networks:

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

  end for
end for

Figure 2.12 *DDPG Pseudocode*

Source: from [45]

2.3.4 Proximal Policy Optimization

Defining the step size(learning rate) α becomes the thorny issue in policy gradient methods. Because if the step size is too large, the policy will not coverage, opposite, to finish learning the policy will take a century. The new robust policy gradient methods, which we call proximal policy optimization[71, 26] was proposed to solve the problem, have some of the benefits of trust region policy optimization[70], but they are much simpler to implement, more general, and have better sample complexity (empirically). It bounds parameter updates to a trust region to ensure stability. Several approaches have been proposed to make policy gradient algorithms more robust. One effective measure is to employ a *trust region* constraint that restricts the amount by which any update is allowed to change the policy. A popular algorithm that makes use of this idea is trust region policy optimization[70], This algorithm is similar to natural policy gradient methods which we mentioned in section 2.1.2. PPO is one variant of TRPO, it directly uses first order optimization methods to optimize the objective.

Surrogate objective function

In we described in Policy gradient section 2.1.2, policy gradient estimator $\nabla_\theta J^\theta$, here we construct an objective function $L(\theta)$ whose gradient is the policy gradient estimator, the estimator is obtained by differentiating the objective function, where $\hat{A}$ is advantage function:

$$L(\theta) = \mathbf{E}[\log \pi_\theta(s,a)\hat{A}^\pi(s,a)]$$

We can also understand $L(\theta)$ as expected cumulative return of the policy J^θ. As we want to limit the objective function being in a maximum-minimum bound, we use "surrogate" objective function to replace the original objective. We first find a approximated lower bound of the original objective as the surrogate objective and then maximize the surrogate objective so as to optimize the original objective. The surrogate objection in TRPO is:

$$L_{\theta_{old}}(\theta) = \mathbf{E}[\frac{\pi_\theta(s,a)}{\pi_{\theta_{old}(s,a)}}\hat{A}^\pi(s,a)]$$

Trust region

Trust-region methods define a region around the current iterative within which they trust the model to be an adequate representation of the objective function, and then choose the step to be the approximate minimizer of the model in this region[55]. Simply say, during our optimization procedure, after we decided the gradient direction, we want to constrain our step size to be within a trust region so that the local estimation of the gradient remains to be trusted. In TRPO, they used *average KL divergence* between the old policy and updated policy as a measurement for trust region. The surrogate objective function is maximized subject to a constraint on the size of the policy update.

$$\begin{aligned} &\textbf{maximize } L_{\theta_{old}}(\theta) \\ &\textbf{subject to } \overline{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old},\theta) < \delta \end{aligned}$$

θold is the policy parameters before the update, δ is the constraint parameter, $\rho_{\theta_{old}}$ is the is the discounted visitation frequences in θold.

According to the theories above, PPO[71] presented two alternative ways to maximize the surrogate objective function with constrain: *Clipped Surrogate Objective* and *Adaptive KL Penalty Coefficient*.

Clipped Surrogate Objective

We denote the probability ratio as $r(\theta)$ to represent $\frac{\pi_\theta(s,a)}{\pi_{\theta_{old}(s,a)}}$.

$$L_{\theta_{old}}(\theta) = \mathbf{E}[\frac{\pi_\theta(s,a)}{\pi_{\theta_{old}(s,a)}}\hat{A}^\pi(s,a)] = \mathbf{E}[r(\theta)\hat{A}^\pi(s,a)]$$

We define a hyperparameter $\epsilon \in [0,1]$ to limit the $r(\theta)$ in the interval $[1-\epsilon, 1+\epsilon]$. So this strategy can be written as:

$$L^{CLIP} = \mathbf{E}[\min(r(\theta)\hat{A}^\pi(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^\pi(s,a))]$$

Adaptive KL Penalty Coefficient

Another approach is to use a penalty on KL divergence, and to adapt the penalty coefficient β so that we achieve some target value of the KL divergence d_{target} each policy update. Simply say, β is updated according to a certain comparison between real KL divergence d_{real} and the target divergence d_{target}. There are two steps:

1. Optimize the objective function:

$$L^{KLPEN}(\theta) = \mathbf{E}[\frac{\pi_\theta(s,a)}{\pi_{\theta_{old}(s,a)}}\hat{A}^\pi(s,a) - \beta KL[\pi_{\theta_{old}}, \pi_\theta]]$$

2. Compare d_{real} with d_{target}, adjust β:

$$\begin{cases} \beta \leftarrow \beta \div 2 & \text{if } d_{real<d_{target}\div 1.5} \\ \beta \leftarrow \beta \times 2 & \text{if } d_{real>d_{target}\times 1.5} \\ \beta & \text{otherwise} \end{cases} \tag{2.3}$$

Distributed PPO

Figures 2.13, 2.14 show the pseudocodes from OpenAI and DeepMind. Besides, DeepMind also proposed an asynchronous method for PPO called Distributed PPO, which used the similar structure as A3C. Data collection and gradient calculation are distributed over workers, it aims to achieve good performance in rich, simulated environments. In this paper, we implement and evaluate this method. It uses the pseudocode from OpenAI and combines the asynchronous method which A3C uses. We will discuss the asynchronous method detail in next section.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 2.13 *PPO Pseudocode by OpenAI*

Source: from [71]

Algorithm 1 Proximal Policy Optimization (adapted from [8])

for $i \in \{1, \cdots, N\}$ **do**
  Run policy π_θ for T timesteps, collecting $\{s_t, a_t, r_t\}$
  Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $j \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \text{KL}[\pi_{old}|\pi_\theta]$
    Update θ by a gradient method w.r.t. $J_{PPO}(\theta)$
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
    Update ϕ by a gradient method w.r.t. $L_{BL}(\phi)$
  end for
  if $\text{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \alpha\lambda$
  else if $\text{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\alpha$
  end if
end for

Figure 2.14 *PPO Pseudocode by DeepMind*

Source: from [26]

2.3.5 Asynchronous Advanced Actor Critic

Asynchronous Advanced Actor Critic[50] is an asynchronous method using Advanced Actor Critic(Q Actor Critic in section 2.1.2). Asynchronous means Asynchronously execute multiple agents in parallel, on multiple instances of the environment and all using a replica of the NN (asynchronous data parallelism), it often works in a multi-core CPU or GPU. As in figure 2.15 shows, there is a global network and multiple actor-learners which have their own set of network parameters. A thread is dedicated for each agent, and each thread interacts with its own copy of the environment. Given each thread a different exploration policy also improves robustness, since the overall experience available for training becomes more diverse.

Moreover, in A3C just one deep neural network is used both for estimation of policy $\pi(s)$ and value function $V_\pi(s)$; because we optimize both of these goals together, we learn much faster and effectively(figure 2.16). We also don't need to consider the data correlation and oscillations issues because different agent get different transitions when playing in same environments.

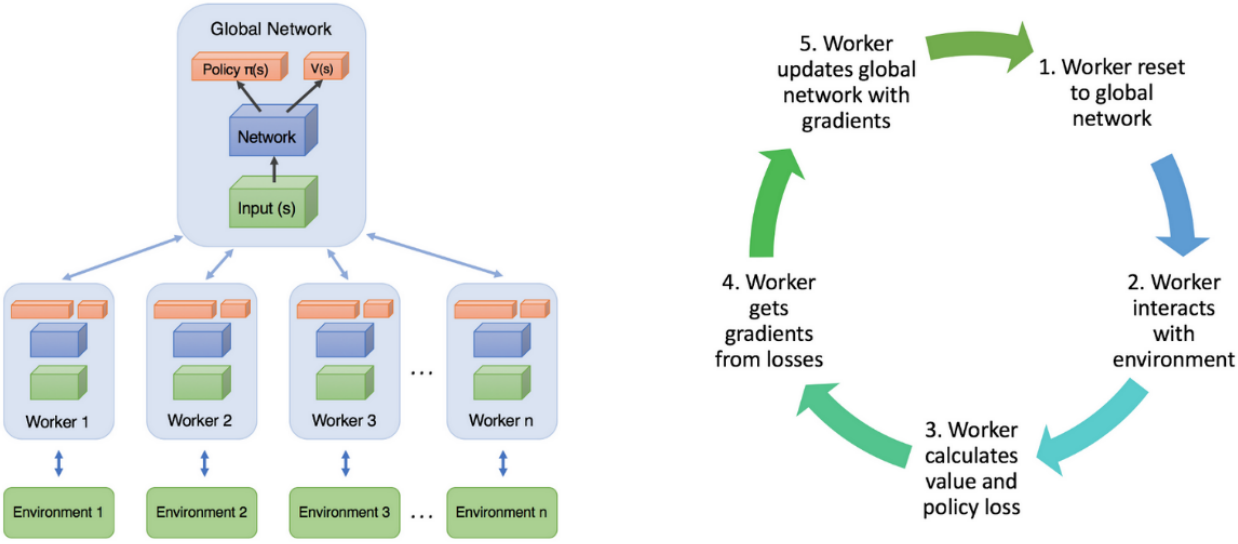


Figure 2.15 *A3C procedure*

Source: https://morvanzhou.github.io/

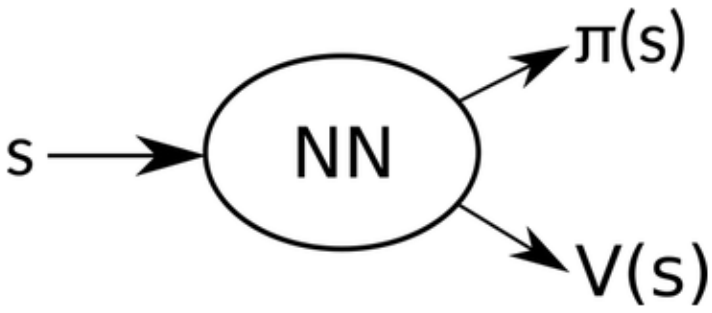


Figure 2.16 *A3C Neural network structure*

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Figure 2.17 *A3C Pseudocode*

Source: from [50]

2.4 Summary

In this chapter, we discussed the background about Reinforcement learning and Deep reinforcement learning which combines RL with Deep learning. Q-learning and policy gradient methods are presented, as well as the limitations of RL. We also present Convolution neural network and Recurrent neural network which are the popular DNNs for image and text processing. Alongside, the hyper-parameters of DL have been discussed, they may effect our following experiment results and evaluation. Last but not least, we present four up-to-date advance DRL methods: Deep Q Network, A3C, DDPG, PPO.

In the next chapter we will discuss the engineering applications of reinforcement learning and the challenges to implement. Also, we will propose the current challenges of DRL for engineering applications.

3. BACKGROUND: DEEP REINFORCEMENT LEARNING FOR ENGINEERING APPLICATIONS

In this chapter, we present an overview of Deep reinforcement learning for engineering applications. First of all, we discuss the necessities and requirements to use RL in engineering applications. Then we list current implemented applications using RL. We also discuss the knotty issues to implement RL in engineering applications. Since deep reinforcement learning is still under developing, we propose the challenges in DRL for engineering applications.

3.1 Reinforcement Learning for Engineering Applications

3.1.1 Engineering Applications

The word 'Engineering' has existed since the start of human civilization. The American Engineers' Council for Professional Development (ECPD, the predecessor of ABET) has defined "Engineering" as: The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property[81]. For engineering applications engineers apply mathematics and sciences such as physics to find novel solutions to problems or to improve existing solutions. More than that, engineers are now required to be proficient in the knowledge of relevant sciences for their design projects. As a result, engineers continue to learn new material throughout their careers. In the past, humans devised inventions such as the wedge, lever, wheel and pulley, which represented the start of engineering applications. As humans discovered more and more about mathematics and sciences, we are about to arrive the new century: industry 4.0, which aiming to build the smart industry.

Categories for Engineering Applications

Engineering is a broad discipline which is often broken down into several sub-disciplines. No doubt that, we are surrounded by miscellaneous engineering applications. Nowadays, we can't live without engineering. Engineering is often characterized as having four main branches: chemical engineering, civil engineering, electrical engineering, and mechanical engineering[81]. There are various applications in each branch, and humans discovered and applied more and more applications during engineering fields fusion. According to the classification in Wikipedia, there are several applications in those branches.

- **chemical engineering**

  Oil refinery, microfabrication, fermentation, and biomolecule production

- **civil engineering**

  structural engineering, environmental engineering, and surveying

- **electrical engineering**

  optoelectronic devices, computer systems, telecommunications, instrumentation, controls, and electronics

- **mechanical engineering**

  kinematic chains, vacuum technology, vibration isolation equipment, manufacturing, and mechatronics

Applications of Artificial Intelligence in Engineering

As time goes, industry continues to revolution, we would like to make the applications more autonomous and human-aware, Artificial intelligence was proposed alongside Industry 4.0. Artificial intelligence is a branch of computer science that aims to create intelligent machines. It has become an essential part of the technology industry. AI techniques are now being used by the practicing engineer to solve a whole range of hitherto intractable problems. More and more AI methods are applied in all branches of engineering. AI technologies already pervade our lives. As they become a central force in society, the field is shifting from simply building systems that are intelligent to building intelligent systems that are human-aware and trustworthy[29]. Knowledge engineering and Machine learning are the cores of AI. They perfectly complement each other, also each performs its own functions. The

role of AI in engineering applications is to improve existing engineering solutions by switching from manual to autonomous. In industrial, engineers usually apply AI in monitor, maintain, optimize, automate, these four areas[24].

In [69], several AI engineering applications were proposed and related papers were published in the following categories:

- Engineering Design
- Engineering Analysis and Simulation
- Planning and Scheduling
- Monitoring and Control
- Diagnosis, Safety and Reliability
- Robotics
- Knowledge Elicitation and Representation
- Theory and Methods for System Development

3.1.2 What is the Role of RL in Engineering Applications?

Reinforcement learning is a framework that shifts the focus of machine learning from pattern recognition to experience-driven sequential decision-making. It promises to carry AI applications forward toward taking actions in the real world. While largely confined to academia over the past several decades, it is now seeing some practical, real-world successes[29].

In engineering, pattern recognition refers to the automatic discovery of regularities in data for decision-making, prediction or data mining. The goal of machine learning is to develop efficient pattern recognition methods that are able to scale well with the size of the problem domain and of the data sets[43]. As Reinforcement Learning is a subfield of Machine Learning, it shares the same goal, particularly it refers to the problem of inferring optimal actions based on rewards or punishments received as a result of previous actions. This is called a reinforcement learning model. Applications which meet the reinforcement learning model can be generated and solved by RL. Reinforcement learning plays a distinctive role in engineering applications.

The goal of RL in engineering applications is about to build human-aware intelligent systems which can be applied to real-world engineering situations, in order to be

more convenient and intelligent. More precisely is to discover an optimal policy that maps states (or observations) to actions so as to maximize the expected return J, which corresponds to the cumulative expected reward[36].

As [24] summarized, Reinforcement Learning can be applied in three aspects of engineering: optimization, control, monitor and maintenance. There are several applications of them, which we list in Figure 3.1.

One reason for the popularity of reinforcement learning is that it serves as a theoretical tool for studying the principles of agents learning to act. But it is unsurprising that it has also been used by a number of researchers as a practical computational tool for constructing autonomous systems that improve themselves with experience. These applications have ranged from robotics, to industrial manufacturing, to combinatorial search problems such as computer game playing[31]. RL algorithms can provide solutions to very large-scale optimal control problems. It has achieved many successful applications in engineering[48, 14, 85, 47, 80, 33]. I searched Google with key words: reinforcement learning applications.

In following table, there are up-to-now engineering applications using RL. We would like to demonstrate three application examples which were done by [47, 80, 33], respectively in Optimize, Control, Monitor and Maintenance domains.

Building an optimize the production inventory system

A Production Inventory Task was done by using a new model-free average-reward algorithm(SMART), which is a improved Reinforcement learning method, to optimize the preventive maintenance in a discrete part production inventory system.

Building HVAC control system

This paper presents a deep reinforcement learning based data-driven approach to control building HVAC(heating, ventilation, and air conditioning) systems. A co-simulation framework based on EnergyPlus is developed for offline training and validation of the DRL-based approach. Experiments with detailed EnergyPlus models and real weather and pricing data demonstrate that the DRL-based algorithms (including the regular DRL algorithm and a heuristic adaptation for efficient multi-zone control) are able to significantly reduce energy cost while maintaining the room temperature within desired range.

Building an intelligent plant monitoring and predictive maintenance system

People came out an idea of learning the fault patterns and fault sequences by trial and error to reformulate the fault prediction problem, so that to make an intelligent plant monitoring and predictive maintenance system. Reinforcement learning methods appear to be a viable way to solve this kind of problem.

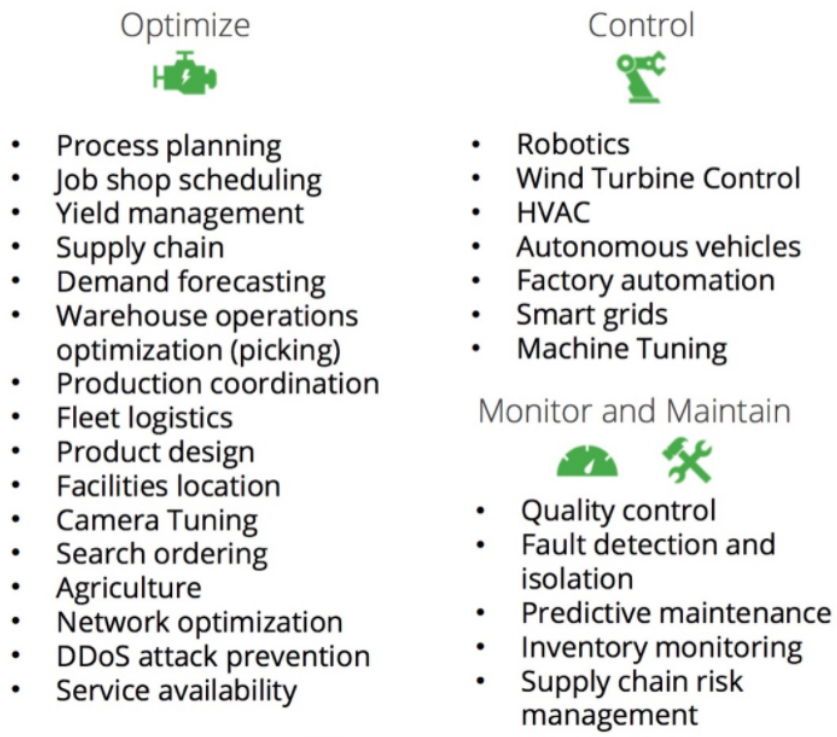


Figure 3.1 *RL in engineering applications*

3.1.3 Characteristics of RL in Engineering Applications

In order to use Reinforcement Learning methods to Engineering Applications, a specific application must be generalized as an RL model. Necessarily, the design of an RL model must meet the Markov assumption, which means that the probability of next state depends only on the current state and action, but not on the preceding states and actions. Basic reinforcement is modeled as a Markov decision process. Basically, according to the documentation of OpenAI Gym, an basic RL model includes: Description, Observation, Actions, Rewards, Starting State, Episode Termination, Solved Requirements.

- Description: simply describe the model, focus on the aim.

- Observation: also be called as State, generalizing and defining required information as input.
- Actions: generalize the possible actions and related limits as output.
- Rewards: the value of a state transition, reinforcement signal.
- Starting State: give a start state, so the agent can start an episode.
- Episode Termination: define some conditions to end the iterations.
- Solved Requirements: define conditions as problem be solved.

Domains	RL Engineering applications
Optimize	Reinforcement Learning Approaches to Biological Manufacturing Systems[77], Distributed reinforcement learning control for batch sequencing and sizing in just-in-time manufacturing systems[28], Application of reinforcement learning for agent-based production scheduling[79], Dynamic job-shop scheduling using reinforcement learning agents[4], Self-improving factory simulation using continuous-time average-reward reinforcement learning[47], Inventory management in supply chains: a reinforcement learning approach[22], The application of Reinforcement learning in Optimization of Planting Strategy for Large Scale Crop production[56], Personalized Attention-Aware Exposure Control using Reinforcement Learning[83]; etc.
Control	A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning[8], Reinforcement learning for humanoid robotics[62], Autonomous inverted helicopter flight via reinforcement learning[53], Reinforcement learning in multi-robot domain[48], Strategy learning for autonomous agents in smart grid markets[68], Using smart devices for system-level management and control in the smart grid: A reinforcement learning framework[34], Deep Reinforcement learning for building HAVC control[80], Learning and tuning fuzzy logic controllers through reinforcements[9], Adaptive PID controller based on reinforcement learning for wind turbine control[72], Reinforcement learning applied to linear quadratic regulation[10], Neural networks and reinforcement learning in control of water systems[]; etc.
Monitor and Maintain	Intelligent monitoring and maintenance of power plants[33], A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning[20], Using reinforcement learning for pro-active network fault management[25], Reinforcement learning of Normative monitoring intensities[42]; etc.

Table 3.1 *Reinforcement learning for engineering applications*

3.1.4 RL lifecycle in Engineering Applications

As applying reinforcement learning in engineering, first of all, we need to define a systems development life cycle(SDLC). A systems development life cycle is composed of a number of clearly defined and distinct work phases which are used by systems engineers and systems developers to plan for, design, build, test, and deliver information systems. I searched Google with key words "lifecycle software", it showed the graphs of lifecycles. This RL lifecycle is derived from software development lifecycle.

1. **Gather Knowledge**

   Gather data and knowledge from the problem to help make appropriate modeling assumptions.

2. **Visualise and define RL model**

   Visualise the gathered data and knowledge to items RL needed(states, actions, rewards, transactions, terminations).

3. **Build training environment according to real life systems** according to real world system, build the simulated environment for training.

4. **choose RL method**

   choose appropriate RL methods to implement

5. **perform training**

   perform training process

6. **applying**

   apply the trained structure to the real problem

7. **Evaluate results**

8. **Diagnose issues**

9. **Refine the system**

3.1.5 Challenges in RL for Engineering Applications

Comparing to other optimal strategies like optimal control solving problems, RL has its own advantages. It is convenient to modify model parameters and robust to external disturbances. For example, it is possible to start from a good enough demonstration and gradually refine it. Another example would be the ability to dynamically adapt to changes in the agent itself, such as a robot adapting to hardware changesheating up, mechanical wear, growing body parts, etc[37]. However, we are still facing a bunch of challenges when applying to engineering applications. In [36], it listed several challenges apparent in the robotics setting. These challenges can be also be considered in general engineering applications, but not enough. Reinforcement Learning (RL) constitutes a significant aspect of the Artificial Intelligence field with numerous applications ranging from finance to robotics and a plethora of proposed approaches. Since there has several RL methods be developed and evaluated through video games, people raised critical challenges when applying RL to

Engineering applications. It's not so much related to algorithm implantation, but building RL model and experimental and error cost.

RL algorithms limits and high requirements of policies improvement

Classical reinforcement learning like Q-learning, Sarsa could deal problems with low-dimensional, discrete observations and actions. However, in engineering applications, the states and actions are inherently continuous, the dimensionality of both states and actions can be high. Facing such problems, only use classical RL algorithms are not enough to solve them. The reinforcement learning community has a long history of dealing with dimensionality using computational abstractions. It offers a larger set of applicable tools ranging from adaptive discretizations and function approximation approaches to macro-actions or options[36]. As the new development of Deep reinforcement learning, which combine deep learning methods with reinforcement learning algorithms, RL in engineering applications make a breakthrough over previously intractable problems. deep learning enables RL to scale to decision-making problems, for example, settings with high-dimensional state and action spaces.

Physical world uncertainty

As we are moving towards Artificial General Intelligence(AGI), designing a system which can solve multiple tasks (i.e Classifying an image, playing a game ..) is really challenging. The current scope of machine learning techniques, be it supervised or unsupervised learning are good at dealing with one task at any instant. This limits the scope of AI to achieve the generality . To achieve AGI, the goal of RL to make the agent perform many different type of tasks, rather than specializing in just one. This can be achieved by multi task learning and remembering the learning. Weve seen recent work of Google Deep Mind on multi task learning, where the agent learns to recognize a digit and playing Atari. However this is really a very challenging task when you scale the process. It requires a lot of training time and huge number of iterations to learn tasks.

Also, in many real world tasks agents do not have the scope to observe the complete environment. This partial observations make the agent to take the best action not just from current observation, also from the past observations. So remembering the past states and taking the best action w.r.t current observation is key for RL to succeed in solving real world problems.

Further more, the system dynamics or parameters may be unknown and subject to noise, so the controller must be robust and able to deal with uncertainty[19]. For such reasons, real-world samples are expensive in terms of time, labor and, potentially, finances.

