Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

USE OF MACHINE LEARNING TECHNIQUES TO SOLVE SCHEDULING PROBLEMS

Bc. Evgeniya Brichkova

Supervisor: doc. Ing. Přemysl Šůcha, Ph.D.
Field of study: Open Informatics
Subfield: Data Science
May 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Brichkova** Jméno: **Evgeniya** Osobní číslo: **421672**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Datové vědy**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Využití metod strojového učení pro řešení úloh rozvrhování

Název diplomové práce anglicky:

Use of machine learning techniques to solve scheduling problems

Pokyny pro vypracování:

Práce si klade za cíl analýzu a návrh algoritmů pro řešení problémů rozvrhování s využitím metod strojového učení. Hlavní důraz je kladen na synergii mezi tradičními state-of-the-art přístupy a metodami strojovým učením. Postupujte podle následujících bodů:
1) Proveďte rešerši literatury zaměřenou na studovanou oblast.
2) Na základě výsledků v práci [2] vyberte vhodný problém pro aplikaci metod strojového učení. Můžete zvážit například problémy „1 | rj | sum Uj“, „1 | rj | sum wj Uj“ a „P | | sum wj Cj“
3) Navrhněte a implementujte algoritmus řešící vybraný problém.
4) Na základě struktury problému navrhněte způsob generování trénovacích instancí.
5) Navržený algoritmu otestujte a výsledky porovnejte s výsledky dosaženými v literatuře.

Seznam doporučené literatury:

[1] Václavík R. - Novák A. - Šůcha, P. - Hanzálek, Z. Accelerating the Branch-and-Price Algorithm Using Machine Learning In: European Journal of Operational Research. 2018, vol. 271, no. 3, pp. 1055-1069.
[2] Michal Bouska, Antonin Novak, Premysl Sucha, István Módos, Zdenek Hanzálek: Data-driven Algorithm for Scheduling with Total Tardiness. International Conference on Operations Research and Enterprise Systems 2020: 59-68.
[3] Anton Milan, Seyed Hamid Rezatofighi, Ravi Garg, Anthony R. Dick, Ian D. Reid: Data-Driven Approximations to NP-Hard Problems. Conference on Artificial Intelligence AAAI 2017: 1453-1459.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Přemysl Šůcha, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **29.07.2020** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **19.02.2022**

doc. Ing. Přemysl Šůcha, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomantka bere na vědomí, že je povinna vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studentky

Acknowledgements

I thank my supervisor doc. Ing. Přemysl Šůcha for the guidance and valuable advice during this work. I thank Ing. Michal Bouška for the help with developing ideas. I thank my fiancé Ananías Hilario for the incredible support and help during this work. I thank my mother Marina Brichkova for the support and believing in me during my whole studies and I thank my son Kalev David, who inspired me to do my best.

Thank you.

Declaration

Abstract

This work studies algorithms for solving scheduling problems using machine learning, specifically, deep learning methods. We concentrate on a scheduling problem characterized as $1 \mid r_j \mid \sum_{j \in J} U_j$ in the standard notation, i.e., scheduling a set of jobs on a single machine. Each job is characterized by its processing time, release time, and due date. The objective of this NP-hard combinatorial problem is to minimize the number of jobs that were not completed before the due date. The thesis surveys the literature related to the addressed scheduling problem and the existing machine learning techniques used to solve combinatorial problems. Further, we design several machine learning-based methods to solve $1|r_j|\sum_{j \in J} U_j$ that emphasize the combination and interaction between traditional state-of-the-art approaches for scheduling and machine learning methods. The experimental results show that the proposed methods can solve the problem instances with up to 160 jobs with a very low optimality gap.

Keywords

scheduling, number of tardy jobs, machine learning, deep learning, recurrent neural networks, transformer, attention mechanisms, seq2seq, sequence to sequence, encoder-decoder model

Supervisor

doc. Ing. Přemysl Šůcha, Ph.D.

Abstrakt

Tato diplomová práce studuje algoritmy pro řešení úloh plánování pomocí strojového učení, konkrétně metod hlubokého učení. Soustředí se na problém rozvrhování, který je charakterizován jako $1 \mid r_j \mid \sum_{j \in J} U_j$, tj. rozvrhování sady úloh na jednom počítači. Každá úloha se vyznačuje časem zpracování, časem vydání a termínem splnění. Cílem tohoto NP-těžkého kombinatorického problému je minimalizovat počet úloh, které nebyly dokončeny před termínem splnění. Diplomová práce studuje literaturu týkající se řešeného problému plánování a existujících technik strojového učení používaných k řešení kombinatorických problémů. Dále navrhujeme několik metod založených na strojovém učení, abychom vyřešili $1|r_j|\sum_{j \in J} U_j$, které zdůrazňují kombinaci a interakci mezi state-of-the-art přístupy pro rozvrhování a metody strojového učení. Výsledky experimntů ukazují, že navrhované metody mohou vyřešit problémové instance až se 160 úlohami s velmi malou relativní odchylku od optima.

Klíčová slova

plánování, počet opožděných úloh, strojové učení, hluboké učení, rekurentní neuronové sítě, transformátor, mechanismy pozornosti, seq2seq, sekvence po sekvenci, model kodér-dekodér

Překlad názvu

Využití metod strojového učení pro řešení úloh rozvrhování

Contents

Chapter 1

Introduction

This thesis studies the scheduling problem of minimizing the amount of tardy jobs on a single machine $1 \mid r_i \mid \sum U_i$. According to the definitions given by Pinedo [40], a scheduling problem is described by a triplet $\alpha \mid \beta \mid \gamma$. The problem studied in this thesis is interpreted as follows: in the field α we have 1, representing the usage of a single processing machine; in the field β we have release dates, indicating that job i cannot start before its release date r_i; the objective, as seen in the γ field, indicates that we are trying to minimize the total amount of tardy jobs. In the following work, the processing times of jobs are referred as to p_i and the due dates of jobs are referred to as d_i.

This target has wide applications in many production and service environments. It reflects factors of external costs based on due dates, such as customer satisfaction. Therefore, the number of tardy jobs can represent orders of customers that are not satisfied [36]. The possible applications include, for example, logistics services and factory businesses. The goal of this thesis is to approach the problem from the perspective of machine learning, particularly deep learning. The problem is represented as a regression and classification task, where the goal is to predict which jobs are tardy within the given instance. Therefore, the algorithm assigns tardiness to jobs based on the features that are learned and extracted by the neural network. This work focuses on two main approaches: an LSTM-based Sequence-to-Sequence architecture [26] and a Transformer-based architecture [47]. The data generation method is borrowed from [46] and is not included in the focus of this thesis. The text is organized in the following way: the second chapter studies the state of the art, including the scheduling problems and the machine learning techniques applied to them. The third chapter gives the overview of the machine learning techniques used in this thesis. The fourth chapter explains the proposed methods. The fifth chapter describes the experiments, shows the results of the experiments and presents a brief analysis of each of them. The sixth chapter concludes the work with an overall discussion of the results and possible research directions that can be taken in the future on this topic.

Chapter 2

Related work

2.1 Scheduling

Problem $1 \,|\, r_j \,|\, \sum_{j \in J} U_j$ has been proven to be NP-hard by Lenstra et al. [33]. When the release and due dates are similarly ordered, i.e., $r_i < r_j \implies d_i \leq d_j, \forall (J_i, J_j))$, Kise et al. [29] gave an $O(n^2)$ algorithm and Lawler [32] gave a $O(n \log n)$ algorithm for the same special case.

Dauzère-Pérès [14] proposed a lower bound based on the relevance of a Mixed-Integer Linear Programming (MILP) formulation, where new constraints are added in which *big-M* is required. They found out, however, that by using the structural properties of the problem, a lower bound independent of *big-M* can be derived by solving a maximum of $\log(2(n-1))$ Linear Programming (LP) problems; thus, they developed a method for obtaining the lower bound which doesn't depend on *big-M*. Then, the heuristic was presented and its effectiveness was computationally studied by comparison with the provided lower bound. Their testing instances assume up to 50 jobs. Computing times to find the lower bound are usually less than 15 seconds when $n = 10$, less than 1 minute when $n = 20$, but can increase up to 21 minutes when $n = 50$. For the heuristic, computing times are always less than 1 second.

Lasserre and Queyranne [31] proposed an original MILP formulation of several one-machine sequencing problems, based on the generalized due date scheduling problems introduced in Hall [24] and Hall et al. [25]. Among the objectives of these problems are, for example, the minimization of makespan, maximum lateness or mean flow time. In this formulation, jobs are specified by their position in the sequence. Sequencing decisions, then, can be formulated without the *big-M* coefficient usually required in classical formulations. As observed by Dyer and Wolsey [15], the *big-M* coefficient induces very poor lower bounds by LP relaxation.

Baptiste et al. [3] proposed two new lower bounds. The first one is a flow-based lower bound, obtained by Lagrangian relaxation. The other lower bounds are based on the adjustment of the release and due dates in order to match polynomially-solvable cases of the problem at hand. They include a nested-case lower bound, where the basic idea lies on the decrease of the release dates in such a way that jobs become nested, i.e., their time-windows are either included or do not overlap in time. For this purpose, the particular preemptive schedule is obtained by applying the Earliest Due-Date (EDD) priority dispatching rule. The next lower bound is based on the *Mine, Ibaraki, and Kise* algorithm by Kise et al. [29], where the data is adjusted in order to match the conditions: $\forall i, j \in J$, $r_i \geq r_j \implies d_i \leq d_j$. In addition, they introduced the dominance properties, which allow: 1) to decompose the problem into independent sub-problems; 2) to add some precedence constraints between jobs; and, 3) to determine that some jobs are on-time. The elimination rules include those initially designed for $1|r_j|L_{max}$ and are adapted to our problem and the collection of specific time-windows adjustments. Their branch-and-bound approach works optimally up to instances of size 200. The algorithm solved 73.33% of the instances within the one-hour cut-off limit when the processing times of jobs are randomly generated from a uniform distribution Unif $(1, 100)$, but only 56.67% of the problems when the processing times are generated from a uniform distribution Unif $(25, 75)$.

One more approach based on the Lagrangian relaxation was also used by Dauzère-Pérès and Sevaux [13] for the problem $1 \mid r_j \mid \sum_{j \in J} W_j \cdot U_j$. The paper is based on the notion of master sequence, i.e., a sequence from which an optimal sequence can be extracted. In their work a new MILP formulation is introduced. Using this formulation, a Lagrangian relaxation algorithm is derived. They have also introduced the MILP formulation for the non-weighted case problem: $1 \mid r_j \mid \sum_{j \in J} U_j$. The experiments were conducted on instances of size up to 160 jobs. The Lagrangian relaxation algorithm was first compared with solutions obtained by using the MILP formulation in *ILOG-CPLEX* solver. However, even for small size problems of 80 jobs, very large CPU times were observed with the MILP solver for numerous instances. For example, in a given instance, the search was stopped after 19 hours with the optimal solution, but its optimality was not proved. If the accuracy level is reduced to the one of the Lagrangian relaxation algorithm, the CPU time of the MILP solver decreases but remains very large.

For solving $1 \mid r_j \mid \sum_{j \in J} w_j \cdot U_j$, M'Hallah and Bulfin [38] proposed an exact algorithm for which the experiments indicate that the algorithm solves both weighted and unweighted problems. The approach is based on Surrogate Relaxation (SR) resulting in a multiple-choice Knapsack provides the bounds for the *Branch-and-Bound* algorithm. They first developed a mathematical model for the most general problem, $1 \mid r_j \mid \sum_{j \in J} w_j \cdot U_j$. Then, they used multipliers which determine the tightness of the bound provided, in order to form a SR of the $1 \mid r_j \mid \sum_{j \in J} w_j \cdot U_j$. The Surrogate Dual (SD) has a single resource constraint and a set of multiple-choice constraints, so it looks like a multiple-choice Knapsack problem except that it has, in addition to the binary variables, the continuous variables representing job completion times. To obtain a binary Multiple-Choice Knapsack Problem (MCKP), they replaced the completion time-continuous variables by either their upper or lower bounds, which results in a further relaxation of the original problem. The relaxed problem has the special structure of a MCKP. The solution value of MCKP is a bound to $1 \mid r_j \mid \sum_{j \in J} w_j \cdot U_j$. While the bounds MCKP yields may not be as tight as the bounds obtained by the SD of the SR, MCKP remains easier to solve. Thus, the disadvantage that we obtain a looser bound is counterbalanced by its ease of computation. Finally, the solution of MCKP is bounded using the continuous relaxation of MCKP. Then, there is presented a heuristic that gives an initial feasible solution for the *Branch-and-Bound* algorithm.

Ourari et al. [39] presents how, using a MILP formulation, both good-quality lower and upper bounds can be computed for the $1|r_j|\sum_{j \in J} U_j$ problem. The proposed approach differs from the *Branch-and-Bound* approaches described by Baptiste et al. [3], M'Hallah and Bulfin [38] and Lenstra et al. [33], in the fact that, since MILP is used, it is more generic: the model can be extended, so new constraints can easily be added to the model. Their MILP formulation is based on the dominance conditions proposed by Erschler et al. [16], which defines a set S_{dom} of dominant job sequences, with respect to the feasibility problem, for the Single Machine Scheduling Problem (SMSP). The approach has been tested on the instances generated by Baptiste et al. [3] (of size up to 160 jobs) and the results show that the gap between the upper and lower bound is, on average, very tight. Moreover, even though finding an optimal solution is not the aim of their approach, experiments have shown that their approach proves optimality of 48 instances that were not optimally solved by Baptiste et al. [3]. Further, Baptiste et al. [3] did not succeed to solve the instances of over 160 jobs. Experiments on the quality of the bounds show that the lower bound equals the optimal solution in 90% of the cases, while the upper bound is equal or better than the State of the Art (SOTA) in 98% of cases.

In 2018, Garraffa et al. [18] proposed an exact exponential algorithm for the Single Machine Total Tardiness Problem (SMTTP) which exploits the structure of a basic *Branch-and-Reduce* framework based on the Lawler's decomposition property that solves the problem with worst-case complexity in time $O(3^n)$ and polynomial space. The proposed algorithm is an improvement over the referenced technique, with the embedding of a node merging operation. Its time complexity converges to $O(2^n)$ keeping the space complexity polynomial. Bouška et al. [9] note that this the fastest known exact algorithm for SMTTP to this date and is able to solve instances with up to 1300 jobs.

