DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2019*

Machine Learning for Constraint Programming

TIANZE WANG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Machine Learning for Constraint Programming

TIANZE WANG

ICT Innovation Data Science
Date: June 24, 2019
Supervisor: Amir Payberah
Examiners: Christian Schulte, Vladimir Vlassov
School of Electrical Engineering and Computer Science
Swedish title: Maskininlärning för begränsningsprogrammering

Abstract

It is well established that designing good heuristics for solving Constraint Programming models requires years of domain experience and a huge amount of trials and error. In this thesis project, we conduct an empirical study of whether Machine Learning and Deep Learning techniques have the potential to help the design of constraint solvers.

Specifically, this thesis project examines the potential of Machine Learning and Deep Learning models for the regression task of predicting the makespan and solving time of a Job-shopScheduling Problem without actually solving the givenJob-shop Scheduling Problem instance. Several Machine Learning models are tested with manually designed features as input. Different Deep Learning architectures are explored with either just the Job-shop Scheduling Problem instance as input or with an additional input of the previously designed features.

The experiment results justify the potential of several proposed models in predicting the makespan and solving time. For predicting the makespan (unit: machine time unit), the best Random Forest regression model achieves a Mean Squared Error of 0.78 on the test set. The best Deep Learning model achieves a Mean Squared Error of 0.74 on the test set. For predicting the solving time (unit: millisecond) of a Job-shop Scheduling Problem, the best Random Forest regression model achieves a Mean Squared Error of 2.12×10^7 on the test set. The best Deep Learning model achieves a Mean Squared Error of 5.19×10^7 on the test set.

Discussions of the reason behind difference of different Machine Learning and Deep Learning models are provided and future directions are proposed.

Sammanfattning

Det är väl etablerat att det kräver många års erfarenhet av domänexpertis och mycket experimentell felsökning för att utforma en bra sökheuristik för villkorsprogrammeringsmodeller. I denna avhandling beskriver vi genomförandet av en empirisk studie med syftet att utreda potentialen av maskininlärningstekniker för att underlätta framtagandet av villkorsprogrammeringslösare.

Mer specifikt undersöker vi maskininlärningsmodellers regressionsförmåga att förutse makespanöch lösningstid för "Job-Shop Scheduling Problem"utan att för den delen lösa den givna "Job-Shop Scheduling Problem"instansen. Flertalet maskininlärningsmodeller testas med manuellt framtagna särdrag som indata. Olika djupmaskininlärningsarkitekturer utforskas med antingen bara "Job-Shop Scheduling Problem-instanser som indata eller med ytterliggare indata i form av de manuellt framtagna särdragen.

Experimentresultaten motiverar användandet av flertalet av de föreslagna maskininlärningsmodellerna för att förutse makespanöch lösningstid. För förutsägandet av makespan"(enhet: maskintidsenhet) uppnår den bästa Random Forestregressionsmodellen ett medelkvadratfel på 0,78 på testdatamängden. Den bästa djupmaskininlärningsmodellen uppnår ett medelkvadratfel på 0,74 på testdatamängden. För förutsägandet av lösningstiden (enhet: millisekund) av "Job-Shop Scheduling Problem"uppnår den bästa Random Forestregressionsmodellen ett medelkvadratfel på 2.12×10^7 på testdatamängden. Den bästa djupmaskininlärningsmodellen uppnår ett medelkvadratfel på 5.19×10^7 på testdatamängden.

Skillnadsorsakerna rörande de olika maskininlärningsmodellernas prestanda diskuteras i avhandlingen samt framtida forskningsinriktningar.

Acknowledgements

I would like to thank my examiner Christian Schulte and my supervisor Amir Payberah for offering the opportunity of the thesis project and their constant guiding and helping through the course the thesis.

I want to express my thankfulness to Vladimir Vlassov for reviewing and offering the feedback to the thesis.

I want to show my gratitude to Edward Tjörnhammar for his feedback on the thesis and offering the Swedish translation of the abstract.

Last but not least, I would like to express my gratitude to my family, my friends, and all of the people who have inspired and support me during the path.

Contents

1 Introduction **1**
- 1.1 Problem Formation . . . 1
- 1.2 Research Question . . . 2
- 1.3 Goals . . . 2
- 1.4 Thesis Contributions . . . 3
- 1.5 Methodology . . . 3
- 1.6 Ethics . . . 4
- 1.7 Sustainability . . . 4

2 Background **6**
- 2.1 Constraint Programming . . . 6
  - 2.1.1 Parts of a Constraint Programming Model . . . 6
  - 2.1.2 General Procedure of a Constraint Programming Solver 7
- 2.2 Machine Learning . . . 8
  - 2.2.1 Traditional Machine Learning Models . . . 9
  - 2.2.2 Deep Learning models . . . 12
- 2.3 Combination of Constraint Programming and Machine Learning 13
  - 2.3.1 Machine Learning for Constraint Programming . . . 13
  - 2.3.2 Constraint Programming for Machine Learning . . . 19
- 2.4 Job-shop Scheduling Problems . . . 19
  - 2.4.1 Mathematical definition of Job-shop Scheduling Problems . . . 20
  - 2.4.2 General Methods for Solving Job-shop Scheduling Problems . . . 20

3 Implementation **23**
- 3.1 Dataset Acquisition . . . 23
  - 3.1.1 Job-shop Scheduling Problems Instance Data Acquisition and Generation . . . 23

3.1.2 Gecode Model and Benchmarking Setup 25
3.1.3 Machine Learning and Deep Learning Dataset Generation 28
3.2 Traditional Machine Learning Models 30
3.2.1 Experimentation Setup 30
3.3 Deep Learning models 34
3.3.1 Models 34
3.3.2 Experiment setup 38

4 Results and Discussion **41**
4.1 Results of traditional Machine Learning models 41
4.1.1 Makespan 42
4.1.2 Runtime 42
4.2 Results of Deep Learning models 45
4.2.1 Models with vanilla Convolutional Neural Network . . 45
4.2.2 Models with LeNet mockup 46
4.3 Discussion of different results 49

5 Conclusion and Future Work **51**
5.1 Conclusion 51
5.2 Future Directions 52

Bibliography **54**

Chapter 1

Introduction

1.1 Problem Formation

Constraint Programming (CP) is a successful method for solving combinatorial search problems which draws upon a wide range of techniques from AI, computer science, databases, programming languages, and operations research [1]. Different aspects of constraint processing are investigated in theoretical computer science, logic programming, knowledge representation, operations research, and many related application domains [2].

The basic idea behind CP is that the user, instead of coming up with the exact procedure to solve a given program, states a specification of the problem which includes the decision variables, the possible values for the variables, and the set of constraints on the decision variables to make feasible solutions. In order to acquire solutions to the specification of the problem, the user then resort to general purpose or domain specific constraint solvers for help which usually encodes different heuristic for finding solutions for a given specification.

Traditionally, the heuristics in the solvers are hand engineered by domain experts and can be seen as reflections of their expertise in solving those specific kinds of problems which can often require a lot of past experiences from the same domain. One of the most common way to optimize a constraint solver is to come up with additional information during solving that can be utilized to greatly reduce the search space and such information usually comes from deterministic approaches that are hand-engineered.

In this thesis project, we are interested in investigating whether *Machine Learning* (ML) methods, especially *Deep Learning* (DL) methods, can be used to predict information that can be beneficial to prune the search space

and whether such additional information can improve the performance of constraint solvers.

1.2 Research Question

Job Shop Scheduling Problem (JSSP) , which falls into a bigger category scheduling problem, serves as a good candidate for our study as the problem is easily formulated, very well studied in research during the past, and that the hardness of the problem in terms of computation needed to come up with the optimal solution could vary dramatically even with a small change in the problem specification which could be a potential area where ML can make its contributions. JSSPs are concerned with finding a feasible schedule of different jobs each with a certain number of operations across different machines where one machine can only run one operation at a given time point. To be more specific, in solving a JSSP, we are interested in finding a schedule plan that minimize or maximize certain metrics, with *makespan*, which is the time elapses from the start of the first job and the end of the last job, being the most common one.

Constraint solvers for JSSP usually come up with a lower bound and a upper bound for the *makespan* and then start constraint propagation and searching from there to find the minimal *makespan* from there via binary search within the interval. One way to optimize the problem is to provide a estimated value for the optimal *makespan* to the solver at the beginning and let it first search around the estimated optimal *makespan*.

The research question for this master thesis project is that:

- Can we build a Machine Learning model that can predict, within a reasonable amount of time, the minimal *makespan* within a reasonable error range by only looking into the JSSP instance specification and some features from the solving phase that are easy to compute?

We will investigate the research question through empirical studies.

1.3 Goals

The goal of the project is to account for the usefulness of ML models in solving JSSPs, compare the performance of different models, and explore whether the additional information from ML models can actually yield performance gain for JSSP constraint solvers.

1.4 Thesis Contributions

The contributions of this master thesis are as follows:

- An experimental study of using different Machine Learning models to predict the optimal *makespan* of a Job-shop Scheduling Instance. We investigate different traditional Machine Learning models and several Deep Learning architectures and performance experiments and then evaluate the performance of these models. Our study shows that there are Machine Learning models which can predict the makespan within an acceptable range of error.
- Two randomly generated Job-shop Scheduling Problem instances dataset and the corresponding computational record for solving the instances. A Job-shop Scheduling Problem instances generator that can be used to generate more instances if needed. A Constraint Programming model for Job-shop Scheduling Problem.

1.5 Methodology

The research strategy for this dissertation is to perform quantitative and empirical research. We begin with a comprehensive and systematic literature study with the following two goals:

- Identify what are the state of art techniques in CP to solve JSSPs;
- Identify what ML and DL techniques has been applied in the field of CP with an extra focus on how they can be utilized to optimize the performance of constraint solver.

Specifically, we observe the potential of ML techniques to boost the performance of constraint solver in the setting of JSSPs. The experiment design is guided by the research objectives. We implement a CP model and corresponding constraint solvers for JSSPs. After that benchmarking for different JSSP specification is performed and the result are transformed into a dataset that is later used for ML and DL models for predicting the running time and *makespan* of the given JSSP specifications.

Delimitations

The extent to which we try to answer the research question is limited by a number of factors.

First, our evaluation is experimental and focuses on empirical results from quantitative experiments. For example, when comparing the performance of different ML and DL models on predicting the *makespan* and *runtime*, which is the running time that a CP solver to solve a CP model, we only demonstrate empirically that one model has better performance on the dataset used for testing.

Second, our experiments are only concerned with predicting the performance of ML and DL models on predicting the *makespan* and *runtime* of JSSPs. But we are not concerned with the performance of the model on general scheduling problems.

Third, we are making the assumption that the job-shop scheduling instances reflect the actual running time of each operation in the job which means we are assuming that an operation will finish with in the time given by the specification without delay.

Finally, although we make some modification to existing DL networks, we do not propose any new network architectures that are specially designed for JSSPs.

1.6 Ethics

While it might not seem obvious how solving a JSSP could lead to ethical concerns, when we generalize the ML method to boost the performance of other constraint solvers, problems might occur. For example, when performing roster scheduling, a ML model maybe learning from the data in such a way that the prediction of the ML model is in favour of one group of people having more night shifts than the other which might lead to biased predictions that might mislead the constraint solver to make unethical roster schedule.

Thus, when applying ML techniques to problems that might have the potential for ethical concerns, it is very important to not only to filter out features that might lead to unethical decisions but also to inspect the predictions of the model to avoid unethical results.

1.7 Sustainability

This work aims to contribute to Sustainable Development Goal #8[1] which reads “promoting sustained, inclusive and sustainable economic growth, full

[1] https://sustainabledevelopment.un.org/

and productive employment, and decent work for all" by demonstrating that ML has the potential of saving computing power and energy used for solving CP tasks via the use case of ML techniques can be used to fasten the process of JSSPs.

[0] https://sustainabledevelopment.un.org/

Chapter 2

Background

2.1 Constraint Programming

First introduced in Artificial Intelligence (AI) and graphics in the 1960s and 19770s, the concept of CP has led to many techniques that are used and studied in different fields of computing. CP has proven itself suitable for solving many combinatorial optimization problems which can be expressed in different formalism. One of such formalism commonly used in AI are *(*Constraint Satisfaction Problems) (CSPs) and *(*Constrained Optimization Problems) (COPs).

CP is a programming paradigm where, instead of specifying sequence of steps for the program to execute to get the result, one states the relations between variables in the form of constraints that must be satisfied in order to get feasible solutions for the problem which makes CP a form of declarative programming.

2.1.1 Parts of a Constraint Programming Model

CSP captures intuitions of CP problem specifications with variables, values, and constraints. Note that here a problem specification only states what are the solutions to the problem but not how to perform the computations to get those solutions. To be more specific, a CSP constrains the following parts:

- A finite set of variables $V = \{x_0, x_1, ...\}$;
- A universe U made of a finite set of values which specifies the values that each variable can take;
- A finite set of constraints C which states what are the solutions to the problem by specifying what relationships that variables should follow.

COP is another type of problems that can be solved by CP where the major difference from CSP is that, apart from satisfying all the constraints on variables and values, COP also have an objective function that needs to be minimized or maximized depending on the requirement of the task.

2.1.2 General Procedure of a Constraint Programming Solver

After specifying the configuration in a constraint model, when it comes to actually solving the problem, one will need to resort to constraint solvers which will take the constraint model as input and return either solutions to the constraint model or a failure message that no solution has been found. While different solving procedures and optimization techniques might exist for different constraint solvers, the general process of solving a constraint model includes the following steps:

- `Propagation`: where propagators, which are implementations of constraints, remove values that violate the constraints in the problem specification.
- `Search`: when propagator remove values anymore and a solution have not yet occur, we resort to search for help. And the search phase includes the following two steps.
  - `Branching`: defines the structure of a search tree.
  - `Exploration`: specifies the way to explore the search tree to look for solutions.

While constraints state relations among variables in problem specifications, propagators, which prune values that are in conflict with constraints, implement constraints when it comes to programming a constraint model. While in some cases propagation will lead to good results, most of the time propagation alone is not enough and that is when searching comes to the rescue. There are two main steps during the solving phase: branching which creates simpler sub-problems and defines the shape of the search tree; explorations defines the way to traverse the search tree, with depth first search being the most common one.