Simulation environment

In the context of applications, reinforcement learning offers a framework for the design of sophisticated and hard-to-engineer behaviors. The challenge is to build a simple environment where this machine learning techniques can be validated, and later applied in a real scenario[84]. Think about if we training the RL agent in real-world systems, first of all, the stakes are high. Harel Kodesh (former VP and CTO of GE Software) in Forbes said[24]:"If an analytical system on a plane determines an engine is faulty, specialist technicians and engineers must be dispatched to remove and repair the faulty part. Simultaneously, a loaner engine must be provided so the airline can keep up flight operations. The entire deal can easily surpass 200,000 dollars." Second, RL is the process of learning from trial-and-error by exploring the environment and the robots own body. Testing on operating production lines, industrial equipment, warehouses, etcetera is both expensive and disruptive, more then that, they are not plentiful, we can't perform in real world to assume part or product failures. Third, the cost of failure and change is much expensive.

Another reason to use simulation environment for RL training is Safe Exploration Strategies[19]. RL agents always learn from exploration and exploitation. RL is a continuous trial-and-error based learning, where agent tries to apply different combination of actions on a state to find the highest cumulative reward. The exploration becomes nearly impossible in real world. Let us consider an example where you want to make the robot learn to navigate in complex environment avoiding collisions. As the robot moves around the environment to learn, itll explore new states and takes different actions to navigate. However it is not feasible to take best actions in real world where the dynamics of the environment changes very frequently and becomes very expensive for the robot to learn[36].

The current trend is to study (simulated) 3D environments. For example, Microsofts Project Malmo has made the world of Minecraft available to researchers, and DeepMind has open-sourced their own in-house developed 3D environment. These 3D environments focus RL research on challenges such as multi-task learning, learning to remember and safe and effective exploration. There are also many extend packages and softwares based on Open AI gym which can simulate various applications,

such as 'gym-gazebo'. We also use this package to simulate robot in Gazebo and train RL algorithms on Open AI gym.

Once we use simulated environment, one critical problem we must consider, under-modeling and model uncertainty[36]. Ideally, simulation would render environments and RL methods possible to learn the behavior and subsequently transfer it to the real-world applications. Unfortunately, creating a sufficiently accurate environment with actual hyperparameters and get sufficient training data is challenging.

Reward function design

Reward function is the core in reinforcement learning, as it specifies and guides the desired behavior. The goal of reinforcement learning algorithms is to maximize the accumulated long-term reward. An appropriate reward function can accelerate the learning process. It is not easy to define the suitable reward function for correspond application. First question is, how to give a quantitative number of an reward or a punishment. Then is how to define the reward function. The learner must observe variance in the reward signal in order to be able to improve a policy: if the same return is always received, there is no way to determine which policy is better or closer to the optimum[36]. [49]proposed a methodology for designing reward functions that take advantage of implicit domain knowledge.

A design of reward function could also effect the fastness of convergence. If the reward is too fast to meet task achievement, the controller may fall into a local optimal. Opposite way, the controller may slow down the learning process, the worst will be never reach the task achievement.

Professional knowledge requirements

In [24], they proposed that one challenge to breaking the gap between game playing and engineering applications is the reliance on subject matter expertise. Engineers sometimes lack professional knowledge about RL algorithms and programming skills, programmers lack knowledge at specific engineering fields. As the development of AI, various interdisciplinary research programs are raising up. People who has more than one branch of knowledge is desired need.

3.2 Challenges in Deep RL for Engineering Applications

The development of Deep Reinforcement learning solves a part of challenges, such like: traditional RL limits. RL now could deal with continuous state and action space without discretization. Also, Curse of Dimensionality"[36] is no longer a big problem. But there are still some new appeared challenges we need to consider. We wouldve liked to do a survey but it was too much work and we did not have time. Instead, based on some literature review (which is also not exhaustive) we can propose some challenges which give the reader a sense of the motivation for our Thesis and other challenges in the field.

3.2.1 Lack of using cases

In these three years, various Deep RL methods have been proposed and open source implementations are becoming publicly available every day. Google DeepMind and Open AI team published successive four popular DRL algorithms, Deep Q network, A3C, PPO, and DDPG, researchers apply those mostly on the game environments or classical environments which provided by open AI Gym, like CartPole, Pendulum. As this research is up to date and is continue under research. The cases about applying DRL in engineering or related literatures are not plentiful.

3.2.2 The need of comparisons

Researchers in the field face the practical challenge of determining which methods are applicable to their use cases, and what specific design and runtime characteristics of the methods demand consideration. In this case , we will discuss the criteria in this paper.

3.2.3 Lack of standard benchmarks

To date there is no standard benchmark for Deep RL methods. OpenAI Gym has emerged recently as a standardization effort, but it is still evolving. We hope our research could help build the benchmarks.

3.3 Summary

In this chapter, we discussed several aspects of DRL in engineering applications. We started with the theory of engineering applications, then presenting the role, characteristics and lifecycle of RL applying to engineering cases. After that, we discussed

considered issues when applying RL in engineering applications. As some technical issues can be solved by DRL, and DRL is still under developing, we proposed three challenged points in engineering applications.

In next chapter, we propose our research questions in first, then discuss the implementation requirements. The implementation requirements include: environments and experimental setting.

4. PROTOTYPICAL IMPLEMENTATION AND RESEARCH QUESTIONS

In this chapter, our research questions will be proposed and the prototypical implementation will be discussed. In detail, we will propose the research questions, then, we will present the case study about OpenAI Gym and Gazebo. Then we will discuss three different characteristically environments which we use for experiments. At last, we will give out experimental settings.

4.1 Research Questions

The following are the research questions that we have selected for our Thesis:

1. Which hyper-parameters will impact the performance of a specific method over a specific environment, and how?
2. With adjusted hyper-parameters for high performance guaranteed, what are the factors to benchmark different methods over a specific environment?

ITEMS FOR FIRST QUESTION:

- Network architecture
- Model configuration
- Reward function model
- Exploration strategy
- Training configuration for methods relatively

ITEMS FOR SECOND QUESTION:

- running time

- sample efficiency
- highest score
- average score
- robust

4.2 Case Study

OpenAI Gym
OpenAI was founded in late 2015 as a non-profit with a mission to build safe artificial general intelligence (AGI) and ensure AGI's benefits are as widely and evenly distributed as possible. In addition to exploring many issues regarding AGI, one major contribution that OpenAI made to the machine learning world was developing both the Gym and Universe software platforms.

There exists several toolkit for RL environments design, OpenAI Gym leads the place because of its standardization of environments used in publications and the better benchmarks. Gym is a collection of environments designed for testing and developing reinforcement learning algorithms. It is a toolkit for RL. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms[11]. It saves the user from having to create complicated environments. Gym is written in Python, and there are multiple environments such as robot simulations or Atari games. OpenAI Gym also provides an online scoreboard for people to compare results and code locating on the official website(`gym.openai.com`). A diverse collection of tasks are called environments, Gym contains these environments with a common interface, the collection will grow over time step. The gym prepackages with plentiful environments. It's this common API around many environments that makes the gym so great. There are also many environments that doesn't prepackaged in Gym, for example, 3D model environments, robotics using ROS and Gazebo, etc. Therefore, some gym extension packages and libraries are generated like Parallel Game Engine, gym-gazebo, gym-maze to combine Gym with other simulation and rending tools. People can apply Gym to different applications using these extension packages.

OpenAI Gym does not include an agent class or specify what interface the agent should use, it just includes an agent for demonstration purposes[11]. Precisely, Gym only provide the environments. In figure 4.1, the function with a single episode with

100 timesteps is showed in the gym code snippet. All the documentation you can find on Gym Github(`https://github.com/openai/gym`).

```
ob0 = env.reset() # sample environment state, return first observation
a0 = agent.act(ob0) # agent chooses first action
ob1, rew0, done0, info0 = env.step(a0) # environment returns observation,
# reward, and boolean flag indicating if the episode is complete.
a1 = agent.act(ob1)
ob2, rew1, done1, info1 = env.step(a1)
...
a99 = agent.act(o99)
ob100, rew99, done99, info2 = env.step(a99)
# done99 == True  =>  terminal
```

Figure 4.1 *Gym code snippet*

Source: from [11]

Gym-Gazebo

For sophisticated engineering applications like robotics, reinforcement learning methods are hard to apply on reality due to the high error cost. Therefore, emulating the sophisticated behaviors virtual and later applied in a real scenario becomes the alternative solution. Gazebo simulator(`http://gazebosim.org/`) is a 3D modeling and rendering tool, with ROS (Robot Operating System: `http://www.ros.org/`)[67], a set of libraries and tools that help software developers create robot applications. Gazebo is an advanced robotics simulators being developed which help saving costs, reducing time and speeding up the simulation.

A whitepaper[84] presented an extension of the OpenAI Gym for robotics using the Robot Operating System (ROS) and the Gazebo simulator. Abstract information of robotics can be obtained from the real world in order to create an accurate simulated environment. Once the training is done, just the resulting policy is transferred to the real robot. Environments developed in OpenAI Gym interact with the Robot Operating System, which is the connection between the Gym itself and Gazebo simulator(Gigure 4.2). *Erle Robotics* provides a toolkit called Gym-Gazebo, currently they created a collection of six environments for three robots: Turtlebot, Erle-Rover and Erle-Copter. The environments are created in OpenAI Gym style and displayed in Gazebo. It is also flexible that people can create their own robots and environments simulation regarding to this extension package.

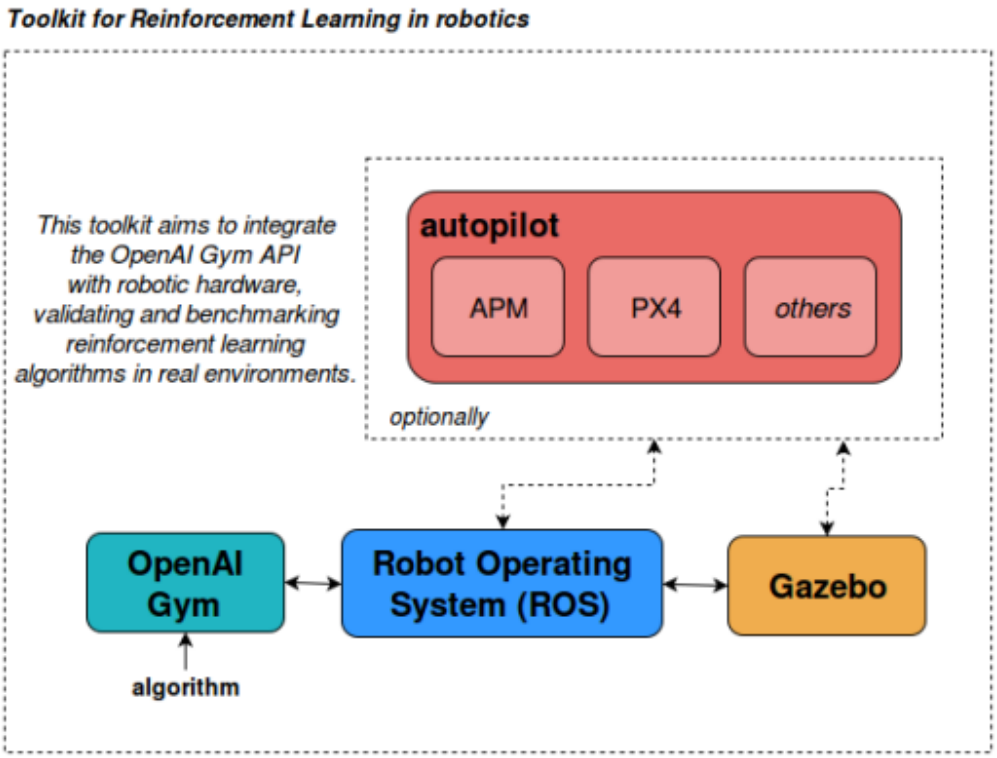


Figure 4.2 *Toolkit for RL in robotics*

Source: from [84]

TensorFlow and Keras

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Googles AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains. You can also find the documentation and tutorial on their Github.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Teano. Keras as the Python Deep Learning library, it offers a simplified way to build your models. Keras contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier. The code is hosted on Github, and community support forums include the GitHub issues page. In this paper, we use Keras to build our agents' models. Alongside, we also used existed RL agent frameworks to

support our algorithms.

- **Keras-RL**
  Keras-RL[64] implements some state-of-the art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras. In this paper, we use their DQN and DDPG agents as references.

- **Morvan-python**
  Morvan-python[64] is a personal and non-profit python, reinforcement learning tutorial supported by a Ph.D student Morvan. It implements some state-of-the art deep reinforcement learning algorithms in Python with Tensorflow. In this paper, we use his DPPO agents as references.

- **rllab**
  rllab is a framework for developing and evaluating reinforcement learning algorithms. Both the benchmark and reference implementations of [17] are based on rllab. It includes a wide range of continuous control tasks plus implementations of the algorithms: REINFORCE, Truncated Natural Policy Gradient, Reward-Weighted Regression, Relative Entropy Policy Search, Trust Region Policy Optimization, Cross Entropy Method, Covariance Matrix Adaption Evolution Strategy, Deep Deterministic Policy Gradient. In this paper, we will use it as a result reference.

4.3 Environments

As we discussed in Chapter 3, due to the high cost of data collection and error, also the dangerous performance, environments simulation is essential for RL in engineering applications. The environments consist of certain factors that determine the impact on the Reinforcement Learning agent. It will be more efficient to select an appropriate method for the agent to interact with an specific environment. These environments can be 2D worlds or grids or even a 3D world. Here are some important features of environments which can effect the choose of appropriate RL methods, described in [52]:

- Deterministic
- Observable
- Discrete or continuous

- Single or multiagent

The Tasks in [17] in the presented benchmark can be divided into four categories: basic tasks, locomotion tasks, partially observable tasks, and hierarchical tasks. In this paper, we used following three environments which can be sorted into basic tasks and hierarchical tasks for experimentation and benchmarking suggestions: CartPole, PlaneBall, CirTurtleBot. These three environments are quite representative, they contains low and high-dimension observations, 2D and 3D models. "CartPole" and "PlaneBall" which belong to basic tasks, are the traits which can be extracted from different engineering applications, they are abstracted from the mechanical dynamical systems. "CirTurtleBot" which belongs to hierarchical tasks, described as a TurtleBot finds path through a maze world. DQN works with environment with discrete actions, DDPG and PPO are better with continuous actions. For engineering applications, it doesn't make sense to take the discrete or continuous actions as the comparable factor. Therefore, we made both discrete and continuous versions for each environment.

4.3.1 CartPole

The "cartpole" system[35] is a classic benchmark for nonlinear control.It is one kind of Mechanical dynamical system, *Mechanical dynamical systems* are easily understandable by people and thus illustrative as examples. Although the "Cart-Pole swing-up" system is a simple system, it is generated into a lot of engineering problems. Following are some engineering applications of cart-pole system[58]:

- The altitude control of a booster rocket during takeoff
- Wheel chair like vehicles
- Seg-ways, Ice-skating
- Helicopter and Aero-plane like aeronautic balancing

"CartPole" is a simulated environment that a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over. It is provided by Gym in classical control domain called "*CartPole-v0*"(Figure 4.3).

In the discrete version, it has two discrete actions(push cart to left and right), four-dimension observation space which means the cart position, cart velocity, pole angle

and pole angle velocity. 4.4. The reward is defined as one for every time step taken, including the termination step. All observations are assigned a uniform random value between ±0.05 as the starting state. Considered solved when the average reward is greater than or equal to 200 holding in 10 consecutive trials. Episode will terminated when:

1. Pole Angle is more than ±12°
2. Cart Position is more than 2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than 200

In the continuous version, it has one dimensional action space with limit[-1,1], which means the force with direction applying to the cart. The rests are same to the discrete version.

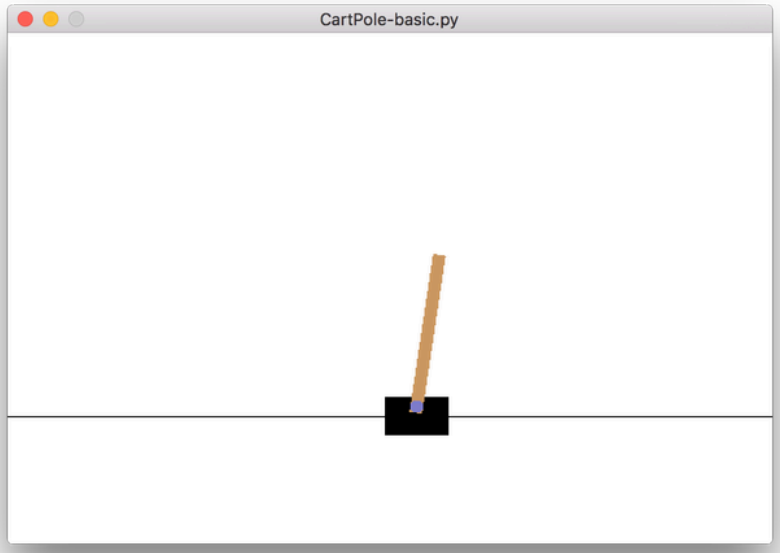


Figure 4.3 *CartPole-v0*

Source: gym.openai.com

Observation

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Actions

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

Figure 4.4 *Observations and Actions in CartPole-v0*

Source: gym.openai.com

4.3.2 PlaneBall

"PlaneBall" is that a ball rolls on a plane, which can rotate in both X and Y axis. The goal is to keep the ball always in the center of the Plane. This system is another mechanical dynamical system. It shares some similarities with CartPole, but also has its own features and more dimensions. I followed the same baseline structure displayed by researchers in the OpenAI Gym and Gym-Gazebo package, and builds a gazebo environment of "PlaneBall" on top of that(Figure 4.5).

In the continuous version, it has two dimensional action space(torque on the plane of X-axis and on Y-axis) with limit[-2, 2], 7-dimension observations including: rotation degree on X-axis α, rotation degree on Y-axis β, rotation speed on X-axis vel_α, rotation speed on Y-axis vel_β, ball position according to plane-frame (x, y), ball velocity vel_{ball}. The reward is defined as: -1 for every time step the ball doesn't move to the middle area, 100 for every time step the ball stays in the middle area, -100 for the ball rolling out of the plane. Random angle α, β in $\pm 90^{circ}$ and random ball position in the limits of the plane-frame for the starting state. Considered solved when the episode reward is greater than or equal to 800 holding in 10 consecutive trials. Episode will terminated when:

1. Ball rolls out of the plane
2. Episode length is greater than 999

In the discrete version, it has 81 discrete actions, we discretize each dimension with limit[-2, 2] to [-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2], then use [0, 1, ..., 81] to replace the combination of two [-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2]. The rests are same to the continuous version.

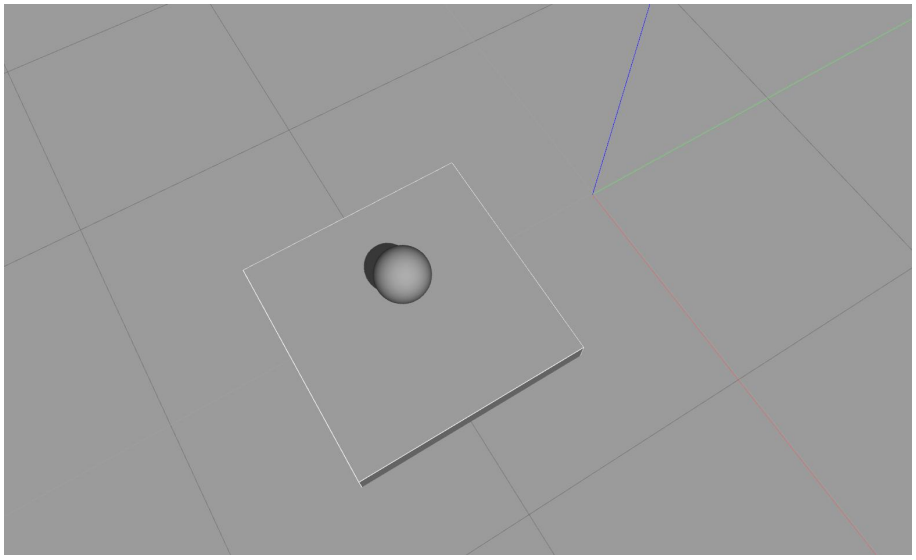


Figure 4.5 *PlaneBall*

4.3.3 CirTurtleBot

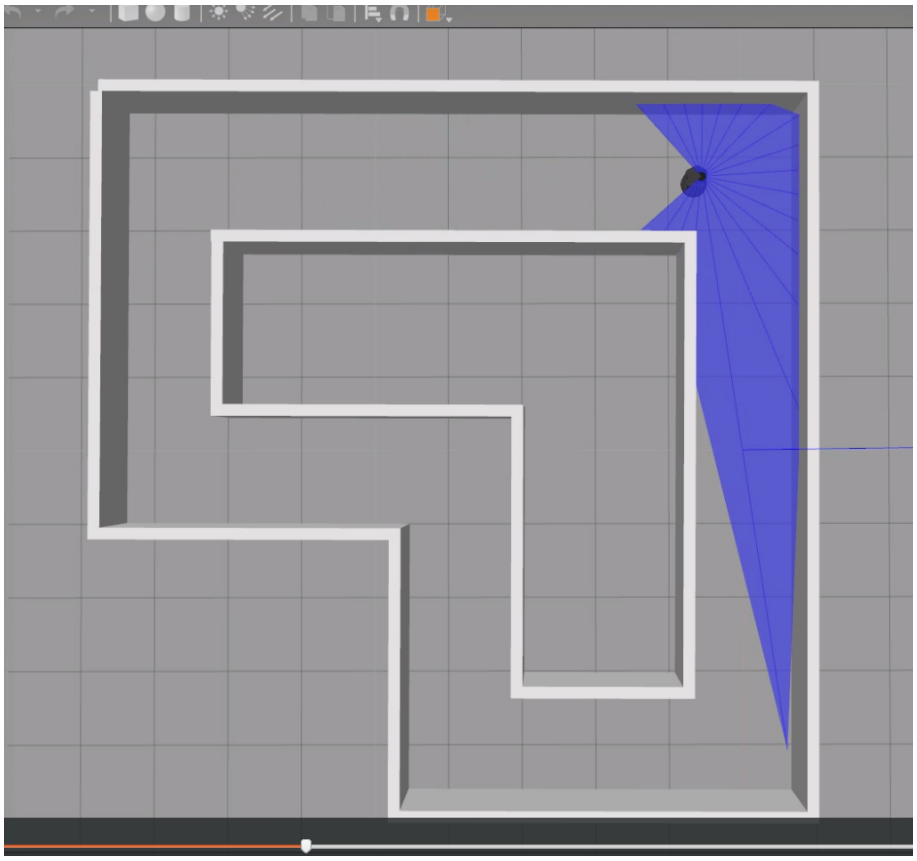
"CirTurtleBot" is a classic robot called Tutlebot(https://www.turtlebot.com/), it moves around a circuit, complex maze with high contrast colors between the floor and the walls. Lidar is used as an input to train the robot for its navigation in the environment. The goal is to navigate robot without collision with walls(Figure**??**). This environment is simulated and controlled by Gazebo, ROS and Gym, it is available in gym-gazebo package called "*GazeboCircuitTurtlebotLIDAR-v0*" (https://github.com/erlero gazebo). In the discrete version, it has three discrete actions(forward, left, right). And five dimensional observation space, which all represent the the range of the scan r in range domain. The reward is defined as one for every time step the robot move left or right, five for forward moving.

In the continuous version, it has one dimensional action space with limit[-0.33, 0.33], which represents the rotate angle acceleration to control the angle velocity(vel_z). As the observations' information are all collected by the Laserscanner on TurtleBot, the

observation space is 20 dimensions, which all represent the range of the scan r in range domain. The reward is defined as a reward function according to the action value. We can understand it as a normal distribution, when it approaches the middle of the action value, the reward will be the highest(5).

Random initializing the robot position at four corners of the maze for the starting state. Considered solved when the episode reward is greater than or equal to 600 holding in 10 consecutive trials. Episode will terminated when:

1. Each laser range is smaller than 0.2
2. Episode length is greater than 300



4.4 Experimental Setting

In this section, we elaborate on the experimental setup used to generate the results.

Hardware settings and specific versions
Our experimental device is a laptop with 8GB RAM, 8 kernel CPU, Ubuntu 16.04

system. The versions of experimental softwares are: Python3.5, Tensorflow 1.2-CPU support, Keras, ROS-Kinetic, Gazebo7. We also installed "Keras-RL", rllab, "RAY" packages for agent utilization.