2.2 Machine Learning

In many cases, when solving problems that require optimizations, such as scheduling or the Traveling Salesman Problem (TSP), we would be content to find a good approximation in a short period of time. This is one of the

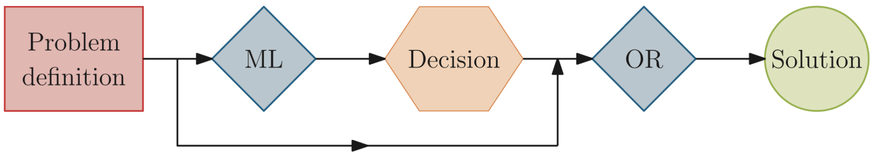

Figure 2.1: The original image is taken from the paper by Bengio et al. [7]. The machine learning model is used to augment an operation research algorithm with valuable pieces of information.

reasons why ML is a good match for Operations Research (OR) problems; we can think of OR algorithms being enhanced by ML models in several ways: 1) providing end-to-end predictions; 2) performing the selection of hyper-parameters which need to be set in order to configure OR algorithms; 3) providing with an initial approximate solution or providing useful initial information to OR algorithms (see Figure 2.1); 4) as a component called all-throughout the execution of OR algorithms (e.g., a component guiding the search based on a data-driven heuristic value provided by the ML model; see Fig. 2.2).

The majority of the ML methods in scheduling are based on the modification of the dispatching rules, done dynamically each at a time step, with the decision to be taken based on a system condition. Some inspirational works for the usage of machine learning in scheduling problems appeared starting from beyond the machine learning field [22, 27]. In most cases, the proposed approaches have selected the dispatching rules by the simulation runs. Later on, the results from the simulations were used as training data for the machine learning algorithms applied for these problems.

One of the examples of this approach is represented by the work of Choi et al. [11], who solve the re-entrant hybrid flow shop problem, $HF \mid Re-entrant \mid C_{max}, \sum_{j \in J} T_j$. They were inspired by the old framework made by Jeong and Kim [28], who in 1998 proposed the real-time scheduling mechanism using simulation and dispatching rules for flexible manufacturing systems. The architecture of both frameworks is based on the interaction between three main components: scheduler, controller, and the system state evaluator. The scheduler determines the appropriate time for choosing new dispatching rules, the controller serves as an engine for the system states monitoring and the communication between components. The system state evaluator is the place of the difference between the approaches: the old approach uses simulation for evaluating candidate dispatching rules while the new approach is rooted in the inductive learning field. They suggested a real-time scheduling mechanism in which a decision tree is used to select an appropriate dispatching rule before considering the rule change. The decision nodes of the tree are represented with the system status such as a total number of remaining operations, total processing time, and others. For the tree construction, the *Iterative Dichotomizer* algorithm by Quinlan [41] is used. The test and evaluation showed that the differences in performances of the old

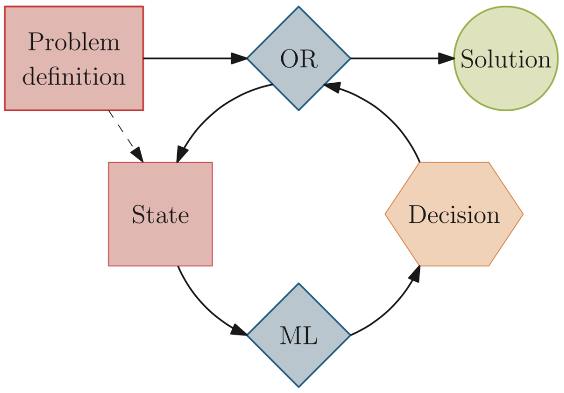

Figure 2.2: The original image is taken from the paper by Bengio et al. [7]. The combinatorial optimization algorithm repeatedly queries the same machine learning model to make decisions. The machine learning model takes as input the current state of the algorithm, which may include the problem definition.

simulation-based approach and the decision tree approach are not significant. The evaluation was performed on models with the following characteristics: 1) jobs with inter-arrival value drawn from an exponential distribution with a mean of 10, $\Delta r \sim \text{Exp}\left(\lambda = \frac{1}{10}\right)$; 2) due dates sampled from a uniform distribution $d_j \sim \text{Unif}\left(2 \cdot p_j, 4 \cdot p_j\right)$ where p_j is the sum of operation times for job j; 3) major breakdowns occur with an inter-failure time sampled $\Delta F^+ \sim \text{Exp}\left(\frac{1}{15000}\right)$, and repair times were sampled $\Delta R^+ \sim \text{Exp}\left(\frac{1}{500}\right)$; 4) minor breakdowns occur with an interval sampled $\Delta F^- \sim \text{Exp}\left(\frac{1}{6000}\right)$ and repair times were sampled $\Delta R^- \sim \text{Exp}\left(\frac{1}{150}\right)$; 5) 200 maximum waiting jobs.

A major application of Reinforcement Learning (RL) in scheduling for data center cooling was reported, although without concrete performance measures, by Evans and Gao [17] at Google. The field of RL has also gained popularity as a means to tackle scheduling problems, especially in the field of manufacturing scheduling. In RL it is possible to train for objectives that are hard-to-optimize directly due to lack of precise models if there exist reward signals that coincide with the objective. Mao et al. [35] use RL to solve job scheduling problems in computer clusters, $Rm \mid online\, r_j \mid \sum_{j \in J} S_j = \frac{C_j - r_j}{p_j}$.

The main idea of RL is the following: an agent faces a particular state of the environment he is located in, and he must choose among several actions to take; when the agent chooses his action, he eventually receives a reward which is representative of how much he achieved his overall goal. In the context of a scheduling environment, the system state can be described with, for example, the number of the jobs or tardy jobs in a buffer, as well as the tardiness or lateness of those jobs.

Wang and Usher [50] proposed the approach for the single-matching Dynamic Job-Shop Scheduling (DJSS), which is based of the dispatching rules selection and RL. Based on the system state, which is represented by the number of jobs in the buffer and the estimation of the total lateness of these jobs, the

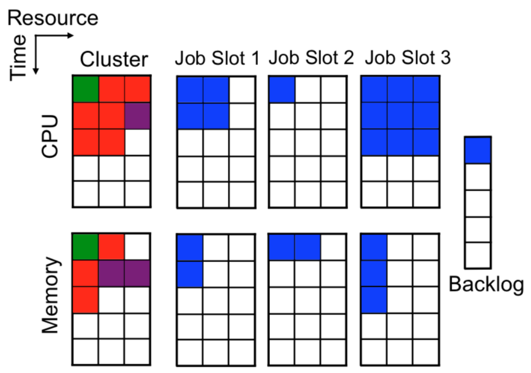

Figure 2.3: The original image is taken from the paper by Mao et al. [35]. An example of a state representation, with two resources and three pending job slots.

agent selects the best dispatching rule. *Q-learning* is used for the dispatching rules selection, in particular when there are at least two jobs in the buffer. The method is adopted for three cases of the system objective: minimize maximum lateness, minimize the number of tardy jobs, minimize mean lateness — and the agent's reward function is generated accordingly: if the currently completed job is tardy, the learning agent receives a penalty of -1, otherwise, a reward of $+1$ is assigned. In their simulations, the time between job arrivals $\Delta r = r_j - r_{j-1}$ follows an exponential distribution with a mean of 10, $\Delta r \sim \text{Exp}\left(\lambda = \frac{1}{10}\right)$; the estimated processing times of jobs were uniformly distributed $p_j \sim \text{Unif}\left(7.5, 8.5\right)$; the due dates were set according to the formula $d_j = r_j - \alpha_j \cdot p_j$, with α_j uniformly distributed $\alpha_j \sim \text{Unif}\left(1.5, 2.5\right)$.

In the year 2017, Shahrabi et al. [42] proposed an improvement over the previous study using Variable Neighborhood Search (VNS) which is introduced to address the DJSS problem. The VNS is a meta-heuristic optimization method that can be used to solve combinatorial problems. With the arrival of each new job, the VNS gets an instance of the static job-shop problem, for which it finds the optimal solution. The parameters of the VNS algorithm crucially affect the algorithm's performance; the agent's action is the selection of these parameters at each point of rescheduling. This happens with the change of the system state (number of jobs on the shop-floor and mean processing time of current operations).

RL has been recently combined with the Deep Learning (DL) techniques in what is now known as Deep Reinforcement Learning (DRL). The usage of a lookup table to store or update the states (or states and actions like in the case of *Q-Learning*) would be very expensive and sometimes impossible for large state spaces, as is the case of many problems of interest, including scheduling problems. In addition, generalization considerations are not taken into account when learning using the table representation, due to the lack of shared parameters between the states. DRL introduces the usage of the DL for efficient value estimation and policy definition.

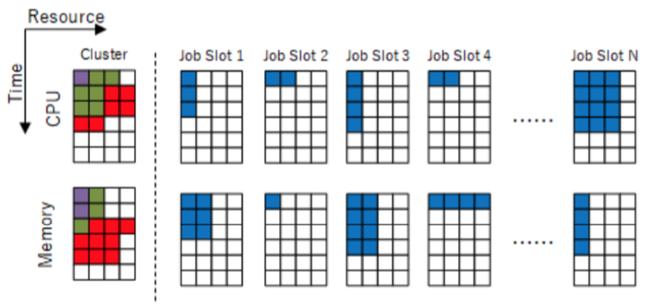

Figure 2.4: The original image is taken from the paper by Ye et al. [51]. The state representation of offline scheduling scheduling.

Mao et al. [35] in the year 2016 introduced the Deep Resource Management with Reinforcement Learning (DeepRM) — the multi-resource cluster scheduler for the resource management systems — to solve the problem $Rm \mid online\ r_j \mid \sum_{j\in J} S_j = \frac{C_j - r_j}{p_j}$: considering a cluster with m resource types and jobs arriving in an online fashion, a resource demand of each job j which is known upon arrival $k_j = (k_{j,1}, \cdots, k_{j,m})$, a job duration p_j, and a criteria of minimizing the average slowdown. Their method was tested on instances lasting for 50 time steps t, with 80% of the jobs being short-term jobs with a duration of $1t - 3t$; 20% of the jobs are long-term jobs with a duration of $10t - 15t$. DeepRM, is an example solution that translates the problem of packing tasks with multiple resource demands into a learning problem. DeepRM performs comparably or better than standard heuristics such as Shortest-Job-First (SJF). It learns strategies directly from experience, such as favoring short jobs over long jobs and keeping some resources free to service future arriving short jobs. The main assumptions of the model are: 1) jobs arrive at the cluster in an online fashion, in discrete timesteps; 2) the scheduler chooses one or more of the waiting jobs to schedule at each timestep; 3) no preemption is allowed; and, 4) the cluster is a single collection of resources. The space state is represented in Fig. 2.3. The leftmost blocks represent the current job allocation of the resources (each job of a separate color), the job slots represent the awaiting jobs and their requirements and a backlog is just a single number representing the number of jobs left beyond M awaiting ones. The state representation is fixed so that it can be applied as an input to the Neural Network (NN), meaning that M number of awaiting jobs is fixed. It is important to note the trick they use to reduce the action space: in the normal case, the action could possibly be generated for every subset of awaiting jobs, which is a large number of 2^M; while, in their approach, the agent is allowed to perform several actions during a single time step. The required function is designed to guide the agent towards the preferable objective, selected in advance.

Ye et al. [51] proposed a new approach of DRL in resource scheduling based on the DeepRM scheduler, solving instances lasting for 50 time steps t, with 80% of the jobs being short-term jobs with a duration of $1t - 3t$; 20% of the jobs are long-term jobs with a duration of $10t - 15t$. As mentioned before, one of the main assumptions of the DeepRM scheduler is the online environment where jobs arrive in a streaming fashion. The new approach proposed scheduling techniques for both: 1) offline (DeepRM Off); and, 2) online environments (DeepRM2). Two different objectives take place: the minimization of the average slowdown and the minimization of the job completion time. The state representation of the DeepRM2 coincides with the DeepRM, while the DeepRM Off representation is different: opposite to the online learning here we have got all the awaiting jobs from the start, so we have the job slots amount equivalent to the number of jobs as well as the backlog information that is missing, as it is shown in Fig. 2.4. The action space and the reward are similar to the DeepRM. One more major difference with the DeepRM approach is the usage of the Convolutional Neural Network (CNN) since the state space is represented as a 2-dimensional lattice, which can be seen as an image, and thus the characteristics of CNN can be leveraged. One more difference is the usage of imitation learning for the CNN initialization.

More recently, Bouška et al. [9] proposed a data-driven algorithm that solves the Single Machine Total Tardiness Problem (SMTTP). It leverages DL all-throughout the execution (that is, the type 4 of the categorization by Bengio et al. [7]) which can efficiently generalize information from the training phase to instances of size up to 350 jobs, where it achieves an optimality gap of about 0.5%.

2.3 Challenges

2.3.1 Feasibility

As noted by Bengio et al. [7], even though ML can be used to directly output solutions to optimization problems, it would be more precise to say that the algorithm is learning a heuristic: the learned algorithms do not usually come with any guarantees in terms of optimality or feasibility. This can be the case for any heuristic — and this issue can be mitigated by using the heuristic within an exact optimization algorithm (such as *Branch-and-Bound*).

Finding the feasible solutions to a combinatorial optimization problem is not an easy task (NP-hard for general MILP), but it is even more challenging in ML, especially by using NNs: trained with gradient descent, NNs must be designed carefully in order not to break the differentiability of the target function, learned by a NN. For instance, the *Pointer Network* by Vinyals et al. [49] is a complex architecture used to make a network output a permutation, a constraint easy to satisfy when writing a classical combinatorial optimization heuristic.

2.3.2 Modeling

Concerning modeling, Bengio et al. [7] remark that the problems studied in combinatorial optimization are different from the ones currently being addressed in ML, where most successful applications target natural signals. The architectures used to learn good policies in combinatorial optimization might be very different from what is currently used with DL.

