Novel Machine Learning Algorithms for NP-hard Combinatorial Problems

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Maschinenbau und Verfahrenstechnik der

Universität Duisburg-Essen

zur Erlangung des akademischen Grades

einer

Doktorin der Ingenieurwissenschaften

(Dr.-Ing.)

genehmigte Dissertation

von

M.Sc. Annika Tonnius

aus

Gütersloh

Tag der Disputation: 15.03.2023

Erstgutachter: Prof. Dr. rer. nat. Johannes Gottschling

Zweitgutachter: Prof. Dr. rer. nat. Robert Martin

// Abstract

For many combinatorial problems encountered in practical applications, there are currently no efficient solution algorithms available. In particular, such problems are often NP-hard and thus especially difficult to solve for larger instances. However, exact solutions are often not necessary in practice. While, depending on the specific problem, algorithms for finding approximate solutions might already be available, methods based on machine learning have not yet found widespread use in this domain.

In general, any systematic approach to NP-hard combinatorial problems by means of machine learning techniques faces two major challenges: First, combinatorial problems are discrete in nature, and it is typically difficult to map discrete results with machine learning methods. Second, due to the large solution space of NP-hard problems, it takes large computational effort, or is even practically impossible, to generate labels for supervised machine learning. Therefore, in this dissertation, we present an approach that circumvents these obstacles, and thus allows for efficiently predicting solutions to combinatorial problems with machine learning. More specifically, we overcome the problem of discrete solutions with a relaxation approach. The problem of label generation, on the other hand, is avoided with label-free learning via the loss function: The loss function represents the objective function, which computes the objective value of the combinatorial problem itself. Instead of labels, the loss function thereby immediately directs the adaption of the parameters during the gradient-based training of the machine learning algorithm.

Although the approach presented here is applicable to a much more general class of problems, we will focus on the so-called hybrid flow shop scheduling problem to demonstrate our methods. It consists of two subproblems: a permutation problem and an assignment problem. We analyze the hybrid flow shop in depth, including a formal mathematical description. Additionally, we give an extensive overview of the theoretical foundations of gradient-based machine learning to provide a theoretical foundation for understanding the proposed approach.

Most importantly, we provide a detailed explanation of the suggested approach of learning via the loss function and the literature that has thus far treated the topic. For each subproblem, we train a neural network and describe the architecture and training method in detail, as well as the connection between the two networks to address the entire hybrid flow shop problem.

To assess the effectiveness of our approach, we compare the results of the neural networks, both individually and combined for the complete approach, with those of a basic Monte Carlo algorithm, as well as a more advanced mixed-integer linear programming approach. The results demonstrate noteworthy benefits in terms of efficiency and scalability of the proposed approach. Finally, we give suggestions for further research concerning the topic.

Zusammenfassung

Für viele kombinatorische Probleme sind derzeit keine hinreichend effizienten exakten Lösungsalgorithmen verfügbar. Insbesondere sind kombinatorische Probleme mit praktischer Relevanz oft NP-schwer und daher für größere Instanzen besonders schwierig zu lösen. Exakte Lösungen sind in der Praxis jedoch oft nicht erforderlich. Obwohl es, je nach Problemstellung, bereits Algorithmen zur Suche nach Näherungslösungen gibt, haben Methoden, die auf maschinellem Lernen basieren, in diesem Bereich bislang noch keine breite Anwendung gefunden.

Systematische Annäherungen an NP-schwere kombinatorische Probleme mittels maschineller Lernverfahren stehen vor zwei großen Herausforderungen: Einerseits sind kombinatorische Probleme von Natur aus diskret, und es ist im Allgemeinen schwierig, diskrete Ergebnisse mit Methoden des maschinellen Lernens abzubilden. Andererseits ist es aufgrund des großen Lösungsraums von NP-schweren Problemen rechenaufwändig bzw. praktisch unmöglich, Labels für überwachtes maschinelles Lernen zu generieren. Diese Dissertation stellt daher einen Ansatz vor, der diese Hürden überwindet und eine effiziente Vorhersage von Lösungen für kombinatorische Probleme durch maschinelles Lernen ermöglicht. Dazu wird das Problem der diskreten Lösungen mit einem Relaxierungsansatz überwunden, während das Problem der Labelgenerierung durch labelfreies Lernen über die Verlustfunktion umgangen wird: Die Verlustfunktion stellt die Zielfunktion dar, die den Zielwert des kombinatorischen Problems selbst berechnet. Anstelle von Labels steuert die Verlustfunktion damit unmittelbar die Anpassung der Parameter während des gradientenbasierten Trainings des maschinellen Lernalgorithmus.

Obwohl der vorgestellte Ansatz auf allgemeinere Probleme anwendbar ist, liegt der Fokus auf der Demonstration der Methoden am sogenannte Hybrid Flow Shop-Problem. Dieses besteht aus zwei Teilproblemen: einem Permutationsproblem und einem Zuordnungsproblem. Das Problem wird eingehend analysiert, einschließlich einer formalen mathematischen Beschreibung. Zudem wird ein umfassender Überblick über die theoretischen Grundlagen des gradientenbasierten maschinellen Lernens bereitgestellt, um eine Basis für das Verständnis des vorgeschlagenen Ansatzes zu schaffen. Insbesondere wird das Lernen über die Verlustfunktion ausführlich erläutert sowie ein Überblick über die Literatur gegeben, die das Thema bisher behandelt hat. Für jedes Teilproblem wird ein neuronales Netz trainiert: Die Architektur sowie die Trainingsmethode werden erklärt und die Verbindung zwischen den beiden Netzen beschrieben, um das Gesamtproblem zu lösen.

Für die Bewertung des Ansatzes erfolgt ein Vergleich der Ausgabe der neuronalen Netze, sowohl separat als auch kombiniert für den gesamten Ansatz, mit den Ergebnissen eines Monte-Carlo-Algorithmus und eines Mixed-Integer-Linear-Programming-Verfahrens (MILP). Dabei zeigen sich signifikante Vorteile in Bezug auf Performance und Skalierbarkeit des vorgestellten Ansatzes.

Contents

List of Figures v

List of Tables ix

List of Symbols xi

List of Abbreviations xiii

1 Introduction 1

1.1 Aim of the Thesis 2

1.2 Structure of the Thesis 3

2 Problem Description 5

2.1 The Hybrid Flow Shop Problem 5

2.1.1 Description 5

2.1.2 Notation 8

2.1.3 Mixed-Integer Linear Program Representation of the Assignment Problem 11

2.1.4 Applications 13

2.2 NP Complexity Classes 15

2.2.1 Complexity of the Hybrid Flow Shop Problem 17

3 Machine Learning 19

3.1 Definition 19

3.1.1 Artificial Intelligence 19

3.1.2 Machine Learning 20

3.1.3 Deep Learning 21

3.2 Structure of Neural Networks . . . 21
3.2.1 Feedforward Neural Networks . . . 21
3.2.2 Regression . . . 23
3.2.3 Classification . . . 26
3.3 Training Process . . . 31
3.3.1 Gradient Descent . . . 31
3.3.2 Optimizers . . . 38
3.3.3 Backpropagation . . . 40
3.3.4 Over- and Underfitting . . . 42
3.4 Deep Learning . . . 45
3.4.1 Convolutional Neural Networks . . . 46
3.4.2 Recurrent Neural Networks . . . 49
3.5 Label-Free Learning via the Loss Function . . . 56

4 Machine Learning for Combinatorial Problems **59**
4.1 Literature review . . . 59
4.2 General Approach . . . 62
4.3 Subproblem 1: Learning the Assignments . . . 65
4.3.1 Assignment Relaxation . . . 66
4.3.2 Loss Function . . . 68
4.3.3 Structure of the Neural Network . . . 72
4.3.4 Training Process . . . 74
4.4 Subproblem 2: Learning the Permutations . . . 77
4.4.1 Permutation Relaxation . . . 77
4.4.2 Loss Function . . . 79
4.4.3 Structure of the Neural Network . . . 80
4.4.4 Training Process . . . 82
4.5 Limitations . . . 83

5 Results **85**
5.1 Evaluation . . . 85
5.1.1 Comparison Algorithms . . . 86

5.1.2 Structure of the Test Data . . . 87
5.1.3 Pure Assignment Problem . . . 88
5.1.4 Pure Permutation Problem . . . 95
5.1.5 Complete Approach . . . 95
5.1.6 Greedy Post-Processing . . . 98

6 Conclusion **101**

6.1 Summary . . . 101
6.2 Outlook . . . 102

Bibliography **103**

List of Figures

2.1 Illustration of the hybrid flow shop scheduling problem with jobs $j = 1 \ldots, J$ and stages $i = 1, \ldots, I$ with M_i machines on stage i 6

2.2 Schedules for different interpretations of the permutation rule for a hybrid flow shop with $J = 3$ jobs, $I = 2$ stages and $M_1 = 3$, $M_2 = 2$ machines on the stages 7

2.3 An optimal schedule for a given job order and given processing times with $J = 7$ jobs, $M = 6$ machines and $I = 3$ stages with two machines each 8

2.4 Multi-stage multi-server internet architecture, cf. [2] 14

2.5 Hierarchical display of complexity classes and their solving difficulty, cf. [45], assuming that $\mathsf{P} \neq \mathsf{NP}$ 16

3.1 Classification of artificial intelligence, machine learning and deep learning, cf. [13, p. 22] . . 19

3.2 Comparison between traditional programming and machine learning [52] 20

3.3 Simple Feed-Forward Neural Network, cf. [9, p. 228] 23

3.4 Plot of the non-linear function $y = x^3 - 3x$ 24

3.5 Layer outputs of a scalar regression problem with one hidden layer comprising three nodes . 25

3.6 Classification example with three groups 26

3.7 Position of the three one-hot encoded labels in $\mathbb{R}^3$ 27

3.8 The activations of an output layer take the values 2, 0.5, 5 and 4.5. They are transformed by a) the argmax function, b) scaling to 1 and c) by the softmax activation function. The transformed values are plotted in the respective sub-figures. 28

3.9 Visualization of the classification results 29

3.10 Output of the last layer, mapped to 3-dimensional space. The gray lines indicate the linear separation of the groups. 30

3.11 Plot of the data points and the corresponding minimum function for the gradient descent example 32

3.12 Function MSE(w) and derivatives for $w = -1$ and $w = 5$ 33

3.13 Learning rate example . . . 34

3.14 Influence of the learning rate η on the learning process for $w_0 = -1$ (first row) and $w_0 = 6$ (second row), cf. [19, p. 312] . . . 35

3.15 Example of learning curve with two minima . . . 36

3.16 Comparison of the gradient descent algorithm with and without momentum, $\eta = 0.02$ and $w_0 = 1.5$. The algorithm converges faster with momentum (b). . . . 36

3.17 Comparison of the gradient descent algorithm with momentum and with Nesterov momentum, $\eta = 0.02$ and $w_0 = 1.5$. The algorithm converges faster with Nesterov momentum (b). . . . 37

3.18 Plots of the current function (red) with $w \approx 2.7$ and $b = 0$, and actual minimum function (green) . . . 38

3.19 Paths of weights and bias adaptions without momentum (a), with momentum (b) and with Nesterov accelerated gradient (c) . . . 39

3.20 Neural network model for the backpropagation example . . . 40

3.21 Comparison of NN models with respect to overfitting . . . 42

3.22 Comparison of NN models with respect to underfitting . . . 44

3.23 Representations of a given input produced by a deep learning Neural Network (cf. [13, p. 28]) 45

3.24 Feature matrix produced by a convolutional layer with a kernel size of 2×2 and an input of size 4×4 (cf. [31, p. 334]) . . . 47

3.25 Feature matrix produced by a max pooling layer with a filter size of 2×2 on an input of size 4×4 (cf. [31, p. 334]) . . . 48

3.26 Structure of a recurrent node (cf. [1, p. 275]) . . . 50

3.27 The vanishing gradient problem over a sequence . . . 52

3.28 Structure of an LSTM cell . . . 53

3.29 Structure of a GRU cell . . . 54

3.30 Schemata of supervised learning with labels and label-free learning . . . 56

4.1 Schedules of two optimal assignments for different permutations of the same problem . . . 63

4.2 General training process of the proposed approach . . . 64

4.3 General inference process of the proposed approach . . . 64

4.4 Schedule with job splitting . . . 67

4.5 Schedule with machine splitting . . . 67

4.6 Examples of job splitting and machine splitting . . . 67

4.7 Neural network architecture of the assignment model . . . 73

4.8 Neural network architecture of the permutation model 80

5.1 Accuracy and deviations from optimum *makespan* values of 1000 predictions for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages and varying numbers of jobs 89

5.2 Accuracy and deviations from optimum makespan values of 1000 predictions for a hybrid flow shop with different machine configurations for $J = 6$ jobs 90

5.3 Accuracy and deviations from optimum *flowtime* values of 1000 predictions for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages with $J = 6$ and $J = 10$ jobs 91

5.4 Comparisons between NN predictions and Monte Carlo results of 1000 problem instances for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages and varying numbers of jobs 92

5.5 Comparison between NN permutation predictions and Monte Carlo results of 1000 problem instances for $J = 10$ and $J = 12$ jobs 96

5.6 Comparison between NN permutation predictions and Monte Carlo results of 1000 problem instances for $J = 12$ and $J = 16$ jobs 97

List of Tables

2.1 Number of possible solutions for the reduced assignment problem and the standard HFS problem with job permutations 17

3.1 x and y coordinates of the points in fig. 3.11 31

5.1 Number of possible assignments for a HFS configuration with I=3 stages and 2 machines on each stage, with varying numbers of jobs J 88

5.2 Comparison of *execution times* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 60 jobs 93

5.3 Comparison of *accuracies* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 20 jobs 93

5.4 Comparison of *mean deviations* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 20 jobs 93

5.5 Relative differences between the NN and MILP results for varying numbers of jobs J and varying time limits for the MILP solver 94

5.6 Mean deviation from optimum for $J = 6$ of the NN predictions and 1000 random permutations 95

5.7 Mean deviation from optimum of the combined prediction of permutation and assignments and random selections for $J = 6$ 96

5.8 Relative differences between the NN and MILP results for varying numbers of jobs J and varying time limits for the MILP solver 97

5.9 Overview of the improvements reached by the greedy algorithm concerning the mean deviation from the optimum, for $J = 6$ jobs and varying tolerances 99

List of Symbols

Hybrid Flow Shop (HFS)

Symbol	Description
$\mathcal{A}$	Set of binary assignments matrices
A	Binary assignments matrix of jobs to machines
A_R	Assignment-relaxed assignments matrix of jobs to machines
C	Completion times matrix
$C_{j,i}$	Completion time of job j on stage i
$C_{\max}$	Makespan objective
C_{sum}	Flowtime objective
f_C	Objective function
I	Number of stages
i	Stage
J	Number of input jobs
j	Input job
k	Position of a job in a sequence
$\mathcal{M}_i$	Set of machines on stage i
M	Sufficiently large number for big-M method
M_i	Number of machines on stage i
m_i	Machine on stage i
$\mathcal{O}$	Set of doubly stochastic permutation matrices
O	Doubly stochastic permutation matrix
P	Processing times matrix
P_σ	Permuted processing times
P_{jm}	Processing time of job j on machine m
σ	Permutation vector
S	Starting times matrix
$S_{j,i}$	Starting time of job j on stage i

Machine Learning (ML)

Symbol	Description
$\widehat{A}$	Rectified assignments prediction matrix
$\widehat{A}_R$	Relaxed assignments prediction matrix
a_j	Activation value of the jth node of a layer
B	Batch size
b_{j0}	Bias values of the jth node of a layer
G	Number of output classes in classification
h	Arbitrary activation function
K	Kernel in a convoluational NN
L	Arbitrary loss function
N	Number of input data points
$\widehat{O}$	Rectified permutation prediction matrix
$\widehat{O}_R$	Relaxed permutation prediction matrix
$\boldsymbol{r}$	Hidden representation in a recurrent NN of an input $\boldsymbol{x}$
w_{ji}	Weight between node i of a layer and node j of the following layer
x	Vector of inputs to a neural network
x_n	Input to a neural network
$\hat{y}$	Predictions of an NN for the inputs x
$\hat{y}_n$	Prediction of an NN for an input x_n
z_j	Output of the jth node of a layer
α	Gradient value or slope
γ	Momentum for gradient descent optimization
υ_t	Size of step t of gradient descent
η	Learning rate
θ	Neural network parameters trained by gradient descent
$\times$	Size indicator
$\odot$	Elementwise matrix multiplication
$*$	Convolutional operation

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CO	Combinatorial Optimization
DEI	Divide-et-Impera
DL	Deep Learning
FIFO	First In First Out
GRU	Gated Recurrent Unit
HFS	Hybrid Flow Shop
LSTM	Long Short-Term Memory
MAJ	Modular Assignment of Jobs
MC	Monte Carlo algorithm
MILP	Mixed-Integer Linear Programming
ML	Machine Learning
MSE	Mean Squared Error
NN	Neural Network
NP	Nondeterministic Polynomial time solvable
NPC	NP-Complete
P	Polynomial time solvable
PGNN	Physics Guided Neural Network
PINN	Physics Informed Neural Network
prmu	Permutation
RL	Reinforcement Learning
RMSprop	Root Mean Squared Propagation
RNN	Recurrent Neural Network
SAT	Boolean SATisfiability problem
SGD	Stochastic Gradient Descent

Chapter 1

Introduction

For many combinatorial problems encountered in practical applications, there are currently no efficient solution algorithms available. In particular, such problems are often NP-hard and thus especially difficult to solve for larger instances.

Countless *exact* approaches to solve such problems have been discussed in the literature, even though these algorithms can only provide solutions to small instances of NP-hard problems within a reasonable amount of time. A well-known representative of an exact algorithm is *linear programming*. Typically, for larger instances, the algorithm is stopped after a defined amount of time and the best computed result so far is returned, which turns it into an *approximation* algorithm. In particular, for NP-hard problems, it is often more practical to employ efficient algorithms which provide only an *approximate* solution, but require significantly less computational effort and stay in an acceptable computation time frame. In general, the choice of an approximation algorithm requires a compromise between the accuracy of the solution and computational performance – the art lies in creating an algorithm that possesses, on the one hand, a short computation time, and on the other hand, provides sufficiently good or, in some instances, optimal solutions. Especially for time-critical tasks or to gain a competitive advantage, we would like to obtain results as quickly as possible.

Then again, there are *machine learning algorithms*: A machine learning *model* has to be *trained in advance*, which may require some time. However, after training, the model can *predict* the results of multiple problem instances very efficiently and concurrently, since the operations comprise mainly matrix operations. Classic (exact) methods like linear programming do not require a pre-trained model; however, the computational effort is high *while solving* the tasks, when we potentially require an answer as quickly as possible. Moreover, classic algorithms are typically not optimized for concurrent problem solving, which is reflected in the comparatively slow problem processing.

Considering a logistics environment where the overall system does not change, e.g. the routes to be scheduled or the processes to be considered stay roughly the same, and only the input parameters vary, e.g. changing distances in a routing problem, it makes sense to train ML models upfront which can be reused for the same process and thus provide quick solutions to the considered problem when needed.

Thus, machine learning seems like a natural choice to solve NP-hard problems. Unfortunately, simple machine learning algorithms can typically not be applied to NP-hard problems without further ado. There are two major obstacles that need to be overcome:

- *Discreteness*: The solutions of NP-hard problems are often of discrete nature, e.g. a sequence that comprises a vector of positive integer values, or assignments that consist of binary variables. Machine learning algorithms are typically not suitable for working with discrete, but rather with *continuous* values.
- *Label generation*: For the training process, classic supervised machine learning algorithms make use of *labeled data*, i.e. problem instances with known (optimal) output. For NP-hard problems, it is often not practically possible to determine the optimal output, or it is too computationally expensive.

These obstacles might be the reason why solving NP-hard problems using machine learning has not yet found widespread use in research.

1.1 Aim of the Thesis

The aim of this thesis is to provide a flexible approach to solve NP-hard combinatorial problem with machine learning by circumventing the aforementioned issues, i.e. the discreteness of combinatorial problems and the challenge of label generation. Additionally, the presented approach offers several advantages compared to classic algorithms solving combinatorial problems and standard machine learning algorithms:

- *Discreteness*: Using a relaxation strategy combined with rectification, the proposed approach is able to produce discretized outputs.
- *Label-free training*: The approach does not require label generation at all. Instead, the parameters of the machine learning model are optimized via the loss function.
- *Scalability*: The created machine learning model can be applied to problems of different sizes, i.e. there is no need to learn a new model for each problem size.
- *Parallelization*: With a trained machine learning model, the solution to multiple problem instances can be computed in parallel, i.e. it is possible to test and compare multiple alternatives without any further waiting time.
- *Flexibility*: The approach will be described and tested on a specific problem type. However, the system can easily be fitted to other problem types by changing the function describing the loss of the machine learning model.
- *Performance*: Overall, the presented approach returns results very quickly compared to e.g. classic optimization algorithms. This feature makes the approach especially suitable for applications where close-to-optimal solutions have to be obtained as fast as possible.

A rather sophisticated problem in the complexity class of NP-hard problems is the *hybrid flow shop scheduling problem*. We will validate the proposed approach with this problem. It comprises two sub-

problems: a sequencing problem and a binary assignment problem. Both subproblems are solved with label-free machine learning algorithms in this thesis, which are then combined to approximate the full hybrid flow shop problem. It is necessary to consider a few peculiarities of the problem, e.g. the interdependence of the sub-problems. Since the underlying mathematical model is more complex than that of many other NP-hard problems, the approach is also applicable to less complex problems. Originally, the hybrid flow shop is a production problem, but it applies to other domains as well.

1.2 Structure of the Thesis

In chapter 2, we will describe the hybrid flow shop problem in detail. We provide, both visually and formally, explanations of the problem, as well as an overview of complexity classes in general. In section 2.2.1 we will categorize the hybrid flowshop problem accordingly. Afterwards, in chapter 3 we will give a broad overview of gradient-based machine learning optimization, as well as deep learning (cf. section 3.4), which is the theoretical foundation of the proposed approach; the approach is then explained in detail in chapter 4, including a literature overview of machine learning for combinatorial problems and the hybrid flow shop problem (cf. section 4.1). The proposed approach is evaluated in chapter 5, where we compare the performance of the neural networks for the separate subproblems, i.e. permutation and assignments, as well as the combined approach with the Monte Carlo method and mixed-integer linear programming. Eventually we summarize the dissertation and provide further research suggestions in concerning machine learning for NP-hard combinatorial problems.

Chapter 2

Problem Description

To evaluate our approach to solving combinatorial problems with machine learning, we address the hybrid flow shop (HFS) scheduling problem as our primary example. It is one of many NP-hard combinatorial problems; however, it has some special features that make it stand out from other problems in this category: The HFS problem comprises two different combinatorial problems that are interdependent, more specifically a *sequencing problem* and an *assignment problem*.

2.1 The Hybrid Flow Shop Problem

Hybrid flow shops (also referred to as *flexible flow shops*, *multiprocessor flow shops* or *flexible flow lines*) are a generalization of both flow shops and parallel machine environments. The model was originally developed for production planning problems in manufacturing plants, where it is used to describe the production of different kinds of goods [68]. However, the HFS problem is also applicable to a wide variety of other domains; in section 2.1.4 we will analyze two completely different practical examples of the HFS to emphasize the versatile application possibilities.

2.1.1 Description

The structure of a production plant following the HFS scheme is shown in fig. 2.1. It consists of different stages $i = 1, \dots, I$ in series, each of which contains a subset $\mathcal{M}_i$ of M_i machines. The production of one item on the plant is called a *job*. Every job j has to pass each stage in sequence and, on each stage i, needs to be processed by exactly one machine $m \in \mathcal{M}_i$. On each stage i, there is an arbitrary number M_i of machines. For a job to be finished, the item must be processed by exactly one machine on each of the stages. Furthermore, the order of stages must be followed, i.e. the job is first processed by any machine on stage 1, then on stage 2 etc., before it is finally processed by any machine of stage I.

Following the definition of Ruiz et al. [68], to be called a hybrid flow shop, the problem configuration must include at least two stages, and at least one of the stages must possess more than one machine. Each job is assigned a *processing time* on each machine.

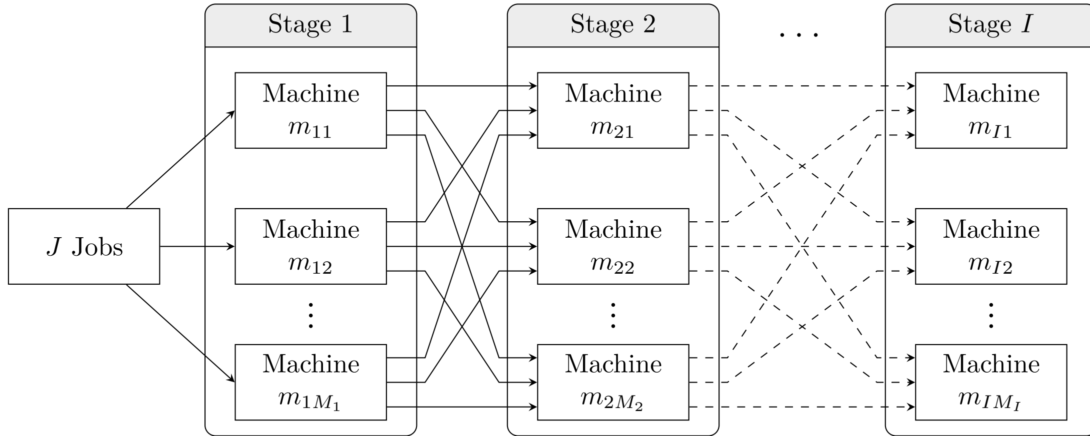


Figure 2.1: Illustration of the hybrid flow shop scheduling problem with jobs $j = 1 \dots, J$ and stages $i = 1, \dots, I$ with M_i machines on stage i

There are three types of machine configurations considered in the literature that determine the relation of the processing times of jobs on a stage: [26, p. 53]

- *identical* parallel machines imply identical processing times for all jobs on all machines on a stage,
- *uniform* parallel machines possess different "speeds" for the machines on a stage, i.e. some machines process *all* incoming jobs slower than other machines on the respective stage,
- *unrelated* parallel machines, which we will consider in the following, comprise different processing times both between jobs on the same machine and between different machines on the same stage.

Constraints

Several constraints must be satisfied by any *schedule*, i.e. any assignment of jobs to machines and the respective starting times:

(1) on each stage, each job is assigned to exactly one machine,

(2) each job must be processed completely on one stage before proceeding to the next,

(3) a machine must fully process one job before beginning to process another.

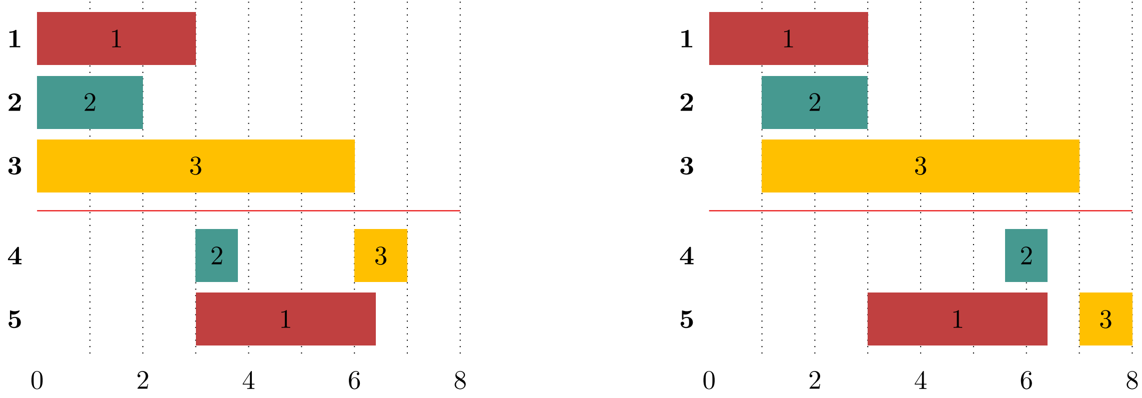
More specifically, we will focus on the *permutation* flow shop in the following. For this problem, the jobs must be processed in a specific *order* [68][56, p. 17], hence they do not *overtake* one another. What this rule implies for the definition of our constraints is often not well defined in the literature and thus can be interpreted differently: Given job j has higher priority than job k, we could only check the *starting times* of the jobs on the stages to avoid overtaking, i.e.

(4) no machine can begin processing job k before processing of job j has begun on the same stage,

which complies with the definition given in [68], focusing on the processing order on *each stage*. Following the definition of [56, p. 17], the permutation constraint is equivalent to the *first in first out* (FIFO) rule.

Hence, the *first* job in a given order must be (among the) *first* jobs to finishing processing and the *last* job in order must be (among the) *last* jobs to finish processing. To respect this definition, we have to add a further constraint considering the *finishing times* of a job on a stage:

(5) no machine can finish processing job k before processing of job j has finished on the same stage.



(a) Schedule considering constraint (4), respecting consecutive starting times of jobs on stages

(b) Schedule considering constraints (4) and (5), respecting consecutive starting and completion times of jobs on stages (FIFO rule)

Figure 2.2: Schedules for different interpretations of the permutation rule for a hybrid flow shop with $J = 3$ jobs, $I = 2$ stages and $M_1 = 3$, $M_2 = 2$ machines on the stages

The schedules for different interpretations of the permutation rule are visualized as *Gantt charts* in fig. 2.2, for a HFS with $J = 3$ jobs, $I = 2$ stages and $M_1 = 3$, $M_2 = 2$ machines on the stages; fig. 2.2a complies with constraints (1) to (4), whereas fig. 2.2b additionally includes constraint (5). The jobs are processed in the order $1 - 2 - 3$. In fig. 2.2a, by respecting the starting times, job 2 is not allowed to start before job 1 on stage 2, i.e. at $t = 3$, even though job 2 has already finished processing on stage 1 at $t = 2$ and machine 4 is unoccupied. Shifting the starting time of processing job 2 on stage 2 to $t = 2$ would lead to job 2 *overtaking* job 1. In the schedule of fig. 2.2b we can observe the impact of constraint (5) already on the first stage: Job 2 has to start later to not finish before job 1 on the stage, which simultaneously leads to job 3 being delayed to obey the starting time restriction. On stage 2, job 2 must not be processed before the completion time equals the completion time of job 1.

Note that the proposed job order for fig. 2.2 is not optimal, e.g. swapping the priorities of jobs 1 and 2 would avoid blocking the processing of job 2 on stage 2. The sub-optimal sequence was chosen for demonstration purposes.

We choose both of the permutation constraints to define the HFS problem in this thesis and thus, given constraints (1) - (5) and assuming no unnecessary gaps during the processing on any stage, the complete schedule is fully determined by the order of the jobs and the assignments of jobs to machines. An example of such a schedule is shown in fig. 2.3. However, the approach described in this thesis to determine the job order and machine assignments can easily be fitted to a different interpretation of the rules, by modifying the loss function of the underlying neural network model (cf. section 4.2).

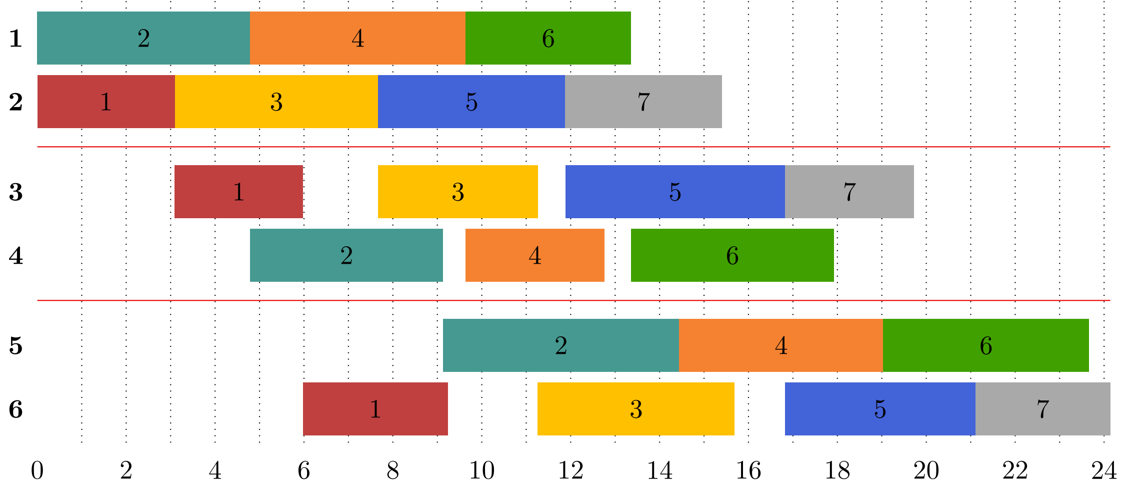


Figure 2.3: An optimal schedule for a given job order and given processing times with $J = 7$ jobs, $M = 6$ machines and $I = 3$ stages with two machines each

Objectives

For the permutation hybrid flow shop, the minimization problem consists of finding the optimal schedule – i.e. the optimal job order and assignments – with respect to a specific objective function which measures the total processing time. Most commonly, either the *makespan* or the *flowtime* is chosen as the objective. The *makespan* C_{max} is defined as the time at which the final job is finished on the last stage: Considering fig. 2.3, the makespan is the point in time when job 7 finishes on stage 3, i.e. $C_{\text{max}} = 24.14$. The *flowtime* C_{sum}, also referred to as *total completion time* in the literature, represents the sum of the final finishing times of *all* jobs; in fig. 2.3, since each job finishes on stage 3, we have to sum up the completion times of the jobs on that stage. Job 1 finishes on 9.241, job 2 on 14.44 and so on. Eventually, we add the completion time of job 7, which equals our makespan, thus we obtain $C_{\text{sum}} = 127.306$. [56, pp. 18, 19]

2.1.2 Notation

There are several variations of the HFS problem, each considering different restrictions or objectives. Following the proposed notation of flow shop problems in [46], the analyzed problem classes in this thesis are denoted with the triplet $\text{FH}_m((\text{RM}^k)_{k=1}^m \mid prmu \mid C_{\text{max}} \vee C_{\text{sum}})$: The first value describes the problem setup, where FH_m refers to the HFS with an arbitrary number of stages m and $(\text{RM}^k)_{k=1}^m$ refers to unrelated machines with any number of k machines on each stage; *prmu* denotes a permutation flow shop, i.e. the jobs are processed in a specific order on each stage; $C_{\text{max}} \vee C_{\text{sum}}$ denotes the objective functions, either the makespan C_{max} or the flowtime C_{sum}.[1]

[1]For the most part, we will choose other names for the variables considered in the general notation of [46].

To describe the problem formally, we will assume that the following parameters are fixed as part of the problem specification:

$$
\begin{aligned}
&J && \text{(number of jobs),}\\
&I && \text{(number of stages),}\\
&M && \text{(number of machines),}\\
&\mathcal{M}_i && \text{(machines on stage } i\text{),}\\
&M_i = |\mathcal{M}_i| && \text{(number of machines on stage } i\text{).}
\end{aligned}
$$

More accurately, $\mathcal{M}_i$ denotes the set of machine *indices* for the stage $i \in \{1, \ldots, I\}$, i.e. $m \in \mathcal{M}_i$ if and only if the m-th machine is part of stage i. Additionally, the indices

$$
\begin{aligned}
&j = 1, \ldots, J && \text{(job index),}\\
&i = 1, \ldots, I && \text{(stage index),}\\
&m = 1, \ldots, M && \text{(machine index),}\\
&k = 1, \ldots, J && \text{(position index)}
\end{aligned}
$$

indicated the affiliation of variables to the aforementioned fixed parameters, with k representing the position a specific job is assigned to in the sequence. The input variable of the problem is given by the matrix

$$
P \in \mathbb{R}_{+}^{J \times M} \qquad \text{(processing times)}
$$

such that P_{jm} denotes the time required for machine m to process job j on the stage i with $m \in \mathcal{M}_i$. Finally, a *schedule* is fully defined by the three matrices

$$
\begin{aligned}
&A \in \{0,1\}^{J \times M} && \text{(assignments),}\\
&S \in \mathbb{R}_{\geq 0}^{J \times I} && \text{(starting times),}\\
&O \in \{0,1\}^{J \times J} && \text{(permutation matrix).}
\end{aligned}
$$

To describe a sequence in vector form, we introduce

$$
\sigma \in \{1, J\}^J \qquad \text{(permutation vector).}
$$

Note that S_{ji} denotes the starting time of job j on the i-th *stage* (not the i-th machine). The assignment of jobs to machines is represented by the *assignment matrix* A: If job j is assigned to machine m, then $A_{jm} = 1$; otherwise, $A_{jm} = 0$. Since in the classical hybrid flow shop problem each job is assigned to exactly one machine on each stage, the requirement $A \in \mathcal{A}$ must hold, where

$$
\mathcal{A} = \left\{ A \in \{0,1\}^{J \times M} \,\middle|\, \sum_{m \in \mathcal{M}_i} A_{jm} = 1 \text{ for all } j \in \{1, \ldots, J\},\, i \in \{1, \ldots, I\} \right\} \tag{2.1.1}
$$

denotes the set of admissible assignment matrices. An extension of this set for the *assignment-relaxed* problem will be discussed in section 4.3.1.

The schedule illustrated in fig. 2.2b can be described with the following matrices:

$$P = \begin{pmatrix} 3.0 & 6.0 & 1.0 & 4.2 & 3.4 \\ 7.0 & 2.0 & 8.4 & 0.8 & 2.8 \\ 4.0 & 5.0 & 6.0 & 2.0 & 1.0 \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}, \qquad S = \begin{pmatrix} 0.0 & 3.0 \\ 1.0 & 5.6 \\ 1.0 & 7.0 \end{pmatrix}.$$

Note that P and A also apply to the schedule in fig. 2.2a, whereas the starting times S only apply to fig. 2.2b, implementing restrictions (4) and (5).

The permutation of a given job order is expressed with the *permutation matrix* O: If job j is assigned to *position* k in the sequence, then $O_{kj} = 1$; otherwise $O_{kj} = 0$. To ensure that each job j is assigned to exactly one position k, the requirement $O \in \mathcal{O}$ must hold, where

$$\mathcal{O} = \left\{ O \in \{0,1\}^{J \times J} \,\middle|\, \sum_{k=1}^{J} O_{kj} = 1 \text{ for all } j \in \{1, \ldots, J\}, \sum_{j=1}^{J} O_{kj} = 1 \text{ for all } k \in \{1, \ldots, J\} \right\} \tag{2.1.2}$$

denotes the set of admissible permutation matrices. The permuted matrix P_σ of processing times can be computed by simple matrix multiplication, i.e.

$$P_\sigma = O \cdot P. \tag{2.1.3}$$

Consequently, if O is the *identity matrix*, the original job order persists. [11, pp. 2–3] We can easily translate a given sequence σ into the respective permutation matrix O with

$$O(\sigma)_{k,j} = \begin{cases} 1 & \text{if } \sigma_k = j \\ 0 & \text{otherwise.} \end{cases} \tag{2.1.4}$$

For example, to reorder a processing times matrix P to follow the sequence $\sigma = (2,3,1)$, we multiply P with the respective permutation matrix O of size 3×3, i.e.

$$P_{(2,3,1)} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 3.0 & 6.0 & 1.0 & 4.2 & 3.4 \\ 7.0 & 2.0 & 8.4 & 0.8 & 2.8 \\ 4.0 & 5.0 & 6.0 & 2.0 & 1.0 \end{pmatrix} = \begin{pmatrix} 7.0 & 2.0 & 8.4 & 0.8 & 2.8 \\ 4.0 & 5.0 & 6.0 & 2.0 & 1.0 \\ 3.0 & 6.0 & 1.0 & 4.2 & 3.4 \end{pmatrix}. \tag{2.1.5}$$

Note that it is also necessary to reorder the assignment matrix A, if the assignments of the jobs to the machines was specified *before* the permutation was applied. However, we would typically determine a permutation first and seek for the corresponding machine assignments *after* permuting P: Different permutations lead to different optimal machine assignments; thus, defining the machine assignments without knowing about the final permutation of the jobs is not reasonable for most applications.

2.1.3 Mixed-Integer Linear Program Representation of the Assignment Problem

To formally describe the HFS problem, we can set up a mathematical model that considers the previously introduced constraints and variables. In particular, we will create a *mixed-integer linear programming* (MILP) model, which is suitable for constraint based problems like the HFS, comprising binary (e.g. the assignment matrix A) and continuous variables (e.g. the starting times S). Typically, MILP models are fed to a *solver* that uses techniques like *branch-and-bound* or a *simplex* approach to compute a solution. [26, p. 195] With MILP we can determine the *exact* solution; however, because of the computational effort, the exact solution can often only be computed for small problem instances. [26, p. 133] This also applies to the HFS problem (cf. section 2.2). Setting e.g. a time restriction for early stopping converts MILP into an *approximation algorithm*. We will use MILP in the course of this thesis to obtain exact solutions for small instances and approximated solutions for larger instances as a comparison for the results of the proposed novel approach.