Training configuration
We run each algorithm five times with different random seeds. Then we average these five to get the result. We train 'CartPole' in 300k steps, 'PlaneBall' in 500k steps, 'CirTurtleBot' in 200k steps.

Reward function model
We use *infinite-horizon discounted model* $\mathbf{E}(\sum_{t=0}^{\infty} \gamma^t r_t)$, $0 < \gamma < 1$ for the performance analysis. In DQN, discounted Q-value(action-value) function is used for Q_{target} and advantage function calculation.

Exploration strategy
For DQN, we utilized 2 main exploration approaches: Epsilon Greedy with annealing policy and Boltzmann Policy. The principle behind these two we discussed in chapter2. We compared the two policies responded to the first research question in the experiment. For DDPG, we utilized a random process function called: OrnsteinUhlenbeck process. For DPPO, we random sample from a defined normal distribution.

Policy Representation
For the function approximation of our evaluated algorithms, we utilized normal Deep Neural Network architectures. Figure 5.1 shows the NN architecture we used in DQN method. There are three hidden layers with 24 neurons and rectified linear unit activation for each. This architecture we also use for both actor-critic networks of DDPG and DPPO with different input and output.

4.5 Summary

In this chapter, we proposed three research questions as references for evaluation in next chapter. We also presented three environments we used during our work. CartPole is the classical environment provided by Gym, PlaneBall is a "medium hard" environment which created by ourself. CirTurtleBot is an environment with a TurtleBot moving in a maze environment, provided by Gazebo, implemented by Gym-Gazebo package. Also, our experimental settings are given.

In next chapter, we discuss our evaluation and experimental results regarding to the research questions in previous chapter.

5. EVALUATION AND RESULTS

In this chapter, we discuss the evaluation and results. Each chapter begins by reminding the reader of the research question. Then we describe the test that we designed to evaluate it. Next we give the results and discuss them. To conclude the chapter we try to give a Takeway paragraph, where saying what the reader can take from your test and results.

5.1 "One-to-One" performance Comparison

In this evaluation section, we focus on the first research question. We tun the factors which might have performance influence for a specific method apply to a specific environment. We evaluate the results in terms of sample complexity, value accuracy, policy quality. We listed all the influenced factors of the three methods, in figure 5.2, 5.14, 5.21. In order to reduce the large amount of comparisons of hyper-parameters tunning, we fixed some parameters which have the better performance according to some related works and tuned some specific parameters. Performance of the implemented algorithms in terms of average return over all training iterations for five different random seeds (same across all algorithms to all environments). we tested and used the smallest fully training steps of each "one-to-one" experiment.

We listed the influenced hyper-parameters in each methods with graph format and the fixed and tuned parameters of each "one-to one" pair with table format. The tuned parameters with red fonts represent the higher performance after comparing.

5.1.1 DQN method in environments

In our evaluation, we utilized advanced DQN method, which is Double DQN[78] with fixed Q target and experience replay mechanisms. There are the influenced hyper-parameters in DQN algorithm, which shows in figure 5.2. We used the Neural network structure as figure 5.1 shows. The input and output layers are different regarding to different environments.

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 4)	0
dense_1 (Dense)	(None, 24)	120
activation_1 (Activation)	(None, 24)	0
dense_2 (Dense)	(None, 24)	600
activation_2 (Activation)	(None, 24)	0
dense_3 (Dense)	(None, 24)	600
activation_3 (Activation)	(None, 24)	0
dense_4 (Dense)	(None, 2)	50
activation_4 (Activation)	(None, 2)	0

Total params: 1,370
Trainable params: 1,370
Non-trainable params: 0

Figure 5.1 *Neural Network architecture of DQN*

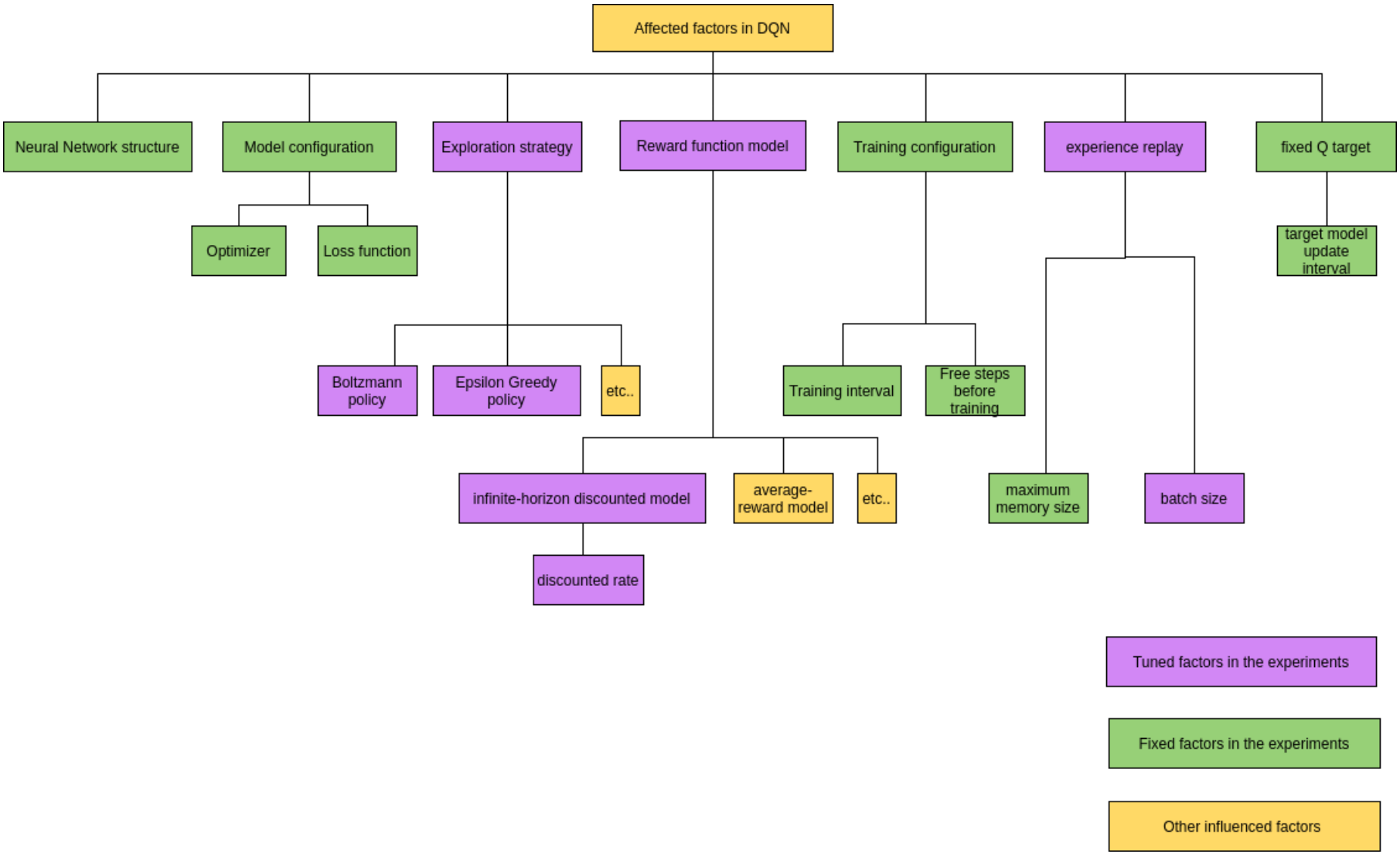


Figure 5.2 *Influenced hyper-parameters in DQN*

- **DQN in "CartPole"**

  In this experiment, we used the suggested Neural Network structure 5.1, there are three hidden layers with 24 neurons and rectified linear unit activation for each. We found this structure is good enough for "CartPole" training responding to training time, performance. For the training compiling, we found the Adam optimizer with learning rate 0.01 can give a better performance. The loss function we used Huber Loss with formula: $0.5 * square(q\_target - q\_predict)$.

  We evaluated the performance with aspects to "steps per episode" and "reward per episode" according to total training steps as well as the training time. Due to the characteristic of "CartPole" environment, these two performance aspects are equal, so we can evaluate only the "reward per episode".

  Table 5.1 listed the fixed parameters of DQN in "CartPole", which we guaranteed that will give a better performance. Table 5.2 listed the parameters that we tuned in this experiment. Here we choose two exploration strategies Epsilon discounted greedy and Boltzmann with fix parameters' values for the comparison. The batch size we choose 32, 64, 96, and the discounted rate we choose 0.1, 0.5, 0.99. In this experiment, we run 2500 training episodes and 500 testing episodes.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.1
Optimizer	Adam optimizer, learning rate=0.001
Loss function	Huber loss function, 0.5*squre(q_target-q_predict)
Q-learning function	$Q_{target} = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta'_t); \theta_t)$
Maximum memory size	10000
Steps before training	2000
Batch size	64
Training interval	train_interval=1
Target model update interval	update factor=0.01

Table 5.1 *Fixed parameters of DQN in "CartPole"*

Tuned parameters	**Comparing Value**	
Exploration strategy	Epsilon discounted greedy policy (eps_min=0.0001, eps_decay=0.999)	Boltzmann policy (tau=1, clip=[-500,500])
Reward function model (infinite-horizon discounted model)	discounted_rate=0.5	discounted_rate=0.99

Table 5.2 *Tuned parameters of DQN in "CartPole"*

As table 5.2 shows, we evaluated the 'exploration strategy' and 'discounted rate' factors. Figure 5.4 shows the performance of two different exploration strategy, DisEpsGreedy, Boltzmann and two discounted rate, 0.5, 0.99. Red line represents Boltzmann policy with tau=1.0, clip=(-500, 500), discounted_rate=0.99. Green line represents Discounted Epsilon Greedy policy with minimum_epsilon=0.0001, decay=0.999, discounted_rate=0.99. Blue line represents Boltzmann policy with tau=1.0, clip=(-500, 500), discounted_rate=0.5.

As we can see, green line coverage and reach to the termination faster than the red line .However, the green line can't always reach the highest reward in the test steps. The blue line holds in a much low reward which means that it didn't coverage. In conclusion, the agent with fixed parameters(table 5.7) and exploration strategy=Boltzmann, discounted_rate=0.99 have a such high performance.

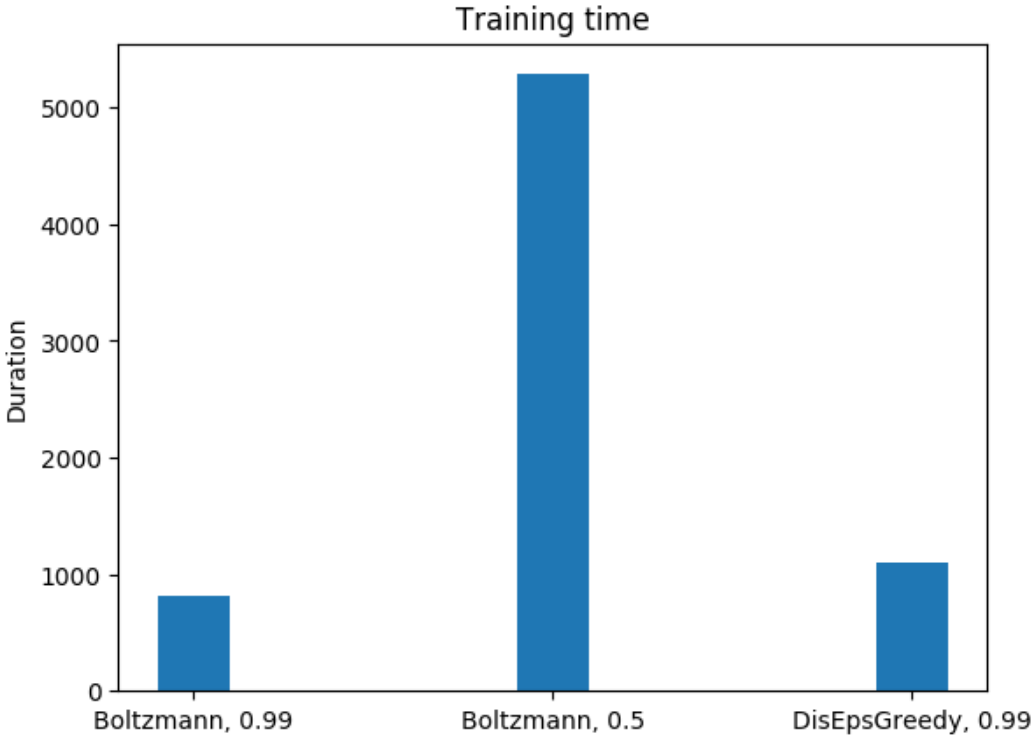


Figure 5.3 *DQN-CartPole: Training time*

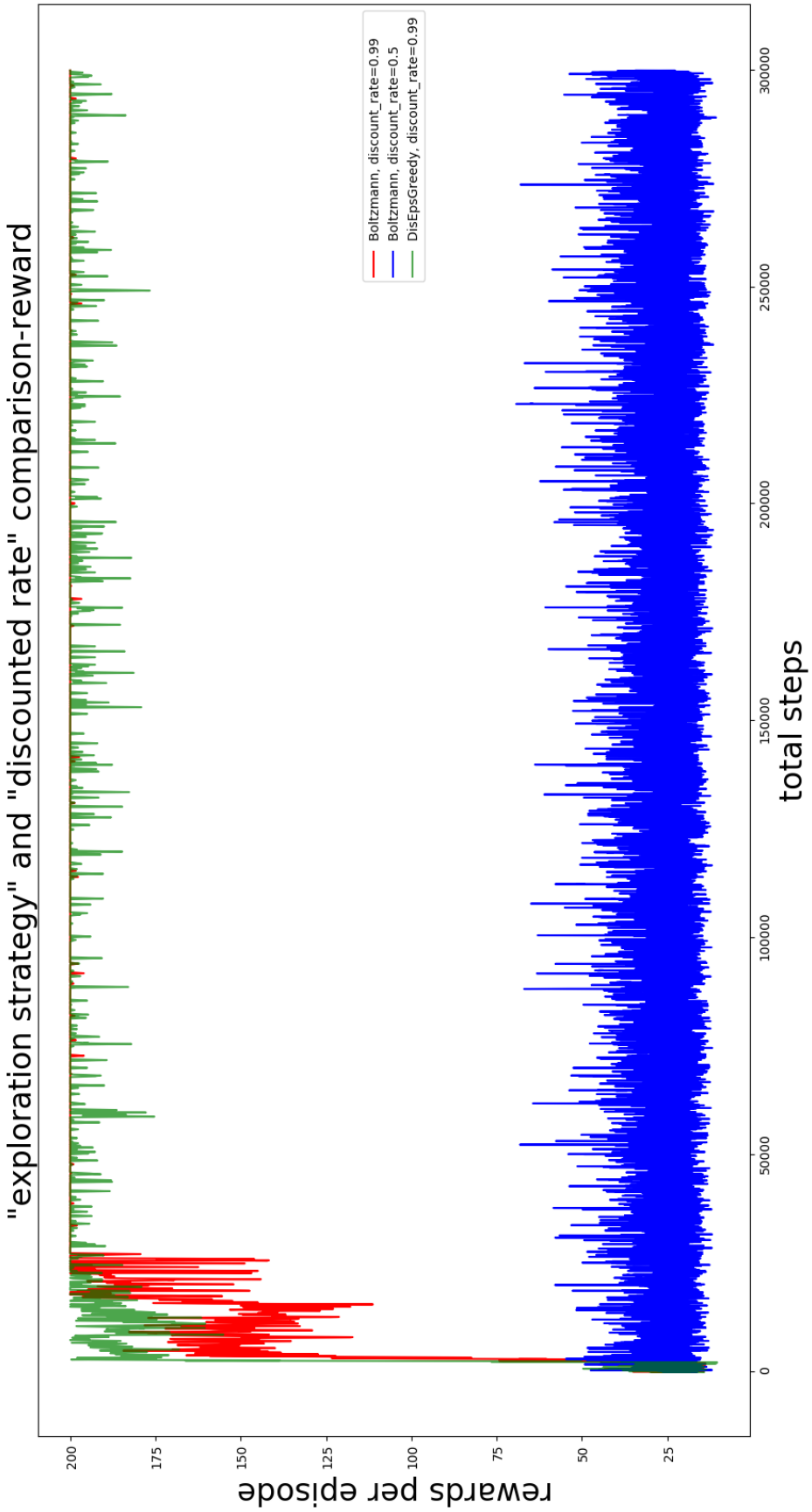


Figure 5.4 *DQN-CartPole: Exploration strategy and discounted rate comparison*

- **DQN in "PlaneBall"**

  In this experiment, we used the suggested Neural Network structure 5.1, there are three hidden layers with 24 neurons and rectified linear unit activation for each. We found this structure is good enough for "PlaneBall" training responding to training time, performance. We perform 2000 episodes for training, after training episodes, we use 500 episodes for testing. We evaluated the performance with aspects to "reward per episode" according to total training steps as well as the training time. As we discussed in chapter4, the goal of "PlaneBall" environment is to hold in the middle area of the plane as long as possible.

  Table 5.3 listed the fixed parameters of DQN in "PlaneBall", which we guaranteed that will give a better performance. Table 5.4 listed the parameters that we tuned in this experiment. Here we use Adam optimizer with the learning rate of 0.001 and 0.01. We also compared the target network update frequency, we utilized soft update with update rate of 0.001 and 0.01, hard update with 50000 steps. In this experiment, we run 2000 training episodes and 500 testing episodes. Due to the hardware limitations, we didn't guarantee the fully training steps. we use the fixed training steps to evaluate algorithms with different hyper-parameters.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.1
Loss function	Huber loss function, 0.5*square(q_target-q_predict)
Q-learning function	$Q_{target} = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta'_t); \theta_t)$
Memory size	100000
Steps before training	20000
Batch size	640
Training interval	train_interval=1
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration strategy	Boltzmann policy, tau=1

Table 5.3 *Fixed parameters of DQN in "PlaneBall"*

Tuned parameters	Comparing Value		
Optimizer (Adam optimizer)	learning_rate=0.001	learning_rate=0.01	
Target model update interval	soft update, rate=0.001	soft update, rate=0.01	hard update, 50000 steps

Table 5.4 *Tuned parameters of DQN in "PlaneBall"*

With fixed learning_rate=0.01, we compared the algorithms with three different target_network update rate: 0.001, 0.01, 50000. For the target network update, we have two mechanisms, soft update with certain update rate and hard update with certain steps. In this experiment, we evaluated soft update with rate 0.001 and 0.01, and hard update with 50000 steps. As figure 5.7 showed, we compared the target_net update interval with 0.001(red line), 0.01(green line), 50000(blue line). In the training steps, we can see that algorithm with target_net update interval=0.1 reached the high score fast than other two. After 2000 episodes training episodes, we use 500 episodes for testing. Algorithm with target_net update interval=50000 presented the worst average reward, algorithm with target_net update interval=0.001, 0.01 both presented quite good average reward. After training steps, algorithm with target_net update interval=0.01 could hold the maximum steps.

With fixed target_net update interval=0.01, we compared the algorithms with two different learning rate: 0.001 and 0.01. As figure 5.8 showed, we compared the learning rate with 0.001(red line), 0.01(green line) according to "reward per episode" factor. In the training steps, we can see that algorithm with learning rate=0.01 reached the high score fast than another. After 2000 episodes training episodes, we use 500 episodes for testing. Algorithm with learning rate=0.01 presented the higher average reward than the algorithm with learning rate=0.001. After training steps, algorithm with learning rate=0.01 could hold the maximum steps.

Figure 5.5, 5.6 showed the training time according to different comparison. Overall, in our experiment of "PlaneBall" environment, the algorithm with fixed hyper-parameter in 5.1 and tuned hyper-parameter of learning_rate=0.01, target_net_update_interval=0.01.