2.3.3 Scaling

Bengio et al. [7] note that scaling to larger problems can be a challenge. If a model trained on instances up to some size, say TSP up to size $n = 50$, is evaluated on larger instances, say TSP of size $n = 100$, $n = 500$, etc, the challenge exists in terms of generalization: papers tackling TSP through ML and attempting to solve larger instances see degrading performance as size increases much beyond the sizes seen during training [5, 12, 30, 49]. To tackle this issue learning on larger instances would be an option, but this may turn out to be a computational and generalizational issue. Except for very simple ML models and strong assumptions about the data distribution, it is impossible to know the model capacity and the architectural characteristics required to capture the target distribution. It is also hard to predict the sample complexity, i.e., the number of observations required for learning, since one is unaware of the true data generating distribution.

Bengio et al. [7] suggest that learning should occur on a distribution small enough that the policy could fully exploit the structure of the problem to achieve generalization. Bengio et al. [7] believe that end-to-end machine learning approaches to combinatorial optimization can be improved by combining them with current combinatorial optimization algorithms, to benefit from the theoretical guarantees and SOTA algorithms already available.

2.3.4 Data Generation

Bengio et al. [7] remark that collecting instances of optimization problems is a subtle task. Given an external process on which we observe instances of an optimization problem, we can collect data to train some policy needed for optimization, and expect the policy to generalize on future instances of this application. A practical example would be a business that frequently encounters optimization problems related to their activities, such as a delivery company. In cases where we are not targeting a specific application for which we would have historical data, proactively training a policy for problems that we do not yet know of poses a complex question: we need to define over which family of instances we want to generalize; even so, it remains a complex effort to generate problems that capture the essence of such family of instances. Smith-Miles and Bowly [43] propose a problem instance generating method.

Chapter 3

Background Theory

The following review of fundamentals of Machine Learning was gathered mainly from the work by Goodfellow et al. [21].

3.1 Types of ML Tasks

ML tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that we want the ML system to process. The ML tasks concerning this work are reviewed.

- **Classification.** In this type of task, the model is asked to specify to which of the categories some input belongs. To solve this task, the learning algorithm is usually asked to learn a function $f : \mathbb{R}^n \longmapsto \{1, \cdots, k\}$. When $y = f(x)$, the model assigns an input described by vector x to a category identified by numeric code y.
- **Regression.** In this type of task, the model is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to learn a function $f : \mathbb{R}^n \longmapsto \mathbb{R}$. This type of task is similar to classification, except that the format of output is different.
- **Structured output.** Tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements. This is a broad category and subsumes transcription and translation tasks, as well as many other tasks.

3.2 Performance Measure

To evaluate the ability of an ML algorithm, we must design a quantitative measure of its performance. Usually this performance measure is specific to the task being carried out by the system. For tasks such as classification, we often measure the accuracy of the model. Accuracy is the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the error rate. The error rate is the proportion of examples for which the model produces an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not. However, these measures suit for the classification tasks. For such tasks as ,for example, regression, we must use performance metrics that give the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

3.3 ML Model Generalization and the Bias-Variance Trade-Off

The central challenge in ML is generalization: the learned model must perform well on new, previously unseen, inputs. We typically estimate the generalization error of a ML model by measuring its performance on a test set of samples that were collected separately from the training set.

The bias-variance trade-off was first formally introduced by Geman et al. [19]. It refers to the fact that when trying to make a statistical prediction (e.g., estimating a parameter of a distribution or fitting a function), there is a trade-off between the accuracy of the prediction and its precision, or equivalently between its bias and variance, resp. Suppose we are trying to predict some target function $f : \mathbb{R}^n \longmapsto \mathbb{R}$. Let $y = f(x)$, where $(x, y) \sim P_D$. Let us define a dataset D as a collection of independent identically distributed (iid) realizations of pairs (x, y) sampled from the real data distribution P_D. We use a dataset $D = (x_i, y_i)_{i=1}^n \sim P_D$ to fit an estimator $\hat{f}_D$ of the target f using some statistical algorithm, which searches over some function class $\mathcal{F}$ to find the best $\hat{f}_D$ available in $\mathcal{F}$.

For any data point (x, y), the bias B of the predictor's function class $\mathcal{F}$ is the difference between the expected value over dataset realizations $D \sim P_D$ with their corresponding predictor $\hat{f}_D$, of the predicted value $\hat{f}_D(x)$ and the true value at $f(x)$:

$$B = \mathbb{E}_{P_D}[\hat{f}_D(x)] - f(x) \tag{3.1}$$

For any data point (x, y), the variance V of the predictor's function class $\mathcal{F}$ is the expected difference between the predicted value of the predictor fitted on a particular dataset realization, $\hat{f}_D(x)$, and the expected value over dataset realizations $D \sim P_D$ with their corresponding predictor $\hat{f}_D$, of the predicted value:

$$\begin{aligned} V &= \mathbb{E}_{P_D}\left[\left(\hat{f}_D(x) - \mathbb{E}_{P_D}[\hat{f}_D(x)]\right)^2\right] \\ &= \mathbb{E}_{P_D}\left[\hat{f}_D(x)^2 - 2\cdot\hat{f}_D(x)\cdot\mathbb{E}_{P_D}[\hat{f}_D(x)] + \mathbb{E}_{P_D}[\hat{f}_D(x)]^2\right] \\ &= \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)^2] - 2\cdot\mathbb{E}_{P_D}[\hat{f}_{P_D}(x)]\cdot\mathbb{E}_{P_D}[\hat{f}_{P_D}(x)] + \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)]^2 \\ &= \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)^2] - 2\cdot\mathbb{E}_{P_D}[\hat{f}_{P_D}(x)]^2 + \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)]^2 \\ &= \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)^2] - \mathbb{E}_{P_D}[\hat{f}_{P_D}(x)]^2 \end{aligned} \tag{3.2}$$

A standard measure for the error in a prediction is the Mean Squared Error (MSE) is

$$MSE = \mathbb{E}_{P_D}[(\hat{f}_D(x) - f(x))^2] \tag{3.3}$$

The MSE increases as the bias or variance increases. In fact, a well-known result is that the MSE decomposes into a bias and variance term:

$$\begin{aligned} MSE &= \mathbb{E}_{P_D}[(\hat{f}_D(x) - f(x))^2] \\ &= \mathbb{E}_{P_D}\left[\hat{f}_D(x)^2 - 2\cdot\hat{f}_D(x)\cdot f(x) + f(x)^2\right] \\ &= \mathbb{E}_{P_D}[\hat{f}_D(x)^2] - 2\cdot\mathbb{E}_{P_D}[\hat{f}_D(x)]\cdot f(x) + f(x)^2] \\ &\quad + \mathbb{E}_{P_D}[\hat{f}_D(x)]^2 - \mathbb{E}_{P_D}[\hat{f}_D(x)]^2 \\ &= \underbrace{\mathbb{E}_{P_D}[\hat{f}_D(x)^2] - \mathbb{E}_{P_D}[\hat{f}_D(x)]^2}_{V} + \underbrace{(\mathbb{E}_{P_D}[\hat{f}_D(x)] - f(x))^2}_{B^2} \\ &= V + B^2 \end{aligned} \tag{3.4}$$

For two function classes that have the same prediction error, if one function class is more biased than the other, then we know the other must have higher variance. Thus, to effectively use ML one tries to use a function class that balances the bias-variance trade-off. Ideally, we would use the smallest function class that can capture the target function.

3.4 Overfitting and Underfitting

The factors determining how well a ML algorithm will perform are its ability to make the training error small and make the gap between training and test error small. These two factors correspond to the two central challenges of machine learning: underfitting and overfitting. *Underfitting* occurs when the model is not able to obtain a sufficiently low error on the training set; *overfitting* occurs when the gap between the training and test error is too large. Note that this gap is representative of the generalization performance of the model. We can control whether a model is more likely to overfit or underfit by altering its *capacity*, which is its ability to fit a wide variety of functions. One way to control the capacity of a learning algorithm is by choosing its hypothesis space, the set of functions that the learning algorithm is allowed to select as being the solution. ML algorithms will generally perform best when their capacity is appropriate for the true complexity of the task and the amount of training data they are provided with. The capacity of the model directly affects its variance since the variance represents the amount of variability in functions of the function class, or in other words is a measure of the complexity or size of the function class.

Capacity is not only determined by the choice of model. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective; this is called the representational capacity of the model. Additional limitations, such as the imperfection of the optimization algorithm, mean that the learning algorithm's effective capacity may be less than the representational capacity of the model family.

3.5 Hyperparameters

Most ML algorithms have hyperparameters, settings that we can use to control the algorithm's behavior. Sometimes a parameter is chosen to be a hyperparameter (i.e., that the learning algorithm does not learn) because the parameter is difficult to optimize, or because it is not appropriate to learn it on the training set. One of the exmaples is the learning rate. Learning rate is a hyperparameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. Another example is the model capacity. As mentioned in the previous section, the model capacity directly affects the model's ability to overfit on the training data. Therefore, the bigger the model capacity is, the more complicated function it is capable of learning in a specific piece of data. This creates bigger variance. On the other hand, too little model capacity leads to bigger bias and incapability of learning the training data.

3.6 Recurrent Neural Networks

RNN serve for processing sequential data: it operates on a sequence that contains vectors $x^{(t)}$ with the time step index t ranging from 1 to τ, where τ

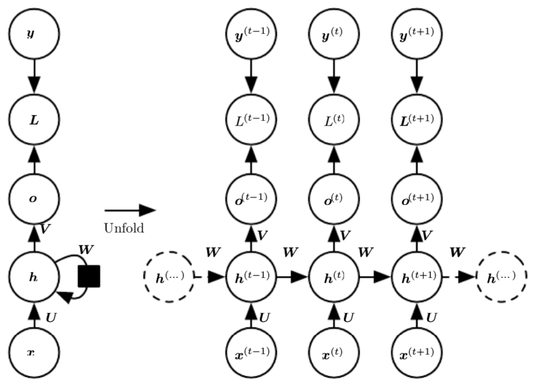

Figure 3.1: Diagram by Goodfellow et al. [21, see II.10]. RNNs that produce an output at each time step and have recurrent connections between hidden units.

is length of the sequence. The main idea of the RNN is parameter sharing, which makes it possible to extend the same model and apply it to inputs of varying lengths. It is able to learn a fixed size representation for any sequence length and generalize across various lengths. Each RNN unit at each time step t has a hidden state. The current time step's hidden state is calculated using information of the previous time step's hidden state and the current input. This process helps to retain information on what the model saw in the previous time step when processing the current time steps information. Many RNNs use Eq. 3.5 — or a similar equation — to define the values of their hidden state.

$$h^{(t)} = f(h^{(t-1)},\ x^{(t)};\ \theta) \tag{3.5}$$

The network typically learns to use $h^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t. This summary is, in general, necessarily lossy, since it maps an arbitrary length sequence $(x^{(1)}, \cdots, x^{(t-1)}, x^{(t)})$ to a fixed length vector x^h. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than others. Some issues arise due to this concerning long-term dependencies; these issues have been tackled by more sophisticated models such as the Long Short-Term Memory (LSTM), and by the development of so-called *Attention Mechanisms*, which will be presented in the following sections.

Goodfellow et al. [21] bring some important use cases for RNNs:

- **Many-to-Many Hidden-to-Hidden.** In this use case, the RNN produces an output at each time step and has recurrent connections between hidden units and produces an output at each time step. In the Fig. 3.1 we can see the computational graph to compute the training loss of an RNN that maps an input sequence of $\mathbf{x} = (x^{(1)}, \cdots, x^{(t)})$ values to a corresponding sequence of $\mathbf{o} = (o^{(1)}, \cdots, o^{(t)})$ values. A loss $L^{(t)}$

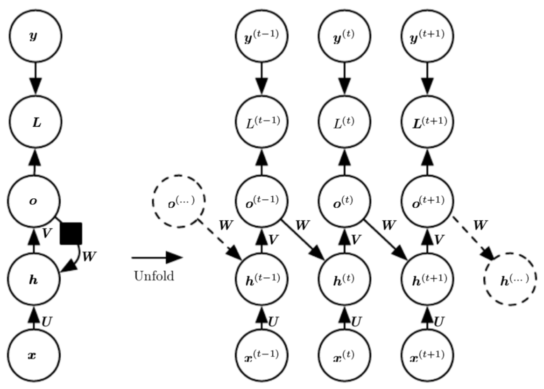

Figure 3.2: Diagram by Goodfellow et al. [21, see II.10]. RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.

measures how far each $o^{(t)}$ is from the corresponding training target $y^{(t)}$. The RNN has an input to hidden connections parametrized by a weight matrix U, hidden-to-hidden recurrent connections parametrized by a weight matrix W, and hidden-to-output connections parametrized by a weight matrix V.

- **Many-to-Many Output-to-Hidden.** In this use case, the RNN produces an output at each time step and has recurrent connections only from the output at one time step to the hidden units at the next time step. In the Fig. 3.2 we can see an RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $x^{(t)}$, the hidden layer activations are $h^{(t)}$, the outputs are $o^{(t)}$, the targets are $y^{(t)}$ and the loss is $L^{(t)}$. To the left, the circuit diagram; to the right, the unfolded computational graph. Such an RNN is less powerful (i.e., can express a smaller set of functions) than those in the family represented by Fig. 3.1. The RNN in Fig. 3.1 can choose to put any information it wants about the past into its hidden representation $h^{(t)}$ and transmit $h^{(t)}$ to the future. The RNN in this figure is trained to put a specific output value into $o^{(t)}$, and $o^{(t)}$ is the only information it is allowed to send to the future. There are no direct connections from $h^{(t)}$ going forward. The previous $h^{(t)}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $o^{(t)}$ is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training.

- **Many-to-One Hidden-to-Hidden** This use case refers to RNNs with recurrent connections between hidden units, that read an entire sequence and then produce a single output. In the Fig. 3.3 we can see the time-unfolded RNN with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a

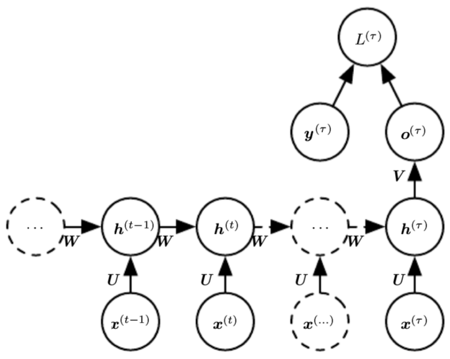

Figure 3.3: Diagram and caption by Goodfellow et al. [21, see II.10]. Time-unfolded RNN with a single output at the end of the sequence.

fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $o^{(t)}$ can be obtained by back-propagating from further downstream modules.

