修　　士　　論　　文

Improving the Action Branching Architecture for Multi-dimensional Hybrid Action Spaces in Deep Reinforcement Learning

(深層強化学習における多次元混合行動空間のための行動分岐アーキテクチャの改善)

指導教員　　鶴岡 慶雅 教授

東京大学大学院情報理工学系研究科
電子情報学専攻

氏 名　　**48186452** 彭来鴿

提 出 日　　令和 **02** 年 **1** 月 **30** 日

Abstract

Recently deep reinforcement learning (DRL) has achieved several great successes in traditional chess games such as chess, shogi and Go, and in 2D environment games such as Atari 2600 and StarCraft II. DRL solves the complicated state-space problem in 2D environment games by using frames as inputs. However, 3D environment games such as ViZDoom and Minecraft still challenge researchers for their partial observability and more complex action spaces. This thesis aims to address a complicated and general action space, the multi-dimensional hybrid action-space problem, in 3D environment games.

There is not much research on the complex action-space problem. Widely used value-based DRL algorithms such as Deep Q-Network (DQN) usually deal with one-dimensional discrete action spaces. As continuous action spaces are also general and important in control systems, policy-based DRL algorithms such as deep deterministic policy gradients (DDPG) and proximal policy optimization (PPO) were proposed to address this problem. However, both these discrete action spaces and continuous action spaces are one-dimensional spaces with only one degree of freedom (DOF). In 3D environment games, even in the real world, multi-dimensional action spaces with more than one DOFs are more general. Branching dueling Q-network (BDQ) used the action branching architecture to solve a multi-dimensional discrete action-space problem. However, when there are continuous action spaces in the DOFs, BDQ can not deal with this problem.

In this thesis, we aims to address the multi-dimensional discrete-continuous hybrid action space problem by improving the action branching architecture. The branching PPO (BPPO) algorithm was proposed as a combination of the action branching architecture and the PPO learning algorithm. In order to improve the sample efficiency and accelerate the learning process, a method of applying human demonstration data in BPPO was proposed. We evaluated four different models in two different environments to solve different tasks in a Minecraft simulator. By comparing the training curves, average rewards, and testing curves, we show that the proposed method BPPO outperforms other baselines in both learning speed and final performance. Experiments with human demonstration also show that BPPO has better compatibility with human demonstration data and better sample efficiency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Recent years, there have been several huge successes in the field of game AI with deep reinforcement learning (DRL). In 2013, Google DeepMind developed an algorithm called Deep Q-network (DQN) [1] that outperformed human players in most of the Atari 2600 games. The publication of DQN has a far-reaching impact on the field of game AI. There have been incessant numbers of new reports on state-of-art solutions to some new games or new adaptions of the DQN algorithms. In 2016, AlphaGo [2] defeated the top human player. AlphaGo Zero [3] was proposed by DeepMind in 2017. Compared to AlphaGo, AlphaGo Zero can master the game of Go without human knowledge. In 2019, DeepMind developed MuZero [4] that masters Atari, Go, Chess and Shogi by planning with a learned model. Also in 2019, AlphStar [5] was developed by DeepMind and achieved grandmaster level in the game of StarCraft II by using multi-agent reinforcement learning. So far deep reinforcement learning in game AI has shown great power in both traditional chess games and 2D environment video games.

DRL has wide applications and can be seen as a means to achieve Artificial General Intelligence (AGI). Although reinforcement learning (RL) has been studied for decades, the application of RL is mostly limited in simple toy tasks. Traditional RL algorithms such as Q-learning [6], REINFORCE [7] and SARSA [8] have not been widely applied until the boom of the deep learning technology. DRL combines deep neural networks and the RL learning algorithms together, making RL more general and useful.

Deep Q-Network (DQN) [1, 9] is a typical value-based DRL algorithm and it outperforms human players in most Atari games. There are many improvements and extensions of the original DQN to improve the performance of DQN and solve specific problems in DRL, such as the partial observability problem and the sample efficiency problem when applying this DRL algorithm. Double DQN [10], dueling DQN [11], distributed DQN

[12] and prioritized experience replay [13] are well-known extensions. Hausknecht, M., and Stone, P. [14] proposed a deep recurrent Q-learning (DRQN) which combines the LSTM layer [15] with the original DQN, outperforming standard 4-frame DQN and augmented 10-frame DQN in the game of flickering Pon. Hester et al. [16] proposed an algorithm called Deep Q-learning from Demonstrations (DQfD) to apply the demonstration data in DQN. In tasks requiring complicated policies, hierarchical DRL is also introduced.

Besides the DQN family, policy-based DRL algorithms also achieve the state-of-the-art. Well-known policy-based DRL algorithms such as asynchronous advantage actor-critic (A3C) [17], deep deterministic policy gradients (DDPG) [18] and proximal policy optimization (PPO) [19] are also widely used. Compared to the DQN family, policy-based DRL algorithms are more flexible with action spaces and can deal with continuous action spaces.

Recent DRL algorithms mainly address the complexity problem of the state spaces in RL by using deep neural networks to approximate state values or policies. As the development of DRL advances, many challenging games with much more complicated state spaces such as ViZDoom have been overcome; however, there still remain challenges to solve games with both complex state spaces and action spaces such as Minecraft, which requires better sequence decisions. Previous value-based research mainly focuses on the state spaces and assumes that the action space is discrete and has only one dimension.

However, in robot control tasks, the angle values of the robot's joints may be continuous. For such continuous action spaces, there are mainly two solutions: discretize the action space and utilize value-based algorithms or directly generate continuous values by applying policy-based algorithms such as DDPG. Also, a robot may have more than one joint, and these joints are independent with each other. Therefore, it is necessary to control these degrees of freedom (DOFs) at the same time. This can be defined as a multi-dimensional action space. To address such problems, Tavakoli et al. [20] proposed an action branching architecture to use a separate action branch for each action dimension.

There are other research efforts to address complicated action-space problems. Xiong et al. [21] proposed a parametrized DQN (P-DQN) method which can learn with a one dimension discrete-continuous hybrid action space by combining the DQN algorithm and the DDPG algorithm. Vinyals et al. [22] designed a parametrized action space for the game StarCraft II in which each action contains a function identifier and a sequence of arguments. They made a baseline with A3C as the learning algorithm and represented the policy in an auto-regressive manner by using the chain rule.

1.2 Motivation and contribution

This work aims to address a more general and complicated action-space problem, the multi-dimensional hybrid action space. Multi-dimensional means that there are more than one degree of freedom in the action space, and hybrid means that in all DOFs, some action values are discrete and some values are continuous. This kind of action space is quite general in complicated tasks, even in the real world. By defining this kind of action space, in one time step, an agent can do more primitive actions, which will improve the efficiency to learn a good policy.

Considering the complexity of the action space, we assume that each action dimension is independent to each other and utilize the action branching architecture proposed by Tavakoli et al. [20] to generate separate actions for each dimension. For continuous action spaces, discretization may violate the mechanism of the action space, and generate too many discrete points. Thus, in this work, we improved the action branching architecture to address the multi-dimensional hybrid action-space problem by applying the PPO learning algorithm. we call the proposed model as Branching PPO (BPPO). We also made use of non-pixel observations to augment the state representation. Categorial distributions and Gaussian distributions are used to sample discrete actions and continuous actions respectively. We also proposed a new learning method to use supervised learning during a pre-training phase by using human demonstration data to quickly start the training by providing a better start policy and accelerate learning process.

As for the experiment environment, We take Minecraft, a 3D environment sandbox game, as the platform for its challenging environment, high-dimension representation, complex action space and hierarchical item systems. Compared to 2D environment games, DRL in 3D environment games still challenges researchers. 3D environment games create complex 3D world for game players. The complexity gives 3D environment games the advantage of resembling the real world, which makes it a suitable testbed for solving problems in the real world. We used MineRL [23], a Minecraft simulator, and took a navigation task and a treechop task as training domains. MineRL provides a real game environment in which the map is very large and contains all the possible entities in it. Therefore, it is very hard for the agent to explore the environment with normal one dimension action space. What is more, besides the discrete movement actions, the agent should also control some continuous camera parameter to control the first-person view. Therefore, this environment is suitable for evaluating the performance of the proposed model.

This work compares the proposed method BPPO with other baselines in two environments. In the navigation task with dense rewards, BPPO learns faster and achieves better performance. In the treechop task with extremely sparse rewards, BPPO shows better compatibility with human demonstration data and achieves higher scores. The experi-

ments show efficiency of our proposed method.

1.3 Overview of this thesis

In following sections, firstly, we will give introduction to the background knowledge of reinforcement learning and explain some value-based RL algorithms and policy-based RL algorithms in chapter 2.

In chapter 3, we will introduce several methods related to my work, including the action branching architecture.

In chapter 4, we will firstly introduce the details of the experiment environments, and then the main architecture and learning methods used in this work. Then we will talk about my detailed experiment setting, and finally we will show the experiment results and our analysis.

Chapter 5 is the conclusion of this thesis.

Chapter 2

Background

2.1 Reinforcement learning

Reinforcement learning [24] is an area of machine learning which learns what to do—how to map situations to actions—from experiences so as to maximize a numerical reward signal. Reinforcement learning focuses on the interaction between an agent and the environment. An agent may refers to a robot, a control system or some other kinds of system. More specifically, at time t, the agent observes an observation o or a state s from the environment, and then chooses an action a and receives a reward r from the environment. The environment makes a transition to another state $s\prime$ because of the effect of action a, and then the agent repeats to choose a new action at the next time step $t+1$. In reinforcement learning, the agent at time t always tries to maximize the expectation of the cumulative rewards obtained from time step t. The whole process is described as a Markov Decision Process (MDP).

2.1.1 Markov decision process (MDP)

In a Markov Decision Process, there are several definitions to describe the terminology:

- S : A set of states.
- A : A set of actions.
- $P(s' \mid s, a)$: The probability that a state s at time t will transit to a state s' at time $t+1$ when given an action a.
- $R_a(s, s')$: The immediately received reward after a state s transits to a state s' when given an action a.

- $\gamma \in [0, 1]$: The discount factor, represents the priority attention of the future rewards and the present reward. Larger discount factor means the agent focuses more on long-term rewards while smaller discount factor means the agent pay more attention to the short-term rewards.

According to the definitions explained above, the cumulative reward R is calculated as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0} \gamma^k r_{t+k+1}. \tag{2.1.1}$$

A state value function is defined as

$$V(s) = \mathbb{E}[R_t \mid S_t = s]. \tag{2.1.2}$$

The agent tries to choose a series of actions that maximize the expected rewards when starting from every state s, in order to achieve the max rewards it can obtain from whatever state it is in. A policy π is defined as

$$\pi(a \mid s) = P[A_t = a \mid S_t = s]. \tag{2.1.3}$$

A optimal policy π^* represents an expected policy that achieves max value function $V(s)$ given any starting state s. We use $V^*(s)$ to represent the optimal state value function of the state s. The Bellman equation contributes to an iterative way to calculate the state value function, here we show the derivation of the Bellman equation:

$$\begin{aligned} V(s) &= \mathbb{E}[R_t \mid S_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...] \\ &= \mathbb{E}[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...)] \\ &= \mathbb{E}[r_{t+1} + \gamma R_{t+1}] \\ &= \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]. \end{aligned} \tag{2.1.4}$$

The final equation showing above gives us the Bellman equation. By removing the expectation, we get the following equation:

$$V(s) = r_{t+1} + \gamma \sum_{s' \in S} P(s' \mid s, a) V(s'). \tag{2.1.5}$$

For optimal state value function $V^*(s)$, we have $V^*(s) = \mathbb{E}[V(s)]$, thus the Bellman equation can be rewrite as:

$$V^*(s) = r_{t+1} + \gamma V^*(s_{t+1}). \tag{2.1.6}$$

A state value function following a policy π is denoted as

$$V_\pi(s) = r_{t+1} + \gamma \sum_{s' \in S} P(s' \mid s, a) V_\pi(s'), \tag{2.1.7}$$

which means actions are taken according to the policy π from the state s_t. The agent aims to learn a policy π which achieves $V^*(s)$ as $V^*(s) = \max_\pi V_\pi(s)$.

Reinforcement learning is widespread used in game AI problems because games are easier to be modeled as MDPs than many other artificial intelligence problems for that most games have explicit rules. It is imperative to recognize that reinforcement learning has a deep influence in game AI research. Besides game AI, reinforcement learning is also popular in control theory and multi-agent systems.

2.1.2 Value-based methods

There are mainly two kinds of methods in reinforcement learning: value-based and policy-based methods. As indicated in the name, value-based methods try to approximate the optimal state value function $V^*(s)$ to learn an optimal policy π^* indirectly, while policy-based methods try to approximate $\pi^*(a \mid s)$ directly. For value-based methods, we will introduce a most well-known algorithm, Q-learning, its deep version deep Q-network, and several variants of deep Q-network.

2.1.2.1 Q-learning

Q-learning [6] is a well-known reinforcement learning method, aiming to learn how to act optimally in controlled Markovian domains.

A state-action value function $Q(s, a)$ is defined as

$$Q(s, a) = \mathbb{E}[R_t \mid S_t = s, A_t = a]. \tag{2.1.8}$$

$Q^*(s, a)$ is used to denote the optimal state-action value function given a state s and an action a. According to the definition of the $Q(s, a)$, we can know that:

$$V(s) = \sum_{a \in A} P(a \mid s) Q(s, a). \tag{2.1.9}$$

We can also apply the Bellman equation in the state-action value function by using the same derivation process with the state value function $V(s)$, then we will get the following Bellman equation:

$$Q^*(s_t, a_t) = r_t + \gamma \max_{a'} Q^*(s_{t+1}, a'). \tag{2.1.10}$$

Q-learning aims to approximate the optimal state-action value function $Q^*(s, a)$ by using a iterative method called Q-table which is a table that records the value of $Q(s, a)$ for all (s, a) pairs given $a \in A$ and $s \in S$. In the beginning of the learning process, Q-learning randomly initializes the state-action value function $Q(s, a)$ for each (s, a) pair. Then Q-learning learns from experiences with exploration of all the possible state-action (s, a) pairs to gradually correct their values to approach the expected optimal value $Q^*(s_t, a_t)$.

