# Implement reinforcement learning algorithms to control robot's locomotion in Simulation.

Zhipeng Liu

(ID: 1772961)

School of Computer Science

University of Birmingham

Supervisor: Dr. Morteza Azad

Master of Science in Robotics

MSc Summer Project

10th September 2018

# Acknowledgments

First and foremost, I want to use this opportunity to express my very great appreciation to my supervisor *Dr. Morteza Azad* who was guiding me patiently in the last nine months. This project would have been impossible without his expert advice and useful critiques. Also, I would like to thank *prof. Jeremy L Wyatt* for his valuable comments. Finally, I wish to thank my family for their support and encouragements throughout my study.

# Abstract

Reinforcement learning has made a set of significant progress in the last few years. This project implements various reinforcement learning algorithms, including state-of-art approaches DDPG and D4PG (incomplete), and use them to solve a challenging balancing task. Moreover, a systematic comparison among implemented algorithms is given.

## Keywords

Reinforcement Learning; Locomotion Control; Deep Learning.

# Content

# List of Figures

# Chapter 1

## Introduction

The major objective of this project is to implement various reinforcement learning algorithms, such as deep Q-network (DQN), and deep deterministic policy gradient (DDPG). Moreover, a robot's locomotion control task is designed for testing and comparing those algorithms. This thesis assumes all readers have a basic knowledge of deep learning.

Reinforcement learning (RL) is a subfield of machine learning specializing in learning how to make optimal decisions over time. Another two important subfields are supervised learning and unsupervised learning, and reinforcement learning lays between them. RL does not learn optimal policies from labelled examples which is a necessary component in the supervised learning. But it uses another kind of label called reward to guide learning, which is also different from unsupervised learning. The objective of reinforcement learning is to find out an optimal strategy in order to maximize reward. As a consequence, one of the main applications of reinforcement learning is the robot's locomotion control. Through reinforcement learning, the robot can learn how to act in some environments by trial and error regiments without any expert knowledge.

In the last few years, reinforcement learning has made a set of significant progress, which is attracting increasing researchers to work on it. In 2017, the world's best Go player was beaten by Google DeepMind's AI which is based on deep reinforcement learning (i.e. reinforcement learning with deep neural network). More recently, OpenAI Five, a team of five reinforcement learning based neural networks, defeated 99.95% best amateur human teams at Dota 2 which is a real-time strategy game played between two teams of five players. In the robot's locomotion control field, the latest achievements of deep reinforcement learning are demonstrated by mastering parkour domain. The

parkour domain is shown in figure 1.

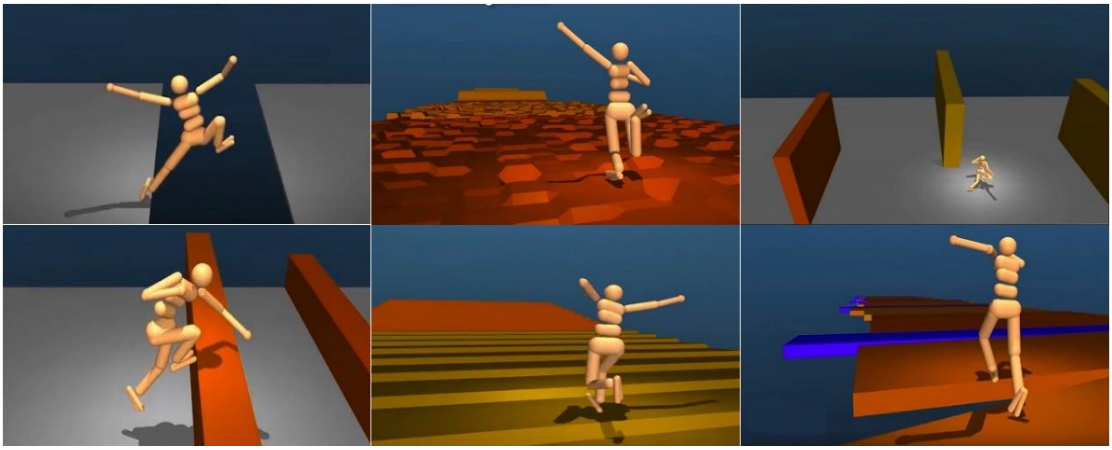


*Figure 1: Parkour domain (Barth-Maron et al., 2018)*

The humanoid robot in figure 1 learns not only how to walk and run on perfect and rough terrain, but also how to avoid obstacles (e.g. walls), jump over hurdles and gaps, and climb stairs though behaviour looks adorable.

Inspired by this, the designed task used to test RL algorithms is based on a two-wheel balance robot. The goal of the robot is to keep the balance and move forward on flat surface and slopes. The screenshot of the task in simulation is shown in figure 2.

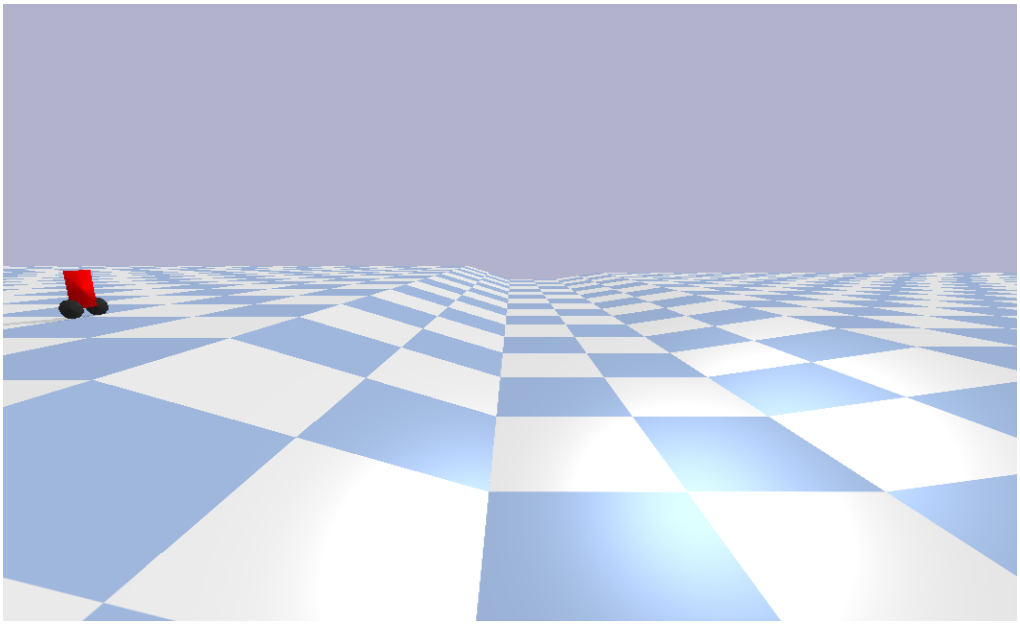


*Figure 2: The designed task*

In the designed task, the robot starts from the flat surface. It needs to go down a slope, climb another slope and keep the red body balanced. It is a torque control robot with two actuated joints on two wheels.

Chapter 2 (Background) explains all the reinforcement learning algorithms implemented in this project.

Chapter 3 (Related work) introduces some other very famous deep reinforcement learning algorithms which are not implemented in this project. Also, we will talk about deep genetic algorithms (deep GA).

Chapter 4 (The designed task) shows all the details of the designed task.

Chapter 5 (Experiments and Results) demonstrates achieved results.

Chapter 6 (Further development) discusses several things can do about this project in the future.

Chapter 7 (Conclusion) concludes the whole project.

# Chapter 2

## Background

The first deep reinforcement learning algorithm, deep Q-network (DQN), was proposed by DeepMind in 2015. It is based on a basic reinforcement learning algorithm Q learning. Since then, reinforcement learning can be empowered by deep neural network directly. One of the most significant achievements is that reinforcement learning is capable of dealing with high-dimensional states like raw image pixels. After that, the ideas underlying the success of deep Q-network was adapted to the continuous action space, then we have the deep deterministic policy gradient (DDPG) algorithm which is presented in 2017. It relies on actor-critic architecture. Furthermore, distributional distributed deep deterministic policy gradient (D4PG) algorithm, an extended version of DDPG, was proposed in 2018. It adopts several very successful improvements (e.g. distributional perspective) and works within a distributed framework.

