Mario Innerkofler, BSc

Learning Knapsack

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Mathematics

submitted to

Graz University of Technology

Supervisor

Eranda Dragoti-Cela, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.

Institut für Diskrete Mathematik

Nikolaus Furian, Ass.Prof. Dipl.-Ing. Dr.techn.

Institut für Maschinenbau- und Betriebsinformatik

Graz, December 2020

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

In this master thesis we take a look at applications of machine learning techniques to the weakly NP-hard 0-1-knapsack problem. Applications of these techniques to combinatorial optimization problems opened a new and well diversified field of research. We explain and demonstrate two entry points for machine learning into the knapsack problem. The first entry point is a direct end-to-end approach where we aim to learn solutions or merely the optimal profit directly from given instances. The second approach is rather indirect and consists in applying machine learning within well established algorithmic frameworks. In this regard we aim to learn node-selection rules used in a branch-and-bound algorithm so as to obtain optimal solutions quickly. We however will show that under mild assumptions the best-first search strategy provides a minimal search tree for a fixed basic variant of a branch-and-bound algorithm. Still a learned node-selection rule can be used in a heuristically truncated branch-and-bound algorithm. The goal of such an algorithm is to find good (or optimal) solutions quickly and skip the algorithmic proof of optimality by only exploring a hopefully small subtree of a respective search tree.

Kurzfassung

In dieser Masterarbeit befassen wir uns mit Anwendungen Maschinellen Lernens auf das schwach NP-schwere 0-1-Rucksackproblem. Die Anwendung dieser Techniken auf kombinatorische Optimierungsprobleme hat ein neues Forschungsfeld eröffnet, welches vielschichtige Ansatzpunkte verfolgt. Wir adressieren und demonstrieren zwei verschiedene Ansatzpunkte, welche dem Maschinellen Lernen Einzug in das Rucksackproblem gewähren. Als erstes wählen wir einen direkten Ansatz, welcher darin besteht, optimale Lösungen, oder als Alternative nur deren Kosten, direkt auf Basis der gegebenen Instanz zu lernen. Ein zweiter, indirekter Ansatz ergibt sich, indem man Techniken des Maschinellen Lernens in etablierte Algorithmen einflechtet. Wir legen das branch-and-bound Regime zugrunde und versuchen, Knotenauswahlstrategien zu lernen, um Optimallösungen schnell zu entdecken. Jedoch können wir zeigen, dass bereits unter schwachen Voraussetzungen in einem speziell konfigurierten branch-and-bound Algorithmus die best-first-seach Strategie einen minimalen Suchbaum liefert. Nichtsdestotrotz können gelernte Knotensuchstrategien in heuristisch abgebrochenen branch-and-bound Verfahren verwendet werden. Jene zielen nämlich nur darauf ab, eine gute, oder sogar optimale, Lösung schnell zu finden und verzichten auf den algorithmischen Beweis der Optimalität, wodurch wir nur einen möglichst kleinen Teil des Suchbaums erkunden wollen.

Acknowledgments

I would like to express my great gratitude towards my parents Daniela and Wilhelm. Only by their generous support I am able to fully enjoy the excellent education offered by TU Graz and KFU Graz. My parents offer any assistance to the greatest extent possible. My equally magnificent brothers Manuel and Stephan deserve a special thanks for providing help, company and support at any time.

Second of all allow me to thank my dear friends and also my dear colleagues from the student's council SIGMA Graz. You made the last years quite memorable, allow me to grow personally and offer any kind of assistance when needed.

Collective thanks also goes to the professors at TU Graz and KFU Graz who I was honored to learn from during the course of my studies. In particular let me point out the outstanding quality of guidance offered by my supervisors professor Dragoti-Cela and professor Furian.

At last I would like to thank Amber Foster for offering linguistic support during writing.

List of Figures

3.1 Handwritten digits; Bishop [6, p.2] 30
3.2 biological neuron; Neves, Gonzalez, Leander Karoumi [30] 46
3.3 ANN neuron; Mitchell, Machine Learning 47
3.4 SNN; example 48
3.5 biological neural network; Neves et al. [30] 49
3.6 unfolding RNN; Goodfellow et al. [13, p.376] 59
3.7 LSTM-cell; Goodfellow et al. [13, p.409] 61
3.8 seq-to-seq RNN; Goodfellow et al. [13, p.378] 64
3.9 seq-to-1 RNN; Goodfellow et al. [13, p.382] 65

4.1 DNNC-pto; PR-curve on test set 77
4.2 DNNC-pto; performance in BB 78

5.1 RNNC-red; item predictions 84
5.2 RNNC-red; wrong item predictions 84
5.3 RNNC-red; sets of perf. pred.; Venn diagrams 91
5.4 TBB; incumbent updates 94

List of Algorithms

1 A branch-and-bound algorithm for KP (BB) 12
2 Generating the full branch-and-bound tree for KP 14
3 Reconstruct the minimal branch-and-bound tree 24

4 Stochastic gradient descent (SGD) 35
5 Stochastic gradient descent with momentum (SGDM) 36
6 Root mean square propagation (RMSprop) 36
7 Adaptive moments (Adam) 37

8 Generating a random instance of KP 74

9 $\rho - r$ local search heuristic 86
10 $\rho - r$ local search hybrid heuristic 88

List of Tables

3.1 Gaussian, Poisson, binomial, and exponential are in EDM 38
3.2 Def, confusion matrix 70

4.1 DNNC-pto; architecture 75
4.2 PTO-prediction, evaluation on test set 76
4.3 PTO-prediction, performance in BB 80

5.1 profit prediction model architectures 82
5.2 RNNC-red; architecture 83
5.3 RNNC-red; bit error 83
5.4 RNNC-red; classification report 85
5.5 RNNC-red; confusion matrix 85
5.6 solution prediction heuristics, bit-errors 89
5.7 TBB, duration 93
5.8 profit-prediction; DNN-full, DNN-red, RNN-full, RNN-red 96
5.9 profit-prediction; RNNC-red 96
5.10 profit-prediction; local search (hybrid) 97
5.11 profit-prediction, TBB 98

Contents

1 Introduction **1**
1.1 Machine learning meets combinatorial optimization 1
1.2 The contribution and outline of this elaboration 2
1.3 Basics and notations 3

2 The knapsack problem **6**
2.1 The 0-1-knapsack problem (KP) 6
2.2 Solving the 0-1-knapsack problem 7
2.2.1 Greedy and linear relaxation KP 8
2.2.2 Branching rules for KP 10
2.2.3 Branch-and-bound algorithm for KP 11
2.2.4 Node-selection rules 18
2.2.5 Optimal node-selection rules 19
2.2.6 Performance and phases of a branch-and-bound algorithm 25

3 Introduction to machine learning **29**
3.1 Notations for data 31
3.2 Modeling in a machine learning context 33
3.2.1 Stochastic gradient descent methods 34
3.3 Regression 38
3.4 Linear classification 40
3.4.1 Binary logistic regression 42
3.4.2 Multi logistic regression 44
3.5 Artificial neural networks (ANN) 45
3.5.1 (Simple) neurons 46
3.5.2 Single layer neural networks (SNN) 47
3.5.3 Feed-forward neural networks (FFNN) 51
3.5.4 Recurrent neural networks (RNN) 57
3.6 Other modeling techniques 65
3.7 Assessing the quality of models 67
3.7.1 Assessing the fit of regression models 68
3.7.2 Assessing the fit of classification models 68

4 Learning node selection rules **73**
4.1 Training and test data 74
4.2 Learning PTO 75
4.3 Using PTO prediction within BB 77

5 ML-aided heuristics for KP **81**
5.1 Training data and test data 82
5.2 Learning optimal profit 82
5.3 Learning solutions 82
5.3.1 Local search heuristic 86
5.4 Truncated branch-and-bound algorithm for KP 92
5.5 Evaluation of predicted profits 95

6 Implementation **99**
6.1 Setting up a branch-and-bound algorithm 99
6.2 Setting up the ML-tasks 100
6.2.1 Learning profits and solutions 100
6.2.2 Learning node-selection rules 102
6.3 Real time component 102
6.4 Heuristics 103

7 Conclusion **104**

Chapter 1

Introduction

In the recent past, the field of machine learning has risen to significant popularity far beyond the mathematics and computer science circles. The term machine learning seemingly spreads a mystical aura and reports of superhuman capabilities as evident in almost any kind of media outlet. Indeed the capabilities and, more importantly, the ways in which we use these techniques seem to justify the adoration of this data driven tool. May it be the support provided to medical personnel for disease diagnostics, the ability to accurately model the human behavior, or maybe only a funny facemask being perfectly rendered on any user's image – machine learning is present in our everyday lives.

1.1 Machine learning meets combinatorial optimization

The machine learning paradigm increasingly pierces through the shell of established mathematical work. In this elaboration we integrate machine learning further into the field of combinatorial optimization. Not long ago (2018) Bengio, Lodi and Prouvost announced that this rendezvous is strongly believed to be the beginning of a new era in the field [5]. Our small piece of contribution concerns the knapsack problem which is one of the most prominent and well researched combinatorial optimization problems. Bengio et al. proposed several entry points for machine learning into a combinatorial optimization context [5]. One is via ***end-to-end methods*** which directly take an instance as input, do some machine learning, and propose a solution. Another approach is ***learning properties*** of a given problem which help us to solve the problem more efficiently. For example, we may aim to predict the best suited solution method. The third lever presented is using a ***model within a well known (potentially exact) technique*** serving as a framework. It follows the goal of learning decisions to be made within the algorithm to find better (or optimal) solutions (more quickly). For instance Balcan, Dick, Sandholm and Vitercik discussed learning variable selection rules within a fixed branch-and-bound framework such that a small branch-and-bound tree is produced while leaving all other parameters untouched [2].

1.2 The contribution and outline of this elaboration

In the following we will present techniques which follow the fixed framework approach and which follow the end-to-end approach.

Concerning the fixed framework approach, we also move in the context of branch-and-bound and ***focus on the node-selection policy***. He, Hal and Jason discuss such learning rules which try to reach optimal nodes as quickly as possible [14]. In the case of the knapsack problem however, we will constructively prove that under relatively weak additional assumptions on the instance and when using a particular variant of a branch-and-bound algorithm, the best-first-search strategy is optimal with regard to producing small trees. This is mainly due to the fact there is an lp-solution for the knapsack problem which admits only one fractional variable giving rise to a trivial branching rule. It allows for establishing a global upper bound (w.r.t. set inclusion) for all branch-and-bound trees which contain any conceivable branch-and-bound tree all with respect to a fixed instance. To the best of our knowledge such a result has not been available yet.

Moreover we will show that such an optimal selection policy has some room for mistakes and needs not find an optimal node in the quickest possible way. It will turn out that the best-first-search strategy fully exploits this room for mistakes and hence is the worst optimal node selection in this regard. Spinning on this result we still can make use of ***learned node-selections in a heuristically truncated branch-and-bound algorithm***. It follows the novel goal of quickly detecting optimal nodes for knapsack and serves us as an entry point into the field of heuristics.

In this field we will furthermore introduce some examples of ***end-to-end methods***. Some of these methods provide us with an associated solution while others simply estimate the profit disregarding the interpretation of packing items. Martini [29] demonstrated that this can work well for knapsack, but we will encounter certain obstacles which we will address. For example, by using a local search heuristic on the prediction we are able to construct an input better than the greedy solution on average.

Hopefully the reader is now excited for a journey into the intersection of machine learning and combinatorial optimization. Throughout the first chapters we will present the basics and some notable results in the respective fields. Later on, we will focus on the modeling tasks and evaluate our findings. At the very end we will permit a glimpse into the technical realization of such a machine learning project which may help a beginner getting started with the minimal amount of frustration necessary.

1.3 Basics and notations

Notations and conventions

Definition 1.1.

(1) By $[n] := \{k \in \mathbb{N} \mid 1 \le k \le n\}$ we denote the ***discrete interval*** containing all integers between 1 and n.

(2) Recall the convention of setting $\max \varnothing := -\infty$ and $\min \varnothing = \infty$

(3) For vectors $x \in \mathbb{R}^n$ we denote x_i the i-th component of x. Elements in $\mathbb{R}^n$ are column-vectors. For matrices $A \in \mathbb{R}^{m\times n}$ we denote A_j the j-th column (unless specified differently) and by A_{ij} the i-j-th entry. By $e_i \in \mathbb{R}^n$ we denote the i-th unit basis vector $e_i = (0, \dots, 0, 1, 0, \dots, 0)$ where a 1 is at the i-the position.

(4) For a set X we denote by 2^X the ***power set*** of X being the set of all subsetes of X. By $\binom{X}{k}$ we denote the elements in 2^X having cardinality k. By $X^k := X \times \dots \times X$ we denote the k-fold cartesian product of X being the set of k-tuples of X.

Directed and undirected graphs

Definition 1.2. ***Undirected graphs***

(1) Given an ***undirected regular graph*** $G = (V, E)$. The vertex set or node set of a specified graph is denoted by $V(G) = [n]$ and the edge set is denoted by $E(G) \subseteq \binom{V}{2} = \{\{v_i, v_j\} \mid v_i, v_j \in V, i \neq j\}$. For vertices v and edges e we may shorthand write $u \in G$ and $e \in G$ meaning $u \in V(G)$ and $e \in E(G)$. The empty graph $(\varnothing, \varnothing)$ is simply but sloppily denoted by $\varnothing$. We measure the size of a graph by the number of vertices and denote it by $|G| = n$.

(2) The ***neighborhood*** $N_G(u)$ of a node u is defined by $N(u) = \{v \in V(G) \mid \{u, v\} \in E(G)\}$ being a set of nodes adjacent to u. If the underlying graph is clear from the context we shorthand denote the neighborhood by $N(u) := N_G(u)$. The degree d_u of a node u in the graph is defined as the cardinality of its neighborhood $|N(u)|$.

(3) Given another graph H the ***subgraph inclusion*** $H \subseteq G$ denotes $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. For subsets of vertices $U \subseteq V(G)$ the graph induced on U by G is defined by $G[U] = (U, E(G[U]))$ where $E(G[U]) := \{\{u, v\} \mid u, v \in U \wedge \{u, v\} \in E(G)\}$.

Definition 1.3. ***Directed graphs***

(1) A ***regular directed graph*** $D = (V(D), E(D))$ is defined by a vertex set $V(D) = [n]$ and a set of directed edges $E(D) \subseteq V(D)^2 \backslash \{(v, v) \mid v \in V(D)\}$.

(2) A ***path*** in sequence notation is defined by by $P = (v_1, \dots, v_k)$ and the path is defined as a directed graph by $P = (V(P), E(P)) := (\{v_1, \dots, v_k\}, \{(v_i, v_{i+1}) \mid 1 \le i \le k-1\})$. The first and respectively last nodes v_1, v_k are called the start and end of P and P is called a v_1-v_k-path. The number of edges $k-1$ in P defines the length $k-1$ of the path.

(3) Two directed graphs H, G are contained as ***subgraphs*** $H \subseteq G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If one of G, H, say H, is an undirected graph, the subgraph inclusion $H \subseteq G$ is defined by disregarding the orientation of edges meaning we identify $(u, v) \sim \{u, v\}$. First and foremost, a path P is said to be in an undirected graph G if P is a path with $P \subseteq G$.

(4) Finally, this allows us to define the ***distance of vertices*** u, v in G as the minimal length of an u-v path in G or ∞ if no u-v-path exists by convention.

Definition 1.4. ***Rooted trees***

(1) A ***tree*** T is an undirected, acyclic, and connected regular graph.

(2) A root $r \in V(T)$ is a specialized vertex in a tree and thus defines a ***rooted tree*** $T_r = (V(T), E(T), r)$. Demanding the existence of a root in a rooted tree implies that every rooted tree is non-empty. If T_r consists only of one vertex we call T_r trivial. In rooted trees a direction of edges $e = (u, v)$ is naturally defined by arranging the tuple (u, v) such that the ***distance*** (i.e. number of edges) between u and r is less than the distance between v and r. u is called the ***ancestor*** or predecessor of v and v is called the ***descendant*** or child of u. Distinct nodes having the same ancestor are called siblings. The ***depth of a node*** v is defined as the distance to the root r

(3) A tree T'_r is called a ***rooted subtree*** of a rooted tree T_r if and only if $T' \subseteq T_r$ and $r \in T'$ is defined as the root of $T' := T'_r$.

(4) Paths starting in the root r are called ***trajectories***. Given two trajectories which both contain a common node but each respective trajectory contains a distinct child of the node. Then the common ancestor is called the fork of the trajectories.

(5) ***Leaves in rooted trees*** are defined as nodes having no descendants. With this convention, a trivial rooted tree T_r has a root of degree zero which is also a leaf. If T_r is non-trivial the set of leaves is exactly the set of degree-one vertices. Nodes which are not leaves are called inner nodes.

Definition 1.5. ***(Canonically rooted) binary trees***
A tree $T = (V(T), E(T))$ is called a ***binary tree*** if it either consists of a single vertex or if it consists of a vertex and two edges respectively ending in two new binary trees. Binary trees can be regarded as ***canonically rooted trees*** as follows. In trivial binary trees the root is defined as the only vertex in the tree, and in non-trivial binary trees the root is defined as the unique vertex having degree two. Unless specified differently we from now on always identify binary trees as canonically rooted trees. The unique canonical root allows using notation $T = (V(T), E(T))$ also for canonically rooted binary trees. Having this canonical rooted binary tree available, we can equivalently define binary rooted trees as a rooted tree where each node has either two or zero descendants.

Function spaces

Definition 1.6. ***Space of continuous functions***
By $C(X) = \{f : X \to \mathbb{R} \mid f \text{ continuous}\}$ we denote the vector space of continuous functions on a subset X of a finite-dimensional Banach-space. For $f \in C(X)$ we define the supremum norm by $||f||_X = \sup_{x\in X} |f(x)|$.

Definition 1.7. L^p***-spaces***

(1) We define the L^p-norm of a function $f : \mathbb{R}^m \to \mathbb{R}$ as $||f||_{p,\mu} = (\int_{\mathbb{R}^m} |f(x)|^p d\mu)^{\frac{1}{p}}$.

(2) The L^p-space is then defined as $L^p_\mu(\mathbb{R}^m) = \{f : \mathbb{R}^m \to \mathbb{R} \mid ||f||_{p,\mu} < \infty\}$

Definition 1.8. ***dense subsets***
We call a subset $S \subseteq Q$ dense in the metric space (Q, d) if for all $q \in Q$ and each $\epsilon > 0$ there is an $s \in S$ with $d(q, s) < \epsilon$. If $(Q, ||.||)$ is a normed space we use the induced metric $d(x, y) = ||y - x||$.

Special functions and properties

Definition 1.9. ***Sigmoidal function***
We call a function $f : \mathbb{R} \to \mathbb{R}$ sigmoidal if $\lim_{x\to\infty} f(x) = 1$ and $\lim_{x\to-\infty} f(x) = 0$

Definition 1.10. λ***-increasing and*** λ***-decreasing functions***
Given some $\lambda > 0$. We call a function $f : (a, b) \to \mathbb{R}$ a λ- (strictly)-increasing function if there exists $u : (a, b) \to \mathbb{R}$ (strictly) increasing with $|f(x) - u(x)| \leq \lambda \ \forall x \in (a, b)$. If, in contrast, u is (strictly) decreasing, we call f a λ-(strictly) -decreasing function .

Definition 1.11. ***The sigmoid function*** σ
Let $\sigma : \mathbb{R} \to (0, 1)$ be defined by $\sigma(x) = \frac{1}{1+e^{-x}}$. This function is also called the sigmoid function, the logistic function or also the logistic sigmoid function. Images $\sigma(x) \in (0, 1)$ under this function can be interpreted as a probabilities.

Definition 1.12. ***Softmax***
The softmax is a well known function and defined as follows.

$$\begin{aligned} softmax &: \mathbb{R}^k \to (0,1)^k \\ (y_1, \dots, y_k) &\mapsto \Big(\frac{\exp(y_1)}{\sum_{i=1}^k \exp(y_i)}, \dots, \frac{\exp(y_k)}{\sum_{i=1}^k \exp(y_i)} \Big) \end{aligned}$$

By construction $||softmax(y_1, \dots, y_k)||_1 = 1$ and thus each image can be interpreted as a probability vector.

Chapter 2

The knapsack problem

Knapsack problems are among the most popular and most investigated combinatorial problems. The ***typical setup for a knapsack problem*** is a given finite set of items where each item has an associated profit and an associated weight. The goal is to find a composition of the items which maximizes the sum of the profits while the sum of weights does not exceed a certain weight threshold. Variants may require certain bundles of items being packed or allow packing items multiple times. Because of the popularity of knapsack problems, the available literature treating this topic is vast. For example Martello and Toth [28], Kellerer, Pferschy, and Pisinger [22] or Martello, Pisinger and Toth [27] provide literature which exclusively treats Knapsack Problems in great detail and generality. We base this section on the excellent book on combinatorial optimization by Korte and Vygen [25, p.459 ff.].

Concerning the ***hardness of knapsack problems***, Karp showed that already a very simple version of a Knapsack Problem is (weakly) NP-hard [21]. This makes knapsack problems candidates for experimentation with heuristics and approximation algorithms. Indeed for the simple version being mentioned Ibarra and Kim (1975) constructed an approximation algorithm of arbitrary accuracy ϵ running in polynomial time $O(n^2\frac{1}{\epsilon})$ where n is the number of items, see Korte et al. [25, p.464 ff.]. Not only are knapsack problems interesting from an academic perspective but also in practice they find many applications. Let us formally introduce the simple version of a knapsack problem and let us discuss some properties relevant for our further proceedings.

2.1 The 0-1-knapsack problem (KP)

We are given a set of items having associated a respective weight and a profit. The task in the 0-1-knapsack problem KP is to determine a combination of items which maximize the sum of profits while the sum of weights does not exceed a given capacity.

Definition 2.1. ***0-1-knapsack problem (KP)***
Let $c_1, \dots, c_n \in \mathbb{R}_{>0}$, $w_1, \dots, w_n \in \mathbb{R}_{>0}$ be profits, respectively weights of items $[n]$, $n \in \mathbb{N}$. Furthermore let $W \in \mathbb{R}_{>0}$ be a given capacity of the knapsack.

(1) Let the ***objective function*** be denoted by $c(x) := \sum_{i=1}^{n} c_i x_i = c^t x$ and denote the weight function by $w(x) := \sum_{i=1}^{n} w_i x_i = w^t x$. The 0-1-knapsack problem is defined by the task of solving the following integer program.

$$c^* := \max\{c^t x \text{ s.t. } w^t x \leq W,\ x \in \{0,1\}^n\} \tag{KP}$$

The constraints $x_i \in \{0,1\}$, $1 \leq i \leq n$, are called ***integrality constraints*** and $w(x) \leq W$ is called the capacity or ***weight constraint***. While the optimal profit c^* is unique for any given instance an optimal solution x^* and its weight $w^* := w(x^*)$ may not be unique.

(2) An element $x \in \{0,1\}^n$ is called a ***binary vector*** or a packing. Whenever a binary vector x satisfies the capacity constraint $w(x) \leq W$ we call it feasible or (an) integral (solution/packing). An item i is said to be ***included***, packed or contained in a packing x if $x_i = 1$ and otherwise we say i is ***excluded*** or not contained. Vectors $x \in [0,1]^n$ having at least one component $x_i \in (0,1)$ are called fractional.

(3) Denote by $b_i := \frac{c_i}{w_i}$ the ***benefit*** of the respective items i, $1 \leq i \leq n$.

Permanent Assumption 1.
From this point onward we assume the following without loss of generality. For any given instance of KP the items are labeled according to $b_1 \geq \cdots \geq b_n$, the profits are scaled by the maximum providing $\max_{1 \leq i \leq n} c_i = 1$ and the weights are scaled by the capacity allowing us to assume $W = 1$. Furthermore we may assume $w_i \leq W,\ 1 \leq i \leq n$ or equivalently $\max_{1 \leq i \leq n} w_i \leq W$ because items i having $w_i > W$ can not occur in any feasible solution and thus can be discarded. In this way all weights are in $(0,1]$ and reflect the percentage of the consumed capacity W if packed.

2.2 Solving the 0-1-knapsack problem

The ***branch-and-bound method*** is a very generally defined method for solving hard combinatorial optimization problems. In order to apply it to a certain problem we need to specify the individual components which define such a method.

First of all, we require an ***initial feasible solution***. In the case of the 0-1-knapsack problem this is a binary vector which obeys the weight constraint and therefore offers a lower bound on the optimal objective function value. Throughout a branch-and-bound algorithm, the set of feasible solutions is split repeatedly by applying a so-called ***branching rule*** on the problem. In order to decide whether one shall further apply branching a method for computing ***upper bounds*** is required. A sufficiently small upper bound shows there is no optimal solution in the considered subset and we no longer consider the subproblem. In this case we say the subproblem is fathomed or pruned. There are also further cases in which subproblems can be pruned. We can apply pruning if an optimal solution of the subproblem which is integral is found or if no integral feasible solution exist for the subproblem. After pruning and branching a ***node-selection rule*** is applied which tells which subproblem is considered next.

We define commonly used methods which serve the above purposes with the goal of defining a ***basic version of a branch-and-bound algorithm*** for solving KP. This concrete version is referred to as BB. As mentioned the hardness of the 0-1-knapsack problem was first shown by Karp and it justifies to proceed with the introduction of a branch-and-bound method [21].

2.2.1 Greedy and linear relaxation KP

The concept of ***constructing a greedy-solution*** is a general method being described by incrementally improving an objective by the locally best option. In regard to KP the greedy method provides a solution in linear time which approximates the optimal solution up to a factor of 2, see Korte et al. [25, p.462]. This solution which is called the greedy solution plays an important role in the initialization of a branch-and-bound algorithm which we will present.

Conversely the ***linear relaxation of integer programs*** is a common technique applied in order to obtain upper bounds. A linear relaxation embeds the feasible integral points in a convex subset of a real vector space. It allows for finding upper bounds via methods like linear programming corresponding to possibly fractional vectors. Perhaps the most natural way to perform such an embedding is by simply dropping the integrality constraints of the considered integer program. Hence this particular relaxation is referred to as the ***natural linear programming relaxation*** of an integer program. The reason for looking at upper bounds in the context of integer programming is the aim to prove that certain subproblems do not contain an optimal solution. This event justifies to discard the considered subproblem by applying a prune by bound which is one of several pruning rules. Smaller upper bounds are wishful in this regard since they lead to more sensitive pruning.

Definition 2.2. ***Greedy solution***
Given an instance of KP.

(1) The greedy-solution x^{greedy} is defined by going through the items ordered starting from the most to the least beneficial where an item is included if it fits.

$$(x^{greedy})_i = 1 \text{ if } \sum_{j=1}^{i-1} x_j^{greedy} w_j + w_i \leq W \text{ and } (x^{greedy})_i = 0 \text{ else, } 1 \leq i \leq n$$

A greedy-solution can be computed in $O(n)$ via the weighted median as it is pointed out by Korte et al. [25, p.462]. A straight-forward sorting method provides the greedy solution in $O(n\log(n))$ and for small instances this may be preferable due to its simplicity. In particular if the items are already sorted accordingly this simple method also takes $O(n)$ time.

(2) The profit and weight of the greedy solutions $c^{greedy} := c(x^{greedy})$, $w^{greedy} := w(x^{greedy})$ are called the ***greedy-profit*** respectively the ***greedy-weight***. An instance is called ***greedy-optimal*** if the greedy solution is optimal $c^* = c^{greedy}$.

Definition 2.3. ***Critical item***
Given an instance of KP where $\sum_{i=1}^n w_i > W$ meaning it is not feasible to pack all items. In such instances a ***critical item*** $k \in [n]$ is defined and the associated variable x_k is called the ***critical variable***. By packing the items in a greedy fashion the critical item is given by $k := \min\{j \in [n] | \sum_{i=1}^j w_i > W\}$. It is the first item which does not fit in the partly packed knapsack. Note that for any fixed ordering of items k according to non-increasing benefits the greedy solution is unique. However, from the point of a given instance KP the ordering and thus also the greedy-solution is not unique in case two benefits coincide.

Definition 2.4. ***Linear relaxation of the 0-1-knapsack problem (LP)***
By dropping the integrality constraints we obtain the natural linear programming relaxation LP for a given instance of KP as already mentioned above. In this way, we formally define the relaxed problem LP also called the ***fractional knapsack problem*** as in Korte et al. [25, p.459].

$$c^{lp} := \max\{c^t x \text{ s.t. } w^t x \leq W,\ x \in [0,1]^n\} \tag{LP}$$

Lemma 2.5. ***Solving LP***
Given an instance of KP. A solution to the natural linear programming relaxation can be determined directly without using commonly known methods for solving linear programs like the simplex method or inner point methods.

