TOOLPATH PLANNING IN ADDITIVE MANUFACTURING

By

Yixiao Wang

Thesis Project
Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF MECHANICAL ENGINEERING

Advisor

Prof. Kornel Ehmann

March 2021

ABSTRACT

Toolpath planning in additive manufacturing is a series of typical planar coverage path planning of cross sections of the product. The thesis focuses on planar coverage path planning. The basic goal is to plan a path which fills the whole cross section correctly and as soon as possible. Traditional methods are realized and improved. First, segment the pixels of the cross section into connected graphs. Second, solve nonlinear integer optimization problem through genetic algorithm with the cost function calculated by Lin-Kernighan-Helsgaun method to determine the local path types and visit order. Traditional methods require artificial customization so learning method is studied. Muzero, one of the most advanced reinforcement learning structure, is realized, well-tuned and applied well in the same case. Multiple hyperparameters are analyzed, experimented and tuned. Input design and reward assignment are proved to be vital to the performance. The higher goal is to plan a path which strengths the mechanical properties of the product. Special path patterns or structures make a significant impact on the strength, the strain-at-fracture, toughness and other properties, which cost much computational resources and time to simulate. Besides, the variable dimension of the toolpath increases in factorial order with the length. In order to solve this problem when traditional method is not general and simple enough, reinforcement learning method is tested and shows initial and rough potential in this area.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Toolpath planning in additive manufacturing is a complex, tough but vital for a good-quality product. From a superficial level, toolpath planning determines the path length, the number of lifting the nozzle (or turning laser on and off) and so on. These factors will influence the manufacturing efficiency. From a deep level, toolpath planning determines the path pattern (ring-shaped, zig zag, etc.), feed direction and other factors which will affect the final product quality including mechanical properties and surface finish.

Toolpath planning in additive manufacturing is a series of typical planar coverage path planning of cross sections of the product. The variable dimension of the toolpath increases in factorial order with the number of actions (movements). How to solve the toolpath planning problem within finite time and resources with the goal of human demands remains a open question to researchers.

1.1 Background and Challenge

Additive Manufacturing (AM) produces physical objects layer by layer through a series of manufacturing process like extrusion, sintering, melting, light curing, spraying, etc. Compared with the traditional processing, for example, modeling, cutting and assembly, it is a ”bottom-up” manufacturing method through the accumulation of materials, starting from nothing. This makes it possible to manufacture complex structural parts that were restricted by traditional manufacturing methods in the past. Metal additive manufacturing is the focus of this thesis.

From a superficial level, the idea path has the shortest length, minimal number of switching lasers and minimal number of turns. Path length will influence the manufacturing time. Frequently switching lasers will reduce the efficiency, damage the laser transmitter, and cost more energy. Too many turns will reduce the efficiency and may influence the product quality. For example, there may be holes in the turns. However, the current toolpath planning method in commercial software

only consider whether the path covers the whole target area. Deeper research is needed to increase the production efficiency and product quality.

From a deep level, the idea path has some patterns or feed directions which will potentially influence the product quality. However, the inner mechanism remains unknown or requires very complex simulation and calculation. Besides, most toolpath planning considering the mechanical properties are artificially-designed for special purposes. To increase the mechanical properties, Xia et al. [1] designed the toolpath based on the predicted stress field of the product in the application to strengthen itself. Allum et al. [2] designed nonplanar toolpaths ZigZagZ which increase the toughness, strength and other mechanical properties. However, these toolpaths are planned by artificial designing with certain goal. When the demand of the user changes, the methodology should be designed by the user again. And whether artificial methodology is optimal remains a question.

1.2 Scope and Outline

Chapter 2 introduces the improved methodology of planar coverage path planning in additive manufacturing based on traditional methods. Chapter 3 introduces toolpath planning through reinforcement learning. Hyperparameter analysis, reward assignment and input design are discussed. Chapter 4 introduces the conclusion and future direction.

CHAPTER 2

IMPROVED TRADITIONAL TOOLPATH PLANNING

Toolpath planning in additive manufacturing is a series of 2D coverage path planning (CPP) because AM should print the product layer by layer and in each layer, toolpath planning problem is a 2D CPP problem. In the current commercial software, toolpath planning algorithms are simple, stable but not efficient. Figure 2.1 shows two typical toolpath. The objective is to fill the grey area and the red lines are toolpaths. The toolpath consists of horizontal lines. This kind of paths pass through many white areas which are useless, and the machine should take the laser off.

2.1 Segementation of the Target Area

First, I divide the target area, the grey area, into small areas through connected graph. One pixel is connected to another pixel if it is at the left, right, up, or down of the other. Figure 2.2 shows the details. Arrows in (a) represent the connection direction. (b) shows pixel 1 is connected to pixel 0 but pixel 2 is not connected to pixel 0. However, if defining upper left is also a connection direction, pixel 2 is connected to pixel 0.

After dividing target grey area into connected graphs, local path planning can be done in each

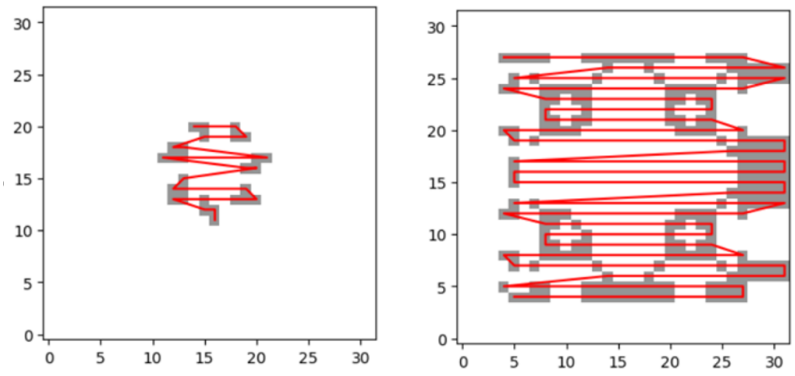


Figure 2.1: two typical toolpath

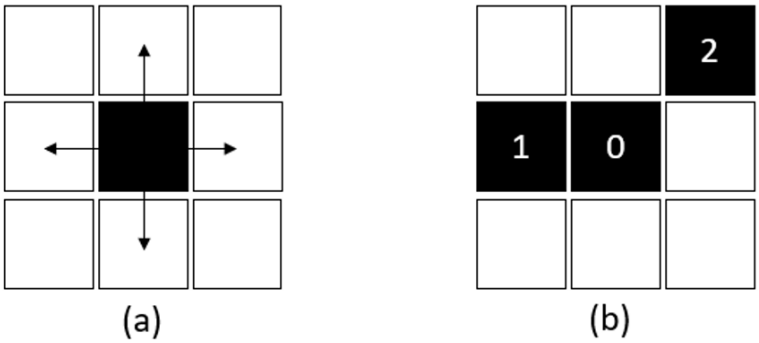


Figure 2.2: Connected graph

graph. Figure 2.3 shows two examples.

After local path planning, total toolpath can be obtained if connecting local paths. Figure 2.4 shows an example. It is obvious that this kind of path pass less white areas.

2.2 Visit Order Planning

It is obvious that visit order of local paths will influence the total path length. In fact, visit order planning is a Traveling Salesman Problem (TSP).

Define the number of local paths as n. The path length of ith area is pl_i, the start point and final point are s_i and f_i, respectively. Define the virtual places as $vp_i, i \in \{1, 2, 3, \ldots, n\}$. For $i \in \{2, 3, 4, \ldots, n\}$, the distance between vp_{i-1} and vp_i is distance$(f_{i-1}, s_i) + pl_{i-1}$. Now, the problem is to plan a visit order from vp_0 to pass through all the $\{vp_i\}$ with the minimal length, which is a typical TSP. Lin-Kernighan-Helsgaun Method (LKH) is used to solve it.Figure 2.5 shows three examples of total paths before and after connection order planning. Total path length of (c) after connection order planning decrease from 564.8 to 332.4.

2.3 Path Type Planning

Local path has several types. For example, path line can be vertical and horizontal. Figure 2.6 shows 8 typical path types.

Define the type in ith local path as α_i. If $\alpha_i, i \in \{1, 2, 3, \ldots, n\}$ are known, visit order can be