The process of the Q-learning algorithms is as follows:

- **1.** Initialize the Q-table arbitrarily.
- **2.** For a state s at time step t, choose an action a with highest state-action value $Q(s, a)$ in the current table, observe the next state s' at time $t + 1$ and the reward r.
- **3.** According to the Bellman equation, the target value of $Q(s, a)$ is $r+\gamma \max_{a'} Q(s, a')$, then we iteratively improve the estimate value in the table by the following rule:

$$Q(s, a) := Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s, a') - Q(s, a)], \tag{2.1.11}$$

  where α is the learning rate.
- **4.** Repeat **2** and **3** until the state-action value function $Q(s, a)$ converges.

In Q-learning with Q value table to store the value for each (s, a) pair, it needs $S \times A$ memory. However, as the set of states or the set of actions becomes bigger or even infinite, it is hard to create such a Q-table to store all the state-action values. Therefore, Q-learning is limited to solve MDPs with finite states and finite actions. However, many environments such as Chess, Go, gym AI environments and Atari games in the game field, and some other real world environments such as robot arms control, have infinite states spaces and it is impossible to create a Q-table to store all the state-action values. In order to solve this kind of problem, there comes the inspiring deep Q-network (DQN).

2.1.2.2 Deep Q-Network (DQN)

Deep Q-Network [1, 9] is a value-based reinforcement learning algorithm. It is a method that combines reinforcement learning with a deep neural network to approximate the state-action value function $Q(s, a)$ instead of using a Q-table to store all values to address the problem in infinite states environments. The notable success in Atari 2600 showed the potential of the DQN algorithm.

A DQN takes an observation s as the input to the deep neural network with parameter θ, and calculates the state-action value function as:

$$Q(s, a) = f(s, a; \theta), \tag{2.1.12}$$

for any possible state-action pairs (s, a). Usually inputs s of the DQN are pixel features of frames in the game, the output of the neural network is a vector with size $\mid A \mid$ which is the number of accessible actions, each element in the output vector represents the Q value for each action a.

In DQN, we use gradient descent to update the network parameter θ. The same as Q-learning, the target state-action value function of (s, a) is:

$$\hat{y}_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_t), \tag{2.1.13}$$

and the estimation of the network is:

$$y_t = Q(s, a; \theta_t). \tag{2.1.14}$$

Then we could create an objective loss function as:

$$\begin{aligned} L_t(\theta_t) &= \mathbb{E}_{(s,a,r,s')\sim U(D)}[(\hat{y}_t - y_t)^2] \\ &= \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_t) - Q(s, a; \theta_t))^2], \end{aligned} \tag{2.1.15}$$

in which γ is the discount factor and θ_t is the parameter of the network at time step t. From the loss function shown above we can see that the change of the parameter θ will also affect the value of the target state-action value $\hat{y}$, thus making it unstable to do the gradient descent. The solution of this problem is to use another separate network θ^- to calculate the target $\hat{y}$. We can then rewrite the objective loss function as:

$$L_t(\theta_t) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_t^-) - Q(s, a; \theta_t))^2]. \tag{2.1.16}$$

The target network parameters θ^- updates by copying from the θ only every C steps in order to eliminate the correlations between the network $Q(\theta)$ and the target network $Q(\theta^-)$. This modification makes the algorithm more stable compared to the original one.

In reinforcement learning, the agent acts in a time sequence, thus the states in the same sequence are related to each other and make the learning process unstable. A key idea to break the correlations presenting in the sequence of past observations in DQN is the experience replay buffer [25]. When the agent interacts with the environment, the agent creates a memory buffer to store the (s, a, r, s') tuples as experiences in the buffer. When updating the model parameter θ, we randomly sample the learning data batches from the memory buffer, because randomly choosing the data reduces the influence of the correlations. Empirically, we eliminate the expectation in the equation (2.1.16) by using batch sampling from the experience replay buffer, and therefore we get the following objective loss function:

$$L_t(\theta_t) = (r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_t^-) - Q(s, a; \theta_t))^2. \tag{2.1.17}$$

Unlike Q-learning that updates the Q-table every time step, the agent chooses an action according the Q value predicted by the network θ every time step, but updates the parameter every N time steps by batch sampling past experiences from the replay buffer.

Algorithms that combine traditional reinforcement learning and deep learning are called deep reinforcement learning (DRL). As the development of DRL, there are some improved variants of DQN such as Double DQN [10], Dueling DQN [11] and Distributed DQN [12]. We call them DQN-liked algorithms for that those variants are proposed based on the DQN algorithm.

2.1.2.3 Exploration

In reinforcement learning, an agent learns from its past experience. However, if the experience leads to bad results in the beginning, it may cause even worse performance in later learning process. The key point to address this problem is to explore the environment as much as possible in order to get more experiences and find a global optimal solution. A widely used exploration method in DQN-liked algorithms is called ϵ-greedy.

Generally, the agent's policy $\pi(a \mid s)$ in DQN-liked algorithms is to choose the action a with highest state-action value $Q(s, a)$. To encourage the agent to explore the environment, we also make the agent to choose random actions rather than following the policy. ϵ-greedy algorithm defines a threshold ϵ that satisfy $0 <= \epsilon <= 1$ to decide whether to act according to the policy or sample a random action. For each time step, we generate a random number a between 0 and 1, if $a > \epsilon$, select an action according to the policy, otherwise sample a random action from the action space $\mathcal{A}$.

In the beginning of the learning process, we encourage the agent to do random exploration to get more experiences from the exploration by setting the ϵ to a high value such as 1. As the agent learns from its past experiences, it learns a policy to get higher reward from the environment, thus we want to make the agent to follow the policy as the learning process goes on. We anneal the ϵ value from 1 to a smaller value, usually it is 0.1, until the policy converges. There are different annealing schedules can be used in ϵ-greedy algorithm, the most common one is the linear schedule.

One thing to be noticed is that we use ϵ-greedy algorithm during the training process; however, we use a greedy algorithm when testing our models. Greedy algorithm means that the agent completely follows the policy.

Exploration is very important in reinforcement learning especially in those complicated environments where it is prone to achieve local optimum. There are much research about the exploration problem and many results show that facilitating the exploration in reinforcement learning improves the agent performance.

2.1.2.4 Double DQN

The popular DQN algorithm is shown to overestimate action values in some conditions. Hasselt et al. developed Double DQN (DDQN) in order to solve the overestimation problem in some Atari 2600 games and the algorithm is shown to be able to be generalized to work with large-scale function approximation [10]. In the vanilla DQN method, the target estimation of the state-action value $Q(s, a)$ is as:

$$\hat{y}_t = r_t + \gamma \max_{a'} Q^-(s_{t+1}, a'; \theta_t). \tag{2.1.18}$$

In this equation, it firstly calculates the $Q^-(s_{t+1}, a'; \theta_t)$ value for all possible actions a', chooses one action with the highest value, and then use the Q value of the selected action to calculate the target estimation. This uses the same value to both choose and evaluate the action, resulting overestimation of the state-action value. In order to prevent this problem, one solution is to decouple the selection part from the evaluation part.

Double Q-learning [26] tries to solve this problem by separating two different state-action value functions θ and θ'. For each update, one set of parameter is used to select the greedy action and the other one is used to calculate the state-action value of that chosen greedy action. To make the difference between original Q-learning and the Double Q-learning clear, we firstly rewrite the above target estimation of the original Q-learning as:

$$\hat{y}_t^Q = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t). \tag{2.1.19}$$

The Double Q-learning target estimation can be written as:

$$\hat{y}_t^{DoubleQ} = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta'_t). \tag{2.1.20}$$

In the argmax operation, the Double Q-learning uses the current online parameter θ to select a greedy action a. However, it uses another set of parameter θ' to evaluate the value of that selected action a. The current online parameter θ is updated by the following loss function:

$$L = \hat{y}_t^{DoubleQ} - Q(s, a; \theta_t). \tag{2.1.21}$$

The second set of parameter θ' can be updated by symmetrically switching with θ in next time step.

When apply the idea of Double Q-learning to DQN, the target network θ^- in DQN provides a candidate for the second set of parameter θ' and thus there is no need to use another new neural network. Hasselt et al. refered to the combination algorithm as Double DQN from both Double Q-learning and DQN [10]. Then we can get the target estimation for Double DQN from Double Q-learning as follows:

$$\hat{y}_t^{DoubleDQN} = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t^-). \tag{2.1.22}$$

In this equation, compared to that in the Double Q-learning, we replace the second set of parameter θ' with the target network θ^- to evaluate the current online policy.

In Double DQN, we use the online network θ to select the argmax action and use the target network θ^- to evaluate the value of this action. The online network θ is updated for every N time steps while keeping the target network θ^- to stay unchanged. The target network θ^- copies the parameter from the online network θ every C time steps.

This version of Double DQN has the minimal change to the original DQN and improves the agent performance, and is widely used instead of the original DQN.

2.1.2.5 Dueling DQN

The original DQN algorithm uses convolutional neural networks, the Double DQN algorithm improves the loss function of the original DQN to prevent the overestimation problem but still uses the same network architecture. Wang et al. [11] proposed a new neural network architecture called Dueling DQN which achieved the state-of-the-art performance in the Atari 2600 domain.

In reinforcement learning, we have the state value function $V(s)$ for each state s and the state-action value function $Q(s,a)$ for each state-action pair (s,a). Wang et al. defined another important quantity called advantage function from the state value function and the Q value function:

$$A(s,a) = Q(s,a) - V(s). \tag{2.1.23}$$

We can rewrite equation (2.1.9) as:

$$V(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q(s,a)], \tag{2.1.24}$$

thus we have:

$$\begin{aligned}\mathbb{E}_{a\sim\pi(a|s)}[A(s,a)] &= \mathbb{E}_{a\sim\pi(a|s)}[Q(s,a)] - \mathbb{E}_{a\sim\pi(a|s)}[V(s)] \\ &= V(s) - V(s) \\ &= 0.\end{aligned} \tag{2.1.25}$$

From previous introduction we know that $V(s)$ evaluates how good it is given a particular state s, and how much value an agent can get by taking a particular action a when in this state s. The advantage function subtracts the state value from the Q value, giving a relative measurement of the importance of each action.

Figure 2.1 shows the architecture of the dueling network. Unlike conventional single stream architecture to directly estimate the $Q(s,a)$ value, dueling network separate it into two streams to estimate the state value $V(s)$ and the advantage function $A(s,a)$. As shown in the Figure 2.1, those two streams share the same convolutional neural network.

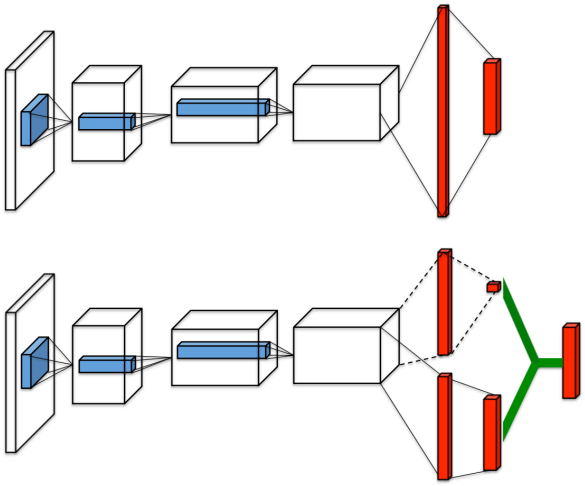


Figure 2.1: The popular single stream neural network use in DQN and Double DQN (top) and the dueling network architecture (bottom).

Suppose that the parameter of the convolutional neural network is θ, the parameter of the advantage function is α, and the parameter of the state value function is β. The final $Q(s, a)$ value is estimated by adding up as:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha), \tag{2.1.26}$$

and this function can then be used in value-based algorithms such as the DQN algorithm and the Double DQN algorithm.

However, an unidentifiable problem of calculating Q value by the equation (2.1.26) is that when given $Q(s, a; \theta, \alpha, \beta)$ we can not recover $V(s; \theta, \beta)$ and $A(s, a; \theta, \alpha)$ uniquely. To solve this issue of identifiability, the solution is to force the advantage function estimator to be zero at the chosen action. That can be done by subtracting the max advantage function value from the advantage function as:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \max_{a' \in A} A(s, a'; \theta, \alpha)). \tag{2.1.27}$$

Now, for the action $a^* = \arg\max_{a' \in A} Q(s, a'; \theta, \alpha, \beta)$, we have $A(s, a^*; \theta, \alpha) = 0$, thus we know that $V(s; \theta, \beta) = Q(s, a^*; \theta, \alpha, \beta)$.

An alternative way is to replace the max operator by an average operator as:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha))). \tag{2.1.28}$$

Although the equation (2.1.28) loses the semantic meaning of $V(s;\theta,\beta)$ and $A(s,a;\theta,\alpha)$ because it now subtracts the mean of the advantage function value instead of the max value and makes it unable to recover V and A, it still increases the stability of the learning process because the advantage value function only needs to change as fast as the mean value, instead of changing as the max value. This equation provides better performance.

As we get the estimation of $Q(s,a;\theta,\alpha,\beta)$ from the dueling architecture, it can generalize learning across actions without changing current deep reinforcement learning algorithms. For an example, we can use this Q value in algorithms such as Double DQN to get the target estimation $\hat{y}_t$ as:

$$\hat{y}_t^{DoubleDQN} = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t, \alpha_t, \beta_t); \theta_t^-, \alpha_t^-, \beta_t^-). \tag{2.1.29}$$

The training method of this dueling network is also simple by applying back-propagation. The parameters θ, α, β are updated automatically without any extra modifications.

2.1.3 Policy-based methods

As indicated as its name, policy-based algorithms optimizes the policy $\pi(a \mid s) = P(a \mid s)$ directly. In Value-based algorithms, the policy is a greedy policy obtained from the $Q(s,a)$ value as:

$$\pi(a \mid s) = \arg\max_a Q(s,a) \tag{2.1.30}$$

in an indirectly way by estimating the state-action function. However, in policy-based algorithms, we estimate the distribution of the action space $\mathcal{A}$ given the current state s. We also call policy-based methods as policy gradient methods [27, 28].

