國立東華大學運籌管理研究所

碩士論文

指導教授：溫日華 博士

貨櫃遷儲問題 ── 以人工神經網絡為基礎之方法

An Artificial Neural Network-Based Method for the Container Relocation Problem

研究生：楊文富　撰

中華民國一〇八年三月

摘要

現代貨櫃運輸越來越重要，因為櫃中的貨物比其他海運貨物更具價值，所以人們希望能夠更有效率地運輸貨櫃。貨櫃碼頭是船與貨車的中轉站。在貨櫃碼頭中，有很多改進貨櫃轉運效率的議題，貨櫃遷儲問題(Container Relocation Problem)也是其中之一。對此問題，目前已有不少解決辦法，本論文挑選了 Min-Max 和 Look-ahead *N* 兩種啟發式演算法搬運、提取貨櫃的方法，用人工神經網絡(Artificial Neural Network, *ANN*)去學習這兩種啟發式演算法，測量學習成效，再用人工神經網絡學習兩者間較好的搬運提取方式，觀察人工神經網絡能否超越這兩種方法，找到更好的解。

根據貨櫃堆的大小，我們設計了兩組了不同的實驗：四高三寬、內含 7 個貨櫃的貨櫃堆(小型貨櫃堆)以及四高六寬、內含 18 個貨櫃的貨櫃堆。我們做了三種不同類型的資料集，除了模仿 Min-Max 和 Look-ahead *N* 兩種啟發式演算法所創造出的資料集，另外從兩種資料集中，挑出表現比較好的資料，組成新的資料集供 *ANN* 學習，觀察人工神經網路是否能超越兩種啟發式演算法。

實驗結果顯示，在小型貨櫃堆中，人工神經網絡不但能完美地學習兩種啟發式演算法如何遷儲貨櫃，還能超越原本的兩種方法，減少遷儲貨櫃的次數。在大型貨櫃場裡，人工神經網路無法完美地學習兩種啟發式演算法，亦不能超越他們，不過遷儲貨櫃的次數與兩種啟發式演算法相當接近。最後，論文討論 Adam 和 Mini-batch 兩種減少 *ANN* 運算時間的方法。

關鍵字： 人工神經網絡、貨櫃遷儲問題、Min-Max 演算法、Look-ahead *N* 演算法

Abstract

Container transportation have become more important in modern world because goods in containers are more valuable than other means of maritime transportation. Thus, people hope to effectively transport containers. Container terminals help transship containers between vessels and trucks. The Container Relocation Problem (*CRP*) is an issue related to the improvement of container terminals. There have been methods proposed for the *CRP*. Here, we first choose two different heuristics, Look-ahead *N* and Min-Max for the *CRP*, and apply Artificial Neural Network (*ANN*) to imitate how these two heuristics reshuffle containers; then by learning from the best of the two heuristics we check whether the performance of *ANN* can surpass them.

We do experiments on two types of bay size: 4-row, 3-column, and 7-container bay size (small bay) and 4-row, 6-row, and 18-container bay size (large bay). Besides following the logic of two heuristics to generate datasets, we form a new type of datasets by combining best data instances of two heuristics. We train many *ANNs* for Min-Max, Look-ahead *N* and Best-of-Two to set their parameter values. Then we use the trained parameters from different *ANNs* to reshuffle containers and compare the results with the original reshuffle results of heuristics. *ANN* perfectly imitates the two heuristics and surpasses them in combined datasets that we generate for small bay size. For large bay size, *ANN* is unable to imitate nor surpass the two heuristics but the results are very close to them. In the end, we do further analysis on two methods to reduce computational time of training *ANN*s.

Keywords: Artificial Neural Network; Container Relocation Problem; Min-Max Heuristic; Look-ahead *N* Heuristic

Table of Contents

List of Figures

List of Tables

I. Introduction

1.1 Study Background

Maritime transportation has been a primary means of transporting goods around the world for centuries. By now, nearly 80% goods of trade is transported by Maritime transportation. In maritime transportation, there are different ways to transport goods according to types of goods. Including liquid bulk, dry bulk, break bulk, roll-on / roll-off (ro-ro) and container cargo.

The amount of international seaborne trades in 2006 was 7,700 million tons, growing to 10,287 million tons in 2016. Dry bulk goods accounted for 4,888 million tons, the largest in 2016. Oil and gas are of 3,055 million tons. Containerized goods of 1,720 million tons are the third largest (UNSTAD 2017). With increasing demand for consumer goods transported around the world, the need for container vessels grows higher year by year, and for the same weight, the value of goods in the containers is usually higher than dry bulk and liquid bulk. The invention of container transportation speeds up maritime transportation with improved security in goods. The need for container has increased from 1,076 to 1,720 million tons from 2006 to 2016. At the same time, the amount of 20-foot equivalent units (TEUs) has increased from 90 million to 140 million from 2006 to 2016 (UNSTAD 2017). People hope to improve the efficiency of container transportation more than the other ways of transportation.

Container terminals play an important role on processing containers for vessels. A container terminal is a place for vessels to transship containers to trucks, and vice versa. Vessels need to be loaded and unloaded containers at container terminals. With less processing time in container terminal, containers can be loaded on vessels in shorter time. It is critical to efficiently utilize container terminals to handle such a large amount of daily transportation. Figure 1.1.1 shows a diagram of the container terminal (朱威倫 2017). The operations of container terminals are complex. Many issues are discussed to save time and reduce transshipment cost of container terminals. Before the arrivals of vessels, scheduling vessels to various berths in advance is essential due to limited number of berths at a container terminal. This is called the *berth allocation problem* (BAP). There are different factors influencing the problem, such as waiting time, berthing time, different types of vessels, etc. See, for example Cordeau et al. (2005).

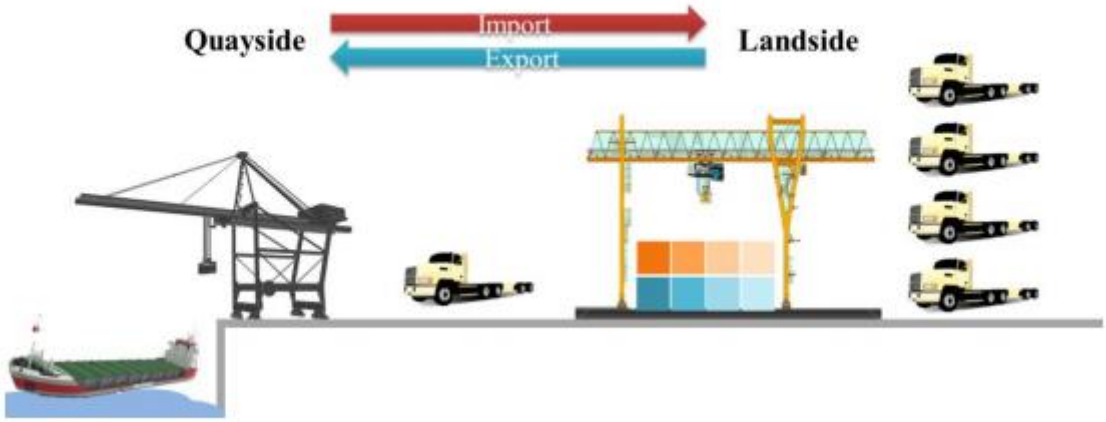


Figure 1.1.1. Container terminal
圖 1 港口流程圖 in（朱威倫 2017）

When a vessel arrives at a berth, *quay cranes* (QCs) will unload containers to trucks, which transport containers to the storage yard; during loading, the process is reversed, with trucks taking containers from the storage yard and load them to vessels by QCs. The determination of the number of QCs allocated to a vessel is called the *crane allocation problem*. The objective

can be to load and unload containers faster in lower cost with limited number of cranes. The problem of deciding the positions of containers on vessel is called the *stowage planning problem*. The storage slots of containers can be allocated by weight, destination, and requirement of refrigeration or not. Zhang et al. (2003) discuss more detail about the problem.

Figure 1.1.2 shows a diagram of container block, which is a place for temporarily storing containers. Lots of container blocks are set in a container terminal. A container block includes lots of bays. Figure 1.1.3 shows a bay. Several containers form a stack which means a column in a bay. Several stacks form a bay. A truck transports containers from QCs to the storage yard, where yard cranes take and put the containers to store in bays of blocks. The process can also be done by *straddle cranes* (SCs) which can both transport and load the container to the stack. Nowadays, *automatic guided vehicles* (AGVs) are utilized to transport containers in automated container terminals. No matter what transport means is applied, vehicles, SCs, or AGVs, to find a suitable route to the storage yard can be modeled as the *vehicle routing problem* (VRP). Besides these problems mentioned in Section 1.1, Carlo et al. (2014) give more comprehensive review about container terminals, including the trend of container terminals, and research directions.

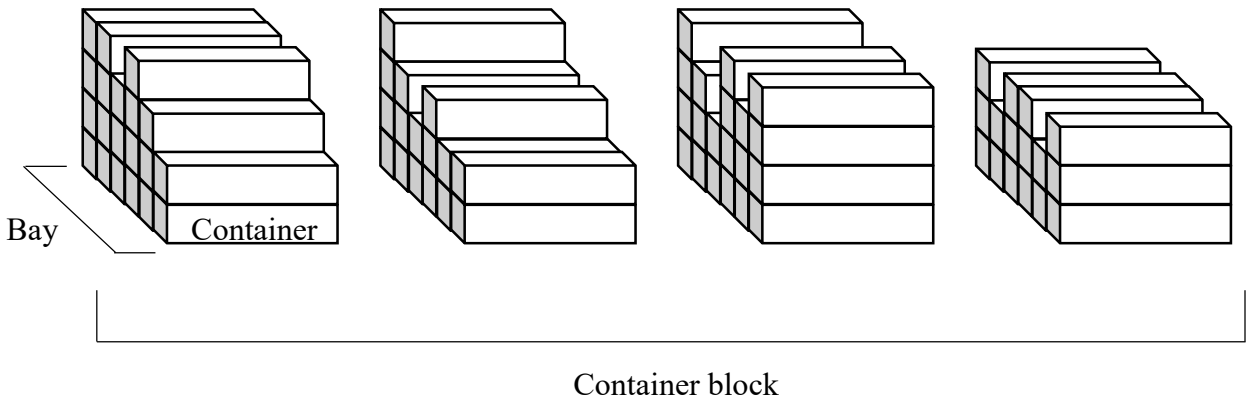


Figure 1.1.2. Container block

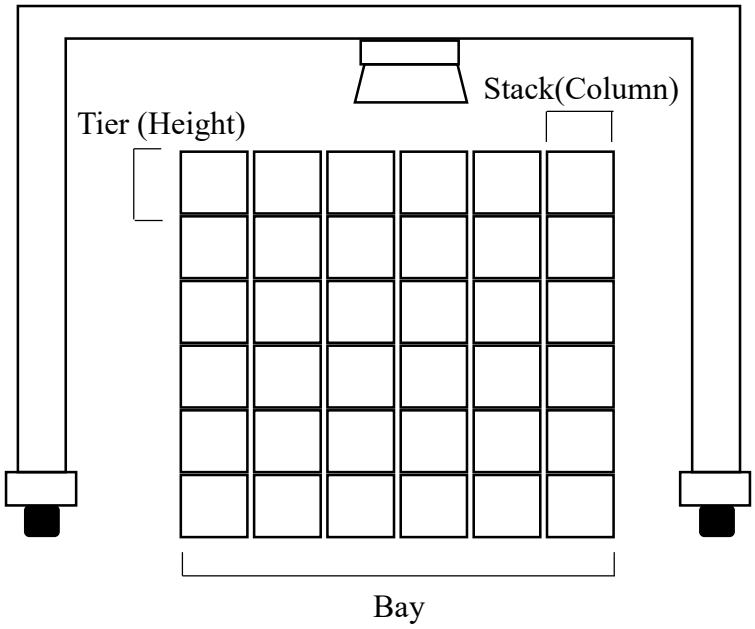
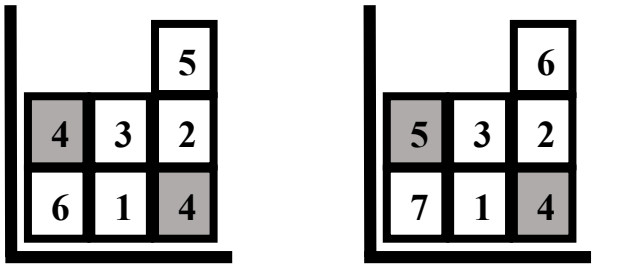


Figure 1.1.3. Bay.

Before a vessel arrives at the berth, containers will be allocated to storage locations on the vessel according to the weight, destination, category and other factors. The storage locations determine the loading sequence of containers and containers are encoded to represent the loading sequence into vessels. There are two ways to encode: Individual identification and group identification. Containers of the same destination and similar weight can be put in the same group, and the loading sequence of containers in the same group does not influence the whole loading plan of containers. This is called group identification. In our study, we use individual identifications. We encode the *retrieving* sequence of containers by consecutive positive integers. Lower positive number means higher retrieving priority. In Figure 1.1.4, containers with same number are in the same group. In group identification, when the retrieval order requests Container 4, the crane is able to retrieve container 4 in either Column 1 or Column 3. The choice does not affect the loading process on vessel. On the contrary, if containers are encoded as individual identification, the loading sequence are fixed without any flexibility in group identification. The container bay on the right side of Figure 1.1.4 shows

individual identifications that containers with gray are turned to different numbers for which the loading order is inflexible. In our study, we assume that all containers are marked with individual identifications.



Container with group identification

Container with individual identification

Figure 1.1.4. Different identification

Here are assumptions for *retrieving* containers on the bay:

1. The yard crane only moves containers on the top of stacks.
2. The yard crane only moves one container each time.
3. The retrieving sequence follows the identifications of containers.
4. The number of containers in a stack are not allowed to surpass the limited height.

The process of taking a container away from a bay by a yard crane is called *retrieving.* The process of moving containers among stacks in the bay is called *reshuffling.* Containers with higher priority to be retrieved on the vessel sometimes are under containers with lower priority. The higher priority container is *blocked*, leading to a *deadlock.* In Figure 1.1.4, Container 1 is blocked by Container 3. Before retrieving a container, any container above it needs to be reshuffled to other stacks. The objective of reshuffling is to smooth the bay that no blocked container is in the resulted bay. Reshuffles are unprofitable moves. More reshuffles increase time on loading containers and the vessel stays longer at berth. Thus, less reshuffles can reduce

the total loading time and increase the turnover rate of berths and vessels. The problem of smoothing containers and retrieving containers according to the encoded sequence is called the *container stacking problem.*

Container stacking problem is divided into two subproblems, the *pre-marshalling problem* and *container relocation problem*. In the pre-marshalling problem, the crane reshuffles containers until the whole bay is smooth (i.e., without any deadlock). The objective of pre-marshalling problem is to minimize the number of reshuffling times or the total time on reshuffling. There have been many papers proposing different methods, including integer programming or heuristic algorithms to solve the problem. See Lee and Hsu (2007), Caserta and Voß (2011), and Tierney et al. (2015).

Another subproblem is called the container relocation problem (*CRP*), which basically empties a bay. In pre-marshalling problem, the yard crane reshuffles containers in advance without any container retrieval until the bay is smooth. In *CRP*, the crane retrieves containers no matter the bay is not smooth or not. The crane does reshuffle when the container is blocked by other containers. The crane keeps reshuffling and retrieving until the bay is empty. There have been lots of methods on solving the *CRP*. The objective of those methods is to minimize the total moves or minimize the total time on retrieving containers. With fewer moves of containers or shorter time to reshuffle containers on the yard, the quicker the yard crane can empty all containers, the higher the bay utilization. In this thesis, we will solve the container relocation problem (*CRP*) to empty a container bay. We plan to use the artificial neural network (*ANN*) to learn from two different heuristic algorithms on solving the *CRP*, making *ANN* learn from the better of two heuristic algorithms and seeing whether *ANN* can get a better solution than both heuristic algorithms. The coding is by Python.

1.2 Study Motivation

In 2016, AlphaGo, Deepmind's artificial intelligence software for playing Go game, defeated Lee Sedol, one of the top professional Go players in the world. It is not the first time that a human player was defeated by artificial intelligence in a board game. However, people around the world are surprised because Go game is one of the most complex board games in the world. AlphaGo's achievement inspires more people to discuss the areas that can be applied by machine learning.

Artificial neural network (*ANN*) is one of the algorithms in machine learning. Though *ANN* has been proposed for decades, it is not popular until two major improvements. First, data is easier to collect and save now. Neural network needs lots of data to make optimal decisions. Second, the computer infrastructure is much better than 20 years ago. The modern computer has more memory, and more powerful CPU to deal with computational problems.

ANN has been widely applied in different fields, including medical diagnosis, image recognition, and natural language processing, etc. More and more algorithms derived from artificial neural networks help those fields to have huge breakthroughs in recent years. Even though there have been many algorithms to deal with the *CRP*, the relocating moves of containers can still be improved. Due to the astonishing performance on *ANN* in recent years, perhaps the method can help people find unprecedented solutions on *CRP*. It gives us the motivation to solve the *CRP* by applying the *ANN* algorithm.

1.3 Problem Framework

Ideally we would like to have a self-learning system that can learn from its own mistakes: whenever our system is beaten by other algorithms in emptying a given bay configuration, the system can learn the procedures of the other algorithms such that eventually our system outperforms these algorithms. AlphaGo learns from lots of Go playbooks. In our study, we imitate the two heuristic algorithms for *CRP* as the start. In fact, we build an *ANN*-based system for this purpose. In the *ANN*-based system, many *ANN*s are trained, each *ANN* specifically for a pre-defined set of *bay characteristics* (defined by the numbers of tiers, columns, and deadlocks of the bay). For any bay configuration, the retrieving and reshuffling sequences of two different heuristic algorithms to empty the bay are broken down into small steps taking care of one container move, and the *ANN*s learn such moves in these small steps that match with the pre-defined bay characteristics of the *ANN*s. At first, the *ANN*s learn from each of the two chosen heuristic algorithms to empty bays. Then, the *ANN*s learn from the *better* solutions of two heuristic algorithms, making the *ANN*-based system surpass the performance of two heuristic algorithms.

To calibrate the performance of the *ANN*-based system, we work on two types of experiments: 4-row, 3-column, and 7-container bay (small bay) and 4-row, 6-column, and 18-container bay (large bay). We want to see how the *ANN*-based system works in small and large bay size. The framework includes the following and Chapter 3 describes more detail on our methods:

I. Creating datasets for training the *ANN*s

1. Learning the logic of two heuristic algorithms on *CRP* and programming them.
2. Generating random bay configurations as problem instances for retrieving and reshuffling.

3. Reshuffling and retrieving random bay configurations by two heuristics and recording each move done by heuristics.
4. Interpreting each container move of two heuristics and incorporate them into the appropriate datasets that match with the bay characteristics of the move.
5. Choosing better reshuffle moves between the two heuristics for a bay configuration to form new datasets, which is called Better-of-Two datasets.

II. Training the *ANN*s from the datasets of two heuristics and Better-of-Two

1. Programming the *ANN*s and learning from three kinds of datasets, respectively.

III. Verifying the training result

1. Generating random bay configurations for verifying the performance of the *ANN*-based system.
2. Reshuffling random bay configurations by the trained *ANN* for the three kinds of datasets, respectively.
3. Comparing reshuffling results with the original results of the heuristics.