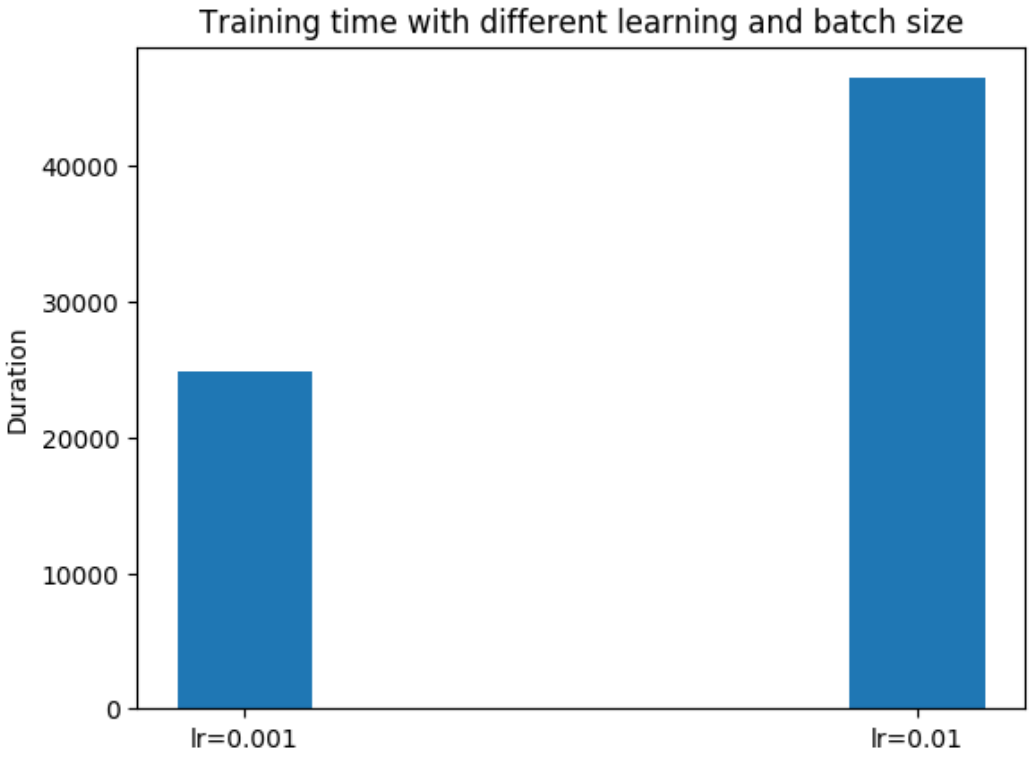


Figure 5.5 *DQN-PlaneBall: Training time of learning rate comparison*

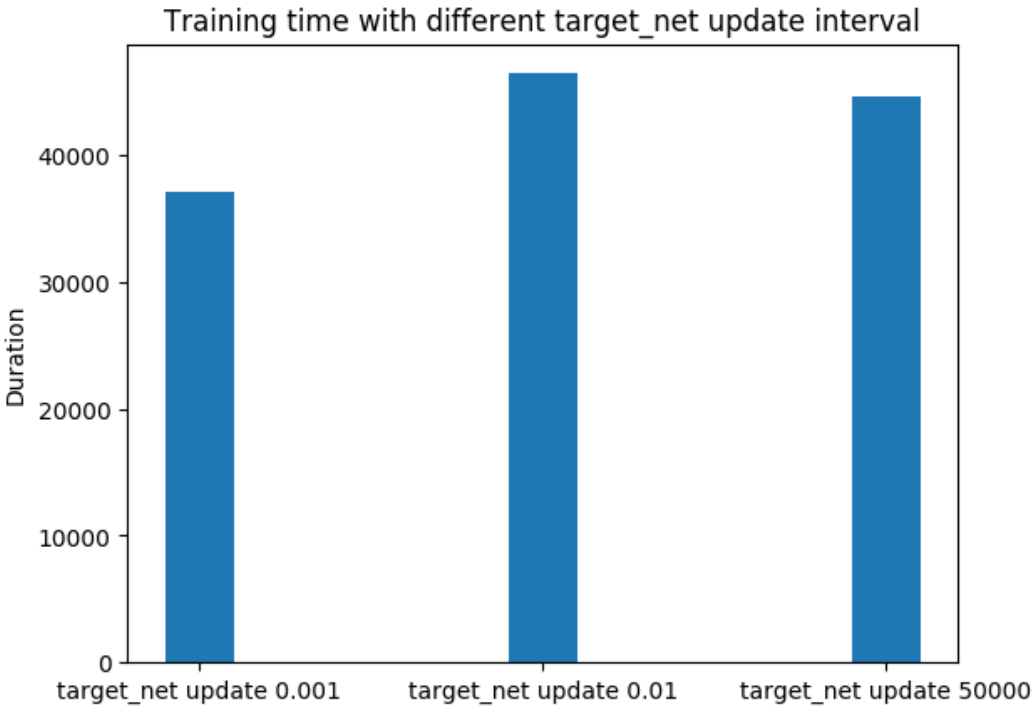


Figure 5.6 *DQN-PlaneBall: Training time of target net update comparison*

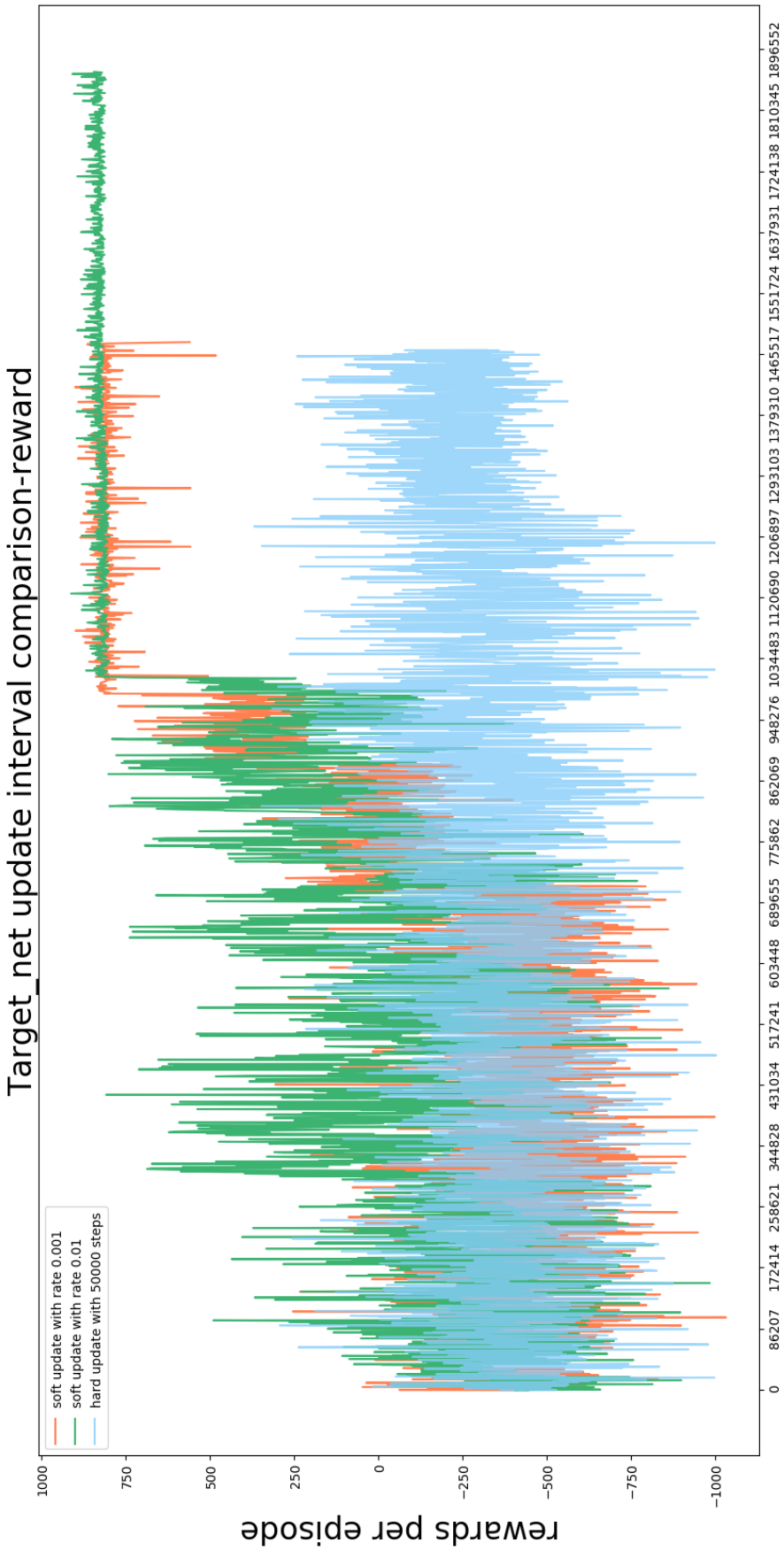


Figure 5.7 *DQN-PlaneBall: target_net update comparison*

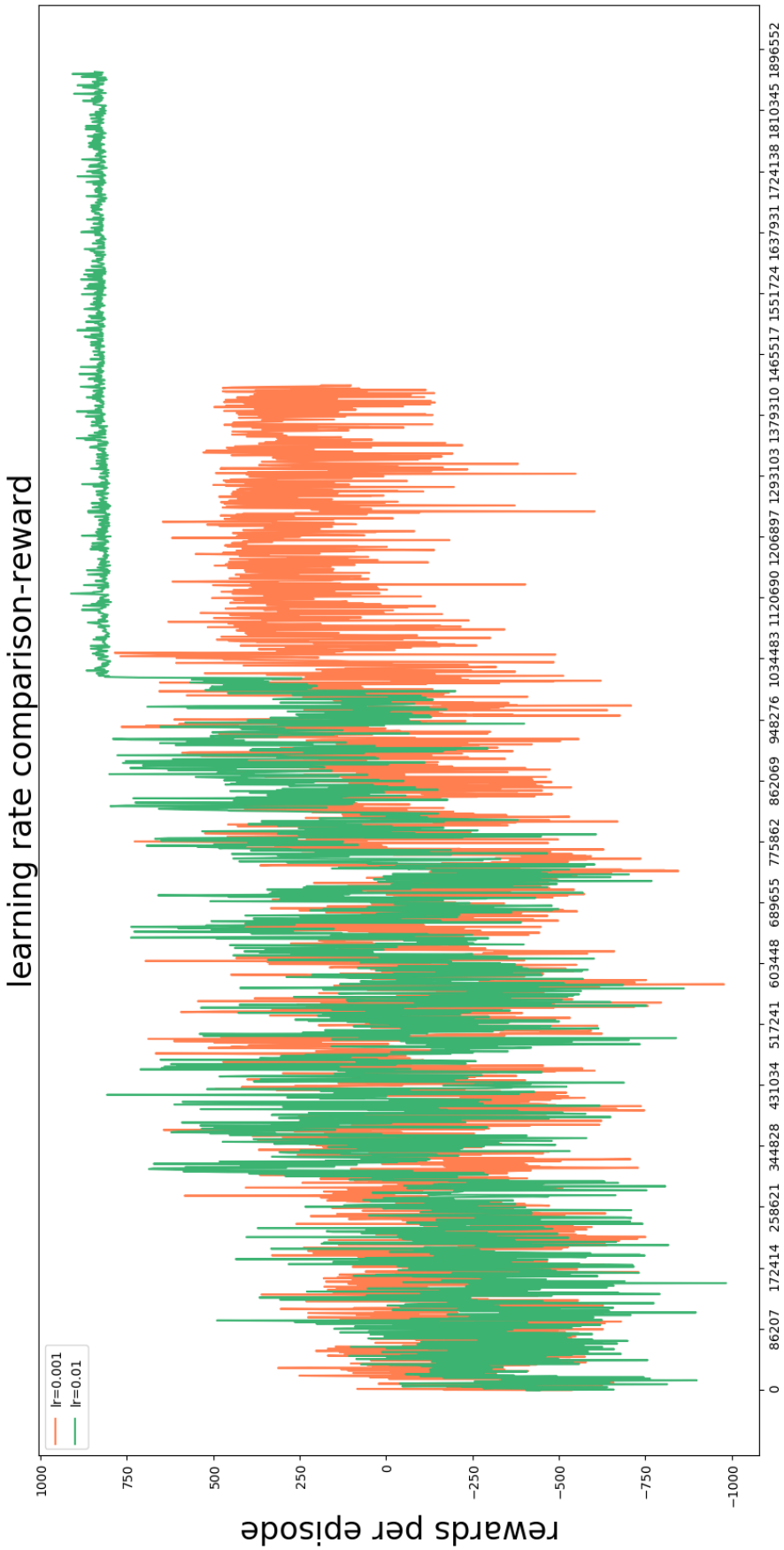


Figure 5.8 *DQN-PlaneBall: learning rate comparison*

- **DQN in "CirTurtleBot"**

  In this experiment, we used the suggested Neural Network structure 5.1, there are three hidden layers with 24 neurons and rectified linear unit activation for each. We found this structure is good enough for "CirTurtleBot" training responding to training time, performance. We used the Q-learning function as table 5.7showed. The loss function we used Huber Loss with formula: $0.5 * squre(q\_target - q\_predict)$. We evaluated the performance with aspects to "steps per episode" and "reward per episode" according to total training steps as well as the training time. Table 5.3 listed the fixed parameters of DQN in "CirTurtleBot", which we guaranteed that will give a better performance. Table 5.4 listed the parameters that we tuned in this experiment. Here we use Adam optimizer with the learning rate of 0.1, 0.01 and 0.001. The batch size we choose 32, 64, 96. We also compared the warm up steps before training, we utilized 1000 and 2000 steps. In this experiment, we run 2500 training episodes and 500 testing episodes. Due to the hardware limitations, we didn't guarantee the fully training steps. we use the fixed training steps to evaluate algorithms with different hyper-parameters.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.1
Loss function	Huber loss function, 0.5*square(q_target-q_predict)
Q-learning function	$Q_{target} = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta'_t); \theta_t)$
Steps before training	1000
Batch size	96
Memory size	50000
Training interval	training_interval=1
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration strategy	Boltzmann policy, tau=0.8

Table 5.5 *Fixed parameters in DQN of "CirTurtleBot"*

Tuned parameters	**Comparing Value**	
Optimizer (Adam optimizer)	learning_rate=0.001	learning_rate=0.01
Target Net update interval	0.001	0.01

Table 5.6 *Tuned parameters in DQN of "CirTurtleBot"*

As table 5.6 shows, we evaluated the 'learning rate' and 'target net update interval' factors. Figure 5.10 shows the performance of two different learning rate: 0.001, 0.01 with fixed target_net_update=0.01. Figure 5.11 shows the performance of two different target net update interval: 0.001, 0.01 with fixed learning_rate=0.01. Because the vibration amplitude of the three curves are really large, we separate the comparison to two figures. The red line represents learning_rate=0.01, target_net_update=0.01; the blue line represents learning_rate=0.001, target_net_update=0.01; the green line represents learning_rate=0.01, target_net_update=0.001.

In figure 5.10, two lines coverage both at around 15000 steps. The red line oscillates between 400 and 850 rewards, the blue line oscillates between 0 and 1200 rewards. Although the the blue line performs higher score than the red one, it is not stabler than the red one. In real engineering applications, we rather like the performance of the red one. In figure 5.11, the red line coverages faster and it is stabler than the green on. Since we can't get higher performance using this method, we would admit that the red one which represents learning_rate=0.01, target_net_update=0.01 has the highest performance.

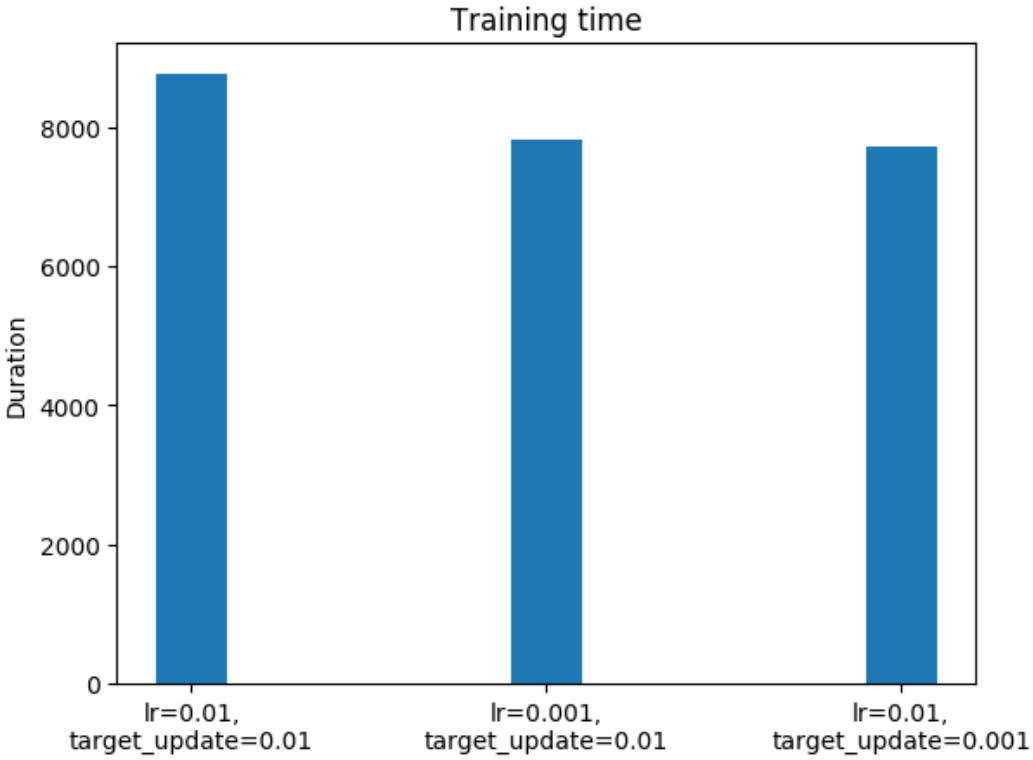


Figure 5.9 *DQN-CirturtleBot: Training time*

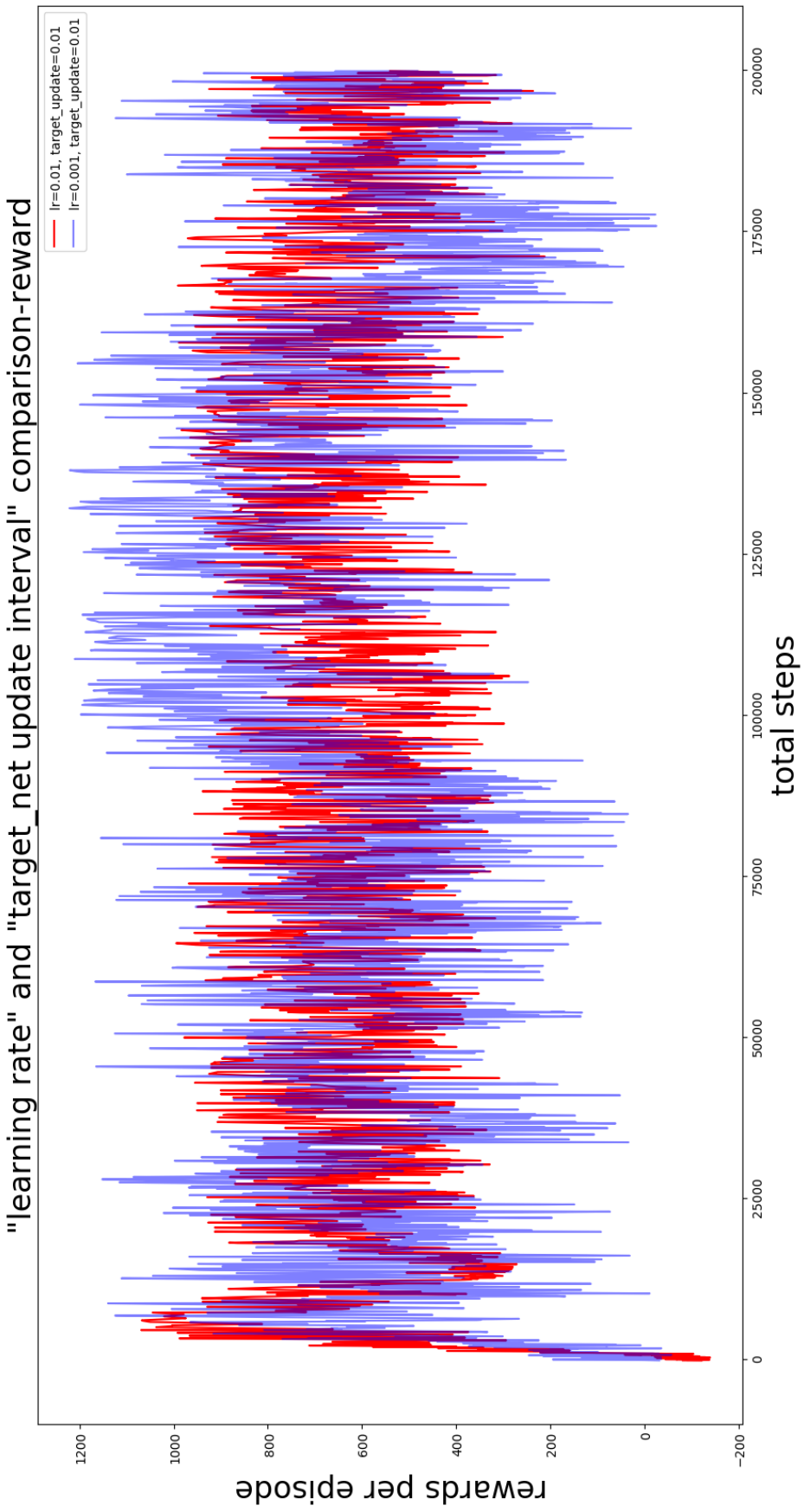


Figure 5.10 *DQN-CirturtleBot: Exploration strategy and discounted rate comparison*

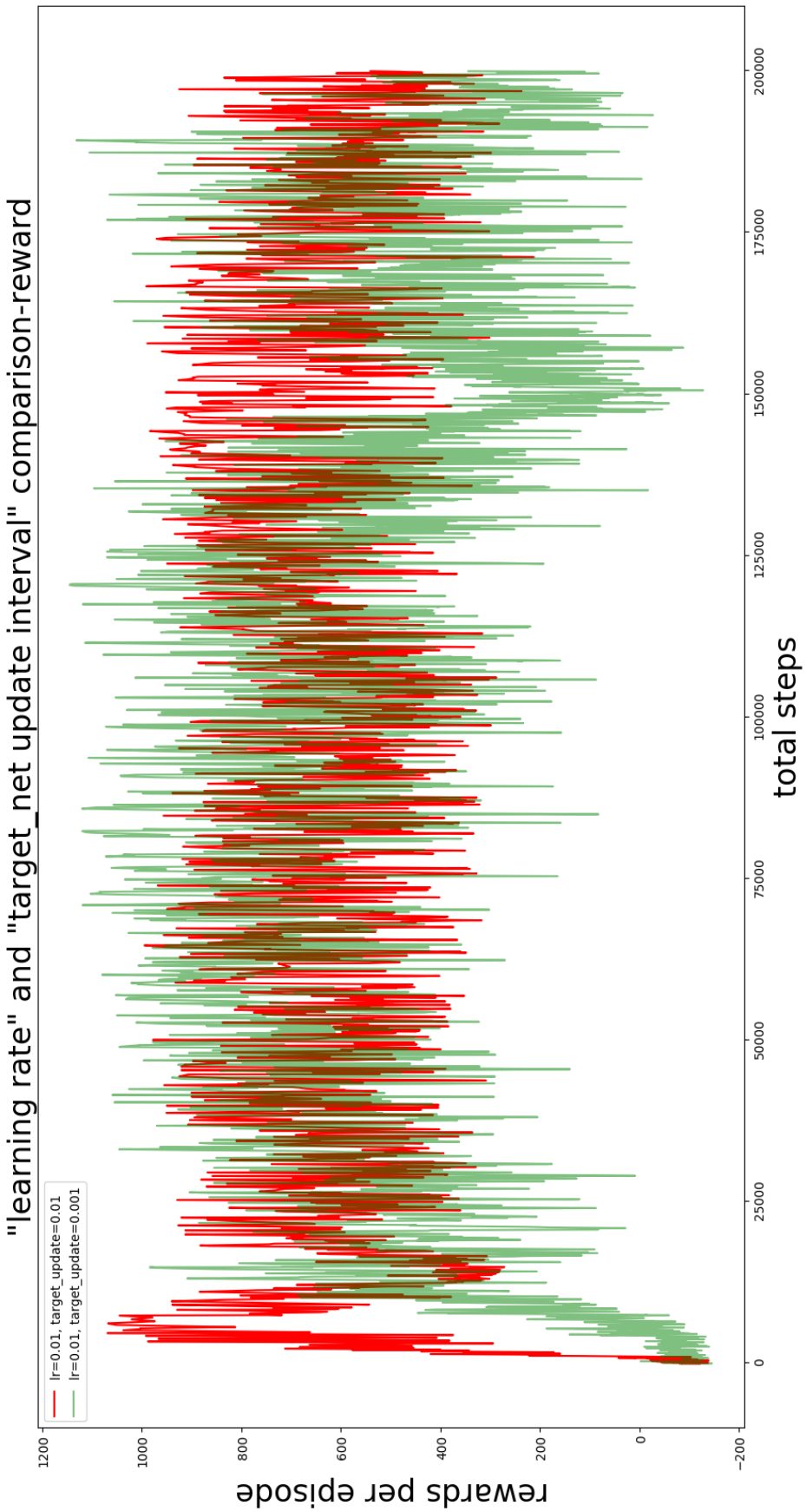


Figure 5.11 *DQN-CirturtleBot: Exploration strategy and discounted rate comparison*

5.1.2 DDPG method in environments

In our evaluation, we evaluate DDPG method for three environment, we tuned the hyperparameters to compare the performance in order to get the high performance relatively. There are the influenced hyper-parameters in DDPG algorithm 5.14. We used the actor-critic neural network architectures as figure 5.12, 5.13 showed. The input and output layers are different regarding to different environments.

