Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science

Master's Thesis

Comparing Deep Reinforcement Learning Methods for Engineering Applications

Author:

Shengnan Chen

August 25, 2018

Advisors:

Prof. Dr.-Ing. habil. Sanaz Mostaghim

Intelligent Cooperating Systems (IKS)

M.Sc. Gabriel Campero Durand

Technical and Operational Information Systems (ITI)

Chen, Shengnan:
Comparing Deep Reinforcement Learning Methods for Engineering Applications
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2018.

Abstract

In recent years the field of Reinforcement Learning has come across a series of breakthroughs. By combining with developments from working with complex neural networks, popularly called Deep Learning, practitioners have started proposing various Deep Reinforcement Learning (DRL) solutions, which are capable of learning complex tasks while keeping a limited representation of the information learned. Nowadays, researchers have started applying such methods to different kinds of applications, seeking to exploit the ability of this learning approach to improve with experience. However, the DRL methods proposed in this recent period are still under development, as a result researchers in the field face several practical challenges in determining which methods are applicable to their use cases, and which specific design and runtime characteristics of the methods demand consideration for optimal applications.

In this Thesis we study the requirements expected from DRL in engineering applications, and we evaluate to which extent these can be addressed through specific configurations of the DRL methods. To this end we implement different agents using three Deep Reinforcement Learning methods (Deep Q-Learning, Deep Deterministic Policy Gradient and Distributed Proximal Policy Optimization); to solve tasks on three different environments (CartPole, PlaneBall, and CirTurtleBot), intended to represent mechanical and strategical tasks. Through our evaluation we are able to report characteristics of the methods studied. Our results can support decision making for ranking the three methods according to their suitability to given specific application requirements. We expect that our evaluation approach can also contribute to showing how comparison across models could be carried out, providing researchers with the information they need about methods.

Acknowledgements

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 Master's 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.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, August 25th 2018

Shengnan Chen

Shengnan Chen

Contents

List of Figures

1. Introduction

In this chapter, we present the context for our work (Section 1.1). We follow a problem statement that establishes the motivation behind the Thesis (Section 1.2), we also scope the goals of our work (Section 1.3). The chapter concludes with a description on the methodology we follow (Section 1.4), and the structure for the subsequent chapters (Section 1.5).

1.1 Context for our work

1.1.1 Artificial intelligence and its relevance to industry

Since the concept of "Industry 4.0" has been proposed, "How to build a 'smart factory'?" became one of the foremost questions to consider. Without a doubt, a large number of researchers are currently interested in the field of artificial intelligence (AI), seeking to employ it in different aspects of industry and business, to improve everyday tasks. In order to quantify objectively the relevance to society from these technologies, in 2017 a team from several universities and companies, developed an AI Index, a report on the state of AI [Sho17][1]. Among the collected findings authors report that each year from the last 12 years there has been a continued 9x increase in the number of AI-related academic publications. Authors also report that the number of startup ventures for AI-related business, the amount of investment in the technologies, and the job market shares for skills in this area have grown in the last decade. Such observations highlight a trend of increasing relevance in this field.

Artificial intelligence can be defined in many ways. For example, authors have suggested diverse definitions based on qualities of machines and applications of being able to act or think, either humanly or rationally [RN16]. According to Russell and Norvig, it can also be defined as a field of study that considers methods and technologies to

[1]This index has been made publicly available here: http://cdn.aiindex.org/2017-report.pdf

build intelligent agents, where such agents are systems that perceive their environment and take actions that maximize their chances of reaching a given goal or state [RN16]. Thus, one commonly used definition of AI consists of building agents capable of rational behavior.

One of the main subfields of AI is machine learning, which encompasses a set of techniques based on statistics, through which programs or agents can progressively improve their performance at a given task, without being explicitly programmed. Machine learning is usually subdivided into three categories(Figure 2.12), according to the existence or not of a feedback given to the learning process. These categories are supervised and unsupervised learning, wherewith in the first there is a feedback and in the latter, there is generally none. Each of these approaches is applicable to different scenarios [RN16]. Supervised learning is pertinent for cases where the agent can learn a given task based on sufficient labeled examples provided as training data. This includes reinforcement learning, an approach where the agent interacts with the environment receiving rewards, and it seeks to learn, by evaluating the impact of actions through trial and error, which set actions to perform to maximize a given reward [KBP13a]. Finally, unsupervised learning does not presuppose labeled data, and instead, the goal of the agent is only to learn the structure of the data (e.g. clusters or the occurrence of frequent patterns).

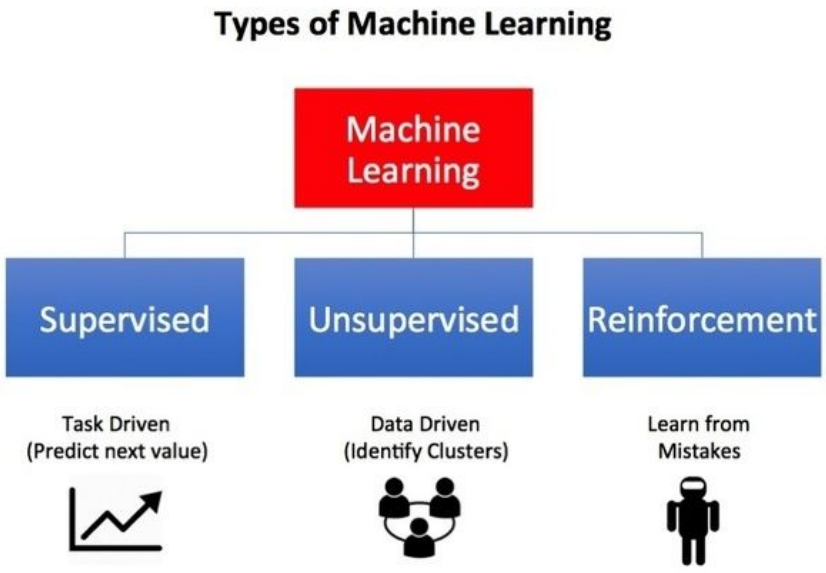

Figure 1.1: Machine learning types[2]

[2]Source:http://ginkgobilobahelp.info/?q=Machine+Learning+With+Big+Data++Coursera

1.1.2 Reinforcement learning and deep reinforcement learning

From the approaches of machine learning, reinforcement learning (RL) methods provide a means for solving optimal control problems when accurate models are unavailable [Li17]. Using RL can save programming time when building a control system. This can be achieved by considering the *controller* as an RL *agent*: the agent learns how to act from what it receives (reward) from the environment as a consequence of its actions (which usually change the state of the environment). The agent can select actions either by being based strongly on its past experience (exploiting actions) or by choosing entirely unvisited actions (exploring actions). The best trade-off between exploitation and exploration helps the agent to understand how to achieve correctly the behavior required by the task to be learned [SB+98]. A good exploration strategy is essential for the agent to be able to learn a good policy of actions to perform. Other aspects such as the optimization function, parameters, and how well the learning model is able to capture the assignation of credit (for rewards received) to past actions, can also play a large role in deciding the goodness of the trained RL agents.

Although RL had some successes in the past, previous approaches lacked scalability and were inherently limited to fairly low-dimensional problems, hence reducing the number of use cases that could be addressed with RL. These limitations exist because RL algorithms can be understood as an optimal control problem, and they share the same complexity issues as optimization algorithms [ADBB17a]. When Bellman (1957) [Bel57] explored optimal control in discrete high-dimensional spaces, he noted an exponential explosion of states and actions for which he coined the term "Curse of Dimensionality". In order to alleviate the impact of this issue RL researchers traditionally employ a series of strategies from adaptive discretization to function approximation [SB+98]. Extending such approaches, recent developments in deep learning technologies have brought forward the possibility of new function approximation solutions through the use of deep neural networks to store the learned model. This fusion of deep learning with reinforcement learning represents a new area for reinforcement learning research: deep reinforcement learning (DRL). The approach championed in this field holds practical importance since it could extend the applicability of RL to more complex real-world scenarios, and it benefits from technological developments that facilitate the use of deep learning.

In recent years many successful DRL algorithms have been proposed. Deep Q network (DQN) is one of the first DRL algorithms able to succeed in several high-dimensional challenging tasks [MKS+13]. For example, it has been shown to be able to successfully learn a behavior by observing only the raw images (pixels) of a video game it plays and receiving a reward signal based on the scores achieved at each time step [MKS+13]. DQN uses Q-learning as the policy updating strategy, wherewith the agent is designed to learn the long-term value of performing an action given a state, represented by the image of the game at a given time. Convolutional neural networks (which are used for extracting structured information from images) have been used to provide function approximation for the Q-values that are learned for pairs of states and actions. Furthermore, these networks fulfill the role of helping the model to reach an internal representation of states

that enables to provide values for unvisited states based on the proximity to already visited states. In order to achieve this some specialized techniques such as experience replay have been developed to improve the process of training the neural network by breaking the data correlations between different steps in episodes during training.

Various novel DRL methods [LHP+15, SWD+17, HSL+17, MBM+16] have been proposed in recent years. These methods are, in general, similar to DQN but they variate the RL policy-estimating methods and can involve the use of more than one neural network. Deep deterministic policy gradient (DDPG) is one of these methods, which combines a deterministic actor-critic approach with DNNs[3]. Other two popular methods are asynchronous advanced actor-critic (A3C) and proximal policy optimization (PPO).

Apart from these methods there are several optimizations available to basic DQN, like Prioritized Experience Replay and Double DQN. However, since studies so far suggest that these optimizations contribute to one another [HMVH+17], and there seem to be no trade-offs in choosing between them, we do not consider them in our work.

RL has a wide range of applications. Li lists several of them [Li17], such as games, robotics, natural language processing, computer vision, neural architecture design, business management, finance, healthcare, industry 4.0, smart grids, intelligent transportation systems, among others. From designing state-of-the-art machine translation models for constructing new optimization functions, DRL has already been used to approach several kinds 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 [ADBB17b].

1.1.3 RL for engineering applications and its challenges

When focusing on engineering applications, RL can help in the monitoring, optimization and control of systems, to improve their performance according to pre-determined targets[4]. Whereas common ML applications are tasked with learning how to make predictions, for example for speech recognition or customer segmentation, RL for engineering applications is expected to help in automation and optimization, for example in tasks pertaining to autonomous vehicles and robotics.

Since engineering is a large field including various categories, like mechanical, chemical, electrical, etc., the number of diverse RL applications included in the field could be quite large. Practitioners propose[5] that, for ease of understanding, the areas of applications could be grouped into three general functional aspects: optimization, control, and monitoring and maintenance. These are discussed in Chapter 3.

RL in engineering applications is still facing several challenges[6], such as the need for simulated environments (used for agent training) to accurately model reality, the intrinsic

[3]This and the other approaches mentioned are presented in detail in Chapter 2

[4]See: https://conferences.oreilly.com/artificial-intelligence/ai-ca-2017/public/schedule/detail/60500

[5]ibid.

[6]ibid.

uncertainty in the physical world (which makes it challenging to train agents that perform predictably even for unexpected events), the question of selecting the most suitable RL method and configuration for a given problem, the complexity in training and evaluating these models for large state spaces, among others. We will discuss more these issues in Chapter 3.

1.1.4 The need for benchmarks for RL in engineering applications

DRL algorithms have already been applied to a wide range of problems. Various DRL methods have been proposed and open source implementations are becoming publicly available every day. Researchers in the engineering 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 existing research selects DRL methods according to solely to the characteristics of the action space (whether it is discrete or continuous) and the state space (low or high dimensional) [WWZ17, KBKK12, SR08]. No doubt, these are essential factors, however, relying exclusively on these factors misses other valuable information, and might not provide sufficient guidance, especially considering that environment representation can also be adapted during building RL models.

For real-world engineering applications, the physical environments are often complex and interactions with the real environment might be too costly to use during development, consequently, there is usually a need to simulate the environments such that the agents can be trained in simulations. There are many ways to design simulated environments. For example, the "Cart-Pole" environment in OpenAI Gym is designed as providing agents with two discrete actions and a four-dimensional state space. If an engineering application finds that this environment is sufficiently close, it can be adapted to match the application, for example for balancing cable car vehicles the action space can be designed as a one-dimensional continuous action (e.g giving a push force in the range of -2 to 2 Kilograms-force). After defining such environment, however, a large number of methods are still applicable, and other criteria apart from just evaluating the reward obtained given the spaces defined but instead considering the complexity of training, or the hyper-parameter tuning required, could be useful to guide end users in selecting the method to apply.

Standard benchmarks for DRL methods need to be generated and provided to end users, helping them to determine how well a method and its configuration could match their use case.

Along with this recent progress, the Arcade Learning Environment(ALE) [BNVB13] has become a popular benchmark for evaluating RL algorithms designed for tasks with high-dimensional state inputs and discrete actions. Other benchmarks [DCH+16, GDK+15, TW09] have also been proposed regarding different aspects and requirements for RL.

The state of the art in evaluations shows some limitations, especially in how they can serve engineering RL applications. As we said before, current state-of-the-art benchmarks have been proposed regarding specific fields and aspects. ALE [BNVB13] is a popular benchmark for different RL methods in Atari game environments. However, these algorithms do not always generalize straightforwardly to tasks with continuous actions. Other work [DEK+05] contains tasks with relatively low-dimensional actions. There are also benchmarks containing a wider range of tasks with high-dimensional continuous state and action spaces [DCH+16]. However current DRL methods are not evaluated and compared in the former work. We will discuss more on the existing benchmarks in Chapter 6.

As the new successful DRL methods being proposed, like asynchronous advanced actor-critic (A3C) and proximal policy optimization (PPO), are not contained in the previous benchmarks, the performance of these new algorithms needs to be evaluated. As previously described, for engineering applications, the prototypical simulated environments could be designed in many ways. It is necessary thus to benchmark according to specific changing engineering application requirements, rather than just evaluating standard environment features.

The lack of a standardized and challenging testbed for DRL methods makes it difficult to quantify scientific progress and does not help end users from engineering applications to compare DRL approaches for a given task. A systematic evaluation and comparison can be expected to not only further our understanding of the strengths of existing algorithms but also to reveal their limitations and suggest directions for future research.

Taken together the aspects that we have presented thus far provide the context for the problem we will research in this Thesis: the relevance of AI, the potential new applications for RL based on the use of DRL, the challenges in applying these techniques for engineering applications, and the research gap in benchmarks for the aforementioned use case. Building on this we can formulate our problem statement.

1.2 Problem statement

Ideally practitioners from engineering applications who intend to use RL for a given problem/application should be provided with: a) representative environment configurations, and b) evaluations of RL agents following different methods; for practical purposes this would represent a benchmark for RL methods for engineering applications.

The use of standard environment configurations could help them determine how close is their application to the environment used in evaluations, and the reporting of reproducible evaluations, describing drawbacks and strengths from the agents, could guide the practitioners into understanding which methods could be worthwhile for their application.

Unfortunately, these standard environments and configurations do not exist for engineering applications, and furthermore, published evaluations in comparable use cases fail to consider relevant and novel RL solutions, DRL methods.

Since DRL methods extend the practical applicability of RL to high dimensional spaces, and they benefit from technological developments in working with deep neural networks, it can be expected that they will constitute alternatives for applications where RL was previously non-practical, and, as a result that there will be an increasing need for comparative benchmarks to facilitate their adoption.

The building and establishment of a benchmark is a long process that requires collaboration between researchers, such that there is an agreement on its design and such that it remains unbiased and generally informative. In preparation for such collaborative endeavor it is possible to start with foundational work by establishing potential criteria that should be included in the evaluation, assessing the usefulness of the criteria to compare the characteristics of novel DRL methods in a practical evaluation using environments representative of control and optimization tasks. This work could constitute a reasonable starting point towards a proposal for a more standard benchmark.

1.3 Research aim

In this work we propose to lay out some initial groundwork towards the building of a benchmarking suite for DRL methods in engineering applications. To this end we seek to determine the following two aspects:

- **Criteria and experimental comparison**
  Research question 1: What is the important comparison criteria regarding state-of-the-art research?
  Based on our observations we propose to carry out an experimental comparison with three different DRL methods (DQN, DDPG, DPPO) using three representative environments, reporting on how the methods fare with respect to the criteria we established. Namely we evaluate changing hyper-parameters, assessing the importance that they have and whether they require to be disclosed with benchmarking results.

- **Outline of limitations and generalization**
  Research question 2: What are the factors to benchmark different methods over a specific engineering problem?
  Using our tested environments to capture features from the best DRL models from our training experience. With this question we seek to compare the best configured models on the different tasks 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.

1.4 Research methodology

In this Thesis, we use the CRISP-DM [WH00] process model as our research methodology. This is a chosen method for building models based on data mining, since agents and

their configurations are somehow also models on how a learning process should occur, such that another learning model (i.e, the neural network, or brain of the agent) is properly built.

The steps of the CRISP-DM methodology are shown in Figure 2.11.

- **Business understanding**
  In this phase we need to understand and answer to the following questions:
  What are representative engineering applications and what is the role of reinforcement learning in engineering problems?
  How to build a DRL model according to specific problems?
  Chapter 3 records our efforts in answering these research questions.

- **Data understanding**
  In this phase we seek to understand the ideas behind the DQN, DDPG, PPO methods we compare in this paper and understand the environment data selected for our study.
  Chapter 2 and Chapter 4 collect our results from this phase.

- **Data preparation**
  In this phase we define the reward, observation and action space of the environments and the learning steps of the training phase.
  The results for this phase are summarized in Chapter 4.

- **Modeling**
  Here we prepare and program the DRL agents and define the neural network structures.
  The results for this phase are summarized in Chapter 4.

- **Evaluation**
  We compare the performance of the three methods according to the research questions and evaluate them regarding the evaluation factors.
  The results for this phase are summarized in Chapter 5.

[7]Source:https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining

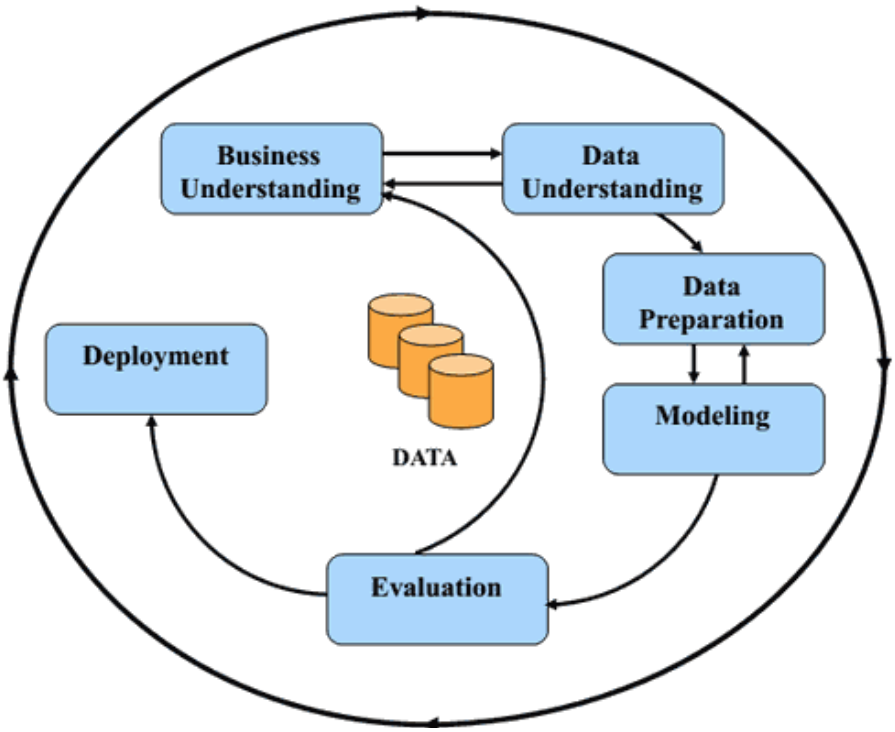

Figure 1.2: CRISP-DM process model [7]

1.5 Thesis structure

This remaining of this Thesis is structured as follows:

- Chapter 2 and Chapter 3: Contains the background for our work, thematically divided in two sections. First we present the state of the art of RL and DRL, and second we survey the use of DRL for engineering applications.
- Chapter 4: In this chapter we establish our research questions, we report on the chosen environments and the implementation for our study.
- Chapter 5: This chapter presents the experimental results and our discussion of them.
- Chapter 6: In this chapter we collect related work about comparisons and benchmarks.
- Chapter 7: We conclude this Thesis in this chapter by summarizing our findings and proposing future work.

2. Background: Reinforcement learning and deep reinforcement learning

In this chapter, we present a theoretical background on reinforcement learning and deep reinforcement learning. We structure the chapter as follows:

- We start by discussing the general theory of RL with a focus on Q-learning and policy gradients (Section 2.1).
- Next we discuss deep learning and the hyper-parameters involved in training neural networks. To provide more insights we discuss two kinds of networks: convolutional and recurrent (CNN, RNN). These are networks used for image processing and for learning on sequential data, like speech or time series (Section 2.2).
- We conclude the chapter with the presentation of DRL. More than that we present a brief selection of state-of-the-art DRL methods (Section 2.3).

2.1 Reinforcement learning

Reinforcement learning is proposed to solve problems by presupposing an agent that must learn the proper behavior to fulfill a task, through trial-and-error interactions with a dynamic and unknown environment. When actions change the environment we are in a Reinforcement Learning scenario, in which the agent is required to imagine the future gains of current actions (i.e., there is a credit assignment problem). When actions do not change the environment, we are facing a simpler problem that is address through Multi-arm Bandits, since agents are only required to learn the immediate expected reward for actions.

Authors have proposed to organize approaches to RL problems into two main groups, which are defined by strategies [KLM96]: one group searches the space of behaviors in order to find one that performs well in the environment; the other one uses statistical techniques and dynamic programming methods to estimate the utility of taking actions in given states of the world. These two approaches correspond to policy and value-based methods of RL.

Since the rise and requirement of artificial intelligence, people take advantages from the second strategy, more and more research is focused on it. In this work we also focus on the second option.

Although reinforcement learning is an area of machine learning fields, it has several differences from normal machine learning: it does not depend on preprocessed data, instead it derives knowledge from its own experience. It focuses on performance, which involves finding a balance between exploration and exploitation. Also, it is generally based on real-world environment interaction scenarios.

Reinforcement learning methods follow a basic RL model, as shown in Figure 2.1. The agent takes a certain action according to the internal action chosen strategy, based on the previous state, then interacts with the environment to observe current state and relevant rewards. This process is called a transition.

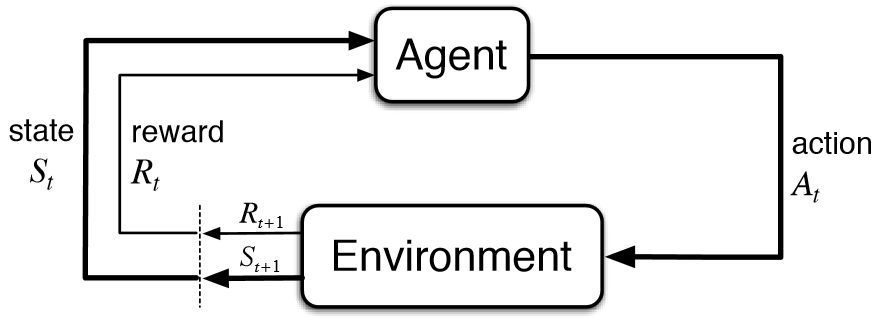

Figure 2.1: The basic reinforcement learning model [1]

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

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

- 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 the Markov assumption, which is explained as the probability of the next state depending only on the current state, and the action taken, but not on preceding states and actions. From the start of the transition to the end it is called an episode. One episode of MDP 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, the reinforcement learning problem can be seen as an optimal control problem [Pow12]. 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? [KLM96] 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 choice between these models depends on the characteristics and requirements of the application. In this paper, our formulas are based on the infinite-horizon discounted model.

After determining one appropriate optimal behaviour model now we can start thinking about algorithms for learning to get optimal policies. According to the summarization of [KCC13], reinforcement learning algorithms can be sorted into two classes:

Value function based RL algorithms
The value function [KLM96] can be represented as a 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 the 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 the 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 the black-box and white-box methods. As [ET16] described, white-box methods have the advantage of leveraging more information, and the disadvantage which could be the advantage of black-box methods is that the performance gains are a trade-off with additional assumptions that may be violated and less mature optimization algorithms with exception of models.