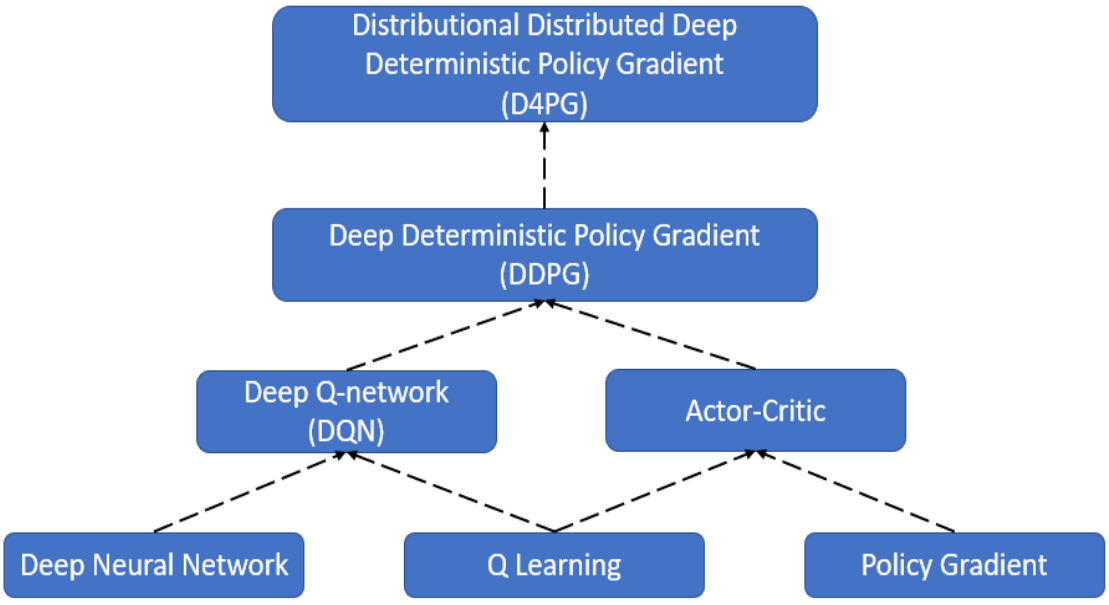



*Figure 3: The path of implementation*

This project starts from implementing very basic reinforcement learning algorithms Q learning and Policy Gradient, then going to DQN, Actor-Critic, and finally implementing DDPG and part of D4PG algorithms. You can see the path of implementation clearly in figure 3.

## 2.1 Q Learning and Deep Q-network

### 2.1.1 Q Learning

The core problem of Q learning is how to estimate an optimal action-value function for any given finite Markov Decision Process (FMDP). In Q learning, we usually use a Q table to represent the function. At each time-step, the agent (e.g. a robot) take an action given the current observation. The action is passed to the environment, and the environment returns a new observation and reward. Q table is updated by the Bellman equation, shown in equation (2.1), depending on this transition. With time passing, the Q table updates iteratively.

$$\mathrm{newQ}(s,\mathrm{a}) = Q(s,a) + \partial\,[R(s,a) + \gamma \max \mathrm{Q}(s',a') - Q(s,a)] \quad \text{equation (2.1)}$$

In the equation (2.1), $s$ and $s'$ are the states before and after taking an action respectively, $\partial$ is learning rate, and $\gamma$ is the discounted factor. Briefly, the Q value of current state (i.e. $Q(s,a)$ ) is updated depending on the reward and the difference between the highest Q value of next state (i.e. $\max \mathrm{Q}(s',a')$) and current Q value. Then, the Q table will be gradually becoming the optimal action-value function. The prove is covered by (Sutton et al., 1998). Using Optimal value function, the agent always select action which maximizes the expected total reward of all successive steps, starting from the current state.

### 2.1.2 Deep Q-network (DQN)

In DQN, the action-value function is estimated by a neural network rather than a Q table. Generally, we use a convolutional neural network (CNN) deal with raw image pixels input (i.e. high-dimensional input). Or use a two-layer fully connected neural network when the observation is low dimensional like position

and velocity of an agent. The network predicts values of every potential action given the current observation. Moreover, the agent executes an action according to a $\epsilon$ - greedy strategy that takes the highest value action (i.e. best action) with probability (1- $\epsilon$) and does a random action with probability $\epsilon$. Epsilon is usually 0.1. This is for exploration. Besides, two innovative ideas are used in DQN to guarantee deep neural network works in reinforcement learning. They are Experience Replay and Target Network.

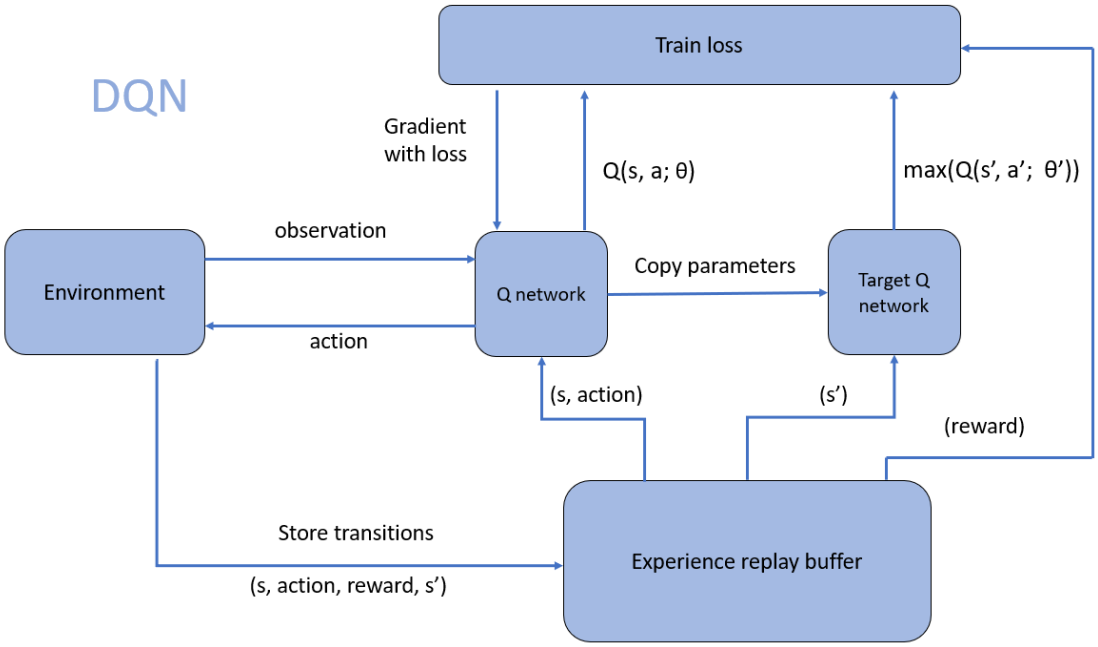



*Figure 4: DQN algorithm*

In most deep learning methods, the training data are assumed independent. However, there is a strong correlation between successive steps in reinforcement learning. In order to avoid bias caused by correlation, a memory buffer is used in DQN which stores all experiences including (state, action, reward, next state) of every single step. Then, the neural network learns from samples which are randomly selected from the buffer. As a result, experience reply can reduce the correlation between samples in updating neural network.

In order to approximate an optimal action-value function by a deep neural network, the loss function is designed according to the Bellman equation. It is shown in equation (2.2).

$$\text{loss} = [targetQ - Q(s, a;\ \theta)]^2 \qquad \text{equation (2.2)}$$

$$\text{targetQ} = \text{R} + \gamma \max[\text{Q}'(\text{s}', \text{a}')] \qquad \text{equation (2.3)}$$

In the equation (2.2), the Q function is the action-value function, and the target Q, shown in equation (2.3), represents target part in the Bellman equation. Intuitively, target Q can be considered as optimal value function. The loss error is the difference between optimal value function and current value function, and that is what we want to reduce. The Q′ function in target Q is estimated by the target network which is used to predict the Q value of the next state $s'$ (i.e. the state after acting), while the $Q$ function is estimated by Q network which is used to predict the Q value of the current state $s$ (i.e. the state before acting). Target network has the same architecture as Q network but different parameters. It is updated every fixed step (e.g. 1000 steps) with a copy of the latest learned parameters of Q network. The main reason why we do not use one neural network to estimate both Q and Q′ function is that change Q′ function too frequently (if it updates at every step just like Q function) causing instability in learning. In the other word, using a separate target network can give the learned neural network (i.e. Q network) a relative constant target which will stabilize learning (Mnih et al., 2015). The whole process of DQN algorithm is shown in figure 4. Besides, from equation (2.1), (2.2), (2.3), it is easy to find out that the update methods of Q learning and DQN are both based on the Bellman equation. So DQN is basically the combination of deep neural network and Q learning.