Heuristics are often applied to CP models with the aim of extra performance gain by exploring the search space smartly. One of the most well-known heuristics is first-fail where we would like the constraint solver to try the variable and value combination that is most likely to fail first. The key observation

behind this is that a failure that happens in the upper level of the search tree wastes less computation that a failure that happens at lower level of the search tree. Also, a failure at the upper level of the search tree prunes more search space than a lower level. There are various approaches to approximate the probability of failure for each variable. A naive approximation is to inversely correlate the probability of failure with the number of values left for the variable that we are interested in. *Accumulated Failure Count* (AFC) is another commonly used heuristic that take advantage of search history that prefers variables involved in failure. Nowadays, the search for better heuristics still remains an active field of research in CP.

Metaheuristics are general algorithmic frameworks designed to solve complex optimization problems. In recent years, metaheuristics are becoming successful alternatives also for solving optimization problems that include mathematical formulation of uncertain, stochastic, and dynamic information. To be more specific, metaheuristic algorithms such as *(*Ant Colony Optimization) (ACO), *Evolutionary Computation* (EC), *Simulated Annealing* (SA), *Tabu Search* (TS), and others, are justifying themselves as successful alternatives to classical approaches based on mathematical and dynamic programming for solving *Stochastic Combinatorial Optimization Problems* (SCOPs) [3].

Uncertainty is another interesting part of CP to be explored. Few CP formalism can deal with both optimization and uncertainty at the same time. [4] proposes a way to deal with combinatorial optimization problems under uncertainty by injecting the Rank Dependent Utility from decision theory and the results shows it is possible to handle uncertainty with regular CP solvers without the need to define a new formalism neither to develop dedicated solvers.

2.2 Machine Learning

ML is the scientific study of algorithms and statistical models that computer systems use in order to learn to perform specific tasks without using explicit step by step instructions, but relying on the ML model to explore the patterns behind and perform the inference. ML is seen as an important part of AI.

ML models are algorithms that build a mathematical model based on training data, in order to make predictions on the test data without explicitly being programmed on how to perform the task given that the training and the test data are independent and identically distributed.

ML models have been applied to a wide variety of applications, such as spam filtering, fraud detection, image recognition and classification, where it is almost infeasible even for domain experts with years of experience to de-

velop algorithms of specific instructions on performing the task with good performance. ML techniques can also be applied to the domain of feature selections by automatically learning important features or combination of features with regard to certain objective measurements via the power of computing. ML greatly facilitates the process of feature selection where domain experts hand engineer features and test the correlation with statistical analysis manually.

The types of ML algorithms can be classified into different categories based on the type of input and output data, and the type of problem they are intended to solve. A brief summary can be found as follows:

- `Supervised and semi-supervised learning`:

  Supervised learning models are built on top of a dataset where the output datasets have desired labels, while in the case of semi-supervised learning, parts of the labels are missing.

- `Unsupervised learning`:

  Unsupervised learning models takes only a set of input data and are supposed to explore and find the structure of the data without the help of desired output labels.

- `Reinforcement learning`:

  Reinforcement Learning (RL) models perform learning on the dataset with the help of feedback agents that give feedback on the action of the model and then take on the next step so as to maximize some predefined cumulative reward function.

In this thesis project, we are more interested in supervised learning methods which can be further divided into two categories:

- `Regression` where the prediction of a given input is a variable in a continuous space;

- `Classification` where the prediction of a given input falls into one of the pre-defined categories in a discrete space.

2.2.1 Traditional Machine Learning Models

In this subsection, we are going to briefly go through the traditional ML models that we have experimented with.

Linear regression

Linear models make a prediction using a function that is linear combination of input features. For regression problems, the prediction formula for input features x, w_i and b are the parameters, where $0 \leq i \leq p$ and $i \in \mathbb{N}$, that are involved in the following function:

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + ... + w_p * x_p + b$$

where $\hat{y}$ is the prediction from the linear model and w_i and b are the parameters to learn.

In order to train a linear regression model, an objective function is needed to evaluate the "goodness" of the parameters w_i and b which, in our case, is the *Mean Squared Error* (MSE) between the predictions and the true values. Perhaps, one of the disadvantages of linear regression is that there is no hyper-parameters which means the complexity of the model cannot be controlled.

Ridge regression

Ridge regression is very similar to linear regression in that they share the same formula. However, when it comes to objective function, a penalty term of $\lambda \sum_{i=0}^{p} {w_i}^2$ is added in additional the mean square error which serves regularization method that explicitly restricts the model to avoid overfitting. The regularization term is designed in such a way that it will requires the coefficients w_i to be close to zero which achieves the effect that each feature should have as little effect on the prediction as possible.

The value of λ controls the strength of the regularization with a larger λ lead coefficients close to zero which translate to more regularization and a smaller λ lead coefficients further away from zero which translate to less regularization.

Lasso regression

Lasso regression is another type of linear regression model with regularization that is similar to ridge regression except that the penalty term is $\lambda \sum_{i=0}^{p} \|w_i\|$ which prefers some coefficients to be exactly zero. This means that some features from the input data are entirely ignored by the model which can be viewed as a form of automatic feature selection.

Support Vector Machine

In the case of classification problems, while in some linear models we are optimizing the model in a way such that the misclassification error is as small as we can get, in Support Vector Machine (SVM), the optimization objective is the maximize the margin between different classes where the margin is defined as the distance between the separating hyperplane and the support vectors between different classes. The key intuition behind SVM is that a SVM model with large margins between different classes generalize better whereas the a model small margins may have the tendency to overfit on the training data.

Support Vector Regression (SVR) is a bit different from its classification counterpart, which is namely SVM, so that SVR can work with continuous values instead of discrete classification labels. While in SVM we try to maximize the margin between different classes, in SVR we aim to fit the error within certain threshold. As for the choice of kernel, which is the function that maps a lower dimensional data into a higher dimensional data, we tried "linear", "poly", "rbf", and "sigmoid" kernels.

Decision tree

Decision tree (DT) regression builds the regression model in the form of a tree structure. The tree split the dataset into parts by searching over all possible splits and choose the on that leads to the highest degrees of purity among subnodes of the tree which is usually measured by entropy. The tree is built through recursive partitioning until no further split is available or one of the predefined stopping criteria is reached, e.g. the minimum number of nodes in a node.

When making predictions, there is a little difference between classification and regression where the they both find the leaf for the new data point but classification gives the majority class label while regression returns the mean of all values in the leaf as prediction.

As for regularization, there are a number of hyperparameters that can be maneuvered, e.g. the maximum depth of the tree, the number of features to consider when looking for the best split, the minimum number of samples required to be at an internal node or a leaf node.

Random forest

Random forest (RF) regression learning methods fall into the category of ensemble learning methods where the key intuition behind is that the predictions

that are made by a set of weak learns through voting are more powerful that the decision made by a single strong learner. RF builds randomize trees on random samples of the training dataset.

When it comes to making predictions, the final prediction is made by aggregating the predictions of individual tree. For example, in the case of classification where every weak learner returns the probability for each class, the final prediction is the class with the highest probability. In the regression case, the final prediction could be as simple as the mean of all predictions of all the weak learners in the ensemble.

For regularization, in addition to the hyperparameters that can be tuned for individual DT regressor, there is also a number of other parameters related to the ensemble learning, e.g. the number of trees in the forest, whether bootstrap samples are used when building trees.

2.2.2 Deep Learning models

Artificial Neural Networks (ANNs) are computing systems inspired by biological neural networks that constitute animal brains. Such systems are considered as a sub-category of ML that have gained popularity in the area of research in recent years.

Compared to traditional ML models, ANN models have superior performance on the task where a lot of non-linearity and combinations of features are involved. For example, while traditional ML models such as DTs and RFs might have good performance on spam email filtering, when it comes to more challenging tasks like image classifications of cats and dogs, hand written digit recognition, ANN models usually performs much better with their learning capabilities for much harder problems.

While the idea of ANN has been around for many years, it is not until the recent boost of training power and availability of large datasets, that the performance of ANN begins to shine. However, even with the current computing power and dataset, most of the time, people still need to carefully design model architecture and develop training methods in order to gain the best performance.

2.3 Combination of Constraint Programming and Machine Learning

AI has gained a considerable amount of audience in recent years and has been successfully applied to many places, boosting the performance of a wide range of application in terms of accuracy, efficiency and effectiveness. Combinatorial optimization is such an area with modeling, search, and optimization being the three pillars of COP and CSP solving which makes ML especially relevant. As pointed out by [5], ML and CP have developed mostly independently with each other, but the vision that many opportunities lies at the intersection of the two areas has became more and more clear during the past decade. The combination of ML and CP has opened a research avenue with two directions. In one direction, how ML, especially its subcategory DL, can be used to ease the journey of modeling, solving and explaining CP tasks. In the other direction, how constraint solving can be utilized to facilitate ML applications. In this section, we will be looking into what has been accomplished in these two directions.

2.3.1 Machine Learning for Constraint Programming

One of the key challenges for operation and combinatorial optimization researchers when solving real-world problems is designing and implementing high-quality heuristics to guide the search process of solving a constraint model. In the past, finding such good heuristics highly relies on the experience and domain knowledge of the expert designing the system and very often a heuristic that is highly optimized for a particular process is not guaranteed to be the best heuristic for another problem. Thus, making it somehow time consuming for practitioners to find the best heuristic for a given problem. The problem has already been recognized in the CP community. For example, Jean-Francois Puget gave an invited talk at the 2004 International Conference on Principles and Practice of CP with the title *The next challenge for CP: ease of use* [6].

Much progress has been made in the past twenty years towards the goal to make CP easier to use. As pointed out by the survey [7], the progress towards automation are mainly in three different areas: acquisition which aims at automating problem modeling, solution which aim at automating problem solution, and explanation which aims at explaining the reason behind a success or failure in CP.

Acquisition

Problem acquisition aims at facilitating the problem specification process.

For modeling CP models, modeling languages of higher abstract level like MiniZinc [8] which provides a large set of predefined constraints that can ease the modeling process. Gecode [9] is an open source C++ toolkit for developing CP systems and applications which provides constraint solvers with state-of-the-art performance while making the system easily extensible with its modular design.

As constraint expression can be naturally embedded in spreadsheet environments, [10] describes how decision models can be based on rule families represented in Excel decision tables.

However, making it easier to state constraints during the modeling process has its limitations as the resulting model might be very inefficient to solve. Fortunately, there have been plenty of work on automating the formulation or reformulation process for constraint models to make it easier to solve. Frisch gives an invited talk on progress in constraint modelling and reformulation [11]. [12] discusses how a MiniZinc model can be transformed into a form that is easier for constraint-based local search backend solver.

[13] focuses on the modeling component which has traditionally shaped by optimization and domain experts, interacting to provide realistic results. ML techniques can help to simplify the process by exploiting the data to either create models or refine expert-designed ones. [13] also covers approaches that have been proposed to enhance the modeling process by either single constraints, objective functions, or even the whole model.

One of the ultimate goals could be that the user of a CP tool can give a description in natural language and the constraint model can be automatically and correctly generated which can then be used to calculate solutions to the problem. While this goal might be wonderful, but the reality is still far from the vision. However, there is indeed some process made in the field of automating problem acquisition. [14] provides an early attempt in this direction. [15] makes progress in extracting constraint from natural languages using a structured-output classifier.

Solution

[16] propose one more technique to the *Empirical Model Learning* (EML) technique by devising methods to embed in a CP model two types of tree-based classifiers from ML, namely DT and RF. There are many difficulties to overcome in order to make a successful embedding of ML models into a CP

model although DT and RF have many similarities to a constraint model.

While many attempts have been made to utilize the power of ML techniques to CSPs, not many of them have utilized the recent advances in DL. [17] applies DL to predict the satisfiabilities of CSPs and this is the first effective application of DL to CSPs that yields above 99.99% prediction accuracy on random Boolean binary CSPs whose constraint tightnesses or constraint densities do not determine their satisfiabilities. A deep convolutional neural network on a matrix representation of CSPs has been used to solve the problem and the asymptotic behavior of generalized model along with domain adaptation and data augmentation techniques have been applied to deal with the high cost of generating labeled training dataset.

Apart from that, many other approaches exists. [18] integrates *Deep Neural Networks* (DNNs) into a heuristic tree search procedure to decide which branch to choose next and to estimate a bound for pruning the search tree of an optimization problem. [19] promotes a RL approach for solving a classical job scheduling problem. [20] uses a Q-learning based method to minimize the total transmission cost in caching optimization. [21] employs a DNN to train the optimal scheduling algorithm to solve the problem of minimizing network energy consumption and reducing transmission delay by *opt propagation* (RProp) algorithm as the learning heuristic for supervised learning on the model shown in Figure 2.1. [22] applies neural networks trained with RL and *Constraint-Based Local Search* (CBLS) for solving CP problems. [23] explores whether a DNN can learn how to construct solutions of a CSP in an end-to-end manner without any explicit symbolic information about the problem constraints. In particular, the authors train a DNN to extend a feasible partial solution by making a single, globally consistent, variable assignment. [19] proposes a ML methods DeepRM, which translates the problem of packing tasks with multiple resource demands into a learning problem, as an alternative to manually designed heuristics for resource management. [24] applies RL methods to learn domain-specific heuristics for JSSP. Empirical results indicates that RL can offer a new method for constructing high-performance scheduling systems. [25] presents a new approximate optimization method, namely *Hybrid Imperialist Competitive Algorithm* (HICA), which is based on the imperialist competitive algorithm from Atashpaz-Gargari and Lucas (2007).