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 4)	0
dense_1 (Dense)	(None, 24)	120
activation_1 (Activation)	(None, 24)	0
dense_2 (Dense)	(None, 24)	600
activation_2 (Activation)	(None, 24)	0
dense_3 (Dense)	(None, 24)	600
activation_3 (Activation)	(None, 24)	0
dense_4 (Dense)	(None, 1)	25
activation_4 (Activation)	(None, 1)	0

Total params: 1,345
Trainable params: 1,345
Non-trainable params: 0

Figure 5.12 *Actor Neural Network architecture of DDPG*

Layer (type)	Output Shape	Param #	Connected to
observation_input (InputLayer)	(None, 1, 4)	0	
action_input (InputLayer)	(None, 1)	0	
flatten_2 (Flatten)	(None, 4)	0	observation_input[0][0]
concatenate_1 (Concatenate)	(None, 5)	0	action_input[0][0] flatten_2[0][0]
dense_5 (Dense)	(None, 24)	144	concatenate_1[0][0]
activation_5 (Activation)	(None, 24)	0	dense_5[0][0]
dense_6 (Dense)	(None, 24)	600	activation_5[0][0]
activation_6 (Activation)	(None, 24)	0	dense_6[0][0]
dense_7 (Dense)	(None, 24)	600	activation_6[0][0]
activation_7 (Activation)	(None, 24)	0	dense_7[0][0]
dense_8 (Dense)	(None, 1)	25	activation_7[0][0]
activation_8 (Activation)	(None, 1)	0	dense_8[0][0]

Total params: 1,369
Trainable params: 1,369
Non-trainable params: 0

Figure 5.13 *Critic Neural Network architecture of DDPG*

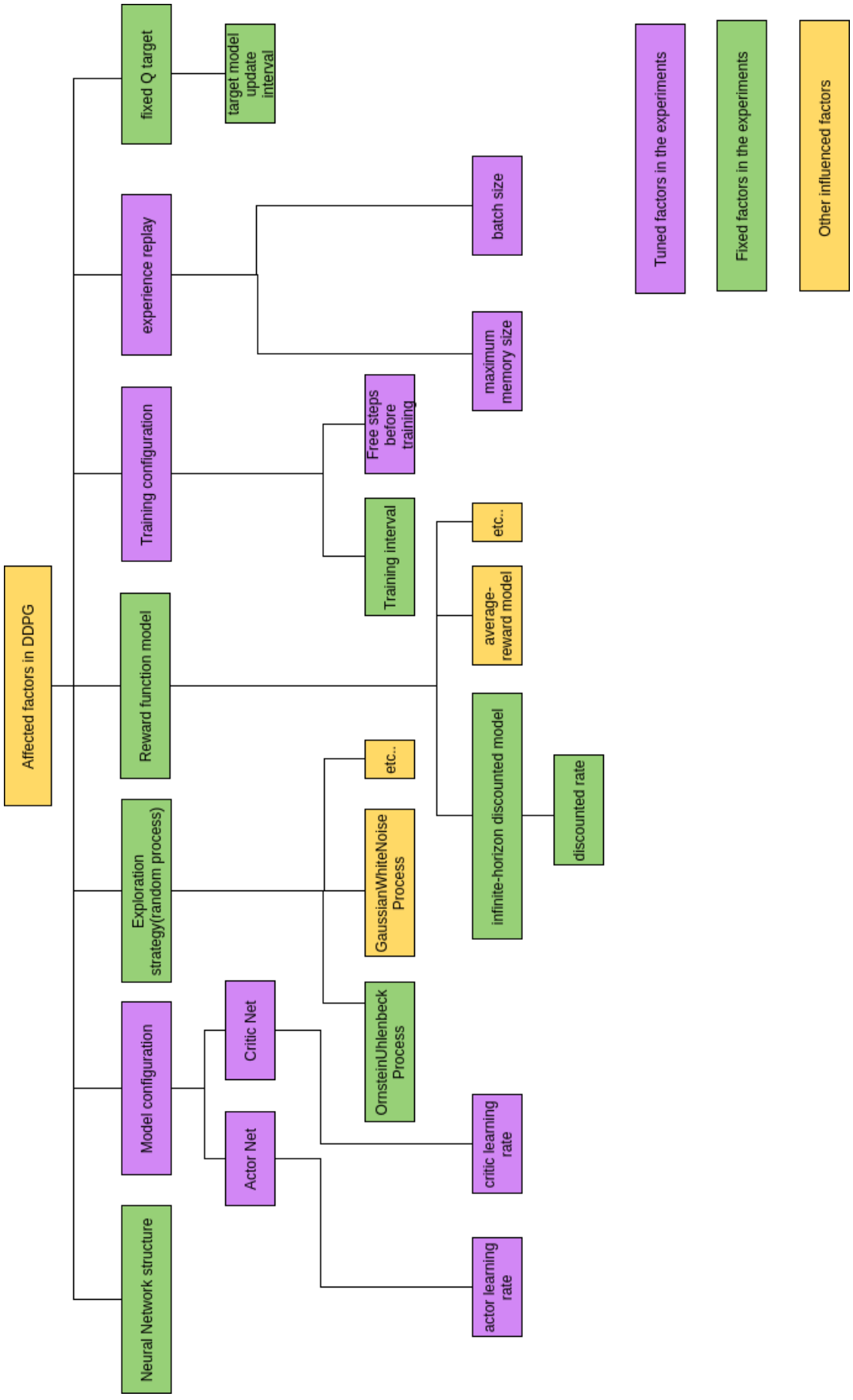


Figure 5.14 *Influenced hyper-parameters in DDPG*

- **DDPG in "CartPole"**

We trained the DDPG agent in "CartPole" in 300000 training steps. When the agent triggers the termination condition, it will stop training and performing the test during the rest steps. In "CartPole", the termination condition is defined as: holding 200 reward in 10 episodes. Table 5.7 and 5.8 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13, with four dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12, 5.13
Critic learning rate	0.001
Steps before training	1000
Batch size	96
Target Net update interval	0.001
Training interval	training_interval=1
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration strategy (Random process)	OrnsteinUhlenbeckProcess

Table 5.7 *Fixed parameters in DDPG of "CartPole"*

Tuned parameters	**Comparing Value**	
Actor learning rate	learning_rate=0.001	learning_rate=0.00001
memory size	memory=50000	memory=10000

Table 5.8 *Tuned parameters in DDPG of "CartPole"*

As table 5.8 shows, we evaluated the 'Actor learning rate' and 'memory size' factors. Figure 5.16 shows the performance of two different memory size, 10000, 50000 and two actor learning rate, 0.00001, 0.001. As we can see, green line coverage and reach the termination faster than the red line. The blue line holds in a much low reward which means that it didn't learn. In conclusion, the agent with fixed parameters(5.7) and memory_size=10000, actor_lr=0.00001 have a such high performance.

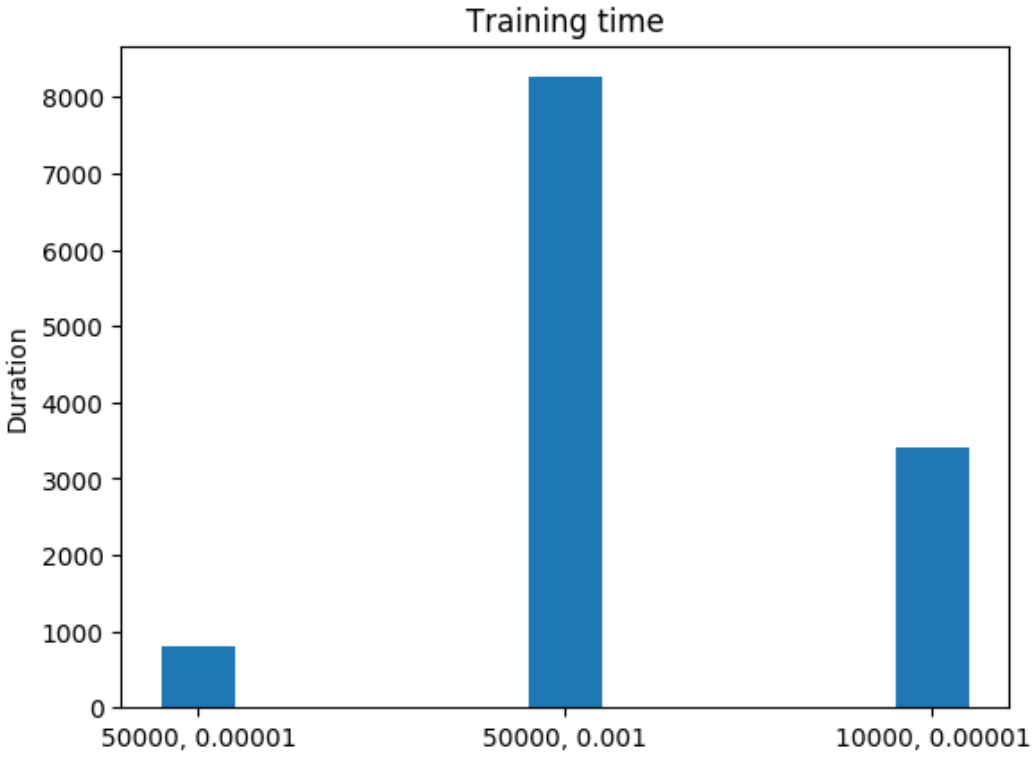


Figure 5.15 *DDPG-CartPole: Training time comparison*

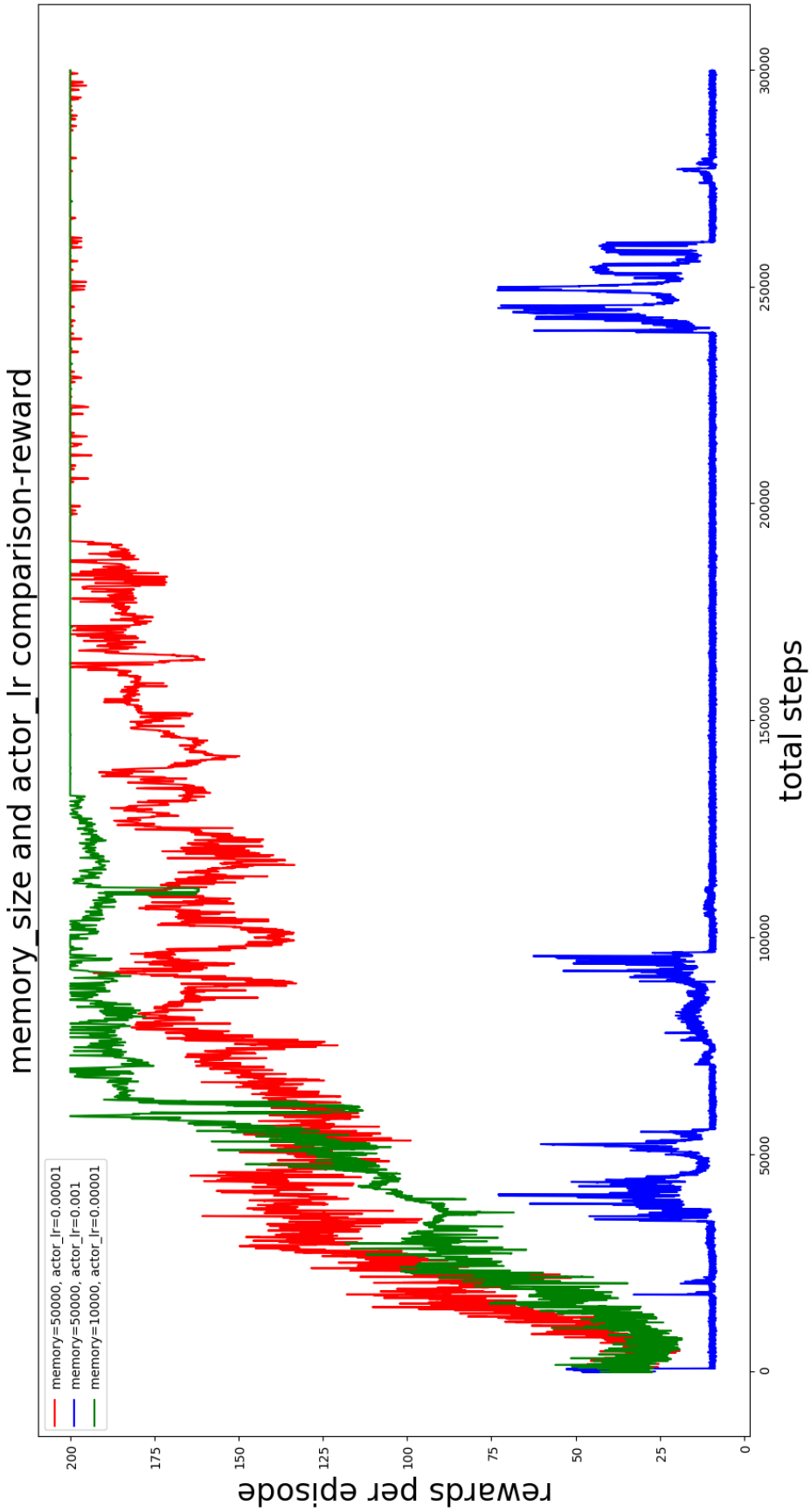


Figure 5.16 *DDPG-CartPole: memory size and actor learning rate comparison*

- **DDPG in "PlaneBall"**

We trained the DDPG agent in "PlaneBall" in 500000 training steps. When the agent triggers the termination condition, it will stop training and performing the test during the rest steps. In "PlaneBall", the termination condition is defined as: holding 800 reward in 10 episodes. Table 5.9 and 5.10 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13 with seven dimensional observation space and two dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12 5.13
Critic learning rate	0.001
Memory size	50000
Steps before training	1000
Batch size	96
Training interval	training_interval=1
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration strategy (Random process)	OrnsteinUhlenbeckProcess

Table 5.9 *Fixed parameters in DDPG of "PlaneBall"*

Tuned parameters	**Comparing Value**	
Actor learning rate	actor_lr=0.00001	actor_lr=0.001
Target net update interval	target_update=0.001	target_update=0.01

Table 5.10 *Tuned parameters in DDPG of "PlaneBall"*

As table 5.10 shows, we evaluated the 'Actor learning rate' and 'target net update interval' factors. Figure 5.18 shows the performance of two different actor learning rate, 0.00001, 0.001 and two target_net_update_interval, 0.001, 0.01. As we can see, green line coverage and reach the termination faster, but the reward in test phase is smaller than the red line. This means that the green line was not fully trained. The blue line holds in a much low reward which means that it didn't learn. In conclusion, the agent with fixed parameters(5.9) and actor_lr=0.00001, target_net_update=0.001 have a such high performance.

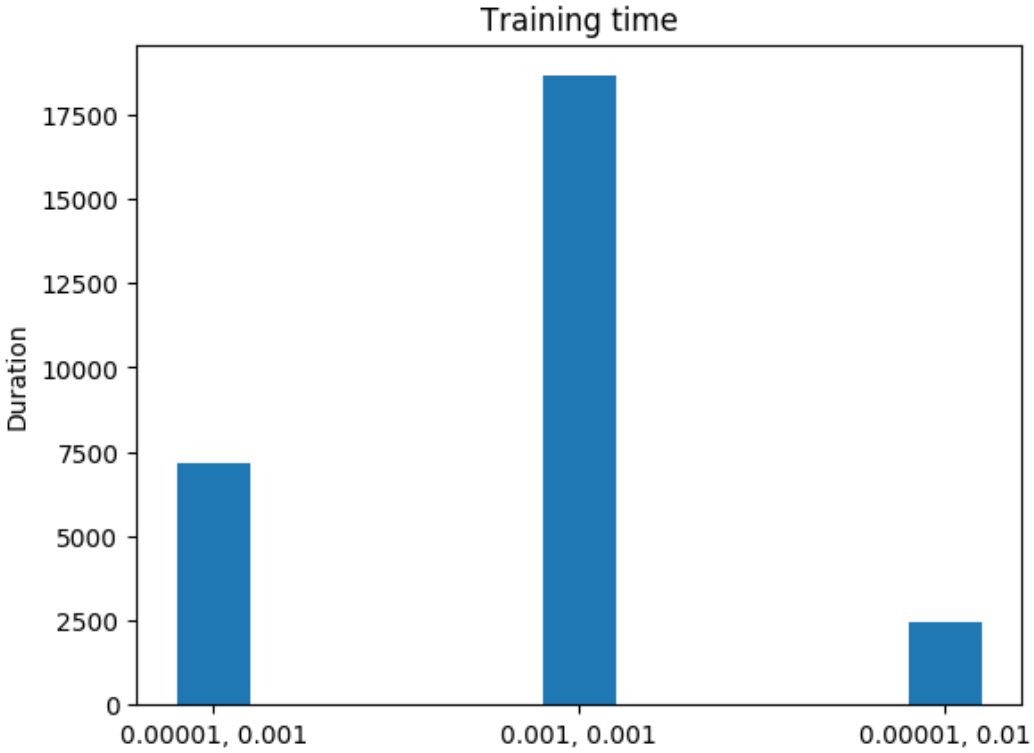


Figure 5.17 *DDPG-PlaneBall: Training time comparison*

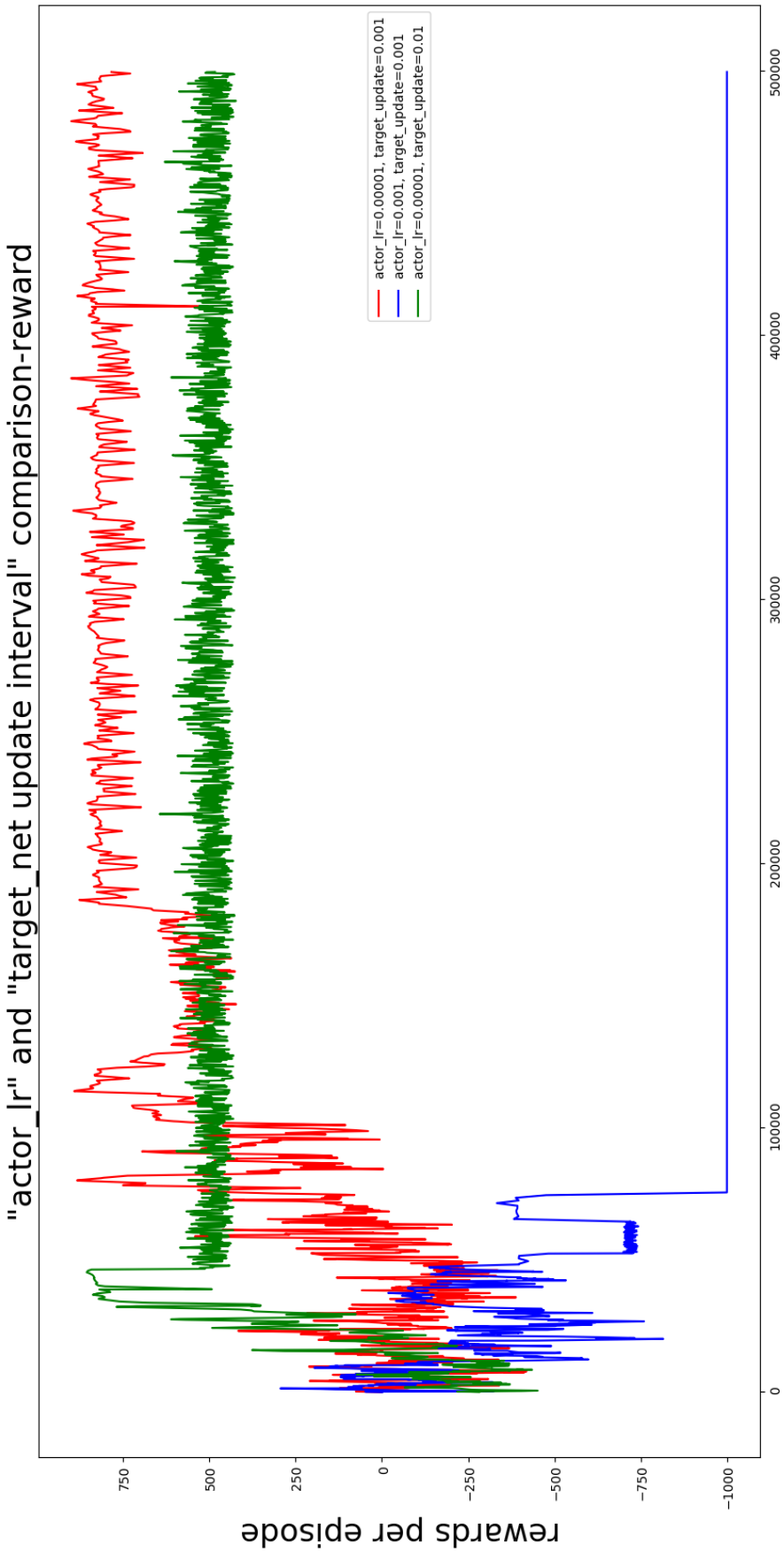


Figure 5.18 *DDPG-PlaneBall: actor learning rate and target net update interval comparison*

- **DDPG in "CirTurtleBot"**

We trained the DDPG agent in "CirTurtleBot" in 200000 training steps. When the agent triggers the termination condition, it will stop training and performing the test during the rest steps. In "PlaneBall", the termination condition is defined as: holding 600 reward in 10 episodes. Table 5.11 and 5.12 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13 with 20 dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12 5.13
Target net update interval	0.001
Memory size	50000
Steps before training	1000
Batch size	96
Training interval	training_interval=1
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration strategy (Random process)	OrnsteinUhlenbeckProcess

Table 5.11 *Fixed parameters in DDPG of "CirTurtleBot"*

Tuned parameters	**Comparing Value**	
Actor learning rate	actor_lr=0.0001	actor_lr=0.00001
Critic learning rate	critic_lr=0.001	critic_lr=0.01

Table 5.12 *Tuned parameters in DDPG of "CirTurtleBot"*

As table 5.12 shows, we evaluated the 'Actor learning rate' and 'Critic learning rate' factors. Figure 5.18 shows the performance of two different actor learning rate, 0.00001, 0.0001 and two critic learning rate, 0.001, 0.01. The red line represents actor_lr=0.0001, critic_lr=0.001; the blue line represents actor_lr=0.00001, critic_lr=0.001; the green line represents actor_lr=0.00001, critic_lr=0.01.

As we can see, the red line coverages faster than the other two, it reaches stable in around 26000 steps. After reaching the termination, the red line holds around 1000 scores, the blue line and the green hold around 900 and 800 scores separately. Therefore, the red line shows the better performance. In conclusion, the agent with fixed parameters(5.11) and actor_lr=0.0001, critic_lr=0.001 have a such high performance.

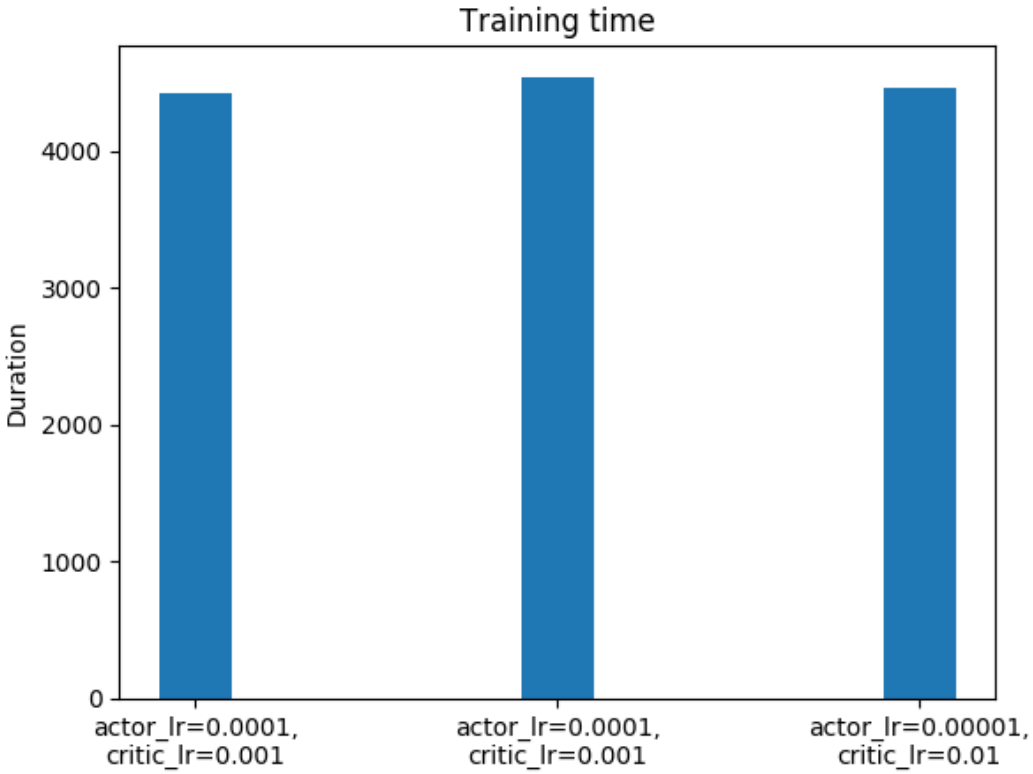


Figure 5.19 *DDPG-CirTurtleBot: Training time comparison*

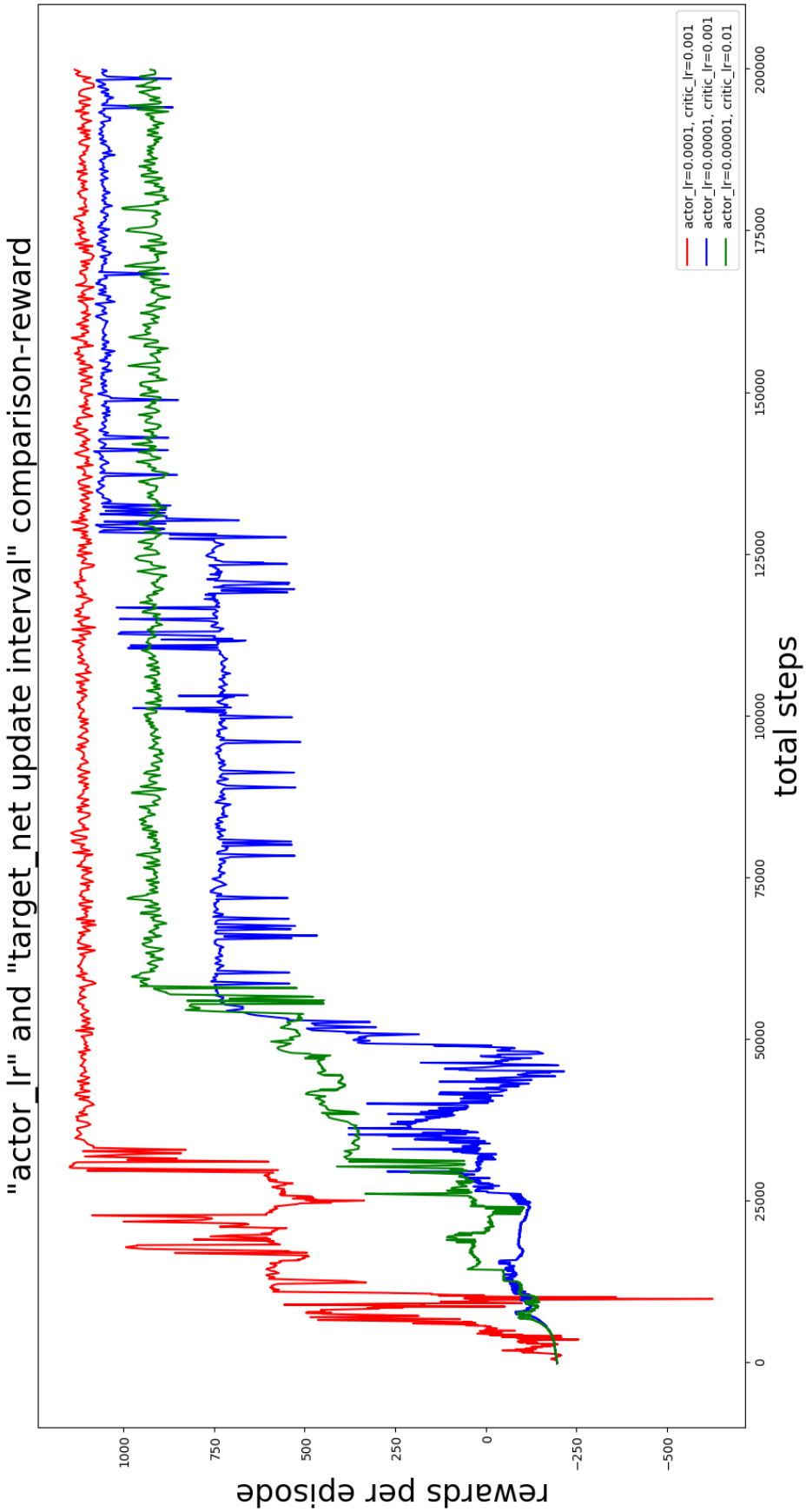


Figure 5.20 *DDPG-CirTurtleBot: actor learning rate and target net update interval comparison*

5.1.3 PPO method in environments

In this evaluation, we evaluate Distributed PPO method for three environment, we tuned the hyperparameters to compare the performance in order to get the high performance relatively. There are the influenced hyper-parameters in DPPO algorithm 5.21. For this method, we combined both PPO and A3C, where we optimize the "clipped surrogate objective" using amount of samples collecting by several parallel workers. There is a global PPO brain using for updating, each worker(agent) run in it's own environment at the same time and upload the collected samples to the global brain. The workers should wait until the global brain finish updating after every batch size samples. We used the actor-critic neural network architectures as figure 5.12, 5.13 showed. The input and output layers are different regarding to different environments.