## 2.2 Policy Gradient and Actor-Critic

### 2.2.1 The advantage of Policy Gradient

In chapter 2.1, Q learning and deep Q-network are both value-based reinforcement learning algorithms. They estimate policy indirectly via approximating values of every potential action and choosing the best one. However, there is an alternative way to handle finite Markov Decision Process (FMDP), which is called Policy Gradient (PG). Instead of finding optimal action-value function, Policy Gradient focuses on estimating policy itself. One of the advantages of it is that it can deal with continuous action space which value-

based methods cannot. For example, if your task needs fine control like the steering wheel angle of a self-driving car, Policy Gradient is much better than value-based methods. Using PG, we do not care about values. The policy will directly make the decision what to do next. But for value-based methods, we have to discrete continuous action and calculate values of all possible actions. As a result, in such cases, it is much more feasible to avoid values and work with policy directly.

### 2.2.2 The issues of Policy Gradient

In chapter 2.2.1, we discussed the advantage of Policy Gradient. But unfortunately, PG also suffer from several problems which limit it only used in simple environments.

First of all, full episodes are required. It means that training does not start until the full episode is completed. This situation is fine for simple tasks like Cart-Pole task, when in the beginning, the pole often falls down in less than 20 steps. But for some other tasks like Pong, each episode usually last for hundreds of or even thousands of frames. In this case, the training batch will be huge, and from sample efficiency aspect, when the agent needs to communicate with the environment a lot, we just perform a single training step.

Another issue is the high gradient variance. Since the training is only at the end of episodes, the total reward of successive episodes could be very different. For example, in the CartPole task, we get reward of 1 for every timestep that the pole does not fall down. If we hold the pole for 5 steps, we will get total reward of 5. In the next episode, a lucky episode, we hold the pole for 50 steps, the total reward is 50. In this case, gradients of the two episodes have a ten times difference. This high gradient variance can seriously affect learning.

### 2.2.3 Actor-Critic

From the chapter 2.2.2, you can find out that the main drawback of policy gradient is that it cannot update at each timestep. In order to overcome this, another algorithm called Actor-Critic was proposed by (Peters et al., 2005). It contains two equally important components Actor and Critic. Actor is

responsible for estimating policy using a neural network. While Critic judges the policy using another neural network. For example, at each timestep, Actor selects an action given the current state (i.e. observation). Then the Critic outputs a value based on the current state and selected action. That value indicates the quality of that action which can be used for updating policy. The whole process of Actor-Critic is shown in figure 5.

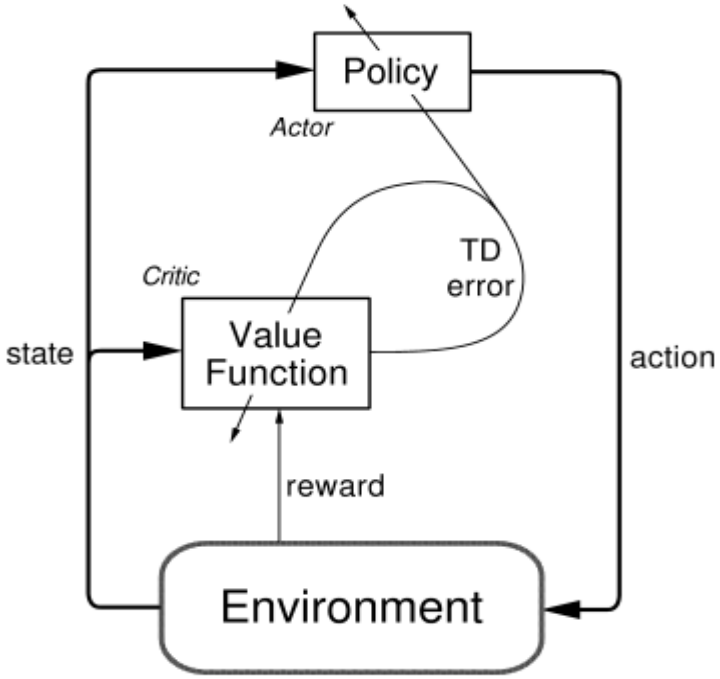



*Figure 5: Actor-Critic structure (Tsitsiklis et al., 1997)*

The gradient of the policy is shown in equation (2.4). It represents the direction in which we need to change the Actor network's parameters to improve the policy. The strong proof of this is covered in (Silver et al., 2014). In the equation, $\mathrm{Q}$ function represents Critic, and $\pi$ function represents Actor.

$$\text{Gradient of policy} = \nabla \mathrm{Q}(\mathrm{s},\mathrm{a}) * \nabla \pi(a|s) \qquad \text{equation (2.4)}$$

$$\text{Actor loss} = -\mathrm{Q}(\mathrm{s},\mathrm{a}) * \log \pi(a|s) \qquad \text{equation (2.5)}$$

$$\text{Critic loss} = \left[\left(\mathrm{R} + \mathrm{Q}(\mathrm{s}',\mathrm{a}')\right) - \mathrm{Q}(\mathrm{s},\mathrm{a})\right]^2 \qquad \text{equation (2.6)}$$

From a practical point of view, Actor can be implemented as performing optimization of the loss function (equation 2.5). The gradient of the $\mathrm{Q}$ function is proportional to the value of action taken, which is $\mathrm{Q}(s,\mathrm{a})$, and the gradient of $\pi$ function is equal to log-probability of the action taken. Additionally, the minus

sign in equation (2.5) is necessary, as the loss function is minimized during the Stochastic Gradient Descent (SGD), but we want to maximize the gradient of policy. Moreover, Critic is updated according to the Bellman equation. As a result, the loss function of Critic is designed as equation (2.6).

Sum up briefly, Actor component is based on policy gradient, and the Critic component is similar to value-based method Q learning. This algorithm overcomes those issues of Policy Gradient because it can update policy at every timestep with the aid of critic value.

## 2.3 Deep Deterministic Policy Gradient (DDPG)

### 2.3.1 The overview of DDPG algorithm

Actor-Critic overcomes drawbacks of Policy Gradient, but at the expense of updating two different neural networks at the same time. The strong correlation between the two neural networks makes them much more difficult to converge. Therefore, the learning process of Actor-Critic is not stable, and sometimes two neural networks even diverge.

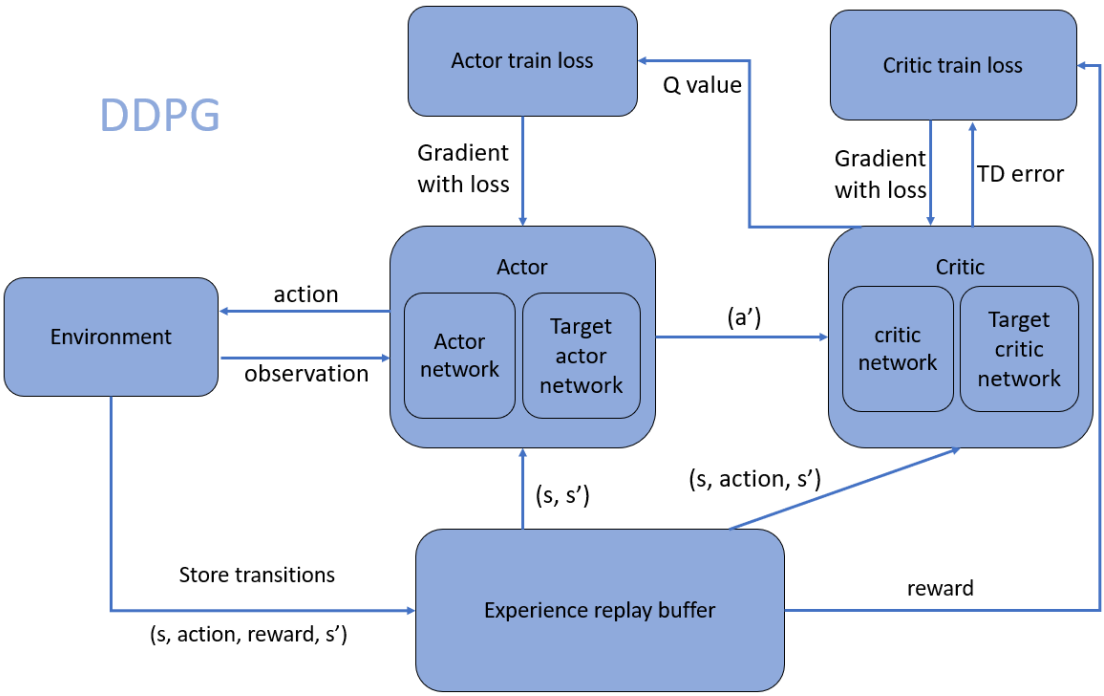



*Figure 6: DDPG algorithm*

As a consequence, Deep Deterministic Policy Gradient (DDPG) algorithm was proposed by (Lillicrap et al., 2016). It relies on Actor-critic structure, has two equally important components Actor and Critic. Thus, it can handle continuous action tasks as well. Also, it adapted ideas from deep Q-network (DQN), which let DDPG becomes a stable deep reinforcement learning algorithm. These ideas are Experience Replay and Target Network, which can stabilize the learning process of neural networks, and this is what Actor-Critic lacks for.

