Solving Train Maintenance Scheduling Problem with Neural Networks and Tree Search

Version of October 10, 2018

Shijian Zhong

Solving Train Maintenance Scheduling Problem with Neural Networks and Tree Search

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Shijian Zhong
born in Jilin, People's Republic of China

Algorithmic Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Netherlands Spoorwegen
Laan van Puntenburg 100, 3511 ER
Utrecht, the Netherlands
www.ns.nl

©2018 Shijian Zhong. All rights reserved.

Solving Train Maintenance Scheduling Problem with Neural Networks and Tree Search

Author:	Shijian Zhong
Student id:	4550951
Email:	`zhongshj@hotmail.com`

Abstract

The Train Maintenance Scheduling Problem (TMSP) is a real-world problem that aims at complete maintenance tasks of trains by scheduling their activities on a service site. Common methods of constructing optimal solutions to this problem are difficult as the problem consists of several highly-related sub-problems. Currently, NS is using a local search algorithm[52] to provide solutions for the problem. However, it has several deficiencies such as solution randomness and lacking flexibility for rescheduling.

In this research, we investigated the applicability of sequential decision making and supervised learning for solving TMSP. First, we formulate the TMSP problem with a reactive sequential mechanism and define the state and action space. Next, we design a feature representation for states and come up with the best kind of neural network structure through comparisons. Then, we conduct experiments to compare several search strategies with the trained network as the heuristic and find the best one. Finally, we evaluate the solvability of our system and conclude that our approach has a certain capability for solving small-scale problems.

Keywords: Train Maintenance Scheduling Problem, Reactive Agent, Supervised Learning, Neural Networks, Tree Search.

Thesis Committee:

Chair:	Dr. S. E. Verwer, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. M. de Weerdt, Faculty EEMCS, TU Delft
Company supervisor:	Dr. W. J. Lee, NS Techniek
Committee Member:	Dr. J. C. van Gemert, Faculty EEMCS, TU Delft

Preface

After nine month efforts, with great happiness, I present this work as the fulfillment of my master's study.

This work cannot be accomplished without the help of many people. Thanks to my supervisor Sicco Verwer, for referring me to this internship and offering precious guidance in our weekly meeting. Thanks to my supervisor Mathijs de Weerdt, for giving feedback on both the work and the thesis, let me know the right way of doing research.

Thanks to Wan-Jui Lee, my supervisor at the company, for the fruitful discussions and suggestions through this work. Thanks to Bob Huisman, for offering the opportunity for this internship. Thanks to Demian de Ruijter, and other colleagues and students in the company, for kindly sharing their experience of working with railways.

Thanks to my friend Liang Guo and Kaixin Ding, for the lunch, beer, and talk we have together.

Thanks to my parents, for supporting my decision of this master's study.

Shijian Zhong
Rotterdam, the Netherlands
October 10, 2018

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The Dutch Railway System is one of the busiest railway systems in the world. To keep the trains to operate well and safely in this large and busy transportation system, routine maintenance for trains is required. For instance, trains need inspection and repair regularly to keep their normal operation. Also, to serve passengers with clean and comfortable carriages, trains also need regular cleaning for both inside and outside of the carriages. As the largest train service operator in the Netherlands, NS is responsible for a large part of the operation of Dutch Railway System. For Nedtrain, the sub-department of NS, the primary task is the regular maintenance for trains.

Train maintenance usually takes place in service sites, which are connected with neighbor stations by tracks. A typical service site consists of several facilities for certain types of maintenance and some additional tracks for parking and shunting moves. In a maintenance period, trains first arrive at the service site, then get desired maintenance, and finally depart at the required time. Due to limited service facilities and available tracks in the service site, the same kind of services cannot be performed in parallel for multiple trains but have to be processed one by one. Also, as some services have to be done on specific tracks, shunting moves are needed in between services for different trains. Therefore, proper scheduling is needed to organize trains' activities and make sure all trains could finish required services and depart on time.

1.2 Train Maintenance Scheduling Problem

The TMSP problem can be described with the following scenario. The service site consists of several tracks with specific connections. Some of the tracks are with maintenance facilities. Fig 1.1 shows the service site 'Kleine Binckhorst,' which is the one we work on in this thesis. In this service site, track 61 to 64 have several different maintenance facilities, and track 906a is the gate track where trains arrive and depart. Rest of the tracks are for parking

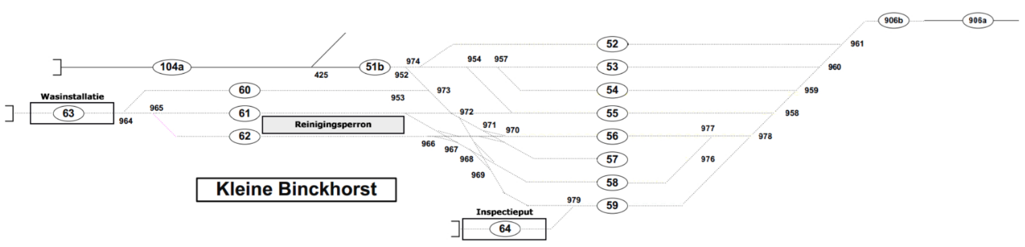


Figure 1.1: Service site 'Kleine Binckhorst', figure by van den Broek[52].

or shunting moves.

In a period, several trains come to the service site for maintenance and parking. Trains arrive at gate track at certain times. Each of them has certain services that need to be done on specific tracks. For the same material and service type, the duration required for maintenance is fixed. Processing maintenance means the train has to stay on the corresponding track for a certain period. Like arrival, departures are also required with fixed time. However, materials are required instead of exact trains, which offers some flexibility to select which train to depart for each departure event. Besides processing maintenance, trains also need to be parked in the service site during the stay in the service site.

1.2.1 Problem Instance

Table 1.1 describes an example problem instance of 3 trains in full context. Train 2000 (SLT-4) arrives first at the 00:20, with 10 minutes of internal cleaning and 23 minutes technical control B required. Then, train 1000 (VIRM-4) arrives at 00:30, requiring only a 23-minute internal cleaning. Train 3000 (VIRM-6) arrives at 01:00, with 35 minutes of internal cleaning and 14 minutes of technical control B needed. The departure section requires an SLT-4 to depart at 01:30, a VIRM-4 to depart at 01:40, and a VIRM-6 to depart at 02:10.

Train	Material	Arrive Time	Services (type/duration)
1000	VIRM-4	00:30	(1, 23min)
2000	SLT-4	00:20	(1, 10min), (5, 23min)
3000	VIRM-6	01:00	(1, 35min), (5, 14min)

Depart Time	Material
01:30	SLT-4
01:40	VIRM-4
02:10	VIRM-6

Table 1.1: An example problem instance.

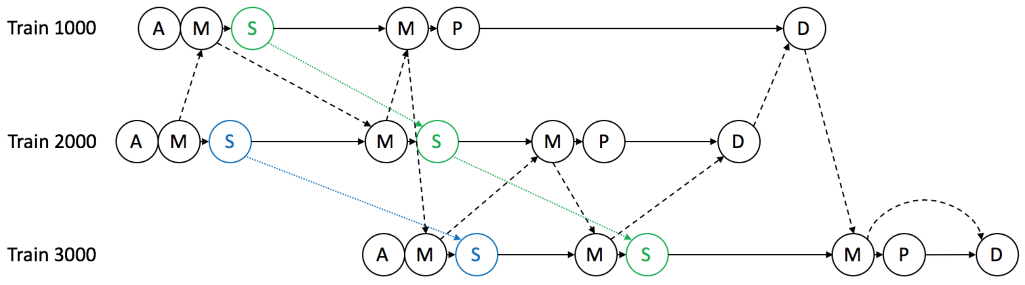


Figure 1.2: The activity graph of the solution.

This process consists of several sub-problems: maintenance scheduling, shunting, and routing. Maintenance scheduling can be summarized as organizing the orders of required services from trains' or facilities' perspective. Shunting aims at allocating trains in limited tracks in the required period. Routing act as a general constraint that has to be respected when solving the above two sub-problems. However, the problematic part of the TMSP is that these sub-problems are highly correlated and dependent on each other. A maintenance schedule can only be performed if the routing to service platforms can be done. Similarly, routing is only possible on non-occupied tracks, which requires careful shunting to provide enough flexibility.

1.2.2 Solution

A solution for TMSP can be described as an activity graph. Fig 1.2 is an example solution for the problem instance. In this activity graph, arrival (A), service (S), parking (P) movement (M), and departure (D) nodes are connected by arcs. Nodes are ordered in the chronological order from left to right. Green service nodes stand for internal cleaning, and blue service nodes mean technical control B. Solid, black arcs connect the activities for the same train. The dashed arcs show the order of movements. The dotted arcs determine the order of service tasks to perform. By processing these activities, all trains can get their required services done and depart as required.

1.3 The Local Search Algorithm

Train Maintenance Scheduling Problem (TMSP) used to get solved by expert schedulers at Nedtrain, but this is usually very heavy labors for the schedulers. Also, for involved problem instances that have many trains and services, solving them manually could be difficult. To reduce people's labor and enhance the capability for solving large-scale problems, researchers at Nedtrain have been trying to make an automatic planning algorithm for solving TMSP. At present, a local search algorithm[52] has been developed to automatically generates scheduling plans for TMSP. The local search algorithm has been developed with two desires. First, it generates solutions more quickly than human planners, which can assist real-world train maintenance scheduling. Moreover, since NS is going to expand its services and more trains will be running, current service sites are expected to serve more trains. Therefore, another intention of the local search algorithm is to investigate the capacity of service sites by testing the problem solvability with a different number of trains involved.

The local search algorithm transforms the TMSP into an optimization problem by relaxing several feasibility constraints. It first builds an initial solution, then searches through neighboring solutions of the current solution to get improvement. The initial solution is generated by the following procedure: First, the service schedule is made. Services are added one by one to each resource (service track). If multiple resources are available, the one with the currently smallest makespan is selected. For a single train, the order of different services is determined by their starting time. After obtaining the service schedule, parking event are assigned between service, arrival and departure events for each train. The exact tracks for parking events are randomly assigned to available tracks. Finally, paths for movements that connect parking, service, arrival and departure events are assigned according to the positions of these events.

After the initial solution is built, the algorithm iteratively performs local search in its neighboring solutions given the current solution. To make the local search procedure flexible, several constraints have been ignored: train collides, service delay, depart delay and so on. For each iteration, the local search procedure modifies the plan by randomly selecting among operators, go to a neighboring solution of this operator, and accept the current solution with the simulated annealing strategy on the cost function. We explain several modifications as follows:

- movement shift: shifting a movement earlier or later in the linear ordering imposed on the train movements.
- movement merges: merge two movements into one.
- parking reassignment: change the track and arrival side of one parking activity.
- parking swap: swapping the track and arrival side of two parking activities that overlap in time.
- parking repositions: split a parking event into two, with a movement in-between, and assign one of the parking events to a different track.
- resource order swap: swap two consecutive service tasks of the same service track.
- train order swap: swap two consecutive service tasks of the same train.
- resource reassignment: assign a service to a different suitable service track.

For each solution, a feasibility check is performed by checking crossing movements, delays to make sure it only returns feasible solutions. The iterations of the local search algorithm could run continuously, so two working status are provided: terminate as soon as a feasible solution is found, or terminate when given time runs out and return the best solution that has been found.

1.4 Problem Statement

The local search algorithm can effectively solve TMSP, but there are still deficiencies. The primary problem is it lacks flexibility for unexpected changes. Suppose an additional train outside of the plan comes for maintenance and shunting while the original plan is in progress, the local search algorithm is not able to adjust the current plan to make it feasible, but can only produce a new plan which is usually very different from the original one. Table 1.2 shows a simple example of plan reconstructing for unexpected incident. In the beginning, only one train with two services needs to be processed. However, an unexpected train arrives for one service at the moment that the plan is already partially executed. The local search algorithm is not able to fix the plan base on the current situation, but can only generate a new plan. In this example, the new plan arranges train 2000 before train 1000 for service 2, which is not feasible in reality since train 1000 is already processing service 2.

Current solution	**New solution**
Service(1000, 1)(finished)	Service(1000, 1)
Service(1000, 2)(processing)	Service(2000, 2)
	Service(1000, 2)

Table 1.2: Rebuild plan with the local search.

Besides, the local search algorithm also produces solutions with randomness. According to the previous section, the algorithm performs local swaps randomly to come up with new solutions from former ones. Therefore, given similar or even precisely the same problem instances, the algorithm could produce different solutions. These solutions are feasible ones, but the maintenance crew at the service site prefers consistent solutions to perform their jobs smoothly.

In recent years, due to the development in sensor technology, data storage, and computational power, vast amounts of data becomes available in many fields. Thaduri *et al.* illustrated the potential of using data technologies specifically in railway systems[51]. At the same time, machine learning also drew people's attention as a technique which utilizes data. Also, neural networks have become favorite techniques in both academy and industry these years. Due to its capability to generalize abstract patterns from historical data, it has been widely used in tasks like classification and regression and has made significant accomplishments in many fields such as images, audios, and languages.

Currently, no research directly uses machine learning in train maintenance scheduling problem. In Cordeau's survey[10], many approaches for train routing and scheduling problem use optimization models. Optimization models can produce high-quality solutions within short computation time, but building optimization models for complex problems with a large variety of constraints is not natural. On the other hand, there are researches on applying machine learning to some other scheduling problems. Shaw *et al.* used a decision tree to classify manufacturing patterns into appropriate dispatching rule in a manufacturing system[44]. Lee *et al.* has combined genetic algorithm with an induced decision tree to develop a job shop scheduling system[30]. Shiue *et al.* used an artificial neural network to select essential features for a C4.5 decision tree in manufacturing systems[45]. These researches have shown the potential of using machine learning on planning and scheduling problems. However, on the other hand, machine learning techniques are only applied at a somewhat abstract level in these studies. The optimization model is still the dominant part while the machine learning approach provides tiny aids such as heuristics.

Impressive data-driven decision-making research that plays the game of Go has been presented by Silver *et al.*[46]. In this research, deep convolutional neural networks are used by generalizing from a large amount of play by professional Go players and give instructions for the best move to take. With its great success, researchers at Nedtrain started to investigate the possibility of solving TMSP also in a sequential way. Similarly, the TMSP problem can be formulated into a chronological, sequential decision-making problem. The action predictor can be solved by supervised learning. Then the solution could be found by iteratively predicting the best action to take until the problem gets solved.

The sequential decision-making approach with machine learning could recompense the deficiencies of the local search we have mentioned. First, this approach is flexible for unexpected incidents. This is because sequential approach decomposes finding the solution into several action-predicting problems, then it can restart half-way by updating the current state with the unexpected change. As for consistency, the specific characteristic of machine learning models guarantee that same inputs always lead to the same outputs. Also, the neural network model is capable of generalizing similar states and perform consistent actions. Finally, the sequential approach is also expected to be fast. Though, training the policy network could be time-consuming, making predictions are almost instantly once trained. In reality, the model only needs to be trained once and could be used continuously.

In this research, we have explored the possibility of solving TMSP with the sequential approach and machine learning technique. For the machine learning model part, the intention is to investigate if there are common patterns in the statuses during maintenance that could be generalized and could be used to give appropriate next actions. Specifically, first training a machine learning model that predict actions from states. Next, using a solver agent to find feasible, complete solutions with the trained machine learning model as a heuristic.

1.5 Research Questions

The primary purpose of this research is to explore the possibility of solving TMSP with supervised learning. We have the following main research question:

Can we solve TMSP with supervised learning?

To structure this research, we have come up with three research questions. Each research question represents part of the research and is answered by the result of the research.

RQ1: How to formulate TMSP in a way that enables supervised learning?

TMSP problem is in continuous time domain where machine learning methods cannot be applied directly. To solve the TMSP problem with machine learning approaches, sequential modeling that transform TMSP into a state-action formula is needed.

RQ2: What kind of neural network architecture works best for TMSP?

Current researches use different neural network architecture to solve supervised learning problems. In researches with convolutional neural networks, the most popular convolutional kernels are 1*1, 3*3, and 5*5[20, 27, 48, 50]. This research question is trying to answer which neural network architecture has the best performance in TMSP. We conduct experiments among 1*1, 3*3 and 5*5 CNN, and also ordinary fully-connected network.

Performance evaluation could be the prediction accuracy of the model at the state level. Besides, we also concern about the computational expense when training.

RQ3: Using the policy network as a heuristic, what search strategy works best for TMSP?

The neural network model learns strategy and behavior in the training data. Therefore, the performance of the model largely depends on the quality of the training data. As we use solutions from local search to train the neural network, the performance could be limited.

To enhance the performance of finding solutions, a possible approach is to use the neural network as a heuristic to guide a search algorithm. In this way, the TMSP becomes a game-tree search problem that can be solved by searching for a target child node. In this research question, we investigate which search strategy works best for this specific problem. Specifically for game-tree search problem, we consider Greedy Randomized Adaptive Search Procedure (GRASP), A* Search, Monte-Carlo Tree Search (MCTS) as competitive search strategies. We conduct experiments to compare among them, focusing on solvability on problem instances and computational complexity to evaluate.

Finally, to answer the primary research question, we evaluate the performance of our system by solving unseen porblem instances. We also compare our system with the local search algorithm[52] that NS is currently using. We mainly focus on problem solvability as the criteria. However, we also assess their computational expenses.

1.6 Outline

This thesis is organized as follows. Chapter 2 includes a review of related works on machine learning, train scheduling, game playing, and search algorithms. From chapter 3 to chapter 5 we introduced the techniques we used to build the scheduling system. Individually, chapter 3 introduced the approach we used for modeling TMSP in sequential decision making. Then, a policy network architecture built with supervised learning is explained in chapter 4. Chapter 5 presented our approach for combining the policy network with search algorithms. In chapter 6, we evaluate our solver system by comparing with the local search algorithm. Finally, in chapter 7, we conclude this thesis and introduce future research perspective.

Chapter 2

Related Work

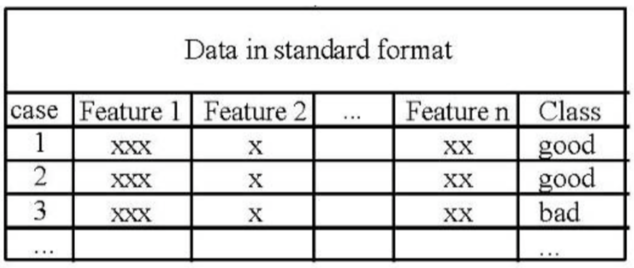
In this chapter, we present a literature study on topics related to our work. As this thesis presents an approach for solving TMSP using a combination of sequential decision making and machine learning, we focus on these topics: train maintenance scheduling, machine learning, and sequential decision making.

This literature study includes four sections. In the first section, we introduced deep learning techniques, including the mainstream neural network algorithms. Section 2 is a review of researches related to the train maintenance scheduling field. In this section, we present existing works on train maintenance scheduling and several similar problems with their methods. For the sequential decision making, our work learns from several studies in game playing, which is presented in the third section. Section 4 is a review of search algorithms which are commonly used for sequential decision making.

2.1 Machine Learning

Mitchell *et al.* has provided a widely quoted formal definition of machine learning: ”A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.”[33] This definition stresses the role of experiences as a way to improve the performance of a model, with the advantage of enhancing it automatically.

There are mainly two kinds of machine learning techniques: supervised learning and unsupervised learning. The two techniques are distinguished by whether the experience includes feedback. From the data’s perspective, a supervised learning model makes use of data with corresponding labels (see Fig 2.1). The learning algorithm can generalize relationships between them and become able to predict labels for new data. An unsupervised learning model utilizes unlabeled data and finds patterns such as similarity from data. From the task’s perspective, supervised learning algorithms can be used in classification and regression, depending on the labels are categorical or numerical. Unsupervised learning algorithms can



Data in standard format

case	Feature 1	Feature 2	...	Feature n	Class
1	xxx	x		xx	good
2	xxx	x		xx	good
3	xxx	x		xx	bad
...					...

Figure 2.1: Data format for supervised learning, figure by Kotsiantis[26].

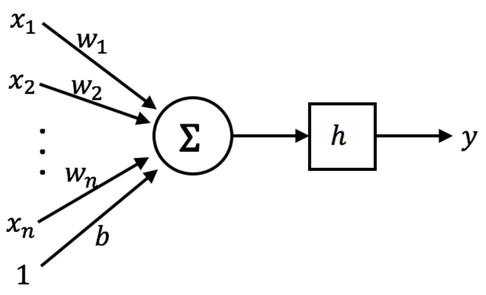


Figure 2.2: A single neuron.

perform tasks such as clustering, which organizes samples into subsets based on their similarity.