In 1959, Wagner [78] formulated the pioneer MILP model for permutation flow shops. It has since become popular to describe scheduling problems in terms of integer programming [26, p. 133]. We will in the following express a MILP model that suits our previously defined constraints and objectives: More specifically, we will concentrate on the model formulation for the machine assignment problem. Descriptions of models including the optimization of the permutation can be reviewed in [53][33][26, p. 198] or [83, pp. 51–52].

To solve the problem with a linear programming approach, we need to define *decision variables*. They are populated and changed during the optimization process in compliance with the system constraints [26, p. 128], which we will formally define later. With the decision variables

$$S_{j,i} \qquad \text{(starting time of job } j \text{ on stage } i\text{)},$$
$$C_{j,i} \qquad \text{(completion time of job } j \text{ on stage } i\text{)},$$

we save the starting times S and completion times C for the jobs j on the stages i. The objective values are denoted with

$$C_{\max} \qquad \text{(makespan: completion time of the last job } J \text{ on the last stage } I\text{)},$$
$$C_{\text{sum}} \qquad \text{(flowtime: sum of all jobs' completion times on stage } I\text{)},$$

respectively. Since we are dealing with an assignment problem, i.e. a *binary* problem, we need to define our binary assignment matrix A with

$$A_{j,m} \qquad = \begin{cases} 1 & \text{if job } j \text{ is assigned to machine } m \\ 0 & \text{otherwise.} \end{cases}$$

The last variable we introduce is

$$\mathsf{M} \qquad \text{(sufficiently large number)},$$

which serves as a *penalty term* and disables linear constraints on conditions where they do not apply. This approach is referred to as the *big-M method*. [41, pp. 129–130] Our objectives are

$$\min \quad C_{\max}, \tag{2.1.6}$$

$$\min \quad C_{sum}. \tag{2.1.7}$$

For the constraint definition it is essential to set the correct boundaries for the linear program to work. Otherwise, we might obtain infeasible solutions. While setting up the constraints in the following, we will refer to the respective verbally formulated constraints from section 2.1.1. The first equation

$$\sum_{m=1}^{M_i} A_{j,m} = 1 \qquad \forall\, j \in \{1, \dots, J\},\ i \in \{1, \dots, I\} \tag{2.1.8}$$

accounts for valid assignments and represents our binary constraint: For each job, only one machine per stage can be set to 1, thus it refers to constraint (1). The next inequalities deal with the *continuous* decision variables. With

$$C_{j,i} \geq S_{j,i} + P_{j,m} - \mathsf{M} \cdot (1 - A_{j,m}) \qquad \forall\, j \in \{1, \dots, J\},\ i \in \{1, \dots, I\},\ m \in \mathcal{M}_i, \tag{2.1.9}$$

we compute the completion time of the current job j on the current stage i by adding the current processing time $P_{j,m}$ to the current starting time $S_{j,i}$. There is also the first appearance of the big-M number M: We iterate through all machines $\mathcal{M}_i$ for a job j on a stage i, however, only one of the machines is occupied. Thus, we want the completion time to increase only if we currently view an occupied machine, i.e. if $A_{j,m} = 1$. In these cases, the big number is multiplied with 0. If the current machine is not occupied, i.e. $A_{j,m} = 0$, we subtract the large number M and hence, the completion time is not increased. The inequality

$$S_{j,i+1} \geq C_{j,i} \qquad \forall\, j \in \{1, \dots, J\},\ i \in \{1, \dots, I-1\} \tag{2.1.10}$$

accounts for rule (2): A job j can only start on the next stage $i+1$ after it was completed on the current stage i. Rule (3) is covered with

$$S_{j,i} \geq C_{k,i} - \mathsf{M} \cdot (2 - A_{j,m} - A_{k,m}) \qquad \forall\, j \in \{2, \dots, J\},\ i \in \{1, \dots, I\},\ m \in \mathcal{M}_i,\ k \in \{1, \dots, I-1\}, \tag{2.1.11}$$

where we increase the starting time of the current job j to the completion time of any prior job on the stage k, if the current machine m is occupied *and* the previous job k was assigned to the current machine. Thus, the constraint ensures that only one job at a time is processed on a machine. Next, we analyze the inequalities that deal with the prevention of jobs overtaking other jobs with higher priority. Rule (4) considering the starting times is satisfied with

$$S_{j+1,i} \geq C_{j,i} - P_{j,m} - \mathsf{M} \cdot (1 - A_{j,m}) \qquad \forall\, j \in \{1, \dots, J\},\ i \in \{1, \dots, I\},\ m \in \mathcal{M}_i. \tag{2.1.12}$$

The starting time of the next job $j+1$ on the current stage i is greater or equal to the completion time of the current job j minus the current processing time, only if the current machine is occupied. The

completion time rule (5) is set up as

$$C_{j+1,i} \geq C_{j,i} \qquad \forall\, j \in \{1,\dots,J\},\ i \in \{1,\dots,I\}, \tag{2.1.13}$$

where we ensure that the next job $j+1$ on the stage i cannot finish earlier than the current job j. Finally, we define the inequalities for the objective functions with

$$C_{max} \geq C_{j,I} \qquad \forall\, j \in \{1,\dots,J\}, \tag{2.1.14}$$

$$C_{sum} \geq \sum_{j=1}^{J} C_{j,I}. \tag{2.1.15}$$

The last equations ensure that the values of the respective variables stay in the right domains, i.e. 0 or 1 for the binary variables of eq. (2.1.8), and values greater or equal to 0 for the continuous variables from eq. (2.1.9) to eq. (2.1.13):

$$A_{j,m} \in \{0,1\} \qquad \forall\, j \in \{1,\dots,J\},\ m \in \{1,\dots,M\}, \tag{2.1.16}$$

$$S_{j,i} \geq 0 \qquad \forall\, j \in \{1,\dots,J\},\ i \in \{1,\dots,I\}, \tag{2.1.17}$$

$$C_{j,i} \geq 0 \qquad \forall\, j \in \{1,\dots,J\},\ i \in \{1,\dots,I\}. \tag{2.1.18}$$

These variables and constraints can be conveyed to a solver to minimize the objective function. Typically, a feasible starting solution is determined by the solver before applying the optimization algorithm. With most solvers it is possible to provide a custom starting solution to the problem; if this starting solution is close enough to the optimum, this might support the solver in finding the best solution faster. [26, p. 242]

2.1.4 Applications

Besides manufacturing applications, the HFS can also be applied to other domains. We will in the following analyze two other applications fields: ship scheduling and server scheduling.

Ship Scheduling

In [75] the HFS is applied to the allocation of ships to berths in a harbor. The ships maneuvering through the harbor are equivalent to the jobs of the standard manufacturing example. There are three stages that the ships have to pass:

1. an *entering zone* with parallel accesses, i.e. parallel *machines*,
2. a *berth zone* where the ships are assigned to one of the shipyards to be unloaded; since the berths may be equipped differently, they are viewed as *unrelated parallel machines*,
3. a *leaving stage*, structured similarly to the entering zone.

The processing times are determined by safety distances between ships that have to be respected, berthing times while unloading and travel duration on the waterway.

The special features of this interpretation of the HFS problem can be expressed with a MILP model, which is fitted to the requirements of the ship scheduling example. The respective model presented in [75] is more complex than the standard HFS problem. Nevertheless, the proposed approach of solving MILP problems with machine learning could be modified in a way to fit the problem introduced in the paper.

Server Scheduling

The next example is based on [2]: A set of clients sends requests via the internet that have to be processed by a multi-stage multi-server setup (see fig. 2.4) resembling a hybrid flow shop. A request has to be processed on one of the servers of each stage. After the request was processed on the final stage I, the response is sent back to the client. It is the goal to reduce the average response time by scheduling the incoming requests efficiently.

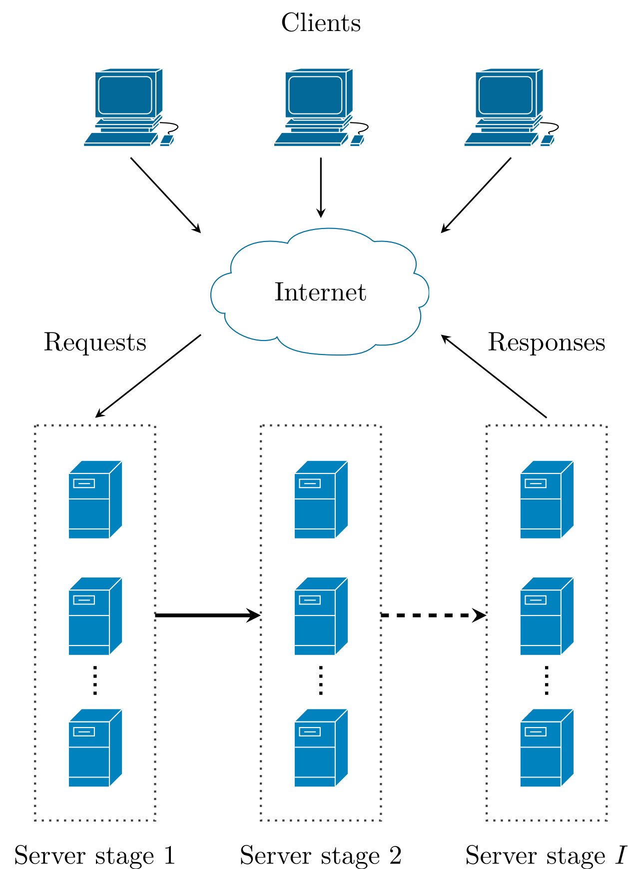


Figure 2.4: Multi-stage multi-server internet architecture, cf. [2]

In this example it becomes obvious that it is sometimes better to accept a good, but not optimal solution to the HFS problem, than to wait for a better or exact solution. The clients are expecting a very quick response: Depending on the number of requests coming in, it is beneficial to use an algorithm that can determine the assignments of multiple (maybe hundreds) jobs in parallel. When processing the requests batch-wise, the algorithm should also account for varying sequence sizes.

Moreover, an algorithm that can be scaled down could be beneficial in the case of server failure: In this case, the algorithm could dynamically adapt to the new node configuration to ensure further processing of the jobs.

2.2 NP Complexity Classes

Computational problems can be divided into complexity classes that state how "difficult" the problems are to solve and how easy it is to determine if a given solution to a problem is the optimal one. [30, pp. 44–45] Referring to the HFS problem, each instance comprises several feasible solutions that lie within the previously described constraints of section 2.1.3. Computing the objective value (e.g. makespan) of a solution is easy, whereas determining if the given solution is optimal requires more elaborate procedures.

There are computational problems for which we can define algorithms to gain the exact solution, e.g. with linear programming. However, some of these problems are *practically unsolvable*: The cost or resources to solve them, i.e. to obtain the best solution, increases with the size of the problem to an extent that exceeds any reasonable limitation, for instance if a solution required years of computation time or an excessive amount of memory. We consider algorithms to be *practically solvable* if they require at most *polynomial* effort. An example is matrix multiplication for square matrices with $O(n^3)$: The algorithm takes $\sim n^3$ operations[2] to solve for an $n \times n$ matrix. These problems belong to the *polynomial time solvable* (P) complexity class. Algorithms growing *exponentially* in computation time, e.g. with $O(2^n)$, are not in P. [36, p. 726]

Another complexity class comprises the *nondeterministic polynomial time solvable* (NP) problems. The additional term "nondeterministic" in the class name reveals that we can only solve these problems in polynomial time using a nondeterministic algorithm. Simply stated, *deterministic* algorithms base their solution steps on predictable decisions: There is only one path to determine the solution. Contrary, *nondeterministic* algorithms comprise several possible paths to obtain the solution. Which branch on the path is taken depends on nondeterministic decisions. Thus, for a classical (stochastic) implementation of a nondeterministic algorithm, the same input may return different results for each run.[3] Hence, we *may* find a solution in polynomial time for a problem in NP, but this is not guaranteed. In contrast, problems in P can be solved in polynomial time using a deterministic algorithm. An interesting property of NP is that, although we may not find a solution in polynomial time, we can *check* in polynomial time if a given solution is the correct one. [36, pp. 728–730]

Unlike what the acronym might suggest, P is *not* disjoint from NP: Each problem in P can also be solved with a non-deterministic algorithm by replacing the arbitrary selections with a fixed decision, thus $\mathsf{P} \subseteq \mathsf{NP}$. [36, p. 729] Interestingly, it has not been proven yet if it is indeed not possible to solve all problems in NP in polynomial time, i.e. whether there is a difference between P and NP. This leads to the *P versus NP question*, which can, in words, be described as: *Are problems whose answer is easily verifiable also easily solved?* And thus, is it true that $\mathsf{P} = \mathsf{NP}$? [30, p. 46] In fact, the *P vs. NP question* is one of the *Millennium Prize Problems* that were selected by the Clay Mathematics Institute in 2000. They comprise

[2] The *O-notation* provides an *upper bound* for the growth rate of a function with respect to its input size n. [15, p. 49]

[3] In order not to go beyond the scope, the interested reader is referred to [36] for more information on deterministic and non-deterministic algorithms.

mathematical problems of high interest that have yet to be solved. A price money of 1 million dollars is offered to anyone who can solve one of the problems. [14]

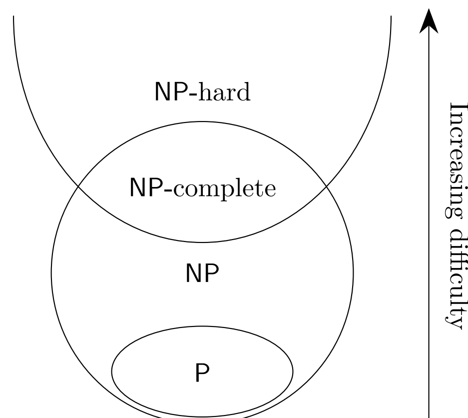


Figure 2.5: Hierarchical display of complexity classes and their solving difficulty, cf. [45], assuming that $\mathsf{P} \neq \mathsf{NP}$

Figure 2.5 shows how the complexity class P is categorized as being contained in NP. The difficulty of solving problems in the respective classes increases from bottom to top. Thus, problems in NP are possibly harder to solve than problems in P. In general, we can specify that a problem A is harder than a problem B, if we can *reduce* B to A in polynomial time, i.e. we can express problem B in terms of problem A.

The *Cook-Levin theorem* states that any problem in NP can be reduced to the *boolean satisfiability* (SAT) problem.[4] All problems that are as difficult as the SAT problem are part of a new subset, namely the NP-complete (NPC) complexity class. Thus, NPC is representative for the NP complexity class and defines an "upper bound" for the difficulty of problems in NP. This is also reflected in fig. 2.5: NPC problems are in NP, thus $\mathsf{NPC} \subseteq \mathsf{NP}$, and they are situated at the top of the NP class as problems in NPC are the most difficult problems in NP. [36, 732 ff.]

The last complexity class to discuss here is NP-hardness. Problems in this class are at least as hard as problems in NPC, i.e. the SAT problem. Thus, all NP-complete problems are also NP-hard and we can express any problem in NP in terms of any NP-hard problem. Not all NP-hard problems are in NP, i.e. the correctness of a given solution cannot be checked in polynomial time. Therefore, the NP-hard set in fig. 2.5 lies partly outside and above of NP, as it makes the problems increasingly difficult. The intersection between the NP-hard and the NP class is the area of NP-complete problems. [30, p. 68]

Knowing the complexity class of e.g. a scheduling problem has practical benefits: It supports the assessment of requirements for algorithms to solve the respective problem. Depending on the complexity, we can expect to gain an exact solution (problems in P) or an approximate solution (problems in NP or NP-hard problems). According to this classification, we can choose or create the solution algorithm. Moreover, the complexity class provides information about how easy it is to check if the results of an algorithm are correct, which is possible for NP-complete or easier problems. If the problem is NP-hard and *not* in NPC, we might not be able to determine the correctness of our results, which makes approaching the problem ultimately harder. This characteristic of a problem is also to be considered when modeling a solution algorithm.

[4] Again, the interested reader is referred to [36] to learn more about the SAT problem.

2.2.1 Complexity of the Hybrid Flow Shop Problem

In [34], the author discusses that the HFS problem is NP-hard, even with $I = 2$ stages and one of the stages comprising only one machine ($M_1 = 1$ or $M_2 = 1$). This result is also supported by the HFS problem overview in [68].

As we have analyzed in section 2.2, NP-hardness is necessary, but not sufficient for NP-completeness. To conclude that the HFS problem is NP-complete would be beneficial for solving the problem, since we could easily determine if a given solution is optimal. Although some special cases of flow shops have been proven to be NP-complete [68][82][64], this is not true for the HFS problem: There has not yet been found an algorithm to determine if an HFS solution is optimal in polynomial time.

The overall number of possible solution for a HFS is

$$J! \left(\prod_{i=1}^{I} M_i \right)^J , \tag{2.2.1}$$

where the product of machines on the stages determines the assignment possibilities for one job on all stages, the J-th power is the number of assignment possibilities for one job permutation over all jobs and stages, and $J!$ denotes the number of possible permutations. [32] Dropping the factorial term $J!$ for all possible permutations would compute the number of possible assignments for a given sequence.

Table 2.1: Number of possible solutions for the reduced assignment problem and the standard HFS problem with job permutations

J	Machine config.	I	Possibilities assignments	Possibilities *prmu*
6	[2,2,2]	3	262 144	188 743 680
6	[2,2,2,2]	4	16 777 216	12 079 595 520
6	[3,3,3]	3	387 420 489	278 942 752 080
10	[2,2,2]	3	1 073 741 824	3 896 394 330 931 200
10	[3,3,3]	3	205 891 132 094 649	$7.471\,377\,401\,450\,6 \cdot 10^{20}$
20	[3,3,3]	3	$4.239\,115\,827\,521\,6 \cdot 10^{28}$	$1.031\,335\,340\,967\,0 \cdot 10^{47}$
50	[3,3,3]	3	$3.699\,884\,850\,351\,3 \cdot 10^{71}$	$1.125\,286\,426\,741\,9 \cdot 10^{136}$

Table 2.1 shows the number of possibilities that arises from different amounts of jobs and various machine configurations for the reduced machine assignment problem and the standard HFS problem including job permutations. We can observe the exponential growth of the problem with increasing J and complexity of the machine configurations for the assignment problem; considering the job permutations, the solution space enlarges rapidly for increasing job numbers.

As an example, if we assume that the computation of one possibility takes only one millisecond, i.e. 0.001 seconds, it would still take over 6500 years[5] to compute all solutions of the assignment problem for $J = 10$ and a machine configuration of $[3, 3, 3]$ – even though this is a comparatively small problem. Including the

[5] $205\,891\,132\,094\,649 \cdot 0.001/(60 \cdot 60 \cdot 24 \cdot 365) \approx 6528.765$

job permutations, it would take approximately $2.37 \cdot 10^{10}$ years to iterate through all possibilities.[6] Thus, to compute the exact solution with *brute-force* is not practicable.

To summarize, as the HFS problem can (currently) not be solved efficiently in polynomial time, it is advised to apply approximation algorithms to compute a solution; exact algorithms might not return a solution in a reasonable time. Along with this, we cannot await that a proposed approximation algorithm returns optimal solutions. Nevertheless, the trick is to find an algorithm that takes as little time as possible to deliver the best possible result.

[6] As a comparison: The sun is $\sim 4.610 \cdot 10^9$ years old.

Chapter 3

Machine Learning

3.1 Definition

Machine learning (ML) and its more sophisticated variation *deep learning* (DL) are the main concepts we will be using to approach the NP-hard combinatorial hybrid flow shop problem in this thesis. Figure 3.1 illustrates how machine learning can be classified – as a subset of *artificial intelligence* (AI). Thus, we will start with a definition of AI, before moving on to machine learning.

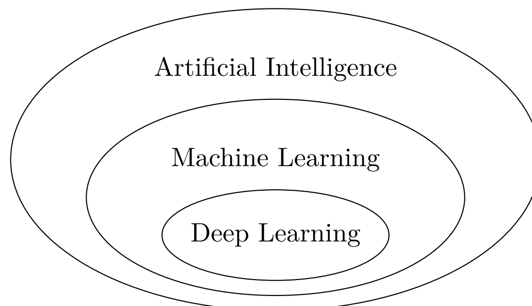


Figure 3.1: Classification of artificial intelligence, machine learning and deep learning, cf. [13, p. 22]

3.1.1 Artificial Intelligence

As early as 1983, Rich gave the following definition:

> Artificial intelligence is the study of how to make computers do things at which, at the moment, people are better. [62]

This definition is still valid today and will be valid for many years to come [22, p. 2]. These "things" at which people are better than computers, for example, refer to *domain knowledge*. In the industrial context, the decisions of domain experts are often driven by their experience. Nowadays, AI is successfully used in those scenarios where it supports experts and sometimes even exceeds their knowledge. [8] Further, Rich

states that, to create artificial intelligence, scientists "[...] need to produce programs that imitate human performance in a wide variety of 'intelligent' tasks" [63]. One mean – but not the only one – to achieve this is machine learning.

3.1.2 Machine Learning

Figure 3.2 illustrates what distinguishes traditional programming from ML. In traditional programming, we have a clear vision of the outcome of a program. When we create a program that sorts numbers from a user input, we use certain data (user input of numbers) and perform certain rules (a sorting algorithm) on this data to get a desired answer (sorted values). However, there exist problems we do not have an algorithm for to obtain the answer. For instance, we want to label e-mails as "spam" or "no spam": We cannot *explicitly* express this task as a mathematical function, but there is an *implicit* one that approaches our desired output sufficiently.

We can compensate the lack of knowledge with a large quantity of data – in this case past e-mails a human could identify as spam. Hence, the labeling of these e-mails is a task, "at which, at the moment, people are better" [62]. ML can identify *patterns* in the data, expressed as mathematical functions. Maybe the human identifying the e-mails was not exactly sure in some cases either. Often, a good *approximation* is enough for a task. [3, pp. 1–3]

The input information to ML differs from traditional programming. Still, we have access to a huge amount of data describing our problem, but contrary to ordinary algorithms, we already know the answer to it. What we do not know is *how* exactly these answers were generated. It is the task of ML to develop a set of rules that leads to the answer. [52]

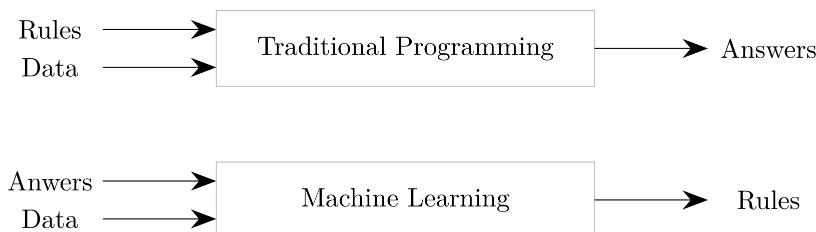


Figure 3.2: Comparison between traditional programming and machine learning [52]

Eventually, we want to use the generated set of rules on *unseen* data, i.e. not yet labeled data. In our spam e-mail example, we would like to categorize new incoming e-mails with our rules. Therefore, it is crucial that they are not only working well on our labeled data, but also on any unseen data that has the same structure as our labeled data: The rules must have the ability to *generalize*. *Generalization* is a key concept of machine learning and specifies, at least in part, the quality of an ML model. [23, p. 6]

To "simulate" unseen data, we divide our labeled data into *training* and *test* data: We only use the training data for learning; after we have set up our ML model, we use the test data to evaluate our results. We further elaborate on this topic in section 3.3.4.

Hence, we can paraphrase ML algorithms as methods to generate mathematical models from observed input data with certain features (attributes) to predict an output attribute of unseen data with the same

structure as the input data. Eventually, the discrepancy between the model output and the observed data has to be minimized, which can be interpreted as an "optimization problem". [2, p. 1] There are several ML techniques that use this approach. The algorithms developed in this thesis rely on *artificial neural networks* (ANNs). Thus, we will focus on this method in the following sections.

Broadly, the data and its features processed by an ANN are represented as linear combinations. A set of linear and non-linear transformations is performed on these matrices to map the input to the output. [35, p. 389][2, p. 41] We will elaborate on these concepts in the following.

3.1.3 Deep Learning

Referring to fig. 3.1, deep learning is a subfield of machine learning [29], and thus follows the same goal of generating a mathematical model that describes a relationship between input and output data. Deep learning models are always neural networks: Contrary to standard ML models, their "building blocks", the *layers* (c.f. section 3.2.1), are increasingly meaningful *representations* of the input data structure. The "deep" in deep learning does not refer to the size of the DL model or that a deeper understanding is gained; it represents the concept of the layer-wise representation of the data, which might involve a more complex architecture than usual. [13, pp. 27–29] We will further discuss the principals of deep learning and the mechanisms behind common DL layers in section 3.4.

3.2 Structure of Neural Networks

The mechanisms of neurons in the human brain inspired the concept of artificial neural networks. From this initial point, the concept has developed considerably. Merely, the terminology is reminiscent of the old concept, like the term *neural network* (NN) itself. [35, p. 389][9, p. 226] We will use the terms *artificial neural networks* and *neural networks* interchangeably.

Neural networks can be used for classification as well as regression tasks. In supervised classification, the aim is to assign input data to given categories or classes. The labeling of e-mails as spam is therefore a classification task with two output classes *yes* (spam) or *no* (no spam). Regression problems forecast a continuous numerical value. [3, p. 10] For example, via given chemical and mechanical properties of a material we want to predict its tensile strength.

The mathematical function produced by an NN comprises "sums of nonlinearly transformed linear models" [35, p. 18]. In fig. 3.3 we see an example of a neural network structure. This structure is also called a *multi-layer perceptron* or *feedforward neural network*.

3.2.1 Feedforward Neural Networks

The overall structure of NNs comprises connected *nodes* which are assigned to consecutive *layers*. There is always an input layer, where the number of nodes matches the dimension of the input examples, and an output layer, where the number of nodes matches the dimension of the desired outputs. Between the input and output layer, there is an arbitrary number of hidden layers. The amount of nodes in hidden layers can be chosen freely. In their basic form, the *dense layers*, each node of a layer is connected to each node

of the previous layer; each connection has a *weight* w and each node has a bias term b. These parameters are already given, either by initialization or after some learning cycles (cf. section 3.3). With forward propagation, we want to determine the outputs $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_N)$ an NN assigns to the respective input data points $x = (x_1, \ldots, x_N)$. To achieve this, we compute the outputs of the (non-)linear transformations the NN performs on the single inputs x_n. During this process, we memorize the results of the single transformations.

Figure 3.3 shows a neural network with two hidden layers: Each layer has an arbitrary number of nodes D (input layer), M, E (hidden layers) and K (output layer). The figure shall serve as a reference for the notation of the following equations. We use [9, p. 227]

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_{j0}^{(1)} \tag{3.2.1}$$

to compute the result of a linear subset, where a_j is called the *activation* of the jth node of a layer. The activations sum up the product of the outputs of all nodes from the previous layer that are connected to the current node[1] and the weights w_{ji} on the connections. The subscript j refers to the current layer's nodes and the subscript i refers to previous layer's nodes. For instance, in the first hidden layer of fig. 3.3 there are M nodes and thus the activation vector is of size M with the elements $\boldsymbol{a} = a_1, \ldots, a_M$. The weights on the connections to the first layer with the superscript (1) can be summarized in a vector $\boldsymbol{w}^{(1)} = w_{11}^{(1)}, \ldots, w_{MD}^{(1)}$ with $M \cdot D$ entries. Since we look at the first layer, the output of the previous layer is simply our non-transformed input to the NN. Furthermore, we add a *bias* term b_{j0}. The bias is the *intercept* of the linear transformation and broadens the solution space to capture any "invariant part of the prediction" [1, p. 6]. We keep a 0 in the subscript of the bias to emphasize that it is not connected any previous layer's nodes i. Nevertheless, like the weight w_{ji} with two subscripts, it is a *trainable parameter* that has to be adjusted to map the input to the output (cf. section 3.3). So far, we have generated a purely linear function. To also model non-linear dependencies, we transform the activations with a non-linear, continuous and differentiable *activation function* h to compute the *output* of a node with [9, p. 227]

$$z_j = h(a_j). \tag{3.2.2}$$

Note that it is not mandatory to perform an activation function to an activation. There are cases when a linear behavior is desired (for an example see section 3.2.2). For any other layer, we substitute x_i with the output values z_i of the previous layer.

To process the links between the remaining layers, we adapt eq. (3.2.1) and eq. (3.2.2) to get

$$z_j^{(2)} = h\left(\sum_{i=1}^{E} w_{ji}^{(2)} x_i + b_{j0}^{(2)}\right) \tag{3.2.3}$$

for the outputs of the second hidden layer and

$$y_j = h\left(\sum_{i=1}^{K} w_{ji}^{(out)} x_i + b_{j0}^{(out)}\right) \tag{3.2.4}$$

[1] For a dense layer this means that *all* of the previous layer's nodes are connected to each node of the following layer.

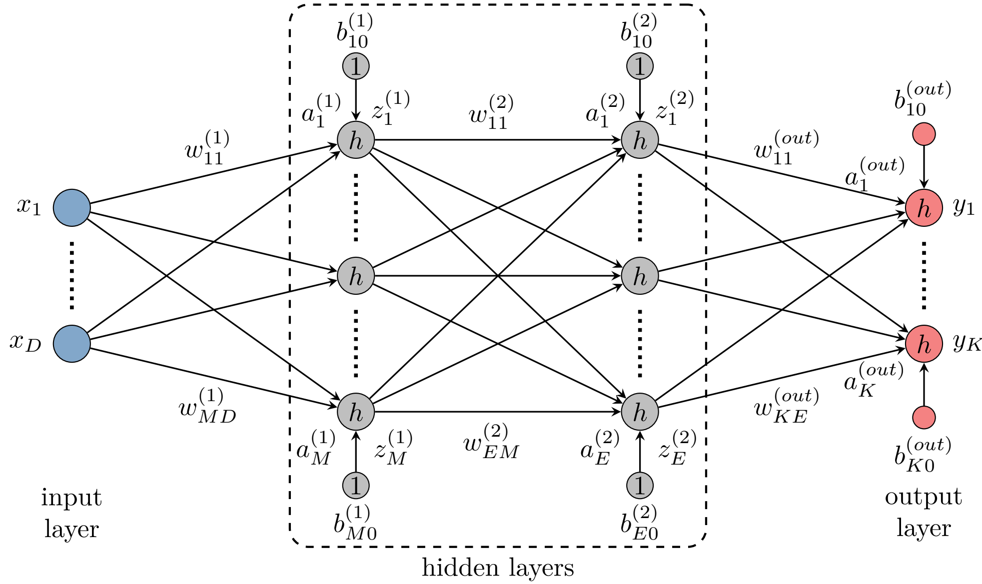


Figure 3.3: Simple Feed-Forward Neural Network, cf. [9, p. 228]

to determine the final outputs of our NN in fig. 3.3. With these equations we can compute the activations and outputs of *all* nodes. We can substitute the outputs z with the according functions of the previous layer's nodes, leading to

$$y_k(x) = h\left(\sum_{v=1}^{M} w_{kv}^{(out)} h\left(\sum_{j=1}^{E} w_{vj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_{j0}^{(1)}\right) + b_{l0}^{(2)}\right) + b_{k0}^{(out)}\right), \tag{3.2.5}$$

to directly express the outputs y_k as a function depending on only the inputs $\mathbf{x}$, with given weights, biases and activation functions for the three layers in fig. 3.3. We can further simplifly the notation by incorporating the bias terms into the weight terms. The bias can be viewed as an additional node attached to each node on the hidden and output layers, with a trainable weight and an input value fixed at 1 (cf. fig. 3.3). This ultimately results in

$$y_k(x) = h\left(\sum_{j=0}^{M} w_{ji}^{(out)} h\left(\sum_{j=0}^{E} w_{ji}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)\right), \tag{3.2.6}$$

where the sum variables start from $j = 0$ instead of $j = 1$ to include the bias values b_{j0}. [9, pp. 228–229]

We have now reached the target of feedforward propagation to compute all activations, node outputs and thus the final output of the neural network. In the following sections we will make use of this process and discuss common neural network model architectures.

3.2.2 Regression

The goal of regression is to identify an output consisting of numerical continuous values (dependent variables) as a function of the input (independent variables) [3, p. 82]. Neural networks modeling a *scalar*

regression problem possess one node in the output layer [35]. Thus, the mapping $\mathbb{R}^N \mapsto \mathbb{R}$ applies to any input x of size N. To illustrate a scalar regression model, the NN in fig. 3.3 would only have the output node y_1; however, regression is also possible for the non-scalar case, i.e. $\mathbb{R}^N \mapsto \mathbb{R}^M$. Additionally, there is typically no activation function h in the last layer of the regression NN, i.e. $y_k = a_k$ [9, p. 228].

As an example, we construct a neural network for scalar regression that takes as an input a real number $x_n \in x$. We define the output labels as $y = bx^3 - 3bx$. Thus, for a single input value, a single output value is returned. After training the NN, the observed values $\hat{y}$ should be close or equal to $f(x) = x^3 - 3x$ for any real input number x_n. To simplify the example, we observe the transformation of values in the interval $[-2, 2] := \{x \in \mathbb{R} \mid -2 \leq x \leq 2\}$ with a step size[2] of 0.1. Figure 3.4 is a plot of the x-values and the corresponding y-values of the function $f(x)$. The points are displayed in different colors to enhance the traceability of each dot in the following graphs. For instance, the input value $x_n = -2$ is tinted dark purple whereas the value $x_n = 2$ uses a bright yellow.

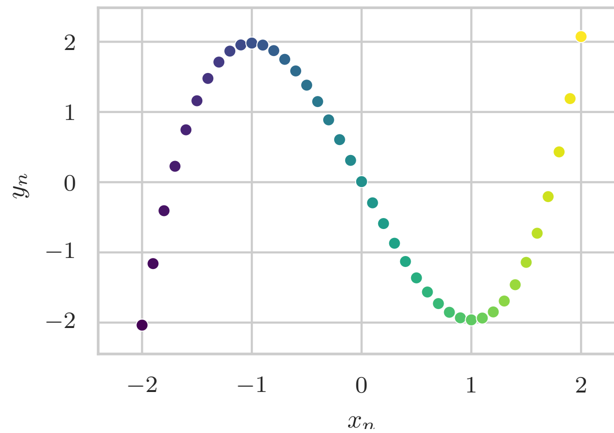


Figure 3.4: Plot of the non-linear function $y = x^3 - 3x$

We create a fairly simple NN to model the function, with one hidden layer with three nodes. The input and output layers have only one node each. Since $f(x)$ is non-linear, we can only map this function using a non-linear activation function. In this particular case, we take the *tanh* activation function to perform the tangens hyperbolicus function to the activations of the hidden layer. Thus, the output for a single node of the first transformation is $z_m = \tanh(xw_m^{(1)} + b_{m0})$.

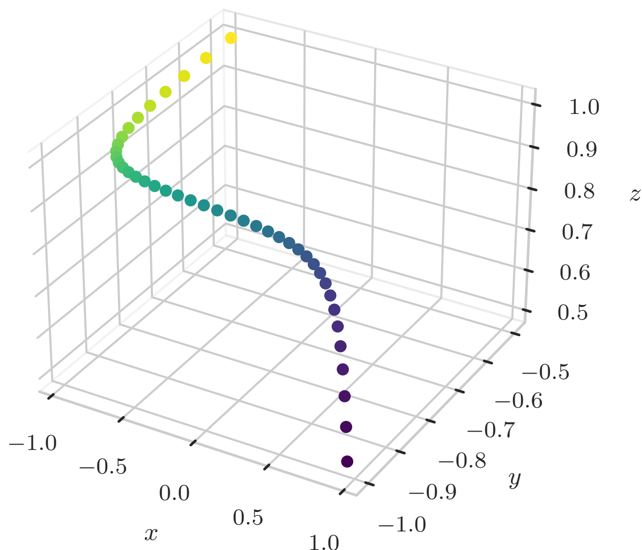
As discussed before, for the weights between hidden and output layer, we use the identity as activation. We compute the output values with $\hat{y} = \sum_{m=0}^{3}(z_m w_{km}^{(out)})$. For the general form

$$\hat{y}(x) = \sum_{m=0}^{3}(\tanh(xw_m^{(1)} + b_{m0})w_m^{(out)}) \tag{3.2.7}$$

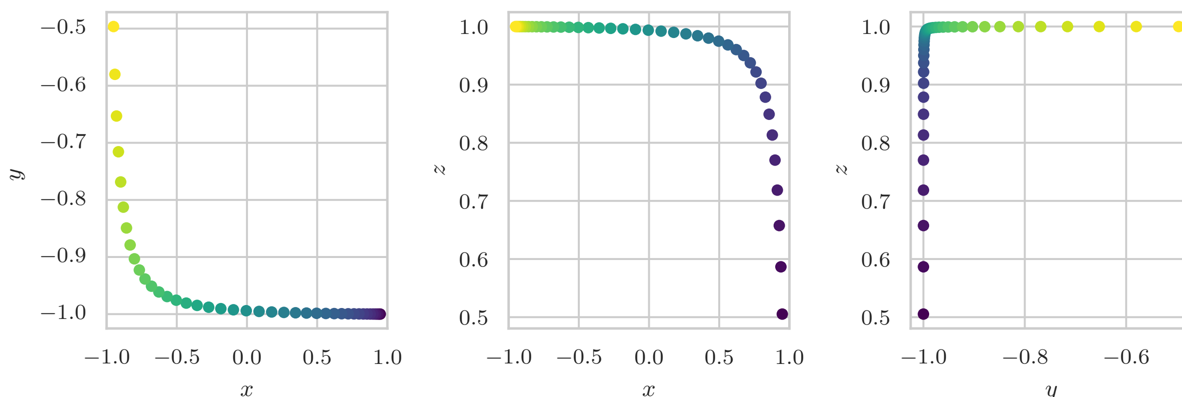
we substitute z_m in the equation.

Due to the three nodes of the hidden layer and the non-linear activation function, the input value x_n is mapped to a point in 3-dimensional space. The results of this transformation can be viewed in fig. 3.5a

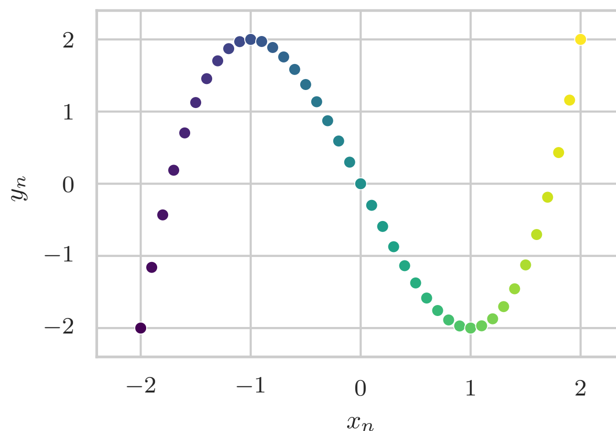
[2]For training, we use the numbers of the same interval, but with a step size of 0.01 to increase the training sample. We do not create a test data set before training, as this example is for demonstration only. For further information on training and test data sets see section 3.3.4.



(a) Mapping of the input to $\mathbb{R}^3$ produced by the hidden layer



(b) Output of the hidden layer from different perspectives



(c) Output of the final layer with one node

Figure 3.5: Layer outputs of a scalar regression problem with one hidden layer comprising three nodes

and fig. 3.5b. The tanh activation function accounts for the maximum values -1 and 1, as these are the limits of the tangens hyperbolicus function. When we compare the plot in fig. 3.5a with the plot of $f(x)$ in fig. 3.4, we notice that the x-values in the extrema, i.e. $x_n = -1$ (dark blue) for the maximum and $x_n = 1$ (light green) for the minimum, are in the area of maximum curvature in the 3D-mapping of the points.

The three values from the hidden layer are eventually combined and transformed to the output y. The deviation of the output function from the original function is very small (MSE ≈ 0.015) and cannot be detected in the corresponding plot in fig. 3.5c. Besides the shown solution, there are others: Since the weights are initialized randomly, we might get a different solution when we reset our model and train again. For example, the weight values can be interchanged, as the sums of equation 3.2.7 are commutative.