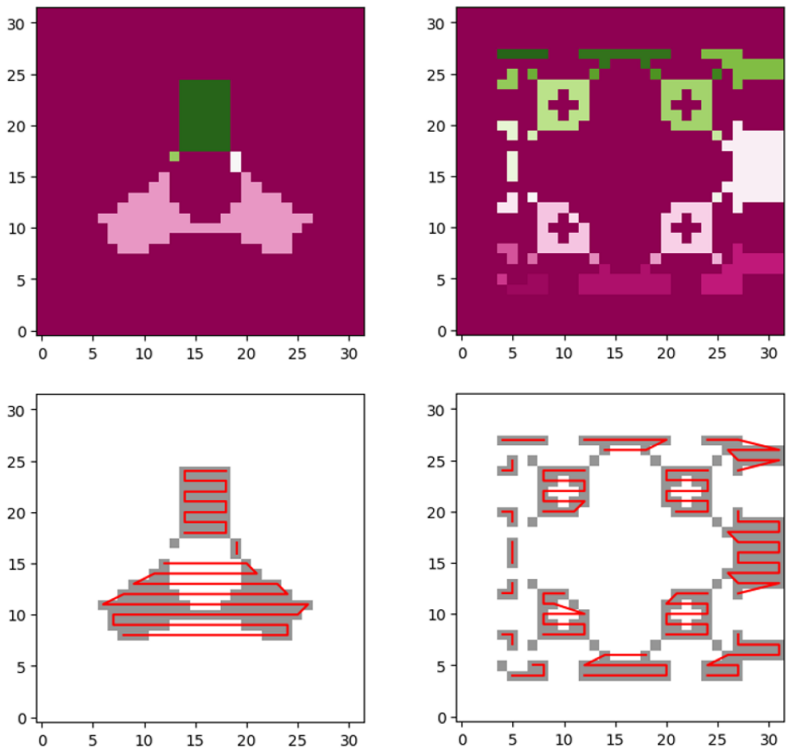


Figure 2.3: Examples of connected graphs and local path planning

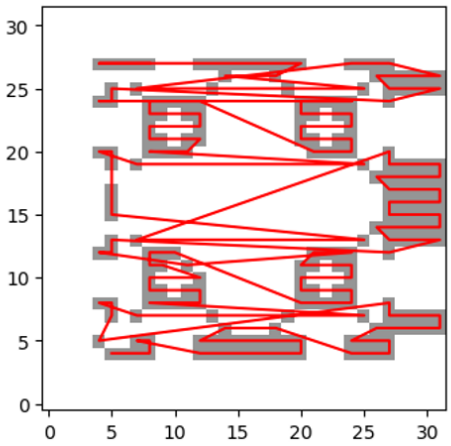


Figure 2.4: Total toolpath after solving TSP

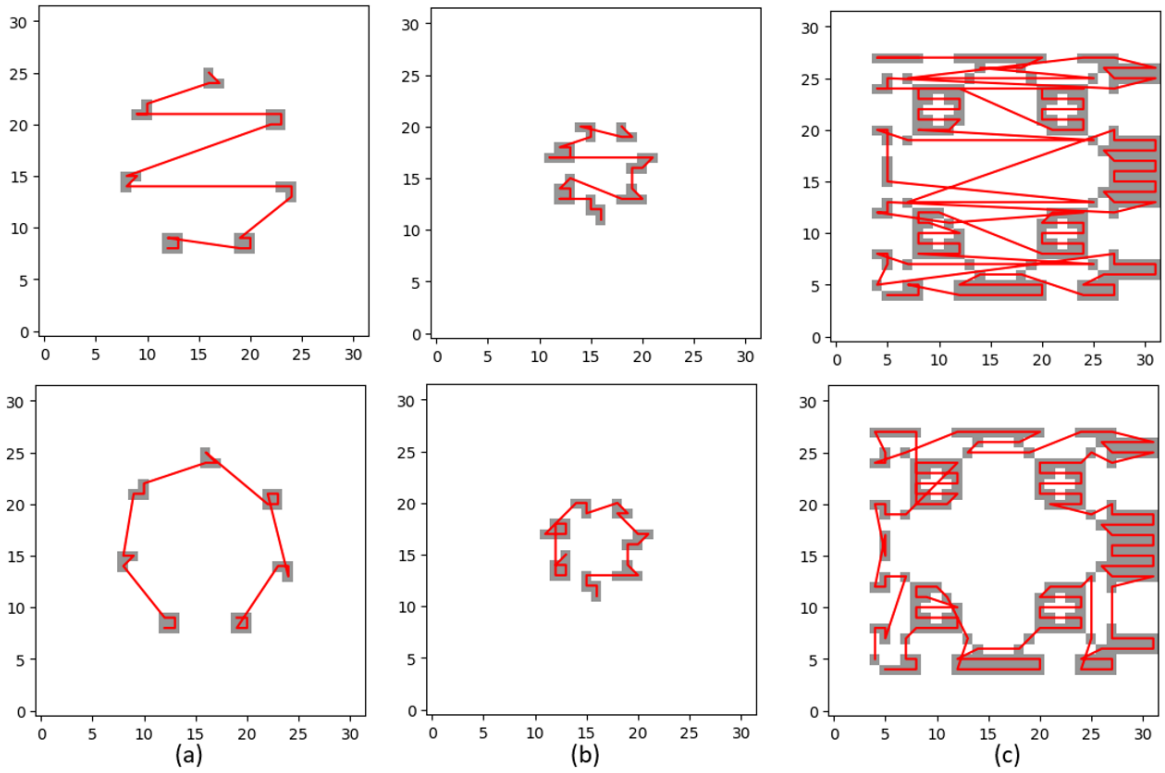


Figure 2.5: Total paths before and after visit order planning

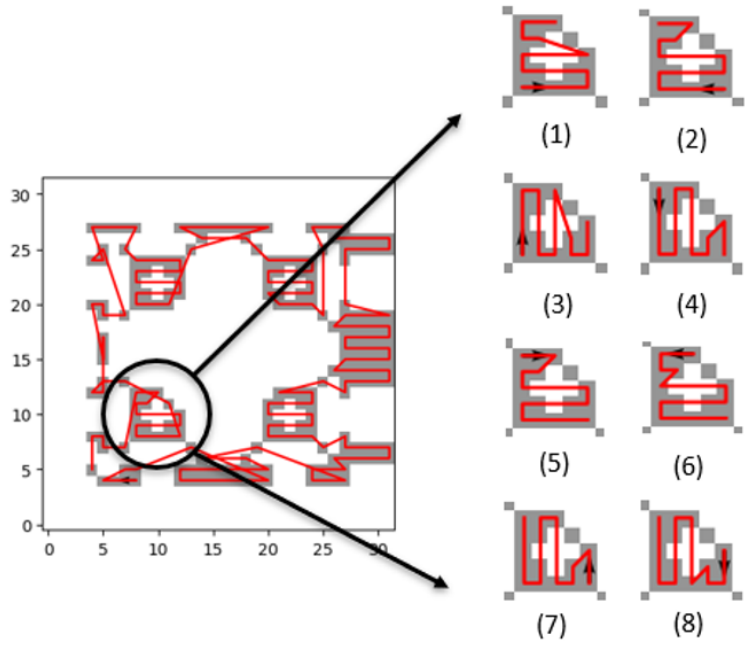


Figure 2.6: Typical eight path types

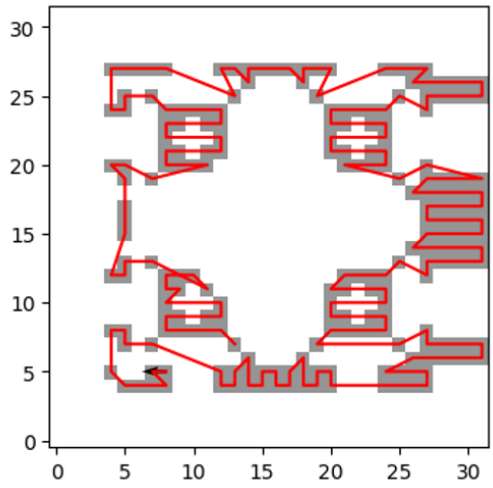


Figure 2.7: Total path after path type planning

calculated through LKH method so the total path length can be obtained. Define total path length as a function $TSP(\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n)$ and a nonlinear integer optimization problem is formed.

$$\begin{aligned} &\min_{\alpha_i} TSP(\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n) \\ &s.t. \alpha_i \in \{1, 2, ..., 8\}, \forall i \end{aligned} \tag{2.1}$$

Genetic Algorithm is suitable to solve it. The reason is that crossover and mutation will reserve the good information. In this problem, in some connected parts, optimal path types can be unique and be affected little from the change of other parts parameters. Figure 2.7 shows total path after path type order planning. The total path length decreases from 332.4 to 298.5.

2.4 Experiments in Industrial Data Set

To be more convincing, I tested the algorithm in the 32x32 section data set, which was established by my coworker Mojtaba. This dataset contains the planar projections of common industrial models, which is representative in metal additive manufacturing. Figure 2.8 shows the frequency distribution graphs. It is obvious that the number of long paths decreases, especially after path type planning. The average path lengths of (a), (b), (c) and (d) are 215.2, 210.0, 200.2 and 144.5.

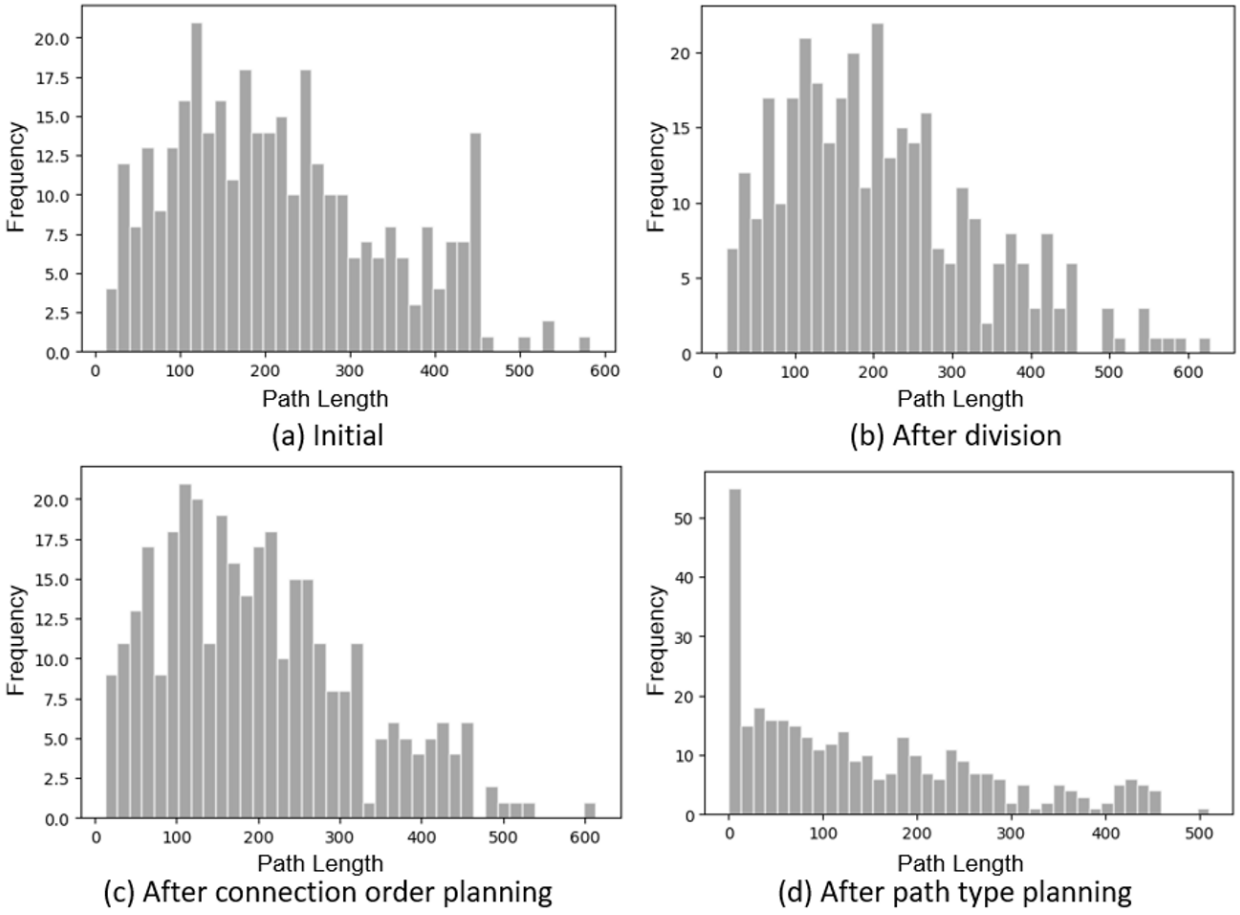


Figure 2.8: Path length frequency distribution graphs

2.5 Summary

This chapter introduces the traditional method to solve planar coverage path planning problem. Definition of the whole problem, division into sub problems and corresponding solutions are contributions of this chapter. The results can significantly decrease the total path length.

CHAPTER 3

TOOLPATH PLANNING THROUGH REINFORCEMENT LEARNING

Traditional toolpath planning is based on human designing. According to certain objectives, such as minimal path length, less turns and properties from complex simulations, people design the path planning methods to approach these targets. For example, to reduce the number of turns, people design spiral path pattern and then calculate parameters to form a spiral path. This kind of path planning method is not universal, requires targeted design and in-depth understanding on the mechanism.

Learning method could overcome this problem. It does not need too much understanding about the mechanism so that prior path patterns or other rules can be obtained, and later, people can design the algorithm to achieve these priorities. Reinforcement learning is utilized to achieve this kind of target. To be more specific, one of the most advanced reinforcement learning structure, Muzero, is applied in toolpath planning.

3.1 RL Structure

Muzero is proposed by DeepMind in 2019. It does not tell the agent what kind of next step (next jet position in additive manufacturing) will lead to a good product. It makes the agent learn next step priorities itself.

Muzero will create several actors to play games simultaneously. When playing games, each actor will use the latest network in the SharedStorage to choose the next step. After game is over, the game history will be stored in ReplayBuffer. Using data in ReplayBuffer, networks will be trained and newest network will be stored in SharedStorage. After some loops, the program stops and Muzero can be utilized to play another different game and achieve great rewards. Figure 3.1 shows the overview structure of Muzero.

The most important part is how each agent plays the game, in other words, how each agent

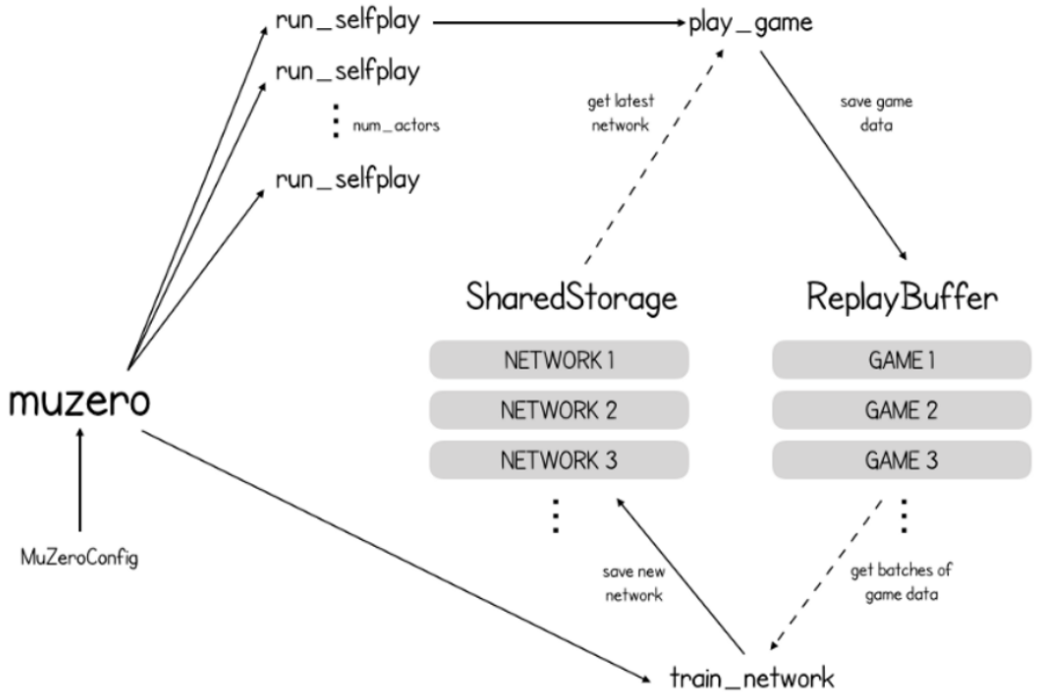


Figure 3.1: Overview of the MuZero [3]

knows what the next step is. Muzero uses Monte Carlo Tree Search (MCTS) to tell the agent which next step will lead to a good future result. The main idea is to simulate what will happen in the next few steps and then to choose the next step which leads to the best future result.

There are three networks in the Muzero. First network h is to convert the input such as images, into hidden state. The function is similar to feature extraction. Second network f is to predict the current value and policy based on hidden state. Policy can be represented as the probability distribution of choosing an action. For example, going up will lead to a better product at the current state in AM. Current value can represent current score. The higher current score is, the better final product will be. Third network g is to predict the probability distribution of next hidden states if an action is applied and to predict the reward of this action if leading to a certain next hidden state. Value can be viewed as accumulated rewards. Figure 3.2 shows how these three networks work in the reinforcement learning. 0 means the current input, S_0 means current hidden state, S_1 means one of the possible next hidden states if action a is taken.

To apply Muzero into toolpath planning problem, we should define the toolpath planning envi-

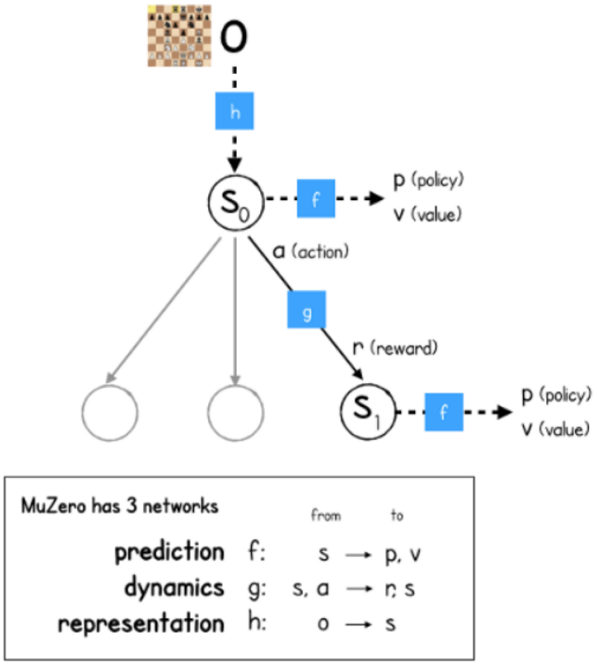


Figure 3.2: How networks work [3]

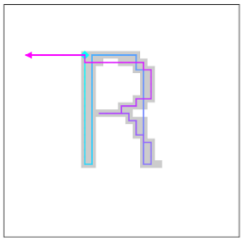


Figure 3.3: An example of toolpath from Muzero

ronment. 332 pictures represented as cross sections of the different products in additive manufacturing are used as data set in this chapter, which has been mentioned in the last chapter. The data set is divided into train one and test one. The train data set contains 322 pictures and test data set contain 10. The aim is to plan a path passing through all the grey pixels as soon as possible. Figure 3.3 shows an example.