Therefore, there are two neural networks with the same architecture but different parameters in each component. For example, in Actor component, there is an actor network and a target actor network. The target network gives the actor network a relative constant target makes learning easier. Four neural networks (i.e. two for Actor and two for Critic) used in DDPG means high computational cost, but a much more stable learning process makes it worth doing. Also, experience replay can help DDPG algorithm reducing correlations between learning samples. The whole process of DDPG algorithm is shown in figure 6.

### 2.3.2 Two new ideas used in DDPG

The target network in DQN is updated every fixed step (e.g. 1000 steps) with a copy of the latest learned parameters, which is also called hard copy. However, DDPG updates target networks by another method called soft copy. Instead of copying directly, the parameters of target networks are updated by slowly tracking the parameters of learned networks. The soft copy equation is shown below.

$$\theta' = \tau * \theta + (1 - \tau) * \theta' \qquad \text{equation (2.7)}$$

In the equation, $\theta'$ represents parameters of target network, and $\theta$ represents parameters of learned network. The hyperparameter $\tau$ is much smaller than 1, as the target network needs to keep relative constant. This simple modification can further improve the stability of learning (Lillicrap et al., 2016).

The other new idea used in DDPG algorithm is batch normalisation (BN). Although BN is a mature idea in deep learning methods, it is the first time used in a reinforcement learning algorithm. Batch normalisation can be not only used

for input to normalise different features into a similar scale, but also used to normalise the output of each hidden layer. Intuitively, if the input layer is beneficial from normalisation, every hidden layer should be beneficial from it as well. In this project, the batch normalisation is implemented by adding batch norm layers immediately before both input layer and every non-linearity. The batch norm layers are provided by Pytorch which is a deep learning framework. There is a detailed explanation of batch normalisation in the paper (Ioffe et al., 2015).

### 2.3.3 The implementation of DDPG algorithm

DDPG consists of two different neural networks for actor component and critic component. Also, each component includes a learned neural network and a target neural network. They have the same architecture but different parameters. The actor network is extremely simple with two hidden layers. Two hidden layers have 400 and 300 units respectively (i.e. about 130000 parameters). The input is an observation vector, while the output is a deterministic action. The output action is transformed with hyperbolic tangent non-linearity to squeeze the values to [-1, 1]. Then times it by maximum action value, e.g. 4, so the action range will be [-4, 4]. However, the critic network is a little bit unusual. It also has two hidden layers. But there are two separate input paths for observation and action respectively. The observation is the same as the normal input, while the action is added until the second hidden layer. Then two paths are concatenated together to be transformed into the critic output of one number. The architectures of actor and critic neural networks are shown in figure 7. They come from the paper (Lillicrap et al., 2016).

On the training step, the critic network is updated by bellman equation (i.e. equation (2.1)). The critic loss function is the same as equation (2.2). This process is similar to the training of DQN. For actor network, we need to update the actor's parameters in a direction that will increase the critic's output. The loss function is the same as equation (2.5). After updating both actor and critic networks, we update two target networks by soft update using equation (2.7). Additionally, we use two different optimizers for actor and critic neural networks. It is easy to find out that most equations mentioned above are also used in DQN

or Actor-Critic algorithms. As a result, DDPG is basically the combination of DQN and Actor-Critic.

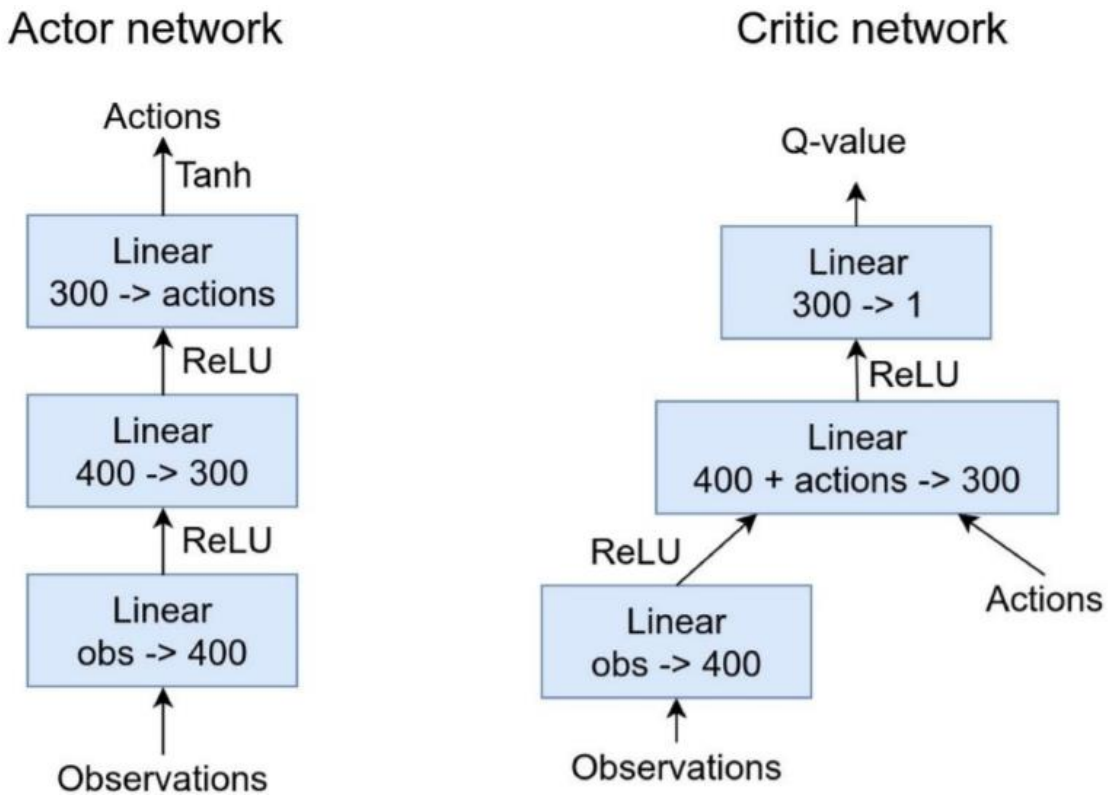


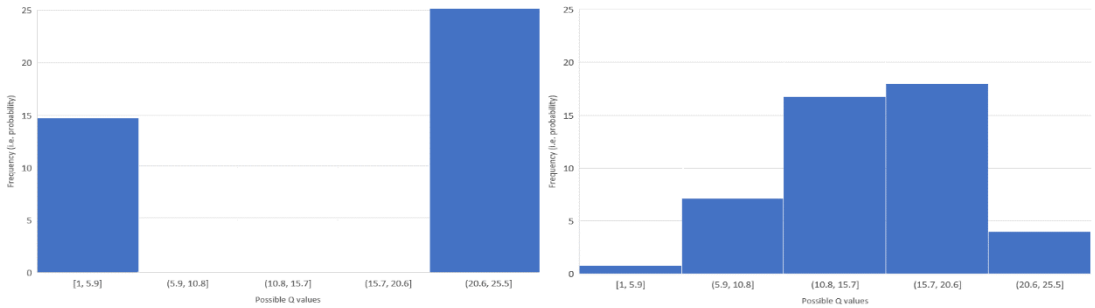

*Figure 7: DDPG actor and critic networks*

## 2.4 Distributional Distributed Deep Deterministic Policy Gradient (D4PG)

D4PG algorithm is an extended version of DDPG. It is a very recent algorithm proposed by (Barth-Maron et al., 2018). The authors came up with four improvements to DDPG, which improves stability, convergence speed and sample efficiency (Barth-Maron et al., 2018). These four improvements are distributional perspective, prioritized experience replay, distributed framework and n-steps return.