3.2.3 Classification

In contrast to regression, where the aim is to predict a real number, classification aims to assign an input to a *class*. To illustrate the concept of classification, we use the example in fig. 3.6, which consists of 2000 points. Each point $x_n \in x$ consists of the two coordinates (x_1, x_2), which are equally distributed on the interval $[0,1]^2 \coloneqq \{x_1, x_2 \in \mathbb{R} \mid 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1\}$ and divided into $G = 3$ groups. The data points are colored depending on which group they belong to: The points in and on the center circle belong to group 0 (blue), the points left from the circle belong to group 1 (yellow) and the remaining points on the right are assigned to group 2 (red).

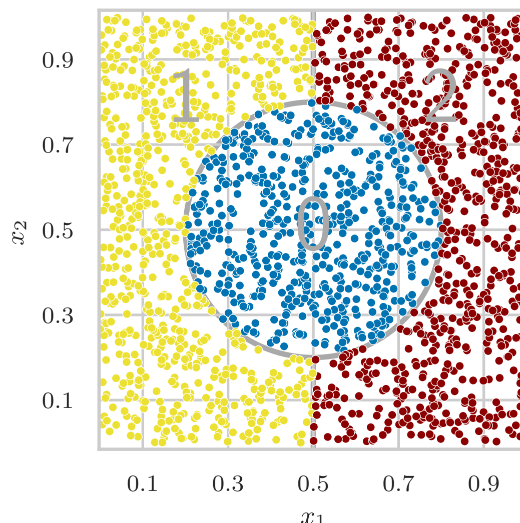


Figure 3.6: Classification example with three groups

Our objective is to build a neural network that assigns an input point, given its x_1 and x_2 values, to one of three groups. The overall mechanism stays the same: With linear and non-linear functions the inputs to the NN are transformed to approach their labels as closely as possible [31, p. 100]. Considering regression, one could come to the conclusion to use the same NN architecture for classification, e.g. by utilizing the discrete group values 0 to 2 as labels and mapping the input to a final output node without activation function. However, another architecture for the last layer has proven more efficient: Rather than having one output node for the predicted value, we use a final layer with several nodes, each corresponding to

a possible class, mapping the input to G-dimensional space [44, p. 25]. The goal for classification is to "group" the data points with (non-)linear transformations in $\mathbb{R}^G$ to make them *linearly separable* [44, p. 44]. Our example in fig. 3.6 is not linearly separably, meaning that we cannot separate the data points into their respective groups using "straight lines". In the following sections, we demonstrate how the required mapping is achieved.

We convert our discrete group label values with *one-hot encoding*. Let G be the number of groups that occur in our labels and g denotes any of our groups. The one-hot encoded representation in our example is a vector of size $G = 3$. It contains mostly zeros, only on the $(g+1)$-th position it is 1. [44, p. 25] This leads to the following label representations for our groups:

- Group 0 (blue): $[1, 0, 0]$,
- Group 1 (yellow): $[0, 1, 0]$,
- Group 2 (red): $[0, 0, 1]$.

We could have also given the red group the value 1, the yellow group the value 0 etc. as long as the grouping convention is consistent. If all predictions matched the corresponding labels, the inputs would all be mapped to three points in $\mathbb{R}^3$, depending on the groups the inputs belong to. The result of this ideal case can be viewed in fig. 3.7.

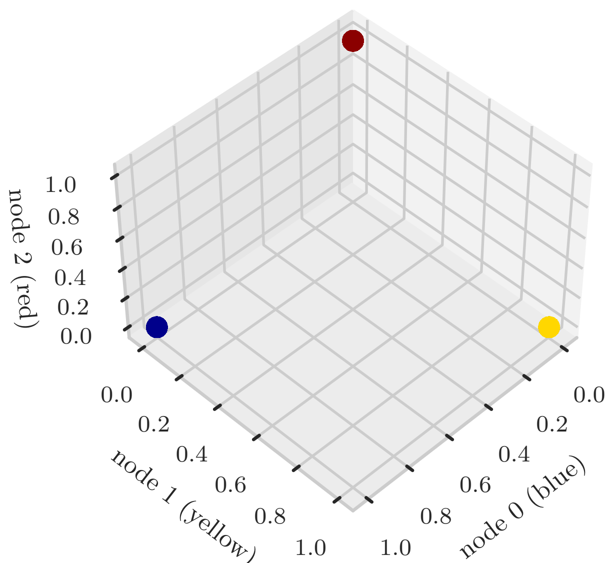


Figure 3.7: Position of the three one-hot encoded labels in $\mathbb{R}^3$

The Softmax Activation Function

Contrary to the NN architecture for regression, the activations of the last layer must be transformed by an activation function. The resulting outputs have to represent the one-hot encoded labels to measure the performance of the model, i.e. one activation of the last layer should approach 1 and the others 0. The

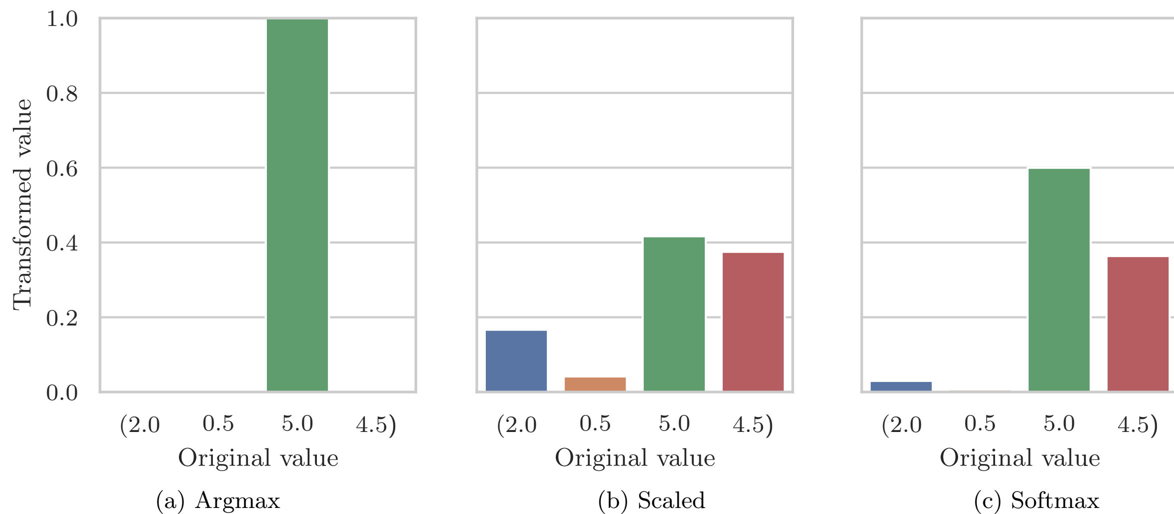
argmax function would achieve this task: It sets the highest activation to 1 and the others to 0. [19, p. 304] However, argmax is not differentiable [61] which does not work for activation functions as will be clarified in section 3.3. Another approach is to *scale* the last layer to 1 by dividing each activation by the sum of all activations. The result is a multinomial distribution, where the sum of the activations in the last layer equals 1 [31, pp. 62–63]:

$$\text{scale}(a)_j = \frac{(a_j)}{\sum_{j=1}^{J}(a_j)}. \tag{3.2.8}$$

This function is *mostly* differentiable: It is not defined for the denominator, i.e. the sum of the activations, equaling 0. Moreover, the activations must have positive values, i.e. $a \in \mathbb{R}^+$. A remedy provides the *softmax* activation function

$$\text{softmax}(a)_j = \frac{\exp(a_j)}{\sum_{j=1}^{J}\exp(a_j)}, \tag{3.2.9}$$

where we apply the exponential function to each activation. [31, p. 81] Hence, we only obtain positive values, leading to differentiability for the whole function. It can be viewed as a "soft" version of the argmax function. [31, p. 187] Due to the exponential function, bigger values are more emphasized and smaller values are diminished. Thus, the weights do not have to take as extreme values as with the scaling function to mimic the labels. In fig. 3.8, we see how four different activation values $a = (2, 0.5, 5, 4.5)$ are transformed by the argmax, scaling and softmax function. With the softmax transformation, the smaller values almost disappear and the gap between the highest and second highest value becomes more obvious.



(a) Argmax (b) Scaled (c) Softmax

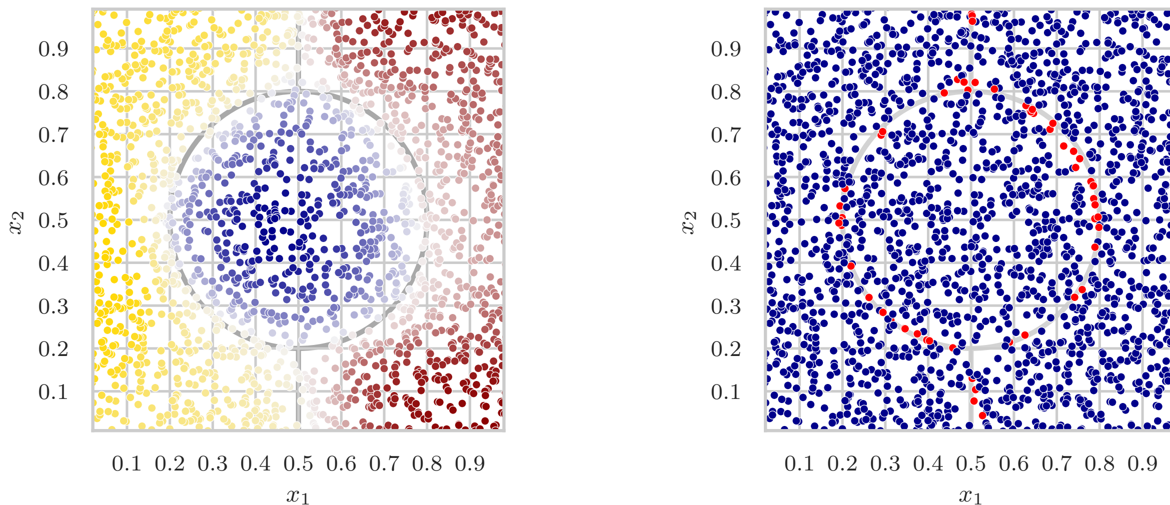
Figure 3.8: The activations of an output layer take the values 2, 0.5, 5 and 4.5. They are transformed by a) the argmax function, b) scaling to 1 and c) by the softmax activation function. The transformed values are plotted in the respective sub-figures.

Coming back to our classification example, for the architecture of our model, we use one hidden layer with three nodes and the *tanh* activation function. The input layer consists of two nodes that take the x_1 and x_2 value, respectively. The output layer consists of three nodes, one node for each class, and the activations of the nodes are transformed by the softmax activation function. The node with the highest output value, thus the highest *probability*, shall indicate which group the input point belongs to.

Evaluation of the Model

Similar to the regression example, we use the mean squared error to evaluate the model. With more than one activation for the output layer, we compute the MSE by taking the mean of the squared differences between the single activations and their respective label values (0 or 1). Additionally, we introduce another metric for evaluation, that expresses what percentage of the data points was correctly predicted – the *accuracy*.

After the learning phase, we evaluated the model's performance using 2000 randomly selected data points. Our results show an overall accuracy of 97.1%, with 58 data points being misclassified. Figure 3.9 visualizes the results: In fig. 3.9a, the data points are colored according to what group the NN assigned them to. The plot also shows the *confidence* of the model's predictions, with darker points indicating higher confidence, i.e. where the maximum value of the activations of a prediction is approaching 1. Near the borders of the groups, the points are lighter colored, indicating lower confidence in the predictions. In fig. 3.9b, the 58 incorrectly classified data points are colored red. Notably, all of these points are located close to the borders of the neighboring groups, indicating that the model was "uncertain" in its classification decisions in these areas.



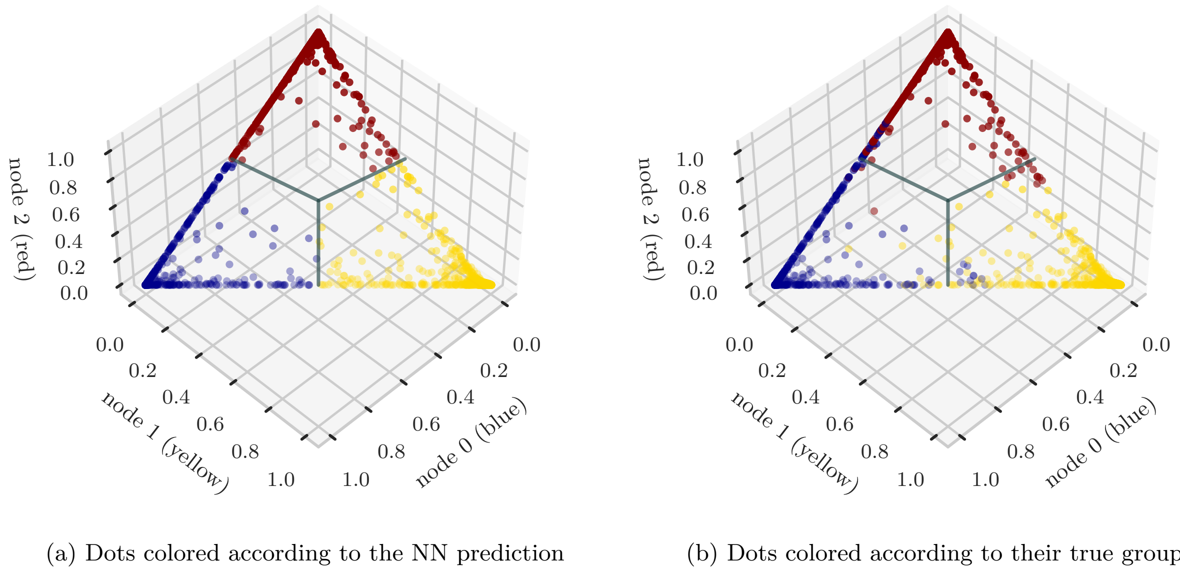
(a) Predicted group assignments and determination of the predictions (the lighter the less determined)

(b) Correct (blue) and incorrect (red) predictions of the test data set

Figure 3.9: Visualization of the classification results

As mentioned previously, with the output activations, the input values (x_1 and x_2) are mapped to points close to their respective class label points (cf. fig. 3.7) in $\mathbb{R}^3$, where the groups are then linearly separable. Figure 3.10 illustrates this transformation. The solution space forms a 2-dimensional equilateral triangle, which is the convex hull of the solution points for the respective groups. This triangular shape is due to the constraint that all coordinates (or activations of the nodes) must sum up to 1. The three gray lines indicate how the groups can be separated linearly. They intersect at (0.33, 0.33, 0.33), where the activation values are equally distributed among the groups. In fig. 3.10a, the data points are colorized according the group the NN assigned them to. Thus, the dots in the groups' areas all have the same color.

In fig. 3.10b, the dots are colored according to which group they *actually* belong to. In direct comparison, we can observe that some points were not correctly assigned by the neural network. The wrongly classified points are mostly located close to the boundaries, which is consistent with fig. 3.9b.



(a) Dots colored according to the NN prediction

(b) Dots colored according to their true group

Figure 3.10: Output of the last layer, mapped to 3-dimensional space. The gray lines indicate the linear separation of the groups.

3.3 Training Process

In the previous sections, we have analyzed how a given input is processed by a Neural Network's weights and activation functions to reach a desired output. How the weights' values can be determined has been neglected so far. Only if the weights have been set correctly, the neural network fulfills its purpose and returns meaningful results.

The correct weights are "learned" by fitting the parameters to the input data. A training cycle considering all training data is called an *epoch* [3, p. 293]: Typically, we need multiple epochs to determine the parameters' values. How the learning process works in detail, is described in the following sections.

3.3.1 Gradient Descent

To adjust the weights, we must first express how much the current output – resulting from the current weights – deviates from the desired output, i.e. the labels. Hence, we introduce the *loss function* L. It typically returns a number as small as possible (e.g. close to 0) if the correct weights were chosen and thus, the correct output was returned.

A common loss function is the *mean squared error* (MSE). If there are N training inputs to a neural network that each possess a label y_n and each produce a prediction $\hat{y}_n$, the MSE is given as [31, p. 108]

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y-\hat{y})_n^2. \tag{3.3.1}$$

Note that further analysis on the loss function and its role in the proposed algorithm is presented in chapter section 3.5. With *gradient descent* we make use of the loss function's results to determine how much and in which direction the weights have to change. The mechanisms of this algorithm are explained by analyzing the following example. In fig. 3.11 we have given a data record with $N = 8$ data points in $\mathbb{Z}^2$, where the x values are our inputs and the y values are the observed results.[3] The red line is the graph of a linear regression model for which the MSE is minimal. With gradient descent, we want to determine the slope and intercept of the linear mapping.

Table 3.1: x and y coordinates of the points in fig. 3.11

x	-4	-2	-1	2	5	6	7	9
y	-3	3	3	12	13	19	23	24

A small neural network without hidden layers is sufficient to model the problem. As the function is linear, there is no need for an activation function for the outputs of the last node. Thus, there are two trainable parameters to determine the activation a of the output layer; a weight w and a bias value b. Considering the explanations of the last chapters, we can conclude that, after learning, the weight value represents the slope of the function and the bias represents the intercept.

[3] For simplicity reasons, only integer values were picked. Of course, gradient descent works with arbitrary values in $\mathbb{R}$, also for the weights and biases.

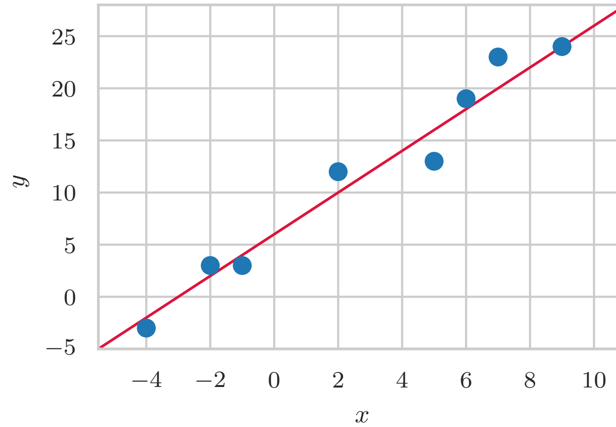


Figure 3.11: Plot of the data points and the corresponding minimum function for the gradient descent example

Adaption of the Weight

At the beginning of the learning process, the weights are typically initialized randomly [19, p. 311]. We first analyze the influence of the activation on the gradient descent by setting the bias (or intercept) to a fixed value $b = 0$. We observe how the MSE changes with different values for $w \in [-2, 7]$, with a step size of 0.5. The plot of the resulting function MSE(w), with the weights on the x-axis and the corresponding MSEs on the y-axis, can be viewed in fig. 3.12. The graph shows that the minimum MSE[4] can be reached with a weight $2 < w < 3$. However, due to the random initialization of the weights, they could have taken any value on the interval.

We analyze two different starting scenarios by assuming that the weight was initialized with $w = -1$ or $w = 5$, which returns a loss of MSE ≈ 505 or MSE ≈ 240, respectively. Using gradient descent we want to adjust the current weight into the right direction to eventually approach the optimum value. We take the derivative

$$\nabla_w L(w) = \frac{d\text{MSE}}{dw} \tag{3.3.2}$$

to acquire the gradient at that point.[5] Referring to eq. (3.3.1), to compute the MSE, we set our labels to $y = xw + b$. Since we currently treat the bias as a constant with $b = 0$, the term is simplified to $y = xw$. With the help of the chain rule, we obtain

$$\frac{d\text{MSE}}{dw} = \frac{1}{M} \sum_{m=1}^{M} 2x_m(x_m w - \hat{y}_m) \tag{3.3.3}$$

for the derivative. The direction of the gradient is an indicator of the direction that leads to a minimum, where the gradient is close to 0 (i.e. in our example where $2 < w < 3$). The name *gradient descent* is derived from the fact that we are using the gradient information to descent the graph and find the minimum.

[4]Note that, in our case, an MSE of 0 cannot be reached, since the data points in fig. 3.11 are not all located on the linear graph and since we neglect the bias value.

[5]As we neglect the bias as a trainable parameter for the moment, we are only interested in the derivative after the weight w. Later, we also include the derivative after the bias $\frac{\partial \text{MSE}}{\partial b}$.

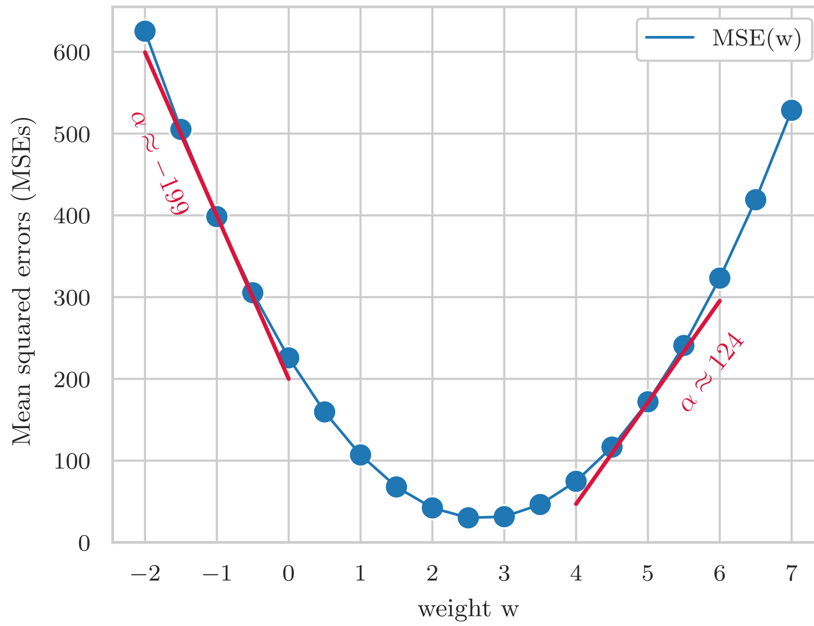


Figure 3.12: Function MSE(w) and derivatives for $w = -1$ and $w = 5$

Inserting our initial weights $w = -1$ and $w = 5$ into eq. (3.3.3) results in the gradients $\alpha \approx -199$ and $\alpha \approx 124$ (cf. fig. 3.12): Considering the case $w = -1$ the weight has to increase, while for $w = 5$ the weight has to decrease. Simply adding the absolute value of the derivative 199 for $w_0 = 5$ to compute the next weight, i.e. $w_1 = w_0 + 199 = 198$, would take us very far from our goal of $w \approx 2$. Therefore, we introduce the *learning rate* η and compute the desired *step size* υ_t with

$$\upsilon_t = \eta \nabla_w L(w_{t-1}). \tag{3.3.4}$$

The current weight is determined by subtracting the step size from the last iteration's weight [73]

$$w_t = w_{t-1} - \upsilon_t, \tag{3.3.5}$$

where choosing $\eta = 0.01$ leads to

$$w_1 = w_0 + \alpha_0 \eta = -1 + 199 \cdot 0.01 = 0.99. \tag{3.3.6}$$

In fig. 3.13 we see that we have further approached the minimum value with $\alpha_1 \approx -92$. However, there is still room for improvement. A next step is performed with

$$w_2 = w_1 + \alpha_1 \eta = 0.99 + 92 \cdot 0.01 = 1.91, \tag{3.3.7}$$

where the new slope is $\alpha_2 \approx -42$. As we are approaching the minimum value, the derivatives of our points become smaller, causing smaller steps. As soon as the derivative drops below 0.01, we stop iterating with an approximated minimum weight $w_{opt} = 2.661 \approx 2.7$. The mean squared error at that point is MSE $= 29$.

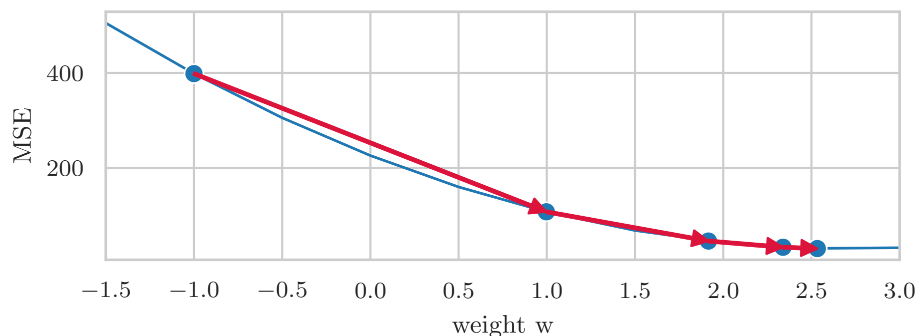


Figure 3.13: Learning rate example

Recall that, in our example, it is impossible to reach an MSE of 0: The minimum of fig. 3.12 does not touch the x-axis, which is caused by fig. 3.11, where the dots deviate from the linear line. Furthermore, a bias value $b > 0$ is necessary to better describe the linear function and to decrease the MSE. It is always possible to construct a neural network that leads to an error of 0, given enough layers and nodes [9, p. 230], which would, however, usually result in *overfitting* (cf. section 3.3.4).

The influence of the learning rate on the convergence behavior of the gradient descent can be observed in fig. 3.14, for the initial weights $w_0 = -1$ (first row) and $w_0 = 6$ (second row). Given the optimal learning rate η_{opt}, the minimum is reached within one step (second column). For $\eta < \eta_{opt}$ (first column) the system will eventually converge to the minimum, but training can be slow. With $\eta_{opt} < \eta < 2\eta_{opt}$ the system also converges, but the resulting weights "oscillate" around the optimum weight. Again, this can lead to a slow training process. With $\eta > 2\eta_{opt}$ the system diverges, i.e. the MSE increases while the weights oscillate. Thus, the size of the learning rate determines how fast the model converges and – considering $\eta > 2\eta_{opt}$ – if the model converges at all. The choice of the right learning rate also plays hand in hand with the initialization of the weights: In our example, we get different optimal learning rates η_{opt} for the initial weights w_0. Typically, the best learning rate is not known initially. It is recommended to start with a learning rate of $\eta = 0.1$ and to experiment whether higher or lower learning rates lead to fast convergence or any convergence at all. [19, pp. 312–313]

Besides the issue with the optimal learning rate, there are several other complications when applying gradient descent. In our current (simple) example, we have only one minimum: With increasing complexity, our weight-loss function can have several minima. The risk arises that the gradient "gets stuck" in a *local minimum*, thus the *global minimum* remains unattained. In fig. 3.15 there is a function with a local minimum on the right and a global minimum on the left. Depending on our initial weight, we reach the global minimum or stay at the local minimum.

Another problem that can occur is *slow learning*. This is especially a problem with *ravines*, where "the surface curves much more steeply in one direction than in another" [74]. They often occur close to minima and lead to an oscillation movement around the optimum weight, resulting in an increased number of steps until the minimum is reached. [67] Likewise *plateaus*, where the gradient of the function and thus the step size become very small, lead to a long training duration.

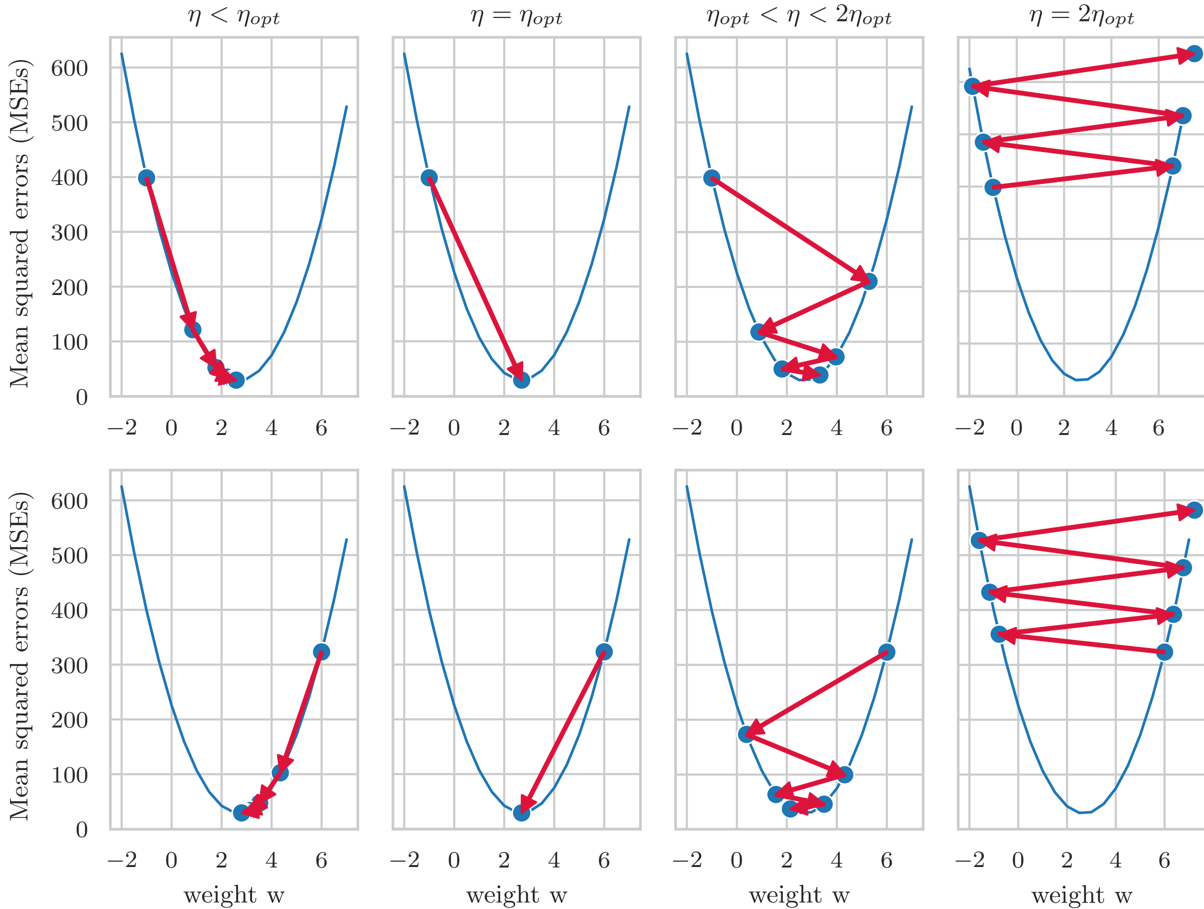


Figure 3.14: Influence of the learning rate η on the learning process for $w_0 = -1$ (first row) and $w_0 = 6$ (second row), cf. [19, p. 312]

Momentum

Therefore, the *momentum* term is introduced. It supports the adaption of new weights by adding the term $\gamma \upsilon_{t-1}$ to the current step size, where γ is the momentum and υ_{t-1} is the step size of the last weight adaption. Thus, the step size is increased depending on the size of the last step and the gradient descent retains "momentum" while moving towards the minimum. The current step size υ_t is given by [67]

$$\upsilon_t = \gamma \upsilon_{t-1} + \eta \nabla_w L(w_{t-1}). \tag{3.3.8}$$

To obtain the weight w_t, we subtract the step size from the previous weight w_{t-1} (cf. eq. (3.3.5)). Figure 3.16 shows a comparison of the gradient descent algorithm without momentum (fig. 3.16a) and with momentum (fig. 3.16b), both with a learning rate of $\eta = 0.02$ and an initial weight of $w_0 = 1.5$. Without momentum we reach the minimum value after 24 steps. With a momentum of $\gamma = 0.3$ we reach the minimum after 8 iterations. Thus, the algorithm converges much faster with momentum.

Typically, the momentum is set to a value around $\gamma = 0.9$ [67]. When we set the momentum to $\gamma = 0.8$ for our example of fig. 3.17a, the weight values oscillate around the minimum (cf. fig. 3.17a). In other words, we "overshoot the target". A "simple modification" [73] of the standard momentum to dampen the

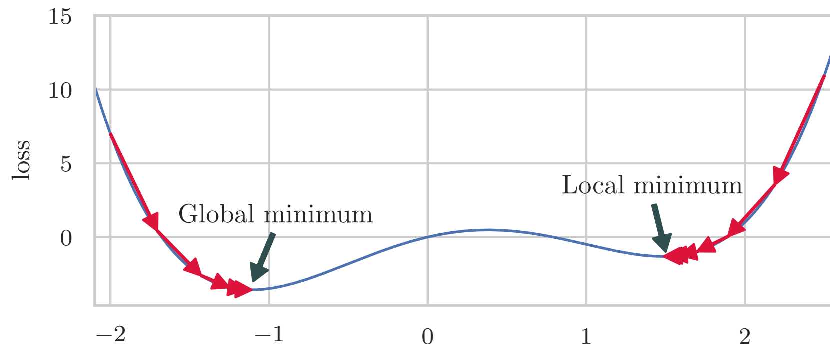
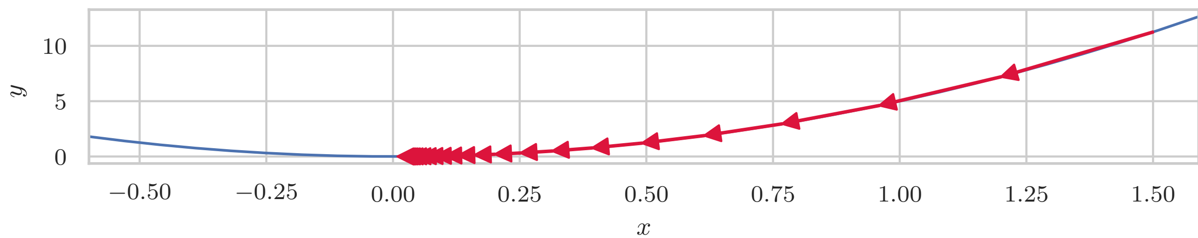
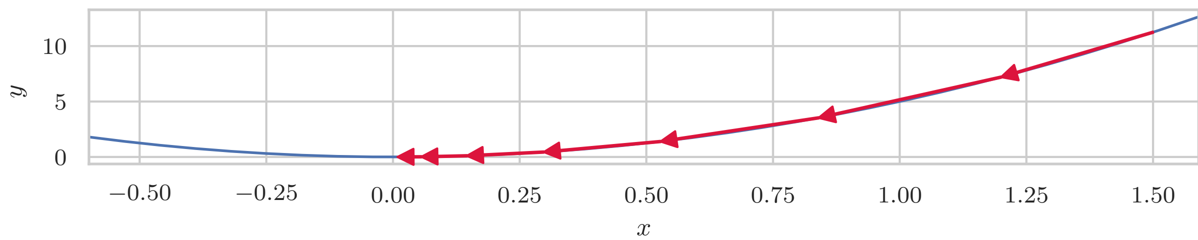


Figure 3.15: Example of learning curve with two minima



(a) Gradient descent without momentum, optimum reached after 32 steps



(b) Gradient descent with momentum ($\gamma = 0.3$), optimum reached after 8 steps

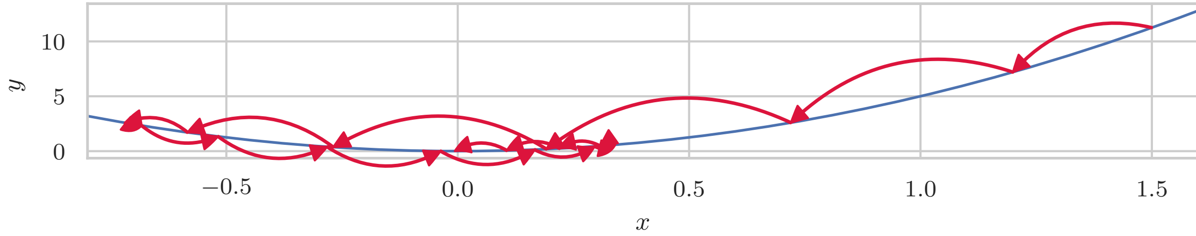
Figure 3.16: Comparison of the gradient descent algorithm with and without momentum, $\eta = 0.02$ and $w_0 = 1.5$. The algorithm converges faster with momentum (b).

oscillation is the *Nesterov accelerated gradient* (NAG), introduced in [54]. The NAG can be interpreted as a friction parameter [1, p. 137]. Contrary to standard momentum, the gradient is evaluated *after* applying the acceleration term of the standard momentum [31, p. 300]. This results in

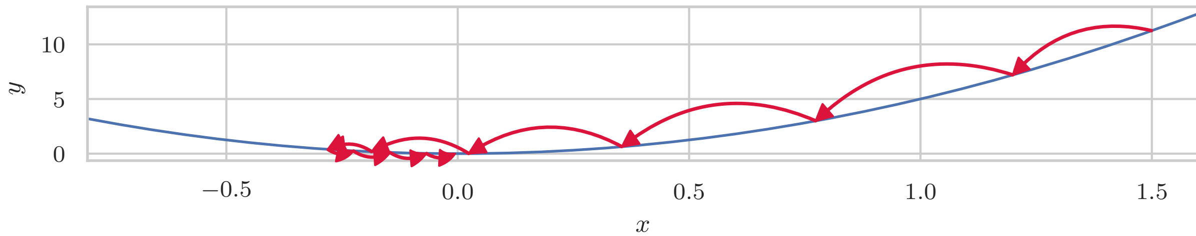
$$\upsilon_t = \gamma\upsilon_{t-1} + \eta\nabla_\theta L(w_{t-1} - \gamma_N\upsilon_{t-1}), \tag{3.3.9}$$

where γ_N is the momentum for gradient evaluation.

Thus, with the NAG we also take a look ahead and consider how the gradient behaves in our step direction. When the gradient changes, it dampens the current movement. The effect can be seen in fig. 3.17: The momentum $\gamma = 0.8$ leads to an oscillation movement around the minimum (fig. 3.17a) and the minimum is reached after 18 iterations. With the NAG, the movement around the minimum is dampened for $\gamma = \gamma_N = 0.8$ (cf. fig. 3.17b) and we reach the optimum already after 12 steps.



(a) Gradient descent with momentum ($\gamma = 0.8$)



(b) Gradient descent with Nesterov momentum ($\gamma = 0.8$)

Figure 3.17: Comparison of the gradient descent algorithm with momentum and with Nesterov momentum, $\eta = 0.02$ and $w_0 = 1.5$. The algorithm converges faster with Nesterov momentum (b).

Adaption of the Bias

So far, we have kept the bias as a constant value $b = 0$. Generally, the bias is also a trainable parameter, that has to be adapted to contribute to a minimum loss. In fig. 3.18 we see how close the current function with $w \approx 2.7$ and $b = 0$ is to the minimum function. It becomes obvious that the intercept has to increase, and thus the bias b. Also, with the change of the intercept, the slope – and thus the weight w – have to decrease. In other words, there is an interdependence between bias and weight and we cannot consider them separately.

As a consequence, we have to compute a gradient with two partial derivatives

$$\nabla_\theta L(\theta) = \left[\frac{\partial \text{MSE}(w,b)}{\partial w}, \frac{\partial \text{MSE}(w,b)}{\partial b}\right]^T = \frac{1}{M}\sum_{m=1}^{M}\begin{bmatrix} 2x_m(x_m w + b - y_m) \\ 2(x_m w + b - y_m)\end{bmatrix} \tag{3.3.10}$$

for the weight w and for the bias b, where $\theta = (w, b)$ denotes all trainable parameters. Note that the bias cannot be neglected in the term of the partial derivative after the weight, since its value can differ from 0. With both weight and bias as adjustable parameters, the solution space stretches to 2-dimensions $\mathbb{R}^2$. In fig. 3.19 we see an excerpt of this solution space with three different convergence paths. The partial derivatives for the weight and bias determine the step size in the direction of the respective axes. Figure 3.19a shows the path for basic gradient descent. The error decreases fast at the beginning; after

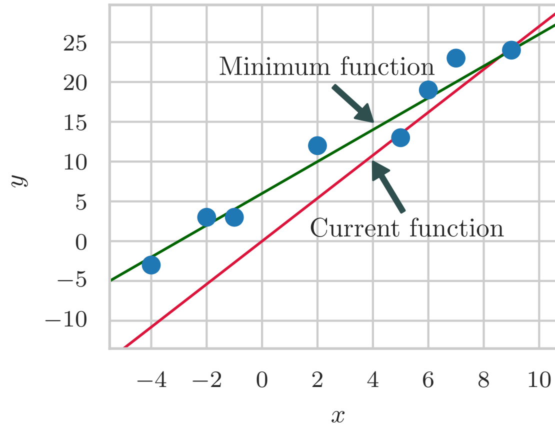


Figure 3.18: Plots of the current function (red) with $w \approx 2.7$ and $b = 0$, and actual minimum function (green)

arriving in the “ravine”, the step size drops rapidly and thus, it converges slowly after 337 steps. In fig. 3.19b we see the convergence path with momentum ($\gamma = 0.85$). The path oscillates around the ravine; close to the optimum, it circles around the minimum value. Nevertheless, it converges much faster than without momentum, requiring only 79 iterations. The third fig. 3.19c traces the path with the NAG ($\gamma = 0.85$, $\gamma_N = 0.99$). The path is more controlled since the oscillation of the momentum path is dampened. Compared to the path without any momentum, the step size is bigger, even on the ravine and the minimum is found after 26 steps. For our example, the minimum is $w = 2$ for the slope and $b = 6$ for the bias.