The core of policy search methods is iteratively updating the policy parameters θ, so that the expected return J will be increased. The optimization process can be formalized as follows:

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

, where θ_i is a set of policy parameters which is parametrized on existing policies π, and $\Delta\theta_i$ 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 dependent on the agent knowledge about 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 [KLM96]. 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 [NB18].

- 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 doesn't 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 between 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 it's a model-based RL algorithm, if it cannot, it's a model-free algorithm. Both methods have 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 learns for the specific model, sometimes it is not suitable for some other model, and it also costs time to learn another model.

Figure 2.2 shows the category of reinforcement learning methods according to [KCC13]. In the following, we focus on Q-learning, which is a classical value function based method, belonging 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 should converge to a solution (e.g., a way of playing) and settle down with that solution (exploitation). If we choose an action that "ALWAYS" maximizes the "Discounted Future Reward", we are acting greedily. This means that we are not exploring and we could miss some better actions. This is called the exploration-exploitation dilemma, and it is essential for . 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 a commonly used technique. 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. This is called an annealed greedy approach, or epoch greedy.

- *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 the 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 the 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.

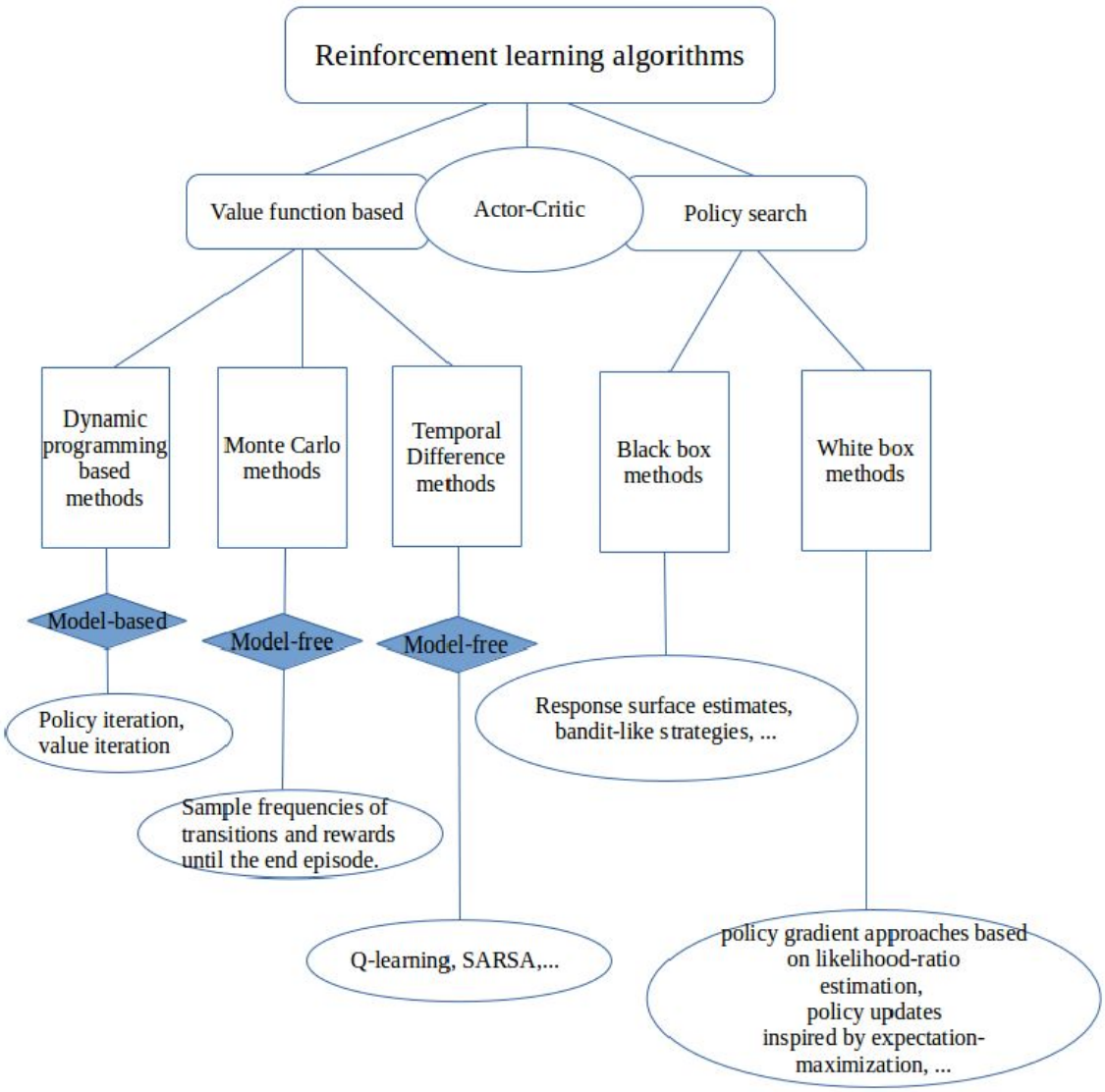

Figure 2.2: Category of Reinforcement learning methods

2.1.1 Q_learning

Q_learning is a value function based RL algorithm which learns an optimal policy, we also call it a Temporal Difference method. In 1989, Watkins proposed the Q-learning algorithm which is typically easier to implement [DW92]. Nowadays, many RL algorithms are based on it, for example, Deep Q Network uses Q-learning as the optimal policy learning, combining with Neural Network as the function approximation. First of all, we need to understand the term "Temporal Difference". One way to estimate the value function is using the 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.

The Q-learning method combines 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 taking action a in state s. As we described before, $V(s)$ is the optimal policy, the value of taking 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 the 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 combines estimate Q value and define the policy together. The Q-learning rule is:

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

This is also called the *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 a loss factor (e.g., for a game, snakes) are less, so, here the future reward is more even though the instantaneous reward is less. α determines how much the agent learns on each experience.

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 the 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 the gradient of the expected return J multiplies by learning rate α to compute the changes which represent as $\alpha\nabla_\theta J$. Therefore, the optimization process form can be written as:

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

There exist several methods to estimate the gradient $\nabla_\theta J$ [PS06], 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 known as PEGASUS in reinforcement learning [NJ00], Likelihood Ratio method [Gly87] (or REINFORCE algorithms [Wil92]). Policy gradient theorem/GPOMDP [SMSM00, BB01] and Optimal Baselines strategy are the improved approaches based on Likelihood Ratio method. Natural Policy Gradient estimation was proposed by [Kak02], based on this, [PVS05] proposed the Episodic Natural Actor-Critic.

Here we discuss *policy gradient theorem/GPOMDP*. Since it is based on the 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.

Combining the two equations above, the policy gradient can be estimated as follows:

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

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, in the likelihood ratio method it 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 [PS08]. 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 characteristics 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, as such we could also call it a Q actor-critic method.

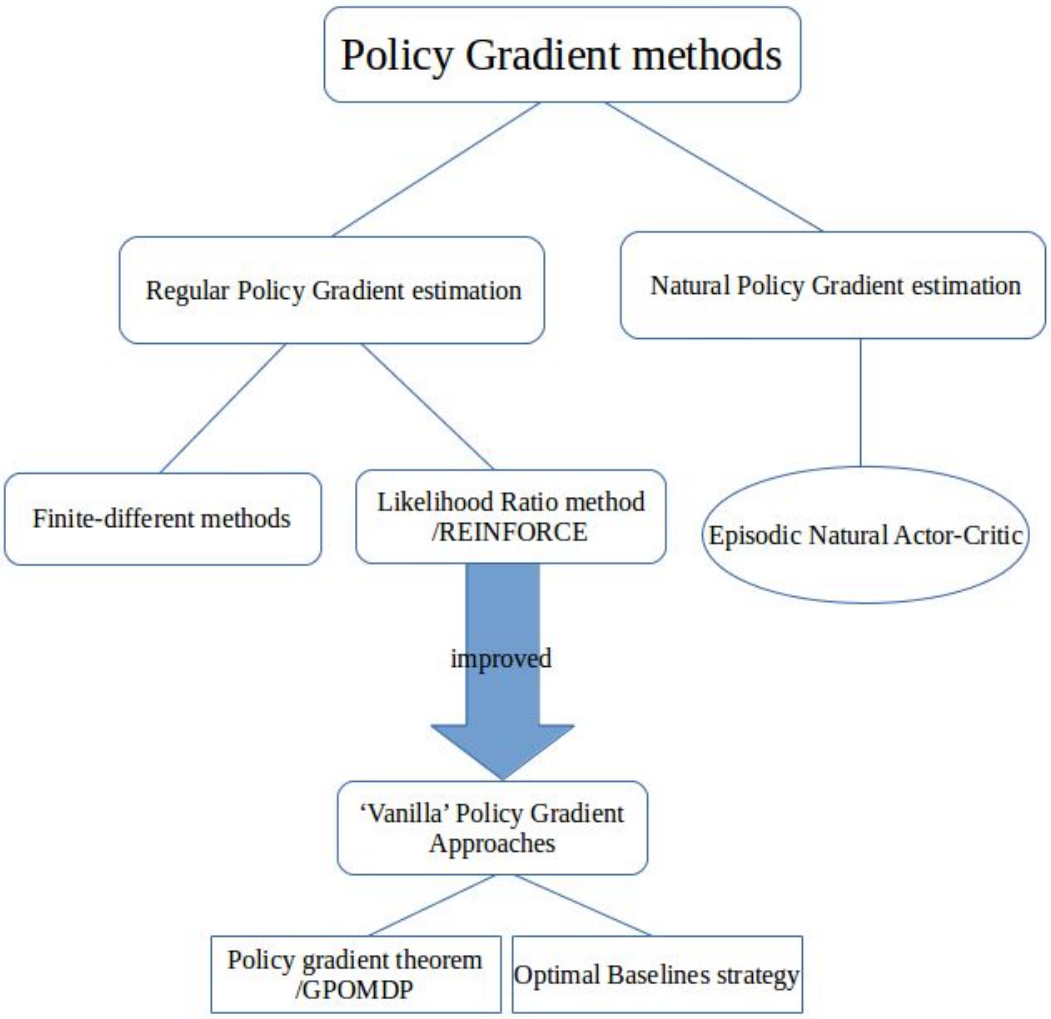

Figure 2.3: Policy Gradient methods

2.1.3 Limitations of RL in practice

Value function approaches(e,g. Q-learning) theoretically require total coverage of state space and the corresponding reinforced 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 though a small change of the local reinforce values may cause a 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 than value function methods. More than that, these methods are also available to continuous features. However, due to the above theory, the policy search approaches may cause local optimal and even cannot reach global optimal.

A combination of the value function and policy search approaches called *actor-critic structure* [BSA83] was proposed for fusing both advantages. The "Actor" is known as 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 [KBP13b]. 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 complex issues. We will discuss in the next section with a focus on deep learning and artificial neural networks.

As [Lin93] 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 a Markov decision process. We will discuss techniques around these issues in the Deep reinforcement learning section.

[1]Source: http://mi.eng.cam.ac.uk/~mg436/LectureSlides/MLSALT7/L5.pdf

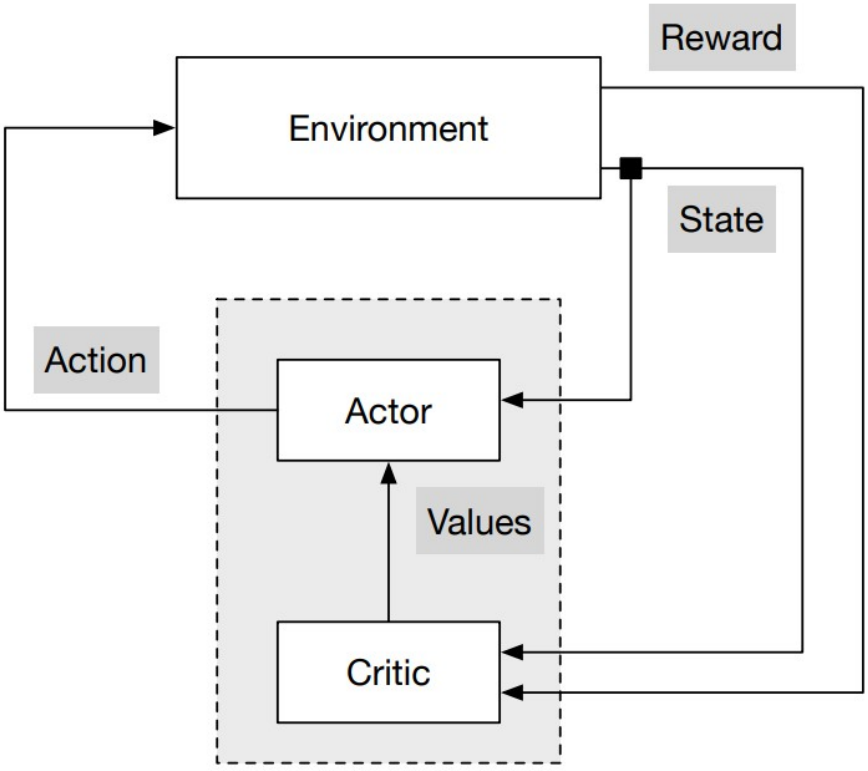

Figure 2.4: Actor-Critic Architecture [2]

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 the search. Increasingly, these applications make use of a class of techniques called *deep learning* [LBH15]. Deep learning algorithms rely on the deep neural network. Deep neural networks can automatically find compact low-dimensional representations (features) of high-dimensional data (e.g., images, text and audio) [ADBB17c]. Deep learning has accelerated the progress in many other machine learning fields, like supervised learning, reinforcement learning, among others.

A deep neural network(DNN) consists of multiple layers of nonlinear processing units (hidden layers). It performs feature extraction and transformation. Figure 2.5 shows, a deep neural network (DNNs) consisting of three layers, the input layer, hidden layers, the output layer. In the 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 the 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. Noting 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 a nonlinear transformation. We need an 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 activation function to fit your own issue. There are also several pre-defined AFs, such as 'relu', 'tanh', etc.

After computations flow forward from input to output, in the output layer and each hidden layer, we can compute error derivatives backwards, and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function [Li17]. This is the core of the learning part, to find the right weights and biases.

Still, DNNs have a knotty issue to solve: Statistical Invariance or Translation Invariance. Imagine we have two images with the exactly same kitty on them, the only difference is the kitty is located in a different position. If we want the DNNs to train to recognize

it is a cat on both images, the DNNs should give different weights. The same issue comes from text or sequences recognition. One way to solve it is Weight Sharing. For the image, people built the Convolution Neural Network(CNN) structure, for text and sequences, people built Recurrent Neural Network(RNN). Simply stated, CNN's use shared parameters across space to extract patterns over the image, RNNs do the same thing across time instead of space.

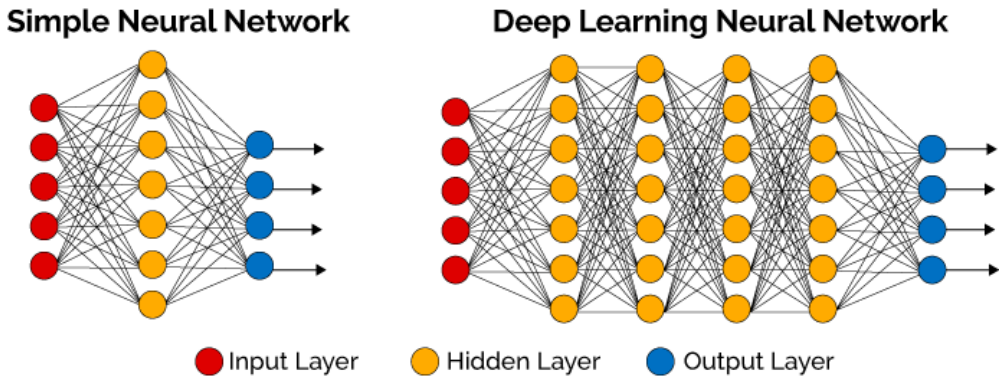

Figure 2.5: Simple NNs and Deep NNs [3]

[3]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, CNNs are used to solve Translation Invariance issues, 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 [cla18]. 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 goes 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 the input image or not.
- **Pooling Layer:** Pooling layers perform a downsampling operation (subsampling) and reduce the input dimensions. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting [cla18]. 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 architectures are summarized in Figure 2.8.

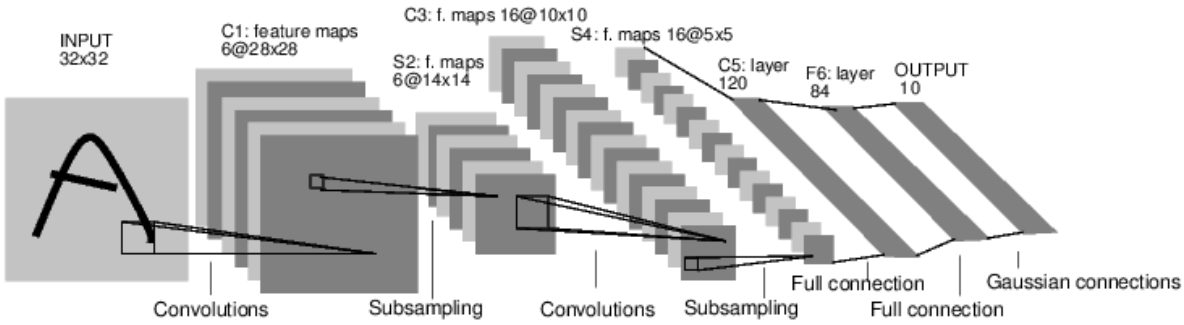

Figure 2.6: Architecture of LeNet-5 [LBBH98]

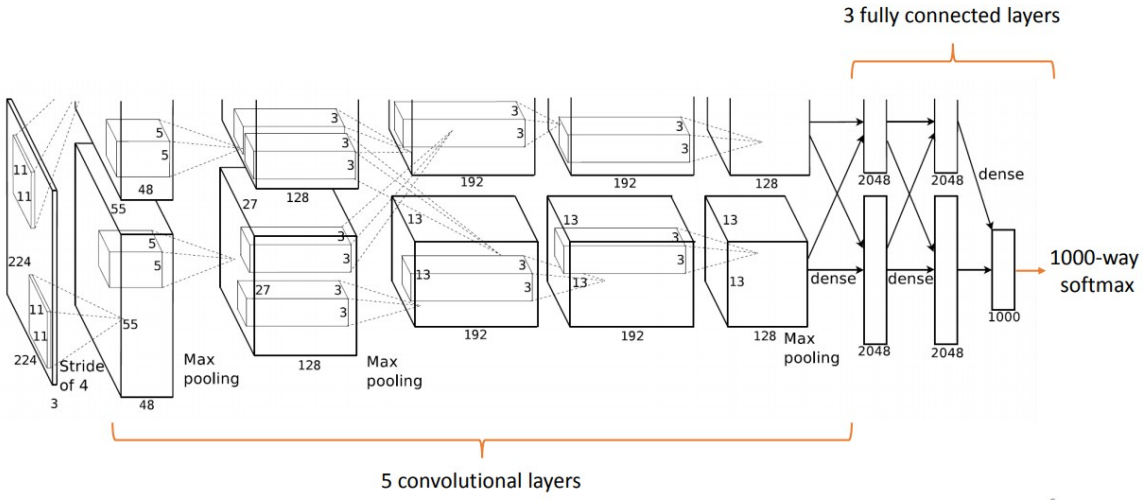

Figure 2.7: Architecture of AlexNet [KSH12]

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 popular CNN architectures [4]

[4]Source: https://medium.com/@siddharthdas_32104/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-66

2.2.2 Recurrent Neural Network(RNN)

As discussed before, RNN shares parameters(weights) to process text over time. In another word, 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. For example, we want to translate the sentence, we better know the whole input sentence and the order of the sequence.

A typical RNN structure is shown 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 state$s_0$ is typically initialized to all zeros; 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 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 infinity (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* [PMB13] 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) [HS97]. LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. In practice also the choice of activation functions can help in dealing with gradient problems.

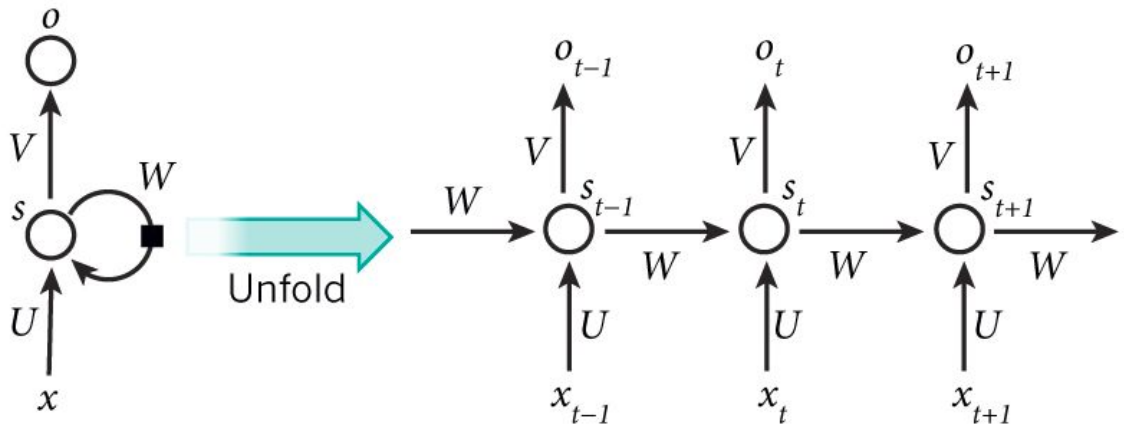

Figure 2.9: Typical RNN structure [5]

[5]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 the first thing to consider is hyperparameters. Here we list our collected understanding on the considerable hyperparameters in this domain.

Ordinary Neural Networks

- Number of layers
- Number of hidden-layers' neurons
- Sparsity or not in connections
- Initial weights
- Choice of activation functions
- Choice of loss functions
- Optimization types (e.g., Adam, Rmsprop, etc.)
- Clipping or not
- Regularization of functions
- Batch sizes
- Learning rate
- Splits and the optional use of k-folds cross validation
- Kinds of numeric inputs and their domains

CNNs specialization

- Number of convolutional layers
- Number of FC layers
- Patch/filter size
- Padding types
- Pooling types
- Layers' connection types
- Stride length

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, 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 the problem of inefficient 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 the Deep neural network which belongs to Deep learning domain, a new arising field *Deep Reinforcement Learning* showed up to help solve RL in high dimensional domains. 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 breaks the "Curse of dimensionality" intrinsic to large spaces of actions to explore.

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 wasteful. What we need is some way to generalize and match by patterns between states and actions. We need our algorithm to say "the value of this kind of states is X" rather than "the value of this exact, 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 the Actor-Critic method which approximating value function and direct policy. In the 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 the most popular and effective algorithms, proposed by DeepMind and OpenAI. In this work, these 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 [MKS+13], 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 [MKS+13] used the images shown on the Atari emulator as input, using convolution neural network to process image data. Q-learning algorithm was used to make the decision, with stochastic gradient descent to update the weights. Since deep learning handles only with independent data

samples, the *experience replay* mechanism was used to break correlations. Generally, DQN algorithm replaces the tabular representation for Q-value function with the 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 previously. 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 led out by actions chosen. Second, the collected data distributions of RL are non-stationary because RL keeps 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) [Lin93]. 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 hyper-parameters could be controlled during DRL method designing and evaluation, the buffer size and mini-batch size. Large buffers mean that the agent will be trained several times on relatively old experience, possibly slowing down the learning process. Large mini-batches, on the other hand, could increase the network training time.

Fixed Q-target
Another essential breaking correlation mechanism called *Fixed Q-target*. We produce two neural network structure for DQN, with the same structure but different parameters(weights). In equation 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 computing of Q-target uses the old parameters and Q-predict use the current parameters, this can also break the data correlation efficiently. Moreover, since policy changes rapidly with slight changes to Q-values, the policy may oscillate. This mechanism can also avoid oscillations. Figure 2.11 shows the complete pseudocode of Deep Q Network with experience reply which is produced by [MKS+13].

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 is ordered text sequences. Moreover, the hyperparameters of NNs, e.g, the layers, neurons, could also be adjusted flexibly. Recently, an advanced DQN algorithm called Double DQN was proposed by [VHGS16]. In Double DQN, the online network predicts the actions while the target network is used to estimate the Q value, which effectively 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.