Table 3.1: Action Space

action	0	1	2	3	4	5	6	7
laser	on	on	on	on	off	off	off	off
move	up	down	left	right	up	down	left	right

Table 3.2: Reward Assignment

condition	filling correctly	moving uselessly	filling wrongly
reward	1	0	-0.1

3.2 Special Hyperparameters in Muzero

This section focuses on special hyperparameters in Muzero which are not common in other reinforcement learning structures. Mathematical analysis is major part and necessary experiments are conducted for validation. The pictures in the data set are all in the form of 32x32. The initial position is random within the range of the picture for training and testing. There are eight typical actions in additive manufacturing as shown in Table 3.1. The reward can be divided into three kinds: filling correctly, moving uselessly, filling wrongly. The assignment is shown in Table 3.2.

3.2.1 Upper Confidence Bound

In the MCTS, the agent chooses the action a which will maximize the Upper Confidence Bound (UCB).

$$a^k = \arg\max_a \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} (c_1 + \log \frac{\sum_b N(s,b) + c_2 + 1}{c_2}) \right] \tag{3.1}$$

where a denotes action, Q denotes value, P denotes policy, N denotes visit counts. b in $N(s,b)$ denotes all the actions from node(state) s, and $N(s,b)$ denotes the visit count of the node after conducting action b from s. Each MCTS will backpropagate to the root node and all the node visited will add one visit count. Thus, $\frac{N(s,a)}{\sqrt{\sum_b N(s,b)}}$ shows the priority of the action a from all the

Figure 3.4: Total reward (UCB)

actions and its reciprocal shows the exploration. Note that if prediction network works well, which shows that $P(s,a)$ is like MCTS,$P(s,a)$ will reduce the influence of the $\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$. So we can conclude that the first term means greedy strategy and the secend term means exploration.

Increasing c_1 will increase exploration obviously because current value Q means greedy strategy, which is to always choose the action leading to greatest current value. Descreasing c_2 will increase exploration but the exploration is affected by the maximum visit count in MCTS according to

$$\log \frac{\sum_b N(s,b) + c_2 + 1}{c_2} = \log \left(\frac{\sum_b N(s,b) + 1}{c_2} + 1\right) \tag{3.2}$$

Note the maximum of $\sum_b N(s,b)$ is the number of simulations which is 50 in the current case if the laser does not pass the same pixel repeatedly. The simulation will be discussed in the later section.

In the go game [4], $c_1 = 1.25$ and $c_2 = 19652$. $c_2 = 19652$ is not suitable for our case, because if $c_2 = 19652$, logarithmic term will always be zero. So I choose $c_2 = 1000$ (whose legend is $new\_section\_baseline$), $c_2 = 500$, $c_2 = 100$ and conduct the experiments. Figure 3.4 shows the results and we should choose $c_2 = 500$ because the total reward increases faster and maximum total reward is higher.

Figure 3.5: Total reward (Temperature)

3.2.2 Temperature

Final step of MCTS is to calculate the probability of choosing action a, which is

$$p_a = \frac{N(s,a)^{\frac{1}{T}}}{\sum_b N(s,b)^{\frac{1}{T}}} \tag{3.3}$$

where p_a is the probability of choosing action a. $T = 1$ means probability is proportional to the visit count. $T < 1$ means the bigger visit count means bigger probability compared with the proportion. $T = 0$ means choosing the maximum one. Small T leads to more randomness which means more exploration. It is obvious that proper temperature assignment is vital.

Two different assignments, T_1 and T_2, are set and tested.

$$\begin{aligned} T_1(x) =& 1, x \in [0,1] \\ T_2(x) =& \begin{cases} 1, x \in [0, 0.5) \\ 0.5, x \in [0.5, 0.75) \\ 0.25, x \in [0.75, 1] \end{cases} \end{aligned} \tag{3.4}$$

where x is the ratio of current training step to total training steps. Figure 3.5 shows the results. Blue line uses $T_1(x)$ as the temperature strategy and orange one uses $T_2(x)$ instead. The results show temperature strategy does not affect the total reward significantly.

In [4], for the Atari game, actions are selected from visit count distribution as just mentioned

Figure 3.6: Total reward (Temperature Threshold)

throughout the duration of the game. For Go, actions are selected from the visit count distribution before T_s moves. And when the number of moves reach T_s, T will be set to 0. From Figure 3.6, our experiments show this hyperparameter does not affect the total reward significantly.

3.2.3 Dirichlet Distribution

When we have the action distribution p_a from MCTS, we should add some randomness p to it with certain ratio r in the training procedure so that the agent could explore more. The final action distribution is

$$P(a) = (1 - r)p_a + rp \tag{3.5}$$

In Muzero, randomness p is Dirichlet distribution whose probability density function is

$$f(x_1, ..., x_k; a_1, ..., a_k) = \frac{1}{B(a)} \prod_{i=1}^{k} x_i^{a_i - 1} \tag{3.6}$$

where $\sum_{i=1}^{k} x_i = 1$, and $\forall x_i > 0$.

Note that we do not need to care about $B(a)$ which is used for normalization. We only need to focus on the assignment of a_i. It is obvious that when $\forall a_i < 1$, the x will be more likely at the boundary which means the probability of choosing one certain action can be much higher than the others, in other words, more exploration. If $\forall a_i = 1$, the probability of this point x appearing at any position within the k-dimensional cube is equal, which means the probability of choosing

any action is the same. Keep r as a constant. x near the boundary may change the order of action probabilities because some probabilities of certain actions may greatly increase. And if $\forall a_i = 1$, the order of action probabilities will not change but the gap between them will shorten. Thus, there are two ways to increase the exploration when the action is chosen by probabilities: first, decrease the a_i to near zero; second, keep $\forall a_i = 1$ and increase the fraction r. It should be noticed that when the action is chosen by maximum probability, the effect of the second way is relatively small. Thus, the first way is recommended.

3.2.4 Td Steps

Td_steps means the number of steps used to calculate the target value and target reward. It should be noticed that increasing Td_steps generally makes positive effect on the performance but not always. More Td_steps costs more computational resources. There exists a trade-off. On the other hand, when the reward is sparse (it is hard for the agent to get a nonzero reward), more Td_steps is very helpful.

3.2.5 Unroll Steps

When getting batches from game_history, firstly sample the games and then sample the game position. After that, get Unroll_steps number of histories for certain game position in certain game. This term is used to train the dynamic network g. So when it is hard for the network to predict the next hidden state, more Unroll_steps will help.

3.2.6 Loss Weight

The loss function l in Muzero is the weighted sum of policy loss l_p, value loss l_v and reward loss l_r.

$$l = w_p l_p + w_v l_v + w_r l_r \tag{3.7}$$

The weights of three losses may influence the total reward. Different loss weights are tested as shown in 3.3. Figure 3.7 shows the result of experiments. After several experiments of lw_1, we

Table 3.3: Loss Weight

group	w_p	w_v	w_r
lw_1	1	0.25	1
lw_2	1	1	4

Figure 3.7: Total reward (Loss weight)

find that policy loss is always around 60, value loss is around 15, reward loss is around 15. So we quadruple the weights of value loss and reward loss and want the agent to train each loss equally. The result shows lw_2 will make the total reward increase much faster at the beginning. The reason is that at the beginning, reward and value predictions have large errors. And the policy is obtained from reward and value prediction. Training the policy with wrong reward and value predictions leads to bad total reward. If increasing the weights of them, reward and value loss can be trained more at the beginning so they will be more accurate. In the end, it will benefit the total reward.

3.3 General Hyperparameters in RL

This section focuses on the general hyperparameters in RL. Methematical analysis and valid experiments are major parts. The experiments are repeated to get more reliable results.

In order to obtain better and reliable performance quickly, there are several adjustments compared to the previous section. In this section, the data set is compressed into the form of 8x8. Because the picture is too big in the previous so that it takes over 8 hours to complete one experi-

Table 3.4: Action Space

action	-1	0	1	2	3	4	5	6	7
laser	off	on	on	on	on	off	off	off	off
move	still	up	down	left	right	up	down	left	right

Table 3.5: Reward Assignment

condition	filling correctly	moving uselessly	filling wrongly	stay still
reward	1	-0.1	-0.3	-0.5

ment. Besides, random initial position causes nonrepeatability of the experiments. Thus, the initial position is fixed in testing. The action space is 9-dimensional as shown in Table 3.4 because when the agent attempts to go out of the image, the agent should stay at the same place by the law of the simulation environment. Thus, it is necessary to add another action which is 'stay still' into the set of actions. As the consequence, the reward assignment should be changed as shown in Table 3.5 where moving uselessly is penalized for a shorter path length.

There are several toolpaths from repeated experiments shown in Figure 3.8. Though all the parameters are set the same, the randomness will lead the policy to different optimums.

3.3.1 Initial Position in Testing

The performance of reinforcement learning is hard to repeat. To increase the repeatability, fixing the initial positions in testing would be a solution. In order to be more rigorous, I have conducted

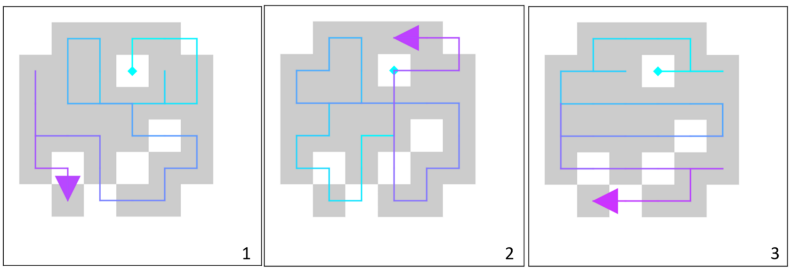


Figure 3.8: Examples of toolpaths (8x8)

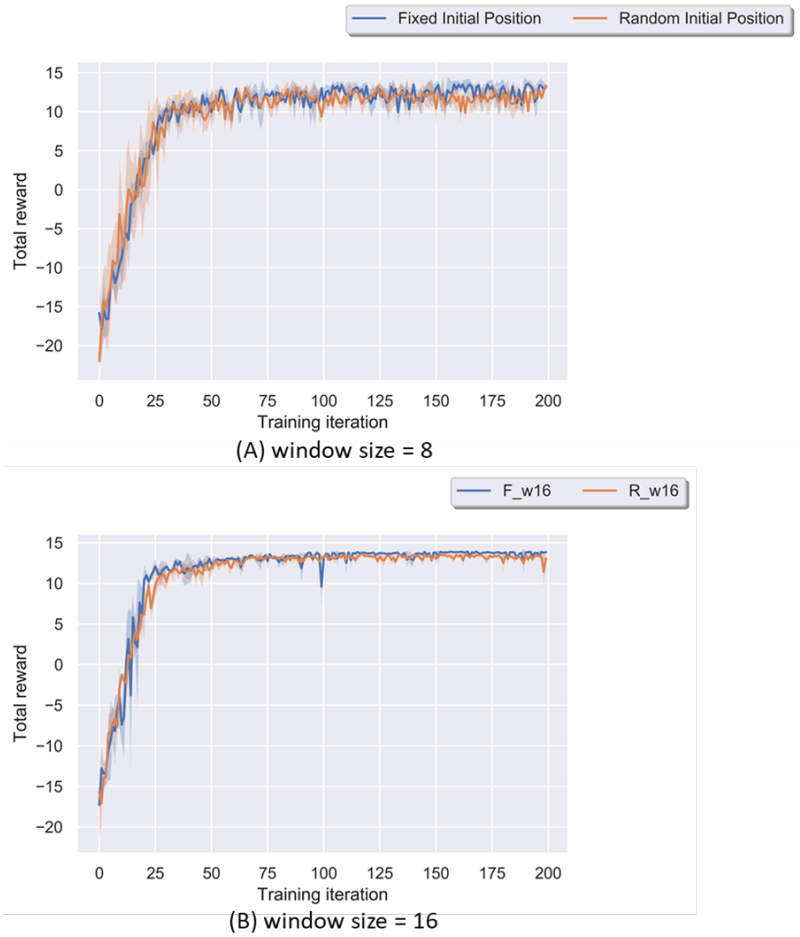


Figure 3.9: The influence of random initial positions for testing

the following experiments to demonstrate the influence of the initial positions for testing. Each experiment are repeated three times. The solid lines and semitransparent areas are the means and stds of the results from repeated experiments. The results in Figure 3.9 show that wherever the initial position is, the agent can have the similar performance after training, especially in the target of filling all the grey pixels which is the most important goal. Thus, in the later discussion, the initial position for testing is fixed and keeps the same. Window size means the size of the input to the network which will be discussed in the later section.

3.3.2 Episode

Episode denotes the situation that the agent is working for its task. The number of episodes denotes how many agents are working simultaneously in each training loop. After the current training loop, the histories of these episodes will be added into replay buffer and the optimizer will use newly updated replay buffer to train the network. Thus, the number of episodes can not be too small; otherwise, not enough histories are added into replay buffer in each training loop. That is to say, the network is optimized based on only a few new episodes working according to the most recent network. The consequent fast updates of the network causes instability and other defects. And the number of episodes can not be too big, either. Too many episodes will cost too much computational resources. If the size of replay buffer is constant, replay buffer will contain the more recent information, making the optimizer more greedy. So proper number of episodes should be determined.