1.4 Contribution

It is a novel research for solving the *CRP*. In the thesis, we apply the self learning idea in line of AlphaGo learning from Go playbooks to train our *ANN*s. We create lots of bay configurations and made the *ANN*s learns how the two different heuristic algorithms solve bay configurations. We choose the better solution among two heuristic algorithms on solving different bay configurations and make the *ANN*s also learn from the better solution to surpass the result of the two heuristic algorithms. As the results demonstrated, our *ANN*-based system

perfectly imitates the two heuristics and surpasses them in combined datasets that we generate for small bay size. For large bay size, the *ANN*-based system is unable to imitate nor surpass the two heuristics but the results are very close to them.

1.5 Organization

The thesis is divided into five chapters. Chapter 1 is on background introduction and study motivation. In background introduction, maritime transportation and container terminal are briefly introduced. Some papers related to container terminal are cited. The study motivation explains why *ANN* is applied to solve the *CRP*. Chapter 2 is literature review, where papers related to solving the container relocation problem by heuristic algorithms and papers related to *ANN* are reviewed. Chapter 3 discusses on process to apply the *ANN*-based system in solving the *CRP*. It includes coding the heuristics, datasets, training *ANN*, and verifying the performance of reshuffling containers by *ANN*-based system. Chapter 4 contains the computational results. There are two types of experiment results, one for small bay size and the another for large bay size. The experiments show how well the *ANN*-based system imitates the two heuristic algorithms respectively, and show whether the *ANN*-based system can get better solutions after training both heuristic algorithms at the same time. Adam and Mini-batches are two methods to help speed up the training process of the *ANN*. We observe the differences when we do not apply the proposed methods in this chapter. Chapter 5 mentions the strength and limitations of our study. Then, the conclusion and future work are subsequently discussed in the chapter.

II. Literature Review

Chapter 2 discusses papers related to *CRP* and *ANN*. To further understand *CRP*, we introduce papers on solving *CRP* by different methods. Two heuristics among them are chosen to train the *ANN*s. The concepts of the two heuristics are further introduced in the chapter. In our study, we apply *ANN* to solve the *CRP*. Thus, papers related to improve the efficiency of *ANN* are reviewed in this chapter.

1. Section 2.1 give an overview of different methods on solving the *CRP*.
2. Section 2.2 and 2.3 illustrate the two heuristics that we choose.
3. Section 2.4 briefly introduces *ANN* and reviews papers related to improving the efficiency on *ANN*.
4. Section 2.5 mentions the methods we apply to improve the efficiency of the *ANN* in our study.

2.1 Literature Review of CRP

Container relocation problem are decomposed into two types: dynamic and static problem. In a dynamic problem, new containers continually arrive at the storage block when reshuffling containers. In a static problem, no new container comes in and the bay is emptied. Yang and Kim (2006) utilize dynamic programming and genetic algorithm to solve the static problem and three rule-based heuristics for the dynamic problem. Wan et al. (2009) propose an integer programming (IP) based heuristic to solve both static and dynamic problem. Murty et al. (2005)

propose the Reshuffle Index (RI) to reduce the number of reshuffles in Hong Kong International Terminals. Kim and Hong (2006) utilize both Branch & Bound and the Expected Number of Additional Relocation (ENAR) heuristic to solve the *CRP*, respectively. Lee and Lee (2010) propose a three-phase heuristic which combines integer programming and heuristic to retrieve containers in least number of moves and least retrieving time. Caserta et al. (2011) propose efficient heuristics called the Min-Max heuristic and Corridor method to reduce container reshuffles. Jovanovic and Voß (2014) propose the Chain heuristic by improving the Min-Max heuristic. Petering and Hussein (2013) present two different methods, BRP- III, a mixed integer linear programming, and Look-ahead *N* heuristic improving the Min-Max heuristic.

Table 2.1.1 gives a summary of papers on solving *CRP*.

Table 2.1.1 The integration of papers on solving *CRP*.

Authors	**Objective**	**Methods**	**State of *CRP***	**Results**
Yang and Kim (2006)	To minimize the number of reshuffles	Dynamic programming, Genetic algorithm, and three rules heuristic	Static and dynamic	The heuristic of utilizing the concept of space waste is better than other two heuristics in dynamic *CRP*
Murty et al. (2005)		Reshuffle Index (RI) heuristic	Static	The number of reshuffles is reduced after applying the RI heuristic
Kim and Hong (2006)		Branch and Bound (B&B), Expected Number of Additional Relocation (ENAR) heuristic	Static	The heuristic is close to B&B in the static *CRP*
Wan et al. (2009)		Integer programming (IP) based heuristic	Static and dynamic	IP-based heuristic is comparable to ENAR heuristic and RI heuristic in reshuffles
Lee and Lee (2010)		Three-phase heuristic	Static	The reshuffles is fewer than the ENAR heuristic
Caserta et al. (2011)		Min-Max heuristic and Corridor method (CM)	Static	The number of reshuffles is fewer than the ENAR heuristic
Petering and Hussein (2013)		Mixed integer programming, Look-ahead *N* heuristic	Static	The number of reshuffles of the Look-ahead *N* heuristic is fewer than the Corridor method, the ENAR heuristic, and the Three-phase heuristic
Jovanovic and Voß (2014)		Chain heuristic	Static	Chain heuristic requires fewer number of reshuffles than the RI heuristic and the Min-Max heuristic

2.2 Min-Max Heuristic

Due to its clear concept, quick computational time, and good solution, Min-Max heuristic in Caserta et al. (2011) is chosen as one of the two heuristics to train the *ANN*s.

There are terminologies for the Min-Max heuristic: the container to be retrieved is called the *target container*. In our study, the target container is the lowest number container in the bay. The stack where the target container locates is called *target stack*. When target container is blocked, the crane will reshuffle the blocking containers until the target container can be retrieved. Min-Max heuristic helps to determine the stacks that the blocking containers should be reshuffled to. The idea of Min-Max is to avoid the creation of new deadlock reshuffling in blocking containers. In reshuffling a blocking container, the Min-Max heuristic compares the individual identification of the blocking container with those of containers in other stacks. Putting a blocking container on top of a container with lower retrieval priority will not cause any new deadlock. Thus, Min-Max method detects the highest priority container in each non-target stack. Those highest priority containers are candidates. Each candidate represents different non-target stack. Reshuffling the blocking container to a non-target stack of lower priority than the blocking container does not create a new deadlock. On the other hand, if the reshuffle of the blocking container has to create a new deadlock, the Min-Max heuristic chooses the non-target stack of the highest priority. The closer priorities ensure less reshuffles in the future. Figure 2.2.1 shows results when reshuffling container 2 to different stacks. Because of insufficient space for reshuffling container in previous loading process, it causes another deadlock when reshuffling the blocking container. Choosing the closer retrieval priorities between the candidate and the blocking when facing the deadlock problem is good for reducing the total moves on loading containers to the vessel. Figure 2.2.1 shows results when reshuffling

container 2 to different stacks. When reshuffling container 2, the candidates in column 1 and 2 are container 6 and 3. Min-Max chooses between containers 2 and 3. There is no container in between. However, container 3, 4, 5 are between containers 2 and 6, which means container 3, 4, 5 can be reshuffled to the container 6. Then container 2 can be reshuffled on the container 3, 4, 5 without creating the new deadlock. In Figure 2.2.1, it causes the new deadlock if container 2 is reshuffled on column 1 and container 4 is reshuffled to column 2. The example reflects why Min-Max chooses the closer retrieval priority container.

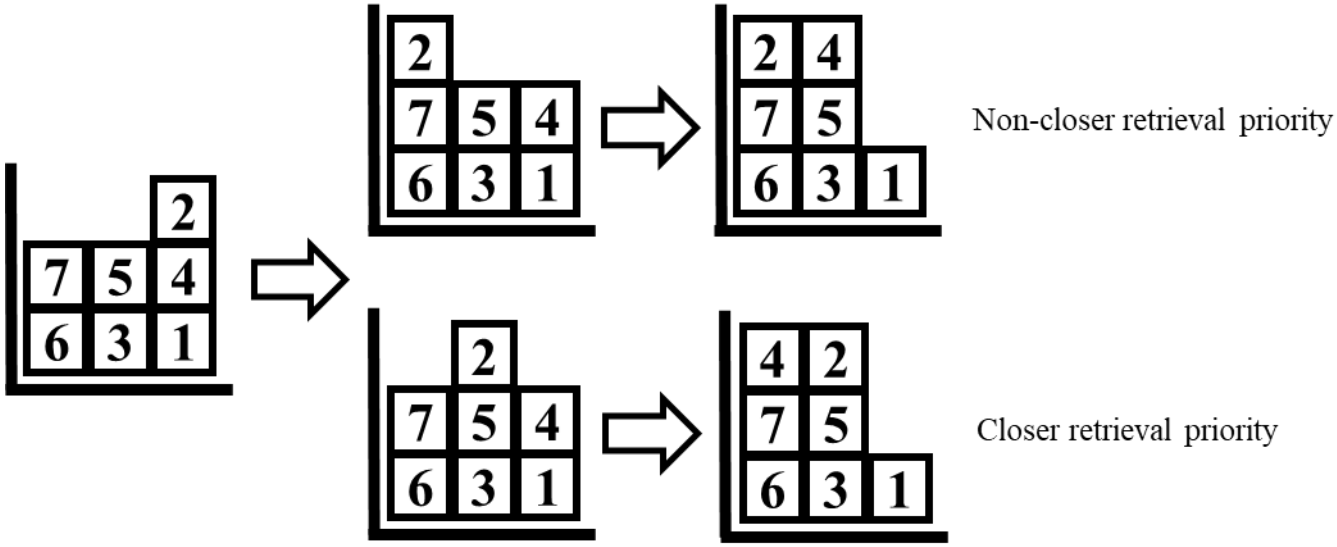


Figure 2.2.1. Comparison of different reshuffles

The following is variable definition in the Min-Max heuristic. Let:

C be the number of containers in the bay;

S be the number of stacks in the bay;

T be the number of tiers in the bay;

i be the retrieval priorities of container, $i = 1$ to NC, smaller number has higher retrieval priority;

i^* be the target container;

s be the number of stack in the bay, $s = 1$ to S;

s^* be the target stack;

$High(s)$ be highest retrieval priority container in each non-target stack;

$Top(s)$ be the top container in stack s;

$Top(s^*)$ be the top container in target stack; and

$Height(s)$ be the height of stack s.

The pseudo code of Min-Max heuristic:

1. If $i^* = Top(s^*)$, the crane directly retrieves container i^*; else continue.
2. If $i^* \neq Top(s^*)$ (i.e., there are blocking containers on the target container), find $High(s)$ in each non-target stack.
3. Let $C = \{s \mid High(s) > Top(s^*)$ *and* $Height(s) < T\}$. The crane reshuffles $Top(s^*)$ to the stack with the highest retrieval priority container in C.
4. If $C = \emptyset$, the crane reshuffles $Top(s^*)$ to the stack with the lowest retrieval priority container in $High(s)$ and $Height(s) < T$. Go back to step 1 until the container bay is empty.

2.3 Look-ahead *N* heuristic

In the Min-Max heuristic, the crane reshuffles containers in the target stack. The Look-ahead *N* heuristic considers to reshuffle containers on the non-target stack. Look-ahead *N* offers more flexibility on reshuffling containers than Min-Max.

"*N*" means *N* lowest retrieval priority containers. As we see the name—Look-ahead *N*—it considers the *N* earliest retrieving order containers. Figure 2.3.1 shows a bay. Containers 1 and 2 are the two lowest retrieval priority containers if $N = 2$. The Look-ahead *N* heuristic considers the blocking containers on the containers 1 and 2. Look-ahead *N* locks stacks 2 and 3 where the *N* earliest retrieving containers locate, the crane will not reshuffle any container to these stacks. It helps to avoid creating any new deadlock on stacks 2 and 3 before retrieving containers 1 and 2.

For Figure 2.3.1, the crane reshuffles one container on the top of stacks 2 and 3—which are containers 3 and 5, respectively, to avoid the new deadlock. In this case, containers of lower priority are reshuffled first. If the crane first reshuffles container 3 to stack 1, it causes deadlock in reshuffling container 5 to stack 1. Thus, Look-ahead *N* chooses the container of lower priority, container 5.

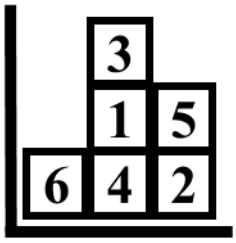


Figure 2.3.1. 6-container bay

The following is the variable definition in the Look-ahead N heuristic. Variables appearing in Min-Max heuristic are not defined again.

Let:

N be N lowest priorities of containers;

$Stack(N)$ be the set of stacks where N earliest containers locate;

$Top(Stack(N))$ be the array of top containers in $Stack(N)$;

r be the order of reshuffling container in $Top(Stack(N))$; and

$Top_r(Stack(N))$ be the container of r-th lowest retrieval priority in $Top(Stack(N))$.

The pseudo codes of Look-ahead N heuristic:

1. Set the value of N, and let $N' = \min\{N, \text{number of present containers in the bay}\}$.
2. If $i^* = Top(s^*)$, the crane directly retrieves container i^* ; else continue.
3. If $i^* \neq Top(s^*)$, create $Stack(N')$.
4. If $Stack(N')$ includes all stacks or stack not in $Stack(N')$ are full, set $N' = N' - 1$ and repeat the step 3; else continue.
5. Create Top($Stack(N')$). The crane has to reshuffle one container in $Top(Stack(N'))$.
6. Let $r = 1$, if reshuffling $Top_r(Stack(N'))$ creates the new deadlock, $r = r + 1$ and repeat the step 6. If $Top_r(Stack(N'))$ is in the s^*, the crane must reshuffle it. If r = the number of container in $Top(Stack(N'))$, reshuffle $Top_r(Stack(N'))$.
7. Let $C = \{s \mid High(s) > Top_r(Stack(N')), s \notin Stack(N')$ and $Height(s) < T\}$. The crane reshuffles $Top(s^*)$ to the stack with the highest retrieving order container in C.
8. If $C = \emptyset$, the crane reshuffles $Top_r(Stack(N'))$ to the stack with the container of lowest retrieval priority in $High(s)$, $s \notin Stack(N')$ and $Height(s) < T$. Then go back to step 1 and repeat the step until the container bay is empty.

2.4 Artificial Neural Network (*ANN*)

Artificial neural network is described to imitate the operations of neurons in human brain to transmit signal to other neurons or receptors through synapses. Suppose a human watches a certain object and observe the features of the object. The neurons transmit what they see as signals to the brain. According to the features of the object, the brain can identify what the object is. If the object has never been seen before, human will learn about it. After learning, the accuracy for a human identifying the object will become higher and higher. Thus, a well-trained *ANN* model can predict unknown data in high accuracy. In fact, it is still uncertain on how human's neurons works. It is just an intuition for understanding *ANN* and gives intuition for human to distinguish the object in *ANN*.

Perceptron is like a neuron. Neuron receives different signals and perceptron receives different values. We assume that neurons are connected. Neuron A transmits signal to neuron B. Neuron B receives signals from many neurons, including neuron A. If those signals are strong enough to activate neuron B, neuron B will transmit the signal to the next neuron. *ANN* imitates the way to activate neurons. It sets threshold to do evaluation. Figure 2.4.1 gives the intuition on how the single perceptron works in an *ANN*. *x* represents features. Lines represent weights w. Weights show the importance of features to identify an object. Dashed line represents bias b, where the bias is added to shift z and give more flexibility to explore the value of z. *w* and *b* are called *parameters*. After computation, we get the value of $z = x_1w_1 + x_2w_2 + x_3w_3 + b$. To evaluate the performance of z value, the *activation function* $g(z)$ is proposed. Thus, $a = g(z)$. The activation function is usually non-linear function. Sigmoid (Cybenko (1989)) and Rectified Linear Unit (ReLU) (Nair and Hinton (2010)) are applied in our study. An activation function not only gives value to represent the performance

of z, but also provides non-linear function to explore more possibility on prediction.

The following are Sigmoid and ReLU functions:

Sigmoid: $a = \frac{1}{1+e^{-z}}$

ReLU : $a = \begin{cases} 0 & for\ z \leq 0 \\ z & for\ z > 0 \end{cases}$

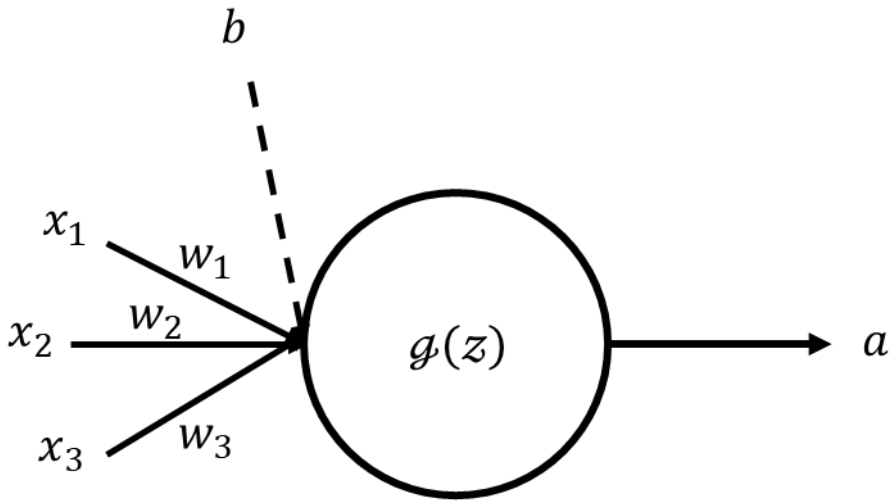


Figure 2.4.1. A perceptron

Figure 2.4.2 shows a basic *ANN*. Perceptrons on the left are grouped as the *input layer*. The perceptrons there represent features. In *ANN*, we need to collect various features for training. When humans learn to identify a tree, height, color and shape are important features for identification. The perceptrons on the right in Figure 2.4.2 are grouped into the *output layer* where the predicted result comes out. For a more complex problem, the larger *ANN* model needs to be applied. Thus, more perceptrons and layers are needed to improve the predicted results. Layers between input layer and output layer are called hidden layers. The connection

to represent parameters for a large *ANN* model are as follow. Let l be an index represneting layers and L is the number of layers in *ANN*. l is from 0 to L -1. The weights = w_{ij}^{l}, and bias = b_i^l represents number of the i^{th} perceptron in layer l, and j represents j^{th} perceptron in the layer $l + 1$. If $l = 3$, i will be number of perceptrons in the layer 3, and j will be in the layer 2. The same computational process in perceptrons continues and the values are passed from the input layer to the output layer. It is called *forward propagation*. The objective of *ANN* is to find suitable value of parameters w_{ij}^{l} and b_i^l to make good prediction. After going through all layers once, we get the predicted output value. We set a threshold to transfer the predicted value. In our study, the predicted value is always between 0 to 1. We talk about how we set the threshold in Chapter 3.