As constraint solvers can be viewed as complicated software that have to make many decisions through the solving process based on a limited amount of information and sometimes even only on a pre-defined heuristic, [26] investigates using ML to make these decisions automatically depending on the

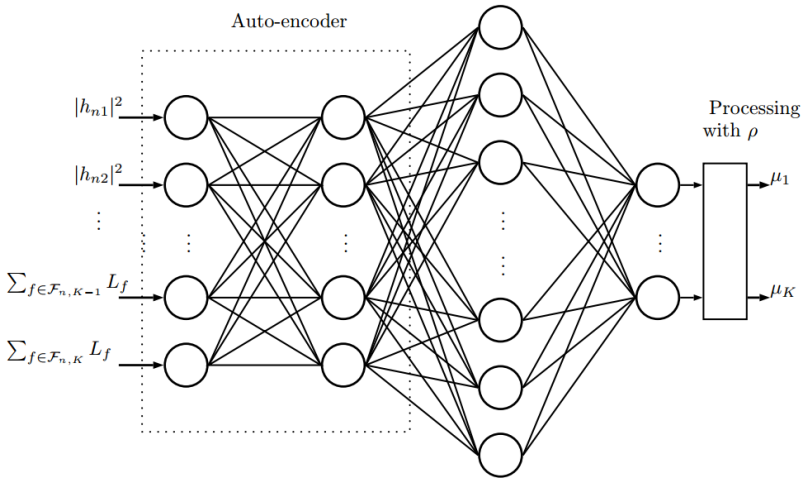


Figure 2.1: A DNN structure

problem to solve.

[27] demonstrates that Empirical Hardness Models, which can be used to predict the *runtime* of search algorithms for hard combinatorial problems, can also make accurate predictions of the *Run-Time Distributions* (RDTs) of incomplete and randomized search methods. The paper also shows that parameter settings of an algorithm can also be incorporated into a model, which can be used to to automatically adjust parameter settings on a per-instance basis in order to optimize for performance.

[28] proposes a new propagator for a set of Neuron Constraints that are employed in the context of the Empirical Model Learning technique. Specifically, a Neural Network can be embedded in a Constraint Model by simply encoding each neuron as a Neuron Constraint. A new propagator for the most common ANN structure in practice is proposed with the assumption of using sigmoid neurons in the hidden layer. But the methods can easily be extended to any differentiable activation function.

ML for algorithm selection is also an active filed of research. The algorithm selection problem is concerned with selecting the best algorithm to solve a given problem based on features and other relevant description of the problem. It is an area of research where researchers are investigating how to identify the most suitable existing algorithm for solving a given problem instead of developing new algorithms.

[29] provides a survey on algorithm selection techniques in the area of combinatorial search problems. Some of the techniques in this area have

achieved significant performance improvements.

Apart from that, [30] investigates the impact of different sets of evolved instances for building models that give predictions on algorithm selection.

Algorithm portfolios are yet another approach that can be used to improve the speed of solving a CP task. [31] proposes a general method for combining existing algorithms into new programs that are unequivocally preferable to any of the component algorithms. The method offers a computational portfolio design procedure that can be widely used. The portfolio is constructed simply by letting algorithms in the portfolio run concurrently but independently on a serial computer. And as soon as one of the algorithms finds a solution, the run terminates. The paper plays around with the setting of fractions of time for each algorithms and have some interesting results.

[32] provides an evaluation of the portfolio approach on hard combinatorial search problems. One of the interesting observations is that it could be advantageous to apply a more "risk-seeking" strategy with a high variance in *runtime*. And the experiment results testify the computational advantage of portfolio approaches when dealing with hard combinatorial optimization problems. Another interesting findings is that one can exploit the large variance in certain randomized search methods by packing them in a portfolio and still achiever better overall performance than more conservative strategies.

[33] compares the performance of off-the-shelf ML classifiers against some well-known portfolio approaches, e.g. CPHydra, ISAC, 3S, SATzilla, and SUNNY. [34] presents an automated approach for the algorithm selection pipeline.

[35] describes extensions and improvements of existing models, new families of models, and a much more thorough treatment of algorithm parameters as model inputs. While such models have been studied for complete, deterministic search algorithms, [27] promotes that *empirical hardness models*, which is models based on linear basis function regression for predicting the time an algorithm will take to solve a given problem instance, can also make accurate predictions on the run-time distributions of incomplete and randomized search methods, e.g. stochastic local search algorithms. Their work extends *empirical hardness models* significantly in the following three ways:

1. shows that the same approach can be used to predict sufficient statistics of the RTDs of incomplete, randomized algorithms, and in particular of *Stochastic Local Search* (SLS) algorithms for *Boolean Satisfiability Problem* (SAT);

2. extends empirical hardness models to include algorithm parameters in addition to features of the given problem instance;

3. explores the potential of such an approach for automatic per-instance parameter tuning. In their experiments, the tuning never hurt and sometimes resulted in substantial and completely automatic performance improvements compared to default or optimized fixed parameter settings;

Explanation

Explaining why a solution is successful might be straight forward, as one can simply check whether all the constraints are satisfied. However, sometimes this might not be enough, as the current constraint model might be incomplete, incorrect or inadequately specified and the users are more interested in getting more information about why a solution works. In other cases where multiple solutions exist, users might be interested in why a particular solution is chosen.

Recent wide adoption of ML has also renewed the interest in interpretable ML as many decision are too important to black-box techniques. One natural approach to get better explanation for a solution is to trace the solving process of the constraint program. However, such tracing generally does not work well when the solver employs search in the solving phase. [36] provides explanations of the problem solving behaviour for logic puzzles using inference-based constraint satisfaction which provides considerably more meaningful explanations than a plain trace of a search process. [37] proposes a MaxSAT-based framework, called MLIC, which allows principled search for interpretable classification rules expressible in propositional logic. The main focus of the paper is to encourage researchers in both interpretable classification and in the CP community to take it further and develop richer formulations, and bespoke solvers attuned to the problem of interpretable ML.

Not surprisingly, much of the work on explanation has been focusing on why there is no solution to the problem, which might be explained by the need of the user to understand why the constraint model does not have any solution so as to weaken or alter the constraint specification to allow solution. For example, [38] studies the problem of deriving explanations for assignments and deletions in terms of previous selections using explanation trees. [39] describe a set of tree-based tools for explaining the solving process in a user-friendly manner where users can explore the constraint model through understandable nodes in a tree architecture. Other failure explanation methods also exists with the idea of explaining intermediate failures during the constraint solving phase in order to make the process more efficient.

2.3.2 Constraint Programming for Machine Learning

While ML has demonstrate its potential in facilitating the automation of CP task, CP techniques can also be utilized to further improve ML.

For example, CP can be a flexible way to tackle data mining tasks. [40] argues that constraints from users can be helpful for itemset mining in the following three ways:

- `Preprocessing`: the dataset after preprocessing satisfy the user-defined constraints which serves as a data cleaning method;
- `Mining process`: constraints can be integrated to data mining systems to facilitate the mining process;
- `Postprocessing`: the result from the mining process can be required to satisfy a set of constraints so that any violation is removed.

[41] introduces a generic CP model for itemset mining taking into account any type of user's constraints. Empirical evaluation shows that the proposed model can handle different types of constraints on different datasets and can find the itemsets that satisfies all users constraints. [42] proposes a new propositional satisfiability based approach for mining maximal frequent itemsets that extends the one in [43] by showing that the maximal itemsets can be obtained by performing clause learning during search. [42] presents an efficient and scalable approach for computing all maximal frequent itemsets using propositional satisfiability. Experimental results on have shown that their approach is more effective compared to Eclat and DMCP,a specialized and CP-based algorithms, respectively.

Apart from its application in data mining, CP are also used in the other areas. For example, [44] presents a new approach for training ANNs using techniques for solving the CSPs.

2.4 Job-shop Scheduling Problems

In this project, we aim to construct a algorithm portfolio based on ML techniques for JSSPs.

2.4.1 Mathematical definition of Job-shop Scheduling Problems

Sequencing and scheduling is concerned with the optimal allocation of scarce resources to activities over time and it has been subject of active research in CP and in *Operations Research* (OR). [45] classifies scheduling problems according to shop environments, including single-machine, parallel machines, flow shop, no-wait flow shop, flexible flow shop, job shop, open shop.

JSSP is one of the most difficult COPs considered [46] and it is, together with a large class of intractable numerical problems, proven to be NP-hard [47], while many of its variations have been proven to be NP-complete [48], [49], [50]. The unary resource scheduling problem can be represented by the disjunctive graph. Disjunctive scheduling problems can be generally defined as the problem of scheduling n jobs $J = \{J_1, J_2, ..., J_n\}$, on a set of m unary resources $R = \{R_1, R_2, ..., R_m\}$. A job J_i consists of a set of tasks $T = \{t_1, t_2, ..., t_k\}$, where each task has a processing time and a resource that it must be processed on associated with it. A resource, which is also referred to as a machine, is exclusive and can only process one task at a time in a non-preemptive fashion.

The classical JSSP consists of scheduling m jobs to n machines. There are many variants of JSSPs, in the setting we are working with, there are sets of machines $M = \{M_1, M_2, ..., M_m\}$ and jobs $J = \{J_1, J_2, ..., J_n\}$, where a job j has m ordered operations $O_j = \{O_{J_1}, O_{J_2}, ..., O_{J_m}\}$, where each of the operations of a job should be processed on each of the m machines exactly once in an given order. The objective of JSSP is to look for a possible schedule of operations to minimize criteria such as *makespan*, which is the time elapses from the start of the first job and the end of the last job, *total weighted tardiness*, *sum of maximum earliness*, etc.

2.4.2 General Methods for Solving Job-shop Scheduling Problems

According to [45], the first systematic approach to scheduling problems was proposed in the mid-1950s. Since then, thousands of research papers aiming at the problem have been published. Different solution techniques including exact methods, heuristics, estimation methods and metaheuristics have been suggested for each JSSP variation.

The majority of methods for solving the problem use meticulously designed heuristics, which requires domain experts to come up with clever heuris-

tics, painstakingly test, and tuning of the heuristics for good performance in practice.

For example, [51] provides a statistical study of the relationship between JSSP features and the optimal *makespan*. The study includes a set of 380 carefully hand-designed features each representing a certain aspect of the scheduling problem. The 380 features can be divided into two categories: configuration features which are taken from the JSSP specification and temporal features which are concerned with the information about the solving process such as the output of a dispathing rule, heuristic applied to the problem. In the first experiment, three different methods to assess the correlation of individual features with C' (the scheduling efficiency metric) and then rank them accordingly. Temporal features, expectedly, ranked among the highest for both batches, whereas some configuration features performed well too. The second and third experiments were an effort to see which feature combinations can have a higher prediction power in the ML problem to classify instances into higher or lower than class average. ML and statistical analysis techniques have been applied to identify features with regard to the optimal *makespan*.

The result of the experiments demonstrate a test accuracy measure of around 80% on classification task of whether the *makespan* is higher or lower than the mean optimal *makespan* with a test set of 15000 randomly generated JSSP instance. Apart from that, a separate section is dedicated to offer an example on how the learned correlations can be applied in practice for feature selections in a prismatic par manufacturing setting. In the discussion, the paper also point out a future direction of applying feature combination approaches using RFs.

[52] presents a simple technique for disjunctive machine scheduling problems by combining several generic search techniques such as restarts, adaptive heuristics and solution guided branching and show that this method can match or even in some cases outperform state of the art algorithms on a number of problem types.

[53] considers the problem of scheduling correlated parallel machines to minimize *makespan* and scheduling correlated parallel machines with release times to minimize total weighted tardiness with different levels and combination of machine correlations, which is the dispersion of processing times among jobs on the same machine, and job correlations, which is the dispersion of processing times of the same job across machines. Mathematical models are used for examine machine correlation and job correlation with regard to computation results and times which reveal that the problem instances becomes harder to solve as with the increment on machine and job correlation. Apart from that, the paper also points out why some of the typical assumptions

in machine scheduling problems are over idealized and why tests performed under those assumptions might be far from reality.

Chapter 3

Implementation

3.1 Dataset Acquisition

In this section, we will introduce how do we prepare the dataset for further experiments. Specifically, there are two different types of dataset, one for CP benchmark which we will refer to as JSSP instance data, another for ML and DL which we will refer to as machine learn and DL dataset.

3.1.1 Job-shop Scheduling Problems Instance Data Acquisition and Generation

In order for a constraint model for JSSP to work, we need to send the problem specification as input to the model which we refer to as a JSSP instance. Given that our constraint model is implemented in Gecode which is an open sources C++ toolkit for building constraint based system, it make sense for us to make the JSSP instance data in a form that is easy to incorporate to our constraint model.

Specifically, we define each JSSP instance as an array of constant integer number in C++, one of the examples is shown as follows:

Listing 3.1 shows an JSSP instance named la01 where the first two number of the array specifies that this JSSP instance is concerned with a scheduling of 5 jobs of 5 operations each over 5 different machines. From line 3 to line 12 in Listing 3.1 is the requirements of the 5 different jobs. For example, line 3 states that job 0 requires 5 operations to finish where the first operation needs to be performed on machine 1 and take 21 units of time, the second operation needs machine 0 and takes 53 units of time and so on. Note that to follow the indexing schema in C++, both the index of the jobs and the machines starts

from 0.

Now that we know what a JSSP instances look like, we will introduce how we compose or generate the dataset of JSSP instances in the rest of this subsection.

```
const int la01[] = {
    10, 5, // Number of jobs and machines
    1, 21, 0, 53, 4, 95, 3, 55, 2, 34,
    0, 21, 3, 52, 4, 16, 2, 26, 1, 71,
    3, 39, 4, 98, 1, 42, 2, 31, 0, 12,
    1, 77, 0, 55, 4, 79, 2, 66, 3, 77,
    0, 83, 3, 34, 2, 64, 1, 19, 4, 37,
    1, 54, 2, 43, 4, 79, 0, 92, 3, 62,
    3, 69, 4, 77, 1, 87, 2, 87, 0, 93,
    2, 38, 0, 60, 1, 41, 3, 24, 4, 83,
    3, 17, 1, 49, 4, 25, 0, 44, 2, 98,
    4, 77, 3, 79, 2, 43, 1, 75, 0, 96
};
```

Listing 3.1: A JSSP Instance

Randomly generated 10 by 10 Job-shop Scheduling Problem instances dataset

In order to get a larger dataset with more jssp instances so that more benchmark results could be used to formulate the training and testing dataset for the ML models, we generate a dataset of 1000 instances where each JSSP instance has 10 jobs and 10 machines and the duration of each operation in each job is randomly sampled from a uniform distribution from 1 to 99 with the boundaries included. And the naming of the instances are from r000 to r999. We will refer to the dataset as `Instance_R1000` from now on.