I conducted group experiments. There are three groups: episode = 10, episode = 20, episode = 40. In each group, three repeated experiments are completed. The results are shown as Figure 3.10. Total reward of episode10 is lower and less stable than others. Total reward of episode40 goes up faster than the others in the beginning but is overtaken by episode20. The result is in accordance with the previous analysis. Fewer episodes cause more instability and more episodes will accelerate convergency. And because of the defects of "greed", too many episodes will cause more susceptibility to local optimum. Considering the cost of the computation, episode = 20 is more suitable in this case.

3.3.3 Learning Rate

Learning rate will influence the speed of training in the supervised learning. In the supervised learning, small learning rate means the low speed towards convergence but more stability. Large learning rate means fast convergence but less stability, even oscillation which probably leads to non-convergence. So proper learning rate is essential in learning procedure and decay learning rate is often used in supervised learning which speeds up the training at the beginning and then benefits

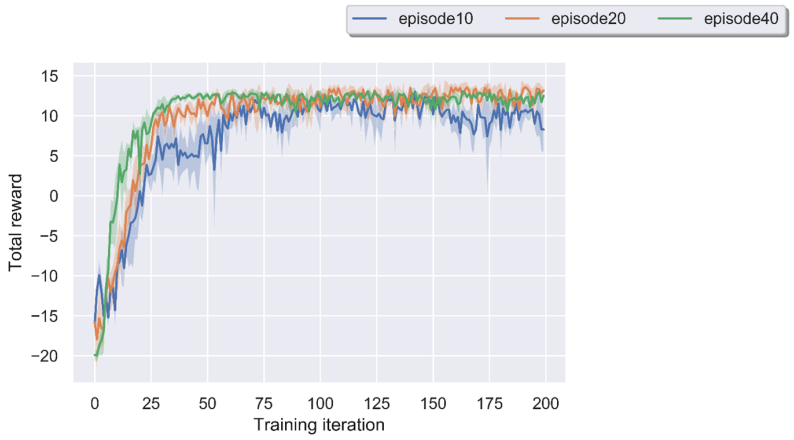


Figure 3.10: The influence of episodes

convergence gradually. However, in reinforcement learning, the influence of learning rate is more complex. When more histories are added into replay buffer, the optimizer will use these data to train the network. So current data in the replay buffer cannot reflect the whole situation of the agent will encounter. If training too much in each training loop, the network may be overfitting. Thus, if learning rate is too small, the optimizer tends to get into local optimum for the current replay buffer. And when new histories are added, the optimizer cannot quickly jump out of the local minimum if learning rate is too small. However, if learning rate is too large, the network may be not trained enough because of oscillation or something else. The most complex thing is that the trained network will be used in the next training loop and the agent will use this network to do its task, creating the new history to used in the next training. In general, from my experiments, the learning rate should not be too small and too large. And the only and practical way to determine the learning rate, from my point of view, is testing. Training loss in each training loop can be saved, which can be used to check whether the number of epochs is enough to train.

I chose 0.01, 0.005, 0.002 as three different learning rates (lr0.01, lr0.005, lr0.002) and tested three times. The results are shown in Figure 3.11, 3.12, 3.13. The conclusion is that constant learning rate 0.005 is more suitable in this case. The total reward is more than others and the convergence is faster. The number of filling currently can reach maximum and be more stable. The total loss of lr0.005 is also lower than others in the later stages of training.

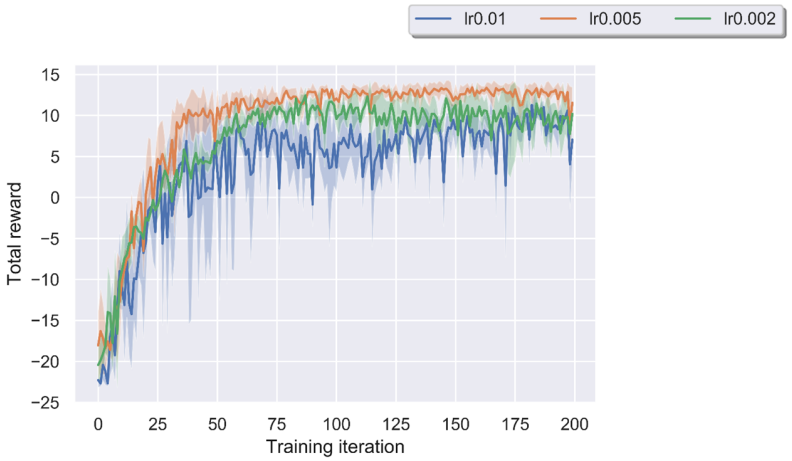


Figure 3.11: Total reward (lr)

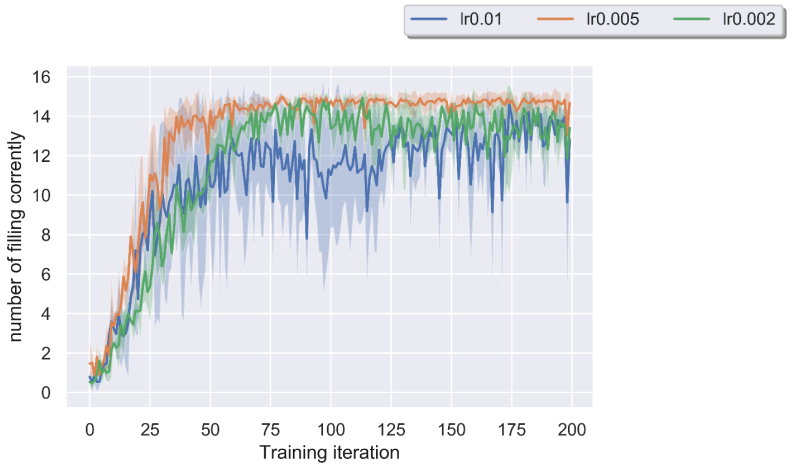


Figure 3.12: Number of filling correctly (lr)

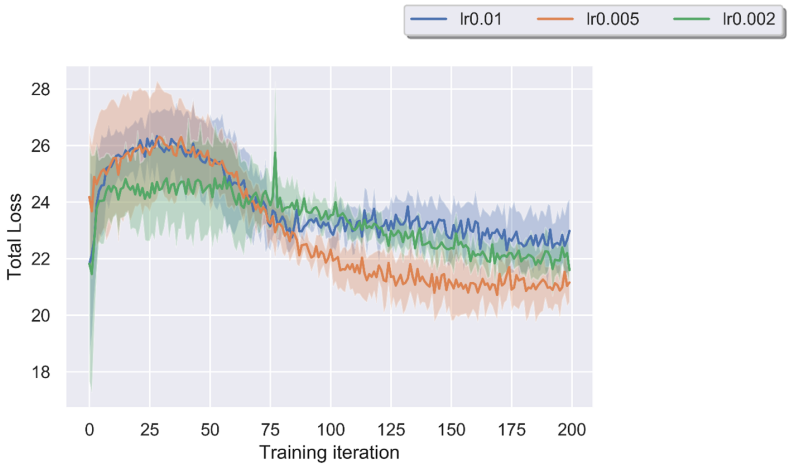


Figure 3.13: Total loss (lr)

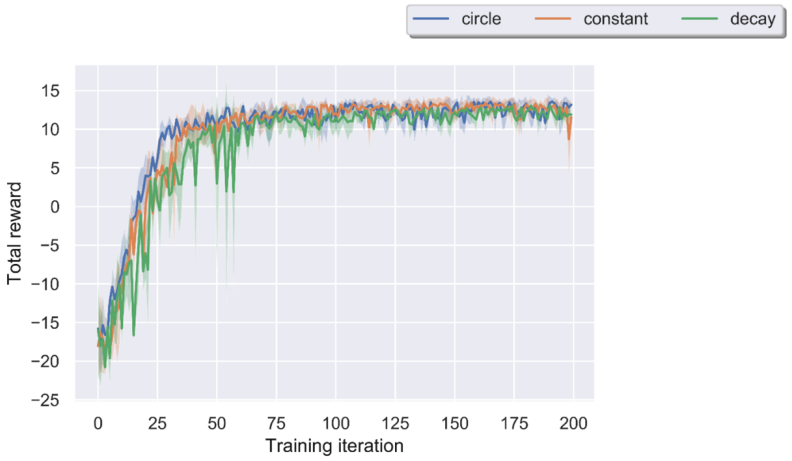


Figure 3.14: Total reward (different lr)

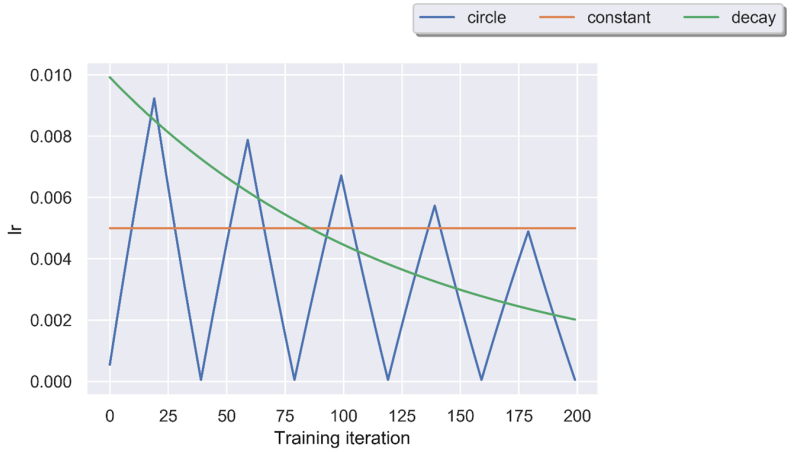


Figure 3.15: lr (different lr)

Exponentially decayed learning rate and circle decayed learning rate are both tested. The former is common in machine learning and the latter is newly designed. Circle decayed learning rate will go up and down like cosine function during the training, which increases the chance to jump out of dead zone like local optimum. As shown in Figure 3.14 and 3.15, circle decayed learning rate is more suitable in this case. Thus, circle decayed learning rate should be chosen.

3.3.4 Number of Simulations

The idea case is to teach the agent to avoid the penalty and get the positive reward. It is true in theory, but not in practice. In theory, the agent after training should avoid all the penalty except

-0.1 (because moving uselessly is unavoidable). The reason for the gap between theoretical and practical results may lie in the error of value prediction and MCTS. The decision made by MCTS is based on the reward and value predicted by the network. Through several experiments, the reward error can be reduced to nearly 0. And the target reward obtained from the environment is absolutely correct. However, the target value may not be right because it is calculated from Temporal-Difference Learning. Thus, the value prediction matters. Another possible reason is about MCTS. Except the structure of MCTS which is hard to improve and do some research on, whether the search in MCTS is sufficient may be the answer. As we all know, MCTS will expand its tree several times and choose the best action based on the values of actions in the root which are calculated by backpropagation. Thus, even if the value and reward prediction are correct, the policy generated by MCTS is locally optimum. On the other hand, the time of expansion is essential. When the number of expansions are not enough, the policy will be too greedy. But when the number of expansions is too large, MCTS tends to become enumeration method which is not necessary and reasonable. Thus, proper number of expansions which are called simulations in Muzero is vital to the optimal policy.

This section is focused on the influence of number of simulations. The action space is nine-dimensional which means when the MCTS expands two leaves, there will be 81 possibilities. It is very large space. In Muzero, the numbers of simulations are over hundreds in chess and go, and 50 in Atari.

To study on the influence of number of simulations, I conducted group experiments: w1_s20: 20 simulations; w1_s40: 40 simulations; w1_s80: 80 simulations. The experiment is repeated three times in each group. The results are shown in Figure 3.16, 3.17, 3.18. After increasing the number of simulations, the performance is enhanced in all the aspects. However, the decrease in the number of filling wrongly is not obvious. Because the increase of simulations is enough because the performance of w1_s80 is similar to w1_s40. Therefore, the only reason is that the optimal policy MCTS can learn in the current reward assignment. To draw the conclusion, the number of simulations should be big enough but it does not need to be too big which will cost too

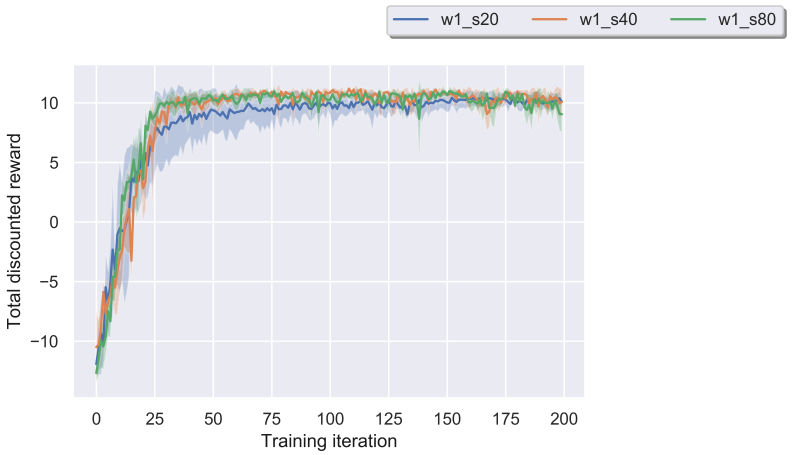


Figure 3.16: Total discounted reward (number of simulations)

Figure 3.17: number of filling wrongly (number of simulations)

much resources. To obtain the optimal policy, reward assignment is more essential.