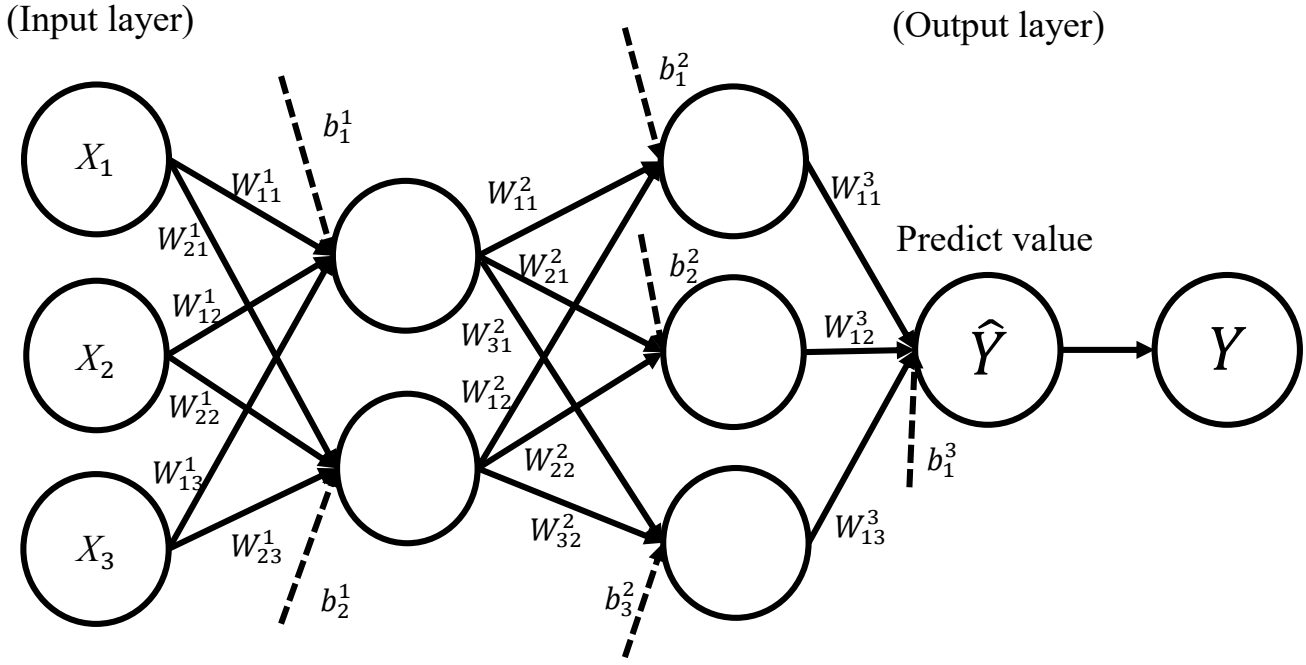


Figure 2.4.2. Basic *ANN* structure

To check whether the difference between predicted value and actual value, the *ANN* computes the error between the predicted value and the actual value. We utilized error functions to compute the error between predicted value and actual value, e.g., *least square error function*,

mean absolute error function, *cross-entropy error function*, *binary cross-entropy function* and so on. We use least square error function and binary cross-entropy function in our study.

The following is the least square error function:

$$\text{Cost} = = - \frac{1}{N}\sum_{i=1}^{N}(\widehat{Y}_i - Y_i)^2 \times \frac{1}{2}$$

N = number of data

The following is the binary cross-entropy function:

$$\text{Cost} = - \frac{1}{N}\sum_{i=1}^{N} Y_i \times \log \widehat{Y}_i + (1 - Y_i) \times \log(1 - \widehat{Y}_i)$$

N = number of data

The initial predicted result is not always good. The predicted accuracy is low and the error is high. Thus, *back propagation* (Rumelhart et al. (1986)) is proposed to adjust the value of weights and bias. To adjust the value of weights and bias, back propagation takes the derivative of error with the respect to each parameter. It is called *partial derivative*. Doing partial derivative helps get the effect of each parameter on the error. The algorithm uses the chain rule in calculus for partial derivative. Let dw_{ij}^l and db_i^l be the value of taking the derivative of error with the respect to weights w_{ij}^l and bias b_i^l. dw_{ij}^l and db_i^l can help to adjust the parameter. In back propagation, the method to adjust the parameters is called *gradient descent*.

Let α be the *learning rate* to adjust the parameters. Weights and bias are adjusted as follow:

$$w_{ij}^l* = w_{ij}^l - \alpha \times dw_{ij}^l,\ b_i^l* = b_{ij}^l - \alpha \times db_i^l,$$

where w_{ij}^l* and b_i^l* represent the updated weights and biases, respectively. If the learning rate is too high, the value of w_{ij}^l* and b_i^l* will be adjusted too much to miss the optimal solution. If the learning rate is too low, w_{ij}^l* and b_i^l* will be time-consuming to reach the optimal

solution. The typical learning rate is [0.1, 0.00001]. The updated parameters will go through the forward propagation and back propagation to update parameters again. To find suitable values for parameters, the process of forward propagation and back propagation need to go through thousands of times.

Many optimization algorithms are proposed to improve the efficiency on *ANN*. Hinton et al. (2012) introduced *mini-batch gradient descent* to deal with the large-scale dataset. In mini-batch gradient descent, dataset is split into many batches and the parameters are updated through the forward propagation and back propagation once batch by batch according to the order of batches. Assume that there are 10,000 data in the dataset. We split the dataset into 10 batches. Each batch contains 1,000 data. The parameters are updated parameters batch by batch. Thus, parameters will be updated 10 times to go through the whole dataset once which is called an *epoch*. The objective of splitting dataset is to train *ANN* with smaller number of data that helps speed up the computational time on training.

Momentum (Sutskever et al. (2013)) helps to improve gradient descent method. Because the traditional gradient descent only updates parameters by the value of dw_{ij}^{l} and db_{i}^{l} in last back propagation. The concept of exponentially weighted average is applied to combine the previous values of dw_{ij}^{l} and db_{i}^{l} together, which provides a more accurate way to update the parameters.

The following is the variable of Momentum. Let:

β be the rate of exponentially weighted average, β = 0.9;

$V_{dw_{ij}^{l}}$ be the exponentially weighted average of dw_{ij}^{l}, and $V_{dw_{ij}^{l}}$ = 0 at the beginning;

$V_{db_{i}^{l}}$ be the exponentially weighted average of db_{i}^{l}, and $V_{db_{i}^{l}}$ = 0 at the beginning;

t be number of back propagation times; and

α be the learning rate.

The following is the function of Momentum:

$V_{dw_{ij}^l} = V_{dw_{ij}^l} \times \beta + dw_{ij}^l \times (1 - \beta)$, (Exponentially weighted average)

$V_{db_i^l} = V_{db_i^l} \times \beta + db_i^l \times (1 - \beta)$, (Exponentially weighted average)

$V_{dw_{ij}^l} = \frac{V_{dw_{ij}^l}}{1-\beta^t}$, $V_{db_i^l} = \frac{V_{db_i^l}}{1-\beta^t}$, (Update parameters)

$w_{ij}^l* = w_{ij}^l - \alpha \times V_{dw_{ij}^l}$, $b_i^l* = b_{ij}^l - \alpha \times V_{db_i^l}$.

We need to do *bias correction* because $V_{db_i^l}$ and $V_{db_i^l} = 0$ in the beginning. The exponentially weighted average is not correct in the first few updates of parameters. With more updates of parameters, β^t will be larger and that the influence of $\frac{1}{1-\beta^t}$ will be smaller. Thus, it is all right not to apply bias correction if *t* is large at the end.

RMSprop (Hinton et al. (2012)) is another approach to speed the updating process of gradient descent method. RMSprop does more investigation than Momentum to update parameters and keeps the parameters from heavily fluctuating to slow down the updating process.

The following is the variable of RMSprop. Variables appearing in Momentum are not defined again:

Let:

$\beta = 0.9$;

ε be epsilon for avoiding 0 in denominator. $\varepsilon = 10^{-8}$;

$S_{dw_{ij}^l}$ be the exponentially weighted average of the square of dw_{ij}^l, $S_{dw_{ij}^l} = 0$ at the beginning; and

$S_{db_i^l}$ be the exponentially weighted average of the square of db_i^l, $S_{db_i^l} = 0$ at the beginning.

The following is the function of RMSprop:

$$S_{dw_{ij}^l} = S_{dw_{ij}^l} \times \beta + dw_{ij}^{l\,2} \times (1 - \beta), \quad \text{(Exponentially weighted average)}$$

$$S_{db_i^l} = S_{db_i^l} \times \beta + db_i^{l\,2} \times (1 - \beta), \quad \text{(Exponentially weighted average)}$$

$$S_{dw_{ij}^l} = \frac{S_{dw_{ij}^l}}{1-\beta^t}, \; S_{db_i^l} = \frac{S_{db_i^l}}{1-\beta^t}, \quad \text{(Bias correction)}$$

$$w_{ij}^l* = w_{ij}^l - \alpha \times \frac{dw_{ij}^l}{\sqrt{S_{dw_{ij}^l}}}, \quad b_i^l* = b_i^l - \alpha \times \frac{db_i^l}{\sqrt{S_{db_i^l}}}. \quad \text{(Update parameters)}$$

When dw_{ij}^l and db_i^l are larger (or smaller), $\sqrt{S_{dw_{ij}^l}}$ and $\sqrt{S_{db_i^l}}$ will be larger (or smaller). Thus, $\sqrt{S_{dw_{ij}^l}}$ and $\sqrt{S_{db_i^l}}$ keep the value of dw_{ij}^l and db_i^l from heavily fluctuating in updating parameters.

Kingma and Ba (2015) propose *Adam* which combines the features of Momentum and RMSprop to update the parameters. Adam is effective for a wide variety of *ANN* architectures. The variables of Adam are defined in Momentum and RMSprop. The setting of β_1 and β_2 are different in Adam.

Let:

$\beta_1 = 0.9$;

$\beta_2 = 0.999$; and

$\varepsilon = 10^{-8}$.

The following is the function of Adam:

$$V_{dw_{ij}^l} = V_{dw_{ij}^l} \times \beta_1 + dw_{ij}^l \times (1 - \beta_1),$$

$$V_{db_i^l} = V_{db_i^l} \times \beta_1 + db_i^l \times (1 - \beta_1),$$

$$S_{dw_{ij}^l} = S_{dw_{ij}^l} \times \beta_2 + dw_{ij}^{l\,2} \times (1 - \beta_2),$$

$$S_{db_i^l} = S_{db_i^l} \times \beta_2 + db_i^{l\,2} \times (1 - \beta_2),$$

$$V_{dw_{ij}^l} = \frac{V_{dw_{ij}^l}}{1-\beta^t}, \; V_{db_i^l} = \frac{V_{db_i^l}}{1-\beta^t},$$

$$S_{dw_{ij}^l} = \frac{S_{dw_{ij}^l}}{1-\beta^t}, \; S_{db_i^l} = \frac{S_{db_i^l}}{1-\beta^t},$$

$$w_{ij}^l* = w_{ij}^l - \alpha \times \frac{V_{dw_{ij}^l}}{\sqrt{S_{dw_{ij}^l}}}, \quad b_i^l* = b_i^l - \alpha \times \frac{V_{db_i^l}}{\sqrt{S_{db_i^l}}}.$$

Weights initialization is important for the convergence of *ANN*, especially for *ANN* with many layers. If the value of weights in each layer is too large (or small), the value of *z* will be larger (or smaller) and larger (or smaller) after computing layer by layer. Thus, the problem in *ANN* with many layers is more serious. It is called *exploding / vanishing gradient*. The problem decreases the efficiency of back propagation to update weights. *Xavier initialization* (Glorot and Bengio, 2010) and *He initialization* (He et al. (2015)) help a lot in weight initialization. Xavier initialization is based on number of perceptrons in a layer. Assume $z = x_i w_i + b$. If there are more perceptrons in a layer, the value of z will be bigger because of more $x_i w_i$. Thus, Xavier initialization decreases the value of w_i when number of perceptrons are larger in a layer. In the *ANN* program, we randomly create weights in the range of [-1, 1] in Gaussian distribution that the weights are with mean 0 and variance 1. Then, let *n* be the number of perceptrons in the layer and variance be $\frac{1}{n}$. The adjusted weights = weights × $\sqrt{\frac{1}{n}}$.

.

In the process of training *ANN*, the dataset is split into three parts: *training set*, *validation set*, and *testing set*. The training set helps train *ANN*. Validation set helps adjust the model on forecasting unknown datasets. Testing set is for verifying the performance of *ANN*. We utilize the training set to train *ANN*. After we finish training *ANN*, we get the accuracy for the training set. Then, we get the another accuracy from the validation set. If the accuracy of training set is very high and accuracy of validation set is relative low, it means that *ANN* are over trained. It is called *overfitting*. Thus, *dropout* (Srivastava et al. (2014)) and *regularization* (Ng, (2004)) are proposed to avoid the overfitting.

2.5 Summary

Many methods to solve the *CRP* are reviewed in Section 2.1. In Sections 2.2 and 2.3, we focus on two heuristic algorithms: Min-Max heuristic and Look-ahead *N* heuristic. In Section 2.4, we briefly introduce artificial neural network and some optimization algorithms to improve the efficiency of *ANN*. In our study, we apply the *ANN*s to imitate how Min-Max heuristic and Look-ahead *N* heuristic solve the *CRP* and observe whether the *ANN*s get better solution than the two heuristics. In the architectures of the *ANN*s, we utilize Xavier initialization to initialize weights. We apply Mini-batch gradient descent and Adam to reduce the training time of the *ANN*s effectively. We give more details about the setting of functions in Chapter 4.

III. Methods

In this chapter, we illustrate the method to solve the CRP. Please refer to Section 1.3 for our solution framework, and see Figure 3.1 for the structure of our method. There are 3 sections in the chapter. Section 3.1 illustrates how we create datasets for training the *ANN*s. We explain the process utilizing two heuristics Min-Max and Look-ahead *N* to create three kinds of datasets. The section includes generating random bay configurations, programming two heuristics, creating datasets according to the logic of heuristics, and choosing better moves among two heuristics to combine new datasets. Section 3.2 shows how we train the *ANN*s. It discusses training the *ANN*s to learn the logic of two heuristics as well as the better of the two heuristics, leading to a total of three kinds of *ANN*s. Section 3.3 discusses how to measure the performance of the training results. We utilize the parameters attained by the *ANN*-based system which includes many *ANN*s to reshuffle containers and compare average moves with those from the heuristics, including the better of Min-Mac and Look-ahead *N*.

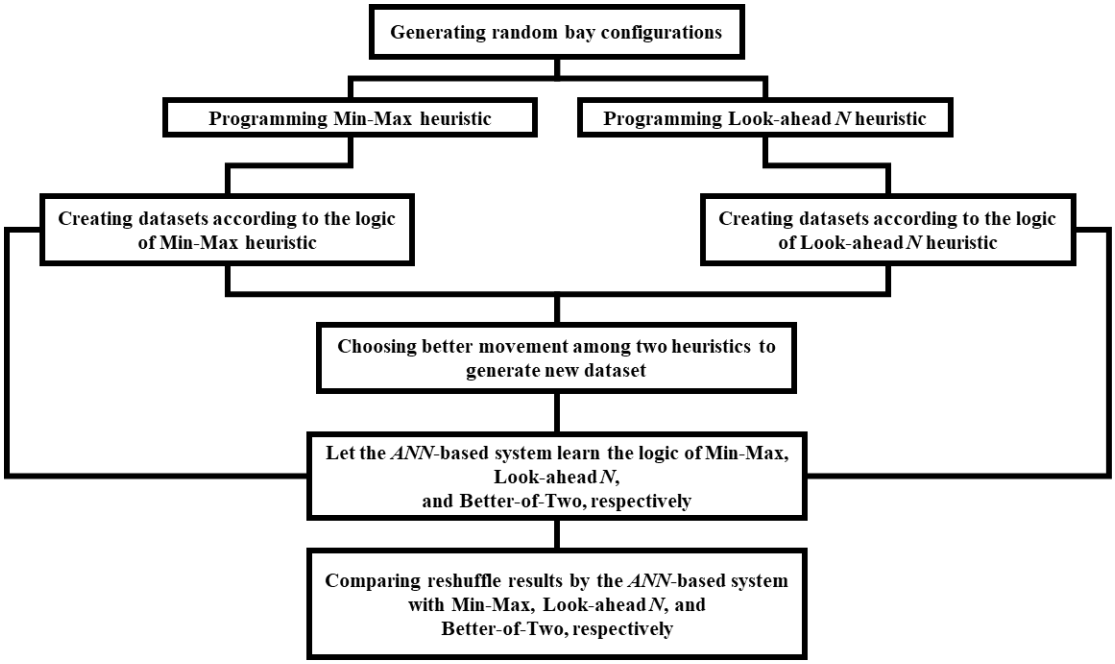


Figure 3.1. Structure of methods

3.1 Creating Datasets

Under any method for the *CRP*, a bay changes shape and gradually reduces in number of containers. We need a data structure to represent the bay configurations as well as the actions to retrieve and reshuffle containers. Section 3.1.1 discusses the representation of bay configurations and Section 3.1.2 of actions. Then Section 3.1.3 discusses the generation of bay configurations for the heuristics and the *ANN*s which are included in the *ANN*-based system to work on, and Section 3.1.4 discusses the compilation of datasets for the *ANN*s, including one set from the better of Min-Max and Look-ahead *N* heuristics. Section 3.1.5 discusses that actually, we have a collection of *ANN*s, not a single one. This is for reducing the training time, and the action actually means sub-dividing the datasets into smaller ones for the *ANN*s. Section 3.1.6 gives the detailed pseudo-codes of various algorithms.

3.1.1 Representation of Bay Configurations

There are $m \times n$ storage slots in an m-tier, n-stack bay, forming a handy matrix structure. The entries of the matrix store the *ID* of the containers stored in the corresponding slot, where the entry 0 represents an empty slot. For ease of programming and computation, we convert the matrix structure into an one-dimensional array, in the order of upper before lower tiers. In our program the array is actually in form of a *column vector* to match the calculation and programming requirement of the *ANN*s. See Figure 3.1.1.A for the representation of a 3-tier, 3-stack, and 7-container bay.

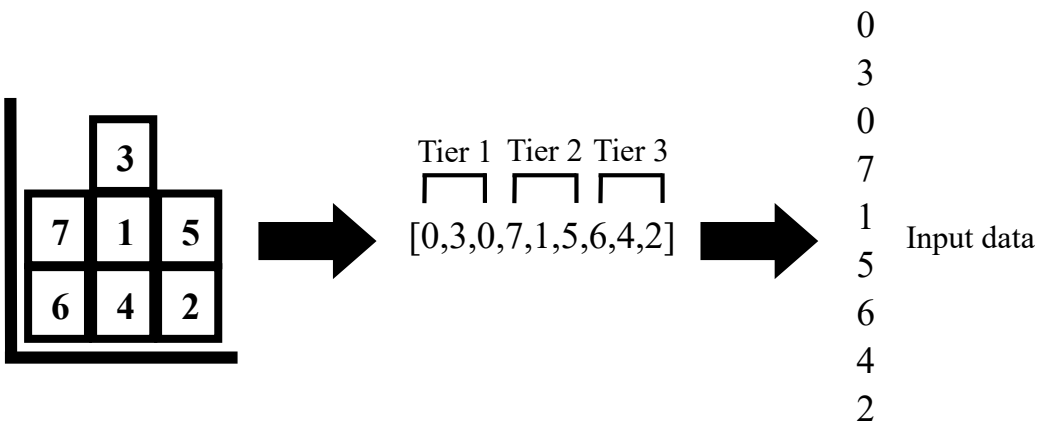


Figure 3.1.1.A. Representation of a 3-tier, 3-stack, and 7-container bay

3.1.2 Representation of Retrieval and Reshuffle Actions

Any method for the *CRP* provides a sequence of retrieval and reshuffle actions. For an $m \times n$ bay, a retrieval action is to take away a container on the top tier of a stack, while a reshuffle action is to move a container on the top tier from one to another stack. Thus, such actions can be represented by a $2n$ array, with the first n elements representing the stacks eligible for taking away a container, and the second n elements for putting down a container onto. A retrieval is

represented by having 1 at the retrieved stack and 0 otherwise, and a reshuffle with 1 at both the retrieved stack and the stack reshuffled to, and 0 otherwise. Again, in our programs, the array is actually in form of a stack vector to match the calculation and programming requirement of *ANN*. See Figure 3.1.2.A for the representation of a reshuffle action for a 3×3 bay.