These years, reinforcement learning is also making success in several fields. A reinforcement learning model runs simulations, and obtain feedback from the environment as its experience to learn and improved.

2.1.1 Deep learning

Deep learning is a branch of machine learning that drew people's attention these years. It transforms input data into multiple levels of representation, with each level generalizing features into a higher and more abstract level[28].

Deep learning models consist of multiple neurons, which receives multiple inputs and produce an output. Fig 2.2 shows the computational details of a single neuron. x_i denotes the input of this neuron and y denotes the output. The neuron first performs a weighted sum of all inputs with $\sum_i w_i x_i + b$ where b is a bias factor. Then this weighted sum passes through an activation function to get the output $y = h(\sum_i w_i x_i + b)$.

Usually, non-linear functions are selected as activation functions. They introduces non-

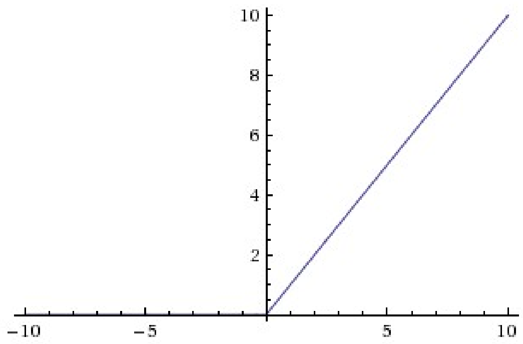


Figure 2.3: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. Figure by CS231n[31].

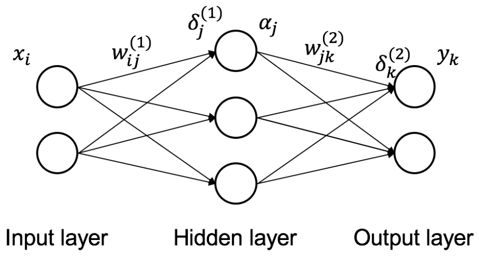


Figure 2.4: Feedforward neural network

linearity into the model which can greatly enhance the capability of generalization. Also, activation functions needs to be differentiable for computing gradients. There are several non-linear functions that have been used as activation function $h(x)$. For instance, Sigmoid($h(x) = \frac{1}{1+e^{-x}}$) and Tanh($h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$). But the most popular one at present is the rectified linear unit (ReLU) ($h(x) = max(0,x)$) since it learns faster than other activation functions[17]. Fig 2.3 shows the ReLU function.

A typical kind of deep learning model is feedforward network. In a feedforward network, computations are processed one way and staged among layers of neurons. Fig 2.4 shows a 2-layer (or 1-hidden-layer) feedforward network. In this network, the input x_i is first transformed with a linear transformation to be the input of the hidden layer: $\delta_j^{(1)} = \sum_i w_{ij} x_i$. In the hidden layer, $\delta_j^{(1)}$ passes a non-linear activation function: $\alpha_j = h(\delta_j^{(1)})$, and becomes the outputs of the hidden layer. Then, another linear transformation passes these outputs to the output layer: $\delta_k^{(2)} = \sum_j w_{jk} \alpha_j$. Finally through the activation function, the output of this network is obtained by $y_k = h(\delta_k^{(2)})$.

Deep learning models benefit from the excessive number of layers. With sufficient layers, deep learning models can learn very complex functions. Deep learning models also succeed in the capability of processing raw data, while conventional machine learning usually requires careful feature engineering beforehand[28].

In supervised learning, a neural network model can be described as connections with adjustable weights w between layers. By training a neural network model f_w, we aim to find optimal weights that minimize the error between correct labels y and the predicted labels from the model $\hat{y} = f_w(x)$. Since these parameters cannot be solved directly due to the complex structure with non-linearity, gradient descent is usually used for finding the minimum.

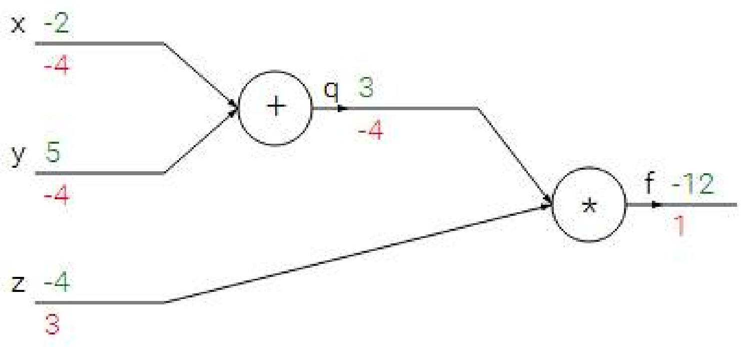


Figure 2.5: Backpropagation, figure by Li[31].

General gradient descent is an optimization method for finding the minimum of functions. Gradient represents the direction of the steepest drop in function value in the current position. For an example function $f(x)$, gradient descent start with a randomly assigned position x_0. Then for each iteration, gradient of current position $\nabla f(x_i)$ is computed, and the factor x is updated with a small step γ opposite to gradient direction $x_{i+1} = x_i - \gamma \nabla f(x_i)$. By processing this iteratively, the function value is expected to decrease gradually. This process stops when the loss stops decreasing, indicating that it has reached the minimum.

For training neural networks, the loss function with weights as factors is used as the objective function $\theta(w) = \sum_i L(y_i, f(x_i))$ to perform a gradient descent. Due to a large amount of training data, computing gradient with all training samples is not computationally feasible. In most researches, stochastic gradient descent or mini-batch gradient descent is used to handle this issue. For stochastic gradient descent, each iteration only one sample is selected from training data to compute gradient. Similarly, mini-batch gradient descent samples a subset. By going through all samples respectively and gradually, the estimated gradient has been proved to be converged to the gradient directly computed from all samples[5].

For multiple layer architectures, computing gradient of errors with respect to weights could be done by backpropagation algorithm[41]. Fig 2.5 shows how backpropagation is performed in a simple network. In this example, the objective is to solve $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$ given $x = 2$, $y = 5$ and $z = -4$. First, a forward propagation is performed to obtain values at each output (shown as green values). In the backpropagation stage, starting from the final output f, gradients with respect to its input q and z are first computed (shown as red values). Here $\frac{\partial f}{\partial q} = \frac{f}{q} = -4$ and $\frac{\partial f}{\partial z} = \frac{f}{z} = 3$. Then $\frac{\partial f}{\partial x}$ can be obtained with chain rule of derivatives $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4$, and same for $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4$.

With backpropagation, the gradient can be spread backward from output to input even in a complex network structure. After the gradients are obtained, gradient descent can be used for updating inputs.

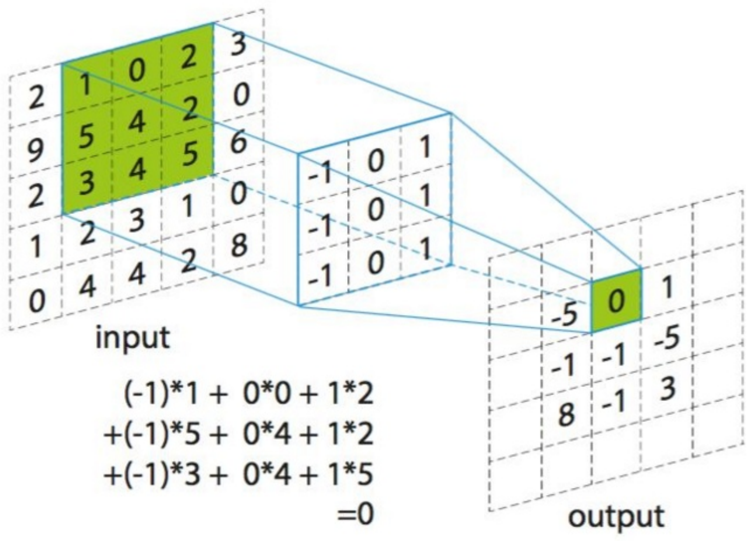


Figure 2.6: Image convolution.

There are several variants of feedforward network that has made many practical successes. Here we introduced a widely used structure: convolutional neural networks (CNN).

2.1.2 Convolutional neural networks

Convolutional neural networks use multiple arrays as input and generalize several local features with their convolutional layers[29]. Fig2.7 shows a typical convolutional neural network. In the beginning, there are several convolutional and pooling layers. In a convolutional layer, one or several convolutional kernels slide through the input feature maps. Fig 2.6 shows a convolution with only one filter. In each sliding position, a weighted sum of several neighboring pixels is computed as a pixel value of the next feature map. The procedure of sliding through feature maps is called convolution, and these weighted sums compose feature maps for the next layer. Pooling layers compute the maximum feature of a local patch and reduce the size of feature maps. Pooling is used to reduce the dimension of features and provide invariance to shifts and noises. After several convolutional and pooling layers, the feature maps are connected with several fully-connected layers for further transformation into labels.

Since CNN looks for local patterns and shares weights by conducting convolution with the same filters, it is suitable for handling euclidean domain data such as signals, images, and videos whose neighbors are strongly related. Also, with fewer parameters than fully-connected networks of the same scale, computation of training as well as predicting also gets more efficient.

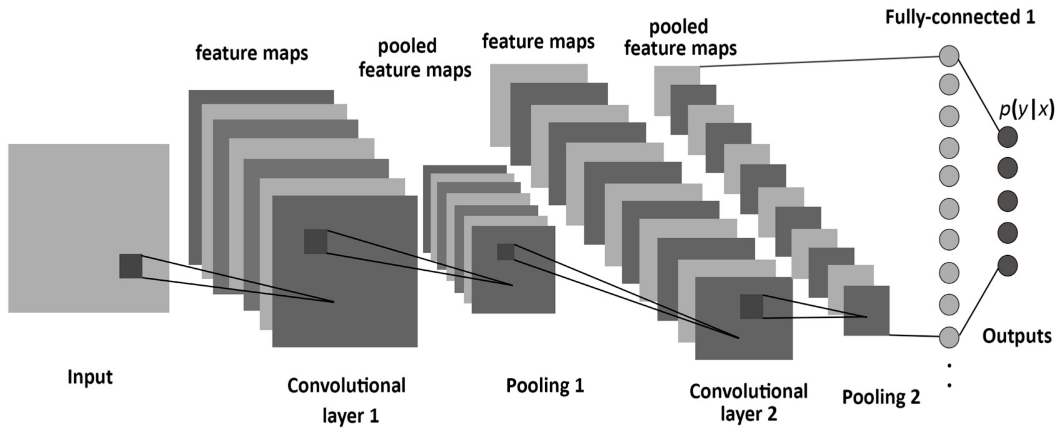


Figure 2.7: Convolutional neural network, figure by Albelwi *et al.*[3].

Convolutional neural networks have made significant accomplishments in many fields. Specifically for images, CNN can be used in detection, segmentation, and recognition. A prominent one is the image classification performed by Krizhevsky *et al.* on LSVRC-2010 ImageNet[27], which has reached an overwhelming performance compared to second-best. Also, in speech recognition, Sainath *et al.* proved that CNN outperforms ordinary deep neural networks and Gaussian mixture model in LVCSR tasks[43].

To conclude, neural network models have a prominent capability of generalization. They are also able to process raw features with not much feature engineering. In TMSP problem with the sequential approach, states could include structural and complex information. Actions also come from the complicate reasoning of features. Therefore, in this thesis, we trained a convolutional neural network as policy heuristic. We represent each state as feature maps, and the output is several available actions.

2.2 Train Maintenance Scheduling

Train Maintenance Scheduling Problem is a complex and specific problem. As mentioned in Chapter 1, TMSP consists of several highly-dependent sub-problems such as train maintenance, train shunting allocation. Therefore, we present a review of related studies concerning these sub-problems. In this section, we first present existing work on train scheduling concerning routes. Then, studies of train maintenance scheduling are shown. Finally, we also provide a review of several approaches to solving general scheduling problem with machine learning.

2.2.1 Train Scheduling

The primary constraint of train scheduling problem is track specification and their connections. Trains have to be on tracks all the time. When scheduling multiple trains in a railway network, track occupation also has to be considered.

Lots of studies used optimization techniques to solve the train scheduling problem. Cordeau *et al.* has separate this field into routing problem and scheduling problem[10]. Routing focuses on organizing train activities concerning railroad models, and scheduling concerns synchronizing the use of available tracks as resources. Cordeau *et al.* has also pointed out that early approach focused on linear programming and network optimization, but integer programming and heuristic methods become famous later on.

A constraint programming model for train scheduling through junctions is given by Rodriguez *et al.*[39], which is shown to have reached a significant improvement within an acceptable computation time. Ghoseiri *et al.* presented a multi-objective optimization approach for solving train-scheduling problem[16]. The work is performed on a scenario which is close to reality, consisting single, multiple tracks and multiple platforms. The objectives include lowering fuel consumption and shortening passenger-time. Caimi *et al.* has developed a Resource Tree Conflict Graph model that encodes tracks as resources[9]. Then maximal conflict cliques for each resource is determined and used in an integer linear programming formulation. D'Ariano *et al.* described a branch and bound algorithm for comprehensive real-time management of a complex railway network[13]. The model is shown to outperform commonly used dispatching rules.

Concerning the computational cost of optimization models, some researches also looked into heuristics. Higgins *et al.* compared several different heuristics for scheduling on a single line track and concluded that the genetic algorithm and hybrid algorithm have the best performance[22].

There are also studies about the complexity of train scheduling problem. Kroon's research has investigated the computational complexity of routing trains through railway stations[2]. They found that the problem could be considered as a Node Packing Problem and solved with integer linear programming.

2.2.2 Maintenance Scheduling

Maintenance is the major issue in TMSP. Purely train maintenance problem with no consideration of track occupation could be simplified as a job-shop scheduling problem.

Standard job-shop scheduling problem can be described as follows: Given n jobs $J_1, J_2, ..., J_n$ with different processing time. Each job has a set of operations $O_1, O_2, ..., O_n$ which needs to be done in a specific order. Each operation needs to be performed on a specific machine. Each job can only process one operation at a time. The objective is to minimize the makespan of the whole process.

In TMSP, trains can be seen as jobs. Operations are like services to be performed, and maintenance tracks are like machines. A relaxation of TMSP is that services do not need to be processed in a specific order. However, a restriction is that maintenance tracks are identical which can only handle one kind of service. For TMSP, each train has an objective of finishing services before departure time. However, maintenance has to be performed concerning several constraints such as track layouts, working schedule of crews or even energy consumption.

Higgins presented a model which finds the best allocation of maintenance activities and crews with tabu search heuristic[21]. The model optimizes the prioritized finishing time of each activity and the expected interference to scheduled trains.

Also, there are studies related to preventive maintenance, which estimate the necessity of performing maintenance in long-term. Research from Budai *et al.*[8] introduced a programming formulation as well as two heuristics for solving preventive maintenance scheduling problem on tracks.

2.2.3 General Scheduling With Machine Learning

Scheduling problems also appeared in other fields such as production and manufacturing. The standard part of these fields with train maintenance scheduling is that they arrange and allocate work and workloads to meet specific requirements. Scheduling is a difficult task since there could be complex constraints to follow, multiple and even conflicting objectives to be optimized and sometimes uncertainties to take into account.

Researchers have investigated the possibilities of combining machine learning with scheduling. In an early review from Aytug *et al.*[4], four kinds of approaches are summarized: rote learning, inductive reasoning, case-based reasoning, and neural networks.

Rote learning is a memorization technique based on repetition. A scheduling system from Bless *et al.* represented knowledge by frames[6]. The system select schedule by comparing the current state with existing frames. When solving a new problem instance, schedule with the same or closest frame is selected. In case no frame meets the requirement, a regular schedule will be executed. Obtained schedules are saved as new frames for further reference.

However, as rote learning memorize success decision through experiences, it could be challenging to find the same frame in complex problems whose state space is enormous. Another approach called inductive reasoning, which tries to generalize evidence from experiences. In this way, it can handle unseen instances better than rote learning.

Piramuthu *et al.* developed a decision support system with inductive learning capability in a flexible manufacturing system[1]. In this study, simulation is used to generate training examples. The decision support system selecting dispatching rules given a set of attributes that represent current status. The results show superior system performance through this framework and selected attributes.

Case-based reasoning is another machine learning technique that solves problems based on the solutions of previous similar problems. The critical difference with inductive reasoning is that inductive reasoning performs generalization before target problem is known, which is called the eager generalization. Case-based reasoning uses a strategy of lazy generalization that delays generalization of its cases until testing time. Cunningham *et al.* presented two CBR techniques for single machine scheduling problem[11]. These approaches are shown to provide good quality solutions.

The above studies show that combining machine learning with scheduling could obtain favorable outcomes in general. However, the exact approaches of incorporating them are usually using machine learning technique as external aids upon conventional scheduling algorithms. For example, in the case of predicting dispatching rules with system attributes, machine learning is used as a rough heuristic in the scheduling system. In this study, we will explore the possibility of using machine learning techniques on a more concrete level.

2.3 Game Playing

In this thesis, we considered the TMSP problem a sequential decision-making problem, which is similar to several existing studies on game playing. We have modeled the scheduling problem from real-time into a state-action formula. A neural network model is trained to predict one-step action given a current state. This section is a review of existing research regarding game playing. We introduce several games with the approaches researchers conducted. We also discuss utilities of these approaches with different characteristics of each game.

In game playing, a game is a closed environment with a set of specific rules where players can perform actions to update the environment. For each player, there is usually a goal that he/she aims to achieve.

2.3.1 Board Games

Unlike real-time games, the interactions between the environment and the players in board games are turn-based. In other word, actions cannot be processed synchronously. This

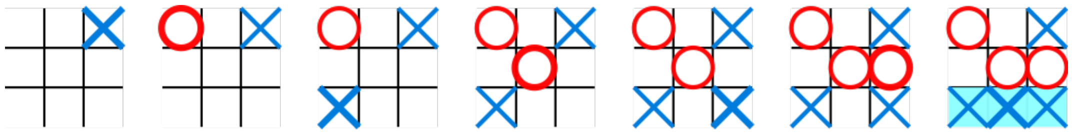


Figure 2.8: Tic-tac-toe, figure by Wikipedia[53].

makes it possible to model board games into sequential decision making. In a sequential modeling for a board game, we can denote $S = \{s_0, s_1, ...\}$ as state of the game, $A = \{a_0, a_1, ...\}$ as action and f as game rules. Then we have $s^{'} = f(s,a)$ as performing an action and going to a new state in this game.

Fig 2.8 shows Tic-tac-toe, a typical two-player competitive board game. In a tic-tac-toe game, two players (X and O) make spaces in a 3*3 grid in turn. The one who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game, and the other one loses.

Board games can be classified into several dimensions. A board game is said to be deterministic if the outcome of the actions can be determined with no stochastic influences. Otherwise, it is said to be stochastic. A game is said to be perfect information if the whole environment is observable to all players. Otherwise, it is imperfect information. Table 2.1 shows example board games of these classes.

	Deterministic	**Stochastic**
Perfect information	Chess, Go	Backgammon
Imperfect information	Texas Hold'em	Scrabble

Table 2.1: Board games classification

There are many studies of building intelligent players for board games. Specifically for deterministic, perfect information case, game-tree search is commonly used. A game tree consists of nodes that represent states, and each node has children that represent possible next states. Then, a search algorithm performs a search in the game tree from the current state to terminal states and obtains playing strategies. Fig 2.9 shows a game tree for tic-tac-toe. Red number shows the evaluation from search regarding player X.