(1) If $W < \sum_{i=1}^n w_i$ let k be the critical item. Let $x_{\leq} = (1, \dots 1, 0 \dots 0)$ be the packing of all items $\{1, \dots, k-1\}$ and denote by $c_{\leq}, w_{\leq}$ the associated profit and weight respectively. By x^f we denote the (fractional) packing which packs a proportion $x_k^f := \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} \in [0,1)$ of the critical item k and no other item $x_i^f := 0, i \neq k$. Then according to ***Dantzig (1957)*** a solution to LP is given by $x^{lp} := x^{\leq} + x^f$ [12]. Again c^{lp}, w^{lp} denote the associated profit and weight respectively.

    Note that in the considered case clearly $W = w^{lp}$. In this way x^{lp} packs items in the greedy fashion and takes the largest possible fraction of the critical item k subject to feasibility. Furthermore, note that $x_{\leq}$ is feasible providing a lower bound not larger than the greedy profit. If $x_k^f = 0$ then x^{lp} is an optimal packing (in particular integral) for KP.

(2) If $W \geq \sum_{i=1}^n w_i$ we define $x^{lp} := (1, \dots, 1)$ the all-ones solution and clearly it is optimal for both KP and LP.

(3) We call an instance of KP integral or ***lp-optimal*** if x^{lp} is an integral solution. In this case $x^{lp} = x^{greedy}$ and in particular the optimal profit is provided by both greedy and lp solution $c^{greedy} = c^* = c^{lp}$. Therefore lp-optimality implies greedy-optimality. The converse is only true if $W \geq \sum_{i=1}^n w_i$ or if $x_k^f = 0$. Having fixed an ordering of the items we call KP fractional whenever it is not integral. Note that by Permanent Assumption 1 instances KP are always feasible and x^{lp} can be constructed.

(4) Computing x^{lp} can be done in $O(n)$ by distinguishing the cases necessary to construct x^{lp} and in the appropriate case we determine the critical item by weighted median as in Definition 2.2. Similarly a straight-forward method provides x^{lp} in $O(n \log n)$ and in case the items are sorted, we merely require $O(n)$.

2.2.2 Branching rules for KP

Generally the partitioning of the feasible set into several subsets is called branching. Balcan et al. [2] showed that branching rules are a ***powerful leverage point*** for reducing the computational cost of branch-and-bound algorithms. There does not exist a generally preferable branching rule and a variety of strategies and rules is available.

When engineering a branching rule, typically a ***trade off*** between the number of subsets in the partitions, the number of times a particular subset needs to be partitioned until a pruning rule applies, and the computational cost for calculating the partition is encountered. All these parameters may largely influence the performance of a branch-and-bound algorithm.

One popular and simple branching rule for general integer programs introduces combinations of inequalities $x_k \leq \lfloor x_k^{lp} \rfloor$ and $x_k \geq \lceil x_k^{lp} \rceil$ to define the subsets comprising the partition. Thereby x_k is a fractional variable and $x_k^{lp} \in \mathbb{R} \backslash \mathbb{Z}$ is its value in an lp-solution. By the close relation of the partitioning schema to variables we may say we ***branch on variables***. In different branching rules often times a close analogy to other structures of the problem can be drawn and analogous terminology is used.

For KP we will explicitly define and use this natural branching rule. For a binary integer program these inequalities reduce to ***fixing fractional variables to zero, respectively one***. Since KP not only can be formulated as a binary integer program, but also has an lp-solution x^{lp} with at most one fractional variable (see Lemma 2.5), this branching rule reduces to fixing x_k to zero, respectively one defining two subsets.

Definition 2.6. ***Knapsack problem with fixed item set***
Consider an instance of KP as in Definition 2.1. Kolpakov and Posypkin introduced the following useful framework allowing for a neat definition of the branching-rule being used for the 0-1-knapsack problem [24, p.100].

(1) Let $I \subseteq [n]$ be a set of ***fixed items*** and accordingly define the set of ***unfixed items*** $N := [n] \backslash I$. For fixed items we are given constants $\theta_i \in \{0,1\}$, $i \in I$. For items $i \in I$ a value $\theta_i = 1$ reflects the demanded inclusion of item i while $\theta_i = 0$ reflects its exclusion. By conventionally setting $\theta_i = 0 \; \forall i \in N$ a binary vector $\theta \in \{0,1\}^n$ is defined. Together with I the vector θ defines which items are fixed and unfixed and which items in the given I are included or excluded. Therefore the tuple (θ, I) uniquely determines the feasible set of a subproblem $\mathrm{KP}_{I,\theta}$ of the original instance.

(2) By $w_{I,\theta} := \sum_{i \in I} \theta_i w_i$, $c_{I,\theta} := \sum_{i \in I} \theta_i c_i$ we denote the ***fixed weight*** respectively the ***fixed profit*** of the subinstance $\mathrm{KP}_{I,\theta}$. The quantity $W - w_{I,\theta}$ is called the residual or ***spare capacity***.

(3) The subproblem $\mathrm{KP}_{I,\theta}$ is defined by optimizing the same ***objective function*** over the smaller feasible sets as follows.

$$c^*_{I,\theta} := \max\{c^t x \text{ s.t. } w^t x \le W,\ x_i \in \{0,1\}\ \forall i \in N, x_i = \theta_i\ \forall i \in I\} \qquad (\mathrm{KP}_{I,\theta})$$

(4) The corresponding ***linear relaxation*** $\mathrm{LP}_{I,\theta}$ is given as follows.

$$c^{lp}_{I,\theta} := \max\{c^t x \text{ s.t. } w^t x \le W,\ x_i \in [0,1]\ \forall i \in N, x_i = \theta_i\ \forall i \in I\} \qquad (\mathrm{LP}_{I,\theta})$$

(5) The optimal solution of $\mathrm{KP}_{I,\theta}$ is denoted by $x^*_{I,\theta}$ and the optimal solution of $\mathrm{LP}_{I,\theta}$ is denoted by $x^{lp}_{I,\theta}$ The respectively associated weights are denoted by $w^*_{I,\theta} := w(x^*_{I,\theta})$, $w^{lp}_{I,\theta} := w(x^{lp}_{I,\theta})$ and the respectively associated profits are denoted by $c^*_{I,\theta} := c(x^*_{I,\theta})$, $c^{lp}_{I,\theta} := c(x^{lp}_{I,\theta})$.

(6) When fixing items we can identify a potentially infeasible fractional 0-1-knapsack problem which we call the ***unfixed part*** of $\mathrm{KP}_{I,\theta}$.

$$\max\{\sum_{i \in N} c_i x_i \text{ s.t. } \sum_{i \in N} w_i x_i \le W - w_{I,\theta},\ x_i \in \{0,1\}\ \forall i \in N\}$$

By adding the fixed profit and by appropriately inserting the respective variable values in a vector we reconstruct the optimal profit and the associated optimal solution of $\mathrm{KP}_{I,\theta}$ from the respective unfixed part.

(7) Regarding a solving method for $\mathrm{KP}_{I,\theta}$ we may encounter infeasibility of the unfixed part and thus also of $\mathrm{KP}_{I,\theta}$ if $W - w_{I,\theta} < 0$. This situation can be detected easily and for feasible unfixed parts we can ***apply the same results and assumptions*** as stated for the usual KP. In particular, Lemma 2.5 allows for constructing lp-optimal solutions in polynomial time. We say $\mathrm{KP}_{I,\theta}$ is infeasible if the unfixed part is infeasible. We call feasible instances of $\mathrm{KP}_{I,\theta}$ integral and fractional if the unfixed part has the corresponding property. Similarly the critical item of $\mathrm{KP}_{I,\theta}$ is the critical item of the unfixed part which exists only if not all unfixed items fit in the unfixed instance. Otherwise the optimal solution is easily given by including all unfixed items.

2.2.3 Branch-and-bound algorithm for KP

In the following, we compose the previously defined components into a specific variant of a branch-and-bound algorithm, referred to as BB, for solving KP. For the short time remaining until we formally define the act of selecting subproblems in Definition 2.17 we treat this aspect of the branch-and-bound algorithm BB as a blackbox.

Permanent Assumption 2.
From this point onward we restrict ourselves to instances of KP where all benefits are pairwise distinct and items obey the unique labeling according to $\frac{c_1}{w_1} > \cdots > \frac{c_n}{w_n}$.

Definition 2.7. ***A branch-and-bound algorithm for KP***
The following algorithm based on the general branch-and-bound schema presented in Korte et al. solves an instance of KP [25, p.584].

Algorithm 1: A branch-and-bound algorithm for KP (BB)

Data: KP
Result: c^* optimal profit

1 Initialize a timer $t = 1$;
2 Compute c^{greedy} the greedy profit of KP and set the incumbent $c^{inc} := c^{greedy}$;
3 Initialize the set of active nodes $\mathcal{N}_1 = \{KP\}$.;
4 **while** $\mathcal{N}_t \neq \varnothing$ **do**
5 Select a node $KP_t = KP_{I,\theta} \in \mathcal{N}_t$ and form the current tree T_t as in Definition 2.8 ;
6 **if** $KP_{I,\theta}$ *is infeasible* **then**
7 ***prune by infeasibility:***
8 pass ;
9 **else if** $KP_{I,\theta}$ *is integral* **then**
10 **if** $c^{lp}_{I,\theta} > c_{inc}$ **then**
11 ***prune by new incumbent:***
12 Update $c^{inc} := c^{lp}_{I,\theta}$
13 **else**
14 ***prune by integrality:***
15 pass ;
16 **else if** $KP_{I,\theta}$ *is fractional* **then**
17 **if** $c^{lp}_{I,\theta} > c_{inc}$ **then**
18 ***branch:***
19 Get $k := k_{I,\theta}$ the critical item of $KP_{I,\theta}$;
20 Define subproblems $KP_{I\cup\{k\},\theta}$ (exclusion), $KP_{I\cup\{k\},\theta+e_k}$ (inclusion) ;
21 Add the new active nodes $\mathcal{N}_t := \mathcal{N}_t \cup \{KP_{I\cup\{k\},\theta}, KP_{I\cup\{k\},\theta+e_k}\}$
22 **else**
23 ***prune by bound:***
24 pass ;
25 Next round active nodes $\mathcal{N}_{t+1} = \mathcal{N}_t \backslash \{KP_{I,\theta}\}$;
26 Increment the timer by one $t++$;
27 return $c^* := c_{inc}$

Definition 2.8. ***branch-and-bound tree***
Given an instance of KP.

(1) All branch-and-bound algorithms naturally define a rooted binary tree T called a ***branch-and-bound tree***. KP=KP_1 defines the root and all encountered subproblems define the vertex-set. Two nodes are connected by an edge if one subproblem

was obtained from the other by branching. The leaves in the branch-and-bound tree are called pruned nodes and the inner nodes are called branched nodes.

(2) The node at which $c^{inc} := c^*$ is updated is called the ***optimal node***. The corresponding time is denoted by t^*. The trajectory to this node is called the ***path to the optimal node (PTO)*** and it may consist only of one node in case the initial solution is optimal.

(3) If we refer to the ***set of active nodes*** $\mathcal{N}_t$ at a time t we more precisely mean $\mathcal{N}_t$ before having picked a new (current, active) node $K_t :=\text{KP}_{I,\theta} \in \mathcal{N}_t$ and before having potentially added new nodes by branching. Furthermore, we say the current node KP_t is processed at time t. The act of processing a node refers to deciding if the branching rule or an according pruning rule is being applied to KP_t.

(4) At all times $1 \leq t \leq N$ in the algorithm we ***track rooted subtrees*** $T_t = T[KP_1, \ldots, KP_t] \subseteq T$ in an online fashion by adjoining the current node to the node which generated KP_t by branching. At time $t = 1$ only the root KP was processed thus far and it forms T_1. The sequence of trees T_t, $1 \leq t \leq N$ satisfies $T_1 \subseteq \cdots \subseteq T_N = T$ where each tree T_t differs from T_t by the node KP_{t+1} which is processed at the respective time $1 \leq t \leq N-1$. Trees of the form T_t are called (branch-and-bound) trees at time t or current (branch-and-bound) trees in case t is clear from the context.

Lemma 2.9. ***Branch-and-bound trees generated by BB are well defined***
For a given instance KP the branch-and-bound tree defined as above in Definition 2.8 is well defined if it generated by BB. This means at any run of BB with the same parameters provides the same branch-and-bound tree.

Proof.
Recall that by the Permanent Assumption 2 we have determined a unique labelling of items according to $b_1 > \cdots > b_n$. Therefore the solution of subinstance $\text{LP}_{I,\theta}$ constructed according to Lemma 2.5 is uniquely determined. In particular the critical item $k = \min\{j \in [n] \mid \sum_{i=1}^{j} w_i > W$ as defined in Definition 2.3 is uniquely determined. In this way, the integrality of a node is well defined while the feasibility is well defined regardless of the Permanent Assumption 2. Therefore, any subproblem $\text{KP}_{I,\theta}$ can be associated with a well defined state of being integral, fractional, or infeasible. By this level of uniqueness, we achieve that subproblems which are generated by branching are uniquely determined with regard to a given instance KP. In particular, because of the Permanent Assumption 2 the unique ordering of items provides a unique sequence of critical items of potential branching candidates along any trajectory.

It is left to verify that the same set of nodes is branched in every run of BB. At time $t = 1$ the incumbent c^{inc} is unique by an application of the greedy method to KP. Since a potential critical item k of KP is unique in particular a subsequent set of active nodes $\mathcal{N}_2$ is uniquely determined. In an inductive fashion suppose that both the incumbent at time $t-1$ and the set of active nodes generated for the next round $\mathcal{N}_t$ are uniquely determined. Then the node-selection rule selects the same node $\text{KP}_{I,\theta}$ being uniquely identified by

(I, θ) in every run of BB. Therefore when processing $\mathrm{KP}_{I,\theta}$ at a time $2 \leq t \leq N$ we obtain the same well defined state of $\mathrm{KP}_{I,\theta}$ and in particular potential application of the branching rule is determined by KP. □

Definition 2.10. ***Full branch-and-Bound tree***
Given an instance of KP. We define the ***full branch-and-bound tree*** T_{full} algorithmically as follows.

Algorithm 2: Generating the full branch-and-bound tree for KP

Data: KP
Result: T_{full}
1 Define T_{full} as one vertex KP being the root;
2 **while** *T_{full} has a fractional leaf* **do**
3   Get a fractional node with no children $\mathrm{KP}_{I,\theta}$;
4   Get $x_k^{lp} \in (0,1)$ its fractional variable.;
5   Connect two newly generated nodes $KP_{I \cup \{k\}, \theta}$, $KP_{I \cup \{k\}, \theta + e_k}$ to $\mathrm{KP}_{I,\theta}$ by an edge ;
6 return T_{full} ;

We denote the vertex-set of T_{full} by $\mathcal{N}_{full} := V(T_{full})$ and call it the ***set of admissible nodes*** for KP. Note that analogously to Lemma 2.9 the tree T_{full} is well defined. In particular, we point out that subproblems $\mathrm{KP}_{I,\theta}$ generated as defined above are obtained according to branching on items $i \in I$. Items in I became critical at the according subproblem along the trajectory to $\mathrm{KP}_{I,\theta}$ in an ordering specified merely by KP.

Lemma 2.11. ***All branch-and-bound trees grow in*** T_{full}
Let KP be an instance of knapsack. Let T be an arbitrary branch-and-bound tree generated by using BB and an arbitrary node selection rule. Then T_{full} is well defined and $T \subseteq T_{full}$.

Proof.
The result is proven analogously to Lemma 2.9. □

Lemma 2.12. ***Structure of*** T_{full}
For a given instance KP the full branch-and-bound tree T_{full} is a binary tree. All inner nodes of T_{full} are fractional nodes and all leaves of T_{full} are either infeasible or integral.

Proof.
Clearly T_{full} is a binary tree since we append either two or zero new descendants to nodes. By construction we thus obtain a leaf $\mathrm{KP}_{I,\theta}$ if and only if we do not append new descendants which is the case only if the node is not fractional. The first case for a non-fractional instance $\mathrm{KP}_{I,\theta}$ is infeasibility. In case of feasibility $x^{lp_{I,\theta}}$ exists. However $x^{lp_{I,\theta}}$ must be an integral packing since we did not append new descendants. □

Remark 2.13.

(1) T_{full} is generated ***analogously to a branch-and-bound tree*** according to Definition 2.8 by a slight alteration. Instead of performing a prune by bound, we apply branching, and the case of pruning by a new incumbent is integrated to the case of pruning by integrality. Thus T_{full} can be obtained by BB if we assume the initial solution to be the all-zero solution. We then select nodes particularly poorly by avoiding integral nodes as long as possible in order to avoid pruning by bound.

(2) The above structural result 2.12 motivates thinking of applying BB as ***growing trees*** $T_1 \subseteq \cdots \subseteq T_N = T \subseteq T_{full}$ into a large binary tree T_{full}. T_{full} is the universal branch-and-bound tree associated with a worst case performance of BB applied on KP. In fact, the existence of such a universal tree structure for KP is merely reasoned by a trivial branching rule which receives a uniquely determined branching variable by Permanent Assumption 2. In particular, T_{full} is ***independent*** of the choice of initial solutions, the choice of upper bounds, and of the node-selection rule. More notably, branch-and-bound trees T generated analogously to BB but by using alternatives of all parameters except for the branching rule which still satisfies $T \subseteq T_{full}$ and most implications still apply in a similar fashion.

(3) When using the branching-rule as defined in Definition 2.7 we can only avoid exploring the entirety of T_{full} by ***pruning trajectories by bound***. The application of a prune by bound depends on the quality of upper bound and on the quality of the current incumbent. Regarding the generation of upper bounds as a fixed parameter throughout the algorithm, we may at runtime only leverage the node-selection in order to increase c^{inc} quickly.

(4) In variants of branch-and-bound algorithms for solving other combinatorial problems there might be ***further degrees of complication*** and a tree analogous to T_{full} may not be attainable. For instance having several fractional variables at a node or having non-binary integral variables opens questions about defining such a full tree. It may well be the case that a suitable assumption which provides us well defined branch-and-bound trees only based on a given instance does not exist or at least is not as straight forward to come up with. In particular, objects associated with integral variables which are indistinguishable are considered problematic. For KP we already revealed but successfully resolved this issue. In case of the strong assumption of unique relaxed solutions, this issue can also be resolved. In this instance, there is potential for several candidate branching variables to form a uniquely determined set. From this point onward, it is conceivable for a definition of T_{full} to be achieved by adding all possible branching variants in the form of a descendant. In case we choose to compute upper bounds via natural linear programming relaxations, the next step of proving that indistinguishable subproblems can not be formed in a cyclic fashion is straightforward as upper bounds decrease along trajectories by uniqueness. In the case of KP we are also able to prove this behavior without assuming unique lp-solutions quite technically as follows. Avoiding constants lp-bounds along trajectories will become a crucial detail in the proofs of further results.

Lemma 2.14. ***Strict monotonicity of trajectories in*** T_{full}
Given an instance KP and an inner node $KP_{I,\theta}$ of T_{full}. Then the lp-bound at a descendant $KP'_{I,\theta}$ of $KP_{I,\theta}$ is strictly smaller than the lp-bound at $KP_{I,\theta}$.

Proof.
Without loss of generality we identify $KP_{I,\theta}$ as KP. We prove this result by a fully fledged case distinction. Since KP is fractional, there exists a critical item $k \in [n]$ and the critical variable defined by Lemma 2.5 is $(x^{lp})_k \in (0,1)$. The descendant which we denote by $\mathrm{KP}_{I,\theta}$ may be infeasible, or in case of feasibility all or no one item might fit. If $\mathrm{KP}_{I,\theta}$ is infeasible however, the inequality is clear. So suppose $\mathrm{KP}_{I,\theta}$ is feasible meaning it has a non-negative capacity $0 \leq W - \theta_k w_k$.

(1) If $\theta_k = 0$ we have $1 = (x^{lp})_i = (x^{lp}_{I,\theta})_i,\ 1 \leq i \leq k-1,\ (x^{lp}_{I,\theta})_k = 0$. If $k = n$ we get $(x^{lp}_{I,\theta})_n = 0 < (x^{lp})_i \in (0,1)$ providing a strict decrease of lp-profit. So suppose $k < n$ and by $(x^{lp})_k w_k > 0$ some $(x^{lp}_{I,\theta})_i > 0, k+1 \leq i \leq n$. We express the residual weight $W - \sum_{i=1}^{k-1} w_i = x^{lp}_k w_k \geq \sum_{i=k+1}^{n} (x^{lp}_{I,\theta})_i w_i > 0$ estimated by feasibility of $x^{lp}_{I,\theta}$. We subtract the common summands $\sum_{i=1}^{k-1} c_i$ from both bounds and compare equivalently by using descending benefits as follows.

$$c_k x^{lp}_k = c_k \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} = b_k(W - \sum_{i=1}^{k-1} w_i) \geq b_k(\sum_{i=k+1}^{n} (x^{lp}_{I,\theta})_i w_i) =$$
$$\sum_{i=k+1}^{n} (x^{lp}_{I,\theta})_i b_k w_i > \sum_{i=k+1}^{n} (x^{lp}_{I,\theta})_i b_i w_i = \sum_{i=k+1}^{n} (x^{lp}_{I,\theta})_i c_i$$

(2) If $\theta_k = 1$ we obtain an instance with capacity $0 \leq W - w_k$ by having assumed feasibility of $\mathrm{KP}_{I,\theta}$. By $0 \leq W - w_k$ we in particular have $2 \leq k$. The characterization of the critical k item provides k minimal such that $\sum_{i=1}^{k} w_i > W$. Therefore $\sum_{i=1}^{k-1} w_i > W - w_k$ and the critical item in $\mathrm{KP}_{I,\theta}$ exists $1 \leq k_{I,\theta}$ and occurs earlier $k_{I,\theta} \leq k-1$. We thus have a solution of $\mathrm{KP}_{I,\theta}$ of the form $(x^{lp}_{I,\theta}) = (1,\ldots,1,f_{I,\theta},0,\ldots 0,1,0\ldots,0)$ where $f_{I,\theta} = \frac{W - w_k - \sum_{i=1}^{k_{I,\theta}-1} w_i}{w_{k_{I,\theta}}} \in [0,1)$ at position $k_{I,\theta}$ and 1 at position k for including k. The all-zero and all-one vectors could potentially be of length zero. We estimate the difference in profit as follows.

$$c^{lp} - c^{lp}_{I,\theta} =^{(i)} [\sum_{i=k_{I,\theta}}^{k-1} c_i + b_k(W - \sum_{i=1}^{k-1} w_i)] - [b_{k_{I,\theta}}(W - w_k - \sum_{i=1}^{k_{I,\theta}-1} w_i) + c_k] =^{(ii)}$$

$$\sum_{i=k_{I,\theta}}^{k-1} c_i + w(b_k - b_{k_{I,\theta}}) - (b_k - b_{k_{I,\theta}})(\sum_{i=1}^{k_{I,\theta}-1} w_i) - (\sum_{i=k_{I,\theta}}^{k-1} b_k w_i) + b_{k_{I,\theta}} w_k - c_k =^{(iii)}$$

$$\sum_{i=k_{I,\theta}}^{k-1} c_i + (b_k - b_{k_{I,\theta}})(W - w_k - \sum_{i=1}^{k_{I,\theta}-1} w_i) > 0 \iff$$

$$\sum_{i=k_{I,\theta}}^{k-1} c_i > (b_{k_{I,\theta}} - b_k)(W - w_k - \sum_{i=1}^{k_{I,\theta}-1} w_i) =^{(iv)} (b_{k_{I,\theta}} - b_k) w_{k_{I,\theta}} f_{I,\theta} =$$

$$c_{k_{I,\theta}} f_{I,\theta} - b_k w_{k_{I,\theta}} f_{I,\theta}$$

In step (i) we cancel the equal terms in both solutions, plug in the expressions for the fractional variables computed according to Lemma 2.5 and shall not forget c_k, the fixed profit. In step (ii) we factor $(b_k - b_{k_{I,\theta}})$ from common summands. In step (iii) we write $c_k = b_k w_k$ and thus, can factor out an additional $(b_k - b_{k_{I,\theta}})$. We look at the equivalent inequality and in step (iv) note that this is the weight-margin in $KP_{I,\theta}$ admitting the expression in the critical variable $f_{I,\theta} \in [0,1)$. By positivity of profits and weights and the fractional variable being in $[0,1)$ we can easily verify $\sum_{i=k_{I,\theta}}^{k-1} c_i \geq c_{k_{I,\theta}} > c_{k_{I,\theta}} f_{I,\theta} - b_k w_{k_{I,\theta}} f_{I,\theta}$.

□

Lemma 2.15. ***Arriving at integral nodes***
Consider an instance of KP and the full branch-and-bound tree T_{full}. Consider a node $KP_{I,\theta}$ which was constructed from its predecessor by fixing the critical variable k. We assume without loss of generality the predecessor of $KP_{I,\theta}$ is KP by looking at the unfixed part.

(1) If $KP_{I,\theta}$ is obtained by fixing $x_k = 1$ then $KP_{I,\theta}$ is integral only if $\sum_{i=1}^{k_{I,\theta}-1} w_i = W - w_k$. This corresponds to the spare capacity being filled exactly by items $\{1, \dots k_{I,\theta} - 1\}$.

(2) If $KP_{I,\theta}$ is obtained by fixing $x_k = 0$ then $KP_{I,\theta}$ is integral only if either $\sum_{i=1}^{k_{I,\theta}-1} w_i = W - w_k$ or if $\sum_{i \neq k}^{n} w_i \leq W$ meaning all unfixed items fit with respect to the residual capacity still being W by $x_k = 0$.

Proof.
The proof of this result is already present along the lines of the proof of Lemma 2.14. Part of the reversed implication of (2), namely the fact that unfixed instances provide an integral solution in case all unfixed items fit is already shown in Lemma 2.5. From the point of search strategies it may be helpful to know scenarios which enable finding integral nodes since only then we can detect an optimal node. □

Lemma 2.16. ***Finding x^{greedy} in T_{full}***
Given an instance of KP and the associated T_{full}. In T_{full} consider a node which is found when traversing from the root along the trajectory in correspondence to excluding the critical item attained at each subproblem. Then the leaf at the end of this trajectory is an integral node with $c^{lp}_{I,\theta} = c^{greedy}_{I,\theta} = c^{greedy}$ admitting the greedy profit with $x^{greedy} = x^{greedy}_{I,\theta} = x^{lp}_{I,\theta}$ even via the greedy solution.

Proof.
Let x^{greedy} be the greedy solution. If the root is integral we have an lp-optimal instance and the statement is trivial by not branching and Lemma 2.5. So suppose the root is fractional and thus a critical item k exists. We show the statement iteratively by moving along the considered trajectory. Clearly $x^{greedy}_k = 0$ in the greedy solution by definition 2.2. We therefore exclude k and do not decrease the capacity by construction of subproblems. Hence the greedy-solution at the descendant coincides with the greedy solution at the root since the same items in the unfixed part fit and $\theta_k = 0$ ensures also the fixed items assume the same variable values. This is an invariant in this procedure and in particular the greedy solution is always contained hindering infeasibility. Therefore the leaf is integral and Lemma 2.15 applies. The scenarios which result in integrality are either that all unfixed items fit or that the residual capacity exactly balances the fitting unfixed items. □

2.2.4 Node-selection rules

Let us formally introduce the procedure of selecting nodes. We will then define a notion of an optimal node-selection for setting up a ***baseline comparison*** for node-selections. In fact we are able to prove the optimality of the well known best-first-search strategy for solving KP by using BB as defined in Definition 2.7.

Definition 2.17. ***Node-selection rule***
Given an instance KP and the full branch-and-bound tree T_{full}. In order to formally define node-selection rules we construct a family of ***node-valued functions***. These functions take a current branch-and-bound tree and a set of active nodes as input and map to some particular node in the set of active nodes.

(1) Let T be a rooted subtree of T_{full} which is not the full tree. The ***neighborhood of the rooted subtree*** T (in T_{full}) is defined as follows.

$$\mathcal{N}_T := \{u' \in T_{full} \backslash T \mid u' \text{ descendant of some } u \in T\}$$

(2) Consider tuples (T, N_T) where T is a rooted subtree of T_{full} and $\emptyset \neq N_T \subseteq \mathcal{N}_T$ is a subset of the neighborhood of T. We collect all such pairs in a set as follows.

$$\mathcal{T}_{\mathcal{N}} = \bigcup_{KP \ni T \subsetneq T_{full}} \{(T, N_T) \mid \emptyset \neq N_T \in 2^{\mathcal{N}_T}\}$$

$\mathcal{N}_T$ reflects the maximal set of active nodes if T is a current branch-and-bound tree. Therefore $\mathcal{T}_{\mathcal{N}}$ reflects ***all combinations of a potential current branch-and-bound tree and a respective set of active nodes***. Note that in particular a

proper subset $N_T \subseteq \mathcal{N}_T$ could form a set of active nodes if some fractional nodes in $\mathcal{N}_T \subseteq T_{full}$ were pruned by bound. Furthermore note that it is justified to exclude empty sets N_T since in this case BB has already terminated.