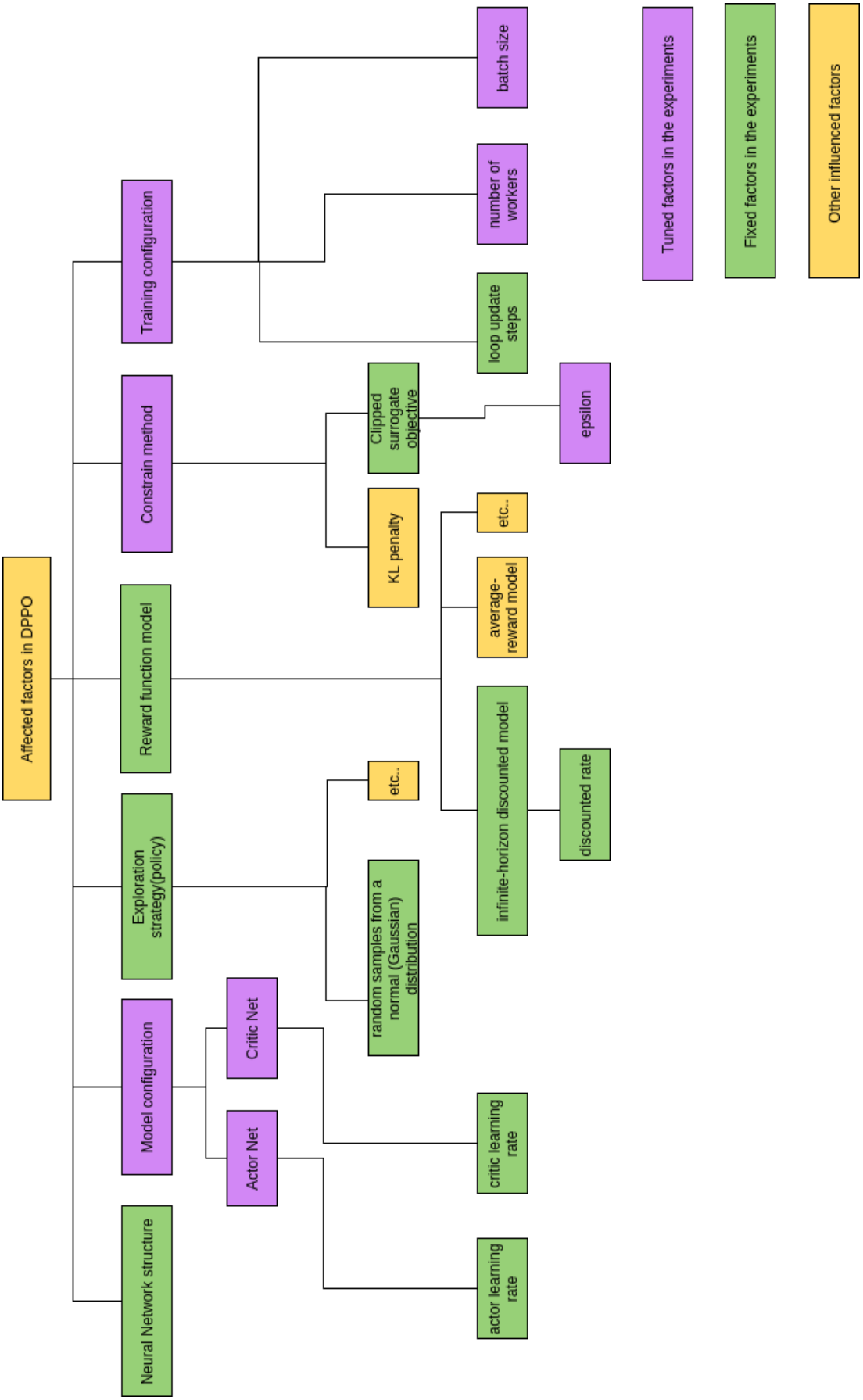


Figure 5.21 *Influenced hyper-parameters in DPPO*

- **PPO in "CartPole"**

  We trained the DDPG agent in "CartPole" in 300000 training steps.Table 5.13 and 5.14 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13, with four dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12, 5.13
Constrain methods	Clipped surrogate objective
Actor learning rate	0.0001
Critic learning rate	0.00002
Loop update steps	(10,10)
number of workers	4
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration	random samples from a normal distribution

Table 5.13 *Fixed parameters in DPPO of "CartPole"*

Tuned parameters	**Comparing Value**	
Clipped surrogate epsilon	epsilon=0.2	epsilon=0.8
Batch size	batch_size = 32	batch_size=320

Table 5.14 *Tuned parameters in DPPO of "CartPole"*

As table 5.14 shows, we evaluated the 'Clipped surrogate epsilon' and 'batch size' factors. Figure 5.23 shows the performance of two different batch size, 32, 320 and two clipped surrogate epsilon, 0.2, 0.8. Here, the batch_size factor has another meaning comparing to other methods. It means that after every batch_size examples which collected by all the parallel workers, the ppo brain will update once. The 'Clipped surrogate epsilon' factor is a hyper-parameter in PPO, which we described in chapter 2.

As we can see, the blue line and the green line appeared in fault curves. After around 30000 steps, the two lines increase rapidly and staying in 150 rewards for a short time, then increase and hold to 200 rewards. Relatively, the green line raised faster and hold the highest score earlier than the blue one. The red line also coverages fast, but the score in not stable all the time. The red line represents algorithm with batch_size = 320, epsilon = 0.2, the green line represents algorithm with batch_size = 32, epsilon = 0.2, the blue line represents algorithm with batch_size = 32, epsilon = 0.8. In conclusion, the agent with fixed parameters(5.13) and batch_size=32, epsilon=0.8 has a such high performance.

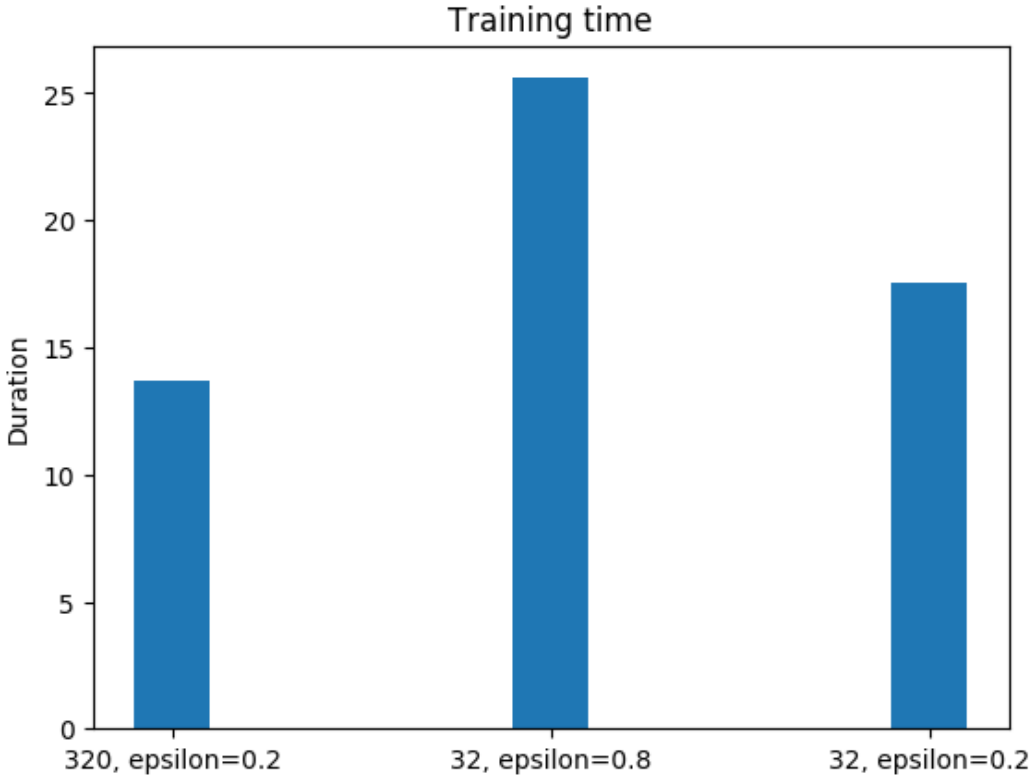


Figure 5.22 *DPPO-CartPole: Training time comparison*

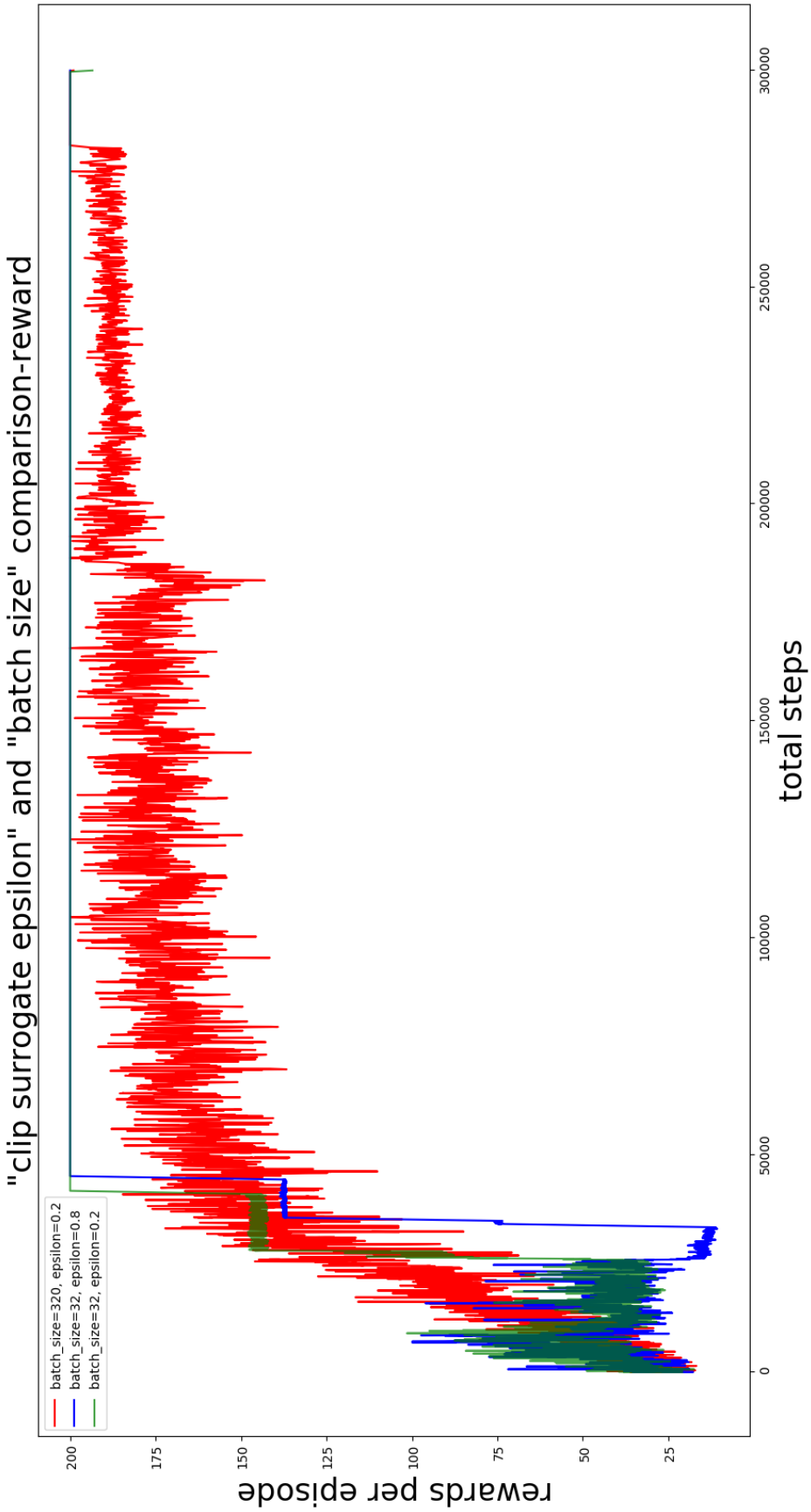


Figure 5.23 *DPPO-CartPole: batch size and clipped surrogate epsilon comparison*

- **PPO in "PlaneBall"**

  We trained the DDPG agent in "PlaneBall" in 500000 training steps.Table 5.15 and 5.16 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13, with seven dimensional observation space and two dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12, 5.13
Constrain methods	Clipped surrogate objective, epsilon=0.2
Actor learning rate	0.00001
Critic learning rate	0.0001
number of workers	4
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration	random samples from a normal distribution

Table 5.15 *Fixed parameters in DPPO of "PlaneBall"*

Tuned parameters	**Comparing Value**	
Loop update steps	actor_update_steps=10 critic_update_steps=100	actor_update_steps=100 critic_update_steps=100
Batch size	batch_size = 200	batch_size=100

Table 5.16 *Tuned parameters in DPPO of "PlaneBall"*

As table 5.16 shows, we evaluated the 'Loop update steps' and 'batch size' factors. Figure 5.25 shows the performance of two different batch size, 100, 200 and two loop update steps, (10,100), (100,100). The 'Loop update steps' factor is a hyper-parameter in PPO, which means that, the looping steps when global PPO doing update. We defined actor_update_steps and critic_update_steps separately. We compare the combination of actor_update_steps=10, critic_update_steps=100, and actor_update_steps=100, critic_update_steps=100. Here, the batch_size factor has another meaning comparing to other methods. It means that after every batch_size examples which collected by all the parallel workers, the ppo brain will update once.

As we can see, both blue line and red line appeared to learn after around 150000 steps. The blue line holds the stable average score after 200000 steps with around 650 average score. The red line holds the stable average score after 400000 steps with around 500 average score. Comparing them two, the blue line shows the better performance than the red one regarding to the coverage speed and the average score. The green line didn't learn because it oscillates between -250 to 250 during the training steps.

The red line represents algorithm with batch_size = 100, update_steps=(10,100), the green line represents algorithm with batch_size = 200, update_steps=(100,100), the blue line represents algorithm with batch_size = 200, update_steps=(10,100). In conclusion, the agent with fixed parameters(5.15) and batch_size = 200, update_steps=(10,100) has a such high performance.

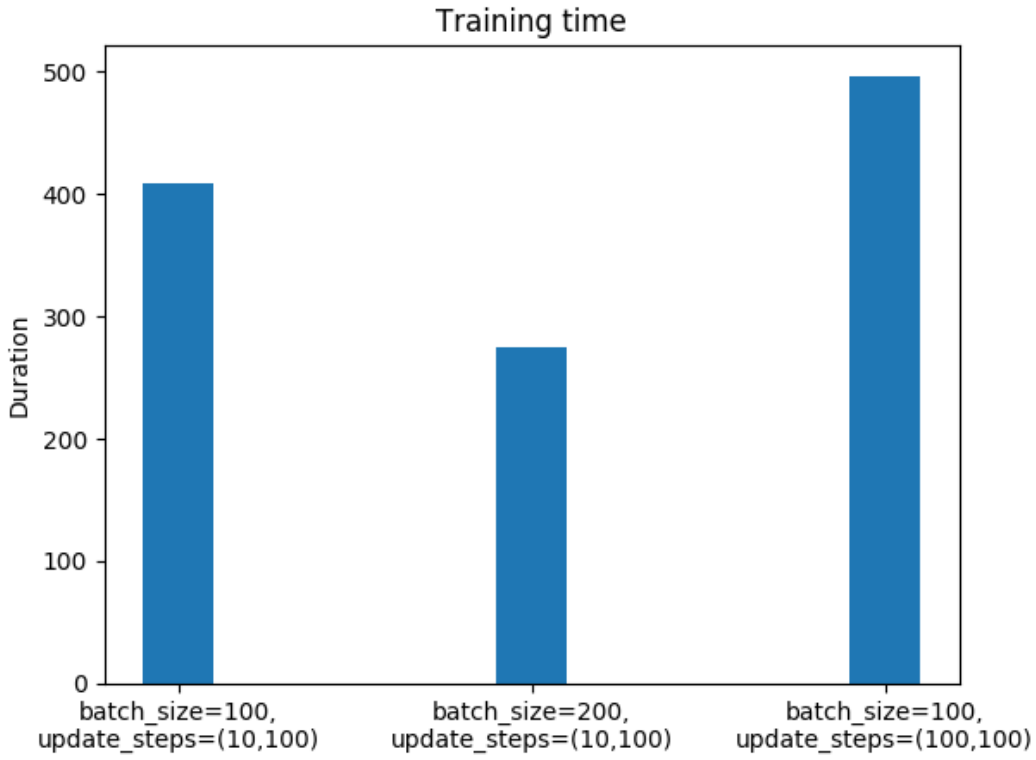


Figure 5.24 *DPPO-PlaneBall: Training time comparison*

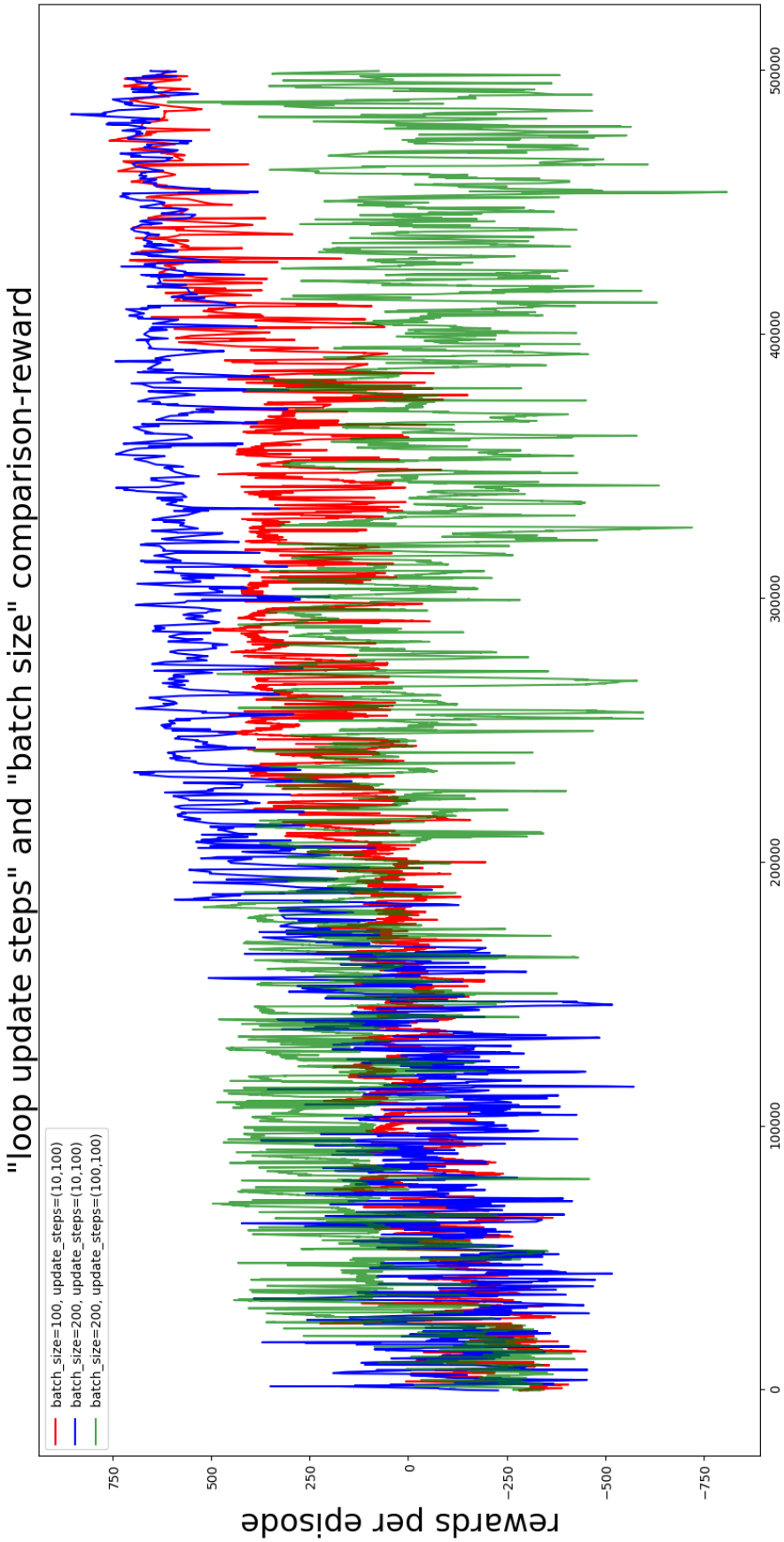


Figure 5.25 *DPPO-PlaneBall: batch size and loop update steps comparison*

- **PPO in "CirTurtleBot"**

  We trained the DDPG agent in "CirTurtleBot" in 200000 training steps.Table 5.17 and 5.18 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in figure 5.12 and 5.13, with 20 dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure: 5.12, 5.13
Constrain methods	Clipped surrogate objective
Actor learning rate	0.0001
Critic learning rate	0.00002
Batch size	32
number of workers	2
Clipped surrogate epsilon	0.2
Reward function model	infinite-horizon discounted model, discount_rate=0.99
Exploration	random samples from a normal distribution

Table 5.17 *Fixed parameters in DPPO of "CirTurtleBot"*

Tuned parameters	**Comparing Value**		
Loop update steps	actor_update_steps=10 critic_update_steps=10	actor_update_steps=1 critic_update_steps=10	actor_update_steps=1 critic_update_steps=10

Table 5.18 *Tuned parameters in DPPO of "CirTurtleBot"*

As table 5.18 shows, we evaluated the 'Loop update steps' factor with three combinations: (actor_update_steps=10, critic_update_steps=10), (actor_update_steps=1, critic_update_steps=10), (actor_update_steps=10, critic_update_steps=1). Figure 5.27 shows the performance of these three different loop update steps combinations' comparison.

As we can see, the blue line oscillates around 100, it didn't learning during the 200k steps. The red line coverages to its high score in around 150000 steps with 1300 rewards. The green line coverages to its high score in around 75000 steps with 1300 rewards. Although the red line and the green line have the same high score in general, the green line coverage faster than the red one. I contrast, after coverage, the red line is stable than the green one.

The red line represents algorithm with (actor_update_steps=10, critic_update_steps=10), the green line represents algorithm with (actor_update_steps=10, critic_update_steps=1), the blue line represents algorithm with (actor_update_steps=1, critic_update_steps=10). In conclusion, the agent with fixed parameters(5.17) and (actor_update_steps=10, critic_update_steps=1) has a such high performance.