However, purely search agent may suffer from complexity problem, especially for some board games with massive search space[36]. For instance, the game of Go has an enormous state space of 10^{170} possible positions[32], which is not possible to search thoroughly with current computational power. To solve this issue, researchers come up with efficient tree search algorithms that could find strategies while not necessary to search through the whole tree. For instance, Alpha-Beta pruning for two-player adversarial games prunes a subtree when it is found to be not promising[24]. Monte-Carlo tree search performs multiple ran-

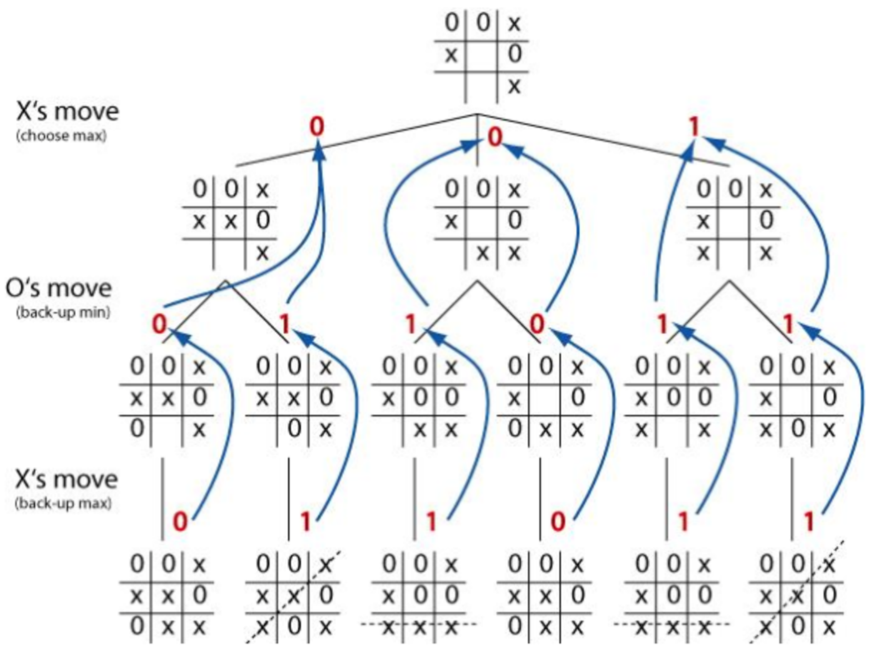


Figure 2.9: Partial game tree for tic-tac-toe, figure by Elnaggar[14].

dom rollouts and gets strategies from this sampled outcomes of the game tree[7].

On the other hand, several studies use supervised learning to learn from existing strategies and predict moves for new games. Groshev *et al.* used a deep neural network to learn a reactive policy from existing plans[18]. The experiment on Sokoban suggested that deep neural networks can extract powerful features from observations. For the game of Go, Maddison *et al.* trained a 12-layer convolutional neural network from human professional games[32]. With no search, it beats a traditional search program *GnuGo* in 97% of games. This research shows us the potential of applying the learning in game playing.

There are also studies that combine search and machine learning. These studies use a trained machine learning model as heuristics to guide a search algorithm. As the heuristics are given, the search space of the game states can be significantly reduced. Silver *et al.*[46] has presented an impressive Go agent $AlphaGo$. It consists of two trained machine learning models: a policy network that gives action probability for states, a value network that evaluate the utility of states. These two heuristics are used to guide a Monte-Carlo tree search from breadth and depth perspective respectively. It had reached 99.8% winning rate against other Go agents and has defeated the human European Go world champion by 5 games to 0.

Learning and search are both complimentary with each other. On the one hand, learning a heuristic can reduce the search space, which releases the complexity issue of search algorithms. On the other hand, the performance of a data-driven supervised learning model strongly relies on the quality of training data. The trained model could make mistakes especially when training data is not perfect. A search mechanism will offer chances to get to promising actions that the trained model ignored.

2.3.2 Real-Time Games

The real-time game is another type of game that is played in a continuous time domain. In a real-time game environment, players also perform actions that change the current status of the environment. Ontañ *et al.* on has summarized the main difference between real-time games and board games[37]. Real-time games are simultaneous action games, where several actions can be executed at the same time. Actions could also be durative instead of instantaneous. With these aspects, there is no discernible state mechanism in this continuous time domain. Also, real-time games have huge state space and action number comparing to board games.

Standard game playing approaches, such as game-tree search, is not directly applicable for real-time cases. Appropriate state modeling is needed for building player agents. A commonly used state modeling approach specifically for video games is to sample continuous time with equal and sufficiently tiny intervals. Sufficiently small sampling intervals guarantee sufficient flexibility for agents to perform actions. In several studies on Atari games, video frames are used as sampling intervals[34][35]. Actions could be performed in each frame, or each k frames to reduce the state space. In Atari games, the video frequency is 60Hz, which means each second consists of 60 states and an agent can perform 60 actions at most during this time.

Specifically for video games, modeling states with frames would be appropriate. This is because video games usually require actions to be performed rather frequently. Also, action space for video games is usually fixed and limited. However, for those games with large action space, sampling with frames could result in enormous problem complexity. For those games that do not require frequently performing actions, modeling states with uniform sampling could be a waste.

There is another state mechanism called Reactive (Reflex) Agent. A reactive agent usually has a set of condition-action rules and sensors which can keep track of the changes in the environment[42]. Fig 2.10 shows a framework of simple reactive agent. In this framework, detected changes in the environment act as triggers for these action rules. Since that, actions will be performed only when there is a change in the environment. In real-time games, this reactive planning strategy results in a brief state sampling.

Several studies used reflex agent for Starcraft – a real-time strategy game (RTS)[37]. In RTS, players need to build an economy (gathering resources and building a base) and military power (training units and researching technologies) to defeat their opponents (destroy-

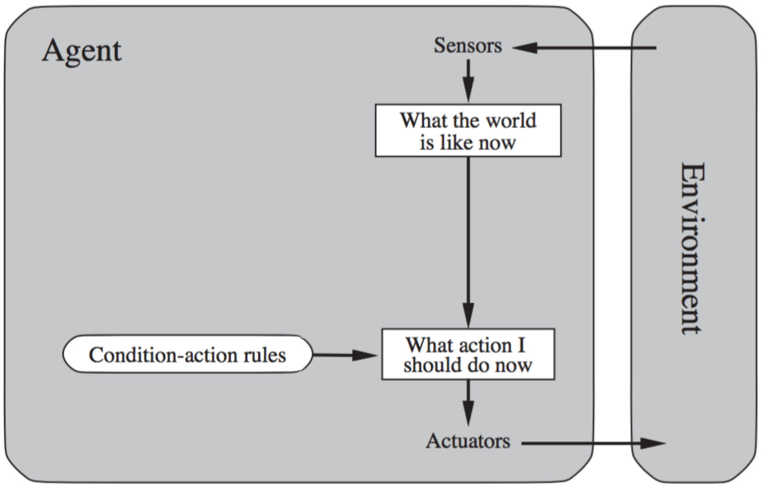


Figure 2.10: Framework of simple reactive agent, figure by Russell[42].

ing their army and base). Unlike Atari games, the action number of Starcraft could be large and highly depending on the state's property (e.g., how many troops the agent have at that moment). Starcraft also does not require frequently taking actions since most of its actions are durative and there will be much idle time.

To conclude, these existing researches offer us insights in the following perspectives: The TMSP problem can be solved by incorporating sequential modeling with machine learning and tree search. As a real-time planning problem, a reactive agent could be used to reduce complexity.

2.4 Search Algorithms

2.4.1 Greedy Randomized Adaptive Search Procedure

The Greedy Randomized Adaptive Search Procedure (GRASP) is an iterative randomized sampling approach for combinatorial optimization problem[15]. Fig 2.11 shows the pseudo-code of general GRASP metaheuristic. In each iteration, it consists of two phases: construction and local search. In the construction phase, the algorithm quickly constructs an initial solution via an adaptive randomized greedy function. Then in the local search phase, the algorithm search for the local optimal solution given the initial solution obtained from the construction phase. From each iteration, the GRASP obtains a locally optimal solution. By iteratively performing these two phases, the algorithm updates a better local optimal solution iteratively and finally return the best solution found.

```
procedure GRASP(Max_Iterations,Seed)
1    Read_Input();
2    for $k = 1, \ldots ,$ Max_Iterations do
3        Solution ← Greedy_Randomized_Construction(Seed);
4        Solution ← Local_Search(Solution);
5        Update_Solution(Solution,Best_Solution);
6    end;
7    return Best_Solution;
end GRASP.
```

```
procedure Greedy_Randomized_Construction(Seed)
1    Solution ← $\emptyset$;
2    Evaluate the incremental costs of the candidate elements;
3    while Solution is not a complete solution do
4        Build the restricted candidate list (RCL);
5        Select an element $s$ from the RCL at random;
6        Solution ← Solution $\cup \{s\}$;
7        Reevaluate the incremental costs;
8    end;
9    return Solution;
end Greedy_Randomized_Construction.
```

```
procedure Local_Search(Solution)
1    while Solution is not locally optimal do
2        Find $s' \in N$(Solution) with $f(s') < f$(Solution);
3        Solution ← $s'$;
4    end;
5    return Solution;
end Local_Search.
```

Figure 2.11: Pseudo-code of the GRASP, figure by Resende et al[38].

2.4.2 A* Search

A* Search algorithm[19] is a graph search algorithm that has been widely used in pathfinding and graph traversal problems. A* Search uses the following node evaluation function:

$$f(n) = g(n) + h(n)$$

The evaluation function has two parts. $g(n)$ represents the cost of the path from the starting node to n. $h(n)$ is a heuristic function that estimates the distance from n to the terminal node. When performing the A* Search, the algorithm maintains a priority queue to select the minimum cost nodes to expand repeatedly. For each expansion, the node with a minimum evaluation $f(n)$ is removed, with its neighboring nodes added to the priority queue. The algorithm performs this iteratively until a goal node has the lowest evaluation in the

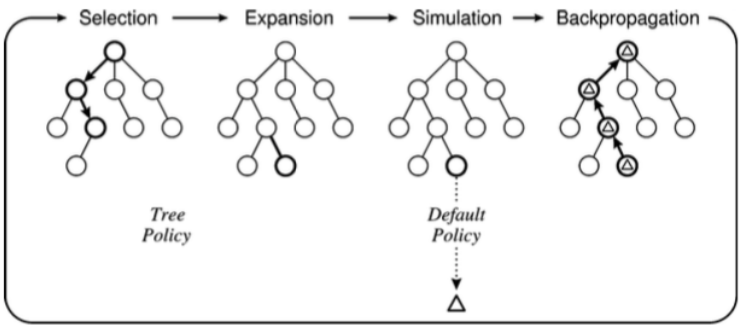


Figure 2.12: General MCTS (one iteration), figure by Browne[7].

priority queue. The f value of the goal node is the cost of the shortest path.

2.4.3 Monte-Carlo Tree Search

The Monte-Carlo tree search (MCTS) is a method for finding optimal decisions in decision-making problems. It takes random samples in the whole decision space and builds a search tree according to sampled results. MCTS has been used in many artificial intelligence approaches, especially in games and planning problems which can be represented as trees of sequential decisions.

The basic MCTS process is simple. Starting with the node representing the current state as the root node, a tree is then expanded iteratively through an expansion policy. Four steps are applied per iteration:

- Selection: Starting from the root node, select a node in the tree that needs to be expanded. The selected node has to be a non-terminal state and has unexpanded children.
- Expansion: As a node is selected in the last step, find a child node and add it to the tree.
- Simulation: From the expanded node, run a rollout to reach a terminal node and obtain a result.
- Backpropagation: backpropagate the simulation result from the terminal node to the root node and update their statistics.

After sufficient iterations performed, an action for the root state is selected based on the values of its child nodes.

Fig 2.12 shows a single iteration for general MCTS. This kind of process offers MCTS many benefits. It does not require any domain knowledge as well as the evaluation of non-terminal states. Also, it is a statistical anytime algorithm. As more computation offered, the performance is generally better.

In Browne's review, modifications of the above process have been grouped into two policies: Tree Policy and Default Policy. Tree policy appends a new leaf node given the current tree. Default policy performs a rollout from a non-terminal state and obtains an evaluation.

Several pieces of research provided variants of MCTS with specific tree policy and default policy. The most important one is the Upper Confidence Bound 1 applied to trees (UCT)[25]. UCT is a specific selection strategy that works for the selection stage of MCTS, trying to balance between exploration and exploitation. In the selection stage, UCT select a node that maximizes the following formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$

In this formula, w_i stands for the number of wins for node i. n_i stands for the total visit count for node i. N is the total visit of the parent node. c is the hyper-parameter that balance between exploration and exploitation. Therefore, the first part of this formula prefers exploitation to child nodes that succeed more during previous iterations, but the second part gives a chance for less visited child nodes to encourage exploration.

Chapter 3

Sequential Modeling

Train Maintenance Scheduling Problem is in the continuous time domain, where sequential decision approaches cannot be used directly. Appropriate sequential modeling that transfers the TMSP into a state-action formula would make it possible to apply machine learning techniques on finding solutions.

In this chapter, we present a state-action modeling specifically for TMSP. This chapter is organized as follows. We first explain the TMSP by introducing the environment for TMSP, several constraints to be respected and the objective of solving it. Then we present our sequential modeling approach for the TMSP using reactive agent modeling.

3.1 Rules for Scheduling

Scheduling in the real world requires lots of details to be taken into account. For instance, the time required for performing a shunting move depends on track length, shunting path, switches on the way, and even different train type. Modeling all these aspects in detail would be complicated. NS uses an abstract and adjustable environment for local search to help scheduling and researches. In this part, we explain an ideal mechanism that we used considering the model complexity and its applicability in reality. This mechanism applies for the simulated solutions from local search as well as our research.

- Shunting moves take a fixed time of 3 minutes regardless of path. Here, 3 minute is a sufficient period for any shunting moves in this service site.
- Change direction does not take time. In reality, change direction is required when a train has to perform a shunting move to the opposite direction. The train driver has to walk to the other side of the train to start it. However, it requires modeling driver's position into state description, which will make states more complex. Considering it is a relatively short time, ignoring it will not significantly affect a plan.

- Shunting moves cannot be performed in parallel. Modeling paralleled shunting moves requires careful inspection on related paths to make sure there is no collide. By disallowing this, we only need to consider obstacles for shunting moves. Considering it is a relatively short time, disallowing this only sacrifice a little bit of planning flexibility.

With these assumptions, the environment for making plans gets simplified. However, this model can still be used directly for TMSP in the real world.

3.2 State Modeling

The goal of this thesis is to solve TMSP with machine learning approach. In other words, finding the feasible activity sequence as the local search algorithm has done. However, predicting the whole solution at one time is not feasible due to the enormous dimension of the whole sequence.

One possible way of using machine learning to find solutions is only to predict one-step action. This approach requires the whole planning procedure to be decomposed into state-action pairs, and then a machine learning model is trained with the state as features and action as labels.

There are two intentions of solving TMSP sequentially. For actual use, a sequential planning model offers the possibility of partial rescheduling. For instance, when unexpected arrival delays or additional service required during the procedure, the sequential planning approach can reschedule from the current state while the local search algorithm has to reschedule the whole plan. Besides, states with the same action may include specific patterns which could be generalized. By learning a machine learning model, an environment level knowledge specifically for TMSP domain can be figured out for further research.

3.2.1 Reactive Agent

TMSP is a real-time scenario, which means activities are performed in the continuous time domain, and some activities are even durative. In this scenario, states are not visible and the timestamp for the agent to take action is not known. We need to formulate the states with assumptions.

Many studies build intelligent agents in real-time scenarios. As illustrated in Chapter 2, several studies on video games used uniform state sampling that select frames as states. In a 60 Hz video game, the agent can perform up to 60 actions in 1 second. This approach is suitable for video games as actions are performed frequently in video games.

Specifically for TMSP, a possible uniform state sampling way would be building states for each minute or second. However, unlike video games, in TMSP most of the time the agent stays idle with no activity. Then solving TMSP does not need actions to be taken frequently. Sampling with minute can also result in large states for training as well as step number for finding solutions.

Another state modeling approach is building a reactive (reflex) agent. A reactive agent detects significant changes and performs actions reactively. With reactive agent modeling, states are sampled whenever changes are detected. Comparing with uniform sampling, the reactive agent samples less but at crucial moments during the process.
We argue that the reactive modeling is more appropriate for TMSP. On one hand, TMSP does not requires actions to be taken frequently. Most of the activities are durative such as movements, services, where the agent only needs to stay idle. On the other hand, TMSP contains apparent trigger activities such as arrival, departure for the agent to detect. For the TMSP, the reactive agent modeling can significantly reduce the state complexity while not sacrificing action flexibility. We present a visualized comparison between uniform sampling and reactive sampling in a two-train example.

Fig 3.1 compares the result of conducting state sampling with uniform sampling and reactive sampling in a simple case. The uniform sampling approach constructs states in every minute, and the reactive sampling only builds states when changes are detected in the environment (e.g., arrival, service is done). A brief evaluation of state quantity shows that the uniform sampling constructs 61 state-action pairs for this solution, with 55 of them have 'Idle' as their actions. On the other hand, the reactive sampling can represent this solution with only 10 state-action pairs (4 'Idle') without sacrificing flexibility. According to the above evaluation, we argue that for TMSP, reactive agent modeling outperforms uniform sampling considering state simplicity and agent flexibility.

3.2.2 Activity Details

In this part, we present our state modeling for TMSP with the reactive approach. We will first explain the modeling details, then give a short example solution under our mechanism.

Reactive modeling sample states when the environment changes. These changes include arrivals, departures, and terminations of durative actions (Move, Service, Depart). Besides, the moment when an action has just been taken can be also regarded as a change, which requires sampling as well. Finally, considering idle times when executing a solution plan, there should be an additional action 'Wait' that let the agent stay still.

First, all activities in TMSP are grouped into two classes: **Action** and **Trigger**, depending on they can be executed by the agent, or they happen spontaneously. Activities are modeled in different ways:

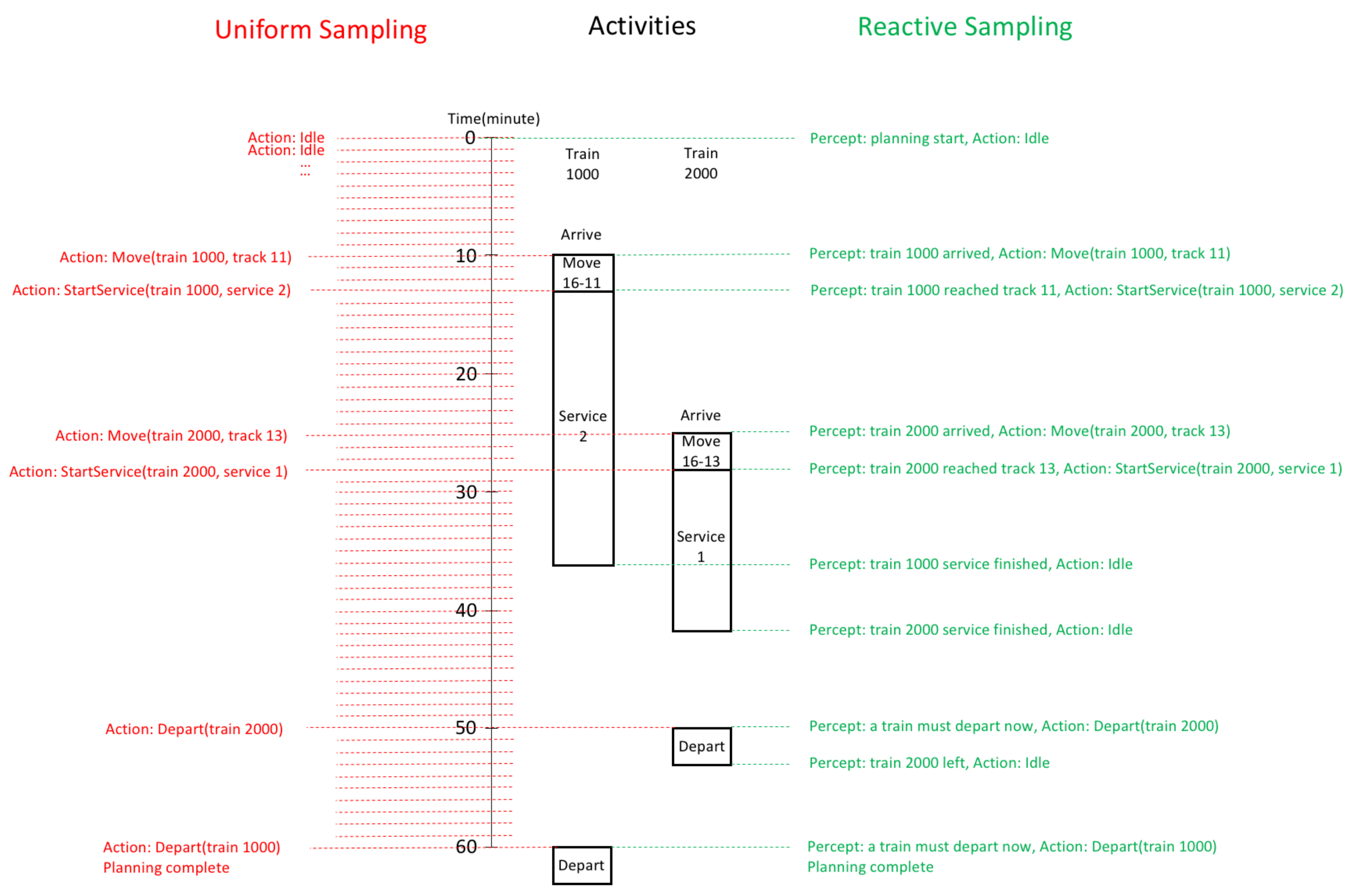

Uniform Sampling
Activities
Reactive Sampling
Time(minute)
Action: Idle
Action: Idle
Action: Move(train 1000, track 11)
Action: StartService(train 1000, service 2)
Action: Move(train 2000, track 13)
Action: StartService(train 2000, service 1)
Action: Depart(train 2000)
Action: Depart(train 1000)
Planning complete
Train 1000
Train 2000
Arrive
Move 16-11
Service 2
Move 16-13
Service 1
Depart
Percept: planning start, Action: Idle
Percept: train 1000 arrived, Action: Move(train 1000, track 11)
Percept: train 1000 reached track 11, Action: StartService(train 1000, service 2)
Percept: train 2000 arrived, Action: Move(train 2000, track 13)
Percept: train 2000 reached track 13, Action: StartService(train 2000, service 1)
Percept: train 1000 service finished, Action: Idle
Percept: train 2000 service finished, Action: Idle
Percept: a train must depart now, Action: Depart(train 2000)
Percept: train 2000 left, Action: Idle
Percept: a train must depart now, Action: Depart(train 1000)
Planning complete

Figure 3.1: Uniform sampling and reactive sampling in a two-train example.