(3) A function $\mathcal{S} : \mathcal{T}_\mathcal{N} \to \mathcal{N}_{full}$ which satisfies $\mathcal{S}(T, N_T) \in N_T \ \forall (T, N_T) \in \mathcal{T}_\mathcal{N}$ is called a ***node-selection rule***. We conventionally extend the domain of $\mathcal{S}$ by setting $\mathcal{S}(\varnothing, \{KP\}) := KP$. This allows us to apply $\mathcal{S}$ for selecting the root KP at time $t = 1$. Note that a node-selection rule has available all information about the current tree T and has available all information about the active nodes N_T in order to select some $\text{KP}_{I,\theta} \in N_T$.

Examples of node selection rules

Typically node-selection rules are defined implicitly by assuming arbitrary sets of active nodes and arbitrary current trees. Oftentimes, a tree-search strategy motivates incorporating the structure of the current tree into a node-selection rule. Another typical approach is selecting active nodes based on quantities such as associated bounds. In practice, a trade off between the number of explored nodes and the computational cost entailed by canny node-selection rules is expected.

Definition 2.18. ***Depth-first-search (DFS)***
We pop the last element from a stack of active nodes $\mathcal{N}_t$ at any time t. Newly generated pairs of nodes are pushed on the stack such that the preferred problem is on top. If we push the problem which fixed the critical variable of the predecessor to one we call the rule ***DFS-inclusion***. When reversing the preference we refer to it as ***DFS-exclusion***.

Definition 2.19. ***Depth-first-search with restart (DFS-w-r)***
We define an additional rule to DFS. If the current node KP_t at time t is pruned, we select an active node KP_{t+1} from $\mathcal{N}_{t+1}$ uniformly at random in the next round. In this way, we ***restart the search*** at a random point in the tree whenever we run into a leaf of the branch-and-bound tree resulting from BB at termination. In the context of search strategies such an element is called diversification.

Definition 2.20. ***Best-first-search (BESTFS)***
We choose an active node KP_t having a ***maximal lp-bound*** c_t^{lp} among all active nodes $\mathcal{N}_t$. We prioritize integral nodes among nodes having the same bound and use an arbitrary (potentially even random) tie break in case we only have available non-integral nodes.

2.2.5 Optimal node-selection rules

Having defined a formal concept of a node selection rules $\mathcal{S}$ we treat it as a parameter for BB and fix all other parameters according to Definition 2.7. In the following, we then aim to ***minimize the number of processed nodes*** by tuning this parameter. It turns out the BESTFS is a minimizer and that minimal trees are unique. Along

the way we prove that each branch-and-bound tree generated according to BB contains unavoidable nodes regardless of the node-selection, see Lemma 2.27. The number of these nodes directly results from the structure of a given instance and our method of choice for computing upper bounds. Having available this unavoidable set of nodes we will prove the optimality of BESTFS in Theorem 2.31 by showing that only these unavoidable nodes are explored. Finally, in Lemma 2.33 we present a straight-forward method allowing us to reconstruct the minimal branch-and-bound tree from any given branch-and-bound tree.

Definition 2.21. ***Optimal node-selection rule***

(1) A node-selection rule $\mathcal{S}^*$ which for a fixed instance KP results in a tree of minimal size among all node selection rules $\mathcal{S}$ when using BB is called an optimal node selection rule for KP.

(2) The corresponding branch-and-bound tree is called minimal for KP. If a node selection rule is optimal for every instance of KP it is simply called optimal and the corresponding branch-and-bound trees are called minimal.

Definition 2.22. ***Closure of a tree***
Consider an instance of KP and the corresponding full branch-and-bound tree T_{full}. The closure of a rooted subtree $T \subseteq T_{full}$ is defined as the induced subgraph $\overline{T} := T_{full}[V(T) \cup \mathcal{N}_T]$ where $\mathcal{N}_T$ is the neighborhood of T as defined in Definition 2.17. We conventionally define the closure of the empty tree $\overline{\emptyset} := (\{KP\}, \emptyset)$ as the rooted subtree containing only the root.

Lemma 2.23. ***Closures are binary trees***
Given KP and a rooted subtree $T \subseteq T_{full}$ or $T = \emptyset$. Then $T \subseteq \overline{T} \subseteq T_{full}$ and $\overline{T}$ is a binary tree.

Proof.
Obviously $T \subseteq \overline{T} \subseteq T_{full}$ holds. Clearly the closure $\overline{T}$ of T is a tree since it stays connected by extending T by connecting vertices and it remains acyclic since the closure is contained in a tree $T \subseteq \overline{T} \subseteq T_{full}$. We proceed by showing each vertex either has zero or two descendants. In case T is empty, the closure $\overline{T}$ consists of only the root and is a binary tree by definition. In the non-empty case, consider a node v in T. Suppose v has descendants v', u' in T_{full} which is a binary tree by 2.12. Descendents which are not yet present in T are added by closing T. If v has no descendants in T_{full} it also has no descendants in $\overline{T}$ as $T \subseteq T_{full}$ and $\overline{T} \subseteq T_{full}$. □

Definition 2.24. ***Admissible nodes and truncating*** T_{full}
Given KP and the corresponding full tree T_{full}. Let $c \in \mathbb{R} \cup \{-\infty, +\infty\}$ be a constant. Analogous to the set of all admissible nodes $\mathcal{N}_{full}$ we define $lp_{full} := \{c^{lp}_{I,\theta} \mid KP_{I,\theta} \in \mathcal{N}_{full}\}$ the ***multi-set of all admissible lp-bounds*** of KP. We define truncations of these (multi-) sets as follows.

(1) We define the set of all ***admissible nodes truncated at*** c as follows.

$$\mathcal{N}(c) := \{KP_{I,\theta} \in \mathcal{N}_{full} \mid c^{lp}_{I,\theta} > c\} \subseteq \mathcal{N}_{full}$$

(2) We define the ***full tree truncated at*** c as follows.

$$T(c) := T_{full}[\mathcal{N}(c)] \subseteq T_{full}$$

(3) We define multi-sets of ***admissible lp-bounds truncated at*** c as follows.

$$lp(c) := \{c^{lp}_{I,\theta} \mid KP_{I,\theta} \in \mathcal{N}(c)\} \subseteq lp_{full}$$

Obviously all truncated sets are empty if and only if $c \geq c^{lp}$. If $c = -\infty$ we obtain $\mathcal{N}(-\infty)$, $T(-\infty)$ and $lp(-\infty)$ directly corresponding to all feasible nodes in T_{full}.

Lemma 2.25. ***Structure of truncated trees***
Given KP and a possibly infinite fixed constant $c \in \mathbb{R} \cup \{-\infty, +\infty\}$. If $c < c^{lp}$ the full tree truncated at c is a rooted subtree of T_{full}. Furthermore, truncated trees get smaller when the constant grows meaning for any two constants $c_1 \leq c_2 \in \mathbb{R} \cup \{-\infty, +\infty\}$ we obtain $T(c_2) \subseteq T(c_1)$ (rooted) subtree containment.

Proof.
If $c < c^{lp}$ at least the root is contained in $\mathcal{N}(c)$. If a node v is in $T(c)$ its predecessor has an lp-bound larger than c^{lp} by Lemma 2.14 and thus also is a node of $T(c)$. Since $T(c) = T_{full}[\mathcal{N}(c)]$ is induced on T_{full} all nodes are connected to the root and $T(c)$ is a tree. Similarly for constants $c_1 \leq c_2$ we have $\mathcal{N}(c_2) \subseteq \mathcal{N}(c_1)$ and we obtain rooted-tree containment $T_2 \subseteq T_1$ by again being graphs induced on T_{full}. □

Definition 2.26. ***Unavoidable tree***
Given an instance of KP with optimal profit c^* and full branch-and-bound tree T_{full}. Define $T_U := \overline{T(c^*)} \subseteq T_{full}$ the unavoidable rooted subtree. Nodes in T_U are called unavoidable nodes. More compactly we simply refer to these nodes as unavoidable and call T_U unavoidable.

Lemma 2.27. ***The unavoidable tree is contained in all branch-and-bound trees***
Given KP and a branch-and-bound tree $T \subseteq T_{full}$ obtained by BB at termination by using an arbitrary node selection rule $\mathcal{S}$. Then $T_U \subseteq T$ regardless of the node-selection rule $\mathcal{S}$ used in BB.

Proof.
In case $\mathcal{N}(c^*) = \varnothing$ in particular the root satisfies $c^{lp} \leq c^*$. Since an optimal solution exists, Lemma 2.14 implies the root is the optimal node and the instance is lp-optimal meaning c^* is admitted at the root and the corresponding solution x^{lp} is integral. In this case, the unavoidable tree consists merely of the root and is a rooted binary subtree.

So suppose $\mathcal{N}(c^*) \neq \varnothing$. First we prove $T(c^*) \subseteq T$ is contained in the branch-and-bound tree T as a rooted subtree. We extend the containment to the closure $\overline{T(c^*)} = T_U$ later. Assume to the contrary there exists a node $s \in T \backslash T(c^*)$. By $s \in T \cap T(c^*)$ the node s is not the root and there exists a node $t \in T \cup T$ with a descendant $t' \in T \backslash T(c^*)$. Therefore t is a fractional and must have been pruned by bound when generating T. Therefore, the lp-bounds at t satisfies $lp_t \leq c_{inc} \leq c^*$ for an incumbent c^{inc} available at the time t was processed. But then $t \notin \mathcal{N}(c^*)$ is a contradiction.

We proceed by showing that even the closure $\overline{T(c^*)} = T_U$ of $T(c^*)$ is contained in T. So given a node in $s \in T(c^*) \subseteq T$, we must show that if a node $s \in T(c^*) \subseteq T$ has two descendants in T_{full} it also has two descendants in T. Then by Lemma 2.11 the descendants in T coincide with descendants in $\overline{T(c^*)} \subseteq T_{full}$ and $\overline{T(c^*)} \subseteq T$ follows. If s has descendants in T_{full} the node s is a fractional node by 2.12 and by $s \in T(c^*)$ the lp-bound $lp_s > c^*$. At the time $s \in T(c^*) \subseteq T$ is processed in BB it is neither pruned by integrality nor does it provide a new incumbent by being fractional. By $lp_s > c^*$ the node s can not be pruned by bound because incumbents are bounded $c^{inc} \leq c^*$ and s is not infeasible by both having descendants in T_{full} and by admitting $lp_s > c^*$. Therefore s was branched at the time it was processed and indeed has two descendants in T. □

Remark 2.28.
In summary Lemma 2.23 and in particular Lemma 2.27 show the unavoidable tree $T_U = \overline{T(c^*)}$ is a binary rooted subtree of T_{full}. It is contained in any branch-and-bound tree regardless of the node-selection when using BB. If we can prove that T_U is attained as a branch-and-bound tree when using BB by finding a suitable node-selection, we constructively prove that T_U is a minimal branch-and-bound tree.

Lemma 2.29.
Given an instance of KP which is solved according to BB by using BESTFS as a node-selection rule. Denote by $t^*-1 := |\mathcal{N}(c^*)|$ and consider an ordering of respective lp-bounds $c_1^{lp} \geq \cdots \geq c_{t^*-1}^{lp} > c^*$ in $lp(c^*)$. The next smaller node in lp_{full} is $c_{t^*}^{lp} := c^*$. Denote by $\tilde{c}_t^{lp}$ the lp-bound of the node processed at time $1 \leq t \leq t^*$ when using BESTFS as a node-selection rule in BB. Then $\tilde{c}_t^{lp} = c_t^{lp}$, $1 \leq t \leq t^*$. In particular, at time t^* an integral node which admits the optimal profit c^* is detected.

Proof.
We show the statement by induction. At time $t = 1$ the root is processed and has a maximal lp-bound c_1^{lp}. Let $2 \leq t \leq t^* - 1$ and suppose at time $t-1$ the statement holds. We have processed a contingent of $t-1$ nodes which admit the $t-1$ largest lp-bounds. Since nodes can be processed only once, the next chosen node must satisfy $\tilde{c}_t^{lp} \leq c_t^{lp}$. Assume to the contrary $\tilde{c}_t^{lp} \neq c_t^{lp}$ and therefore $\tilde{c}_t^{lp} < c_t^{lp}$. Thus there exists $v \in \mathcal{N}_{full} \backslash \{\mathcal{N}_t \cup V(T_{t-1})\}$ an unprocessed node which is not active having an lp-bound $c_v^{lp} := c_t^{lp}$. On the trajectory to v in T_{full} we choose $s' \in T_{full} \backslash T_{t-1}$ a node with minimal distance to the root. Then s' is not the root and the predecessor $s \in T_{t-1}$ of s' admits $c_s^{lp} > c_{s'}^{lp} \geq c_v^{lp}$ by Lemma 2.14. Since $s \in T_{t-1}$ the node s has already been processed and

$c_s^{lp} > c^* \geq c^{inc}$. Thus, at the time s was processed, we branched on s and have available s' as an active node. However $c_{s'}^{lp} \geq c_v^{lp}$ contradicts $\tilde{c}_t^{lp} < c_t^{lp} = c_v^{lp}$.
In order to prove the "in particular" statement, let $t = t^*$ and let s' be an integral node admitting $c_{t^*}^{lp} = c^*$. If $t^* = 1$ we have $\mathcal{N}(c^*) = \emptyset$ and the statement is already shown in Lemma 2.27. If $t^* \geq 2$ the root is not an optimal but there exists an optimal node s' having a predecessor s with a strictly larger bound. By the above $s \in T_{t^*-1}$ is explored while and has a strictly larger bound than its descendant $c_s^{lp} > c_{s'}^{lp}$ by Lemma 2.14. By the above s' is not explored at time $t^* - 1$. However, s' is in the set of active nodes because s is fractional and $s \in T(c^*)$ harbors an lp-bound larger than any incumbent making it impossible to prune by bound. Hence BESTFS has an integral node available admitting c^* which is preferably chosen. □

Remark 2.30.
Note that while all admissible lp-bounds larger than c^* are encountered in descending order when using BESTFS it must not necessarily be the case that the associated nodes correspond.

Theorem 2.31. ***BESTFS is an optimal node-selection rule***
Let T be a branch-and-bound tree constructed by using BESTFS for an arbitrary instance of KP. Then $T = T_U$ and therefore T is a minimal branch-and-bound tree or equivalently BESTFS is an optimal node-selection. In particular T_U is the unique minimal branch-and-bound tree.

Proof.
If we can show $T \subseteq T_U$ we have shown the statement since $T_U \subseteq T$ by Lemma 2.27.
Assume to the contrary there exists $s' \in T \setminus T_U$. s' is not the root and assume without loss of generality s' has minimal distance to the root. Then its predecessor satisfies $s \in T \cap T_U$. T_U is a binary tree and therefore the sibling of s' is also not in T_U. Hence s is a leaf in T_U. However s has a descendant s' in T_{full} and is therefore feasible and fractional. We aim to show $c_s^{lp} \leq c^*$. If this was not the case, $s \in T(c^*)$ and when closing $T(c^*)$ the descendant s' would have been added, and we would obtain a contradiction to assuming $s' \notin T_U = \overline{T(c^*)}$. Therefore, indeed $c_s^{lp} \leq c^*$ and we apply Lemma 2.29. Thus at the time s is processed we have available c^* as incumbent. Hence s is pruned by bound providing $s' \notin T$, a contradiction. In order to prove the uniqueness we suppose a distinct minimal branch-and-bound tree T^*. By Lemma 2.27 $T_U \subseteq T^*$ and whenever there is a node in $T^* \setminus T_U$ the minimality is obstructed. □

Remark 2.32.
One way to construct the minimal branch-and-bound tree is to use BESTFS as a node-selection rule in BB. However when having constructed a branch-and-bound tree with a different node selection already we still are able to reconstruct T_U in a simple and fast way as follows.

Lemma 2.33. ***Reconstruct the minimal branch-and-bound tree***
Given and instance KP, let T be any branch-and-bound tree for KP and let c^* be the optimal profit.

(1) $T_U = \overline{T(c^*)}$ can be reconstructed by applying the following algorithm.

Algorithm 3: Reconstruct the minimal branch-and-bound tree

Data: T branch-and-bound tree, c^* optimal profit
Result: T_U minimal branch-and-bound tree
1 Find $\mathcal{N}(c^*)$ by filtering the nodes of T by lp-bounds;
2 Compute the induced subtree $T[\mathcal{N}(c^*)] = T(c^*)$;
3 Compute the closure $\overline{T(c^*)}$ by appending missing descendants to the feasible fractional vertices of $T(c^*)$;
4 return $\overline{T(c^*)}$

---

(2) The number of nodes of $\overline{T(c^*)}$ is given by $|\overline{T(c^*)}| = 2|T(c^*)| + 1$.

(3) For any rooted subtree $T \subseteq T_{full}$ the inequality $|\overline{T}| \leq 2|T| + 1$ holds.

Proof.

Ad (1) Since $T(c^*) \subseteq T_U \subseteq T$ is contained in any branch-and-bound tree we detect the entirety of $\mathcal{N}(c^*)$ within T by filtering. We can compute the closure in a straight-forward method by directly applying the definition.

Ad (2) If $T(c^*)$ is empty, the statement is clear by convention. In the non-empty case we observe $\mathcal{N}(c^*)$ consists only of fractional nodes which are inner nodes in T_{full} by 2.12. Therefore, they are also inner nodes of the tree $\overline{T(c^*)}$ by definition. We recall $\overline{T(c^*)}$ is a binary tree, see Lemma 2.23. Thus each node in $T(c^*)$ must have a degree of 3 in the closure except for the root which has degree 2 in the closure. We directly compute the difference of the respective sizes by $|\overline{T(c^*)}| - |T(c^*)| = -1 + \sum_{i=1}^{|T(c^*)|}(3 - d_i) = 3|T(c^*)| - 1 - \sum_{i=1}^{|T(c^*)|} d_i = 3|T(c^*)| - 1 - 2(|T(c^*)| - 1) = |T(c^*)| + 1$. Rearranging the term immediately yields the result. Note that we used the well known fact that the sum of degrees of vertices in a tree is twice the number of edges and the number of edges in trees is one less than the number of nodes.

Ad (3) By closing an arbitrary rooted subtree, $T \subseteq T_{full}$ each node in $\overline{T}$ has a degree of at most 3. We analogously compute $|\overline{T}| - |T| \leq |T| + 1$ and prove the claim.

□

Remark 2.34.

(1) The above algorithm in Lemma 2.33 reconstructs the minimal branch-and-bound tree T_U in $O(|T|)$ from any given branch-and-bound tree T for a fixed instance KP. The minimal tree (size) is of interest for ***assessing a node-selection rule***. In case we are only interested in the size $|T_U| = |\overline{T(c^*)}|$ we merely need to determine about

half the nodes of T_U, namely the nodes in $T(c^*)$. We can then use part (2) of the above Lemma 2.33 and directly compute $|T_U|$.

(2) As an alternative to the above algorithm we can obtain T_U by solving KP with BB and by using BESTFS. Let us briefly analyze the running time and ***justify the use of the above algorithm***. At each node we need to solve the linear relaxation which takes $O(n)$, see Lemma 2.5. We may use a Fibonacci heap to keep track of the set of active nodes. In the first $|T(c^*)|$ iterations, two elements are added per round (in constant time) by branching and one element is deleted (in logarithmic time) by selecting an active node. By using Stirling's formula and Lemma 2.33 we estimate $\sum_{t=1}^{|T(c^*)|} \log t \sim |T(c^*)| \log |T(c^*)| \sim |T_U| \log |T_U|$. This term reflects the running time spent merely on keeping track of the active nodes. In total,we have running times of order $\Omega(|T_U|(\log(|T_U|) + n))$ for solving KP with BESTFS and BB. In particular, if $|T| = O(|T_U|(\log(|T_U|) + n))$ is relatively small, the above algorithm is preferable for computing the minimal branch-and-bound tree T_U. In case we only aim for the size $|T_U|$ we can save an additional factor of 2 as opposed to fully determining T_U.

2.2.6 Performance and phases of a branch-and-bound algorithm

In order to assess the performance of a general branch-and-bound algorithm in a more granular fashion, we define ***two phases*** during the algorithm. In the exploration we have not yet detected an optimal node. In the validation phase an optimal node is found and the optimality is verified. This can be done for an arbitrary branch-and-bound algorithm for any given problem but in our case we focus on BB and KP. The duration (or lenght) of the respective phases and the performance of BB are measured by counting the number of processed nodes (attributed to the respective phase).

In Theorem 2.36 we show one way to ***estimate the whole duration*** of BB, given an instance of KP. We obtain a lower bound which is only based on the number of nodes processed in the exploration phase and on the size of the unavoidable tree giving us insight into the validation phase of BB. In this way we can show the size of the generated branch-and-bound tree is bounded by a function which is affine linear in the duration of the exploration phase and the intercept is determined by the number of unavoidable nodes. In a second result formulated in Corollary 2.38 we revisit the use of BESTFS in BB and are able to ***explicitly disclose the phases*** based on the corresponding tree truncated at c^*.

Definition 2.35. ***Phases in BB***
For a given instance of KP consider a branch-and-bound tree T which is generated by employing an arbitrary node-selection rule in BB. Denote by t^* the time when the optimal node was detected as defined in Definition 2.8 and denote by KP_t the subproblem processed at time t. Furthermore, let T_{full} be the full branch-and-bound tree for KP. We divide the set of explored nodes $V(T)$ in two disjoint sets called phases.

(1) We respectively define the ***exploration-phase*** $\mathcal{N}_E$ and the ***validation-phase*** $\mathcal{N}_V$ as follows.

$$\mathcal{N}_E = \{\mathrm{KP}_t \mid t < t^*\}, \mathcal{N}_V = \{\mathrm{KP}_t \mid t \geq t^*\}$$

In this way, the node KP_{t^*}, which admits an optimal solution for the first time, already belongs to the validation phase. We define the tree explored at the end of the exploration phase by $T_E = T[\mathcal{N}_E]$.

(2) The sizes of these sets of nodes obey $0 \leq |\mathcal{N}_E| = t^* - 1$ and $1 \leq |\mathcal{N}_V|$. We call these quantities the length or ***duration of the respective phase***.

(3) The following cases can be encountered.

If the lp-bound of KP is $c^{lp} = c^*$, the root is an optimal node by Lemma 2.5 and we face an ***lp-optimal*** instance. Therefore $t^* = 1$ and both phases are trivial $|\mathcal{N}_E| = 0, |\mathcal{N}_V| = 1$. In particular, the branch-and-bound tree is only a singular node.

If $c^{lp} > c^*$ is not lp-optimal, we distinguish cases depending on the quality of the initial solution being the greedy in BB. If the ***greedy solution is optimal*** $t^* = 1$ and the root is an optimal node. It is a fractional and branched node by $c^{lp} > c^*$. The optimal profit is given by the greedy profit $c^{greedy} = c^*$ at the optimal node. The exploration phase is trivial $|\mathcal{N}_E| = 0$ and the validation phase is non-trivial $|\mathcal{N}_V| > 1$.

If the ***greedy solution is not optimal*** in particular KP is not lp-optimal by Lemma 2.5 and we encounter the optimal node at an integral node not being the root at a time $t^* > 1$. In this case, the optimal profit is given as the lp-bound $c^{lp}_{t^*} = c^*$ at the optimal node. The exploration phase is non-trivial $|\mathcal{N}_E| > 0$ while the validation phase only might be non-trivial $|\mathcal{N}_V| \geq 1$. Note that the optimal node might be the last active node providing $|\mathcal{N}_V| = 1$.

(4) By the phases being disjoint, the number of nodes decomposes into summands $N := |\mathcal{N}_E| + |\mathcal{N}_V| = |T|$. The number of nodes of T is only one way to ***measure the performance of a branch-and-bound algorithm***. Since a branch-and-bound tree T is a binary tree, we can equivalently look at the number of branched nodes $\frac{|T|-1}{2}$ being the inner nodes of T. For a more granular running time analysis for BB for arbitrary node-selection rules $\mathcal{S}$ we at least need to incorporate the complexity of $\mathcal{S}$ and refer to remark 2.34. With regard to actual running time BESTFS may be outperformed by a rule which does not blow up the number of active nodes to an order $O(|T_U|)$ and while introducing a sufficiently small set of additional nodes to T_U. Using stronger upper bounds may entirely change the game since an unavoidable structure $T_U = \overline{T(c^*)}$ is dependent on $lp(c^*)$.

Theorem 2.36. ***Duration of BB***
Given an instance of KP and let $\mathcal{S}$ be an arbitrary node selection rule used in BB yielding the branch-and-bound tree T at termination. Let t^* be the time the optimal node was detected and consider $T_E = T_{t^*}$ the current tree at the end of the exploration phase.

(1) Then $T \subseteq \overline{T(c^*)} \cup \overline{T_E}$.

(2) The size of T is bounded by the number of unavoidable nodes and the duration of the exploration phase by $|T| \leq 2t^* + 2|\mathcal{N}(c^*)| - 1$. In particular we can estimate the duration of the validation phase $|\mathcal{N}_V| \leq t^* + 2|\mathcal{N}(c^*)|$.

Proof.

(1) We partition the nodes of T in $\mathcal{N}_E \cup \mathcal{N}_V$. For nodes $s' \in \mathcal{N}_E \subseteq \overline{T_E}$ the inclusion is clear. So suppose $s' \in \mathcal{N}_V$ meaning at the time s' is processed we have available the optimal profit as incumbent $c^{inc} = c^*$. If $c^{lp}_{s'} > c^*$ we have $s' \in \mathcal{N}(c^*) \subseteq T_U = \overline{T(c^*)}$. So it remains to show the statements for nodes $s' \in \mathcal{N}_V$ with $lp_{s'} \leq c^*$. If s' is the root it is contained in T_U. Otherwise assume an ancestor s of s'. If $lp_s > c^*$ we have $s \in \mathcal{N}(c^*)$ feasible and fractional and obtain $s' \in T_U$ by closure. If $lp_s \leq c^*$ we distinguish two more cases. If $s \in \mathcal{N}_E$ we get $s' \in \overline{T_E}$ by closure. Otherwise we encounter $s \in \mathcal{N}_V$ with $lp_s \leq c^*$. But this implies we prune s by bound and $s' \notin T$, a contradiction.

(2) We can compute a rough estimate on the total duration by using Lemma 2.33 and estimate $|T| \leq |T_U \cup \overline{T_E}| = |T_U| + |\overline{T_E}| - |T_U \cap \overline{T_E}| \leq |\overline{T_E}| + |\overline{T_E}| - 1 \leq 2|\mathcal{N}(c^*)| + 1 + 2(t^* - 1) + 1 - 1 \leq 2t^* + 2|\mathcal{N}(c^*)| - 1$ where $|\mathcal{N}(c^*)|$ is the number of unavoidable nodes independent of the node-selection $\mathcal{S}$. In particular we can write $|T| = |\mathcal{N}_E| + |\mathcal{N}_V| \leq 2t^* + 2|\mathcal{N}(c^*)| - 1$ and thus obtain $|\mathcal{N}_V| \leq t^* + 2|\mathcal{N}(c^*)|$ by $|\mathcal{N}_E| = t^* - 1$. Note that $|T_U| = 2|\mathcal{N}(c^*)| + 1$ is tight and the estimate on $|\overline{T_E}|$ depends on the structure of T_E where large degrees in T_E reduce the error.

□

Remark 2.37.
This results shows that processing nodes which are in $T(c^*)$ in any phase does not obstruct the potential optimality of a node-selection rule. This is somewhat relaxing as in particular one has some room for mistakes in the exploration phase and in general only moving along PTO is not required. In particular BESTFS demonstrated that this is not necessary. In particular, if $T(c^*)$ does not only consist of one PTO optimal node selection rules are not unique. In contrast, if a node-selection rule explores nodes outside of $T(c^*)$ in the exploration phase, these mistakes are punished by having to add up to two nodes. So in this regard BESTFS admits the most generous exploration phase among all optimal node-selection rules. Note that the somewhat degenerate (nonetheless wishful) case of optimal roots does not obstruct this claim.

Corollary 2.38. ***Phases when using BESTFS in BB***
Consider an instance of KP solved by using BESTFS in BB. Let T be the branch-and-bound tree and suppose that the optimal node is detected at time t^*. If the initial solution is not optimal we have $T(c^*) = N_E$ with $|\mathcal{N}(c^*)| = t^* - 1$ and $|N_V| = |N_E| + 1 = t^*$. In total we obtain $|T| = 2t^* - 1 = 2|N_E| + 1$. If the initial solution is optimal $N_E = \varnothing$ and $N_V = 2|\mathcal{N}(c^*)| + 1$.