Randomly generated 9 by 9 Job-shop Scheduling Problems instances dataset

To further enlarge the size of the benchmark dataset while still make all the benchmark to finish within a reasonable amount of time, we generate a dataset of 10000 instances of 9 jobs and 9 machines each where the duration of each operation in each job is again randomly sampled from a uniform distribution from 1 to 99 with the boundaries included. The naming of the instances are from q0000 to q9999. We will refer to the dataset as `Instance_Q10000` from now on.

The random Job-shop Scheduling Problem instances generator

To generate the dataset, we make a generator using Gecode. The generator has a set of pre-defined and changeable parameters which specifies the number of machines, the number of jobs, the uniform distribution from which the operation time is drawn. The generator is available from the following link [1]

3.1.2 Gecode Model and Benchmarking Setup

In this subsection, we are going to introduce how we construct a constraint model for solving JSSP in Gecode and how we run the benchmark on the instances dataset we have acquired earlier.

The gecode model for solving JSSPs

The CP model used for benchmarking is implemented in Gecode [9] and the approach for constructing the constraint model and constraint solver mainly follows the idea from the following paper [52]. However, it is important to note that our implementation is a sketch and does not include all techniques from the paper which could lead to less competitive performance. For details of our implementation of the model, please refer to the following link [2].

When it comes to running the benchmark for different JSSP instances given our model, the user needs to specify at least three different parameters

- The name of the heuristic to use;
- A real number between 0.0 and 1.0 indicating the level of randomness encode in finding the solutions for the constraint model with 0.0 being no randomness at all and 1.0 being completely random in parts of the program;
- The name of the JSSP instance to solve.

A typical log file for the solving process looks is shown in the following code:

```
1 sixteen-16-60-120-120/abz5-action-0.25.log
2 Probing...
3   abz5 [makespan: 1505]
```

[1] https://github.com/chschulte/gecode/blob/job-shop-experiments/examples/job-shop-generate.cpp

[2] https://github.com/chschulte/gecode/blob/job-shop-experiments/examples/job-shop.cpp

```
  abz5 [makespan: 1452]
  abz5 [makespan: 1347]
    nodes:      35338
    failures:   8
    peak depth: 710
    runtime:    1.361 (1361.000 ms)

Adjusting...
  Bounds: [868,1347]
  Bounds: [1107,1347]
  abz5 [makespan: 1227]
  Bounds: [1107,1227]
  Bounds: [1167,1227]
  Bounds: [1197,1227]
  abz5 [makespan: 1212]
  Bounds: [1197,1212]
  Bounds: [1204,1212]
  Bounds: [1208,1212]
    nodes:      2793541
    failures:   1395739
    restarts:   1597
    no-goods:   0
    peak depth: 104
    runtime:    2:00.003 (120003.000 ms)
      stopped due to time-out...

Solving...
    nodes:      1300376
    failures:   649776
    restarts:   1116
    no-goods:   0
    peak depth: 104
    runtime:    52.111 (52111.000 ms)

Found best solution and proved optimality.
```

Listing 3.2: A JSSP Solving Log

The solving process mainly contained three phases: probing, adjusting, and solving where probing examine randomly whether some values are solutions for *makespan*, adjusting applies *interval bisection*, which consists of repeatedly bisecting the interval using the value x that is in the middle of the interval and then selecting x as the new lower bound if no feasible schedule with a *makespan* of x exists and x as the new upper bound otherwise, until either a timeout is reached or the lower bound is the same as the upper bound to get better bounds for the optimal *makespan*, and solving resorts search to

the optimal *makespan* given by the last interval given in the adjusting phase. Line 1 shows the name of the log file which include information about the instance, heuristic, and tbf value combination we have used for this benchmark setting. The following lines shows the three phase of process to find solutions. Line 2 to line 9 shows information about the probing phase. Line 11 to line 28 shows information about the interval bisection method to adjust the lower bound and the upper bound for the optimal *makespan*. Line 30 to line 36 is the solving phase to find the optimal *makespan*. Line 38 indicate that the optimal *makespan* has been found.

Benchmarking methods

We perform benchmark on both datasets: namely `Instance_R1000` and `Instance_Q10000`. For each JSSP instance, we run test with a combination of three different heuristics ("action", "afc", and "chb") and three different tbf values (0.0, 0.1, 0.25) which lead to 9 different benchmark results for each JSSP instance we have. The results are stored in separate log files which can be processed later to form the training and testing data for ML models.

Hardware and software configurations

The computations of for solving the JSSP instances are performed on the machine with the following specification

- CPU: Intel Xeon E-2186G
- Memory: 4 x 16 GB DDR4 2666 MHz ECC
- Operating System: Windows 10 Pro for Workstations

The CP model for JSSP is given as a Gecode model and solved with "GECODE 6.1.1".

Constraint Programm Settings

Note that while Gecode native provide multi-threading to speed up the solution finding process, in all of our settings to acquire the benchmark dataset, we only use a single thread. The main reason behind is that we use running time as the major measurement of computation cost, which we treat as the hardness to solve the problem, and the scale of speed up in terms of the number of threads in not completely problem independent. Thus, to keep a fair comparison of

hardness among different JSSP instances, we choose to use a single thread across all the benchmark.

For the R instances, we set the time out for probing to be 60 seconds, 240 seconds for adjusting, and no time out for the probing so that we always find the optimal solution. For the Q instances, probing times out after 30 seconds, adjusting times out after 15 seconds, and there is no time out for the solving phase.

3.1.3 Machine Learning and Deep Learning Dataset Generation

In the previous subsection, we introduce how we acquire all the benchmark data for different JSSP instances in our three different JSSP instance dataset, in this subsection, we will go through how we generate the dataset for the ML and DL tasks from the JSSP instances and the corresponding benchmark results.

Features

In order to train ML and DL models for predicting the *makespan* and *runtime* of JSSP instances, features that represent certain characteristics of the problem should be developed. As an example, [51] manually develops a set of 380 mostly novel features for ML models to predict the optimal *makespan*.

To this end, we also develop the following features from the JSSP instance itself or the corresponding benchmark log whose names and meanings are shown in the following list:

- `Heuristic`: the name of the heuristic used for the benchmark;
- `Instance`: the name of the JSSP instance used for the benchmark;
- `tbf`: the value of tbf;
- `first_LB`: the first output of lower bound on *makespan* in the benchmark log;
- `first_UB`: the first output of upper bound on *makespan* in the benchmark log;
- `first_makespan`: the first output of *makespan* in the benchmark log;

	Heuristic	Instance	adjusting_time_ms	first_LB	first_UB	first_makespan	makespan	probing_progress	probing_time_ms	runtime_ms	solving_time_ms	tbf	num_of_jobs	num_of_machines	max_operation_time	max_machine_load	r_first_makespan_bound
4297	afc	r477	221.0	628	4762	1151	725.0	0.276281	219.0	440.0	0.0	0.10	10	10	603	630	0.126512
5672	action	r630	14443.0	646	4736	1137	780.0	0.249780	671.0	15114.0	0.0	0.25	10	10	582	620	0.120049
7060	afc	r784	548.0	718	5383	1239	829.0	0.267151	546.0	1094.0	0.0	0.10	10	10	624	742	0.111683
4194	action	r466	11.0	712	5046	1116	858.0	0.178315	223.0	234.0	0.0	0.00	10	10	708	630	0.093216
5252	afc	r583	775.0	672	4909	1148	825.0	0.180314	746.0	1521.0	0.0	0.25	10	10	673	581	0.112344

Figure 3.1: A sample of five items for R1000 instances

- `probing_progress`: the ratio between the difference of the first and the last *makespan* during probing divided by the first *makespan* during probing. This feature captures how much progress can be made during the probing phase which may serve as feature reflecting the hardness of the given JSSP problem;
- `probing_time_ms`: the time spent during probing phase in milliseconds;
- `adjusting_time_ms`: the time spent during adjusting phase in milliseconds;
- `solving_time_ms`: the time spent during solving pahse in milliseconds;
- `runtime_ms`: the arithmetic sum of probing_time_ms, adjusting_time_ms, and solving_time_ms;
- `num_of_jobs`: the number of jobs for a given JSSP;
- `num_of_machines`: the number of machines for a given JSSP;
- `max_operation_time`: the maximal number of time units of the sum of all operations belonging one job among all jobs;
- `max_machine_load`: the maximal number of time units of the sum of all operations belonging to one machine among all machines;
- `r_first_makespan_bound`: $\frac{first\_makespan - first\_LB}{first\_UB - first\_LB}$

Dataset format

As for the organization of the output dataset, we packed the preprocessed data for our three different instances dataset into three different objects of type "pandas.Dataframe" where each row in the dataframe contains the features that we manually designed. Figure 3.1 gives an example of 5 rows randomly selected from the dataframe for R instances.

Instances matrix

As for the data regarding the raw JSSP instances, we simply convert the JSSP specification into a two-dimensional matrix without the two values specifying how many machines and how many jobs are required for the JSSP instances as these two values can easily be acquired from the dimensions of the matrix and more importantly padding of meaningless numbers will be introduced to the two-dimensional matrix if we were to include these two numbers. As an example, the instance matrix of JSSP instance "la01" is shown as follows:

$$\begin{bmatrix} 1 & 21 & 0 & 53 & 4 & 95 & 3 & 55 & 2 & 34 \\ 0 & 21 & 3 & 52 & 4 & 16 & 2 & 26 & 1 & 71 \\ 3 & 39 & 4 & 98 & 1 & 42 & 2 & 31 & 0 & 12 \\ 1 & 77 & 0 & 55 & 4 & 79 & 2 & 66 & 3 & 77 \\ 0 & 83 & 3 & 34 & 2 & 64 & 1 & 19 & 4 & 37 \\ 1 & 54 & 2 & 43 & 4 & 79 & 0 & 92 & 3 & 62 \\ 3 & 69 & 4 & 77 & 1 & 87 & 2 & 87 & 0 & 93 \\ 2 & 38 & 0 & 60 & 1 & 41 & 3 & 24 & 4 & 83 \\ 3 & 17 & 1 & 49 & 4 & 25 & 0 & 44 & 2 & 98 \\ 4 & 77 & 3 & 79 & 2 & 43 & 1 & 75 & 0 & 96 \end{bmatrix}$$

Makespan and runtime distributions

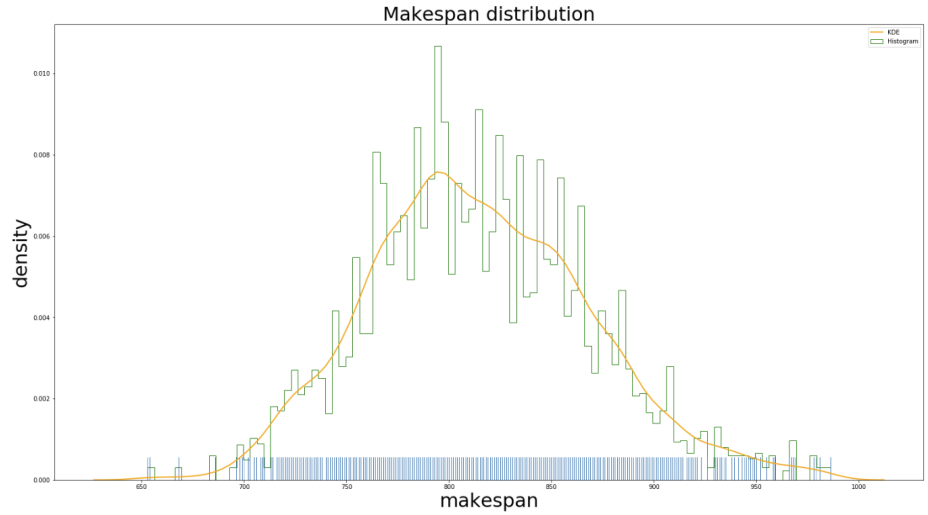
Figure 3.2a shows the *makespan* distribution for R instances and Figure 3.2b for Q instances. As it can be seen from the figures, the two distributions follows normal distributions.

The distribution for *runtime* follows a long tail distribution which is a typical distribution for running time of computer programs and benchmarks, as it can be seen from Figure 3.3a for R instances and 3.3b for Q instances.

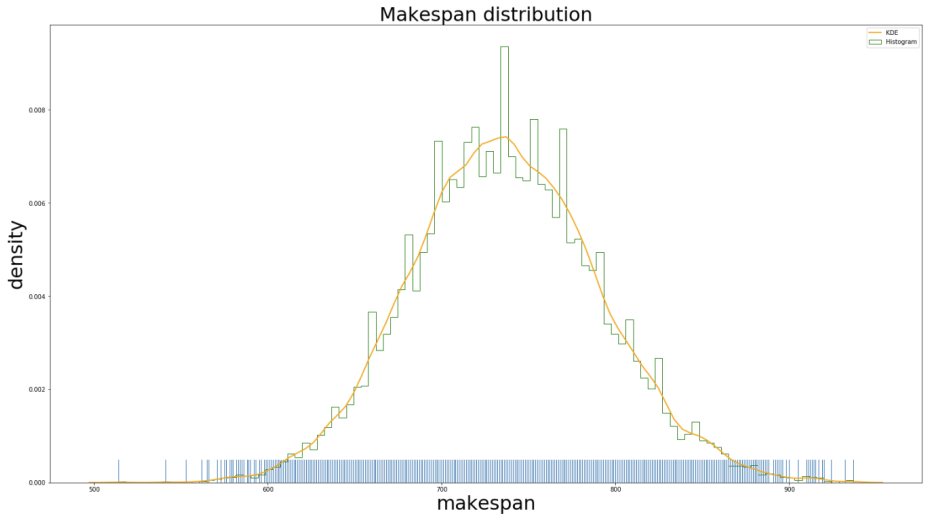
3.2 Traditional Machine Learning Models

3.2.1 Experimentation Setup

In this section we are going to introduce the experiment setup for training and evaluating the traditional ML models and the measurement for model performance.