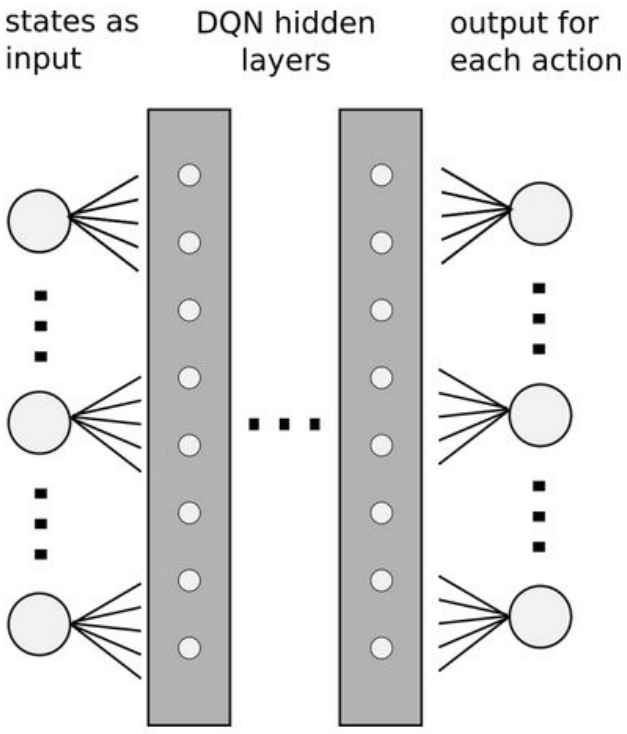

Figure 2.10: Deep Q-learning structure [6]

[6]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 [MKS+13]

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 have been proposed by [LHP+15]. It used an actor-critic approach based on the DPG algorithm [SLH+14], combined with *experience replay* and *fixed Q-target* techniques which inspired by DQN to use such function approximation in a stable and robust way. In this algorithm, a recent advantage in deep learning called *batch normalization* [IS15] is also adopted. 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 [SLH+14] based on Actor-Critic methods which we have discussed previously. 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 for function approximation, 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, the experience replay 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 was updated every k steps, the parameters of the target networks are updated in the 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, the 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 is shown in Figure 2.12.

Batch Normalization

Additionally, a robust strategy called batch normalization [IS15] 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 a unit mean and variance (i.e, normalization). In a deep neural network, each layer thus receives this 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 [LHP+15]

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 converge, opposite to finish learning the policy will take a very long time. The new robust policy gradient methods, which we call proximal policy optimization [SWD+17, HSL+17] was proposed to solve the problem, and it has some of the benefits of trust region policy optimization [SLA+15], 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 [SLA+15], This algorithm is similar to natural policy gradient methods which we mentioned previously. PPO is a variant of TRPO, it directly uses the first order optimization methods to optimize the objective.

Surrogate objective function
Previously we mentioned the 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 an 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 a "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 [NW06]. Simply stated, 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, $\rho_{\theta_{old}}$ are the is the discounted visitation frequencies in θold.

According to the theories above, PPO [SWD+17] presented two alternative ways to maximize the surrogate objective function with constraints: *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 adopt the penalty coefficient β so that we achieve some target value of the KL divergence d_{target} each policy update. Simply stated, 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
Figure 2.13, Figure 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. We will discuss the asynchronous method in detail in the 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 $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$

Figure 2.13: PPO Pseudocode by OpenAI [SWD+17]

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 [HSL+17]

2.3.5 Asynchronous Advanced Actor Critic

Asynchronous Advanced Actor Critic [MBM+16] 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. Giving 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 gets different transitions when playing in the same environments. Figure 2.14 shows the pseudocodes from DeepMind.

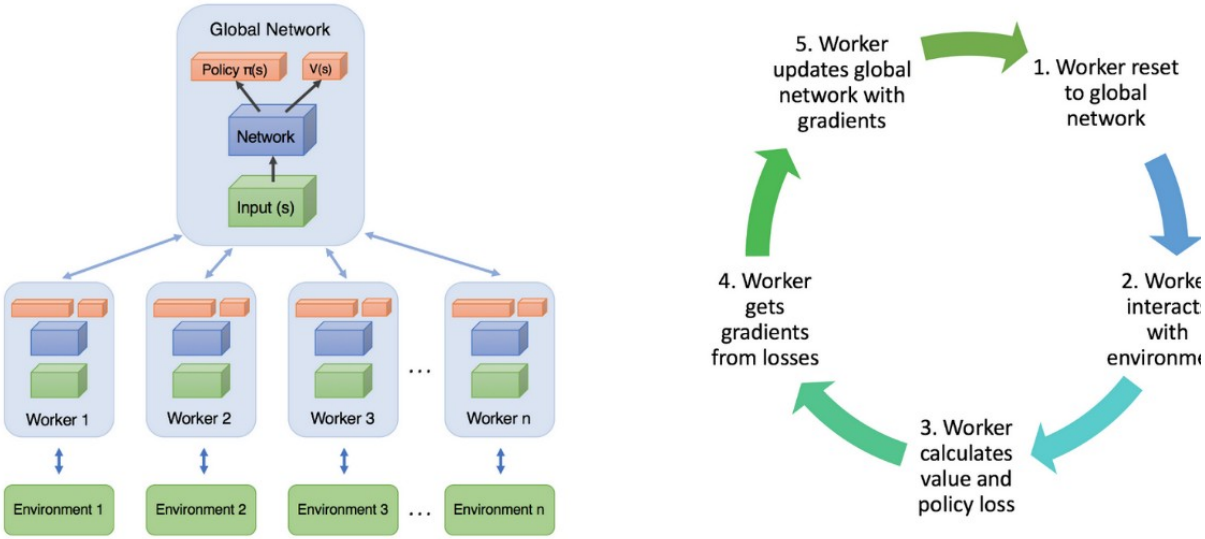

Figure 2.15: A3C procedure [7]

[7]Source: https://morvanzhou.github.io/
[8]ibid

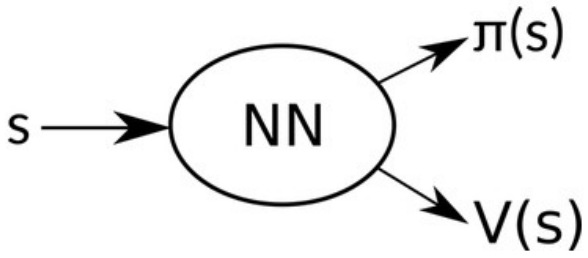

Figure 2.16: A3C Neural network structure [8]

2.4 Summary

In this chapter, we discussed the background of 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. After this we are able to list hyperparmeters, which are pertinent to our discussion on what hyperparameters algorithms must disclose. Alongside, the hyper-parameters of DL have been discussed, they may affect our following experimental results and evaluation. Last but not least, we present four up-to-date advanced DRL methods: Deep Q Network, A3C, DDPG, PPO.

In the next chapter, we will discuss the engineering applications of reinforcement learning and the challenges of its implementation, as disclosed in the literature. Also, we will outline the current challenges of DRL for engineering applications, based on our literature review.

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 [MBM+16]

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 currently 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[1]. 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

[1]https://en.wikipedia.org/wiki/Engineering

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, the 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 to new approaches, such as the one called industry 4.0, which aims to build the smart industry.

3.1.1.1 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[2]. There are various applications in each branch, and humans discovered and applied more and more applications by combining engineering fields.

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

3.1.1.2 Applications of Artificial Intelligence in Engineering

As time goes, the industry continues to be revolutionized, 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 to 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

[2]https://en.wikipedia.org/wiki/Engineering

building intelligent systems that are human-aware and trustworthy[Hor14]. 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 the monitor, maintain, optimize, automate these four areas[Ham17].

In [RA12], 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. This field is now seeing some practical, real-world successes[Hor14].

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[LYZ05]. 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 intelligent systems with human-like performance, 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 cumulatively expected reward[KBP13b].

As [Ham17] 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[KLM96]. RL algorithms can provide solutions to very large-scale optimal control problems. It has achieved many successful applications in engineering[Mat97, CB96, Zha96, MMDG97**?** , KSK99].

In the following table, there are up-to-now engineering applications using RL. We would like to discuss three application examples which were done by [MMDG97**?** , KSK99], respectively in Optimize, Control, Monitor and Maintenance domains.

3.1.2.1 Building an optimize the product inventory system

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

3.1.2.2 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.

3.1.2.3 Building an intelligent plant monitoring and predictive maintenance system

People had the 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.

[3]Source:https://conferences.oreilly.com/artificial-intelligence/ai-ca-2017/public/schedule/detail/60500

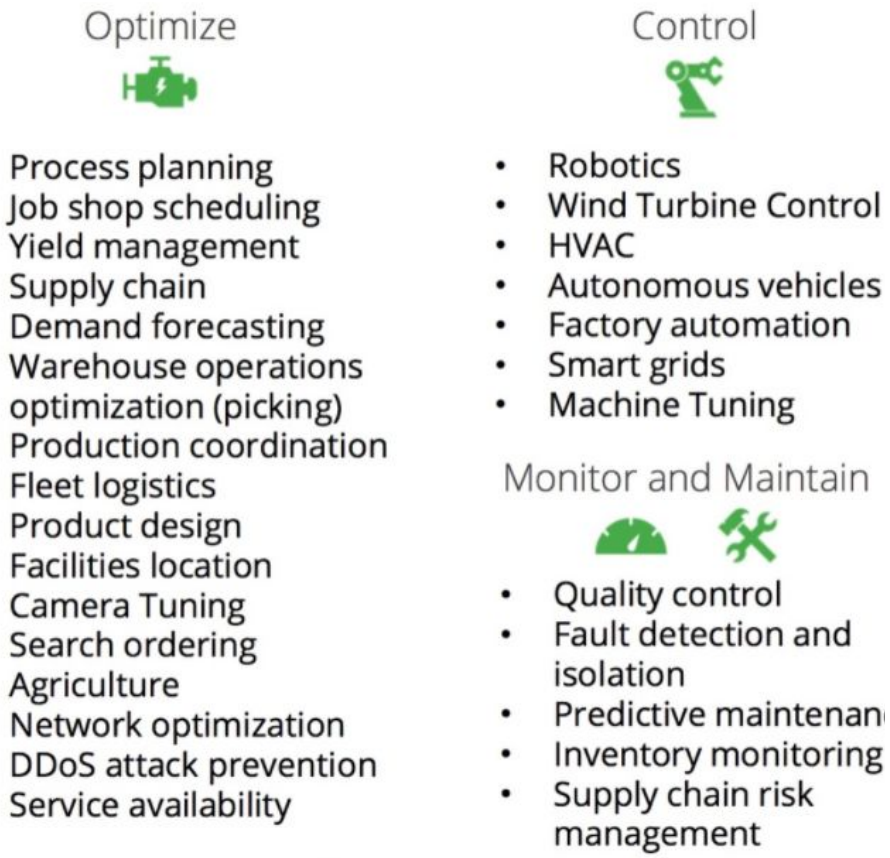

Figure 3.1: RL in engineering applications [3]

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, a RL model includes: Description, Observation, Actions, Rewards, Starting State, Episode Termination, Solved Requirements.

- Description: Simply describe the model, focus on the aim.
- Observation: Also called as State, generalizing and defining required information as input.
- Actions: Generalizing the possible actions and related limits as output.
- Rewards: The value of a state transition, reinforcement signal.

- Starting State: Given a start state, so the agent can start an episode.
- Episode Termination: Defining some conditions to end the iterations.
- Solved Requirements: Define conditions as the problem to be solved.

3.1.4 RL Life Cycle 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.

1. **Gather Knowledge**

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

2. **Visualize and define RL models**

   Visualize 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 the real world system, build the simulated environment for training.

4. **Choose RL method**

   Choose appropriate RL methods to implement.

5. **Perform training**

   Perform the 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 changes—heating up, mechanical wear, growing body parts, etc[KCC13]. However, we are still facing several challenges when applying RL to engineering applications. In [KBP13b], it listed several challenges apparent in the robotics settings. These challenges can be also be considered in general engineering applications.

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 have 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 implementation, but building RL model and experimental and error costs.

3.1.5.1 RL algorithms limits and high requirements of policies improvement

Classical reinforcement learning like Q-learning, Sarsa could deal with 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 uses of 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[KBP13b].

As the new development of Deep reinforcement learning, which combines 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.

3.1.5.2 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 is good at dealing with one task at any instant. This limits the scope of AI to achieve generality. To achieve AGI, the goal of RL to make the agent perform many different types of tasks, rather than specializing in just one. This can be achieved through multi-task learning and remembering the learning.

We've seen recent work from Google Deep Mind on multi-task learning, where the agent learns to recognize a digit and play Atari. However, this is really a very challenging task when you scale the process. It requires a lot of training time and a huge number of iterations to learn tasks.

Also, in many real-world tasks agents do not have the scope to observe the complete environment. These partial observations make the agent to take the best action not just from current observation, but 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.

Furthermore, the system dynamics or parameters may be unknown and subject to noise, so the controller must be robust and able to deal with uncertainty[ET16]. For such reasons, real-world samples are expensive in terms of time, labor and, potentially, finance.

3.1.5.3 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[ZLVC16]. 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[Ham17]:"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 to 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 robot's own body. Testing on operating production lines, industrial equipment, warehouses, etcetera is both expensive and disruptive, more than that, they are not plentiful opportunities fo this, we can't expect the real world to provide in a short span a diverse array of part or product failures. Third, the cost of failure and change is much expensive.

Another reason to use simulation environments for RL training is that they provide Safe Exploration Strategies[ET16]. RL agents always learn from exploration and exploitation. RL is a continuous trial-and-error based learning, where agent tries to apply a different combination of actions on a state to find the highest cumulative reward. The exploration becomes nearly impossible in the real world. Let us consider an example where you want to make the robot learn to navigate in a complex environment avoiding collisions. As the robot moves around the environment to learn, it'll explore new states and takes different actions to navigate. However, it is not feasible to take the best actions in the real world where the dynamics of the environment changes very frequently and becomes very expensive for the robot to learn[KBP13b].

The current trend is to study (simulated) 3D environments. For example, Microsoft's 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 software 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[KBP13b]. Ideally, the 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 hyper-parameters and get sufficient training data is challenging.

3.1.5.4 Reward function design

Reward function is the core of reinforcement learning, as it specifies and guides the desired behaviour. 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 the corresponding application. The first question is, how to give a quantitative number of a 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[KBP13b]. [MLLFP06]proposed a methodology for designing reward functions that take advantage of implicit domain knowledge.

A design of reward function could also affect the speed of convergence. If the reward is too fast to meet task achievement, the controller may fall into a local optimal. The opposite way: the controller may slow down the learning process, the worst will never reach the task achievement.

3.1.5.5 Professional knowledge requirements

In [Ham17], 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.

3.2 Challenges in Deep RL for Engineering Applications

The development of Deep Reinforcement learning solves a part of challenges, such as traditional RL limits. RL now could deal with continuous state and action space without discretization. Also, Curse of Dimensionality[KBP13b] is no longer a big problem. But there are still some newly appeared challenges we need to consider. We would've 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 use 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 to continually under research. The cases about applying DRL in engineering or related works of literature are not plentiful.

3.2.2 The need for 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 life cycle 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 the next chapter, we propose our research questions in first, then discuss the implementation requirements. The implementation requirements include environments and experimental setting.

Domains	**RL Engineering applications**
Optimize	Reinforcement Learning Approaches to Biological Manufacturing Systems[UHFV00], Distributed reinforcement learning control for batch sequencing and sizing in just-in-time manufacturing systems[HP04], Application of reinforcement learning for agent-based production scheduling[WU05], Dynamic job-shop scheduling using reinforcement learning agents[AÖ00], Self-improving factory simulation using continuous-time average-reward reinforcement learning[MMDG97], Inventory management in supply chains: a reinforcement learning approach[GP02], The application of Reinforcement learning in Optimization of Planting Strategy for Large Scale Crop production[OWC04], Personalized Attention-Aware Exposure Control using Reinforcement Learning[YWV+18]; etc.
Control	A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning[BC95], Reinforcement learning for humanoid robotics[PVS03], Autonomous inverted helicopter flight via reinforcement learning[NCD+06], Reinforcement learning in multi-robot domain[Mat97], Strategy learning for autonomous agents in smart grid markets[RV11], Using smart devices for system-level management and control in the smart grid: A reinforcement learning framework[KBKK12], Deep Reinforcement learning for building HAVC control[WWZ17], Learning and tuning fuzzy logic controllers through reinforcements[BK92], Adaptive PID controller based on reinforcement learning for wind turbine control[SR08], Reinforcement learning applied to linear quadratic regulation[Bra93], Neural networks and reinforcement learning in control of water systems[]; etc.
Monitor and Maintain	Intelligent monitoring and maintenance of power plants[KSK99], A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning[FLJ+10], Using reinforcement learning for pro-active network fault management[HS00], Reinforcement learning of Normative monitoring intensities[LMFL15]; etc.

Table 3.1: Reinforcement learning for engineering applications

4. Prototypical implementation and research questions

In this chapter we establish our research questions and disclose all relevant aspects about our prototype implementation. The chapter is organized as follows:

- We start by proposing our research questions (Section 4.1).
- We discuss the basic tools we used, namely the OpenGym AI framework and the Gym-Gazebo libraries (Section 4.2).
- We follow by describing the specific environments (Section 4.3) and the experimental configuration (Section 4.4) that we used for our evaluation.

4.1 Research Questions

In this section we distill the questions that conformed our research aim (Section 1.3), into more specific and measurable research questions that will guide our evaluation.

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 what are their specific influences for the tested methods? This research question is important to determine which factors require to be reported when disclosing information about agents in benchmarking results.
2. With adjusted hyper-parameters for a guaranteed high performance, what are the factors that need to be benchmarked for different methods over a specific environment? How do the tested methods differ with regards to these factors?

For the first question we organize the possible hyper-parameters as follows:

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

For the second question we propose to consider the following factors:

- Execution time
- Sample efficiency
- Highest score achieved
- Average score
- Robustness (i.e, how the learned model oscillates around the optimal scores through tests)

4.2 Tools

OpenAI Gym
OpenAI was founded in 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 exist several toolkits for RL environments design, among them we can name a few like: the Arcade Learning Environment, Roboschool, DeepMind Lab, the DeepMind Control Suite, and ELF.

OpenAI Gym is both an RL environment toolkit and a repository of standard environments which can be used in publications. In Gym there is a collection of environments designed for testing and developing reinforcement learning algorithms. Gym includes a growing collection of benchmark problems (i.e., environment configurations) that expose a common interface, and a website where people can share their results, through a public scoreboard and mechanisms to share the code of agents, and compare the performance of algorithms[BCP+16]. These scoreboards are available in the OpenAI Gym website[1].

[1]https://gym.openai.com

Gym saves the user from having to create complicated environments and it contributes to standardization such that researchers can work on a common set of assumptions. Gym is written in Python. Among its multiple environments it includes robot simulations and Atari games.

There are also many environments that are not 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 rendering tools.