Considering a move consumes relatively short time with disallowing lateral moves, it could be modeled as instantaneous action.

Different from moves, services take a relatively long time while other activities are allowed (usually necessary) in the meantime. Then, service is modeled with two parts: **StartService** as an action and **EndService** as a trigger. As the agent decided to start a service, its termination becomes a trigger in a certain future time.

Depart is a unique activity. There are fixed required departure time as triggers, but an action for selecting which train to depart needs to be taken by the agent. In this way, depart is also modeled with two parts: **Depart** as an action and **Departure** as a trigger.

Table 3.1 shows how actions and triggers correspond to former activities. Here we illustrate these actions and triggers in details:

Activity	**Action**	**Trigger**
Move	Move	-
Arrive	-	Arrival
Depart	Depart	Departure
Service	StartService	EndService
-	Wait	-

Table 3.1: Action and Trigger module

Actions are used for changing the current state into another state. Since actions are also used as labels for machine learning model, we tried to make their descriptions as concise as possible while making sure the next state $s^{'} = f(s,a)$ is always deterministic.

1) A Move action is described with **train** and **destination track**. It takes the train from current track to destination track and consumes 3 minutes. The exact position in destination track is determined by current track and track connection layout.

2) A Depart action is described with **train**. It removes the train from the service site and consumes 3 minutes.

3) A StartService action is described with **train** and **service type**. It makes two changes: It turns the status of that train from 'requiring service k' into 'processing service k,' and add a trigger EndService(train, time) with the same train as **train**, and the service required time as **time**. This action is finished instantly.

4) A Wait action has no further description. When it is performed, the agent stays idle until next coming trigger, then process it. The time it consumes depend on the time interval to the next trigger.

Triggers cannot be processed proactively, but are performed spontaneously when the time comes. Each trigger has a countdown timer indicating how much time left to its execution. In the state representation, a trigger list is maintained as the information of future events. After a trigger is processed, it is deleted from the trigger list.

1) An Arrival trigger is described with **train** and **time**. When it is processed, the corresponding train appears on the gate track.

2) A Departure trigger is described with **material** and **time**. Processing a departure trigger does not change the state, it is only for asking a depart action from the agent.

3) An EndService trigger is described with **train** and **time**. It can only be added to the trigger list by starting a service. When it is processed, the corresponding train deletes the status 'processing service k'.

3.2.3 State Description

Considering our case as a deterministic, perfect information game, state description should be sufficiently detailed. All information that might be used for predicting actions should be included in the state description. For our case, an action is mainly decided by current status in service site. However, future events should also be considered to make long-term plans. Therefore, in our state description, a state consists of three parts: service site status, trigger list, and remaining services. Table 3.2 shows an example state description.

Train	Material	Track	Position	In Service
1000	1	1	0	1
2000	2	2	0	-
3000	2	-	-	-

Trigger	Description
Arrival	train: 3000, time: 10
EndService	train: 1000, time: 20
Departure	material: 2, time: 30
Departure	material: 1, time: 40
Departure	material: 2, time: 50

Train	Service
2000	2
3000	1
3000	2

Table 3.2: State description

Service site status consists of several trains with their properties. In service site status, trains currently in and trains about to arrive are included, but departed trains are not. This is because departed trains no longer influence the current decision. Each train consists of 4 properties: 'Material' denotes the train type, related to service due time and departure requirements. 'Track' denotes which track the train is currently on. The 'Position' is used to distinguish orders of trains when they are on the same track. 'InService' denotes whether the train is processing maintenance and also what type of service.

Trigger list maintains all future triggers. Each trigger has its description, but all of them have a countdown timer indicating how soon it would happen.

Remaining services part consists of all undone services, described with train and service type.

Note that some information is not included in state description, for instance, track length, train length for specific material and service duration for specific material and service type. Since they are fixed for all states, they can be regarded as environment knowledge. They are not described explicitly as features but described implicitly in the existing solutions as these solutions follow these rules.

An example of a complete solution under our state modeling is described in Appendix A.

3.3 Discussion

Here we briefly summarize this chapter: For modeling the TMSP into sequential decision making, we have used the reactive agent modeling. The reactive agent modeling sample states only when significant changes are detected. In a brief evaluation according to state quantity and action flexibility, the reactive modeling uses less states without sacrificing the action flexibility compare to the uniform sampling.

Chapter 4

Policy Network

In this chapter, we present the neural network model part. We first introduce our feature engineering for states in TMSP, including several discussions and motivations. Then, we argue that the 1*1 convolutional kernel is the best architecture in theory for building the policy network. Finally, we conduct experiments on several mainstream neural network architectures to test our hypothesis.

4.1 Feature Engineering

From the previous chapter we know, a state in TMSP consist of structural and complex information. It contains both numerical (e.g., time, order) and categorical (e.g., material, track) information with a structural organization. To train a neural network model, appropriate feature engineering is required.

An existing study by Lu[12] that used machine learning techniques in TMSP has shown us an example of feature engineering. This study used several aggregated features to predict the difficulty level of problem instances. However, as we are predicting detailed actions, aggregated features are not discriminant enough for the neural network to generalize. In the following part, we first explain our considerations on performing feature engineering for states. Then we present our feature engineering approaches.

4.1.1 Considerations

When representing states into features, there is a trade-off between having detail feature representation or an abstract one. From the machine learning perspective, a feature representation with less information is preferred since it reduces the state space. However, the feature engineering has to be enough detail so that the network can at least distinguish significantly different states.

In our feature engineering, we believe that an appropriate level of details would be: if some similar states require different actions, their features should be discriminant for the machine learning model to learn and generalize with different actions. Within this prerequisite, try to make a concise feature representation that includes minimum details to reduce the state space.

We have also considered symmetry of state description. Symmetry means one real-world status can have several different state descriptions. An example in TMSP is the train id. Table 4.1 is an example of representing a real-world status with two state descriptions by switching their train id.

State A

Train	Material	Track	Position	In Service
1000	1	-	-	-
2000	2	-	-	-

Trigger	Description
Arrival	train: 1000, time: 10
Arrival	train: 2000, time: 25
Departure	material: 2, time: 50
Departure	material: 1, time: 60

Train	Service
1000	2
2000	1

State B

Train	Material	Track	Position	In Service
1000	2	-	-	-
2000	1	-	-	-

Trigger	Description
Arrival	train: 2000, time: 10
Arrival	train: 1000, time: 25
Departure	material: 2, time: 50
Departure	material: 1, time: 60

Train	Service
1000	1
2000	2

Table 4.1: Example of different state description for the same real-world status

The problem of symmetry when training a machine learning model would be sparsity, where the state space gets unnecessarily larger than the scenario space in real-world. There are two possible solutions to this problem. One is to perform data augmentation for the training set. The data augmentation checks every training sample, create all possible symmetric samples, and include them all into the training set. For instance, in previous work by Maddison *et al.* on playing the game of Go, the training set is sampled from 8 possible augmentations (rotate, reverse) of the states in original training set[32]. Another way is to design a feature engineering that can merge different states of the same scenario into the same feature set. We use this approach in our work.

Besides, some information that relates to the environment is not described in the state description. This includes track connectivity, track length and service duration for each material, and so on. They are the domain knowledge of TMSP and would be the same for all the states, so we have not explicitly described them in either state description and feature representation. However, since the solutions from training set obey this external knowledge, they can implicitly affect the machine learning model while training. For instance, if none of the tracks have held three trains in training data (which may exceed the track length), the trained model is not likely to perform this action even if it does not know the track length.

4.1.2 Feature maps

Based on the above considerations, we come up with the following feature maps.

We consider that all of the content in the state description is related to either a train or a track, or both of them. Then, feature maps can use train and track as two indexes, with the content encoded sparsely. The size of a feature map is $M * N$, where M is the maximum train number involved and N is the maximum track number involved.

For different kinds of information in state description, we encode them on different feature maps. We use 13 feature maps in total. Each feature map describes one kind of train-track oriented feature. They are summarized in Table 4.2.

Feature	**# of planes**	**Description**
Position	1	Current position of trains
Reachability	1	Tracks that trains can reach
Service	5	Service type 1-5, whether the train requires / where it can process
In service	2	Whether the train is in service / countdown
Arrival	1	Train’s arrival countdown
Departure	3	Whether the train can depart / next countdown / last countdown

Table 4.2: Summary of input feature maps for neural networks

Here we illustrate these features in detail:

- Position: For each train and track, if the train is currently on this track, the value is 1, otherwise 0.
- Reachability: For each train and track, considering track connectivity and obstacles of other trains, if the train can reach the track within one move, then the value is 1, otherwise 0.

- Service: This part has 5 feature maps that correspond to 5 kinds of service. For each train, if it requires this service, then the tracks on which the service can be processed are encoded as 1.
- In service: This part has 2 feature maps: If the train is currently processing a service, then in one map all tracks of this train is encoded as one, in the other map the same part is encoded as the countdown time.
- Arrival: For each train, if it is not arrived yet, encode all tracks of this train as the countdown time, otherwise 0.
- Departure: Considering departure requires only the certain material type, each train may have one or multiple chances to depart. We use 3 feature maps here: A binary map that encodes the train as 1 if it can depart at the current state. Two countdown maps that encode the countdown for the train's next departure chance and last departure chance. All of the 3 feature maps are encoded with all tracks together.

Specifically, when processing features that relate to countdown time, encoding exact values may be inappropriate as countdown times can be integers that range from 0 to ∞. Normalization is needed for maintaining the same scale among different features and benefit the convergence of gradient descent.

Since we care about both of the absolute magnitudes and relative orders of countdown time, we use a negative exponential function $f(t) = e^{-\lambda t}$ for normalization. λ is a hyperparameter for adapting the magnitude level of the input. This normalization scales time values into [0, 1) while preserving their orders and unifying their magnitude among different states. It also makes the value more sensitive around small values, which in reality make sense that we would like our agent to be sensitive for urgent events. For instance, when selecting the current action, whether an arrival event is after 90 or 100 minutes is less important than whether it is after 10 or 20 minutes.

Fig 4.1 shows feature maps from an example state. From these feature maps, we can know the following information of this state:

- Position: The first, second and third arrival trains are on track 10, 11 and 13 respectively.
- Reachability: The first arrival train can reach track 2-9, 13 and 16; the second arrival train can reach track 6-9, 13, 16; the third arrival train can reach track 6-12, 16.
- Service 1-5: The third and fourth arrival trains require service 1 on track 11-12; the third and fourth arrival trains require service 5 on track 2-10.
- In service: The third arrival train is currently in service. From the exponential representation, it will be done shortly.

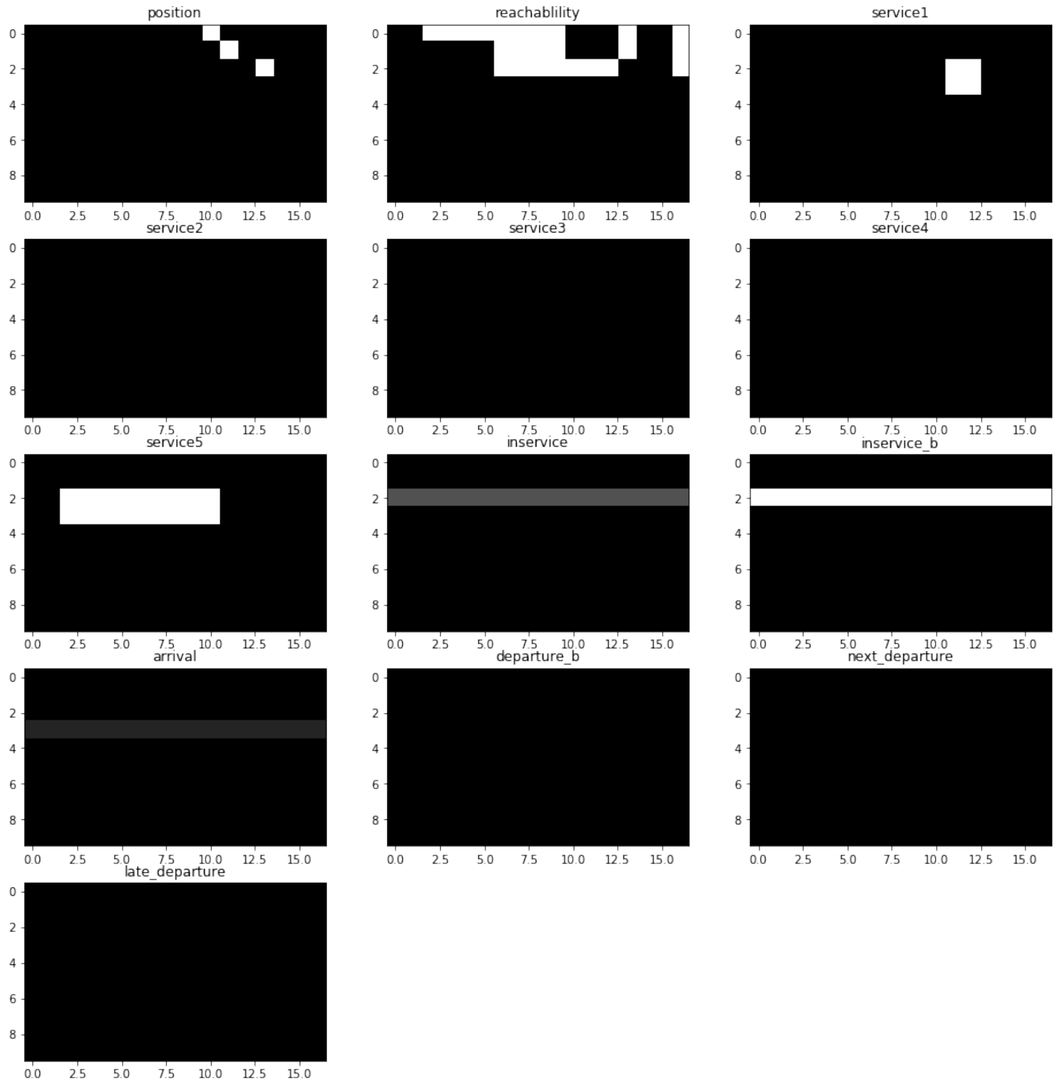


Figure 4.1: A set of feature maps from an example state.

- Arrival: The forth arrival train will arrive after a while.
- Departure: Currently no train is allowed to depart. The departure chances for all trains are still far from now.

From the above description regarding feature maps, we can see that they include most but not all of the information in state description. For example, if a train has 3 chances to depart in future, the feature maps only include the next one and the last one. However, we believe the information from our feature engineering is discriminative enough if different actions are expected.

4.2 1*1 Convolutional Kernel

Previous researches in convolutional neural networks often used convolutional kernels of different size, depending on how much neighboring area to percept in this layer. However, the 1*1 convolutional kernel does not percept neighbors at all. When convolving with a 1*1 kernel, each position in the resulting feature maps only depends on the same position in the previous feature maps. This process does not capture local patterns in feature maps. 1*1 convolution can be simply seen as conducting transformations among multiple feature maps. This approach has been used for implementing dimension reduction[49].

For our case, ordinary convolutional kernels larger than 1*1 are inappropriate. A state in train maintenance scheduling problem has a structural description which has no geometric formulation. Geometric neighbors in our feature maps are not geometrically related in reality. For instance, neighboring trains in feature map only mean their arrivals are right next to each other. Tracks are related as graphs in reality, and it is difficult to find their geometric relations. Given this kind of representation, ordinary 3*3 or 5*5 convolutional kernels that percept geometric neighbors do not make sense in theory for non-geometric feature maps we have.

On the other hand, when performing 1*1 convolution, every position is being filtered with the same kernels. Then, a 1*1 convolutional layer can also be regarded as a fully-connected layer with fewer connections and shared weights. This makes 1*1 CNN requires fewer parameters and less computation than the fully-connected network with the same scale. More importantly, our feature representation can make sure that a 1*1 kernel always percepts the same formula of information.

According to the above illustration, we have made the following hypothesis: CNN with kernels larger than 1*1 will not outperform CNN with 1*1 convolutional kernels.

4.3 Label Encoding

According to the action definition in Chapter 3 we know, given M trains and N tracks, we can have $M*(N+5+1)+1$ possible actions. Table 4.3 shows an example action space of 3 trains and 5 tracks.

	Train 1	Train 2	Train 3
Track 1	Move(train 1, track 1)	Move(train 2, track 1)	Move(train 3, track 1)
Track 2	Move(train 1, track 2)	Move(train 2, track 2)	Move(train 3, track 2)
Track 3	Move(train 1, track 3)	Move(train 2, track 3)	Move(train 3, track 3)
Track 4	Move(train 1, track 4)	Move(train 2, track 4)	Move(train 3, track 4)
Track 5	Move(train 1, track 5)	Move(train 2, track 5)	Move(train 3, track 5)
Service 1	StartService(train 1, service 1)	StartService(train 2, service 1)	StartService(train 3, service 1)
Service 2	StartService(train 1, service 2)	StartService(train 2, service 2)	StartService(train 3, service 2)
Service 3	StartService(train 1, service 3)	StartService(train 2, service 3)	StartService(train 3, service 3)
Service 4	StartService(train 1, service 4)	StartService(train 2, service 4)	StartService(train 3, service 4)
Service 5	StartService(train 1, service 5)	StartService(train 2, service 5)	StartService(train 3, service 5)
Depart	Depart(train 1)	Depart(train 2)	Depart(train 3)
Wait	Wait		

Table 4.3: Possible actions for 3 trains and 5 tracks

When assigning labels to output nodes of neural networks, a common approach is to assign one node for each possible action. However, in this case, the number of the possible actions has an exponential growth as the problem scale increase.

To reduce the number of nodes required for the output layer, we consider this problem as a multi-label classification. Each action is described with two sets of classification: train and activity. For train and activity classification, M nodes and $N+5+1$ nodes are needed respectively. Finally, the action Wait is not related to the train. We use an additional node for it. In this way, $M+N+5+1+1$ nodes can be sufficient to describe the whole action space. Specifically for this multi-label classification, we minimize the binary cross-entropy loss when training the neural network.

4.4 Experiment

From the last section, we come up with a hypothesis: For TMSP problem, 1*1 CNN is better than CNN with larger kernels and fully-connected network considering prediction performance. In this section, we experiment to test our hypothesis and answer our second research question.

4.4.1 Experiment Setup

We have experimented with two scenarios: small scale and large scale. The small-scale scenario has 4 trains, and the large-scale scenario has 14 trains. In both scenarios, we performed a 5-fold cross validation with 200 solutions generated by the local search algorithm[52] with the same setup. In the cross-validation, solutions are split into 5 folds uniformly and then decomposed into states based on the state modeling in Chapter 3. For each scenario, we split the solutions into 3:1:1 for training, validation, and testing. Table 4.4 shows the setup of the 5-fold cross-validation.