3.4 Network

Network is essential to the performance of agents. It can be seen as a function so the relationship between the input and output is vital to whether the function can be optimized as well as expected. In the current dynamic network, as shown in Figure 3.19, current hidden state stacked with next action passes through several convolutional networks and exports the next hidden state. Hidden state means the feature of the information of the environment. At the same time, the output passes through another convolutional layer and fully connected network and exports the reward, which

Figure 3.18: number of moving uselessly (number of simulations)

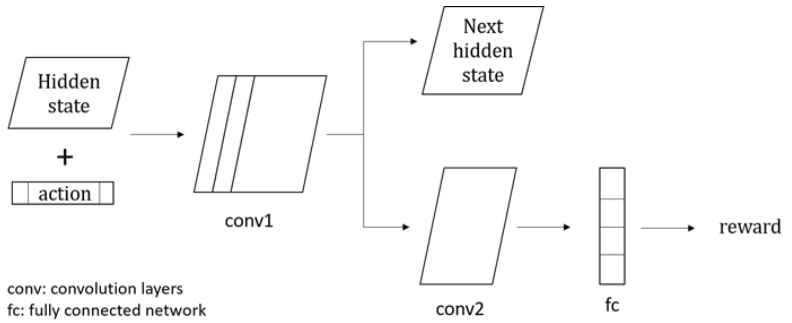


Figure 3.19: Current dynamic network

implicates that the function can predict reward only based on next hidden state. It is not reasonable because reward is also related to the action. The output of conv1 is next hidden state, which should not contain any information about action. So I modified dynamic network structures as shown in Figure 3.20 and 3.21. In Figure 3.21, conv2 and conv3 contain more layers than those in Figure 3.20.

I have conducted three repeated experiments for each kind of dynamic network (Mdnet_plus: Figure 3.21; Mdnet: Figure 3.20; dnet: Figure 3.19) and the results are shown in Figure 3.22. After modification, total reward increases faster. And with more convolutional layers, the dynamic network can obtain higher total rewards and become more stable. Thus, the conclusion is that the structure of "Mdnet_plus" should be utilized in this case.

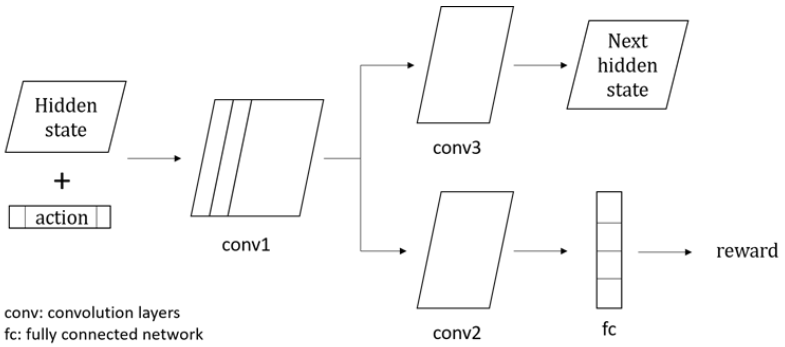


Figure 3.20: Modified dynamic network

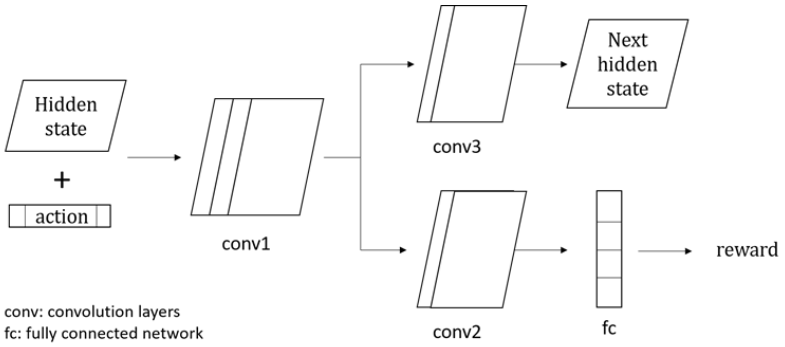


Figure 3.21: Modified dynamic network with more convolutional layers

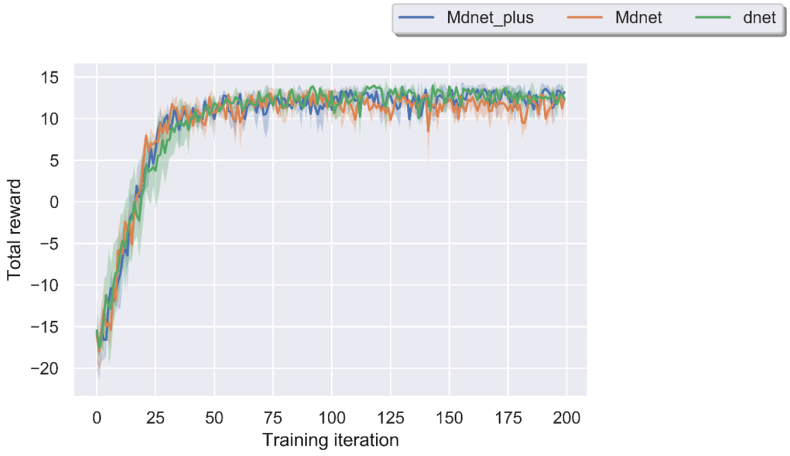


Figure 3.22: Performance of different dynamic networks

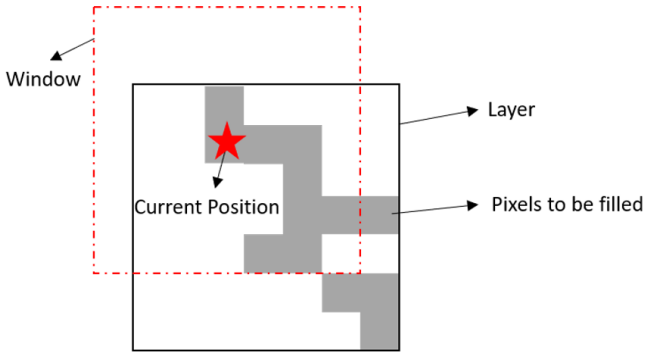


Figure 3.23: Window

3.5 Input Design

The input of the network is all the information obtained and utilized for determining the next action of the agent. So the input should contain enough information for the agent to make the right decision. However, when the information is too much, it is hard for the agent to tell the useful ones from the others. Thus, input design is essential in reinforcement learning.

Current input is the window centered at the laser position with the same size of the section, as shown in Figure 3.23. The part covering the target area in the window is grey and the other part is white. This kind of input contain the information of where the grey pixels are and the current position of laser, which are quite enough for the agent to act rightly in dense reward structure.

3.5.1 Budget

Budget is the number of remaining movements the agent can take. It is used to predict the reward and the value. First of all, budget can teach the agent that they must properly arrange the action of exploring and getting immediate rewards. Second, budget can help reduce the reward loss and value loss. There lies a problem about the assignment of target reward and value: what should be the target value and reward when the agent reaches maximum movements? It remains open. One alternative is to use the output of the network as the last target value and reward. Another alternative is to let them be zeros. In the current reinforcement learning structure, the latter is chosen. However, there is no information about it telling the agent that it will be terminated,

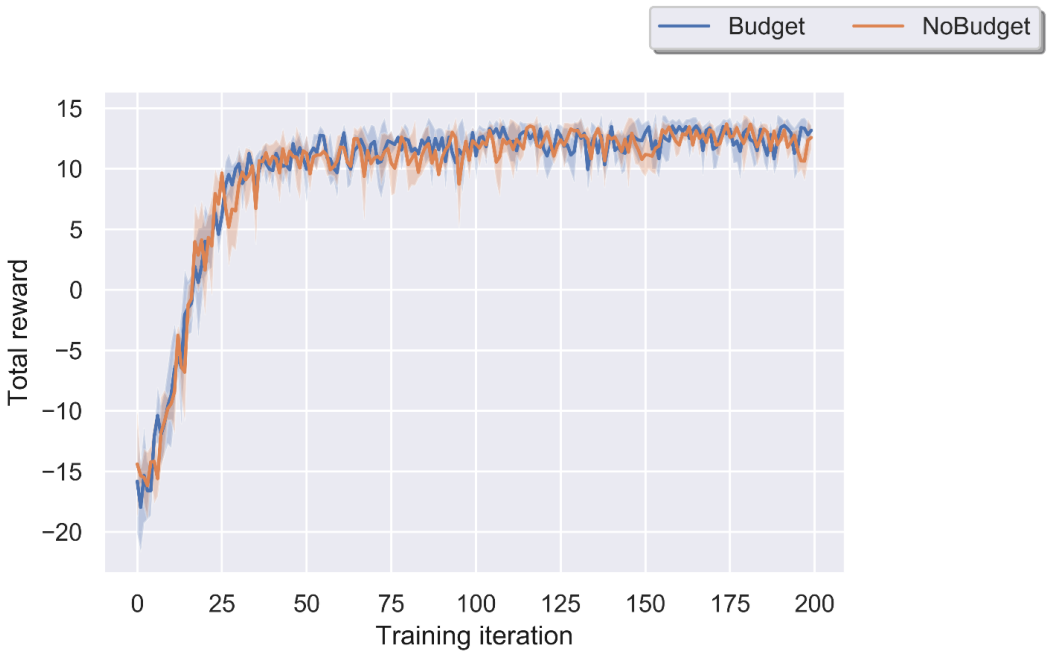


Figure 3.24: Total reward (budget)

the value and reward will be both zero. This will cause errors in network training. If budget is introduced, the agent will know it is going to run out of its "fuel" and the errors will decrease.

Two group experiments are conducted: one with budget and the other without budget. Each group contains three repeated experiments. The results are shown in Figure 3.24, 3.25, 3.26. The performance of two group are close. Total reward of the group with budget increases slightly faster and more stably. But its reward loss and value loss are much lower than the other. So budget should be introduced in this case.

3.5.2 Window Size

It is obvious that when the size of window is equal to that of sections, the window cannot reflect the information of the whole environment. So I doubled the window size and tested its performance. As shown in Figure 3.27, window size = 16 has better performance than window size = 8, which is consistent with theoretical analysis.

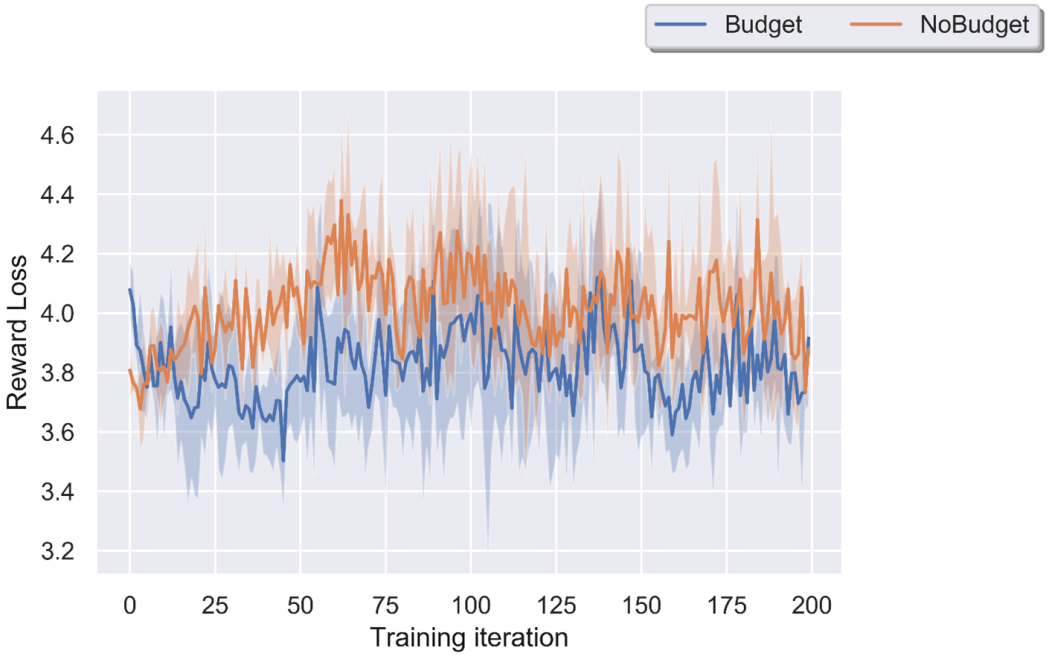


Figure 3.25: reward loss (budget)

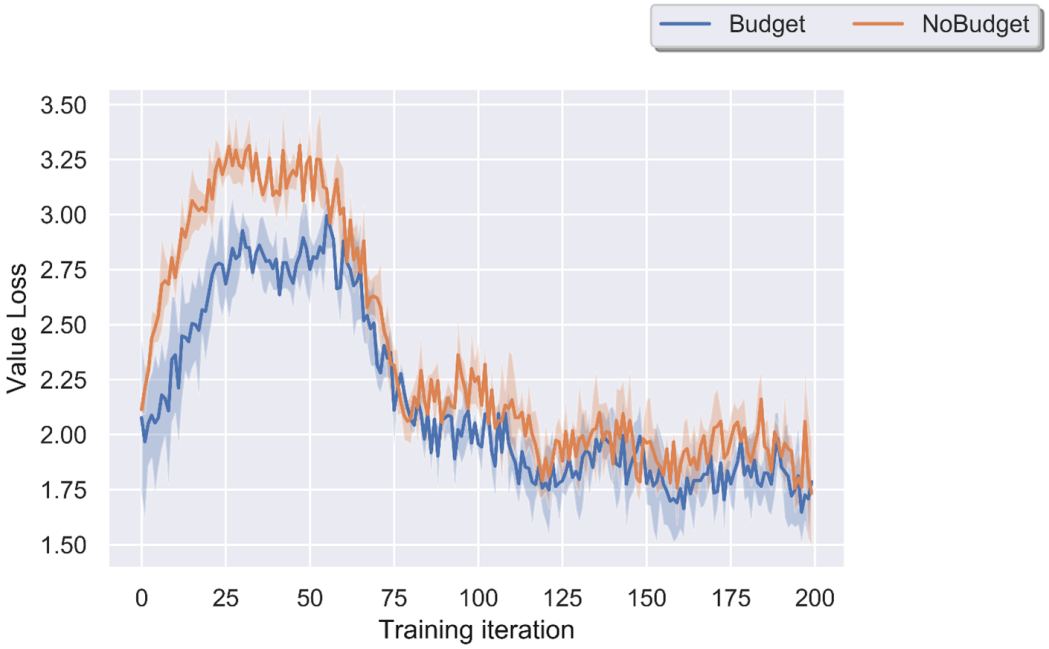


Figure 3.26: value loss (budget)

Figure 3.27: Total reward (window size)

3.5.3 Input Design

The numerical input of the window currently is shown in Figure 3.28(a). The values of grey pixels are 1 and the others are 0. This kind of input contains the information about target area to fill and the position of the laser. It is well-designed and compact. However, the agent cannot determine whether it reaches the boundary of the layer based on the current input. As we all know, the agent is not allowed to get out of the boundary so when the agent reaches, its action is limited. So current input is not capable of offering this message to the agent, impairing its performance. Thus, I added the information of the boundary into the input, as shown in Figure 3.28(b). The values of the pixels out of the boundary in the window are -1.