OpenAI Gym does not include an agent class or specify what interface the agent should use, it just includes an agent for demonstration purposes[BCP+16]. For example, in Figure 4.1 it is shown how a user-side agent can be programmed to interact with a Gym environment by explicitly calling reset and step functions.

```
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 [BCP+16]

Gym-Gazebo[2]

For sophisticated engineering applications like robotics, reinforcement learning methods are hard to train by using real physical systems, due to the high cost of errors. Therefore emulating the sophisticated behaviors virtual and later applied in a real scenario becomes the alternative solution. Gazebo simulator[3] is a 3D modeling and rendering tool, with ROS[4][QCG+09], 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, reduce time and speed up training of RL features for robotics.

A whitepaper[ZLVC16] 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

[2]https://github.com/erlerobot/gym-gazebo
[3]http://gazebosim.org/
[4]Robot Operating System: http://www.ros.org/

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(Figure 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 enough that people can create their own robots and environment simulation by using this extension package.

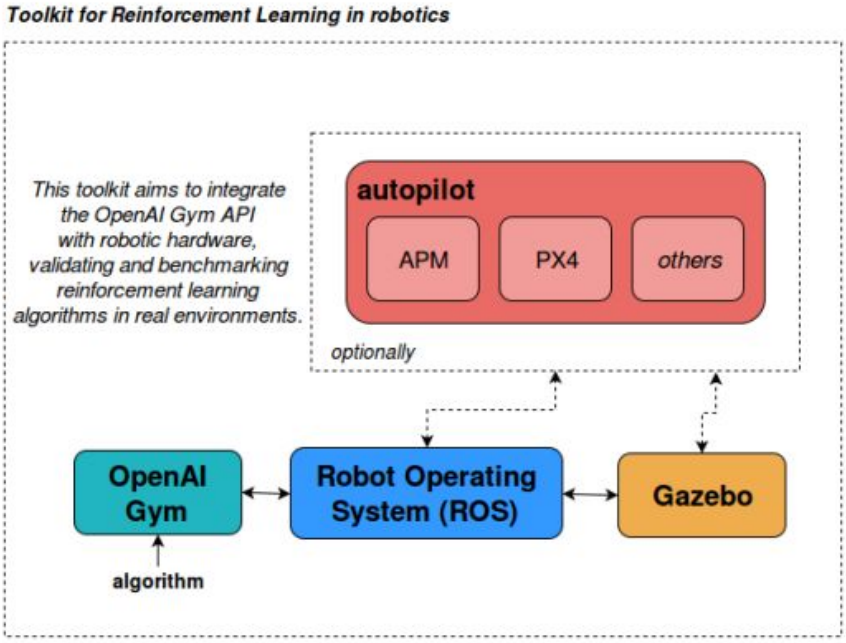

Figure 4.2: Toolkit for RL in robotics [ZLVC16]

TensorFlow and Keras[5]
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 Google's 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.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Teano. Keras is a Python Deep Learning library, it offers a simplified way to build your models in a declarative manner. 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. In this work we use Keras to build our agents' models, with TensorFlow serving as a backend for Keras.

- **Keras-RL**[6]
  Keras-RL 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**[7]
  Morvan-python is a personal and non-profit python, reinforcement learning open source library. It implements some state-of-the art deep reinforcement learning algorithms in Python with Tensorflow. In this work we use the provided DPPO agents as references.

[5]TensorFlow: https://www.tensorflow.org/; Keras: https://keras.io/
[6]https://keras-rl.readthedocs.io/en/latest/
[7]https://morvanzhou.github.io/

4.3 Environments

As we discussed in Chapter 3, due to the high cost of data collection and error, 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, as described in [NB18]:

- Determinism
- Observability
- Discrete or continuous
- Single or multiagent

The tasks considered by authors for benchmarking continuous control [DCH+16] can be divided into four categories: basic tasks, locomotion tasks, partially observable tasks, and hierarchical tasks.

In this paper, we used the following three environments which can be sorted into basic tasks and hierarchical tasks for experimentation and benchmarking suggestions: CartPole, PlaneBall, CirTurtleBot.

We considered these environments to be representative, since they contain low and high-dimensional observations, in addition to 2D and 3D models.

"CartPole" and "PlaneBall" which belong to basic tasks, model are 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 a path through a maze world. DQN works with an 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 "cart–pole" system[Kim99] is a classic benchmark for nonlinear control. It is a kind of mechanical dynamical system. *Mechanical dynamical systems* are easily understandable and thus can serve as illustrative examples. Although the "Cart-Pole swing-up" system is a simple system, it has several real-world applications.

The following are some engineering applications of the cart-pole system [Pat14]:

- The altitude control of a booster rocket during takeoff
- Wheel chairs and similar vehicles
- Ice-skating
- Helicopters and aeronautic balancing

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

In the discrete version, it has two discrete actions(push cart to left and right), a four-dimension observation space which contains the cart position, cart velocity, pole angle and pole angle velocity.Figure 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. The task is considered solved when the average reward is greater than or equal to 200 for in 10 consecutive trials. Episodes will terminated when:

1. The pole angle is more than $\pm 12^{\circ}$
2. The cart position is more than +/-2.4 (i.e., the center of the cart reaches the edge of the display).
3. The episode length is greater than 200.

In the continuous version, it has a one dimensional action space with a limit[-1,1], which means the force with a given direction that is applied to the cart. The remaining parameters are the same as those of the discrete version.

[8]Source: gym.openai.com
[9]Source: gym.openai.com

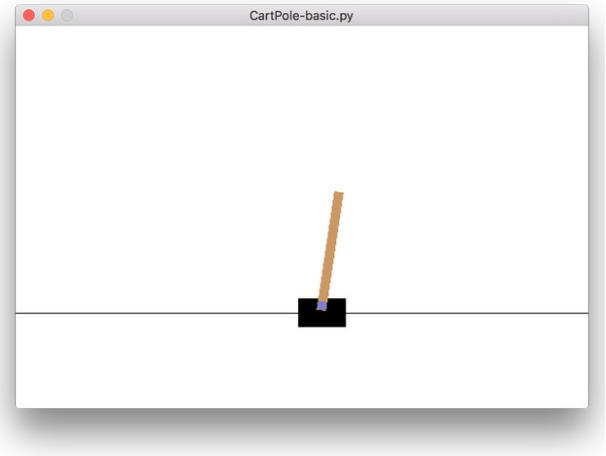

Figure 4.3: CartPole-v0 [8]

4.3.2 PlaneBall

"PlaneBall" consists of a ball that rolls on a plane which can rotate in both the 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. We followed the same baseline structure displayed by researchers in the OpenAI Gym and Gym-Gazebo package, and built a gazebo environment of "PlaneBall" on top of that (Figure 4.5).

In the continuous version the setting was configured to have a 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.

The task is considered solved when the episode reward is greater or equal to 800 holding in 10 consecutive trials. The episode will terminated when:

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

In the discrete version, the setting has 81 discrete actions. For this we discretized each dimension with the limit [-2, 2] to [-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2], then used [0, 1, ...,

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 [9]

81] to replace the combination of two [-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2]. The remaining parameters are the same as those of the continuous version.

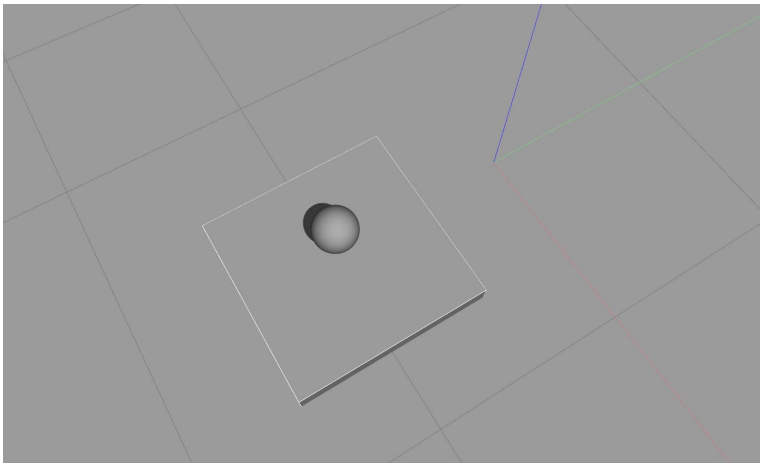

Figure 4.5: PlaneBall

4.3.3 CirTurtleBot

"CirTurtleBot" is a classic robot called Tutlebot(https://www.turtlebot.com/), which moves around a circuit, a 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 the robot without collision with the walls(Figure 4.6). This environment is simulated and controlled by Gazebo, ROS and Gym. It is available in the Gym-Gazebo package called "*GazeboCircuitTurtlebotLIDAR-v0*" (https://github.com/erlerobot/gym-gazebo).

In the discrete version it has three discrete actions (forward, left, right). And a five dimensional observation space, which all represent the range of the scan r in the range domain. The reward is defined as one for every time step the robot moves 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 the 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).

We randomly initialize the robot position at four corners of the maze, as a starting state. The task is considered solved when the episode reward is greater than or equal to 600 holding in 10 consecutive trials.

The episode will terminated when:

1. Each laser range is smaller than 0.2

2. Episode length is greater than 300

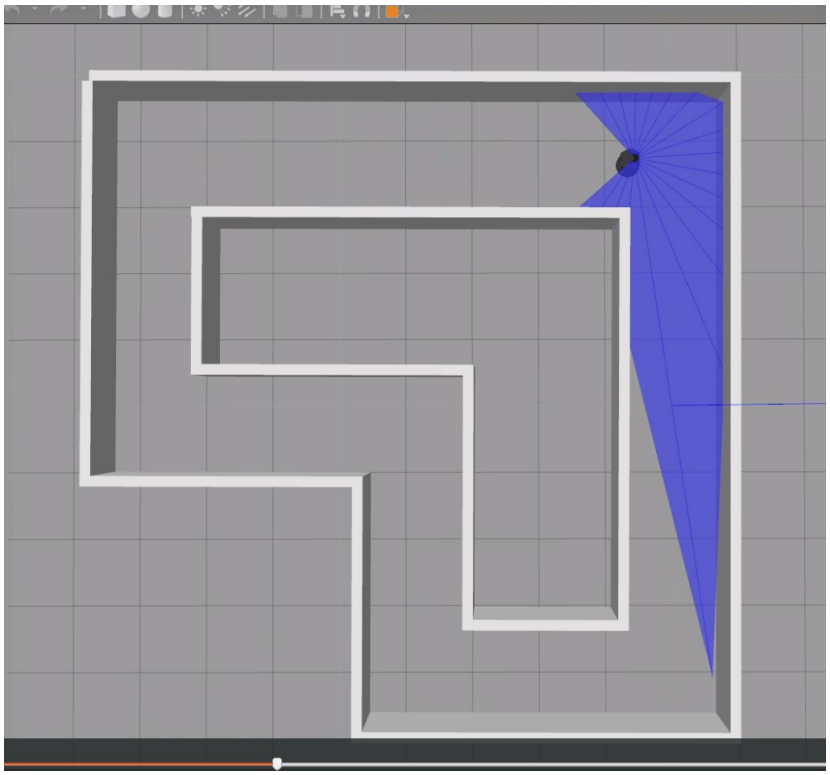

Figure 4.6: CirTurtleBot

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 core CPU (i7-8750H), 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 an *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 Chapter 2. We compared the two policies as a response to the first research question in our evaluation. For DDPG, we utilized a random process function called: Ornstein–Uhlenbeck process[Fin04]. For DPPO, we explored by taking a 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 fully connected hidden layers with 24 neurons and a 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 two research questions to guide our 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 disclosed.

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

5. Evaluation and results

In this chapter, we discuss the evaluation and results. We organize the chapter as follows:

- We start the chapter by presenting our series of tests designed to tackle the first research question (Section 5.1), testing the impact of parameters per method, for the learning process in the different environments.
- The next section in the chapter presents the results of our tests for the second research question (Section 5.2), comparing the different methods.

5.1 "One-by-One" performance Comparison

In this evaluation section, we focus on the first research question. We tune and evaluate the factors which might have a performance influence for a specific method, as applied to a specific environment. We evaluate the results in terms of sample complexity, value accuracy, policy quality.