The forward message for the case of the network from the Fig. 3.1 is shown in the following equations. The activation function is chosen to be the hyperbolic tangent. We assume that the output is discrete, as if the RNN is used to predict words from a dictionary. A natural way to represent discrete variables is to regard the output logits $\mathbf{o}$ as giving the unnormalized log-probabilities of each possible value of the discrete variable. We can then apply the $Softmax$ operation as a post-processing step to obtain a vector $\hat{\mathbf{y}}$ of normalized probabilities over the output that we can use for calculating the loss function. Forward propagation begins with a specification of the initial state $h^{(0)}$. Then, for each time step $t \in \{1, \cdots, \tau\}$, we apply the update equations.

3.6.1 Teacher Forcing

Models that have recurrent connections from their outputs leading back into the model may be trained with teacher forcing. Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t+1$. We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is:

$$\log p\left(y^{(1)}, y^{(2)} \mid x^{(1)}, x^{(2)}\right) \\ = \log p\left(y^{(2)} \mid y^{(1)}, x^{(1)}, x^{(2)}\right) + \log p\left(y^{(1)} \mid x^{(1)}, x^{(2)}\right) \quad (3.6)$$

In this example, we see that at time $t = 2$, the model is trained to maximize the conditional probability of $y^{(2)}$ given both the x sequence so far and the previous y value from the training set. Maximum likelihood thus specifies that during training, rather than feeding the model's own output back into itself, these connections should be fed with the target values specifying what the correct output should be. Teacher forcing was originally motivated for allowing us to avoid Back-Propagation Through Time (BPTT) in models that lack hidden-to-hidden connections. As soon as the hidden units become a function of earlier time steps, however, the BPTT algorithm is necessary.

Definition 3.1 (Teacher Forcing)**.** Derived from the maximum likelihood criterion, the teacher forcing method replaces the input at time $t+1$ with the ground truth instead of the output from time t given by the model.

The teacher forcing method possesses a disadvantage, namely, that the feedback inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time. One way to mitigate this problem is to train with both teacher-forced inputs and free-running inputs. Bengio et al. [6] suggest an approach to mitigate the gap between the inputs seen at training time and the inputs seen at test time, which randomly chooses to use generated values or actual data values as input during training.

3.6.2 Gated RNN and The Long Short-Term Memory

One of the most effective sequence models used in practical applications are called gated RNNs. The gated RNNs introduce the *forget* mechanism, which allows the network to set the old state to zero. This leads to the idea of creating paths through time that have derivatives that neither vanish nor explode.

One of the most successful realizations of the gated RNNs is LSTM. A crucial addition by Gers et al. [20] has been to make the weight on this self-loop conditioned on the context, rather than fixed. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence. The control unit then learns when to forget fully or partially the previous information based on the current sequence. Therefore, once the particular information has been used, the LSTM are able to forget the old state.

The LSTM block diagram is illustrated in Fig. 3.4; it shows a block diagram of the LSTM network cell. Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing

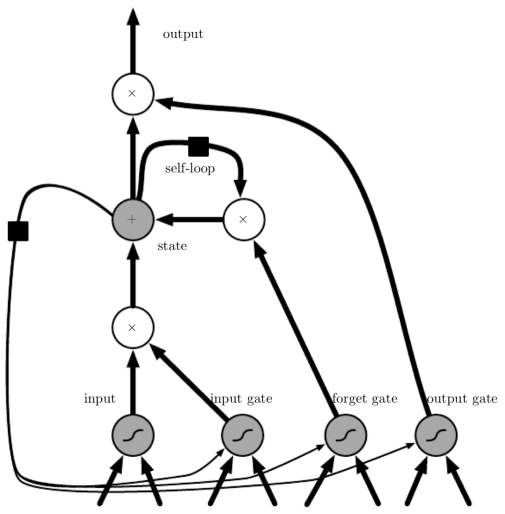

Figure 3.4: Diagram and caption by Goodfellow et al. [21, see II.10]. Block diagram of the LSTM network cell.

nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

The LSTM has been found extremely successful in many applications, such as unconstrained handwriting recognition, speech recognition, handwriting generation, machine translation and image captioning.

3.7 Sequence-to-Sequence Models

The Sequence-to-Sequence models serve to map an input sequence to an output sequence which is not necessarily of the same length. The input can be expressed as a vector or sequence of vectors that summarize the sequence $\mathbf{x} = (x^{(1)}, \cdots, x^{(n_x)})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed by Cho et al. [10] and shortly after by Sutskever et al. [44], who independently developed that architecture and were the first to obtain state-of-the-art translation using this approach. The former system is based on scoring proposals generated by another machine translation system, while the latter uses a standalone recurrent network to generate the translations. These authors respectively called this architecture, illustrated in Fig. 3.5, the encoder-decoder or sequence-to-sequence architecture. The idea is very simple: 1) an encoder or reader or input RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state; 2) a decoder or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence $\mathbf{y} = (y^{(1)}, \cdots, y^{(n_y)})$. In a sequence-to-sequence architecture, the two RNN are trained jointly to maximize the average of

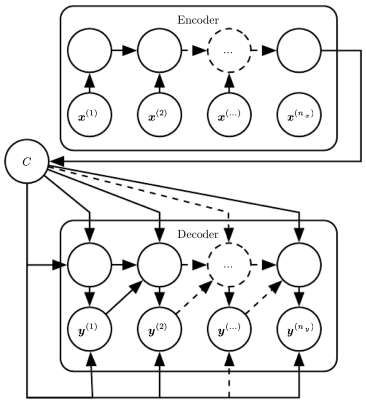

Figure 3.5: Diagram and caption by Goodfellow et al. [21, see II.10]. Example of an encoder-decoder or sequence-to-sequence RNN architecture.

$\log \mathbb{P}(y^{(1)}, \cdots, y^{(n_y)} \mid x^{(1)}, \cdots, x^{(n_x)})$ over all the pairs of x and y sequences in the training set. The last state h_{n_x} of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

We can see in Fig. 3.5 an example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(y^{(1)}, \cdots, y^{(n_y)})$ given an input sequence. It is composed of an encoder RNN that reads the input sequence $(x^{(1)}, \cdots, x^{(n_x)})$ and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

3.8 Attention Mechanisms

Attention mechanism was firstly introduced for the RNN-based encoder-decoder architecture by [2]. One of the biggest problems of RNN-based encoder-decoder architecture is the processing of long sequences. As it was mentioned before, the context C, emitted by the Encoder, summarizes the whole sequence. When processing a long sequence, the information becomes hard to capture within a single fixed-length Encoder state. Attention mechanism addresses the long-range dependencies limitation of RNNs. It learns to emphasize only to the parts of the input sequence that are relevant to the current prediction. Instead of a single last hidden state, the Encoder emits a hidden state vector at each time step. Therefore, the result context collects all the hidden states of the input sequence.

Attention mechanisms map each query to a so-called *attention value*, performing a convex combination of the existing values, where the weight assigned to each is obtained by a similarity function comparing the query with the corresponding key. We can think of it as a form of soft-retrieval; we have: 1) a query q for which we want to retrieve its value v; and, 2) keys k associated to values, to which we will match the query:

$$y = \mathit{Attention}(q, K, V) = \sum_i \mathit{Similarity}(q, k_i) \cdot v_i \tag{3.7}$$

The similarity function expresses the measure of alignment between the query and the key. It depends on the attention mechanism chosen.

We do not assume, that the initial representations of the keys and queries can be directly compared, so we desire to learn representation spaces suitable for comparing the queries and keys. Similarly, we wish to impart more power unto the attention mechanism by allowing it to project the values into a space that is more suitable to produce the final values obtained by attention. Therefore we learn the projections W^Q, W^V and W^K.

Considering $X \in \mathbb{R}^{n \times d_x}$, $Y \in \mathbb{R}^{n \times d_y}$, $Z \in \mathbb{R}^{n \times d_z}$, we let $W^Q \in \mathbb{R}^{d_x \times d_k}$, $W^K \in \mathbb{R}^{d_y \times d_k}$, $W^V \in \mathbb{R}^{d_z \times d_v}$. Then,

$$Q = X \cdot W^Q, \quad K = Y \cdot W^K, \quad V = Z \cdot W^V \tag{3.8}$$

where $Q, K \in \mathbb{R}^{n \times d_k}$, $V \in \mathbb{R}^{n \times d_v}$.

There are various types of attention mechanism, which are outlined in the following subsections.

3.8.1 Self-Attention

Relates elements from an input set between each other. In self-attention, the queries, the keys and the values, all come from the same source. Considering a group of n input vectors $X = (x_1, \cdots, x_n)$, we obtain with self-attention $Y = (y_1, \cdots, y_n)$, where:

$$y_i = \mathit{Attention}(x_i, X, X) = \sum_j \mathit{Similarity}(x_i, x_j) \cdot x_j \tag{3.9}$$

Remark 3.2. The resulting self-attention is permutation equivariant:

$$\mathit{Attention}(\pi(X)) = \pi(\mathit{Attention}(X))$$

Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations.

3.8.2 Scaled Dot-Product Attention

The similarity function of choice by attention mechanisms is the dot product, followed by the $Softmax$ function (used to normalize the weights into probabilistic form). Considering previously defined cardinalities, the attention values are obtained as follows:

$$Attention(Q, K, V) = Softmax_k\left(\frac{Q \cdot K^\intercal}{\sqrt{d_k}}\right) \cdot V \tag{3.10}$$

3.8.3 Multi-Head Attention

Multi-Head Attention (MHA) allows the model to jointly attend to information from different representation subspaces: instead of performing a single attention function with d_k-dimensional queries and keys and d_v-dimensional values, the queries, keys and values are projected with h different learned projections, each with size $\frac{d_k}{h}$ for the queries and the keys, and size $\frac{d_v}{h}$ for the values. Attention is performed in parallel across the h heads, yielding $\frac{d_v}{h}$-dimensional output values. These are concatenated and once again projected with learned projection W^O, resulting in the final d_v-dimensional values. Let $O_i = Attention(X \cdot W_i^Q, Y \cdot W_i^K, Z \cdot W_i^V)$, then [47]:

$$MHA(X, Y, Z) = Concatenate[O_1, \cdots, O_h] \cdot W^O \tag{3.11}$$

3.8.4 Bahdanau Attention

Proposed by Bahdanau et al. [2], this was the original method that started the *attention mechanisms* revolution. In previous encoder-decoder architectures for Sequence-to-Sequence (Seq2Seq) tasks, the decoder would use a context vector c obtained from the encoder which represented the whole sequence. This had the limitation that the encoder had to embed all the relevant information corresponding to the input sequence into the same fixed-length vector, to be used by the decoder to generate the output sequence. *Bahdanau attention* changed this, allowing the context vector to be different for each prediction of the decoder, which has the benefit of being able to emphasize different kinds of information from the input sequence, at each step.

Let us now revisit the original formulation by Bahdanau et al. [2] for Neural Machine Translation (NMT).

Main Idea

When using the same context vector c across the whole prediction, the decoder defines a probability over the translation $(\hat{y}_1, \cdots, \hat{y}_{T_y})$ by decomposing the joint probability into the ordered conditionals:

$$p(\hat{y}_1, \cdots, \hat{y}_{T_y}) = \prod_{t=1}^{T} p(\hat{y}_t \mid \{\hat{y}_1, \cdots, \hat{y}_{t-1}\}, c) \tag{3.12}$$

where an RNN models each conditional probability as:

$$p(\hat{y}_t \mid \{\hat{y}_1, \cdots, \hat{y}_{t-1}\}, c) = g(\hat{y}_{t-1}, s_t, c), \tag{3.13}$$

where g is a nonlinear function that outputs the probability of $\hat{y}_t$, s_t is the hidden state of the RNN, and $c = q(\{h_1, \cdots, h_{T_x}\})$ is a vector generated from the sequence of the hidden states given by the encoder when processing the input $(x_1, \cdots, x_{T_x})$. Now, $h_t = f(x_t, h_{t-1})$, where f is an RNN-based model and q are some nonlinear functions. Previous work by Sutskever et al. [45] used an LSTM as f and $c = h_T = q(\{h_1, \cdots, h_{T_x}\})$.

In the work proposed by Bahdanau et al. [2], the conditional probability is defined as:

$$p(\hat{y}_i \mid \hat{y}_1, \cdots, \hat{y}_{i-1}, x_1, \cdots, x_{T_x}) = g(\hat{y}_{i-1}, s_i, c_i), \tag{3.14}$$

where s_i is an RNN hidden state for time i, computed by $s_i = f(s_{i-1}, \hat{y}_{i-1}, c_i)$. Note that unlike the previously described encoder-decoder approach, here the probability is conditioned on a distinct context vector c_i for each output element $\hat{y}_i$. The context vector c_i depends on a sequence of hidden states $H = (h_1, \cdots, h_{T_x})$, each of which is obtained by the encoder, considering the sequence (or part of it) and a particular element x_i.

If the encoder uses a model that looks at the whole sequence before generating the hidden states for each element (such as the Bidirectional Recurrent Neural Network (Bidirectional RNN)), then each hidden state h_i contains information about the whole input sequence, with a strong focus on the parts surrounding the i-th element. On the other hand, if the encoder uses a simple RNN model, each hidden state will contain information only about the previous elements in the sequence.

In this model, the *attention weights* are given by the $Similarity(s_{i-1}, h_j)$, which is the probability that the output element $\hat{y}_i$ is aligned to a source element x_j; then, the i-th context vector c_i is the expected hidden state over all the hidden states where the probabilities correspond to the *attention weights*. Intuitively, this implements an *attention mechanism* in the decoder. The decoder decides parts of the source sequence to pay *attention* to. By letting the decoder have an *attention mechanism*, we relieve the encoder from the burden of having to encode all information in the source sequence into a fixed-length vector, and the information can be spread throughout the sequence of hidden states, which can be selectively retrieved by the decoder accordingly.