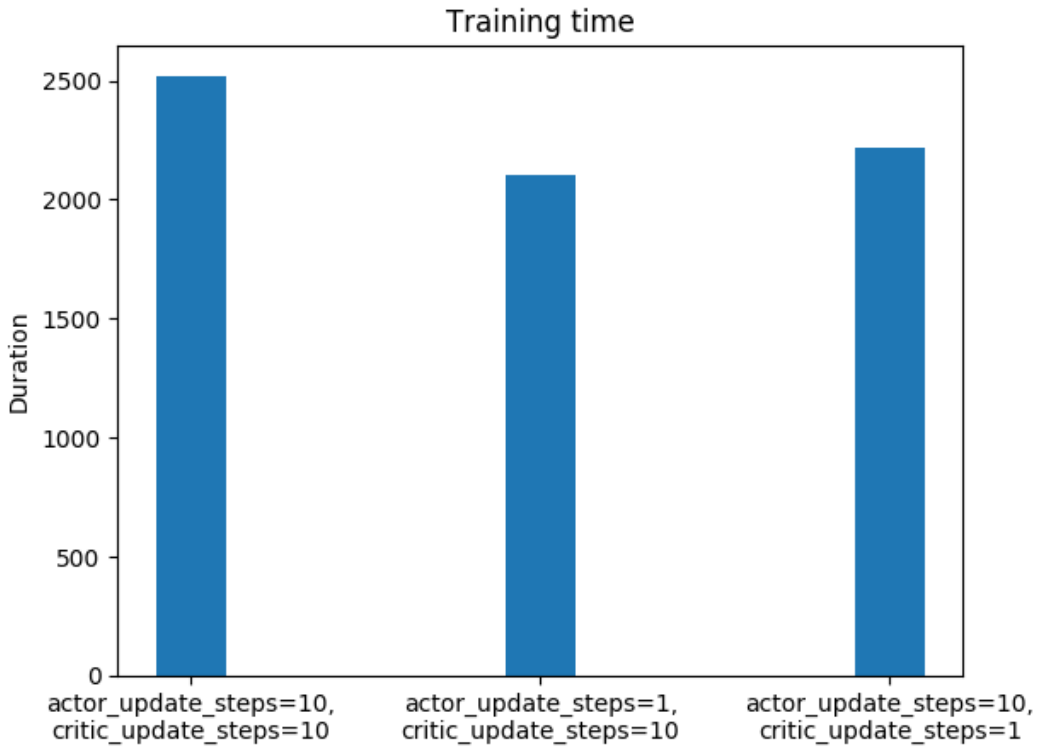


Figure 5.26 *DPPO-CirTurtleBot: Training time comparison*

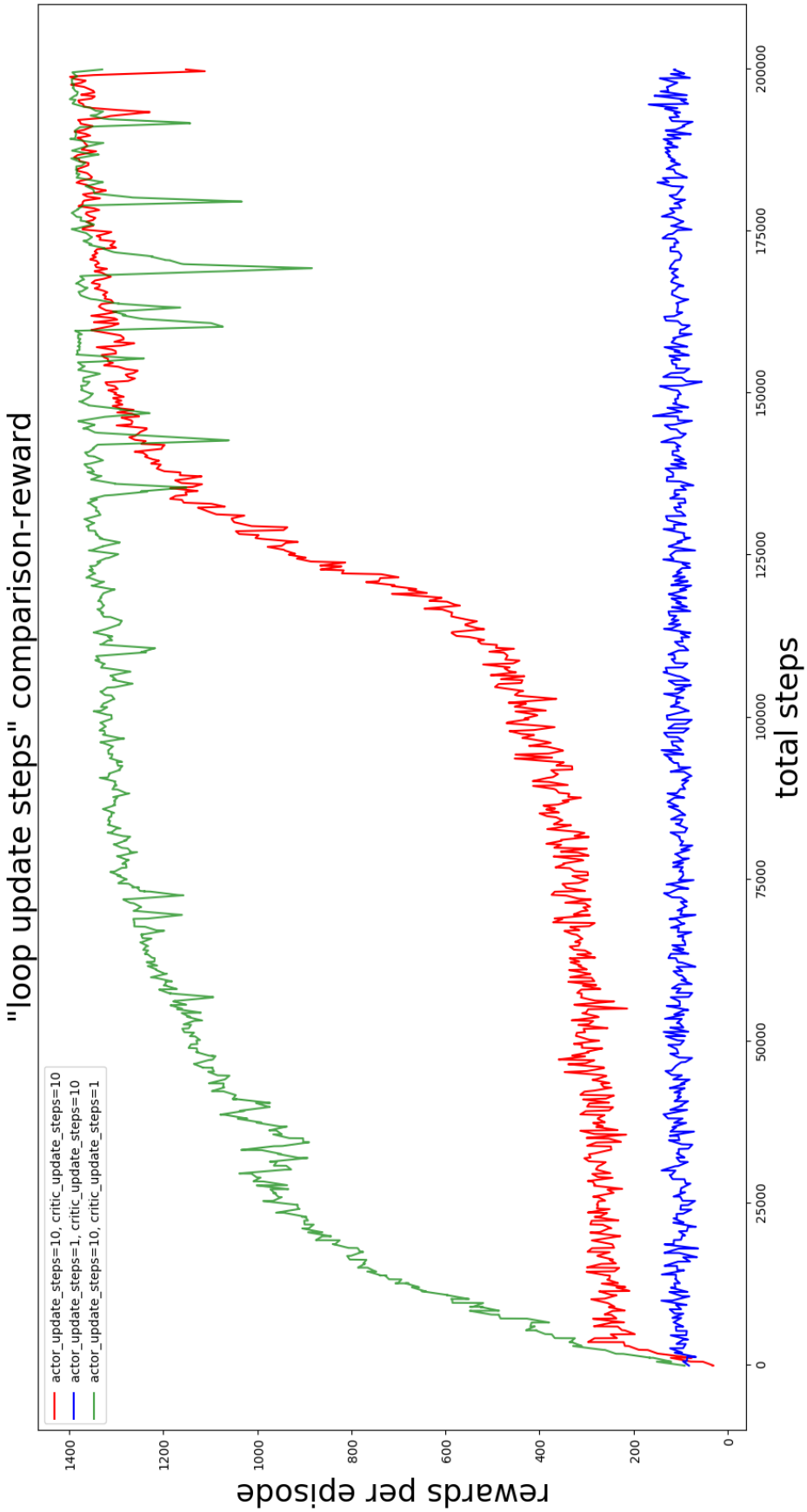


Figure 5.27 *DPPO-CirTurtleBot: loop update steps comparison*

5.2 Methods comparison through environments

In this evaluation section, we focus on the second research question. We compare the three methods with high performance guaranteed by the first evaluation section. We list all the meta-parameters which we tuned for high performance in table 5.19.

We evaluated the three methods regarding to :

- **time complexity**
  With fixed training steps, we compare the running time among the three methods.
- **sample efficiency**
  Sample efficiency is an important criteria in training performance comparison. During the training phases, the method which coverages faster has the higher sample efficiency.
- **highest and average score**
  The highest and average score of the training is an intuitional way for performance evaluation.
- **robustness**
  Robustness here means that after fully trained, the offset amplitude around the highest score.

Entity	Parameter	Value in CartPole	Value in PlaneBall	Value in CirTurtleBot
DQN	learning_rate	0.001	0.01	0.001
	memory _size	10000	100000	10000
	warm_up_steps	2000	20000	2000
	batch_size	64	640	64
	target_net_update	0.01	0.01	0.01
	train_interval	1	1	1
	discount_rate	0.99	0.99	0.99
	exploration	Boltzmann (tau=1)	Boltzmann (tau=1)	Boltzmann (tau=1)
DDPG	actor_lr	0.00001	0.00001	0.0001
	critic_lr	0.001	0.001	0.001
	memory _size	10000	50000	50000
	warm_up_steps	1000	1000	1000
	batch_size	96	96	32
	target_net_update	0.001	0.001	0.001
	train_interval	1	1	1
	discount_rate	0.99	0.99	0.99
	exploration	OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)	OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)	OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)
DPPO	actor_lr	0.0001	0.00001	0.00001
	critic_lr	0.00002	0.0001	0.0001
	batch_size	32	200	32
	discount_rate	0.99	0.99	0.99
	loop_update_steps	(10,10)	(10,100)	(10,1)
	exploration	random samples from a normal distribution	random samples from a normal distribution	random samples from a normal distribution
	constrain method	clipped surrogate objective, epsilon=0.2	clipped surrogate objective, epsilon=0.2	clipped surrogate objective, epsilon=0.2
	num_workers	4	4	2

Table 5.19 *Meta-parameter of DQN, DDPG, DPPO over three environments*

5.2.1 Comparison in "CartPole"

We evaluate the performance of DQN, DDPG and DPPO in CartPole environment. The hyper-parameters of these three methods are chosen from the first evaluation section with relatively high performance guaranteed. We trained in 300k steps, and generate the curves of 'episode reward' responding to 'total steps', as figure 5.29 shows. In this figure, red line represents DQN, blue line represents DDPG, green line represents DPPO. Among these three methods, DQN coverages faster than the other two, it reached the highest score(200) at around 25000 steps. DPPO and DDPG reached the highest score in around 40000 and 170000 steps. This means DQN has better sample efficiency than the other two. After reaching to the highest score, DQN didn't hold 200 for the whole steps, it appeared to get 197 scores in few steps. Instead, DPPO and DDPG both hold the highest score after the first time reaching it. Figure 5.28 shows the training time among these three methods. DPPO had the least training time, DDPG had the most training time with more than 4000 seconds. Both DQN and DPPO had a quite less training time with 800 and 500 seconds. In conclusion, DDPG showed the worst performance than DQN and DDPG responding to time complexity, sample complexity. DQN had better sample complexity than DPPO, but had worse robust.

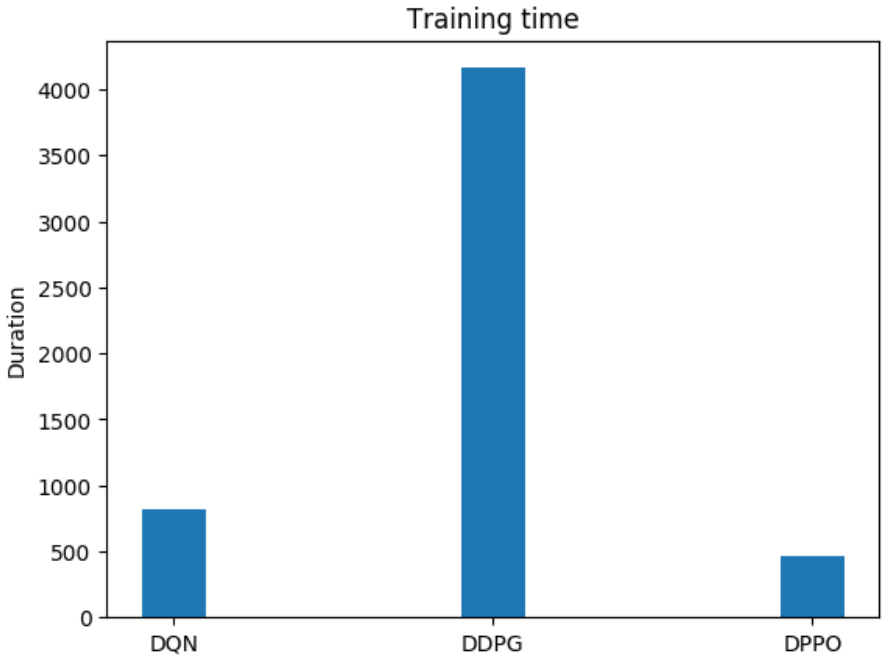


Figure 5.28 *CartPole: Training time comparison*

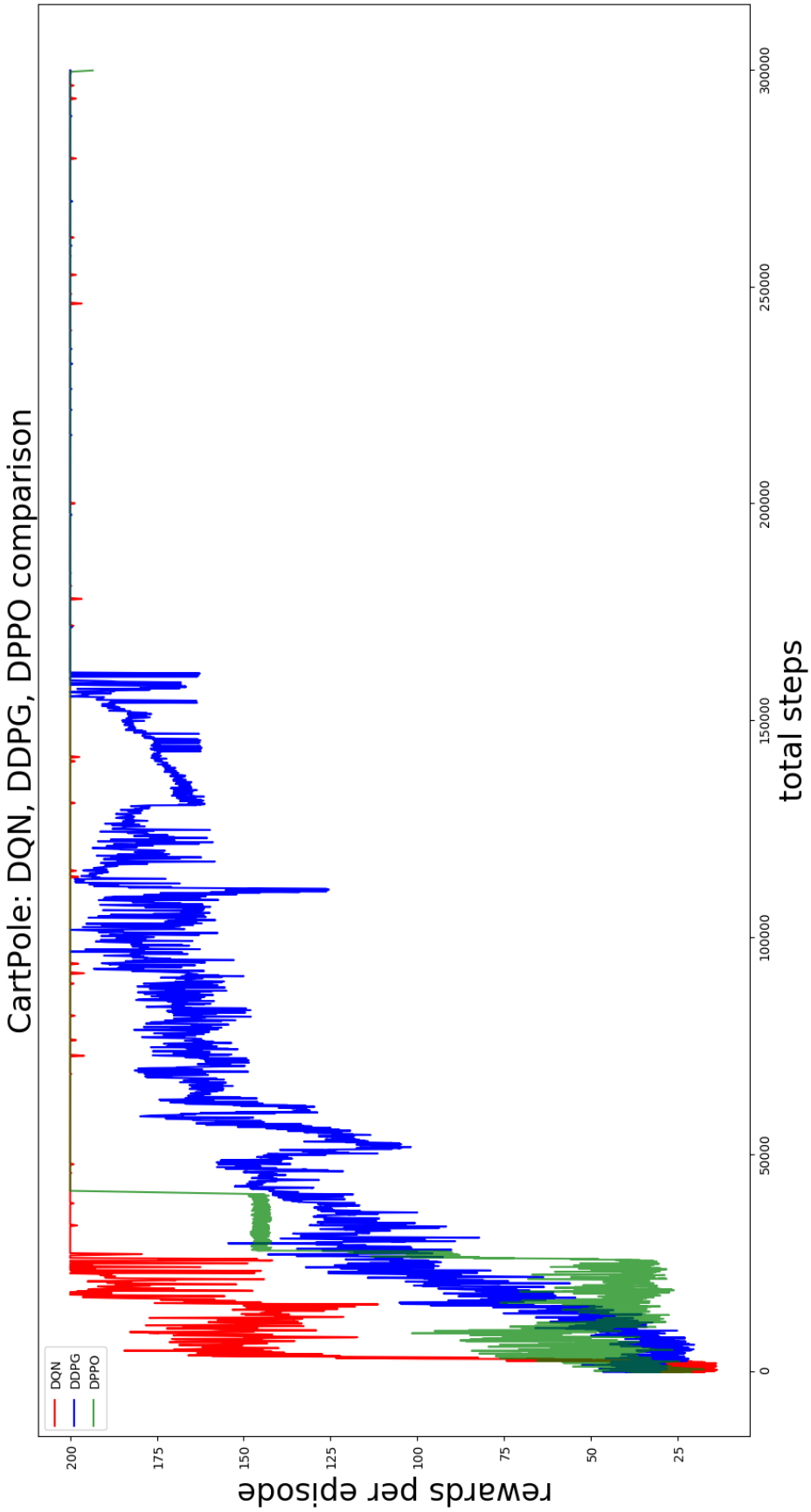


Figure 5.29 *CartPole: DQN, DDPG, DPPO comparison*

5.2.2 Comparison in "PlaneBall"

We evaluate the performance of DQN, DDPG and DPPO in PlaneBall environment. The hyper-parameters of these three methods are chosen from the first evaluation section with relatively high performance guaranteed. We trained in 500k steps, and generate the curves of 'episode reward' responding to 'total steps', as figure 5.31 shows. In this figure, red line represents DQN, blue line represents DDPG, green line represents DPPO. Among these three methods, DDPG coverages faster than the other two, it reached the highest score(900) at around 180000 steps. DPPO and DQN reached the highest score in around 200000 and 300000 steps. This means DDPG has better sample efficiency than the other two. Obviously, DDPG holds the higher score with around 700, than the other two during the training steps. DPPO vibrates around 500 score after coverage. DQN didn't really learn in 50k steps. According to the hyper-parameter tunning of DQN in PlaneBall in the first evaluation section, the DQN agent learned after 1000k steps.

Figure 5.30 shows the training time among these three methods. DPPO had the least training time, DQN had the most training time with more than 40000 seconds. Both DDPG and DPPO had a quite less training time with 8000 and 1000 seconds. In conclusion, DDPG showed the best performance than DQN and DPPO responding to time complexity, sample complexity and high score.

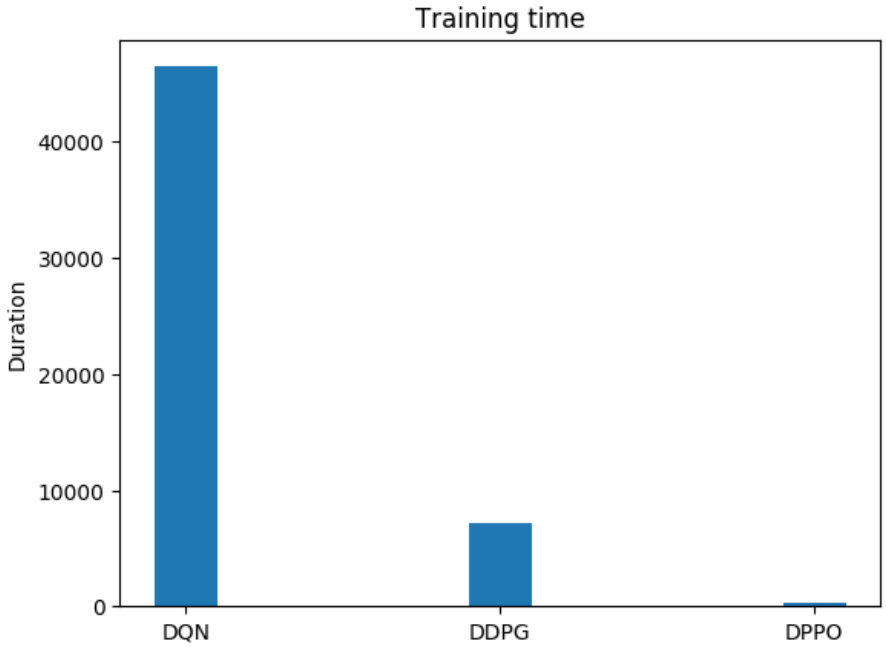


Figure 5.30 *PlaneBall: Training time comparison*

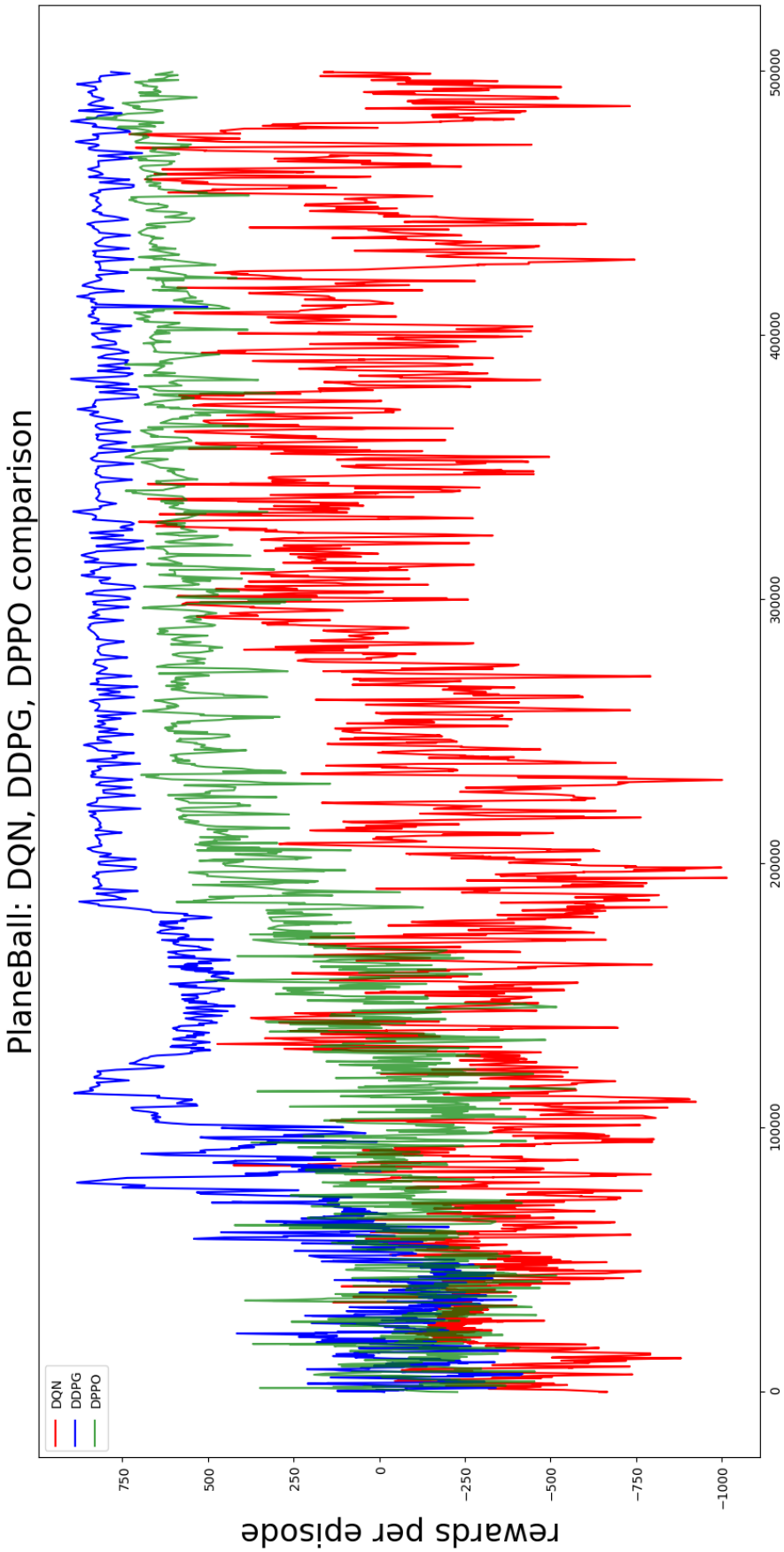


Figure 5.31 *PlaneBall: DQN, DDPG, DPPO comparison*

5.2.3 Comparison in "CirTurtleBot"

We evaluate the performance of DQN, DDPG and DPPO in CirTurtleBot environment. The hyper-parameters of these three methods are chosen from the first evaluation section with relatively high performance guaranteed. We trained in 200k steps, and generate the curves of 'episode reward' responding to 'total steps', as figure 5.33 shows. In this figure, red line represents DQN, blue line represents DDPG, green line represents DPPO. Among these three methods, DDPG coverages faster than the other two, it reached its highest score(1100) at around 30000 steps. DPPO reached its highest score(1300) in around 75000 steps. DQN oscillate between 250 and 1000 after 20000 steps. This means that DQN didn't learn well in 200k steps. Besides, DDPG has better sample efficiency than DPPO but the high score and average score are lower than DPPO.

Figure 5.32 shows the training time among these three methods. DPPO had the least training time, DQN had the most training time with more than 8000 seconds. Both DDPG and DPPO had a quite less training time with 5000 and 3000 seconds. In conclusion, DPPO showed the best performance than DQN and DDPG responding to time complexity, higher score and average score. DDPG had better sample efficiency and robust than DPPO.