We listed all the potentially influential factors of the three methods, in Figure 5.2, Figure 5.14, and Figure 5.21. In order to reduce a large number of comparisons of hyper-parameters tuning, we fixed some parameters which have the better performance according to some related works and tuned some specific parameters.

We report the performance of the implemented algorithms in terms of average return over all training iterations for five different random seeds (the same across all algorithms in all environments). We tested and used the smallest fully trained results of each "one-by-one" experiment.

We disclose the hyper-parameters in each method with a graph format (i.e, a diagram) and the fixed and tuned parameters of each "one-by-one" pair in table formats. The tuned parameters with red fonts represent the highest performance found, after comparing.

5.1.1 DQN method in the environments

In our evaluation we utilized a simple yet advanced DQN method, which is Double DQN[VHGS16] with a fixed Q target and experience replay mechanisms. There are several influential hyper-parameters in the DQN algorithm, which are shown in Figure 5.2. We used the Neural network structure as Figure 5.1 shows. The input and output layers are different regarding different environments.

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

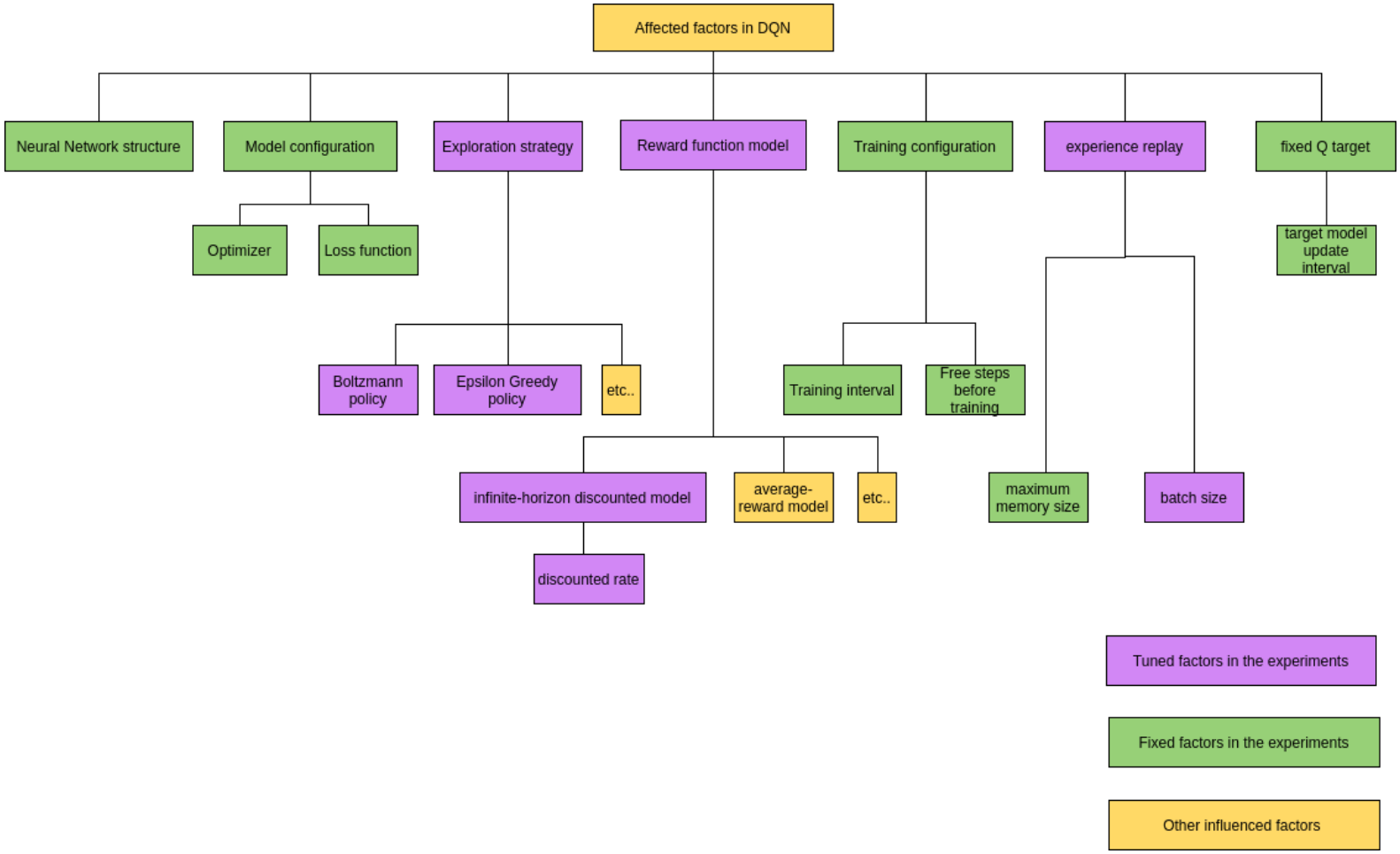

Figure 5.2: Hyper-parameters in DQN

- **DQN in "CartPole"**
  In this experiment, we used the suggested Neural Network structure(Figure 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 with regards to training time, performance. For the training compiling, we found the Adam optimizer with learning rate 0.01 can give a reasonable 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 fixed parameter values for the comparison. The batch sizes 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. The red line represents the Boltzmann policy with tau=1.0, clip=(-500, 500), discounted_rate=0.99. The green line represents Discounted Epsilon Greedy policy with minimum_epsilon=0.0001, decay=0.999, discounted_rate=0.99. The blue line represents Boltzmann policy with tau=1.0, clip=(-500, 500), discounted_rate=0.5.

As we can see the green line converges and reach 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 converge. In conclusion, the agent with fixed parameters(Table 5.7) and exploration strategy=Boltzmann, discounted_rate=0.99 has the highest performance.

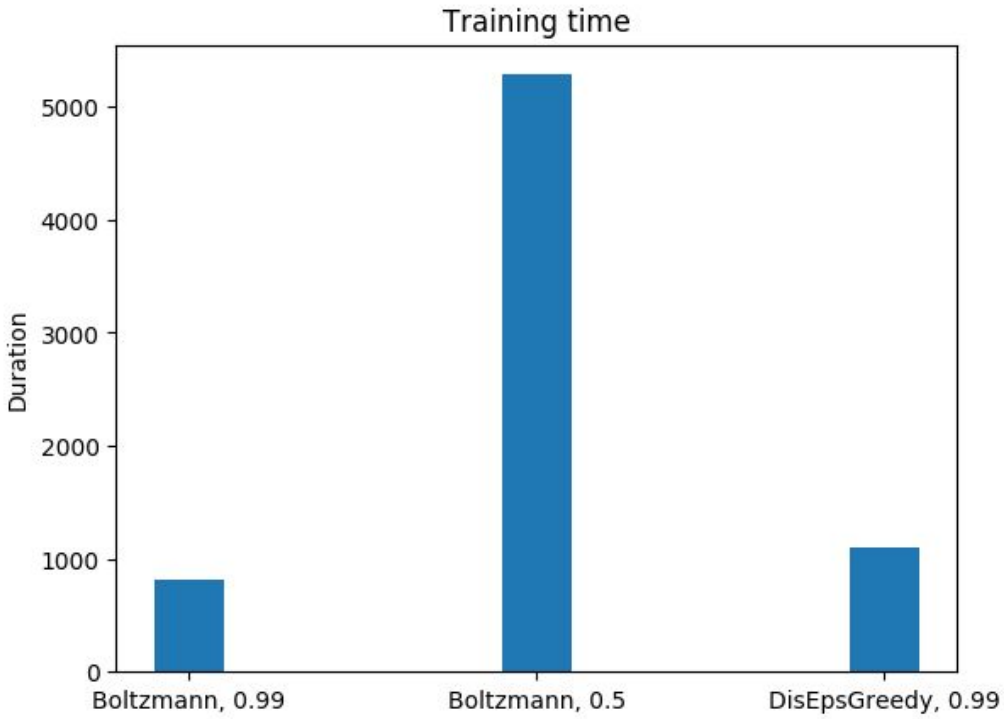

Figure 5.3: DQN-CartPole: Training time

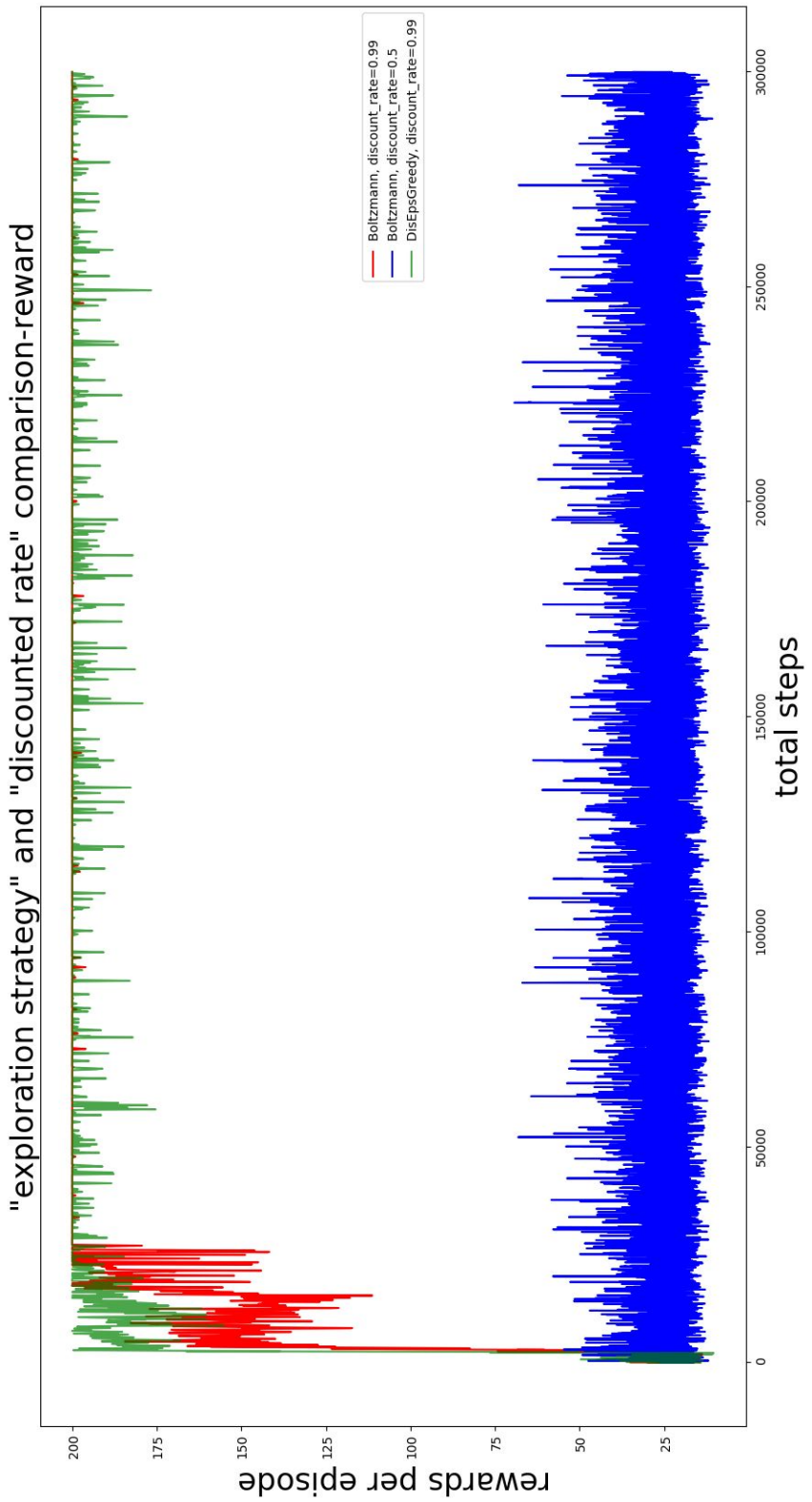

Figure 5.4: DQN-CartPole: Exploration strategy and discounted rate comparison

- **DQN in "PlaneBall"**

  In this experiment we used the suggested Neural Network structure Figure 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 such as "reward per episode", according to total training steps as well as the training time. As we discussed in Chapter 4, the task to be mastered in the "PlaneBall" environment is to hold in the middle area of the plane as long as possible.

  Table 5.3 lists the fixed parameters of DQN in "PlaneBall", which we experimentally observed to give a reasonable performance. Table 5.4 lists 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 a soft update with an 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 full 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 a 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 shows, 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 the algorithm with target_net update interval=0.1 reached the high score faster than the other two. After 2000 episodes training episodes, we use 500 episodes for testing. The 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, the algorithm with target_net update interval=0.01 could hold the maximum steps.

With a fixed target_net update interval=0.01, we compared the algorithms with two different learning rates: 0.001 and 0.01. As Figure 5.8 shows, 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 by learning rate=0.01 reached the high score fast than another. After 2000 episodes training episodes, we use 500 episodes for testing. The algorithm with learning rate=0.01 presented the higher average reward than the algorithm with learning rate=0.001. After training steps, the algorithm with learning rate=0.01 could hold the maximum steps.

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

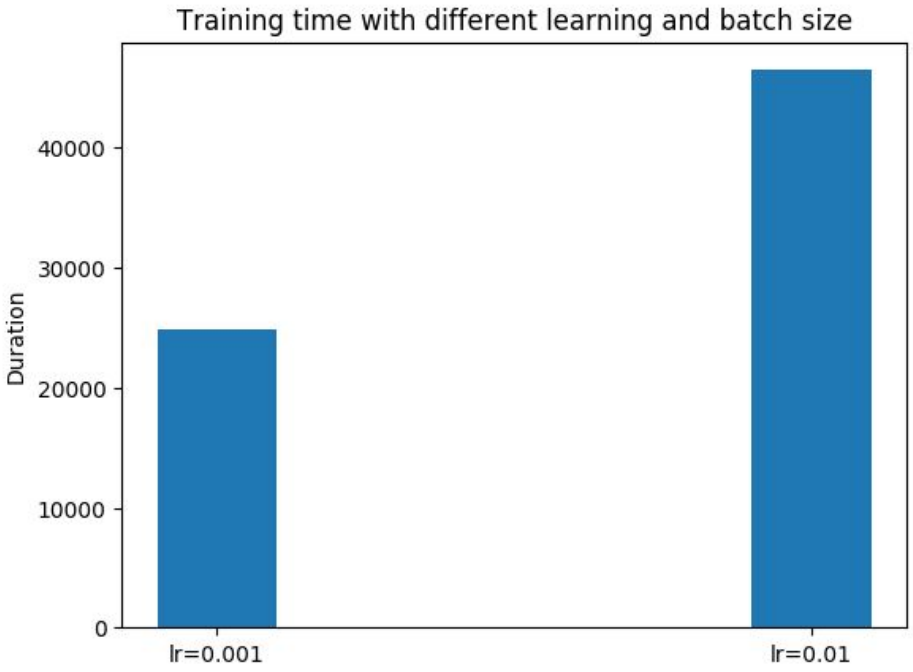

Figure 5.5: DQN-PlaneBall: Training time of learning rate comparison

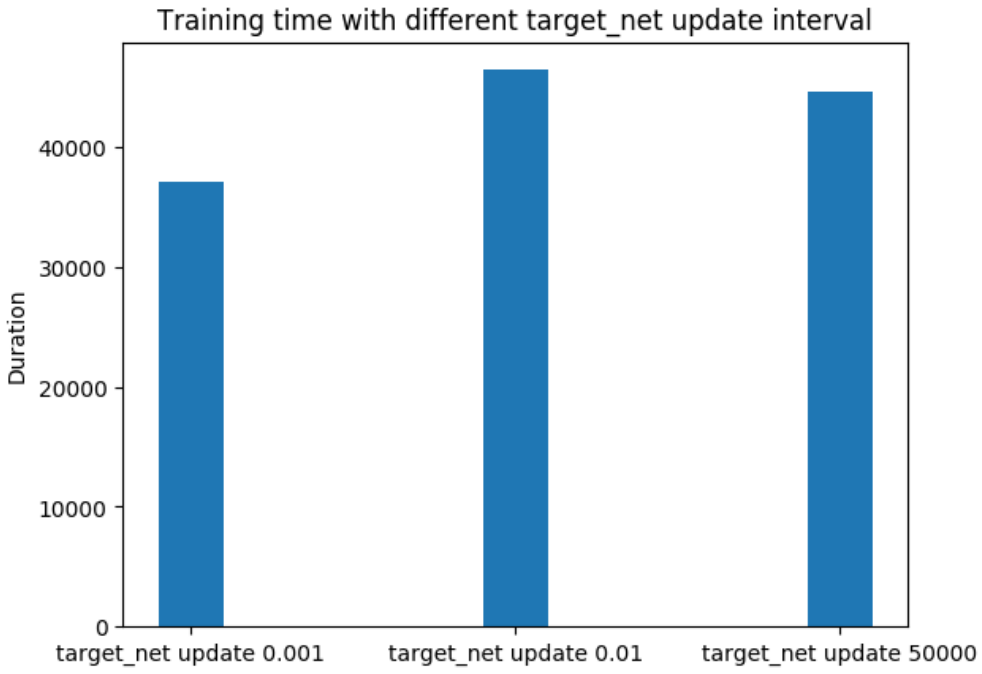

Figure 5.6: DQN-PlaneBall: Training time of target net update comparison

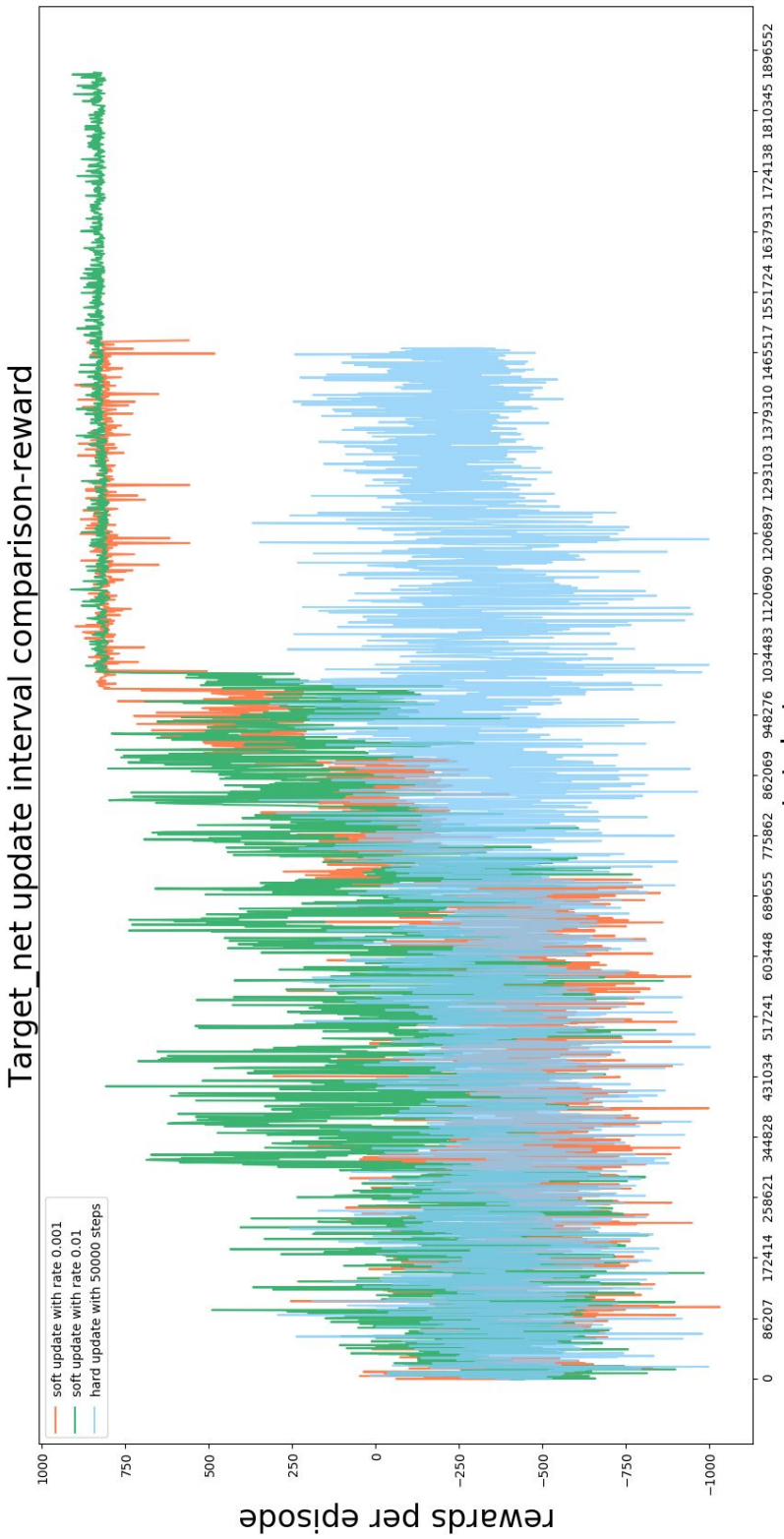

Figure 5.7: DQN-PlaneBall: target_net update comparison

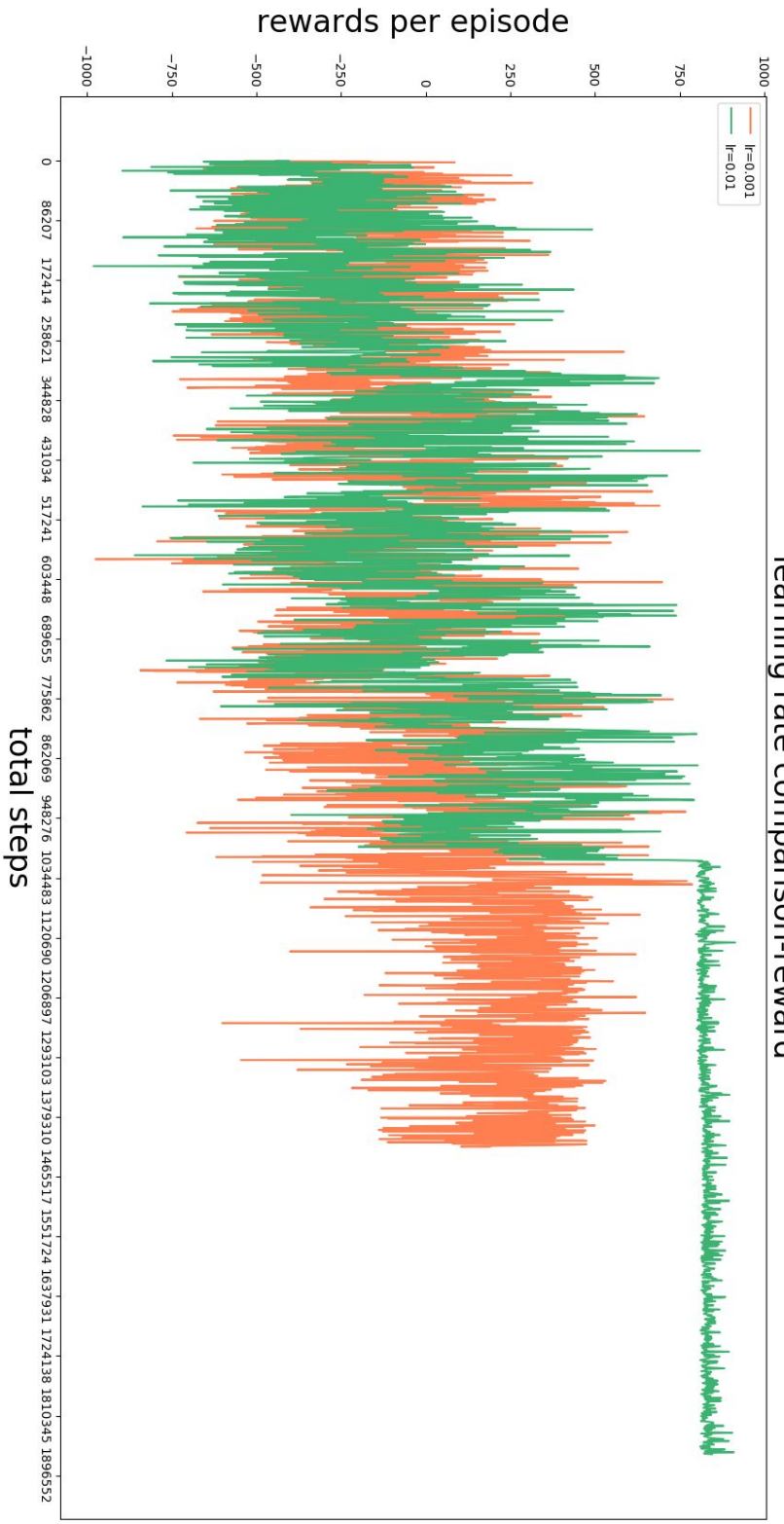

Figure 5.8: DQN-PlaneBall: learning rate comparison

- **DQN in "CirTurtleBot"**

  In this experiment, we used the suggested Neural Network structure Figure 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 Figure 5.7 shows. The loss function we used was the Huber Loss with formula: $0.5 * square(q\_target - q\_predict)$. We evaluated the performance with aspects such as "steps per episode" and "reward per episode" according to total training steps as well as the training time. Table 5.3 lists the fixed parameters of DQN in "CirTurtleBot", which we guaranteed that will give a better performance. Table 5.4 lists 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 full 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 is 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 blue line performs a 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. This is similar to the case for PlaneBall.

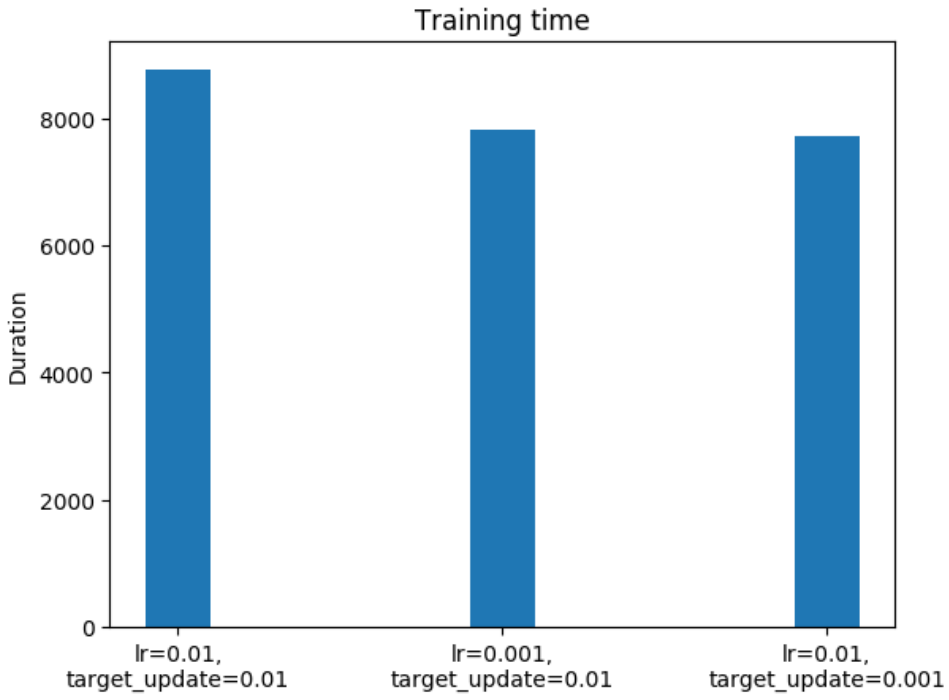

Figure 5.9: DQN-CirturtleBot: Training time

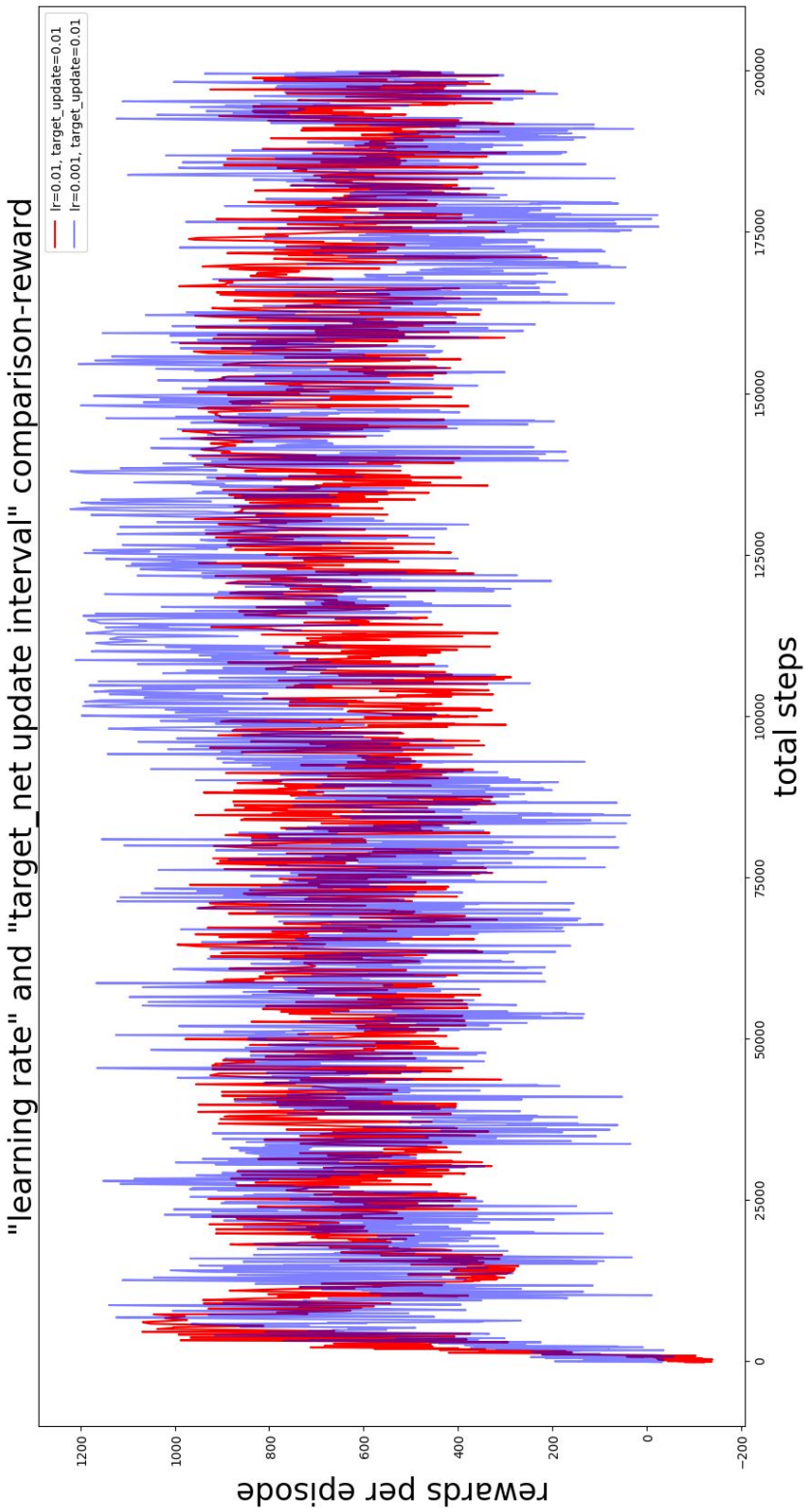

Figure 5.10: DQN-CirturtleBot: Exploration strategy and discounted rate comparison

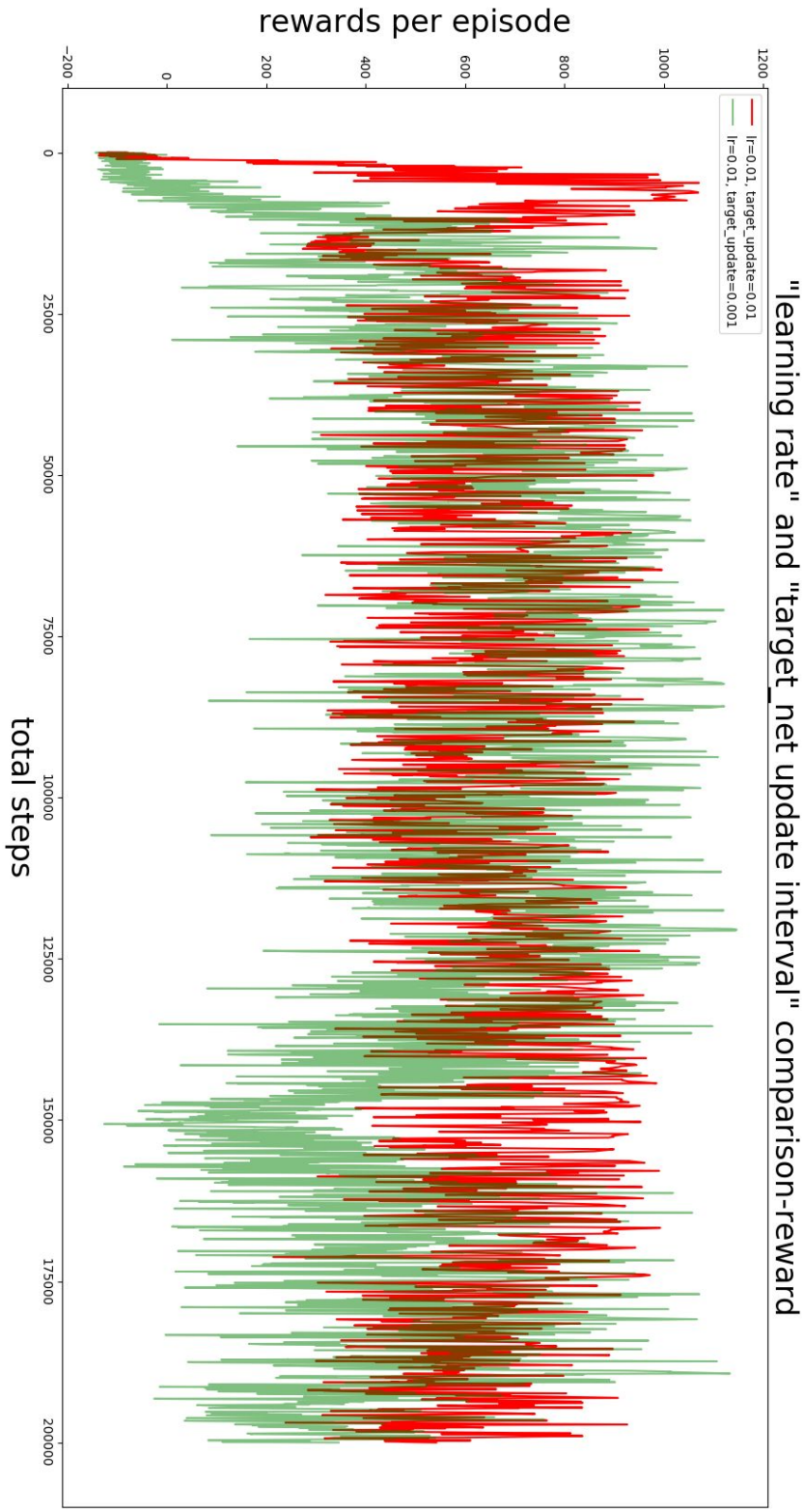

Figure 5.11: DQN-CirturtleBot: Exploration strategy and discounted rate comparison

5.1.2 DDPG method in the environments

In our evaluation, we evaluate the DDPG method for three environments, we tuned the hyperparameters to compare the performance. The hyper-parameters in the DDPG algorithm are presented in Figure 5.14. We used the actor-critic neural network architectures as Figure 5.12, Figure 5.13 show. The input and output layers are, naturally, different for different environments with different observation spaces.

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

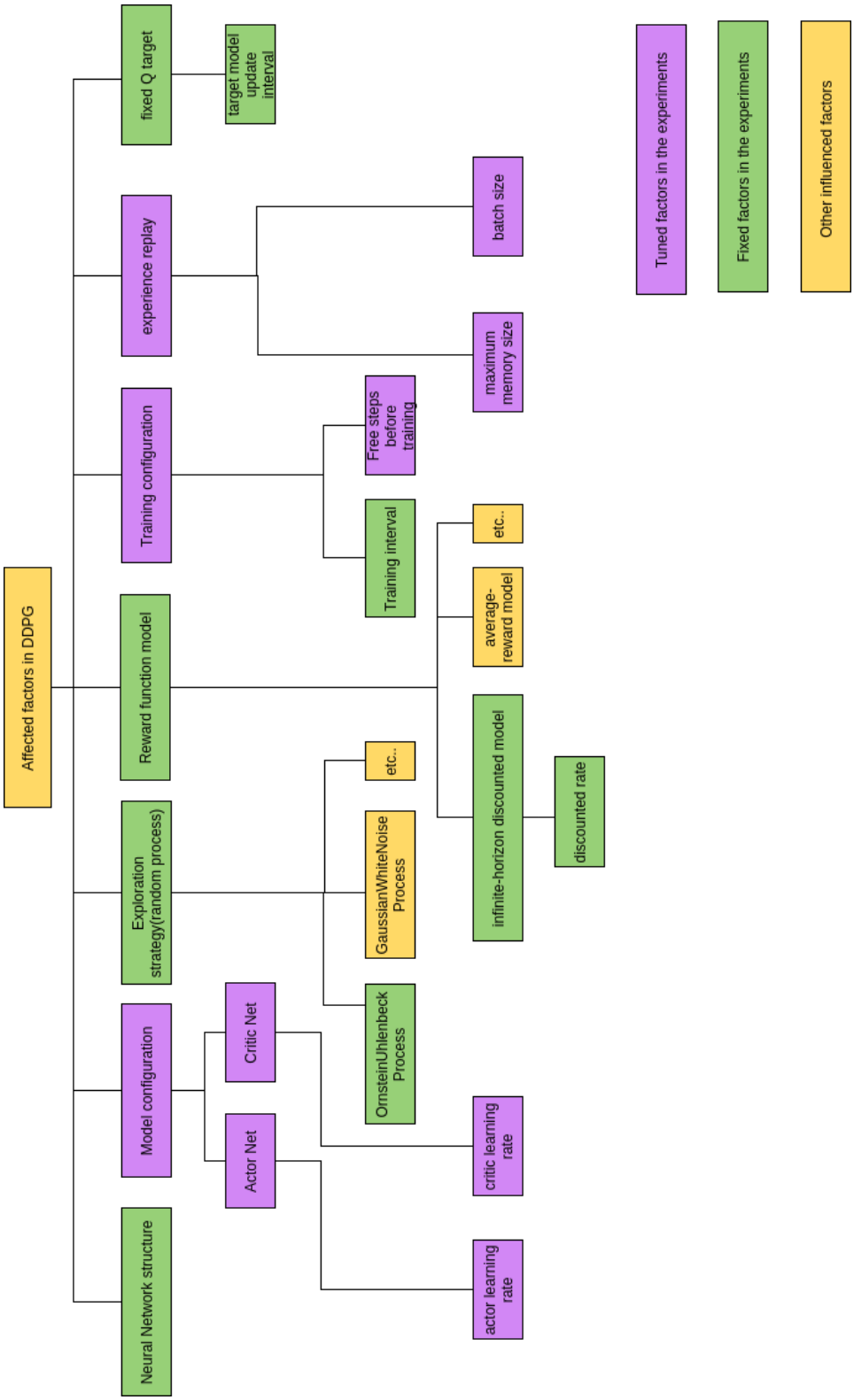

Figure 5.14: Hyper-parameters in DDPG

- **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 be performing the test during the remaining steps. In "CartPole", the termination condition is defined as holding 200 rewards in 10 episodes. Table 5.7 and Table 5.8 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in Figure 5.12 and Figure 5.13, with four-dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure 5.12, Figure 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 sizes, 10000, 50000 and two actor learning rate, 0.00001, 0.001. As we can see, the green line converges and reaches 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(Table 5.7) and memory_size=10000, actor_lr=0.00001 has the highest performance.

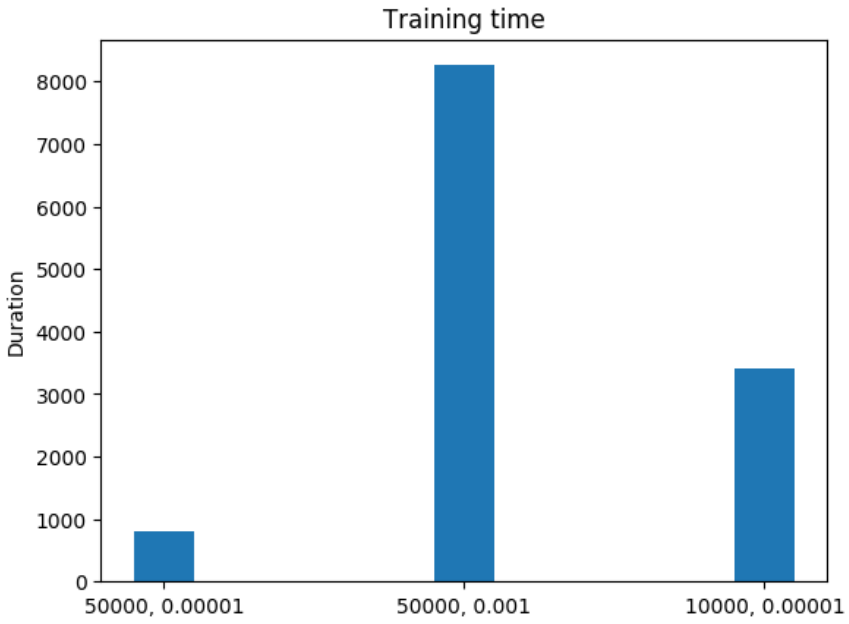

Figure 5.15: DDPG-CartPole: Training time comparison

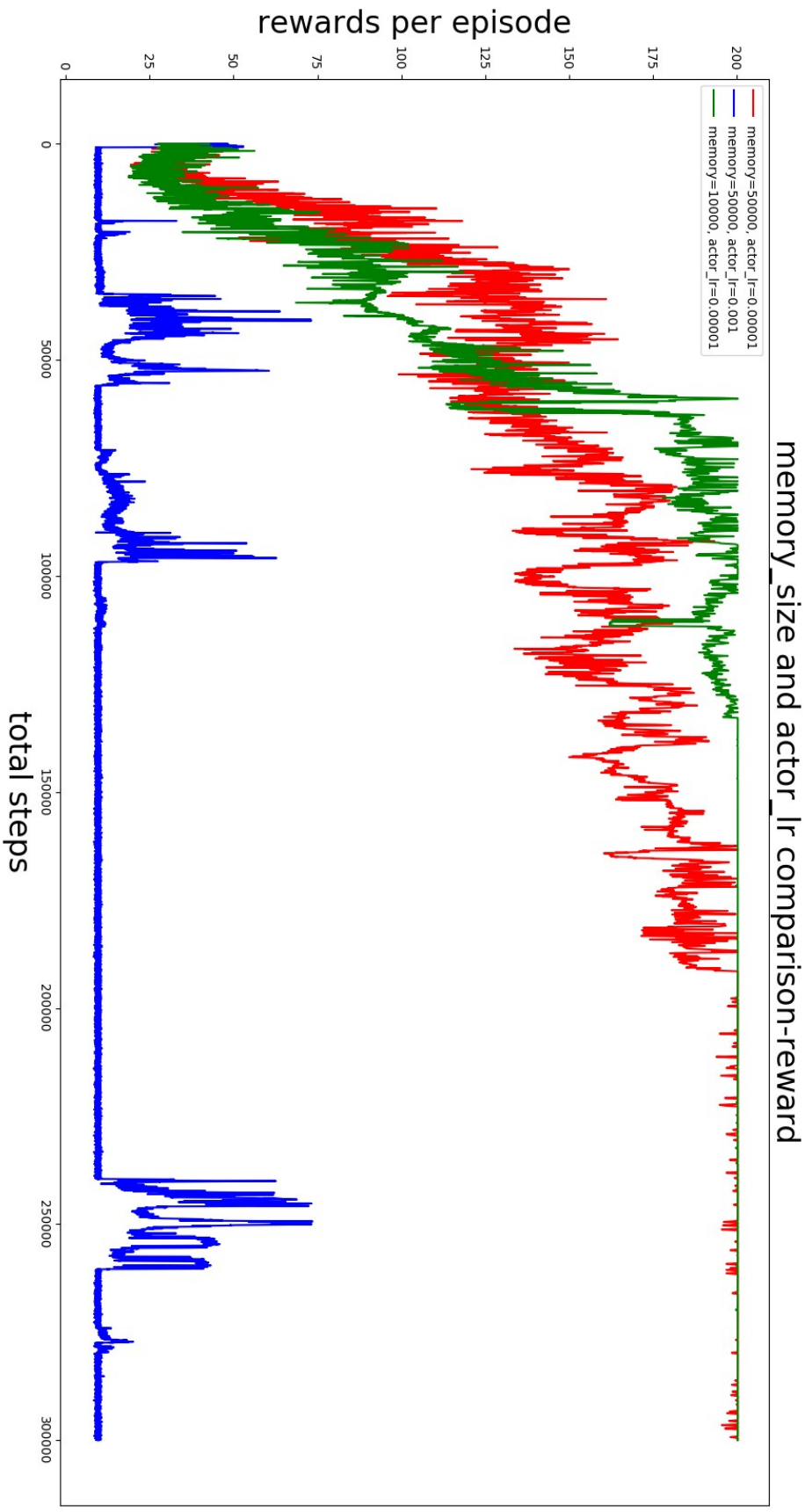

Figure 5.16: DDPG-CartPole: memory size and actor learning rate comparison

- **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 be performing the test during the remaining steps. In "PlaneBall", the termination condition is defined as holding 800 rewards in 10 episodes. Table 5.9 and Table 5.10 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in Figure 5.12 and Figure 5.13 with seven dimensional observation space and two dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure 5.12 Figure 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 converges and reach the termination faster, but the reward in the 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(Table 5.9) and actor_lr=0.00001, target_net_update=0.001 has the best performance.

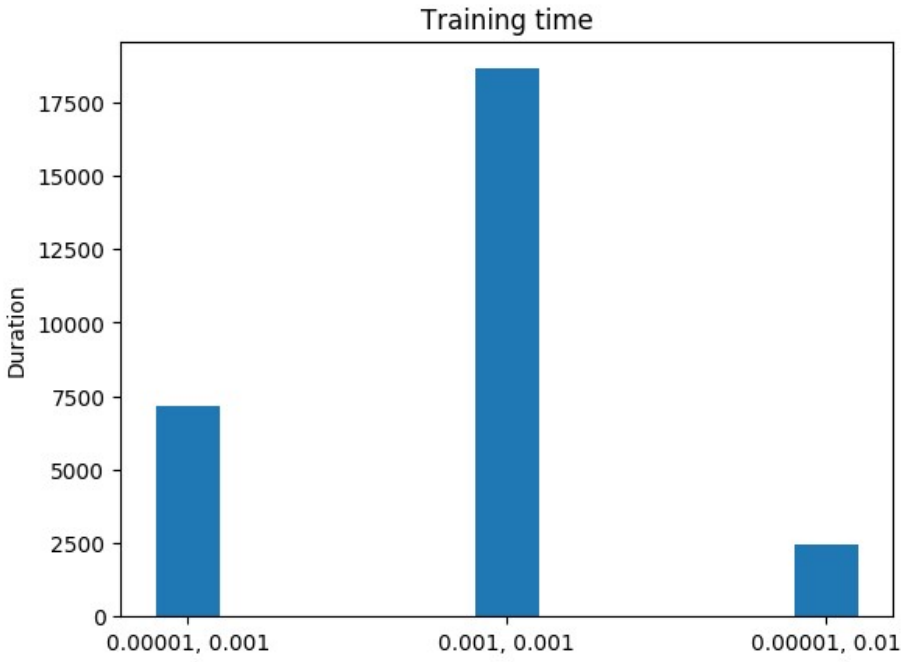

Figure 5.17: DDPG-PlaneBall: Training time comparison

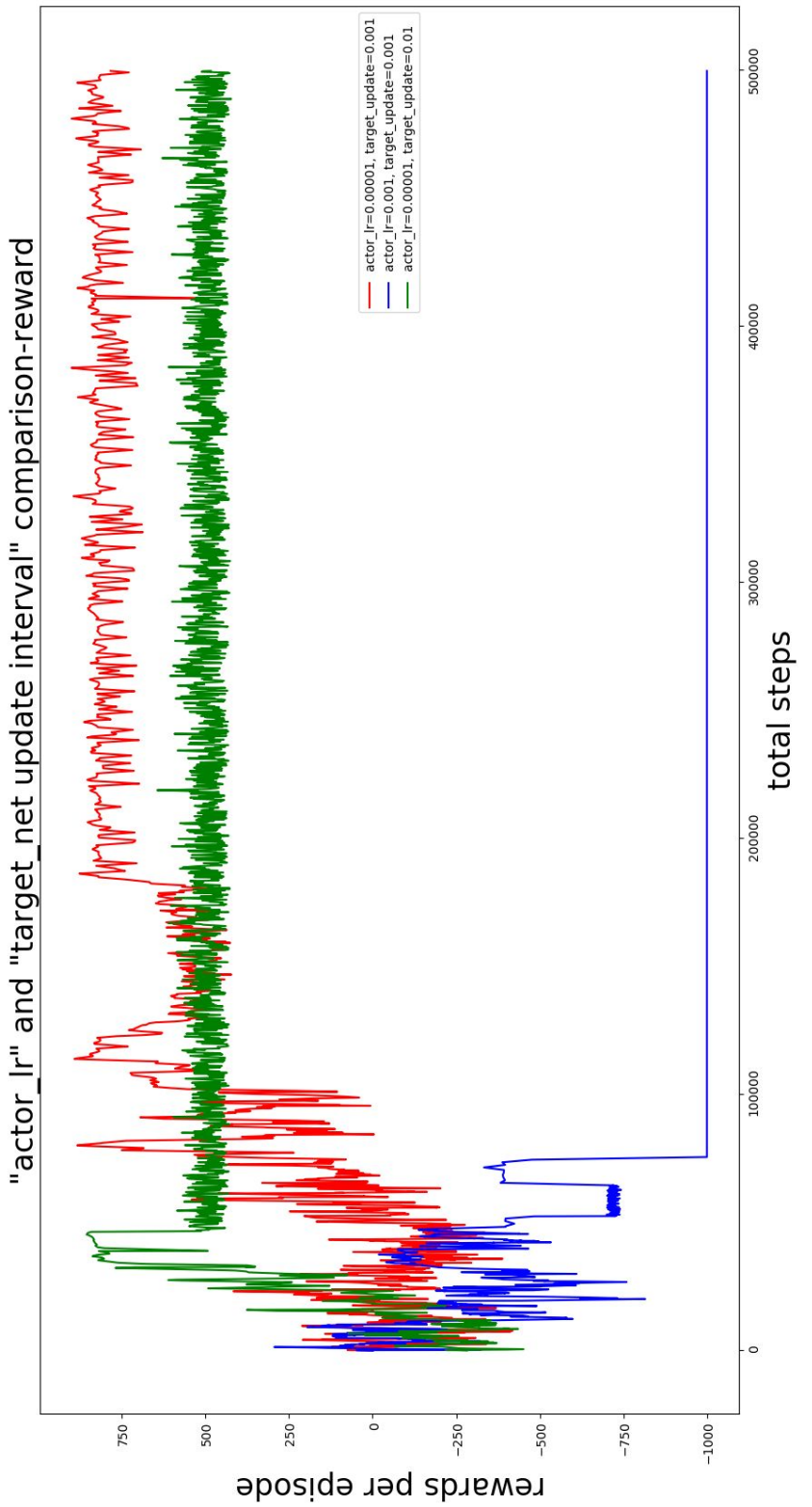

Figure 5.18: DDPG-PlaneBall: actor learning rate and target net update interval comparison

- **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 be performing the test during the rest steps. In "PlaneBall", the termination condition is defined as holding 600 rewards in 10 episodes. Table 5.11 and Table 5.12 describe the fixed parameters and tuned parameters respectively. We used the actor-critic neural network architectures in Figure 5.12 and Figure 5.13 with 20 dimensional observation space and one dimensional action space.

Fixed parameters	**Value**