Suppose a policy is parameterized by the parameter θ, our target is to maximize the cumulative reward G:

$$\max_\theta G = \mathbb{E}[\sum_{t=0}^{H} r(s_t) \mid \pi_\theta)]. \tag{2.1.31}$$

Let τ denotes a state-action sequence $s_0, a_0, s_1, a_1, ..., s_H, a_H$, then we define the cumulative reward R for τ is:

$$R(\tau) = \sum_{t=0}^{H} r(s_t, a_t). \tag{2.1.32}$$

Then we can rewrite equation (2.1.31) as:

$$\begin{aligned} G(\theta) &= \mathbb{E}[\sum_{t=0}^{H} r(s_t, u_t); \pi_\theta] \\ &= \sum_\tau P(\tau;\theta) R(\tau). \end{aligned} \tag{2.1.33}$$

In this new notation, the goal is to find an optimal θ that:

$$\max_{\theta} G(\theta) = \max_{\theta} \sum_{\tau} P(\tau;\theta)R(\tau). \tag{2.1.34}$$

For equation (2.1.34), take the gradient of the parameter θ:

$$\begin{aligned}
\nabla_\theta G(\theta) &= \nabla_\theta \sum_{\tau} P(\tau;\theta)R(\tau) \\
&= \sum_{\tau} \nabla_\theta P(\tau;\theta)R(\tau) \\
&= \sum_{\tau} \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_\theta P(\tau;\theta)R(\tau) \\
&= \sum_{\tau} P(\tau;\theta) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} R(\tau) \\
&= \sum_{\tau} P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau).
\end{aligned} \tag{2.1.35}$$

To calculate the policy gradient, we usually approximate the equation (2.1.35) with empirical estimate by sampling m different trajectories τ under the policy π_θ:

$$\nabla_\theta G(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)};\theta) R(\tau^{(i)}). \tag{2.1.36}$$

This equation is valid even when R is discontinuous or unknown, and when the sample space of trajectories is a discrete set. Because we want to maximize the equation (2.1.34), the gradient will increase the probability of trajectories with positive R and decrease the probability of trajectories with negative R. Note that the gradient only changes the probability of experience trajectories, but not change the trajectories.

$\nabla_\theta \log P(\tau^{(i)};\theta)$ in the equation (2.1.36) can be further decomposed as:

$$\nabla_\theta \log P(\tau^{(i)};\theta) = \nabla_\theta \log \left[\prod_{t=0}^{H} P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)}) \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \right], \tag{2.1.37}$$

where $P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)})$ is the state transition dynamical model, and the $\pi_\theta(a_t^{(i)} \mid s_t^{(i)})$ is

the policy. From equation (2.1.37), we can get:

$$\begin{aligned}\nabla_\theta \log P(\tau^{(i)};\theta) &= \nabla_\theta \Big[\sum_{t=0}^{H} \log P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)}) + \sum_{t=0}^{H} \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \Big] \\ &= \nabla_\theta \sum_{t=0}^{H} \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \\ &= \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}).\end{aligned} \tag{2.1.38}$$

Compared to equation (2.1.37), equation (2.1.38) contains no state transition dynamical model, which means easier without accessing the state transition model to compute the estimation. The $\hat{g}$ in equation (2.1.36) provides an unbiased estimation because $\mathbb{E}[\hat{g}] = \nabla_\theta G(\theta)$. Therefore, we can estimate policy gradient as equation (2.1.34) with equation (2.1.38) by sampling trajectories under the policy π_θ.

Equation (2.1.34) is unbiased but very noisy because R may varies a lot. To fix this problem for real-world practicality, we can add a baseline and/or use a temporal structure [29]. Consider a baseline b, we want to increase the probability of trajectories with R larger than b, and decrease the probability of those with R smaller than b, then we have:

$$\nabla_\theta G(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)};\theta)(R(\tau^{(i)} - b)). \tag{2.1.39}$$

Notice that this equation is still unbiased, we can prove it as:

$$\begin{aligned}\mathbb{E}[\nabla_\theta \log P(\tau;\theta) b] &= \sum_\tau P(\tau;\theta) \log P(\tau;\theta) b \\ &= \sum_\tau P(\tau;\theta) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} b \\ &= \sum_\tau \nabla_\theta P(\tau;\theta) b \\ &= \nabla_\theta \Big(\sum_\tau P(\tau;\theta) b \Big) \\ &= b \nabla_\theta (\sum_\tau P(\tau;\theta)) \\ &= b \times 0 = 0.\end{aligned} \tag{2.1.40}$$

Thus we still have $\mathbb{E}[\hat{g}] = \nabla_\theta G(\theta)$ when subtracting a baseline b.

As for the temporal structure, let us change the format of the equation (2.1.39), then the current estimation of policy gradient is:

$$\begin{aligned}\hat{g} &= \frac{1}{m}\sum_{i=1}^{m}\nabla_\theta \log P(\tau^{(i)};\theta)(R(\tau^{(i)}-b)) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\right)\left(\sum_{t=0}^{H-1}R(s_t^{(i)},a_t^{(i)})-b\right) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\left[(\sum_{k=0}^{t-1}R(s_k^{(i)},a_k^{(i)}))+(\sum_{k=t}^{H-1}R(s_k^{(i)},a_k^{(i)}))-b\right]\right)\end{aligned} \tag{2.1.41}$$

In this equation, the term $\sum_{k=0}^{t-1}R(s_k^{(i)},a_k^{(i)})$ does not depend on the current action $u_t^{(i)}$, thus removing this term can lower variance of the estimation. We can also make the baseline b to depend on $s_t^{(i)}$, which make the equation (2.1.41) rewritten as:

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\left(\sum_{k=t}^{H-1}R(s_k^{(i)},a_k^{(i)})-b(s_t^i)\right). \tag{2.1.42}$$

There are several choices for the baseline b:

- Constant baseline: $b = \mathbb{E}[R(\tau)] \approx \frac{1}{m}\sum_{i=1}^{m}R(\tau^{(i)})$
- Optimal constant baseline: $b = \frac{\sum_i(\nabla_\theta \log P(\tau^{(i)};\theta))^2 R(\tau^{(i)})}{\sum_i(\nabla_\theta \log P(\tau^{(i)};\theta))^2}$
- Time-dependent baseline: $b_t = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=t}^{H-1}R(s_k^{(i)},a_t^{(i)})$
- State-dependent expected return: $b(s_t) = \mathbb{E}[r_t + r_{t+1} + r_{t+2} + ... + r_{H-1}] = V_\pi(s_t)$, this baseline means that we want to increase the probability of those trajectories according to how much their return are better than the expected return under the current policy.

Empirically, we use the forth baseline to estimate the policy gradient. Here comes the problem: how to estimate the $V_\pi(s_t)$. Generally, we initialize the parameter ϕ to estimate the $V_\pi(s_t;\phi)$, collect trajectories $\tau_1, \tau_2, \tau_3, ...\tau_m$, and then use the empirical return to

optimize the $V_\pi(s_t;\phi)$ as:

$$\phi_{t+1} \leftarrow \arg\min_{\phi} \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\left(V_\pi(s_t^{(i)};\phi) - (\sum_{k=t}^{H-1} R(s_t^{(i)}, a_t^{(i)}))\right)^2. \tag{2.1.43}$$

So far, we have introduced all the concepts in the policy gradient, then we can conclude the process of the "vanilla" policy gradient algorithm as follows:

- **1.** Initialize the policy parameter θ and the value parameter ϕ.

  For iteration $i = 1, 2, 3, ...$, do **2, 3, 4, 5**:

- **2.** Let the agent interact with the environment under the current policy π_θ and collect trajectories.

- **3.** For each time step in each trajectories, calculate the return $R_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$, the baseline $b(s_t) = V_\pi(s_t;\phi)$ and the advantage estimation $\hat{A}_t = R_t - b_t$.

- **4.** Update the value parameter ϕ by minimizing $||b(s_t) - R_t||^2$, summing over all trajectories and all time steps.

- **5.** Update the policy parameter θ by using the policy gradient estimation $\hat{g}$ according to the equation (2.1.42) as $\hat{g} = \nabla_\theta \log \pi(a_t \mid s_t;\theta)\hat{A}_t$, $\theta = \theta + \alpha\hat{g}$, where α is the learning rate.

2.1.3.1 Asynchronous advantage actor-critic (A3C)

Asynchronous advantage actor-critic [17] is a kind of variant of deep reinforcement learning algorithm derived from the "vanilla" policy gradient algorithm. Before introducing the main idea of this algorithm, we will firstly explain what is an "actor" and a "critic".

We can use several variants of the baseline b in the equation (2.1.42) to reduce variance in the policy gradient algorithms. However, the other term $\sum_{k=t}^{H-1} R(s_k^{(i)}, a_k^{(i)})$ will also lead to large variance due to per sample based or no generalization. To address this problem, we have two solutions to change this term. One solution is to reduce variance by adding discount factor γ, the second solution is to reduce variance by using a function to approximate this term.

Generally, we can use the estimation Q to approximate the term $\sum_{k=t}^{H-1} R(s_k^{(i)}, a_k^{(i)})$:

$$Q^\pi(s,a) = \mathbb{E}[r_0 + r_1 + r_2 + ... \mid s = s_0, a = a_0]. \tag{2.1.44}$$

By introducing discount factor, the estimation will be rewritten as:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + ... \mid s = s_0, a = a_0] \\ &= \mathbb{E}[r_0 + \gamma V^{\pi}(s_1) \mid s = s_0, a = a_0] \\ &= \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 V^{\pi}(s_2) \mid s = s_0, a = a_0] \\ &= \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 V^{\pi}(s_3) \mid s = s_0, a = a_0] \\ &= ... \end{aligned} \tag{2.1.45}$$

In the above equation, we can flat the $V^{\pi}(s_t) = r_t + \gamma V^{\pi}(s_{t+1})$. We call it 1-step Q estimation for just flatting 1 step forward, and n-step Q estimation for n steps forward. As shown in the equation (2.1.45), to estimate the $Q^{\pi}(s,a)$, we can use the value estimation function $V^{\pi}(s)$, and this function can also be used to compute the baseline b. Then here comes the main idea of the actor-critic algorithm: we call the value estimation function $V^{\pi}(s;\phi)$ with parameter ϕ as "critic" that predicts the state value of state s. The policy $\pi(a \mid s;\theta)$ with parameter θ is the "actor" to predict the possibility distribution of each accessible action given state s. The "critic" evaluates the goodness of an action a predicted by the "actor" π given state s.

The learning process of the advantage actor-critic (A2C) algorithm is:

- **1.** Initialize the policy parameter θ and the value parameter ϕ.

  For iteration $i = 1, 2, 3, ...$, do **2, 3**:

- **2.** Let the agent interact with the environment under the current policy π_θ and collect roll-outs (s, a, r', s'), compute $\hat{Q}_i^{\pi}(s,a)$ for each step.

- **3.** Update the parameter θ and ϕ according to the update rule:

$$\phi_{i+1} \leftarrow \min_{\phi} \sum_{(s,a,r',s')} ||\hat{Q}_i^{\pi}(s,a) - V^{\pi}(s;\phi)||_2^2 + \lambda||\phi - \phi_i||_2^2 \tag{2.1.46}$$

$$\theta_{i+1} \leftarrow \theta_i + \alpha \frac{1}{m} \sum_{k=1}^{m} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^{(k)} \mid s_t^{(k)}) \left(\hat{Q}_i^{\pi}(s_t^{(k)}, a_t^{(k)}) - V^{\pi}(s_t^{(k)};\phi) \right) \tag{2.1.47}$$

Asynchronous advantage actor-critic is an improved version of the advantage actor-critic using asynchronous method. The main idea of A3C is to utilize parallel actor learners to learn policies, which stabilizes the learning process by decreasing the correlation between experiences and accelerates the training process. In DQN-liked algorithms, experience replay is utilized in order to break the correlation of the sequence of experiences,

while in A3C, parallel actor learners collecting different experiences also break the correlation, and facilitate the exploration in policy-based algorithms.

A3C has a global agent to store the main parameters (θ, ϕ). Note that in practical, θ and ϕ usually share a part of parameter just like the architecture of the dueling DQN. The global agent copies its parameters to all the workers $w_1 = (\theta_1, \phi_1), w_2 = (\theta_2, \phi_2), ...$ Workers will execute the actor-critic algorithms, compute the policy gradient according to the equation (2.1.47), and return their updates of parameters to the global agent. Notice that in A3C, n-step Q estimation is used. When all the workers have finished their learning, the global agent updates according to the parameter update of the workers and then assign the new parameter $(\theta_{i+1}, \phi_{i+1})$ to those workers.

2.1.3.2 Exploration

As mentioned in the section 2.1.2.3, exploration is very important for the agent to learn experiences from the environment. In value-based algorithms, we use ϵ-greedy to force the agent to choose some random actions; however, we can not apply ϵ-greedy in policy gradient algorithms because we should sample the trajectories under the policy π. Generally, we can add an entropy of the policy π to encourage exploration in policy-based algorithms. The technical of adding entropy of the policy π to improve exploration was originally proposed by Williams and Peng [30], they found that it was particularly helpful to solve tasks which require hierarchical behavior. Mnih et al. [17] improved this technical to apply it in the A3C algorithms.

The main idea is to add the entropy of the policy π to the objective function of the policy gradient algorithm [17]. So far we know that the objective function of the actor-critic algorithms is $G(\theta)$ in the equation (2.1.34). Adding the entropy $H(\pi(a_t \mid s_t; \theta))$ of the policy π can improve the exploration by discouraging premature convergence to suboptimal deterministic policies. Then we get a new objective function:

$$J(\theta) = G(\theta) + \beta H(\pi(a_t \mid s_t; \theta)), \tag{2.1.48}$$

where β defines the strength of the entropy regularization and controls how much exploration we want. We then take the gradient of the full objective function including the entropy regularization term:

$$\begin{aligned} \hat{g} &= \nabla_\theta J(\theta) \\ &= \nabla_\theta \log \pi(a_t \mid s_t; \theta)(Q^\pi(s_t.a_t) - V^\pi(st; \phi)) + \beta \nabla_\theta H(\pi(a_t \mid s_t; \theta)). \end{aligned} \tag{2.1.49}$$

2.1.3.3 Trust region policy optimization (TRPO)

Schulman et al. proposed Trust region policy optimization (TRPO) which is similar to natural policy gradient methods but is more effective for optimizing large nonlinear

policies such as neural networks and with guaranteed monotonic improvement [31].