First of all, D4PG adapts distributional perspective which was proposed by (Bellemare et al., 2017). The core idea of it is to replace a single Q value, the output of the critic, with a probability distribution. In the other word, instead of

returning a single Q value, the critic network outputs a set of values corresponding to the probabilities of possible Q values. For instance, we have two distributions. They are shown in the diagram below. We assume that the mean of distribution 1 is higher than distribution 2. The horizontal coordinate represents possible Q values, and vertical coordinate represents probability. Previously, we always try to predict an average value (i.e. a single Q value) of an action, which may have a complicated underlying distribution. In this case, distribution 1 is better than distribution 2. However, you may prefer distribution 2, because distributional 1 has a relatively high probability to get very low Q value. Distribution 2 means that at most of the time you will not take a very wrong action. Therefore, distributional perspective considers not only the mean value but also the distribution over possible Q values.




*Distribution 1* *Distribution 2*

The second improvement is prioritized experience replay. Previously, the neural network learns samples which are uniformly sampled from memory buffer. This method tries to improve the efficiency of samples in memory buffer by assigning priorities to samples and then sampling the buffer according to those priorities. Priority is the training loss of a sample. A high training loss means that action is either very good or very bad, so our neural network should train more on these important data.

The last two improvements are distributed framework and n-steps return. Since the limitation of time, these two improvements are not implemented in this project. For details of the distributed framework, please see the paper (Horgan et al., 2018). For details of n-steps return, please see the paper (Hessel et al., 2017). Although only two improvements are implemented, this incomplete D4PG algorithm achieves a better performance than DDPG algorithm.

Additionally, distributional perspective and prioritized experience replay can

also be added into the deep Q-network (DQN) algorithm, and that forms Categorical DQN and Prioritized DQN respectively. As a result, these two RL algorithms are also implemented in this project. Categorical DQN is proposed by (Bellemare et al., 2017), and Prioritized DQN is proposed by (Schaul et al., 2015).

# Chapter 3

# Related work

## 3.1 Some other deep RL algorithms

There are another three widely used deep reinforcement learning algorithms which are not implemented in this project. They are Trust Region Policy Optimization (TRPO), Proximal Policy Optimisation (PPO) and Distributed Proximal Policy Optimisation (DPPO). All of them are recent deep RL algorithms, especially PPO and DPPO published in 2017.

### 3.1.1 Trust Region Policy Optimization (TRPO)

TRPO was proposed by Berkeley researchers in the paper (Schulman et al., 2015). When we update our policy, it is not easy to choose a good step size, since a too small step size slows down training significantly, but a too large step size makes neural network harder to converge. TRPO adds an additional constraint on the policy update, which is maximum Kullback-Leibler (KL) divergence between the old and new policies. This can make sure the policy is not moving too far away from the starting point. Moreover, in order to solve the constrained policy optimization problem, TRPO uses a conjugate gradients method. However, the conjugate gradient method is more complicated than stochastics gradient decent (SGD), which let TRPO less flexible. Also, it has the same issue as Policy Gradient algorithm which is mentioned in chapter 2.2.2. Training does not start until the full episode is completed. This is significantly less data inefficient.

### 3.1.2 Proximal Policy Optimisation (PPO)

PPO was presented by OpenAI team in the paper (Schulman et al., 2017). It is based on actor-critic algorithm which is mentioned in chapter 2.2.3. The main contribution of it is that it changes expression of gradient of policy. Instead of using the gradient of logarithm probability of the action taken which is shown in equation (2.5), PPO applies another objective: clipped surrogate objective. The objective is the ratio between new and old policies scaled by the advantages. This off-policy deep reinforcement learning algorithm PPO can achieve good performance in many complex environments (Schulman et al., 2017). It also can handle continuous action space. However, (Barth-Maron et al., 2018) demonstrates that D4PG algorithm can achieve much better performance than PPO at least in the parkour domain. D4PG algorithm is mentioned in chapter 2.4, and you can see the parkour domain in figure 1.

### 3.1.3 Distributed Proximal Policy Optimisation (DPPO)

DPPO, proposed by (Heess et al., 2017), combines the PPO algorithm with a distributed framework. If you remember, DQN algorithm uses an experience replay buffer to reduce the correlation between samples to improve stability of the learning. This is mentioned in chapter 2.1.2. Distributional framework is doing the similar thing but in a different way.

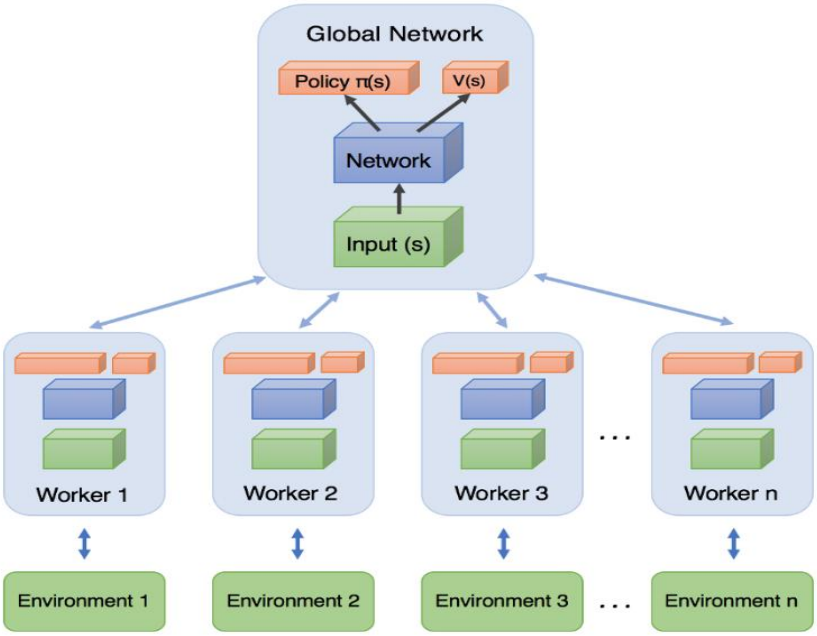



*Figure 8: Distributed Framework (Mnih1 et al., 2016)*

In the framework, there is a global network, and multiple agents each works in their own environment. The global network gathers experiences from several parallel environments to update policy (i.e. global network). Then each agent updates their local policy according to the global network. The whole process of distributional framework is shown in Figure 8. Additionally, this framework is also applied in D4PG algorithm.

## 3.2 Deep Genetic Algorithm (Deep GA)

Deep Genetic Algorithms combines genetic algorithms (GA) with the deep neural network. Recently, several research studies show the applicability of deep genetic algorithms (GA) to large-scale RL problems, and get some really surprising results. (Petroski Such et al., 2018) proposed a simple genetic algorithm can train deep convolutional neural network (CNN) to solve challenging Atari domain and on many games, deep GA outperforms a modern deep RL algorithm deep Q-network (DQN). Moreover, this paper demonstrates that some techniques that improve the power of GA, such as novelty search, also improves the performance of deep GA. Besides, another advantage of deep GA is it can be parallelised more easily than deep RL to achieve a higher training speed (Salimans et al., 2017).

Deep GA algorithms show some advantages when handling RL tasks, but most of the tasks mentioned in those papers are in discrete action space (e.g. Atari domain). In the locomotion control field, we usually need fine control (i.e. continuous action control) of our robot. (Lapan, 2018) implemented the parallelized deep GA on the HalfCheetah environment which is a two-dimensional locomotion control task in continuous action space. Deep GA is able to learn how to stand perfectly but does not figure out running can get more reward. Furthermore, deep RL has more powerful algorithms than DQN, such as DDPG and D4PG, when dealing with continuous action tasks. (P. Lillicrap et al., 2016) and (Barth-Maron et al., 2018) demonstrate that DDPG and D4PG are able to solve the HalfCheetah task.