Formulation

Following the previous definitions of attention already given, we present the formulation of *Bahdanau attention* under the same notation: the query is the previous state of the decoder s_{i-1}, and both, the keys and the values, are the set of hidden states of the encoder H (i.e., each value associated to a key is the key itself). The context vector is computed as a weighted sum of the hidden states from the encoder:

$$c_i = \mathit{Attention}(s_{i-1}, H, H) = \sum_{j=1}^{T_x} \mathit{Similarity}(s_{i-1}, h_j) \cdot h_j \tag{3.15}$$

where

$$\mathit{Similarity}(s_{i-1}, h_j) = \frac{\exp\Big(\mathit{AlignmentScores}(s_{i-1}, h_j)\Big)}{\sum_{k=1}^{T_x} \exp\Big(\mathit{AlignmentScores}(s_{i-1}, h_k)\Big)} \tag{3.16}$$

and

$$\mathit{AlignmentScores}(s_{i-1}, h_j) = \tanh\Big(s_{i-1} \cdot W^s + h_j \cdot W^h\Big) \cdot w^\mathsf{T} \tag{3.17}$$

In the expressions, $s_{i-1} \in \mathbb{R}^{1\times d_s}$, $W^s \in \mathbb{R}^{d_s \times d_k}$, $h_j \in \mathbb{R}^{1\times d_h}$, $W^h \in \mathbb{R}^{d_h \times d_k}$, $w \in \mathbb{R}^{1\times d_k}$; $\mathit{AlignmentScores}(s_{i-1}, h_j)$ indicate how well the inputs around position j, and the output at position i match, and this is normalized with $\mathit{Softmax}$ in order to produce the $\mathit{Similarity}$. The context vector is utilized by concatenating it to the embedding of the input word of the decoder:

$$\widetilde{y}_{i-1} = \mathit{Cat}[c_i;\ \hat{y}_{i-1}] \tag{3.18}$$

Finally, $\widetilde{y}_{i-1}$ is fed together with s_{i-1} to the decoder RNN to produce the output o_i and hidden state s_i. Let $W^y \in \mathbb{R}^{d_o \times |Y|}$; the output o_i is fed to a classifier:

$$p(y \mid \hat{y}_1, \cdots, \hat{y}_{i-1},\ \mathbf{x}) = \mathit{Softmax}_y(o_i \cdot W^y) \tag{3.19}$$

$$\hat{y}_i = \arg\max_y\ p(y \mid \hat{y}_1, \cdots, \hat{y}_{i-1},\ \mathbf{x}) \tag{3.20}$$

3.8.5 Luong Attention

Luong et al. [34] propose several alignment scores, together with a *local attention mechanism* that selectively focuses only on some of the elements of the input sequence, in an attempt to improve the computational performance of the model. They refer to the classical attention mechanism as *global attention mechanism*. We review their *global attention mechanism*, and their proposed alignment scores.

Luong attention utilizes the context vector differently than *Bahdanau attention*: rather than as an input to the decoder, it transforms the output o_i of the decoder with the *attention mechanism*.

$$\tilde{o}_i = \tanh\left(Cat[c_i;\ o_i] \cdot W^o\right) \tag{3.21}$$

Similar to *Bahdanau attention*, the final classifier is defined as:

$$p(y \mid \hat{y}_1, \cdots, \hat{y}_{i-1}, \mathbf{x}) = Softmax_y(\tilde{o}_i \cdot W^y) \tag{3.22}$$

Just as seen previously, the context vector c_i is given by the resulting *attention values*. In order to obtain c_i, the preliminary output o_i produced by the decoder RNN is used as follows:

$$c_i = Attention(o_i, H, H) = \sum_{j=1}^{T_x} Similarity(o_i, h_j) \cdot h_j \tag{3.23}$$

where

$$Similarity(o_i, h_j) = \frac{\exp\Big(AlignmentScores(o_i, h_j)\Big)}{\sum_{k=1}^{T_x} \exp\Big(AlignmentScores(o_i, h_k)\Big)} \tag{3.24}$$

and

$$AlignmentScores(o_i, h_j) = \begin{cases} o_i \cdot h_j^{\mathsf{T}} & Dot \\ o_i \cdot W^g \cdot h_j^{\mathsf{T}} & General \\ \tanh\left(Cat[o_i;\ h_j] \cdot W^c\right) \cdot w^{\mathsf{T}} & Concat \end{cases} \tag{3.25}$$

where $o_i \in \mathbb{R}^{1\times d_o}$, $h_j \in \mathbb{R}^{1\times d_h}$, $W^g \in \mathbb{R}^{d_o\times d_h}$, $W^c \in \mathbb{R}^{d_o+d_h\times d_k}$, $w \in \mathbb{R}^{1\times d_k}$.

3.9 Transformers

Transformer is a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output.

The RNN has particular disadvantages in comparison to the Transformer architecture. Among them are:

- *(Long-range dependencies)* Usually solved using attention mechanisms.
- *(Gradient vanishing and explosion)* When the gradient is passed back through many time steps, it tends to grow or vanish.
- *(Large number of training steps)* RNNs need to be unrolled for as many steps as corresponds in the sequence. The optimization tends to be difficult because of the parameter sharing in the unfolded network.
- *(Recurrence prevents parallel computation)* Due to the sequential nature of RNNs.

Properties of the *Transformer*:

- *(Facilitate long-range dependencies)* Connections can be drawn among any parts of the sequence. Long-range dependencies have the same prior likelihood of being used than short-range dependencies.
- *(No gradient vanishing and explosion)* Instead of having a number of computations that grows linearly with the length of the sequence, we do the computation for the entire sequence simultaneously, passing it through a constant amount of layers.
- *(Fewer training steps)* Easier to optimize.
- *(No recurrence)* Faster training due to parallelization is possible using *Teacher-Forcing* (note that decoder inference still needs to be sequential).

3.9.1 Architecture of the Transformer

Designed for machine translation as an encoder-decoder architecture. In contrast to RNN-based models in which sentences are processed word by word, the *Transformer Encoder* processes all words in parallel.

The *Transformer Encoder* consists of MHA, *Layer Normalization* and a Feed-Forward Neural Network (FFNN). The input to the encoder is a vector of word embeddings representing the input sentence. Since we wish to treat the input as a sequence rather than a set, we use a *Positional Encoder* to inject a value representing the position of the word in the sentence. Then, the MHA will compute the attention between all pairs of embeddings, producing for each word a better embedding containing additional relational information.

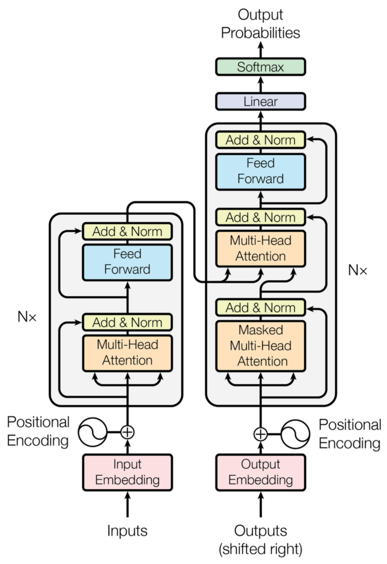

Figure 3.6: Transformer model by Vaswani et al. [48]

The *Transformer Encoder* can be stacked: the first layer combines words with words, the second layer combines pairs with pairs, etc. Following the MHA is an *Add&Norm* layer, which passes a skip connection preceding MHA, and applies *Layer Normalization* by Ba et al. [1]. This is followed by a FFNN, which is again followed by an *Add&Norm* layer. The output of the encoder is a set of embeddings — one per word — each capturing the information of the original word embedding along with the attention given to the other words.

The *Transformer Decoder* is tasked to output a sequence of labels corresponding to translated words in the target dictionary. The decoder also uses a *Positional Encoder* at its input and can be stacked multiple times. In the *Transformer Decoder* we have a masked MHA, which performs self-attention on the embeddings of the target words at the input when applying the *Teacher-Forcing* training strategy. When generating the word $\hat{y}_k$ during training, the target words preceding in position $y_1, \cdots, y_{k-1}$, are used as input instead of the generated words $\hat{y}_1, \cdots, \hat{y}_{k-1}$; this makes training easier, and allows us to parallelize decoding for training. The mask restricts the attention of the decoder to the target words preceding the current target:

$$MaskedAttn(Q, K, V) = Softmax\left(\frac{Q \cdot K^{\intercal} + M}{\sqrt{d_k}}\right) \cdot V \tag{3.26}$$

where M is a mask matrix filled with 0 and $-\infty$, ensuring the probability of masked elements will be zero after applying the $Softmax$ activation, and the distribution will be proper (adding up to 1). Then, another MHA block follows, which associates translated words to original input words. This is followed by a FFNN. Then the output of the *Transformer Decoder* is fed into a classifier which outputs a distribution over words in the target dictionary.

Chapter 4

Proposed Methods

Within this thesis, two DL approaches for solving the $1 \mid r_j \mid \sum_{j \in J} U_j$ are introduced. Both approaches try to predict whether or not a job is tardy in an optimal solution. The choice of the network architecture is motivated by the analogy between scheduling and Natural Language Processing (NLP) problems. In this case, a schedule can be seen as a sentence to be processed within an NLP task, where the vector representation concerning each job (release date, processing time, due date) is analogous to a word. The first approach uses an LSTM-based *Encoder-Decoder* architecture with *attention mechanisms*; in the second approach, a *Transformer*-based architecture is used.

The main components of the proposed methods are explained in the following sections, namely: 1) a data generation method by means of which we implicitly select a target distribution to be learned; 2) several normalization approaches that standardize the input to a specific range; 3) the proposed models for tardiness prediction explained in detail; 4) integration of the tardiness prediction models with a Constraint Programming (CP) solver, in order to produce the final schedules; 6) performance measures by means of which we can assess the effectiveness of the proposed method.

4.1 Data Generation

The data generation method selects a target distribution that we sample to produce the training, validation and testing data. The target distribution provides a characterization of scheduling problems for which the models are optimized. Deviations from the target distribution will result in a degraded performance.

The data generation method utilized is described in the work by Valente and Alves [46]. Let us consider J to be the set of all jobs in a scheduling instance, and let each job be $j \in J$. A set of problems of sizes $\{5, \cdots, 160\}$ was randomly generated in the following way: for each job j an integer processing time p_j was sampled from the uniform distribution Unif$(1, 10)$. For each job j, an integer release date r_j was sampled from the uniform distribution Unif$(0, \alpha \cdot \sum_{j \in J} p_j)$ where α was set at 0.25, 0.50 and 0.75. Instead of determining due dates directly, the slack times between a job's due date and its earliest possible completion time were generated. For each job j, an integer due date slack s^d_j was sampled from the uniform distribution Unif$(0, \beta \cdot \sum_{j \in J} p_j)$. The due date slack range β was set at 0.10, 0.25 and 0.50. The due date d_j of j was then set equal to $d_j = (r_j + p_j) + s^d_j$.

The solutions of the generated instances were produced by two Integer Linear Programming (ILP)s, which are designed to solve two corresponding criteria. The first ILP solves the standard problem of $1 \mid r_j \mid \sum_{j \in J} U_j$ for the purpose of obtaining the optimal value of the objective function. The second ILP solves the problem $1 \mid r_j \mid \sum_{j \in J} U_j$, which searches for the optimal solution that minimizes the difference between the release dates and the start times of each job. The purpose of this two-fold process is to characterize the solutions guiding the learning of the model: since the model is able to predict only one optimal solution per instance, the data contains only one class of optimal solutions.

Let us consider T to be the set of available time units on the resource; r_j, the release date of the job j; p_j, the processing time of the job j; d_j, the due date of the job j. Let us define the indicator variable to be optimized:

$$x_{jt} = \begin{cases} 1 & \text{if job } j \text{ starts at the time step } t, \\ 0 & \text{otherwise.} \end{cases}$$

Finally, the variable $OptVal$ is set as the optimal value of the objective function of the first ILP. In addition, is used as a constraint in the second ILP, so that the solver searches for a new solution only in the space of optimal solutions.

The first ILP is described in the following lines:

$$\begin{aligned} \text{minimize} \quad & \left\{ \sum_{j \in J} \sum_{t \in T \,:\, t > d_j - p_j} x_{jt} \right\} \longrightarrow OptVal \\ \text{subject to} \quad & x_{jt} = 0 && \forall j \in J,\ t \in T : t < r_j \\ & \sum_{j \in J} \sum_{t' \in [t - p_j + 1,\ t]} x_{jt'} \leq 1 && \forall t \in T \\ & \sum_{t \in T} x_{jt} \geq 1 && \forall j \in J \end{aligned}$$

The seconds ILP is described in the following lines:

$$
\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{j\in J}\sum_{t\in T}\{t\cdot x_{jt} - r_j\} & \\
\text{subject to} & \displaystyle\sum_{j\in J}\;\sum_{t\in T\,:\,t>d_j-p_j} x_{jt} = OptVal & \\
& x_{jt} = 0 & \forall j\in J,\;\; t\in T : t < r_j \\
& \displaystyle\sum_{j\in J}\;\sum_{t'\in[t-p_j+1,\; t]} x_{jt'} \le 1 & \forall t\in T \\
& \displaystyle\sum_{t\in T} x_{jt} \ge 1 & \forall j\in J
\end{array}
$$

4.2 Normalization of the Input to the NN Models

The input to the NN is normalized. The reason for it is that we'll generally have different scales for each of the input features in the dataset. This situation could influence the final results for some of the inputs, with an imbalance not due to the intrinsic nature of the data but simply to their original measurement scales. Normalizing all features in the same range avoids this type of problem.

The normalizations considered are the instance-wise scaling normalizations with scalar multiplier $\frac{1}{k}$, which scales in the same way all release dates, processing times and due dates pertaining to the corresponding instance. Thus, k is the normalization factor scaling x-axis of the schedule. In this work we assume several normalization factors, that are defined as:

$$
\begin{aligned}
k_{mrsp} &\triangleq \max_{j\in J:\, r_j} r + \sum_{j\in J:\, p_j} p \\
k_{sp} &\triangleq \sum_{j\in J:\, p_j} p \\
k_{mspmd} &\triangleq \max\left\{\sum_{j\in J:\, p_j} p,\;\; \max_{j\in J:\, r_j} d\right\} \\
k_{md} &\triangleq \max_{j\in J:\, r_j} d
\end{aligned}
$$

The impact of different normalization factors is evaluated within the next chapter.