There are two group experiments. Each group contains three repeated experiments. The results in Figure 3.29, 3.30 and 3.31 shows the performance increases after adding the information of the boundary into the input. And the number of staying is significantly reduced to nearly 0. The reason is that in the simulation environment, when the agent chooses an action (invalid or illegal actions) to go out of the boundary, the agent is forced to stay at the same position. When the information of

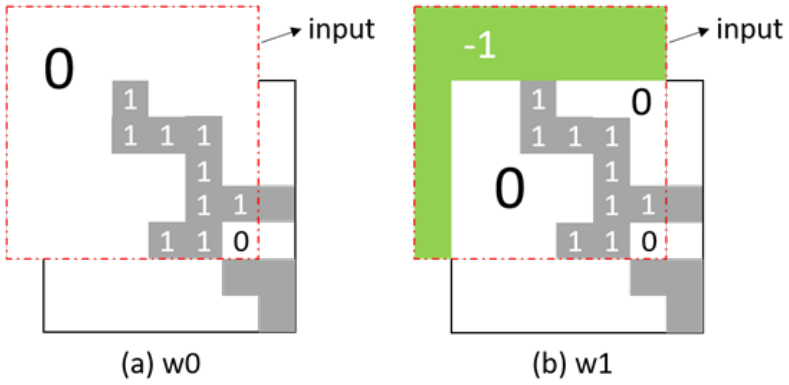


Figure 3.28: numerical input

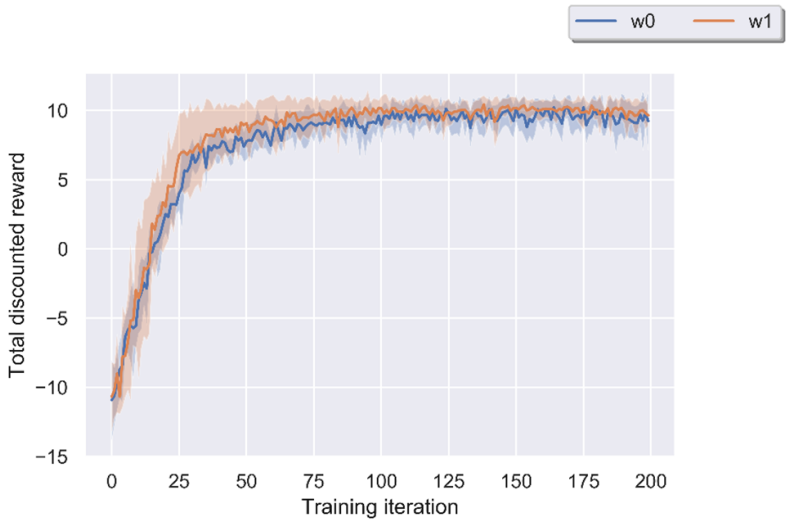


Figure 3.29: Total discounted reward (w0 w1)

the boundary is passed to the agent, the agent will learn it and know it is the boundary so it tends to choose valid actions. As the result, the value loss is smaller because the error to predict the reward is reduced.

Additionally, I designed several different inputs to test their performance. Two typical inputs are shown in the Figure 3.32. The experiment results are shown in Figure 3.33. It is obvious that i1 and i2 are harder for the agent to absorb useful information and the results are in accordance with it.

When the input is changed, the optimal hyperparameters will be different. So I attempted to change the hyperparameters to enhance the performance under i1 and i2. I selected several results

Figure 3.30: number of staying (w0 w1)

Figure 3.31: value loss (w0 w1)

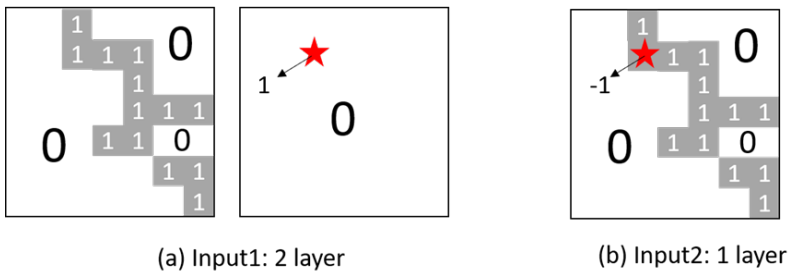


Figure 3.32: two different kinds of inputs

Figure 3.33: the performance of input 1 (i1) and input2 (i2)

and presented them in this report, as shown in Table 3.6 and Figure 3.34. The complexity of the network and other hyperparameters like Temperature Threshold are not prioritized. The first thing is to train the network sufficiently. So I tested the number of episodes, size of replay buffer and decay coefficient of the learning rate. From Figure 3.34, the conclusion is that hyperparameters should be tuned again after changing the input and the network is well trained because more training does not increase the performance significantly. And other attempts like increasing the depth of the network and encouraging the agent to explore more does not show any nice effect and sometimes impairment instead. All the results indicate that simple and compact input is beneficial in the reinforcement learning. Another finding is that the decay coefficient should not be too small, which will make bad effect in the later training loops. The reason may lie in the disability to go out of the local optimum if the learning rate becomes too small. In more depth, small learning rate will lead to local optimum where the reward loss, value loss and policy loss are all minimal locally if the training in each training loop is sufficient. It should be noticed that the training of the policy loss is to reduce the error between the policies of the network and the Monte Carlo Tree Search (MCTS). So when there exists a better policy, local optimum will bounce the attempt to reach the better policy back. Thus, too small learning rate is not always beneficial and that is why the performance becomes bad in the later training loop and is not able to be better again when the

Table 3.6: Hyperparameters

Group	Number of episodes	Size of replay buffer	γ in learning rate
i2	20	1000	0.99998
i2_e40_w2000	40	2000	0.99998
I2_e40_w2000_lrg0.9999	40	2000	0.9999

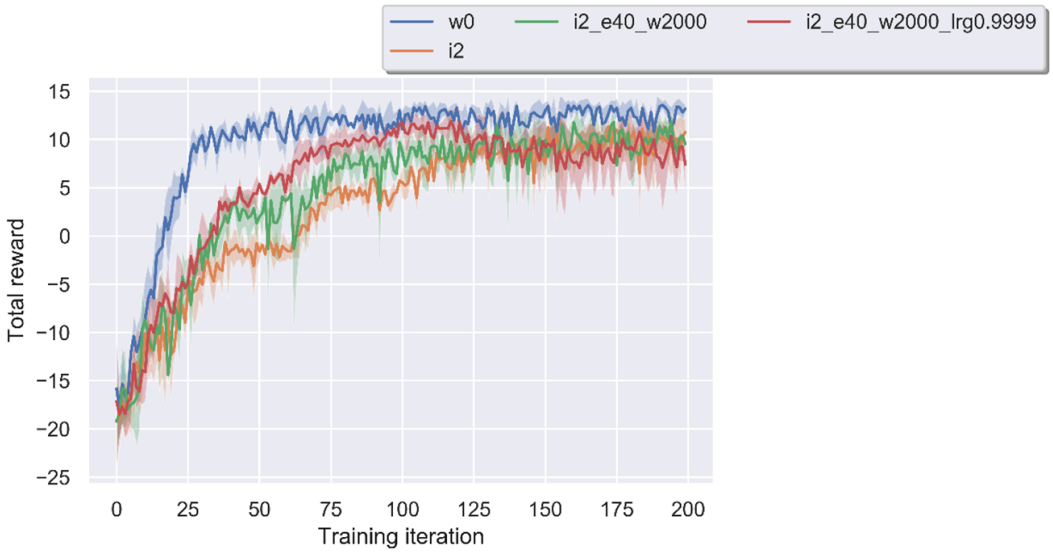


Figure 3.34: the performance of i2 with different hyperparameters

decay coefficient is too small.

3.6 Reward Assignment

The goal of the toolpath planning in additive manufacturing is to fill the target area as soon as possible and there is no area filled if not supposed to be filled. So the number of staying and moving uselessly should be as few as possible and filling wrongly (the laser is on when the position should not be filled) is not allowed at all. To realize this target, the core is to design the reward structure well. For example, if the penalty of staying is more serious than that of filling wrongly, the agent may tend to choose to fill wrongly rather than stay. So well-designed reward structure should reveal the priority between different interactions with the environment.

The idea case is that the agent learns to get no penalty but all the positive rewards. Through

Table 3.7: Different Reward Assignment

Group	filling correctly	moving uselessly	filling wrongly	stay still
Initial	1	-0.1	-0.3	-0.5
NW0	1	-0.1	-0.5	-0.3
NW1	1	-0.1	-0.7	-0.3

experiments, the idea case is hard to realize. MCTS is essential in the mechanism about how the reward assignment influences the optimal policy because the policy generated by MCTS is the target policy that the network tends to learn. However, the relationship between reward assignment and MCTS is hard to determine and requires more academic research. Experimentally, setting the reward based on the priorities is important.

Group experiments are conducted. Three repeated experiments are implemented in all the same settings except reward assignment. The reward assignments are shown in Table 3.7. The results are shown in Figure 3.35, 3.36 and 3.37. From the number of filling correctly, NW0 and NW1 exceed the Initial slightly. They are more stable in the later training loop. When increasing the penalty of filling wrongly, the number of filling wrongly occurs a significant decrease. On the whole, NW0 and NW1 can lead to a more stable performance because the std (the width of the translucent band) is relatively small. The conclusion is that the reward assignment should reveal the priorities and proper reward assignment is beneficial to generate a stable policy.

3.7 Pretraining to Accelerate the Training

Reinforcement learning is time consuming because it contains playing, interacting with the environment and training. In our 32x32 coverage path planning case, it always takes more than one day to finish one experiment. Generally, reinforcement learning policy (RL policy) requires 100,000s of games histories to learn effectively. As mentioned before, inaccurate reward and value prediction is bad for policy convergency. Inspired by [5], we intuitively think pretraining the representation network h may help the reinforcement learning training. The strategy is to establish the dataset of remaining grey pixels, actions, rewards and values. With the same network of Muzero,

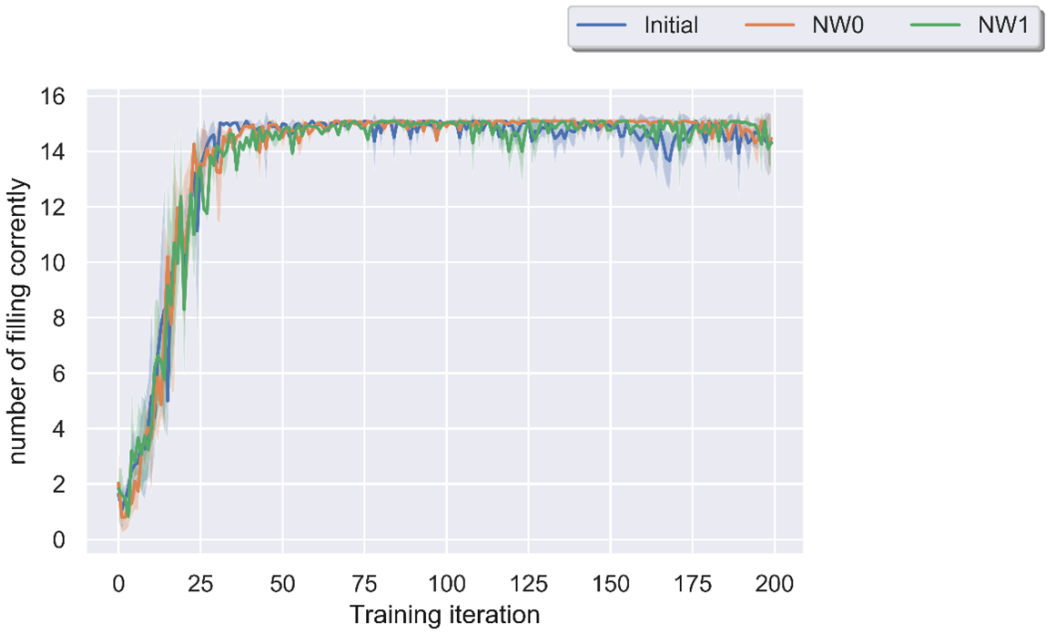


Figure 3.35: number of filling correctly (reward assignment)