# Chapter 4

## The designed task

The designed task of this project is a challenging balance task based on a two-wheel robot. The wheeled robot needs to move forward on flat surface and slopes. It seems that to keep the balance of a two-wheels robot by deep reinforcement learning is not a hard task, because there are already many trained models on the internet can do that. However, as far as I know, it is the first time using deep reinforcement learning to learn how a balancing robot moves forward on slopes. The main challenge is that the slope is not as friendly as the flat surface. It is much more difficult to keep balance on a slope especially for a wheeled robot.

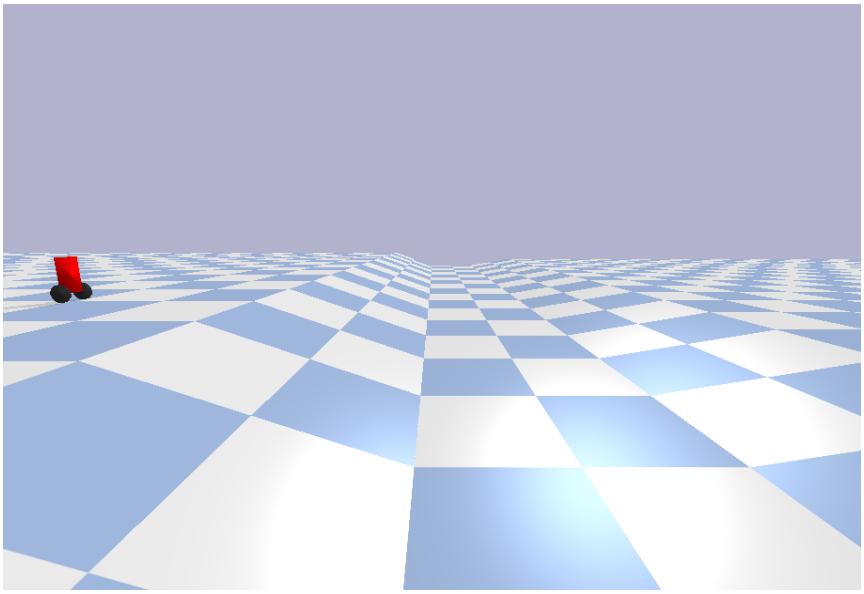


*Figure 9: The designed task*

Figure 9 is the screenshot of the task in simulation. The simulation is powered by PyBullet. It is a Python module for physics simulation for robotics and machine learning.

## 4.1 The robot model

The robot consists of an inverted rectangular body with two cylindrical wheels on each side. Two black wheels are connected by continuous revolute joints. The red body is in a naturally unstable configuration. Thus, the aim is to keep its balance and move forward. The total height of the robot is about 50 centimetres. Each wheel weight 0.4 kilogram, and the body is 1.0 kilogram. The whole robot model is defined in a URDF file which is a widely accepted standard for representing a robot model. It is based on an XML format which can be used in PyBullet. For details of the robot model such as inertia and centre of mass (CoM), please see the URDF file in Appendix A.

## 4.2 Details of the task

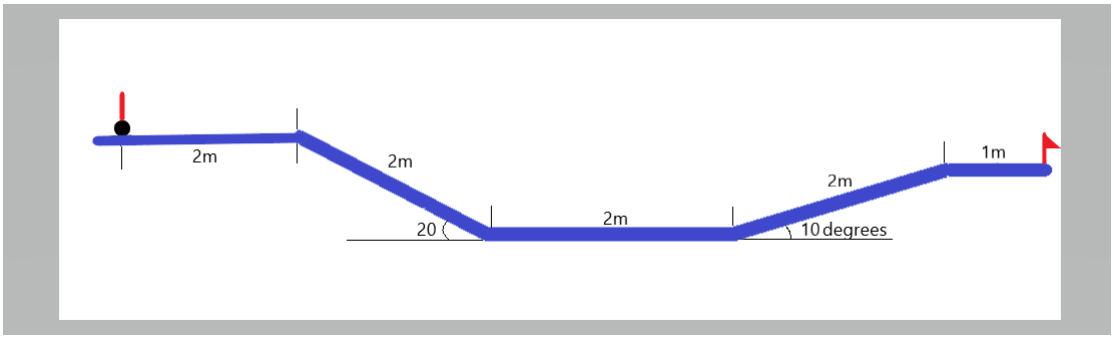



*Figure 10: Section plane of the task*

The section plane of the whole task is shown in figure 10. The robot starts from the left side, keep the balance and move forward on a flat surface which is 2 meters long. Then it needs to go down a 20 degrees slope with the length of 2 meters. Before climbing a 10 degrees slope, there is another 2 meters flat surface. Finally, the robot arrives at the red flag.

## 4.3 The observation of the task

The observation of the designed task consists of five things. They are shown as follows.

- Angle of the red body
- Angular velocity of the red body
- Angular velocity of the wheel
- Linear velocity of the wheel
- Distance

The first observation is the angle between the body and perpendicular direction, and the second one is the angular velocity of that angle. Thus, zero angle means the robot keep upright while a big angle (e.g. $\pm 70°$) means the robot almost fall down. Additionally, the last observation is how far the robot go. If the distance is about 9 meters, the robot arrives at the red flag.

## 4.4 The action of the task

There is only one action in this task, which is the acceleration of two wheels. In order to move forward, two wheels always take the same magnitude of acceleration but in the opposite direction. Moreover, the action range of this task is [-4, 4]. Thus, the maximum acceleration is 4 meter squared per second (i.e. $4\ \mathrm{m}^2/s$). This range is chosen by a small set of experiments which is omitted in this dissertation. A high maximum acceleration will increase the difficulty of the task since velocity change too fast is usually bad for keeping the balance. However, the robot needs a relatively high acceleration to go up a slope. As a result, here is a trade-off, and 4 is one of the optimal choose satisfying both sides.

For DDPG algorithm and D4PG algorithm, the action is really simple which is just one number inside the action range, as they can handle continuous action space. However, for DQN algorithm, we have to discretize the action space.

The action range [-4, 4] is discretized into 21 discrete actions. They are {-4.0, -3.6, -3.2, -2.8, -2.4, -2.0, -1.6, -1.2, -0.8, -0.4, 0.0. 0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6, 4.0}.

## 4.5 The reward function of the task

The reward function is the most important component of the designed task, and how to design a good reward function is always the most difficult thing in reinforcement learning as there is not a general method to do it so far. Generally, there are two kinds of reward function, sparse reward function and continuous reward function. The sparse reward function is easier to define. For example, the agent gets reward 1 when it achieves the goal otherwise gets reward 0. However, sparse rewards slow down learning and make neural network harder to converge because the agent needs to take many actions before getting any reward. Rather than using a sparse reward function, continuous reward function such as a polynomial usually achieves better performance. In this case, you have to design your reward carefully as a slight change of it may seriously affect the performance. However, on the other hand, continuous reward function may decrease the generalisability since you use domain knowledge to design the continuous reward function.

In my task, the reward function is a continuous reward function which is shown in equation (3.1).

- $\mathrm{X}$ is the angle of red body
- $\mathrm{Y}$ is the distance
- $\mathrm{Z}$ is the linear velocity of the wheel

$$\mathrm{reward} = -[\exp(2 * (\mathrm{X} + 0.05)^2) + 1] + 2 * \mathrm{Y} + 4 * \mathrm{Z} \qquad \text{equation (3.1)}$$

There are three parts in the reward function. The first part is for keeping the balance, which is the most complicated part. This part is visualized in figure 11. In figure 11, the horizontal coordinate is $\mathrm{X}$ (i.e. the angle of the body), and vertical coordinate is the reward. From the figure, you can see that the penalty is very small when the angle (in radian) is small. Because the agent needs to

move forward on slopes, a small angle is allowed. But if the angle increases which means that the robot may fall down, the penalty will increase rapidly. Moreover, there is a 0.05 offset in the first part, because the red body tilts forward a little can force the robot to move forward. This small offset is especially useful when the robot is climbing the slope. For example, if the red body is upright when the robot is going up the slope, the body will easily tilt backwards. Then the robot has to move backwards to keep the balance. In that case, the robot will never climb the slope.