Neural Network structure	Figure 5.12 Figure 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 at 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(Table 5.11) and actor_lr=0.0001, critic_lr=0.001 has a higher performance.

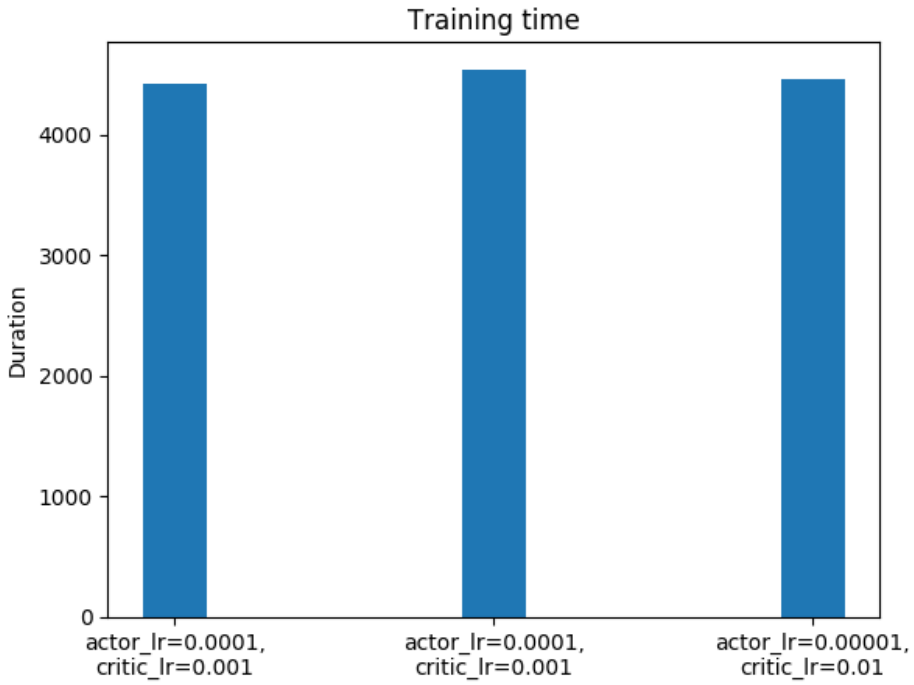

Figure 5.19: DDPG-CirTurtleBot: Training time comparison

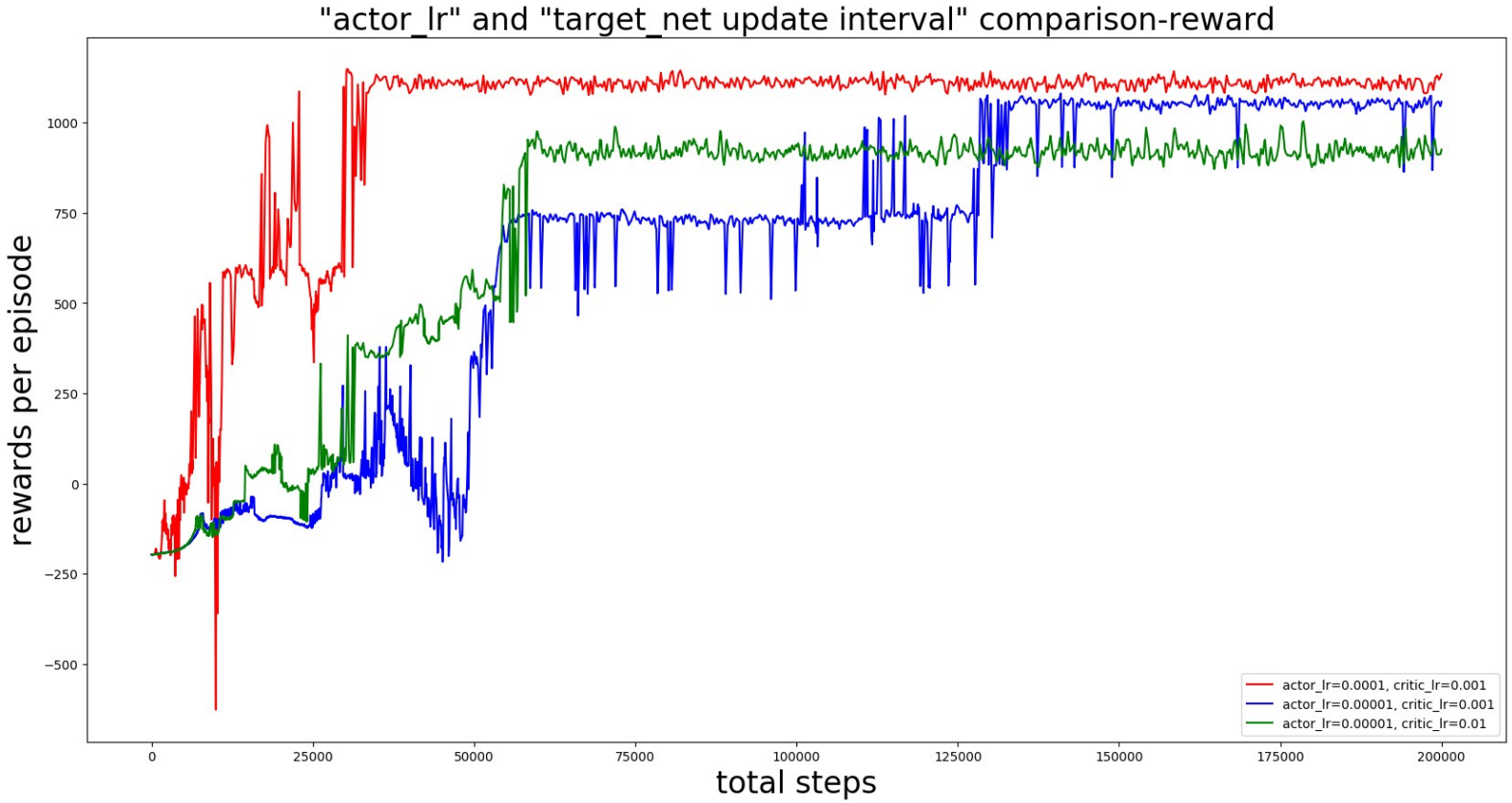

Figure 5.20: DDPG-CirTurtleBot: actor learning rate and target net update interval comparison

5.1.3 PPO method in the environments

In this evaluation we evaluate the Distributed PPO method for three environments, we tuned the hyperparameters to compare the performance. The hyper-parameters in the DPPO algorithm are presented in Figure 5.21. For this method, we combined both PPO and A3C, where we optimize the "clipped surrogate objective" using a number of samples collected by several parallel workers. There is a global PPO brain used for updating, each worker (agent) run in its 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, Figure 5.13 show. The input and output layers are different regarding different environments.

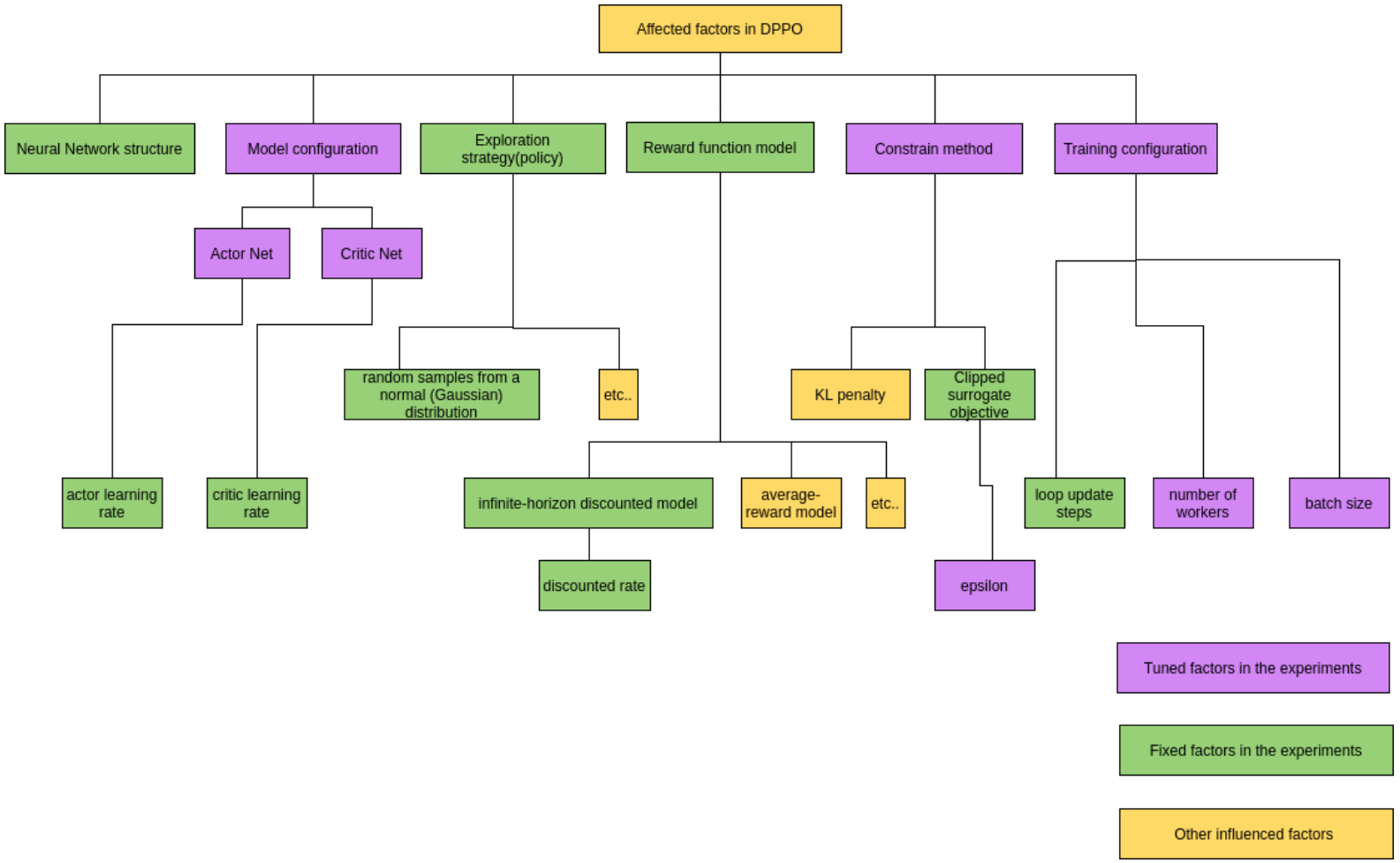

Figure 5.21: Hyper-parameters in DPPO

- **PPO in "CartPole"**

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

Fixed parameters	**Value**
Neural Network structure	Figure 5.12, Figure 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 sizes, 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 are 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 held the highest score earlier than the blue one. The red line also coverages fast, but the score is 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(Table 5.13) and batch_size=32, epsilon=0.8 has a high performance.

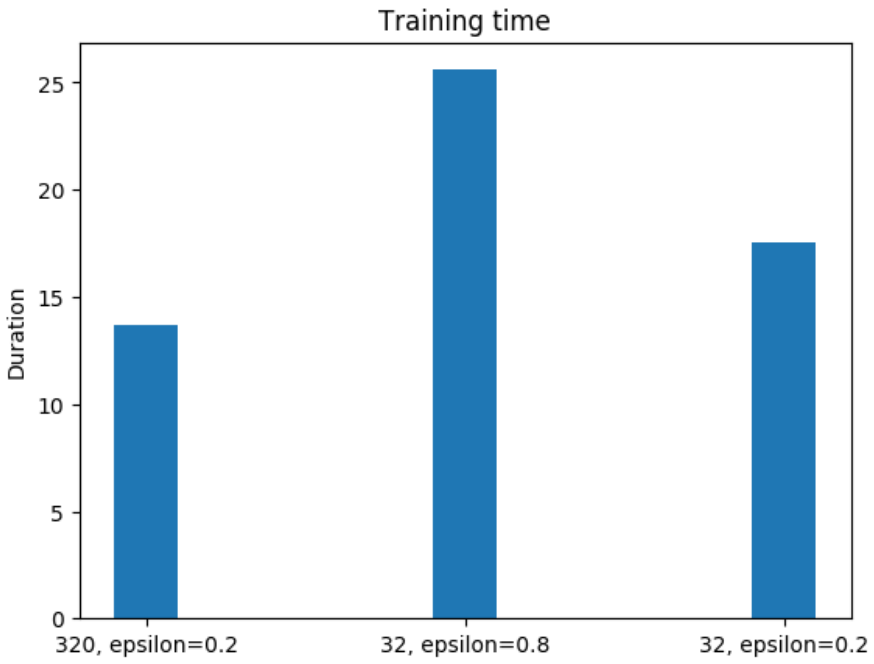

Figure 5.22: DPPO-CartPole: Training time comparison

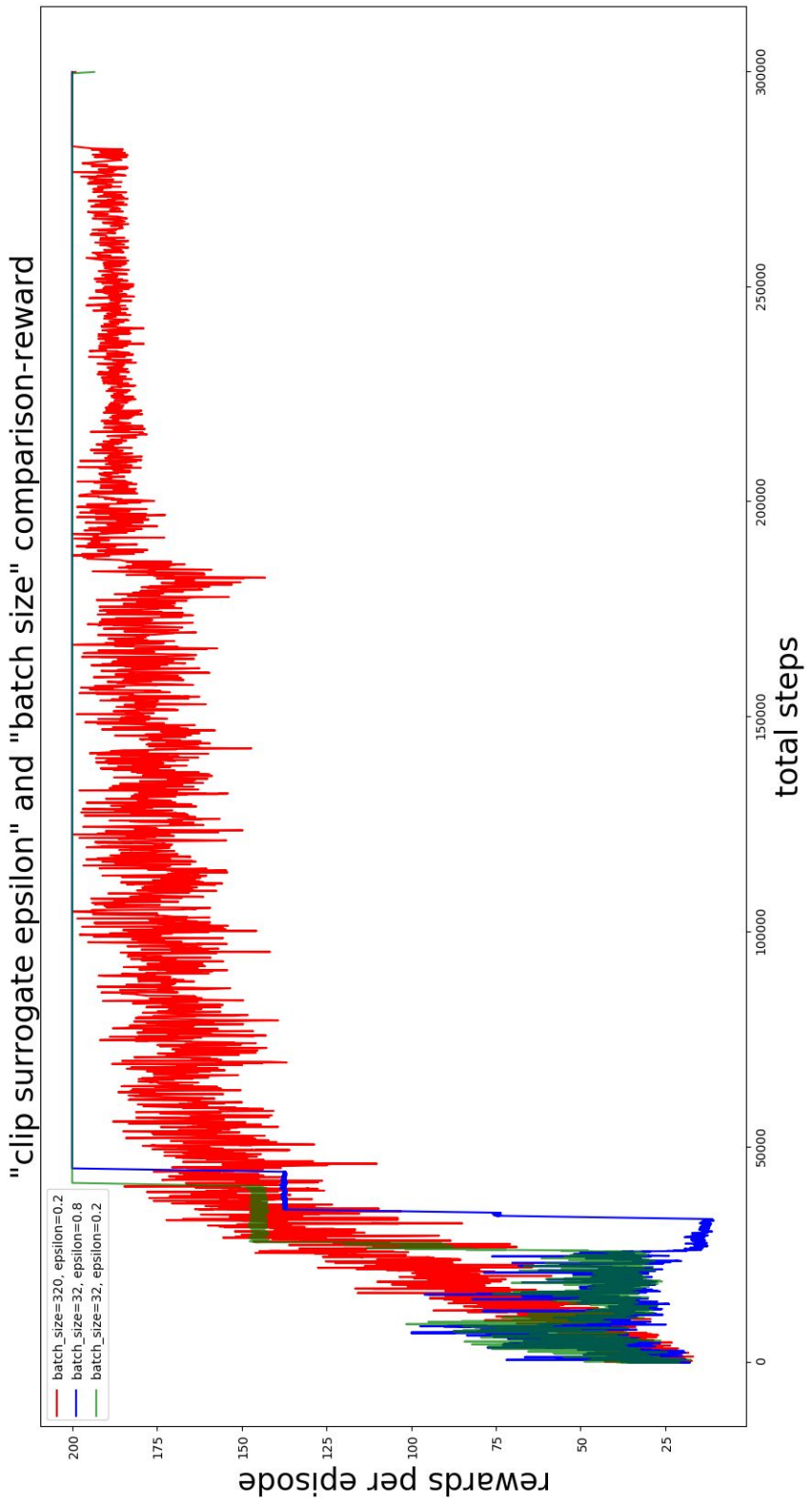

Figure 5.23: DPPO-CartPole: batch size and clipped surrogate epsilon comparison

- **PPO in "PlaneBall"**

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

Fixed parameters	**Value**
Neural Network structure	Figure 5.12, Figure 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 the 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 the blue line and the 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 the two, the blue line shows the better performance than the red one regarding the convergence 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 the 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 the algorithm with batch_size = 200, and update_steps=(10,100). In conclusion, the agent with fixed parameters(Table 5.15) and batch_size = 200, update_steps=(10,100) has a the highest performance.

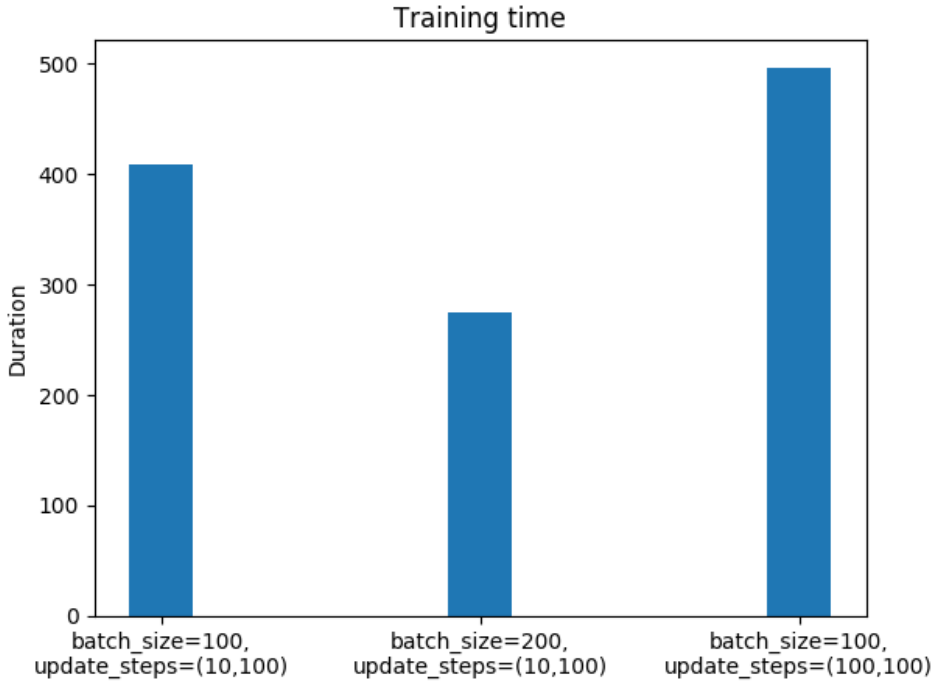

Figure 5.24: DPPO-PlaneBall: Training time comparison

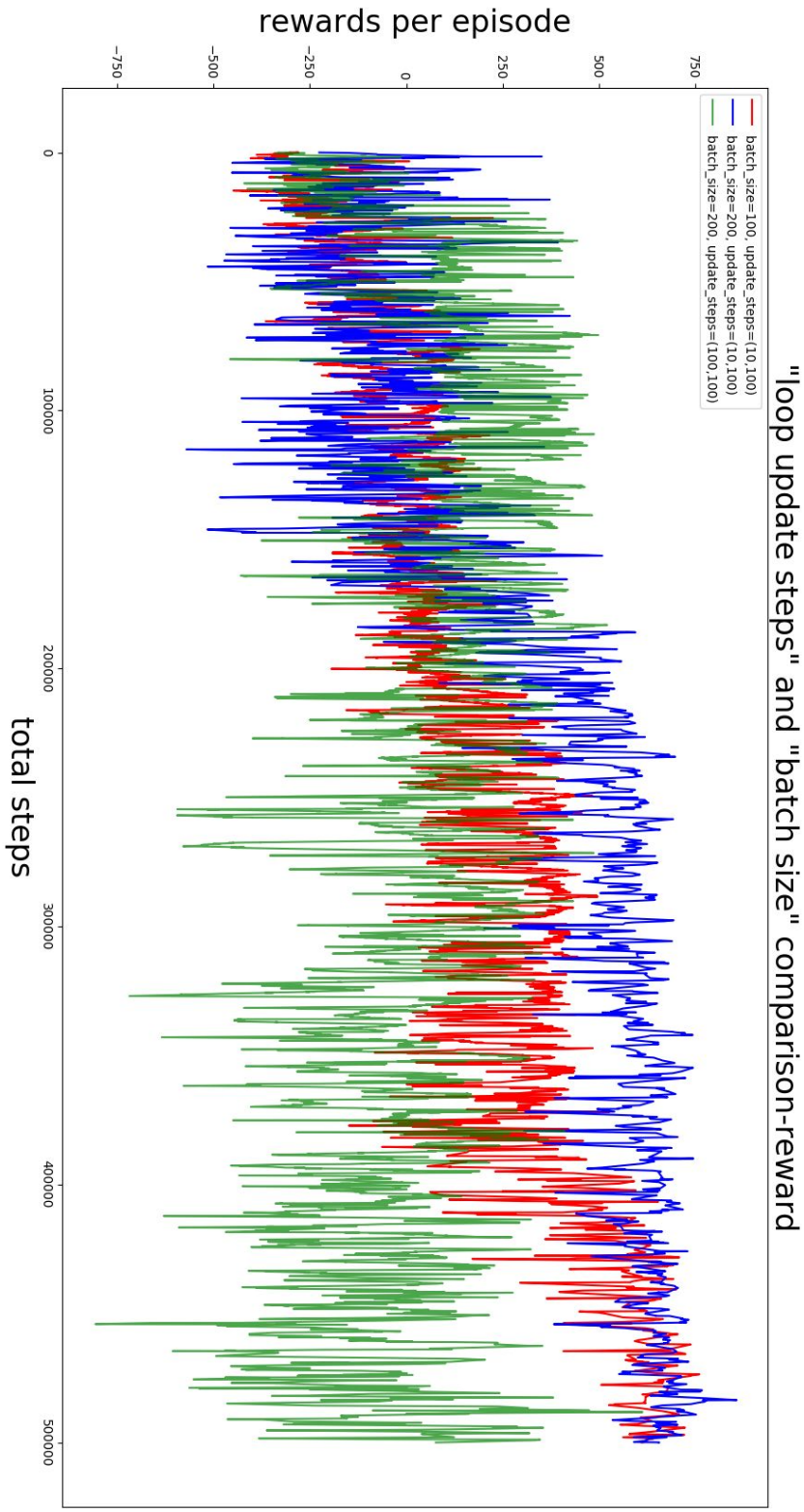

Figure 5.25: DPPO-PlaneBall: batch size and loop update steps comparison

- **PPO in "CirTurtleBot"**

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

Fixed parameters	**Value**
Neural Network structure	Figure 5.12, Figure 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 learn during the 200k steps. The red line converges 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 converges faster than the red one. In contrast, after coverage, the red line is more stable than the green one.

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

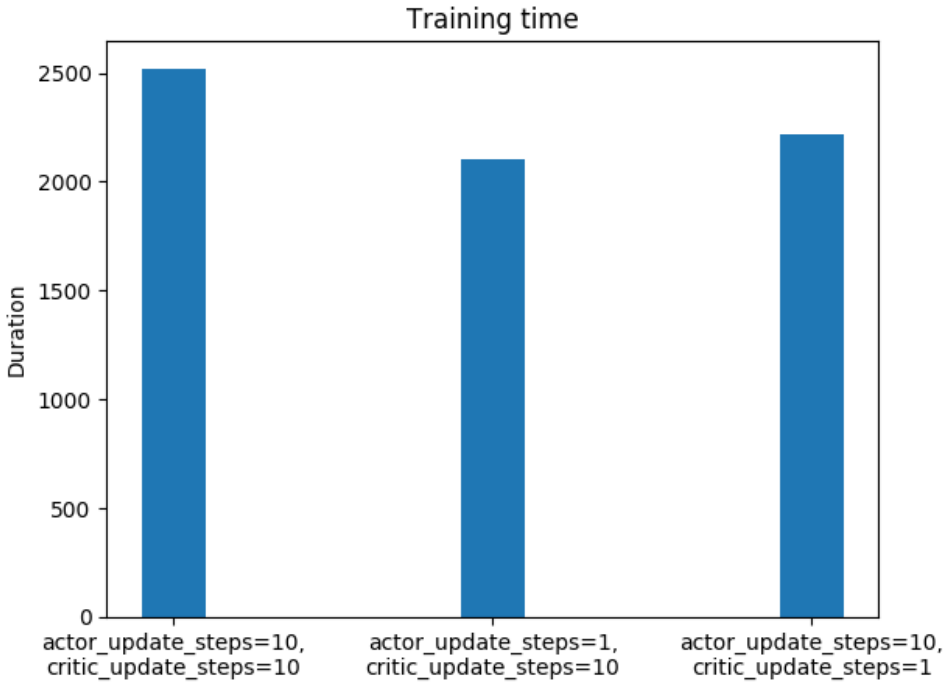

Figure 5.26: DPPO-CirTurtleBot: Training time comparison

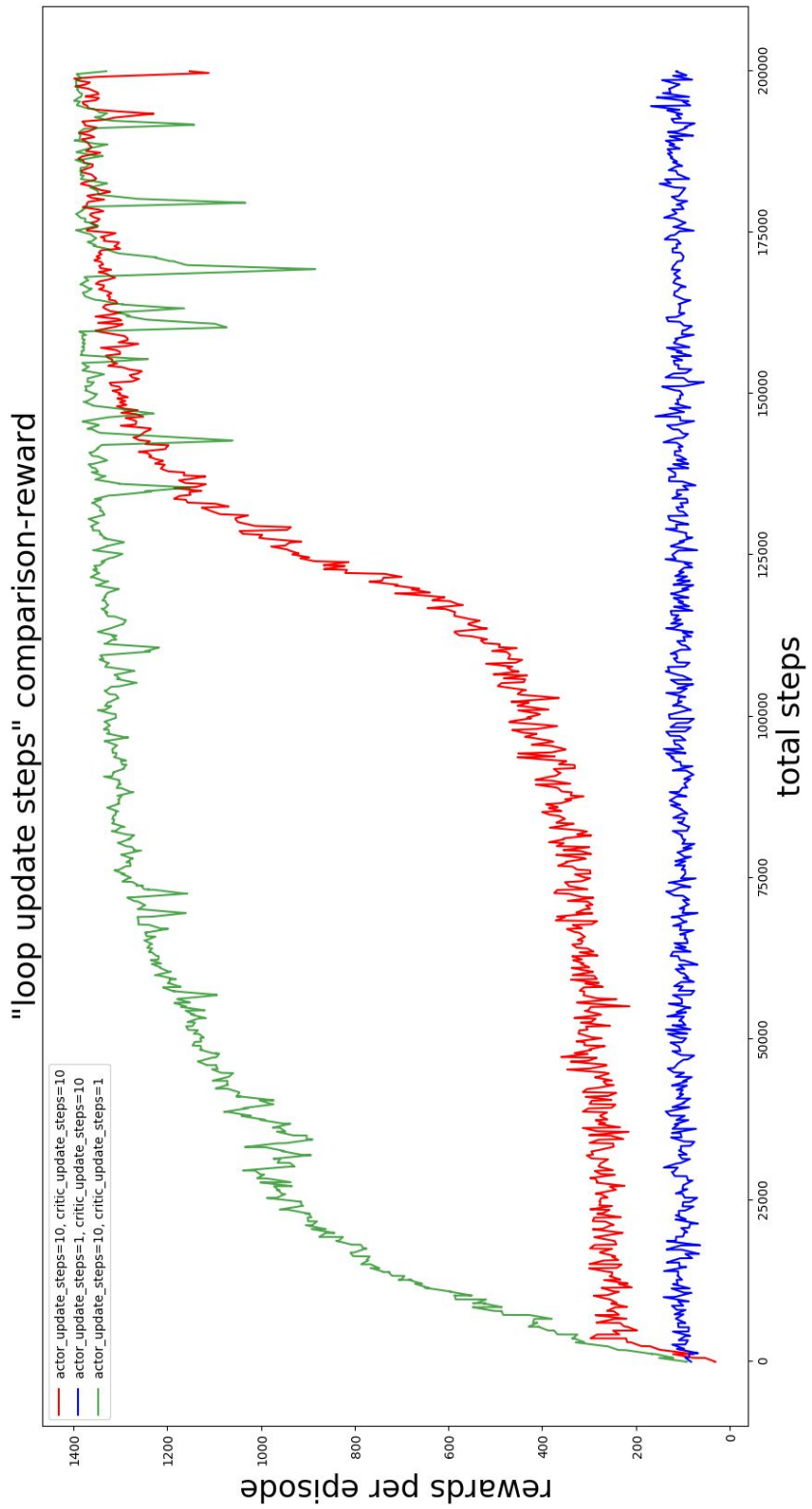

Figure 5.27: DPPO-CirTurtleBot: loop update steps comparison

5.2 Methods comparison through environments

In this evaluation section, we focus on the second research question. We compare the three methods with a high performance guarantee (based on the studies in the previous section). We list all the meta-parameters which we tuned for high performance in Table 5.19.

We evaluated the three methods regarding:

- **Time complexity**
  With fixed training steps, we compare the running time among the three methods.
- **Sample efficiency**
  Sample efficiency is an important criterion 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 are intuitive ways for performance evaluation.
- **robustness**
  Robustness here means the offset amplitude around the highest score, for the fully trained agent.

Entity	Parameter	Value in CartPole	Value in PlaneBall	Value in CirTurtleBot
DQN	learning_rate memory _size warm_up_steps batch_size target_net_update train_interval discount_rate exploration	0.001 10000 2000 64 0.01 1 0.99 Boltzmann (tau=1)	0.01 100000 20000 640 0.01 1 0.99 Boltzmann (tau=1)	0.001 10000 2000 64 0.01 1 0.99 Boltzmann (tau=1)
DDPG	actor_lr critic_lr memory _size warm_up_steps batch_size target_net_update train_interval discount_rate exploration	0.00001 0.001 10000 1000 96 0.001 1 0.99 OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)	0.00001 0.001 50000 1000 96 0.001 1 0.99 OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)	0.0001 0.001 50000 1000 32 0.001 1 0.99 OrnsteinUhlenbeck (theta=0.15, mu=0, sigma=0.3)
DPPO	actor_lr critic_lr batch_size discount_rate loop_update_steps exploration constrain method num_workers	0.0001 0.00002 32 0.99 (10,10) random samples from a normal distribution clipped surrogate objective, epsilon=0.2 4	0.00001 0.0001 200 0.99 (10,100) random samples from a normal distribution clipped surrogate objective, epsilon=0.2 4	0.00001 0.0001 32 0.99 (10,1) random samples from a normal distribution clipped surrogate objective, epsilon=0.2 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 the CartPole environment. The hyper-parameters of these three methods are chosen from the first evaluation section with relatively high performance guarantees. We trained in 300k steps, and generate the curves of 'episode reward' responding to 'total steps', as Figure 5.29 shows. In this figure, the red line represents DQN, the blue line represents DDPG, the green line represents DPPO. Among these three methods, DQN converges 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 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, therefore the methods score higher in robustness. 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 robustness.

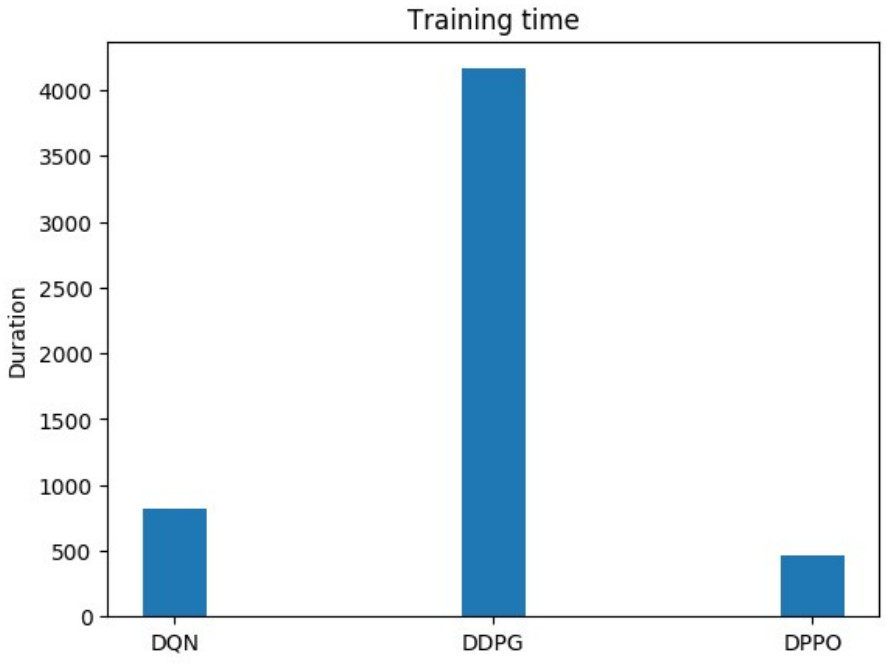

Figure 5.28: CartPole: Training time comparison

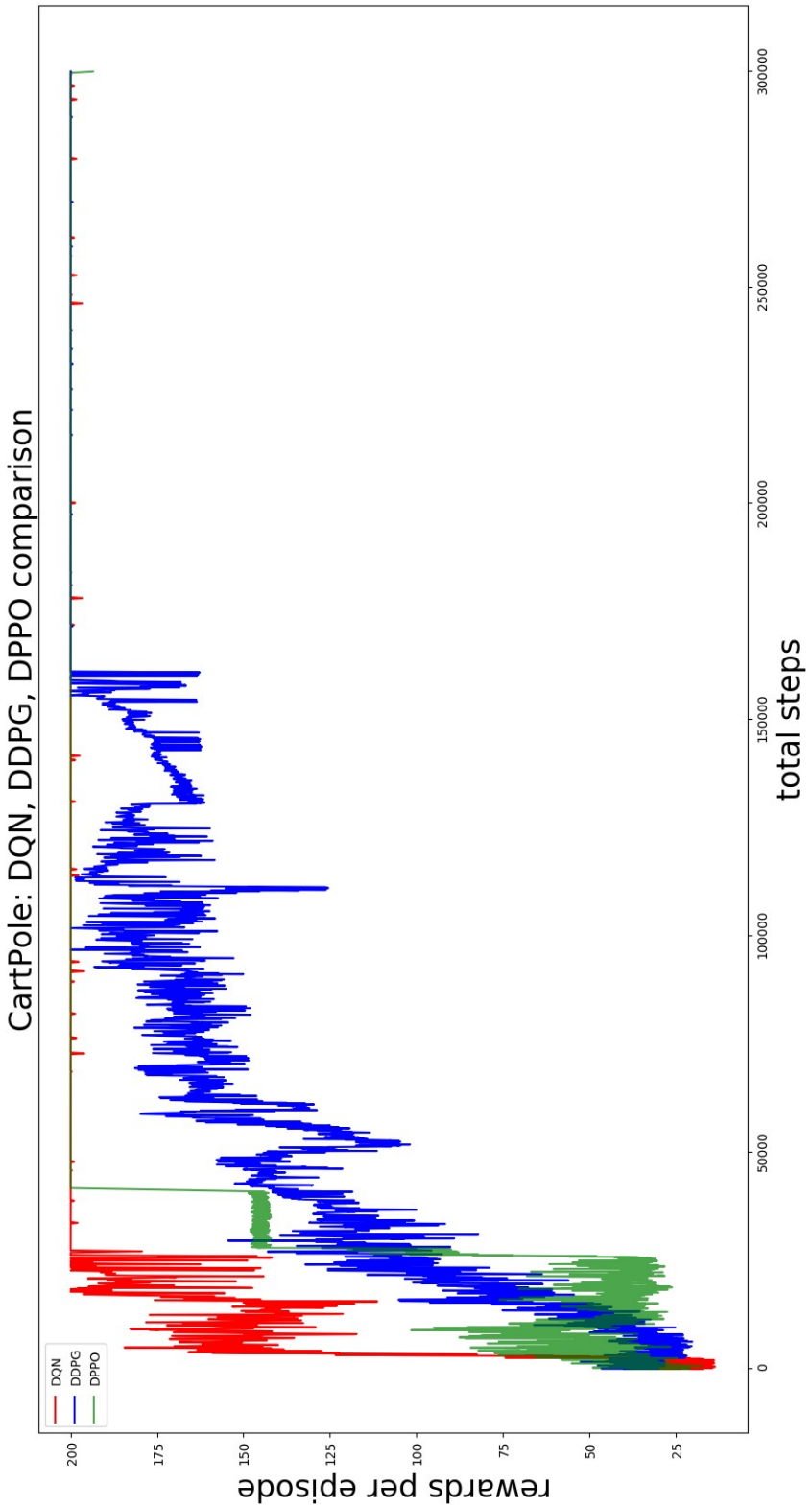

Figure 5.29: CartPole: DQN, DDPG, DPPO comparison

5.2.2 Comparison in "PlaneBall"

We evaluate the performance of DQN, DDPG and DPPO in the PlaneBall environment. The hyper-parameters of these three methods are chosen from the first evaluation section with relatively high performance guarantees. We trained in 500k steps, and generate the curves of 'episode reward' responding to 'total steps', as Figure 5.31 shows. In this figure, the red line represents DQN, the blue line represents DDPG, the green line represents DPPO. Among these three methods, DDPG converges 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 scores after convergence. 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 of 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.

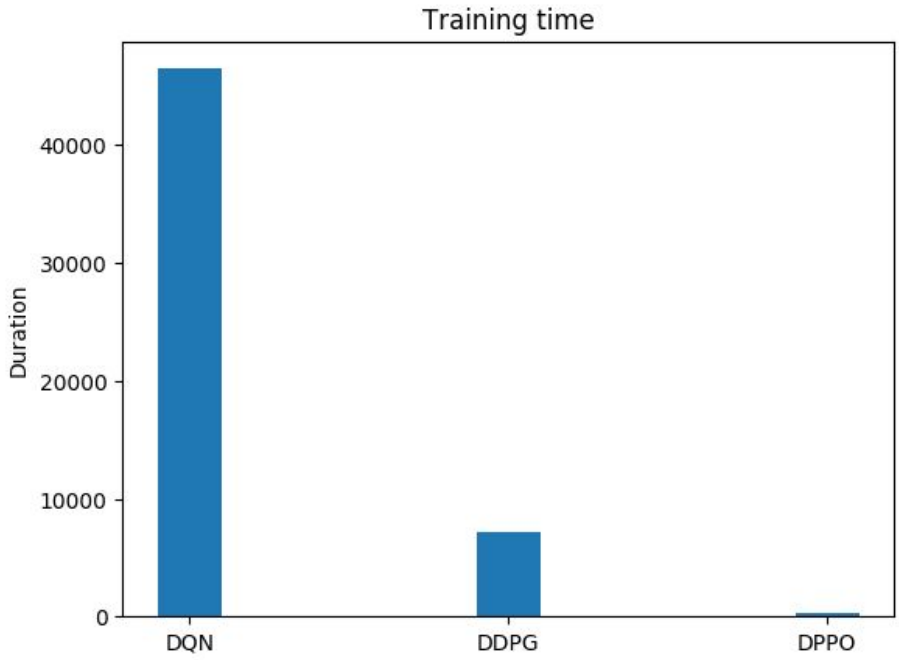

Figure 5.30: PlaneBall: Training time comparison

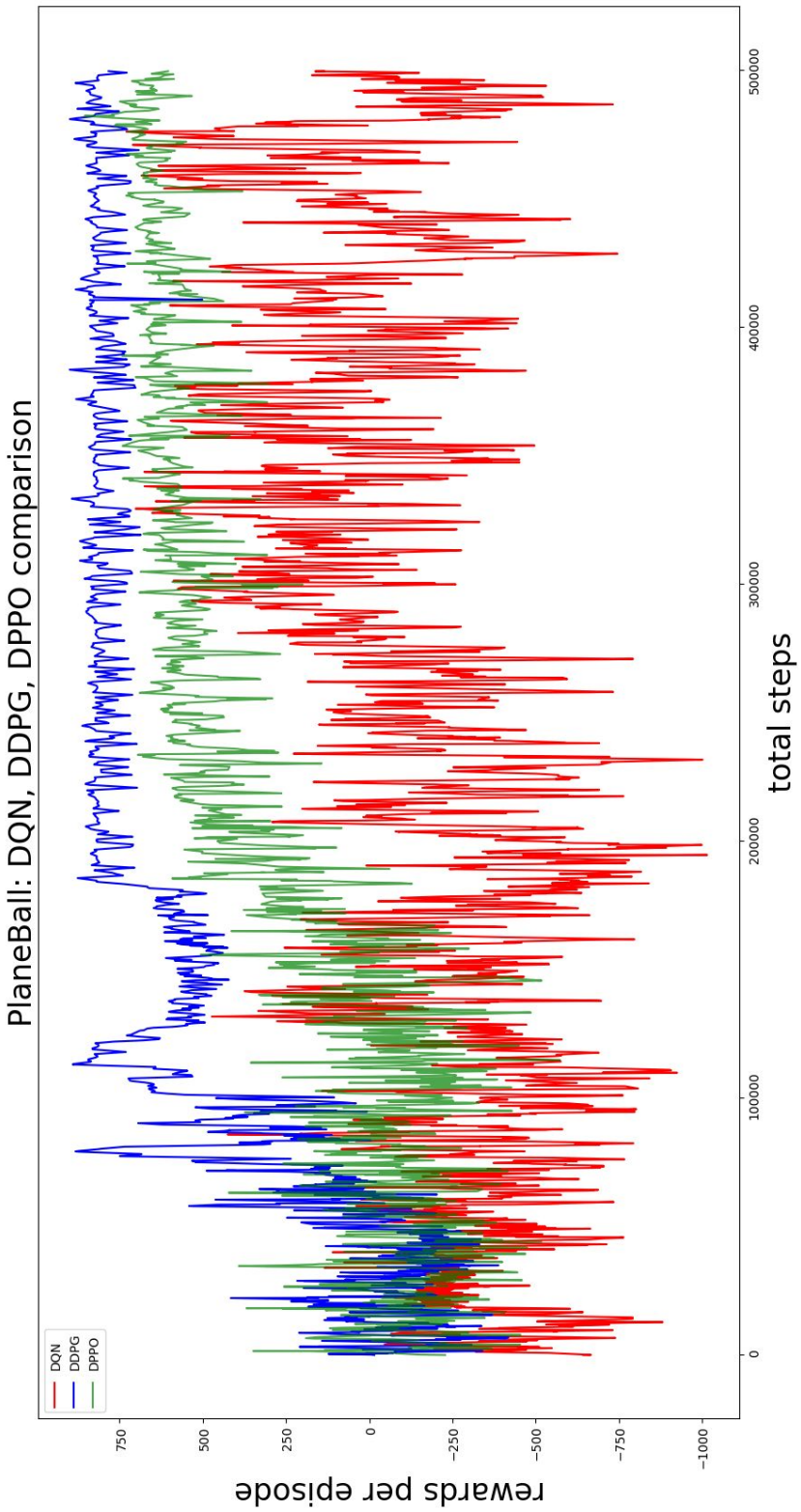

Figure 5.31: PlaneBall: DQN, DDPG, DPPO comparison

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 guarantees. We trained in 200k steps, and generate the curves of 'episode reward' responding to 'total steps', as Figure 5.33 shows. In this figure, the red line represents DQN, the blue line represents DDPG, the green line represents DPPO. Among these three methods, DDPG converges 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 an average score. DDPG had better sample efficiency and robustness than DPPO.

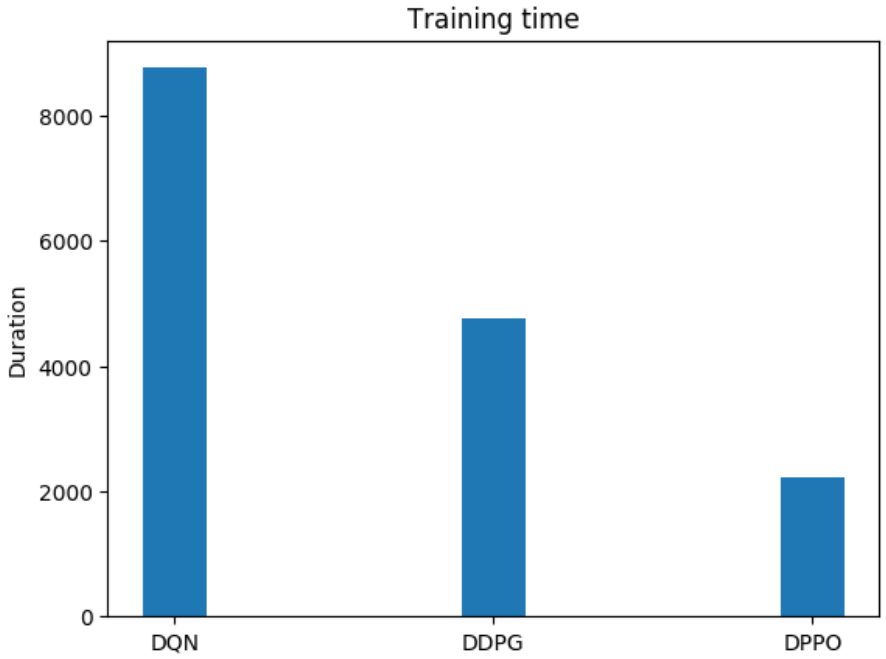

Figure 5.32: CirTurtleBot: Training time comparison

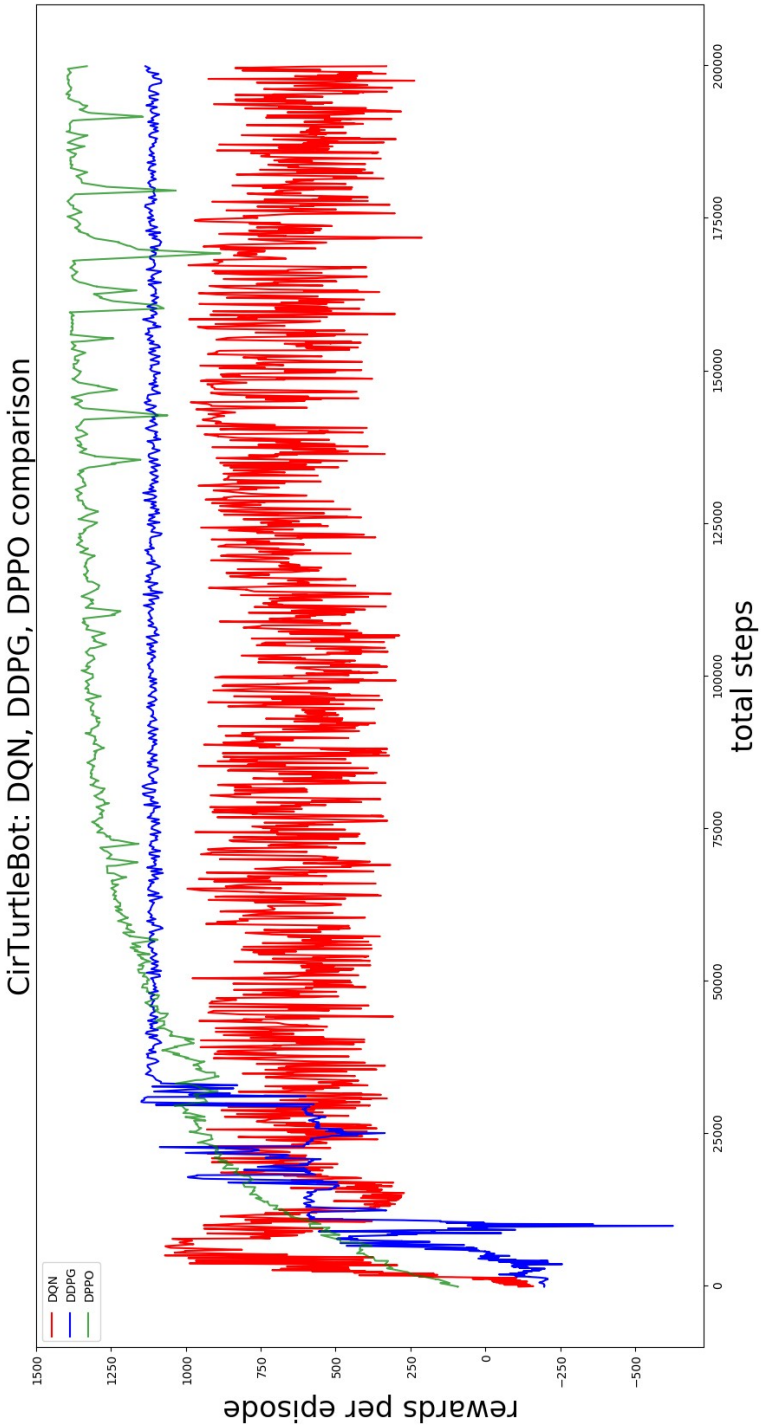

Figure 5.33: CirTurtleBot: DQN, DDPG, DPPO comparison

5.3 Summary

In this chapter, evaluation and results are discussed regarding the research questions.

For the first research question we found that hyper-parameters have a high impact, and thus need to be reported in oder to understand the results of a benchmark. We were specifically able to ascertain experimentally the important effect of exploration policies and their parameters, update rate to target networks, memory sizes, actor critic learning rate and batch size in PPO.

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 a 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 these in Chapter 7 in detail.

In the 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 research in benchmarks for deep reinforcement learning. We first discuss literature of RL in applications, then come to challenges, last but not least, the released benchmarks.

6.0.1 Engineering Applications