Proof.
We apply Theorem 2.31 in order to obtain $T = T_U = \mathcal{N}_E \dot{\cup} \mathcal{N}_V$ and estimate the sizes by using Lemma 2.33. By Lemma 2.29 we visit an integral node which admits the optimal profit at time $|T(c^*)| + 1$. If the initial solution is not optimal, the exploration phase lasts $|T(c^*)|$ steps. If the initial solution is already optimal still the unavoidable tree T_U needs to be fully explored. □

Chapter 3

Introduction to machine learning

The field of machine learning (ML) became popular in recent decades for extracting and modeling patterns in data. We take a brief history excursion referring to James, Witten, Hestie and Tibshirani [20, p.6].

Brief historical overview

The development of machine learning techniques can be traced back to the 1930s when Fisher proposed the ***linear discriminant analysis***. Not much later, further linear methods such as logistic regression were introduced. By the end of the 70s, the class of ***generalized linear models***, containing the logistic regression model and similar, had been defined. Until the 80s, one was primarily concerned with linear models as the non-linear models were computationally infeasible. When the computation power became widely available during the mid-80s, more modern methods such as ***classification*** (Breiman, Friedman, Olshen and Stone) were introduced. In 1986, the class of generalized linear models was extended further by ***generalized additive models*** (Hastie and Tibshirani). Ever since, these types of methods are continually developed and used in all kinds of applications.

Prominent applications of machine learning

For instance, Amazon uses ML techniques for performing ***speech recognition*** in their smart home appliances. Google utilizes the modeling power to perform large scale data-analysis. Even in more manageable tasks such as ***filtering of spam-mails*** or ***face recognition*** used in recent generations of smartphones the machine learning paradigm is present. Notable machine learning models can handle any input of the right format and have the ability to automatically abstract certain soft rules describing the data. Referring to the example of Amazon's Alexa the ability to handle differently pitched voices perhaps spoken at various speeds is remarkable. Another remarkable example is the capability of modern face-recognition techniques to recognize a user's face under different lighting conditions or after having altered the face such as by wearing sunglasses.

Introductory example to a machine learning task

The example in Bishop's book [6, p.2] shall serve us as an introductory example to trace down the machine-learning workflow. We will take a closer look at notable aspects of this exemplary process in the subsequent sections.

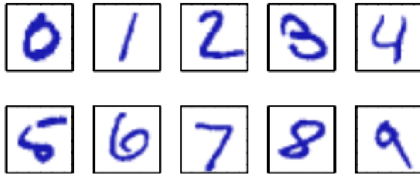


Figure 3.1: Handwritten digits; Bishop [6, p.2]

Suppose we have a huge collection of 28×28 images which all represent one hand-written digit as it is shown in Figure 3.1. Our task is to ***extract the digit which is displayed in the image***. For the example's sake, suppose we aim to automate a scanning of sheer amounts of handwritten zip codes which shall be stored in a database. Our first attempt may be to define a fixed set of rules which describe a digit. In practice however it is virtually impossible to represent each version of a written digit by a fixed set of exceptions and rules. It may well be that a very similar looking version of a digit can not be associated with a number because the set of fixed rules misses a layer of abstraction. Thus we should at least aim for a model which is able to suggest a feasible label for any 28×28 input image. In a sense we wish to ***determine a set of continuous soft rules***.

Speaking a bit more abstractly we want to define a parametric function $f(x, \theta)$ which assigns to each 28×28 image x the represented digit. We ***make some rough assumptions on the form of*** $f(.,.)$ and ***tune the model by finding suitable parameters*** θ by applying suitable mathematical techniques. In order for this to be feasible we need to fix some basic assumptions on the architecture of the model. In order to evaluate the quality of the model for being able to optimize the parameters and perhaps assess the quality of the model, suitable measures are required. This is done by introducing a so-called loss-function $L(f(x, \theta), y)$ which in our example measures the distance of a suggested digit $f(x, \theta)$ and an original known digit y. Equipped with this framework we need a set of n images x_i where we already have correctly abstracted the depicted digit y_i. We now minimize the average loss $\frac{1}{n} \sum L(f(x_i, \theta), y_i)$ with respect to θ by using a suitable optimization technique. Having tuned the model accordingly, we can ***evaluate the model*** $f(., \theta)$ ***on any number of further examples*** and achieved an automation of zip-code scanning. If the model managed to reasonably abstract the images we expect a correct assignment for many but probably not all input instances.

Along this way there are some modeling choices to be made and we have to keep certain pitfalls in mind. Throughout the following sections we aim to cover the basics of machine learning in order to be able to successfully set up a machine learning task. We primarily are targeted towards neural networks and only briefly peek at other notable machine learning techniques.

Machine learning paradigms

The above example follows a specific machine learning paradigm called ***supervised learning***. In order to convey the broadness of machine learning we refer to Bishop and briefly introduce other prominent machine learning paradigms [6, p.3].
The essence of supervised learning methods is having a training set with already known predictions. We select a model-type and fit on the training set usually by using non-linear optimization techniques. The two main categories of supervised learning techniques are classification and regression. In classification, we generate predictions within a finite predefined set of values called class labels. In a regression task we allow the predictions to be real vectors of one or several dimensions.

A further branch of machine learning is the ***unsupervised learning***. The data for unsupervised learning is not equipped with a predicted label beforehand but the model shall figure out meaningful labels on its own. One of the typical examples is clustering. It aims to assign to each data-point one or more labels which represents the membership in a cluster. The model a priory only has a set of vague assumptions such as a notion of closeness within a cluster and a notion of distance between clusters and targets the task only regarding the global demands perhaps not even specifying a number of clusters to be expected. In this way, the learning happens with a large degree of autonomy hence the name.

Arguably, the most degree of freedom is experienced in ***reinforcement learning*** paradigm (Sutton and Barto, 1998). We do not specify a global notion of what the success of a prediction is but we introduce a local notion of reward which accumulates by each action the model takes. The model takes an action by traversing states in a defined environment. It tries to identify which actions are preferable at a given state in order to achieve a maximal sum of rewards when reaching a final state. One fun application of this paradigm is learning to play computer games typically achieving superhuman skills. Another more serious example is the chatbot Tay (Microsoft, 2016) being quickly discontinued since apparently the most outrageous aspects of twitter are associated with large rewards.

3.1 Notations for data

The driving force of machine learning is data. Thus an integral part of an introduction to machine learning is introducing suitable notations and a discussion of formats of data. Many machine learning techniques live up to their full potential only if the data is suitably preprocessed. Since the aspect of data preprocessing is quite contained in our application we guide towards Zheng and Casari who carefully treat and cover this topic in their book [34]. Details regarding the actual implementation are discussed in chapter 6. This section is dedicated to the basic notions and notations necessary for the mathematical aspects following the conventions in Bishop [6, xi ff.].

Definition 3.1. ***Data points and data matrix***

(1) Given $\{x_1, \ldots x_n\} \subseteq \mathbb{R}^m$ a set of $n \in \mathbb{N}$ ***data points*** of dimension $m \in \mathbb{N}$. Define $X := (x_1^T, \ldots, x_n^T)^T \in \mathbb{R}^{n \times m}$ called the ***data matrix***. The element $x_{ij} \in \mathbb{R}$ in this way is the j-th element of the i-th data-point x_i being stored in form of a row in X. Columns correspond to so called ***features***, see Definition 3.2 below.

(2) Let $y \in \mathbb{R}^n$ denote another vector corresponding to data points x_i perhaps given in form of a data matrix $X \in \mathbb{R}^{n \times m}$. The component y_i of y is called the ***label***, response or target of the data-point x_i. A tuple (x_i, y_i) is called a ***labeled data point*** and a set $\{(x_i, y_i) \mid 1 \leq i \leq n\}$ containing only labeled data points is called ***labeled data***. Depending on the context the labels may be restricted to be discrete.

Definition 3.2. ***Features and feature space***

The term ***feature*** in a machine learning context is widely used and refers to a (statistical) attribute of the (suitably transformed) input data. For instance, when encountering data being associated to persons, we may standardize the height of a person and call the standardized height a feature.

The procedure of selecting features, the so called ***feature engineering***, is part of preprocessing the data at hand. Offering an outlook, in a more general setting one might have to apply encoding and decoding techniques to even mathematically capture the problem at hand (e.g. text processing). While in feature engineering there are certain guidelines, techniques, and best practices, one still heavily relies on the domain knowledge of the practitioner. Selecting a set of suitable dimensions of the m-dimensional data cube is a predominating factor at this stage and is known as ***dicing the data***.

Referring to Bishop, we merely provide the formalities to mathematically capture the feature engineering stage since it is not really an issue in our setup [6, p.137 ff.]. The interested reader is referred to Zheng et al. who strive for a comprehensive introduction to feature engineering [34].

(1) Let $\{\phi_j : \mathbb{R}^m \to \mathbb{R}, \mid j \in [M-1]\}\}$ be a finite set of functions and conventionally set $\phi_0 \equiv 1$ to the always-one function. The ϕ_j, $0 \leq j \leq M-1$, are called the ***basis functions***.

(2) The vector-valued function $\phi(x) = (1, \phi_1(x) \ldots, \phi_{M-1}(x)) : \mathbb{R}^m \to \mathbb{R}^M$ is called the ***feature space mapping***. It shall be understood as a transformation of the input-data. In this context the input-data is called the raw data. Images $\phi(x_i) \in \mathbb{R}^M$ of the raw data-points x_i are called ***feature vectors***.

(3) We define the ***design matrix*** as the data matrix of the transformed data
$\Phi = (\phi(x_1), \ldots, \phi(x_n))^T \in \mathbb{R}^{n \times M}$.

Remark 3.3. ***Data preprocessing***
Needless to say an actual machine learning task starts at least one step earlier. Before feature engineering can be applied it is necessary to achieve a certain quality of data by cleaning. This oftentimes has to do with getting rid of faulty rows of a data matrix. Since this type of action does not reduce the dimension m of the data cube this action is also called ***slicing the data***. At this stage, one heavily relies on visualization techniques and statistical methods for outlier detection for example. This is not a big concern in our sterile working environment but in practice makes up a large proportion of an ML task. It typically is stated that 80 to 90% of effort in a data science task is spent on preprocessing the data showing that the actual installment of a suitable model is only a small piece in the puzzle.

Definition 3.4. ***Padding***
By padding we generally understand the concatenation of artificial digits to a data-point x. This measure can be interpreted as a feature-space mapping
$\phi(x) = (p_1, \dots, p_k, x, p'_1, \dots p'_{k'})$. We say we pad x with $(p_1, \dots, p_k)$ at the beginning and with $(p'_1, \dots p'_{k'})$ at the end. Oftentimes, zeros and ones are the digits of choice. Padding is mainly a convenience measure and allows for compact notation. Some models also are able to learn that zeros and ones do not carry any information. In this case one can train a model with respect to a large worst case input dimension and evaluate points of smaller dimension via padding as well.

3.2 Modeling in a machine learning context

In order to embed the machine-learning task in a more formal mathematical framework let us address the underlying assumptions and goals of machine learning. Being equipped with the formal environment we dive into the fitting or training of models which is also called the learning (phase). As already hinted in the introductory example in Section 3 the parameters are typically optimized by applying non-linear gradient descent techniques. We base this section on the excellent book on deep learning by Goodfellow, Bengio and Courville [13, p.275 ff.].

Goals and assumptions in machine learning

We assume an underlying distribution p_{data} of labeled data points. In general we do not know p_{data} but instead we have available a sample of n points $\{(x_1, y_1), \dots, (x_n, y_n)\}$. This sample is called the training set where the associated empirical distribution $\widehat{p}_{data}$ ***approximates the original distribution***. The training set is used to derive a model for p_{data} and in this way there is a natural gap by only working with a sample and its empirical distribution.

A machine-learning model is a parametric function $f(x;\theta) = \widehat{y} \sim y$ approximating the true label y via a prediction $\widehat{y}$. f can be evaluated for any x which merely assumes the right format. For measuring the closeness of labels $y, \widehat{y}$ we assume a very general performance-measure P. As most times the true performance measure P is intractable

with regard to mathematical optimization we work with an ***indirect measure***. We pick up on the loss function L already being hinted in Section 3. L is required to admit reasonable optimization properties and shall be tightly correlated to P. The loss function sometimes even has better modeling properties as opposed to P and not only preferred with regard to mathematical properties. A loss-function typically is defined via averaging over the point loss terms $L(f(x;\theta),y) \coloneqq \frac{1}{n}\sum_{i=1}^{n} L(f(x_i,\theta),y_i)$ and in the following we focus only on loss functions having this particular form. We define the (true) risk associated to the model as expected loss $\mathbb{E}_{(x,y)\sim p_{data}}(L(f(x;\theta),y)) = J^*(\theta)$. The risk tells how well the model with the respective choice of the parameters θ fits the true distribution p_{data}. Naturally wish to minimize the risk with respect to θ yielding a precise model. However p_{data} is typically unknown and we ***estimate the true risk by the empirical risk*** $\mathbb{E}_{(x,y)\sim \widehat{p}_{data}}(L(f(x;\theta),y)) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i,\theta),y_i) = J(\theta)$ involving the empirical distribution. The ***minimization of the empirical risk*** is the appropriate formal concept being executed by fitting a model and it substantiates the learning of the data.

We often ***evaluate the quality*** of the fitted model by means of an independent test set. The test set is also interpreted as a sample from p_{data} but it was not used for training. The gap we observe when evaluating on the training set and the test set gives insight into the generalization capabilities of the model.

3.2.1 Stochastic gradient descent methods

As mentioned above, gradient descent methods are predominately used for fitting (non-linear) models. By the form of loss functions averaging the per point losses already the mere evaluation of loss functions and their derivatives can be a costly endeavor. Reasonable sizes for training data ranges from thousands to millions of samples. In order to ***speed up the gradient update***, one samples only a small batch of data points. The gradient can be approximated from this random batch by a straightforward scaling and thus in and off itself is a realization of a random variable. In the following, we introduce this stochastic gradient approach and highlight the commonly used versions all being based on this idea.

Definition 3.5. ***Stochastic gradient descent (SGD)***
We aim to approximate the gradient $\nabla_\theta L$ by drawing uniform random samples $\{x'_1,\dots x'_{n'}\}$ of fixed size $1 \leq n' \leq n$. These samples are called mini-batches and n' is called the batch size. We refer to Goodfellow et al. [13, p.294] where a standard variant of stochastic gradient-descent is presented.

Algorithm 4: Stochastic gradient descent (SGD)

Data: $(\epsilon_k) \in \mathbb{R}^{\mathbb{N}}$ sequence of learning rates; θ initial parameter
Result: θ model parameter
1 Initialize time $k = 1$;
2 **while** *not converged* **do**
3 sample n' data points $\{(x'_1, y'_1), \dots (x'_{n'}, y'_{n'})\} \subseteq \{(x_1, y_1), \dots, (x_n, y_n)\}$;
4 Approximate the gradient $\nabla_\theta J \sim g := \frac{1}{n'} \sum_{i=1}^{n'} \nabla_\theta L(f(x'_i, \theta), y'_i)$;
5 Update parameter $\theta := \theta - \epsilon_k g$;
6 $k{+}{+}$
7 return θ

Remark 3.6.

(1) Oftentimes we use the word ***batch*** to describe a mini batch and the term batch size is used for describing the size of a mini batch. This shall not be confused with a batch referring to the entire data set in the paradigm of deterministic descent methods.

(2) Using only a single data point at the time for approximating the gradient defines the concept of online descent methods. Formerly the online methods were also called stochastic methods but the mini patch methods described in SGD are now also called stochastic methods. In a practical application of SGD and similar methods one performs the random sampling by shuffling the training data set. The sampling is then simulated by picking the subsets $\{i+1, \dots, i+n'-1\}$, $i \equiv 0 \mod n'$ where i is gradually increased. Cycling through the data set once in this fashion is called learning for one ***epoch***, see Goodfellow et al. [13, p.280]. In each new epoch again the data set is shuffled randomly. Implementations of SGD variants thus typically ask for a specification of the batch size and the number of epochs used for training. These types of inputs which concern the learning phase are called the ***hyperparameters***.

(3) The choice of an appropriate batch-size is often a ***trade off*** between number of iterations and cost of evaluations. A notable observation is motivated by the quick convergence of standard error $\frac{\sigma}{\sqrt{n'}}$ of the random mini batch in comparison to a singular error σ. The gain in accuracy when drawing batches of size 100 or respective batches of size 10000 is only a factor of 10 while the evaluation of the larger mini batch would take 100 times longer. The inaccurate gradient however may add unto the number of iterations.

(4) The choice of the ***learning rate*** is often done by tracking the loss over time and adapting the learning rate accordingly. A common step size sequence is formed by linearly decreasing the step size until some fixed time τ is reached and keeping it constant afterward. There is a theoretical convergence result which guarantees convergence if $\sum_{k>0} \epsilon_k = \infty$ and $\sum_{k>0} \epsilon_k^2 < \infty$ being satisfied by the proposed sequence and by constant learning rates.

Definition 3.7. ***Stochastic gradient descent with momentum (SGDM)***
We track the motion of the model parameter θ in space and associate a unit mass and velocity v to the point θ. Motivated by physics we introduce a notion of momentum being speed times mass. During the optimization we store an exponentially decaying history reflecting the momentum which partly steers the motion of the optimization parameter. With this technique we hope to counter the noise in the gradients coming from sampling and poor conditioning.

Algorithm 5: Stochastic gradient descent with momentum (SGDM)

Data: ϵ learning rate; θ initial parameter, α momentum parameter, v initial velocity
Result: θ model parameter
1 **while** *not converged* **do**
2 sample n' data points $\{(x'_1, y'_1), \dots (x'_{n'}, y'_{n'})\} \subseteq \{(x_1, y_1), \dots, (x_n, y_n)\}$;
3 Approximate the gradient $\nabla_J(\theta) \sim g := \frac{1}{n'} \sum_{i=1}^{n'} \nabla_\theta L(f(x'_i, \theta), y'_i)$;
4 Update velocity $v := \alpha v - \epsilon g$;
5 Update parameter $\theta := \theta + v$
6 return θ

Definition 3.8. ***Root mean square propagation (RMSprop)***
This optimization technique incorporates adaptive learning. Component wise, we incorporate an exponentially decaying history by looking at the (component wise) squared gradient. Using this update rule we can quickly traverse flat regions of the objective function and automatically slow down in steep regions for each individual model parameter θ_i.

Algorithm 6: Root mean square propagation (RMSprop)

Data: ϵ learning rate; θ initial parameter, ρ decay rate, δ small constant for numeric stability
Result: θ model parameter
1 Initialize $r = (0, \dots, 0)$ **while** *not converged* **do**
2 sample n' data points $\{(x'_1, y'_1), \dots (x'_{n'}, y'_{n'})\} \subseteq \{(x_1, y_1), \dots, (x_n, y_n)\}$;
3 Approximate the gradient $\nabla_J(\theta) \sim g := \frac{1}{n'} \sum_{i=1}^{n'} \nabla_\theta L(f(x'_i, \theta), y'_i)$;
4 Accumulate squared gradient component wise $r_i := \rho r_i + (1 - \rho) g_i^2$;
5 Compute parameter update component wise $\Delta\theta_i = -\frac{\epsilon}{\sqrt{\delta + r_i}} g_i$;
6 Update parameter $\theta := \theta + \Delta\theta$
7 return θ

Remark 3.9.
The RMSprop is seen as an adaptation of AdaGrad. AdaGrad incorporates a summed history and is particularly well suited for optimizing convex functions. In RMSprop we forget the history quicker by applying an exponential decay. Thus having moved to a convex location we have a larger influence of the current gradient and try to achieve a behavior similar to AdaGrad in this convex region.

Definition 3.10. ***Adaptive moments (Adam)***
We use an exponential decay rate and incorporate the first and second moments of the gradient estimates in a momentum. We introduce a bias correction which shall compensate for the bias introduced by the initialization. We refer to Kingma and Ba for more theoretical motivation and a careful analysis [23]. To be clear by ρ_1^t, ρ_2^t we simply denote taking the t-th power of constants ρ_1 respectively ρ_2 .

Algorithm 7: Adaptive moments (Adam)

Data: ϵ learning rate; θ initial parameter, ρ_1, ρ_2 decay rate in $[0, 1)$ for moment estimates, δ small constant for numeric stability
Result: θ model parameter

1 Initialize $s = r = (0, \dots, 0)$, time $t = 0$;
2 **while** *not converged* **do**
3 sample n' data points $\{(x'_1, y'_1), \dots (x'_{n'}, y'_{n'})\} \subseteq \{(x_1, y_1), \dots, (x_n, y_n)\}$;
4 Approximate the gradient $\nabla_J(\theta) \sim g := \frac{1}{n'} \sum_{i=1}^{n'} \nabla_\theta L(f(x'_i, \theta), y'_i)$;
5 $t{+}{+}$;
6 Update biased first moment estimate $s := \rho_1 s + (1 - \rho_1) g$;
7 Update biased second moment estimate element-wise $r_i := \rho_2 r_i + (1 - \rho_1) g_i^2$;
8 Correct bias in first moment $\widehat{s} = \frac{1}{1-\rho_1^t} s$;
9 Correct bias in second moment $\widehat{r} = \frac{1}{1-\rho_2^t} r$;
10 Compute parameter update component wise $\Delta\theta_i = -\epsilon \frac{\widehat{s}_i}{\sqrt{\widehat{r}} + \delta}$;
11 Update parameter $\theta := \theta + \Delta\theta$
12 return θ

Remark 3.11.
The default parameters for Adam are $\rho_1 = 0.9$, $\rho_2 = 0.999$, $\delta = 10^{-8}$, $\epsilon = 0.001$. Whenever we use Adam, we only may adapt the learning rate ϵ and leave the other parameters on the default settings.

3.3 Regression

Regression tries to model the relation between a set of input variables and a ***continuous response variable***. We introduce the regression formally as a statistical model and point out that the systematic component is used as what is called a regression model in machine learning. Oftentimes the modeling power of straight forward regression is already sufficient to model relationships in data. Moreover an introduction to regression will provide a better understanding of more modern approaches which are undeniably related to basic variants of regression models. We base this section on the book by Dunn and Smyth [9].

Definition 3.12. ***Exponential dispersion model family (EDMs)***
Let $EDM(x;\theta,\phi) = a(x,\theta)\exp(\frac{x\theta-\kappa(\theta)}{\phi})$ be a probability density function. The parameter θ is called the ***canonical parameter***, $\kappa(.)$ the ***cumulant function***, $\phi > 0$ the ***dispersion parameter*** and $a(.)$ is a ***normalization function*** ensuring that EDM is a density. The exponential dispersion model family EDMs is defined as the family of all distributions of the above form. An extension to the multivariate case is possible. If a random variable y has a density of the form $EDM(x;\theta,\phi)$, we use shorthand notation $y \sim EDM(\theta,\phi)$.

Lemma 3.13.
The Gaussian , Poisson, binomial, and the exponential distributions are members of EDMs.

Proof.
We simply mention how the parameters of EMD need to be set. The full details are available in Dunn et al. [9, p.213].

distribution	density	θ	ϕ	$\kappa(\theta)$	$a(x,\phi)$
Gaussian $p(x;\mu,\sigma^2)$	$\frac{1}{\sqrt{2\pi\sigma^2}}\exp(\frac{(x-\mu)^2}{2\sigma^2})$	μ	σ^2	$\frac{\theta^2}{2}$	$\frac{1}{\sqrt{2\pi\sigma^2}}\exp(\frac{-x^2}{2\sigma^2})$
Poisson $p(k;\mu)$	$\frac{\exp(-\mu)\mu^k}{k!}$	$\log(\mu)$	1	μ	$\frac{1}{k!}$
binomial $p(x;\mu,m)$	$\binom{m}{mx}\mu^x(1-\mu)^{m(1-x)}$	$\log(\frac{\mu}{1-\mu})$	$\frac{1}{m}$	$-\log(1-\mu)$	$\binom{m}{mx}$
exponential $p(x;\gamma)$	$\frac{\exp(-\frac{x}{\gamma})}{\gamma}$	$-\frac{1}{\gamma}$	1	$\log(\gamma)$	1

Table 3.1: Gaussian, Poisson, binomial, and exponential are in EDM

□

Lemma 3.14. ***Expectation and variance of EDM***
Given $y \sim EDM(\theta,\phi)$. We refer to Dunn at al. who present the expectation and variance of random variables distributed according to EDM [9, p.216].

$$\mathbb{E}(y) = \frac{d\kappa}{d\theta}(\theta) \qquad Var(y) = \phi\frac{d^2\kappa}{d\theta^2}(\theta) = \phi\frac{d\mu}{d\theta}(\theta)$$

Definition 3.15. ***Regression models***
Let $x_1, \ldots, x_m, y$ be random variables and given n independent observations $x_{i1}, \ldots, x_{im}, y_i,\ 1 \le i \le n$. Furthermore let $\beta_i, w_i, o_i\ 0 \le i \le m$ be real constants.

(1) Let us introduce ***regression models*** as in Dunn et al. [9, p.12]. A regression model assumes $\mathbf{E}(y_i) = \mu_i = f(x_{i1}, \ldots, x_{im}; \beta_0, \beta_1, \ldots, \beta_m)$ meaning the mean response is a parametric function f of the input variables. f is called the systematic part of the model and has available the so called regression parameters $\beta_0, \ldots, \beta_m$, see . The random part of the model is compromised by assumptions on the distribution of y_i.

(2) Moving on, let us formally define ***linear regression models*** again referring to Dunn at al. [9, p.12]. Regression models of the form $\mu_i = f(\beta_0 + \sum_{i=j}^{m} \beta_j x_{ij})$ are called linear in the parameters and $\beta_0 + \sum_{i=j}^{m} \beta_j x_{ij}$ is called the linear predictor. β_0 is called intercept (or bias in a machine learning context), and the $\beta_i, 1 \le i \le n$ are called slopes for the corresponding explanatory variables.

According to Dunn et al. the ***random part of linear regression models*** is defined as follows [9, p.32].

$$Var(y_i) = \frac{\sigma^2}{w_i} \text{ for all } 1 \le i \le n \tag{LM}$$
$$\mathbb{E}(y_i) = \mu_i = \beta_0 + \sum_{i=j}^{m} \beta_j x_{ij} \text{ for all } 1 \le i \le n$$

σ^2 denotes the common part of the variance of the y_i and the w_i are weights.

(3) Referring to Dunn et al. ***generalized linear models*** are defined by two conditions as follows [9, p.211 ff.].

$$y_i \sim EDM(\mu_i, \frac{\phi}{w_i}),\ 1 \le i \le n \text{ for all } 1 \le i \le n \tag{GLM}$$
$$g(\mu_i) = \eta_i = o_i + \beta_0 + \sum_{i=j}^{m} \beta_j x_{ij} \text{ for all } 1 \le i \le n$$

g is a monotonic differentiable function called the link function. In a machine learning context the inverse g^{-1} is called the activation function. The o_i are called offsets and the w_i are called weights, $1 \le i \le n$.

Lemma 3.16. ***Maximum likelihood parameters***
Given a data-matrix X and labels $y_i \sim N(\mu_i, \sigma)$ being normally distributed and let $\mu_i = \beta_0 + \sum_{j=0}^{m} \beta_j x_j$ satisfy LM. Furthermore consider the mean squared error as defined later in Section 3.7.1. Then according to Dunn et al. a parameter vector β which admits the minimal mean squared error on the given data set is a maximum likelihood parameter for this normal linear regression model [9, p.174].

3.4 Linear classification

The core idea of a classification algorithm is well described by Bishop who we refer to throughout this section [6, p.43]. The aim of classification is to model a structure of some given data explaining an ***assignment of data points to discrete labels*** which are comprising the finite and predefined label set $\mathcal{T}$. Such a label assignment is called a discriminant function and the sets of points having the same label are called classes $\mathcal{C}_k$. Since we encounter discrete class labels the concept of replacing a relentless true performance measure P by an appropriate loss function L is crucial for successful modeling of classification. In the following, we thus explain how this is done and interpreted in a meaningful way.

Classification is an instance where optimizing L as opposed to P has many perks which we will briefly get into. The application of P though is valuable when assessing the fit of the model, see Section 3.7.2. In the context of classification, we may not merely aim to model the actual label assignment but moreover we can incorporate more fundamental aspects. Let us discuss common variants and let us point out the advantages and disadvantages.

Generative models

One powerful approach is ***modeling the joint distribution*** of data points given by $p(x, \mathcal{C})$. This approach yields us so-called generative models. They allow us to generate synthetic input data by sampling from the modeled distribution, and we have access to all aspects of the distribution. Usually these models demand large amounts of data. One aspect demonstrating the power of generative models is being able to compute the marginal distribution $p(x)$. By this we can identify unlikely data points which can be used to perform outlier detection.