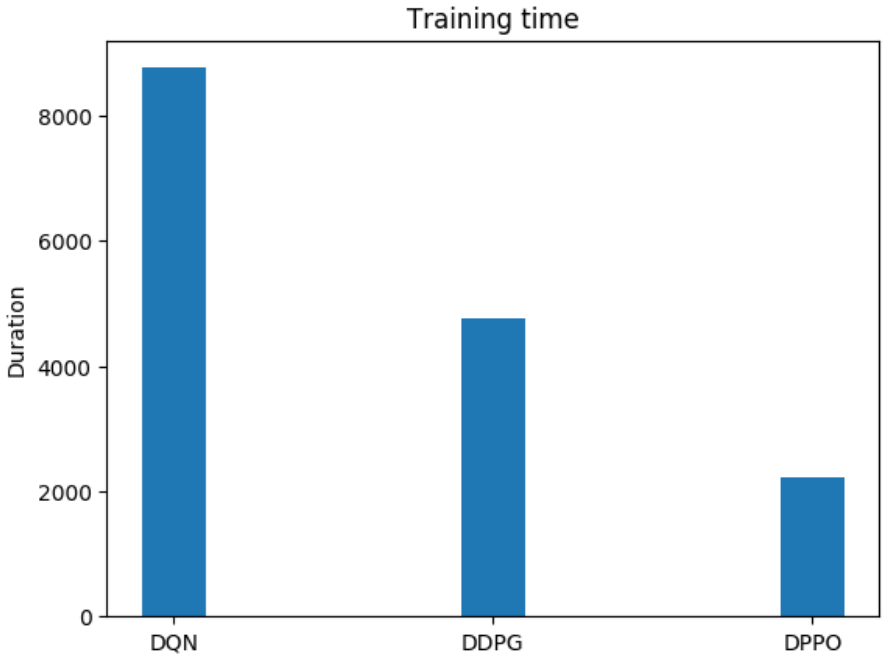


Figure 5.32 *CirTurtleBot: Training time comparison*

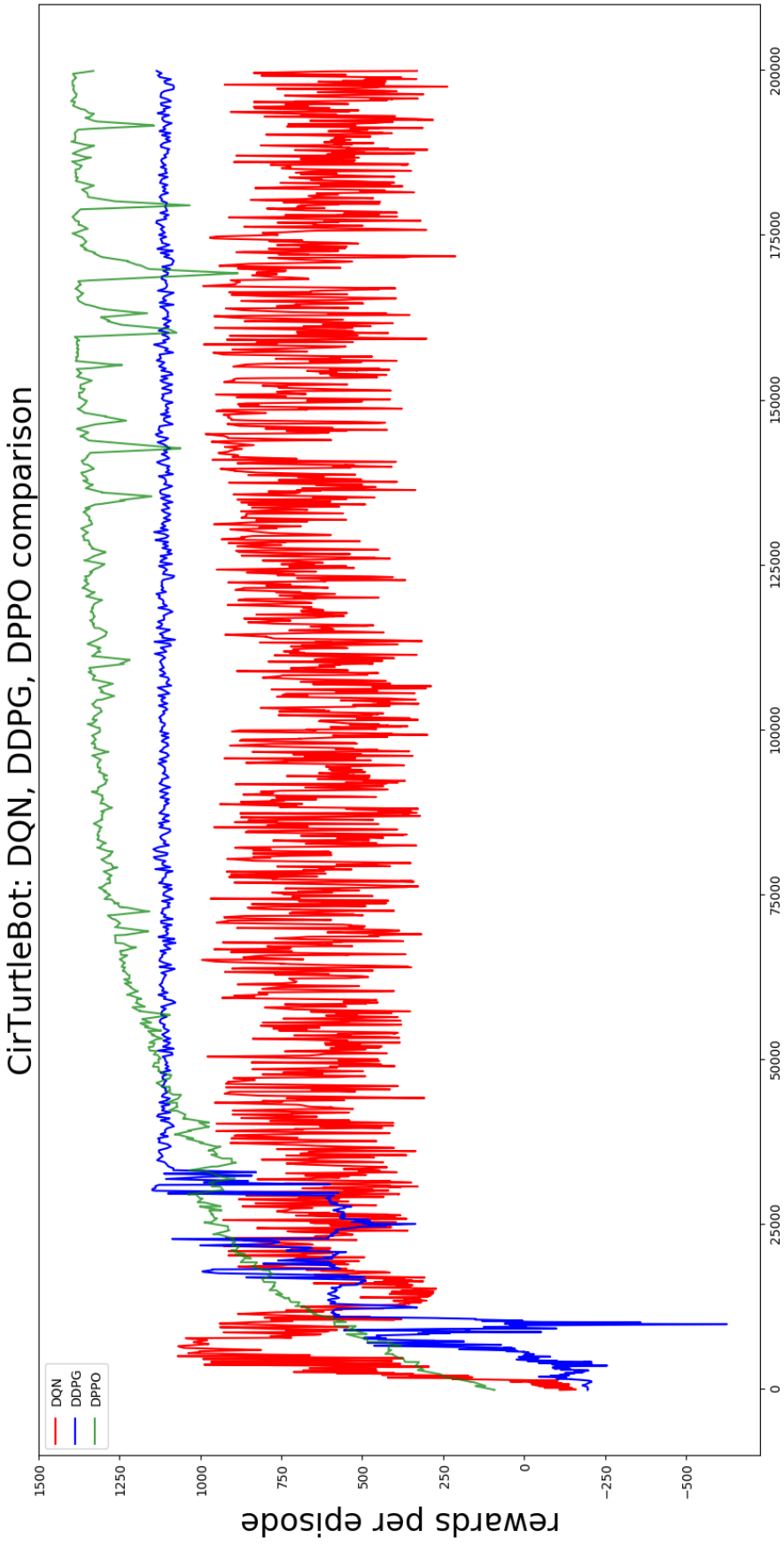


Figure 5.33 *CirTurtleBot: DQN, DDPG, DPPO comparison*

5.3 Summary

In this chapter, evaluation and results are discussed regarding to the research questions. With high performance guaranteed algorithms, for 'CartPole', DQN and DPPO both have higher performance; for 'PlaneBall', DDPG has the highest performance, DQN couldn't learn in 500k steps, and its training is much higher than the other two; for 'CirTurtleBot', DDPG and DPPO both have higher performance, DPPO has higher score and lower training time. DDPG is robust and has better sample efficiency. However, there are many threats in our experiments which have influences on the results validity, we will discuss in chapter 7 in detail.

In next chapter, related work will be discussed and compared.

6. RELATED WORK

In this chapter, we review papers and work that is similar to our task and existing benchmarks of reinforcement learning in aspect of different comparison fields. We fist discuss literatures of RL in applications, then come to challenges, last but not least, the released benchmarks.

Some papers have researched with Deep Reinforcement Learning in applications. [41] first presents the general DRL framework, then introduces three specific engineering applications: the cloud computing resource allocation problem, the residential smart grid task scheduling problem, and building HVAC system optimal control problem. The effectiveness of the DRL technique in these three cyber-physical applications have been validated. Finally, the paper study on the stochastic computing-based hardware implementations of the DRL framework, which consumes a significant improvement in area efficiency and power consumption compared with binary-based implementation counterparts. Comparing with our paper, we also list these three engineering applications, however, we focus much on the specific DRL methods and the evaluation rather than a brief definition.

In book [59], a new algorithm we call C-Trace, a variant of the P-Trace RL algorithm is introduced, and some possible advantages of using algorithms of this type are discussed. They also present experimental results from three domains: A simulated noisy pole-and-cart system, an artificial non-Markovian decision problem, and a real six-legged walking robot. The results from each of these domains suggest that that actual return (Monte Carlo) approaches to the credit-assignment problem may be more suited than temporal difference (TD) methods for many real-world control applications. Comparing to our work, we also use an cart-pole system as the experimental environment, engineering control applications are also our main focus. However, the algorithms we implemented are the recently new-arising Deep RL algorithms, and we focused on the comparison of those methods.

In [12], recent Deep RL methods like Deep Q Network, Deep Deterministic Policy Gradient and Asynchronous Advanced Actor Critic are presented in detail. Starting with traditional reinforcement learning concepts, methods, then comes to deep

reinforcement learning and function approximation. Two industrial applications robotics and autonomous driving are utilized for experiments. The testbeds are Double Inverted Pendulum , Hopper and TORCS simulator. The DDPG and A3C methods are implemented and evaluated of these three environments. Comparing with this paper, we both present the state-of-art Deep RL methods DDPG and A3C, and one experimental environment. In our paper, we also present DQN and PPO, and evaluated these four methods on three environments: "CartPole", "PlaneBall", "CirTurtleBot". Our environment are chosen respectively, they can generate to different aspects of engineering application, and their environment features are outstanding. The goal of our task is to generate the beneficial and practical criteria for further Deep RL in engineering application benchmark design.

[65, 37, 62, 48, 36] concern only in robotic field, which is one of the important Deep RL application domain of engineering applications. In robotics, the ultimate goal of reinforcement learning is to endow robots with the ability to learn, improve, adapt and reproduce tasks with dynamically changing constraints based on exploration and autonomous learning. Since robotic applications are more complex and hard to apply DRL into them, these papers focus implement DRL in robotic field to make a comprehensive analysis. In [37], they give a summary of the state-of-the-art of reinforcement learning in the context of robotics. Numerous challenges faced by the policy representation in robotics are identified. Three recent examples for the application of reinforcement learning to real-world robots are described: a pancake flipping task, a bipedal walking energy minimization task and an archery-based aiming task. In all examples, a state-of-the-art expectation-maximization-based reinforcement learning is used, and different policy representations are proposed and evaluated for each task. The proposed policy representations offer viable solutions to six rarely-addressed challenges in policy representations: correlations, adaptability, multi-resolution, globality, multi-dimensionality and convergence. Both the successes and the practical difficulties encountered in these examples are discussed. Finally, conclusions are drawn about the state-of-the-art and the future perspective directions for reinforcement learning in robotics. In [36], challenges of RL in robotics are discussed with four aspects: Curse of Dimensionality, Curse of Real-World Samples, Curse of Under-Modeling and Model Uncertainty, Curse of Goal Specification. In our paper, we discussed the challenges of RL in application with above-mentioned aspects, differently, we also presented other challenged regarding to general engineering applications.

To build on recent progress in reinforcement learning, the research community needs good benchmarks on which to compare algorithms. A variety of benchmarks have been released, such as the Arcade Learning Environment(ALE)[7] which is

currently popular benchmark,exposing a collection of Atari 2600 games as reinforcement learning problems, and recently the *RLLab* benchmark for continuous control[17], there are also on RL benchmarks regarding different comparison aspects, like[21, 76, 1, 1, 18, 16].

The Arcade Learning Environment(ALE)[7] becomes nowadays most likable benchmark. Arcade Learning Environment (ALE) is generated as a new challenge problem, platform, and experimental methodology for empirically assessing agents designed for general competency. ALE is a software framework for interfacing with emulated Atari 2600 game environments. It also provides a strict testbed for evaluating and comparing approaches. ALE is released as free, open-source software, the latest version of the source code is publicly available at: `http://arcadelearningenvironment.org`. Their purpose in presenting benchmark results for both of these formulations is two-fold. First, these results provide a baseline performance for traditional techniques, establishing a point of comparison with future, more advanced, approaches. Second, in describing these results we illustrate our proposed methodology for doing empirical validation with ALE. They provided the benchmark results using SARSA(λ), which is a traditional RL method. Based on this method, they generated five different feature representation approaches: Basic, BASS, DISCO, LSH, RAM. The cumulative rewards are compared among these approaches. They first constructed two sets of games, one for training and the other for testing. They used the training games for parameter tuning as well as design refinements, and the testing games for the final evaluation of our methods. Our training set consisted of five games: Asterix, Beam Rider, Freeway, Seaquest and Space Invaders. After that, 50 games were chosen at random to form the test set. Comparing to our work, in terms of environments, we concerned more in simulated environments which can be applied to engineering applications. In terms of RL methods, we used four recent Deep RL methods with same function approximation for results generating. In aspect of comparison criteria, cumulative reward is not only the criteria, but also other factors like algorithm robust, running time, etc.

In 2016, another RL benchmark for continuous control[17] was released. Since [7] is aim to evaluate algorithms designed for tasks with high-dimensional state inputs and discrete actions, there is a blank in comparison environments with continuous action space. This paper is focus on tasks with continuous actions. This benchmark consists of 31 continuous control tasks with sort of Basic Tasks, locomotion tasks, Partially Observable Tasks and Hierarchical Tasks. Nine RL algorithms were implemented in the benchmark, they are: Random, REINFORCE, TNPG, RWR, REPS, TRPO, CEM, CMEA-ES, DDPG. They also provide more criteria for the comparison, such like the iteration numbers of each algorithm, average reward, etc. All the source

code and the agents you can find in their rllab. Comparing to our work, our aim is to provide the practical criteria to make the comparison regarding to engineering application features. We proposed more criteria for the comparison. Alongside, the algorithms we used for experiments are DQN, DDPG, and DPPO, which is recently released Deep RL algorithm. In aspect of environments, we also choose them respectively, from simple classical control problem CartPole system to Hierarchical robotic Task.

[16] dedicated to compare the sample complexity of RL algorithms. They proposed that in robotics, improving some cost function in order to reach the controller efficiency generally requires to perform many evaluations of controllers on the real robot with different parameter values. This process is often time consuming, it may lead to abrasion of the mechanical structure or even to damage if the tested controllers generate dangerous behaviors. As a result, sample efficiency is a crucial property of any robot learning method. This paper implemented Deep Deterministic Policy Gradient(DDPG) and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithms, and are evaluated on a continuous version of the mountain car problem. Based on their evaluations, the general finding is that DDPG requires far less interactions with then environment than CMA-ES and with less variance between different runs. The reason of choosing DDPG and CMA-ES is that, DDPG is a Deep RL method based on actor-critic policy estimation approach, CMA-ES is a direct policy search method. Their evaluation could be generated as the comparison between two type of RL methods. Comparing with our work, we also did the evaluation with respect to sample complexity. Rather than that, we also compared other factors like robust, time complexity, and the implement algorithms are DQN, DDPG, and DPPO which are the currently most competitive methods.

6.1 Summary

In this chapter, we compared to the papers that are similar to what we did. We discussed what they did that is like ours and what we did that is different.

In next chapter, we give conclusions about our work and the drawbacks wich need to be improved.

7. CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize all the work and give some general ideas about further work.

7.1 Work summary

In this work, the most popular deep reinforcement learning methods are compared in three representative engineering tasks(CartPole, PlaneBall, CirTurtleBot). The algorithms include DQN, DDPG and DPPO with deep neural networks as policy representation. In order to break the gap of comparison between discrete task method and continuous task method, we implemented both discrete and continuous version for each environment. In the evaluation, we perform two aspects of comparisons regarding to the two research questions. First of all, we did hyper-parameter comparison. We evaluate one method in one environment, comparing the method with different hyper-parameters' values, choosing the one who has the highest performance. After that, we compared different methods with high performance guaranteed hyper-parameters in one specific environment. During the second comparison, we evaluate these methods through training time, sample efficiency and robust.

According to our evaluation, DQN method had the higher performance only in the "CartPole" environment. This environment is famous as the classical problem with low dimension space. We can assume that, in engineering problems, if we can generate those problem as low dimension training model, DQN is a good choice for training. DDPG had the higher performance in "CirTurtleBot" environment, and especially in "PlaneBall" environment which has the highest dimension and did the worst in "CartPole" environment. We may consider to use DDPG for higher dimension engineering model training. DPPO showed the relative higher performance among these three environments. It seems like it is suitable for both low and high dimension models. However, it we use DPPO in engineering problems, we should also take the hardware and training cost into consider, Because the performance of DPPO relies on the workers number hardly. This means that it needs better hard-

ware support than the other two methods. The training time and robust of DPPO are dependent on how many workers training at the same time. In reality, if we don't have such high hardware support for training, we could use DDPG instead.

7.2 Threats to validity

- **hardware limits**
  We have to admit that the computer's performance really can affect the running time, the training time is different of the same algorithm running in different computers. And the GPUs are much faster than CPUs. This may effect the performance of DPPO method, because we can't train the agent with more than 8 workers parallelly. Definitely, the training speed and accuracy will be improved if more workers train at the same time.

- **algorithm implementation**
  Although the pseudo code of the methods are proposed and we could get the implementations from different researchers, the way of code implementation has the influence on training time. More than that, the mechanisms for optimizing the methods are proposed and updated daily, the performance could be effected by using the newest optimize mechanism or not. For example, DQM method has many variations, such like Double-DQN. Recently, DDPG method has been updated by a mechanism call batch normalization. PPO has two methods to constrain the surrogate objective function. Due to the time and ability limitations, we can't implement all the methods with the newest mechanisms. In our work, we implemented Double-DQN, DDPG without batch normalization, DPPO with clipped surrogate objective. This may effects the comparison performance.

- **environment designs**
  As we said before, we implemented both discrete and continuous versions for each environment. The discretized size is an important issue we need to consider because this can affect the performance.

- **experiment setups**
  First of all, we perform 5 times for each algorithm and average them to get the results. The result will be more accuracy if taking more trails for the experiment. Second, we perform 30k, 50k, 20k steps for CartPole, PlaneBall, CirTurtleBot training. It will affect the robust performance comparison. Last but not least, for choosing a such high performance guaranteed hyper-parameters combination is complicated. Because we usually can't compare all the possibilities of the hyper-parameters combination.

7.3 Future work

- training with higher-performance computer, especially with GPUs.
- taking more engineering tasks for comparing, especially locomotion tasks, partially observable tasks, and hierarchical tasks .
- Implementing and evaluating A3C and DPPO with KL-Penalty, also newly proposed algorithms from now on.
- Proposing more other criterion for methods comparison.
- Benchmarking general engineering tasks.

BIBLIOGRAPHY

[1] Saminda Abeyruwan and Ubbo Visser. Rllib: C++ library to predict, control, and represent learnable knowledge using on/off policy reinforcement learning. In *Robot Soccer World Cup*, pages 356–364. Springer, 2015.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[4] M Emin Aydin and Ercan Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3):169–178, 2000.

[5] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[6] Jonathan Baxter and Peter L Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] Hee Rak Beom and Hyung Suck Cho. A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning. *IEEE transactions on Systems, Man, and Cybernetics*, 25(3):464–477, 1995.

[9] Hamid R Berenji and Pratap Khedkar. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on neural networks*, 3(5):724–740, 1992.

[10] Steven J Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in neural information processing systems*, pages 295–302, 1993.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[12] Antonio Cappiello. Deep reinforcement learning algorithms for industrial applications, 2018.

[13] Stanford CS class. Cs231n: Convolutional neural networks for visual recognition, 2018.

[14] Robert H Crites and Andrew G Barto. Improving elevator performance using reinforcement learning. In *Advances in neural information processing systems*, pages 1017–1023, 1996.

[15] Peter Dayan and CJCH Watkins. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[16] Arnaud de Froissard de Broissia and Olivier Sigaud. Actor-critic versus direct policy search: a comparison based on sample complexity. *arXiv preprint arXiv:1606.09152*, 2016.

[17] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[18] Alain Dutech, Timothy Edmunds, Jelle Kok, Michail Lagoudakis, Michael Littman, Martin Riedmiller, Bryan Russell, Bruno Scherrer, Richard Sutton, Stephan Timmer, et al. Reinforcement learning benchmarks and bake-offs ii. *Advances in Neural Information Processing Systems (NIPS)*, 17, 2005.

[19] Mohammed E El-Telbany. The challenges of reinforcement learning in robotics and optimal control. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 881–890. Springer, 2016.

[20] Chaochao Feng, Zhonghai Lu, Axel Jantsch, Jinwen Li, and Minxuan Zhang. A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for network-on-chip. In *Proceedings of the Third International Workshop on Network on Chip Architectures*, pages 11–16. ACM, 2010.

[21] Alborz Geramifard, Christoph Dann, Robert H Klein, William Dabney, and Jonathan P How. Rlpy: a value-function-based reinforcement learning framework for education and research. *Journal of Machine Learning Research*, 16:1573–1578, 2015.

[22] Ilaria Giannoccaro and Pierpaolo Pontrandolfo. Inventory management in supply chains: a reinforcement learning approach. *International Journal of Production Economics*, 78(2):153–161, 2002.

[23] Peter W Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987.

[24] Mark Hammond. Deep reinforcement learning in the enterprise: Bridging the gap from games to industry, 2017.

[25] Qiming He and Mark A Shayman. Using reinforcement learning for proactive network fault management. In *Communication Technology Proceedings, 2000. WCC-ICCT 2000. International Conference on*, volume 1, pages 515–521. IEEE, 2000.

[26] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Joonki Hong and Vittaldas V Prabhu. Distributed reinforcement learning control for batch sequencing and sizing in just-in-time manufacturing systems. *Applied Intelligence*, 20(1):71–87, 2004.

[29] Eric Horvitz. One hundred year study on artificial intelligence: Reflections and framing, 2014.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[31] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[32] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[33] Dimitrios Kalles, Anna Stathaki, and Robert E King. Intelligent monitoring and maintenance of power plants. In *Workshop on «Machine learning applications in the electric power industry», Chania, Greece*, 1999.

[34] Emre Can Kara, Mario Berges, Bruce Krogh, and Soummya Kar. Using smart devices for system-level management and control in the smart grid: A reinforcement learning framework. In *Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on*, pages 85–90. IEEE, 2012.

[35] Hajime Kimura. Efficient non-linear control by combining q-learning with local linear controllers. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 210–219. Morgan Kaufmann, 1999.

[36] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[37] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Hongjia Li, Tianshu Wei, Ao Ren, Qi Zhu, and Yanzhi Wang. Deep reinforcement learning: Framework, applications, and embedded implementations. *arXiv preprint arXiv:1710.03792*, 2017.

[42] Jiaqi Li, Felipe Meneguzzi, Moser Fagundes, and Brian Logan. Reinforcement learning of normative monitoring intensities. In *International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems*, pages 209–223. Springer, 2015.

[43] Wei Li, Qingtai Ye, and Changming Zhu. Application of hierarchical reinforcement learning in engineering domain. *Journal of Systems Science and Systems Engineering*, 14(2):207–217, 2005.

[44] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[45] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[46] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[47] Sridhar Mahadevan, Nicholas Marchalleck, Tapas K Das, and Abhijit Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 202–210. MORGAN KAUFMANN PUBLISHERS, INC., 1997.

[48] Maja J Matarić. Reinforcement learning in the multi-robot domain. In *Robot colonies*, pages 73–83. Springer, 1997.

[49] Laëtitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Reward function and initial values: better choices for accelerated goal-directed reinforcement learning. In *International Conference on Artificial Neural Networks*, pages 840–849. Springer, 2006.

[50] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[52] Abhishek Nandy and Manisha Biswas. Reinforcement learning with keras, tensorflow, and chainerrl. In *Reinforcement Learning*, pages 129–153. Springer, 2018.

[53] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer, 2006.

[54] Andrew Y Ng and Michael Jordan. Pegasus: A policy search method for large mdps and pomdps. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 406–415. Morgan Kaufmann Publishers Inc., 2000.

[55] Jorge Nocedal and Stephen J Wright. *Sequential quadratic programming*. Springer, 2006.

[56] Shichao Ou, Xinghua Wang, and Liancheng Chen. The application of reinforcement learning in optimization of planting strategy for large scale crop production. In *2004 ASAE Annual Meeting*, page 1. American Society of Agricultural and Biological Engineers, 2004.

[57] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[58] Jyoti Ranjan Pati. Modeling, identification and control of cart-pole system, 2014.

[59] Mark Pendrith and Malcolm Ryan. Reinforcement learning for real-world control applications. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 257–270. Springer, 1996.

[60] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219–2225. IEEE, 2006.

[61] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[62] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pages 1–20, 2003.

[63] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *European Conference on Machine Learning*, pages 280–291. Springer, 2005.

[64] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[65] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[66] Warren B Powell. Ai, or and control theory: A rosetta stone for stochastic optimization. *Princeton University*, 2012.

[67] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating

system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[68] Prashant P Reddy and Manuela M Veloso. Strategy learning for autonomous agents in smart grid markets. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1446, 2011.

[69] George Rzevski and RA Adey. *Applications of artificial intelligence in engineering VI*. Springer Science & Business Media, 2012.

[70] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[71] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[72] M Sedighizadeh and A Rezazadeh. Adaptive pid controller based on reinforcement learning for wind turbine control. In *Proceedings of world academy of science, engineering and technology*, volume 27, pages 257–262, 2008.

[73] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[74] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[75] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[76] Brian Tanner and Adam White. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10(Sep):2133–2136, 2009.

[77] Kanji Ueda, Itsuo Hatono, Nobutada Fujii, and Jari Vaario. Reinforcement learning approaches to biological manufacturing systems. *CIRP Annals-Manufacturing Technology*, 49(1):343–346, 2000.

[78] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[79] Yi-Chi Wang and John M Usher. Application of reinforcement learning for agent-based production scheduling. *Engineering Applications of Artificial Intelligence*, 18(1):73–82, 2005.

[80] Tianshu Wei, Yanzhi Wang, and Qi Zhu. Deep reinforcement learning for building hvac control. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[81] Wikipedia. Engineering, abet history, 2018.

[82] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[83] Huan Yang, Baoyuan Wang, Noranart Vesdapunt, Minyi Guo, and Sing Bing Kang. Personalized attention-aware exposure control using reinforcement learning. *arXiv preprint arXiv:1803.02269*, 2018.

[84] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742*, 2016.

[85] Wei Zhang. Reinforcement learning for job-shop scheduling. 1996.