Iteration	Training Set	Validation Set	Test Set
1	Fold 1,2,3	Fold 4	Fold 5
2	Fold 2,3,4	Fold 5	Fold 1
3	Fold 3,4,5	Fold 1	Fold 2
4	Fold 4,5,1	Fold 2	Fold 3
5	Fold 5,1,2	Fold 3	Fold 4

Table 4.4: Fold setup for the cross validation.

Based on this scenario, we set the size of our feature map as 20*17, means there will be 17 tracks and maximal 20 trains. The number of the output node is 44, with 20 trains, 17 tracks, 5 services, 1 departure, and 1 wait. The time normalization factor λ is 0.05 to fit the scale of our time value.

We design the following network architectures:

1*1 CNN:
(1) A convolution of 16 filters of kernel size 1*1
(2) A rectifier nonlinearity
(3) A convolution of 32 filters of kernel size 1*1
(4) A rectifier nonlinearity
(5) A fully connected linear layer to a hidden layer of size 1024
(6) A rectifier nonlinearity
(7) A fully connected linear layer to the output layer of size 44

3*3 CNN:
(1) A convolution of 16 filters of kernel size 3*3 with padding 1
(2) A rectifier nonlinearity
(3) A convolution of 32 filters of kernel size 3*3 with padding 1
(4) A rectifier nonlinearity
(5) A fully connected linear layer to a hidden layer of size 1024
(6) A rectifier nonlinearity

(7) A fully connected linear layer to the output layer of size 44

5*5 CNN:
(1) A convolution of 16 filters of kernel size 5*5 with padding 2
(2) A rectifier nonlinearity
(3) A convolution of 32 filters of kernel size 5*5 with padding 2
(4) A rectifier nonlinearity
(5) A fully connected linear layer to a hidden layer of size 1024
(6) A rectifier nonlinearity
(7) A fully connected linear layer to the output layer of size 44

Fully connected network:
(1) A fully connected linear layer to the output layer of size 16*20*17
(2) A rectifier nonlinearity
(3) A fully connected linear layer to the output layer of size 30*20*17
(4) A rectifier nonlinearity
(5) A fully connected linear layer to a hidden layer of size 1024
(6) A rectifier nonlinearity
(7) A fully connected linear layer to the output layer of size 44

We make sure that these framework parameters have the same scale: All of these networks are 4-layer feedforward networks. Also, the number of nodes for each hidden layer, in other words, the capacity of information it could represent, are designed to be equal among these networks. From the perspective of representation learning, the only difference between these networks is the way it transforms features a current layer to the next layer (filtering with 1*1, 3*3, 5*5 convolutional kernels, or a complete linear transform). We consider our comparison to be fair under this setup.

According to the training issue, all of the networks are trained with the same hyper-parameters:

- Framework: PyTorch 0.3.1.post2, CUDA 8.0, GTX 1060
- Loss function: Binary cross-entropy loss (BCELogitLoss in PyTorch)
- Optimizer: Adam[23]
- Epoch: 50
- Learning rate: 1e-3, decay by 0.2 every 10 epoches
- Batch size: 100

As there is no valid research of tuning parameters for training neural networks, scientists usually select them by experience or existing successful results. We select commonly used training hyper-parameters. Specifically for the loss function, we use the binary cross-entropy loss function for multiple labels as output. For each architecture, we select the model with the lowest validation loss through training as the best model.

4.4.2 Prediction

Table 4.5 and 4.6 shows the testing error of these network architectures in both problem scales.

	1	2	3	4	5	Mean
1*1 CNN	0.2729	0.2619	0.2448	0.2500	0.2468	0.2553
3*3 CNN	0.2600	0.2458	0.2534	0.2541	0.2370	0.2500
5*5 CNN	0.2481	0.2411	0.2562	0.2452	0.2342	0.2450
Fully-connected	0.2639	0.2512	0.2484	0.2637	0.2623	0.2579

Table 4.5: Testing error of different models in the small-scale scenario.

	1	2	3	4	5	Mean
1*1 CNN	0.4887	0.4788	0.4817	0.4978	0.4774	0.4849
3*3 CNN	0.4976	0.4762	0.4791	0.4807	0.4775	0.4822
5*5 CNN	0.4820	0.4911	0.4707	0.4860	0.4803	0.4820
Fully-connected	0.5284	0.5191	0.5088	0.5419	0.5316	0.5260

Table 4.6: Testing error of different models in the large-scale scenario.

We can see in the small-scale scenario, the prediction accuracy among these networks does not differ significantly. In the large-scale scenario, the performance of convolutional models is close and slightly better than the fully-connected model.

4.4.3 Error Evaluation

Evaluating with only the prediction accuracy could be insufficient for comparing among these models. Models with similar prediction accuracy could have a different preference of prediction and make different kinds of mistake.

To compare these models thoroughly, we used two new evaluation metrics: class confusion matrix and error evaluation matrix. The class confusion matrix roughly evaluates the confusion among action class while ignoring mistakes from the action detail. We assign true labels in the y-axis and predictions in the x-axis. This metric shows us how a neural network model confuses among action classes.

The class confusion matrix could be insufficient for a thorough evaluation since it simply returns correct when inner-class errors are made. To provide more detail, we use the error evaluation matrix to describe inner-class errors. The error evaluation matrix classifies the predictions into 4 levels from mild to severe by the action class:

- correct: the prediction is totally correct, i.e. Move(train 1, track 3) - Move(train 1, track 3).
- train & class: the train and the action class is correctly predicted, but the action detail is wrong, i.e. Move(train 1, track 3) - Move(train 1, track 4).
- class: the action class is correctly predicted, but the train and the action detail are wrong, i.e. Move(train 1, track 3) - Move(train 2, track 4).
- wrong: the action class is wrong, i.e. Move(train 1, track 3) - StartService(train 2, service 3).

These levels intend that some kinds of incorrect prediction are not fatal in real planning. For instance, a **Move** prediction with the correct train but wrong destination track can still lead to solving. With this metric, we can see what kind of mistake does a model usually makes for each action class. Note that for **Depart** action, the first error (train & class) does not exist because a **Depart** action is described with the train only. This makes the second column of depart in the error evaluation matrix always 0. Similarly, the second and the third column for the wait action also remains 0 as the only possible error for the wait is confusing with other class.

Fig 4.2 and 4.3 shows the accumulated evaluation metrics from the testing result of 5-fold cross validation among networks in both scenarios. In the confusion matrix, the x-axis stands for predicted labels, and y-axis stands for true labels. In the error evaluation matrix, y-axis stands for true labels, and the x-axis stands for the error type a prediction lies in.

For the small-scale scenario, the class confusion matrices show that **Depart** has the least class error since in all of the models it reached 100% class accuracy. **StartService** and **Wait** also have very few class mistakes. However, the **Move** makes many mistakes. The most significant class error is predicting **Wait** upon the **Move**. On the other hand, the error evaluation matrices indicate that **Move** seldom gets correctly predicted. Beside confusing with other classes, the majority prediction picks the correct train but fails to predict the correct destination track. The performance of the other three action class is much better than the **Move** class.

For the large-scale scenario, more confusions could be found in the class confusion matrices. A large portion of **Move** is classified into **Wait**, with a small portion into **StartService**.

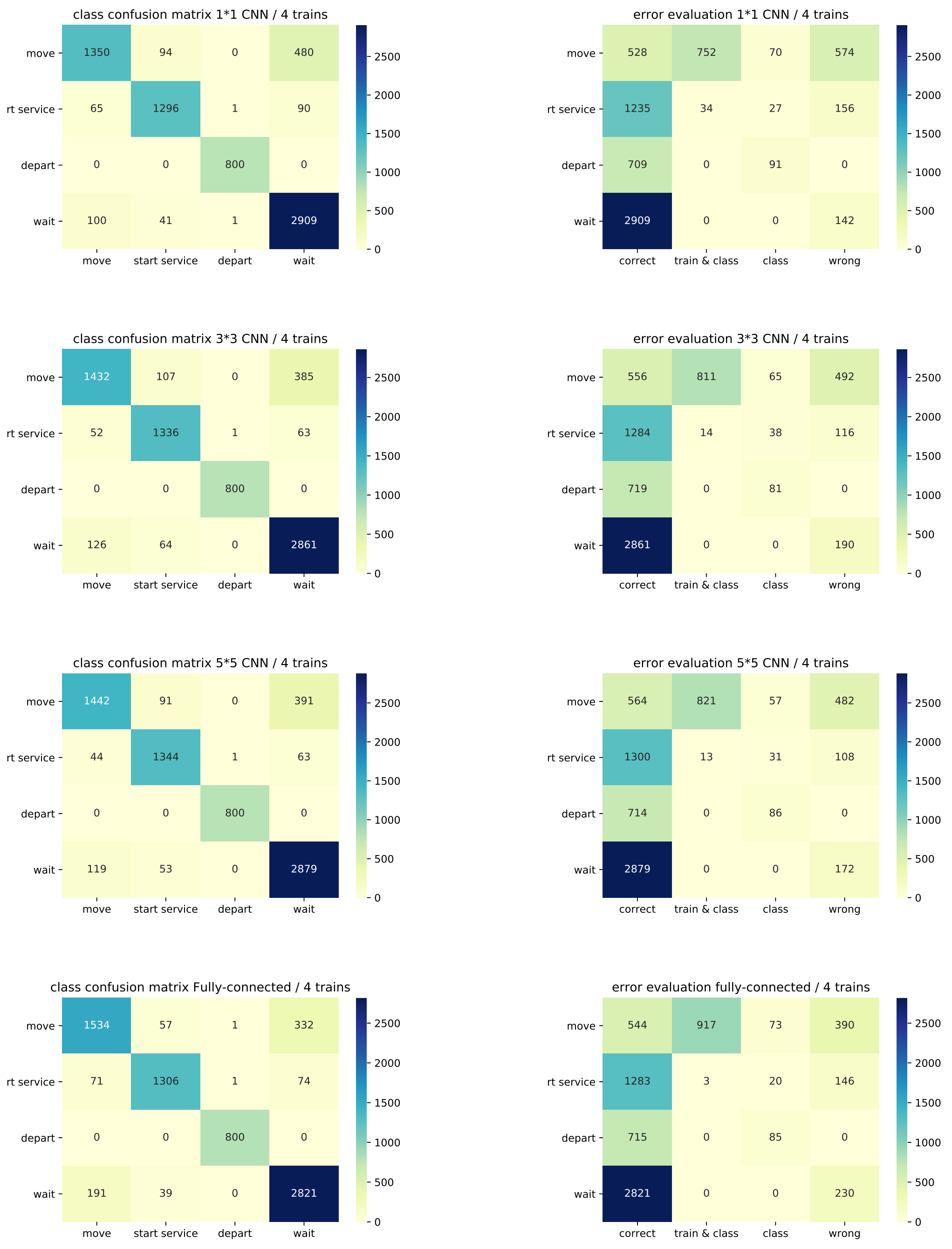


Figure 4.2: Class confusion matrix and error evaluation matrix of different network architectures in the small-scale scenario.

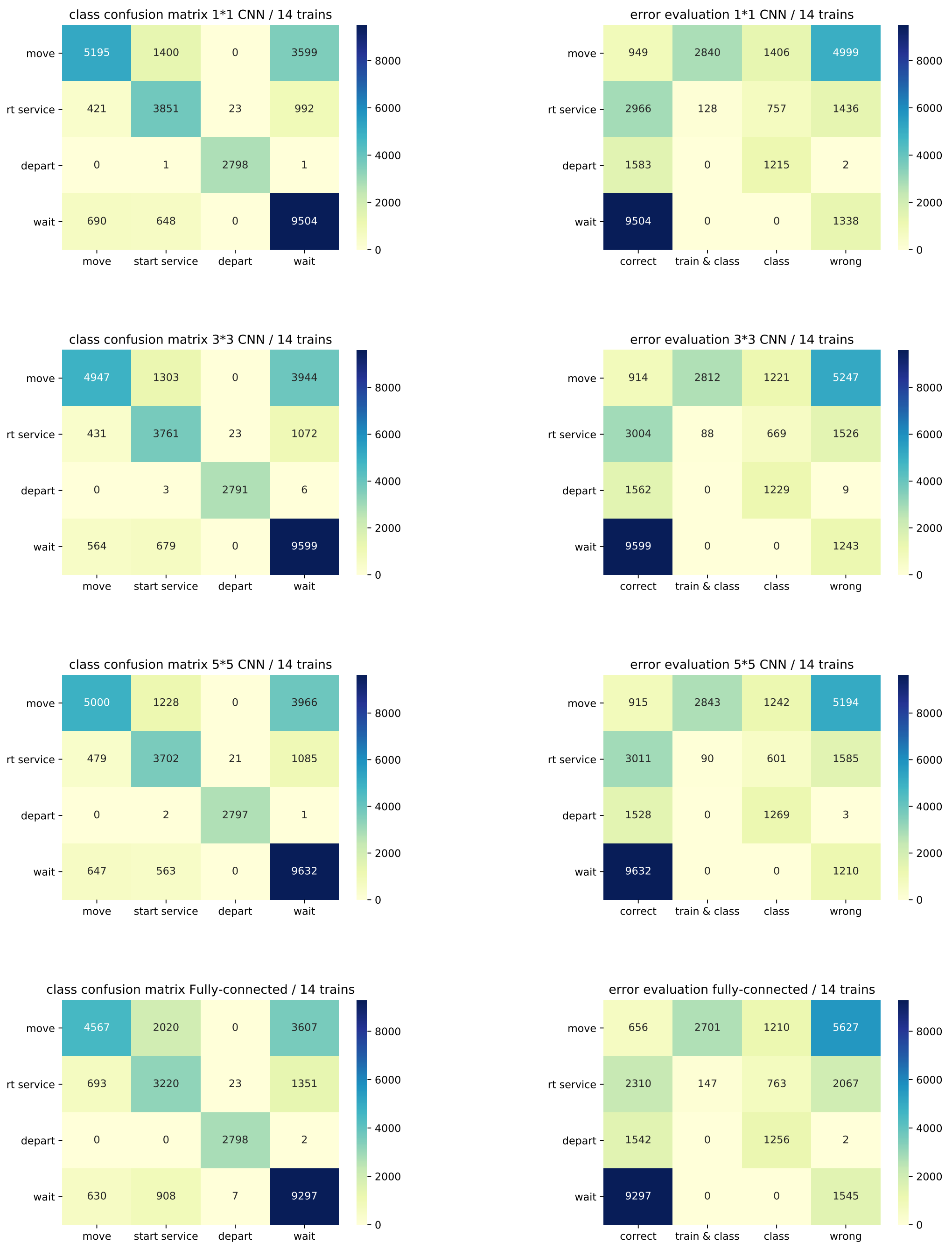


Figure 4.3: Class confusion matrix and error evaluation matrix of different network architectures in the large-scale scenario.

StartService is classified into the **Move** and **Wait** occasionally. **Wait** is also sometimes regarded as **Move** and **StartService**. **Depart** action have almost no class confusion error with other classes. For the error evaluation matrices, the main inner-class error is still in confusing destination track for **Move**. Another glaring error is confusing departure train for **Depart**. Generally, when extending the problem scale with more trains, the prediction performance of these networks gets worth.

Comparing among different network structures, the evaluation metrics do not reveal a significant difference in both scenarios.

4.4.4 Learning Curve

We have also evaluated the learning curves of these experiments to see the performance through training. Fig 4.4 and fig 4.5 show the learning curves of the fifth iteration from the cross-validation among different models in both of the scenarios.

Comparing among network structures, we can see that a more complex network can achieve lower training error. However, the validation error does not get reduced as the model get more complicated. Comparing among scenarios, we can see that the large-scale scenario have higher validation error but less training error than the small-scale scenario. The training error of 3*3 CNN, 5*5 CNN, and fully-connected network even reached 0, which means the current model complexity is capable of handling the problem complexity.

A preliminary conclusion of this result is that these network frameworks have almost the same performance in both of the scenarios.

4.4.5 Further Evaluation

According to the learning curve through training, significant overfitting could be observed. We come up with several hypotheses:

1. Concerning the state space: 200 instances are not sufficient to fill the state space for the TMSP. This result in significant bias between the training set and the validation set. A network that learns from insufficient training data has poor performance when predicting unseen new data.

2. Concerning the data quality: the previous evaluation metrics show that a large portion of inner-class error is confusing destination tracks to move actions. The hypothesis is that the solutions from the local search algorithm has randomness, and could be even more random at the state level, especially for move actions.

We perform two additional evaluation, trying to give insight into these two hypotheses. For the first hypothesis, we conduct experiments on how the amount of training data affect the

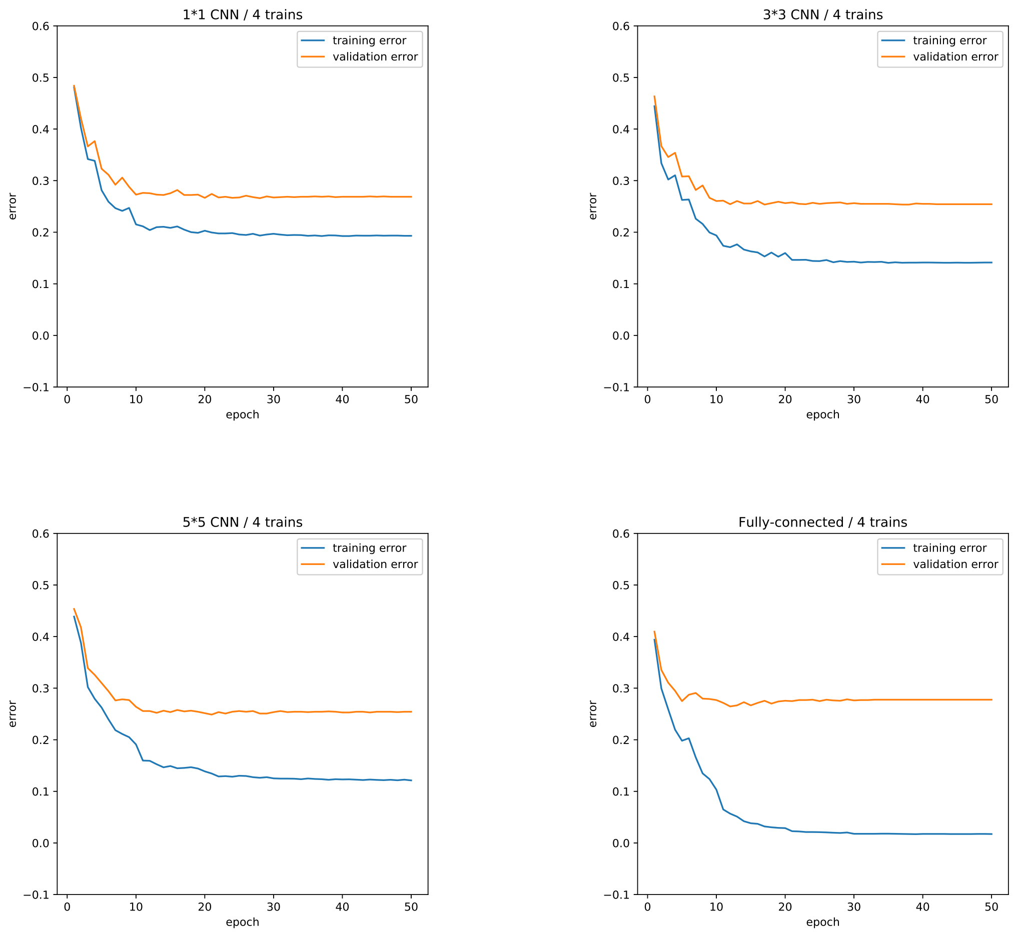


Figure 4.4: Learning curves of different network architectures in small scale scenario.

model's capability of generalizing this problem. For the second hypothesis, we examine the purity of the outputs of the policy network by action class. This measurement means how sure the network makes predictions, revealing the noise in the training data.

4.4.5.1 State Space

The experiment is designed as follows: 500 training instances and 100 validation instances of the large-scale scenario is used. We train several 1*1 CNN with the same hyper-parameters as in previous content but with a different number of training instances. Fig 4.6 shows the learning curve of validation error by training set size.