4.3 Proposed Models for Tardiness Prediction

4.3.1 Definitions

The following definitions are used in the description of both proposed NN models.

Let the dataset $D = (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i=1}^{n} \sim P_D$ be a collection of iid realizations of pairs $(\mathbf{x}, \mathbf{y})$ sampled from the real data distribution P_D. Let $\mathbf{x} = (x_1, \cdots, x_n)$, let $J_{\mathbf{x}}$ be the job indeces for instance $(\mathbf{x}, \mathbf{y})$, and let $x_j \triangleq \{(r_j, p_j, d_j)\}$ represent a job j, where r_j, p_j, d_j are the release date, the processing time and the due date of the job, respectively.

Let $\mathbf{o} = (o_1, \cdots, o_n)$, and let o_j be a binary indicator whether job j is tardy in the optimal solution, defined as:

$$o_j \triangleq \begin{cases} 1 & \text{if job } j \text{ is tardy in the optimal solution} \\ 0 & \text{otherwise} \end{cases}$$

Let $\mathbf{e} = (e_1, \cdots, e_n)$, and let e_j be the predicted probability of a job j of being tardy in the optimal solution. Furthermore, we define v being the predicted amount of tardy jobs in the optimal solution.

Then, the DL-based model is defined in equation 4.1, with $\Theta = (\theta_e, \theta_v, \theta_{shared})$, where θ_{shared} represents common parameters, and θ_e, θ_v represent specific parameters corresponding to tardiness prediction and amount of tardy jobs prediction, respectively; n is the amount of jobs in the instance. The idea of both proposed methods is based on estimating the following two functions:

$$\begin{aligned} f_e(x_1, x_2, \cdots, x_n; \theta_e, \theta_{shared}) &= (e_1, e_2, \cdots, e_n) \approx \mathbb{P}(o_j = 1 \mid \mathbf{x};\ \Theta)_{j=1}^{n} \\ f_v(x_1, x_2, \cdots, x_n; \theta_v, \theta_{shared}) &= v \approx \sum_{j \in J} o_j \end{aligned} \tag{4.1}$$

4.3.2 Model 1: Structured Tardiness Probability $\mathbb{P}(\mathbf{o} \mid \mathbf{x};\ \Theta)$ Using the LSTM-based Sequence-to-Sequence Model

The first NN model is based on the *Encoder-Decoder* architecture, extended with the Luong attention mechanism.

The architecture of the NN is represented in Fig. 4.1.

The *Encoder* accepts the sequence of jobs $(x_1, x_2, \cdots, x_n)$ in EDD order. It transforms the sequence to the latent space, producing the hidden states which are then passed to the *Decoder*.

The *Decoder* accepts the set of the hidden states from the *Encoder* and generates the probability of tardiness for each job at each time step t. The probability is then passed together with the *Decoder*'s hidden state back to the *Decoder* in the next time step. After n time steps the *Decoder*-loop terminates as it has processed all the n jobs in the sequence.

The *Regressor* is used to predict the amount of tardy jobs in the instance. It is directly connected to the *Encoder*, accepting the last hidden state as input features. The *Regressor* contains a linear layer and a Rectified Linear Unit (ReLU).

The loss function is a multi-objective criteria: 1) for the tardiness prediction loss function defined in equation 4.2, we use the weighted variant of Binary Cross Entropy (BCE) loss in order to address the issue of imbalance among tardy and not tardy targets in the training data; 2) for the value prediction loss function defined in equation 4.3, we use the Mean Absolute Error (MAE), as it is the preferred criterion for regression tasks over integer targets. The loss function looks in the following way:

$$\begin{aligned} \ell_{BCE} &\triangleq \frac{1}{|D|} \sum_{(\mathbf{x},\mathbf{y}) \in D} \frac{1}{|J_{\mathbf{x}}|} \sum_{j \in J_{\mathbf{x}}} \left\{ w_{\frac{0}{1}} \cdot y_j \cdot \log \hat{y}_j + (1 - y_j) \cdot \log (1 - \hat{y}_j) \right\} \\ w_{\frac{0}{1}} &\triangleq \frac{|\{(\mathbf{x},\mathbf{y}) \in D : y_j = 0\}|}{|\{(\mathbf{x},\mathbf{y}) \in D : y_j = 1\}|} \\ \hat{y}_j &= f_e(\mathbf{x}; \theta_e, \theta_{shared})_j \end{aligned} \tag{4.2}$$

$$\begin{aligned} \ell_{MAE} &\triangleq \frac{1}{|D|} \sum_{(\mathbf{x},\mathbf{y}) \in D} \frac{1}{|J_{\mathbf{x}}|} \sum_{j \in J_{\mathbf{x}}} |y_j - \hat{y}_j| \\ \hat{y}_j &= f_v(\mathbf{x}; \theta_v, \theta_{shared})_j \end{aligned} \tag{4.3}$$

$$\ell \triangleq \alpha \cdot \ell_{BCE} + \beta \cdot \ell_{MAE} \tag{4.4}$$

The architecture is represented in Fig. 4.1.

The tardiness assignment is done in the following way: we take predicted by the Regressor amount of tardy jobs with the highest probability e_j of being tardy and set them as tardy, so the remaining jobs are set as early.

During this research one more LSTM-based Sequence-to-Sequence Model approach was examined. The model was considering several optimal solutions for labelling an instance. The preliminary results were promising since the model could find a way to generalize in an easier way by combining convenient optimal solutions from each instance. The main disadvantage and the reason

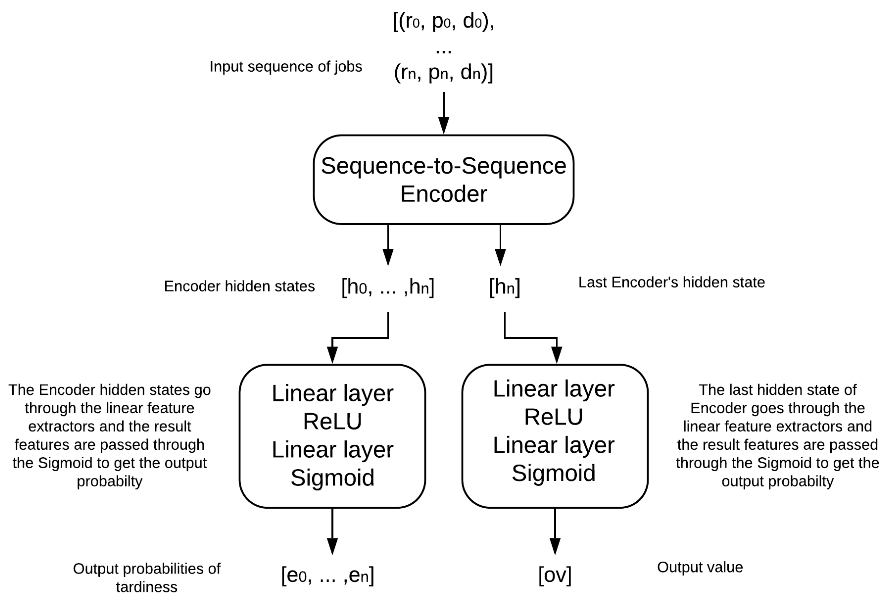
[(r0, p0, d0),
...
(rn, pn, dn)]
Input sequence of jobs
Sequence-to-Sequence Encoder
Encoder hidden states
[h0, ... ,hn]
[hn]
Last Encoder's hidden state
Linear layer ReLU Linear layer Sigmoid
Linear layer ReLU Linear layer Sigmoid
The Encoder hidden states go through the linear feature extractors and the result features are passed through the Sigmoid to get the output probabilty
The last hidden state of Encoder goes through the linear feature extractors and the result features are passed through the Sigmoid to get the output probabilty
Output probabilities of tardiness
[e0, ... ,en]
[ov]
Output value

Figure 4.1: LSTM-based *Encoder-Decoder* model architecture.

why this was not explored further is due to the time consumption of generating all the optimal solutions.

The second proposed approach is using the Transformer architecture and it is fully based on the attention mechanism.

4.3.3 Model 2: Structured Tardiness Probability $\mathbb{P}(\mathrm{o} \mid \mathrm{x};\, \Theta)$ Using the Transformer-based Model

The architecture is used to give a structured prediction of tardiness to the sequence of jobs $(x_1, x_2, \cdots, x_n)$, defining the instance of size n.

The model architecture is represented in Fig. 4.2. It includes the *Transformer Encoder*, a *Classifier* and a *Regressor*. The *Transformer Encoder* outputs the features for each job, which are passed to the *Classifier* to get the final probability for each job. The *Regressor* takes the *Transformer Encoder* features and predicts the proportion of jobs in the sequence that are tardy, $\frac{|\{j \in J : o_j = 1\}|}{|J|}$. While the *Regressor* outputs directly the proportion just described, the tardiness *Classifier* outputs a feature vector. This is done in favor of the BCE with logits loss, which together with the feature vector takes the dataset statistics in the input. This loss is numerically more stable than the standard BCE loss since it takes into account the fact, that the amount of the tardy and non-tardy jobs across the training dataset is uneven. The target loss function is the combination of two, as in the LSTM-based Seq2Seq

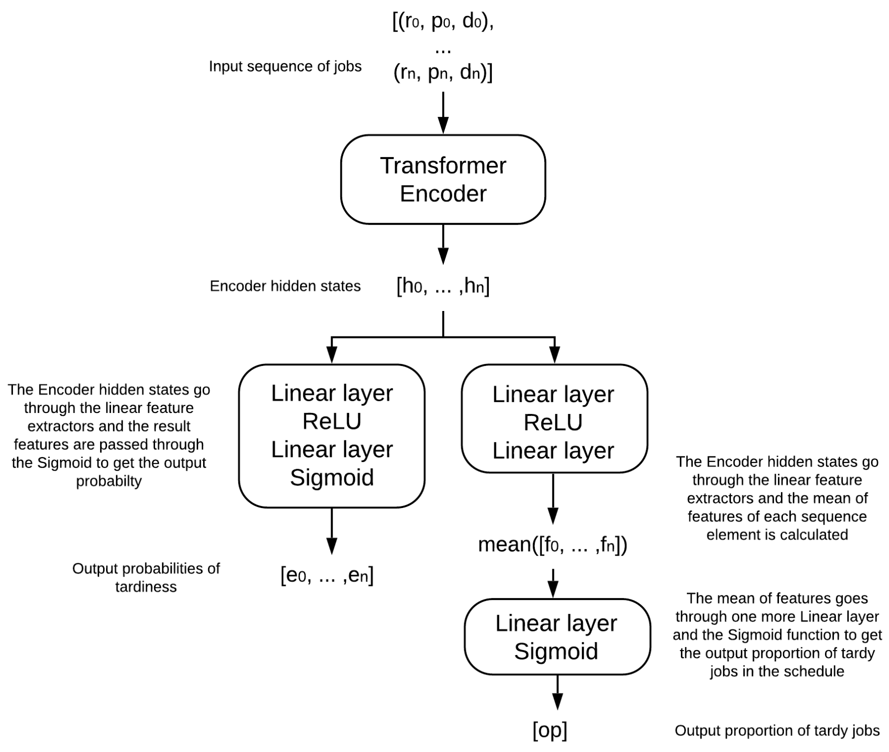
Input sequence of jobs
[(r0, p0, d0),
...
(rn, pn, dn)]
Transformer Encoder
Encoder hidden states
[h0, ... ,hn]
The Encoder hidden states go through the linear feature extractors and the result features are passed through the Sigmoid to get the output probabilty
Linear layer ReLU Linear layer Sigmoid
Linear layer ReLU Linear layer
The Encoder hidden states go through the linear feature extractors and the mean of features of each sequence element is calculated
mean([f0, ... ,fn])
Output probabilities of tardiness
[e0, ... ,en]
Linear layer Sigmoid
The mean of features goes through one more Linear layer and the Sigmoid function to get the output proportion of tardy jobs in the schedule
[op]
Output proportion of tardy jobs

Figure 4.2: *Transformer*-based model architecture.

model. The MAE loss is defined below; the rest of definitions pertaining to the loss function are identical to those in the previous model.

$$\begin{aligned} \ell_{MAE} &\triangleq \frac{1}{|D|} \sum_{(\mathbf{x},\mathbf{y}) \in D} \frac{1}{|J_{\mathbf{x}}|} \sum_{j \in J_{\mathbf{x}}} |y_j - \hat{y}_j| \\ \hat{y}_j &= |J_{\mathbf{x}}| \cdot f_v(\mathbf{x}; \theta_v, \theta_{shared})_j \end{aligned} \tag{4.5}$$

The architecture is represented in Fig. 4.1.

The tardiness assignment is done in the following way: we take v jobs with the highest probability of being tardy and set them as tardy, so the remaining jobs are set as early.

The standard *Transformer* architecture also includes the *Positional Encoder*. It is not used in this work, since in our problem, the order of the input sequence can be considered as not carrying essential information (in contrast to NLP tasks). That is, there might exist such an ordering (e.g. based on the EDD dispatching rule), but we don't know for sure. Therefore, liberating the model from this assumption could have a positive effect in its performance.

In consequence, we proceed to adopt an order-invariant formulation of the *Transformer Encoder*.

The *Transformer Decoder* is not used in the current approach for two reasons. The first one is the performance issue: in both, inference and training, the decoder assumes a sequential processing of its input. It adduces a linear computational complexity growth $O(n)$ with respect to the size of the instance $n = |J|$. Even though the issue during training can be circumvented using *Teacher-Forcing* in the *Transformer Decoder*, it remains a problem at inference time. The second reason is that the *Encoder* already takes into account all the jobs in the sequence to extract the features needed for the prediction. Therefore, it considers them all together and how they interact with each other; in other words, the *Transformer Encoder* learns to compute features for each job considering the whole structure of the problem. Nonetheless, to investigate the complete *Transformer* (including the *Transformer Decoder*) would be an interesting follow-up to this work.