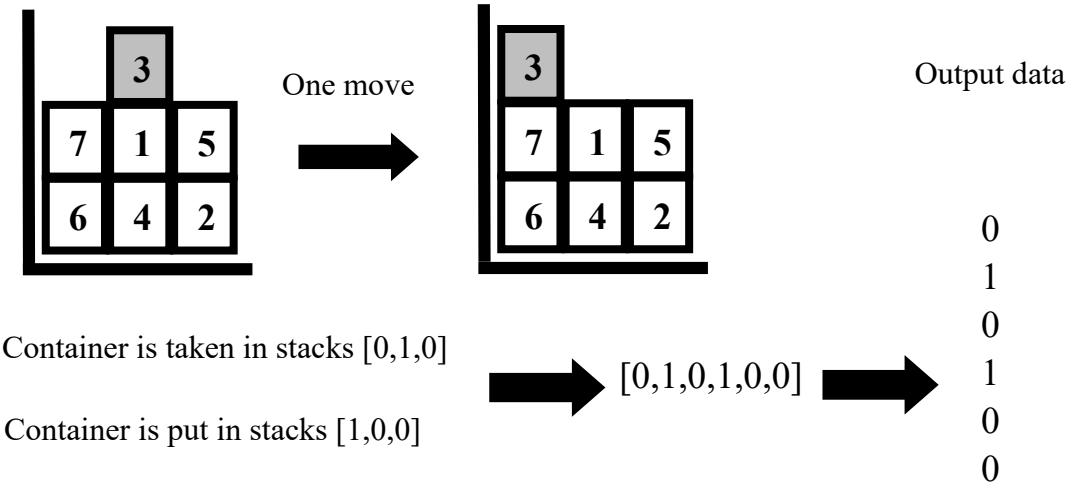


Figure. 3.1.2.A. Representation of a reshuffle of a 3×3 bay

3.1.3 Generation of Bay Configurations

In our computation, we work with two bay sizes: small bay (4-tier, 3-stack, and 7-container) and large bay (4-row, 6-stack, and 18-container).

No matter which method, a bay configuration is made up of two components, the *allocation* of containers to storage slots and the *stacking pattern* showing the height of each stack. The small- and large-size configurations are generated by different methods in these components.

The generation of small-size configurations is exhausting all permutations of allocating containers to all permutations of stack patterns. Figure 3.1.3.A illustrates the idea for a 2×2 bay with 3 containers. There are a total of 6 permutations to allocate 3 containers into 3 slots, and 2 stack patterns for a 2×2 bay with 3 containers, making a total of 12 possible bay configurations in this setting. The small-size configurations are generated in this fashion.

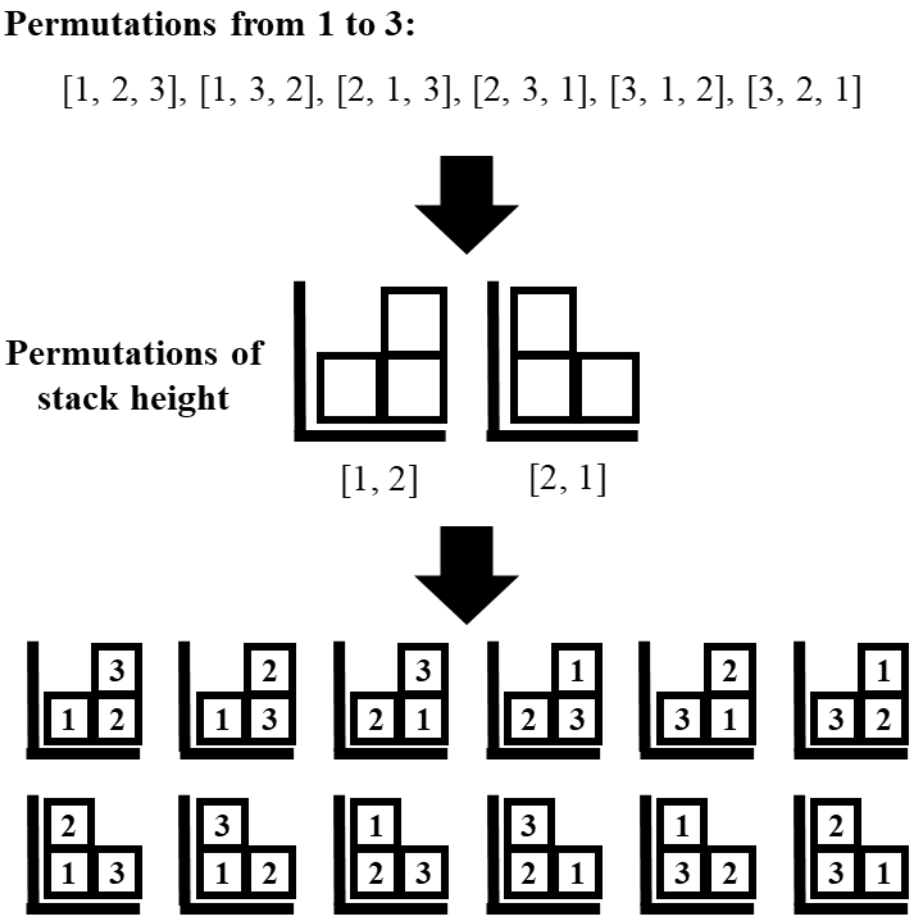


Figure 3.1.3.A. 2-tier, 2-stack, and 3-container bay configurations

For small bay size, the memory and computational speed of computer allow us to work with all possible bay configurations of the given setting; however, for large bay size, the memory requirement is too large and the computation takes too much time so that we can only work with a fraction of all possible configurations which are randomly generated bay configurations for the given setting. Thus, we have two different methods to prepare bay configurations according to the size of the bay.

It is impossible to work with all configurations for the large-size bay. In the given setting, we control the total number of configurations to generate; in each configuration, we randomly generate one allocation scheme for 18 containers, and put them into one randomly generated stack pattern. Section 3.1.6 shows the pseudo codes on the procedure of randomly generating configurations.

3.1.4 Compilation of Datasets for the *ANN*s

The bay configurations generated according to the procedure in Section 3.1.3 are represented by the convention in Section 3.1.1. The two heuristics, Min-Max and Look-ahead *N*, are then applied to the bay configurations to eventually empty them. In each iteration, the action is represented by the convention in Section 3.1.2, and the resulted bay configuration by the convention in Section 3.1.1. For each heuristic, each iteration leads to an *input* (i.e., the bay configuration) and an *output* (i.e., the action) of the iteration. The input-output pair forms a *data instance* for the *ANN*s. The collection of all data instances from a heuristic forms the *dataset* of the heuristic. Training the *ANN*s by this dataset will give the *ANN*s following the logic of the heuristic.

Note that we can easily get a better heuristic by choosing *better* of Min-Max and Look-ahead *N*, where "better" is defined by the sequence of actions and bay configurations that takes less number of moves to empty the original bay configuration. Thus, for each bay size, we actually have three datasets for the *ANN*s to learn from, one dataset from each heuristic, and one from the Better-of-Two heuristic. See the following subsection that actually the datasets are further divided into smaller ones according to bay size, number of deadlocks, and number of containers of the bay.

3.1.5 Collection of *ANN*s

To reduce computation time and to reduce complexity of the *ANN*s, we actually work with a collection of *ANN*s, instead of one *ANN* to take care of everything. Each *ANN* works specially for a given heuristic, a given bay size, with a given number of containers in the bay, and a given number of deadlocks in bay configuration for the current action. Figure 3.1.5.A shows the different data instances for a 3×2 bay of 3 containers. The three cases show that there can be 2 deadlocks, 1 deadlock, and no deadlock. The containers in gray color are deadlocks. The data instances of these three cases are separated into three different datasets, for two *ANNs* for two and one deadlock, respectively. Because for zero deadlock it is intuitive to retrieve a container without training any *ANN* model, we do not create any *ANN* to handle zero deadlock.

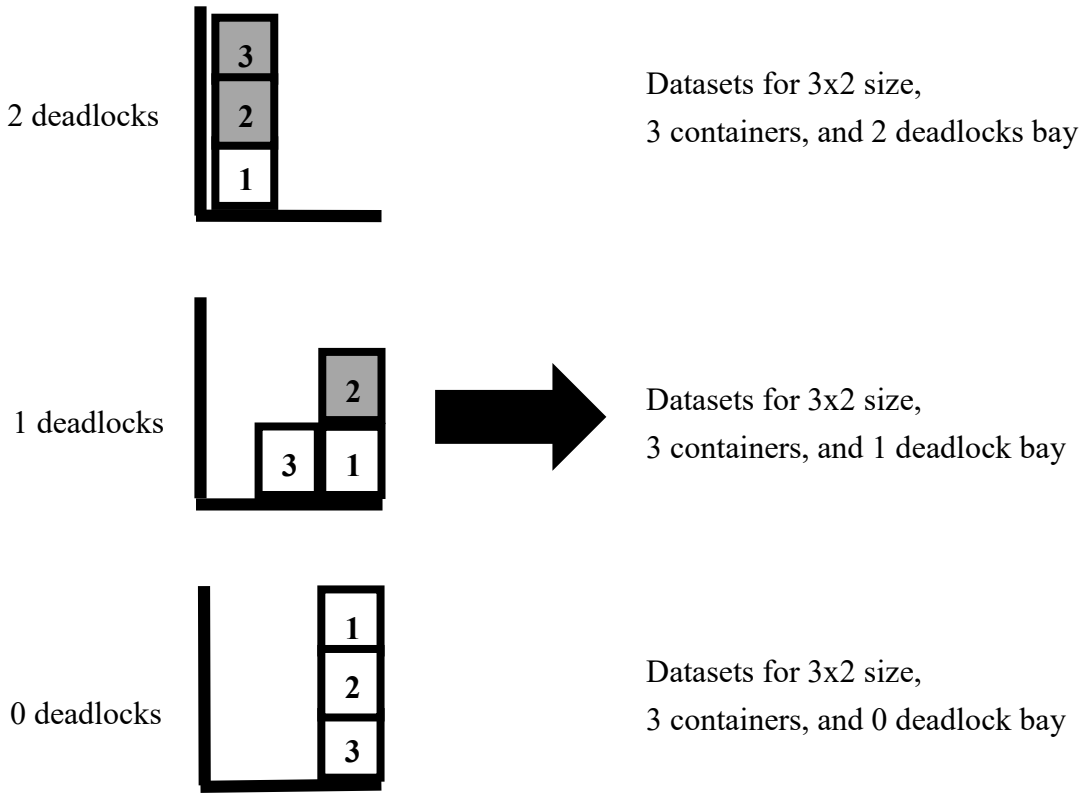


Figure 3.1.5.A. Data instances in three different datasets

3.1.6 Pseudo-codes of Algorithms

This section documents the pseudo-codes for the algorithms for the tasks in early sections. The pseudo-codes are:

- pseudo-code A: Create Stacking Patterns and Generate Permutations of [1,…,*NC*]
- pseudo-code B: Create All Combination of Bay Configurations
- pseudo-code C: Categorize the Bay Configurations and Turn Them into Input Data
- pseudo-code D: Randomly Generate Bay Configurations
- pseudo-code E: Generate Output Data for Heuristics

Pseudo-codes to Create Stacking Patterns and Generate Permutations of [1, …, *NC*]

The inputs of this set of codes are *T* (= the number of tiers of the bay), *S* (= the number of stacks of the bay), and *NC* (= the number of containers in the bay). The outputs are the stacking patterns and the permutations of [1, …, *NC*].

Pseudo-code A: Create Stacking Patterns and Generate Permutations of [1, …, *NC*]

```
1  function create stacking pattern (T, S, NC)
2          set an empty list
3          create Cartesian product for each {0 to T} in S spaces
4          for each combination in Cartesian product
5              if sums of the combination = NC
6                  add the combination into the list
7          return list
8  end function
9  Stacking patterns = create stacking pattern (T, S, NC)
10 Generate permutation of list which contains 1 to NC (permutation_1_to_NC)
```

Line 1 is the name of the function. Line 2 creates a list to save each stacking pattern. Line 3 creates the cartesian product of *T* and *S*. For *T* = 4 and *S* = 3, their Cartesian product will be the collection of triplets [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 1, 0),…, (4, 4, 1), (4, 4, 2), (4, 4, 3), (4, 4, 4)]. Lines 4, 5, 6 form a **for** loop to calculate the sum of each triplet. Triplets with their element sum = *NC* stand for the heights of stack patterns with *NC* containers in the bay. We add such triplets into the list. Line 7 returns the stack patterns. Line 8 ends the function and Line 9 executes the function to create the data structures for the stack patterns. Line 10 creates all permutations for [1, …, *NC*]; e.g., for *NC* = 3, the results are [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

Pseudo-codes to Generate All Combinations of Bay Configurations

This set of codes generates the bay configurations for small-size bays as documented in Section 3.1.3. The inputs are stacking patterns and permutations of [1,…,*NC*] ; the outputs are all combinations of bay configurations in the given bay size and containers. The idea is illustrated in Figure 3.1.6.A.

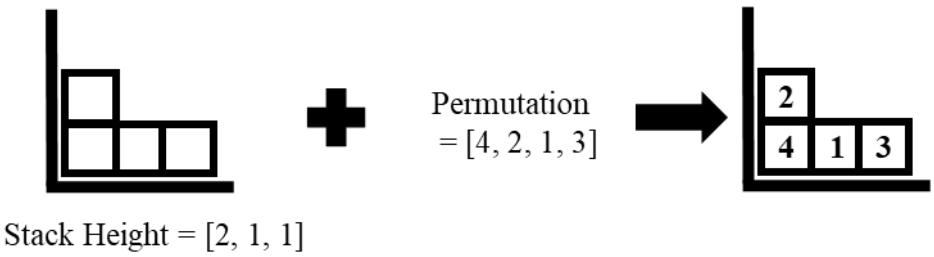


Figure 3.1.6.A. Process of generating a bay configuration

Pseudo-code B: Generate All Combinations of Bay Configurations

```
1 function create bay configuration (stacking patterns, permutation_1_to_NC)
2     for each stacking pattern in stacking patterns
3         order = 0
4         for each permutation in permutation_1_to_NC
5             create an empty bay
6             for each stack height in stacking pattern
7                 tier position = lowest position in the bay
8                 while stack height > 0
9                     bay [tier, stack] = permutation[order]
10                    stack height = stack height – 1
11                    order = order + 1
12                    tier position = tier position – 1
13            record the bay configuration
14 end function
15 All bay configuration = create bay configuration (Height combination, permutation_1_to_NC)
```

The set of codes creates all bay configurations by allocating all permutations of 1 to *NC* to all possible stacking patterns. The first **for** loop in Line 2 goes through each stacking pattern. To create bay configurations, for each stacking pattern, the second **for** loop in Line 4 first creates an empty bay and the third **for** loop in Line 6 allocates each permutation of [1, …, *NC*], i.e., the container *ID*s (numbers), into the storage slots of the bay according to the height of each stack. Figure 3.1.6.A shows to how form a bay configuration. We put containers in the bay from lower to upper tiers, and from left to right stacks. Thus, the sequence of putting containers in Figure 3.1.6.A is 4–2–1–3. The priority to put container in the bay is from bottom to top and left to right. The **while** loop in Line 8 puts containers in the same stack as long as the height limit of the stack is observed.

Pseudo-codes to Categorize the Bay Configurations and Turn them into Input Data

This set of codes reads in instances from a full dataset and categories them into datasets for the collections of *ANN*s described in Section 3.1.5. The inputs are generated bay configurations in Pseudo-code B and the outputs are input data from the categorized bay configurations.

Pseudo-code C: Categorize the Bay Configurations and Turn them into Input Data

```
1   function extract destined bay configuration
2       for each bay configuration in all bay configurations
3           find the position of target container, container 1
4           find number of deadlocks
5           if number of deadlocks is the given number
6               record the bay configuration
7   end function
8   Destined bay configurations = create bay configuration (All bay configurations)
9   Shape the size of destined bay configurations to match input layer size
10  Save the created input data as a file
```

This program works for instances with a given number of containers. The main task is to find the number of deadlocks in the bay configuration. The **for** loop in Line 2 checks each bay configuration. Lines 3 to 6 show how to record bay configurations with a pre-specified number of deadlocks.

Pseudo-codes to Randomly Generate Large-size Bay Configurations

As discussed in the last paragraph of Section 3.1.3, for large-size bays, we only randomly generate a fraction of the all possible bay configurations for the heuristics and *ANN*s to work on. This set of codes precisely serves this purpose. The outputs are input data of random bay configurations in given size, containers, and deadlocks.

Pseudo-code D: Randomly Generate Large-size Bay Configurations

```
1  Create the empty list for adding container bay
2  Set number of data = 0
3  while number of data
4        Create a list that contains 1 to NC and random the sequence
5        Create an empty bay
6        Randomly put the number in the list of Line 4 into the bay
7        When the bay configuration is created, it is reverse. Thus, we turn it upside down
8        if deadlock = given number
9              record the bay configuration
10             number of data = number of data + 1
11 Turn the list into numpy array
12 Shape the numpy array size to match the input layer size
13 Save the created input data as a file
```

Line 1 creates an empty list for saving random bay configurations. The **while** loop in Line 3 creates random bay configurations according to the number of bay configurations required. Line 4 in the while loop generates a list that contains a random permutation of [1,…, *NC*]. Line 5 creates an empty bay and Line 6 randomly saves container *ID*s into stacks of the bay. Line 7 is simply an intermediate to align the data structure with the computational convention. Lines 8 and 9 keep bay configurations of the pre-specified number of deadlocks. Line 10 counts the number of bay configurations collected so far, and the **while** loop ends if sufficient number of bay configurations is got. Lines 11 and 12 shape the generated bay configurations into desirable input layer size to suit programming.

Pseudo-codes to Generate Output Data for Heuristics

This set of codes generates the actions for the heuristics. See Figure 3.1.2.A for the vectors documenting the actions. The inputs are input data; and the outputs are output data.

Pseudo-code E: Generate Output Data from Heuristics

```
1  Open the file of input layer
2  Shape the input layer to the size of bay configuration
3  Program heuristic (Min-Max, Look-ahead N) and set heuristic to only do a reshuffle
4  Create an empty list
5  for each bay configuration in all bay configurations
6      Record the stack height A of the bay configuration
7      Apply Min-Max (Look-ahead N) heuristic to do a reshuffle on the bay configuration
8      Record the stack height B of the bay configuration after a reshuffle
9      Check the difference between height A and height B
10     Utilizing the difference to create output data
11     Add output data to the list
12 turn the list into numpy array
13 Shape the size of array to suits output layer
14 Save the output layer to the file
```

Line 1 opens the file of input data and Line 2 turns it into the bay configuration. Line 3 calls a heuristic to generate the action of the read in bay configuration. Line 4 creates the data structure to save the action vectors. The **for** loop Lines 5 to 9 observes the difference of stack height after reshuffling a container. It helps us record stacks that a container is taken from or put into. We have mentioned the concept of two heuristics in Chapter 2 and program in Line 7. When reshuffling containers, we set the program to make sure the container reshuffle to right stack. The stack pattern is recorded when reshuffling containers to avoid surpassing height limit. Thus, there is no need to use mathematical constraints to set number of containers, height, stacks. We program to simulate container bay and how heuristics reshuffle containers.

Then, Line 10 records the action vectors like those Figure 3.1.2.A in the listed created in Line 4. Lines 12 to 14 turn the list into numpy data structure, shape it to satisfy the need of output layer size, and save it in the file of output data.