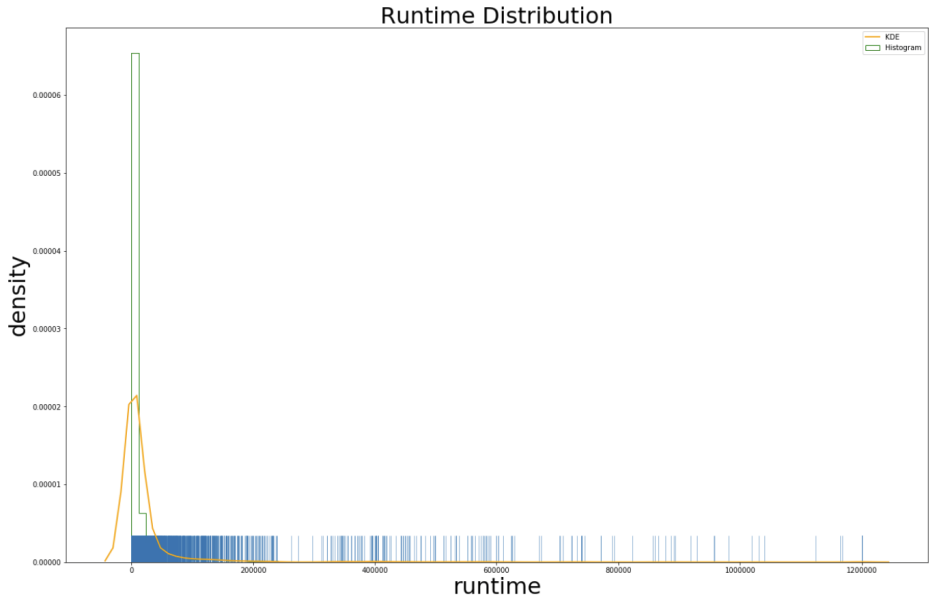


(a) Makespan distribution for R instances

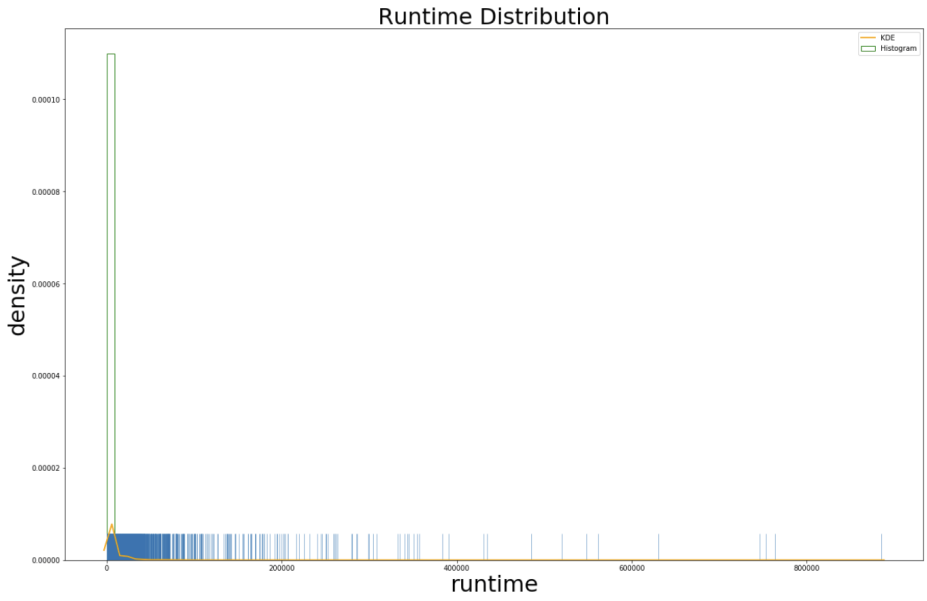


(b) Makespan distribution for Q instances

Figure 3.2: Makespan distribution



(a) Runtime distribution for R instances



(b) Runtime distribution for Q instances

Figure 3.3: Runtime distribution

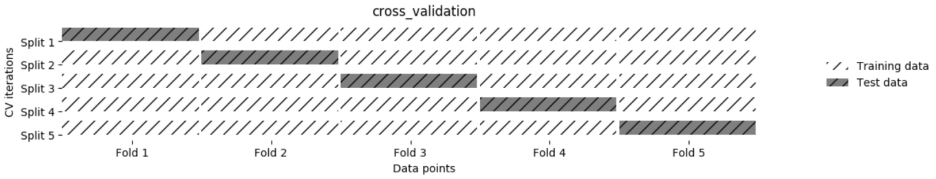


Figure 3.4: An example of five-fold cross validation

Performance measurement

To have a quantitative evaluation of the ML model, we need a measurement for performance. We employ one of the most common measurement for regression which is the MSE between the predictions and the true values of the data points whose mathematical definition is shown in the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\{y_i - \hat{y}_i\}^2 \tag{3.1}$$

where n is the number of items in the dataset, y_i is the true value of the i_{th} data point, and $\hat{y}_i$ is the predicted value of the i_{th} data point.

Cross validation

Cross validation is a method to estimate generalization performance of a model where k-fold cross validation randomly split the data into k parts of equal sizes which is referred as folds. After the folds are made, a five-fold cross validation first takes fold 1 as the test set and folds 2-5 as the training set. Then, it takes fold 2 as the test set and folds 1, 3, 4, and 5 as the test set. Such an iteration goes on until each fold is taken as the test set for exactly once. Finally, the mean of the five evaluation scores are returned as the evaluation score from cross validation. An example of five-fold cross validation is shown in Figure 3.4.

Software and library configurations

For performing experiments on traditional ML models, we use the scikit-learn [54] version “0.20.0”. And the experiments are performed on a computer of “Ubuntu 18.04.1 LTS” with “4.15.0-51-generic” Linux kernel.

3.3 Deep Learning models

3.3.1 Models

In this subsection, we will introduce the DL models that we have applied to solve the regression problem for *makespan* and *runtime*. Note that we employed the same model architecture for *makespan* and *runtime* regression but they are trained separately. Specifically, we have trained four different models where Vanilla CNN model and Lenet 5 model takes JSSP instance matrix as input and the other two hybrid models take an additional vector as input including the values for "first_LB", "first_UB", "first_makespan", "probing_progress", "tbf", "num_of_jobs", "num_of_machines", "max_operation_time", "max_machine_load", and "r_first_makespan_bound", and an onehot encoding of which of the three heuristic to use.

Vanilla Convolutional Neural Network model

The vanilla CNN model we apply here perhaps one of the simplest model one can come up with which contains several 2-d convolutional layers, a flatten layer, and a few fully connected layers. Figure 3.5 shows a summary of the model. Note that as the dimension of our input data is significantly smaller than the dimension of modern image data, we do not employ any pooling layer in the model so that more convolutional layers can be applied before running out of dimensions.

Le-Net 5 model

The LeNet-5 architecture is perhaps one of the most known CNN architecture proposed by Yann LeCun in 1998 [55] and widely used for the handwritten digit recognition on MNIST [56] dataset. Here we build a model that is similar to the LeNet-5 architecture whose summary can be found in Figure 3.6. The major differences are:

- We remove the average pooling layer due to the fact that our input data is of dimension 9×18 instead of 32×32.
- We do not account for the complex connection between layer C3 and layer S2. See table 1 in [55] for more details.
- We do not employ the complex mechanism for measuring how much the input image belongs to each class as we are working with a regression problem now.

Layer (type)	Output Shape	Param #
aux_input (InputLayer)	(None, 9, 18, 1)	0
conv2d_1 (Conv2D)	(None, 7, 16, 32)	320
conv2d_2 (Conv2D)	(None, 5, 14, 32)	9248
conv2d_3 (Conv2D)	(None, 3, 12, 16)	4624
conv2d_4 (Conv2D)	(None, 1, 10, 16)	2320
flatten_1 (Flatten)	(None, 160)	0
dense_7 (Dense)	(None, 32)	5152
dense_8 (Dense)	(None, 16)	528
dense_9 (Dense)	(None, 1)	17

Total params: 22,209
Trainable params: 22,209
Non-trainable params: 0

Figure 3.5: Summary for vanilla CNN model

- Other modifications to the network structures to avoid negative dimensions.

Hybrid vanilla Convolutional Neural Network model

The hybrid vanilla CNN model takes an auxiliary input of features in addition to JSSP instance matrix. The main purpose for testing the hybrid model is to see whether the additional information can further improve the performance of the model or at least fasten the training process. A detailed summary of the model can is shown in Figure 3.7 and a visualization can be found in Figure 3.10.

Hybrid Le-Net 5 model

The hybrid Le-Net 5 model is designed with same idea as in the case of hybrid vanilla CNN model. A detailed summary of the model can is shown in Figure 3.9 and a visualization can be found in Figure 3.10.

Layer (type)	Output Shape	Param #
aux_input (InputLayer)	(None, 9, 18, 1)	0
conv2d_1 (Conv2D)	(None, 7, 16, 6)	60
depthwise_conv2d_1 (Depthwis	(None, 7, 16, 6)	12
conv2d_2 (Conv2D)	(None, 5, 14, 16)	880
depthwise_conv2d_2 (Depthwis	(None, 5, 14, 16)	32
conv2d_3 (Conv2D)	(None, 3, 12, 120)	17400
flatten_1 (Flatten)	(None, 4320)	0
dense_7 (Dense)	(None, 32)	138272
dense_8 (Dense)	(None, 16)	528
dense_9 (Dense)	(None, 1)	17

Total params: 157,201
Trainable params: 157,201
Non-trainable params: 0

Figure 3.6: Summary for LeNet mockup model

Layer (type)	Output Shape	Param #	Connected to
aux_input (InputLayer)	(None, 9, 18, 1)	0	
conv2d_1 (Conv2D)	(None, 7, 16, 32)	320	aux_input[0][0]
main_input (InputLayer)	(None, 13)	0	
conv2d_2 (Conv2D)	(None, 5, 14, 32)	9248	conv2d_1[0][0]
dense_1 (Dense)	(None, 64)	896	main_input[0][0]
conv2d_3 (Conv2D)	(None, 3, 12, 16)	4624	conv2d_2[0][0]
dense_2 (Dense)	(None, 32)	2080	dense_1[0][0]
conv2d_4 (Conv2D)	(None, 1, 10, 16)	2320	conv2d_3[0][0]
dense_3 (Dense)	(None, 16)	528	dense_2[0][0]
flatten_1 (Flatten)	(None, 160)	0	conv2d_4[0][0]
concatenate_1 (Concatenate)	(None, 176)	0	dense_3[0][0] flatten_1[0][0]
dense_4 (Dense)	(None, 32)	5664	concatenate_1[0][0]
dense_5 (Dense)	(None, 16)	528	dense_4[0][0]
dense_6 (Dense)	(None, 1)	17	dense_5[0][0]

Total params: 26,225
Trainable params: 26,225
Non-trainable params: 0

Figure 3.7: Summary for hybrid vanilla CNN model

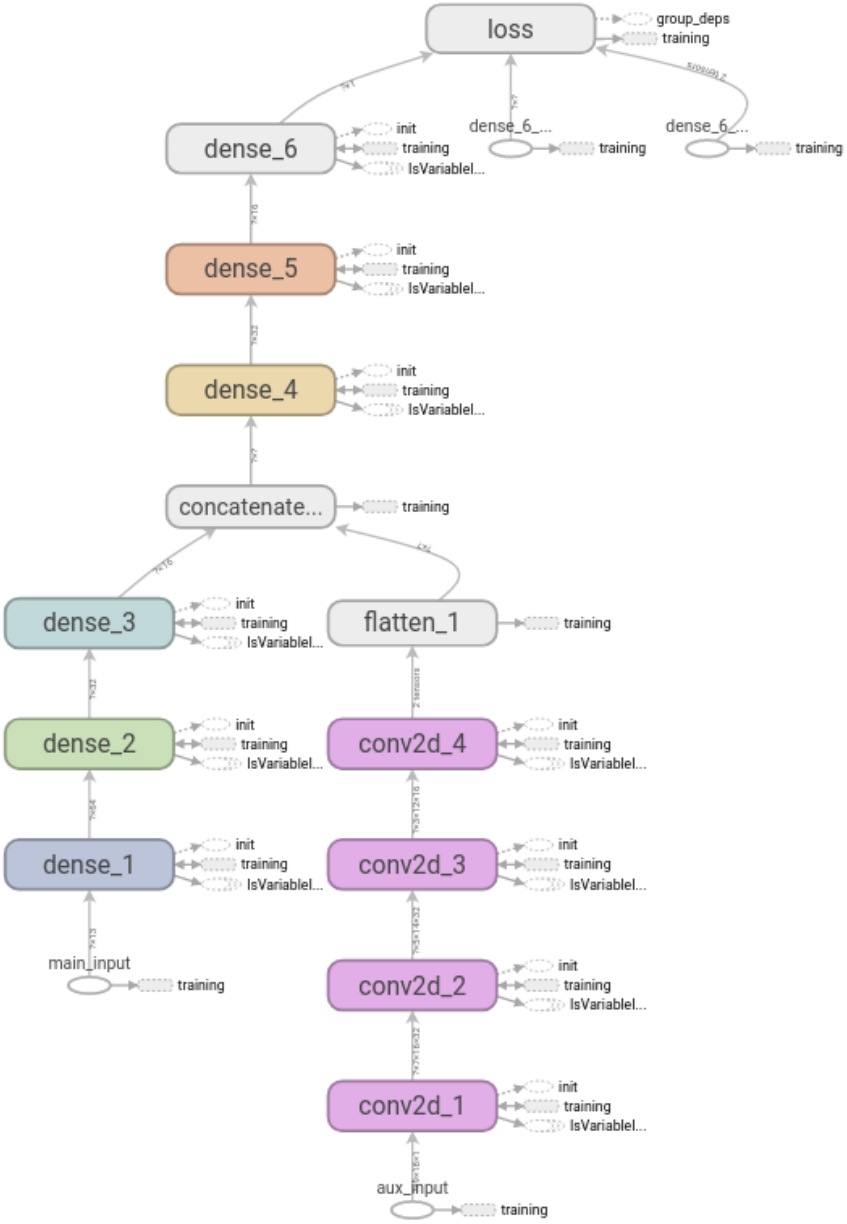


Figure 3.8: Visualization for hybrid vanilla CNN model

Layer (type)	Output Shape	Param #	Connected to
aux_input (InputLayer)	(None, 9, 18, 1)	0	
conv2d_1 (Conv2D)	(None, 7, 16, 6)	60	aux_input[0][0]
depthwise_conv2d_1 (DepthwiseCo	(None, 7, 16, 6)	12	conv2d_1[0][0]
main_input (InputLayer)	(None, 13)	0	
conv2d_2 (Conv2D)	(None, 5, 14, 16)	880	depthwise_conv2d_1[0][0]
dense_1 (Dense)	(None, 64)	896	main_input[0][0]
depthwise_conv2d_2 (DepthwiseCo	(None, 5, 14, 16)	32	conv2d_2[0][0]
dense_2 (Dense)	(None, 32)	2080	dense_1[0][0]
conv2d_3 (Conv2D)	(None, 3, 12, 120)	17400	depthwise_conv2d_2[0][0]
dense_3 (Dense)	(None, 16)	528	dense_2[0][0]
flatten_1 (Flatten)	(None, 4320)	0	conv2d_3[0][0]
concatenate_1 (Concatenate)	(None, 4336)	0	dense_3[0][0] flatten_1[0][0]
dense_4 (Dense)	(None, 32)	138784	concatenate_1[0][0]
dense_5 (Dense)	(None, 16)	528	dense_4[0][0]
dense_6 (Dense)	(None, 1)	17	dense_5[0][0]

Total params: 161,217
Trainable params: 161,217
Non-trainable params: 0

Figure 3.9: Summary for hybrid LeNet mockup model

3.3.2 Experiment setup

Performance measurement

We choose the same performance measurement as in the case of traditional ML experiments, namely MSE whose definition is given by equation (3.1).