From the learning curve, we can see, the validation error gets decreased as more training

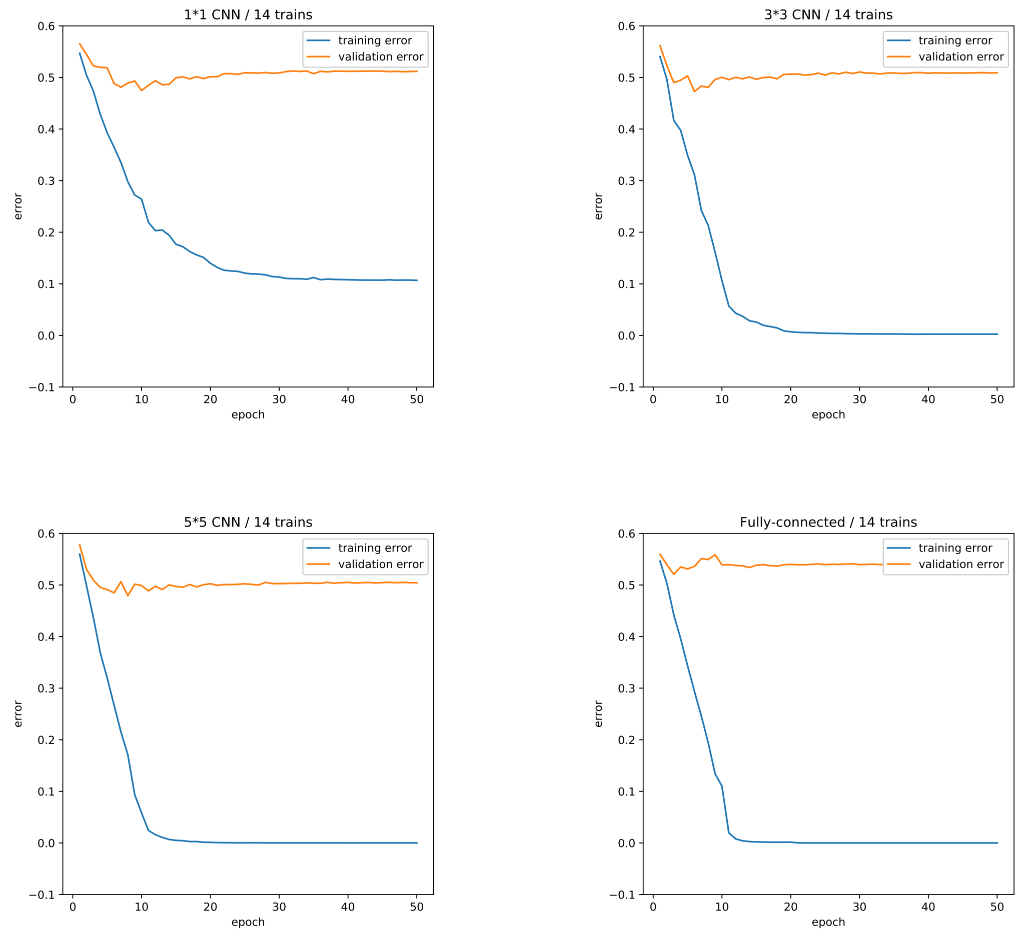


Figure 4.5: Learning curves of different network architectures in large scale scenario.

data is provided. Though the trend gets less evident as the training set gets larger, the validation error does not seem to converge even when 500 training instances are used. From this experiment, we have a rough estimation of the state space in the large-scale scenario. This could be one of the reasons of the overfitting in our previous experiment (where only 120 for training).

4.4.5.2 Data Quality

The local search solutions we used for training are all feasible at the solution level. However, at the state level, we cannot claim that those actions are ground truth since they result from our subjective state modeling. The previous error evaluation matrix has shown much error in confusing tracks for move actions. Also, from the algorithm, we know that the local search algorithm assigns movements randomly. Further evaluation of how much noise is in

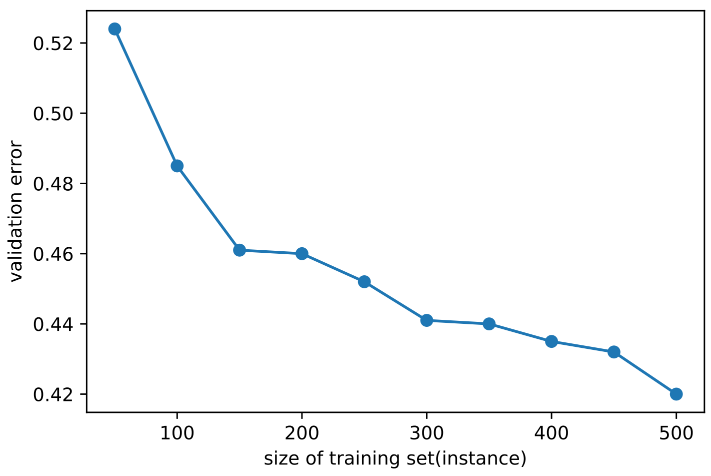


Figure 4.6: Validation error curve by training set size.

our dataset would help us explaining the overfitting problem.

Since the state is a structural data formula, it is challenging to conduct visualizations on it. Instead, we look into the outputs of the trained 1*1 CNN and calculate the entropy of output actions. A low entropy value means the output distribution is pure and the network is sure for this input state. A large entropy value reveals dispersed distribution on action probability, and the input state is confusing for the network.

The entropy is computed with the following formula: Given $T = [t_1, t_2, ...t_M]$ as the output scores for train, $A = [a_1, a_2, ...a_N]$ as the scores for action, and w for the score of wait, we have:

$P = [p_{11}, ..., p_{ij}, ..., p_{MN}, p_w]$ as the scores for each action, where:

$p_{ij} = Sigmoid(t_i)Sigmoid(a_j)$, $p_w = Sigmoid(w)Sigmoid(max(T))$.

We compute the entropy of normalized action scores: $norm(P) = [\frac{p_{11}}{sum(P)}, ..., \frac{p_{ij}}{sum(P)}, ..., \frac{p_{MN}}{sum(P)}, \frac{p_w}{sum(P)}]$, and plot the histogram of entropy value by action class in both scenarios (see Fig 4.7).

The entropy histogram shows that in both scenarios, the **Move** actions have the most significant output entropy, then the **Wait** and the **StartService**. The **Depart** action is the most certain action class for the network. This confirms the observation in the error evaluation matrix that there is much noise to **Move** actions in our dataset. Comparing among scenarios, the large-scale scenario has more noises but with similar distribution among action classes as in the small-scale scenario.

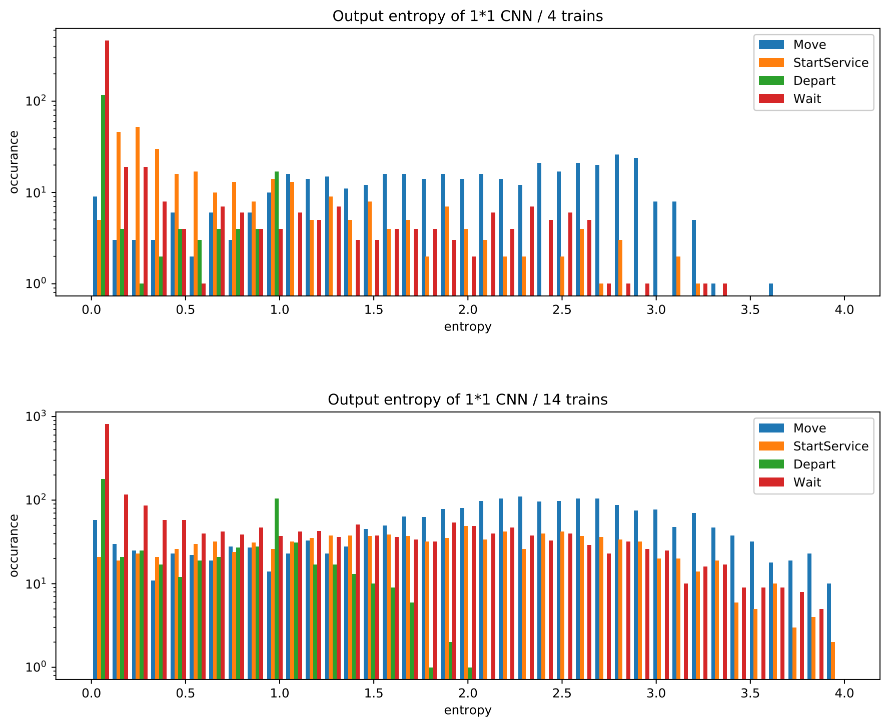


Figure 4.7: Entropy histogram by action in both scenarios.

We have the following conclusion for our two hypothesis: According to the local search solutions and our state modeling, two factors contribute to the overfitting. The training set we used for training is not sufficient. The dataset also includes noises which make the neural network challenging to generalize from the training set.

4.5 Discussion

From the above experiment, we come up with the following conclusions: The prediction accuracy, as well as our specific error metrics among these 4 networks, does not differ significantly. However, complex models like the fully-connected network and larger kernel CNNs requires more parameters. This result in sizeable computational expense and severe overfitting.

This experiment cannot validate our hypothesis since 1*1 CNN did not outperform other networks significantly. However, while having a close performance with other models, 1*1 CNN has the advantage of fewer parameters. The learning curves also show that there is less overfitting with 1*1 CNN than with other frameworks.

An evaluation for errors is performed by evaluating the class confusion matrix and the error evaluation matrix. These evaluation metrics shows that the most frequent mistakes are made to move actions.

We also explore through overfitting and come up with two insights for this problem. The state space of this problem is enormous. A larger training set is suggested to achieve better prediction performance. Also, solutions from the local search algorithm have randomness at the state level, especially for move actions.

Through the above reasons, we select 1*1 CNN to build our system in later chapters.

Chapter 5

Search Strategy

In this chapter, we have implemented the search strategy for solving the TMSP. We first explain the motivation of using a search algorithm in our system. Then, we present experiments among three tree-search algorithms: GRASP, A* Search, and Monte-Carlo Tree Search. Finally, we evaluate the performance of these approaches and give a conclusion.

5.1 Motivation

Solving TMSP problem instances with greedy strategy from heuristic is easy to fail. It requires the heuristic to be correct whenever processing next step. For instance, given a problem instance that is expected to be solved with 30 steps, a heuristic with 90% accuracy has a probability of $1-0.9^{30} = 95.76\%$ to fail with greedy strategy in theory.

To solve this issue, we can implement search to provide more chances when the heuristic make mistakes.

5.2 Experiment

5.2.1 Experiment Setup

Similar to the experiments in chapter 4, we conduct experiments in the small-scale scenario (4 trains). We select the first trained 1*1 CNN in the cross-validation as the heuristic, with the corresponding test set for evaluation on the instance level. Table 5.1 shows the data setup for this experiment.

For each of the three search algorithms, given $P(s,a)$ as the output score of state s and action a from the policy network, we use the following strategy and hyper-parameters follows:

	Training Set	Validation Set	Test Set
Fold	Fold 1,2,3	Fold 4	Fold 5
Instance Quantity	120	40	40

Table 5.1: Data setup for the search.

GRASP: since the local search cannot be applied in a tree search, we only perform the adaptive randomized greedy search for each iteration.

The adaptive randomized greedy search is performed by maximizing the following formula:

$$a = \operatorname*{argmax}_{j} r_{ij} * P(s_i, a_{ij})$$

where r_{ij} is a random number from $X \sim U[0,1]$.

A* Search: we assume the cost for reaching current node $g(n)$ is always 0. $f(n)$ is the output score from the network. With this setup, the strategy becomes a best-first search with a greedy strategy.

MCTS:
Selection: PUCB strategy[40] with $c = 1$:

$$a = \operatorname*{argmax}_{a} \frac{W(s,a)}{N(s,a)} + cP(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

Expansion: random expansion.
Rollout: randomly sample the action with $P(s,a)$ as weights.
Backpropagation: reward for solved node: $W(s,a) = 1$, otherwise $W(s,a) = 0$.

5.2.2 Evaluation Metrics

The most critical evaluation metric is the problem instance solvability. We evaluate this with how many problem instances can these search algorithms solve.

Concerning the computational efficiency, a possible evaluation metric could be how much time the search algorithm uses when a solution is found. Consider solving the TMSP problem by searching for a target node in a game tree, all of the three search strategies aimed at reaching a target node without going through the whole tree. Less exploration on unpromising nodes saves computation resources. However, on the other hand, the computational expense for making a single expansion is different among these three search algorithms. For instance, GRASP conducts randomization for each expansion, but MCTS needs to compute complex utility score from both heuristic and previous outcomes. Therefore, we examine how much time has the search algorithm used until a target node is reached. A search algorithm that requires less time is considered to be more efficient.

Also, aggregated metrics from all problem instances could be insufficient to prove the performance of search algorithms. This is because each problem instance is different and could receive different evaluation among those three search algorithms. In this experiment, we also evaluate the performance per instance to see if a search algorithm could consistently perform better for multiple instances.

5.2.3 Result

Considering that GRASP and MCTS are stochastic search approaches, we have replicated the problem instances 5 times for these two algorithms to get more results.

We set an upper-bound of 20 seconds for the maximum runtime of node exploration to make sure the computational expense of the experiment is feasible. The search algorithm stops as soon as a target node is found or it has spent more than 20 seconds. If the search of a problem instance exceeds this upper-bound, it is considered unfeasible with this search strategy.

Table 5.2 shows the general solvability of the three search strategies. We can see MCTS has the best general instance solvability. GRASP is less good, and the A* Search has the worst performance.

Strategy	GRASP	A*	MCTS
Instance	200	40	200
Solved	132	14	179
Solved Rate	0.66	0.35	0.895

Table 5.2: Instance solvability of the tree search algorithms.

Fig 5.1 shows the histogram of our evaluation metrics with these three algorithms. The red line is a boundary that distinguishes feasible and unfeasible result according to our evaluation metrics.

From this evaluation metric, we can see that MCTS generally requires the least time to reach target nodes. GRASP is also less good in this evaluation metric, and A* Search has the worst performance.

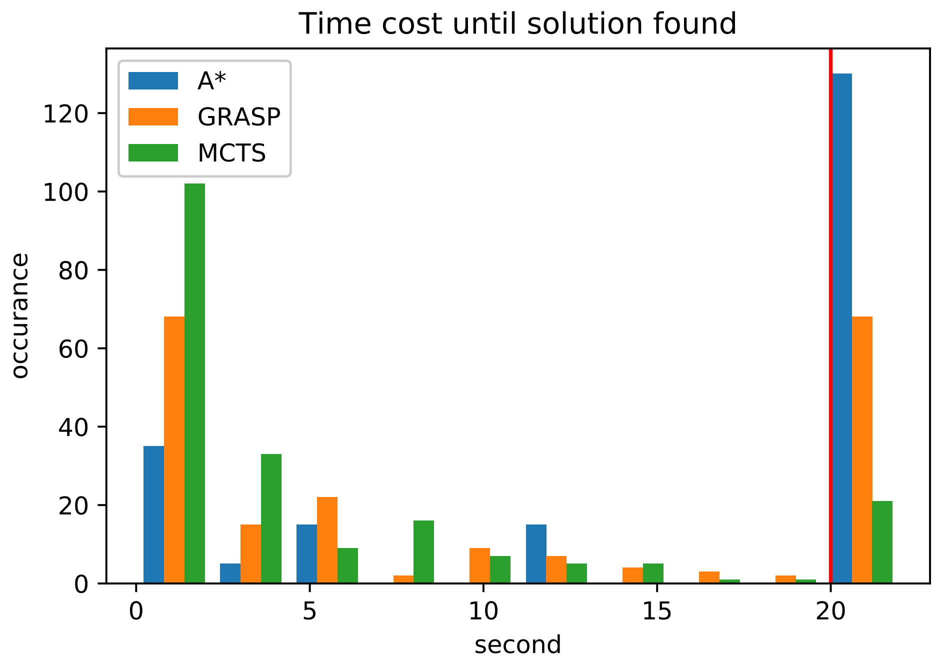


Figure 5.1: Evaluation of three search algorithms.

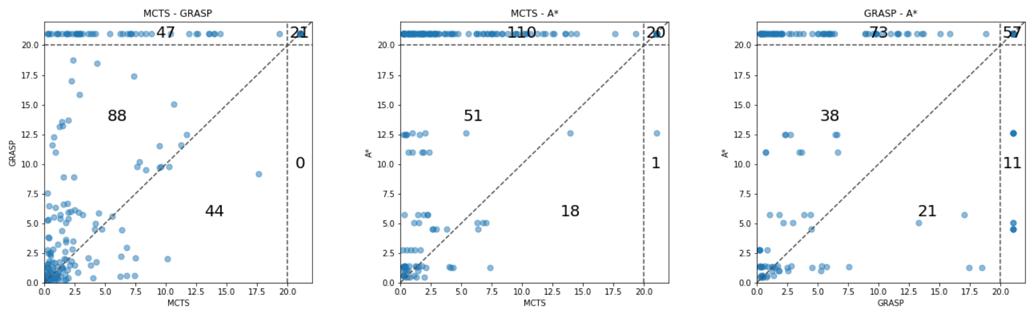


Figure 5.2: Visited node number per instance between search algorithms.

Fig 5.2 shows pair visualizations of the time expense per instance between search algorithms. Each of these scatterplots can be segmented into 5 sections, representing 'only strategy A (B) solved,' 'both strategies solved but A (B) used less time,' and 'neither of them solved.' We use annotations to describe the quantity of instance lies in each section.

Through these scatter plots we can see that previous conclusion (MCTS>GRASP>A*) does not consistently hold for all of the problem instances. There are distinct portions of instances that a better search strategy could not outperform a less good one. This shows that the performance still partially depends on the instances themselves. Specifically, give a problem instance, MCTS is not certain but more likely to solve it with less computation than the other two approaches. Beside MCTS, GRASP performs better than A* in the same way.

5.3 Discussion

From the above experiments, we have the following conclusion: When searching for solutions of TMSP problems with the policy network heuristic, MCTS generally outperforms GRASP and A* with the criteria of both instance solvability and computational expense. However, MCTS does not consistently outperform the other two among different problem instances. Though most of the problem instances are solved best with MCTS, there are still some problem instances that GRASP, or A* performs best.

According to the above conclusions, We select MCTS to build the system in later chapters.

Chapter 6

Solution Evaluation

From the last three chapters, we have first presented how to formulate the TMSP sequentially to enable supervised learning. We have also conducted experiments that compare the performance of different network frameworks and search strategies. In this chapter, we present a thorough evaluation of our solver system. This includes three parts: First, we evaluate the performance of the 1*1 CNN as the action predictor. Then, we build a MCTS algorithm with the network as a heuristic and test its performance with problem instances. Finally, we compare solutions obtained by our approach with solutions from the local search algorithm concerning several aspects. Finally, we evaluate solutions obtained by our approach concerning their quality to assess the performance of our system.

6.1 Dataset

The dataset we used for this experiment includes scenarios of 4, 6, 8, 10, 12, 14 trains respectively in the service site 'Kleine Binckhorst.' Each scenario has 200 problem instances. The solutions of these instances are obtained by allowing 20 seconds for the local search, which means they are the best among all solutions generated in 20 seconds per instance.

Figure 6.1 shows the histogram of state quantity according to solutions of the local search algorithm. We can see that scenarios with more trains have more states per solution.

6.2 Prediction Evaluation

Considering the imbalance state quantity of solutions among different scenarios, training only one policy network with all of the states could result in bias that the network may concern large-scale scenarios more than small-scale scenarios. For this reason, we trained independent networks for each scenario and conducted evaluations separately. This also makes it easy for us to find out how problem scale affect the performance through our evaluation.

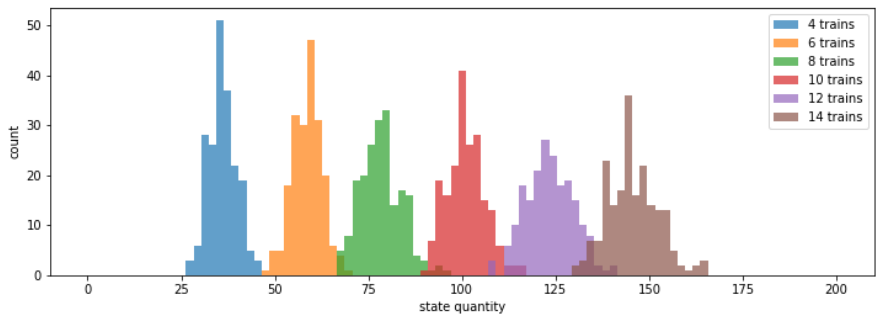


Figure 6.1: State quantity of solutions in each scenario.