The accuracy criteria of both models are described in the following section.

4.4 Accuracy Criteria of the NN

Both NN models have two evaluation criteria during validation phase:

- **Tardiness accuracy.** The tardiness accuracy represents the percentage of instances, where the tardiness assignment is predicted correctly. In other words, the NN predicted correctly all the tardy jobs in the instance, comparing to the tardiness in the label. This evaluation criterion in used during validation phase to choose the best model.
- **Value accuracy.** The value accuracy represents the percentage of instances, where the value of the objective function is predicted correctly, comparing to the label. This evaluation criteria is used only for the monitoring the progress of learning.

After the best trained model is obtained, we would like to construct the final schedule. We use the prediction of the NN in the construction of the final schedule with the help of Constraint Programming.

4.5 Construction of the Schedule with the CP

As a remainder, in the output of the NN we get the probabilities of tardiness for each job. Then, we retrieve the set of early jobs using those probabilities and the prediction of the objective value (see Section 4.3.1). For the construction of the schedule, the initial problem $1 \mid r_j \mid \sum_{j \in J} U_j$ is transformed into $1 \mid r_j, \widetilde{d_j} \mid \sum_{j \in J} U_j$, by setting the due dates of early jobs as their deadlines.

Then, the CP is searching for the feasible solution of the transformed problem. It accepts at the input the release dates, the processing times and the deadlines for each job. If the feasible solution is found, then the CP returns the destination schedule and the problem is solved. If the solution is infeasible, we remove the deadline of an early job with the lowest probability of being early and pass the modified instance to the CP again. The procedure repeats until the feasible solution is found. In the worst case - if the NN predicted all the tardy jobs wrongly - we remove all the deadlines, so the feasible solution will be found.

Let $T = \{(t_1, t_2, \cdots, t_n)\}$ be flags representing the tardiness for each job. The pseudo code of CP algorithm is in the picture below.

Algorithm 1 CP

```
1:  function SCHEDULE($r$, $p$, $d$, $T$)
2:      $interval \leftarrow max(r) + sum(p)$
3:      $mdl \leftarrow CpoModel()$
4:      $jobs \leftarrow []$
5:      for all i in $\{0, \cdots, length(T)\}$ do
6:          if T[i] = 0 then                ▷ If job at position i is early, set deadline
7:              $jobs.append(mdl.interval($
                    $start = (r[i], interval), size = p[i], end = (r[i], d[i])$
                $))$
8:          else
9:              $jobs.append(mdl.interval(start = (r[i], inteval), size = p[i]))$
10:     $seq \leftarrow mdl.sequence\_var([jobs])$
11:     $mdl \leftarrow mdl.no\_overlap(seq)$
12:     $schedule \leftarrow mdl.solve()$
```

The result schedule is evaluated in the following way: first, the number of tardy jobs is calculated. Then, this number is compared with the number of tardy jobs in the optimal solution using the gap error. Let y be the target value and $\hat{y}$ be the prediction. The gap error is calculated as $gap = \frac{\hat{y}-y}{\hat{y}}$. The *gap* error is used for the performance evaluation in all the conducted experiments, which are described in the following chapter.

4.6 Implementation details

The NN models and the data loaders were implemented in the *PyTorch v1.8.1* framework. For the storage and retrieval of the data the *Postgres* database was used. For the ILP *Gurobi Optimizer v9.0.3* was used. For the CP, the *CPLEX v20.1* solver was used. The NN models were trained on Tesla V100-PCIE-16GB GPU. The evaluation of the NN models was conducted on *macOS Mojave 10.14.6*.

Chapter 5

Experiments

The following experiments describe the behaviour of the proposed solutions on various settings of the models and the data, including: data normalization, instance amounts, instance sizes and model architectures. Due to time constraints, data generation was restricted, and the resulting dataset turned out to be unbalanced, with less instances generated for bigger schedule sizes. The approximate amount of instances generated per instance sizes are: 100000 instances were generated for instances containing 5-10 jobs, 13000 instances were generated for instances containing 11-33 jobs, 7000 instances for instances containing 34-40 jobs, 1200 instances for instances containing 45, 50, 55, and 60 jobs. The instances until size 100 were generated by increasing steps of 5, and from 100 to 160 by increasing steps of 10; 600 instances were generated in both cases. For every instance the data is divided into training, validation and testing in proportion 60%, 20%, 20% respectively. Then, for every instance size where generated equal amount of instances for every combination α and β, where $\alpha \in \{0.25, 0.5, 0.75\}$ and $\beta \in \{0.1, 0.25, 0.75\}$.

All the experiments are conducted on instances where the tardiness of each job is predicted by the selected model (either the *Transformer*-based model or the LSTM-based model), and the output labels are sent to the CP solver, effectively restricting the solution space available to the solver with the deadlines for the jobs that are defined to be early by the predictive model (i.e., those that were predicted not to be tardy in the optimal solution).

The plot of each of the following experiments shows (per instance size) the median of the gap error, with its Median Absolute Deviation (MAD), and the mean of the gap error. Note that the MAD is presented while the Standard Deviation (SD) is not, since the MAD is robust to outliers while the SD is strongly affected by them. Graphs with the SD were considered, but these turned out to be cluttered and harder to visualize because of the big SD values that resulted due to the outliers.

For each experiment there is a brief discussion of the results containing conjectures and plausible explanations for the observed behavior. Comparisons are drawn.

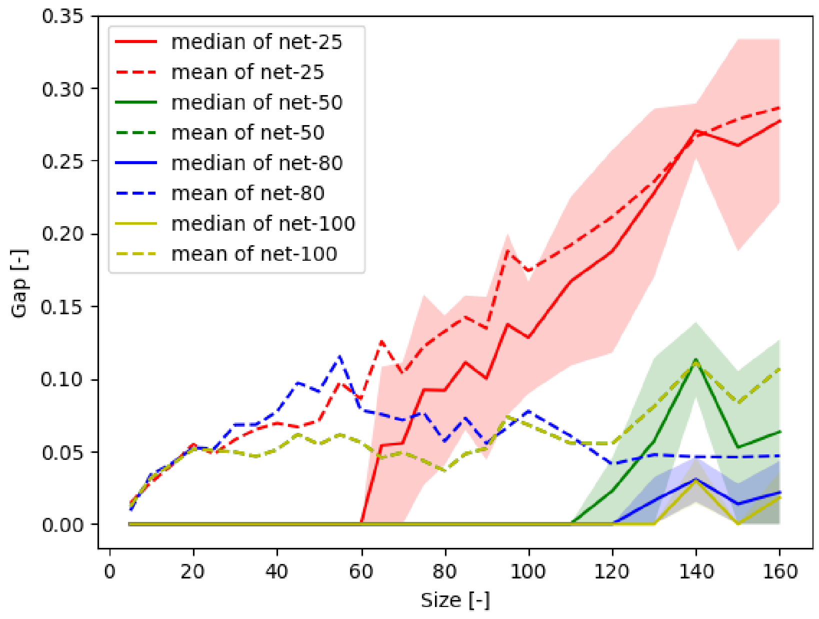

Figure 5.1: Transformer-based models learned on the datasets constructed from various instance sizes

5.1 Training on Varying Instance Sizes

The purpose of this experiment is to examine how the *Transformer*-based model learns on the datasets constructed from various instance sizes.

The model was trained on the datasets containing the instance sizes 5-25 (around 700000 instances), 5-50 (around 850000 instances), 5-80 (around 860000 instances) and 5-100 (around 863000 instances) indicated as net-25, net-50, net-80 and net-100 respectively.

Fig. 5.1 shows that the generalization ability varies according to the various instance sizes included into the dataset: the network learned up to 25 sized instances generalizes well until instances of size 60, then the result gets much worse. Also, as we can see in the picture, the network learned on the instances of sizes up to 50 and 80, generalize well up to the instances of sizes 110 and 120 respectively. The best performance is showed by the network learned on the most sizes (i.e., the model trained with instances of sizes up to 100).

A good follow-up experiment would be to continue this evaluation into bigger instances; this wasn't done in the present work due to time constraints, as generating bigger instances is a very computationally intensive and time-consuming process.

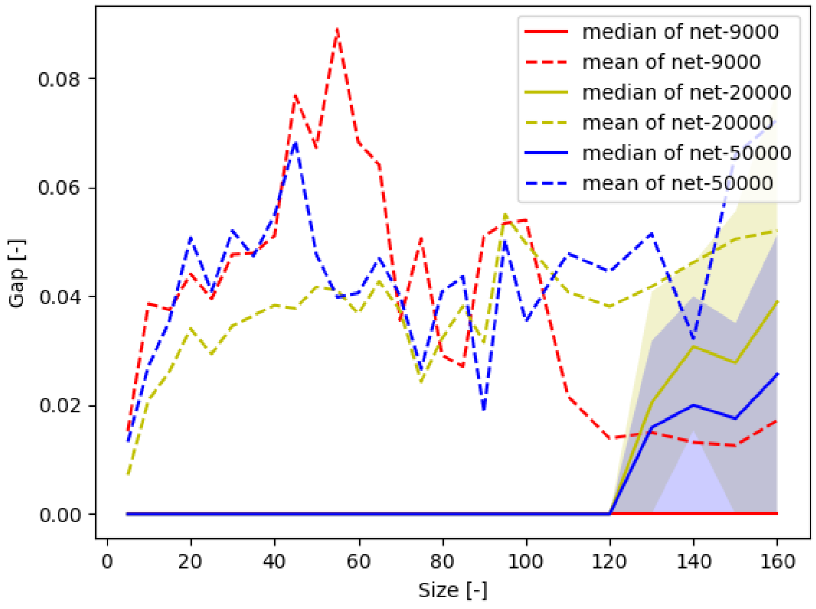

Figure 5.2: Transformer-based models learned on the datasets constructed from various amounts of data per instance size.

5.2 Training on Datasets with Varying Amounts of Data per Instance Size

The purpose of this experiment is to examine how the *Transformer*-based model learns on datasets containing various numbers of instances per an instance size.

Three scenarios of training the *Transformer*-based model were considered. In each scenario the maximum number of instances per an instance size used for the whole dataset was of 9000, 20000 and 50000 respectively. The dataset was internally divided into training, validation and testing data. The three corresponding models (net-9000, net-20000, net-50000) were trained on the range of instance sizes 5-100.

The resulting gaps per instance size can be observed in Fig. 5.2. The results show, that the best performance has the network, trained on 9000 of instances. This can be explained by the fact, that reducing the amount of instances leads to more balanced dataset, which affects the result positively.

A good follow-up experiment would be to continue this evaluation into bigger instances; this wasn't done in the present work due to time constraints, as generating bigger instances is a very computationally intensive and time-consuming process.

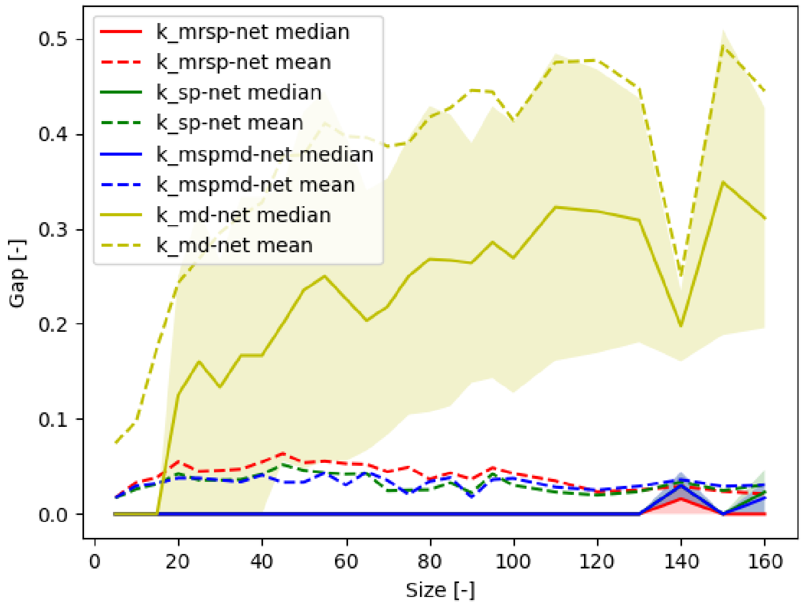

Figure 5.3: Transformer-based models learned with differently normalized inputs.

5.3 Training with Varying Normalization Types

Several normalization schemes were applied to the same data in both the training phase and the testing phase. The results of utilizing each of the considered normalization schemes are presented in Fig. 5.3.

The nomenclature utilized to identify the models of the NN has the form of *normalization*-net, where *normalization* is replaced by the corresponding formula representing the scaling factor normalization.

The normalizations studied were introduced above; refer to section 4.2 for more detail. We consider scaling normalizations of the form $\frac{1}{k}$, where $k \in \{k_{mrsp}, k_{sp}, k_{mspmd}, k_{md}\}$, as previously defined.

In the graph presented in Fig. 5.3, we visualize the mean, the median and the MAD of the gap, for the models trained and tested using each corresponding normalization. The figure shows, that the best results are given by the models trained with the normalizations which include the sum of the processing times. The means of these models are very similar as well as the medians. The worst result is given by the model trained using the normalization which doesn't include the sum of processing times.

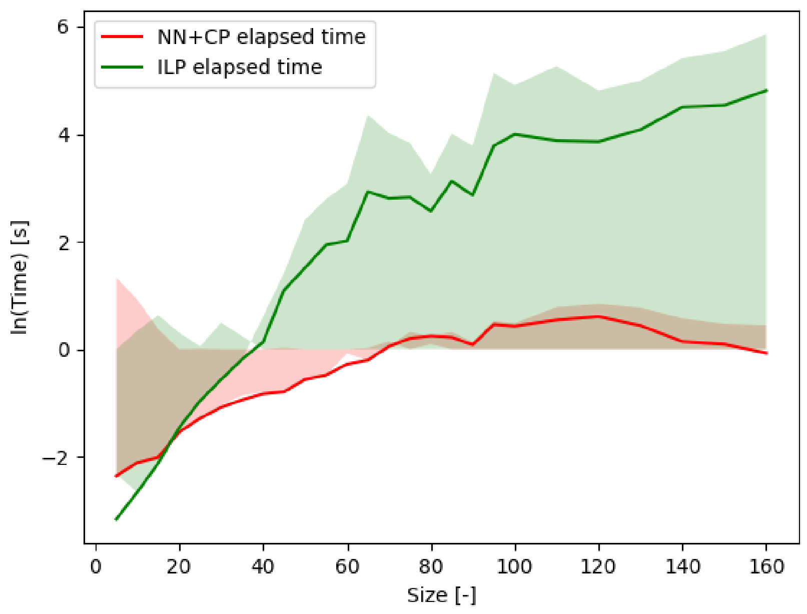

Figure 5.4: The comparison of the time spent in obtaining a solution by the ILP solver and the *Transformer*-based CP solver.