In TRPO, let $R(\pi)$ denotes the expected discounted reward:

$$R(\pi) = \mathbb{E}_{s_0,a_0,\dots}\left[\sum_{t=0}\gamma^t r(s_t)\right], \tag{2.1.50}$$

and the following standard definitions of the state-action value function Q^π, the value function V^π, the advantage function A^π:

$$Q^\pi(s_t,a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\dots}\left[\sum_{l=0}\gamma^l r(s_{t+l})\right], \tag{2.1.51}$$

$$V^\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},\dots}\left[\sum_{l=0}\gamma^l r(s_{t+l})\right], \tag{2.1.52}$$

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{2.1.53}$$

Consider the expected reward $R(\tilde{\pi})$ of another policy $\tilde{\pi}$ in terms of the advantage value over π:

$$R(\tilde{\pi}) = R(\pi) + \mathbb{E}_{s_0,a_0,\dots\sim\tilde{\pi}}\left[\sum_{t=0}\gamma^t A^\pi(s_t,a_t)\right]. \tag{2.1.54}$$

Let ρ_π be the discounted visitation frequencies:

$$\rho_\pi(s) = P(s_0=s) + \gamma P(s_1=s) + \gamma^2 P(s_2=s) + \dots. \tag{2.1.55}$$

Then we can rewrite the equation (2.1.54) with a sum over states instead of time-steps:

$$\begin{aligned} R(\tilde{\pi}) &= R(\pi) + \sum_{t=0}\sum_s P(s_t=s|\tilde{\pi})\sum_a \tilde{\pi}(a|s)\gamma^t A^\pi(s,a) \\ &= R(\pi) + \sum_s\sum_{t=0}\gamma^t P(s_t=s|\tilde{\pi})\sum_a \tilde{\pi}(a|s)A^\pi(s,a) \\ &= R(\pi) + \sum_s \rho_{\tilde{\pi}}(s)\sum_a \tilde{\pi}(a|s)A^\pi(s,a). \end{aligned} \tag{2.1.56}$$

This equation implies that any update of the policy from π to $\tilde{\pi}$ with a nonnegative expected advantage at every state s, making $\sum_a \tilde{\pi}(a|s)A^\pi(s,a) \geq 0$, is guaranteed to improve the policy performance R or just keep it constant. However, the complex dependency of $\rho_{\tilde{\pi}}(s)$ on $\tilde{\pi}$ makes this equation difficult to be directly optimized. Instead, they introduced another local approximation in which replace $\rho_{\tilde{\pi}}(s)$ with $\rho_\pi(s)$:

$$L_\pi(\tilde{\pi}) = R(\pi) + \sum_s \rho_\pi(s)\sum_a \tilde{\pi}(a|s)A^\pi(s,a). \tag{2.1.57}$$

Suppose that we have a policy with parameter $\pi(a|s;\theta)$, then for any parameter value θ_0, we have:

$$L_{\pi_{\theta_0}}(\pi_{\theta_0}) = R(\pi_{\theta_0}), \tag{2.1.58}$$

$$\nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)|_{\theta=\theta_0} = \nabla_\theta R(\pi_\theta)|_{\theta=\theta_0}. \tag{2.1.59}$$

This equation indicates that a small step from π_{θ_0} to $\tilde{\pi}$ that improves $L_{\pi_{\theta_{old}}}$ will also improve R, but this does not tell us how big of a step should take and leads to an excessively large policy update. To address this problem, they provide a lower bound to the improvement of R. The new policy can be defined as:

$$\pi_{new}(a|s) = (1-\alpha)\pi_{old}(a|s) + \alpha\pi'(a|s), \tag{2.1.60}$$

where $\pi' = \arg\max_{\pi'} L_{\pi_{old}}(\pi')$ and this equation is under the following constraints:

$$R(\pi_{new}) \geq L_{\pi_{old}}(\pi_{new}) - CD_{KL}^{max}(\pi_{old}, \pi_{new}), \tag{2.1.61}$$

where $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$ and D_{KL} is the KL divergence [32]. Thus, we are guaranteed to improve the true objective R by maximize the following function:

$$\max_\theta [L_{\theta_{old}}(\theta) - CD_{KL}^{max}(\theta_{old}, \theta)]. \tag{2.1.62}$$

In practice, we usually choose to maximize $L_{\theta_{old}}(\theta)$ under a trust region constraint as: $\overline{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \leq \delta$, where $\overline{D}_{KL}^{\rho}(\theta_1, \theta_2) := \mathbb{E}_{s\sim\rho}[D_{KL}(\pi_{\theta_1}(\cdot|s)||\pi_{\theta_2}(s|\cdot))]$.

To compute $L_{\theta_{old}}(\theta)$, practically, Schulman et al. used a sample-based estimation method called importance sampling [33] in which used the past trajectories collected by the old policy $\pi_{\theta_{old}}(a|s)$:

$$\max_\theta L_{\theta_{old}}(\theta) = \mathbb{E}_t[\frac{\log \pi_\theta(a_t|s_t)}{\log \pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, \overline{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \leq \delta. \tag{2.1.63}$$

This objective function can also be rewritten with a penalty instead of a constraint as the following function:

$$\max_\theta \mathbb{E}_t[\frac{\log \pi_\theta(a_t|s_t)}{\log \pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta\overline{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta)]. \tag{2.1.64}$$

2.1.3.4 Proximal policy optimization (PPO)

Schulman et al. proposed a new family of policy gradient methods, proximal policy optimization (PPO) [19]. PPO have some benefits of the TRPO algorithm, but are much simpler to implement, more general, and have better sample efficiency.

In TRPO, a constraint is added to the objective function (2.1.64) to avoid large policy update. To do this, PPO proposed a clipped surrogate objective loss function to penalize changes of the policy as the following form:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{2.1.65}$$

where the probability ratio $r_t(\theta)$ is defined as $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and ϵ is a hyper-parameter. Compared to TRPO, this objective function is simpler and easier to implement.

In order to reduce the variance, most advantage function estimators make use the state value function $V^\pi(s)$. If we use neural network architectures to approximate those functions, parameter sharing between the policy function $\pi(a|s;\theta)$ and the value function $V^\pi(s;\theta)$ is helpful. Note that the parameter θ is the shared part. In this case, we must use an objective loss function that combines the policy objective function $L^{CLIP}(\theta)$ and the value objective function $L^{VF}(\theta)$:

$$L^{VF}(\theta) = (V_\theta(s_t) - V_t^{targ})^2. \tag{2.1.66}$$

The combination objective loss function can further be augmented by combining an entropy bonus $S[\pi_\theta](s)$ of the policy $\pi(a|s;\theta)$ to ensure sufficient exploration, as mentioned in the section 2.1.3.2. By adding all these terms together, PPO proposed the following objective loss function:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)], \tag{2.1.67}$$

where c_1, c_2 are coefficients.

To compute the advantage function estimation $\hat{A}_t$ in the equation (2.1.65), one popular way is to compute it as $\hat{A}_t = Q_t(s,a) - V(s)$ according to the equation (2.1.45). For an example, let the agent run the policy for T steps, notice that T is much less than the length of an episode, then we use the collected trajectory for an update. For each time step $t \in [0, T]$, we compute the advantage estimation as:

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t-1} r_{T-1} + \gamma^{T-t} V(s_T). \tag{2.1.68}$$

In conclusion, for each iteration in the PPO algorithm, each of N parallel actors collect a trajectory with fixed length T time steps, compute the advantage estimation $\hat{A}_t$ for each time step, then optimize the objective loss function (2.1.67) with K epochs and mini-batch size $M \leq NT$.

Chapter 3

Related work

In this chapter, we will introduce some previous work related to the multi-dimensional hybrid action-space problem and my proposed method. Firstly, we will introduce an extension to the DQN-liked algorithms, then we will introduce an algorithm to address the multi-dimensional action-space problem. Finally, a novel method that uses human demonstration data to accelerate the learning process and provide better starting policies will be mentioned.

3.1 Prioritized replay buffer

DQN-liked algorithms use experience replay to avoid the correlations in the time sequences. Experience replay buffer is used to store the experience transition tuple (s, a, r, s') and those stored transitions are sampled randomly to optimize the objective loss function. In original replay buffer, every sample of (s, a, r, s') has the same probability to be sampled when doing mini-batch. However, it is natural to find that the agent learns more from some of its past experiences while some other provide less to learn. Generally, we want the agent to pay more attention to some transitions than others because it can learn more from those transitions. Schaul et al. proposed prioritized replay buffer [13] to solve this sample efficiency problem.

Priority replay buffer tends to more frequently sample transitions with high expected learning progress, which is measured by their temporal-difference (TD) errors δ. TD error is the temporal difference between the true objective function and the estimation. The TD error δ indicates how unexpected the transition is. Thus, the transition with larger absolute TD error is replayed more frequently from the replay buffer.

Schaul et al. proposed a stochastic prioritization method that interpolated between pure greedy prioritization according to the TD error and uniform random sampling. This method made the probability of a transition being sampled to increase as the transition's

priority grows, and guaranteed a non-zero probability even for the transition with lowest priority. Concretely, the probability of sampling a transition i is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{3.1.1}$$

in which p_i is the priority of a transition i and we have $p_i > 0$, the parameter α indicates how much prioritization will be used. $\alpha = 0$ means uniform sampling.

There are mainly two ways to calculate the priority p_i for each transition. One direct way is a proportional prioritization where $p_i = |\delta_i| + \epsilon$, ϵ is a small positive value to prevent that a transition is not resampled once when the TD loss is zero, and $|\delta_i|$ is the absolute value of the transition's TD error. The other way is an indirect rank-based prioritization by calculating $p_i = \frac{1}{rank(i)}$, where rank(i) for transition i is decided when it is stored into the replay buffer with TD loss $|\delta_i|$. Both of the two ways are monotonic in $|\delta|$. However, the second way is more likely to be more robust because it is more insensitive to outliers.

3.2 Branching Dueling Q-Network (BDQ)

In order to solve a multi-dimensional action-space problem, Tavakoli et al. proposed a novel neural action branching architecture (Figure 3.1 [20]) called Branching Dueling Q-Network (BDQ) to address this issue [20].

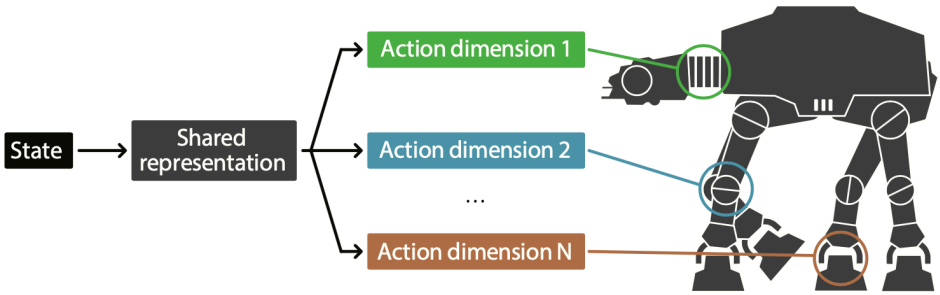


Figure 3.1: A conceptual illustration of the BDQ.

Suppose that we have several number of degrees of freedom to control the motion of an agent, and we allow a level of independence for each of those individual dimensions. Then we can control this dimensions separately. As indicated as the name of this methodology, BDQ separated several network branches for each action dimension, while each dimension shared the common convolutional neural network with parameter

θ to obtain the same state representation vector v_s (Figure 3.2 [20]). The state representation vector v_s was then fed into different action branches and was used to compute the action advantage estimation $A_d^\pi(s, a; \theta, \alpha)$ for each dimension. d denotes the dth action branch. A common value function estimator $V^\pi(s; \theta, \beta)$ was also used to approximate state value $V(s)$, and then was used to compute the state-action value estimation $Q_d^\pi(s, a; \theta, \alpha, \beta) = A_d^\pi(s, a; \theta, \alpha) + V^\pi(s; \theta, \beta)$ for each dimension d. Dueling Double DQN (DDDQN) and prioritized experience replay were used in this architecture.

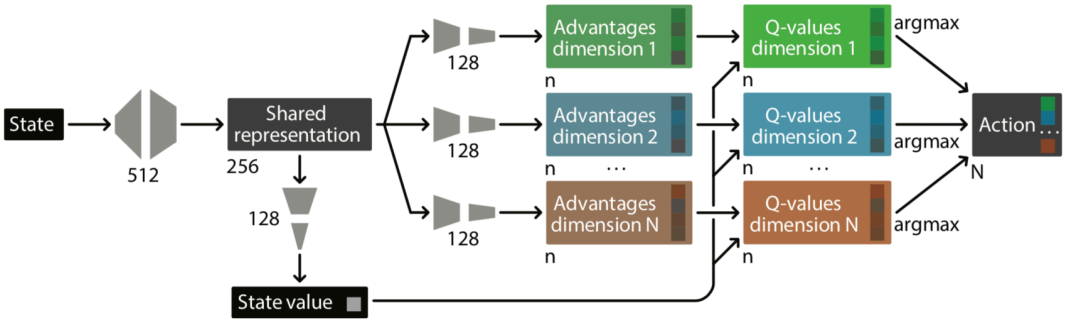


Figure 3.2: The architecture details of the branching dueling Q-network.

Tavakoli et al. [20] experimented on several different aggregation methods to combine the advantage estimation A_d and the state value estimation $V(s)$, and found that the best performing method was to subtract the mean advantage value from the old advantage value in each branch, and then added the value estimation. The $Q(s, a)$ value in each action branch was computed as:

$$Q_d(s, a_d) = V(s) + \left(A_d(s, a_d) - \frac{1}{n} \sum_{a'_d} A_d(s, a'_d) \right). \tag{3.2.1}$$

Tavakoli et al. [20] proposed several temporal-difference (TD) targets for the DQN updating. One simplest way was to calculate a TD target by using the TD target in Double DQN for each individual branch separately as:

$$\hat{y_d} = r + \gamma Q_d^-(s', \arg\max_{a'_d \in A_d} Q_d(s', a'_d)), \tag{3.2.2}$$

where Q_d^- was the target network for the dth branch.

Alternatively, a single global TD target was used for all branches by using the maximum TD target over all branches with the following form:

$$\hat{y} = r + \gamma \max_d Q_d^-(s', \arg\max_{a'_d \in A_d} Q_d(s', a'_d)). \tag{3.2.3}$$

However, BDQ applied another method which showed the best performance. This method replaced the maximum operation in the equation (3.2.3) with a mean operation:

$$\hat{y} = r + \gamma \frac{1}{N} \sum_{d} Q_d^-(s', \arg\max_{a'_d \in A_d} Q_d(s', a'_d)). \tag{3.2.4}$$

Since they had already got the TD target through the aggregation methods mentioned above, the next step was to aggregation the TD targets across all the branches to get a final objective loss function. A simple way was to define the loss function as an expected value of the averaged TD errors across the branches. However, since the TD errors may have different signs and this errors may cancel out with each other, this will reduce the value of the loss function. To overcame this problem, they replaced the TD errors with the absolute TD errors. In practice, Tavakoli et al. [20] found that defining the objective loss function as the averaged squared TD errors across the branches enhanced the performance. The objective loss function was defined as:

$$L = \mathbb{E}_{(s,a,r,s') \sim D}[\frac{1}{N} \sum_{d} (\hat{y_d} - Q_d(s, a_d))^2], \tag{3.2.5}$$

where the TD target $\hat{y_d}$ was computed according the equation (3.2.2), (3.2.3) or (3.2.4).