Discriminative models

In a typical setup however we can give up some of the power of generative models in trade for performance. Typically it is sufficient to ***only model the posterior class probabilities*** $p(\mathcal{C}_k|x)$ telling us how likely the corresponding class label for an encountered data point x is. Approaches which directly model the posterior class probabilities are called discriminative models, and they form a well established family of models for classification. The modeling task typically is subdivided into two phases.

In the ***inference phase*** we aim to construct a probabilistic model for $p(\mathcal{C}_k|x)$ based on a training set. In the ***decision phase***, we then take care of the class-label assignment of data-points based on this probabilistic model.

Having available said probabilistic model in the background as opposed to perhaps a direct class-label assignment we encounter a variety of advantages. Besides being able to ***quantify the certainty of a classification***, we can balance data-sets where one class label is very rare and utilize the thus permitted ***bias correction***. A balanced data

set is desirable because intuitively a learning algorithm needs to first of all see sufficiently many items of each class in order to learn the relation to the respective label. Second of all a small class may have less impact on the loss function and therefore it is learned not as thoroughly while large classes may tend to be learned disproportionately well. However by perhaps artificially balancing the set a bias toward these rare instances is introduced as the model regards them as overly common. With help of $p(\mathcal{C}_k|x)$ and Bayes Theorem (Theorem 3.22) one is able to compensate for this effect. Another advantage lies in the ability to ***combine several classification models*** in a meaningful way. For instance we may want to determine whether a patient has a disease based on a positive or negative blood test and based on a positive or negative X-ray result. By setting up tests which estimate a probability of a negative result as opposed to only telling a positive or negative result, we can carry over the certainty of the individual predictions to a combined model. This combined model incorporated the individual results in a naturally weighted way by respecting the significance.

Basics of classification

In this section let us briefly expand the notions for data in order to appropriately formalize classification. The notations again are due Bishop [6, p.179 ff.]. For sake of completeness let us also present the well known theorem of Bayes which is omnipresent in this particular subfield of supervised learning.

Definition 3.17. ***Data in a classification context***
In a classification context only discrete targets are considered. Given a set of k ***class labels*** $\mathcal{T} = \{0, \dots, k-1\}$ we denote the class label associated to each individual data point x_i by t_i. The vector $t = (t_1, \dots, t_n)$ is called the target vector. For a set of labeled data $\{(x_i, t_i) \mid 1 \le i \le n\}$ equipped with discrete class labels we define ***classes*** $\mathcal{C}_{j+1} = \{x_i \mid t_i = j,\ 1 \le i \le n\}$. It is common practice to shift the labels by one. The classes naturally define a partition of the data-set $\{x_1, \dots, x_n\} = \dot{\bigcup}_{j=1}^{|\mathcal{T}|} \mathcal{C}_j$ and $|\mathcal{T}|$ is called the number of classes.

Definition 3.18. ***One-hot encoding***
Consider $t \in \mathcal{T}$ a fixed class label. We define the one-hot encoding of t as $\tilde{t} = e_t \in \mathbb{R}^{|\mathcal{T}|}$ the t-th unit basis vector. The one-hot encoding is also called a 1-of-K coding scheme.

Definition 3.19. ***Discriminant function***
Given class labels $\mathcal{T} = \{0, \dots, k-1\}$. A function $f : \mathbb{R}^m \to \mathcal{T}$ which assigns each datapoint $x \in \mathbb{R}^m$ to a class is called a discriminant function.

Definition 3.20. ***Decision regions and boundaries***
Suppose we are given a discrimination function $f : \mathbb{R}^m \to \{0, \dots, k-1\}$. The subsets $\mathcal{R}_{i+1} = f^{-1}(i) \subseteq \mathbb{R}^n,\ i = 0, \dots, k-1$ are called the decision regions. The boundaries $\partial\mathcal{R}_i$ of these sets are called decision boundaries.

Definition 3.21. ***Linear discriminant function***
A discriminant function $f : \mathbb{R}^m \to \mathcal{T}$ is called linear if the decision boundaries are $m-1$ dimensional hyperplanes in $\mathbb{R}^m$ as defined in Bishop's book [6, p.179].

Theorem 3.22. ***Bayes (1763)***
For any joint probability distribution $p(A,B)$ we have $p(A|B) = \frac{p(B|A)p(A)}{p(B)}$ called Bayes rule or Bayes Theorem for conditional probabilities [4].

3.4.1 Binary logistic regression

We define a particular generalized linear model namely the binary logistic regression. It is widely used as a discriminative classification model for the two class case $|\mathcal{T}| = 2$. We will point out that a large class of instances is modeled by binary logistic regression. We then refer to Banerjee [3] who showed that this class covers all instances being modeled by logistic regression. Finally we will introduce a function for measuring the fit of a binary logistic regression which can be used to train a logistic regression.

Definition 3.23. ***Logistic regression***
Let $x = (x_1, \dots, x_m)$ be a data-point and let $\beta = (\beta_0, \dots \beta_m)$ be model parameters. Given two classes $\mathcal{C}_1, \mathcal{C}_2$ and the according posterior class probabilities $p(\mathcal{C}_1|x), p(\mathcal{C}_2|x)$ where $p(\mathcal{C}_2|x) = 1 - p(\mathcal{C}_1|x)$. By this it is sufficient to model only one posterior and typically we choose $\mathcal{C}_2$ on which the label 1 is assumed. We introduce (the systemic component of) logistic regression according to Bishop as follows [6, p.205].

$$p(\mathcal{C}_1|x) = \sigma(\beta_0 + \sum_{i=j}^{m} \beta_j x_j),\ p(\mathcal{C}_2|x) = 1 - p(\mathcal{C}_1|x)$$

The binary logistic regression is a generalized linear model GLM with assuming the sigmoid function σ in the role of an activation function.

Definition 3.24. ***Exponential family***
We refer to Banerjee and consider densities of the form $p(x;\theta,\phi) = \exp(x^T\theta - \phi(\theta))p_0(x)$ where p_0 is a non-negative function [3]. By $\mathcal{F}_\phi = \{p_\phi(x;\theta,\phi) | \theta \in \Theta \subseteq \mathbb{R}^d\}$ an ***exponential family*** is defined. $\mathcal{F}_\phi$ is characterized by the function ϕ being called the ***log-partition function*** or cumulant function.

Lemma 3.25.
Let the class conditional probabilities $p(x|\mathcal{C}_j) = p(x;\theta_j,\phi) \in \mathcal{F}_\phi,\ 1 \le j \le k$ be members of the same exponential family $\mathcal{F}_\phi$ where each parameter θ_j corresponds to the respective class $\mathcal{C}_j,\ 1 \le j \le k$. Then the posterior class probability is modeled by a logistic regression, i.e. there exist parameters $\beta_0, \dots \beta_m$ with $p(\mathcal{C}_1|x) = \sigma(\beta_0 + \sum_{i=j}^{m} \beta_j x_j)$. This result is well known and we compose a compact proof by referring to Bishop and Banerjee [6, p.203], [3]

Proof.
We set $a(x) = \log(\frac{p(x|\mathcal{C}_1)p(\mathcal{C}_1)}{p(x|\mathcal{C}_2)p(\mathcal{C}_2)}) = \log(\frac{p(x|\mathcal{C}_1)}{p(x|\mathcal{C}_2)}) + \log(\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)})$. With Bayes Theorem (Theorem 3.22) we can easily show $p(\mathcal{C}_1|x) = \frac{p(x|\mathcal{C}_1)p(\mathcal{C}_1)}{p(x|\mathcal{C}_1)p(\mathcal{C}_1)+p(x|\mathcal{C}_2)p(\mathcal{C}_2)} = \sigma(a(x))$. It remains to show that the so-called log-odds $\log(\frac{p(x|\mathcal{C}_1)}{p(x|\mathcal{C}_2)})$ have an affine linear representation in x. This follows from the assumed structure of members of the same exponential family and the cancellation of mutual factor $p_0(x)$ independent of the parameter θ. □

Remark 3.26.
Lemma 3.25 demonstrates a criterion which identifies instances where logistic regression is applicable. Many common distributions satisfy this sufficient condition making the logistic regression widely usable. However at this point it is not yet known whether a larger class of distributions can be modeled by logistic regression. Banerjee proved a relatively simple criterion characterizing all instances which can be modelled [3]. For a rigorous proof we refer to Banerjee [3].

Definition 3.27. ***Logistic family***
Two densities p_1, p_2 are said to be in the same logistic family $\mathcal{F}_{log}$ if their log-odds is affine-linear $\log(\frac{p_1(x)}{p_2(x)}) = ax + b,\ a \in \mathbb{R}^n, b \in \mathbb{R}$.

Theorem 3.28.

(1) Two class conditionals $p(x|\mathcal{C}_i), p(x|\mathcal{C}_j)$ are in the same exponential family if and only if they are in the same logistic family, see Banerjee [3].

(2) If the log-odds $\log(\frac{p(x|\mathcal{C}_1)}{p(x|\mathcal{C}_2)})$ is affine-linear in x, all class conditionals are in the same exponential family by part (1) of this theorem and the posterior class probability is represented by a logistic regression model $p(\mathcal{C}_1|x) = \sigma(\beta_0 + \sum_{i=j}^{m} \beta_j x_j)$ by Lemma 3.25).

Definition 3.29. ***Binary cross-entropy loss***
Given a data-matrix $X = (x_1^T, \dots, x_n^T)^T$ with a target vector $t = (t_1, \dots, t_n)$. Let $\widehat{y_i} = \sigma(\widehat{\beta}_0 + \sum_{j=1}^{m} \widehat{\beta}_j x_{ij})$ be the estimated labels coming from a logistic regression model. By $\widehat{\beta} = (\widehat{\beta}_0, \dots, \widehat{\beta}_m)$ we denote the parameter vector for the model. We evaluate the likelihood function for Bernoulli-trials $L(t|\widehat{\beta}) = \prod_{i=1}^{n} \widehat{y}_i^{t_i}(1 - \widehat{y}_i)^{1-t_i}$. Then the cross-entropy error is defined as the negative log-likelihood evaluated for the given data and the considered model [6, p.206].

$$E(\widehat{\beta}; t) = -\log(L(t|\widehat{\beta})) = -\sum_{i=1}^{n} t_i \log(\widehat{y}_i) + (1 - t_i)\log(1 - \widehat{y}_i)$$

Note that for minimization with respect to a gradient descent method we need the gradient of $E(\beta, t)$ with respect to the parameters β. Evaluated for the model and data at hand we obtain the following convenient form, see Bishop [6, p.206].

$$\nabla_\beta E(\widehat{\beta}, t) = \sum_{i=1}^{n} (\widehat{y}_i - t_i) x_i$$

3.4.2 Multi logistic regression

Let us generalize the binary logistic regression to the cases $|\mathcal{T}| = k \geq 2$. This introduces multi logistic regression also being called softmax regression. The softmax regression can be embedded in a setup of $k-1$ binary logistic regressions and is also encountered in various more complex models. In this way, the theory concerning binary logistic regression carries over to the multi class case under certain conditions. In addition we will introduce a loss function for softmax regression which is well suited for both training and assessing these models.

Definition 3.30. ***Multiclass logistic regression***
Given k classes $\mathcal{T}$ and the probability vector of class posteriors $(p(\mathcal{C}_1|x),\dots,p(\mathcal{C}_k|x)) \in \mathbb{R}^k$. The probability vector is modelled by applying the softmax on k independant linear models as follows, see Bishop [6, p.209].

$$(p(\mathcal{C}_1|x),\dots,p(\mathcal{C}_k|x)) = softmax(\beta_0^1 + \textstyle\sum_{i=j}^m \beta_j^1 x_j, \dots, \beta_0^k + \sum_{i=j}^m \beta_j^k x_j).$$

This model is also called the softmax-regression.

Lemma 3.31.
The softmax-regression is equivalent to $k-1$ binary logistic regressions, see [6, p.182].

Proof.
Suppose we are given a vector of posterior distributions satisfying a softmax regression $(p(\mathcal{C}_1|x),\dots,p(\mathcal{C}_k|x)) = softmax(\beta_0^1 + \sum_{i=j}^m \beta_j^1 x_j, \dots, \beta_0^k + \sum_{i=j}^m \beta_j^k x_j)$. We aim to identify $k-1$ independent logistic regression models which equivalently model the vector.
By padding $\tilde{x} = (1, x)$ we can define parameters vectors $\beta_i = (\beta_0^i, \dots, \beta_m^i)$ and write the i-th component of the softmax as $p(\mathcal{C}_i|x) = \frac{\exp(\beta_i^T \tilde{x})}{\sum_{j=1}^k \exp(\beta_j^T \tilde{x})}$. By cancellation we can introduce $\tilde{\beta}_k = 0$, $\tilde{\beta}_i = \beta_i - \beta_k$ and rewrite $p(\mathcal{C}_i|x) = \frac{\exp(\tilde{\beta}_i \tilde{x})}{1+\sum_{i=1}^{k-1}\exp(\tilde{\beta}_i \tilde{x})}$, $p(\mathcal{C}_k|x) = \frac{1}{1+\sum_{i=1}^{k-1}\exp(\tilde{\beta}_i \tilde{x})}$. As $p(\mathcal{C}_k|x) = 1 - \sum_{j=1}^{k-1} p(\mathcal{C}_j|x)$ it is sufficient to model $p(\mathcal{C}_j|x), 1 \leq j \leq k-1$. By dividing we obtain $\frac{p(\mathcal{C}_j|x)}{p(\mathcal{C}_k|x)} = \exp(\tilde{\beta}_i \tilde{x}), 1 \leq j \leq k-1$. Thus term $\log(\frac{p(\mathcal{C}_j|x)}{p(\mathcal{C}_k|x)}) = \tilde{\beta}_i \tilde{x}, 1 \leq j \leq k-1$ is linear in $\tilde{x}$ and affine-linear in x. By using Bayes Theorem Theorem 3.22 also the log-odds $\log(\frac{p(x|\mathcal{C}_j)}{p(x|\mathcal{C}_k)}), 1 \leq j \leq k-1$ are affine linear with an added term $\log(\frac{p(\mathcal{C}_j)}{p(\mathcal{C}_k)})$ independent of x. Therefore we identified $k-1$ binary logistic regression tasks which decide if a label is in class k or in a different class j, $1 \leq j \leq k-1$.

Conversely whenever we find a class k (typically called the pivot) satisfying affine-linear log-odds against all other classes we can reconstruct the softmax-regression modeling all posterior class probabilities. □

Remark 3.32.
Thus Lemma 3.31 allows us to extend the theory on binary logistic regression to the softmax regression in the following way. Whenever we are able to identify a class k which admits affine-linear log-odds to all other posterior class probabilities, there exists

a softmax regression which models the posteriors. This is exactly the case if all priors belong to the same exponential family by the characterization of the logistic family $\mathcal{F}_{log}$.

Definition 3.33. ***Cross-entropy loss***
Given a padded data-matrix $X = (x_1^T, \ldots, x_n^T)^T$ meaning $x_i = (1, x_{i2}, \ldots, x_{im}),\ 1 \leq i \leq n$ and one-hot encoded target vectors t_i collected in a matrix $t = (t_1^T, \ldots, t_n^T)^T \in [0,1]^{n \times |\mathcal{T}|}$. Let $\widehat{y}_i = softmax(\widehat{\beta}_1 x_1, \ldots, \widehat{\beta}_k x_k)$ be the probability vector for the i-th data point being estimated by a multi logistic regression with parameter vectors $\widehat{\beta}_1, \ldots, \widehat{\beta}_k \in \mathbb{R}^{m+1}$. We deduce the likelihood function assuming a Bernoulli distribution similar to the binary case $L(t|\widehat{\beta}_1, \ldots, \widehat{\beta}_k) = \prod_{i=1}^n \prod_{j=1}^k \widehat{y}_{ij}^{t_{ij}}$. Then the cross-entropy error is defined as the negative log-likelihood evaluated with respect to a considered model [6, p.209].

$$E(\widehat{\beta}_1, \ldots, \widehat{\beta}_k; t) = -\log(L(t|\widehat{\beta}_1, \ldots, \widehat{\beta}_k)) = -\sum_{i=1}^n \sum_{j=1}^k t_{ij} \log(\widehat{y}_{ij})$$

Again we deduce the gradient with respect to some parameter vector β_j evaluated for given parameters $\widehat{\beta}_1, \ldots, \widehat{\beta}_k$ which can be used in a gradient-descent method for fitting the model [6, p.209].

$$\nabla_{\beta_j} E(\widehat{\beta}_1, \ldots, \widehat{\beta}_k; t) = \sum_{i=1}^n (\widehat{y}_{ij} - t_{ij}) x_i$$

3.5 Artificial neural networks (ANN)

Neural networks are one remarkable technique suitable for numerous modeling tasks. In fact Hornik, Stinchcombe and White showed that ANN can approximate every non-pathological function. Hence it is justified to call neural networks ***universal approximators*** [18]. Another remarkable property of neural networks is their resilience against the ***curse of dimensionality*** which is a serious issue for many other techniques. This phenomenon constitutes by the exponentially increasing demand of data if the dimensionality is increased. Poggio and Liao [31] could formally prove results in this regard.

Throughout the following sections we give a smooth introduction into the field of neural networks. We will start at the ***central building block*** of these models being a neuron. We will then compose increasingly more complex models and highlight their capabilities and drawbacks. When appropriate we will point out further variants of networks while staying within the respective paradigm. One non trivial task is the act of ***fitting a neural network*** to data. The potential amount of parameters requires specialized techniques utilizing the network structure in order to achieve reasonable convergence. In this regard, the generally applicable error backpropagation method turns out to be helpful allowing for feasible training times. The goal of this section will be to convey the techniques in sufficient accuracy and detail such that one would be able to build and train basic neural nets from the ground up. In practice however the widely available libraries are highly useful and recommended. Still we firmly believe it is important to look under the hood of neural networks in great detail at least once.

3.5.1 (Simple) neurons

The basic building blocks of neural networks are called ***neurons***, units, or nodes. The analogy to biological nervous systems is evident and helps grasping the fundamental ideas, see below Figure 3.2. Put simply, a biological neuron receives an electrical impulse from other neurons and based on this may or may not generate an output signal itself. This signal in turn is sent to other neurons forming a complex chain of activation and reaction. The key feature of these networks appears to be the network structure which lifts the power of the quite simple atomic units being neurons. Learning the networks forms new links, and if one forgets, the unused links vanish gradually.

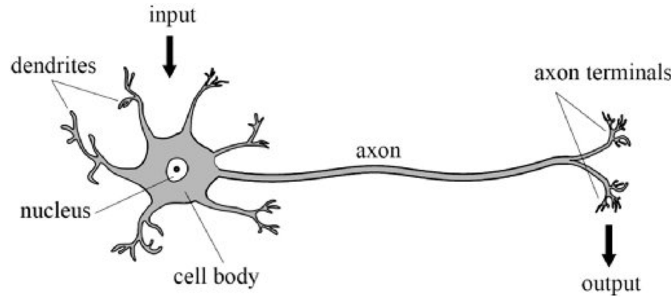


Figure 3.2: biological neuron; Neves, Gonzalez, Leander Karoumi [30]

Inspired by this biological construction, mainly computer scientists at first implemented this core idea in the form of neural networks. Now let us dive into the mathematics of neurons and in particular define what is called the simple neuron. A neuron may assume many concrete forms but ***in general is a function*** mapping from $\mathbb{R}^m$ to $\mathbb{R}$ where m is an input dimension not being specified any closer for now.

While this definition captures any feasible and perhaps even infeasible approach, we fix a certain structure of neurons admitting wishful properties. The first restriction one generally imposes on neurons is a two stage composition. In the first state a so-called ***net input function*** (or transfer function) reduces the dimension of an m-fold input vector to a single number. The second stage is then formed by the application of an ***activation function***. The activation function serves the task of adding a non-linearity and we will reason upon this below in Section 3.5.3. Another demand posed on an activation function is differentiability which is discussed in Section 3.5.3. Typically activation functions also limit the output range. In case the range is limited to an interval one calls such activation functions squishing functions.

Now let us restrain this two stage set up even more by demanding the net input function to simply be a summation. This set of restraints defines the ***simple neuron***. Let us narrowly expand the view from a simple neuron to its connecting links as depicted in Figure 3.3.

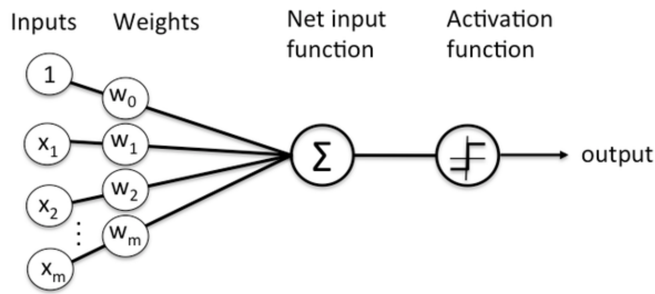


Figure 3.3: ANN neuron; Mitchell, Machine Learning

The idea of links (in a biological sense) is transferred to mathematics in the form of directed weighted edges (in the sense of graph theory). The neuron being a function is interpreted as a node in a ***directed graph***. This is the typical abstraction within which neural networks are being discussed. The direction of edges which connect neurons implies which neuron sends its output number to another neuron as an input. Sending an output x_i of a neuron i via an edge involves multiplying a parameter $w_i \in \mathbb{R}$ which reflects the strength of the connection. In this way, a simple neuron embedded in the network assumes the form of a ***generalized linear model*** $y = a(\sum_{i=1}^{m} w_i x_i)$. y denotes the output of the particular neuron, the $x_i,\ 1 \le i \le m$, are outputs of origin neurons and the $w_i,\ 0 \le i \le m$, reflect the trainable parameters. In this sense, parameters of GLM exactly correspond to the trainable parameters of ANN.

Note that in order to linearly shift the input of the activation function an artificial ***bias neuron*** corresponding to the added bias parameters (in the sense of linear regression models) is added. This principle is reflected by adding an artificial neuron which merely sends a 1. Bias neurons are independent of the input (or context) x and do not receive any input itself. In this way, the bias neurons enhance the abilities of simple neurons.

3.5.2 Single layer neural networks (SNN)

Let us expand from the simple neuron and its close neighborhood by arranging several simple neurons in the form of an array. This forms what is called a (hidden) layer. The output of hidden layers can only be experienced indirectly hence the name. The final layer in this regard is not hidden and is called the output layer. We refer to Bishop's book where a compressed introduction to the following is presented [6, p.225]. In order to avoid confusion we note that there are many names reflecting one and the same concept of the following simple neural network. Most notably the terms multilayer perceptron, and single layer neural network (SNN) are being used. If one was precise perhaps the most suitable name is single hidden layer simple feed forward neural network. In our narrow context the abbreviate (SNN) will do.

Structure of the simple neural network

In order to point out the feedforward nature of SNN let us accompany a data point $x = (x_1, \dots x_m)$ on its journey through an SNN, see Figure 3.4.

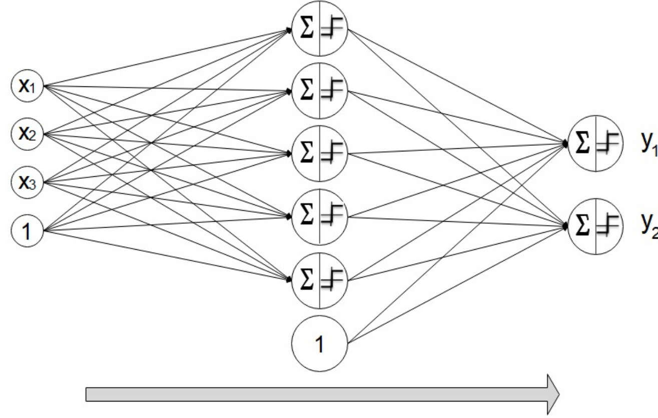


Figure 3.4: SNN; example

The starting point of x is being abstracted by m ***input neurons***, each holding some value x_j. Input neurons merely serve the task of feeding forward a point via edges. It is visible that within the previously mentioned hidden layer there are no connections, but all possible connections to both neighboring layers are present. Let us fix one of the h hidden neurons, say i. The neuron i receives its input in form of a weighted vector $(\beta_{i0}, x_1\beta_{i1}, \dots x_m\beta_{im}) \in \mathbb{R}^{m+1}$, recall the act of sending inputs via edges. Note that already the input layer harbors a bias neuron.

This input vector is then ***processed by simple neurons*** as defined in Section 3.5.1. By applying the net activation function in i we obtain a weighted sum $s_i^{(1)} := \sum_{j=0}^{m} \beta_{ij}^{(1)} x_j = \beta_{i0} + \beta_i^{(1)} x^T \in \mathbb{R}$. On $s_i^{(1)}$ a suitable activation function $a^{(1)}$ is applied providing the output of neuron i by $y_i = a^{(1)}(s_i^{(1)})$. Note that within any layer we generally assume the same scalar activation function being used in each neuron. Between different layers the activation function may however vary. We understand all actions within a layer being taken in parallel and thus compose a vector $(y_1, \dots y_h)$ being associated to the hidden layer.

We again add a bias neuron and propagate $(1, y_1, \dots y_h)$ towards the ***output layer*** via edges meaning we again apply a collection of independent weights. In contrast to the input layer, the output layer is comprised of simple neurons again performing the tasks in parallel. In the instance of SNN, we assume the output layer activation functions in each node to be the identity $a^{(2)} := id$. The appropriate size (i.e. number of nodes) k of the output layer matches the size of the target space being modeled by default. Briefly summarizing the journey of x we look at a particular output neuron κ, $1 \leq \kappa \leq k$, where

the following equation is providing a comprehensive description. In this notation the y_κ is one component of the predicted output vector of SNN.

$$y_\kappa = a^{(2)}(\sum_{j=1}^{h} \beta_{\kappa j}^{(2)} a^{(1)}(\sum_{i=1}^{m} \beta_{ji}^{(1)} x_i + \beta_{j0}^{(1)}) + \beta_{\kappa 0}^{(2)})), \ 1 \leq \kappa \leq k$$

Please note that in Figure 3.4 the input layer is of size $m = 3$, the hidden layer is of size $h = 5$ and the output layer is of size $k = 2$. Thus the relation between a 3-fold data point and a respective 2-fold target vector is being modeled. By the following Figure 3.5 let us again draw the analogy to biology and look at the composition of a few neurons to a small biological neural net.

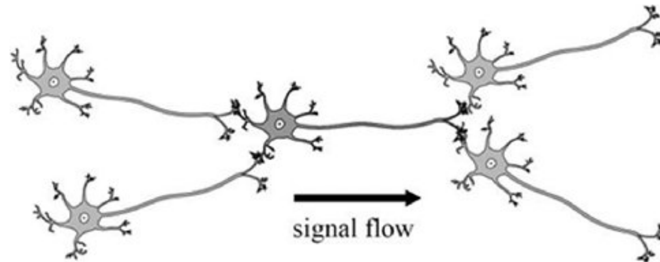


Figure 3.5: biological neural network; Neves et al. [30]

Choosing activation functions

One notable fact concerning hidden layer activation function, is they only enable the full power of a network if chosen to be non linear. In SNN it is clearly visible that by choosing $a^{(1)} = id$ we simply obtain a collapsed SNN being a GLM. Perhaps it is also clear that a scaling of inputs and outputs is realized via the weights being associated to edges. Thus activation functions which only differ by a multiplied constant to the input respectively the output form an equivalent class and add the same abilities to ANN. Regarding the activation function ***prominent candidates*** are σ, tanh, id or the rectified linear unit $relu = max(0, .)$. As it will become clear in Section 3.5.3 the differentiability of activation function is a common claim. In case of using $relu$ we conventionally set the derivative in 0 to 0. Typically the hidden layer is named according to the activation function, e.g. relu-layer, linear-layer (using id), sigmoid-layer or tanh-layer.

In our setup SNN is well suited to perform regression tasks since $a^{(2)} = id$ does not limit the output range. By a slight alteration of SNN also classification on k classes in the form of a discriminative model can be performed. Instead of applying a component wise output activation $a^{(2)}$ we allow for vector-to-vector functions being applied on the weighted sums present in the output layer. In particular the application of softmax provides a model suitable for modeling k-fold vectors of class posterior probabilities. In the case $k = 2$ a single output neuron with $a^{(2)} = \sigma$ is sufficient by $p(\mathcal{C}_2|x) = 1 - p(\mathcal{C}_1|x)$ and a generalization to vector-to-vector activation is not required.

Moving on from SNN

Let us briefly point out the leverage point which allows us to naturally extend the concept of SNN.

Perhaps most apparent is the ***addition of hidden layers*** to the network. The interpretation of neural networks as graphs or sometimes also as stacks of layers is commonly used and allows us to compactly describe the form of said models. Usually individual layers are connected to all respective neurons in the neighboring layers. This instance is called a dense layer.