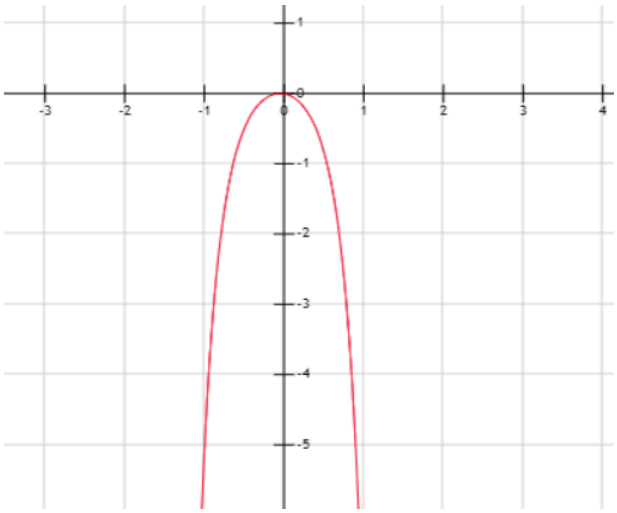



*Figure 11: The first part of reward function*

The second part encourages the robot move as far as possible since a further distance gets a higher reward. The last part is designed for encouraging robot to move relatively faster. If the robot velocity is too low, it cannot climb the slope. However, here is another trade-off, because the robot needs to be as slow as possible when it goes down a slope. The coefficients of those parameters in the reward function are chosen through a set of experiments and finally, the reward function is designed as the equation (3.1). This reward function achieves a very good performance which let the task solved by both DDPG and incomplete D4PG algorithms in about 30 minutes using one CPU.

# Chapter 5

## Experiments and Results

The main objective of this project is trying to solve the designed task, the balancing task, by various deep reinforcement learning algorithms explained in chapter 2. Moreover, a CartPole task provided by OpenAI Gym is used to compare different versions of DQN. Since there are quite a few algorithms are implemented, to be clear, those algorithms are listed below. There are five deep RL algorithms will be compared in this chapter.

- Deep Q-network (DQN)
- Prioritized Deep Q-network (Prioritized DQN)
- Categorical Deep Q-network (Categorical DQN)
- Deep Deterministic Policy Gradient (DDPG)
- Incomplete Distributional Distributed Deep Deterministic Policy Gradient (Incomplete D4PG)

Q learning and Policy Gradient are not on the list because they are obviously not able to solve the designed task. First of all, three algorithms, vanilla DQN, DDPG and incomplete D4PG are tested by the designed task. The result is shown in figure 12.

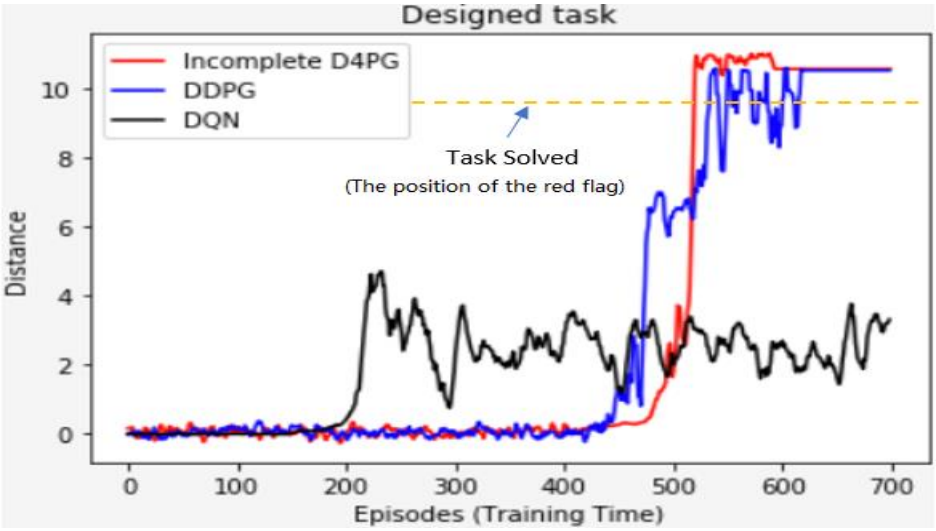



*Figure 12: Results of the designed task*

The horizontal coordinate represents episodes which can be roughly considered as training time. The vertical coordinate represents distance which is how far the robot goes in an episode. Form the figure 12, you can find out incomplete D4PG and DDPG (i.e. red line and blue line) solve the designed task in about 520 and 620 episodes respectively (i.e. about half an hour). Incomplete D4PG is faster and more stable than DDPG due to two improvements, distributional perspective and prioritized experience replay. However, DQN is not able to solve the designed task as the black line fluctuates between 2 and 4 meters after 250 episodes. If you remember figure 10, the first slope starts at the position of 2 meters. As a result, DQN is able to learn how to keep the balance and move forward on the flat surface, but it is not able to learn how to keep the balance on the slope. Figure 13 shows the whole process of the designed task. Since the original image is too colourful, figure 13 is changed to a grey image.

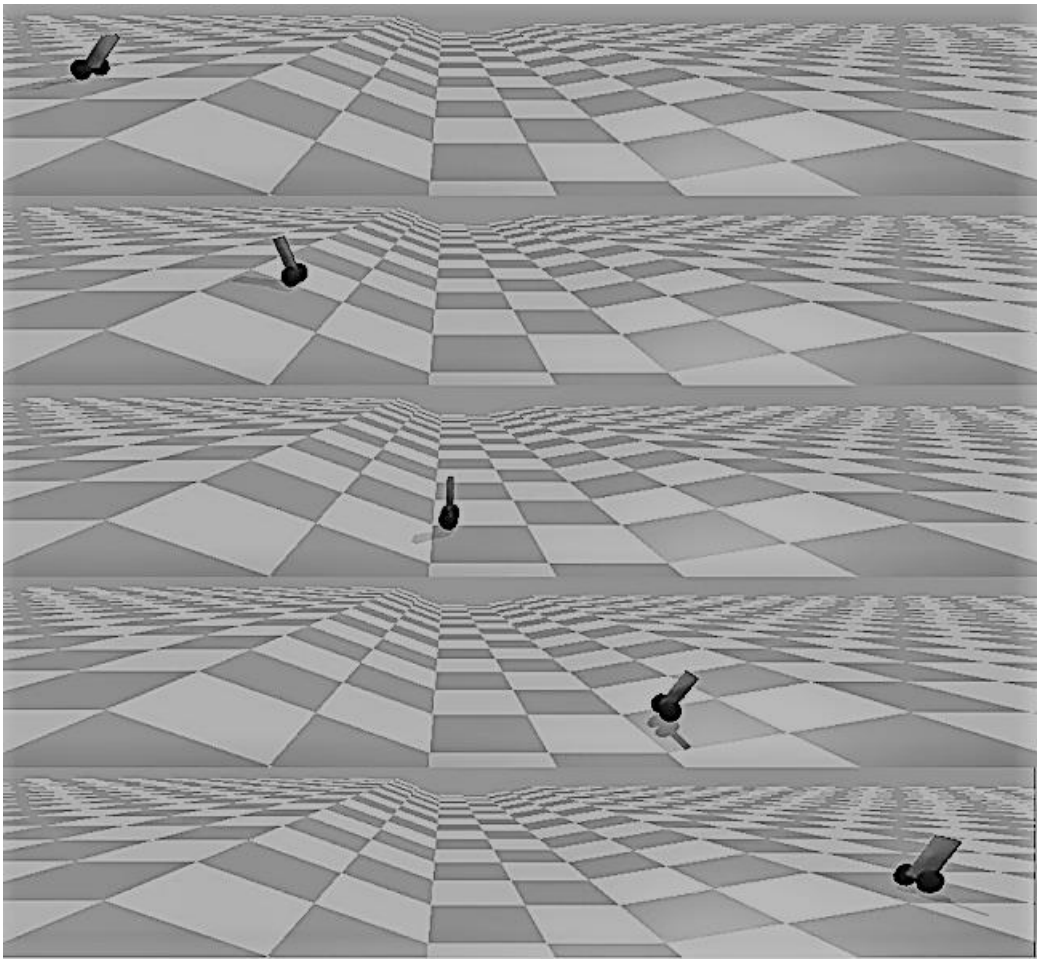


*Figure 13: The whole process of the designed task*

In figure 13, in the beginning, the body of the robot tilts forward to force it moves forward. After 2 meters, the robot slows down in order to keep the balance when going down the first slope. Then the body tilts forward again to force robot move forward and climb the second slope. The full video is uploaded to the YouTube website https://youtu.be/oOzKpN154ng.