Some papers have researched Deep Reinforcement Learning in applications. [LWR+17] 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 studies on the stochastic computing-based hardware implementations of the DRL framework, which constitutes an improvement in area efficiency and power consumption compared with binary-based implementation counterparts. Comparing this research with our work, we also list three applications that intend to represent engineering tasks of varying complexity, however, we focus on specific DRL methods, and we provide a comparative evaluation.

In recent work [PR96], a new algorithm called C-Trace, a variant of the P-Trace RL algorithm is introduced, and some possible advantages of using algorithms of this type are discussed. Authors 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 a cart-pole system as the experimental environment, engineering control applications are also our main focus. However, the algorithms we implement are recent Deep RL algorithms, and we focus solely on the comparison of these methods with regards to their sensitivity to hyper-parameter configuration issues.

Similar to our work, in [Cap18], 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, authors come to deep reinforcement learning and function approximation. Two industrial applications, robotics and autonomous driving, are utilized for experiments. The testbeds are a Double Inverted Pendulum , Hopper and a TORCS simulator. The DDPG and A3C methods are implemented and evaluated in these three environments. Comparing with our work, we too present state-of-the-art Deep RL methods DDPG and A3C, and one experimental environment. We also present DQN and PPO, and evaluate these methods on three environments: "CartPole", "PlaneBall", "CirTurtleBot". Our environments are chosen such that they can encompass different aspects of engineering application. The goal of our task is to provide results on the practical criteria that needs to be considered (hyper-parameter disclosure, and essential metrics per method) for further Deep RL in engineering application benchmark design.

6.0.2 Challenges of RL in Engineering Applications

There is preceding work to ours which is concerned only in the field of robotics [PN17, KCC13, PVS03, Mat97, KBP13b], which is one of the important Deep RL application domains within 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, the existing research focuses on implementing DRL in the robotic field and in making comprehensive analysis. In [KCC13], authors 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 directions for reinforcement learning in robotics.

In [KBP13b], 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, and finally, the Curse of Goal Specification. In our paper, we discussed the challenges of RL in applications that, when considering the real world counterparts, are also facing the aforementioned aspects. Differently, we also present other challenges regarding to general engineering applications but in more simple, simulated environment.

6.0.3 Benchmarks for RL

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)[BNVB13] which is currently popular benchmark, offering a collection of Atari 2600 games as reinforcement learning problems, and recently the *RLLab* benchmark for continuous control[DCH$^+$16]. There are also RL benchmarks regarding different comparison aspects, like[GDK$^+$15, TW09, AV15, AV15, DEK$^+$05, dBS16].

The Arcade Learning Environment(ALE)[BNVB13] is nowadays a well-known benchmark. The 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. Authors provide 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.Comparing to our work, in terms of environments, we are concerned more in simulated environments which can be applied to engineering applications. In terms of RL methods, we used recent Deep RL methods. With regards to the comparison criteria, cumulative reward was not the only criteria that we considered, but we report other factors like algorithm robustness, running time, etc.

In 2016, another RL benchmark for continuous control[DCH$^+$16] was released. Since most work in ARCADE [BNVB13] was targeting aim algorithms designed for tasks with high-dimensional state inputs and discrete actions, there was a gap regarding the comparison of algorithms for environments with continuous action spaces. This novel benchmark consists of 31 continuous control tasks with 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 when compared to ARCADE, such like the iteration numbers of each algorithm, average reward, etc. All the source code and the agents are made publicly available rllab[1]. Comparing to our work, our aim is to provide practical criteria to make comparisons regarding engineering application features. Alongside, the algorithms we used for experiments are DQN, DDPG, and DPPO, which are recently released Deep RL algorithm. In the aspect of environments, we also choose them from simple classical control problems, like the CartPole system to a Hierarchical robotic Task.

[1]http://ray.readthedocs.io/en/latest/index.html

There is also related wok [dBS16] dedicated to comparing the sample complexity of RL algorithms. Authors have 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 the 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 understood as the comparison between two very different kinds 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 robustness, time complexity, and the implemented algorithms are DQN, DDPG, and DPPO which are currently competitive methods.

6.1 Summary

In this chapter, we present a comparative review on work that is similar to our research.

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

7. Conclusions and future work

In this chapter, we summarize all our work and give some general ideas for 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 kinds of comparisons regarding two research questions. First of all we did a hyper-parameter comparison, where methods were compared one by one. We evaluate one method in one environment, comparing the method with different hyper-parameters' values, choosing by the end the values that result in the highest performance. After that, we compared different methods with high performance hyper-parameters in one specific environment. During the second comparison, we evaluate these methods with regards to training time, sample efficiency and robustness.

According to our evaluation, the DQN method had the higher performance only in the "CartPole" environment. This environment is famous as a classical problem with low dimension space. We can assume that, in engineering problems, if we can generate those problems as a 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 using DDPG for a 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, if we use DPPO in engineering problems, we should also take the hardware and training cost into

consideration, Because the performance of DPPO relies on the workers' number and thus requires more computational power. This means that it needs better hardware support than the other two methods. The training time and robustness of DPPO are dependent on how many workers are 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

- **Limitations in Hardware**
  We have to admit that the computer's performance really can affect the running time, the training time is different for the same algorithm running on different computers. And GPUs are considered to be much faster than CPUs. This may affect the performance of the DPPO method because we can't train the agent with more than 8 workers in parallel. 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 find comparable 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 being proposed and updated with high speed, the performance could be affected by using the newest optimization mechanism. For example, the DQN method has many variations, such as Double-DQN, Distributed DQN, Prioritized Experience Replay, and others. Recently, the 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 the clipped surrogate objective. This may affect 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 accurate if taking more trials for the experiment, smoothing out variances due to the randomness in exploration and highlighting trends with more clarity. Second, we perform 30k, 50k, 20k steps for CartPole, PlaneBall, CirTurtleBot training. It will affect the robust performance comparison. Last but not least, choosing a high performance hyper-parameters combination is complicated because we usually can't compare all the possibilities of the hyper-parameters combinations due to time and computational limits.

7.3 Future work

- Training with a higher-performance computer, especially with GPUs.
- Taking more engineering tasks for the comparison, 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 criteria for methods comparison.
- Benchmarking general engineering tasks.

Bibliography

[ADBB17a] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017. (cited on Page 3)

[ADBB17b] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017. (cited on Page 4)