5.4 Prediction Time Comparison Between Transformer-based CP Solver and ILP Solver

The purpose of this experiment is to compare the amount of time spent by the ILP solver producing an optimal solution, with the amount of time taken by the *Transformer*-based CP solver in giving a final prediction of an approx. optimal solution. The logarithm of the computational time is used for enabling a more meaningful visualization of the results. The results of this experiment can be seen in Fig. 5.4. Note that the model utilized in this experiment is the one with the best performance based on the gap metric, the net-100, which can be seen in Fig. 5.1.

We can see how the amount of time the ILP solver takes to find an optimal solution increases in (approx.) exponential fashion with respect to the instance size, while the time taken by the *Transformer*-based CP solver remains (approx.) linear, decreasing to constant towards the end. Note that the solution obtained by the *Transformer*-based CP solver is approximate, as reported in the experiment 5.1; this might explain the variability in time spent towards the higher values in the graph (i.e., there might be a trade-off between decreasing the gap error and decreasing the solver time spent). A good follow-up experiment would be to verify this intuition.

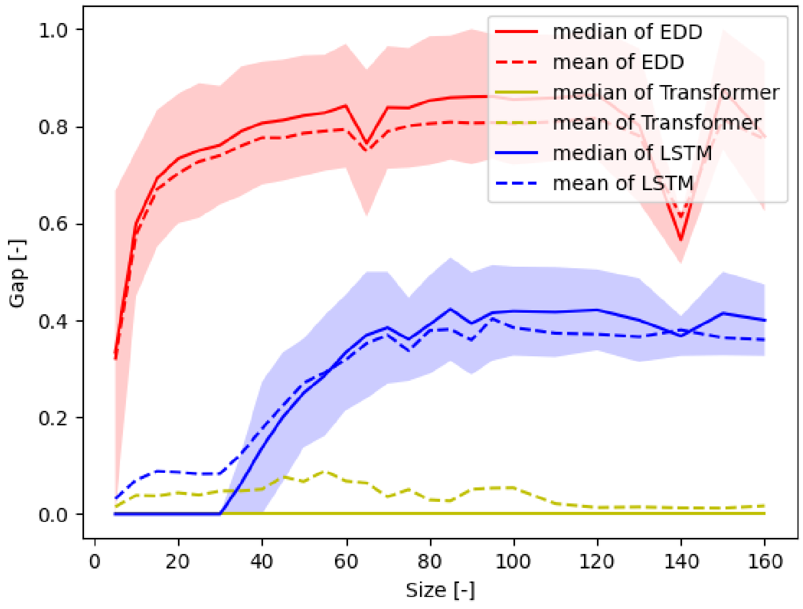

Figure 5.5: The comparison of the *Transformer*-based CP solver, the LSTM-based Seq2Seq CP solver and the EDD algorithm.

5.5 Performance Comparison Between Transformer-based CP Solver and LSTM-based Sequence-to-Sequence CP Solver

The purpose of this experiment is to compare the performance, measured by the optimality gap, of the *Transformer*-based CP solver and the LSTM-based Seq2Seq CP solver.

The *Transformer*-based model and the LSTM-based model were trained on the instance size ranges of 5-50 and 5-40, resp. The training settings are different because the results obtained for the LSTM-based model trained on the instance size range of 5-50 were worse on the testing data, thus it was discarded. The results of this experiment can be observed in Fig. 5.5. The EDD algorithm is evaluated as the baseline method. The LSTM-based solver performs good until the instance size 30, while the *Transformer*-based solver performs stably well all along the given instance sizes and only gets slightly worse after the size 120. One possible cause for the results is that the *Transformer* is indifferent to the order of the input jobs while the LSTM-based model takes it into account while learning. Since it is not clear which order of input jobs would be optimal to use, it can affect the results negatively. An additional possible cause is the recursive nature of the LSTM-based model that results in BPTT, which is known to be harder to optimize [8].

Chapter 6

Conclusion

6.1 Discussion

This work is mainly focused on the design of scheduling algorithms for the problem $1 \mid r_j \mid \sum_{j \in J} U_j$. We proposed two deep learning approaches: the *Transformer*-based CP model and the LSTM-based Seq2Seq CP model. The results of the experiments presented in sections 5.4 and 5.5, have shown that the *Transformer*-based CP model outperforms the LSTM-based Seq2Seq CP model with regards to time complexity of finding a solution, the quality of the solution given, and the generalization ability.

The two models differ not only in their architectures, but also in how the regression task was approached: the LSTM-based Seq2Seq CP model does regression directly over the value of tardy jobs, while the *Transformer*-based CP model learns to predict what is the fraction of jobs in the instance that are tardy. An interesting follow-up experiment would be to verify that the conclusions drawn from the experiments under the present conditions, extend to when both models have the same targets. Given the results of these experiments, the rest of the investigation was focused on the *Transformer*-based CP model.

The results of the experiment found in Section 5.3 have shown that the sum of the processing times is a very important factor in the scaling normalization, and it has to be present in order to obtain a good result. The experiments found in Section 5.1 and Section 5.2 were focused on restricting the dataset: 1) by the instance sizes; and, 2) by the amount of instances per size, respectively. In the first case, the result matched the expectations: the model learns better (i.e., it performs better and generalizes better) having bigger and more varied instance sizes in the dataset. The second experiment has shown that unbalance in the dataset plays a significant role: a smaller amount of data which makes the dataset more balanced leads to better results.

6.2 Future Work

Regarding follow-up work, an important investigation topic to consider is model calibration [23, 37]. The model calibration tunes the predicted probabilities to be representative of the confidence (i.e., the expected correctness) of the prediction, given the dependency of the CP component on the quality of the tardiness probabilities predicted. Also, it would be interesting to try out different loss functions; in particular, the cosine loss is a good candidate given some recent results on small data [4]. Aspects related to the data used for training and testing, such as the balance of the data, the maximum instance size, the distribution of the data, etc., can also be worked on, to further understand the characteristics and limitations of the proposed models. It is very important to derive an understanding of the generalization abilities of the model, not only with respect to increasing instance sizes, but also with respect to distribution shift (for instance, a small change in the relationship between the scales of the release dates, processing times and due dates). An issue that needs emphasizing is the generation of the dataset required for training, evaluation and testing: as mentioned before, the generation of bigger sized instances was not possible due to time restrictions, thus, the resulting dataset turned out to be unbalanced. This could be addressed and done better in future work, with more time for arranging the dataset. Finally, it would be interesting to study a model that learns using more than one optimal solution per instance, since it is possible that it would be able to combine solutions that provide structural insights that the model could leverage to learn a better predictor.

Bibliography

[1] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint*, arXiv:1607.06450.

[2] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint*, arXiv:1409.0473.

[3] Baptiste, P., Peridy, L., and Pinson, E. (2003). A branch and bound to minimize the number of late jobs on a single machine with release time constraints. *European Journal of Operational Research*, 144(1):1–11.

[4] Barz, B. and Denzler, J. (2019). Deep learning on small datasets without pre-training using cosine loss. *arXiv preprint*, arXiv:1901.09054.

[5] Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural combinatorial optimization with reinforcement learning. *arXiv preprint*, arXiv:1611.09940.

[6] Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. *arXiv preprint*, arXiv:1506.03099.

[7] Bengio, Y., Lodi, A., and Prouvost, A. (2018). Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint*, arXiv:1811.06128.

[8] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

[9] Bouška, M., Novák, A., Šůcha, P., Módos, I., and Hanzálek, Z. (2020). Data-driven algorithm for scheduling with total tardiness. *Proceedings of the 9th International Conference on Operations Research and Enterprise Systems*, pages 59–68.

[10] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using

RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

[11] Choi, H.-S., Kim, J.-S., and Lee, D.-H. (2011). Real-time scheduling for reentrant hybrid flow shops: A decision tree based mechanism and its application to a tft-lcd line. *Expert Systems With Applications*, 38(4):3514–3521.

[12] Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2018). Learning combinatorial optimization algorithms over graphs. *arXiv preprint*, arXiv:1704.01665.

[13] Dauzère-Pérès, S. and Sevaux, M. (2003). Using lagrangean relaxation to minimize the weighted number of late jobs on a single machine. *Naval Research Logistics (NRL)*, 50(3):273–288.

[14] Dauzère-Pérès, S. (1995). Minimizing late jobs in the general one machine scheduling problem. *European Journal of Operational Research*, 81(1):134–142.

[15] Dyer, M. E. and Wolsey, L. A. (1990). Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26(2):255–270.

[16] Erschler, J., Fontan, G., Mercé, C., and Roubellat, F. (1983). A new dominance concept in scheduling n jobs on a single machine with ready times and due dates. *Operations Research*, 31(1):114–127.

[17] Evans, R. and Gao, J. (2016). Deepmind ai reduces google data centre cooling bill by 40%. *URL: https://deepmind. com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40*.

[18] Garraffa, M., Shang, L., Della Croce, F., and T'kindt, V. (2018). An exact exponential branch-and-merge algorithm for the single machine total tardiness problem. *Theoretical Computer Science*, 745:133–149.

[19] Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58.

[20] Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. (2003). Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research*, 3:115–143.

[21] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[22] Grabot, B. and Geneste, L. (1994). Dispatching rules in scheduling dispatching rules in scheduling: a fuzzy approach. *International Journal of Production Research*, 32(4):903–915.

[23] Guo, C., Pleiss, G., Sun, Y., and Weinberger, K. Q. (2017). On calibration of modern neural networks. *arXiv preprint*, arXiv:1706.04599.

[24] Hall, N. G. (1986). Scheduling problems with generalized due dates. *IIE transactions*, 18(2):220–222.

[25] Hall, N. G., Sethi, S. P., and Sriskandarajah, C. (1991). On the complexity of generalized due date scheduling problems. *European Journal of Operational Research*, 51(1):100–109.

[26] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[27] Jeong, K.-C. and Kim, Y.-D. (1998a). A real-time scheduling mechanism for a flexible manufacturing system: Using simulation and dispatching rules. *International Journal of Production Research*, 36(9):2609–2626.

[28] Jeong, K.-C. and Kim, Y.-D. (1998b). A real-time scheduling mechanism for a flexible manufacturing system: Using simulation and dispatching rules. *International Journal of Production Research*, 36(9):2609–2626.

[29] Kise, H., Ibaraki, T., and Mine, H. (1978). A solvable case of the one-machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126.

[30] Kool, W., van Hoof, H., and Welling, M. (2019). Attention, learn to solve routing problems! *arXiv preprint*, arXiv:1803.08475.

[31] Lasserre, J. B. and Queyranne, M. (1992). Generic scheduling polyhedra and a new mixed-integer formulation for single-machine scheduling. *International Publisher C.O*, pages 136–149.

[32] Lawler, E. L. (1994). Knapsack-like scheduling problems, the moore-hodgson algorithm and the 'tower of sets' property. *Mathematical and Computer Modelling*, 20(2):91–106.

[33] Lenstra, J., Rinnooy Kan, A., and Brucker, P. (1977). Complexity of machine scheduling problems. In Hammer, P., Johnson, E., Korte, B., and Nemhauser, G., editors, *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 343–362. Elsevier.

[34] Luong, M., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint*, arXiv:1508.04025.

[35] Mao, H., Alizadeh, M., Menache, I., and Kandula, S. (2016). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA. Association for Computing Machinery.

[36] Molaee, E., Moslehi, G., and Reisi, M. (2011). Minimizing maximum earliness and number of tardy jobs in the single machine scheduling problem with availability constraint. *Computers Mathematics with Applications*, 62(9):3622–3641.

[37] Müller, R., Kornblith, S., and Hinton, G. E. (2019). When does label smoothing help? *arXiv preprint*, arXiv:1906.02629.

[38] M'Hallah, R. and Bulfin, R. (2007). Minimizing the weighted number of tardy jobs on a single machine with release dates. *European Journal of Operational Research*, 176(2):727–744.

[39] Ourari, S., Briand, C., and Bouzouia, B. (2009). A MIP approach for the minimization of the number of late jobs in single machine scheduling.

[40] Pinedo, M. L. (2008). *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition.

[41] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

[42] Shahrabi, J., Adibi, M. A., and Mahootchi, M. (2017). A reinforcement learning approach to parameter estimation in dynamic job shop scheduling. *Computers Industrial Engineering*, 110:75–82.

[43] Smith-Miles, K. and Bowly, S. (2015). Generating new test instances by evolving in instance space. *Computers Operations Research*, 63(C):102–113.

[44] Sutskever, I., Vinyals, O., and Le, Q. V. (2014a). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA. MIT Press.

[45] Sutskever, I., Vinyals, O., and Le, Q. V. (2014b). Sequence to sequence learning with neural networks. *arXiv preprint*, arXiv:1409.3215.

[46] Valente, J. M. S. and Alves, R. A. F. S. (2005). An exact approach to early/tardy scheduling with release dates. *Computers Operations Research*, 32(11):2905–2917.

[47] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017a). Attention is all you need. *arXiv preprint*, arXiv:1706.03762.

[48] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017b). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[49] Vinyals, O., Fortunato, M., and Jaitly, N. (2017). Pointer networks. *arXiv preprint*, arXiv:1506.03134.

[50] Wang, Y.-C. and Usher, J. M. (2005). Application of reinforcement learning for agent-based production scheduling. *Engineering Applications of Artificial Intelligence*, 18(1):73–82.

[51] Ye, Y., Ren, X., Wang, J., Xu, L., Guo, W., Huang, W., and Tian, W. (2018). A new approach for resource scheduling with deep reinforcement learning. *arXiv preprint*, arXiv:1806.08122.