Another component which can be adapted in SNN are the ***specific neurons***. In order to dive into this subject, it is convenient to stick with the interpretation of networks as a stack of layers. For instance a convolutional layer compresses a specific set of outputs from the previous layer to a single number being well suited to perform image related tasks. A dropout layer serves the task of adding a random noise avoiding overfitting. In order to define recurrent layers in Section 3.5.4 we will however again return to the neuron-level. This version of nodes is able to pick up on sequential information well suited for text and language processing.

More generally a recurrent neural network can be interpreted as an extension of a network by ***adding edges*** forming loops. This is yet another point of expanding from SNN. Being able to interpret (simple) recurrent neurons as an addition of self loops allows for training (simple) recurrent neural networks by relying on well established techniques, see Section 3.5.3 and in particular Section 3.5.4 where we thoroughly introduce recurrent neural networks.

Let us once more point out the addition of ***bias neurons*** which shall expand the capabilities of neural networks. In the case of using summation as a net input function, they allow for shifting the input of the activation function. When working with other types of neurons though this concept may be revised.

Fitting neural networks

Thus far, we were concerned with the structure of neural networks. In order to fit neural networks to given data we fix some aspects of the model which defines what is called its ***architecture***. These aspects are the sizes and respective activation functions being used in the individual layers. Unless specified differently we assume the addition of a bias neuron in each layer except for the output layer, and we assume all layers to be dense. In this way when using SNN the only parameters left are the ***weights of edges*** denoted by β. We thus identify a neural network having a fixed specified architecture as a parametric function $f : \mathbb{R}^m \to \mathbb{R}^k$. We use the notation $f(x;\beta) = y$ where x is a data point and y is the prediction of f evaluated according to the concrete weights β. It is apparent that neural networks which assume a fixed architecture can be fitted by using non-linear optimization techniques as presented in Section 3.2.1.

3.5.3 Feed-forward neural networks (FFNN)

Having discussed the basic set up of neural networks we employ the leverage point of adding several hidden layers as suggested in Section 3.5.2. This natural extension allows the formation of so-called feed forward neural networks (FFNN). Again the terminology used is not unique across the literature. FFNN can again be defined very generally and we will settle with a simple but widely used version which we will refer to as the deep neural network DNN. This instance is perhaps first thought of when neural networks are being mentioned. DNN (already) allows for astonishing modeling capabilities, see Section 3.5.3. Moreover they motivate the term ***deep learning***. This term does not only refer to the depth (i.e. number of layers) of these networks but also reflects the deep insights into data being permitted by DNN and extensions of such. They are able to pick up seemingly irrelevant aspects and compose them into strong predictions. However, by the complexity of these models the predictions can hardly be explained which raises all sorts of issues – not only regarding the mathematics.

Definition 3.34. ***Feed-forward neural network (FFNN)***
We refer to Goodfellow et al. where a definition of feed-forward neural networks (FFNN) via a collection of iterated functions is outlined [13, p.168 ff.]. Let $D \in \mathbb{N}$ denote the ***depth of FFNN***.
Let $f^{(d)} : \mathbb{R}^{(h^{(d-1)})} \to \mathbb{R}^{(h^{(d)})},\ d \in [D]$ be functions called ***layers*** mapping between spaces of dimensions $h^{(d)} \in \mathbb{N},\ d \in [D] \cup \{0\}$ called the respective ***layer size*** (or the width of the layer or the number of neurons). The last layer $f^{(D)}$ is called the ***output layer*** and all other layers are called ***hidden layers***. The input size is denoted by $m := h^{(0)}$ and the output size is denoted by $k := h^{(D)}$. An FFNN is defined as a mapping f simply by iterating the layers as follows.

$$f : \mathbb{R}^m \to \mathbb{R}^k \text{ where } f := f^{(D)} \circ \cdots \circ f^{(1)} \tag{FFNN}$$

Each layer d is being interpreted as $h^{(d)}$ individual component functions forming the neuron in the sense of Section 3.5.1. The neurons comprising one layer are thought of as acting in parallel.

Remark 3.35.

(1) Note that the notion of an edge in FFNN (in the sense of graph theory) is not clearly defined at this point. Nonetheless such a notion can be introduced by looking at which input arguments being the outputs of the previous layer are present in the individual component functions of a considered layer. In particular, note FFNN does not call for the multiplication of weight parameters to outputs of neurons, and the integration of parameters into FFNN can take broad forms.

(2) Note that an FFNN may also be called a deep feed-forward network or a multi-layer-perceptron in the literature. It captures a broad variety of artificial neural networks. The very similar term "deep neural network" (DNN) however refers only to a specific variant of a FFNN and shall not be confused.

Definition 3.36. ***Deep simple feed forward neural networks (DNN)***
We define simple deep neural networks (DNN) by restricting the generality of FFNN by using only simple neurons as follows.

Let $y^{(0)} := x \in \mathbb{R}^m$ be an input data point and denote by $h^{(D)} = k$ the output size. Furthermore consider $D-1$ hidden layers sizes $h^{(1)}, \dots h^{(D-1)}$. In order to define DNN each hidden neuron is required to ***assume the form of a simple neuron***, see Section 3.5.1. Thus the i-th output of the d-th layer is given as follows.

$$f_i^{(d)}(y^{(d-1)}) = a^{(d)}(\beta_i^{(d)} y^{(d-1)} + \beta_{i0}^{(d)}) = y_i^{(d)}, \ 1 \le d \le D-1, 1 \le i \le h^{(d)}$$

Note that the same real activation function $a^{(d)}$ is applied in each neuron of the hidden layer $d \in [D-1]$ but the ***activation functions*** may vary between layers. In the output layer D we more generally allow a vector-to-vector function $a^{(D)} : \mathbb{R}^k \to \mathbb{R}^k$. This allows for performing both regression and classification, see Section 3.5.2. The action performed in the output layer is given as follows.

$$f^{(D)}(y^{(D-1)}) = a^{(D)}(\beta^{(D)} y^{(D-1)} + \beta_0^{(D)}) = y^{(D)}.$$

The $\beta^{(d)}, d \in [D]$ are matrices of appropriate size. A single index reflects the extraction of the corresponding row and a double index reflects the extraction of an according element. We omit the transpositions for sake of a compact notation.

It is apparent that DNN are a ***natural extension of SNN*** by adding dense hidden layers comprised of simple neurons (except for perhaps the output layer) and in turn DNN is a special instance of FFNN. The visualization of DDN as a network similar to Figure 3.4 is evident and the propagation of values along edges corresponds to multiplication of weight parameters. Also the concept of bias neurons which we use by default shall not remain unmentioned. A DNN is interpreted as a parametric function $f(x;\beta)$ mapping from $\mathbb{R}^m$ to $\mathbb{R}^k$. This allows DNN to be used as a machine learning model and parameters are in general being determined by applying nonlinear optimization, see Section 3.2.1.

Definition 3.37. ***Identify DNN by architecture***

(1) Let us formally introduce classes of DNN which agree up to the parameters. The class of all DNN having the same architecture is denoted by $\aleph^m_{(h^{(1)},\dots,h^{(D)})}(a^{(1)},\dots,a^{(D)})$. The $h^{(d)}$ specify the hidden layer sizes and output size $h^{(D)} = k$. The $a^{(d)}$ specify the scalar-to-scalar activation functions and $a^{(D)}$ specifies the vector-to-vector output activation. m indicates the input size and DNN is ***determined up to weight parameters*** β.

(2) Denote by $\aleph^m_{(*,\dots,*,h^{(D)})}(a^{(1)},\dots,a^{(D)}) := \bigcup_{d=1}^{D-1} \bigcup_{h^{(d)}=1}^{\infty} \aleph^m_{(h^{(1)},\dots,h^{(D)})}(a^{(1)},\dots,a^{(D)})$ the family of networks mapping from $\mathbb{R}^m$ to $\mathbb{R}^{(h^{(D)})}$ by assuming ***arbitrary hidden layer sizes***. The $*$ shall simply indicate we assume this parameter to be arbitrary.

Approximating functions with DNN

Similar to logistic regression and generalized linear models we outline the modeling power of DNN. Results regarding this are called ***universal approximation theorems***.

One of the first results of this type was due to Cybenko where only ***one hidden layer of arbitrary size*** with sigmoid-activation is assumed [8]. Hornik et al. proved a similar result in the same year of 1989 which extends to all measurable functions [18]. This particular result is often referred to as the universal approximation theorem for DNN. It shows DNN are universal approximators meaning all non-pathological functions can be approximated in arbitrary accuracy by a suitable DNN.

Maiorov and Pinkus and independently also Ismailov presented results proving that already a ***bounded number of neurons*** is sufficient for universal approximation abilities [26], [19]. There are many further variants of such theorems for instance also concerning the depth as opposed to the width of DNN.

Theorem 3.38. ***Cybenko (1989)***
If $a^{(1)}$ is continuous and sigmoidal then $\aleph^{m}_{(*,1)}(a^{(1)}, id)$ is dense (w.r.t. supremum norm) in the space of continuous functions on the unit cube $C([0,1]^m)$ [8].

Remark 3.39.
Note that the extension to several output-neurons and specific output-activation is straight-forward. Similar extensions hold true for the other results of this type. We will roughly sketch the ideas to prove our assertions.

(1) To obtain the multi-output case on k neurons we can simply copy the hidden layer k times. The associated weights are independent of each other and we can obtain arbitrary closeness on all outputs simultaneously.

(2) Suppose we want to approximate $g \in C([0,1]^m)$ with some $\tilde{g} \in \aleph^{m}_{(*,1)}(a^{(1)}, a^{(2)})$. Crucially, whenever $f \in \aleph^{m}_{(*,1)}(a^{(1)}, id)$ then $a^{(2)} \circ f \in \aleph^{m}_{(*,1)}(a^{(1)}, a^{(2)})$. We can extend to homeomorphic output activation $a^{(2)}$ as follows. By the theorem and continuity we can find δ sufficiently small and $f \in \aleph^{m}_{(*,1)}(a^{(1)}, id)$ such that whenever $||f - {a^{(2)}}^{-1} \circ g|| < \delta$ is close, also the images under $a^{(2)}$ are close $||a^{(2)} \circ f - g|| < \epsilon$. Thus g is approximated arbitrarily well by $\tilde{g} := a^{(2)} \circ f \in \aleph^{m}_{(*,1)}(a^{(1)}, a^{(2)})$.

(3) Furthermore note that the restriction on the unit cube can also be addressed. Suppose we aim to model a continuous function $g : X \subseteq \mathbb{R}^m \to \mathbb{R}$ where X is compact. Then we can homeomorphically deform X into $[0,1]^m$ by a contraction c. We approximate c by a neural network f_1 and use said outputs as inputs for the neural network f_2 approximating $g \circ c^{-1}$. Note that the activation functions are continuous and thus also f_1, f_2 are continuous functions. Via concatenation we obtain a neural network $f_1 \circ f_2$ having three hidden layers which is a continuous function. Now in order to ensure a small error in the end we need to be able to satisfy $||f_1 \circ f_2 - g|| < \epsilon$ for any small ϵ. Now by demanding $||Df_2|| < M$ a bounded differential we can estimate $||f_2(x) - f_2(x + \epsilon')|| < \epsilon' M \overset{!}{=} \epsilon$ by using a mean value equality. Thus we need

to choose f_1 such that the errors on it's output x are at most $\frac{\epsilon}{M}$ which provides a sufficiently small error at the end.

A slightly dirtier approach would be to use an input transformation which is not required to be realized by a network. We are using $(g \circ c^{-1}) \circ c = f \circ c$ by assuming said contraction c and thus have defined some function f. We transform the input data to $c(X) \subseteq [0,1]^m$ and can find a single-hidden layer network approximating $f = g \circ c^{-1}$. In particular, this shall point out that preprocessing the data and in particular scaling data to the unit cube is good practice in the context of neural nets.

In the following, we display results which bound the required number of hidden neurons. It shows that a smart choice of activation functions determines the modeling power whence the hidden layers are sufficiently (but not awfully) large compared to the input size.

Theorem 3.40. ***Maiorov and Pinkus (1999)***
Let $X = [0,1]^m \subseteq \mathbb{R}^m$ be a unit cube. Then there is an analytic, strictly increasing sigmoidal function $\widehat{\sigma}$ with $\aleph^m_{(3m,6m+3,1)}(\widehat{\sigma},\widehat{\sigma},id)$ dense (w.r.t. supremum norm) in $C([0,1]^m)$, [26].

Remark 3.41.

(1) Maiorov and Pinkus presented a constructive proof giving us insight into the choice of $\widehat{\sigma}$ [26]. Also a version of the theorem which only demands the function being C^∞ instead of analytic is mentioned.

(2) The following result shows that also less hidden neurons are sufficient for arbitrary approximation quality. Instead of an analytic activation function we only require a C^∞ function which has a slightly more specific monotonicity behaviour.

Theorem 3.42. ***Ismailov (2014)***

(1) Let $X \subseteq \mathbb{R}^m$ be compact. Then for any $\alpha \in \mathbb{R}$ and $\lambda > 0$ there is a $C^\infty(\mathbb{R})$ sigmoidal function $\widehat{\sigma}$ which on $(-\infty,\alpha)$ is strictly increasing and λ-strictly increasing on $[\alpha,\infty)$ with $\aleph^m_{(m,2m+2,1)}(\widehat{\sigma},\widehat{\sigma},id)$ dense in $C(X)$.

(2) If we restrict to positive α we obtain density on the unit-cube. Meaning let $X = [0,1]^m \subseteq \mathbb{R}^m$ be a unit cube. Then for any $\alpha > 0$ and $\lambda > 0$ there is a $C^\infty(\mathbb{R})$ sigmoidal function $\widehat{\sigma}$ which on $(-\infty,\alpha)$ is strictly increasing and λ-strictly increasing on $[\alpha,\infty)$ with $\aleph^m_{(m,2m+2,1)}(\widehat{\sigma},\widehat{\sigma},id)$ dense in $C([0,1]^m)$.

Training of DNN

As already pointed out for SNN in section, 3.5.2 the training of a simple DNN similarly fixes the architecture and only ***optimizes the weight parameters*** being associated to edges by using a suitable nonlinear optimization technique, see Section 3.2.1. By typically encountering a large number of parameters, we aim to speed up the training process by utilizing the underlying network structure in the form of dynamic programming. The technique of ***error backpropagation*** implements this and allows for feasible training times even for relatively large networks. In contrast to the evaluation of DNN being a forward propagation of a data-point through the network, the parameter updates are generally thought of as a backwards propagation (hence the name) of error terms starting with an error measure at the output layer with help of a loss function. In the following, let us show how this looks in detail.

Lemma 3.43. ***Error backpropagation (BP or backprop)***
Suppose we are given a loss function of the form $L(f(x,\beta),y) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i,\beta),y)$ averaging a per point loss term $L(f(x_i,\beta),y)$. Suppose $f \in \aleph^{m}_{(h^{(1)},\dots,h^{(D)})}(a^{(1)},\dots,a^{(D)})$ assumes the form of a simple deep neural network with parameters β with the limitation of all $a^{(d)}$ being differentiable activation functions. All notations are according to Definition 3.36.

(1) Then the ***gradient at an arbitrary edge*** is given as follows.

$$\nabla_{\beta_{ij}^{(d)}} L(f(x_i,\beta),y) = \delta_i^{(d)} y_j^{(d-1)} \text{ if } j \geq 1 \text{ and}$$
$$\nabla_{\beta_{ij}^{(d)}} L(f(x_i,\beta),y) = \delta_i^{(d)} \text{ for } j = 0 \text{ the bias neuron.}$$

(2) The quantity $\delta_i^{(d)}$ is called the ***error at the neuron*** i in layer d. for $d = D$ the errors are directly dependent on the loss-function.

$$\delta_i^{(D)} = \frac{\partial L(f(x_i,\beta),y)}{\partial s_i^{(D)}}$$

Note that $f(x_i,\beta) = f^{(D)}(y^{(D-1)}) = a^{(D)}(\beta^{(D)}y^{(D-1)} + \beta_0^{(D)}) = y^{(D)}$ is indeed a function in $s_i^{(D)} = \beta_i^{(D)}y^{(D-1)} + \beta_{i0}^{(D)}$ and we can apply the chain rule.

(3) For all neurons i, $0 \leq i \leq h^{(d)}, 1 \leq d \leq D-1$, in a some hidden layer d the respective error ***only depends on the current and subsequent layer***. The following equation encapsulates this observation and gives rise to the dynamic program.

$$\delta_i^{(d)} = a'^{\,(d)}(s_i^{(d)}) \sum_{j=1}^{h^{(d+1)}} \delta_j^{(d+1)} \beta_{ij}^{(d+1)}$$

Note that $a'^{\,(d)}$ is the derivative function of the univariate $a^{(d)}$.

(4) Thus, summarizing, we can compute the gradient $\nabla_{\beta_{ij}^{(d)}}$ with respect to any parameter $\beta_{ij}^{(d)}$ by computing the errors associated to the last layer as in (2) and then

propagating back the error through the layers by using the equation in (3) This defines the dynamic program which is generally referred to as ***error backpropagation*** (BP or backprop).

Proof.
The proof is slightly technical and involves the multivariate chain-rule. Still we regard the proof being valuable in order to gain insight on how the quantities arise.

Ad (1) We start by calculating the gradient $\nabla_{\beta_{ij}^{(d)}} L(f(x_i, \beta), y)$ associated to an arbitrary edge (j, i) leading to some neuron i in some hidden layer $d \in [D]$.

As each $s_i^{(d)}$ is a function in $\beta_{ij}^{(d)}$ the chain rule is applicable and defines the error as follows.

$$\nabla_{\beta_{ij}^{(d)}} L(f(x_i, \beta), y) = \frac{\partial L(f(x_i, \beta), y)}{\partial s_i^{(d)}} \frac{\partial s_i^{(d)}}{\partial \beta_{ij}^{(d)}} =: \delta_i^{(d)} \cdot \frac{\partial s_i^{(d)}}{\partial \beta_{ij}^{(d)}}.$$

The error is thus obtained by looking at the change of the loss when manipulating the output of the respective net input function $s_i^{(d)}$. The inner derivative captures the change of the net input function if the weight is manipulated. In the case of a bias neuron $j = 0$ we merely send a 1 regardless of the context and observe an direct impact on the net input function $\frac{\partial s_i^{(d)}}{\partial \beta_{ij}^{(d)}} = 1$. For other nodes j in the previous layer the change in loss is proportional to the context $\frac{\partial s_i^{(d)}}{\partial \beta_{ij}^{(d)}} = y_j^{(d-1)}$. The context simply is the value being propagated through the considered edge if x is evaluated. In this regard we again recall that bias neurons are thought of as neurons sending a 1 and this situation exactly corresponds also for $j = 0$.

Ad (2) If we plug in $d = D$ we immediately observe the equation in (2). The errors at the output layer are then propagated backwards through the network. Thus this explains the initialization of the back propagation.

Ad (3) Now let us determine the remaining errors which results in a full explanation of the back propagation procedure. Thus suppose $d < D$ and we look at the subsequent layer $d + 1$. We track the influence of $\beta_{ij}^{(d)}$.

$$\beta_{ij}^{(d)} \to s_i^{(d)} \to a^{(d)}(s_i^{(d)}) \to (s_1^{(d+1)}, \dots, s_{h^{(d+1)}}^{(d+1)}) \to \dots \to L(f(x_i, \beta), y)$$

In layer d the parameters $\beta_{ij}^{(d)}$ only influences the computations at one neuron i via the net input $s_i^{(d+1)}$ present in i. This term is then passed through the real differentiable activation function $a^{(d)}$. The obtained output is passed to the entire subsequent layer where weighted sums $(s_1^{(d+1)}, \dots, s_{h^{(d+1)}}^{(d+1)})$ are computed. Those are then pushed forward through the network until we finally compute $L(f(x_i, \beta), y)$. As this sequential analysis entirely captures the influence of $\beta_{ij}^{(d)}$

we can apply the multidimensional chain-rule by factoring out an inner function $(s_1^{(d+1)}, \dots, s_{h^{(d+1)}}^{(d+1)})$ depending on $\beta_{ij}^{(d)}$ in order to compute the error $\delta_i^{(d)}$ as follows.

$$\begin{aligned}\delta_i^{(d)} =& \frac{\partial L(f(x_i,\beta),y)}{\partial s_i^{(d)}} = a'^{\,(d)}(s_i^{(d)}) \sum_{j=1}^{h^{(d+1)}} \frac{\partial E(\beta)}{\partial s_j^{(d+1)}} \frac{\partial s_j^{(d+1)}}{\partial s_i^{(d)}} \\ =& a'^{\,(d)}(s_i^{(d)}) \cdot \sum_{j=1}^{h^{(d+1)}} \delta_j^{(d+1)} \beta_{ij}^{(d+1)}\end{aligned}$$

Note that the error depends on the errors of later layers, the outgoing edge weights and a quantity $a'^{\,(d)}(s_i^{(d)})$ depending on the sensitivity (derivative) of the activation function at the obtained input $s_i^{(d)}$.

□

Remark 3.44.
As stated in the above point (4), this lemma tells us how we can compute the gradients with respect to an individual parameter in a neural network by using dynamic programming. The back-propagation can be extended quite easily to non-simple networks which for example assume vector-to-vector activation functions, disallow biases or disallow certain edges. Moreover, we will see in the following section that this back-propagation schema can also be applied in a straight-forward fashion to networks having loops, see Section 3.5.4.

3.5.4 Recurrent neural networks (RNN)

Regarding a neural network as a graph we extend these models by ***allowing self-loops and backward edges***. Such networks are called recurrent neural networks (RNN). The feeding back of previously processed information thus introduces an inherent sequence notion making RNN well suited for processing sequential data such as speech, text, time dependent data or similar.

We will first introduce ***notations*** regarding sequential data and point out how such feedback-loop is incorporated in a neural network in terms of mathematics. Again we will ***reduce the full generality of RNN*** and introduce Elman (or simple) recurrent neural networks. Then we will point out one large problem of this recurrent structure namely the ***vanishing (and exploding) gradient problem***. Hochreiter and Schmidthuber addressed this issue among others via constructing a particular kind of gated neuron, the ***LSTM-cell*** [16]. This invention contributed largely to the success of RNN. We base this section on Goodfellow et al. introducing the key concepts of recurrent neural networks [13, p.375 ff.]. Furthermore we refer to Schäfer and Zimmermann who provide ***approximation results*** for RNN and a slightly more general notion of RNN [32].

Definition 3.45. ***Time series***
Let $[\tau]$ be a discrete time window of length τ. We define $[\infty] := \mathbb{N}$ a time window of infinite length. An element $(x^{(1)}, \dots, x^{(\tau)}) \in (\mathbb{R}^m)^\tau$ being a a τ-fold vector of elements in $\mathbb{R}^m$ is called a discrete time series of length τ. We understand a time series as a sequence of data-points.

Definition 3.46. ***Open time discrete dynamical systems***
Given time series $(x^{(1)}, \dots, x^{(\tau)}) \in \mathbb{R}^{m[\tau]}$ and $(h^{(0)}, \dots, h^{(\tau)}) \in \mathbb{R}^{h[\tau+1]}$ and let $f(.,.,\theta) : \mathbb{R}^m \times \mathbb{R}^h \to \mathbb{R}^h$ be a parametric function satisfying the following set of equations.

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}, \theta),\ 1 \leq t \leq \tau$$

In this setup $f(.,.,\theta)$ is called the ***state transition function***, $h^{(t)}$ is called the ***state*** at time t, $0 \leq t \leq \tau$, and $x^{(t)} \in \mathbb{R}^m$ is called the input or ***external signal*** at time $t \in [\tau]$.

The dynamical system is already determined by the input-data, the initial state $h^{(0)}$ and the parametric transition function $f(.,.,\theta)$. An open dynamic system is also called a dynamic system driven by an external signal. As a side remark, if f does not depend on the external signal but only on the previous state we call such a dynamical system autonomous.

Definition 3.47. ***Recurrent neural network***
A recurrent neural network in its full generality is given as a dynamical system. Referring to Schäfer et al. we present recurrent neural network in the thus resulting so called state space representation and introduce the associated terminology [32]. We furthermore refer to Goodfellow at al. who provides a slighly different definition of RNN by disregarding the output equation [13, p.375 ff.].

(1) A ***recurrent layer*** is defined with help of a dynamical system as follows.

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}, \theta),\ t \in [\tau]$$

Furthermore let the ***output equations*** producing an output $y^{(t)} \in \mathbb{R}^k$ at each time-step $t \in [\tau]$ be defined as follows.

$$y^{(t)} = o(h^{(t)}),\ t \in [\tau]$$

Similar to non-recurrent neural networks $o(.)$ is a generally defined output activation function mapping between spaces of appropriate dimension. In this way, an RNN maps a time series $(x^{(1)}, \dots, x^{(\tau)})$ to another time series $(y^{(1)}, \dots, y^{(\tau)})$.

The states of the dynamic system $h^{(t)}$ are called ***hidden states***. The initial hidden state $h^{(0)}$ is a model parameter. We interpret an RNN as a function by feeding in the $x^{(t)}$ and the previous hidden-state $h^{(t-1)}$ in an online fashion and successively obtain hidden states $h^{(t)}$ and the outputs $y^{(t)}$. The hidden state $h^{(t)}$ therefore resemble a memory about the past $\{x^{(1)}, \dots, x^{(t)}\}$ and is are also called the ***context*** (at time t), $t \in [\tau]$. It is crucial to note that in each time-step the same state transition function $f(.,.,\theta)$ is applied.

(2) Oftentimes we slightly reduce the full generality of the output activation function and assume a linear function $y^{(t)} = Vh^{(t)}$ where V is a matrix of appropriate, see Schäfer et al. [32]. In this way we can interpret the output layer as a densely connected layer and the usual correspondence between weight parameters and edges applies.

(3) We define the Elman RNN, or the ***simple recurrent neural network (SRNN)*** as published in [10] by further reducing the generality of RNN. Let $a : \mathbb{R} \to \mathbb{R}$ be scalar activation function, $U \in \mathbb{R}^{h\times m}, W \in \mathbb{R}^{h\times h}$ be weight matrices and $\beta \in \mathbb{R}^h$ a bias vector. The hidden states are defined component wise and assume the form of GLM.

$$h_i^{(t)} = a(U_i x^{(t)} + W_i h^{(t-1)} + \beta_i), 1 \le i \le h,\ 1 \le t \le \tau$$

The output equation is a linear transformation using a weight-matrix $V \in \mathbb{R}^{k\times h}$.

$$y^{(t)} = Vh^{(t)},\ 1 \le t \le \tau$$

Please note that this allows for the Elman RNN to be interpreted as a network with the usual weight-edge relation. The recurrent layer in this sense is a dense layer. Moreover the recurrent layer has a self loop at each neuron and in addition has all possible edges within a layer. Most importantly, note that edge weights are time independent. In the following, we will present another possible view of Elman RNN as an equivalent network structure.

Training of SRNN

The essence for being able to train SRNN or similar variants is a technique called ***unfolding*** which allows us to transform a SRNN into a DNN. In the following diagram the idea is laid out.

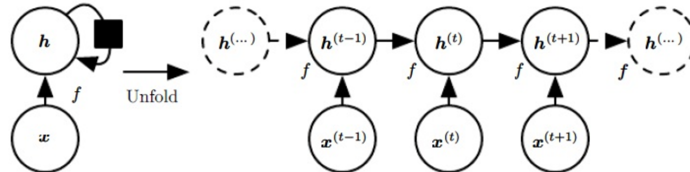


Figure 3.6: unfolding RNN; Goodfellow et al. [13, p.376]

Each circle in Figure 3.6 represents an entire recurrent layer and each edge in the diagram represents having all possible connections between the respective layers. The way in which we obtain the unfolded RNN (right hand side) from the compact interpretation using loops (left hand side) is by copying the recurrent layer $|\tau|$ times. The time invariant edge weights of SRNN are reflected by copies of the respective edge-weights being placed in between the hidden layers $h^{(t)}, t \in [\tau]$, of the unfolded network. Moreover, the input $x^{(t)}, t \in [\tau]$, in this interpretation sends the input via an edge but skips the previous $t-1$

layers which we note is not covered by DNN. Later in Section 3.5.4 we will also include the output generation and for now omit this part of the network.

This unfolding of an SRNN allows for training by using error back propagation. Combining the unfolding and the application of backpropagation we obtain the so called ***backpropagation through time (BPTT)*** which is one variant of training (simple) recurrent neural nets. Note that in particular skipping layers is not problematic for this to work and the updates regarding the copies of edges are simply being added to an aggregate update.

This however introduces one phenomenon which we only very briefly address. By summing updates regarding within one run of BPTT we may generate a sequence of updates referring to the same edge with canceling terms. Since in this instance still parameter updates are proposed, we may infer that it is not yet optimal and the conflicting proposals elongate or prohibit training. These ***conflicting updates*** are for instance being addressed by Hochreiter et al. [16]. Generalizations of BPTT to other recurrent neural networks are possible and widely used. We refer the interested reader to Goodfellow et al. [13, p.384 ff.].