Training setup

When it comes to train the DL models, we have only used the ML dataset generated from the `Instance_Q10000` dataset as it contains ten times more instances than the ML dataset generated from the `Instance_R1000` dataset which can be used to better train the networks.

Since the dataset is reasonably large, which means the result across different train test dataset split are more stable than smaller datasets, and that the computational cost for a ten-fold cross validation is simply too much to pay, we employ a traditional train test split method where 80% of the whole dataset is used for training purpose and the other 20% is used for testing and evaluation. The data that is used for training is further split into a training dataset (70%) and a validation dataset (30%).

The batch size of each model is set to 4000 and we train each model until the model converges which takes from 3000 to 10000 epochs with the Adam

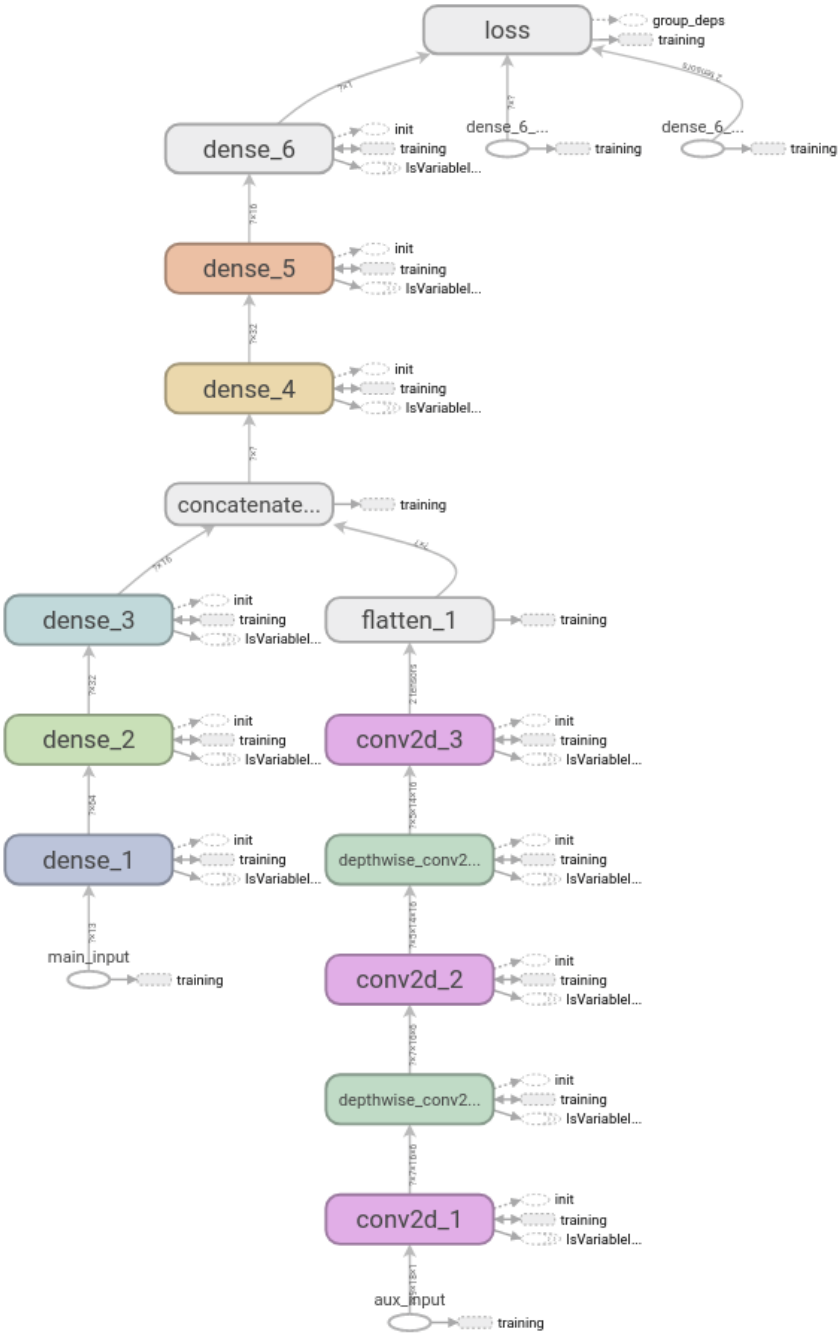


Figure 3.10: Visualization for hybrid LeNet mockup model

optimizer [57].

Software and library configurations

For performing experiments on neural networks, we use the Keras [58] version "2.2.4" with a TensorFlow [59] backend of version "1.12.0". And the experiments are performed on a computer of "Ubuntu 18.04.1 LTS" with "4.15.0-51-generic" Linux kernel.

Chapter 4

Results and Discussion

In this chapter we first show some results from our experiments and then discuss some of the interesting findings we have observed. Note that as our main objective is to investigate the performance of different ML and DL models, we focus on the difference of performance of different models instead of trying to get the best performance of every single model through fine tuning or other hyperparameter optimization techniques.

4.1 Results of traditional Machine Learning models

In this section, we will show the results of experiments on traditional ML models with both the R instances and the Q instances for both *makespan* and *runtime* regression.

To evaluate the performance of the family of linear models (linear regression, ridge regression, and lasso regression), ten-fold cross validation is performed on the whole dataset and the mean and variance of the MSE are reported. As for SVR, DT regression and RF regression, as the ten-fold cross validation is much more expensive to afford computationally, we use the traditional train test split method where 67% of the whole dataset is used for training and 33% of the whole dataset is used for testing purpose. Also, the input features are standardized through sklearn.preprocessing.StandardScaler which is trained only on the training set to avoid the leakage of information from test set to the training phase.

Table 4.1: Makespan regression - Linear models

Model	R_MSE_Avg	R_MSE_Var	Q_MSE_Avg	Q_MSE_Var
Linear	407.11	4987.66	403.40	230.93
Ridge	407.02	4974.56	405.13	223.72
Lasso	407.69	5089.36	406.36	223.38

Table 4.2: Makespan regression - Support Vector Regression

SVR kernel	R_MSE_Train	R_MSE_Test	Q_MSE_Train	Q_MSE_Test
linear	404.40	395.58	404.41	402.77
poly	823.00	877.74	444.77	445.43
rbf	615.89	604.47	400.11	399.57
sigmoid	1125.64	1094.16	124423.05	129020.02

4.1.1 Makespan

The experiment results of the linear model family is shown in Table 4.1 where "R_MSE_Avg" represents the average of MSEs of the ten-fold cross validation on R instances and "Q_MSE_Var" represents the variance of MSEs of the ten-fold cross validation score on Q instances.

The regression results for *makespan* with different kernel options is shown in Table 4.2 where "R_MSE_Train" represents the MSE of R instances on the training set and "Q_MSE_Test" represents the MSE of Q instances on the testing set.

The regression results for *makespan* with DT and RF is shown in Table 4.3 where "R_MSE_Train" represents the MSE of R instances on the training set and "Q_MSE_Test" represents the MSE of Q instances on the testing set.

DT and RF

4.1.2 Runtime

The result of regression for *runtime* follows the same organization of tables and naming schema for columns as for *makespan* regression. Specifically, the result for linear model family is shown in Table 4.4, the result for SVR with different kernels is shown in Table 4.5, and the result for DT regression and RF regression is shown in Table 4.6.

Table 4.3: Makespan regression - Decision Tree and Random Forest

Model	R_MSE_Train	R_MSE_Test	Q_MSE_Train	Q_MSE_Test
Decision Tree	0.00	0.17	0.00	0.00
Random Forest (max_depth=10, n_estimators=100)	36.136	46.031	202.360	221.983
Random Forest (max_depth=80, n_estimators=100)	0.143	0.565	0.121	0.775
Random Forest (max_depth=20, n_estimators=2)	1.319	2.952	3.963	8.006

Table 4.4: Runtime regression - Linear models

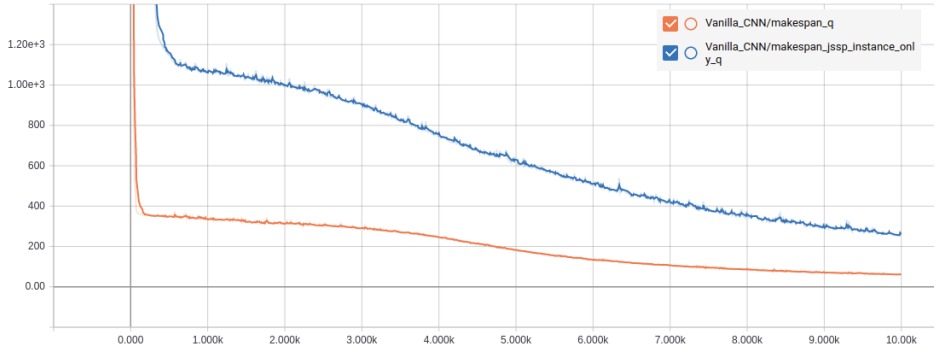
Model	R_MSE_Avg	R_MSE_Var	Q_MSE_Avg	Q_MSE_Var
Linear	7.52×10^9	1.52×10^{19}	1.16×10^8	5.58×10^{15}
Ridge	7.51×10^9	1.54×10^{19}	1.16×10^8	5.58×10^{15}
Lasso	7.51×10^9	1.53×10^{19}	1.17×10^8	5.61×10^{15}

Table 4.5: Runtime regression - Support Vector Regression

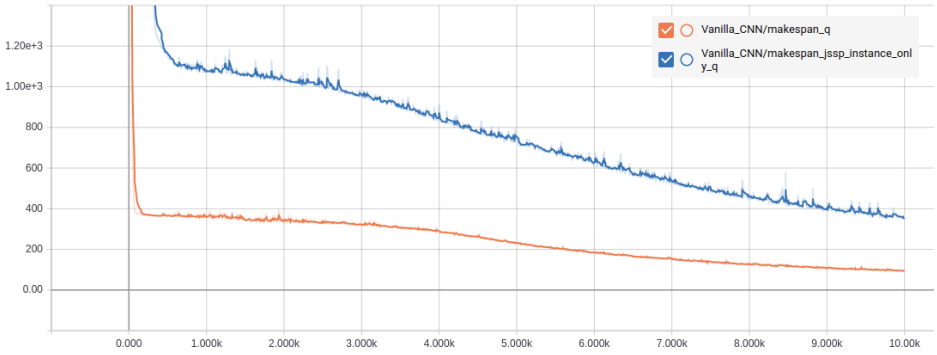
SVR kernel	R_MSE_Train	R_MSE_Test	Q_MSE_Train	Q_MSE_Test
linear	8.40×10^9	8.15×10^9	1.24×10^8	1.11×10^8
poly	8.43×10^9	8.18×10^9	1.24×10^8	1.12×10^8
rbf	8.43×10^9	8.18×10^9	1.24×10^8	1.12×10^8
sigmoid	8.43×10^9	8.18×10^9	1.24×10^8	1.12×10^8

Table 4.6: Runtime regression - Decision Tree and Random Forest

Model	R_MSE_Train	R_MSE_Test	Q_MSE_Train	Q_MSE_Test
Decision Tree	0.000	2.01×10^9	0.00	2.38×10^7
Random Forest (max_depth=60, n_estimators=150, min_samples_leaf=2)	1.19×10^9	2.19×10^9	1.98×10^7	3.34×10^7
Random Forest (max_depth=60, n_estimators=150, min_samples_leaf=1)	3.26×10^8	1.70×10^9	5.58×10^6	2.17×10^7
Random Forest (max_depth=60, n_estimators=150, min_samples_leaf=1, max_feature=0.5)	3.27×10^8	1.65×10^9	5.74×10^6	2.12×10^7
Random Forest (max_depth=10, n_estimators=1000, min_samples_leaf=1)	1.89×10^9	2.74×10^9	1.76×10^7	2.93×10^7
Random Forest (max_depth=10, n_estimators=1000, min_samples_leaf=1, max_features=0.5)	1.82×10^9	2.56×10^9	2.08×10^7	3.34×10^7
Random Forest (max_depth=10, n_estimators=1000, min_samples_leaf=1, max_features=0.1)	2.50×10^9	4.36×10^9	4.34×10^7	6.31×10^7



(a) Training Loss



(b) Validation Loss

Figure 4.1: Makespan regression with vanilla CNN

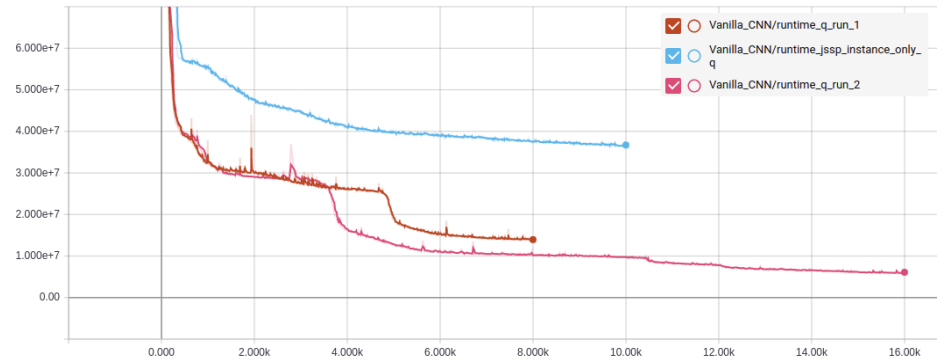
4.2 Results of Deep Learning models

In this section, we will show the result of experiments of *makespan* and *runtime* regression on Q instances. The visualization of the training and validation loss is generated from TensorBoard tool from TensorFlow [59] with a smoothing factor of 0.6.