3.3.2 Optimizers

Algorithms that aim at minimizing the loss function are called *optimizers*. Optimizers that only use the gradient to find a minimum are *first-order optimization algorithms*[6] [31, p. 92]. They are based on the gradient descent method.

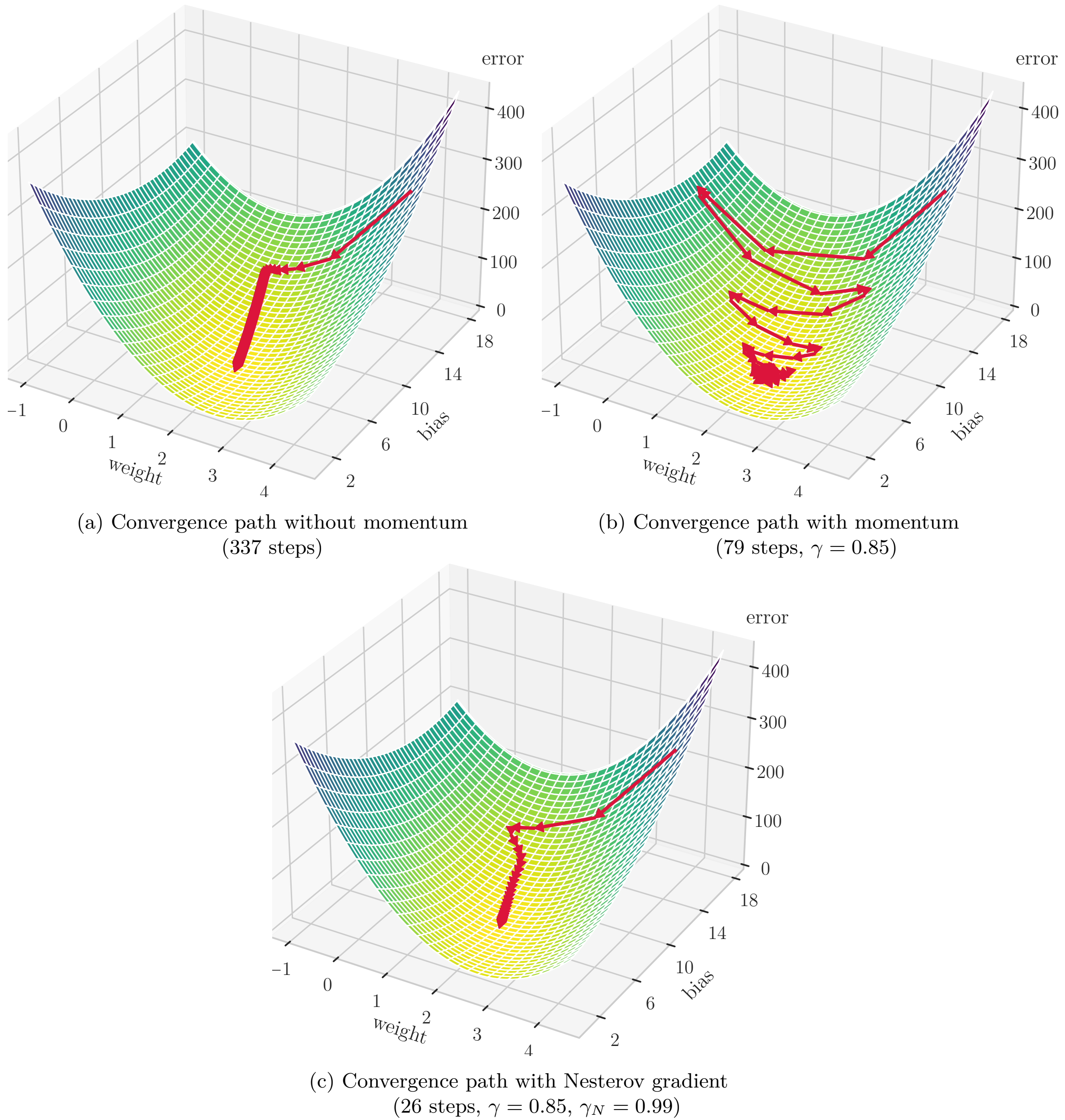
In the last section we analyzed the gradient descent algorithm with a simple example consisting of 8 data points. Generally, the size of data records in ML is substantially larger, with thousands or even millions of data points. Additionally, the architecture of the NN model can be a lot more complex with several layers, nodes and nonlinear activation functions. Likewise, the loss function can be more computationally expensive. Hence, the determination of a single gradient descent step can consume a lot of time and resources. In order to delimit this problem, there is a modification of the ordinary gradient descent algorithm – the *stochastic gradient descent* (SGD): Instead of computing the gradient over the whole *batch* with size B, i.e.

$$\nabla_\theta L = \sum_{m=1}^{B} \nabla L(x_m, \hat{y}_m, \theta), \tag{3.3.11}$$

we draw *one* sample $[x_b, \hat{y}_b]$ uniformly at random from our batch to update θ, resulting in [44, p. 172]

$$\nabla_\theta L \approx \nabla L(x_b, \hat{y}_b, \theta). \tag{3.3.12}$$

[6]Second-order optimizers make use of the *Hessian Matrix*. We will only elaborate first-order optimization in this thesis.



(a) Convergence path without momentum (337 steps)

(b) Convergence path with momentum (79 steps, $\gamma = 0.85$)

(c) Convergence path with Nesterov gradient (26 steps, $\gamma = 0.85$, $\gamma_N = 0.99$)

Figure 3.19: Paths of weights and bias adaptions without momentum (a), with momentum (b) and with Nesterov accelerated gradient (c)

The idea behind SGD is that the resulting gradient is an *expectation*. Contrary to classic gradient descent, updating the gradient for single data points leads to fluctuations in the loss curve while learning. [31, pp. 151–152] Then again, these fluctuations might help to "jump" to better local minima [67].

To further improve the learning process, a batch can be subdivided into *mini-batches* of size B'. For example, if $B = 512$ and $B' = 16$ we obtain $512/16 = 32$ mini-batches. To each of these mini-batches, our optimization algorithm is applied. For SGD this means that a random data point is drawn from *each*

of the mini-batches, which further improves the estimation of our gradient. An iteration cycle over all mini-batches referred as an *epoch*. [31, p. 152]

Generally, SGD has proven to be very efficient. This algorithm, and others deviated from it, are still dominant in modern neural networks. [31, p. 15] In the last section we have addressed techniques to improve the ordinary gradient descent algorithm, like momentum and NAG. They can be easily adapted by SGD or other optimizers.

Other optimizers extend SGD: The *Adagrad* optimizer [18] adapts the learning rate for all parameters in θ individually, taking into consideration all past parameter values. The learning rates for parameters with a high gradient components drop faster than for parameters with small gradients. This can, however, lead to rapidly falling learning rates when applied from the beginning, where gradients are generally larger than later in the training process. The *Root Mean Squared Propagation* optimizer, better known under the acronym *RMSprop* [76], is an enhanced version of Adagrad; it uses a weighted moving average to compute the learning rates for the parameters, rather than taking all parameter values (that might have been large at the beginning) into consideration. The optimizers *Adam* [43] and *Nadam* [17] can be broadly viewed as RMSprop with momentum and NAG, respectively. [31, pp. 307–310][67]

3.3.3 Backpropagation

Via gradient descent we are able to optimize the weight and the bias of an NN for a linear function (section 3.3.1) with respect to the loss L, i.e. $\frac{\partial L}{\partial \theta}$. We have already considered other models – the regression example in section 3.2.2 or the classification example in section 3.2.3 – with an NN comprising at least one hidden layer with several nodes, which results in more than one weight and one bias parameter to optimize. Considering the process of feedforward propagation (cf. section 3.2.1), it is obvious that the activations and outputs of a layer's nodes highly depend on the outputs of the previous layer. Likewise, when we change the last layer's parameters according to gradient descent, we must adjust the previous layers' parameters accordingly.

Generally speaking, backpropagation applies the gradient operation to the layers throughout an NN model, to compute the partial derivatives of the parameters (weights and biases) with respect to the criterion (error) function [19, p. 292]. For this purpose, it is necessary to establish a connection between the gradient of a layer and the previous layer's gradient.

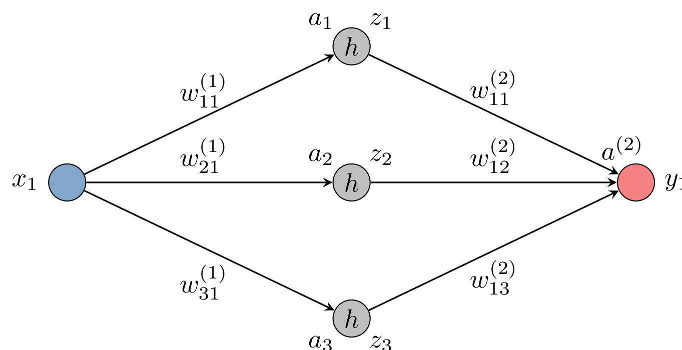


Figure 3.20: Neural network model for the backpropagation example

Figure 3.20 serves as a reference for variable names for the following explanations. It depicts a representation of the classification example model (cf. section 3.2.2) and consists of one node at the input layer, three nodes at the hidden layer, and one node at the output layer. We apply a *tanh* activation function to the hidden layer's activations. The single output node does not have an activation function since we deal with a regression problem. It is assumed that we have determined all activations via *feedforward propagation* for an arbitrary input x, with the respective predicted output value $\hat{y}$, and the loss L of the parameter combinations with respect to a label y. We compute the squared error loss for a particular input x with $L_x = (\hat{y} - y)^2$ and substitute $\hat{y}$ with the output node's activation value a_j, which produces

$$L_x = (a_j - y)^2. \tag{3.3.13}$$

Referring to eq. (3.2.1), recall that the activation value of a node a_j on a layer j consists of the sum of the last layer's outputs z_i multiplied with the weights on the connection between the respective nodes w_{ji}, hence

$$a_j = \sum_i w_{ji} z_i. \tag{3.3.14}$$

We omit the bias term, as it can be seen as an additional weight connected to the current node (cf. section 3.2.1). Next, we substitute the activation in eq. (3.3.13) with eq. (3.3.14) to obtain

$$L_x = (\sum_i w_{ji} z_i - y)^2. \tag{3.3.15}$$

Thus, the resulting function defines the loss L in terms of w. We use the *chain rule* for partial derivatives to compute

$$\nabla L_x(w_{ji}) = \frac{\partial L_x}{\partial w_{ji}} = \frac{\partial L_x}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \tag{3.3.16}$$

Considering eq. (3.3.14), we substitute

$$z_i = \frac{\partial a_j}{\partial w_{ji}}, \tag{3.3.17}$$

and to further shorten the notation, we introduce

$$\delta_j = \frac{\partial L_x}{\partial a_j}, \tag{3.3.18}$$

where δ_j is denoted as the *errors* term. Finally, we obtain

$$\frac{\partial L_x}{\partial w_{ji}} = \delta_j z_i \tag{3.3.19}$$

to describe the influence of the error value change on the weight via the last layer's output. Thus, we have solved the task to produce an equation that builds a connection between the layers' parameters. [9, pp. 243–245]

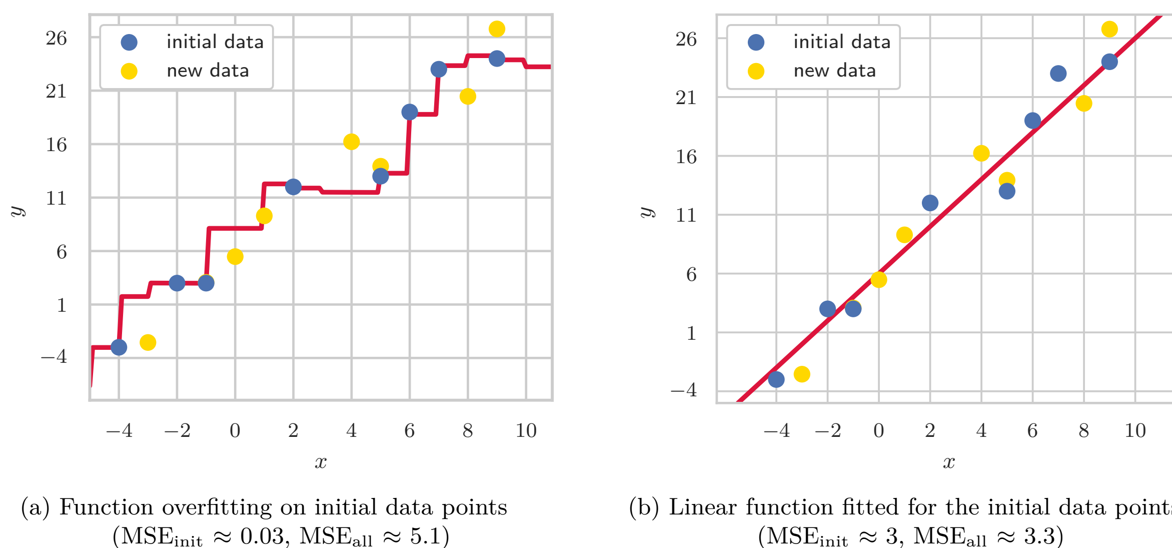
Equation (3.3.19) is applicable to any weight in the NN model, given we have already computed the changes of weights for the following nodes, and the loss L represents any loss function. The size of the parameters' gradients depends on the selected optimization algorithm and parameter values (e.g. for the

learning rate η). If we want to determine the partial derivative with respect to a weight at the beginning of the NN, we need to chain the partial derivatives of all the following nodes connected to that weight.

Since we have to compute the derivatives of the feed-forward function of the neural network and the loss function for gradient descent, it is crucial that all parts of the neural network structure that play a role in the calculations of the forward passing are differentiable. This includes the activation functions used inside an NN and the loss function. As soon as a part of the NN structure is not differentiable, the backpropagation chain is interrupted and the gradients – and thus the parameters – of the previous layers cannot be determined.

3.3.4 Over- and Underfitting

In section 3.3.1 we determined the parameters of a linear function that minimizes the MSE for 8 data points, with a result of MSE $\approx$ 3. Choosing a more complex NN model architecture, we could further decrease the final loss. Figure 3.21a shows the result of a trained model comprising one hidden layer with 4 nodes, that returns a much lower MSE_{init} of 0.03. The graph is closer to the initial points (blue dots) than the linear plot in fig. 3.21b, which we determined via gradient descent for the first data points.



(a) Function overfitting on initial data points ($\text{MSE}_{\text{init}} \approx 0.03$, $\text{MSE}_{\text{all}} \approx 5.1$)

(b) Linear function fitted for the initial data points ($\text{MSE}_{\text{init}} \approx 3$, $\text{MSE}_{\text{all}} \approx 3.3$)

Figure 3.21: Comparison of NN models with respect to overfitting

For a better understanding of the following explanations, we assume that the 8 existing data points are measurement data. For a given circumstance x_i we measure a reaction y_i. Our goal is to create an ML model that predicts the y_i value for any given x_i, so measuring is no longer necessary. Next we assume that 8 more measurements from the same population are added to the data set. They are represented by the yellow dots in fig. 3.21. We observe that the once perfectly fitting line in fig. 3.21a now misses the new data points ($\text{MSE}_{\text{all}} \approx 5.1$) – whereas the original linear function in fig. 3.21b returns a better result for the data points altogether ($\text{MSE}_{\text{all}} \approx 3.3$). This is called *overfitting*: The NN function is not able to *generalize* well on new data [3, p. 41] and thus, it possesses a high *estimation error* [69, p. 114].

To detect an overfitted model, we divide our measurement data into a *training set* and a *test set*: As the name suggests, we use the training set for learning, whereas the test set is not involved in the training process at all – *after* the training process, we use the test set to *evaluate* the model. To the model, the test set behaves like unseen data, thus it simulates future measurements. [31, p. 121] In a way, we have already made use of this technique to detect the current overfitting issue: Initially, we created a model for the first 8 data points and then we evaluated this model on all 16 data points together. Typically, we would only use the test set to estimate the error and not all measurements.

In addition to training and test set, a *validation set* is usually created. It originates from the training set, and, unlike the test set, it is available *during* training: Yet, it is not used to adjust the parameters of the model while learning (like the training set) but it is used to evaluate the model after each epoch. Hence, we can observe how the error of the model changes during the learning phase. The validation set serves to improve the *hyperparameters* of the model, e.g. which loss function or optimizer, how many layers and nodes on the layers to choose, and it gives a first estimation on the performance of the model. Typically, we take 20% from the training data as validation data. [31, p. 121]

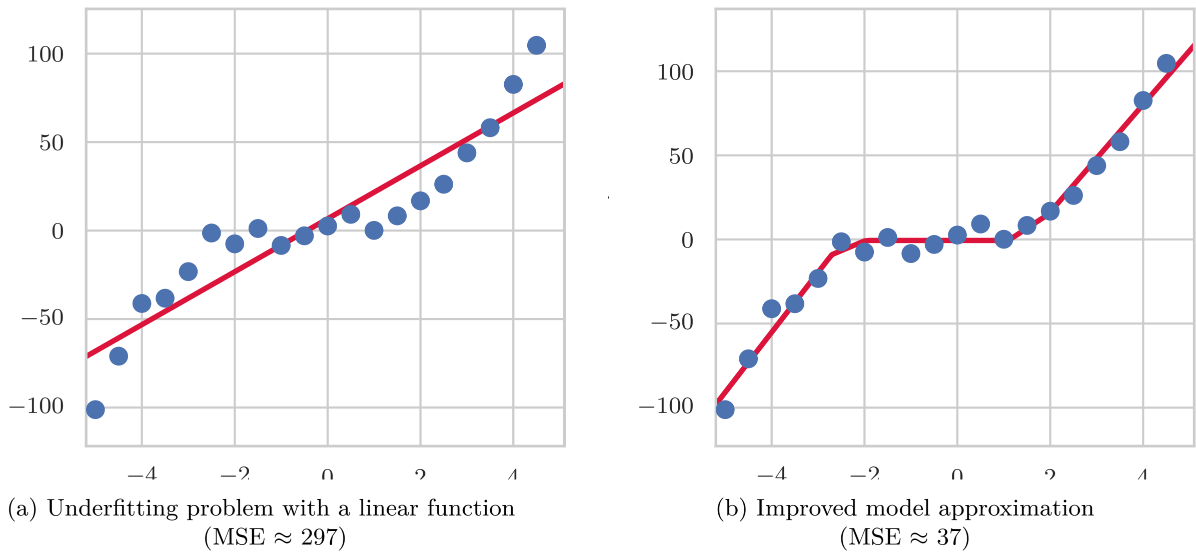
After we have determined the best model for our use case, we can then use the test set to obtain an estimator of the true loss of our function. To improve validation for smaller data sets and to make the error estimations produced by validation more representative, we can use a technique called *k-fold cross validation*: The training set is divided into k equally sized folds and we train the chosen model k times. Each time a different fold is left out while training and the model is evaluated afterwards on the left-out fold. The estimate of the true error is the average of all k validation errors. [69, pp. 119–120]

One countermeasure against overfitting is to include more training data, since the estimation error decreases on a larger training data set. However, in practice, it is not always possible to extend the training data due to limited resources. Similarly, reducing the model complexity e.g. by removing nodes or even layers, improves its generalization ability. However, a less complex model might not be able to capture the relations between data input and labels. [1, p. 25] Thus, multiple possible models should be tested using the validation data set; a model that returns a training error close to the validation error is to be chosen to reduce the risk of overfitting.

Nevertheless, when preventing overfitting, it has to be considered that a higher error on the validation and test data compared to the training data is to be expected [3, p. 41]. Hence we do not have to strive for a model that performs equally good on training and test data.

Another technique to efficiently reduce overfitting is *dropout*: A previously defined amount of nodes in an NN input or hidden layer is deactivated. These nodes are chosen randomly on each training step and thus, the NN model is confronted with varying signals it has to account for. Additionally, with nodes randomly remove from the system, the model's complexity is reduced. Note that dropout is only active during training. [71]

An excessive reduction of the model complexity might lead to the opposite problem: *underfitting*. In fig. 3.22a we see data points with the plot of a linear function that reduces the loss to MSE = 297: The linear function does not seem to be a good choice to reduce the error. With underfitting, we have the problem of a large *approximation error* [69, p. 114]. The model complexity has to be increased by adding non-linear elements and more nodes to avoid underfitting. In fig. 3.22b the approximation of a NN model with one hidden layer with 5 nodes reduces the loss to MSE = 37.



(a) Underfitting problem with a linear function (MSE $\approx$ 297)

(b) Improved model approximation (MSE $\approx$ 37)

Figure 3.22: Comparison of NN models with respect to underfitting

In summary, determining the correct hyperparameters of an NN model is often an iterative process. Beginning from an underfitting or overfitting state, we can step by step approximate the desired solution. Using different data sets to validate and test the models supports the learning process.

3.4 Deep Learning

Recall from section 3.1.3 that we introduced deep learning as a subfield of machine learning, where the layers of NN models are increasingly meaningful representations of the input. To visually explain these representation layers and other concepts in machine learning, we make use of the popular MNIST dataset [47]. It contains 60.000 training and 10.000 test images of handwritten digits. The images were centered and size-normalized to 28x28 pixels. Given an image of a handwritten digit, the goal is to predict which digit is currently considered.

In fig. 3.23 we see how a handwritten digit 9 as the input x is transformed by three consecutive layers of a deep learning neural network. The new representations are generated with *convolutional layers* (cf. section 3.4.1). With each layer, the representation of the original input changes. Some of the representations still remind the human reader of the input – others don't. In the first layer representations the corners of the handwritten digit are emphasized and hence, they seem to be of relevance for the transformation process. In the second layer representation we obtain blurred versions of the original digit. The third layer representations seem like a random collection of pixels to the human observer, whereas for the NN, the level of information is still high enough to recognize the original digit in the last layer.

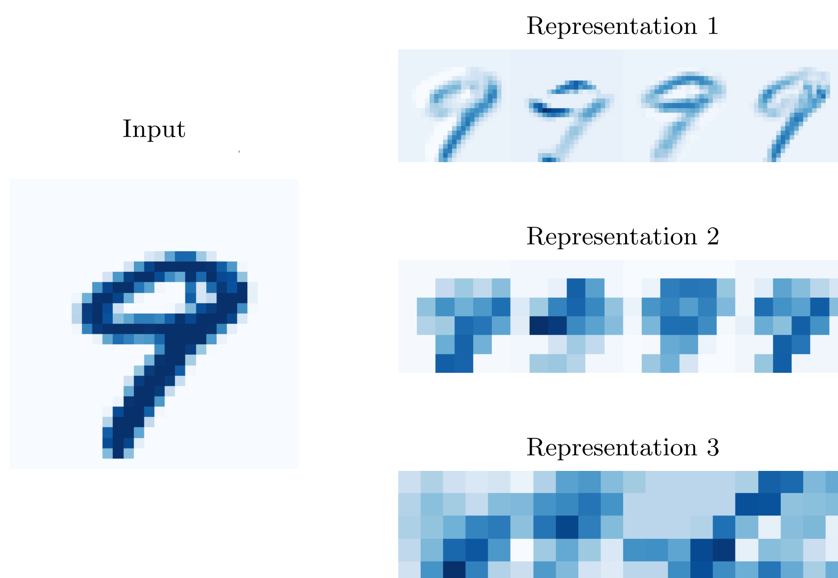


Figure 3.23: Representations of a given input produced by a deep learning Neural Network (cf. [13, p. 28])

Besides convolutional layers, which are often used in image recognition, there are other types of layers that support the representation of the analyzed data. *Recurrent neural networks* are used for sequential data as they have the ability to "remember" data points earlier in the sequence and account for them during prediction. *Autoencoders* are a type of Neural Network structure. They encode the input to find a representation of the data in a lower dimension; then decoding layers restore the original dimensionality of the data. The result is an alternative representation of the input data that emphasizes important features. [31, pp. 1–26]

We will further analyze convolutional neural networks (cf. section 3.4.1) and recurrent neural networks (cf. section 3.4.2) in the next sections, since these types of deep learning neural networks are of particular relevance for deep learning or for the presented approach in this thesis.

3.4.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are specialized to transform "grid-structured" data with "strong spatial dependencies in local regions of the grid" [1, p. 315]. Therefore, CNNs are useful for image analysis as each pixel of an image represents a cell in a 2D-grid [31, p. 330].

Characteristics

In contrast to the previously considered dense layers, convolutional layers perform *sparse interactions*: The input I is transformed to a new representation of smaller size. We achieve this down scaling with a *kernel* K, a matrix typically smaller than the input, which we have defined beforehand for our convolutional layer. We will later consider the exact mathematical operations performed as well as an example that illustrates the sparse interaction. It is important to note that the kernel does not interact with the whole input at once, but it is sliding over the input, transforming an area as big as the kernel on each step. For each operation, the kernel uses the same parameters, which brings us the next characteristic of convolutional layers, the *parameter sharing*: For each transformation the kernel K performs on the input I, the kernel's parameters stay the same. Thus, compared to dense layers, the memory needed to save the parameters of the layer is significantly lower. The parameter sharing is also important for the next characteristic, the *equivariant representations*: When the kernel processes a specific part of the input, e.g. 4 neighboring pixels of an image, it will produce a similar output for other neighboring pixels with a similar structure, even if those pixels are located at the other side of the image. Due to this mechanism, specific *features* of an input can be identified and reflected in the output. For this reason the output S is also referred to as *feature matrix.* [31, pp. 335–338]

Convolutional neural networks have had great influence on deep learning and they are often associated with being *the* deep learning approach. They outperformed existing ML approaches in different applications even in their early stage. Moreover, they act as a blueprint for more advanced deep learning techniques. [1, p. 316]

Mathematical Background

As the name suggests, contrary to dense layers, convolutional layers use *convolutions* instead of matrix multiplications to compute outputs of a layer. The convolution $x * w$ between a function x and a weighting function w is defined by

$$s(t) = (x * w)(t) = \int_{a,t \in \mathbb{R}^n} x(a)w(t-a)da \tag{3.4.1}$$

Applied to neural networks, the function x corresponds to the input to our layer I and the function w corresponds to a set of parameters, i.e. the kernel K. Input and kernel are matrices, also called *tensors*, of *finite* size. Thus, we require a *discretization* of the convolution. We assume that the input and kernel functions are 0 outside of our scope for which we defined the values for kernel and input. When analyzing

2-dimensional inputs, like image data, we may want to define a kernel with rank 2 and process the input with the finite variant of convolutions,

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m,j-n). \tag{3.4.2}$$

Note that convolutional operations are commutative. Hence, we can equivalently write

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m,j-n)K(m,n), \tag{3.4.3}$$

which is more convenient to implement, since the values of the kernel stay constant. Typically, machine learning libraries use *cross-correlation* instead of pure convolutions. However, the terms are often used interchangeably. Cross-correlation is denoted by the operator $*$ as well and defined by

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m,j+n)K(m,n). \tag{3.4.4}$$

We will use cross-correlation in the following examples. The Kernel values m and n are also called the kernel's *weights* – they are typically trainable parameters. [31, pp. 331–333]

Convolutional Layer Examples

In fig. 3.24 we see how an input grid I with the cell values a to p is transformed by a 2×2 kernel with the weights $K = [[\omega, \xi], [\upsilon, \zeta]]$. Starting from the top left corner, the kernel slides from left to right and from top to bottom while it performs cross-correlation to a 2×2 sub-matrix in its current scope. Due to

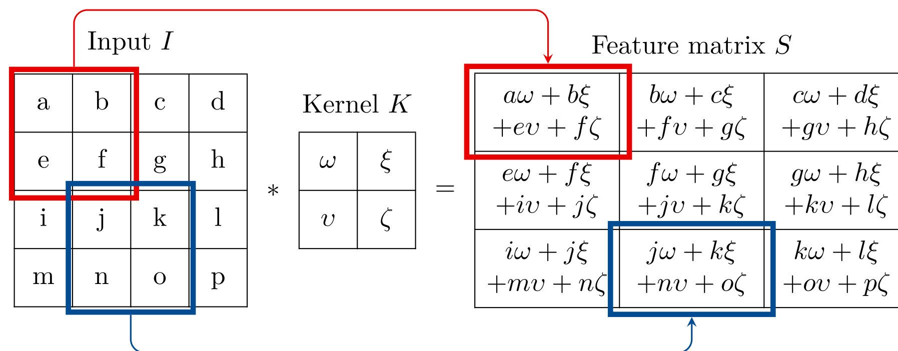


Figure 3.24: Feature matrix produced by a convolutional layer with a kernel size of 2×2 and an input of size 4×4 (cf. [31, p. 334])

the convolutions the feature matrix is of smaller size than the input tensor. If we wish to keep the input's dimensions with the output, we can use *zero padding*: The input image is enlarged with zeros on the left and top side. The kernel incorporates the zeros when sliding over the input, which leads to more kernel operations. This way, we keep the dimension of the input for our output. Likewise, it is also possible that the kernel takes bigger steps when sliding over the input: With a step size of $(2,2)$, the kernel would slide two steps to the side or below after a convolution. Consequently, our feature matrix would only have 4 entries spread over two rows. This would further decrease the output size compared to the standard strides of $(1,1)$.

Referring to fig. 3.23, the first representation was reached with a convolutional layer with a kernel size of 4×4, which downsized the original image from 28×28 to 25×25 pixels. There are 4 similar images in the first representation, since the *filter* variable was set to 4, which means that the layer produced an output for four different kernels and the resulting convolutions. To illustrate all filter results, they were placed next to each other in fig. 3.23. However, the model stores them in a rank-3 tensor of size $25 \times 25 \times 4$.

We always apply the same kernel values with the convolutional operation, and thus we make use of *parameter sharing* during the process. Consequently, the feature matrix contains similar output values in cells that have similar input values. We observe this effect in representation 1 of fig. 3.23: Along the edges of the written number, the resulting filter cells resemble each other. This allows for a reduction of the number of model parameters without losing valuable information to classify the image – a trait that is especially important for highly resolved images that take up a lot of memory to process. [31, p. 254]

Pooling layers

Besides the typical convolutional layers that perform convolutions – or cross-correlations – with a kernel on the input, deep neural networks contain *pooling layers*: They possess a *filter* with a specific size, similar to a kernel, but instead of convolutions, the filter transforms the input with *aggregate functions*. Most common are *max-pooling* and *mean-pooling* layers. Note that the pooling filters are not to be confused with the filters produced by different kernel parameters in standard convolutional layers. Therefore, we will refer to the latter as *kernel-filters*. In fig. 3.25 we see an example of a filter performing max-pooling on the input. The movement of the filter over the input is the same as with the kernel in fig. 3.24, seeking the maximum of the grid currently under the filter's scope. Thus, we have the same downsizing effect as with the convolutional layer in fig. 3.24. A striking difference to convolutions is the lack of trainable parameters within the pooling operation, as there are no filter values to optimize. [1, pp. 326–327]

In fig. 3.23 we have used max-pooling layers to reach the second representation of the input. Compared to representation 1, which was gained with a convolutional layer, they appear far more "pixelated", the edges are not as well recognizable. The second representation was downsized to only 6×6 pixels because we used strides of $(4, 4)$ with a 4×4 filter. Thus, the filter processed each pixel only once.

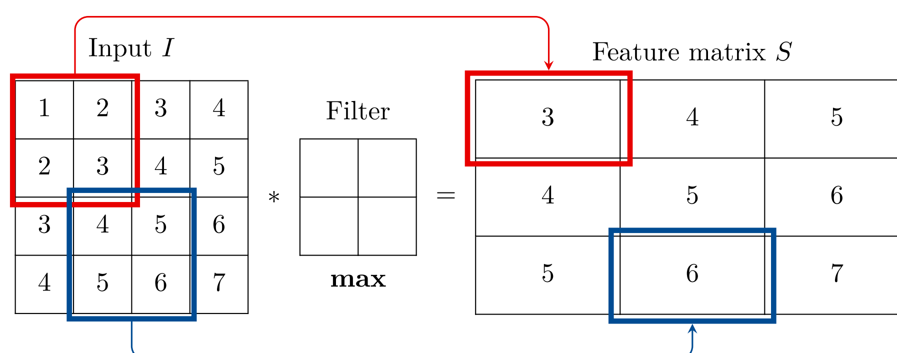


Figure 3.25: Feature matrix produced by a max pooling layer with a filter size of 2×2 on an input of size 4×4 (cf. [31, p. 334])

Note that we cannot create multiple filters, i.e. increase the dimensionality of the input, with max-pooling, since there are no filter parameters to vary [1, p. 326]. The max-pooling operation was applied to all kernel-

filters of representation 1 separately, which diminished the output size of a single kernel-filter to 6×6. For representation 3, we used a convolutional layer again, with a kernel size of $4 \times 4 \times 4$.

The representation produced by a pooling layer is less sensitive to "small translations": When we shift the input in any direction, the max-pooling layer would still be able to filter out specific features. Additionally, striking features of the input are emphasized, whereas smaller details are suppressed. [31, p. 342]

3.4.2 Recurrent Neural Networks

The type of data that we have analyzed in our examples so far comprises *independent* samples: In which *order* we feed the data to the network is irrelevant for the training result.[7] The same is true for inference: Here, we can evaluate all data points independently of each other.

Yet, there are data structures where one sample depends on the previous one. By changing the order, we could no longer make sense of the data and identify a pattern. These structures are referred to as *sequential* data. Typically, the goal is to determine the values that join the sequence next. Sequential data can be divided in *real-valued* and *symbolic* data. The most common data types where these requirements apply are [1, p. 271]

- *Time-series data*: Generally real-valued data that originates from a time sequence, e.g. measurements. For example, we can estimate the temperature of the next day if we know the weather conditions (pressure, humidity) of the previous days. The order of the input data matters for the prediction: We cannot predict the temperature of a day in August if we consider data from a day in January as the previous day in the sequence.
- *Text data*: Words in a text can be seen as a sequence. Although the overall message of a text can often be determined by analyzing all words at once, thus without an order (*bag of words*), more fine-grained, semantic analyses, e.g. for *natural language processing*, can only be produced with sequenced data. Words are symbolic features, as they cannot be compare by numerical values, i.e. the encoded characters. For instance, the words "dessert" and "desert" look almost the same and thus, might be encoded similarly, but have completely different meanings.

The pattern in sequential data can only be identified if the previous values in the sequence are included in the forecasting. As we have analyzed in section 3.2.1, during the feedforward process for an input data point x_t, there is no provision for previous values $x_{t-1}, ..., x_0$ to be considered when predicting the output. This shortcoming led to the development of *recurrent neural networks* (RNNs), which, in short, are feedforward neural networks with "feedback connections" [31, p. 168].

Most commonly, with RNNs we want to predict the next value in the sequence – be it a numerical value or a word.[8] However, we will in the following concentrate on the prediction of numerical sequence values. To make the examples easier to understand, matrix variable will be in bold print throughout this chapter.

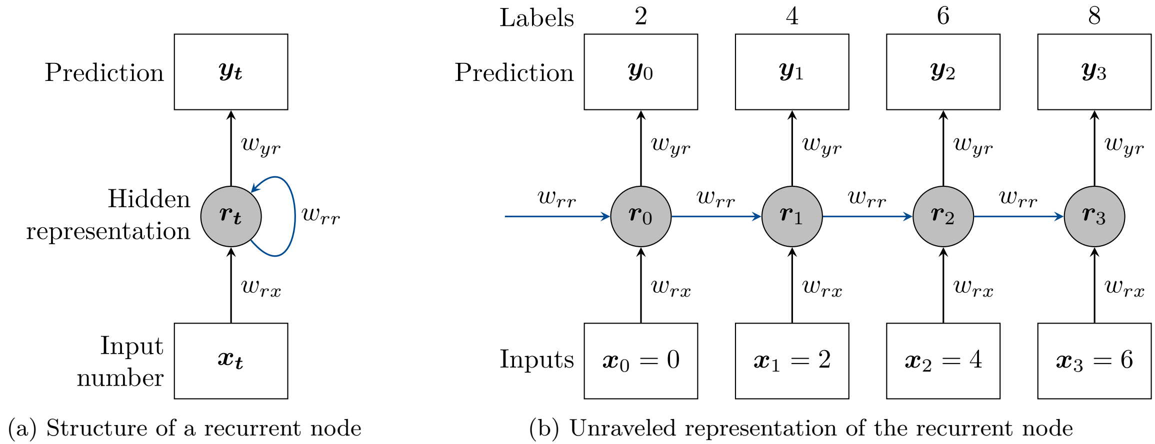
---

[7]Naturally, the training results can differ. However, this is due to chance, e.g. how the parameters were initialized.

[8]To be precise, we cannot simply "predict a word": The words must have been transformed to numerical values before training, e.g. via an *embedding layer*. As natural language processing is not part of this thesis, we will not further elaborate on this topic.

Structure

The feedback connections are realized by *hidden states* attached to the nodes of a recurrent layer: They take into account – in other words “remember” – the values that have (recently) passed through the node. In fig. 3.26 we see two representations of a recurrent node. The first one in fig. 3.26a describes the general form. As with ordinary feedforward nodes, each value $\boldsymbol{x}_t$ in an input sequence $\boldsymbol{x}$ is transformed, i.e. multiplied, by the weight w_{rx} directed to the node. The node itself possesses the hidden states r, indicated by the blue loop going from and to the node, which represents the biggest difference to a feedforward node. [1, p. 276]



(a) Structure of a recurrent node

(b) Unraveled representation of the recurrent node

Figure 3.26: Structure of a recurrent node (cf. [1, p. 275])

When passing the node, each value $\boldsymbol{w}_{rx}\boldsymbol{x}_t$ is transformed by the additional weight $\boldsymbol{w}_{rr}$, and the result is added to the hidden state. The current value of the hidden state is then transformed by the following weight $\boldsymbol{w}_{yr}$ and returned as the prediction of the value next in sequence. With

$$\boldsymbol{r}_t = f(\boldsymbol{r}_{t-1}, \boldsymbol{w}_{rx}\boldsymbol{x}_t) \tag{3.4.5}$$

we can describe this procedure in general terms. The current hidden state $\boldsymbol{r}_t$ is both a function the previous hidden state $\boldsymbol{r}_{t-1}$ and the current input value.

Figure 3.26b shows the “unraveled” representation of the hidden unit: The iterations of the loop are displayed next to each other as feedforward layers. Now it is more visible that, unlike the hidden states, the weights around it, including the “loop weight” $\boldsymbol{w}_{rr}$, *do not change* between inputs.

Thus, they are *shared weights*. With[9]

$$\boldsymbol{r}_t = h(\boldsymbol{w}_{rx}\boldsymbol{x}_t + \boldsymbol{w}_{rr}\boldsymbol{r}_{t-1}) \tag{3.4.6}$$

we compute the hidden states in $\boldsymbol{r}$. Analogous to feedforward nodes, we sum up the weights connected to the node, multiplied by the previous nodes: In this case an input value $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{r}_{t-1}$. We can apply any activation function h to add non-linearity to the term.

[9]The bias parameter of the hidden state was neglected to enhance readability. It is treated like bias parameters in a feedforward NN.

The final output $\boldsymbol{y}_t$ is then evaluated by [1, p. 276]

$$\boldsymbol{y}_t = w_{yr}\boldsymbol{r}_t. \tag{3.4.7}$$

As an example we feed an input sequence $x = (0, 2, 4, 6)$ to the small RNN. One after another, the values are processed by the hidden unit. As we want to predict the value coming next in sequence, the label corresponding to $\boldsymbol{x}_t$ is the following value $\boldsymbol{x}_{t+1}$. The error is then computed via a loss function taking the label and the predicted number as a sequence.

For backpropagation, the procedure discussed in section 3.3.3 can be applied to the unraveled form of an RNN to adjust the parameters without any additional algorithms [31, pp. 384–385].

A crucial advantage of this structure is that RNNs can process arrays of arbitrary length: Both during training and prediction, we can pass arrays of different sizes to the network. This makes RNNs comparatively flexible as we do not need to set up a model and train a network for each possible length. [31, p. 474] This is due to the hidden state's structure, where the number of iterations for the feedback loop is not determined. Thus, the number of sequential nodes in the unraveled representation can vary.