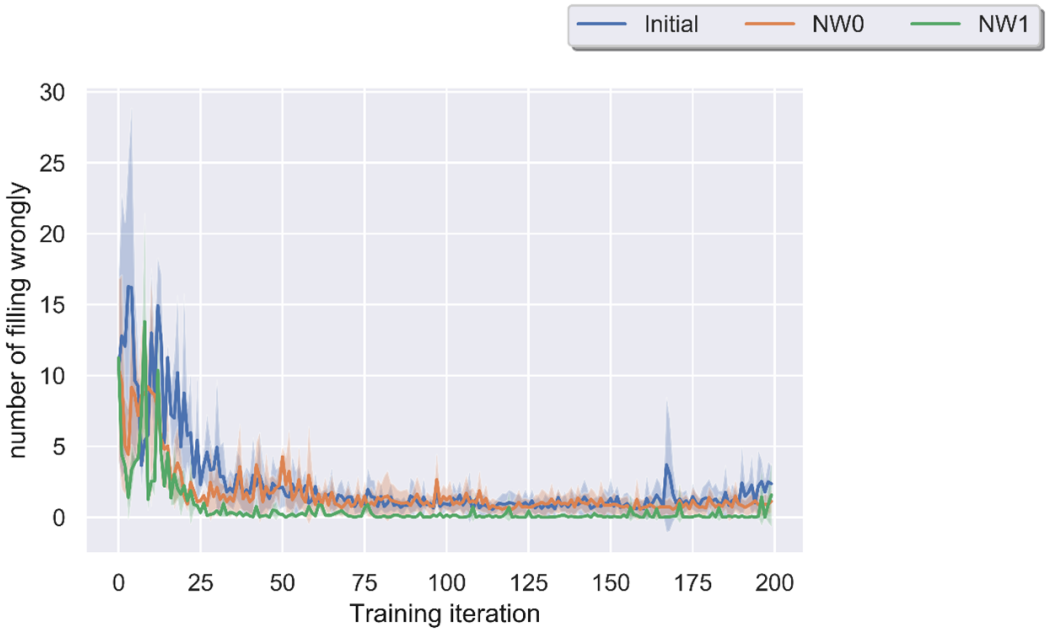


Figure 3.36: number of filling wrongly (reward assignment)

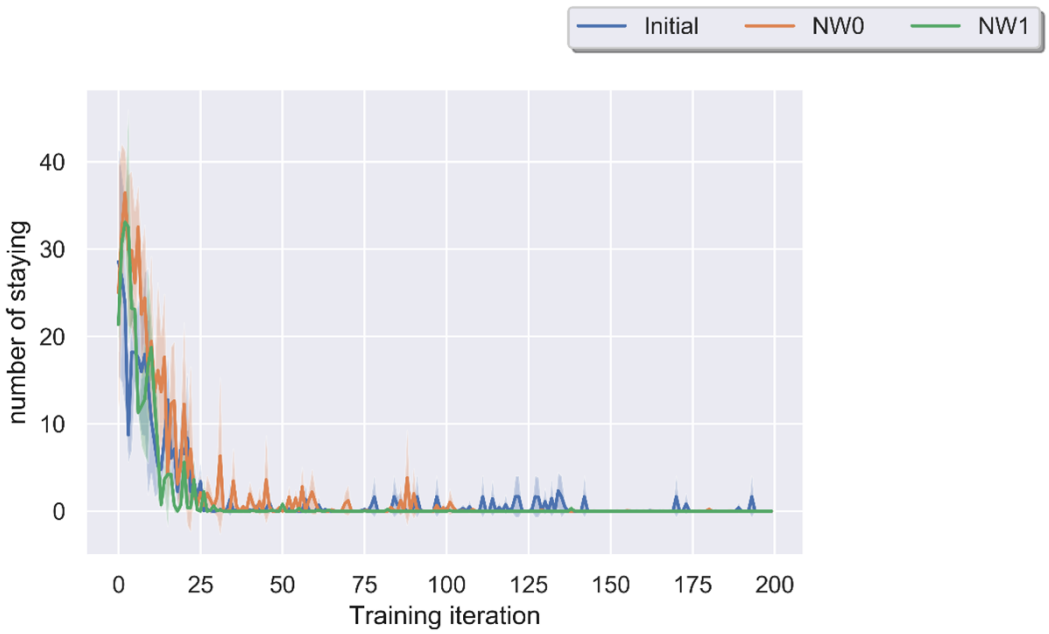


Figure 3.37: number of staying (reward assignment)

we can predict the rewards and values through remaining grey pixels and actions. Here comes to a supervised learning problem. We believe that after pretraining procedure, representation network can extract useful information and predict the value and reward accurately so that RL policy can fast go to convergency.

Dataset is established by 32x32 section data set, the same as what we use in toolpath planning problem. When training the network of Muzero, real inputs are the windows of sections where the machine has conducted some operations. So we set several kinds of toolpath like lines and rectangles, and add them randomly into the section in the dataset. Then we choose the current position of laser randomly and finally get the window as network input. Reward can be determined by the laser position, window and randomly chosen action. Value can be predicted as

$$V = r(\gamma^0 + \gamma^1 + +\gamma^{n-1}) \tag{3.8}$$

where r is the reward when filling a target pixel, n is the number of remaining target pixels, γ is the discount rate.

3.7.1 Neural Network Structure

When we train this supervised network, we find the total loss of value and reward cannot converge to a very low number as expected. As we all know, neural network can fit any function so supervised learning under idea settings can reduce the loss into a small number. We begin to suspect whether our neural network is capable to predict the value and reward. And the capability of the neural network is essential in reinforcement learning.

First, we change the number of channels for each layer. Figure 3.38 shows the result. We can conclude that with the increase of number of channels, total loss of training can be reduced but results from testing say that overfitting will occur if we choose too many channels. Because the loss here is calculated from support vector [4] of the scale, which does not represent the scale value directly. To be more intuitive, (b) shows mean absolute error (MAE) of value (not reward) dialog. Maximum value is less than 32x32=1024. Assume average value is 300. So the MAE rate is around 0.8%˜4%. This is acceptable in our case because the target value is not accurate. And the MAE of reward is always less than 0.01, which means MAE rate is less than 1%. So we can conclude our neural network with 32 channels has the capability to predict the value and reward accurately. MAE rates of value and reward are both less than 1%. To increase the capability, we can choose 32 channels and stop the training when test loss increases. But 32 channels will cost much more time than 8 channels.

The effects of expanding the neural network in depth and breadth are similar. So second, we turn to another factor the structure of residual blocks. Traditional residual block structure is shown in Figure 3.39 (a), which is used in the initial Muzero. According to [6] , it would be better if we put BN layer and ReLu layer before Conv layer, which is Residual block 2 as shown in Figure 3.39(b). To enhance the adaptability of the residual block, we make it work when the number of input channels differs with that of output channels based on [7], which is shown in Figure 3.39(c). Figure 3.40 shows experiment results. From Figure 3.40(a), we can conclude residual block 12 have similar and better performance. From Figure 3.40(b), residual block 2 has a good calculation efficiency. All in all, we should utilize the residual block 2 structure.

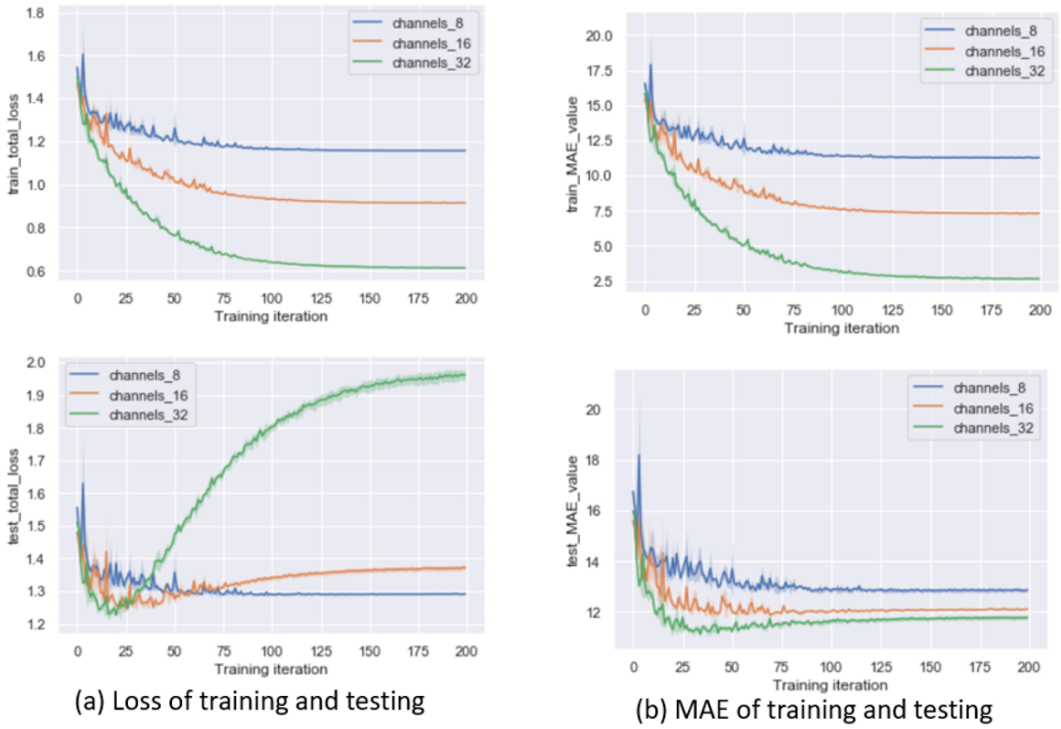


Figure 3.38: Loss and MAE

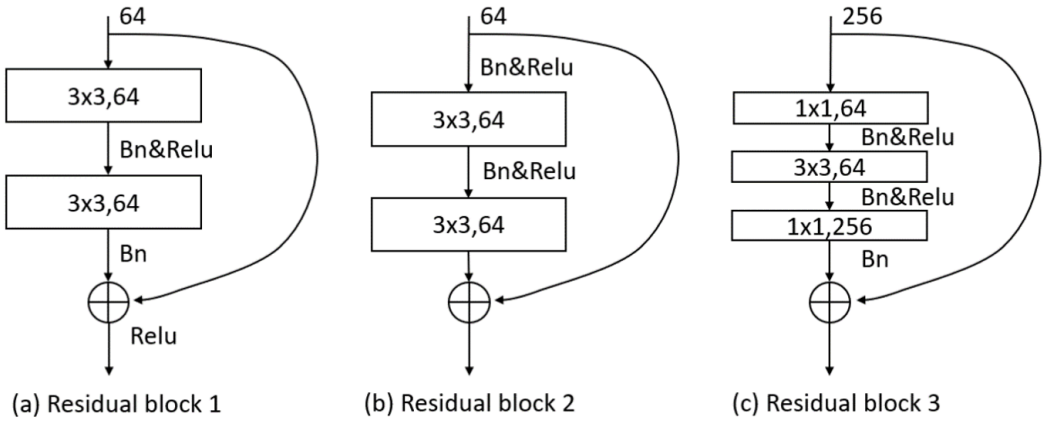
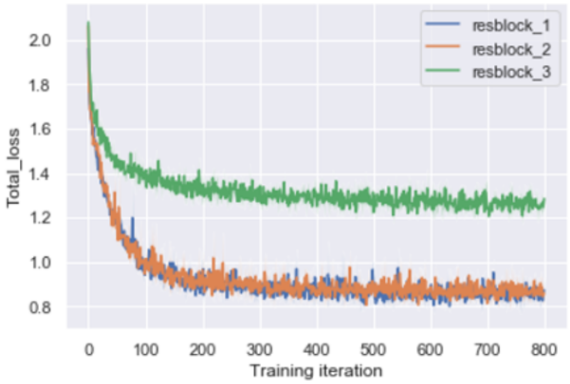


Figure 3.39: Residual block structures



(a) Total loss

resblock	experiment	time
resblock_1	1	1h2m32s
resblock_1	2	1h1m59s
resblock_1	3	1h1m19s
resblock_2	1	37m42s
resblock_2	2	37m37s
resblock_2	3	37m22s

(b) Cost time

Figure 3.40: Results (ResNet)

3.7.2 Pretraining Results

After pretraining, we can fix the weight of representation network, which means using the hidden state calculated by pretrained network. We can also use the weight of pretrained network as the initial network of Muzero. This is similar to transfer learning. From experiments, the result of the latter procedure is similar to that with no pretraining. So we fix the weight in the later experiments.

We can pretrain the reward and value at the same time. But the result is bad. Total reward cannot increase an obvious number. The reason we suppose is that the target value in the dataset is not accurate. So we pretrain the reward only. Figure 3.41 shows the result which is not good as expected. The total reward increases faster than that with no pretraining but later the total reward cannot reach the expected number. The reason we suppose is that hidden state extracted is only useful to reward prediction.

From above analysis, we can conclude that pretraining methods we have tries are useless in our case.

3.8 Sparse Reward

In sparse reward structure, the agent rarely gets nonzero reward and commonly it gets a nonzero reward in the end of the task. Whether the agent performs well in sparse reward structure is essential to toolpath planning in additive manufacturing. Toolpath directly influences the mechanical

Figure 3.41: Total reward (Pretraining)

properties of the product. For example, certain path patterns like zigzagz [2] will strengthen the toughness, strain-at-fracture and other mechanical properties. Sequence of filling also makes an impact because of cooling down of the materials [8]. Some influence can be predicted through simulation but it is also quite hard and costly to simulate. In general cases, the influence can only be determined in the end of the whole manufacturing process or in several points during the process. To capture these influence, sparse reward structure should be utilized.