Specifically, we trained each network with 160 solutions (120 for training, 40 for validation) in each scenario. Rest of the 40 instances are used for evaluation. We use the same hyper-parameters for training as in chapter 4.

Table 6.1 shows the prediction error for each scenario. We can see a significant trend that as the problem scale gets larger, the error increases.

Scenario	4 trains	6 trains	8 trains	10 trains	12 trains	14 trains
Testing Error	27.9	30.2	36.2	41.1	45.8	49.9

Table 6.1: Testing error for different problem scales.

To investigate more about what kind of mistake the policy network usually makes as well as its reason, again we use the two evaluation metrics from chapter 4. The class confusion matrix is used to present how the policy network confuses its predictions among different action classes (**Move, StartService, Depart, Wait**). The error evaluation matrix presents what kind of error does the policy network usually make.

Figure 6.2 and 6.3 shows the class confusion matrix and the error evaluation matrix among different problem scales.

According to the class confusion matrix, we have similar observations as in chapter 4. **Depart** has the least class error since in most of the scenarios it reached 100% class accuracy. **StartService** and **Wait** also have very few class mistakes. However, **Move** makes many mistakes. The most significant class error for the **Move** is predicting **Wait**.

Comparing among different scenarios, we can see that the general accuracy from class confusion matrix slightly decrease as the problem scale gets more massive. However, the distribution of these kinds of error almost remains the same.

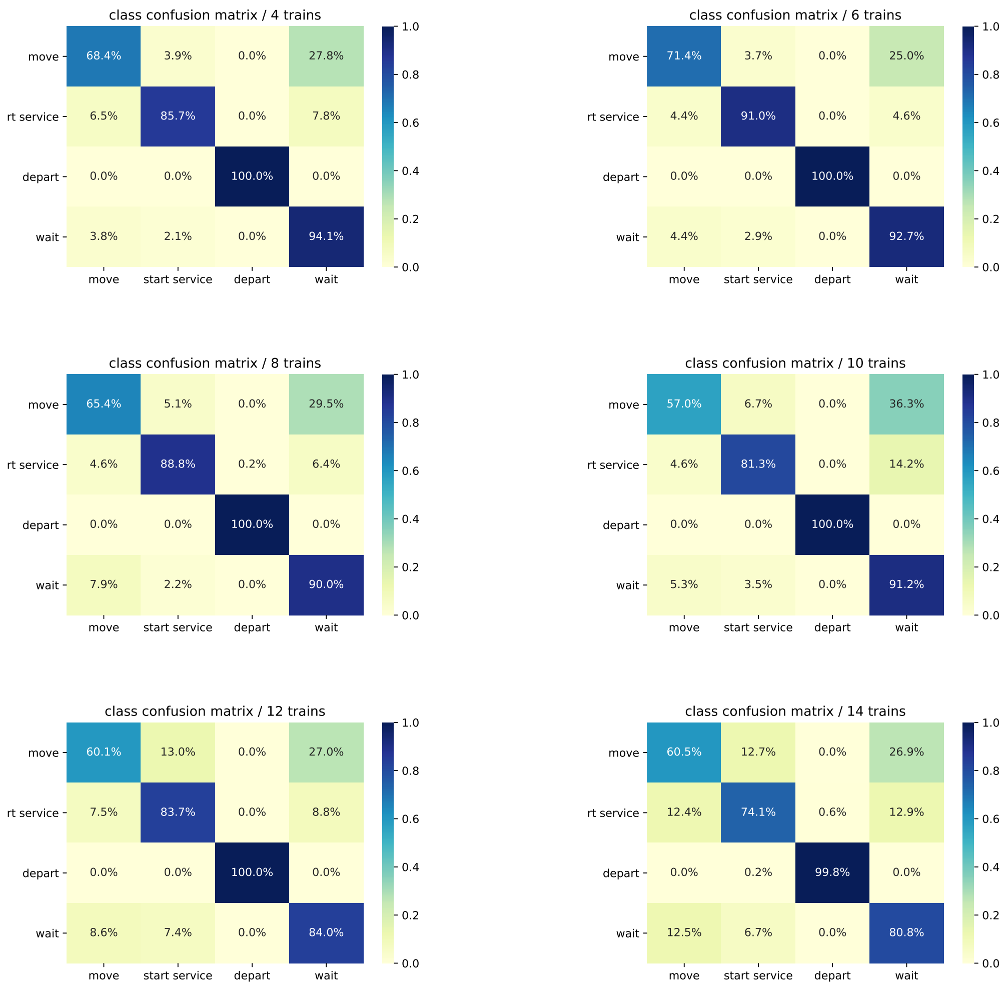


Figure 6.2: Class confusion matrix among different problem scales.

From the error evaluation matrix, we can see that for each action class, what kind of error is usually made beside class error. First, **Move** seldom gets correctly predicted. Beside confusing with other classes, the majority prediction picks the correct train but fails to predict the correct destination track. The performance of the other three action class is much better than the **Move** class. In **Depart** class, an exposed portion of error that selects the wrong train for departure has been made.

As the problem scale gets more substantial, we have the following trends in the error evaluation matrix: The performance generally decreases, with fewer prediction could be entirely correct in all of the four action classes. Specifically for the **Move** class and **Depart** class, more inner-class errors are made. For **StartService** class and **Wait** class, more confusion

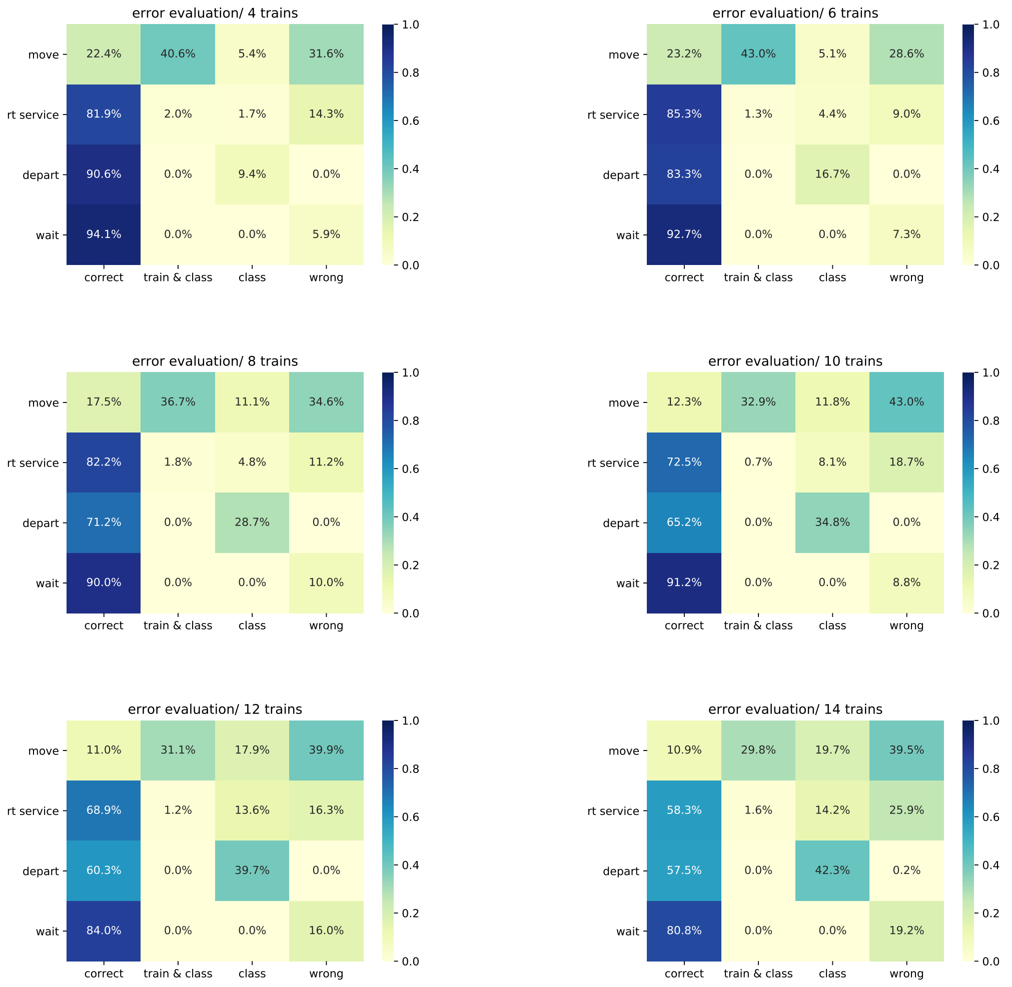


Figure 6.3: Error evaluation matrix among different problem scales.

between classes happen.

To summarize, the primary deficiency of the policy network is that it fails to predict **Move** easily by either mistake the destination track or predicting wait actions. The general performance also gets worse as the problem scale grows (more trains involved). Specifically, **Depart** is very sensitive to the problem scale.

In reality, different kinds of error from the above analysis could have a different severity level for planning. For example, confusing the destination track of a **Move** is not crucial for finding solutions. The train still has chances to get to the correct destination. However, unlike the **Move** class, performing incorrect **StartService** and **Depart** action is irreversible. These errors are considered to be deadly and often lead to immediate termination for the planning.

6.3 Solvability

In this section, we use our solver system (policy network with MCTS) to solve unseen problem instances for each scenario. We simply evaluate the solvability by counting how many problems instances our solver can solve successfully.

We use the 40 testing problem instances illustrated in former section to test the instance solvability for each scenario. These testing instances have been first duplicated to 200 instances. In other words, each instance is solved 5 times with the MCTS to handle its randomness. Due to the limited computational resource, the MCTS cannot be performed unlimitedly. We set 20 seconds as a restrict upper bound of run time allowed. If the solution cannot be found within this time, it is regarded as not solved. The hyper-parameters, as well as strategies for selection, expansion, rollout, and backpropagation in the MCTS, remain the same as in chapter 5.

Table 6.2 shows the instance solvability for each scenario. We can see the solved rate is susceptible to the problem scale. In the 4-train scenario, most of the instances could be solved. When the problem scale increases to 6 and 8 trains, the solved rate drops obviously. For the scenarios with more than 8 train, none of the instances could be solved within 20 seconds.

Scenario	4 trains	6 trains	8 trains	10 trains	12 trains	14 trains
Solved Instance	182	134	40	0	0	0
Solved Rate	0.915	0.67	0.2	0	0	0

Table 6.2: Instance solvability.

Table 6.3 shows how many times instances get solved in the 5-round MCTS for each scenario. We can see in each scenario, most of the instances are either solved 5 times or not solved even once. However, some of the instances lie in between that they only get solved part of the rounds. This result shows that the MCTS has certain randomness in the search procedure. Given the same instances, the search time could be inconsistent.

Fig 6.4 shows the histogram of time cost for finding solutions for 4, 6, 8 trains. From the histogram, we can see that for the 4-train scenario, most of the problem instances are solved within 5 seconds. But as the problem scale gets large, the distribution of solving time tends to be uniform in this 20 seconds interval. A possible reason is that larger problem scales use more actions to perform a solution (according to Fig 6.1), which requires the search to go deeper in the game-tree. The 20-second limitation could be insufficient when there are

Solved count	5	4	3	2	1	0
4 trains	30	7	1	0	1	1
6 trains	16	3	10	4	4	3
8 trains	1	1	3	6	10	19
10 trains	0	0	0	0	0	40
12 trains	0	0	0	0	0	40
14 trains	0	0	0	0	0	40

Table 6.3: Solved count in 5 rounds.

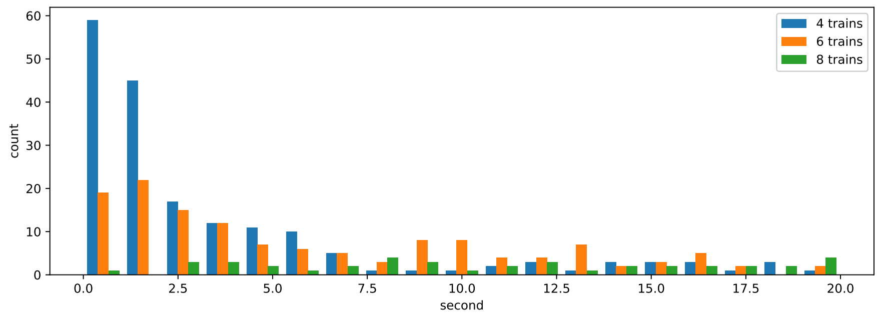


Figure 6.4: Time cost for finding solutions.

more than 8 trains.

We can conclude this section with the following conclusions: Both instance solvability and runtime are sensitive to the problem scale. A more significant problem scale results in less solvability and more runtime. Also, finding solutions slightly depends on the chance due to the randomness from the MCTS framework.

6.4 Solution Quality

Besides the capability of solving problem instances, we also care about the solution quality of the solutions from our approach. According to our research goal, we focus on two evaluation metrics: solution conciseness and consistency. The solution conciseness can be explained as how much operation is needed to execute a solution. It briefly shows the resources (time, maintenance crew, and energy). The consistency describes how similar the solutions from the solver could be given the same or similar problem instances.

6.4.1 Evaluation Metrics

We define the solution conciseness with the quantity of action in a solution according to our state modeling. This definition makes it possible for us to compare this metric with the local search algorithm. Defining similarity among complete solutions is difficult. We evaluate the consistency from a realistic perspective, only looking into the consistency of service as it is the primary part of the solution. The consistency is defined with the following formula:

Given a solution with m trains $T = [t_1, ...t_m]$ and n kinds of service $S = [s_1, ..., s_n]$, we construct vector O and P to map how services are processed:

$$O = [o_{t_1 s_1}, ..., o_{t_m s_n},]$$

$$P = [p_{t_1 s_1}, ..., p_{t_m s_n}]$$

where $o_{t_i s_j}$ is the order of s_j in t_i, $p_{t_i s_j}$ is the track where s_j is performed for t_i. O represents the consistency of service order and P stands for the consistency of where services have been processed.

The similarity between the two solutions can be estimated by computing the hamming distance of O_i, O_j and P_i, P_j. We define similarity between two solutions with the following score:

$$Similarity(Solution_i, Solution_j) = 1 - \frac{Hamming(O_i, O_j) + Hamming(P_i, P_j)}{Length(O_i) + Length(O_j)}$$

The similarity score is defined with ratio because the vectors could have different length due to different problem scales and service requirement among instances. By evaluating with a ratio, solution sets of different scenarios are comparable. In this definition, the similarity score gets 1 when O and P are the same and 0 for entirely different. Elements in either O or P have equal importance on the similarity score. Based on the similarity score, we have the consistency for multiple solutions with the following score:

$$Consistency(Solution_1, ..., Solution_n) = \frac{\sum_{i \neq j} Similarity(Solution_i, Solution_j)}{n(n-1)}$$

Since the solution number we obtained for each instance differs, we compute the average similarity score among all possible pairs as the consistency score.

We give a simple example of two trains and two services, calculating the consistency of among two solutions. Suppose two trains are involved, with train 1 needs service 2 and train 2 needs service 1 and 2. Solution 1 plans service 1 prior than service 2 for train 2 while solution 2 on the contrary. For train 1, solution 1 arranges the only service at track 4 while solution 2 arranges at track 3. We have the status vector O and P in Table 6.4. In this example, there are 6 elements for the instance. Two order elements and one track element are different, so the hamming distance between solution 1 and solution 2 is 3. The similarity score is $1 - \frac{3}{6} = 0.5$. With only two solutions in this computation, the consistency score is 0.5.

(train, service)	order(1,1)	order(1,2)	order(2,1)	order(2,2)	track(1,1)	track(1,2)	track(2,1)	track(2,2)
Solution 1	-	1	1	2	-	4	5	6
Solution 2	-	1	**2**	**1**	-	**3**	5	6

Table 6.4: Example of calculating consistency score.

6.4.2 Result

We evaluate our solutions with the two metrics. For the solution conciseness, we make a comparison with the local search. However, for the consistency score, we are not able to compare with the local search because we only have one local search solution for each problem instance. Instead, we provide a comparison between our approach and randomly assigning services within possible orders and possible tracks (but without examining the plan's feasibility). With this comparison, we can see how much better our approach can do compared with a random strategy.

6.4.2.1 Solution Conciseness

We compare the solution conciseness of the solutions from our approach with the solutions from the local search algorithm. Figure 6.5 shows the histograms of the gaps between local search solutions and our solutions in the instance level. We can see that the local search algorithm generally use fewer steps to solve problem instances. However, in some instances, our supervised learning approach get better solutions than the local search algorithm.

For further comparison, we look into an example that local search solution outperforms our solution with 8 fewer actions in the 4-train scenario.

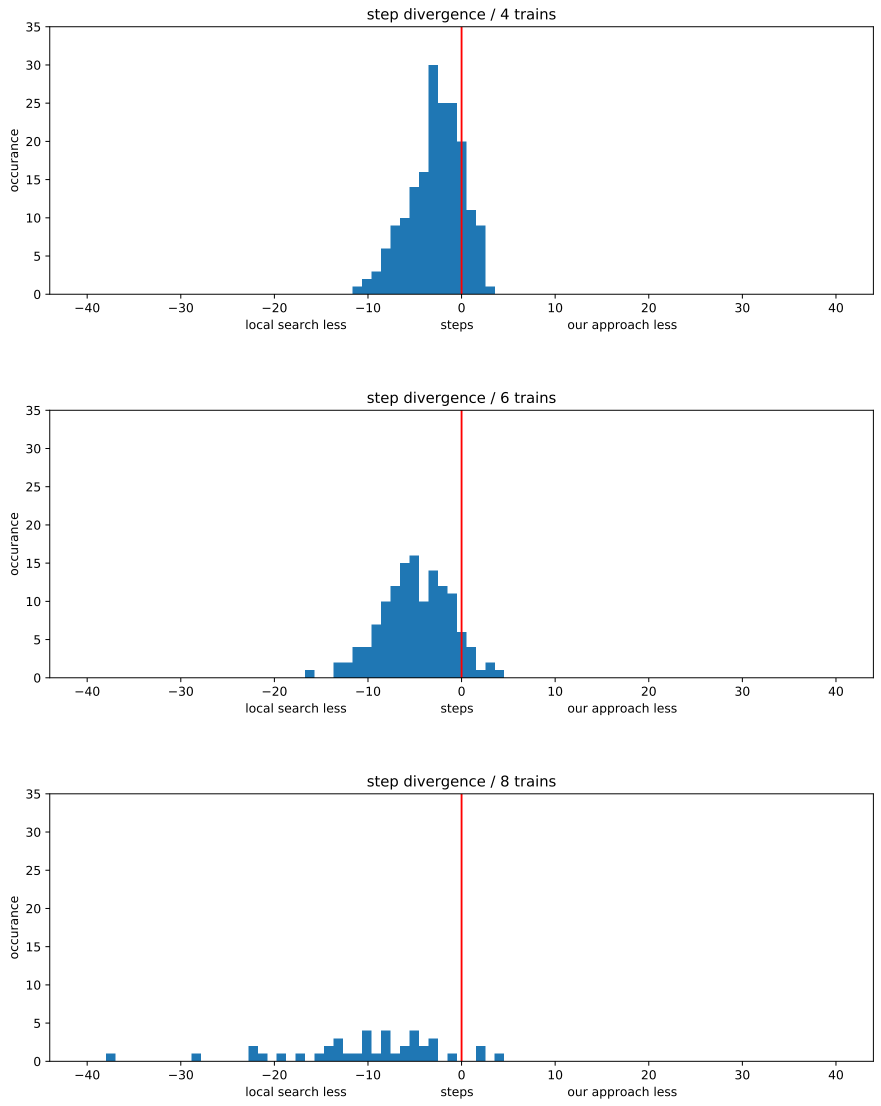


Figure 6.5: Conciseness difference per instance between two approaches.