It should be noted that there are different "design patterns" for RNNs. This affects primarily the number of inputs and outputs of the network. In fig. 3.26 each iteration of the hidden representation loop receives an input and produces an output. This pattern is especially useful if a sequence element consists of multiple values: We can predict the following value for each value in the sequence element, i.e. $\boldsymbol{x}_{t+1}$. For example, to predict the weather for the next day, there are several input values to be processed, e.g. temperature, humidity, air pressure. Returning the hidden state of each variable leads to an information gain and it enables to obtain a prediction for each variable. We can also lay emphasis on the target value $\boldsymbol{y}_3$, which is the element following the whole sequence $\boldsymbol{x}$. Likewise, we can retain only the first input $\boldsymbol{x}_0$: For example, the input could be a single picture and we would like to obtain a sequence of words $\boldsymbol{y}$, i.e. a sentence, that describes the picture. [31, pp. 387–388][1, p. 283]

Vanishing and exploding gradients

A pitfall when training RNNs is the problem of *vanishing and exploding gradients*. This issue also occurs when training any other deep network, e.g. CNNs. However, the problem is more present with RNNs. This is due to the weight $\boldsymbol{w}_{rr}$ of the hidden units: Depending on its value, the hidden representation $\boldsymbol{r}_t$ increases or decreases. Assume that the loss L depends only on the final output $\boldsymbol{y}_t$ Then the gradient applied to a simple hidden unit can be determined with

$$\frac{\partial L}{\partial \boldsymbol{r}_t} = h'(\boldsymbol{r}_{t+1})\boldsymbol{w}_{rr}\frac{\partial L}{\partial \boldsymbol{r}_{t+1}}, \tag{3.4.8}$$

where h' is the derivative of the arbitrary activation function h. By induction, in order to compute the gradient for a hidden representation $\boldsymbol{r}_t$ of an input $\boldsymbol{x}_t$, all hidden representations of the following values in the sequence $\boldsymbol{r}_{t+1}, ..., \boldsymbol{r}_T$ need to be multiplied by the shared weight $\boldsymbol{w}_{rr}$. If $|\boldsymbol{w}_{rr}| < 1$, the gradient $\nabla_{\boldsymbol{r}_t} L$ becomes consistently smaller, i.e. it might *vanish*; likewise, provided that $|\boldsymbol{w}_{rr}| > 1$, the gradient increases consistently, *the further we go back in the sequence*, i.e. it might *explode*. Hence, with vanishing gradients, values at the beginning of the sequence lose influence, and values at the end of the sequence obtain too much influence on the predictions. In the same manner, exploding gradients form the reverse

case. Additionally, the activation function h does also have influence on the gradients' sizes. For example, the *tanh* activation function limits the results of $h'(\boldsymbol{r}_{t+1})$ to the interval $(0,1]$, which alleviates the problem of exploding gradients. [1, pp. 286–287]

The circumstance of vanishing gradients is illustrated in fig. 3.27. The sequence $\boldsymbol{x} = (16, 8, 4, 2)$ is fed to a recurrent unit. We want to predict the value coming next in sequence. Since the values are halved at each step, the sequence has to be extended with $0.5, 0.25, 0.125$ and so on. The colored bars represent the hidden states $\boldsymbol{r}_t$ and how much the inputs $\boldsymbol{x}_t$ contribute to the results, given $\boldsymbol{w}_{rr} < 1$. At the beginning, 16 (red) has the highest influence. We can observe that the influence of this value drops continuously. At the end, its contribution to the recurrent state is negligible. As the values 4 (green) and 2 (orange) have the biggest share, the prediction of the next number in sequence will mostly depend on them. Thus, we might come to false conclusions: For instance, looking only at the last 2 values, the next numbers could be 0, -2, -4 and so on.

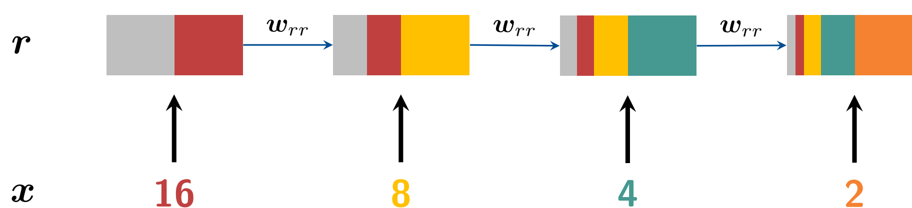


Figure 3.27: The vanishing gradient problem over a sequence

We have only considered one hidden unit and still, the problem of vanishing gradients has become evident. The problem increases with multiple layers connected in series, since the error is propagated as an input from layer to layer. Thus, ordinary RNN units fail at retaining long-term dependencies [37].

Gated RNNs

To address the problem of vanishing and exploding gradients, *gated* RNNs like *gated recurrent units* (GRUs) and *long short-term memory* (LSTM) layers were developed. These RNN units have the ability to "forget" certain parts of the sequence that have a negligible influence on the solution to not "pollute" the value of the hidden state. The gates regulate the data flow: They decide on which information has to be retained and which information can be forgotten.

The mechanism of the gates uses the *sigmoid* (σ) activation function

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{x}} \tag{3.4.9}$$

to map incoming values to the interval $[0, 1]$. The closer the actvation to 1, the more of the information has to be retained; when the value approaches 0, the information is erased.

LSTMs

LSTMs were introduced in [38] and contain 3 regulating gates. In fig. 3.28 the structure of an LSTM cell with the *input gate*, *forget gate* and *output gate* is illustrated. There is also an ungated input node that uses the *tanh* activation function to transform the information. Like simple RNN cells, LSTM cells possess a hidden unit $\boldsymbol{r}_t$ that is passed on to the next cell in line and saves information about previous inputs.

Additionally, a *cell state* $\boldsymbol{c}_t$ is computed; similar to the hidden unit, it is passed on to the next cell in the sequence, where it influences both $\boldsymbol{c}_{t+1}$ and $\boldsymbol{r}_{t+1}$.

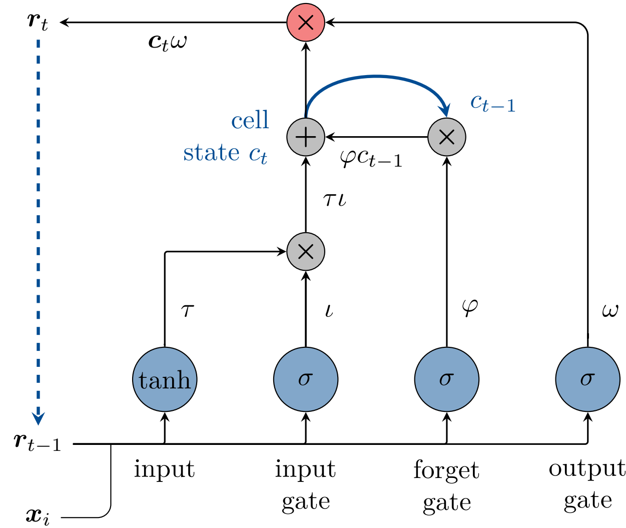


Figure 3.28: Structure of an LSTM cell

The input values to the LSTM cell $\boldsymbol{r}_{t-1}$ and $\boldsymbol{x}_i$ can be viewed as a layer with two nodes, densely connected to a layer of 4 nodes, comprising the input node and the 3 gate nodes. Thus, we can compute τ, ι, φ and ω, the outputs of input, input gate, forget gate and output gate nodes, with

$$\tau = \tanh\left(\boldsymbol{w}_\tau\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\tau\right), \tag{3.4.10}$$

$$\iota = \sigma\left(\boldsymbol{w}_\iota\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\iota\right), \tag{3.4.11}$$

$$\varphi = \sigma\left(\boldsymbol{w}_\varphi\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\varphi\right), \tag{3.4.12}$$

$$\omega = \sigma\left(\boldsymbol{w}_\omega\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\omega\right), \tag{3.4.13}$$

where $\boldsymbol{w}$ and b are the weights and the bias of the respective input and gate nodes.

The input node transforms the input values with a *tanh* activation function (τ). Simultaneously, the combination passes the **input gate**, also called *update gate* in literature, where the sigmoid activation function creates an output between 0 and 1 (ι). Afterwards, we multiply τ and ι ($\tau\iota$). The closer ι is to 1, the more of τ is kept in the system. Hence the name “input gate”: We determine how much of the ungated input is retained.

The output of the **forget gate** (φ) is multiplied by the previous cell's cell state $\boldsymbol{c}_{t-1}$. Thus, the forget gate influences how much of the current cell can be “forgotten”. We have now all the information to determine the new cell state with

$$\boldsymbol{c}_t = \tau \odot \iota + \varphi \odot \boldsymbol{c}_{t-1}. \tag{3.4.14}$$

This cell state is directly passed on the next LSTM cell in sequence, symbolized by the blue arrow going back to where $\boldsymbol{c}_{t-1}$ is entering the system.

The other output of the cell, the hidden state $\boldsymbol{r}_t$, is the product of the cell state $\boldsymbol{c}_t$ and the output of the **output gate** ω. Thus, the output gate determines via

$$\boldsymbol{r}_t = \boldsymbol{c}_t \odot \omega \tag{3.4.15}$$

how much of the computed cell state $\boldsymbol{c}_t$ is passed on as the hidden state $\boldsymbol{r}_t$. In the next LSTM unit it turns into $\boldsymbol{r}_{t-1}$, denoted in fig. 3.28 by the blue dashed line.

To summarize, unlike simple RNN cells, LSTMs comprise a further state that is passed on to the next cell: The cell state $\boldsymbol{c}_t$. They use three different gates to determine how much of the input (input gate), the output (output gate) and the cell state (forget gate) is kept in the system. The input and output gates directly influence the cell state and thus, indirectly the output of the cell. The output of the cell is the hidden state $\boldsymbol{r}_t$, which is then fed to the next cell, together with the cell state $\boldsymbol{c}_t$.

GRUs

Although GRUs were introduced later than LSTMs [12], they can be viewed as a slimmed down version of the latter. GRUs have no cell state, thus they only pass on the hidden unit's value to the next cell. Moreover, they only comprise two gates: the *reset gate* and the *update gate*. [1, p. 412]

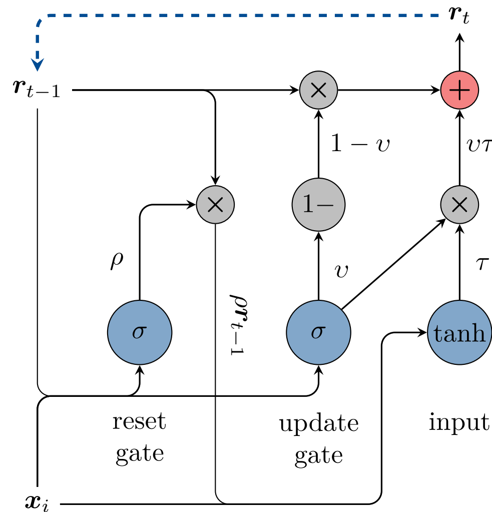


Figure 3.29: Structure of a GRU cell

Figure 3.29 illustrates the structure of a GRU cell. The inputs $\boldsymbol{x}_i$ and $\boldsymbol{r}_{t-1}$ are again fed to the gates, resulting in

$$\rho = \sigma\left(\boldsymbol{w}_\rho\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\rho\right) \tag{3.4.16}$$

and

$$\upsilon = \sigma\left(\boldsymbol{w}_\upsilon\left(\boldsymbol{x}_i, \boldsymbol{r}_{t-1}\right) + b_\upsilon\right). \tag{3.4.17}$$

Compared to LSTMs, we do not proceed in the same way with the input node: Before we can compute τ, the reset gate's result ρ has to be determined. Instead of feeding $\boldsymbol{r}_{t-1}$ to the input node, it is first multiplied by ρ, leading to

$$\tau = \tanh\left(\boldsymbol{w}_\tau\left(\boldsymbol{x}_i, \rho \odot \boldsymbol{r}_{t-1}\right) + b_\rho\right) \tag{3.4.18}$$

to finally determine τ. Thus, the reset gate has the capability of completely resetting the influence of the previous hidden state $\boldsymbol{r}_{t-1}$ on the input τ. The input is further multiplied with the update gate's output υ. Hence, the update gate influences the input value as well: If $\upsilon = 0$, the input value τ is completely erased.

The update gate is also connected the previous hidden state. Before $\boldsymbol{r}_{t-1}$ is passed on to compute $\boldsymbol{r}_t$, the value is multiplied by $(1-\upsilon)$. Thus, the influence of $\boldsymbol{r}_{t-1}$ is inverse to the influence of the input τ to the new hidden state: The update gate determines by how much $\boldsymbol{r}_t$ is updated with the input τ or kept at the old hidden state $\boldsymbol{r}_{t-1}$. In the case of $\upsilon = 0.5$, both τ and $\boldsymbol{r}_{t-1}$ equally contribute to $\boldsymbol{r}_t$ with

$$\boldsymbol{r}_t = \boldsymbol{r}_{t-1}(1-\upsilon) \odot \omega + \upsilon \odot \tau. \tag{3.4.19}$$

The result is then passed on the next GRU cell, illustrated by the blue dashed line in fig. 3.29. [1, p. 293]

Since GRUs lack the cell state and a third gate, we save vector operations when processing a GRU cell compared to an LSTM cell, which in turn leads to smaller computation times for GRUs. Depending on the ML problem, LSTMs might perform better than GRUs. However, they have both shown efficient in ML applications. [31, p. 412]

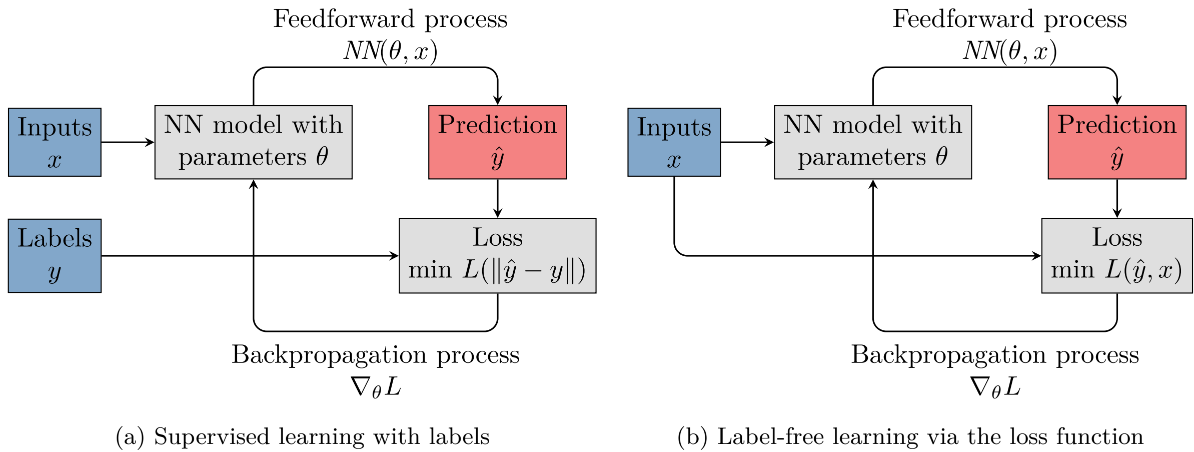
3.5 Label-Free Learning via the Loss Function

In the previous sections we have analyzed the process of machine learning for the *supervised learning* approach, which means that we need labeled input data to compute the loss for parameter adjustment [31, p. 105]. However, sourcing the labels is especially resource-expensive or not possible at all for problems in NP, like the HFS problem. Therefore, we will analyze a new approach that allows us to circumvent the creation of labels by exploiting the loss function itself to adapt the model's parameters.

We will first explain the general concept of this approach and continue with a review of the literature available on this topic. Afterwards, we will analyze how the approach can be classified in the context of machine learning.

General Concept

In fig. 3.30a we see an illustration of the procedures during supervised learning: With the feedforward process we generate predictions $\hat{y}$ by feeding the inputs x to our NN model with the parameters θ. To adjust the parameters during the backpropagation process, we compute a loss that assesses the distance between the predictions $\hat{y}$ and the corresponding labels y. It is the overall goal to *minimize* this loss, i.e. to decrease the distance between the predictions and labels.



(a) Supervised learning with labels (b) Label-free learning via the loss function

Figure 3.30: Schemata of supervised learning with labels and label-free learning

A beneficial characteristic of NP-hard problems is that we can typically compute the quality of a given solution via the minimization function, e.g. we can express the quality of a HFS schedule with the objective function, makespan or flowtime. We can then compare the objective values for different solutions, and the lower the value the better. This feature can be used to derive a different approach to compute the gradient in an NN model: In fig. 3.30b, we have removed the labels from the learning process. Instead, the inputs x are passed on to the loss function, where the predictions $\hat{y}$ are combined with the inputs to compute the *objective function* we want to minimize. For the HFS problem the predictions could comprise a (binary) assignment matrix, and the objective function loss is computed with

$$\min C_{\max}(\hat{y} \odot x) = \min C_{\max}(\widehat{A} \odot P), \tag{3.5.1}$$

where $\widehat{A}$ is our predicted assignment matrix and P is our processing times matrix. With each epoch in the learning process, the overall loss, and thus the makespan, is minimized. This way, we can avoid the creation of labels and nevertheless train the model.

However, we cannot use this approach for all supervised learning problems: Often, there is no underlying mathematical dependency available before training. This is in particular applies to non-deterministic processes, e.g. involving human interactions. For NP-hard problems, on the other hand, we can typically derive a mathematical model and thus develop a loss function that makes use of it.

Literature Review

There is a paucity of literature on this topic since only little scientific research has been done in the area of label-free learning via the loss function. To the best of our knowledge, as of now, the approach has merely been used as support or supplement for other ML techniques.

One example using the approach are *physics-informed neural networks* (PINNs), introduced by Raissi et. al. [59]: A neural network is employed to solve a differential equation (e.g. Navier-Stokes) and the loss is determined by evaluating the differential operators. In [80] the approach was successfully adapted to improve the trajectory of magnetic resonance imaging, speeding up the process and enhancing the image quality.

A similar approach, the *physics guided neural network* (PGNN), was introduced by Karpatne et. al. [42]. In their work, predictions violating physical constraints are penalized by the loss function. An analysis of PGNNs is presented by Wang et. al. [79], who conclude that PGNNs have many areas of application, e.g. the acceleration of data simulation, solving complex partial differential equations or searching for physical laws. The authors state that PGNNs are easy to use and implement; a downside is that the constraints included in the loss functions are *soft constraints*, i.e. there is no guarantee that the constraints are strictly enforced with the loss. The integration of constraints inside the loss function, as well as within the NN's architecture, is also utilized in [7]. Stewart and Ermon [72] predicted movements of objects in computer vision by creating a loss function including domain knowledge and laws of physics. In [20] the authors introduce a method to approximate variational problems with deep learning neural networks whose parameters are mapping the respective function.

Aragon-Calvo [4] created physics-aware neural networks using a *semantic autoencoder*, which encodes input images of galaxies to a set of semantic physical parameters. Thus, the parameters act as a reduced *representation* of the input. Afterwards, the parameters are decoded again to fit the input image, which is also used as the output label. With this approach, the author can predict the physical parameters in the bottleneck. The output image is not the training goal of the neural network, it serves merely as a comparison to the input image and to predict the loss.

Besides approaches modeling physical constraints, there are a few other applications resembling label-free learning with the loss function in the literature. Garg et. al. [27] followed a similar approach as Aragon-Calvo [4] to estimate the depth of images, i.e. the distance of the image pixels to the camera: They encode and decode an input image, which is, in a modified state, used as the target. Hence, there is no need for the creation of labels. Milan et. al. [51] address the problem of label creation for NP-hard problems, which is often computationally expensive: They create approximate solutions as labels and improve the training process by including the *objective costs* of the problem via the loss function.

Classification of the Method in Machine Learning

Typically, ML techniques can be assigned to the categories of *supervised* or *unsupervised* learning [31, p. 104]. Hence, we want to provide an overview of how the approach of learning via the loss function can be classified. Note that there is a third machine learning technique that falls in between supervised and unsupervised learning – *reinforcement learning*. It follows a completely different approach: There is no input data set, instead the model gets into different states by *Markov decisions*. The states are rated by an *agent* and the system can remember certain states and go back to better valued states later in the process. [31, p. 106][5, p. 51] Here, we will not further consider this category of machine learning.

In this chapter we have primarily analyzed classical supervised learning. A simple definition of this method concerns the labels: With supervised learning techniques, we are always given a training data set that is *labeled*, i.e. the correct solution to this data was determined beforehand and thus, we can compute the loss as a distance between labels y and predictions $\hat{y}$ [35, p. 485][28][31, p. 105]. Since our approach clearly states that it is *label-free*, it cannot be accounted for by supervised learning from that perspective. However, even in the course of the simple label definitions, a broader definition of supervised learning is given: An "instructor" or a "teacher" guides the learning process and provides feedback or a direction to the learning algorithm [69, p. 4][31, p. 105][35, p. 29]. This definition fits our approach again, since the loss function evaluates the objective function of the prediction and can thus provide a gradient for learning.

Conversely, a simple definition of unsupervised learning is that the data is not labeled [40, p. 133]. Again, broader definitions of unsupervised learning are given in literature: The learning process is not given any "feedback from its environment" [28] or "degree-of-error for each observation" [35, p. 438] and there is "no measurements of the outcome" [35, p. 2] available. This is certainly not true for the proposed approach since the loss function provides an evaluation of the results. Moreover, unsupervised learning often aims at exploring hidden features in a data set [40, p. 133], which is not the goal of learning via the loss function, where the objective function is clearly defined.

There are two more ML approaches whose names suggest an application that is neither supervised nor unsupervised. *Semi-supervised learning* is a combination of learning with labeled and unlabeled data [44]. Typically, this approach is used to support learning with unsupervised techniques when there is only little labeled data available [28]. Since our approach comprises only unlabeled data, we do not make use of this technique. In *self-supervised learning* the weights of an NN are initialized with a modified subset of the actual data. The learning task itself can be supervised or unsupervised and continuous with the pretrained network. [85] This concept does not describe our approach either.

Eventually, as stated in [31, p. 105], "unsupervised learning and supervised learning are not formally defined terms". This explains the differing understandings authors convey when defining the two terms. In our opinion, label-free learning via the loss function is *supervised learning*: There are indeed no labels, but we can still assess the *quality* of a solution via the objective function. Thereby, a gradient for parameter optimization is provided. Thus, the labels are substituted by the loss function. Furthermore, the aim of the learning process is clear (minimizing the objective function), whereas in unsupervised learning, unknown features are explored.

Chapter 4

Machine Learning for Combinatorial Problems

In the following, we will present an approach that solves the HFS problem with machine learning.

As we will conclude from our literature review in section 4.1, there are two main challenges that arise when solving such combinatorial problems:

1. the *discreteness* of combinatorial problems and
2. the *creation of labels* for supervised ML, in particular for NP-hard combinatorial problems.

To address the issue of discrete (binary) results, we suggest a relaxation approach for the assignments (cf. section 4.3.1) and the permutations (cf. section 4.4.1) of HFS problems. To circumvent the creation of labels, we make use of the label-free learning approach as indicated by section 3.5. The machine learning models and their loss functions play pivotal roles in the solution of both sub-problems.

4.1 Literature review

We refer to the literature reviews of Zacharias [83] and Bengio et al. [6] for machine learning for scheduling problems and machine learning for combinatorial problems, respectively. Both publications provide an extensive overview of publications and developments in the respective fields.

Bengio et al. [6] concentrate in their review specifically on machine learning applications for *combinatorial problems*. They point out that this problem type is typically NP-hard, which makes the computation of results with traditional approximation algorithms resource-expensive. Thus, ML approaches seem like a good choice for combinatorial optimization (CO) problems. They divide the overall *algorithm structure* of ML approaches solving CO problems into three groups:

1. *end-to-end learning*: the ML model is applied to the input problem to directly produce a solution,
2. *learning to configure algorithms*: the ML model is trained to learn the configuration (e.g. hyperparameters) of an operations research model (e.g. MILP),
3. *machine learning alongside optimization algorithms*: the ML model is frequently called inside an optimization algorithm to support decision making for the algorithm's path executions.

As we are aspiring an *end-to-end learning* approach in this thesis, we point out two publications in this field. To produce immediate results, the ML model has to generate *discrete predictions* or predictions that can easily be mapped to a discrete formulation. This is problematic since the elements of ML models typically have to be differentiable (cf. section 3.3.1), thus a *relaxation* is needed. In [21], Emami and Ranka suggest an algorithm that provides a *doubly stochastic matrix* as a representation for a permutation, taking an input matrix with values different from 0 or 1 as an input (see section 4.4.1 for further elaboration). In [77], Vinyals et al. introduce *pointer networks*: To describe a possible output sequence, the network points to members of the input sequence with a probability distribution, generated with the *softmax* activation function. The input with the highest probability is chosen next in sequence by the pointer. Thus, with this approach it is possible to approximate the discrete values of a permutation vector.

The other two algorithm categories described by Bengio et al. comprise completely different use cases of ML for combinatorial problems. One approach is given in [48], where machine learning is used to predict the MILP model for a combinatorial problem itself, e.g. the model constraints. This approach falls under category 2 in the list above.

Furthermore, Bengio et al. point out persisting challenges concerning ML for combinatorial optimization. One aspect is the *optimality* of the solution, which cannot always be proven for NP-hard problems. The previously described articles by Emami and Ranka [21] and Vinyals et al. [77] provide solutions to reach feasibility for the permutation case. Nonetheless, the implementation of feasibility constraints is problematic given the fact that combinatorial problems are of discrete nature. Another challenge for combinatorial problems is to create a *suitable machine learning model* that is capable of learning the underlying data structure. For instance, CNNs are typically the right choice to analyze image data, whereas for combinatorial problems, there is no best practice yet. Additionally, *scaling* the learned model can be a challenge for combinatorial problems, depending on the model architecture used. The last mentioned challenge is *data generation*: Collecting data of a real process or generating data that represents realistic conditions can be difficult. It must be ensured that the generalization capability of the model can be measured.

Summarized, Bengio et al. see great potential in solving combinatorial problems with machine learning. In particular, they point out that the end-to-end learning approach could benefit from already existing optimization algorithms, and that a combination could leverage the performance of the ML approach. Note that in the review of Bengio et al. the approach of label-free learning was not mentioned, which again hints to the lack of research for this approach.

In her literature review [83], Zacharias focuses on machine learning for scheduling problems. She states that proposed algorithms are often delimited to a certain problem configuration and lack flexibility: In the case of HFS scheduling, most algorithms are only applicable to small instances with a certain amount of stages and machines. Typically, ML techniques are not implemented as an end-to-end solution, but

as a support to determine the best *dispatching rule*[1] considering current system parameters – heuristics providing the assignments to machines or even the full schedule are comparatively rare.

The majority of the analyzed approaches (60 publications) concentrates on the makespan objective, whereas only 10 of the publications focus on the flowtime (total completion time) objective. Furthermore, most approaches consider identical machines; uniform or unrelated machine settings are so far neglected in research. This observation is also confirmed by Ruiz and Vázquez-Rodríguez [68]: Uniform machine environments are less complex and thus, solutions for problems referring to this machine setting can be derived more easily [26, p. 53].

In her dissertation, Zacharias proposes the novel *MAJ* (Modular Assignment of Jobs) algorithm that combines ML and classic heuristics to solve the HFS problem efficiently. Therefore, the problem is divided into sub-problems, which are further divided and solved independently, following a *divide-et-impera* (DEI, cf. section 4.2) approach: With supervised ML methods, Zacharias predicts the objective value (makespan or flowtime) of “job packages”, i.e. subsets of all input jobs. Due to the smaller size of the job packages, they can easily be permuted (e.g. $4! = 24$ for $J = 4$) and the permutation promising the best objective value is selected. The assignments to the machines are determined with a *local search*[2] algorithm. Afterwards, the job packages are merged back together to solve the main problem. Thus, we obtain a hybrid approach (category 3 of the solution classes for CO problems by Bengio et al.) that approximates the optimal solutions efficiently. The sub-division of the problem according to DEI as well as the ML part lead to a fast return of results.

Concerning the reviewed publications, most ML approaches rely on neural networks, followed by hybrid approaches, e.g. in combination with linear programming, and recurrent neural networks. Reinforcement learning (RL) is on the last place with only two publications considering this approach among around 100 reviewed papers until 2019. Addressing more recent publications, we can observe a trend towards RL, both for permutation flow shops [10][60][25] and the specific case of hybrid flow shops [24][55]. This development could arise from the fact that reinforcement learning has become more popular in the last years [57, p. v] and thus, the RL research has progressed. Regarding combinatorial problems, RL brings the advantage that no labeling of the input data is required. However, it comprises its own challenges, like formulating an RL model that outputs a final sequence directly. Due to this, hybrid approaches combining RL with classic heuristics [25] or neural networks [60] were developed, or the permutation is obtained with RL iteratively [10]. Another downside is that RL approaches can be computationally expensive [57, p. 91] or slow to learn [57, p. 135] and often require a complicated setup [1, p. 395], especially for complex problems like the HFS.

Overall, Zacharias discovered a paucity of flexible approaches solving scheduling problems. The discrete nature of the problem represents a barrier for the application of ML to combinatorial problems like the HFS. She assumes that the trend towards heuristics combining ML and other algorithms, like linear programming or genetic algorithms, stems from the discretization problem. Another challenge is the generation of labels for the often NP-hard scheduling problems. As a result, supervised approaches often rely on labels generated by simulation [58].

---

[1] Dispatching rules are simple directives to sort jobs waiting to be processed quickly, e.g. *FIFO* or *earliest-due-date*. [26, p. 165]

[2] Local search algorithms improve a given starting solution by searching for better results in the neighboring solution space. [56, p. 382]

The considered literature reviewers have in common that they see potential in ML for combinatorial problems. However, there are some obstacles that have be overcome to successfully implement machine learning in this domain: the discretization of the model output is one of the main issues mentioned, as well as the creation of reliable labels for the hard-to-compute and often NP-hard combinatorial problems. These issues are mirrored in the trends of publications leaning towards ML algorithms avoiding them. For instance, end-to-end solutions are often displaced by hybrid forms that support ML algorithms in decision making: The crucial steps facing discretization are often replaced with other algorithms like MILP. Furthermore, reinforcement learning has recently entered the field to provide solutions that avoid the creation of labels, making them a promising approach for NP-hard problems. Nonetheless, RL approaches have their downsides, considering the complicated setup and resource-intensive execution.

It is important to note that, as of our best knowledge, there are no publications available that process combinatorial problems with label-free learning via the loss function, as we will present it in the following sections; neither for scheduling problems in particular, nor for combinatorial problems in general.

4.2 General Approach

We adapt parts of the approach presented in [83] and divide the HFS problem into two subproblems: on the one hand the *permutation* learning, and on the other hand the learning of the *assignments*. However, we are not directly following a *divide-et-impera* (DEI) approach as proposed in [83]: In the presented approach, the problem is not further subdivided into independently solvable sub-problems, which is an essential step of the DEI method [15, p. 65].

Solving both subproblems completely independently is not reasonable, since the optimal assignments depend on the permutation of the problem matrix, i.e. the optimal assignments for one permutation do not necessarily lead to the optimal solution for another permutation. This circumstance is illustrated in the following example: We have given a problem matrix P and two different permutations of P with $\sigma_1 = (1, 2, 3, 4)$ and $\sigma_2 = (2, 3, 4, 1)$. We compare the optimal assignments of the permutations. Elementwise multiplication of the permuted P with the respective assignments A yields

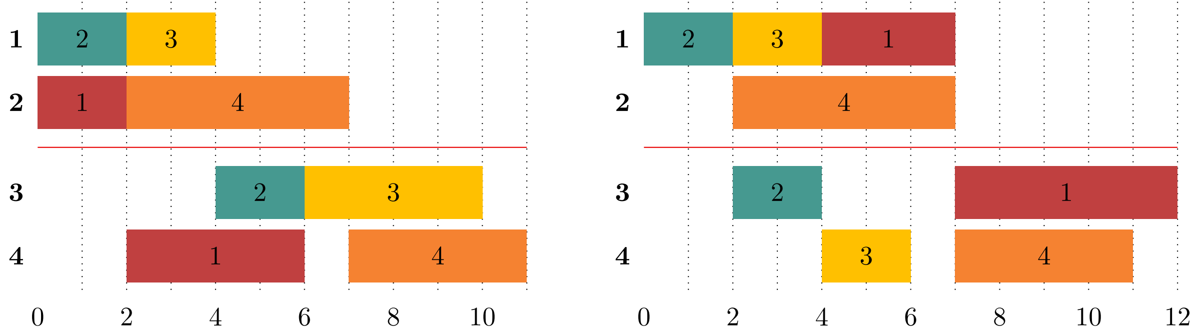
$$P_{A_1} = P_{(1,2,3,4)} \odot A_1 = \begin{pmatrix} 3 & 2 & 5 & 4 \\ 2 & 3 & 2 & 4 \\ 2 & 5 & 4 & 2 \\ 4 & 5 & 6 & 4 \end{pmatrix} \odot \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 0 & 4 \\ 2 & 0 & 2 & 0 \\ 2 & 0 & 4 & 0 \\ 0 & 5 & 0 & 4 \end{pmatrix} \tag{4.2.1}$$

for the first permutation with $\sigma_1 = (1, 2, 3, 4)$ and

$$P_{A_2} = P_{(2,3,4,1)} \odot A_2 = \begin{pmatrix} 2 & 3 & 2 & 4 \\ 2 & 5 & 4 & 2 \\ 4 & 5 & 6 & 4 \\ 3 & 2 & 5 & 4 \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 2 & 0 \\ 2 & 0 & 0 & 2 \\ 0 & 5 & 0 & 4 \\ 3 & 0 & 5 & 0 \end{pmatrix} \tag{4.2.2}$$

for the second permutation with $\sigma_2 = (2, 3, 4, 1)$. The respective schedules can be viewed in fig. 4.1, where we identify machine switches for job 1 (stage 1 and 2) and job 3 (stage 2). For instance, if job 1 kept the

assignments of A_1 in fig. 4.1b, its processing would take place after job 4 in stage 2 and therefore, the total processing time would increase.



(a) Schedule of the optimal assignments for eq. (4.2.1), $\sigma = (1, 2, 3, 4)$

(b) Schedule of the optimal assignments for eq. (4.2.2), $\sigma = (2, 3, 4, 1)$

Figure 4.1: Schedules of two optimal assignments for different permutations of the same problem

As a consequence, to evaluate the quality of a predicted permutation with the objective value, e.g. $C_{\max}$, we need to know the (optimal) assignments for the respective permuted matrix. This fact must be considered for solution strategies of the HFS problem.

Training Process

Figure 4.2 and fig. 4.3 illustrate the procedures for *training* and *inference* of the proposed approach. During training, we create and train a neural network machine learning model for each subproblem, i.e. to predict an assignment matrix $\widehat{A}$ and a permutation matrix $\widehat{O}$. The learned models are utilized during prediction, combined with additional rectification steps. A detailed description of the training processes is given in section 4.3 and section 4.4.

To train the model NN_A for assignment prediction, we use batches of size B_A comprising problem matrices P with an *arbitrary* number of jobs J as training data. When we feed an input matrix P to NN_A, we assume that P is already in the desired order, since NN_A cannot infer a sequence, only the assignments. The result of the feedforward-process is a *relaxed* assignment matrix $\widehat{A}_R$, which is passed on to the loss function.[3] Since we follow a label-free approach, inside the loss function, the quality of the assignments is evaluated using both $\widehat{A}_R$ and the input matrix P: The objective value function f_C, e.g. to compute $C_{\max}$, takes the elementwise product of P and $\widehat{A}_R$ as input and returns the respective objective value. Additionally, a penalty function p is applied to the network's output $\widehat{A}_R$ to support the generation of viable (binary) assignments (cf. section 4.3.2).

For sequence inference, we train the model NN_σ with batches of size B_σ comprising problem matrices P as training data. Contrary to the assignment model training, the number of jobs for P is fixed, e.g. $J = 6$. The matrix size limitation is due to the selected layers for the model, which cannot process inputs of varying size (cf. section 4.4.4). The result of the feedforward-process is a *relaxed* permutation matrix

[3]The assignment matrix is not necessarily binary since we have to consider a relaxed approach to ensure differentiability (cf. section 4.3).

$\widehat{O}_R$, which is passed on to the loss function.[4] As mentioned before, we can only determine the quality of a sequence with knowledge of the respective assignments. Therefore, NN_A is integrated into the loss function of P_σ: We predict the assignments for the permuted input $P_\sigma = P \cdot \widehat{O}_R$ with $\widehat{A} = \text{rectify}_A(NN_A(P_\sigma))$, and compute the objective function value $f_C(P_\sigma \odot \widehat{A})$ to be used for the loss function. Thus, it is necessary to train a reliable assignment model for the respective machine configuration *before* we can train the permutation model. If the assignment model returns weak results, then the permutation model adapts the errors and produces faulty results itself. Similar to the loss of the assignment model, a penalty term p is applied to the output, i.e. $\widehat{O}_R$, to push the outputs towards feasible results.

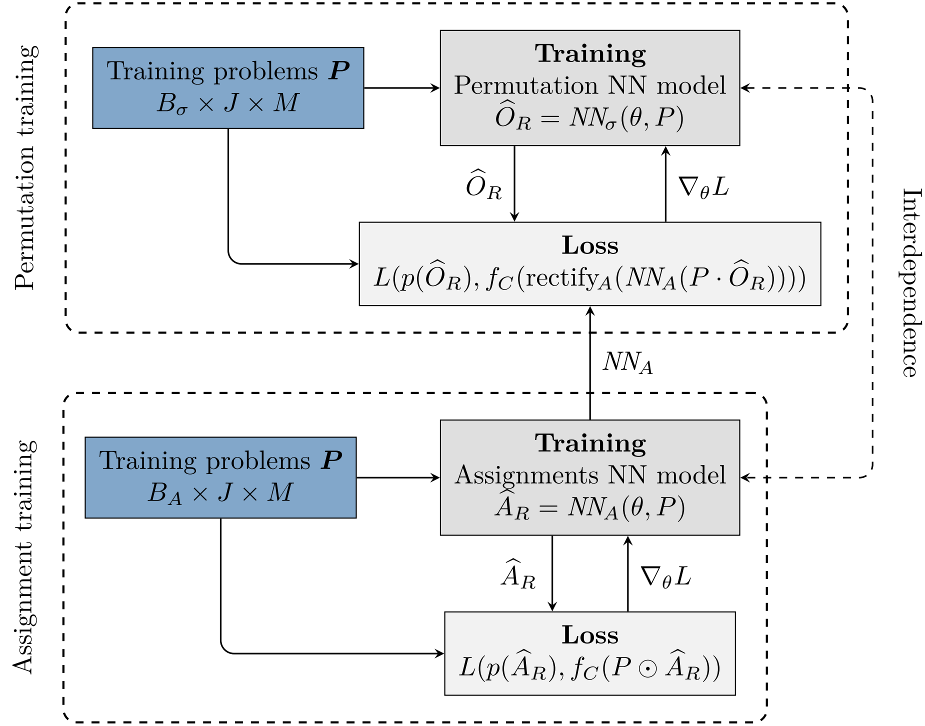


Figure 4.2: General training process of the proposed approach

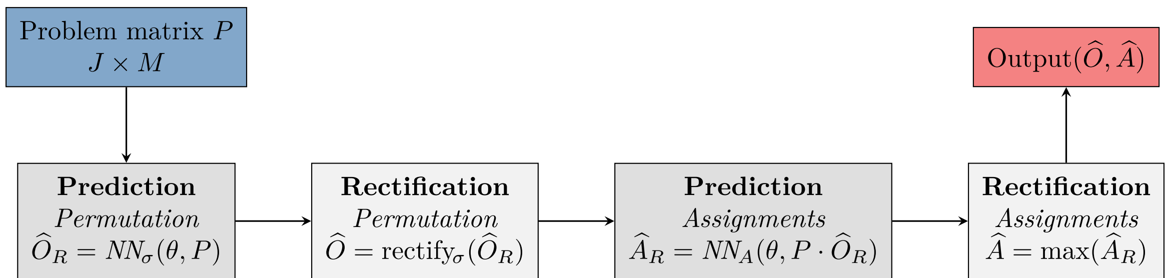


Figure 4.3: General inference process of the proposed approach

[4]Analogous to $\widehat{A}$, the permutation matrix $\widehat{O}$ is not necessarily binary to ensure differentiability.

Inference Process

The procedure of predicting a sequence and the respective assignments for an input matrix P are illustrated in fig. 4.3. The input matrix P, for which we want to predict the permutation and the assignments, is fed to the trained permutation NN model, which returns a (relaxed) permutation prediction $\widehat{O}_R$. Since the permutation matrix might not yet be feasible, the next step involves the *permutation rectification* algorithm (cf. section 4.4.1) which produces a valid, binary permutation matrix for any given relaxed permutation matrix $\widehat{O} = \text{rectify}_\sigma(\widehat{O}_R)$. The input P is then multiplied by the admissible permutation matrix and used as input to the assignment model NN_A, which returns a (relaxed) predicted assignments matrix $\widehat{A}_R$. Similar to the relaxed permutation matrix, the assignment matrix is rectified with $\widehat{A} = \text{rectify}_A(\widehat{A}_R)$ to ensure a feasible result, cf. section 4.3.4. Combining the permutation matrix $\widehat{O}$ and the processing times P with the assignment matrix $\widehat{A}$ defines a full schedule.