In BDQ, all branches shared a part of convolutional neural network, the great differences between different branches affected the stability of the back-propagation in the shared part. Therefore, to guarantee the stability, gradient rescaling was applied to rescale the combined gradient before entering to the shared network module by $1/(N+1)$, where N was the number of the branches.

BDQ applied the prioritized experience replay. In order to use this technical, TD errors were aggregated across branches into a unified one value. This combined TD error was used to compute the priority of the transition. To keep the magnitudes of those errors, Tavakoli et al. [20] proposed a unified TD error as the sum of the absolute TD errors across the branches:

$$e_D(s, a, r, s') = \sum_{d} |\hat{y_d} - Q_d(s, a_d)|. \tag{3.2.6}$$

Since BDQ used the Dueling Double DQN as the learning algorithm, this value-based algorithms dealed with discrete action spaces well but was unable to deal with continuous action spaces. Tavakoli et al. [20] thought that discrete-action algorithms were powerful and discretized the continuous action spaces to apply discrete-action algorithms. However, this discretization may destroy the mechanism of the continuous system. For more complicated cases such as control systems with multi-dimensional hybrid action spaces, BDQ is not that suitable. Therefore, we can improve this action branching architecture to make it more general.

3.3 Deep Q-learning from demonstrations (DQfD)

Although deep reinforcement learning has achieved several great successes in difficult decision-making tasks, agents usually require a great large amount of data to learn from before they reach some reasonable performance. The performance of an agent may seem bad and stupid during the beginning of the learning process, and it needs millions or billions time steps to learn to act smartly in the same environment.

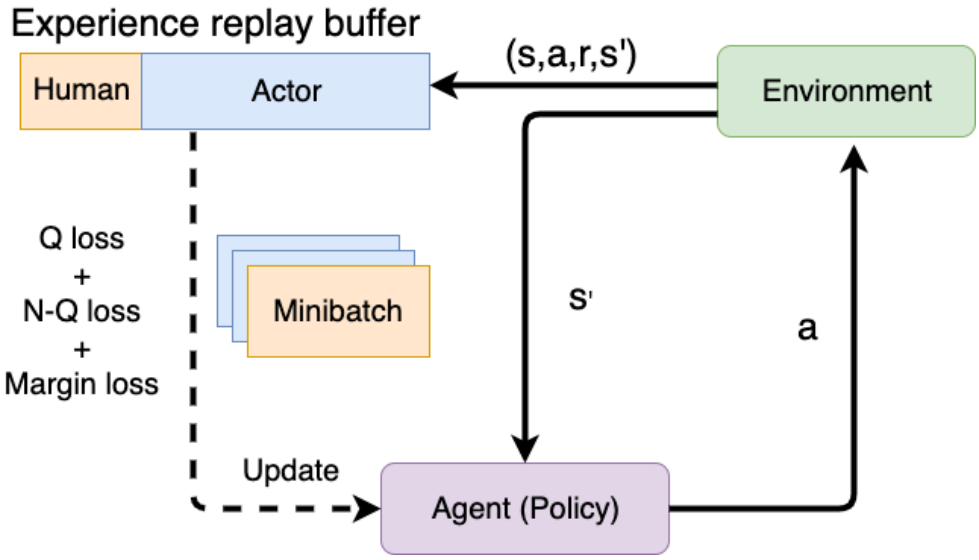


Figure 3.3: The architecture of the DQfD algorithm.

Hester et al. studied a setting where the previous control trajectories were accessible for an agent [16]. They proposed an algorithm called Deep Q-learning from Demonstrations (DQfD) to apply the previous data in deep reinforcement learning algorithms. By using a small set of demonstration data, the learning process was massively accelerated. Their experiments showed that the agent using DQfD had better initial performance than that using prioritized dueling double DQN without demonstration data. This indicated that DQfD also provided a better starting policy than a randomized policy, which implied application in sparse-reward environments. For an example, in games with sparse rewards, it is very difficult for a random agent to get some rewards from the environment, thus it is nearly impossible to learn an efficient policy from those non-reward experiences. In such cases, DQfD can be used to let the agent learn a better starting policy from existed demonstrations and thus collect experiences with positive rewards.

DQfD separated the learning process into a pre-training phase and an interacting phase. The pre-training phase aimed to let the agent imitate the demonstrator and make

use of the demonstration data as much as possible before interacting with the environment. A variant of replay buffer was designed for this algorithm (Figure 3.3). In this replay buffer, human demonstration data was firstly stored, and was never replaced by transitions collected by the agent itself. The agent's self-experiences were then stored after the human demonstration data and were replaced as the replay buffer fulls.

During the pre-training phase, the agent sampled mini-batches from the demonstration data and updated the network parameter. Generally, double DQN uses a 1-step Q-learning loss or a n-step Q-learning loss in the objective loss function. In DQfD, Hester et al. [16] proposed a mixed objective loss function for the pre-training phase. The loss value was composed of four parts: a 1-step double Q-learning loss $J_{DQ}(Q)$, a n-step double Q-learning loss $J_n(Q)$, a supervised large margin classification loss $J_E(Q)$, and an L2 regularization loss $J_{L2}(Q)$. The supervised loss was used as a classification of the demonstrator's actions, while the Q-learning loss avoided simple action imitation and guaranteed that the network satisfied the Bellman equation.

The supervised loss was crucial for the pre-training phase to be effective. Demonstration data usually contained only a small part of the state space and did not take all possible actions. Therefore, many state-action pairs had never been taken and it was hard to ground them to their true value. If only learned with the Q-learning loss during the pre-training phase, the network updated toward the highest of those ungrounded variables and led to wrong estimation. Therefore, a large margin classification loss [34] was added:

$$J_E(Q) = \max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E), \tag{3.3.1}$$

where a_E was the demonstration action and $l(a_E, a)$ was a margin function that equals to 0 when $a = a_E$ and a positive value otherwise. This loss forced that the value of other actions other than a_E had at least a margin lower than the demonstrator's action a_E. This avoided the values of those unseen actions to be too large.

The combination loss was represented as:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q), \tag{3.3.2}$$

where λs were used to control the weights of the losses.

Once the pre-training phase ended, the agent began to interact with the environment as normal DQN methods do. Two things different from normal methods were the replay buffer and the proportional prioritized sampling. The replay buffer in DQfD fixed the demonstration data and never replaced those data. The agent sampled a mixed mini-batch of demonstration data and self-experiences data from the replay buffer and used different priorities to control the ratio of these two different data. Positive constants ϵ_a and ϵ_d were used as priority bonus for self-experience data and demonstration data respectively to control the relative sampling.

Chapter 4

Proposed method: Branching PPO

In this thesis, in order to address the complex multi-dimensional hybrid action-space problem, the proposed method, branching PPO, combined the action branching architecture proposed by Tavakoli et al. [20] and the PPO learning algorithm [19]. Besides using the pixel features of the frames, branching PPO also made use of the non-pixel observations from the environment. A unique learning method was also proposed to using human demonstration data. Experiments in difficult tasks in the 3D environment game of the Minecraft showed the effectiveness of the branching PPO.

4.1 Experiment Environments: MineRL

Deep reinforcement learning has achieved several successes in the game AI field. Researchers claimed to train agents that outperform human beings in various different games, from traditional chess games to complex video games such as Atari 2600 [1]. However, compared to 2D environment video games like Atari 2600, 3D environment games such as ViZDoom, a famous first-person shooting game, Labyrinth and Minecraft are more challenging for both human beings and agents. What is more, the complexity gives 3D environment games the advantage of resembling the real world, which makes it a suitable testbed for solving problems in real world.

Compared to 2D environment games, the partial observability becomes a main difficulty in 3D games. In 2D video games, agent is able to observe the whole state of the environment from the screen, while in 3D games, the agent only sees things in front of it because of the first-person view. The observation from first-person view is unable to refer the whole state of the environment such as what is behind the agent or what happens in other areas, resulting in partial observability.

Besides the partial observability problem, the high-dimension visual observation and complicated action spaces are also difficult problems. Generally, a 3D games often con-

tains various entities and items in the environment. For example, in Minecraft, there are large quantities of entities, blocks and topographies. These not only lead to high-dimension visual observation problem but also increase the complexity of the accessible action spaces.

Figure 4.1: A screenshot of the game Minecraft.

Among those 3D environment games, Minecraft, which is a sandbox game providing a free 3D world with less limits, attracts many researchers for its challenging environment, high-dimension representation, complex action spaces and hierarchical item systems. Minecraft creates a 3D world for players to explore various topographies and scenes. Its players can enjoy an exciting experience of ultimate survival or the pleasure of creating their own worlds. Figure 4.1 shows a screenshot of the game. The world is very complex, containing thousands of different elements, items, blocks and operations. This complexity makes it very difficult to have an artificial intelligence agent learn to act like a human in the Minecraft world. It is hard to define the action space because the actions in Minecraft are continuous, since the action time depends on how long players press their keyboard. The complicated survival system is confusing even to human players. For example, new players may not know where to start or what actions to make, facing such a huge plane when they enter the world for the first time. These properties of the special environment in Minecraft present a challenge for researchers to create an artificial intelligence agent for it.

MineRL is a research project started at Carnegie Mellon University aiming at developing various aspects of artificial intelligence within Minecraft [23]. MineRL provides a simulator of Minecraft and defines several tasks from basic navigation task to difficult

diamond obtaining task. It also defines the state space and the action space for each task and provides simple APIs to control the agent in this simulator. In this simulator, the observations obtained from the task environment are not only the pixel feature of the frame, but also contain some non-pixel features. The action space is a multi-dimensional hybrid action space with both discrete action spaces and continuous action spaces. We will introduce the details of the state space and the action space for specific tasks later. In this thesis, we take the "navigation" task and the "treechop" task as the experiment environments.

MineRL also provides researchers with a great amount of human demonstration data of those tasks. Each trajectory in the demonstration data contains a video recording the human player's game process, a file including the transitions (s, a, r, s') of each time step in the trajectory, and meta information such as whether this trajectory finishes the task successfully or not.

4.1.1 Navigation

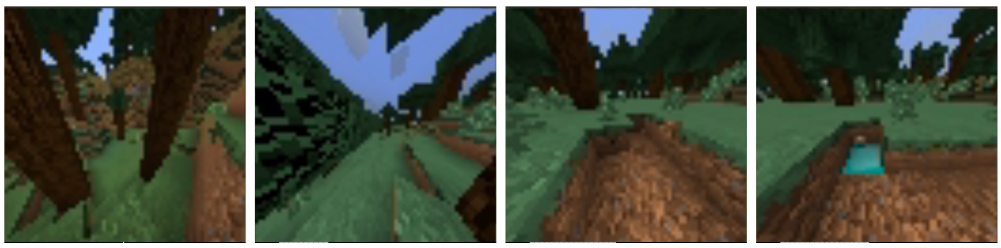


Figure 4.2: The environment of the navigation task. The goal is to find a diamond block in a random environment, and the diamond may be slightly below surface level.

In this task, the goal is to find a diamond block in a random generated survival mode map in a Minecraft world. This represents a basic primitive skill used in many other tasks throughout Minecraft. The agent is randomly initialized in the environment.

Accessible observations are:

- **Pixel observation**: an RGB image observation of the agent's first-person perspective with the size of $64 \times 64 \times 3$.
- **Inventory**: the number of the item "dirt" in stock. An integer number.
- **Compass angle**: a compass angle which points near the goal location, and the goal location is 64 meters far from the start location of the agent.

There are 11 action dimensions in total:

- **Forward**: discrete action space, $\mathcal{A}_d = \{0, 1\}$, where 0 means not to move, 1 means to move 1 block forward.
- **Back**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Left**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Right**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Attack**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Place**: discrete action space, may have different items to choose according to the inventory. In this navigation task, $A_d =$\{"none", "dirt"\}.
- **Jump**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sneak**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sprint**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Direction**: continuous action space, $\mathcal{A}_d = [-180, 180]$. This action space represents the agent's moving direction. This value can be chosen from -180 degree to 180 degree. 0 means to keep going straight, a positive value d means to turn right with d degree, while a negative value means to turn left.
- **Sightline angle**: continuous action space, $\mathcal{A}_d = [-180, 180]$. This action space represents the agent's sightline direction. For an example, 0 means that the agent will look straightly at the front, a negative value d means to look down with d degree, while a positive value d means to look up with d degree.

The agent is given a +100 reward upon touching the diamond block, which is a very sparse reward; however, we use a dense reward variant of this environment in which the agent is given a positive reward every tick for how much closer (or a negative reward for farther) the agent gets to the target. The diamond block has a small random horizontal offset from the compass location. It also has a probability to be set slightly below the surface level, which means that the agent may need to destroy the "dirt" block to find this diamond block under the ground or to search around the place by using local visual features. Episode terminates when agent reaching the goal block or using up maximum 6000 steps.

4.1.2 Treechop

As logs are necessary and significant as a basic resource to craft a wide range of items in Minecraft, it is a very important skill to get log units from the environment. In this task, the agent needs to chop the tree and collect 64 logs in a random forest biome given an iron axe for cutting trees. The agent is randomly initialized in the environment.

Accessible observations are:

- **Pixel observation**: an RGB image observation of the agent's first-person perspective with the size of $64 \times 64 \times 3$.

Compared to the navigation task, there is no "place" action in the treechop task. Other action dimensions are the same as the navigation task. Thus, there are totally 10 action dimensions in this task.

The agent is given a +1 reward when it obtains a unit of log. Episode terminates when agent obtains 64 units or uses up maximum 8000 steps.

Figure 4.3: The environment of the treechop task. The goal is to obtain 64 log units from a random forest biome.

4.2 Architecture and learning methods

The proposed method, branching PPO, was a combination of the action branching architecture [20] and the policy gradient algorithm PPO [19]. In this section, we will talk about the main neural network structure of branching PPO, the TD target, the loss function, a unique learning method to use human demonstration data, and a multi-process simulation method.