Solution from the local search:	Solution from our approach:
Wait	Wait
Move(4000, 11)	**Move(4000, 12)**
-	**Move(4000, 7)**
-	**Move(4000, 11)**
StartService(4000, 1)	StartService(4000, 1)
Wait	Wait
Move(4000, 9)	Move(4000, 7)
Wait	Wait
Move(3000, 8)	Move(3000, 8)
StartService(3000, 5)	StartService(3000, 5)
Wait	Wait
Move(3000, 12)	Move(3000, 11)
StartService(3000, 1)	StartService(3000, 1)
Wait	Wait
Move(3000, 8)	Move(3000, 8)
Wait	Wait
Move(1000, 6)	**Move(1000, 13)**
-	**Move(1000, 11)**
-	**Move(1000, 13)**
-	**Move(1000, 9)**
-	**Move(1000, 11)**
-	**Move(1000, 6)**
StartService(1000, 5)	StartService(1000, 5)
Wait	Wait
Move(2000, 6)	Move(2000, 10)
Wait	Move(2000, 8)
StartService(2000, 5)	StartService(2000, 5)
-	Wait
-	Wait
Move(1000, 12)	Move(1000, 11)
StartService(1000, 1)	StartService(1000, 1)
Wait	Wait
Move(2000, 11)	Move(2000, 12)
Wait	-
StartService(2000, 1)	StartService(2000, 1)
Wait	Wait
Wait	Wait
Depart(1000)	Depart(1000)
Wait	Wait
Depart(4000)	Depart(4000)
Wait	Wait
Depart(2000)	Depart(2000)
Wait	Wait
Depart(3000)	Depart(3000)

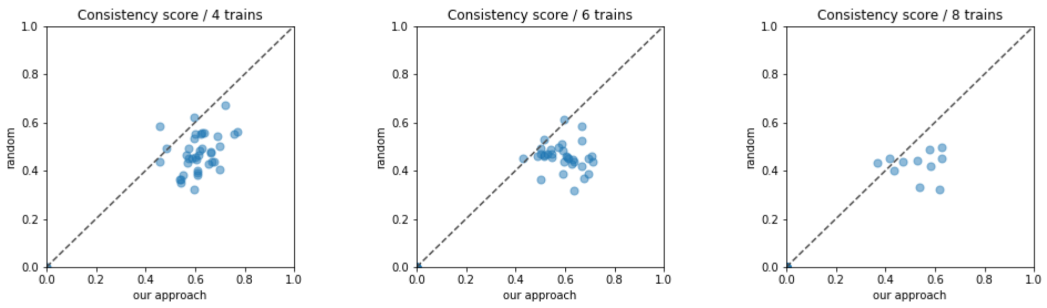


Figure 6.6: Consistency score by scenarios.

We can see two solutions are very similar in the way they schedule service activities. However, our approach performs a lot of unnecessary movements when scheduling, which increase the quantity of action in the solution. These unnecessary movements also verified our previous conclusion for the policy network that confusions of destination tracks happen very often.

6.4.2.2 Consistency

Fig 6.6 compares the consistency score between our approach and the random scheduling per instance. We can see that the general consistency value of the solutions from our approach is around 0.6. Comparing with the random strategy, our system outperforms it on most of the problem instances. Therefore, our approach is considered to have little consistency for the service scheduling as it is slightly better than a random service assignment. This figure also indicated that the consistency value is not very sensitive to the problem scale since the distribution for the 4-train scenario, 6-train scenario, and 8-train scenario almost have the same mean and variance.

To conclude the evaluation on solution quality, we summarize the results as follows: We compare the solution conciseness of our approach with the local search algorithms. The local search algorithm outperforms our approach in most of the cases but not always. In the consistency evaluation specifically for service activities, solutions from our approach are slightly better than scheduling services randomly.

6.5 Discussion

In this chapter, we conduct experiments to test the performance of our solver system from three perspectives: prediction performance of the network, instance solvability, and solution quality. In the prediction performance part, our policy network has good performance only in small problem scales. As the problem scale grows large, the prediction accuracy decreased dramatically. The instance solvability is also sensitive to the problem scale when limited to a fixed runtime. As for the solution quality, we evaluate the solution conciseness and service consistency. In comparison with the local search algorithm in solution conciseness, most of the solutions from our approach are less concise than solutions from the local search algorithm. The consistency evaluation shows that our approach can achieve little consistency from the service scheduling perspective.

Chapter 7

Conclusions

In this chapter, we discuss our work from several perspectives. We first present a summary of this work with comments. Then, we answer the research questions we put forward in chapter 1. Finally, we suggest several future work perspectives.

7.1 Summary

The main contribution of this work is that it is the first research that solves the Train Maintenance Scheduling Problem on a full scale sequentially with the machine learning approach. Comparing to the local search algorithm, our approach has an apparent advantage in rescheduling part of the plan, which is an essential benefit for handling unexpected incidents in reality.

This work can be divided into three parts: sequential modeling, training policy network, and implementing search strategy. In the sequential modeling part, we first present a formulation by reactive modeling that transforms the TMSP from the continuous time domain into a chronological state-action formula. Then, we compare our approach with the mainstream uniform sampling concerning the complexity and flexibility of the state-action mechanism. The conclusion is that our reactive approach obtains a more concise mechanism without sacrificing flexibility compared with the uniform sampling approach.

Based on the sequential modeling approach, we investigate how to build a policy network. We first design a feature representation for states. We make this feature representation complex enough to make sure different states are also distinguishable by features. Since the TMSP has regular structure but contains few geometric relations, we come up with a hypothesis that 1*1 CNN is the best structure for this problem. Then, we conduct experiments to compare the 1*1 CNN with several modern network structures (3*3 CNN, 5*5 CNN, and fully-connected), and prove that 1*1 CNN achieve competitive performance with less model complexity. We provide experiment results on what kind of error the network usually makes, giving insights to what are the hard cases for the policy network to handle.

We found that the primary error from the policy network is confusing destination track on movements, which implies noise in the movement part of the training set. Since we also observed gaps between training error and validation error, we investigate possible reasons for this overfitting problem by exploring state space and data quality. By examining the prediction performance under different amount of training instances, we found that our training set is insufficient to cover the state space, which leads to overfitting. We also calculate entropy for the network's outputs to see noise in the training data. The result indicate a lot of noise for **Move** actions, which not only implies overfitting, but also confirms our error analysis that movements are difficult to predict.

In the search strategy part, we perform experiments on three commonly-used tree search strategies: GRASP, A*, and MCTS, comparing their performance on solving unknown problem instances. With computation time as the evaluation metric, the result shows that the MCTS solves more problem instances than the other two approaches within a fixed time period. Also, for the problem instances, the MCTS uses less time to reach solutions than the other two. These results prove that the MCTS has better performance than GRASP and A* in searching for solutions. Considering the search mechanism of these search strategies, we can say that the globally greedy strategy of A* does not work in TMSP. Also, the stochastic strategy that uses memory is better than the one without memory.

We also provide an evaluation of the whole system on solvability, solution conciseness, and consistency. The evaluation result shows that our approach succeeds in small-scale scenario but lack the capability of handling large-scale problems. For the solution conciseness, we compare solutions from our approach with solutions from the local search algorithms given same problem instances. Using the number of actions required as the evaluation metric, we found that our approach is generally less useful than the local search algorithm. By comparing exact solutions, we found that the main reason for using more steps is our solver usually makes a lot of unnecessary movements. This also result from the noises in the dataset that make **Move** actions confusing. As for the consistency, we first define a similarity metric that only concern position and order of service activities. Then, we use this metric to compare our approach with a randomly assigning strategy, and found that our approach has a higher consistency value than assigning services randomly. This proves that our approach is at least better than randomly assigning service activities.

To summarize, our approach is capable of handling most of the aspects of TMSP. By the results, we are sure that the policy network has a specific capability of shunting an arrival train, starting the service if needed, recognizing the departure moment by selecting the correct train to depart. The primary deficiency of the policy network is that it tends to perform random shunting moves rather than purposeful moves. Also, our agent only succeeds in small-scale scenarios that are more flexible and do not require careful routing. We cannot conclude that it can make intelligent shunting and routing plan. Generally, our approach did not proved to have obtain a comparable result with the local search algorithm.

7.2 Answering Research Questions

From the outcomes of this research, we answer the research questions in the first chapter:

RQ1: How to formulate TMSP problem in a way that enables supervised learning?

The TMSP problem could be formulated with the reactive agent modeling. We provide arguments that specifically for the TMSP, the reactive agent modeling approach could obtain more concise mechanism without sacrificing action flexibility compared with the commonly-used uniform sampling approach.

RQ2: What kind of neural network architecture works best for TMSP?

According to the evaluation results, the prediction accuracy does not differ a lot among these network structures (1*1, 3*3, 5*5 CNN, and fully-connected network). However, more complex models suffer more overfitting. Considering both results, we conclude that the 1*1 CNN is the best network structure for the TMSP problem.

RQ3: Using the policy network as a heuristic, what search strategy works best for TMSP?

The experiment results show that the MCTS generally solves more instances and uses less time to find solutions than GRASP and A*. We conclude that the MCTS is the best search strategy among three of our candidates.

Can we solve TMSP with supervised learning?

The evaluation results in chapter 6 show that our system has successfully solved a lot of unseen problem instances in the small-scale scenario, but fails in the large-scale scenario. The solvability of our approach is not comparable with the local search algorithm.

We conclude that the supervised learning is proven to work for small-scale TMSP only. For large-scale problems, it is not proved yet. But we proved that there are ways to enhance the performance and the possibility to work.

7.3 Future Work

There are several deficiencies in this research. First, the training data we use is not sufficient. The state space investigation in chapter 4 shows that for the large-scale scenario, the prediction performance still get improved as the training set grows to 500 instances. This proved that 120 training instances for our evaluation are not sufficient. We expect the prediction performance to get better than the current result given more training instances. Also, evaluating solvability with current time limit is insufficient for large-scale scenarios.

From Fig 6.1 we can see, a larger scenario requires the algorithm to search more in-depth in the game tree, which is expected to use more time. In this research, we regard an instance as not solved if the search time exceeds 20 seconds. However, large-scale scenarios could possibly get solved if given more search time. In addition, we have not sufficiently explored the possible hyper-parameter space but simply select commonly used hyper-parameters for network structure, training techniques, and detailed strategies in MCTS. This limits the performance of the system.

7.3.1 Dataset

From the further experiments in chapter 4, we found two possible improvements related to the dataset. First, we suggest using more training data to train the policy network as a result shows that our current training set size is not sufficient. Also, the output entropy result in chapter 4 shows that the local search solutions have much noise from the state perspective. These noises mainly affect **Move** actions. A suggestion to this is to train the network with more consistent solutions, which could reduce the noise and get less prediction error.

We also suggest explorations in the reinforcement learning approach. The limitation of supervised learning is that the performance of the obtained policy network largely depends on the quality of the training set. It sometimes results in local optimal heuristic function[47]. On the other hand, a reinforcement learning approach with appropriate objective function is expected to obtain the optimal one-step heuristic. In this research, we provide a state formulation and a feature representation that covers the TMSP in detail, enabling the direct application of reinforcement learning.

7.3.2 Explore Hyper-Parameters

We have performed comparisons among different network structures, but this is only part of the hyper-parameter comparisons of the policy network. There are more directions to investigate. For instance, our current network suffers from overfitting, which indicates that it could be too complicated. Then, a less complex network with fewer layers or fewer nodes may achieve better prediction performance. Also, we have not tried all possible hyper-parameters. Instead, we select a commonly used optimizer, batch size, MCTS selection strategy, etc. This hyper-parameter still worth exploration to find the one with the best performance.

7.3.3 Large-Scale Experiments

Due to computational complexity, we only examined the solvability with a 20-second upper bound. However, for the problem instances that are not solved within 20 seconds, we still do not know whether they are totally unfeasible or feasible with more time. We suggest large-scale experiments to figure it out. Possible improvement includes offering more time for the search, and also using more computational resources, optimizing codes (search and state transition), and even paralleling the MCTS.

Bibliography

[1] Integration of simulation modeling and inductive learning in an adaptive decision support system. *Decision Support Systems*, 9(1):127 – 142, 1993. ISSN 0167-9236. Model Management Systems.

[2] Routing trains through railway stations: complexity issues. *European Journal of Operational Research*, 98(3):485 – 498, 1997. ISSN 0377-2217.

[3] Saleh Albelwi and Ausif Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, 19(6), 2017. ISSN 1099-4300. doi: 10.3390/e19060242.

[4] H. Aytug, S. Bhattacharyya, G. J. Koehler, and J. L. Snowdon. A review of machine learning in scheduling. *IEEE Transactions on Engineering Management*, 41(2):165–171, 1994. ISSN 0018-9391.

[5] Abdelkrim Bennar, Jean marie Monnez, and Casablanca Maroc. Almost sure convergence of a stochastic approximation process in a convex set, 2009.

[6] J. A. Blessing and B. A. Watford. Infmss: An intelligent fms scheduling system. *World Productivity Forum and 1987 Ann. Int. Industrial Engineering Conf. Proc.*, pages 82–88, 1987.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. ISSN 1943-068X.

[8] G. Budai, D. Huisman, and R. Dekker. Scheduling preventive railway maintenance activities. *Journal of the Operational Research Society*, 57(9):1035–1044, Sep 2006. ISSN 1476-9360.

[9] G. Caimi, F. Chudak, M. Fuchsberger, M. Laumanns, and R. Zenklusen. A new resource-constrained multicommodity flow model for conflict-free train routing and scheduling. *Transportation Science*, 45(2):212–227, 2011.

[10] Jean-Franois Cordeau, Paolo Toth, and Daniele Vigo. A survey of optimization models for train routing and scheduling. *Transportation Science*, 32(4):380–404, 1998.

[11] P. Cunningham and B. Smyth. Case-based reasoning in scheduling: Reusing solution components. *International Journal of Production Research*, 35(11):2947–2962, 1997.

[12] Lu Dai. A machine learning approach for optimisation in railway planning., 2018.

[13] Andrea DAriano, Dario Pacciarelli, and Marco Pranzo. A branch and bound algorithm for scheduling trains in a railway network. *European Journal of Operational Research*, 183(2):643 – 657, 2007. ISSN 0377-2217.

[14] Ahmed A. Elnaggar, Mahmoud Gadallah, Mostafa Abdel Aziem, and Hesham Eldeeb. A comparative study of game tree searching methods.

[15] Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[16] Keivan Ghoseiri, Ferenc Szidarovszky, and Mohammad Jawad Asgharpour. A multi-objective train scheduling model and solution. *Transportation Research Part B: Methodological*, 38(10):927 – 952, 2004. ISSN 0191-2615.

[17] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[18] Edward Groshev, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. *CoRR*, abs/1708.07280, 2017.

[19] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] A. Higgins. Scheduling of railway track maintenance activities and crews. *Journal of the Operational Research Society*, 49(10):1026–1033, Oct 1998. ISSN 1476-9360.

[22] A. Higgins, E. Kozan, and L. Ferreira. Heuristic techniques for single line train scheduling. *Journal of Heuristics*, 3(1):43–62, Mar 1997. ISSN 1572-9397.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[24] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975. ISSN 0004-3702.

[25] Levente Kocsis and Csaba Szepesvri. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[26] Sotiris Kotsiantis. Supervised machine learning: A review of classification techniques. 31, 10 2007.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[28] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. , 521:436–444, May 2015. doi: 10.1038/nature14539.

[29] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990.

[30] C.-Y. Lee, S. Piramuthu, and Y.-K. Tsai. Job shop scheduling with a genetic algorithm and machine learning. *International Journal of Production Research*, 35(4):1171–1191, 1997. doi: 10.1080/002075497195605.

[31] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Cs231n: Convolutional neural networks for visual recognition. *University Lecture*, 2015.

[32] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. 2015.

[33] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

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

[36] Allen Newell, Cliff Shaw, and Herbert Simon. *Chess Playing Programs and the Problem of Complexity*, pages 29–42. Springer New York, New York, NY, 1988.

[37] Santiago Ontañon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):1–19, 2013.

[38] Mauricio GC Resende and Celso C Ribeiro. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In *Handbook of metaheuristics*, pages 283–319. Springer, 2010.

[39] Joaqun Rodriguez. A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41(2):231 – 245, 2007. ISSN 0191-2615. Advanced Modelling of Train Operations in Stations and Networks.

[40] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[41] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6.

[42] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

[43] Tara N. Sainath, Abdel-rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for lvcsr.

[44] M.J. Shaw, S. Park, and N. Ramen. Intelligent scheduling with machine learning capabilities: The induction of scheduling knowledge. *IIE Transactions*, 24(2):156–168, 1992. doi: 10.1080/07408179208964213.

[45] Yeou-Ren Shiue and Chao-Ton Su. An enhanced knowledge representation for decision-tree based learning adaptive scheduling. *International Journal of Computer Integrated Manufacturing*, 16(1):48–60, 2003. doi: 10.1080/713804978.

[46] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. 529:484–489, 01 2016.

[47] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[48] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[51] Adithya Thaduri, Diego Galar, and Uday Kumar. Railway assets: A potential domain for big data analytics. *Procedia Computer Science*, 53:457 – 467, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.07.323. URL `http://www.sciencedirect.com/science/article/pii/S1877050915018268`. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[52] R.W. van den Broek. Train shunting and service scheduling: an integrated local search approach., 2016.

[53] Wikipedia contributors. Tic-tac-toe — Wikipedia, the free encyclopedia, 2018. [Online; accessed 13-July-2018].

Appendix A

An Example of the State Modeling

This example is in the service site 'Kleine Binckhorst' with only two trains. In each state, we have pointed out obvious changes by the last action with bold fonts.

State 1

Train	Material	Track	Position	In Service
1000	1	-	-	-
2000	2	-	-	-

Trigger	Description
Arrival	train: 1000, time: 10
Arrival	train: 2000, time: 25
Departure	material: 2, time: 50
Departure	material: 1, time: 60

Train	Service
1000	2
2000	1

Action 1: Wait

State 2

Train	Material	Track	Position	In Service
1000	1	**16**	**0**	-
2000	2	-	-	-

Trigger	Description
Arrival	train: 2000, time: 15
Departure	material: 2, time: 40
Departure	material: 1, time: 50

Train	Service
1000	2
2000	1

Action 2: Move(train: 1000, track: 11)

State 3

Train	Material	Track	Position	In Service
1000	1	**11**	0	-
2000	2	-	-	-

Trigger	Description
Arrival	train: 2000, time: 12
Departure	material: 2, time: 37
Departure	material: 1, time: 47

Train	Service
1000	2
2000	1

Action 3: StartService(train: 1000, service: 2)

State 4

Train	Material	Track	Position	In Service
1000	1	11	0	**2**
2000	2	-	-	-

Trigger	Description
Arrival	train: 2000, time: 12
EndService	**train: 1000, time: 23**
Departure	material: 2, time: 37
Departure	material: 1, time: 47

Train	Service
2000	1

Action 4: Wait

State 5

Train	Material	Track	Position	In Service
1000	1	11	0	2
2000	2	**16**	**0**	-

Trigger	Description
EndService	train: 1000, time: 11
Departure	material: 2, time: 25
Departure	material: 1, time: 35

Train	Service
2000	1

Action 5: Move(train: 2000, track: 13)

State 6

Train	Material	Track	Position	In Service
1000	1	11	0	2
2000	2	**13**	0	-

Trigger	Description
EndService	train: 1000, time: 8
Departure	material: 2, time: 22
Departure	material: 1, time: 32

Train	Service
2000	1

Action 6: StartService(train: 2000, service: 1)

State 7

Train	Material	Track	Position	In Service
1000	1	11	0	2
2000	2	13	0	**1**

Trigger	Description
EndService	train: 1000, time: 8
EndService	**train: 2000, time:13**
Departure	material: 2, time: 22
Departure	material: 1, time: 32

Train	Service
-	-

Action 7: Wait

State 8

Train	Material	Track	Position	In Service
1000	1	11	0	**-**
2000	2	13	0	1

Trigger	Description
EndService	train: 2000, time:5
Departure	material: 2, time: 14
Departure	material: 1, time: 24

Train	Service
-	-

Action 8: Wait

State 9

Train	Material	Track	Position	In Service
1000	1	11	0	-
2000	2	13	0	-

Trigger	Description
Departure	material: 2, time: 9
Departure	material: 1, time: 19

Train	Service
-	-

Action 9: Wait

State 10

Train	Material	Track	Position	In Service
1000	1	11	0	-
2000	2	13	0	-

Trigger	Description
Departure	material: 2, time: 0
Departure	material: 1, time: 10

Train	Service
-	-

Action 10: Depart(train: 2000)

State 11

Train	Material	Track	Position	In Service
1000	1	11	0	-

Trigger	Description
Departure	material: 1, time: 7

Train	Service
-	-

Action 11: Wait

State 12

Train	Material	Track	Position	In Service
1000	1	11	0	-

Trigger	Description
Departure	material: 1, time: 0

Train	Service
-	-

Action 12: Depart(train: 1000)

State 13(terminal state)

Train	Material	Track	Position	In Service
-	-	-	-	-

Trigger	Description
-	-

Train	Service
-	-