3.8.1 Td_step

Td_step is essential to RL in sparse reward structure. It denotes the number of steps used to calculate the target value and target reward. If the reward is sparse and Td_step is small, the training will be often useless because the reward is often zero.

In the experiment of this section, we denote the pattern in the action history as the toolpath pattern which is a series of continuous actions of the agent. Specifically, we denote "up, left, down, right" as the pattern. In the end of the task, the agent will be given a reward of n if the agent goes up, left, down and right in order for n times. Thus, it is not suitable to take one single window of the environment as the input because it does not contain any information about action history. So I designed the trajectory window as the input. The trajectory window is in the same size of

Table 3.8: Reward Assignment

n patterns in the end	stay still	Others
n	-0.5	0

Figure 3.42: total reward (sparse reward)

the section. Initially, the trajectory window is a zero matrix. At ith step, the value of the current position in trajectory window becomes i/max_movements. So the trajectory window records the trajectory of the path indicating the action history. The reward assignment is shown in Table 3.8

Considering the reward is nonzero until the end of the game (at max_movement step), td_step should be max_movement. So it can always get a valid target reward and value, which is supposed to benefit the training. To valid this analytical conclusion, two group experiments are conducted: first is td_step=10 and the other is td_step=50 (max_movement = 50). Figure 3.42 shows two good group experiments (each group experiments are repeated 5 times and the rest cannot get any positive reward). We can initially conclude that big td_step is beneficial in sparse reward structure.

3.8.2 Prioritized Replay Buffer

Prioritized replay buffer is to sample the data for training with certain priority [9]. Define a transition as the atomic unit of interaction in RL, which is used for training. For example, transition i is (state S_{i-1}, action A_{i-1}, reward R_i, discount γ_i, next state S_i). In RL, the targets for training the network are obtained from sampled transitions. The probability of sampling transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.9}$$

where p_i is the priority of sampling transition i. In general, the goal of prioritized replay buffer is to decrease the td errors. So the transitions with bigger td errors should be more likely to be sampled to decrease the whole loss faster. Denote the td error of transition i as δ_i. Thus, there are two ways to realize it. First is that p_i is proportional to $|\delta_i| + \sigma$ where σ is the bias. Second is that p_i is proportional to $\frac{1}{rank(|\delta_i|)}$. Through experiments, the second method is generally more stable and reliable.

Apply the ranked prioritized replay buffer into the experiments, and prioritized replay buffer decreases the loss significantly but the total reward is much lower than randomly sampling. The reason may be the loss reaches the local optimum and the network is not capable of learning a better policy. What prioritized replay buffer really do is to accelerate it and strengthen it because it always samples the data with the large TD errors. So the policy is similar to the current policy generated by MCTS and cannot be a better one. There are several methods to avoid it. For example, changing the exponential coefficients or adding a bias can make prioritized sampling more like randomly sampling. But the performance cannot exceed that of randomly sampling through theoretical analysis and experiments. How to solve it remains a problem. Additionally, priority can also be based on the total reward (for game sampling) or positive reward (for position sampling), not the td error. The agent should learn more from the successful experiments compared to those failures. Through experiments, sampling the games based on total reward and sampling the position of each game based on td error will be better. But the results do not show imposing increase

compared to randomly sampling. However, prioritized replay buffer is common in reinforcement learning. It should work. The reason and the solution to it require more research. On the other hand, the number of the game stored in replay buffer is limited. So how to choose the games to stay or leave may be a problem in RL. However, from Figure 3.34, we can know that in some cases, at least in dense reward structure, the network is trained sufficiently and the size of replay buffer is enough. So the explore of the method to sample the games and game positions may not have much profit.

3.8.3 Reward Assignment

Only giving positive reward for the appearance of desired pattern is not in accordance with reality. Actually, the toolpath should fill the target area and at the same time contain as many desired patterns as possible. In anther perspective, desired patterns are wanted which should be in the target area. The reward is also sparse and directly learning from sparse reward is hard. But teaching the agent to fill the target area (grey pixels) is relatively easier. And filling grey pixels is at a lower level in the task hierarchy. So giving a small positive reward to the agent when it fills a grey pixel and giving a much bigger positive when desired pattern appears is reasonable and widely-used in RL. On the current stage, a simple setting of the sparse reward structure where the agent can perform well should be valuable. Once simple case can be solved, harder and more general cases can be further studied with more experience and methods obtained from simple cases. So the reward of the appearance of desired pattern is not given to the agent in the end of the game. Whenever the desired pattern appears, the agent will be rewarded. Of course, the same desired pattern will not be rewarded twice.

Denote the desired pattern as 'Π' patterns and its three other varieties which rotate by 90, 180, 270 degrees. The reward assignments are shown in Table 3.9. r0 is a typical sparse reward structure and the others are not. The input is two layer with size of 8x8. One layer is the window with boundary w1 and the other is similar to trajectory window which only records the last four positions. And if the laser is on, the value is positive; otherwise, the value is negative.

Table 3.9: Reward Assignment

Group	'Π' pattern	filling correctly	filling wrongly	moving uselessly	stay still
r0	4.0	0.0	-0.3	-0.1	-0.5
r0.3	3.7	0.3	-0.3	-0.1	-0.5
r0.5	3.5	0.5	-0.3	-0.1	-0.5
r1	3.0	1.0	-0.3	-0.1	-0.5

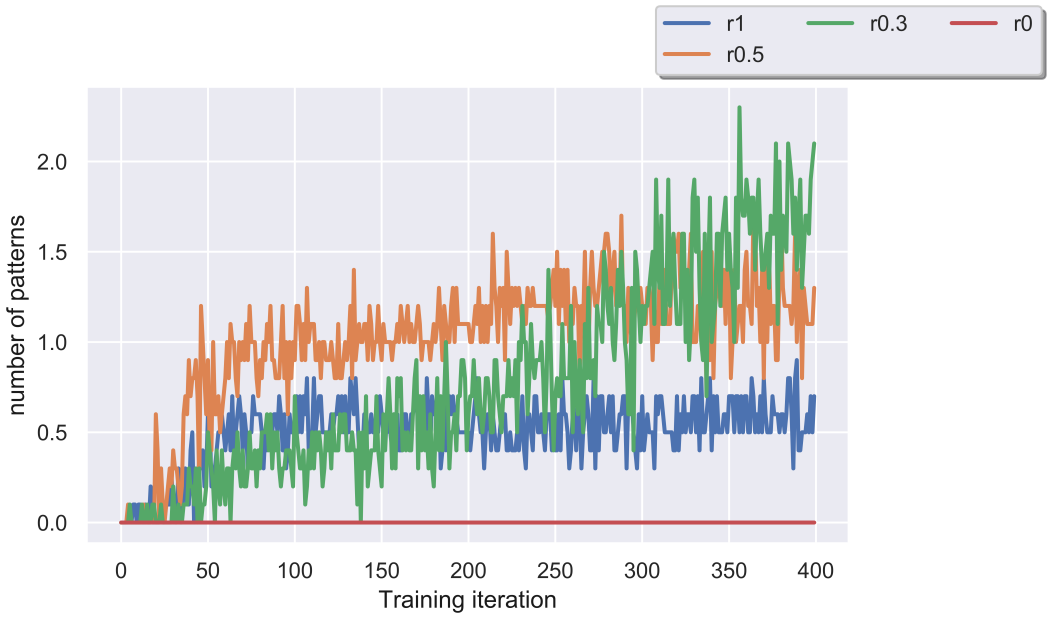


Figure 3.43: number of patterns (reward assignment r)

The results in Figure 3.43 show that the performance is r0.3 $>$ r0.5 $>$ r1 $>$ r0. So setting small target will help the agent perform well in sparse reward structure. Proper greedy strategy is beneficial but the hierarchy between the targets should be more obvious so that the agent will tend to get the most valuable reward.

3.8.4 Input Design

I tried several kinds of inputs from the most common ones like sequential images of the environment and those stacked with action histories as described in the paper of Muzero. Though the performance is not very promising, the agent does learn something. And the lesson from the

experiments is that simple and compact input is vital to RL.

3.9 Technical Approach

3.9.1 Distributed Calculation

Training reinforcement learning network is time consuming. In our case, it takes several days to finish one experiment. We should take some methods to accelerate the training.

In the initial version of the Muzero, we adopt a linear calculation method. First, we make agents play games one by one and then train the network. We can use different CPU cores to play games simultaneously and when agents are playing, we make GPU to training the network. We realize this kind of distributed calculation through python package ray.

3.10 Summary

This chapter introduces reinforcement learning application to toolpath planning in additive manufacturing. In dense reward structure, most hyperparameters are analyzed and tested. Reward assignment and input design are discussed with sound theoretical analysis and experiments. In sparse reward structure, initial exploration is attempted and the results are beneficial to the future research.

CHAPTER 4
CONCLUSION AND FUTURE DIRECTIONS

4.1 Improved Traditional Toolpath Planning

4.1.1 Conclusion

1. General solution to planar coverage path planning problem is to first segment surface, local path planning and combined visit order planning. In the previous case, surface segmentation is based on connected graph. Local path planning is based on zig-zag. Combined visit order planning is path type and visit order planning.

2. Improved traditional toolpath planning can significantly increase the production efficiency and decrease the number of switching the laser. And this algorithm is reliable and suitable for arbitrary product.

4.1.2 Future Direction

1. Connected graph is not a suitable standard to segment surface. Morse decomposition [10] can be utilized. Or convex connected graph will be more useful.

2. Number of turns is not considered in this thesis because the experimental influence of turns is not determined well. In the future research, when number of turns become essential to additive manufacturing, it can be taken into consideration.

3. Local path planning algorithm can be changed to suit certain purposes in additive manufacutring.

4.2 Toolpath Planning through RL

4.2.1 Dense Reward

Conclusion

1. Fine-tuned reinforcement learning method can handle toolpath planning problem in dense reward structure well.

2. Compared to special hyperparameters of Muzero, general hyperparameters in RL are more vital to the performance. It is recommended to keep the iteration-varying hyperparameters of Muzero constant because the performance is usually sensitive to them. There are mathematical explanation behind special hyperparameters of Muzero. Keep them within a reasonable range and tune general hyperparameters. Tune one hyperparameter and check whether the change of performance agrees with theory through inspecting the intermediate variables.

3. Input should contain enough information about the environment and be brief for the agent to understand. Proper artificial design is vital to a great performance

4. Reward assignment should reflect the target of the task. The hierarchy of the reward determines the priority of the reactions in the current environment. Value of rewards influence where is the convergence of the policy. Multiple attempts are required to find a proper reward assignment.

Future Direction

1. Reward assignment can influence the final optimal policy but there exist the hierarchy of the policy in reality. For example, filling wrongly is not allowed in reality. Post-processing can be used but whether we can get the perfect policy is a interesting research direction. What in my mind is to divide complex task into simple tasks, , train the policies respectively and create the hierarchy of the policies. The inspiration is that when task is simple, the optimal policy is more likely to be perfect.

2. Whether the agent can learn how to design the input and reward itself is very challenging and has much potential in RL.

3. How the values of rewards influence the optimal policy requires theoretical machine learning research.

4.2.2 Sparse Reward

Conclusion

1. Through proper tuning and process mentioned in this thesis, the agent can learn sparse reward to some extent. Td_step should be big enough. Proper input design and reward assignment are beneficial.

2. Setting small target is helpful in sparse reward structure.

Future Direction

1. It can not draw to conclusion that prioritized replay buffer may not be very useful in current stage. Theoretical reason should be studied.

2. Hyperparameters for sparse reward structure may not be well-tuned. And its tuning methods may be different from those in dense reward structure, which will be a valuable technical topic.

3. More current algorithms for sparse reward structure should be tested.

4. How to teach the agent in the sparse reward structure only with a few attempts will be very challenging and interesting topic.

5. The reward assignment in sparse reward becomes complex, causing the agent hard to complete the task well. Policy hierarchy may be used to solve this problem

REFERENCE

[1] Xia, L., Lin, S., Ma, G. (2020). Stress-based tool-path planning methodology for fused filament fabrication. Additive Manufacturing, 32, 101020.

[2] Allum, J.,Kitzinger, J.,Li, Y., etc. ZigZagZ: Improving mechanical performance in extrusion additive manufacturing by nonplanar toolpaths[J]. Additive Manufacturing, 2021, 38 101715.

[3] https://medium.com/applied-data-science/how-to-build-your-own-muzero-in-python-f77d5718061a.

[4] Schrittwieser, Julian, et al. "Mastering atari, go, chess and shogi by planning with a learned model." arXiv preprint arXiv:1911.08265 (2019).

[5] Mirhoseini, Azalia, et al. "Chip Placement with Deep Reinforcement Learning." arXiv preprint arXiv:2004.10746 (2020).

[6] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.

[7] He K, Zhang X, Ren S, et al. Identity mappings in deep residual networks[C]//European conference on computer vision. Springer, Cham, 2016: 630-645.

[8] Volpato, N.,Zanotto, T. T. Analysis of deposition sequence in tool-path optimization for low-cost material extrusion additive manufacturing[J]. The International Journal of Advanced Manufacturing Technology, 2019, 101 (5): 1855-1863.

[9] Schaul, T.,Quan, J.,Antonoglou, I., etc. Prioritized experience replay[J]. arXiv preprint arXiv:1511.05952, 2015.

[10] Acar, E. U.,Choset, H.,Rizzi, A. A., etc. Morse decompositions for coverage tasks[J]. The International Journal of Robotics Research, 2002, 21 (4): 331-344.