3.2 Training Artificial Neural Network

We follow the pseudo codes in Section 3.1 to generate 15 datasets for small-size and 47 datasets for large-size bays. All the possible bay configurations are generated for small-size bays. For large bay size, we are unable to generate all bay configurations. Thus, we randomly generate bay configurations. The detailed number of bay configurations is shown in Table A.4, Table A.5, and Table A.6. In this section, we discuss pseudo-codes for programs for training the *ANN*s by these datasets.

The input-output pairs of the datasets are used to train the *ANN* models. The objective is to optimize the *parameters*, which are *weights* and *bias* of *ANN*. The effect of training depends on the values of the *hyper-parameters*, e.g., the number of hidden layers, the number of perceptrons in hidden layers, the value of learning rate, the number of epochs, and the choice of activations, etc. The value of hyper-parameters can be set before training. We separate the pseudo code into 3 parts. There are many functions in *ANN*. Thus, we introduce each function of *ANN* in the first part. The second part combines these functions in the first part to form an *ANN* model. The last part is to execute *ANN* model.

Pseudo-code: Training Artificial Neural Network – Part 1.1

```
Open the file of the input and output data
Random the sequence in input data and out data
Split the dataset into training set, validation set, and testing set
Decide number of hidden layer
Decide number of perceptrons in each hidden layer
function Initialize the parameters (input data, perceptrons in each layer, output data)
    Creating a dictionary for storing parameters
    for each layer in [1, number of layer – 1]
        Generate and store parameters
    return parameters
end function
function Sigmoid function (value)
function ReLU function (value)
function Mini-batch (input data, output data, seed, mini-batch size)
    Calculate number of mini-batches with complete size
    Change the order of input and output data
    for i in [0, number of mini-batches with complete size – 1]
        Create a mini-batch and store it in list
    if there is a mini-batch with incomplete size
        Create the mini-batch with remain data and store it in list
    return Mini-batches
end function
function Forward propagation (input data, parameters, output data)
    Caches = {}
    x = a_0
    if there is no hidden layer
        Apply the function z_1 = w_1 a_0 + b_1
        Apply Sigmoid function, a_1 = g(z_1), and store a_1 in Caches
    else
        for i in [1 to number of layer - 2]
            Apply the function z_i = w_i a_{i-1} + b_i
            Apply ReLU function, a_i = g(z_i), and store a_i in Caches
        Apply the function z_{i+1} = w_{i+1} a_i + b_{i+1}
        Apply Sigmoid function, a_{i+1} = g(z_{i+1}), and store a_{i+1} in Caches
    A = a_{i+1}
    return A, Caches
 end function
```

Line 1 loads the datasets. To increase the accuracy of prediction, Line 2 randomizes the sequence data instances in the datasets. We split the dataset into training set and validation set in Line 3 because we verify the training results by reshuffling containers by trained parameters rather than computing the accuracy of the testing set. Lines 4 and 5 set the number of hidden layers and perceptrons in each hidden layer. Line 6 initializes the parameters. We input the perceptrons in each layer to know the size of parameters and Xavier initialization (Glorot and Bengio (2010)) is applied to set parameters. Then, we store the parameters into *Dictionary* which is a data structure in Python. We put Lines 12 and 13 as activation functions: Sigmoid and ReLU functions. Section 2.4 mentions the two activation functions. Lines 14 to 22 split training set into mini-batches. Line 16 changes the order of input and output data. The order of the data instances is changed in the dataset; e.g., a sequence of [1, 2, 3] may be turned into [3, 1, 2]. We use the *seed* in Python to change the order of the dataset. For a given number of data instances and the batch size, the last mini-batch will not be of full size; see, e.g., dividing 50,000 data instances into mini-batches of 150 data instances. Lines 17 to 18 are for creating full-size mini-batches, and Lines 19 to 20 are for the last mini-batch that may not be of full size. Lines 23 to 37 are for *forward propagation*. Line 24 creates a dictionary called Caches for storing calculated value of a. Line 25 turns input data into a_0 that is easily applied in forward propagation. Line 26 to 34 set a condition to check whether there is any hidden layer. If there is no hidden layer, the program applies Sigmoid function to make a between [0, 1] to do prediction; else the program applies ReLU function until calculating a in the last layer by Sigmoid function to do prediction.

```
function Cost function (A, output data)
    Cost in every data = output data * log (A) + (1 - output data) * log (1 - A)
    Cost = average of cost in every data
    return cost
end function
function Back propagation (parameters, caches, input data, output data, A)
    grads = {}
    dA = Do partial deviation on A
    dZ = Do partial deviation on z
    Do partial deviation on w in last layer and store them in grads
    Do partial deviation on b in last layer and store them in grads
    if number of layers > 2
        for i in second last layer to first layer
            Do partial deviation on w in i-th layer and store them in grads
            Do partial deviation on b in i-th layer and store them in grads
    return grads
end function
function Hyperparameter for Adam optimizer (parameters)
    v = {}
    s = {}
    for i in [1, number of layers – 1]
        Copy the shape of parameters in each layer and store them in v, s, respectively
end function
function update parameters (parameters, grads, v, s, t, beta 1, beta 2, epsilon, learning rate)
    v_correction = {}
    s_correction = {}
    for i in 1 to number of layers
        v[dwi] = beta 1 × v[dwi] + (1 – beta 1) × grads[dwi]
        v[dbi] = beta 1 × v[dbi] + (1 – beta 1) × grads[dbi]
        s[dwi] = beta 2 × s[dwi] + (1 – beta 2) × grads[dwi]^2
        s[dbi] = beta 2 × s[dbi] + (1 – beta 2) × grads[dbi]^2
        v_correction[dwi] = v[dwi] ÷ (1-beta 1^t)
        v_correction[dbi] = v[dbi] ÷ (1-beta 1^t)
        s_correction[dwi] = s[dwi] ÷ (1-beta 2^t)
        s_correction[dbi] = s[dbi] ÷ (1-beta 2^t)
        parameters[wi] = parameters[wi] - v_correction[dwi] ÷ (s_correction[dwi] + epsilon)^1/2 ×
        learning rate
        parameters[bi] = parameters[bi] - v_correction[dbi] ÷ (s_correction[dbi] + epsilon)^1/2 ×
        learning rate
    return parameters
end function
```

Lines 38 to 42 follow the cross-entropy function in Section 2.4 to program. Lines 43 to 54 are for *back propagation* of *ANN*. Line 44 creates the Python data structure dictionary for saving values of partial derivatives on parameters. The process of back propagation takes place from the last layer to the first layer. The partial derivatives on z in the last layer are different from those of the other layers because the Sigmoid function is used only in last layer. As the derivative of the ReLU function is different from that of the Sigmoid function, Line 49 sets a condition for the hidden layers to do derivative on the ReLU function. At the end of back propagation, all the value of the partial derivatives of the parameters are saved. Lines 55 to 60 set the hyper parameters for Adam optimizer in the value of 0. Lines 61 to 76 follow the Adam in Section 2.4 to program.

Pseudo-code: Training Artificial Neural Network – Part 1.3

```
77 function predict (parameters, input data)
78     a, Caches = forward propagation (input data, parameters, output data)
79     for j in 0 to number of data – 1
80         Max take = 0
81         Max put = 0
82         for take in 0 to 5
83             if a[take][j] >= Max take
84             take row = take
85         for put in 6 to 11
86             if a[put][j] = Max put
87             put row = put
88         for i in 0 to 11
89             a[i][j] = 0
90         a[take row][j] = 1
91         a[put row][j] = 1
92     return a
93 end function
```

Lines 77 to 93 are for prediction. The array a defined in Line 78 records the output prediction. The values of the elements of the array give the possibility of taking an action, taking away or putting down, at the stacks. The larger the value is, the more likely for an action to be taken place at the stack. As discussed in Section 3.1.2, for an n-stack bay, the output is a vector of $2n$ elements, with the first half representing the stack that a container is taken away, and the second half representing the stack that a container is put onto. The largest number in the first and the second half are taken as 1; and others as 0 for each output.

Pseudo-code: Training Artificial Neural Network – Part 2

```
94 function nn model (input data, output data, number of epochs, beta 1, beta 2, epsilon, learning rate)
95        seed = 10
96        parameters = Initialize parameters (input data, all of layers, output data)
97        v, s = hyper parameter (parameters)
98        t = 0
99        Create a list for putting cost
100       for i in 0 to number of epochs
101             seed = seed + 1
102             mini-batches = mini-batch (input data, output data, seed, mini-batch size = = 1024)
103             for mini-batch in mini-batches
104                   mini-batch of input data, mini-batch of output data = mini-batch
105                   a, Caches = forward propagation (mini-batch of input data, all of layers, parameters, mini-batch
                                    of output data)
106                   cost = cost function (a, output data of mini-batch)
107                   grads = back propagation (parameters, Caches, input data of mini-batch, output data of mini-
                              batch, a, all of layers)
108             if remainder of i ÷ 100 = 0
109                   print "Cost after epoch =" i
110                   Add cost in list
111       Plot the line chart with every 100 epoch
112       return parameters
113 end function
```

Lines 94 to 113 can be regarded as the main function that calls the previous functions to execute training. Line 95 defines the value of the seed for the random functions, which are used, e.g., in randomizing data instances into various mini-batches in each epoch. Lines 96, 97, and 98 initialize parameters and set the values for hyper parameters for the Adam optimizer. The **for** loop starting in Line 100 trains the parameters and records the changes for the cost function in each epoch. At the end of the destined number of epochs, the function gives the updated values of parameters after training.

Pseudo-code: Training Artificial Neural Network – Part 3

```
114 parameters = ANN model (input data, output data, number of epochs)
115 predict validation set = predict (parameters, all of layers, input data of validation set)
116 Calculate the difference between predict validation set and output data of validation set
117 Utilize the difference between predict validation set and output data to calculate accuracy on validation set
118 Repeat the process to calculate accuracy on training set
```

Line 114 executes training and the accuracy are got from Lines 115 to 118.

3.3 Generating Better of Two Datasets

One of the purposes in the study is to check whether the *ANN*-based system can learn from different heuristics to make better decisions. Here we use the total number of moves to empty a given bay configuration as the yardstick to measure how good a heuristic (or a method) is, i.e., the less number of moves, the better a heuristic is.

Section 3.1 discusses how to general random bay configurations and applies the Min-Max and Look-ahead *N* heuristics to empty the bay configurations. Section 3.1.4 further gives the idea of forming the Better-of-Two heuristic by collecting the data instances of the better

heuristic for any random bay configurations. The first half of this section discusses how to choose better data instances from two heuristics to form the dataset for the Better-of-Two heuristic. The second half of this section discusses the actual training and result recording for the datasets.

Pseudo-codes of Forming the Dataset for the Better-of-Two Heuristic

The following algorithm is only used for 8 or more containers, because there is no difference between the Min-Max and the Look-ahead *N* heuristics for 7 or less containers. The dataset includes input-output pairs. We only load the output data of the dataset to form the new output part of the dataset in the program. In the given number of container in the bay, three categories of output data are generated: 3 deadlocks, 2 deadlocks, and a deadlock. We put these 3 categories of output data per heuristic in the same program to form a new output data of Better-of-Two Heuristic, respectively. The inputs are the 3 categories of output data of the Min-Max and 3 categories of output data of the Look-ahead *N*. The output are 3 categories output data that combine the better of the two heuristics.

Pseudo-code: Forming the Dataset for the Better-of-Two Heuristic

```
1.Open the output dataset of two heuristics
2 MM-output = put 3 categories of output data of Min-Max heuristic together
3 LA-output = put 3 categories of output data of Look-ahead N together
4 Look-ahead N = open the total move of each bay configuration emptied by Look-ahead N
5 Min-Max = open the total move of each bay configuration emptied by Min-Max
6 Compare = Min-Max – Look-ahead N
7 MM_better_LA = []
8 LA_better_MM = []
9 for i in (0 to number of data – 1)
10      if compare [i] > 0
11              add i in LA_better_MM
12      if compare [i] < 0
13              add i in MM_better_LA

14 for i in LA_better_MM
        MM-output [i] = LA-output [i]
15 Create the combined dataset by splitting Min-Max and store them as the file
```

We take an example to explain the program of forming the dataset for the Better-of-Two heuristic. Assume we want to form a new dataset of 4-tier, 6-stack, and 8-container bay in the program. In the given bay, there are 3 categories of datasets to be generated: 3 deadlocks, 2 deadlocks, and 1 deadlock. We use *tier_stack_container_deadlock* to represent a dataset. 4_6_8_3 represents the dataset of 4-iter, 6-stack, 8-container, and 3 deadlocks bay. There are 3 datasets in each heuristic: 4_6_8_1, 4_6_8_2, and 4_6_8_3, making a total of 6 datasets. We take the input data of 6 datasets. We turn them into bay configurations, empty them, and record the total moves. We put these moves together according to the heuristic. Thus, Line 4 contains the total moves of each bay configuration, from the input data of 4_6_8_1, 4_6_8_2, and 4_6_8_3, emptied by the Look-ahead *N* heuristic. Line 5 contains total moves emptied by the

Min-Max heuristic. Line 1 loads the output data of 4_6_8_1, 4_6_8_2, and 4_6_8_3 in each heuristic. Lines 2 and 3 put these output data together according to the heuristic. We record the total moves by the order of 4_6_8_1, 4_6_8_2, and 4_6_8_3. Line 6 computes the difference of total moves among two heuristics. The **for** loop from Line 9 to Line 13 sorts the data instances into two groups, of Look-ahead *N* heuristic better (if *i*-th value > 0), and of Min-Max heuristic better (if *i*-th value < 0) We only use the list LA_better_MM. Line 14 uses the output data in Line 2 and 3 to form a new output data. If the total moves of Look-ahead *N* heuristic is better than Min-Max heuristic in *i*-th bay configuration, the *i*-th in MM-output will be replaced with *i*-th in LA-output which the output data reshuffled by Min-Max heuristic will be replaced by Look-ahead *N* in *i*-th bay configurations. If the total moves are equal, we do not replace. Because the performance of the *ANN*-based system imitates Min-Max heuristic is better than Look-ahead *N*. Thus, we decide to form datasets in Min-Max based and improved by Look-ahead *N*. After replacement, Line 15 splits them into three categories of output data: 4_6_8_1, 4_6_8_2, and 4_6_8_3 and saves these output data. We follow the same method to form the dataset for the Better-of-Two Heuristic in other programs with different given containers.

Pseudo-codes to Calibrate the Performance of the Heuristics

The generated datasets, from Min-Max, Look-ahead *N*, and Better-of-Two, are used to train the *ANN*-based system. At the end of the training and validation, there are three collections of *ANN*s, one collection for each of the three heuristics. These three collections of trained *ANN*s are used to empty 1,000,000 randomly generated 4-tier, 6-column, and 18-container bay configurations to verify the performance of the *ANN*s. We empty container when the target container is on the top. Thus, the **while loop** related to present containers needs to be adjusted. Three performance measures are used to calibrate the *ANN*s, i.e., in emptying a bay configuration, the total number of moves, the computational time, and the number of errors are noted for each heuristic. An error occurs if the suggested action by an *ANN* is infeasible. In that case we apply one of two heuristics to do reshuffle and the counter of errors increases by one. The choice of heuristic depends on the total moves of the bay configurations. We empty the 1,000,000 bay configurations by two heuristics, respectively and record which heuristic is better in the *i*-th bay configuration. When the program faces an error, it will check which heuristic performs well in the *i*-th bay configuration. If the total moves are equal, we choose Look-ahead *N* to fix the error.

The following are the pseudo codes on emptying bay configurations by the trained *ANN*s:

Pseudo-code: Calibrate the Performance of the Heuristics

```
1 Open the file of bay configurations
2 Open the file of Min-Max is better than Look-ahead N on reshuffling bay configurations
3 Turn the bay configurations into input data
4 Bay_move = []
5 error = 0
6 for i in (0 to number of bay configurations -1)
7       move = 0
8       while remain container > 2
9           if target container on the top
10                  retrieve target container
11          else
12                  load parameters according to features (number of container and deadlock)
13                  do forward propagation
14                  do prediction according to the output of forward propagation
15                  get position of taking and putting container

16                  if column for putting container is not full;
                        column for putting container does not contain target container;
                        column for taking container is not empty;
                        same container is not taken in a row;
                        column for taking and putting is not the same
17                          reshuffle container by parameters
18                  else
19                          utilize heuristics to reshuffle container
20                          error = error + 1
21          move = move + 1
22      empty the bay
23      move = move + move for reshuffling and retrieving last two containers
24 add move into Bay_move
25 print error
26 print average move..
27 print computational time
```

Line 1 loads the file of generated bay configurations to be emptied by the heuristics. For the large bay size, there are 1,000,000 bay configurations. For the small bay size, there are 90,720 bay configurations. Line 2 opens and works on the dataset for instances with Min-Max better than Look-ahead *N*. Line 6 marks the beginning of a **for** loop to reshuffle and retrieve each bay configuration. Line 8 marks the beginning of a **while** loop that keeps on taking action of the bay, with less containers as they are taken away. When there are only 2 containers left, Line 23 empties these last two containers. Within the **while** loop, Lines 9 and 10 are conditions to retrieve the *target container* when it is on the top. If the target container is not on the top, the output action from the *ANN*s directs action to reshuffle containers. Lines 12 to 15 explain how to get the position to reshuffle a container. Line 16 checks for possible error before reshuffling a container. If there is no error, the codes reshuffle the container according to the output actions; else reshuffle the container by a heuristic and record the error. The program chooses Min-Max heuristic to fix the error if Min-Max heuristic performs better than Look-ahead *N* heuristic in the bay configuration.

Currently the set of codes is for training the *ANN*s to behave as if the Min-Max heuristics. When the *ANN*s are trained for the other two heuristics, Line 2 will be changed accordingly, with minimal changes in other parts of the codes to suit the heuristic under consideration.

IV. Experiment and Analysis

Section 4.1 mentions the settings such as the hyper-parameters, activation function, and cost function for training the *ANN*s in small and large bay sizes. Section 4.2 shows the accuracy of training set and validation set in each dataset. Section 4.3 shows the results of reshuffling containers by trained parameters and analysis. There are two parts in Section 4.3: Main results and secondary results. The main results are related to the objective of our study. The secondary results are related to comparing different methods or hyper-parameters in the *ANNs*.

4.1 Setting of the *ANN*s

4.1.1 Setting of the *ANN*s in Small Bay Size

Table 4.1.1.A shows the setting for training the *ANN*s in small bay size (4-row, 3-column, and 7-container). We use the Sigmoid function as the activation function in the small bay size. We hope to get 100% accuracy in training dataset of small bay. Thus, we increase either the number of perceptrons of a hidden layer or number of epochs. Because the number of data instances is large in some datasets, we increase the number of perceptrons to hundreds of perceptrons to get 100% accuracy. We sometimes train the *ANN*s longer by increasing epochs to reduce hidden layers and perceptrons for 100% accuracy. In the most *ANN* models, we set the learning rate to 0.0035 to ensure the smooth convergence of cost. The number of data instances in some datasets are too small that we can quickly train them to 100% accuracy by increasing learning rate to, e.g., 0.05. The size of Mini-batch is based on 2^n(32, 64, 128, 256,…), we choose 1,024 as the size of Mini-batch. Section 2.4 mentions the meaning of β_1,β_2, and epsilon. The setting of β_1 is common to be 0.9. The authors of Adam (Kingma and Ba

(2015)) suggested the setting of β_2 to be 0.999. The epsilon keeps the value from 0.