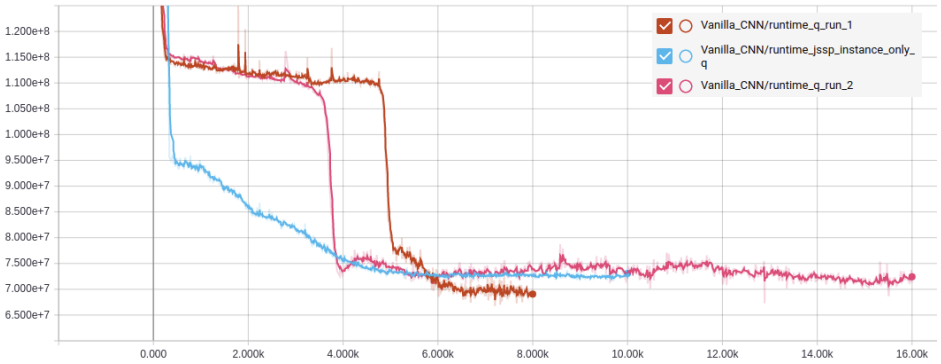
4.2.1 Models with vanilla Convolutional Neural Network

The result of *makespan* regression of the vanilla cnn model is shown in Figure 4.1 where "Vanilla_CNN/makespan_q" shows the result of hybrid model and "Vanilla_CNN/makespan_jssp_instance_only_q" shows the result of the model with only the JSSP instance matrix as input.

The result of *runtime* regression of the vanilla cnn model is shown in Figure 4.2 where "Vanilla_CNN/runtime_q_run_1" shows the result of the first run of



(a) Training Loss



(b) Validation Loss

Figure 4.2: Runtime regression with vanilla CNN

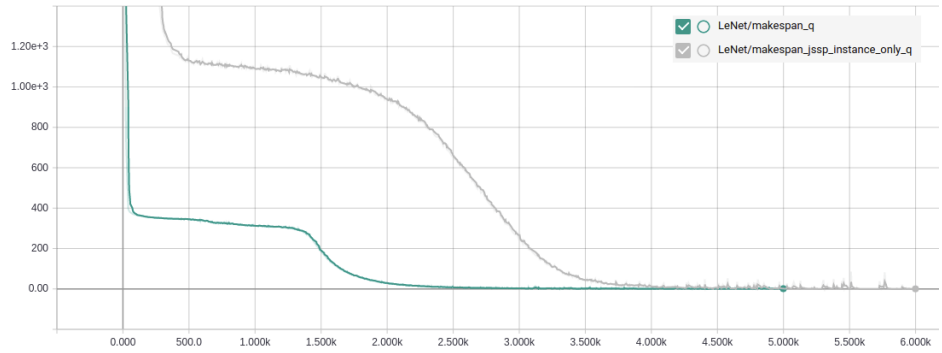
hybrid model, “Vanilla_CNN/runtime_q_run_2” shows the result of the second run of hybrid model, and “Vanilla_CNN/runtime_jssp_instance_only_q” shows the result of the model with only the JSSP instance matrix as input.

As for the performance on the test dataset, the hybrid CNN model achieves a MSE of 93.87 on the test set for predicting *makespan* after 10000 epochs of training and 5.19×10^7 on the test set for predicting *runtime* after 10000 epochs of traning.

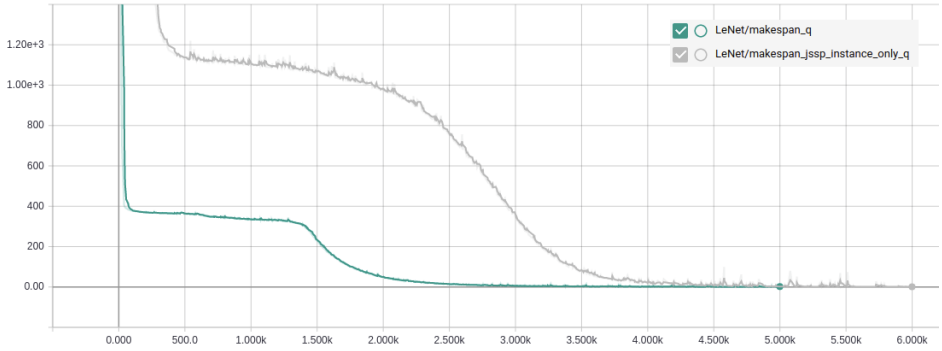
The CNN model that takes only the JSSP instance matrix as input achieves a MSE of 355 on the test set for predicting *makespan* after 10000 epochs of traning and 8.43×10^7 on the test set for predicting *runtime* after 10000 epochs of training.

4.2.2 Models with LeNet mockup

The result for *makespan* regression using the LeNet mockup model is shown in Figure 4.3 and the *runtime* regression result is shown in Figure 4.4. Note that



(a) Training Loss



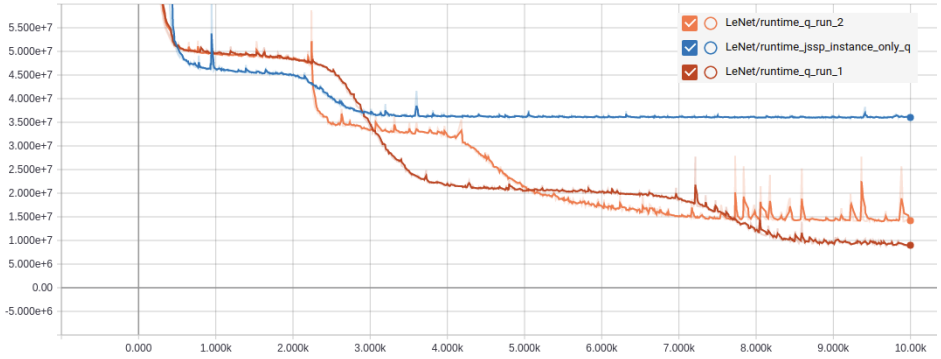
(b) Validation Loss

Figure 4.3: Makespan regression with LeNet mockup

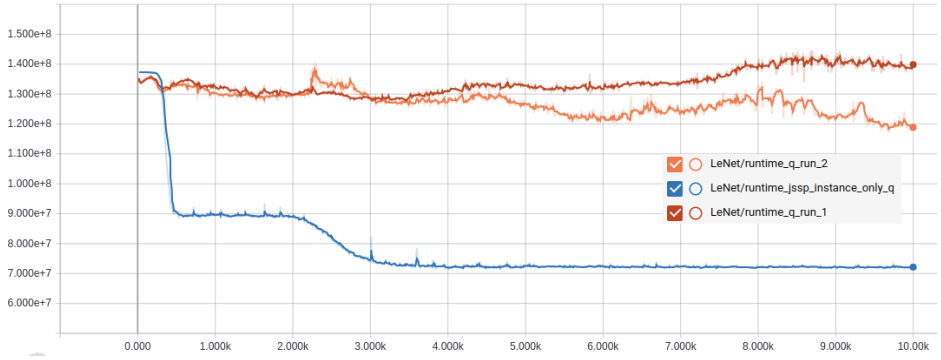
the naming schema is the same where one training record with "jssp_instance_only" indicates that the result is from the model that only takes the JSSP instance matrix as input and a training record without such characters is from a hybrid model that also the one dimensional vector as input in addition to the instance matrix.

As for the performance on the test dataset, the hybrid LeNet mockup model achieves a MSE of 2.85 on the test set for predicting *makespan* after 5000 epochs of training and 1.11×10^8 on the test set for predicting *runtime* after 10000 epochs of traning.

The LeNet mockup model that takes only the JSSP instance matrix as input achieves a MSE of 0.74 on the test set for predicting *makespan* after 6000 epochs of traning and 8.43×10^7 on the test set for predicting *runtime* after 6000 epochs of training.



(a) Training Loss



(b) Validation Loss

Figure 4.4: Runtime regression with LeNet mockup

4.3 Discussion of different results

As can be seen from the result in Table 4.1, the performance of predicting *makespan* of different linear models do not differ much from each other with a average MSE at around 400. While increase the number of instances does not further decrease the average of MSE, it does help the variance of the MSE in cross validation to decrease dramatically which can be a sign that the model is becoming more stable with the larger dataset. The same observation also holds for *runtime* regression (see Table 4.4) with the linear model family.

As for SVR, the result for *makespan* and *runtime* might be leading to different conclusions. From Table 4.2, it seems that the choice of kernel affects the regression performance and some kernels, e.g. "sigmoid", are more sensitive to the size of the dataset while other kernels might be less affected. However, Table 4.5 tells a different story where the choice of kernel does not matter much in terms of the result and an increase of the data volume leads to a performance gain. One guess of the reason behind is that some kernels are better at approximating the makspan than others while for *runtime* regression, which we consider to be harder than the *makespan* regression as can be seen from later examples, all the kernels approximate the distribution equally bad so that performances do not differ much.

As for DT regression and RF regression, as can be seen from Table 4.3 and 4.6, a single DT without any regularization shows a typical sign of overfitting by remembering the almost the whole training data through its complex branches and leaves. A carefully chosen RF regressor can achieve on par performance as a single overfitting DT but may require some efforts for hyperparameter tuning through methods like grid search or more advanced techniques which might be time consuming. However, given the performance gain, both in *makespan* and *runtime* regression, compared to linear models and support vector regressors, we would like to argue that such effort is worth it.

As for the DL models, all of the four models have really good performance with the LeNet mockup hybrid model being extremely efficient both in training and predicting the *makespan*, and the hybrid CNN model being good at prediciting the *runtime* at least compared to the performance of DT and RF in these settings.

Another interesting observation we have found during the training phase is that, most of the time the model that took the JSSP instance matrix as the only input is sufficient to achieve the same performance at its corresponding hybrid models which can be seen from Figure 4.3 and the trend is also reflected in Figure 4.1. This could potentially be the sign that the RFs that we have

applied to the problem can learn the important features from the JSSP instance matrix alone without the input of domain knowledge. Also, this indicates that the features we have developed are actually important features in terms of the value that we want to estimate for the simple reason that using these features as auxiliary input can greatly speed up the training process.

Through different tasks, we have found that different CNN architectures might have different performance on the same task and some may also be less computationally expensive than others to train. One of the reasons behind might be that some special design in the CNN architecture might be better at the learning problem. For example, the depth wise convolutional layer might be at least one of the reasons why our LeNet mockup models are easier to train.

Finally, we would like to point out that in Figure 4.2 and 4.4 the performance of the model is quite unstable. One the reason behind could be that the *runtime* for solving a given JSSP instance involves randomness that might not yet captured in our training dataset as the dataset only contains a single result of *runtime* for each unique combination of features. This might be an interesting directiong to look into.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis project, we have demonstrate the potential of ML and DL models to predict the optimal *makespan* and the *runtime* of solving a JSSP instance based on the JSSP instance matrix and other hand-designed features within an acceptable range of error and within an reasonable amount of time. In particular, we have found that DT and RF from traditional ML models to have good performance in predicting *makespan* and *runtime* with the hand-designed features and that DL models can learned important features automatically from the raw JSSP intances.

For predicting the makespan (unit: machine time unit), the best Random Forest regression model achieves a Mean Squared Error of 0.78 on the test set. The best Deep Learning model achieves a Mean Squared Error of 0.74 on the test set. For predicting the solving time (unit: millisecond) of a Job-shop Scheduling Problem, the best Random Forest regression model achieves a Mean Squared Error of 2.12×10^7 on the test set. The best Deep Learning model achieves a Mean Squared Error of 5.19×10^7 on the test set.

Finally, we would also like to mention that the time used for training and predicting model are also reasonable. The Random Forest regressor takes a few minutes to train on a dataset with 90000 records on a modern computer with 4 CPU cores and the time used for predicting is at a level of seconds. The Deep Leaning models take about an hour or two to train depending on the exact model on a modern desktop with a single GTX 1080 GPU and the predicting time on the test set is within 1 second.

5.2 Future Directions

In this section, we identify several aspects of this thesis project that can be expanded upon and improved.

Encode the regression models into Gecode

Perhaps, one of the most obvious and interesting directions that one could look into is to encode the ML and DL model for predicting the *makespan* into the Gecode program and see if such a combination actually decrease the amount of time needed for getting the optimal *makespan* of a given JSSP instances.

Explore the influence of different loss functions

In this thesis project, we choose Mean Squared Error as the loss function for all of over prediction models as this is the most common choice of loss functions for regression problems. In the future, it would be interesting to explore whether the choice of different loss function will affect evaluation of the performance across different models.

Make the model compact

While the computational time is relative acceptable in our training setting which have access to GPU for training the network, when deploying the DL models in production it could very well be the case that the program can only rely on CPU for the prediction. Thus, making the model compact could reduce the running time of the model during inference.

Explore different architectures

As we have observed from the experiments, different DL architectures might be good at different tasks. Thus, it might be interesting to further explore other DL architectures for potential performance gain. Up till now, we treat the JSSP instances as two-dimensional matrices and apply CNN models for the regression task. This process is quite similar to using CNN for processing images.

Apart from this, the following aspects are also interesting: First, instead of treating the JSSP instance as a two-dimensional matrix, we can flatten the instance and tackle the regression problem using sequence modeling. Another aspect that might be interested to look into is to train an JSSP instance

embeddings using autoencoders and test whether such embeddings could be sufficient in the regression tasks.

Another interesting technique that might be interesting to look into is applying differentiable architecture search to find the most suitable DL architecture for the regression tasks.

Choice of tbf values

Perhaps, one of the most counter-intuitive findings from solving a CP task is that randomness in the solving process can actually improve the performance with high probability. However, what is the level of randomness to use, or is there even an optimal level of randomness yet remain to be the question to answer. Thus, it would be very interesting to target this direction.