4.2.1 Main structure

Figure 4.4 shows the architecture of the branching PPO method. In the experiment environments, there are two different kinds of observations obtained from the environ-

ment, the pixel observation and the non-pixel observation. Deep reinforcement learning algorithms take pixel observations such as RGB images as inputs to the neural networks. To deal with those non-pixel observations in the navigation task environment, we proposed to create a non-pixel feature vector, encode it and make a combination state representation vector $\hat{s}_t$. For an example, in the navigation task, the non-pixel observations are a compass angle and an inventory number of the item "dirt". So we made a vector $v_{\mathrm{non-pixel}}$=[compass angle, number of "dirt"] $\in \mathbb{R}^2$ as another input of the neural network architecture. We regularized the compass angle value by divided by 180 degree. The number of "dirt" is divided by 64.

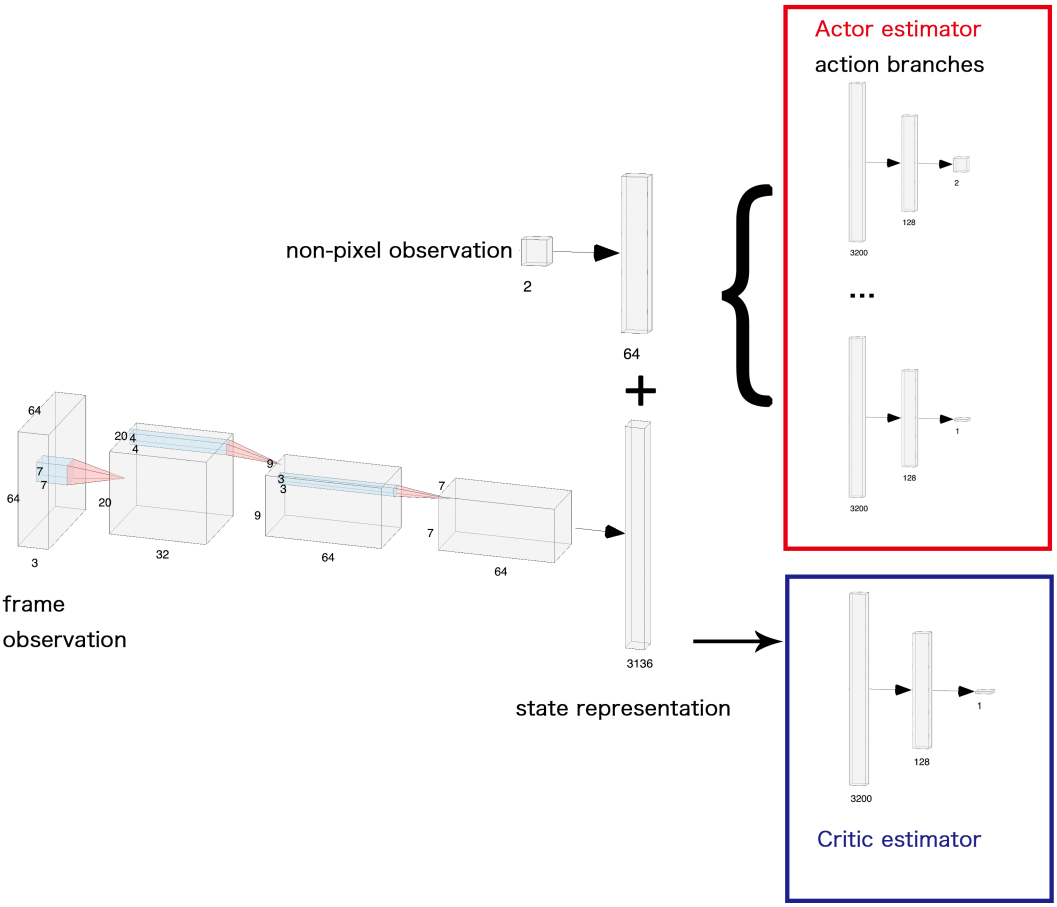


Figure 4.4: The main neural network structure of branching PPO.

This non-pixel observation vector was then fed into extra fully connected layers and we got the feature vector $\hat{s}_{\mathrm{non-pixel}}$. The pixel observation with size $64 \times 64 \times 3$ was fed into a convolutional neural network, and generated a feature vector $\hat{s}_{\mathrm{pixel}}$. These two

feature vectors were then added together to get a combination feature vector $\hat{s}_t$. We took the combination of the non-pixel feature vector and the pixel feature vector as the final state representation and then fed it into a value function estimator and an action branching part.

The value function estimator $V^\pi(s)$ took the state representation vector $\hat{s}_t$ as the input and outputted the estimation of the state value of the current state. This critic value $V^\pi(s)$ was then used to compute the advantage value function A_t^π according to the equation (2.1.68).

The action branching part contained N branches to estimate the policy $\pi(a_d|s) = P(a_d|s), a_d \in \mathcal{A}_d$ for N dimensions. The input of the action branching part was also the state representation vector $\hat{s}_t$. All branches had the same neural network architecture as shown in the Figure 4.4.

Note that we had both discrete action spaces and continuous action spaces. For discrete action spaces, we used the categorical distribution to approximate the distribution over discrete action spaces, and then sampled an action according to this distribution. Softmax was used to output the probabilities of each action in the discrete action space $\mathcal{A}_d$. For an example, in the "forward" dimension, there were two different actions: move forward or not. Therefore, the output of this action branch was a vector $\in \mathbb{R}^2$, where each element represented the probability of one action.

For continuous action spaces "direction" and "sightline angle', we used the Gaussian distribution to approximate the distribution over actions, and then sampled an action according to the Gaussian distribution. The action branch outputted the mean μ, and the standard deviation σ was fixed as 1.

4.2.2 Loss function

Original PPO [19] used a combination objective loss function:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)], \tag{4.2.1}$$

where the policy gradient $L_t^{CLIP}(\theta)$ was computed as:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{4.2.2}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)}. \tag{4.2.3}$$

In branching PPO, we calculated the policy gradient $L_d^{CLIP}(\theta)$ and the entropy bonus $S_d[\pi_\theta](s)$ for each different action branch d. Since there was only one critic estimator, we computed a global value estimation loss:

$$L^{VF}(\theta) = (V_\theta(s_t) - V_t^{targ})^2. \tag{4.2.4}$$

One most direct and simplest way was to define a final global objective loss function by taking the average loss function across all branches as:

$$L(\theta) = \frac{1}{N}\sum_{d=1}^{N} L_d^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 \frac{1}{N}\sum_{d=1}^{N} S_d[\pi_\theta](s), \tag{4.2.5}$$

where N was the total numbers of action branches. In the experiments, we mainly used this global loss function to train the agent.

4.2.3 Human demonstration application

MineRL provides researchers with a great amount of human demonstration data, which is helpful to train the agent and improve the performance. Human demonstration is even crucial in some extremely difficult environments with sparse reward, in which a random policy may never find the goal in the task. As mentioned in the section 3.3, for DQN-liked value-based algorithms, we can apply DQfD [16] to utilize human demonstration data to improve sample efficiency, provide the agent with a better starting policy, and accelerate the learning process. However, DQfD can not be used in policy-based algorithms such as PPO. Actually, policy-based algorithms are more flexible to apply human demonstration data. Since the actor estimator directly approximates the policy $\pi(a|s)$, it is convenient to directly apply supervised learning to update the actor estimator.

We proposed a learning method to use human demonstration data in our branching PPO architecture. Similar to DQfD, there were a pre-training phase and an interacting phase in this method. During the pre-training phase, supervised learning was used to train the actor estimator. It sampled transitions (s_E, r_E, a_E, s'_E) in which E denoted human demonstration data with mini-batch. For discrete action spaces, the actor $\pi(a|s)$ outputted the distribution $P(a)$ for each action. We used the negative log likelihood loss to train the actor as:

$$L_d = -\frac{1}{n}\sum_{1}^{n}\sum_{a\in\mathbb{A}_d} y_a \log P(a), \tag{4.2.6}$$

where $y_a = 1$ when $a = a_E$, otherwise $y_a = 0$. n is the mini-batch size. For continuous action spaces, the actor $\pi(a|s)$ outputted the mean value a_μ of the Gaussian distribution, then we used the mean square error (MSE) loss function to train the actor:

$$L_d = \frac{1}{n}\sum_{1}^{n}(a_E - a_\mu)^2. \tag{4.2.7}$$

We combined loss functions of all branches together and obtained a final loss function:

$$L = \frac{1}{N}\sum_{1}^{N} L_d, \tag{4.2.8}$$

where N was the number of action branches. We also used an experience buffer to store the human demonstration data for mini-batch sampling. Note that in the pre-training phase, only the actors were trained by using the human demonstration data.

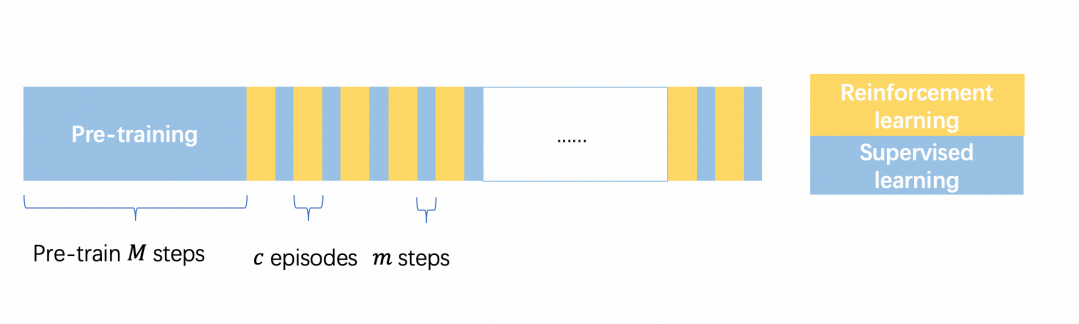


Figure 4.5: A diagram of the proposed learning process to combine branching PPO algorithm and the human demonstration data. In the beginning of the training process, supervised learning was used to train the agent for M steps. Then the agent began to interact with the environment. The reinforcement learning part and the supervised learning part were switched by controlling the parameter c and m.

During the interacting phase, We separated the learning process into two part, the reinforcement learning part and the supervised learning part. In the reinforcement learning part, the agent interacted with the environment, collected trajectories, and updated the actor estimators $\pi(a|s)$ and the value estimator $V^{\pi}(s)$ with the proposed branching PPO algorithm. The reinforcement learning lasted for a fixed number of c episodes, and then switched to the supervised learning part. In the supervised learning part, the agent stopped to interact with the environment, and the actor estimators were trained by sampling the human demonstration data for m steps. This process was just similar to that in the pre-training phase. After the supervised learning part ended, it switched back to the reinforcement learning part. This two parts were switched by controlling the parameter c and m, see Figure 4.6.

4.2.4 Multi-process simulation

Policy gradient algorithms update the policy by sampling trajectory under the current policy. However, only sampling one trajectory leads to huge variance. In order to reduce the variance of the policy gradient, a common method for policy-based algorithms is to use multiple environments to sample multiple trajectories and take the average policy gradient.

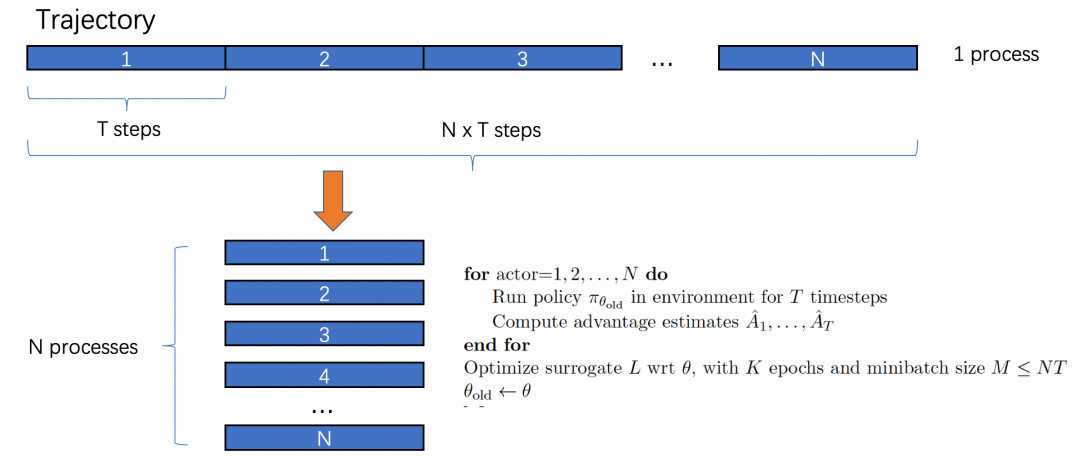


Figure 4.6: A explanation of the multi-process simulation method.

However, the experiment MineRL environments do not support multi-processing. As a result, We proposed a multi-process simulation method to address this problem by using only one game process to simulate multiple processes to reduce variance. Generally, PPO [19] creates N processes, executes T time-steps in each environment in one update. Compute the advantage $\hat{A}_t$ for $t \in [1, T]$ in each process, and samples mini-batches with size M from those NT time-steps to update the policy parameter. In our multi-process simulation, we did it by only using one game process. Instead of excuting T steps in N processes, the agent took $N \times T$ steps in only one process. Then we divided this $N \times T$ steps into N separated trajectories, and treated them as N trajectories from N different processes with length T. Therefore, it was possible to sample the data and updated the policy without huge variance.

4.3 Models

We proposed to use four different models in the experiences: BDQ, BDQ with human demonstration data, branching PPO, and branching PPO with human demonstration data. The same neural network architecture was used for those four models, while the outputs of value-based algorithms and policy-based algorithms were different.

The detail of the shared neural network architecture is shown as follows:

- **Inputs**: For BDQ and its demonstration version, we took the last four frames as the current pixel observation, each frame had the size of $3 \times 64 \times 64$. Therefore, the input size was $12 \times 64 \times 64$. For branching PPO and its demonstration version, we only took the current frame as the input. The input size was $3 \times 64 \times 64$. For the

navigation task, the non-pixel observation was a vector with dimension 2. There was no non-pixel observation in the treechop task.

- **Convolutional neural network 1**: We took the pixel observation as the input. Let (n, k, s) denotes the parameter of a convolutional neural network, where n is the number of kernels, k is the kernel size, and s is the stride length. This network had a size of $(32, 7, 3)$. Outputted o_1 with the size $32 \times 20 \times 20$.

- **Convolutional neural network 2**: We took the o_1 as the input. The size was $(64, 4, 2)$, and it outputted o_2 with the size $64 \times 9 \times 9$.