Both prediction models can be used *independently* of one another as a stand-alone solution: We can predict a permutation matrix without an assignment model, and we can predict assignments without executing a permutation model beforehand. However, the model parameters of NN_σ are influenced by the assignment model used during training inside the loss function. Thus, although the loss function does not play a role when predicting, the permutation model always assumes certain assignments when producing a permutation.

The assignment model, on the other hand, always assumes the permutation conveyed by the current input matrix P, and we can easily predict the assignments for the given sequence. We are not forced to preprocess the input with NN_{PRMU}, and thus, we can use any other algorithm, for example a simple dispatching rule, to order the jobs. Likewise, we do not have to reorder the input at all, e.g. if we want to process the matrix with a *FIFO* approach.

In the following sections we will go into further detail concerning the structure of the neural networks NN_A and NN_σ, and the mechanisms involved in the training process of the respective models.

4.3 Subproblem 1: Learning the Assignments

As discussed in section 4.2, to predict a schedule with machine learning, the assignment model must be determined first. Hence, we will start to elaborate on the neural network model designed for the prediction of the assignments. For the pure assignment problem, we assume that the order in which the jobs need to be processed is fixed. We aim to find the optimal assignment strategy, i.e. the assignment $A \in \mathcal{A}$ such that the resulting objective function – either the makespan or the flowtime – is minimal for the given underlying permutation.

For the relaxed problem, the neural network itself can then be described as a mapping

$$F\colon \mathbb{R}_+^{J\times M} \to \mathcal{A}_\mathrm{R}\,, \quad P \mapsto \widehat{A}_R = F(P)\,,$$

with A_R as defined in eq. (4.3.2), where $\widehat{A}_R$ denotes the predicted relaxed assignment for the processing times P. In a final step, the matrix $\widehat{A}_R$ is then rectified (cf. section 4.3.4) in order to obtain a classical admissible schedule $A \in \mathcal{A}$.

4.3.1 Assignment Relaxation

The equalities and inequalities in section 2.1 describing the HFS problem rely on the *discreteness* of the assignments: All but one of the values of the machine assignment matrix A_{jm} are zero for each job on any given stage. However, as indicated in section 4.1, this discreteness is one of the major obstacles for the application of machine learning techniques, since any loss function defined on a discrete set is not directly suitable for gradient-based optimization procedures. As discussed in section 3.3.1, the *argmax* activation function would fulfill the task of setting only one value to 1. However, this activation function is not applicable because it is not *differentiable*. In the following, we will therefore consider a relaxed approach by allowing *fractional* assignments, which efficiently extends the loss function to the resulting larger domain of admissible matrices.

Recall from eq. (2.1.1) that for the classical hybrid flow shop problem, the set of admissible assignments is given by

$$\mathcal{A} = \Big\{ A \in \{0,1\}^{J \times M} \ \Big| \ \sum_{m \in \mathcal{M}_i} A_{jm} = 1 \ \text{ for all } j \in \{1,\dots,J\},\, i \in \{1,\dots,I\} \Big\}, \tag{4.3.1}$$

to ensure that each job is assigned to exactly one machine on each stage. In the *assignment-relaxed* hybrid flow shop model, we allow for a *split* (or *fractional*) assignment of a job to multiple machines on a single stage in the form of a convex combination, which is described by the extended set

$$\mathcal{A}_{\mathrm{R}} = \Big\{ A \in [0,1]^{J \times M} \ \Big| \ \sum_{m \in \mathcal{M}_i} A_{jm} = 1 \ \text{ for all } j \in \{1,\dots,J\},\, i \in \{1,\dots,I\} \Big\} \tag{4.3.2}$$

of admissible assignment matrices.

The notion of jobs being processed simultaneously on multiple machines has previously been discussed in the literature, both as a relaxation technique [81] and as an optimization problem in its own right [16]. Although our focus lies on the former approach, i.e. on viewing relaxation as a mathematical tool for obtaining classical solutions, it is nevertheless important to distinguish between different *interpretations* of fractional assignments in order to determine how the objective function can be reasonably extended to $\mathcal{A}_{\mathrm{R}}$.

In particular, we will analyze the *job splitting* and *machine splitting* interpretations using the following example: For $J = 3$, $M = 5$ and $I = 2$ with $\mathcal{M}_1 = \{1,2,3\}$ and $\mathcal{M}_2 = \{4,5\}$, we consider the processing times P and relaxed assignment A with

$$P = \begin{pmatrix} 3.0 & 6.0 & 1.0 & 5.0 & 3.5 \\ 7.0 & 2.0 & 8.4 & 1.0 & 4.0 \\ 5.0 & 6.0 & 4.0 & 2.0 & 1.0 \end{pmatrix} \qquad \text{and} \qquad A = \begin{pmatrix} 1 & 0 & 0 & 0.3 & 0.7 \\ 0 & 1 & 0 & 0.7 & 0.3 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

For the respective starting times S

$$S_{job\ split} = \begin{pmatrix} 0.0 & 3.0 \\ 1.0 & 5.45 \\ 1.0 & 6.65 \end{pmatrix} \qquad \text{and} \qquad S_{machine\ split} = \begin{pmatrix} 0.0 & 3.0 \\ 1.0 & 5.556 \\ 1.0 & 6.846 \end{pmatrix},$$

we obtain the schedules shown in fig. 4.6 for the interpretation of the relaxation as job splitting and machine splitting.

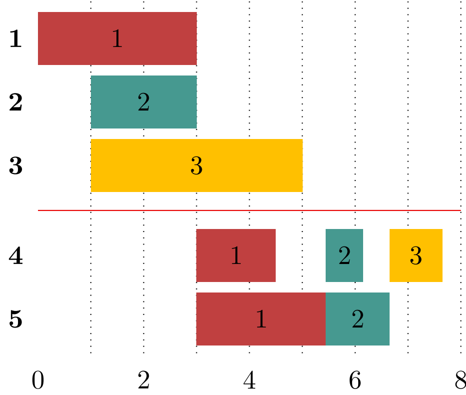


Figure 4.4: Schedule with job splitting

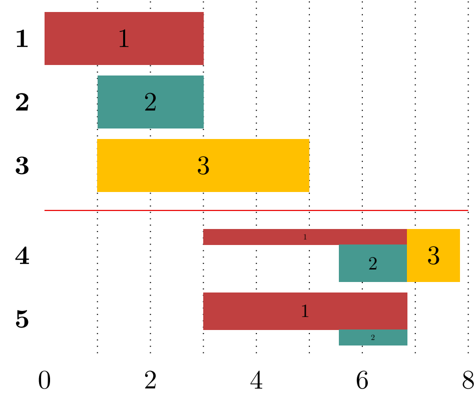


Figure 4.5: Schedule with machine splitting

Figure 4.6: Examples of job splitting and machine splitting

Job Splitting

For the job splitting interpretation, we consider an assignment of a job j to multiple machines $m_1, \ldots, m_n$ on a stage i as a split of the job itself into multiple parts. In this case, the weight factor $A_{jm} \in [0, 1]$ determines the fraction of the job which is being processed by machine m. Then the processing time for this part is naturally given by $A_{jm} \cdot P_{jm}$. During this time, machine m is considered to be fully occupied. The total processing time Δt for job j on stage i is then given by

$$\Delta t = \max\{A_{jm} \cdot P_{jm} \,|\, m \in \mathcal{M}_i\}\,, \tag{4.3.3}$$

i.e. the job is finished on the stage once all parts have been processed. In fig. 4.4 we can observe the effects of job splitting on stage 2 for jobs 1 and 2: Both jobs are split up between machines 4 and 5. The processing time on a stage for split jobs is determined by the maximum processing time, i.e. job 1 is considered finished at $t = 5.45$. Due to the constraint of ordered processing, job 2 is only allowed to start after that. Likewise, job 3 must not start on machine 4 before job 2 has been fully processed on machine 5, which results in gap of processing on machine 4.

However, to obtain a feasible solution, job splitting must be reduced as much as possible. Therefore, we introduce *penalties*, which is further explained in section 4.3.2.

Machine Splitting

Alternatively, we can interpret a fractional assignment as a partial occupation of a particular machine such that $A_{jm} \in [0, 1]$ represents the fraction of machine m which is dedicated to the processing of job j. In this case, the job itself is considered to be undivided, with the required processing time Δt on stage i given by the weighted harmonic mean[5]

$$\Delta t = \frac{\sum\limits_{m \in \mathcal{M}_i} A_{jm}}{\sum\limits_{m \in \mathcal{M}_i} \frac{A_{jm}}{P_{jm}}} = \frac{1}{\sum\limits_{m \in \mathcal{M}_i} \frac{A_{jm}}{P_{jm}}} . \tag{4.3.4}$$

In fig. 4.5 we can see that machines 4 and 5 are not completely occupied by the respective jobs: job 1 takes up 30% of machine 4, which means that there is still a capacity of 70% left for job 2 to run *simultaneously* on this particular machine. Likewise, job 1 and 2 share the capacity of machine 5. However, job 2 may not finish before job 1 has finished and thus cannot immediately start on stage 2 after finishing on stage 1. Since the notion of machine splitting would introduce additional rectification discontinuities for the purely assignment-relaxed hybrid flow shop (cf. section 4.3.4), we will mainly focus on the case of job splitting in the following.

4.3.2 Loss Function

The loss function is a crucial part of the proposed approach: It contributes significantly to the training process, as it determines the gradients for backpropagation. The loss function has to represent the objective functions and – supported by the assignment relaxation – it has to stay *differentiable* at the same time. Thus, we have to extend the definition of the two main objective functions C_{max} and C_{sum} from section 2.1.3 to the assignment relaxed case. We follow the same procedure as described in section 2.1. However, referring to the job-splitting interpretation of non-binary assignments, the actual processing time on each stage needs to be modified according to eq. (4.3.3). Furthermore, a single job might occupy multiple machines on a single stage for different time intervals, which needs to be taken into account for the scheduling as well.

Once the schedule is set up, the relaxed makespan $C^{\text{R}}_{\text{max}}$ and the relaxed flowtime $C^{\text{R}}_{\text{sum}}$ are easily computed, respectively, via the maximum and the sum of the end-of-processing times over all jobs on the final stage. The whole procedure is described in algorithm 1 in tensorial form, as is required by many machine learning frameworks. In the practical implementation, the input is processed *batch-wise*, i.e. the same steps are performed on each input matrix in a batch concurrently: This speeds up the computation time significantly compared to matrix-wise processing. However, this implies some additional lines of code to manage the batches. To enhance readability, these lines were omitted in algorithm 1. As far as possible, the variables were named according to the formerly introduced notation of the hybrid flow shop. Additionally, the comments hint to the implementation of each of the 5 HFS rules (see section 2.1), where applicable. However, since the inputs have to be processed in tensorial form, we cannot completely adapt the linear programming form of section 2.1.

[5] The use of the harmonic mean is due to the interpretation of A_{jm} as the ratio of the machine's performance p_{jm}, which is related to the processing time via $P_{jm} = 1/p_{jm}$.

The output of the forward propagation, which is in turn the input to the loss function, is an assignment-relaxed schedule A_R of size $J \times M$ that follows the restrictions of eq. (4.3.2). The layout of the underlying neural network is described in detail in section 4.3.3. The assignment-relaxed schedule A_R as well as the plain matrix of processing times, which is also the input to our neural network (cf. fig. 4.2), are the inputs to our loss function. Furthermore, we introduce the vector variable penalty weights $w^p \in \mathrm{R}_+^3$: the final loss L is composed of three values – the makespan, the flowtime and an additional penalty term. With w^p we can adjust the influence of those values on the final loss.

In fact, the output of algorithm 1 equals the value of an objective function (makespan or flowtime) only if the penalties are set accordingly: For any admissible assignment $A \in \mathcal{A} \subset \mathcal{A}_\mathrm{R}$, the relaxed makespan is equal to the classical makespan, if the first value of w^p is set to 1. We will further elaborate on the penalty function and its benefit for training in the next section.

To compute the relaxed assignment loss, we loop through all jobs j, stages i and machines on the stages $\mathcal{M}_i$ of the assignment relaxed schedule $P_R = P \odot A_R$. For each of the HFS constraints (2) – (5) we determine a value that ensures that the starting time on the machine accord to said constraints[6]. The final admissible starting time $S_{j,m}$ is then determined by taking the maximum of the 4 restriction values (line 21 in algorithm 1). The respective completion time can then be easily determined by adding the current processing time to the starting time, i.e. $C_{j,m} = S_{j,m} + (P_R)_{j,m}$ (line 22). With the completion times, we can then derive the makespan (line 26) and the flowtime (line 27). The additional penalty term is computed with the function *assignmentMeantimes*, which takes the relaxed assignments A_R and the processing times P as an input. In line 29 we sum all penalty terms and multiply them by the respective penalty weights to obtain the output loss L.

[6]Note that the first constraint (*on each stage, each job is assigned to exactly one machine*) is omitted since the relaxed version of the assignments violates this rule. We will later return to this issue.

Algorithm 1: Relaxed Loss for the Assignments

```
1  AssignmentsRelaxedLoss (P, Â_R)
     inputs : Processing times matrix P of size J × M,
              Predicted relaxed assignment matrix Â_R of size J × M,
              Penalty weights w^p of size 1 × 3
     output: The loss L comprising C^R_max, C^R_sum and a penalty term
2    P_R ← P ⊙ Â_R                              // Assignment-relaxed processing times
3    S_J,M ← 0⃗_{J×M}                           // Starting times matrix of all jobs and machines
4    C_J,M ← 0⃗_{J×M}                           // Completion times matrix of all jobs and machines
5    S_j-1,i ← 0                                // Starting time of previous job on current stage
6    C_shift ← C_(j-1,i)-P[j,m] ← 0              // Admissible completion time for j without overtaking j − 1
7    C^max_m ← 0                                // Maximum processing time of job m
8    foreach j ∈ [0, J − 1]^ℤ do
9      foreach i ∈ [0, I − 1]^ℤ do
10       if j ≠ 0 then
11         C_j-1,i ← max(S_J,M[ j,𝓜_i-1 ])
12         S_j-1,i ← max(S_J,M[ j-1, 𝓜_i ])      // (4) Do not start job j before j − 1 on stage i
13       if i ≠ 0 then
14         C_j,i-1 ← max(C_J,M[j,𝓜_i-1])         // (2) Start job j on i after it is finished on i-1
15       else
16         C_j,i-1 ← 0                          // Set the completion time of j on the previous stage to 0 if i = 0
17       foreach m ∈ 𝓜_i do
18         if j ≠ 0 then
19           C^max_m ← max(C_J,M[:j,m])          // (3) Each machine m processes one job at a time
           // (5) Finish a job on stage i only after all earlier jobs (no overtaking):
20         C_shift ← C_j-1,i − max(P_R[j-1,𝓜_i])
           // Set starting time of job j on machine m according to rules (2)-(5):
21         S_j,m ← max(C_j,i-1, C^max_m, S_j-1,i, C_shift)
22         C_j,m ← S_j,m + P_R[j,m]              // Compute the finishing time of job j on machine m
           // Update starting time and completion time matrices:
23         S_J,M[j,m] ← S_j,m
24         C_J,M[j,m] ← C_j,m
25   C_J ← max(C_J,M[:,𝓜_I])                     // Final completion times of all jobs
     // Computation of the objective functions:
26   C^R_max ← max(C_J)                          // Makespan: maximum of job completion times
27   C^R_sum ← sum(C_J)                          // Flowtime: sum of job completion times
     // Computation of the penalty and final loss L:
28   penalty ← p(Â_R)
29   L = w^p_1*C^R_max + w^p_2*C^R_sum + w^p_3*penalty
30   return L
```

Penalty Functions

In general, optimal solutions to relaxed problems are not admissible for the classical, non-relaxed hybrid flow shop problem; for example, the machine assignment for which the relaxed makespan attains its minimum might require the assignments 0.3 and 0.7, respectively, on two machines of the first stage for the first job, whereas the classical problem is *binary*, i.e. a single machine would need to be selected in the classical problem.[7] While a conversion to the "nearest" admissible solution (cf. section 4.3.4) is always possible – in this case by selecting the machine with the highest assignment value – it cannot be ensured that the resulting classical solution is in any sense optimal, especially if the relaxed solution is not "sufficiently close" to any classical one. Therefore, the relaxed objective function needs to be supplemented by additional *penalty terms* which penalize deviations from the set of classically admissible solutions [39].

A straightforward penalty of the split between assignments is given by the function

$$p_3\colon \mathcal{A}_{\mathrm{R}} \times \mathbb{R}_+^{J\times M} \to \mathbb{R}\,, \qquad p_3(A,P) = \sum_{j=1}^{J}\sum_{i=1}^{I} \sum_{\substack{m\in\mathcal{M}_i \\ m\neq \underset{n\in\mathcal{M}_i}{\operatorname{argmax}} A_{jn}}} A_{jm}\cdot P_{jm}\,.$$

For a given assignment A and a processing times matrix P, the term $p_3(A,P)$ represents the sum over all processing times *except* the ones on machines with the maximum assignment value for a given stage-job combination. Therefore, p_3 penalizes all processing on multiple machines and thereby all deviations from a *discrete*, admissible assignment on each stage.

Alternatively, we can consider the penalty

$$p_3\colon \mathcal{A}_{\mathrm{R}} \to \mathbb{R}\,, \quad p_3(A) = \sum_{j=1}^{J}\sum_{i=1}^{I} \sum_{m,n\in\mathcal{M}_i m\neq n} A_{jm}\cdot A_{jn}\,,$$

which does not depend on the processing times P and is differentiable at any $\in \mathcal{A}_{\mathrm{R}}$. For every combination of jobs and stages, the function p_3 additively weighs the split of the job between any combination m,n of machines on the stage. Note that

$$\sum_{m\in\mathcal{M}_i} A_{jm} = 1$$

for every $j \in \{1,\dots,J\}$ and $i \in \{1,\dots,I\}$ due to the choice of $\mathcal{A}_{\mathrm{R}}$ as the domain of definition for p_1, which ensures that $p_1(A) \geq 0$ for all $A \in \mathcal{A}_{\mathrm{R}}$ and that $p_3(A) = 0$ if and only if $A \in D$, i.e. p_3 attains the minimum value 0 exactly at the classically admissible assignments.[8]

In the following, we will us a weighted sum between the objective function itself and the two penalties (line 29 in algorithm 1) as the loss function for training our neural network.

[7] This corresponds to rule (1) of our HFS constraints.

[8] It is also easy to see that p_3 does not attain any other local minima and that $\sum_{m\neq n} A_{jm}\cdot A_{jn}$ is maximal if and only if $A_{jm} = \frac{1}{|\mathcal{M}_i|}$ for all $m \in \mathcal{M}_i$.

4.3.3 Structure of the Neural Network

The NN model's configuration must realize the mapping of the input matrices to their respective assignment matrices, i.e. $P \mapsto \widehat{A}_R$. Ideally, we make use of our knowledge about the input problems' structure to fulfill this task efficiently, i.e. in the sense of deep learning, always under the condition that differentiability must be retained.

The NN model we suggest primarily comprises recurrent layers, since the sequence of the input matrix P is given and thus, we determine the assignments according to this sequence. With RNNs, the sequence information can be extracted to predict the respective assignments. In particular, we use LSTM layers, since these layers yielded slightly more precise results for our approach than GRUs.

In contrast to the NN models considered in chapter 3, the assignment model keeps the rank of the input matrix during the feedforward process, i.e. the output of each layer is a tensor of rank 2. Therefore, we have chosen layer types (dense and LSTM layers) which are capable of processing and returning tensors of rank 2. Additionally, the number of rows is also consistent throughout for each transformation, whereas the number of columns differs. A special feature of LSTM layers is the ability to process inputs of varying size, in particular of varying numbers of rows. That means, in our case, that we can feed input matrices P with different numbers of jobs J to our NN and obtain a prediction *without* training the NN again for different problem sizes. Note that this does only apply to the number of jobs, not the machine configuration.

Figure 4.7 depicts the structure of the NN model. The respective activation functions, as well as the output sizes of the layers, are indicated in the diagram. During training, the input is fed to the model in batches; to enhance readability, we neglect the batches and view the case for a single problem matrix P of size $J \times M$ processed by the model's parameters. The input is passing the following layers:

- *Bidirectional LSTM layers*: The input matrix is fed to two LSTM layers: One determines the hidden state for the sequence as indicated by the input matrix P; the other one considers the sequence *backwards*. We use *all* hidden states of the layers, i.e. M^3 hidden states per job, as the return value. Hence, we obtain an output matrix with the same number of rows J as the input. We concatenate the LSTM outputs, which leads to a tensor size of $J \times 2M^3$.
- *LSTM layer*: The tensor is further processed by an LSTM layer with the same size and structure as the forward layer of the previous bidirectional LSTM layer. Thus, the size of the tensor is halved to $J \times M^3$.
- *Dense layer*: The following dense layer transforms each row of the input tensor independently. Thus, each row is treated like a flat input tensor. The output size is again $J \times M^3$.
- *Parallel dense layers*: In a relaxed assignment matrix, for each job on each stage, the sum of the assignments must equal 1. To ensure this, the last layer is split into I parts – one for each stage i. The output of the previous dense layer is passed to each of the stage layers in parallel. Depending on the number of jobs on the stage M_i, we set the number of nodes on each stage layer, which mimics the final output. Thus, the number of nodes on a stage layer is $J \times M_i$ and the number of rows stays again the same. In contrast to the previous layers, we choose the softmax activation function per stage layer, i.e. the sum of each row on a stage layer equals 1. Eventually, the stage layers are concatenated, resulting in the final assignment prediction with size $J \times M$.

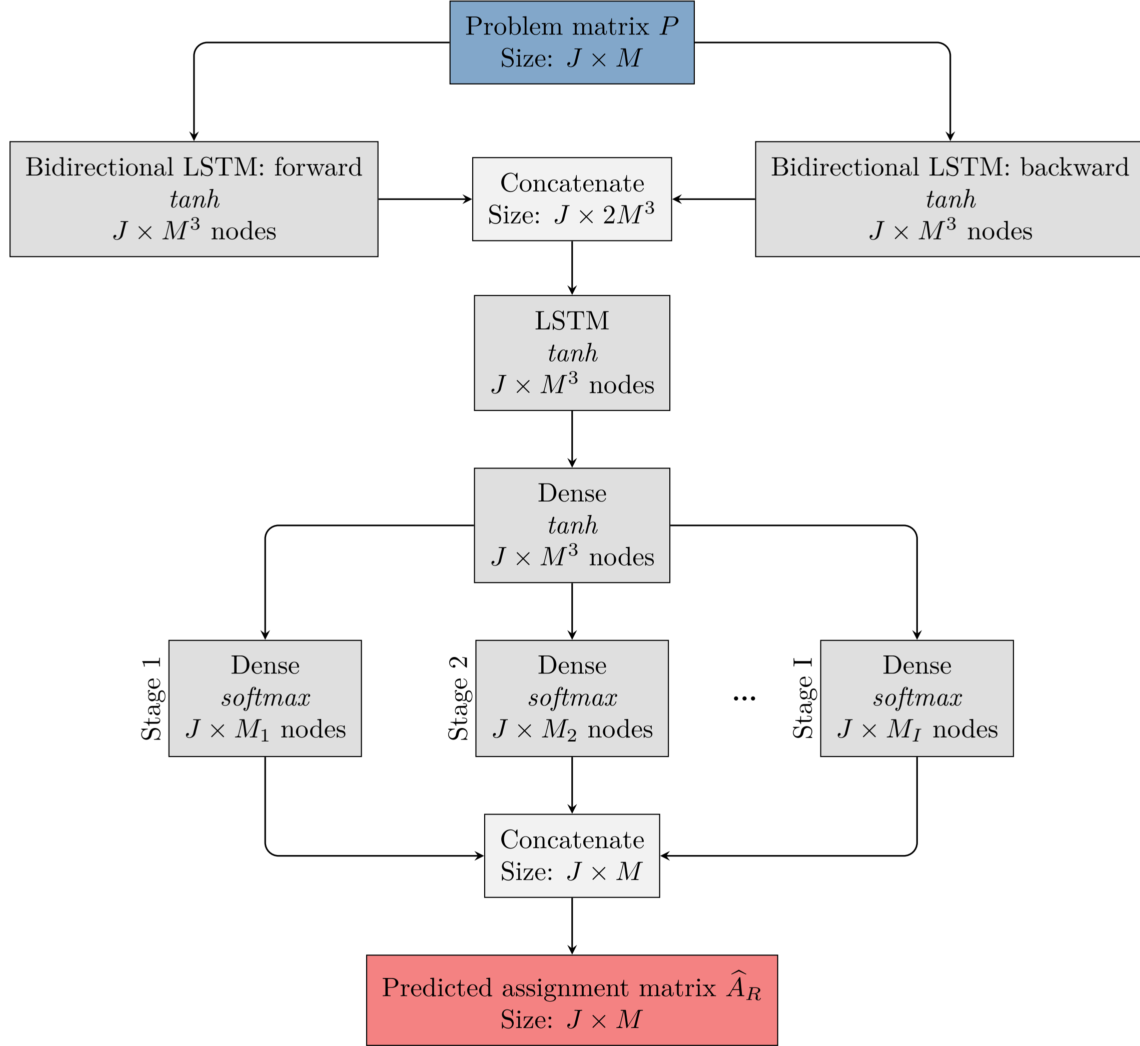


Figure 4.7: Neural network architecture of the assignment model

Strict Constraints

As previously analyzed, it is not directly possible to predict a *binary* assignment matrix and sustain differentiability at the same time. Thus, while the requirement of discrete machine selections has been relaxed in our loss function, we still require the condition

$$\sum_{m \in \mathcal{M}_i} \widehat{A}_{jm} = 1 \qquad \text{for all } \ j \in \{1, \ldots, J\}\,,\ i \in \{1, \ldots, I\} \tag{4.3.5}$$

to be strictly satisfied by the assignment output of the neural network. This strict constraint can be ensured by a suitable choice of the network's output layer, taking into consideration the machine configuration of the input matrix. In particular, we produce a suitable relaxed assignment matrix with the *softmax* activation function (cf. section 3.2.3), which is applied to each job on each stage to enforce properly

constrained outputs. Thus, each assignment matrix $\widehat{A}_R$ predicted by the neural network must then satisfy the constraint (4.3.5) by construction.[9]

The following example illustrates the final output produced by the parallel dense layers: For an input matrix with $J = 4$ jobs, $I = 3$ stages and $M_1 = 2$, $M_3 = 3$ and $M_2 = 2$ machines on each stage, the outputs of the parallel dense layers could contain the following values:

$$\widehat{A}_R^{I=1} = \begin{pmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \\ 0.75 & 0.25 \\ 0.1 & 0.9 \end{pmatrix}, \quad \widehat{A}_R^{I=2} = \begin{pmatrix} 0.11 & 0.21 & 0.68 \\ 0.16 & 0.54 & 0.3 \\ 0.98 & 0.0 & 0.02 \\ 0.3 & 0.3 & 0.4 \end{pmatrix} \quad \text{and} \quad \widehat{A}_R^{I=3} = \begin{pmatrix} 0 & 1 \\ 0.6 & 0.4 \\ 0.22 & 0.78 \\ 0.36 & 0.64 \end{pmatrix}.$$

Due to the softmax activation function applied to each job on each stage, the values sum to 1. The respective concatenated final assignment matrix takes the form

$$\widehat{A}_R = \begin{pmatrix} 0.8 & 0.2 & 0.11 & 0.21 & 0.68 & 0 & 1 \\ 0.4 & 0.6 & 0.16 & 0.54 & 0.3 & 0.6 & 0.4 \\ 0.75 & 0.25 & 0.98 & 0.0 & 0.02 & 0.22 & 0.78 \\ 0.1 & 0.9 & 0.3 & 0.3 & 0.4 & 0.36 & 0.64 \end{pmatrix}. \tag{4.3.6}$$

Categorization of the Model

Referring to the definition of deep learning in section 3.4, this NN model is a *recurrent deep learning neural network model*: We create different representations of the input matrix by consecutively applying LSTM layers. At the same time, we take into consideration the structure of the input matrix, i.e. the sequence information for the LSTM layers and the machine configuration to obtain the final layer.

4.3.4 Training Process

In order to train the neural network, i.e. to determine the network parameters such that the rectified version of the relaxed prediction $\widehat{A}_R = NN_A(P)$ represents an optimal assignment, we directly employ the relaxed makespan – or relaxed flowtime – function from algorithm 1 as the loss function for the training process. Thereby, it is sufficient to create a random training dataset $\mathcal{T} \subset \mathbb{R}_+^{J \times M}$ without pre-computing any labels. For given input $P \in \mathcal{T}$, the loss L is then simply defined for the objective C, i.e. the makespan $C_{\max}^{\mathrm{R}}$ or the flowtime $C_{\mathrm{sum}}^{\mathrm{R}}$, as

$$L(P, \widehat{A}_R) = C(P, \widehat{A}_R) + p(\widehat{A}_R) \tag{4.3.7}$$

where $\widehat{A}_R = NN_A(P)$ denotes the predicted assignment matrix and $p(\widehat{A}_R)$ is a suitable penalty term, as described in section 4.3.2.

Due to the continuous structure of both the output and the input space, and since $C_{\max}^{\mathrm{R}}$ and $C_{\mathrm{sum}}^{\mathrm{R}}$ are sufficiently regular, it is then possible to employ gradient-descent based methods in order to find optimal

[9]Note that this method is closely related to the classical approach of minimizing a loss function L under strict constraints by finding a minimizer of $L \circ g$, where the range of g is the proper (constrained) domain of L.

network parameters such that the loss L is minimized on the training dataset, i.e. the loss

$$L(\mathcal{T}) = \sum_{P \in \mathcal{T}} L(P, NN_A(P))$$

is minimized over the parameter space. By the choice of L, such a minimization ensures that the relaxed objective function value, e.g. $C^{\mathrm{R}}_{\max}(\widehat{A}_R)$, is sufficiently small. With the penalty term p the loss decreases for assignments close to the classical binary case $A \in \mathcal{A}$ for any $P \in \mathcal{T}$; note again that $C^{\mathrm{R}}_{\max}(A) = C_{\max}(A)$ if $A \in \mathcal{A}$.

Since NN_A is a recurrent neural network, it can take input instances of varying sizes during inference as well as during training. It has shown sufficient to train with input instances with job numbers that do not exceed $J = 6$ jobs. Any training with larger instances does not improve the quality of the results.

Rectification

Whether the rectification of an assignment $\widehat{A}_R = NN_A(P)$ also results in an optimal or close-to-optimal solution for $P \notin \mathcal{T}$ is monitored during the training process: For a validation data set with $J = 6$, the optimal assignments were computed with linear programming for the given machine configuration. After each epoch, the assignments are predicted for each problem matrix in the set to determine the amount of correctly predicted matrices and the mean deviation from the optimal makespan that would be produced by the optimal assignments. Since the output of the neural network is, in general, not an admissible schedule due to the employed relaxation, it is necessary to apply a *rectification* method to obtain a classical prediction from the ANN for the validation data set. For the assignment matrix, a simple rounding technique can be used by stagewise assigning each job to the machine with the highest relaxed assignment value:

$$A_{jm} = \begin{cases} 1 & \text{if } \widehat{A}_{jm} = \max_{n \in \mathcal{M}_i} \widehat{A}_{jn}\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

Note that the computational overhead due to this conversion is negligible and that output matrices which are already admissible are invariant under this rectification method, i.e. if $\widehat{A}_R$ satisfies the hybrid flow shop constraints, then $\widehat{A}_R = \widehat{A}$.

As an example,

$$\widehat{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}. \tag{4.3.8}$$

is the rectified version of eq. (4.3.6). During inference, we simply replace the maximum values of a relaxed assignment matrix $\widehat{A}_R$ produced by the softmax function with 1, and the other values are replaced with 0 (cf. section 4.3.4).

Application of the Penalties

As stated previously, we can make use of the same loss function presented in algorithm 1 for both the makespan and flowtime objective. However, the weights assigned to the penalties, represented by the

variable w^p, must be adapted depending on the objective viewed. Particularly, to train the model for the makespan objective, it has proven advantageous to take into consideration the flowtime objective as an additional penalty. The reasons for this become clear in the following example: For the problem matrix P with $J = 4$, $M_1 = 2$ and $M_2 = 2$ there are two assignment matrices given, A_1 and A_2:

$$P = \begin{pmatrix} 2 & 4 & 3 & 2 \\ 5 & 4 & 8 & 3 \\ 3 & 8 & 5 & 6 \\ 4 & 5 & 2 & 4 \end{pmatrix}, \qquad A_1 = \begin{pmatrix} 0 & 4 & 0 & 2 \\ 0 & 4 & 0 & 3 \\ 3 & 0 & 5 & 0 \\ 0 & 5 & 0 & 4 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} 2 & 0 & 3 & 0 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 6 \\ 4 & 0 & 2 & 0 \end{pmatrix},$$

The makespan is *the same* for both assignment variants with $C_{\max} = 17$, whereas the flowtime is *different* with $C^1_{\text{sum}} = 47$ and $C^2_{\text{sum}} = 53$. The flowtime objective is more "sensitive" towards changes in the assignments than the makespan objective, due to the fact that, with the flowtime, the change of *multiple* values, i.e. the completion times of all jobs, is viewed, whereas for the makespan, only the change of *one* value, i.e. the completion time of the last job, is taken into account. Thus, for the flowtime objective, the gradient receives additional "feedback" on changes in the assignments.

Makespan and flowtime are competing objectives: The assignment returning the optimal flowtime is not necessarily leading to the optimal makespan and vice versa. However, the two objective are not opposing, i.e. the assignment for the optimal makespan leads typically to an acceptable flowtime value, even if it is not optimal. The same applies, again, vice versa. We make use of this relation by taking into consideration the flowtime objective when training the makespan – at least at the beginning of training, until the process has stabilized. This approach strongly improves the learning of the makespan objective. During training, the flowtime penalty can be further decreased to emphasize the makespan objective. When training the flowtime, we can set the makespan penalty to 0.

On the other hand, we can simply make use of the additional penalty mentioned in section 4.3.2, which takes into consideration all non-maximum values of jobs on stages. This pushes the predicted relaxed assignment matrix $\widehat{A}_R$ towards binary solutions, i.e. such that $\widehat{A}_R = \widehat{A}$.

In summary, to train the NN model for the makespan and flowtime objective, we have to apply different penalty weights. For the makespan objective, the flowtime objective has to be included at the beginning of the learning process to stabilize the training. For both the makespan and flowtime objective, the term penalizing the non-maximum values has to be added as well. For example, the penalty weight for the makespan objective could take the value $w^p = (1, 1, 1)$ and for the flowtime objective, $w^p = (1, 0, 1)$. The weights can be changed during training to emphasize the actual prediction goal. The results obtained by the training process are evaluated in chapter 5.

4.4 Subproblem 2: Learning the Permutations

For the permutation problem, we aim for a permutation matrix $O \in \mathcal{O}$ which permutes the input matrix P such that, given the optimal assignments are applied to this permutation, the objective function is minimal.

For the relaxed problem, the neural network itself can then be described as a mapping

$$F\colon \mathbb{R}_{+}^{J\times J} \to \mathcal{O}_{\mathrm{R}}\,, \quad P \mapsto \widehat{O}_R = F(P)\,,$$

where $\widehat{O}_R$ denotes the predicted relaxed permutation for the processing times P. Afterwards, the matrix $\widehat{O}_R$ is rectified (cf. section 4.4.4), where we have to pay particular attention to the special requirements of admissible permutation matrices $O \in \mathcal{O}$.

4.4.1 Permutation Relaxation

An admissible result of the permutation prediction is a *binary.* i.e. *discrete* permutation matrix. For the relaxed assignment matrix A_R it was sufficient to apply the *softmax* activation function to each job on each stage: The assignment of a job to a machine on a stage is independent from other assignments, i.e. the change of one such assignment does not cause an inconsistency with another. This is different for the permutation problem, since per row and column, there is only one value allowed to be 1. We recall from (2.1.2) that the set of admissible permutation matrices is given by

$$\mathcal{O} = \Big\{ O \in \{0,1\}^{J\times J} \,\Big|\, \sum_{k=1}^{J} O_{kj} = 1 \ \text{ for all } j \in \{1,\dots,J\}, \sum_{j=1}^{J} O_{kj} = 1 \ \text{ for all } k \in \{1,\dots,J\} \Big\}. \tag{4.4.1}$$

From the following example matrices, only O_1 complies with eq. (4.4.1) and permutes P:

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{pmatrix}, \quad O_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad X_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad X_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Within matrices X_2 and X_3, the position index is shifted to the right or to the top, respectively, which conflicts with the position of job 2 and causes invalid states. "Permuting" P with the presented matrices O_1, X_2 and X_3 leads to

$$P_{\sigma_1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 & 8 \end{pmatrix}, \quad P_{X_2} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{pmatrix}, \quad P_{X_3} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 7 & 9 & 11 & 13 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 5 & 6 & 7 & 8 \end{pmatrix}.$$

The invalidity in X_2 causes job 3 to run twice on position 2 and 3, whereas job 2 is removed from the sequence. Multiplying with X_3 results in job 2 and 3 running simultaneously on position 2, whereas position 3 is unoccupied.

Using the relaxed approach, we would, as with the assignments, accept inconsistencies with jobs running at the same time. However, the relaxed results should again be contained in the convex hull of the discrete asdmissible set. The convex hull of the permutation matrices is given by the set of *doubly stochastic matrices*

$$\mathcal{O}_{\mathrm{R}} = \Big\{ O \in [0,1]^{J\times J} \,\Big|\, \sum_{k=1}^{J} O_{kj} = 1 \ \text{ for all } j \in \{1,\dots,J\},\ \sum_{j=1}^{J} O_{kj} = 1 \ \text{ for all } k \in \{1,\dots,J\} \Big\}, \quad (4.4.2)$$

which we will use as the extended solution space.

The simple *softmax* activation function, which was used for the relaxed assignment matrix, would only account for feasible relaxation of *either* the rows *or* the columns and can therefore not be used directly to enforce the constraints in eq. (4.4.2). Thus, to obtain a relaxed permutation matrix $\widehat{O}_R$, we make use of the *temperature-controlled sinkhorn layer* as proposed in [21]. The procedure is based on the *Sinkhorn-Knopp algorithm* [70]: Through the repeated normalization of the row and column values of a given matrix X, the sum of rows and columns approaches 1. In other words: The softmax activation function is repeatedly applied to the rows and columns of X.

Before normalization, the input matrix X is pre-processed with

$$\mathcal{S}^0 = \exp(X), \quad (4.4.3)$$

where the Sinkhorn operator $\mathcal{S}^0$ describes the state of the matrix X at iteration $i = 0$. With this initial step, the entries of X are pushed towards more extreme values, similar to the exponential function inside the *softmax* activation function (cf. section 3.2.3). Recursively, $\mathcal{S}$ is modified for M iterations with

$$\mathcal{S}^i = \mathcal{T}_C(\mathcal{T}_R(\mathcal{S}^{i-1}(X))), \quad (4.4.4)$$

where

$$\mathcal{T}_C^{j,k} = \frac{X_{j,k}}{\sum_m^M X_{l,j}} \quad \text{and} \quad \mathcal{T}_R^{j,k} = \frac{X_{j,k}}{\sum_m^M X_{j,l}} \quad (4.4.5)$$

normalize the columns and rows, respectively. With the temperature-controlled sinkhorn layer, the pre-processing step $\mathcal{S}^0$ is extended by the *temperature value* τ, i.e.

$$\mathcal{S}^0 = \exp(X/\tau). \quad (4.4.6)$$

Typically, $\tau < 1$, in which case the values in $\mathcal{S}^0$ become even more extreme – and more distinct – compared to the initial step without τ. The effect becomes more pronounced the closer τ is to 0: In the following iterations, the values in $\mathcal{S}^i$ tend more towards either 0 or 1.

The Sinkhorn algorithm is implemented as a *sinkhorn layer*, which applies the temperature-controlled sinkhorn algorithm batch-wise to the input. The number of iterations, as well as τ, are defined before learning. The more iterations the Sinkhorn algorithm performs on the input matrix X, the more closely a doubly stochastic matrix is approached; likewise, the lower the temperature value τ, the closer the output is to a binary permutation matrix. However, it is not advisable to set the number of iterations too high for training, since each iteration can be compared with and additional layer added to the architecture. This

makes the whole network structure more complex and might lead to a vanishing gradient in the earlier layers.