The vanishing/exploding gradient problem

Another more prominent problem which occurs in RNN, in particular when modeling long time sequences, was addressed and analyzed by Hochreiter in his diploma thesis [15]. When using BPTT, we feed back errors through a perhaps very deep unfolded network (when modeling long time series). By having associated the same weights between all hidden layers let us briefly recapitulate the error ***back propagation formula*** in Lemma 3.43 which in this instance allows for simplifications. By encountering only copies of hidden layers and weights we may also lighten the notation and omit the subscript indicating the layer for many quantities. By a straightforward computation it can be shown that the error encountered at a layer d associated to a neuron $j_0 := i$ is given as follows.

$$\delta_i^{(d)} = \sum_{j_1=1}^{h^{(d+1)}} \cdots \sum_{j_t=1}^{h^{(d+t)}} \delta_{j_t}^{(D)} \prod_{\tau=0}^{t} a'(s_{j_\tau}^{d+\tau}) \beta_{j_\tau j_{\tau+1}}$$

It is visible that the weight update for β_{ij} depends on products of the $\beta_{j_\tau j_{\tau+1}}$ in later layers. For larger t (earlier stages) the many factors form a very sensitive error term which is likely to ***become exponentially large or exponentially small*** as Hochreiter elaborated in [15]. By the error terms being directly linked to the sizes of the propagated gradient, this corresponds to vanishing/exploding gradients being propagated backwards in time and substantiates the vanishing/exploding gradient problem.

Interpreted on a higher level, only the more recent data points are remembered properly. The data points which come early in a time-series experience a highly biased gradient update and thus are not incorporated accurately when learning with BPTT. This is interpreted as a ***short term memory effect***. Clearly by this instability the training of RNN by using BPTT is a sensitive and troublesome endeavor which needs be addressed.

In order to resolve this Hochreiter and Schmidhuber suggested a clever architecture which circumvents this problem and moreover addressed conflicting gradient updates [16].

Long short term memory cells (LSTM)

In this section, we refer to Goodfellow et al. [13, p.410] and Hochreiter at al. [16]. The long short term memory cell (LSTM) is a slightly more complex neuron designed for RNN. The key feature of LSTM addresses the vanishing/exploding gradient problem and comprises an in-cell self loop which encapsulates a concept of ***in-cell memory***. This so-called constant error carousel (CEC) only performs affine linear updates and thus is robust towards vanishing/exploding gradients as the gradient propagated along this self loop by BPTT remains constant over time. Therefore this in cell memory does not get biased over time and counters the short term memory effect. The second key feature of LSTM addresses the conflicting gradient update problem by implementing a ***gating structure*** regulating the data flow within and between LSTM cells in a context dependent fashion. This allows for training a context-dependent in-cell filter which only propagates the relevant (desirable) data for performing the gradient update.

In the following Figure 3.7 let us demonstrate the layout of a single LSTM neuron. Note that edges are associated with weight parameters being constant over time. A square on an edge indicates a time delay of one an no square means the quantities are propagated at the current time step. The LSTM in and of itself is a small neural net and the nodes within it are specific functions. We will define the particular components more closely below. For a more theoretical motivation of the design patterns and a careful analysis we refer the interested reader to the original publication by Hochreiter et al. [16].

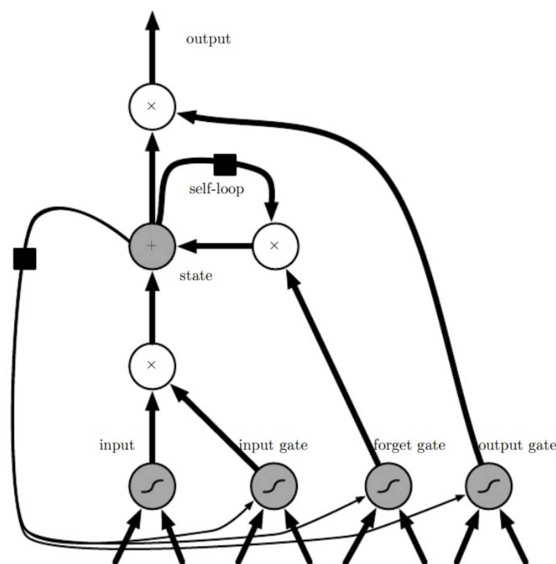


Figure 3.7: LSTM-cell; Goodfellow et al. [13, p.409]

Definition 3.48. ***LSTM-layer***
We define an LSTM-layer of size $h \in \mathbb{N}$ by the actions being taken in an individual LSTM-cell i at a time t and refer to Goodfellow et al. [13, p.410].
By $x^{(t)} \in \mathbb{R}^m$ we denote the current element in the input time series. By $h_j^{(t-1)} \in \mathbb{R}^h$ we denote the hidden state coming from the LSTM-cell j at the previous time-step $t-1$. For defining the individual architectural elements we require a collection of bias vectors $b, b^f, b^g, b^o \in \mathbb{R}^h$, of input-weight matrices $u, u^f, u^g, u^o \in \mathbb{R}^{h\times m}$ and of recurrent weight matrices $w, w^f, w^g, w^o \in \mathbb{R}^{h\times h}$ which in total comprises the parameters of an LSTM cell.

(1) The ***forget-gate*** $f_i^{(t)}$ in cell i is defined as follows.

$$f_i^{(t)} = \sigma\Big(b_i^f + \sum_{j=1}^{m} u_{ij}^f x_j^{(t)} + \sum_{j=1}^{h} w_{ij}^f h_j^{(t-1)}\Big)$$

The input data point $x^{(t)}$ at the time t is weighted and all previously generated hidden states from LSTM-cells $1 \le j \le h$, are incorporated. The bias-term b_i^f enables shifting the input of the activation function σ.

(2) The ***input-gate*** $g_i^{(t)}$ is defined similar to the forget-gate as follows.

$$g_i^{(t)} = \sigma\Big(b_i^g + \sum_{j=1}^{m} u_{ij}^g x_j^{(t)} + \sum_{j=1}^{h} w_{ij}^g h_j^{(t-1)}\Big)$$

(3) The ***cell-state*** $s_i^{(t)}$ at evaluation combines the previous cell-state, the previous hidden state and the current input data point as follows by using gates.

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\Big(b_i + \sum_{j=1}^{m} u_{ij} x_j^{(t)} + \sum_{j} w_{ij} h_j^{(t-1)}\Big)$$

The cell-state resembles the in-cell memory, in particular recall the concept of CEC. Qualitatively speaking, the forget-gate $f_i^{(t)} \in (0,1)$ decides the proportion of the past memory being carried over. The input-gate $g_i^{(t)} \in (0,1)$ decides how much information coming from the other cells and external signal via the term

$$\sigma\Big(b_i + \sum_{j} u_{ij} x_j^{(t)} + \sum_{j=1}^{h} w_{ij} h_j^{(t-1)}\Big)$$

should be added to the in-cell memory. The cell-state experiences only affine-linear updates and implements the CEC as mentioned earlier. It is crucially important to point out that the gates are acting according to the external signal (i.e. input data point) and the hidden-state and thus are context sensitive and not constant through time.

(4) The ***output-gate*** $q_i^{(t)}$ and the hidden-state update $h_i^{(t)}$ are defined as follows.

$$q_i^{(t)} = \sigma\big(b_i^o + \sum_{j=1}^{m} u_{ij}^o x_j^{(t)} + \sum_{j=1}^{h} w_{ij}^o h_j^{(t-1)}\big)$$
$$h_i^{(t)} = \tanh\big(s_i^{(t)}\big) q_i^{(t)}$$

The output unit gate $q_i^{(t)} \in (0,1)$ regulates the portion of in-cell memory which is passed on in the form of hidden state $h_i^{(t)} \in (-1,1)$ to all other cells in the next time-step. Again we notice the output-gate $q_i^{(t)}$ is context dependent.

Approximation power of recurrent neural networks

As already for DNN we ask which types of instances can be modeled by using RNN. For RNN the objects being modeled are time series and thus we ask for approximation results concerning dynamical systems. We refer to Schäfer and Zimmermann who presented such an approximation result as follows [32].

Theorem 3.49. ***Schäfer, Zimmermann (2007)***
Let $f : \mathbb{R}^m \times \mathbb{R}^h \to \mathbb{R}^h$ be measurable and $o : \mathbb{R}^h \to \mathbb{R}^m$ be continuous. Consider a time series $x^{(1)}, \dots, x^{(\tau)}$ and suppose we are given hidden states $h^{(0)}, \dots, h^{(\tau)}$ and outputs $y^{(1)} \dots, y^{(\tau)}$ which satisfy the following equations describing a dynamical system.

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}),\ 1 \le t \le \tau$$
$$y^{(t)} = o(h^{(t)}),\ 1 \le t \le \tau$$

Then there for each $\epsilon > 0$ there exists a simple recurrent neural network with sigmoidal activation $a = (\widehat{\sigma}, \dots, \widehat{\sigma})$ driven by the same external time series $x^{(1)}, \dots, x^{(\tau)}$ which generates outputs $\widehat{y}^{(1)} \dots, \widehat{y}^{(\tau)}$ and hidden states $\widehat{h}^{(0)}, \dots, \widehat{h}^{(\tau)}$ that uniformly approximate the dynamical system. By this we mean $|\widehat{y}^{(t)} - y^{(t)}| < \epsilon,\ t \in [\tau]$ and $|\widehat{h}^{(t)} - h^{(t)}| < \epsilon,\ 0 \le t \le \tau$. The approximating dynamical systems if given as follows.

$$\widehat{h}^{(t)} = a(Ux^{(t)} + W\widehat{h}^{(t-1)} + \beta_0),\ 1 \le t \le \tau$$
$$\widehat{y}^{(t)} = V\widehat{h}^{(t)},\ 1 \le t \le \tau$$

The sigmoidal activation function is defined by component-functions $\widehat{\sigma} : \mathbb{R} \to \mathbb{R}$ which are sigmoidal and monotonically increasing. The $U \in \mathbb{R}^{h' \times m}, W \in \mathbb{R}^{h' \times h'}, V \in \mathbb{R}^{k \times h'}$ are weight matrices and $\beta_0 \in \mathbb{R}^{h'}$ is a bias vector.

Remark 3.50.

(1) Please note that Schäfer et al. used a slightly different definition of sigmoidal namely that the function tends to ± 1 at $\pm\infty$ wheres we defined sigmoidal functions to tend to 0 and 1 respectively. The result however is true in both regards since the weight matrices can compensate for this altered range of hidden states.

(2) The proof presented by Schäfer et al. in [32] involves the universal approximation theorem for DNN for measurable functions by Hornik et al. (1991) [18] which we briefly mentioned earlier.

Design patterns for RNN

We aim to demonstrate slight variants of the SRNN which allow us to model various instances involving time series. Depending whether we generate an output vector $y^{(t)}$ in each time-step or only propagate the last output vector $y^{(\tau)}$, we can deduce ***two basic design patterns*** which both serve distinct purposes.

The first design pattern constitutes by ***generating a network output*** $y^{(t)}$ ***at each time step*** $t \in [\tau]$. It allows for modeling functions between spaces of time-series and hence belongs to the realm of ***sequence-to-sequence models*** (seq-to-seq).

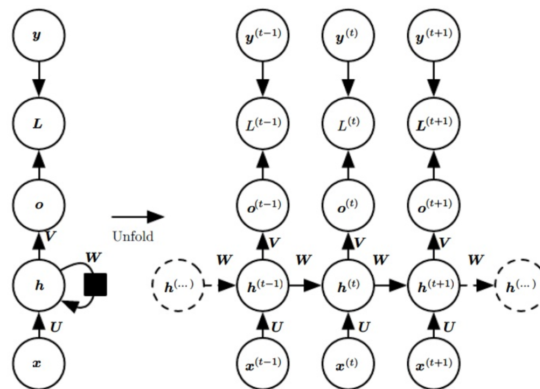


Figure 3.8: seq-to-seq RNN; Goodfellow et al. [13, p.378]

Figure 3.8 depicts both the compact and unfolded view of an RNN which follows this seq-to-seq paradigm. Note that similar to Figure 3.6 each node represents an entire recurrent layer and edges correspond to densely connected layers. Where a respective letter U, V, W is annotated, edge weights (being constant through time) are applied. Looking on the left hand side of the figure, x denotes the input data point, h denotes the hidden state and o denotes a generally defined output neuron mapping into a target space. The encircled L and y merely denote the act of comparing a target y via a loss function L in order to perform BPTT. This generally defined framework allows for using simple input neurons as well as for using LSTM cells. This design pattern may for instance be applied for modeling relations between time series. Furthermore, the output generated by o can be propagated to other layers which are then called time distributed layers since at each time step they receive an input proceeding in an online fashion. In particular this design pattern allows for stacking several recurrent layers.

The second design pattern is based on the idea of ***only propagating the final hidden state***. It thus provides a summary of a time series after the full input data has been processed by the recurrent layer. This design pattern can again be followed up by a neural network which this time does not receive data in an online fashion. By factoring out the time component this paradigm belongs to the collection of ***sequence-to-one models*** (seq-to-1).

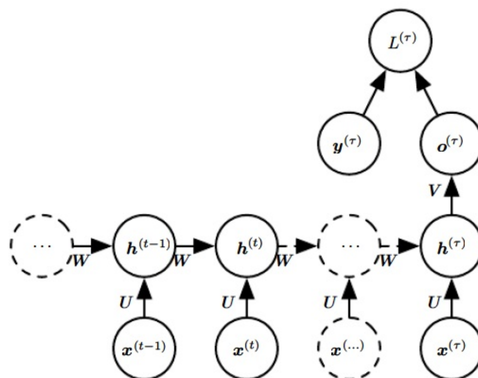


Figure 3.9: seq-to-1 RNN; Goodfellow et al. [13, p.382]

Figure 3.9 uses notation analogous to Figure 3.8. It only represents the unfolded view of an RNN since the more compact representation can easily be confused with seq-2-seq unless suitable notations are introduced. By being able to generate a single output vector (potentially only a single number) as opposed to a time-series this design pattern can be used to perform regression on a real vector space (potentially of dimension one) or classification. Note that the idea of simply raveling a time series into a vector and using DNN may seem equivalent but such time-delayed models inherently are unable to properly factor out the time component by implicitly having fixed an assignment of samples at a time $x^{(t)} \in \mathbb{R}^m$ to the same m neurons. Therefore whenever a trend is present in a time series but occurs at a different time, a time-delayed model relies on having seen exactly such a sample before as opposed to RNN recursively modeling the fresh data points and using hidden states as aggregate memory.

3.6 Other modeling techniques

Besides the previously discussed techniques of which we will only use the neural network there are a variety of other techniques which are state of the art methods in machine learning. As it is impossible to adequately represent all techniques in great detail we roughly explain other prominent models and kindly refer to appropriate literature.

Support vector machines (SVM)

One prominent method which can be used for both classification and regression is the support vector machine (SVM). We refer to Bishop's book where this method is presented and discussed [6]. The underlying idea of an SVM is to transform the data into a higher dimensional space where a linear model is able to perform the task. In this regard SVM extensively ***utilizes the feature space mapping*** as defined in Section 3.1. In a classification setting the aim is to map the data into high dimensions such that the classes become linearly separable. In this high dimensional space, we fit a separating hyperplane typically so that it has maximal distance to the individual classes. The regression task is a bit more finicky and is built off the classification idea. We surround the graph, being a surface by what is called an ϵ-tube by using an error function which is 0 around a small ϵ-region of the graph. Such a function is called an ϵ-intensive function. In this way, points near the graph do not introduce an error and are regarded as well modeled. Now the graph is again transformed into a higher dimensional space where this ϵ-tube contains a linear hyperplane which can be detected by SVM. A notable fact about SVM is that they do not provide a discriminative classification model but are decision machines only suggesting a label directly. Another aspect worth pointing out is the fact that when using suitable transformations, SVM are able to exploit the so-called ***kernel trick***. The kernel trick basically enables us to avoid costly computations which would emerge by transforming the data into a higher dimensional space.

Decision trees

Another more intuitive method is realized by decision trees. For full detail about this technique we refer to Alpaydin and again only point out the key concepts [1]. A decision tree can be represented as a rooted tree in the sense of graph theory. At each node a certain decision rule is implemented. A data-point is then propagated from the root through the tree where the decision rule at each node determines the descendant being visited next. At each leaf, an output result representing a class label or a numerical value for regression is stored. The training of a decision tree involves introducing and pruning nodes and adapting the decision rules such that a ***training point travels from the root to a leaf*** which holds its respective label. Decision trees have the advantage of being easily interpretable since they can be converted in a construct of if-else decisions. Decision trees, however, are prone to overfitting. Nonetheless they find many applications and in particular serve as a basis for ensemble methods (i.e. a composition of several ML-models into a single one) like random forests.

Pointer networks

Finally, we point out a technique which specifically addressed the modeling of combinatorial problems, namely pointer networks. We refer to Vinyals, Fortunato and Jaitly who first presented this technique [33]. Pointer networks ***address the limitation of having a fixed output size when using classical sequence-to-sequence models***

such as RNN. The way this is circumvented by pointer networks is by embedding an RNN into a specific architecture. At each time step, the vector formed by all hidden states of a same sized auxiliary network is transformed into a probability vector by using softmax. The resulting so-called attention vector is therefore of the same length τ as the input and the maximal element is interpreted as a recommended item $t \in [\tau]$. In this way, at each time step the maximal probability points at the suggested item which is again fed into RNN as hidden states. After feeding through the entire time series, we have τ such recommendations where some items may have been recommended for several times. In this way, the output vector has a flexible length of at most τ.

3.7 Assessing the quality of models

After preprocessing the data and fitting a model we require measurements and techniques to evaluate the predictive power of the model. We limit to regression- and two-class-classification models since later we will only use these two types of methods. The multi class case can be attained by natural extensions and other machine learning techniques may require a completely different palette of measurements.

Potential sources of error

We recall the machine learning setting where we assume an unknown underlying distribution p_{data} of which we have samples training data points. This provides us an empirical (approximate) distribution $\widehat{p}_{data}$ which is used for training by minimizing the empirical risk.

In order to capture potential pitfall it is good practice to reserve a fraction of the training data for assessing the model. This test set is not used for training. Having fitted the model only on the training set we evaluate a performance measure on both the test set and the training set. In this way, it is assumed for now the test set well represents p_{data} and the training set well represents $\widehat{p}_{data}$ (i.e. having enough data at hand).

First off, when evaluating on the training set we find how well the model is fit to $\widehat{p}_{data}$. If this error is large we perhaps used a too simple model, called ***underfitting***, or did not carefully choose the ***hyperparameters*** for training. If the error is small we proceed by evaluating the test set. If it is large we check a number of implications. Generally this case indicates a lack of generalization abilities. Then one prominent reason for observing large errors on the test set but (very) small errors on the training set is ***overfitting***. This happens if the model assumes a way too large complexity in order to achieve extremely small errors. One way to address this is by using regularization. A regularization term basically integrates a measure of simplicity of the model into the loss and in this way penalizes overly complex assumptions in trade for small improvements of error by training. Another counteraction is to end training earlier or perhaps to try a more robust or simple modeling technique after all.

In case p_{data} is not well represented we should aim for more training data as it is impossible to generalize to p_{data} based on ***too little information***. In case only $\widehat{p}_{data}$ is

not represented well. The error occurs by insignificant test sample sizes. This can be countered by using estimates like confidence intervals or simply by splitting off a larger fraction of the training data.

A second source of error is introduced by fitting a model on $\widehat{p}_{data}$ with respect to a ***loss function*** L as opposed to a more accurate error measure P in order to achieve better optimization properties. This difference is in particular prevalent in classification since we aim to model probabilities instead of the true class labels. However, in this instance we simply fit the model according to L on $\widehat{p}_{data}$ and evaluate P on the training set being distributed according to $\widehat{p}_{data}$ still. The difference gives us insight into the discrepancies between P and L which typically is not that much of an issue though. After all we are interested in minimizing P and thus on the test set it is the measure of choice as opposed to L. However interpreting P in a classification setup has its own hazards which we will specifically address in Section 3.7.2.

3.7.1 Assessing the fit of regression models

We look at regression models and measure the error between the modeled and the true labels. We assume this content to be commonly known and do not refer to specific literature.

Definition 3.51. ***Measuring error of regression***
Given a data-matrix X and associated a target vector y. Let $f(x;\theta)$ be a (systemic part of a) model for regression on y. We define the following quantities which give insight on the modeling quality.

$$\frac{1}{n}\sum_{i=1}^{n}(f(x_i;\theta)-y_i)^2 \text{ is called the } \textbf{mean-square error (MSE)}$$
$$\frac{1}{n}\sum_{i=1}^{n}|f(x_i;\theta)-y_i| \text{ is called the } \textbf{mean-absolute error (MAE)}$$
$$\frac{1}{n}\sum_{i=1}^{n}(f(x_i;\theta)-y_i) \text{ is called the } \textbf{mean-bias error (MBE)}$$

For the individual measures we identify the respective summands as the per data point error. Thus may also use other aggregates over the data points for instance providing the minimal maximal or median error. For the mean we recall the ***confidence interval*** (for unknown variance) given by the end points $\overline{x} \pm t^* \frac{s}{\sqrt{n}}$ where t^* denotes the quantile being chosen.

3.7.2 Assessing the fit of classification models

In this section we will mainly cover the evaluation of binary classification models. These can be generalized to the multi class case in many instances. Throughout we assume a discriminative model $f(x,\theta)$ modeling the posterior class probabilities. This canonically gives rise to formulating discriminant functions by using a threshold. First, we

will introduce actual performance measures P for the binary classification and refer to Chicco and Jurman [7]. Whenever assessing a performance it is important to look for some baseline comparison which tells what one should at least expect. The expected performance may depend on the particular instance for certain measures whereas others are able to maintain comparability between instances. We will particularly aim for opportunities to improve in this regard to achieve a maximal level of truthfulness.

Evaluation by the discriminant function

In classification it is our primary goal to optimize the actual performance measure P. The performance measure compares the predicted label to the actual label and measures the correct guesses. While this may seem like a straightforward task of counting correct guesses, there are considerations to be made in order to avoid incorrect interpretations. Before evaluating P we shall present the perhaps most natural way to obtain a discriminant function from a discriminative model.

Definition 3.52. ***Discriminant function for binary classification***
Suppose we are given classes $\mathcal{C}_1, \mathcal{C}_2$ and a discriminative model $f(x, \theta) \sim p(\mathcal{C}_1|x)$ providing an estimate posterior for a given data point x. By $T \in [0,1]$ denote a ***threshold***. We define two-class discriminant functions $y_T(x)$ by setting $y_T(x) := 0$ if $p(\mathcal{C}_1|x) \geq T$ and $y_T(x) = 1$ otherwise.

Definition 3.53. ***Performance measures for discriminant functions***
Given $y_T(.)$ a two class discriminant function, a data matrix X and an associated target-vector t. We refer to $t_i = 0$ as negative and to $t_i = 1$ as positive. We present the following well established performance measures for y_T. Note that the following quantities are functions in y_T, X, t. If the arguments are clear from the context we omit them for light notation. We refer to Chicco at al. [7] where an extensive list of the following well known scores can be found.

(1) In the following $I_{\{\}}$ is the indicator function.

$$TP(y_T, X, t) = \sum_{i=1}^{n} I_{\{y_T(x_i)=1\}} I_{\{t_i=1\}} \text{ the number of } \textbf{\textit{true positives,}}$$

$$TN(y_T, X, t) = \sum_{i=1}^{n} I_{\{y_T(x_i)=0\}} I_{\{t_i=0\}} \text{ the number of } \textbf{true negatives,}$$

$$FP(y_T, X, t) = \sum_{i=1}^{n} I_{\{y_T(x_i)=1\}} I_{\{t_i=0\}} \text{ the number of } \textbf{\textit{false positives,}}$$

$$FN(y_T, X, t) = \sum_{i=1}^{n} I_{\{y_T(x_i)=0\}} I_{\{t_i=1\}} \text{ the number of } \textbf{\textit{false negatives;}}$$

We define the number of ***(actual) positives*** $P = TP + FN$ and the number of ***(actual) negatives*** $N = TN + FP$ and summarize these basic measures in what is called a confusion matrix or ***contingency table***.

true class	predicted class		
	positive	negative	total
positive	TP	FN	P
negative	FP	TN	N

Table 3.2: Def, confusion matrix

(2) We define the ***accuracy*** (ACC), the ***positive predictive value*** (PPV), the ***true positive rate*** (TPR), the ***negative predictive value*** (NPV) and the ***true negative rate*** (TNR) as follows.

$$ACC = \frac{TP + TN}{P + N}, \quad PPV = \frac{TP}{TP + FP}, \quad TPR = \frac{TP}{P}$$
$$NPV = \frac{TN}{TN + FN}, \quad TNR = \frac{TN}{N}$$

PPV is also called ***precision***, TPR is also called ***recall*** or ***sensitivity***, and TNR is also called ***specificity***. The precision measures the fraction of true positives among all positive predictions while the recall measures the fraction of true positives which are identified as such. All the above quantities take values on $[0,1]$ where larger is better.

(3) We define the ***F_1-score*** as the harmonic mean of PPV and TPR. We can generalize the F_1-score to the ***F_β-score*** by taking a weighted harmonic mean.

$$F_1 = \frac{2PPV * TPR}{PPV + TPR}, \quad F_\beta = \frac{1}{\frac{1}{1+\beta^2}\frac{1}{PPV} + \frac{\beta^2}{1+\beta^2}\frac{1}{TPR}}$$

These quantities again take values in $[0,1]$ where larger is better.

(4) We define a function which maps the unit interval to points in $[0,1]^2$ by

$$T \to (TPR(y_T, X, t), PPV(y_T, X, t)).$$

The image of this function is called the ***precision-recall curve*** (PR-curve), see Alpaydin [1, p.492]. By AUC-PR we denote the ***area under the precision-recall curve*** assuming values in $[0,1]$. We define the no-skill line by a mapping of T to $(T, \frac{P}{P+N})$ yielding a constant line in $[0,1]^2$ which we will motivate in Lemma 3.56. There is no meaningful theoretical interpretation of the PR-curve and alternatives were suggested by Flach and Kull in [11]. The PR-curve however captures the trade off between precision and recall when changing the threshold. The AUC-PR aims to factor out all possible thresholds and gives insight into the predictive power of

the model. In general, we look for a PR-curve which moves close to the upper-right corner and a larger AUC-PR is better. For a specific choice of a threshold T we obtain a point on the precision recall curve. Typically, we aim to choose T such that we can not increase the precision or recall without decreasing the other (called pareto-optimum, or efficient point).

(5) ***Matthews correlation coefficient*** (by Brian W. Matthews), also called the phi coefficient (by Karl Pearson) or mean square contingency coefficient is defined as follows.

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

Since MCC takes into account all four class sizes in a contingency table it is often considered as the most informative and fair score regarding binary classifications. Chicco et al. [7] advocate for the MCC being preferred over accuracy and F_1-score and we will briefly demonstrate an example where this becomes evident below. Observe that $(TP \cdot TN)$ corresponds to the diagonal entries of the contingency table and $FP \cdot FN$ corresponds to elements which are not on the diagonal. MCC thus considers their difference and the denominator simply is the appropriate weighting factor for limiting the range on $(-1, 1)$. Stronger negative MCC scores indicate the prediction being more of the opposite of the true labels and a more positive MCC indicates stronger correlation. Values close to zero indicate random classification. Note that MCC is not defined if one label is never predicted or if the data is single labeled. In any case, an undefined MCC has a serious implication.

Remark 3.54.

(1) The following reveals the dangers of looking at the accuracy score or even the precision and recall scores. Suppose we are asked to implement a machine learning model which shall detect faulty apples in shipments. We evaluate $PPV = 0.9990$, $TPR = 0.9091$, $ACC = 0.9083$, $F_1 = 0.9519$ which indicates a reasonable predictive power. The corresponding contingency table is $TP = 10000$, $TN = 1$, $FP = 10$, $FN = 1000$. Thus $P = 11000$ and $N = 11$ so there is only a small proportion of faulty apples. Our model is able to detect a 90% of good apples while it was only able to detect 10% of faulty ones meaning it is a very weak model for rejecting faulty apples and accepting good ones as we desire. In this situation Matthews correlation coefficient would indicate the flaws as we obtain $MCC = 0$.

(2) Besides using MCC another countermeasure to this phenomenon is to derive base lines for the F_1-score and the precision which correspond to so-called no-skill classification. To that end we introduce random classifiers and compute the expected scores. We will see how to set up a random classifier in order to obtain the maximal expected scores among all random classifiers for a problem at hand. Whenever a classifier exceeds the best scores attainable by random classification, we can claim the classifier is better than random.