- **Convolutional neural network 3**: We took the o_2 as the input. The size was $(64, 3, 1)$, and it outputted o_3 with the size $64 \times 7 \times 7$. o_3 was flatted as the $\hat{s}_{\text{pixel}}$ vector with dimension 3136.

- **Fully connected layer 1**: We took the non-pixel observation as the input. The size was 2×64, and it outputted the $\hat{s}_{\text{non-pixel}}$ with dimension 64.

We concatenated the $\hat{s}_{\text{non-pixel}}$ and the $\hat{s}_{\text{pixel}}$ to get a combined state representation vector $\hat{s}$ with dimension 3200. This vector was taken as the input to the value estimator and the action branches as Figure 4.4.

The detail of the value estimator is as follows:

- **Fully connected layer 2**: We took the state representation vector $\hat{s}$ as the input. The size was 3200×128, and it outputted a vector o_4 with dimension 128.

- **Output layer 1**: We took the vector o_4 as the input. The size was 128×1, and it outputted a value estimation $V^{\pi}(s)$.

For different action branches, the neural networks were similar:

- **Fully connected layer 3**: We took the state representation vector $\hat{s}$ as the input. The size was 3200×128, and it outputted a vector o_d with dimension 128 in branch d

However, the outputs for discrete action spaces and continuous action spaces were different. For discrete action spaces, the output layers were:

- **Discrete output layers**: We took the vector o_d as the input. Outputted a vector with dimension $|\mathbb{A}_d|$. In value-based models, BDQ and its demonstration version, this vector represented the state-action value estimation $Q(s, a)$ for each $a \in \mathbb{A}_d$. In branching PPO and its demonstration version, it represented the probability $P(a|s)$ for each $a \in \mathbb{A}_d$.

- **Continuous output layers**: We took the vector o_d as the input. In BDQ and its demonstration version, we discretized the continuous action space $[-180, 180]$ into 36 discrete points {-180, -170, ...,0,...,170,180}. Therefor, the output size was 36. However, in branching PPO and its demonstration version, it outputted the mean μ in a Gaussian distribution, thus the output size was 1.

4.3.1 BDQ

We have introduced the main idea of the BDQ model in the section 3.1. BDQ was a model combining the action branching architecture and the dueling double DQN learning algorithm. Prioritized experience replay was also used in this model. What is more, compared to the original BDQ in [20], in our model, extra neural networks were added to deal with non-pixel observations.

Parameters that used in this model are shown as follows:

- Batch size: 32
- Discount factor γ: 0.99
- Learning rate: 0.00025
- Prioritized experience replay ϵ: 0.0001
- ϵ-greedy annealing steps: 1,000,000
- Experience replay buffer size: 1,000,000

4.3.2 BDQ with demonstration data

This is a demonstration variant of the BDQ model, which used the DQfD [16] to pre-train the agent with human demonstration data. Prioritized replay was also used in this model, and this model had the same parameters in BDQ, and some extra parameters:

- Pre-training steps: 50,000
- Human data prioritized experience replay ϵ: 0.01
- Self-data prioritized experience replay ϵ: 0.0001
- Weights λs for different parts in the combination loss function (3.3.2): $\lambda_1 = 0.0, \lambda_2 = 1.0, \lambda_3 = 0.00001$.
- Human demonstration data: 63152 frames in the navigation task, 439900 frames for the treechop task.

4.3.3 Branching PPO (BPPO)

The main architecture of the branching PPO has been explained in previous section. The main parameters of this model are:

- Batch size: 32
- Discount factor γ: 0.99
- Learning rate: 0.00025
- Clip ϵ: 0.2
- T steps in one trajectory: 8
- Number of processes N: 128

4.3.4 Branching PPO with demonstration data

The learning method of this model is explained in the section 4.2.3. Besides the parameters in branching PPO, some extra parameters are:

- Pre-training steps: 50,000
- Switch parameters: $c = 2, m = 500$.

4.3.5 Action dimensions reduction

The details of the action space is mentioned in the section 4.1.1. We assumed that all these dimensions were independent with each other, thus these action dimensions were predicted by separate action branches. However, as we can see that some action dimensions are not completely independent with each other, such as the "Forward" dimension and the "Back" dimension. These two dimensions could be combined together as one dimension. The number of action dimensions could be reduced by doing this combination. Therefore, we proposed an action dimension reduction variant of the original action space and trained with the four models mentioned before.

The reduction version of the action space in the navigation environment is shown below:

- **Forward_back**: discrete action space, $\mathcal{A}_d = \{0, 1, 2\}$, where 0 means to move 1 block forward, 1 means to move 1 block back and 2 means not to move.

- **Left_right**: discrete action space, $\mathcal{A}_d = \{0, 1, 2\}$, where 0 means to move 1 block left, 1 means to move 1 block right and 2 means not to move.
- **Attack_place**: discrete action space, $\mathcal{A}_d = \{0, 1, 2\}$, where 0 means to attack, 1 means to place a "dirt" block, and 2 means to do nothing.
- **Jump**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sneak**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sprint**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Direction**
- **Sightline angle**

The reduction version of the action space in the treechop environment is as follows:

- **Forward_back**: discrete action space, $\mathcal{A}_d = \{0, 1, 2\}$, where 0 means to move 1 block forward, 1 means to move 1 block back and 2 means not to move.
- **Left_right**: discrete action space, $\mathcal{A}_d = \{0, 1, 2\}$, where 0 means to move 1 block left, 1 means to move 1 block right and 2 means not to move.
- **Attack**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Jump**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sneak**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Sprint**: discrete action space, $\mathcal{A}_d = \{0, 1\}$.
- **Direction**
- **Sightline angle**

Notice that the "direction" dimension and the "sightline angle" dimension were not changed in the reduction versions.

4.4 Experiments results and analysis

We conducted experiments on both the navigation task and the treechop task by using the models mentioned in the previous section. The training results of the four models showed the efficiency of the proposed branching PPO algorithm. For the navigation task, we trained the agent in the dense version environment, and tested the agent performance in the sparse version environment. Finally, we compared the original action space and the reduction version of the action space.

4.4.1 Navigation task

Training results of the four different models in the dense navigation environment are shown in Figure 4.7 and Table 4.1. From the figure, we can see that agents with BPPO algorithms learn faster and achieve higher performance than BDQ agents. Models pre-trained by using human demonstration data indeed improve the agent performance. BPPO shows better compatibility and sample efficiency with human demonstration data.

In the table, mean reward, median reward, min reward and max reward are computed over 1000 episodes. The first 4 rows in Table 4.1 are the results of our models. The last 4 rows are the results from the William H. et al., who are authors of the MineRl platform [35]. They trained a DQN agent, a A2C agent, a behavior cloning agent, and a pre-train DQN agent. Although in their paper they wrote DQN, actually it was a dueling double DQN model. In all their models, they did not use the multi-dimensional action space. They discretized continuous action spaces. However, they only provided the best average reward over 100 episodes.

BPPO algorithms have higher mean reward, median reward, and min reward than BDQ algorithms. BPPO without human demonstration data even obtains better performance than BDQ with human demonstration data. Higher min reward also indicates that BPPO algorithms are good at avoiding extremely bad actions in the beginning of the learning process. When compared to William H. et al.'s work, PPO_DEMO model has lower best average reward over 100 episodes than their pre-train DQN [35] model. However, they did not explain their details of the pre-train methods. The difference may come from the pre-training methods, and that they did not use multi-dimensional action space. BPPO and A2C are both policy-based algorithms, but A2C has very bad performance and can hardly get any reward from the environment.

4.4.2 Treechop task

Training results of those models in the treechop environment are shown in Figure 4.8 and Table 4.2. In the figure, we can see that BPPO and BDQ without human demonstra-

Figure 4.7: Training results over four different models in the navigation environment.

Model	Mean reward	Median reward	Min reward	Max reward	Best average reward over 100 episodes
BDQ	34.90	33.36	-18.76	175.18	53.81
BDQ_DEMO	41.44	46.53	-33.60	175.11	56.10
BPPO	49.27	**53.15**	-14.64	172.28	54.04
BPPO_DEMO	**53.20**	51.67	**-7.53**	174.14	**60.47**
DQN [35]	-	-	-	-	55.59
A2C [35]	-	-	-	-	-0.97
Behavior Cloning [35]	-	-	-	-	5.57
Pre-train DQN [35]	-	-	-	-	**94.96**

Table 4.1: Episode rewards analysis for different models in the navigate environment

Figure 4.8: Training results over four different models in the treechop environment.

Model	Mean reward	Median reward	Min reward	Max reward	Best average reward over 100 episodes
BDQ	0.0	0.0	0.0	0.0	0.0
BDQ_DEMO	8.52	8.0	0.0	37.0	13.62
BPPO	0.0	0.0	0.0	0.0	0.0
BPPO_DEMO	**36.62**	**38.0**	0.0	**63.0**	**47.0**
DQN [35]	-	-	-	-	3.73
A2C [35]	-	-	-	-	2.61
Behavior Cloning [35]	-	-	-	-	43.9
Pre-train DQN [35]	-	-	-	-	4.16

Table 4.2: Episode rewards analysis for different models in the treechop environment

tion data can not get any reward from the environment. This indicates that in the treechop task, which has an environment with extremely sparse reward and is very hard to get any reward with random actions from, pre-training with demonstration data is crucial for the agent to learn better starting policies to have a quick start. With human demonstration data, BPPO_DEMO obtains much more higher rewards than BDQ_DEMO, and shows better compatibility with demonstration data and higher sample efficiency.

In Table 4.2, we also compared the results of our models and the models provided by William H. et al. [35]. The max reward of BPPO_DEMO is 63, which means this model can finish the goal of this task, which is to collect 64 log blocks. The max reward is 63 because when the agent obtains the 64th log block, the environment terminates. The proposed BPPO_DEMO even outperforms William H. et al.'s best model, behavior cloning model.

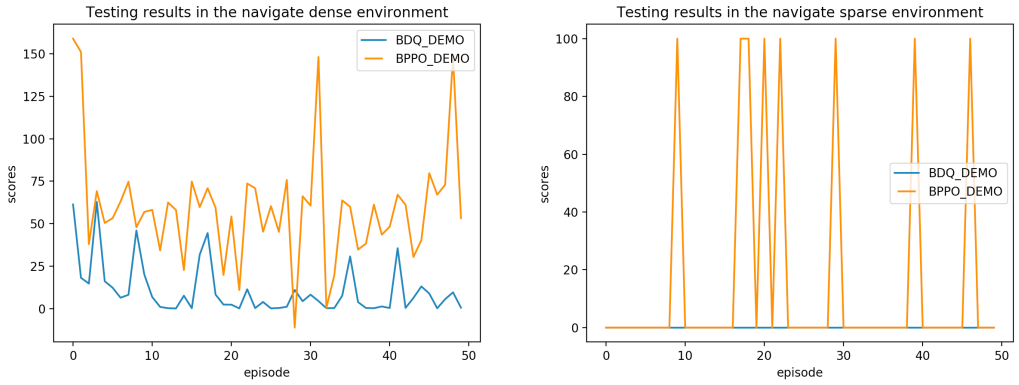
In the navigation environment, pre-train DQN got very high performance, and behavior cloning had poor performance. However, in the treechop environment, behavior cloning outperformed pre-train DQN. We think this indicates the difference between the two tasks. Navigation is more difficult than treechop because the agent can get less information from the pixel observation unless it approaches the diamond block, while in the treechop environment, the agent sees many trees from the pixel observation and it is easy for the agent to learn to obtain a log block by attacking the tree. Therefore, behavior cloning is better in the treechop task. However, our proposed BPPO_DEMO performem well in both environments.

4.4.3 Agent tests

In the navigation task, we trained the agent in the dense version environment. However, agent performing well in dense version environment does not mean that it can also have good performance in the sparse version environment. Therefore, we tested the BDQ_DEMO and the BPPO_DEMO models in the sparse navigation environment.

Figure 4.9 shows the test performance of different models in both the dense and the sparse navigation environment. Each model was evaluated for 50 episodes. Figure 4.9(a) is the testing result in the dense version environment, which is also the training environment. Notice that in this version, the agent gets a small positive reward when it gets closer to the diamond block, and gets a $+100$ reward for reaching the diamond block. Therefore, the agent may get a total reward over 100 when it finally reaches a diamond block. However, in the sparse version, the agent only receives a $+100$ reward for approaching a diamond block. Thus the total reward is 100 only when the agent successfully reaches the goal block, otherwise 0. Figure 4.9(b) is the testing result in the sparse version environment.

Results show that the BDQ_DEMO model can not finish the task in neither the dense



(a) Dense version environment (b) Sparse version environment

Figure 4.9: Testing results for two models in both dense and sparse navigation environments.

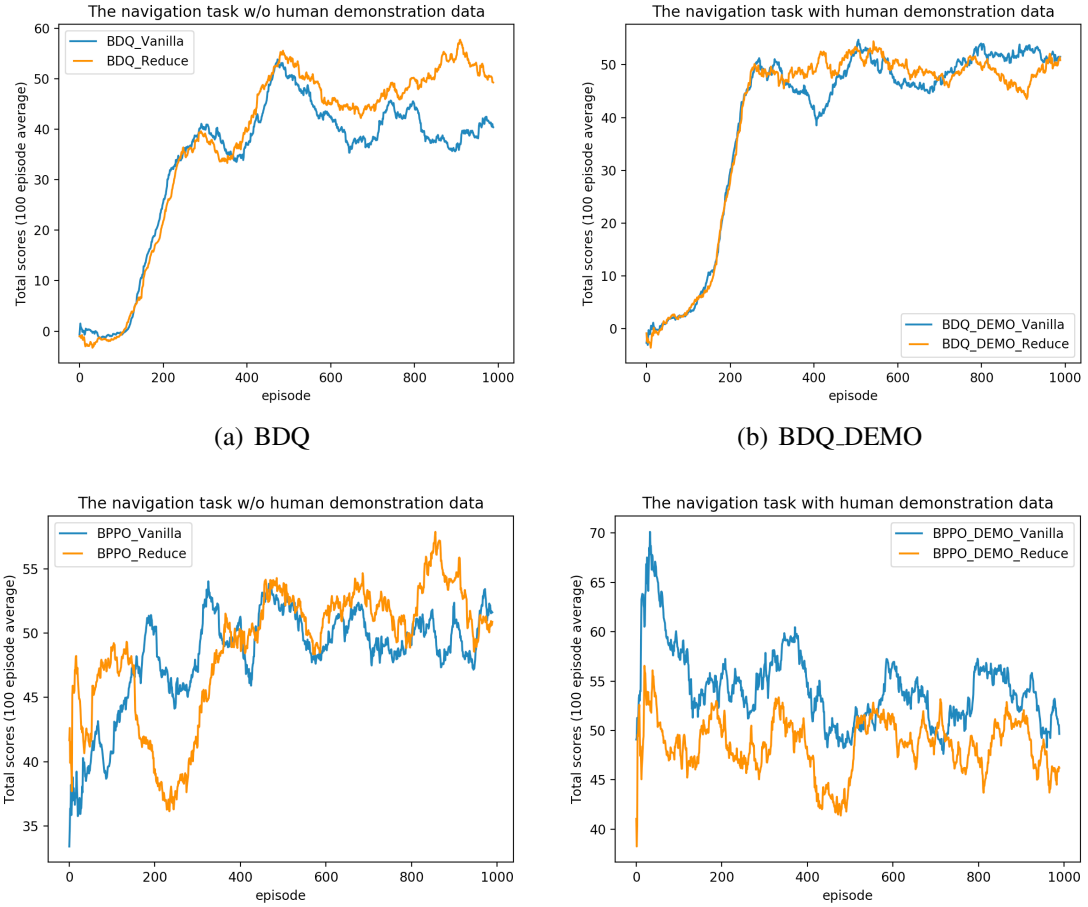
version nor the sparse version, while the BPPO_DEMO model successes 3 times in the dense version and 8 times in the sparse version. This shows the potential of the BPPO_DEMO model that can be reused and applied in more complicated tasks as a skill.