To obtain a *strictly* binary permutation matrix, we could also use a modified form of the Sinkhorn algorithm for rectification, cf. section 4.4.4.

4.4.2 Loss Function

Compared to the loss function of the assignment NN model (cf. algorithm 1), the loss function of the permutation NN model described in algorithm 2 seems relatively simple. Nevertheless, it actually exceeds the complexity of algorithm 2, since the loss function for the permutations *includes* the loss function of the assignment sub-problem and thus, the loss has to be minimized accounting for both the assignments and the permutations.

In algorithm 2 the input matrix P is permuted with the relaxed permutation matrix $\widehat{O}_R$. Afterwards, it becomes evident that an assignment neural network model has to be trained before the permutation model: The learned assignment model is an input to the loss function and used to compute a relaxed assignment matrix $\widehat{A}_R$ based on the permuted input. The relaxed assignment matrix is then rectified to obtain a viable solution, after which we use the loss function of the assignment prediction L_A (cf. algorithm 1) to compute the objective value, i.e. makespan or flowtime, based on the relaxed permutation matrix and the rectified predicted assignments. It is important that the penalty weights w^p for the assignment relaxed loss are set according to the objective, so that the returned result represents it (cf. section 4.3.2).

Similar to the loss function for the assignment problem, we compute a penalty that punishes all values deviating from the maximum value within a row. Thus, permutation matrices comprising only binary values are favored. The total loss is the sum of the objective value and the penalty term, each multiplied by the corresponding penalty weight for the permutation loss.

Algorithm 2: Relaxed Loss for the Permutation

```
1 PermutationRelaxedLoss (P, Ô_R)
    inputs : Processing times matrix P of size J × M,
             Predicted relaxed permutation matrix Ô_R of size J × M
             Trained assignment model NN_A for makespan or flowtime objective
             Penalty weights w^p of size 1 × 2
    output: The loss L comprising C^R_max ∨ C^R_sum and a penalty term
2   P_σR ← P · Ô_R                         // Assignment-relaxed processing times
3   Â_R ← NN_A(P_σR)                       // Assignment prediction for the permuted processing times
4   Â ← rectify(Â_R)                       // Rectification of the predicted assignments
    // Computation of the objective function with the loss function of the assignments:
5   C^R_max ∨ C^R_sum ← L_A(Â, P_σR)
    // Computation of the penalty and final loss L:
6   penalty ← p(Ô_R)
7   L = (w^p_1*C^R_max ∨ w^p_1*C^R_sum) + w^p_2*penalty)
8   return L
```

Penalty Functions

Similar to the penalty for the assignment model, we create a penalty for the permutation model p_σ that punishes all values that are different from 0 *and* the maximum value of a row, i.e. they do not account for the rectified permutation matrix:

$$p_\sigma = \sum_{k=1}^{J} (\sum_{j=1}^{J} \widehat{O}_{kj}) - \max_{j \in \{1,\dots,J\}} \widehat{O}_{kj}. \tag{4.4.7}$$

4.4.3 Structure of the Neural Network

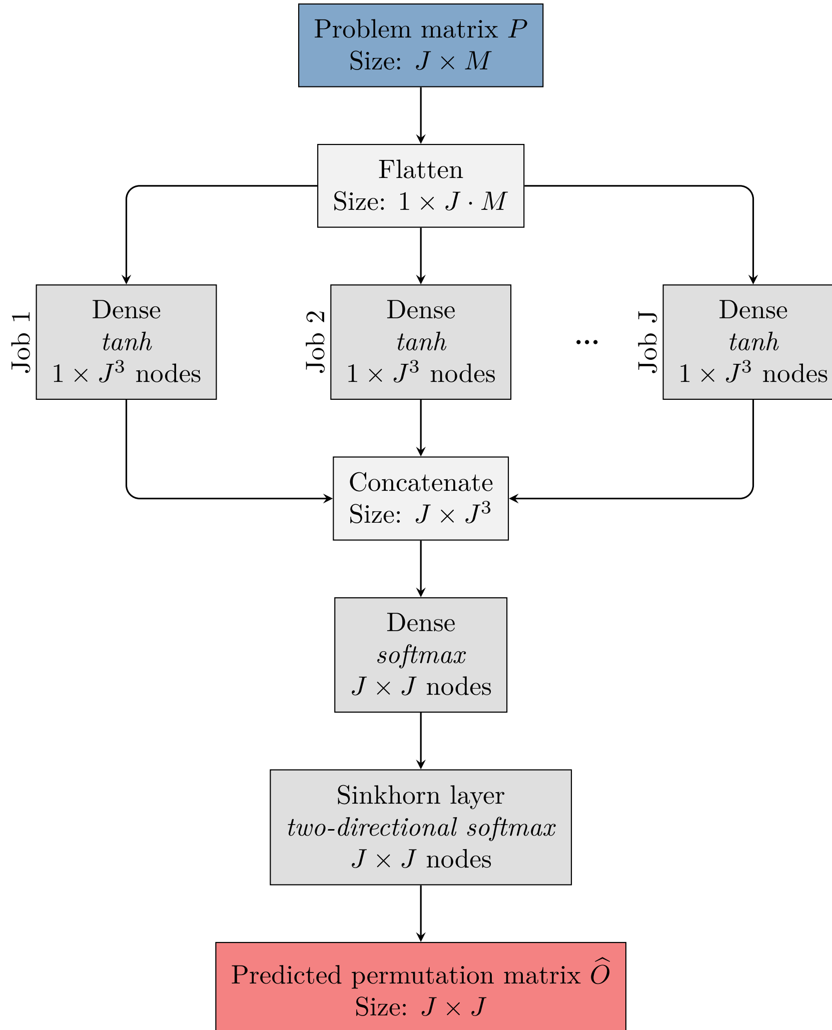


Figure 4.8: Neural network architecture of the permutation model

Figure 4.8 depicts the architecture of the neural network model for the permutation prediction. The model has to map the input processing times P to the respective permutation matrix $\widehat{O}$, i.e. $P \mapsto \widehat{O}$. The input matrix is passed on to a *flatten* layer: This layer type produces a rank-1 tensor from a tensor of higher rank. In this case, the rows of the input matrix are concatenated consecutively to reduce the rank. The

flattened tensor is then passed on to J parallel dense layers using a *tanh* activation function. This structure resembles the parallel layers of the assignment model in fig. 4.7. However, it is important to note that, for the permutation model, there are J parallel layers, i.e. one layer for each *job*, whereas for the assignment model, there are I parallel layers, i.e. one for each *stage*. Each of the job layers produces a flat tensor of size $1 \times J^3$ and the tensors are then concatenated to a tensor of rank 2 with size $J \times J^3$. Thus, the single job layers account for the rows in the newly formed tensor.

The concatenation layer is followed by a dense layer, which is capable of keeping the row dimensions constant by processing them individually. The output size is $J \times J$, which is already our desired output size. This is important since the tensor is afterwards processed by the Sinkhorn layer: As described in section 4.4.4, the Sinkhorn layer transforms the input tensor in a way that, for each row and column, the sum of values approaches 1, and for a sufficiently small temperature value τ, one of the values approaches 1 and the others 0. The size of the input is not changed during this modification. The result is a relaxed permutation matrix of size $J \times J$, which permutes the input matrix inside the loss function (cf. section 4.4.2).

Contrary to the assignment NN model, we do not make use of LSTMs: Naturally, the input matrix P is fed to the NN model with a random job order. Since recurrent layers are made to pay attention to the order and to process it for parameter optimization, we would send "false signals" to the recurrent layer with the input matrix of random order. Therefore, we use a dense layer on the flattened input, so each processing time is densely connected to the next layer and thus considered equally for parameter optimization.

Constraints

In contrast to the final *softmax* layer of the neural network for the assignment problem (cf. fig. 4.7), we cannot strictly enforce the constraints from eq. (4.4.1) directly. Instead, as described in section 4.4.1, we apply multiple iterations of the Sinkhorn algorithm. Note that since the final operation is the normalization $\mathcal{T}_C$ of the columns, the column-wise sum constraint is indeed satisfied strictly in this case, whereas the row-wise sum is only approximately equal to the required value 1.

Categorization of the Model

Since we are not making use of any typical deep learning layers, like RNNs or CNNs, one could argue that this NN is not a deep learning NN. Referring the definition of deep learning in section 3.4, deep learning means that different representations of the input are generated throughout the feedforward process. However, the flattening of the input at the beginning of the model makes it hard to talk about different representations of the input produced within the NN. Thus, unlike the NN model for the assignments, this model might not fall under the category of deep learning. Nonetheless, the architecture of the NN is on the more complex side, in particular considering the sinkhorn layer, which is basically a concatenation of softmax operations.

4.4.4 Training Process

For the training process of the permutation NN, we again renounce any labels and create a random training dataset $\mathcal{T} \subset \mathbb{R}_+^{J \times M}$. Recall that for the permutation training, the number of jobs J is fixed for a specific network, which is due to the network's structure not allowing inputs of varying size (cf. section 4.4.3). However, we can make use of the scalability of the assignment model, which is called inside the loss function: It does not have to be trained for varying job numbers and thus, we can fall back to the same assignment model for the training of permutation NNs with varying job numbers J.

Eventually, we want to set the parameters of the NN model using gradient-based methods such that the loss function

$$L(\mathcal{T}) = \sum_{P \in \mathcal{T}} L(P, NN_\sigma(P)) \tag{4.4.8}$$

returns a loss as small as possible.

To train the network parameters such that the prediction $\widehat{O}_R = NN_\sigma(P)$ leads to an optimal objective value, we have to enable the loss function to determine the objective value for a given prediction $\widehat{O}_R$. However, to compute e.g. $C_{\max}$ for any input matrix P, we also need information about the respective machine assignments A – otherwise, according to section 2.1.1, the schedule is not completely defined. As explained in section 4.2, we therefore feed a trained *assignment model* to the loss function: Inside the loss function, P is permuted by $\widehat{O}_R$ and then used as an input to the assignment model, which returns a relaxed assignment matrix $\widehat{A}_R$, i.e. $\widehat{A}_R = NN_A(P \cdot \widehat{O}_R)$. Afterwards, the relaxed assignment matrix is rectified using the rectification approach of section 4.3.4.

Including the given assignments $\widehat{A}$, we have defined a *full schedule* and thus, it is now possible to determine the objective value. For this, we make use of the differentiable loss function of the assignment model from algorithm 1. Hence, we apply a *deep recurrent neural network inside a loss function*, which further increases the complexity of the gradient-based optimization. Note that, depending on the viewed objective, we have to set the penalty weights of the assignment loss function accordingly. Including the penalty term p (cf. section 4.4.2), which penalizes all values but the maximum for each row, to bring the predictions closer to feasibility, i.e. binary permutations matrices, the loss function L is defined by

$$L(P, \widehat{O}_R) = C(P, \widehat{O}_R, \widehat{A}) + p(\widehat{O}_R) \tag{4.4.9}$$

for the makespan or flowtime objective.

Rectification

By virtue of the Sinkhorn layer, the output $\widehat{O}_R$ of the neural network NN_σ is (approximately) a doubly stochastic matrix. In order to convert this output into an actual permutation matrix, we can employ the temperature-controlled Sinkhorn function again: Since [50]

$$\lim_{\tau \to 0} \lim_{i \to \infty} \mathcal{S}^i(X/\tau) \in \mathcal{O}\,,$$

where $\mathcal{S}^i$ is given by eq. (4.4.4), we can choose a sufficiently low temperature value $\tau > 0$ and a sufficiently large $i \in \mathbb{N}$ such that $\mathcal{S}^i(\widehat{O}_R/\tau)$ can simply be rounded to a permutation matrix $\widehat{O} = \text{rectify}_\sigma(\widehat{O}_R) \in \mathcal{O}$.

The matrix obtained this way is equal to the permutation *closest* to $\widehat{O}_R$ that can be computed via the Hungarian algorithm as suggested by Emami and Ranka [21]. In particular, $\text{rectify}_\sigma(\widehat{O}_R) = \widehat{O}_R$ if $\widehat{O}_R$ is already a permutation matrix.

4.5 Limitations

The viewed problem is an NP-hard problem: As indicated by section 2.2, there exists no algorithm that can solve the problem *optimally* in polynomial time. Due to that, it cannot be expected that the described neural networks are able to predict globally optimal schedules for given processing times.

Furthermore, both the applied relaxation techniques and the machine learning methods themselves introduce possible sources of error. Although gradient-descent-based optimization methods, as presented in section 3.3.2, are capable of finding local minima for very general classes of functions, they usually cannot ensure that *global minima* are identified. Particularly, regarding the large solution space and the relatively large output matrix for the given problems, a binary matrix of size $J \times M$, it can be hardly avoided that the gradients get trapped within a local minimum. The neural-network-based approach presented therefore does not provide an exact solution, but rather an approximation method for the hybrid flow shop scheduling problem.

Furthermore, while the computational cost for finding a schedule using a trained neural network is extremely low when compared to other optimization methods (cf. chapter 5), the effort required during the training process needs to be considered as well. However, the training does not need to be performed "online" during high-frequency scheduling applications; instead, the network can be prepared at any point, and the training can be continued in the background over time to further improve results.

Finally, it should be noted that, due to the fixed topology of the trained neural network, the number of jobs[10], stages and machines per stage cannot be changed without restarting the training process. However, for a reduced number of jobs, a schedule can still be predicted via zero padding of the input, i.e. by including additional jobs with zero processing time. The number of stages can be reduced in a similar fashion, whereas the number of machines per stage could be decreased by adding machines with prohibitively large processing times to the input ("infinity padding"). The maximum number of jobs, stages and machines for which the network is applicable, however, remains bounded by the selected training data. Due to the increasing number of parameters and the resulting computational effort in training the network for larger input dimensions, a limit for these numbers needs to be carefully chosen beforehand.

[10] Note that this restriction does not apply to the pure assignment problem, where a recurrent network was used to allow for variable input sizes with respect to the number of jobs.

Chapter 5

Results

As already pointed out, it is challenging to generate labels, i.e. optimal solutions, for NP-hard problems. This complicates not only training but also *testing*: According to section 3.3.4, to evaluate the learned model, we use a test data set comprising problem instances and their respective labels. The test data set is not involved in the training process to ensure that the model's parameters are not specifically adjusted to the test data. Thus, since it is not possible to create labels for sufficiently large test cases for the viewed NP-hard problem, we will first create *exact results for small problem instances* using MILP (cf. section 2.1.3). With respect to the pure assignment sub-problem, we can create exact solutions, i.e. truly optimal assignments, for relatively large problem instances, with up to $J = 20$ jobs; whereas for the permutation problem, we are limited to problem sizes of $J = 6$. In addition, we will implement a *comparison to other approximation algorithms* and evaluate the relative performance regarding numerical results and computation time. All experiments are performed on a 3.4 Ghz Intel i7 CPU with 4 cores and 16 GB of memory.

5.1 Evaluation

Particularly, for the *pure assignment problem*, we will conduct the following performance tests in section 5.1.3:

- comparison to optimal results for up to $J = 20$ jobs via the accuracy and mean deviation from the optimum,
- comparison to random result generation via *Monte Carlo* methods with up to 1,000 trials and for up to $J = 60$ jobs,
- comparison to *mixed-integer linear programming* with different time limits and varying numbers of jobs.

Afterwards, we consider the pure permutation problem in section 5.1.4, where we perform a

- comparison to optimal results for $J = 6$ jobs and a
- comparison to random result generation (Monte Carlo) for $J = 12$ and $J = 16$ jobs.

Eventually, we view the complete problem in section 5.1.5, i.e. the combination of assignment and permutation prediction, with a

- comparison to optimal objective values for up to $J = 6$ jobs and a
- comparison to random result generation (Monte Carlo) for up to $J = 12$ jobs and $M = 6$ machines, as well as a
- comparison to MILP with different time limits and varying numbers of jobs.

5.1.1 Comparison Algorithms

For both comparison algorithms, the *Monte Carlo method* and *mixed-integer linear programming*, the quality of the results depends on the number of trials and the computation time, respectively. Given enough time or trials, these algorithms return an exact solution. Since our proposed method is intended to be highly performant in terms of computation time, it is not meaningful to simply compare the results to other heuristics without considering the computation time. Therefore, we indicate the computation times for the comparison algorithms and the machine learning approach.

Monte Carlo Method

For each input P, we apply a simple Monte Carlo (MC) algorithm by selecting a predefined amount of random (classical) outputs for our prediction goal, i.e. assignment matrices $A \in \mathcal{A}$ or permutation matrices $O \in \mathcal{O}$. We compute the objective value for each of the *trials* and select the *best* assignment or permutation matrix: Thus, more trials improve the results. We limit the number of generated input matrices to 1000, which ensures that the computation time stays within reasonable limits. For instance, to compute the results for 1000 instances with 1000 trials each, would require $1000 \cdot 1000 = 1,000,000$ operations, which can, depending on the input matrix size, lead to extensive computation times.

Although one of the most simple algorithms imaginable, this method of *guessing* solutions tends to yield remarkably good results in practice, especially for comparatively small problem sizes. Additionally, it is very easy to implement. However, the method is based on chance: For each execution we obtain different results. The more input matrices we produce, the more stable the algorithm becomes, which, in return, leads to high computational effort despite the simplicity of the algorithm.

Mixed-Integer Linear Programming

For the assignment case, we feed the input P to an optimizer solving a mixed-integer linear programming model to obtain a viable assignment matrix A as a result. The algorithm is used for both *exact* and *approximate* problem solving: Contrary to the exact approach, for the approximation case, the algorithm is stopped after a previously defined time interval, and the best result so far is returned. Thus, it functions as a benchmark for bigger problem instances. Similar to the Monte Carlo method, no relaxation is required in this case, since integer programming enables constraints on the solution variables which ensure that the resulting assignment matrix A contains only binary values.

The approximation version of the algorithm is advantageous for comparison to results of larger problem sizes for which we cannot generate exact results. For smaller problem sizes (up to $J = 20$) of the pure

assignment problem, the MILP algorithm, in combination with the highly performant *Gurobi* optimizer, delivers optimal results in reasonable time, which gives the opportunity to benchmark with optimal results.

To implement the MILP algorithm, we feed the linear programming model defined in section 2.1.3 to a solver: In particular, we make use of the commercial optimization tool *Gurobi*, which runs under an academic license. Note that Gurobi is, compared to free solvers, remarkably faster. Due to its high degree of optimization, it should be considered a state-of-the-art competitor in terms of computational performance. [49][66] Gurobi's performance makes it possible to generate optimal results in sufficiently large numbers to ensure high significance of our evaluation: Prior attempts with open-source solvers made it seem impossible to obtain a sufficiently large test set with exact results for varying problem sizes, whereas with Gurobi, we could compute the optimal objective value, e.g. $C_{\max}^{\text{opt}}$, for 1000 problem instances P for *each* viewed machine configuration and for up to 20 jobs.

Establishing a MILP model is, especially compared to the Monte Carlo method, difficult to implement. The model from section 2.1.3 has to be translated into code that the solver can process. The method is also error prone: Setting one of the numerous restrictions incorrectly can lead to hidden infeasibilities or prevent the program from running. Furthermore, the performance of open-source alternatives is, at least for more complex problems, poor, and one has to fall back to commercial optimizers to solve these problems competitively. In turn, Gurobi, as a commercial solver, is a "black box", which makes it impossible to understand precisely how the results are computed or to modify the source code [65, p. 15]. For example, we cannot comprehend how the starting solutions for the MILP algorithm are determined.

5.1.2 Structure of the Test Data

Most numerical experiments in this section consider a hybrid flow shop comprising three stages with two machines each. For the machines on each stage i, we generate normally distributed processing times P_{jm} which are then rounded to integer values. For the specific distributions $P_{jm} \sim \mathcal{N}(\mu_i, \sigma_i^2)$ on the stages, we randomly select a mean value $\mu_i \in [10, 900]$ between 10 and 900 as well as a standard deviation $\sigma_i^2 \in [1, \mu_i - 10]$; if any $P_{jm} < 1$ in a sample, we add the difference of the respective value to 1 to the whole matrix which avoids negative entries. This randomization method results in problem matrices with varied processing times.

The matrices

$$P_1 = \begin{pmatrix} 112 & 113 & 471 & 465 & 28 & 28 \\ 112 & 111 & 469 & 471 & 27 & 25 \\ 110 & 110 & 470 & 467 & 24 & 23 \\ 113 & 109 & 471 & 467 & 24 & 27 \\ 111 & 110 & 471 & 466 & 25 & 25 \\ 111 & 110 & 473 & 468 & 26 & 24 \end{pmatrix} \quad \text{and} \quad P_2 = \begin{pmatrix} 195 & 142 & 855 & 436 & 275 & 486 \\ 53 & 902 & 1048 & 1121 & 445 & 70 \\ 1899 & 661 & 1130 & 1167 & 2206 & 842 \\ 1983 & 907 & 536 & 1178 & 1706 & 1261 \\ 347 & 1067 & 785 & 478 & 2051 & 168 \\ 248 & 490 & 451 & 1798 & 93 & 1292 \end{pmatrix}$$

represent two examples of processing times matrices generated by the above method: They posses normally distributed processing times with different mean values and standard deviations at each stage. The stages in P_1 exhibit very small standard deviations, which emphasize the division between the stages, whereas the high standard deviations in P_2 result in a less structured problem.

5.1.3 Pure Assignment Problem

The focus is on a hybrid flow shop with the machine configuration $(M_1, M_2, M_3) = (2, 2, 2)$ on $I = 3$ stages. We perform the predictions for the makespan objective using the neural network described in section 4.3.3 trained with the procedure described in section 4.3.4 for 1000 test data matrices.

From eq. (2.2.1) we recall that the number of possible solutions for a HFS is

$$J! \left(\prod_{i=1}^{I} M_i \right)^J . \tag{5.1.1}$$

Leaving out the factorial term at the beginning computes the number of possibilities without permutations, i.e. for the pure assignment problem. In table 5.1 we see the amount of possibilities for varying job sizes J for the previously described HFS configuration. Considering the exponential growth of the solution space, it becomes obvious that computing the optimal makespan by brute-force, even for only the assignment problem, does not lead to a result in sufficient time. Nonetheless, the proposed neural network architecture is able to produce results for any number of jobs for a given machine configuration. We will in the next sections indicate the reliability of the results.

Table 5.1: Number of possible assignments for a HFS configuration with I=3 stages and 2 machines on each stage, with varying numbers of jobs J

J	**assignment possibilities**
6	262 144
10	1 073 741 824
20	$1.152\,922 \cdot 10^{18}$
40	$1.329\,228 \cdot 10^{36}$
100	$2.037\,036 \cdot 10^{90}$

Comparison to Optimal Solutions

For the predictions of the assignments and the respective makespans, the histograms in fig. 5.1 depict the deviation of the predictions from the optimal makespans. The tests were performed on 1000 problem matrices for $J = 6, 10, 16$ and 20 jobs. The *accuracy* states what percentage of the predictions reaches the optimum value, whereas the *mean deviation* describes the average deviation of the predictions from the optimum value. There is a clear tendency towards small deviations, e.g. more than half of the predictions deviate by only 1% or less from the optimum objective value for $J = 6$.

We can observe that, expectedly, the accuracy drops with increasing number of jobs J. However, the deviation does not increase in equal manner, as it stays around 2%. Unfortunately, we cannot check how the deviation behaves for bigger problem sizes, but we can expect that it will stay in the same range. We explain this behavior as follows: The size of the output matrix A increases with J, i.e. A contains $6 \cdot 6 = 36$ or $20 \cdot 6 = 120$ entries. The more entries, the higher the probability that one entry is “faulty”, which immediately leads to a drop in accuracy, since we only take into account completely correct results.

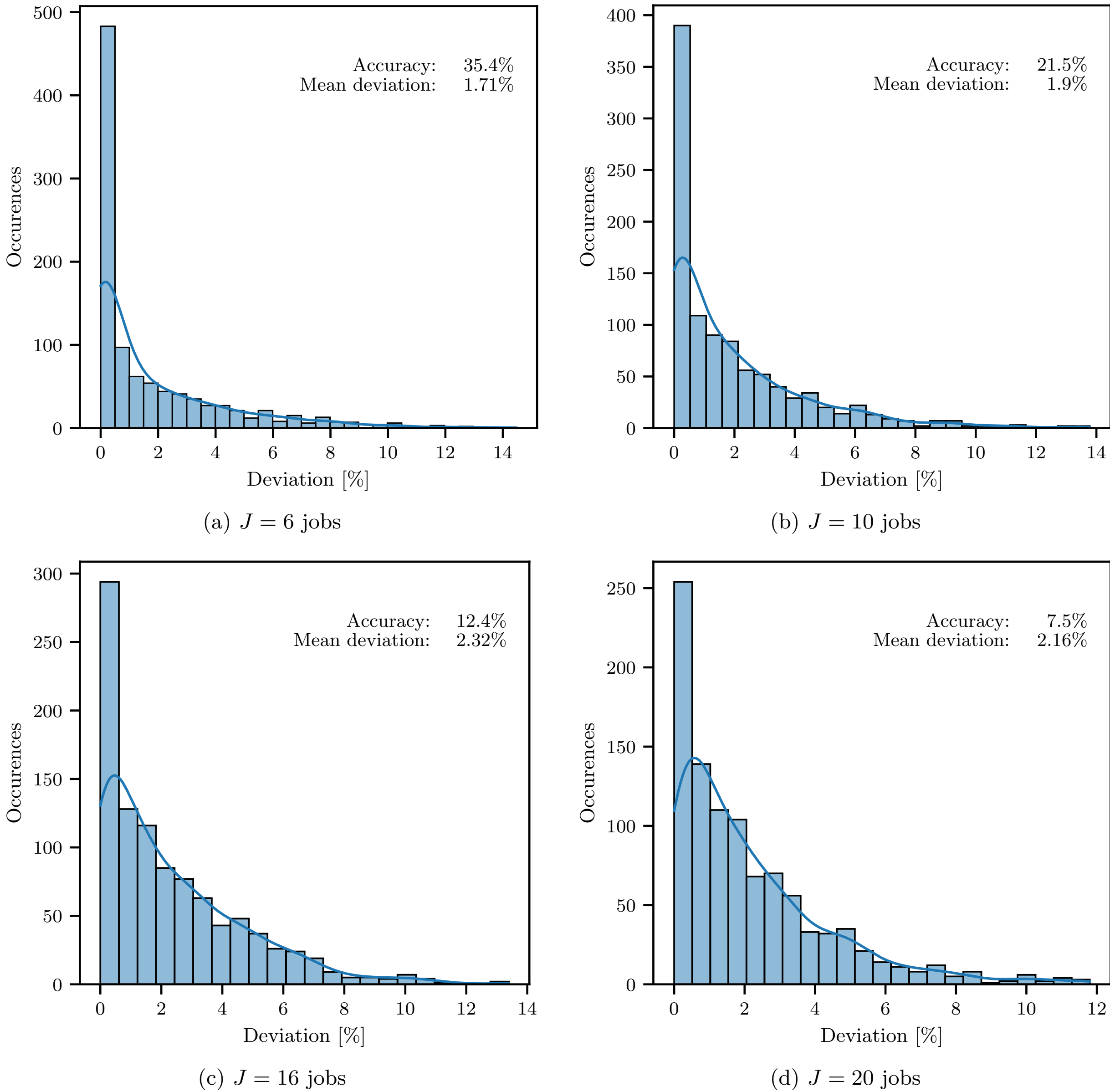


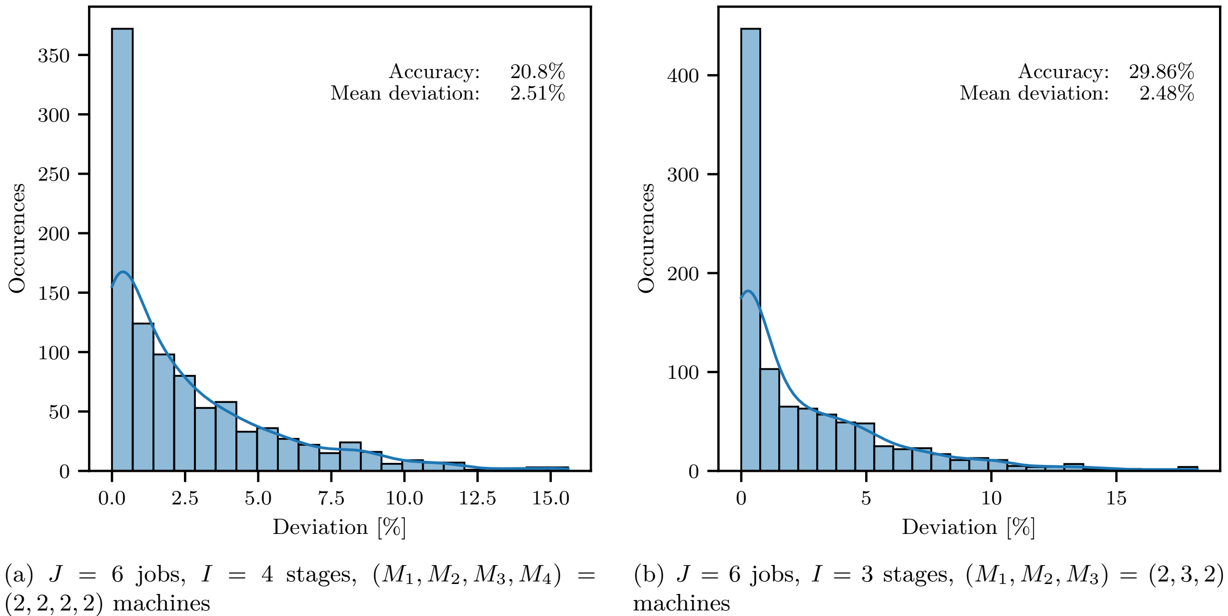
Figure 5.1: Accuracy and deviations from optimum *makespan* values of 1000 predictions for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages and varying numbers of jobs

However, the deviation is not affected in the same manner by single wrongly predicted values, as its contribution to, and influence on, the objective value decreases with rising J.

Other Machine Configurations

To show that the approach is also suitable for training neural networks for more complex machine configurations, we present the results for either $I = 4$ stages or the increased number $M_2 = 3$ of machines on stage 2 for $J = 6$ in fig. 5.2. Since the solution space expands with more machines or stages, the results are not as precise as for the above analyzed case with $I = 3$ and two machines on each stage. However, the mean deviation is still only around 2.5% – presuming that the mechanisms of the above example also apply for

the more complex case, the mean deviation will only increase moderately for larger inputs. The solution space for $I = 4$ with $(2 \cdot 2 \cdot 2 \cdot 2)^6 = 16777216$ possibilities is larger than for $I = 3$ with $(2 \cdot 3 \cdot 2)^6 = 2985984$ possibilities, which can explain why the accuracy is higher for the latter case.



(a) $J = 6$ jobs, $I = 4$ stages, $(M_1, M_2, M_3, M_4) = (2, 2, 2, 2)$ machines

(b) $J = 6$ jobs, $I = 3$ stages, $(M_1, M_2, M_3) = (2, 3, 2)$ machines

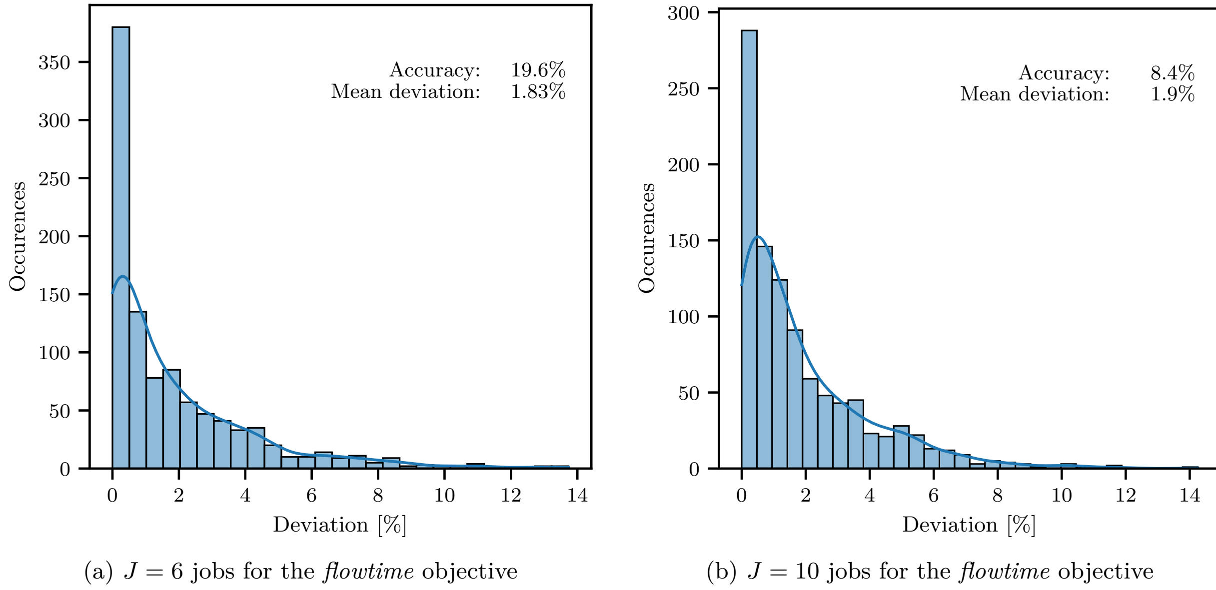
Figure 5.2: Accuracy and deviations from optimum makespan values of 1000 predictions for a hybrid flow shop with different machine configurations for $J = 6$ jobs

Flowtime Objective

In section 4.3.2 we described how the loss function can easily be modified to train an NN model for the flowtime objective instead of the makespan. Therefore, we present two examples of deviations from the optimum for the *flowtime* objective for $J = 6$ and $J = 10$ in fig. 5.3. Again, we can observe a low mean deviation from the optimum, whereas the accuracy is lower than for the makespan prediction with the same configuration. To understand this behavior, we recall from section 2.1.1 that the flowtime is defined as the sum of *all* completion times for the jobs in the last stage; the makespan comprises only the completion time of the *last* job processed. Thus, a globally optimal assignment needs to minimize *all* completion times, which in turn leads to a lower accuracy if only one entry is set incorrectly. However, the mean deviation is not as strongly affected.

Comparison to Monte Carlo

To compare the performance of the proposed approach with the Monte Carlo method, we created test data sets for job numbers between $J = 6$ and $J = 60$ with 1000 examples each. Although the neural network can handle larger problem inputs, the Monte Carlo method requires significantly more time to produce results for larger numbers of trials and thus, we limit the comparison to $J = 60$ jobs. We visually explain the results for selected job sizes in fig. 5.4 and provide more detailed information on the required execution times in table 5.2. Similarly, table 5.3, table 5.4 display the accuracies and mean deviations reached with different numbers of trials for MC based on the optimal solutions for up to $J = 20$, respectively.



(a) $J = 6$ jobs for the *flowtime* objective

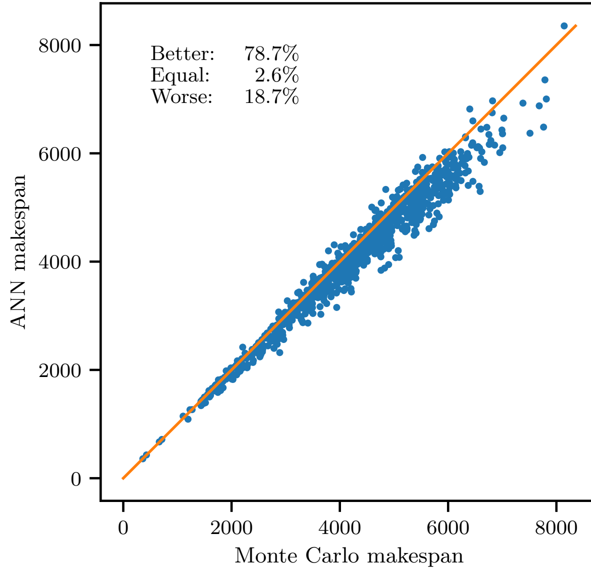
(b) $J = 10$ jobs for the *flowtime* objective

Figure 5.3: Accuracy and deviations from optimum *flowtime* values of 1000 predictions for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages with $J = 6$ and $J = 10$ jobs

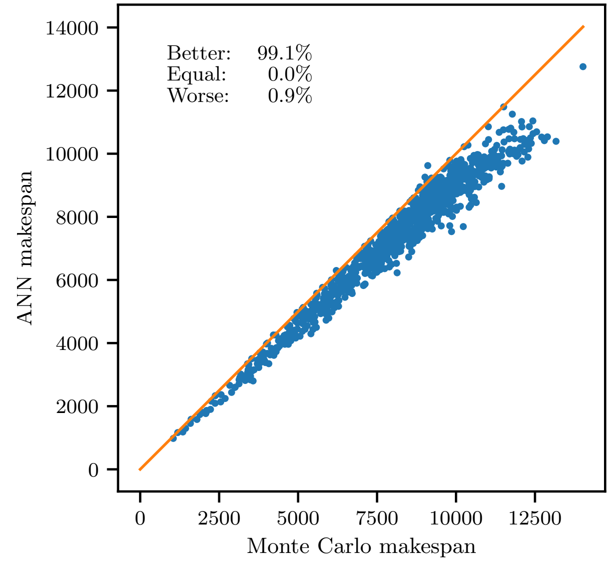
The plots in fig. 5.4 indicate the relative performance between the objective values returned by the trained ANN and the MC method. The x and y coordinates of the plotted dots represent the results of the MC method and the NN, respectively. Hence, all dots that lie on the orange diagonal line stand for results that are equal for MC and NN; dots below the line represent results where the ANN performed better, since the makespan provided by the MC method is *higher*. Dots above the line represent the reverse case. On the respective plots, the percentages of dots below (better), on (equal) and above (worse) the line are indicated. Additionally, we provide the mean relative difference between the results, i.e. for $J = 10$ jobs, the predictions of the NN are on average 4.41% *better* than the MC results.

We can observe a movement of the cloud of dots to the bottom for increasing numbers of jobs, as well as a decrease in the relative difference: Around 80% of the NN predictions are better than or equal to the MC results for $J = 10$ – whereas for $J = 40$ and $J = 60$, *all* the dots are *below* the line and thus, the predictions of the NN are superior in each case. As shown in table 5.3, the accuracy of the results produced by the MC method increases together with the number of trials, and drops rapidly with increasing job numbers. Even though the MC method delivers a single result that exceeds the proposed approach (accuracy for $J = 6$) with 1000 trials, the required time to obtain the outputs have to be considered: The times can be viewed in table 5.2, together with the prediction time of the NN for 1000 problem matrices, which already *include* the rectification process to obtain feasible results.[1] The MC method requires ~ 35 seconds for 1000 trials for $J = 6$, whereas the NN requires only ~ 0.73 seconds. Moreover, for this particular case, the MC method only performs better in terms of the accuracy – the mean deviation is larger compared to the NN result. On the other hand, for example, we would have to wait over 5 minutes to obtain a result for 1000 trials for $J = 40$, although the MC performance is significantly lower, as shown by fig. 5.4c.

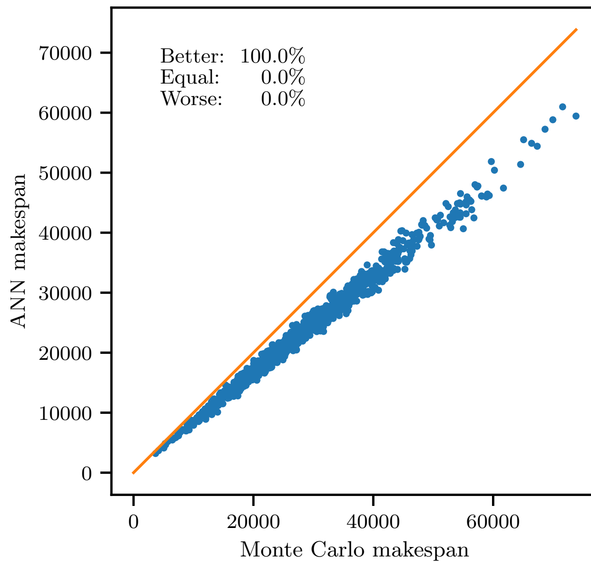
[1]Note that the displayed times are *approximate* and might slightly vary for independent executions.