Table 4.1.1.A. Setting of the *ANN*s in small bay size

Setting of the *ANN*s	
Learning rate	0.001 ~ 0.05
Number of hidden layers	0 ~ 2
Perceptrons in each hidden layer	5 ~ 600
Epochs	1,000 ~ 30,000
Mini-batch size	1,024
Optimizer	Adam
β_1 for Adam optimizer	0.9
β_2 for Adam optimizer	0.999
Epsilon	10^{-8}
Activation function	Sigmoid
Cost function	Least square

Bengio (2012) gives suggestions on tuning hyper-parameters. However, the tuning process is empirical in our study. In small bay size, we hope to make the *ANN*s over-fitted by training them to 100%. We usually set one or two hidden layer and many nodes in the hidden layer to get high accuracy in small bay size. The number of perceptrons is between 5 to 600 depending on how hard to train the *ANN*s to over-fitting. It is easier to train *ANN*s from the Min-Max and Look-ahead *N* heuristic. We set at most 70 perceptrons in a hidden layer. Training the Better-of-Two is harder. Some *ANN*s apply hundreds of nodes to get 100% accuracy. As for number of epoch, we usually set the epoch between 1,000 to 5,000. However, we set epochs

more than 10,000 in some *ANN*s to get 100% accuracy in setting less nodes in a hidden layer.

We give an example of the *ANN* architecture by following the setting in Table 4.1.2. We show two different *ANN* architectures in the dataset of smaller data instances and larger data instances.

Table 4.1.1.B shows *ANN* architectures in the datasets of small and large data. We utilize the dataset of 4-tier, 3-stack, 4-container, and 3-deadlock as the dataset of small data and 4-tier, 3-stack, 7-container, and 1-deadlock as the dataset of large data. The setting of activation function, cost function, and optimizer follow Table 4.1.1.

Table 4.1.1.B. Setting of *ANN* architectures of small and large data instances in small bay size

Dataset	Number of data instances	Learning rate	Number of hidden layers	Perceptrons in each hidden layer	Epochs
Small	18	0.0035	1	10	10,000
Large	28,080	0.0035	1	500	1,000

The parameters are generated between two layers. The size of parameters is given by the number of perceptrons in two layers. The number of perceptrons in input and output layer are 12 and 6. For the dataset of large data instances, the size of w_1 and b_1 are the 2-dimensional array of [10, 12] and [10, 1]. The sizes of w_2 and b_2 are [6, 10] and [6, 1]. For the dataset of large data instances, the sizes of w_1 and b_1 are [500, 10] and [500, 1]. The sizes of w_2 and b_2 are [6, 500] and [6, 1].

4.1.2 Setting of the *ANN*s in large bay

Table 4.1.2.A shows the setting for training the *ANN*s in large bay size (4-row, 6-column, and 18-container). For the activation function, we use Sigmoid in the last layer and ReLU in other layers. We set more hidden layers in this case than the small bay size. We utilize less epochs and perceptrons because ReLU and Binary cross-entropy can help the *ANN*s with quicker convergence than Sigmoid function and Least square method.

In tuning the hyper parameters in large bay size, we set more hidden layers in each *ANN*. With more data in the dataset, the number of hidden layers is increased from 1 to 4 and the number of perceptrons in hidden layer from 5 to 300. We usually set number of epochs between 100 to 200. We set 1,000 epochs when we want to set less perceptrons and spend more time on training. The setting of learning rate is empirical. Setting between 0.001 to 0.007 is suitable for our *ANN* models in both bay sizes. The detail setting of hyper-parameters is in the Appendix.

Table 4.1.2.A. Setting of the *ANN*s in large bay size

Settings of the *ANNs*	
Learning rate	0.0035 ~ 0.007
Number of hidden layers	1 ~ 4
Perceptrons in each hidden layer	5 ~ 300
Epochs	90 ~ 1,000
Mini-batch size	1,024
Optimizer	Adam
β_1 for Adam optimizer	0.9
β_2 for Adam optimizer	0.999
Epsilon	10^{-8}
Activation function	ReLU, Sigmoid
Cost function	Binary cross-entropy

Table 4.1.2.B shows two examples of *ANN* architecture with two different datasets: 4-tier, 6-stack, 3-container, and 2-deadlock bay (small) and 4-tier, 6-stack, 18-container, and 3deadlock bay (large).

Table 4.1.2.B. Setting of *ANN* architectures of small and large data instances in large bay size

Dataset	**Number of data instances**	**Learning rate**	**Number of hidden layers**	**Perceptrons in each hidden layer**	**Epochs**
Small	12	0.0035	1	5	1,000
Large	100,000	0.0035	4	150	100

The size of parameters is 2-dimensional array as well. The number of perceptrons in input and output layers are 24 and 12, respectively. For the dataset of small data instances, the sizes of w_1 and b_1 are [5, 24] and [5, 1]. The sizes of w_2 and b_2 are [12, 5] and [12, 1]. For the dataset of large data instances, the sizes of w_1 and b_1 are [150, 24] and [150, 1]. The sizes ofw_2 to w_4 and b_2 to b_4 are all [150, 150] and [150, 1]. The sizes of w_5 and b_5 are [12, 150] and [12, 1].

4.1.3 The activation Function and cost function between small and large bay size

Table 4.1.3 shows the use of activation and cost function in small and large bay size. We use two activation functions in hidden layer because there are some drawbacks in Sigmoid function. First, exponential function in Sigmoid is time-consuming to compute. Second, Sigmoid faces saturation for extreme values. When doing derivative on Sigmoid, the value will be $a \times (1 - a)$. With larger value of a, the value after derivative will be smaller which makes parameters update slower. Third, the value of a after Sigmoid is always [0, 1], and $a \times (1 - a)$ is always positive. Thus, parameters follow the same direction (toward positive or negative) to be updated. The computational time in ReLU is faster than Sigmoid and there is no saturation problem in ReLU. Thus, we apply ReLU in more complicated problem which is large bay size.

Sigmoid function is unable to be applied in hidden layers of large bay size. Thus, we apply Sigmoid function in small bay size to see whether it works or not.

In the use of cost function, binary cross-entropy is more suitable than least-square in our ANN model. We define our ANN model as a classification problem. Least-square is more suitable in regression problem. Using Sigmoid and least-square still solve the CRP in the small bay size. For large bay size, using ReLU and binary cross entropy are more efficient. As for the activation in output layer, we apply Sigmoid to ensure the value between 0 to 1 and binary

cross-entropy work. Softmax function is not suitable for our problem. Softmax is suitable for multi-class classification problems where there are many classes but only one element is labeled as 1; the other elements 0 in the column vector. The function makes the predicted value between [0, 1] and the sum of each label is 1. Thus, each value *a* will be turned into a probability. It utilizes the exponential function to highlight the maximum value in *a*. Thus, the probability of maximum value will be much higher than others after applying softmax function. Our problem is multi-label classification problem which allows more labels to be assigned 1 and others are 0. In fact, we allow two labels to be assign 1. It follows the rule in section 3.1.2 to assign the value. The feature of highlight maximum value in softmax does not fit the multi-label.

Table 4.1.3. The activation and cost function between small and large bay size

Bay size	Small	Large
Activation function in hidden layer	Sigmoid	ReLU
Activation function in output layer	Sigmoid	Sigmoid
Cost function	Least-square	Binary cross-entropy

4.2 Accuracy of Training Set and Validation Set

In small bay size, we create all combinations of bay configurations to generate datasets. Thus, we do not need to split data into different sets. We make each dataset over-fitting to 100% accuracy in small size. In large bay size, Table 4.2.1, Table 4.2.2, and Table 4.2.3 represent the accuracy of Min-max, Look-ahead *N*, and Better-of-Two datasets. The name of each dataset is n_m, where n and m represent number of containers and deadlocks of the dataset. There are

the accuracy of 1 training set and 2 validation sets for each *ANN*. The column “Train” means the accuracy of the training set. The column “Valid_1” and “Valid_2” mean the accuracy of the 2 validation sets. Setting 2 validation sets helps us to find more suitable trained parameters. We do not compute the testing set because we will verify the performance of the *ANN*-based system by reshuffling containers rather than computing the accuracy of the training set. The training set has contained all combination of data if column Valid_1 and Valid_2 are empty. “Accuracy A” is the average accuracy of each training set. “Accuracy B” is the average accuracy of the training set for the datasets with validation sets.

Table 4.2.1 shows training results of the *ANNs* in imitating the Min-Max heuristic. We try to make each dataset without validation sets over-fitting. Thus, 100% accuracy in some datasets is over-fitting. For datasets with “Valid_1” and “Valid_2”, most of the accuracy is more than 90%. There are 3 datasets which the accuracy of validation sets is worse: 7_1, 7_2, and 8_2. These 3 datasets are generated in different ways: we generate all combination of bay configurations and randomly choose the ideal number of data instances to form a dataset. The other random datasets are directly generated in random. It might be the reason for the lower accuracy of validation set than other datasets.

Table 4.2.1. Accuracy of Min-Max dataset in large bay size

Dataset	Train	Valid_1	Valid_2	Dataset	Train	Valid_1	Valid_2
3_1	100.00%	---	---	11_2	98.17%	93.40%	92.50%
3_2	100.00%	---	---	11_3	98.46%	91.60%	91.10%
4_1	100.00%	---	---	12_1	98.83%	95.00%	95.40%
4_2	100.00%	---	---	12_2	99.02%	94.90%	94.70%
4_3	100.00%	---	---	12_3	98.05%	93.30%	92.70%
5_1	100.00%	---	---	13_1	97.98%	95.30%	95.60%
5_2	100.00%	---	---	13_2	97.94%	92.20%	93.50%
5_3	100.00%	---	---	13_3	99.22%	94.20%	94.40%
6_1	99.77%	---	---	14_1	97.59%	93.10%	94.55%
6_2	100.00%	---	---	14_2	97.45%	93.85%	93.75%
6_3	100.00%	---	---	14_3	98.99%	95.40%	95.80%
7_1	94.54%	78.30%	76.30%	15_1	96.15%	91.50%	91.85%
7_2	95.58%	75.30%	74.40%	15_2	97.82%	93.20%	93.65%
7_3	93.57%	---	---	15_3	97.91%	94.80%	94.60%
8_1	98.51%	90.10%	90.40%	16_1	96.80%	92.45%	92.30%
8_2	93.96%	76.90%	77.70%	16_2	96.55%	91.20%	91.20%
8_3	96.51%	83.50%	84.00%	16_3	97.84%	93.55%	93.55%
9_1	99.29%	91.70%	92.00%	17_1	94.79%	91.35%	91.35%
9_2	97.11%	90.40%	90.70%	17_2	97.10%	92.75%	92.60%
9_3	95.55%	86.50%	87.90%	17_3	98.13%	94.90%	95.30%
10_1	97.98%	91.80%	93.50%	18_1	96.33%	92.10%	92.05%
10_2	98.01%	89.50%	90.60%	18_2	97.36%	92.90%	92.80%
10_3	97.26%	86.80%	90.00%	18_3	97.18%	94.15%	93.25%
11_1	98.18%	93.60%	93.80%	**Accuracy A**	97.90%	90.90%	91.14%
				Accuracy B	97.37%		

Figure 4.2.1 shows the comparison of accuracy with more containers. We do not compare the accuracy from 3_1 to 6_3 because we make them 100% which is over-fitting. we add the same accuracy of Train to Valid_1 and Valid_2 in 7_3 to enable us to do comparison. We calculate the average accuracy of each deadlock with the same container. For example, the average accuracy of 7_1, 7_2, and 7_3 is the accuracy of container 7 in Figure 4.2.1.

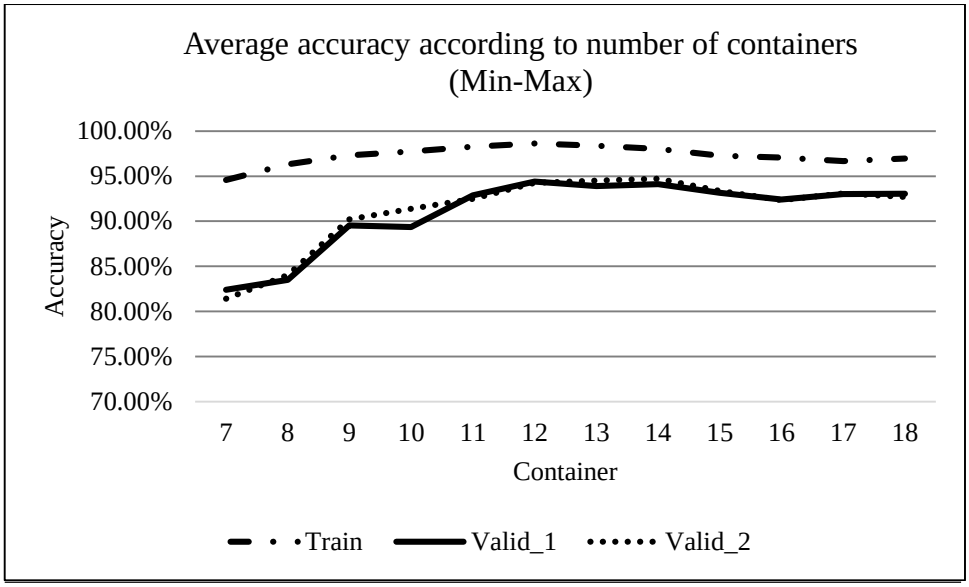


Figure 4.2.1. Average accuracy according to number of containers (Min-Max)

The accuracy in training set is stable. The two validation sets in Container 7 and 8 are affected by 7_1, 7_2, and 8_2. The accuracy of validation sets slightly increases from Container 9 to 15, decreases from 15 to 18.

Table 4.2.2 shows the accuracy of Look-ahead *N* dataset. The average accuracy of Look-ahead *N* dataset is slightly lower than Min-Max dataset. With more containers, the accuracy of the validation set in Look-ahead *N* is lower. However, it is still uncertain about the large different accuracy between Min-Max and Look-ahead *N* in validation sets of dataset 7_1, 7_2, and 8_2.

Table 4.2.2. Accuracy of Look-ahead dataset in large bay size

Dataset	Train	Valid_1	Valid_2	Dataset	Train	Valid_1	Valid_2
3_1	100.00%	---	---	11_2	97.48%	93.10%	94.40%
3_2	100.00%	---	---	11_3	97.95%	93.90%	93.50%
4_1	100.00%	---	---	12_1	95.10%	92.00%	89.10%
4_2	100.00%	---	---	12_2	96.71%	93.30%	92.30%
4_3	100.00%	---	---	12_3	97.06%	91.20%	90.80%
5_1	100.00%	---	---	13_1	94.76%	90.70%	90.40%
5_2	100.00%	---	---	13_2	95.80%	90.30%	92.20%
5_3	100.00%	---	---	13_3	97.52%	90.70%	91.80%
6_1	100.00%	---	---	14_1	95.44%	88.40%	87.90%
6_2	100.00%	---	---	14_2	95.84%	87.50%	87.80%
6_3	100.00%	---	---	14_3	94.08%	89.80%	90.50%
7_1	99.63%	98.00%	97.90%	15_1	92.26%	90.10%	89.60%
7_2	99.66%	98.90%	98.50%	15_2	92.58%	88.90%	87.00%
7_3	100.00%	---	---	15_3	93.27%	87.80%	88.80%
8_1	98.61%	97.90%	97.30%	16_1	90.13%	87.70%	87.00%
8_2	99.47%	99.00%	98.60%	16_2	91.81%	85.10%	85.10%
8_3	99.89%	99.00%	99.80%	16_3	92.55%	88.00%	86.20%
9_1	98.71%	95.30%	95.80%	17_1	92.20%	84.10%	85.60%
9_2	98.88%	95.00%	95.70%	17_2	92.97%	85.20%	85.70%
9_3	99.48%	97.20%	97.60%	17_3	93.20%	81.00%	80.20%
10_1	96.76%	92.40%	91.80%	18_1	90.85%	83.80%	84.30%
10_2	98.38%	94.10%	92.40%	18_2	87.53%	84.00%	82.70%
10_3	99.10%	95.70%	95.70%	18_3	90.61%	80.20%	81.10%
11_1	96.31%	91.40%	93.30%	**Accuracy A**	96.65%	90.88%	90.81%
				Accuracy B	95.50%		

Figure 4.2.2 follows the same method in Figure 4.2.1 for comparison. It is obvious that the accuracies of three sets decrease with the increase in the number of containers in the datasets.

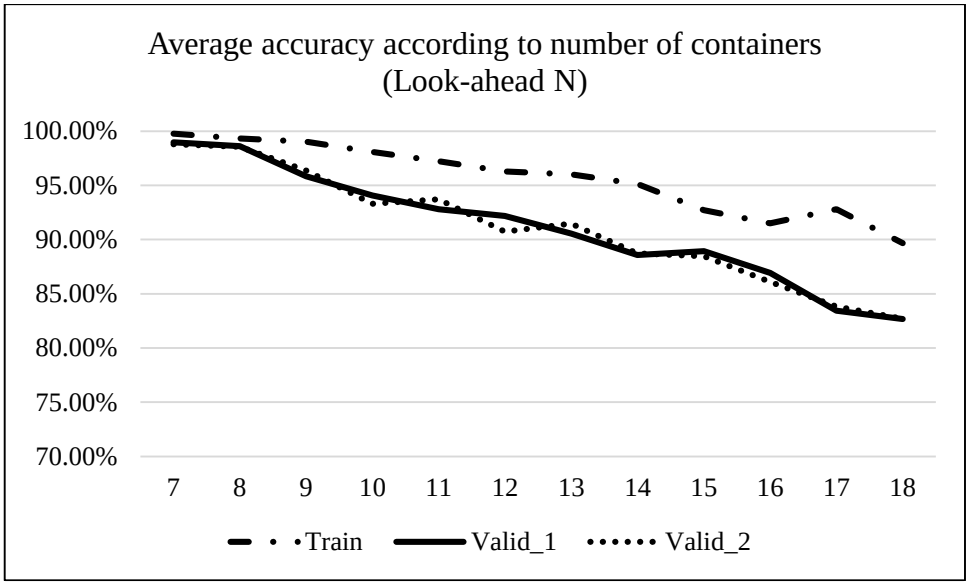


Figure 4.2.2. Average accuracy according to number of containers (Look-ahead *N*)

Table 4.2.3 shows the accuracy of Better-of-Two. We do not combine the datasets for less than 8 because the total reshuffle times of applying Min-Max and Look-ahead *N* to 4-row, 6-column, and 7-container bay are the same. In Table 4.2.3, the accuracies of training and validation sets are lower than the 2 previous results. The accuracy is relatively low with more containers in the dataset. It is uncertain about the lower accuracy of dataset 8_2 when we use a different method to generate the dataset. In general, most of the datasets in Min-Max, Look-ahead *N*, and Better-of-Two are well-trained. We reshuffle containers by the trained *ANN*s.