Ways to generalize the Machine Learning methods to other scheduling problems

Once we have demonstrated the potential of ML and DL in improving the performance of solving a JSSP, another question that comes directly after is that how can be justify that these techniques can be used to facilitate other scheduling or general CP problems. While experimentation results can show the usefulness of such techniques in particular use cases of CP problems, proving these techniques are useful in general can be quite trick which requires serious thinking and proving.

Bibliography

[1] Francesca Rossi, Peter van Beek, and Toby Walsh. "Handbook of Constraint Programming (Foundations of Artificial Intelligence)". In: New York, NY, USA: Elsevier Science Inc., 2006. ISBN: 0444527265.

[2] Brian H. Mayoh, Enn Tyugu, and Jaan Penjam, eds. *Constraint Programming, Proceedings of the NATO Advanced Study Institute on Constraint Programming, Parnu, Estonia, August 13-24, 1993*. Vol. 131. NATO ASI Series. Springer, 1994. ISBN: 978-3-642-85985-4. DOI: 10.1007/978-3-642-85983-0. URL: https://doi.org/10.1007/978-3-642-85983-0.

[3] Leonora Bianchi et al. "A Survey on Metaheuristics for Stochastic Combinatorial Optimization". In: 8.2 (June 2009), pp. 239–287. ISSN: 1567-7818. DOI: 10.1007/s11047-008-9098-4. URL: http://dx.doi.org/10.1007/s11047-008-9098-4.

[4] Valentin Antuori and Florian Richoux. "Constrained optimization under uncertainty for decision-making problems: Application to Real-Time Strategy games". In: Jan. 2019.

[5] Luc De Raedt et al. "Constraint Programming meets Machine Learning and Data Mining (Dagstuhl Seminar 11201)." In: *Dagstuhl Reports* 1 (Jan. 2011), pp. 61–83.

[6] Jean-Francois Puget. "Constraint Programming Next Challenge: Simplicity of Use". In: *Principles and Practice of Constraint Programming – CP 2004*. Ed. by Mark Wallace. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 5–8. ISBN: 978-3-540-30201-8.

[7] Eugene C. Freuder. "Progress towards the Holy Grail". In: *Constraints* 23.2 (Apr. 2018), pp. 158–171. ISSN: 1572-9354. DOI: 10.1007/s10601-017-9275-0. URL: https://doi.org/10.1007/s10601-017-9275-0.

[8] Nicholas Nethercote et al. “MiniZinc: Towards a Standard CP Modelling Language”. In: *CP*. 2007.

[9] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. “Modeling”. In: *Modeling and Programming with Gecode*. Ed. by Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Corresponds to Gecode 6.2.0. 2019.

[10] Jacob Feldman. “Representing and Solving Rule-Based Decision Models with Constraint Solvers”. In: *RuleML America*. 2011.

[11] Alan Frisch. “A Decade of Progress in Constraint Modelling and Reformulation: The Quest for Abstraction and Automation”. In: *Invited Talk slides*. 2011. URL: https://www-users.cs.york.ac.uk/~frisch/Research/decade.pdf.

[12] Gustav Bjordal et al. “A constraint-based local search backend for MiniZinc”. In: *Constraints* 20.3 (July 2015), pp. 325–345. ISSN: 1572-9354. DOI: 10.1007/s10601-015-9184-z. URL: https://doi.org/10.1007/s10601-015-9184-z.

[13] Michele Lombardi and Michela Milano. “Boosting Combinatorial Problem Modeling with Machine Learning”. In: *CoRR* abs/1807.05517 (2018). arXiv: 1807.05517. URL: http://arxiv.org/abs/1807.05517.

[14] Mihaela Sabin and Eugene C. Freuder. “Automated Formulation of Constraint Satisfaction Problems.” In: Jan. 1996, pp. 1407–1407.

[15] Zeynep Kiziltan, Marco Lippi, and Paolo Torroni. “Constraint Detection in Natural Language Problem Descriptions”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI’16. New York, New York, USA: AAAI Press, 2016, pp. 744–750. ISBN: 978-1-57735-770-4. URL: http://dl.acm.org/citation.cfm?id=3060621.3060725.

[16] Alessio Bonfietti, Michele Lombardi, and Michela Milano. “Embedding Decision Trees and Random Forests in Constraint Programming”. In: *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*. 2015, pp. 74–90. DOI: 10.1007/978-3-319-18008-3_6. URL: https://doi.org/10.1007/978-3-319-18008-3%5C_6.

[17] Hong Xu, Sven Koenig, and T K. Satish Kumar. "Towards Effective Deep Learning for Constraint Satisfaction Problems". In: Aug. 2018. DOI: 10.1007/978-3-319-98334-9_38.

[18] André Hottung, Shunji Tanaka, and Kevin Tierney. "Deep Learning Assisted Heuristic Tree Search for the Container Pre-marshalling Problem". In: (Sept. 2017).

[19] Hongzi Mao et al. "Resource Management with Deep Reinforcement Learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. HotNets '16. Atlanta, GA, USA: ACM, 2016, pp. 50–56. ISBN: 978-1-4503-4661-0. DOI: 10.1145/3005745.3005750. URL: http://doi.acm.org/10.1145/3005745.3005750.

[20] Wei Wang et al. "Edge Caching at Base Stations with Device-to-Device Offloading". In: *IEEE Access* PP (Mar. 2017), pp. 1–1. DOI: 10.1109/ACCESS.2017.2679198.

[21] Lei Lei et al. "A deep learning approach for optimizing content delivering in cache-enabled HetNet". In: *2017 International Symposium on Wireless Communication Systems (ISWCS)* (2017), pp. 449–453.

[22] Helge Spieker and Arnaud Gotlieb. "Towards Hybrid Constraint Solving with Reinforcement Learning and Constraint-Based Local Search". In: 2018.

[23] Andrea Galassi et al. "Model Agnostic Solution of CSPs via Deep Learning: A Preliminary Study". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings*. 2018, pp. 254–262. DOI: 10.1007/978-3-319-93031-2_18. URL: https://doi.org/10.1007/978-3-319-93031-2%5C_18.

[24] Wei Zhang and Thomas G. Dietterich. "A Reinforcement Learning Approach to job-shop Scheduling". In: *IJCAI*. 1995.

[25] Maziar Yazdani et al. "Optimizing the sum of maximum earliness and tardiness of the job shop scheduling problem". In: *Computers & Industrial Engineering* 107 (2017), pp. 12–24. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2017.02.019. URL: http://www.sciencedirect.com/science/article/pii/S0360835217300803.

[26] Ian Gent et al. "Machine learning for constraint solver design – A case study for the alldifferent constraint". In: (Aug. 2010).

[27] Frank Hutter et al. “Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms”. In: *Principles and Practice of Constraint Programming - CP 2006*. Ed. by Frédéric Benhamou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 213–228. ISBN: 978-3-540-46268-2.

[28] Michele Lombardi and Stefano Gualandi. “A New Propagator for Two-Layer Neural Networks in Empirical Model Learning”. In: *CP*. 2013.

[29] Lars Kotthoff. “Algorithm Selection for Combinatorial Search Problems: A Survey”. In: *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*. Ed. by Christian Bessiere et al. Cham: Springer International Publishing, 2016, pp. 149–190. ISBN: 978-3-319-50137-6. DOI: 10.1007/978-3-319-50137-6_7. URL: https://doi.org/10.1007/978-3-319-50137-6_7.

[30] F. Neumann and S. Poursoltan. “Feature-based algorithm selection for constrained continuous optimisation”. In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. July 2016, pp. 1461–1468. DOI: 10.1109/CEC.2016.7743962.

[31] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. “An Economics Approach to Hard Computational Problems”. In: *Science* 275.5296 (1997), pp. 51–54. ISSN: 0036-8075. DOI: 10.1126/science.275.5296.51. eprint: https://science.sciencemag.org/content/275/5296/51.full.pdf. URL: https://science.sciencemag.org/content/275/5296/51.

[32] Carla P. Gomes and Bart Selman. “Algorithm portfolios”. In: *Artificial Intelligence* 126.1 (2001). Tradeoffs under Bounded Resources, pp. 43–62. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(00)00081-3. URL: http://www.sciencedirect.com/science/article/pii/S0004370200000813.

[33] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. “An Extensive Evaluation of Portfolio Approaches for Constraint Satisfaction Problems”. In: *International Journal of Interactive Multimedia and Artificial Intelligence* 3.7 (June 2016), pp. 81–86. ISSN: 1989-1660. DOI: 10.9781/ijimai.2016.3712. URL: http://www.ijimai.org/journal/sites/default/files/files/2016/05/ijimai20163_7_12_pdf_13932.pdf.

[34] Andrea Loreggia et al. “Deep Learning for Algorithm Portfolios”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, pp. 1280–1286. URL: http://dl.acm.org/citation.cfm?id=3015812.3016001.

[35] Frank Hutter et al. “Algorithm runtime prediction: Methods & evaluation”. In: *Artif. Intell.* 206 (2014), pp. 79–111.

[36] M.H. Sqalli and E.C. Freuder. “Inference-based constraint satisfaction supports explanation”. In: (Dec. 1996).

[37] Dmitry Maliotov and Kuldeep S. Meel. “MLIC: A MaxSAT-Based framework for learning interpretable classification rules”. In: *CoRR* abs/1812.01843 (2018). arXiv: 1812.01843. URL: http://arxiv.org/abs/1812.01843.

[38] Eugene C. Freuder, Chavalit Likitvivatanavong, and Richard J. Wallace. “Deriving Explanations and Implications for Constraint Satisfaction Problems”. In: *Principles and Practice of Constraint Programming — CP 2001*. Ed. by Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 585–589. ISBN: 978-3-540-45578-3.

[39] Narendra Jussien and Samir Ouis. “User-friendly explanations for constraint programming”. In: *Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE’01), Paphos, Cyprus, December 1, 2001*. 2001. URL: http://arxiv.org/abs/cs.PL/0111037.

[40] Marek Wojciechowski and Maciej Zakrzewicz. “Dataset Filtering Techniques in Constraint-Based Frequent Pattern Mining”. In: *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery*. London, UK, UK: Springer-Verlag, 2002, pp. 77–91. ISBN: 3-540-44148-4. URL: http://dl.acm.org/citation.cfm?id=647915.738873.

[41] Christian Bessiere et al. “Users Constraints in Itemset Mining”. In: *CoRR* abs/1801.00345 (2018). arXiv: 1801.00345. URL: http://arxiv.org/abs/1801.00345.

[42] Said Jabbour et al. “On Maximal Frequent Itemsets Mining with Constraints: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings”. In: Aug. 2018, pp. 554–569. ISBN: 978-3-319-98333-2. DOI: 10.1007/978-3-319-98334-9_36.

[43] Said Jabbour, Lakhdar Sais, and Yakoub Salhi. “Top-k Frequent Closed Itemset Mining Using Top-k SAT Problem”. In: Sept. 2013. DOI: 10.1007/978-3-642-40994-3_26.

[44] Hamid Khodabandehlou and Sami Fadali. “Training Recurrent Neural Networks as a Constraint Satisfaction Problem”. In: (Mar. 2018).

[45] Ali Allahverdi et al. “A survey of scheduling problems with setup times or costs”. In: *European Journal of Operational Research* 187.3 (2008), pp. 985–1032. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2006.06.060. URL: http://www.sciencedirect.com/science/article/pii/S0377221706008174.

[46] E.L. Lawler et al. “Sequencing and scheduling : algorithms and complexity”. English. In: *Logistics of Production and Inventory*. Ed. by S.S. Graves, A.H.G. Rinnooy Kan, and P. Zipkin. Handbooks in Operations Research and Management Science. Netherlands: North-Holland Publishing Company, 1993, pp. 445–522. ISBN: 0-444-87472-0.

[47] J.K. Lenstra and A.H.G. Rinnooy Kan. “Computational Complexity of Discrete Optimization Problems”. In: *Discrete Optimization I*. Ed. by P.L. Hammer, E.L. Johnson, and B.H. Korte. Vol. 4. Annals of Discrete Mathematics. Elsevier, 1979, pp. 121–140. DOI: https://doi.org/10.1016/S0167-5060(08)70821-5. URL: http://www.sciencedirect.com/science/article/pii/S0167506008708215.

[48] M. R. Garey, D. S. Johnson, and Ravi Sethi. “The Complexity of Flowshop and Jobshop Scheduling”. In: *Math. Oper. Res.* 1.2 (May 1976), pp. 117–129. ISSN: 0364-765X. DOI: 10.1287/moor.1.2.117. URL: http://dx.doi.org/10.1287/moor.1.2.117.

[49] Teofilo Gonzalez and Sartaj Sahni. “Flowshop and Jobshop Schedules: Complexity and Approximation”. In: *Operations Research* 26 (Feb. 1978), pp. 36–52. DOI: 10.1287/opre.26.1.36.

[50] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. “Complexity of Machine Scheduling Problems”. In: *Studies in Integer Programming*. Ed. by P.L. Hammer et al. Vol. 1. Annals of Discrete Mathematics. Elsevier, 1977, pp. 343–362. DOI: https://doi.org/10.1016/S0167-5060(08)70743-X. URL: http://www.sciencedirect.com/science/article/pii/S016750600870743X.

[51] Sadegh Mirshekarian and Dušan N. Šormaz. “Correlation of job-shop scheduling problem features with scheduling efficiency”. In: *Expert Systems with Applications* 62 (2016), pp. 131–147. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2016.06.014. URL: http://www.sciencedirect.com/science/article/pii/S0957417416302949.

[52] Diarmuid Grimes and Emmanuel Hebrard. “Solving Variants of the Job Shop Scheduling Problem Through Conflict-Directed Search”. In: *INFORMS Journal on Computing* 27 (Apr. 2015), pp. 268–284. DOI: 10.1287/ijoc.2014.0625.

[53] Yang-Kuei Lin. “Scheduling efficiency on correlated parallel machine scheduling problems”. In: *Operational Research* 18 (Oct. 2017). DOI: 10.1007/s12351-017-0355-0.

[54] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[55] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[56] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[57] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[58] François Chollet et al. *Keras*. https://keras.io. 2015.

[59] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.