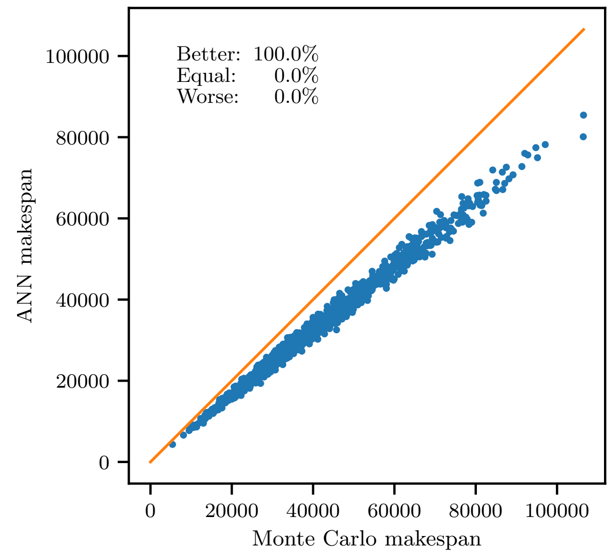
(a) $J = 10$ jobs, relative difference: -4.41%



(b) $J = 20$ jobs, relative difference: -9.86%



(c) $J = 40$ jobs, relative difference: -17.29%



(d) $J = 60$ jobs, relative difference: -19.26%

Figure 5.4: Comparisons between NN predictions and Monte Carlo results of 1000 problem instances for a hybrid flow shop with $(M_1, M_2, M_3) = (2, 2, 2)$ machines on $I = 3$ stages and varying numbers of jobs

Time-wise, we notice a "break-even point" at after approximately 10 trials: Before that, the MC method is faster, whereas for more than ~ 10 trials, the MC method demands more time than the NN predictions with rectification. However, for such few trials, the quality of the MC results is very poor and the time difference is marginal. Thus, the proposed approach should clearly be the preferred way to obtain the results in time-critical applications.

Table 5.2: Comparison of *execution times* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 60 jobs

J	**NN** [s]	**Monte Carlo trials** [s]					
		1	**10**	**100**	**250**	**500**	**1000**
6	0.73	0.21	0.62	3.64	8.45	15.81	35.98
10	1.16	0.26	1.06	5.71	13.26	25.63	54.02
12	1.32	0.3	1.02	7.33	17.8	32.78	67.87
16	1.84	0.45	1.43	9.8	22.8	45.13	94.54
20	3.17	0.63	2.03	14.04	56.36	76.02	148.45
40	4.15	1.28	4.15	30.47	74.36	151.5	348.05
60	7.36	1.84	7.36	54.82	134.35	284.73	1781.93

Table 5.3: Comparison of *accuracies* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 20 jobs

J	**NN** [%]	**Monte Carlo trials** [%]					
		1	**10**	**100**	**250**	**500**	**1000**
6	35.51	0.1	1.1	5.52	13.74	23.57	37.51
10	21.56	0.0	0.2	0.3	0.6	2.11	3.81
12	17.45	0.0	0.0	0.0	0.2	0.3	0.7
16	12.44	0.0	0.0	0.0	0.0	0.0	0.0
20	7.52	0.0	0.0	0.0	0.0	0.0	0.0

Table 5.4: Comparison of *mean deviations* for the prediction of 1000 test data inputs with an NN and varying test data sizes for Monte Carlo trials for up to 20 jobs

J	**NN** [%]	**Monte Carlo trials** [%]					
		1	**10**	**100**	**250**	**500**	**1000**
6	1.7	49.74	20.98	8.05	5.19	3.59	2.31
10	1.89	48.7	24.77	13.19	10.07	8.13	6.44
12	2.09	51.6	28.53	17.2	14.07	11.78	10.11
16	2.31	50.77	31.41	20.89	17.4	15.61	13.83
20	2.15	46.5	28.74	19.52	16.87	15.05	13.35

Comparison to Linear Programming

In addition to the Monte Carlo approach, we compare the prediction results for 500 matrices with the results returned by a MILP model (cf. section 2.1.3), which is processed by the high-performing Gurobi solver, where the optimization process is stopped after 0.5, 1, 2 or 5 seconds, respectively. In particular, we compare the relative difference of the results, which are shown in table 5.5.

Table 5.5: Relative differences between the NN and MILP results for varying numbers of jobs J and varying time limits for the MILP solver

J	**Relative difference from MILP** [%]			
	Limit 0.5s	**Limit 1s**	**Limit 2s**	**Limit 5s**
10	-3.2	0.2	1.9	1.9
20	-19.2	-10.5	-5.8	1.0
40	-30.9	-24.0	-20.2	-13.6
60	-38.3	-30.0	-23.0	-18.0

For $J = 10$, the MILP approach produces slightly better results for an execution limit of 1 second; both for a limit of 2 and 5 seconds, the relative difference is 1.9%. This is explained by the fact that the MILP solver already returns the *exact* solution for most of the inputs for a time interval larger than 2 seconds. This result is consistent with the mean deviation from the *optimum* in table 5.4 for $J = 10$. As indicated by section 5.1.1, for up to $J = 20$, we can compute the exact results for comparison in a reasonable amount of time. For $J = 20$, we see MILP performs worse than for $J = 10$; however, after 5 seconds, the MILP algorithm again performs slightly better. The picture changes for larger job numbers: Due to the expanding solution space, the results cannot compete with the results of the NN anymore. We would have to provide larger time spans to get competitive results for $J = 40$ or $J = 60$, which might be outside a reasonable range.

Note again that we have to take into consideration the *time* required by the algorithms: The MILP algorithm demands substantially more time to process all 200 problem matrices. For most input matrices, in particular for $J \geq 20$, the solver has not finished computing before exceeding the time limit. Thus, the total execution times are approximately 100, 200, 400 and 1000 seconds for the respective restraints – in contrast, the neural network demands only a fraction of this time. Even if we use the durations from table 5.2, which actually refer to a larger input of 1000 problem matrices, the NN outperforms the MILP algorithm by orders of magnitude.

Summary

Especially for time-critical problems, our results suggest that the proposed algorithm is superior to the comparison algorithms. Although for small instances, i.e. $J \leq 6$, the comparison algorithms return acceptable or – in the case of MILP – slightly better results, their execution times are *significantly* higher.

When the solution space expands because of e.g. a rising number of jobs, the algorithms cannot keep up in the same manner as the trained NN: The MILP algorithm requires exponentially more time and the MC method exponentially more trials to reach a consistent accuracy for large J.

5.1.4 Pure Permutation Problem

As analyzed in section 4.2, we cannot measure the performance of a given permutation without the respective assignments, since we need the additional information to compute the objective value. To nevertheless evaluate the performance of the predicted permutations, we will determine the assignments of the comparison algorithm in the following manner: For 500 problem matrices with $J = 6$ jobs we have determined the *optimal* assignments for *each* of the $6! = 720$ possible permutations using MILP (without time constraints). Thus, for each predicted permutation, we know the *respective* optimal objective value, as well as the *global* optimal objective value across all permutations (i.e. the optimal permutation).

We compare the mean deviation from the optimum for the NN predictions with the deviation from the optimum produced on average by 1000 random choices of permutations. Since we only have 720 possible permutations, a comparison to the Monte Carlo method with 1000 trials is not suitable in this case; we will make use of this comparison method for higher job numbers. As a result, the permutation predictions deviate only 6.02% from the optimum values, whereas the random choices deviate on average 15.62%. In $\sim 86\%$ of the cases, the NN predictions are superior to the random selections of permutations.

Table 5.6: Mean deviation from optimum for $J = 6$ of the NN predictions and 1000 random permutations

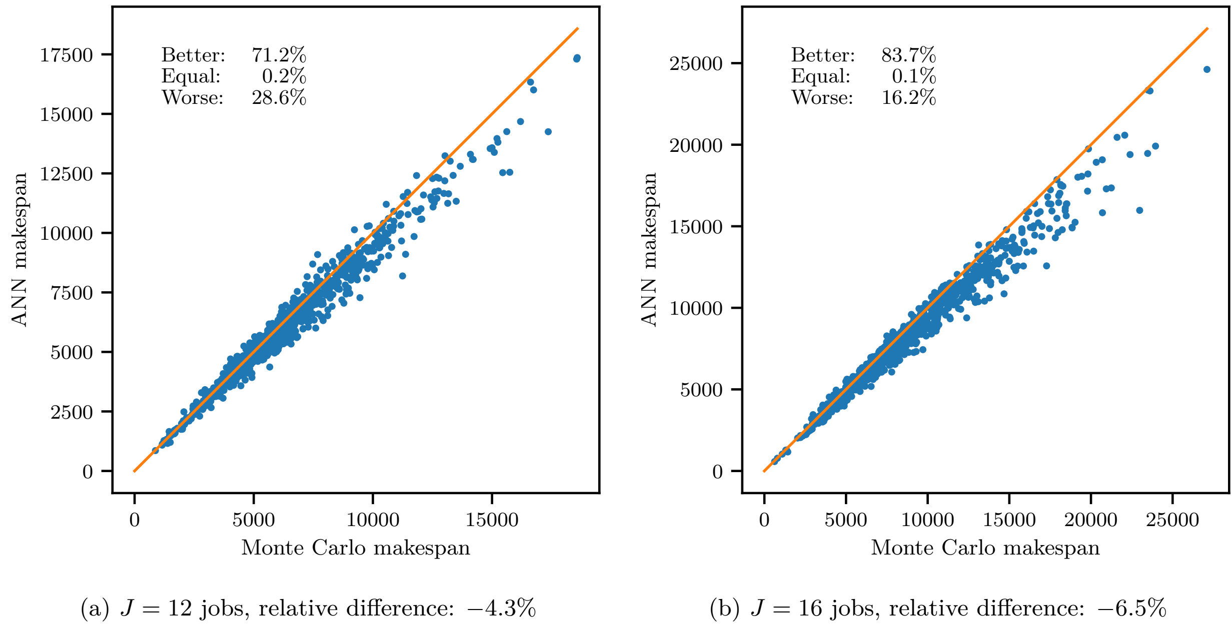
$J = 6$	**Mean deviation from optimum** [%]
NN prediction	6.02
Random permutation	15.62

For permutations with higher job numbers, we determine the assignments with the trained assignment NN. Naturally, there might be some bias in these computations since we have used the assignment NN to train the permutation NN, which might be favorable for the predictions. We could determine optimal assignments for $J \leq 20$ using MILP; however, the 1 000 000 MILP optimizations necessary for 1000 input matrices with 1000 Monte Carlo trials each, would go beyond scope, even with a strong time limitation. We conduct the tests for $J = 10$ and $J = 12$ jobs: Referring to table 5.4, the expected mean deviations from the optimum is only 1.89% and 2.09%, respectively, and in 21.56% and 17.45% of the cases we even obtain the optimal results. Thus, we can use the assignment prediction as a very good estimation of the true optimal makespan.

The results in fig. 5.5 reveal again that for an increasing input problem size the comparison algorithm cannot provide competitive solutions. For $J = 12$ with 1000 MC trials, the NN provides better results in $\sim 71\%$ of the cases. For $J = 16$, the discrepancy increases again with $\sim 83\%$ better results produce by the NN. Recall again, that the predictions produced with the NN are available almost immediately, whereas the MC take tremendously longer to evaluate.

5.1.5 Complete Approach

Testing the complete approach, i.e. the permutation and assignment prediction together, ultimately stretches the solution space according to eq. (2.2.1). As an example, for the least complex case we have viewed in this chapter, with $J = 6$ and $M = 6$, there are $6! = 720$ possible permutations and, according to



(a) $J = 12$ jobs, relative difference: -4.3%

(b) $J = 16$ jobs, relative difference: -6.5%

Figure 5.5: Comparison between NN permutation predictions and Monte Carlo results of 1000 problem instances for $J = 10$ and $J = 12$ jobs

table 5.1, 262 144 assignment combinations: Thus, in total, we obtain $720 \cdot 262\,144 = 188\,743\,680$ possible solutions.

This suggests that it is even harder for e.g. the Monte Carlo method to discover acceptable solutions in the expanded solution space. For $J = 6$ we have determined the optimal assignment for each possible permutation for each of the 500 analyzed problem instances. Thus, we can derive the globally optimal objective values including the respective permutation(s) and assignment matrix, and easily compare the NN prediction to these optimal values. A first test shows that the NN predictions deviate 7.31% from the optimum, whereas random selections for permutation and assignments possess a mean deviation of 71.62% (cf. table 5.7). Compared to the deviation from the optimum only regarding the permutation discussed in section 5.1.4 (15.62%), there is an immense increase of uncertainty – which is explained by the solution space growing by a factor of 262 144. The NN approach returns better results than the random selection in 100% of the cases.

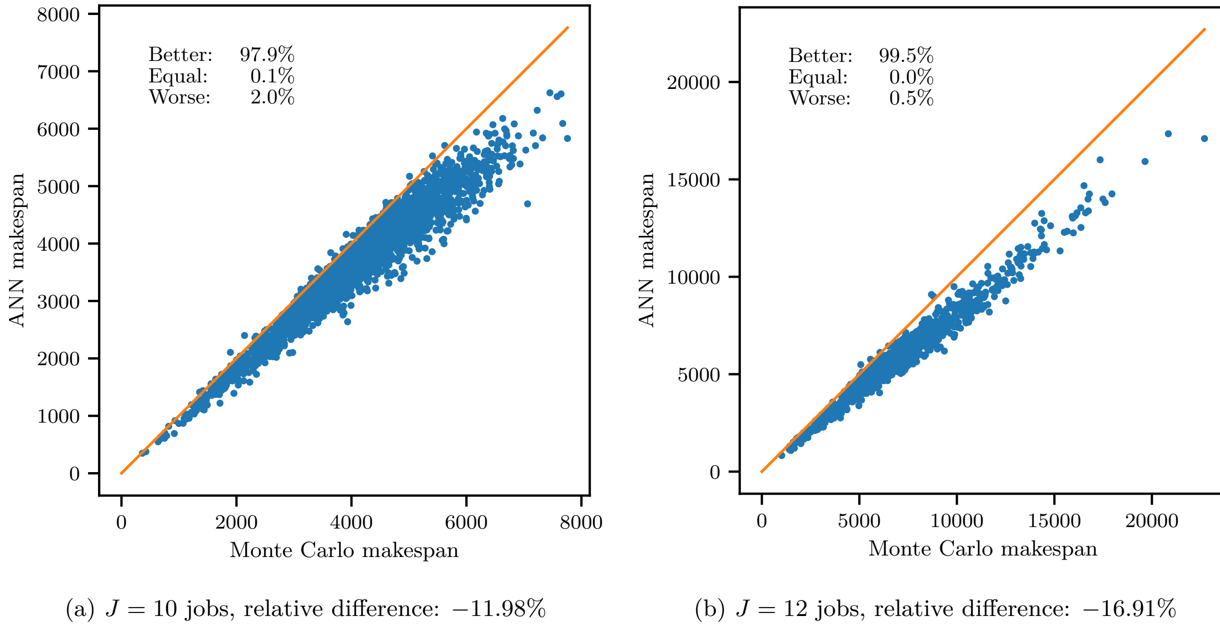
Table 5.7: Mean deviation from optimum of the combined prediction of permutation and assignments and random selections for $J = 6$

$J = 6$	**Mean deviation from optimum** [%]
NN prediction	7.31
Random permutation	71.62

Monte Carlo Comparison

In fig. 5.6 we see the Monte Carlo method comparison for $J = 10$ and $J = 12$ with 1000 trials. The result is consistent with the previous observation: The MC method cannot keep up with the NN predictions

because of the expanded solution space. For almost every input, the NN approach is superior to the Monte Carlo method.



(a) $J = 10$ jobs, relative difference: -11.98%

(b) $J = 12$ jobs, relative difference: -16.91%

Figure 5.6: Comparison between NN permutation predictions and Monte Carlo results of 1000 problem instances for $J = 12$ and $J = 16$ jobs

MILP Comparison

For the comparison of the complete approach with linear programming, we extend the MILP model from section 2.1.3 with additional constraints to enable a concurrent permutation optimization. We referred to the MILP model described in [84] to append the needed inequalities while considering the assignment rules defined in section 2.1, which remain unchanged.

Table 5.8: Relative differences between the NN and MILP results for varying numbers of jobs J and varying time limits for the MILP solver

J	**Relative difference from MILP** [%]			
	Limit 1s	**Limit 2s**	**Limit 5s**	**Limit 10s**
6	3.9	5.5	6.6	6.9
10	-13.8	-5.7	-4.4	-1.3
12	-38.2	-16.6	-8.8	-7.2
16	-47.6	-43.4	-18.0	-13.1

We compare the performance of our approach to the results generated by the Gurobi optimizer, which applies the aforementioned MILP model to the input. The procedure is repeated for different time limits (1, 2, 5 and 10 seconds) for 500 problem instances each. Due to the increased complexity, we leave out the time limit of 0.5 we used for comparison with the assignment NN and add the new time limit of 10 seconds.

Table 5.8 shows an overview of the mean deviation of the NN predictions from the MILP results for the respective time limit. For $J = 6$, the MILP approach returns better results than the NN; however, it takes significantly longer to compute the results with MILP. Note that even after 10 seconds, the optimizer has not yet reached the optimal result for all input problems, since the mean deviation from the optimum of the NN has not yet reached 7.31%, cf. table 5.7.

With increasing J, the accuracy of the MILP approach again drops rapidly. This is consistent with the observations of the pure assignment case, although for the combined approach, the effect is more significant: The solution space grows more rapidly for the full HFS problem than for either of the sub-problems alone. Even after 10 seconds, the MILP approach has not yet reached optimal solutions for $J = 10$, whereas for the pure assignment problem, optimality was reached after ~ 2 seconds.

Summary

The results of this section suggest that the proposed algorithm is superior to the comparison algorithms: Although for small problem instances, i.e. with $J = 6$, the time-constrained MILP algorithm provides better results, it demands a lot more time to return the required results. For larger instances, the comparison algorithms are not able to compete with the proposed algorithm concerning *both* performance *and* accuracy. Both MILP and Monte Carlo fail to scale to the expanding solution space regarding the results, whereas the NN can largely maintain the accuracy with growing problem size.

5.1.6 Greedy Post-Processing

Recall that the outputs of the neural networks NN_A and NN_σ are not always immediately feasible, since the values in the result matrices generally differ from 0 or 1. There is still some *uncertainty* in the results. We make use of the additional information that is conveyed by the relaxed results and perform a simple *greedy algorithm*[2] on the outputs of the neural networks. We set a tolerance of 0.75: For each maximum value in the result matrices below this tolerance, we check if the *closest alternative* results in an improvement. Due to its simplicity and easy implementability, the algorithm has a short run time and can process inputs batch-wise.

For the assignment output, this means that for each job on each stage we check if the maximum value, i.e. the value that would lead to an assignment for the rectified version, is smaller than 0.75. In that case, we assign the job to the machine with the second largest value instead and compute the objective value: If it improves, we keep the new assignment matrix, and if not, we reject the alternative. Similarly, for the permutation prediction, if a prediction is below 0.75, we swap the current position of the predicted value with the position with the second-highest prediction value. For the combined approach, we first improve the predicted sequence, predict the respective assignments and improve these assignments with the greedy algorithm.

Table 5.9 shows the improvement that is achieved by this post-processing procedure for $J = 6$ and different tolerances. For the pure assignment problem, the enhancement is not as evident as for the pure permutation problem, where we can already observe an improvement for a tolerance of 0.6. The combined approach,

[2] Greedy algorithms accept a change option that currently leads to improved results, while neglecting future options that might provide even better results [56, p. 88].

where both the permutations as well as the respective predicted assignments are improved, profits most from the greedy algorithm.

Table 5.9: Overview of the improvements reached by the greedy algorithm concerning the mean deviation from the optimum, for $J = 6$ jobs and varying tolerances

J=6	**Mean deviation from optimum**					
Tolerance	**Assignments** [%]		**Permutations** [%]		**Combined** [%]	
	Original	**Greedy**	**Original**	**Greedy**	**Original**	**Greedy**
0.6	1.71	1.71	6.02	5.22	7.31	6.34
0.75	1.71	1.52	6.02	4.62	7.31	5.62
0.95	1.71	1.46	6.02	4.29	7.31	5.15

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we described a new approach to solve NP-hard combinatorial problems with state-of-the-art machine learning techniques, using the hybrid flow shop scheduling problem as an example, which consists of two subproblems, i.e. a permutation optimization and an assignment optimization. After a thorough introduction of the analyzed problem and a classification according to complexity classes, we presented an extensive analysis of the underlying machine learning principles, both mathematically and visually.

Afterwards, we presented the general process of the approach: No labels are required for the training of the NN, since a sophisticated *loss function*, which represents the objective function of the viewed problem, steps in for them. We directly adapt the parameters of the NN using the results returned by the loss function, which have to be as low as possible. The *discreteness* problem is solved with a relaxation approach and a following rectification process. We also provided a literature overview of machine learning and combinatorial problems, especially regarding the hybrid flow shop problem. Concerning the approach of *label-free learning* via the loss function, there is only little literature available that uses a similar approach.

To approximate the hybrid flow shop, we considered both subproblems individually and eventually combined them to obtain solutions for the entire problem. Therefore, we created and trained two neural networks: one for the permutation inference and one for the assignments inference. Since both subproblems depend on each other, we integrated the loss function of the assignment NN into the loss function of the permutation NN, which increases the complexity of the permutation training process. Since both subproblems can be solved with the respective neural networks individually, we evaluated our results for both networks separately, besides the combined approach. As comparison algorithms, we used the simple Monte Carlo method and the more sophisticated mixed-integer linear programming model, which was fed to the state-of-the-art commercial solver Gurobi.

Our numerical results show that neural networks can provide a highly performant method for finding approximate solutions to the entire hybrid flow shop scheduling problem as well as the underlying subproblems. The neural networks outpace the comparison algorithms concerning *performance* and also

accuracy: Due to the *concurrent* processing of multiple problem instances, the trained neural networks return the desired results in a fraction of the time that the comparison algorithms take. With the growing solution space, e.g. when the number of jobs increases, the neural networks do not lose accuracy as much as the comparison algorithms, which makes the neural network approach favorable for larger problem sizes concerning both performance and accuracy.

Additionally, the neural network for the assignments inference can process inputs of varying size: Even for different job input sizes J we only have to train one network per machine configuration, and thus, the approach shows a high degree of *scalability*. We have also shown that, by a simple modification of the loss function, we can achieve similar results towards other optimization aims, i.e. flowtime instead of makespan, which demonstrates the *flexibility* of the approach.

In summary, for smaller problem matrices and less time-restrictive optimizations, one might consider the effort to establish a mixed-integer linear programming model to exactly solve these instances. With growing complexity of the input problems, for time-critical application or if the testing of multiple alternatives is desired, we cannot get around the proposed high-performant neural network approach. It does not only deliver results substantially faster than comparable algorithms, but it is also more reliable concerning the solution quality for more complex input instances.

6.2 Outlook

Thus far, we have applied the proposed approach to the complex hybrid flow shop problem. For future research, the approach should also be applied to other combinatorial problems, e.g. scheduling problems or classical optimization problems like the traveling salesman problem. Since we have presented neural network architectures for both the permutation and the assignment problem, as well as a combined approach, the presented work can be applied to a large portion of combinatorial problems.

The neural network for the assignment subproblem is capable of handling varying numbers of jobs as inputs. Currently, this is not the case of the permutation NN model: For each input size, we have to train a new neural network to produce outputs of suitable size. Thus, to further improve the approach, the permutation NN could possibly be modified in a way that it also accepts varying input sizes, e.g. with recurrent layers. A simpler way to extend the approach could be *zero padding*: For a permutation NN that was trained for J jobs, we can thereby use any problem instance with k jobs where $k \leq j$. To adapt a smaller matrix to a bigger input, the respective rows are filled with zeros.

The training of the neural networks as well as the testing were executed with limited computational resources. Considering the fast development of computational power, the same algorithm could yield better results in the future. Thus, with further improvement of the proposed method and additional computational power, the approach could become even more performant and versatile.

In this dissertation, we have overcome the two major obstacles that are encountered when solving NP-hard combinatorial problems with machine learning: the *discreteness* of the results and the difficulty of *label generation*. Thus, the dissertation has laid the foundation for further research in this area and can serve as a starting point for additional approaches, as it might have unleashed the potential of machine learning for (NP-hard) combinatorial problems.

Bibliography

[1] C. C. Aggarwal. *Neural networks and deep learning: A textbook*. Cham: Springer, 2018. ISBN: 9783319944630.

[2] A. Allahverdi and F. S. Al-Anzi. "Scheduling multi-stage parallel-processor services to minimize average response time". *Journal of the Operational Research Society* 57.1 (2006). Pp. 101–110. ISSN: 0160-5682. DOI: 10.1057/palgrave.jors.2601987.

[3] E. Alpaydin. *Maschinelles Lernen*. 2. Auflage. De Gruyter Studium. Berlin and Boston: De Gruyter Oldenbourg, 2019. ISBN: 9783110617887.

[4] M. A. Aragon-Calvo. "Self-supervised learning with physics-aware neural networks - I. Galaxy model fitting". *Monthly Notices of the Royal Astronomical Society* 498.3 (2019). Pp. 3713–3719.

[5] A. G. Barto, D. Wunsch, J. Si, and W. B. Powell. *Handbook of Learning and Approximate Dynamic Programming*. IEEE Press Series on Computational Intelligence. Wiley, 2004. ISBN: 9780471660545.

[6] Y. Bengio, A. Lodi, and A. Prouvost. *Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon*. 2018. URL: http://arxiv.org/pdf/1811.06128v2.

[7] T. Beucler, M. Pritchard, S. Rasp, J. Ott, P. Baldi, and P. Gentine. "Enforcing Analytic Constraints in Neural Networks Emulating Physical Systems". *Physical Review Letters* 126.9 (2021). DOI: 098302.

[8] D. Beyer. *Artificial intelligence and machine learning in industry: perspectives from leading practitioners*. First edition. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 1-4920-4891-7.

[9] C. M. Bishop. *Neural networks for pattern recognition*. Reprint. Oxford: Oxford Univ. Press, 2005. ISBN: 0198538642.

[10] J. Brammer, B. Lutz, and D. Neumann. "Permutation flow shop scheduling with multiple lines and demand plans using reinforcement learning". *European Journal of Operational Research* 299.1 (2022). Pp. 75–86. ISSN: 03772217. DOI: 10.1016/j.ejor.2021.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0377221721006743.

[11] R. A. Brualdi. *Combinatorial Matrix Theory*. Advanced Courses in Mathematics - CRM Barcelona. Cham: Springer International Publishing, 2018. ISBN: 9783319709536.

[12] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. 2014. DOI: 10.48550/ARXIV.1409.1259.

[13] F. Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. 1. Auflage. Frechen: mitp, 2018. ISBN: 3-95845-840-8.

[14] Clay Mathematics Institute. *Millennium problems: P vs NP problem*. URL: http://www.claymath.org/millennium-problems/p-vs-np-problem (visited on 09/04/2022).

[15] T. H. Cormen. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press, 2022. ISBN: 9780262367509.

[16] F. M. Defersha. “A comprehensive mathematical model for hybrid flexible flowshop lot streaming problem”. *International Journal of Industrial Engineering Computations* 2.2 (2011). Pp. 283–294. ISSN: 19232926. DOI: 10.5267/j.ijiec.2010.07.006.

[17] T. Dozat. “Incorporating Nesterov Momentum into Adam”.

[18] J. Duchi, E. Hazan, and Y. Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. *J. Mach. Learn. Res.* 12.null (2011). Pp. 2121–2159. ISSN: 1532-4435.

[19] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. 2. ed. A Wiley-Interscience publication. New York: Wiley, 2001. ISBN: 0471056693.

[20] W. E and B. Yu. *The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*. 2017. DOI: 10.48550/ARXIV.1710.00211.

[21] P. Emami and S. Ranka. “Learning Permutations with Sinkhorn Policy Gradient”. *CoRR* abs/1805.07010 (2018).

[22] W. Ertel. *Grundkurs Künstliche Intelligenz*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016. ISBN: 978-3-658-13548-5. DOI: 10.1007/978-3-658-13549-2.

[23] P. Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge: Cambridge Univ. Press, 2012. ISBN: 9781107096394.

[24] Y. C. Fonseca-Reyna, Y. Martinez-Jimenez, A. V. Cabrera, and E. A. R. Sanchez. “Optimization of Heavily Constrained Hybrid-Flexible Flowshop Problems Using a Multi-Agent Reinforcement Learning Approach”. *Investigación operacional* 40.1 (2019). ISSN: 0257-4306.

[25] Y. C. Fonseca-Reyna, A. Puris, Y. Martinez-Jimenez, and Y. Trujillo. “An Improvement of Reinforcement Learning Approach for Permutation of Flow-Shop Scheduling Problems”. *RISTI - Revista Iberica de Sistemas e Tecnologias de Informacao* (2019). Pp. 257–270.

[26] J. M. Framinan, R. Leisten, and R. R. García. *Manufacturing scheduling systems: an integrated view on models, methods and tools*. London: Springer, 2014. ISBN: 9781447162711.

[27] R. Garg, V. Kumar, G. Carneiro, and I. Reid. *Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue*. 2016. DOI: 10.48550/ARXIV.1603.04992.

[28] Z. Ghahramani. “Unsupervised Learning”. *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*. Ed. by O. Bousquet, U. von Luxburg, and G. Rätsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 72–112. ISBN: 978-3-540-28650-9. DOI: 10.1007/978-3-540-28650-9_5.

[29] A. Glassner. *Deep Learning: A Visual Approach*. No Starch Press, 2021. ISBN: 1-0981-2901-6.

[30] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511804106.

[31] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts and London, England: The MIT Press, 2016. ISBN: 9780262035613.

[32] M. Gourgand, N. Grangeon, and S. Norre. “Meta-heuristics for the deterministic hybrid flow shop problem”. *Journal Europeen des Systemes Automatises* 34 (2000). Pp. 1107–1135.

[33] A. Guinet, M. M. Solomon, P. K. Kedia, and A. Dussauchoy. “A computational study of heuristics for two-stage flexible flowshops”. *International Journal of Production Research* 34.5 (1996). Pp. 1399–1415. ISSN: 0020-7543. DOI: 10.1080/00207549608904972.

[34] J. N. D. Gupta. “Two-Stage, Hybrid Flowshop Scheduling Problem”. *Journal of the Operational Research Society* 39.4 (1988). Pp. 359–364. ISSN: 0160-5682. URL: http://www.jstor.org/stable/2582115 (visited on 05/17/2022).

[35] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. New York, NY: Springer New York, 2009. ISBN: 978-0-387-84857-0. DOI: 10.1007/b94608.

[36] H. Herold. *Grundlagen der Informatik*. 3., aktualisierte Auflage. It, Informatik. Hallbergmoos: Pearson, 2017. ISBN: 9783868943160.

[37] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. “Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies”. *A Field Guide to Dynamical Recurrent Neural Networks* (2003).

[38] S. Hochreiter and J. Schmidhuber. “Long Short-term Memory”. *Neural computation* 9 (1997). Pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[39] J. J. Hopfield and D. W. Tank. “”Neural” computation of decisions in optimization problems”. *Biological cybernetics* 52.3 (1985). Pp. 141–152. ISSN: 0340-1200. DOI: 10.1007/BF00339943.

[40] H. Huang. *Statistical Mechanics of Neural Networks*. Springer Singapore Pte. Limited, 2021. ISBN: 9789811675706.

[41] D. Jungnickel. *Optimierungsmethoden: Eine Einführung*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2008. ISBN: 9783540767909.

[42] A. Karpatne, W. Watkins, J. S. Read, and V. Kumar. “Physics-guided Neural Networks (PGNN): An Application in Lake Temperature Modeling”. *CoRR* abs/1710.11431 (2017).

[43] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. Dec. 22, 2014. DOI: paper. URL: http://arxiv.org/pdf/1412.6980v9.

[44] S. W. Knox. *Machine learning: A concise introduction*. Wiley series in probability and statistics. Hoboken New Jersey: John Wiley & Sons, 2018. ISBN: 9781119439073.

[45] L. L. Laudis, S. Shyam, V. Suresh, and A. Kumar. “A Study: Various NP-Hard Problems in VLSI and the Need for Biologically Inspired Heuristics”. *Recent Findings in Intelligent Computing Techniques*. Ed. by P. K. Sa, S. Bakshi, I. K. Hatzilygeroudis, and M. N. Sahoo. Singapore: Springer Singapore, 2018, pp. 193–204. ISBN: 978-981-10-8636-6.

[46] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. “Sequencing and scheduling: algorithms and complexity”. *Designing decision support systems notes* Vol. 8903 (1989).

[47] Y. LeCun, C. Cortes, and C. J. C. Burger. *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 08/04/2022).

[48] M. Lombardi and M. Milano. “Boosting Combinatorial Problem Modeling with Machine Learning” (2018). Pp. 1270–1276. DOI: 10.24963/ijcai.2018/177. URL: http://arxiv.org/pdf/1807.05517v1.

[49] B. Meindl and M. Templ. *Analysis of commercial and free and open source solvers for linear optimization problems*. 2012.

[50] G. Mena, D. Belanger, S. Linderman, and J. Snoek. "Learning Latent Permutations with Gumbel-Sinkhorn Networks" (2018). DOI: 10.48550/ARXIV.1802.08665.

[51] A. Milan, S. H. Rezatofighi, R. Garg, A. R. Dick, and I. D. Reid. "Data-Driven Approximations to NP-Hard Problems". *AAAI*. 2017.

[52] L. Moroney. *AI and machine learning for coders: a programmer's guide to artificial intelligence*. 1st edition. Sebastopol, California: O'Reilly Media, Incorporated, 2020. ISBN: 1-4920-7814-X.

[53] B. Naderi, S. Gohari, and M. Yazdani. "Hybrid flexible flowshop problems: Models and solution methods". *Applied Mathematical Modelling* 38.24 (2014). Pp. 5767–5780. ISSN: 0307-904X. DOI: 10.1016/j.apm.2014.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0307904X14001826.

[54] Y. Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$". *Doklady Akademii Nauk SSSR* 269.3 (1983). Pp. 543–547.

[55] F. Ni et al. "A Multi-Graph Attributed Reinforcement Learning Based Optimization Algorithm for Large-Scale Hybrid Flow Shop Scheduling Problem". *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3441–3451. ISBN: 9781450383325. DOI: 10.1145/3447548.3467135.

[56] M. L. Pinedo. "Flow Shops and Flexible Flow Shops (Deterministic)". *Scheduling*. Cham: Springer International Publishing, 2016. ISBN: 9783319265780.

[57] A. Plaat. *Deep Reinforcement Learning*. Singapore: Springer Nature, 2022. URL: 10.1007/978-981-19-0638-1.

[58] P. Priore, A. Gómez, R. Pino, and R. Rosillo. "Dynamic scheduling of manufacturing systems using machine learning: An updated review". *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 28.1 (2014). Pp. 83–97. DOI: 10.1017/S0890060413000516.

[59] M. Raissi, P. Perdikaris, and G. E. Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. 2017. URL: http://arxiv.org/pdf/1711.10561v1.

[60] J. Ren, C. Ye, and F. Yang. "Solving flow-shop scheduling problem with a reinforcement learning algorithm that generalizes the value function with neural network". *Alexandria Engineering Journal* 60.3 (2021). Pp. 2787–2800. ISSN: 1110-0168. DOI: 10.1016/j.aej.2021.01.030. URL: https://www.sciencedirect.com/science/article/pii/S1110016821000338.

[61] Y. Ren, L. Li, X. Yang, and J. Zhou. "AutoTransformer: Automatic Transformer Architecture Design for Time Series Classification". *Advances in Knowledge Discovery and Data Mining*. Ed. by J. Gama, T. Li, Y. Yu, E. Chen, Y. Zheng, and F. Teng. Cham: Springer International Publishing, 2022, pp. 143–155. ISBN: 978-3-031-05933-9.

[62] E. Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[63] E. Rich. "The Gradual Expansion of Artificial Intelligence". *Computer* 17.5 (1984). Pp. 4–12. ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659134.

[64] A. H. Rinnooy Kan. *Machine scheduling problems: classification, complexity and computations*. The Hague, 1976.

[65] A. Rizvanolli. "The ship crew scheduling problem with rest hours constraints". Doctoral Thesis. Technische Universität Hamburg, 2022. URL: http://hdl.handle.net/11420/13835.

[66] S. F. Roselli, K. Bengtsson, and K. Åkesson. "SMT Solvers for Job-Shop Scheduling Problems: Models Comparison and Performance Evaluation". *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. 2018, pp. 547–552. DOI: 10.1109/COASE.2018.8560344.

[67] S. Ruder. *An overview of gradient descent optimization algorithms*. Sept. 15, 2016. DOI: Added. URL: http://arxiv.org/pdf/1609.04747v2.

[68] R. Ruiz and J. A. Vázquez-Rodríguez. "The hybrid flow shop scheduling problem". *European Journal of Operational Research* 205.1 (2010). Pp. 1–18. ISSN: 03772217. DOI: 10.1016/j.ejor.2009.09.024.

[69] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning*. Cambridge: Cambridge University Press, 2014. ISBN: 9781107298019. DOI: 10.1017/CBO9781107298019.

[70] R. Sinkhorn and P. Knopp. "Concerning nonnegative matrices and doubly stochastic matrices". *Pacific Journal of Mathematics* 21 (1967). Pp. 343–348.

[71] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research* 15 (2014). Pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[72] R. Stewart and S. Ermon. *Label-Free Supervision of Neural Networks with Physics and Domain Knowledge*. 2016. DOI: 10.48550/ARXIV.1609.05566.

[73] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. "On the importance of initialization and momentum in deep learning". *Proceedings of the 30th International Conference on Machine Learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR, 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[74] R. S. Sutton. "Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks". *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986.

[75] R. Tavakkoli-Moghaddam, S. Fatemi-Anaraki, D. Abdolhamidi, and B. Vahedi-Nouri. "Integrated Waterway Scheduling, Berth Allocation and Quay Crane Assignment Problem by Using a Hybrid Flow Shop Concept". *2019 International Conference on Industrial Engineering and Systems Management (IESM)*. IEEE, 2019, pp. 1–5. ISBN: 1728115663.

[76] T. Tieleman and G. Hinton. *Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude*.

[77] O. Vinyals, M. Fortunato, and N. Jaitly. "Pointer Networks" (2015). URL: http://arxiv.org/pdf/1506.03134v2.

[78] H. M. Wagner. "An integer linear-programming model for machine scheduling". *Naval Research Logistics Quarterly* 6.2 (1959). Pp. 131–140. DOI: 10.1002/nav.3800060205.

[79] R. Wang and R. Yu. *Physics-Guided Deep Learning for Dynamical Systems: A Survey*. 2021. DOI: 10.48550/ARXIV.2107.01272.

[80] T. Weiss, O. Senouf, S. Vedula, O. Michailovich, M. Zibulevsky, and A. Bronstein. *PILOT: Physics-Informed Learned Optimized Trajectories for Accelerated MRI*. 2019. URL: http://arxiv.org/pdf/1909.05773v5.

[81] F. Yalaoui and C. Chu. "New exact method to solve the $Pm/r_j/\sum C_j$ schedule problem". *International Journal of Production Economics* 100 (2006). Pp. 168–179. DOI: 10.1016/j.ijpe.2004.11.002.

[82] F. S. Yao, M. Zhao, and H. Zhang. "Two-stage hybrid flow shop scheduling with dynamic job arrivals". *Computers & Operations Research* 39.7 (2012). Pp. 1701–1712. ISSN: 0305-0548. DOI: 10.1016/j.cor.2011.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0305054811002942.

[83] M. Zacharias. "Combining heuristics and machine learning for hybrid flow shop scheduling problems". PhD thesis. Duisburg and Essen.

[84] M. Zacharias, A. Tonnius, and J. Gottschling. "Proceedings of the 2019 International Conference on Industrial Engineering and Systems Management (IESM 2019): 25-27 September 2019, Shanghai, China" (2019). DOI: 10.1109/IESM45758.2019. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=8943787.

[85] X. Zhai, A. Oliver, A. Kolesnikov, and L. Beyer. "S^4L: Self-Supervised Semi-Supervised Learning". *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 1476–1485. DOI: 10.1109/ICCV.2019.00156.

DuEPublico

Duisburg-Essen Publications online

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

ub | universitäts bibliothek

Diese Dissertation wird via DuEPublico, dem Dokumenten- und Publikationsserver der Universität Duisburg-Essen, zur Verfügung gestellt und liegt auch als Print-Version vor.

DOI: 10.17185/duepublico/78228
URN: urn:nbn:de:hbz:465-20230517-132200-7

Dieses Werk kann unter einer Creative Commons Namensnennung - Keine Bearbeitungen 4.0 Lizenz (CC BY-ND 4.0) genutzt werden.