Table 4.2.3. Accuracy of Better-of-Two dataset in large bay size

Dataset	Train	Valid_1	Valid_2	Dataset	Train	Valid_1	Valid_2
8_1	90.49%	89.60%	90.00%	13_3	97.61%	93.30%	92.20%
8_2	83.17%	76.90%	76.20%	14_1	95.04%	89.70%	90.50%
8_3	91.16%	82.30%	83.70%	14_2	96.49%	89.60%	90.80%
9_1	98.10%	91.20%	91.00%	14_3	98.36%	94.00%	92.60%
9_2	95.91%	87.40%	87.90%	15_1	96.17%	93.10%	91.80%
9_3	94.63%	86.10%	87.70%	15_2	95.55%	89.00%	88.00%
10_1	96.95%	92.40%	92.40%	15_3	97.21%	90.10%	90.50%
10_2	96.26%	88.30%	89.00%	16_1	95.92%	88.80%	86.50%
10_3	96.01%	89.20%	89.90%	16_2	95.70%	85.20%	85.80%
11_1	95.05%	91.30%	91.20%	16_3	97.13%	89.70%	89.50%
11_2	96.42%	92.60%	93.60%	17_1	96.47%	88.50%	88.10%
11_3	95.57%	90.60%	89.60%	17_2	95.00%	84.80%	84.50%
12_1	95.69%	92.70%	91.40%	17_3	97.00%	87.50%	88.30%
12_2	96.65%	91.40%	90.80%	18_1	96.29%	86.50%	85.70%
12_3	97.82%	92.90%	92.40%	18_2	95.78%	82.30%	84.30%
13_1	96.33%	91.60%	91.70%	18_3	97.13%	86.70%	86.80%
13_2	96.18%	89.40%	90.70%	Accuracy A	95.61%	88.93%	88.94%

Figure 4.2.3 follows the same method in Figure 4.2.1 and Figure 4.2.2 for comparison. The accuracies of Container 8 are affected by 8_2. The accuracy of training set is stable from Container 9 to 15. The accuracies of the two validation sets increase from Container 9 to 15, decrease from 12 to 18, especially from 15 to 18.

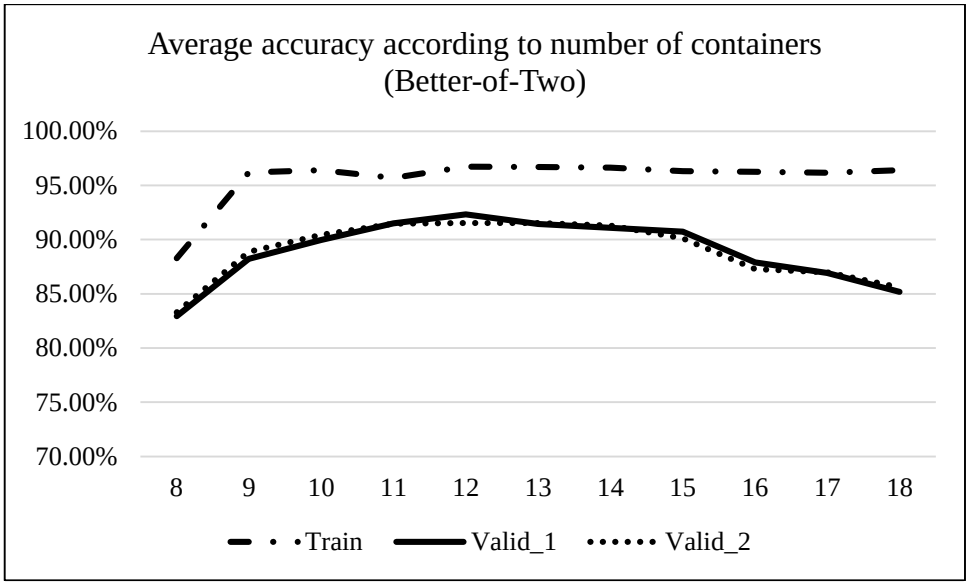


Figure 4.2.3. Average accuracy according to number of containers (Better-of-Two)

4.3 Computational Results and Analysis

4.3.1 Main Results and Analysis

We have mentioned that we do not use testing data in each dataset to verify the performance of the *ANN*-based system. We reshuffle containers by the *ANN*-based system to verify the performance. Table 4.3.1 shows the results of reshuffling containers by heuristics and the *ANN*-based system in small bay size. The parameters completely learn the logic of heuristics to reshuffle containers. There is no error when reshuffling containers. However, the computational time in reshuffling by parameters is more than the heuristics. They are about 3 and 6 times of Min-Max and Look-ahead *N*.

Table 4.3.1. Results of reshuffling containers by heuristics and the *ANN*-based system in small bay size

Reshuffled by Heuristics (90,720 bay configurations)				
Name	**Reshuffle times**	**Average reshuffle times**	**Computational time(s)**	
Min-max	295,800	3.26	42.55	
Look-ahead *N*	289,212	3.188	25.24	
Combined	288,912	3.185		
Reshuffled by the *ANN*-based system (90,720 bay configurations)				
Name	**Reshuffle times**	**Average reshuffle times**	**Computational time(s)**	**Errors**
Min-max	295,800	3.26	124.74	0
Look-ahead *N*	289,212	3.188	139.32	0
Combined	288,912	3.185	143.59	0

Table 4.3.2 shows reshuffle results in large bay size. When facing the error, the program will reshuffle container by Min-max or Look-ahead *N*. “Error_MM” or “Error_LA” are the container reshuffled by Min-Max or Look-ahead N when there is an error that the *ANN*s in the *ANN*-based system are unable to reshuffle the container. The parameters do not completely

learn the logic of heuristics or surpass them. Among three types of parameters, the performance of Min-Max parameter is the closest on average reshuffle times. It reflects on the highest average accuracy in the last section. The Min-Max *ANN*-based system also makes fewer errors when reshuffling containers. Look-ahead *N* is better than Min-Max in Heuristics. However, the performance of the Look-ahead *N ANN*-based system is the worst in reshuffle times, computational time, and number of errors. It is harder for the *ANN*s to learn the logic of Look-ahead *N* than Min-Max in our experiments. The performance of the Better-of-Two *ANN*-based system is between the Min-Max and Look-ahead *N ANN*-based system in reshuffle times and errors. Because of better performance in the Min-Max *ANN*-based system, the Better-of-Two *ANN*-based system learns some logic of Min-Max heuristic to get fewer reshuffle times.

Table 4.3.2. Results of reshuffling containers by heuristics and the *ANN*-based system in large bay size

Reshuffled by Heuristics					
Name	**Reshuffle moves**	**Average reshuffle moves**	**Computational time(s)**		
Min-max	8,809,699	8.81	992.97		
Look-ahead *N*	8,724,514	8.72	846.53		
Combined	8,675,800	8.68	1482.83		
Reshuffled by the *ANN*-based system					
Name	**Reshuffle moves**	**Average reshuffle moves**	**Computational time(s)**	**Errors_MM**	**Errors_LA**
Min-max	8,929,728	8.93	6,172.91	9088 (0.1%)	---
Look-ahead *N*	9,009,937	9.01	6,341.09	---	15,837 (0.18%)
Combined	8,967,150	8.97	10,156.51	600 (0.007%)	11,370 (0.13%)

We generated all combinations of data in small bay size to train the *ANN*s. Thus, the *ANN*s can completely learn the logic of the two heuristics and learn how to surpass these two heuristics. Though we are able to generate few data in large bay size, the trained parameters

still can reshuffle most of the containers and the average reshuffle times are very close to results from the heuristics. However, computational time is a concern. With larger bay size, the programs spend more time on reshuffling. Their average computation time is seven to eight times of the two heuristics on reshuffling containers.

4.3.2 Secondary Results and Analysis

In this section, we compare the effects of two methods: Adam and Mini-batch gradient descent. We apply the methods in the literature to improve the efficiency of the *ANN*s. We test these methods in the datasets of container bay. There are many datasets to train in our study. We only pick a dataset to do the comparison. We want to measure the performance of the *ANN*s with and without Adam optimizer (Kingma and Ba (2015)). We use the *ANN* trained by 4-row, 6-column, 9-container, 2 deadlocks, and Min-Max dataset to measure the Adam optimizer. In the setting of hyper-parameter, Table 4.3.3 shows the hyper-parameters and functions we use. Table 4.3.4 shows the computational time and the accuracy of training set and validation sets. There is no significant difference in computational time. However, the Adam optimizer helps the *ANNs* to increase accuracy. In Figure 4.3.1 and Figure 4.3.2, epoch 0 means the *ANN*s has calculated the first epoch. The difference in accuracy is significant from epoch 0 to epoch 10. Without Adam, the accuracy grows from 15.4% to 23.8% in 10 epochs. With Adam, the accuracy grows from 26.8% to 81.7%.

Table 4.3.3. Setting of the *ANN* in 4-row, 6-column, 9-container, and 2-deadlock

Settings of the *ANN* in 4-row, 6-column, 9-container, and 2-deadlock			
Learning rate	0.0035	V for Adam optimizer	0.9
Number of hidden layers	4	S for Adam optimizer	0.999
Perceptrons in each hidden layer	150	Epsilon	10^{-8}
Epochs	100	Mini-batch size	1,024
Optimizer	Adam	Activation function	ReLU, Sigmoid
Cost function	Binary cross-entropy		

Table 4.3.4 Results of computational time and the accuracy

Name	**Without Adam With Mini-batch**	**With Adam With Mini-batch**
Computational time (s)	357.3	378.81
Accuracy of Train	30.52%	97.11%
Accuracy of Valid_1	28.90%	90.40%
Accuracy of Valid_2	29.50%	90.70%

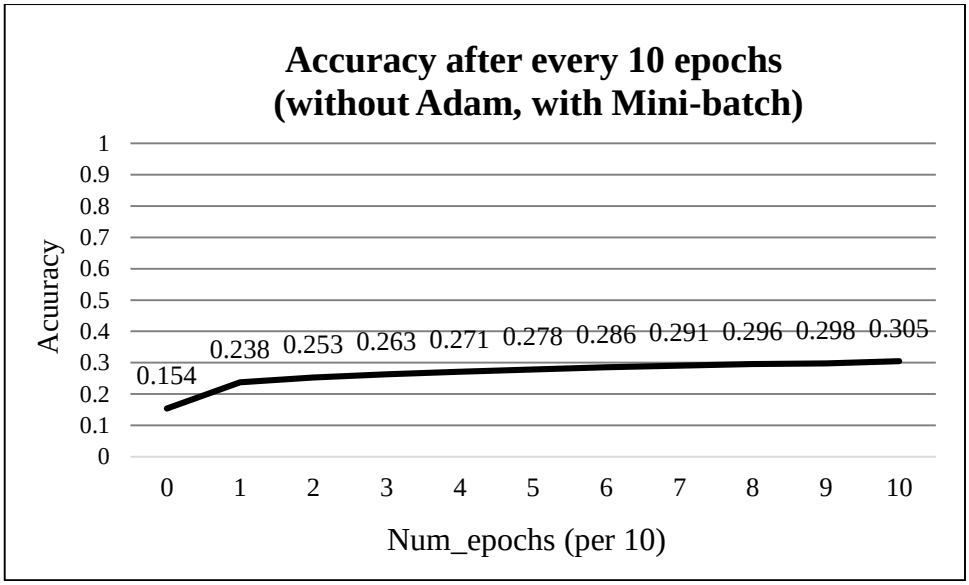


Figure 4.3.1 Accuracy after every 10 epochs without Adam, with Mini-batch

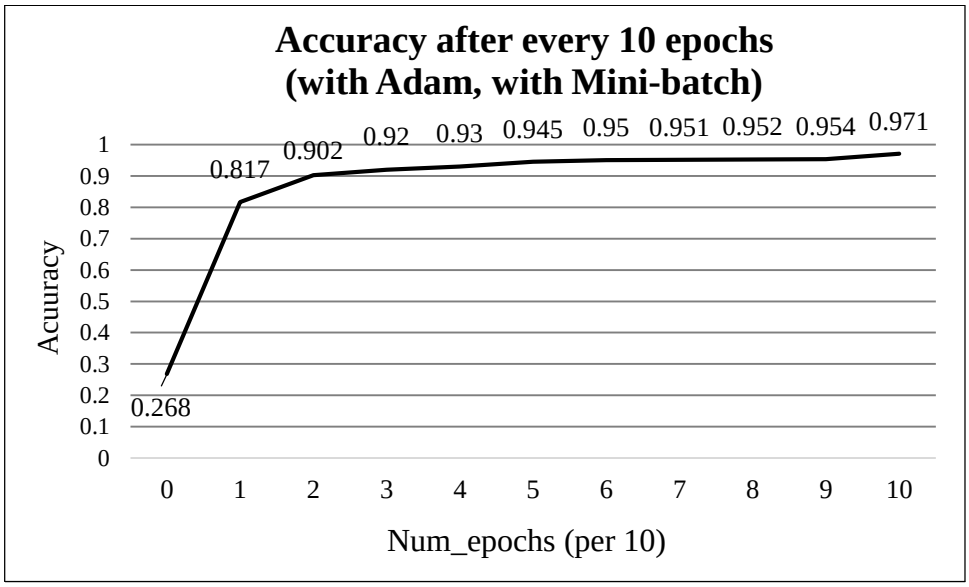


Figure 4.3.2 Accuracy after every 10 epochs with Adam, with Mini-batch

In the next comparison, we measure the effect of Mini-batch gradient descent (Hinton et al. (2012)). We use the same dataset and setting in Table 4.3.5 except for mini-batch size: with 1,024 mini-batch size and without mini-batch size. The application of Mini-batch gradient

descent helps *ANN* to improve the accuracy in limited epochs. Figure 4.3.3 and 4.3.4 show the difference in accuracy when applying Mini-batch gradient descent in *ANN*. Thus, we apply both Adam optimizer and Mini-batch gradient descent to improve the efficiency of *ANN*.

Table 4.3.5 Results of computational time and the accuracy

Name	**Without Mini-batch, With Adam**	**With Mini-batch, With Adam**
Computational time (s)	357.30	469.23
Accuracy of Train	33.03%	97.11%
Accuracy of Valid_1	32.30%	90.40%
Accuracy of Valid_2	31.20%	90.70%

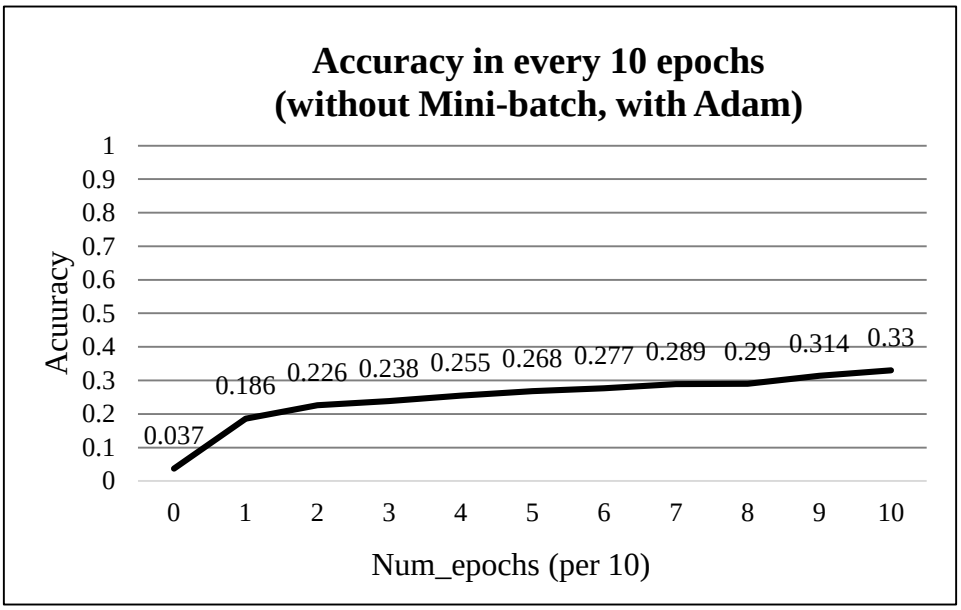


Figure 4.3.3 Accuracy after every 10 epochs without Mini-batch, with Adam

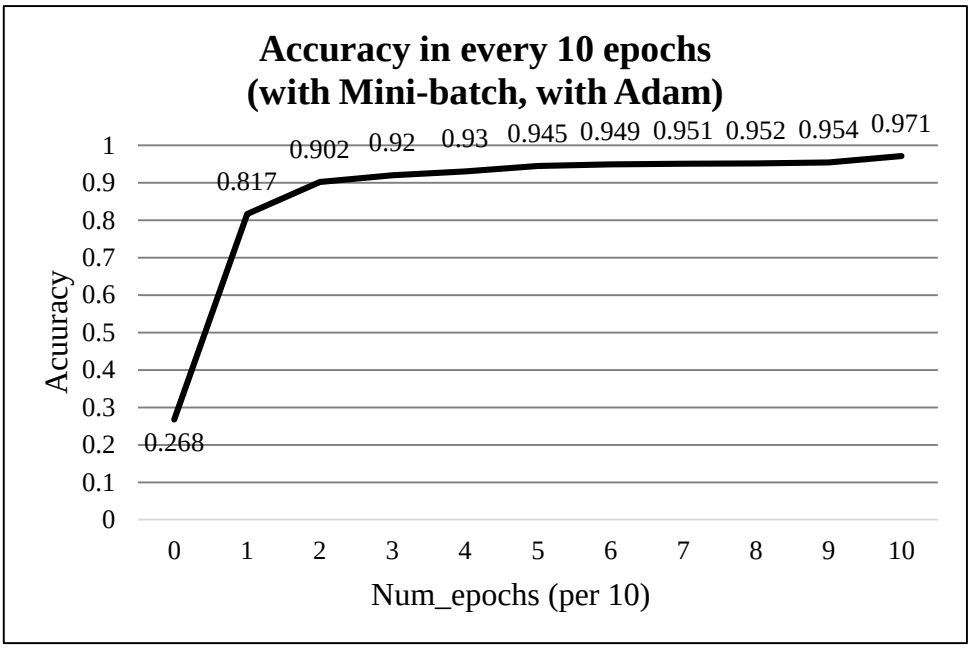


Figure 4.3.4 Accuracy after every 10 epochs with Mini-batch, with Adam

V. Discussion

5.1 Limitations of the Study

There are limitations in our study. We are unable to train the *ANN*s with a large number of data because the computer will run out memory. We do not use libraries such as Tensorflow or Keras to help us to train the *ANN*s. These affect the accuracy in large bay size. The lack of programming skill may influence the computational time on reshuffling containers.

5.2 Conclusion

Our study is a novel research in container relocation problem (CRP). We solve the CRP by applying the *ANN*-based system. The system contains many trained *ANN*s according to the size, container, and deadlock of the bay and the two heuristics to reshuffle containers. The *ANN*-based system can surpass Min-Max and Look-ahead *N* in small bay size if we can train all kinds of data from bay configurations. In large bay size, in most of the time the *ANN*-based system can still effectively reshuffle containers even for new, unknown bay configurations. Only few errors exist. The performance of the *ANN*-based system in large bay size does not surpass heuristics but close to them.

5.3 Future Research

1. In our study, we use two heuristics to train the *ANNs*. Adding more heuristics for solving CRP to generate the datasets to train the *ANNs* may lead to more powerful *ANN*-based system that reshuffle containers in fewer moves.

2. Using *Reinforcement Learning*. It is one of the algorithms in machine learning that is different from *ANN*. In Reinforcement Learning, the program learns how to reshuffle containers according to the feedback of results. The program will get a positive or negative reward depending on the performance of reshuffle.

References

朱威倫.(2017). 以走廊演算法結合堆疊規則處理預先整櫃問題. 國立東華大學運籌管理所碩士論文

Bengio, Y. (2012) Practical recommendation for gradient-based training of deep architectures *Neural Networks: Tricks of the Trade*, pp 437-478.

Carlo, H.J., Vis, I. F. A., & Roodbergen, K. J. (2014). Transport operations in container terminals: Literature overview, trends, research directions and classification scheme. *European Journal of Operational Research,* 236 (1), pp. 1–13.

Caserta, M., & Voß, S. (2009). A corridor method-based algorithm for the pre-marshalling problem. *Applications of Evolutionary Computing*, pp. 788-797.

Caserta, M., Voß, S. & Sniedovich, M. (2011). Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33, pp. 915-929.