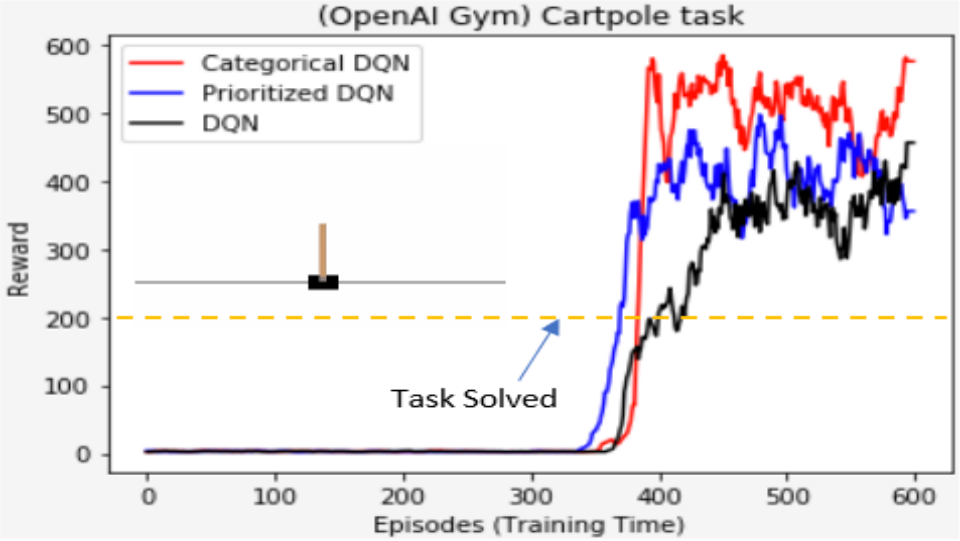



*Figure 14: Results of the CartPole task*

Since the DQN algorithms are not able to solve the designed task, a CartPole task is used to compare three versions of DQN algorithms. CartPole task is much easier than the designed task. The results of the task are shown in figure 14. The horizontal coordinate is episodes, and vertical coordinate is the total reward of an episode. From the figure, you can see that three DQN algorithms are all able to solve the task. the convergence speed of prioritized DQN (i.e. blue line) is higher than vanilla DQN (i.e. black line) due to the improvement, prioritized experience replay, but this improvement does not improve the performance. After about 500 episodes, the total reward of two algorithms (i.e. blue and black lines) are almost the same. However, categorical DQN achieves a better performance than other two version of DQN algorithms, which means that distributional perspective can indeed improve performance.

# Chapter 6

## Further Development

There are several things can do in the future. First of all, implement the complete D4PG algorithm which seems is the best deep reinforcement learning algorithm so far. Secondly, design a more general reward function for the designed task. The reward function used so far is a little complicated which contains many domain knowledge. Using a more precise reward function can increase neural network convergence speed, but the downside is also very obvious. It will decrease the generalisation.


*Figure 15: Handle Robot*

Furthermore, the results of experiments demonstrate that DDPG and D4PG are two powerful and stable deep RL algorithms. They are able to solve the designed task in about half an hour using one CPU. As a result, it is worth implementing those algorithms on lots of more interesting and challenging environments such as a Handle robot. Handle robot also is a two-wheel

balancing robot built by Boston Dynamics in 2017. But it seems can do almost everything with 10 actuated joints. It can walk on the rough slope, jump over the obstacle, down the stairs and transport items. For details, please watch a video on the website https://youtu.be/-7xvqQeoA8c. However, the Handle robot is based on conventional control theory. As a consequence, it is definitely worth trying to implement deep reinforcement learning (e.g. D4PG) on a Handle robot in simulation (if build a real one is too expensive) to see how smart it can be. The Handle robot is shown in figure 15.

# Chapter 7

## Conclusion

This project implements various reinforcement learning algorithms from scratch, including state-of-the-art approaches DDPG and D4PG (incomplete). Solve a challenging balance task by a novel approach (i.e. deep RL) instead of conventional control theory. Moreover, we compare five different deep reinforcement learning algorithms and proves that D4PG is the best one among them. However, there are still many things can do in the future which we have already discussed in Chapter 6.

# Appendix A

## URDF of the robot model

```
<robot name="balance">

  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <material name="red">
    <color rgba="1 0 0 1"/>
  </material>

  <material name="black">
    <color rgba="0.2 0.2 0.2 1"/>
  </material>

  <link name="torso">
    <visual>
      <geometry>
        <box size="0.2 0.05 0.4"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
      <material name="red"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.2 0.05 0.4"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
    </collision>
    <inertial>
      <mass value="0.8"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
    </inertial>
  </link>

  <link name="l_wheel">
    <visual>
      <geometry>
```

```
      <cylinder length="0.02" radius="0.1"/>
    </geometry>
    <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
    <material name="black"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.02" radius="0.1"/>
    </geometry>
    <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
    <contact_coefficients mu="2.5" />
  </collision>
  <inertial>
    <mass value="0.4"/>
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </inertial>
</link>

<link name="r_wheel">
  <visual>
    <geometry>
      <cylinder length="0.02" radius="0.1"/>
    </geometry>
    <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
    <material name="black"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.02" radius="0.1"/>
    </geometry>
    <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
    <contact_coefficients mu="2.5" />
 <origin rpy="0 0 0" xyz="0.12 0.0 0.1"/>
</joint>

</robot>
```

# Appendix B

## 1. The usage of the Git repository

Each python file implements one reinforcement learning algorithm on either Balancing task (i.e. the designed task) or CartPole task. There are 10 python files totally including 8 RL algorithms. These eight implemented RL algorithms are Q Learning, Policy Gradient, Actor-Critic, Deep Q-network (DQN), Prioritized DQN, Categorical DQN, Deep Deterministic Policy Gradient (DDPG), and Incomplete Distributional Distributed Deep Deterministic Policy Gradient (Incomplete D4PG). All implementations are from scratch. The main libraries used in this project are PyTorch, a deep learning framework, and PyBullet, a physic engine.

## 2. Dependencies

- PyTorch v0.4.1
- PyBullet 2.1
- OpenAI Gym (Classic Control)
- Python 3.5

## 3. Git repository address

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/zxl661.git

# List of References

(Bellemare et al., 2017) Marc G Bellemare, Will Dabney, and Remi Munos. A distributional perspective on reinforcement learning. In International Conference on Machine Learning, pp. 449–458, 2017.

(Lillicrap et al., 2016) 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. In International Conference on Learning Representations, 2016.

(Barth-Maron et al., 2018) Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, Timothy Lillicrap. Distributional distributed deterministic policy gradient. In International Conference on Learning Representations, 2018.

(Hessel et al., 2017) 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.

(Horgan et al., 2018) Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. International Conference on Learning Representations, 2018.

(Schulman et al., 2015) J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. arXiv preprint arXiv:1502.05477, 2015.

(Schulman et al., 2017) John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347, 2017.

(Heess et al., 2017) Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin Riedmiller, David Silver. Emergence of Locomotion Behaviours in Rich Environments. arXiv preprint arXiv:1707.02286, 2017.

(Mnih et al., 2016) Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. arXiv preprint arXiv:1602.01783, 2016.

(Petroski Such et al., 2018) Felipe Petroski Such, Vashisht Madhavan, and others. Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. arXiv preprint arXiv:1712.06567, 2018.

(Salimans et al., 2017) Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. arXiv preprint arXiv:1703.03864, 2017.

(Sutton et al., 1998) Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1998.

(Bellemare et al., 2017) Bellemare, Dabney and Munos. A distributional perspective on reinforcement learning. In ICML. 2017.

(Peters et al., 2005) Peters, J., Vijayakumar, S., and Schaal, S. (2005). Natural actor-critic. In 16th European Conference on Machine Learning, pages 280–291.

(Silver et al., 2014) Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin. Deterministic policy gradient algorithms. In ICML, 2014.

(Lapan, 2018) Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Birmingham: Packt, 2018. Print.

(Tsitsiklis et al., 1997) J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. IEEE Transactions on Automatic Control, 42(5):674-690, 1997.

(Mnih et al., 2015) Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

(Ioffe et al., 2015) Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

(Schaul et al., 2015) Schaul, Quan, Antonoglou and Silver. Prioritized experience replay. In Proc. of ICLR. 2015.