[ADBB17c] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017. (cited on Page 23)

[AÖ00] M Emin Aydin and Ercan Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3):169–178, 2000. (cited on Page 53)

[AV15] 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. (cited on Page 115)

[BB01] Jonathan Baxter and Peter L Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001. (cited on Page 18)

[BC95] 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. (cited on Page 53)

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. (cited on Page ix, 55, and 56)

[Bel57] Richard Ernest Bellman. Dynamic programming. 1957. (cited on Page 3)

[BK92] Hamid R Berenji and Pratap Khedkar. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on neural networks*, 3(5):724–740, 1992. (cited on Page 53)

[BNVB13] 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. (cited on Page 5, 6, and 115)

[Bra93] Steven J Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in neural information processing systems*, pages 295–302, 1993. (cited on Page 53)

[BSA83] 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. (cited on Page 21)

[Cap18] Antonio Cappiello. Deep reinforcement learning algorithms for industrial applications, 2018. (cited on Page 114)

[CB96] Robert H Crites and Andrew G Barto. Improving elevator performance using reinforcement learning. In *Advances in neural information processing systems*, pages 1017–1023, 1996. (cited on Page 46)

[cla18] Stanford CS class. Cs231n: Convolutional neural networks for visual recognition, 2018. (cited on Page 25)

[dBS16] 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. (cited on Page 115 and 116)

[DCH+16] 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. (cited on Page 5, 6, 59, and 115)

[DEK+05] 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. (cited on Page 6 and 115)

[DW92] Peter Dayan and CJCH Watkins. Q-learning. *Machine learning*, 8(3):279–292, 1992. (cited on Page 17)

[ET16] 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. (cited on Page 13 and 50)

[Fin04] Steven Finch. Ornstein-uhlenbeck process. 2004. (cited on Page 65)

[FLJ+10] 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. (cited on Page 53)

[GDK+15] 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. (cited on Page 5 and 115)

[Gly87] Peter W Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987. (cited on Page 18)

[GP02] Ilaria Giannoccaro and Pierpaolo Pontrandolfo. Inventory management in supply chains: a reinforcement learning approach. *International Journal of Production Economics*, 78(2):153–161, 2002. (cited on Page 53)

[Ham17] Mark Hammond. Deep reinforcement learning in the enterprise: Bridging the gap from games to industry, 2017. (cited on Page 45, 46, 50, and 51)

[HMVH+17] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017. (cited on Page 4)

[Hor14] Eric Horvitz. One hundred year study on artificial intelligence: Reflections and framing, 2014. (cited on Page 45)

[HP04] 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. (cited on Page 53)

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. (cited on Page 28)

[HS00] 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. (cited on Page 53)

[HSL+17] 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. (cited on Page ix, 4, 37, and 39)

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. (cited on Page 35 and 36)

[Kak02] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002. (cited on Page 18)

[KBKK12] 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. (cited on Page 5 and 53)

[KBP13a] 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. (cited on Page 2)

[KBP13b] 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. (cited on Page 21, 45, 49, 50, 51, 52, and 114)

[KCC13] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013. (cited on Page 12, 14, 49, and 114)

[Kim99] 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. (cited on Page 60)

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. (cited on Page 11, 12, 13, and 46)

[KSH12] 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. (cited on Page ix and 26)

[KSK99] 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. (cited on Page 46 and 53)

[LBBH98] 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. (cited on Page ix and 26)

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015. (cited on Page 23)

[LHP+15] 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. (cited on Page ix, 4, 35, and 36)

[Li17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017. (cited on Page 3, 4, and 23)

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993. (cited on Page 21 and 32)

[LMFL15] 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. (cited on Page 53)

[LWR+17] 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. (cited on Page 113)

[LYZ05] 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. (cited on Page 45)

[Mat97] Maja J Matarić. Reinforcement learning in the multi-robot domain. In *Robot colonies*, pages 73–83. Springer, 1997. (cited on Page 46, 53, and 114)

[MBM+16] 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. (cited on Page ix, 4, 40, and 42)

[MKS+13] 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. (cited on Page ix, 3, 31, 33, and 34)

[MLLFP06] 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. (cited on Page 51)

[MMDG97] 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. (cited on Page 46 and 53)

[NB18] Abhishek Nandy and Manisha Biswas. Reinforcement learning with keras, tensorflow, and chainerrl. In *Reinforcement Learning*, pages 129–153. Springer, 2018. (cited on Page 13 and 59)

[NCD+06] 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. (cited on Page 53)

[NJ00] 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. (cited on Page 18)

[NW06] Jorge Nocedal and Stephen J Wright. *Sequential quadratic programming*. Springer, 2006. (cited on Page 37)

[OWC04] 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. (cited on Page 53)

[Pat14] Jyoti Ranjan Pati. Modeling, identification and control of cart-pole system, 2014. (cited on Page 60)

[PMB13] 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. (cited on Page 28)

[PN17] 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. (cited on Page 114)

[Pow12] Warren B Powell. Ai, or and control theory: A rosetta stone for stochastic optimization. *Princeton University*, 2012. (cited on Page 12)

[PR96] 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. (cited on Page 113)

[PS06] 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. (cited on Page 18)

[PS08] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008. (cited on Page 19)

[PVS03] 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. (cited on Page 53 and 114)

[PVS05] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *European Conference on Machine Learning*, pages 280–291. Springer, 2005. (cited on Page 18)

[QCG+09] 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. (cited on Page 56)

[RA12] George Rzevski and RA Adey. *Applications of artificial intelligence in engineering VI*. Springer Science & Business Media, 2012. (cited on Page 45)

[RN16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016. (cited on Page 1 and 2)

[RV11] 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. (cited on Page 53)

[SB+98] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998. (cited on Page 3)

[Sho17] Yoav Shoham. Toward the ai index. *AI Magazine*, 38(4), 2017. (cited on Page 1)

[SLA+15] 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. (cited on Page 37)

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014. (cited on Page 35)

[SMSM00] 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. (cited on Page 18)

[SR08] 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. (cited on Page 5 and 53)

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. (cited on Page ix, 4, 37, 38, and 39)

[TW09] Brian Tanner and Adam White. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10(Sep):2133–2136, 2009. (cited on Page 5 and 115)

[UHFV00] 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. (cited on Page 53)

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016. (cited on Page 33 and 67)

[WH00] Rüdiger Wirth and Jochen Hipp. Crisp-dm: Towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, pages 29–39. Citeseer, 2000. (cited on Page 7)

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992. (cited on Page 18)

[WU05] 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. (cited on Page 53)

[WWZ17] 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. (cited on Page 5 and 53)

[YWV+18] 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. (cited on Page 53)

[Zha96] Wei Zhang. Reinforcement learning for job-shop scheduling. 1996. (cited on Page 46)

[ZLVC16] 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. (cited on Page x, 50, 56, and 57)