Cordeau, J.F., Laporte, G., Legato, P., & Moccia, L. (2005). Models and tabu search heuristics for the berth allocation problem. *Transportation Science*, 39(4), pp. 526-538.

Glorot, X., & Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistic*, 9, pp. 249–256.

Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signal, and Systems*, 4, pp. 303-314.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. *IEEE International Conference on Computer Vision*, pp 1026–1034, December, Santiago, Chile, 2015.

Hinton, G. E., Srivastava, N., & Swersky, K. (2012). Lecture 6a overview of mini–batch gradient descent.

https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, Browsed on October, 2018.

Jovanovic, R., & Voß, S. (2014). A chain heuristic for the blocks relocation problem. *Computer & Industrial Engineering*, 75, pp. 79-86.

Kim, K. H., & Hong, G. P. (2006). A heuristic rule for relocating blocks, Computers & Operations Research, 33(4), pp. 940-954.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. International Conference on Learning Representations, May, San Diego, USA, 2015.

Li, Fei-Fei., Johnson, J., Serena, Y. (2017). Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/2017/, Browsed on December, 2018.

Lee, Y., & Lee, Y. J. (2010). A heuristic for retrieving containers from a yard. *Computers & Operations Research*, 37, pp. 1139–1147.

Lee, Y., & Hsu, N. Y. (2007). An optimization model for the container pre-marshalling problem. *Computers & Operations Research*, 34 (11), pp. 3295–3313.

Murty, K. G., Liu, J., Wan, Y-w., & Linn, R. (2005). A decision support system for operations in a container terminal. *Decision Support System*, 39(3), pp. 309–332.

Nair, V & Hinton, G. E. (2010). Rectified linear units improve restricted boltzman machine, *Proceedings of the 27th International Conference on International Conference on Machine Learning*, pp. 807-814, June, Haifa, Isreal, 2010.

Ng, A. (2017). Deep learning specialization. https://www.coursera.org/specializations/deeplearning, Browsed on December, 2018.

Ng, A. (2004). Feature selection, L1 vs. L2 regularization, and rotational invariance.

Petering, M. E. H., & Hussein, M. I. (2013). A new mixed integer program and extended look-ahead heuristic algorithm for the block relocation problem. *European Journal of Operational Research*, 231, pp. 120–130.

Rumelhart, D. E., Hinton, G. E., & Willams, R. J. (1986) Learning representations by back-propagating errors. *Nature*, 323, pp. 533-536.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning*, pp. 1139–1147, June, Atlanta, USA, 2013.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014) Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, pp. 1929-1958.

Tierney, K., & Malitsky, Y. (2015). An algorithm selection benchmark of the container pre-marshalling problem. *Lecture Notes in Computer Science,* 8994, pp. 17-22.

Wan, Y-w., Liu, J, & Tsai, P. C. (2009). The assignment of storage locations to containers for a container stack. *Naval Research Logistics*, 56(8), pp. 699-713.

Yang, J. H., & Kim, K. H. (2006). A grouped storage method for minimizing relocations in block stacking systems. *Journal of Intelligent Manufacturing*, 17, pp.453–463.

Zhang, C., Liu, J., Wan, Y-w., Murty, K. G., & Linn, R. J. (2003) Storage space allocation in container terminals. *Transportation Research-B*, (37) pp. 883–903.

UNCTAD Secretariat (2017) Review of maritime transport. *United Nations Conference on Trade and Development*, Geneva, Switzerland.

Appendix

We provide the architecture and accuracy of each *ANN* that we train in small and large bay size in Appendix. In small bay size, we train the dataset of 0 deadlock. Thus, there are 4 categories of datasets: 0 deadlock, 1 deadlocks, 2 deadlocks, and 3 deadlocks. Figure 7.1 shows the architecture and accuracy of each *ANN* from Min-Max heuristic. The name of dataset follows the form of T_S_C_D. T and S are tiers and stacks which describe the size of the bay. C is number of containers. D is number of deadlocks. The choice of activation function, cost function and optimizer follow Table 4.1.1 to do setting. In the column of Perceptrons, the list contains number of perceptrons in each hidden layer.

Table A.1. Architecture and accuracy of each *ANN* from the Min-Max heuristic in small bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy	Computational Time(s)
4_3_7_0	34,560	0.0035	2	[10, 5]	3,000	100%	88.05
4_3_7_1	28,080	0.0035	1	[70]	10,000	100%	1189.37
4_3_7_2	19,440	0.0035	1	[70]	10,000	100%	343.66
4_3_7_3	8,640	0.0035	1	[70]	10,000	100%	225.91
4_3_6_0	5,670	0.0035	2	[10, 5]	10,000	100%	57.27
4_3_6_1	4,320	0.002	1	[50]	10,000	100%	113.17
4_3_6_2	2,520	0.001	1	[35]	10,000	100%	40.18
4_3_6_3	1,080	0.0035	2	[10, 5]	50,000	100%	113.48
4_3_5_0	1,008	0.001	2	[10, 5]	20,000	100%	13.51
4_3_5_1	648	0.003	1	[30]	15,000	100%	13.34
4_3_5_2	360	0.0035	2	[10, 6]	30,000	100%	16.44
4_3_5_3	144	0.001	2	[10, 5]	20,000	100%	7.77
4_3_4_0	180	0.0035	1	[10]	20,000	100%	6.19
4_3_4_1	108	0.0035	1	[15]	15,000	100%	6.66
4_3_4_2	54	0.0035	1	[15]	10,000	100%	5.79
4_3_4_3	18	0.0035	1	[10]	10,000	100%	3.71
4_3_3_0	36	0.0035	1	[10]	10,000	100%	3.66
4_3_3_1	18	0.0035	1	[5]	15,000	100%	3.76
4_3_3_2	6	0.05	0	---	1,000	100%	1.71

Due to the low amount of data instances in datasets 4_3_3_2 to 4_3_6_0, we put data together according to T_S_C. We erase the dataset in number of deadlocks from container 1 to

6. Thus, some datasets from Look-ahead *N* and Better-of-Two are different to the datasets from the Min-Max heuristic. Table A.2 shows the architecture and accuracy of each *ANN* from the Look-ahead *N* heuristic. For example, the dataset 4_3_4_0, 4_3_4_1, 4_3_4_2, and 4_3_4_3 are put in one dataset called 4_3_4.

Table A.2. Architecture and accuracy of each *ANN* from the Look-ahead *N* heuristic in small bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy	Computational time(s)
4_3_7_0	34,560	0.0035	1	[50]	356	100%	43.45
4_3_7_1	28,080	0.0035	1	[160]	1,130	100%	293.73
4_3_7_2	19,440	0.0035	2	[150, 150]	1,155	100%	533.05
4_3_7_3	8,640	0.0035	2	[150, 150]	1,000	100%	185.02
4_3_6	13,680	0.0035	2	[170, 170]	1,000	100%	406.19
4_3_5	2,160	0.0035	1	[150]	5,000	100%	101.99
4_3_4	360	0.0035	1	[50]	5,000	100%	6.26
4_3_3	60	0.0035	1	[15]	5,000	100%	3.54

Table A.3 shows the architecture and accuracy of each *ANN* from the Better-of-Two. Dataset 4_3_7_0 is not included because the two heuristics do the same moves when there is no deadlock on the target stack. Thus, we do not need to create a new dataset 4_3_7_0.

Table A.3. Architecture and accuracy of each *ANN* from the Better-of-Two in small bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy	Computational time(s)
4_3_7_1	28,080	0.0035	1	[500]	1,000	100%	1160.14
4_6_7_2	19,440	0.0035	1	[600]	1,000	100%	693.60
4_6_7_3	8,640	0.0035	2	[150, 150]	1,000	100%	164.28
4_6_6	13,680	0.0035	2	[170, 170]	1,000	100%	347.06
4_6_5	2,160	0.0035	1	[150]	5,000	100%	97.91
4_6_4	360	0.0035	1	[50]	5,000	100%	5.88
4_6_3	60	0.0035	1	[15]	5,000	100%	3.45

Table A.4. Architecture and accuracy of each *ANN* from the Min-Max heuristic in large bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy of Train	Accuracy of Valid_1	Accuracy of Valid_2	Computational time(s)
4_6_18_1						96.33%	92.10%	92.05%	380.71
4_6_18_2						97.36%	92.90%	92.80%	403.88
4_6_18_3						97.18%	94.15%	93.25%	385.28
4_6_17_1						94.79%	91.35%	91.35%	436.29
4_6_17_2						97.10%	92.75%	92.60%	395.84
4_6_17_3						98.13%	94.90%	95.30%	384.70
4_6_16_1						96.80%	92.45%	92.30%	384.74
4_6_16_2						96.55%	91.20%	91.20%	388.61
4_6_16_3						97.84%	93.55%	93.55%	381.02
4_6_15_1						96.15%	91.50%	91.85%	387.69
4_6_15_2						97.82%	93.20%	93.65%	383.16
4_6_15_3						97.91%	94.80%	94.60%	385.18
4_6_14_1						97.59%	93.10%	94.55%	354.52
4_6_14_2			4	[150,150,150,150]	100	97.45%	93.85%	93.75%	344.77
4_6_14_3	100,000					98.99%	95.40%	95.80%	396.77
4_6_13_1						97.98%	95.30%	95.60%	346.11
4_6_13_2						97.94%	92.20%	93.50%	377.96
4_6_13_3						99.22%	94.20%	94.40%	353.29
4_6_12_1						98.83%	95.00%	95.40%	372.11
4_6_12_2		0.0035				99.02%	94.90%	94.70%	391.40
4_6_12_3						98.05%	93.30%	92.70%	334.60
4_6_11_1						98.18%	93.60%	93.80%	339.03
4_6_11_2						98.17%	93.40%	92.50%	330.33
4_6_11_3						98.46%	91.60%	91.10%	342.61
4_6_10_1						97.98%	91.80%	93.50%	381.19
4_6_10_2						98.01%	89.50%	90.60%	502.23
4_6_10_3						97.26%	86.80%	90.00%	330.28
4_6_9_1			3	[200, 200, 200]	200	99.29%	91.70%	92.00%	838.73
4_6_9_2			4	[150,150,150,150]	100	97.11%	90.40%	90.70%	378.81
4_6_9_3			4	[150,150,150,150]	100	95.55%	86.50%	87.90%	401.25
4_6_8_1	200,000		4	[300,300,300,300]	300	98.51%	90.10%	90.40%	4967.66
4_6_8_2	200,000		4	[300,300,300,300]	200	93.96%	76.90%	77.70%	4344.48
4_6_8_3	200,000		4	[300,300,300,300]	200	96.51%	83.50%	84.00%	3304.54
4_6_7_1	100,000		4	[150,150,150,150]	300	94.54%	78.30%	76.30%	1102.24
4_6_7_2	100,000		3	[200, 200, 200]	300	95.58%	75.30%	74.40%	1061.68
4_6_7_3	151,200		3	[300, 300, 300]	300	93.57%	---	---	2886.51
4_6_6_1	86,400		4	[300,300,300,300]	390	99.77%	---	---	3139.38
4_6_6_2	36,000		3	[270, 270, 270]	290	100.00%	---	---	557.45
4_6_6_3	10,800		3	[200, 200, 200]	500	100.00%	---	---	222.71
4_6_5_1	7,920	0.005	2	[200, 200]	1,000	100.00%	---	---	218.67
4_6_5_2	2,880	0.005	2	[130, 130]	1,000	100.00%	---	---	53.54
4_6_5_3	720	0.005	2	[100, 100]	1,000	100.00%	---	---	5.53
4_6_4_1	756	0.005	2	[50, 50]	1,000	100.00%	---	---	3.05
4_6_4_2	216	0.005	2	[30, 30]	1,000	100.00%	---	---	1.25
4_6_4_3	36	0.005	1	[15]	1,000	100.00%	---	---	0.41
4_6_3_1	72	0.0035	1	[15]	2,000	100.00%	---	---	0.73
4_6_3_2	12	0.0035	1	[5]	1,000	100.00%	---	---	0.32

Table A.5. Architecture and accuracy of each *ANN* from the Look-ahead heuristic in large bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy of Train	Accuracy of Valid_1	Accuracy of Valid_2	Computational time(s)
4_6_18_1		0.005		[200,200,200,200,200]	100	90.85%	83.80%	84.30%	664.99
4_6_18_2		0.007		[200,200,200,200,200]	100	87.53%	84.00%	82.70%	696.23
4_6_18_3		0.005		[200,200,200,200,200]	100	90.61%	80.20%	81.10%	739.32
4_6_17_1		0.005		[150,150,150,150,150]	150	92.20%	84.10%	85.60%	786.05
4_6_17_2		0.005		[180,180,180,180,180]	150	92.97%	85.20%	85.70%	947.28
4_6_17_3		0.005		[180,180,180,180,180]	150	93.20%	81.00%	80.20%	990.40
4_6_16_1		0.007		[150,150,150,150,150]	150	90.13%	87.70%	87.00%	919.93
4_6_16_2		0.005		[150,150,150,150,150]	150	91.81%	85.10%	85.10%	837.77
4_6_16_3		0.005		[100,100,100,100,100]	200	92.55%	88.00%	86.20%	336.73
4_6_15_1		0.007	5	[200,200,200,200,200]	200	92.26%	90.10%	89.60%	1823.16
4_6_15_2		0.007		[200,200,200,200,200]	150	92.58%	88.90%	87.00%	1239.33
4_6_15_3		0.007		[150,150,150,150,150]	150	93.27%	87.80%	88.80%	866.92
4_6_14_1		0.005		[180,180,180,180,180]	150	95.44%	88.40%	87.90%	920.03
4_6_14_2		0.005		[200,200,200,200,200]	200	95.84%	87.50%	87.80%	1209.26
4_6_14_3	100,000	0.007		[200,200,200,200,200]	150	94.08%	89.80%	90.50%	1057.52
4_6_13_1				[150,150,150,150,150]	130	94.76%	90.70%	90.40%	559.45
4_6_13_2				[150,150,150,150,150]	150	95.80%	90.30%	92.20%	675.76
4_6_13_3				[160,160,160,160,160]	150	97.52%	90.70%	91.80%	869.24
4_6_12_1				[120,120,120,120,120]	130	95.10%	92.00%	89.10%	342.67
4_6_12_2			4	[180,180,180,180]	100	96.71%	93.30%	92.30%	486.90
4_6_12_3			4	[180,180,180,180]	100	97.06%	91.20%	90.80%	481.26
4_6_11_1			4	[130,130,130,130]	100	96.31%	91.40%	93.30%	298.35
4_6_11_2			4	[130,130,130,130]	100	97.48%	93.10%	94.40%	324.00
4_6_11_3			4	[130,130,130,130]	100	97.95%	93.90%	93.50%	289.39
4_6_10_1			3	[150,150,150]	100	96.76%	92.40%	91.80%	280.23
4_6_10_2			3	[150,150,150]	100	98.38%	94.10%	92.40%	255.68
4_6_10_3			4	[130,130,130,130]	100	99.10%	95.70%	95.70%	299.14
4_6_9_1			3	[150,150,150]	100	98.71%	95.30%	95.80%	388.73
4_6_9_2		0.005	3	[150,150,150]	100	98.88%	95.00%	95.70%	342.82
4_6_9_3			3	[150,150,150]	100	99.48%	97.20%	97.60%	277.22
4_6_8_1	200,000			[200,200]	100	98.61%	97.90%	97.30%	511.98
4_6_8_2	200,000			[200,200]	100	99.47%	99.00%	98.60%	529.30
4_6_8_3	200,000			[200,200]	100	99.89%	99.00%	99.80%	538.44
4_6_7_1	100,000			[150,150]	200	99.63%	98.00%	97.90%	487.62
4_6_7_2	100,000			[150,150]	100	99.66%	98.90%	98.50%	263.52
4_6_7_3	151,200			[200,200]	100	100.00%	---	---	537.02
4_6_6_1	86,400		2	[300,300]	200	100.00%	---	---	759.81
4_6_6_2	36,000			[200,200]	200	100.00%	---	---	188.09
4_6_6_3	10,800			[100,100]	200	100.00%	---	---	17.78
4_6_5_1	7,920			[150,150]	500	100.00%	---	---	71.16
4_6_5_2	2,880			[60,60]	500	100.00%	---	---	7.05
4_6_5_3	720			[30,30]	500	100.00%	---	---	1.44
4_6_4_1	756			[50,50]	1,000	100.00%	---	---	3.14
4_6_4_2	216			[30,30]	1,000	100.00%	---	---	1.21
4_6_4_3	36	0.0035	1	[15]	1,000	100.00%	---	---	0.43
4_6_3_1	72		1	[15]	1,000	100.00%	---	---	0.51
4_6_3_2	12		1	[5]	1,000	100.00%	---	---	0.43

Table A.6. Architecture and accuracy of each *ANN* from the Better-of-Two in large bay size

Name	Number of Data	Learning Rate	Number of Hidden Layer	Perceptrons	Epochs	Accuracy of Train	Accuracy of Valid_1	Accuracy of Valid_2	Computational time(s)
4_6_18_1					180	96.29%	86.50%	85.70%	890.49
4_6_18_2					210	95.78%	82.30%	84.30%	818.98
4_6_18_3					210	97.13%	86.70%	86.80%	736.68
4_6_17_1					180	96.47%	88.50%	88.10%	742.70
4_6_17_2		0.0035			180	95.00%	84.80%	84.50%	688.61
4_6_17_3					180	97.00%	87.50%	88.30%	696.16
4_6_16_1					150	95.92%	88.80%	86.50%	588.41
4_6_16_2					150	95.70%	85.20%	85.80%	582.07
4_6_16_3					180	97.13%	89.70%	89.50%	745.95
4_6_15_1		0.005			150	96.17%	93.10%	91.80%	602.46
4_6_15_2		0.005			150	95.55%	89.00%	88.00%	589.35
4_6_15_3		0.0035			150	97.21%	90.10%	90.50%	593.75
4_6_14_1					130	95.04%	89.70%	90.50%	504.31
4_6_14_2			4	[150,150,150,150]	150	96.49%	89.60%	90.80%	592.84
4_6_14_3	100,000				150	98.36%	94.00%	92.60%	562.88
4_6_13_1					100	96.33%	91.60%	91.70%	399.51
4_6_13_2					100	96.18%	89.40%	90.70%	398.54
4_6_13_3					130	97.61%	93.30%	92.20%	521.02
4_6_12_1					100	95.69%	92.70%	91.40%	434.34
4_6_12_2		0.005			100	96.65%	91.40%	90.80%	380.54
4_6_12_3					100	97.82%	92.90%	92.40%	363.39
4_6_11_1					100	95.05%	91.30%	91.20%	494.71
4_6_11_2					90	96.42%	92.60%	93.60%	372.28
4_6_11_3					90	95.57%	90.60%	89.60%	378.67
4_6_10_1					100	96.95%	92.40%	92.40%	416.54
4_6_10_2					100	96.26%	88.30%	89.00%	429.94
4_6_10_3					100	96.01%	89.20%	89.90%	393.40
4_6_9_1		0.0035	3	[200,200,200]	100	98.10%	91.20%	91.00%	399.93
4_6_9_2		0.0035	4	[150,150,150,150]	100	95.91%	87.40%	87.90%	392.32
4_6_9_3		0.005	4	[150,150,150,150]	100	94.63%	86.10%	87.70%	429.50
4_6_8_1	200,000	0.005	4	[100.100.100.100]	150	90.49%	89.60%	90.00%	360.89
4_6_8_2	200,000	0.005	4	[200,200,200,200]	200	83.17%	76.90%	76.20%	1899.35
4_6_8_3	200,000	0.0035	4	[250,250,250,250]	100	91.16%	82.30%	83.70%	1458.82