4.4.4 Reduction version

We proposed a reduction version of the action space, which was explained in the section 4.3.5. We also trained four models with the reduced action space in the navigation task and the treechop task.

Figure 4.10 shows the comparison between the original action space and the reduced action spaces in the navigation task with four different models. Figure 4.11 shows the comparison results in the treechop task. From the figures we can see that the reduced action space does not benefit the agent performance, it even causes worse performance with the BPPO_DEMO model.

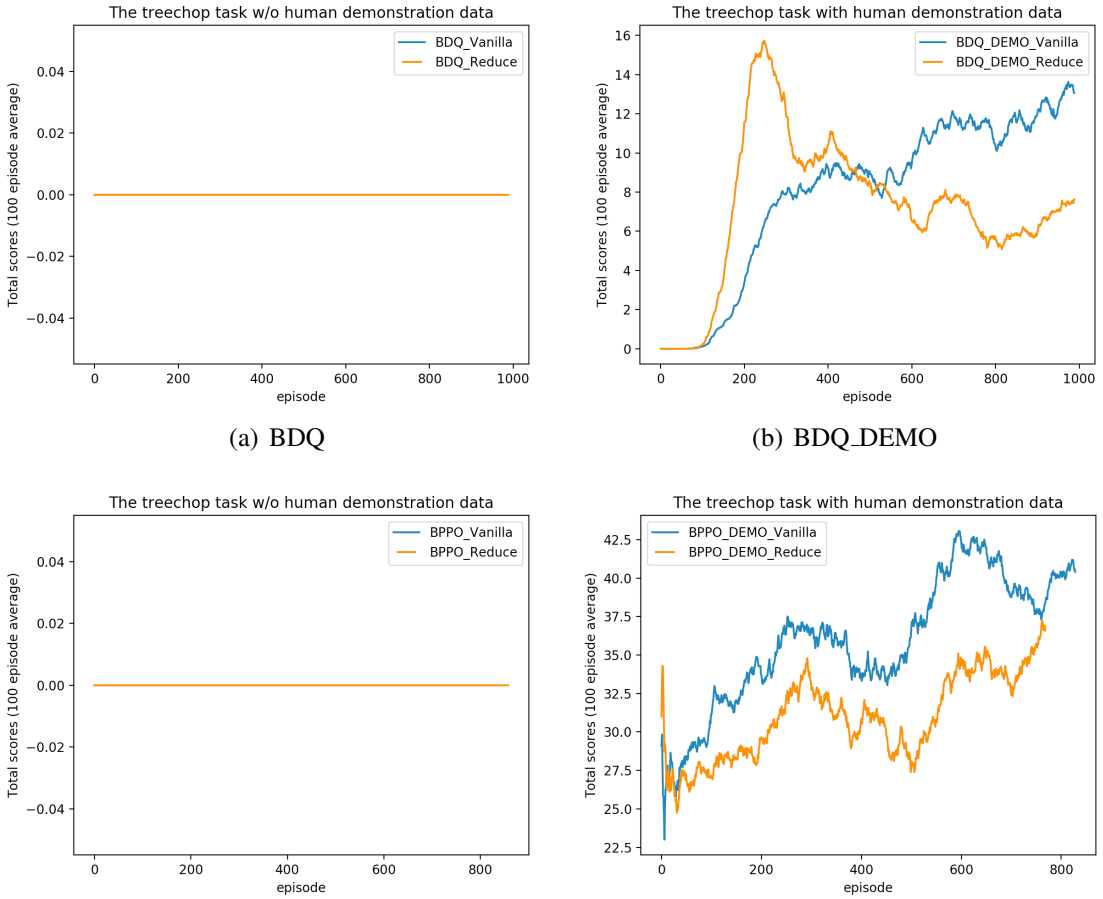
These results contradict with our assumption that combining dependent action dimensions together will improve the performance. This may because that the action dimension "forward" and "back" do not completely contradict with each other, because in the multi-dimensional action space, it has an order to execute each action from different dimensions. The agent can firstly move forward and then move back. These results also indicate that a combination "action" of primitive actions from different independent dimensions with multiple DOFs helps the agent to have more flexible motions.



(a) BDQ (b) BDQ_DEMO

(c) BPPO (d) BPPO_DEMO

Figure 4.10: Comparison between the original action space and the reduced action space in the navigation environment.



(a) BDQ (b) BDQ_DEMO

(c) BPPO (d) BPPO_DEMO

Figure 4.11: Comparison between the original action space and the reduced action space in the treechop environment.

Chapter 5

Conclusion

5.1 Summary

In this work, in order to address a kind of general and complex action-space problem, the multi-dimensional hybrid action space in 3D environment games, we proposed a branching PPO algorithm which combined the action branching architecture and the PPO algorithm. By adding extra fully connected layers to the original action branching architecture, the proposed model can also deal with non-pixel observation inputs. To improve sample efficiency and make use of human demonstration data, we also proposed a learning method which switched between supervised learning and deep reinforcement learning.

we conducted experiments in Minecraft environment with the MineRL platform to evaluate the proposed algorithm. Four models were evaluated in both two environments. Experiment results showed that the proposed algorithm BPPO greatly accelerated the learning process and achieved higher performance in both environments. When we trained the agent with human demonstration data, BPPO showed better compatibility and sample efficiency with demonstration data.

5.2 Future work

Although the experiment results show the efficiency of the BPPO, there are still some ways to improve this proposed method. In this work, we only applied the simplest way to combine branch losses into a final loss function, which was the average value over action branches. It could be improved to use other kinds of aggregation, such as averaged square value.

The second problem was an overfitting problem when we used human demonstration

data in the proposed method. It was difficult to make a balance between the supervised learning and the reinforcement learning parameter m and c. Overfitting occurred when the training steps m of supervised learning part was high during the interacting phase while less m decreased the agent performance. We tried several parameters to find a set of parameters that let the agent have good performance as well as less overfitting. So next we could find a better way to leverage the supervised learning part and the reinforcement learning part.

The third point was that in this work we mainly compared the BPPO models with the BDQ models. All these models were designed to deal with multi-dimensional action spaces. Although we also compared those models with the baselines made by the authors of the MineRL platform, they did not provide the details of their models. We think it could be better to show the efficiency of the proposed model by implementing a one dimensional action space by ourself. Another idea is that in our recent BDQ models, we discretized the continuous action space every 10 degrees and got 36 discretized points. Another way is to remove other points but only keep 10 degrees, 0, and -10 degrees. This was also the way that used in the baselines.

In Minecraft, navigation and tree chopping are basic tasks to collect many resources in this game. Usually in very complicated environments and tasks, hierarchical RL is often used to decompose the difficult tasks into small simple sub-tasks. In this work, we chose the navigation task and the treechop task as the experiment environments to evaluate the proposed model. The results showed the efficiency of the BPPO. Therefore, in future work, a direction is to use the trained models in hierarchical RL to solve more difficult tasks.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints*, 2013.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, Vol. 529, pp. 484 EP –, 2016.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, Vol. 550, No. 7676, pp. 354–359, 2017.

[4] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv e-prints*, 2019.

[5] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, Vol. 575, No. 7782, pp. 350–354, 2019.

[6] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, Vol. 8, No. 3, pp. 279–292, May 1992.

[7] 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*, 2000.

[8] Gavin A Rummery and Mahesan Niranjan. *Online Q-learning using connectionist systems*, Vol. 37. 1994.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, Vol. 518, pp. 529 EP –, 2015.

[10] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. In *AAAI Conference on Artificial Intelligence*, 2016.

[11] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In *International Conference on Machine Learning*, 2016.

[12] H. Y. Ong, K. Chavez, and A. Hong. Distributed Deep Q-Learning. *ArXiv e-prints*, 2015.

[13] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *ArXiv e-prints*, 2015.

[14] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI Conference on Artificial Intelligence*, 2015.

[15] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

[16] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *AAAI Conference on Artificial Intelligence*, 2018.

[17] 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*, 2016.

[18] Timothy P Lillicrap, et al. Continuous control with deep reinforcement learning. *ArXiv e-prints*, 2015.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv e-prints*, 2017.

[20] Arash Tavakoli, et al. Action branching architectures for deep reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2018.

[21] Jiechao Xiong, et al. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. *ArXiv e-prints*, 2018.

[22] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *ArXiv e-prints*, 2017.

[23] Pieter Abbeel. Minerl. `http://www.minerl.io/docs/index.html`.

[24] R.S. Sutton, A.G. Barto, R.S.S.A.G. Barto, C.D.A.L.L.A.G. Barto, and F. Bach. *Reinforcement Learning: An Introduction*. Bradford Book, 1998.

[25] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, Vol. 8, No. 3-4, pp. 293–321, 1992.

[26] Hado V. Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, 2010.

[27] 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*, 2000.

[28] Pieter Abbeel. Policy gradients and actor critic. `https://sites.google.com/view/deep-rl-bootcamp/lectures`.

[29] 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*, 2000.

[30] Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, Vol. 3, No. 3, pp. 241–268, 1991.

[31] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, 2015.

[32] D Pollard. Asymptopia: an exposition of statistical asymptotic theory. 2000.

[33] Jie Tang and Pieter Abbeel. On a connection between importance sampling and the likelihood ratio policy gradient. In *Advances in Neural Information Processing Systems*, 2010.

[34] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Boosted bellman residual minimization handling expert demonstrations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2014.

[35] William H. Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, Manuela Veloso, and Phillip Wang. The MineRL Competition on Sample Efficient Reinforcement Learning using Human Priors. *ArXiv e-prints*, 2019.

Publications

Publications related to this work are as follows:

- Laige Peng and Yoshimasa Tsuruoka. Improving Action Branching for Deep Reinforcement Learning with A Multi-dimensional Hybrid Action Space. ゲームプログラミングワークショップ 2019 論文集, pp. 80-85, 2019.
- Laige Peng and Yoshimasa Tsuruoka. Improving Action Branching for Deep Reinforcement Learning with A Multi-dimensional Hybrid Action Space. 14th Women in Machine Learning Workshop, 2019.

Other publication during the master program is as follows:

- Laige Peng and Yoshimasa Tsuruoka. Learning Basic Skills to Survive the First Day in Minecraft with Life-long Learning. ゲームプログラミングワークショップ 2018 論文集, pp. 101 - 107, 2018.

Acknowledgment

I would first like to thank my supervisor and thesis advisor Prof. Tsuruoka in the School of Information Science and Technology at The University of Tokyo. When I entered the lab, it was very kind of Prof. Tsuruoka to help me get used to the study life in the lab. The door to Prof. Tsuruoka's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed me to do what I was interested in, and steered me in the right direction whenever he thought I needed it. He gave me many advices and suggestions for my research topic, reviewed my paper and thesis, and instructed my academic english writing. I am gratefully indebted to him for his very valuable comments on my thesis. He also supported me and gave me a chance to represent my work at an international workshop in Canada. Prof. Tsuruoka helped me in various aspects. I am so grateful to thank for the support my supervisor giving to me, which made my student life in this lab very happy and fulfilled.

I would also like to thank my families. They supported my idea of studying abroad and gave me the financial support. They respected my opinions and let me do things I wanted to do. My parents always listened to me and comforted me without condition when I felt frustrated. They consistently encouraged me never to give up when I was upset during job hunting, and never pushed me. I really thank my parents.

I would also like to thank the lab members who create a very good and academic atmosphere in the lab. Members in our lab are very kind and they would like to give me suggestions and comments when I did my progress reports in lab meetings. Thank the peer members, Zhang-san, Tong-san, Kano-san, Yasui-san, Ryokan-san, with whom I took courses and prepared for rinko representations together. They helped me a lot for information exchange, mental support and in many other aspects. Thank the Chinese members, Zhang-san, Tong-san, Even-san, David-san, Yu-san, and Rui-san. They comforted me and gave me mental support when I felt lonely as a foreigner in Japan. With all those kind members, I felt so lucky to be a member in the Tsuruoka Lab and had such a fantastic period of time.

During my two and half years in the University of Tokyo, there were so many people who helped me. I am so grateful to express my thanks to them. My former roommate, Zhao, who helped me to get out of depression when I first came to Japan as a foreign

student. She also helped and supported me a lot in my daily life. For this, I am extremely grateful. I would also like to thank a Japanese volunteer, Saito-san from the FACE program in the University of Tokyo, who chatted with me once a week more than one year to help me improve my Japanese. Without Saito-san, I could not have the confidence to communicate with other people and even got a job chance in a Japanese company. I would also like to than Harada-sensei, who is an advisor in the Go-global center. She gave me advice for job hunting and helped me to analysis the situation.

During the time when I wrote this thesis, I would also like to thank Tong-san for helping me to modify the latex file. I would also like to thank Matiss-san, who read my thesis and gave me helpful advise. Finally, I would like to thank my boyfriend, Yin-san, who accompanied me and supported me during this time. I feel so grateful to thank those people who helped me to finish this thesis.

Finally, I must express my very profound gratitude again to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.