Definition 3.55. ***Random and constant classifiers***

(1) Let $p \in [0,1]$. We define a ***random classifier*** $y^p(x)$ as a random function taking the value 1 with probability p and taking the value 0 with probability $1 - p$.

(2) We define $y \equiv 0$ the always-zero-classifier and $y \equiv 1$ the always-one-classifier. Classifiers which always return the same class label are called ***constant classifiers***.

Lemma 3.56. ***Expected F_1-score and expected precision***
Given a random classifier y^p and data X, t. Denote by $\pi = \frac{P}{P+N}$ the fraction of positives. We refer to Flach and Kull stating $\mathbb{E}(F_1(y^p, X, t)) = \frac{2p\pi}{p+\pi}$ and $\mathbb{E}(PPV(y^p, X, t)) = \pi$, see [11, p.3,4].

Corollary 3.57.
The F_1-score of y^p evaluated for any given data-set with $0 < \pi = \frac{P}{P+N}$ attains a unique maximum at $p = 1$. In particular the F_1 score of the always-one classifier is maximal $\max_p \mathbb{E}(F_1(y^p, X, t)) = \frac{2\pi}{1+\pi} = F_1(1, X, t)$.

Proof.
We compute $\frac{\partial}{\partial p}(\frac{2p\pi}{p+\pi}) = \frac{2p(p+\pi)+2p\pi(1+\pi)}{(p+\pi)^2} > 0$ if $p > 0$. For $p = 0$ the F_1-score is 0 and for $p > 0$ the F_1-score is positive. Hence the maximum is attained at $p = 1$. The maximality of the expected F_1 score for the always-one classifier follows since $p = 0$ on a zero-set. □

Remark 3.58.

(1) As we have proven, the always-one classifier provides the same F_1 score as the best random classifier making it suitable for a baseline comparison. Since random and constant classifiers do not incorporate any information hidden in the data we refer to such classifiers as no-skill classifiers. They provide us no-skill scores which we can use for comparing other classifiers. One way we incorporate such a no-skill score is by drawing a constant line at the no skill precision π reflecting a no-skill precision recall curve. Thus we can read of the quality of a model by looking at whether the precision recall curve is above (better) or below (worse) this no-skill line. The resulting no skill AUCPR clearly is π.

(2) We refer to the previous example and compute an expected F_1-score of around 0.9995 and expected precision of about 0.9990. For our classifier of choice we attained $PPV = 0.9990$ the no-skill precision and $F_1 = 0.9519$ is worse than the no-skill F_1 score. Therefore, if we were to use the always-one classifier we would achieve a better classification which clearly hints our classifier is not satisfactory.

Chapter 4

Learning node selection rules

As proven in Theorem 2.31 BESTFS provides us a minimal branch-and-bound tree among all node selections when using BB to solve KP. Still a node-selection which provides us a ***short exploration phase*** is not yet found. In fact a polynomial bound on the number of nodes in the exploration phase could prove P=NP if a node can be processed and selected in polynomial time as well. We do not hope for such a result and thus try to at least heuristically minimize the exploration phase by using machine-learning. We steer a node-selection such that ideally it moves along a PTO. We can use this to truncate BB at an appropriate point and obtain a heuristic solution. In order to train a model which aids a node-selection rule we require a training set and maybe a test set to evaluate the model. To that end we introduce random instances which serve this purpose. In machine learning it is good practice to scale the inputs leading to the following assumption.

Permanent Assumption 3.
From this point onward for any given instance of KP we assume the benefits to be scaled by the maximum $b_i = \frac{b_i}{\max_{1 \le i \le n} b_i}$. These quantities are still referred to as benefits. Clearly in general they no longer coincide with the profit over weight ratios.

Definition 4.1. ***Random instance***
We generate random instances of KP of size $n \in \mathbb{N}$ with pairwise distinct profits, weights and benefits. Furthermore, we apply scaling of items such that $W = 1$ and $\max_{1 \le i \le n} c_i = 1$. We also scale the benefits by the maximum and obtain values in $(0, 1]$. Random instances thus satisfy all permanent assumptions 1, 2, 3 imposed on KP.

Algorithm 8: Generating a random instance of KP

Data: instance size $n \in \mathbb{N}$

Result: random instance KP

1 Initialize $c_i = w_i = 1, b_i = \frac{c_i}{w_i}\ 1 \leq i \leq n$;

2 **while** *any* $c_i = c_j$ *or any* $w_i = w_j$ *or any* $b_i = b_j$ **do**

3 Sample $c_i, w_i \sim U(0,1),\ 1 \leq i \leq n$ until all $c_i, w_i > 0$;

4 Sample capacity $W \sim U(\max_{1\leq i\leq n} w_i, \sum_{i=1}^{n} w_i)$;

5 Compute benefits $b_i = \frac{c_i}{w_i},\ 1 \leq i \leq n$

6 Apply scaling $c_i := \frac{c_i}{\max_{1\leq i\leq n} c_i},\ w_i := \frac{w_i}{W},\ b_i := \frac{b_i}{\max_{1\leq i\leq n} b_i},\ 1 \leq i \leq n$;

7 return $c_i, w_i, b_i\ 1 \leq i \leq n$ labelled according to descending order of benefits

This algorithm provides an instance of KP almost surely after one iteration.

Definition 4.2. ***Tension of an item***
We define for each item i of a random instance of KP notions being called tensions which reflect the existence of similar items. We evaluate the so-called tension with regard to both profit and weight of items.

$$\tilde{f}_w(i) = -\sum_{\substack{j=1 \\ i\neq j}}^{n} \log(|w_i - w_j|) \text{ the unnormalized profit tension of item } i,$$

$$\tilde{f}_c(i) = -\sum_{\substack{j=1 \\ i\neq j}}^{n} \log(|c_i - c_j|) \text{ the unnormalized weight tension of item } i.$$

We obtain the normalized versions of these quantities by dividing the maximum and refer to these as the ***weight tension*** respectively ***profit tension*** of item i.

$$f_w(i) = \frac{\tilde{f}_w(i)}{\max_j \tilde{f}_w(j)},\ f_c(i) = \frac{\tilde{f}_c(i)}{\max_j \tilde{f}_c(j)}.$$

Note that the respective tension is finite and positive for random instances KP. For random instances we can show easily that the tensions are also finite and non-zero in expectation.

4.1 Training and test data

Data generation

We compute the minimal branch-and-bound tree for 1000 instance for each size $n \in \{25, 50, 75, 100\}$ providing us a ***training set*** of 4000 random instances. For generating a ***test set*** we proceed analogously where only 250 random instances for each size $n \in \{25, 50, 75, 100\}$ are considered. The minimal branch-and-bound tree is obtained by using BB with BESTFS.

Slicing and dicing the training/test set

At each node in the tree being an instance $\text{KP}_{I,\theta}$ we consider the quantities $(f_c(k), f_w(k), c_k \cdot x_k, w_k \cdot x_k, w_{I,\theta}, c_{I,\theta}, c^{greedy}, w^{greedy}, d_i, \theta_k)$. From left to right the corresponding ***features*** are called the profit tension (of k), weight tension (of k), profit contribution (of k), weight-contribution (of k), fixed weight, fixed profit, greedy profit, depth (of the node) and fixed variable value (of the node). k denotes the critical item, x_k denotes the critical variable value as defined Lemma 2.5 and c_k and w_k respectively are the profit respectively weight of the critical item. By respectively multiplying with x_k we obtain the amount of profit/weight to be added to $c_{\lesssim}$ to obtain an lp-optimal (however potentially fractional) solution. The tensions $f_c(k), f_w(k)$ of the critical item k are defined as above in Definition 4.2. The ***label vectors*** for both training and test set are composed of ones and zeros respectively indicating whether a node was in the PTO obtained when solving KP with BESTFS. In fact, for training we ***only consider branched nodes or nodes pruned by bound and exclude the root*** since only then all quantities are defined. The prediction of the root clearly can be set to be 1 and we explain later how we predict infeasible and integral nodes.

4.2 Learning PTO

We set up a DNN which we refer to as DNNC-pto having the following architecture.

model	hidden layer size	output layer size	types of layers
DNNC-pto	6	2	*relu*-softmax

Table 4.1: DNNC-pto; architecture

In this way DNNC-pto is used as a discriminative model to predict the conditional probability of being in PTO based on a given node.

We fit the model as specified in Chapter 6. We evaluate DNNC-pto on the fixed test set consisting of all nodes of 250 instances as described above. We evaluate scores both over all test nodes and on slices generated according to depth. The threshold being used to obtain the considered discriminant function according to Definition 3.52 is $T := 0.5$.

depth	mcc	recall	precision	f-1	no-skill precision	no-skill f-1-score	samples
all	0.68	0.67	0.75	0.71	0.11	0.20	160079
0 to 5	0.26	0.15	0.59	0.24	0.10	0.19	28781
5 to 10	0.50	0.39	0.69	0.50	0.06	0.12	53229
10 to 15	0.65	0.63	0.73	0.68	0.08	0.15	35038
15 to 20	0.72	0.78	0.73	0.75	0.11	0.19	22183
20 to 25	0.76	0.88	0.71	0.79	0.13	0.23	15482
25 to 30	0.80	0.94	0.73	0.82	0.15	0.27	10979
30 to 35	0.82	0.97	0.76	0.85	0.18	0.30	7896
35 to 40	0.83	0.98	0.77	0.86	0.20	0.33	5629
40 to 45	0.87	0.98	0.83	0.90	0.23	0.37	3901
45 to 50	0.90	0.98	0.88	0.93	0.24	0.39	2917
50 to 55	0.93	0.98	0.91	0.94	0.24	0.39	2133
55 to 60	0.94	0.98	0.93	0.96	0.24	0.39	1505
60 to 65	0.95	0.98	0.96	0.97	0.26	0.41	949
65 to 70	0.99	0.99	0.99	0.99	0.27	0.43	590
70 to 75	1.00	1.00	1.00	1.00	0.31	0.48	337
75 to 80	1.00	1.00	1.00	1.00	0.32	0.48	176
80 to 85	1.00	1.00	1.00	1.00	0.24	0.38	68
85 to 90	1.00	1.00	1.00	1.00	0.22	0.36	23
90 to 95	0.00	0.00	0.00	0.00	0.00	0.00	13
95 to 100	0.00	0.00	0.00	0.00	0.00	0.00	4

Table 4.2: PTO-prediction, evaluation on test set

The first row evaluates over ***all nodes in the test sets***. The number of nodes respectively is represented in the sample column. Since the test set is highly imbalanced, we first and foremost look at MCC indicating a strong correlation between predicted and actual labels. Recall that the expected precision is exactly the fraction of positives as proven in Lemma 3.56. We observe that the achieved precision and recall both lie above the expected scores by a large margin as well.

The subsequent part of the table considers a ***partition of the test set according to depths***. The MCC throughout the depths suggests that predictions in large depth become increasingly easy and eventually even become perfect. The fraction of positives grows closer to 0.5 while the number of nodes shrinks significantly in larger depths. We note that the maximal depth of a node in an instance with n items is n. In our test set we have 250 instances of each size $\{25, 50, 75, 100\}$. Therefore only 25% of instances can even produce a node at a depth between 76 and 100, 50 % of instances can produce a node at a depth between 51 and 100, and so on. This in part explains the sparsity at increasing depths. On the other hand, predictions close to the root (say depth 0-5) are quite weak which we should bear in mind when constructing node selection rules using DNNC-pto.

Now let us evaluate the classification independent of the chosen threshold $T = 0.5$ used to obtain the above discriminant function. To that end we look at the corresponding ***precision-recall curve***.

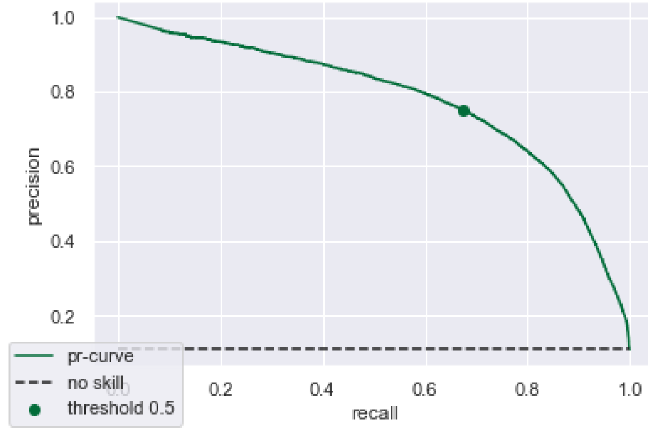


Figure 4.1: DNNC-pto; PR-curve on test set

We observe in Figure 4.1 that regardless of the threshold, we can achieve reasonable predictions and stay well above the no skill line. Note that we will only use DNNC-pto as a discriminative model providing us probabilities interpreted as scores. The probabilities let us order active nodes according to their certainty of being in PTO and thus naturally define a node-selection rule by choosing nodes with largest probability.

4.3 Using PTO prediction within BB

Now let us employ DNNC-pto in a node-selection rule for BB. As hinted above only the modeled posterior probabilities of a node $\mathrm{KP}_{I,\theta}$ being in PTO given the features observed in the node $\mathrm{KP}_{I,\theta}$ will be used in our node selection rule. The model DNNC-pto models this conditional probability which we denote by DNNC-pto$(\mathrm{KP}_{I,\theta}) \sim p(\mathrm{KP}_{I,\theta} \in \mathrm{PTO} \mid \mathrm{KP}_{I,\theta})$. We will then again solve another set of randomly generated instances and look at suitable measures which tell us how DNNC-pto performed in this context.

Definition 4.3. ***DFS-based ML-aided node-selection rules***
Given an arbitrary branch-and-bound tree T for an arbitrary instance of KP.

(1) We define the ***DFS-in-pto*** node-selection rule by assuming an arbitrary current tree and an arbitrary non-empty set of active nodes as follows. We push newly generated nodes on the stack of active nodes such that the node fixing the prior critical variable to 1 is on top. For the respective node we compute DNNC-pto at insertion if none of its input arguments as defined in Section 4.1 is null. Otherwise we conventionally set the modeled probability to 1. Among the two most recent nodes we select the one having a larger score being the predicted posterior probability. Thus we traverse through T in a DFS fashion where the decision of going left or right is made by the model. In case there is only one active node left we perform the unique choice. A tie-break is performed by preferring the later node in the stack (inclusion) similar to Definition 2.18.

(2) At insertion we evaluate for all siblings $KP_{I,\theta}, KP'_{I,\theta}$ their ***sibling similarity*** defined by $s(KP_{I,\theta}, KP'_{I,\theta}) := 1 - |\text{DNNC-pto}(KP_{I,\theta}) - \text{DNNC-pto}(KP'_{I,\theta})|$. We again use the convention of setting DNNC-pto$(KP_{I,\theta})$ to 1 if some input argument is null. In this way a pair of leaves in T provides the highest similarity of 1. Another way to achieve a large sibling similarity is having similar estimated probabilities.

(3) We add a rule to DFS-in-pto in order to obtain the ***DFS-in-pto with restart*** node-selection rule (shorthand DFS-in-pto-w-r). Whenever we run into a leaf of a current tree T we select an active node $KP_{I,\theta}$ among the set of active nodes which maximizes the product $s(KP_{I,\theta}, KP'_{I,\theta}) \cdot \text{DNNC-pto}(KP_{I,\theta})$. This product associated to $\text{KP}_{I,\theta}$ is large whenever it has a high probability of being in PTO and has a similar sibling. Thus, whenever a DFS-run terminates by running into a leaf, we jump to an ambiguous but worthy spot instead of exploring the closest nodes (diversification).

In the following let us evaluate the two newly defined node-selection rules from Definition 4.3 and compare them to their deterministic counterparts. We generate 250 random instances for each size $n \in \{25, 50, 75, 100\}$. Each instance is solved with all considered node-selection rules providing a fair comparison on the same set of 1000 test instances. We look at the sizes of branch-and-bound trees (left hand side) achieved by some prominent node-selection rules and in particular we are interested in the length of the exploration phases (right hand side) as defined in Definition 2.35.

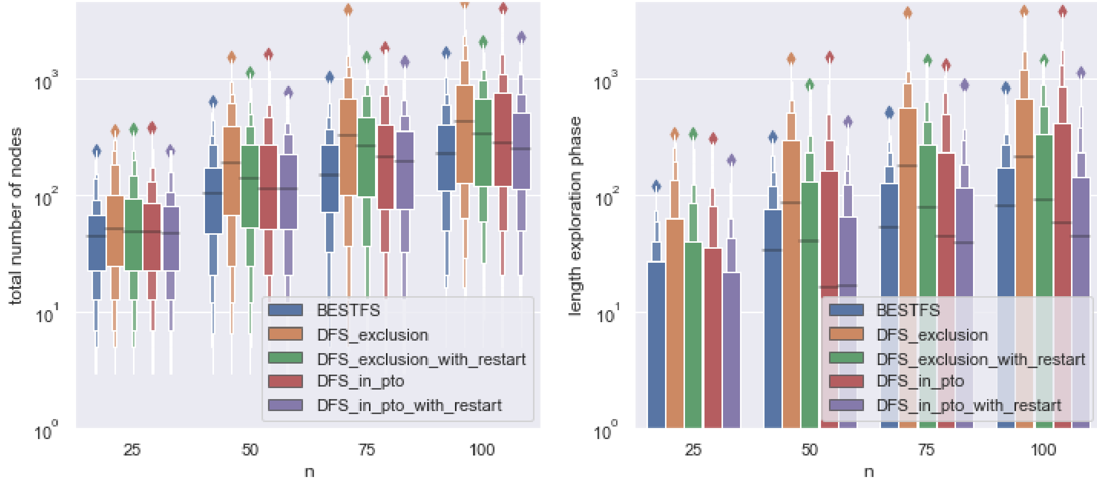


Figure 4.2: DNNC-pto; performance in BB

Note that both boxplots are on a logarithmic scale and only show the y-axis from 1 onward. An empty branch-and-bound tree can not be encountered. Moreover an empty exploration phase implies greedy optimality. In this case the duration (even of the individual phases) is independent of the node-selection rule being used, see Theorem 2.36.

At first let us consider the left hand side of Figure 4.2 reflecting the ***total number of nodes*** processed during the respective runs of BB. As proven in Theorem 2.31 BESTFS always achieves the smallest branch-and-bound trees. DFS-exclusion has the worst overall performance and even a random restart seems to help improve the performance. By employing DNNC-pto the performance is improved significantly, and when using the restart at ambiguous spots, we almost achieve minimal branch-and-bound trees.

Now let us focus on the ***exploration phase*** analysis looking at the right hand side of Figure 4.2. Note that in case that the greedy-solution is optimal, the exploration phase has length 0. Since we evaluate all node-selection rules on the same set of instances, the exploration phase analysis shows the behavior of node-selection rules only on ***non-greedy optimal instances*** in our test-set. For now let us focus only on this case. By Lemma 2.29 BESTFS will explore the entirety of $T(c^*)$ which is the maximal exploration phase for non-greedy optimal instances among all optimal node selection rules, see Theorem 2.36. By this, any node-selection rule which explores more vertices in the exploration phase than BESTFS is not optimal and we can see in Figure 4.2 that this holds true for DFS-exclusion (with restart). They behave non-optimally on many instances. Conversely the ML-aided node selection rules DFS-in-pto (with restart) admit a shorter exploration phase than BESTFS in median. If we briefly look at the overall performance (left hand side) we can deduce that still, DFS-in-pto (with restart) does not behave optimally on all instances since they perform (slightly) worse than BESTFS in general. Thus there are some nodes outside of $T(c^*)$ being explored by DFS-in-pto (with restart) which is penalized by up to two additional nodes in the validation phase, see Theorem 2.36. In this regard the ML-aided node-selection rules find an optimal solution rather quickly on the typical instance but make mistakes (i.e. exploring outside of $T(c^*)$) which increase the number of nodes in the validation phase.

Now let us consider the behavior of the node-selection rules on the ***greedy-optimal instances***. For the sake of argument suppose we use the all-zero solution as initial solution which fully exposes the capabilities of node-selection rules. Still the search behaves the same but we get rid of the concealing case of having optimal roots. The deterministic DFS-exclusion (with restart) finds the optimal greedy-solution on the first run of DFS, see Lemma 2.16. Since by Permanent Assumption 2 the greedy solution is unique, DFS-exclusion (with restart) provide us a minimal exploration phase in this case. Since the PTO coincides with $\overline{T_E}$ and is contained in T_U, Theorem 2.36 implies that DFS-exclusion (with restart) is optimal on greedy-optimal instances. Considering the ML-aided node-selection rules we of course can not prove something similar as we make heuristic choices. However if we recall that we only train DNN-in-pto on PTO being obtained by BB (using the greedy as initial), we never learned the trajectory to a greedy-solution and therefore can not expect good performance in the exploration phase in this case.

In the following table we present the corresponding numerical values concerning Figure 4.2.

n	node-selection	total number of nodes mean	std	median	length exploration phase mean	std	median	samples
25	BESTFS	52.8±5.4	43.9	44	16.0±3.0	24.3	0.0	250
	DFS-exclusion	79.6±9.9	80.0	51	45.1±9.2	74.0	0.0	250
	DFS-exclusion-w-r	70.4±8.2	66.5	49	31.2±6.9	55.4	0.0	250
	DFS-in-pto	64.0±7.3	58.5	49	26.8±6.0	48.0	0.0	250
	DFS-in-pto-w-r	59.3±6.3	50.6	47	16.8±3.7	30.2	0.0	250
50	BESTFS	132.1±14.7	118.3	102	51.1±7.9	63.6	33.5	250
	DFS-exclusion	266.1±34.0	274.1	187	190.6±31.7	256.1	87.0	250
	DFS-exclusion-w-r	193.7±23.5	189.4	141	96.2±18.4	148.8	41.0	250
	DFS-in-pto	204.6±29.6	238.8	114	115.9±26.6	214.3	16.5	250
	DFS-in-pto-w-r	158.4±18.0	145.4	112	46.4±8.9	71.8	17.0	250
75	BESTFS	194.8±20.5	165.5	147	77.1±10.9	87.9	53.5	250
	DFS-exclusion	470.1±63.9	515.5	330	358.9±61.6	497.3	176.5	250
	DFS-exclusion-w-r	326.6±37.2	300.4	264	183.2±30.8	248.4	79.0	250
	DFS-in-pto	308.1±39.5	318.8	215	181.0±34.4	277.9	44.5	250
	DFS-in-pto-w-r	248.0±28.0	225.7	195	79.8±14.2	114.8	39.0	250
100	BESTFS	300.5±34.2	275.5	226	117.5±18.2	147.1	80.5	250
	DFS-exclusion	648.6±88.7	715.2	434	470.0±80.9	652.9	212.0	250
	DFS-exclusion-w-r	448.9±51.0	411.3	339	227.8±40.0	322.7	92.5	250
	DFS-in-pto	520.4±75.3	607.7	283	324.6±69.7	561.9	59.0	250
	DFS-in-pto-w-r	371.9±45.6	368.2	246	105.6±21.3	171.5	44.0	250

Table 4.3: PTO-prediction, performance in BB

By looking at the table it becomes more evident that for fixed n the difference of means and medians for different node selection rules is quite large, in particular regarding the exploration phase. This indicates the existence of far outliers, meaning some instances are particularly hard to solve, given a particular node-selection rule is being used. This observation is particularly pronounced for DFS-in-pto which quickly finds an optimal node in many instances. However, for some particular instances the search with DFS-in-pto was widely unsuccessful. For DFS-in-pto with restart, we observe a much more desirable behavior. The restart rule largely helps the search getting back on track in case we get lost in these few particularly hard instances.

Chapter 5

ML-aided heuristics for KP

A first entry point for machine learning for setting up heuristics for KP is ***truncating a branch-and-bound algorithm*** at a suitable point in time. A second possibility is directly applying ML to an instance instead of using it to guide a predefined method. Martini showed multiple such direct applications of neural networks to the knapsack problem in [29]. We will address the ***direct prediction of optimal profit*** and investigate the influence of additionally providing the greedy and lp-bound to the model. A more indirect approach for accurately predicting the optimal profit is ***directly predicting an optimal solution***. This approach has the advantage that the predicted profit is associated with a (potentially infeasible) solution which could be further processed by using a heuristic such as neighborhood search. In the following section we will present the models being used and then evaluate model specific properties. For testing we use the same test set throughout the section. Only at the end we will launch all models on this test set in order to guarantee a fair comparison of estimated profits.

Permanent Assumption 4.
From this point onward we limit ourselves to instances of KP of size $n = 50$.

Definition 5.1. ***greedy-lp-mean estimator***
Given an instance of KP. Let c^{lp}, c^{greedy} be the respective lp- and greedy-profit. Then we define the greedy-lp-mean-estimator as the mean $c_{int-lp} := \frac{c^{lp}+c^{greedy}}{2}$.

Lemma 5.2. ***greedy-lp-mean estimator approximates the optimal value***
For the optimal profit c^* of an instance of KP we have

$$|c^* - c_{int-lp}| \leq 1.$$

Proof.
By the Permanent Assumption 1 all item profits are in $(0, 1]$. If KP is lp-optimal it is also greedy-optimal by Lemma 2.5 and the greedy-lp-mean-estimator is c^*. Otherwise we consider the feasible packing $x_{\leq}$ packing items $[k-1]$. Then the associated profits satisfy $c_{\leq} \leq c^{greedy} \leq c^* \leq c^{lp}$. Since x^{lp} packs $x_{\leq}$ and a fraction $[0, 1)$ of k, their profits

differ by at most one. In particular the profit difference of $c_{int-lp} \in (c^{greedy}, c^{lp})$ and c^* is at most one. □

Remark 5.3.
This justifies using c_{int-lp} as an estimator for c^* having an absolute error of at most 1. All subsequently presented models aim to predict an optimal objective function value which is closer to the optimal objective function value than c_{int-lp}.

5.1 Training data and test data

In order to obtain a training and test set for our model we generate 10.000 random instances of size $n = 50$ which are solved with BB using BESTFS. We split off 1.000 instances for testing and the other 9.000 instances are used for training the models. Whenever we refer to ***full input data*** we use the item profits c_i, the item weights w_i, the greedy-bound c^{greedy} and the lp-bound c^{lp} as inputs for a model. The ***reduced input data*** uses only the item's profits and weights. For implementation details like the format of the data we refer to Chapter 6.

5.2 Learning optimal profit

In this section we aim to directly predict the optimal profit. We use DNN and RNN similar to Martini in [29]. We use slightly different networks though. This is mainly due to the smaller size of the instances and the limited computational resources. In addition to Martini's work we will also use the greedy and lp-bounds as inputs and investigate the impact on the quality of predictions. In Table 5.1 we define architectures of RNN and DNN which are respectively trained on the full and reduced datasets. The models are called ***DDN-full, DNN-reduced, RNN-full, RNN-reduced*** accordingly. For details and formats of the inputs we refer to Chapter 6.

model	hidden layer sizes	output layer size	types of layers
DNN-full	100-100	1	σ-σ-lin
DNN-reduced	100-100	1	σ-σ-lin
RNN-full	64-32-100-100	1	LSTM-LSTM-lin-lin-lin
RNN-reduced	64-32-100-100	1	LSTM-LSTM-lin-lin-lin

Table 5.1: profit prediction model architectures

5.3 Learning solutions

Let us proceed by defining models which predict solutions for KP. We do so by training a classifier which models the posterior probability of items for being packed in an optimal solution. We assess the predictive power in this section and we postpone the evaluation of the quality of the associated profit predictions later. We introduce one additional score for comparing vectors as follows.

Definition 5.4. ***bit-error***
Given two data-points $s, y \in \mathbb{R}^n$. The bit-error is defined as $err_{bit}(s,y) = \sum_{i=1}^{m} I_{\{s_i \neq y_i\}}$ where $I_{\{\}}$ is the indicator function. In case s, y are binary vector the bit-error is equal to the Manhattan distance $err_{bit}(s,y) = ||s-y||_1$. The bit error allows us to quantify the ***dissimilarity of two vectors***. In order to evaluate sets of vectors we may aggregate the per vector bit-error for instance by using mean, median, and standard deviation.

Now let us define the architecture of the discriminative classification model ***RNNC-reduced*** which is based on an RNN. The model follows the seq-to-seq paradigm (see Section 3.5.4) and the items are fed through the network as a time-series one by one in an online fashion. The classification is then performed by evaluating the model and applying a threshold of $T := 0.5$ as defined in Section 3.7.2.

model	hidden layer sizes	output layer size	types of layers
RNNC-reduced	64-32-100-100	2	LSTM-LSTM-lin-lin-softmax

Table 5.2: RNNC-red; architecture

Having trained the model on the reduced dataset as described in Chapter 6 we perform all subsequent evaluations on the same test set defined in Section 5.1. Let us ***evaluate the bit-errors*** achieved by RNNC-reduced and compare to the greedy and lp-solutions.

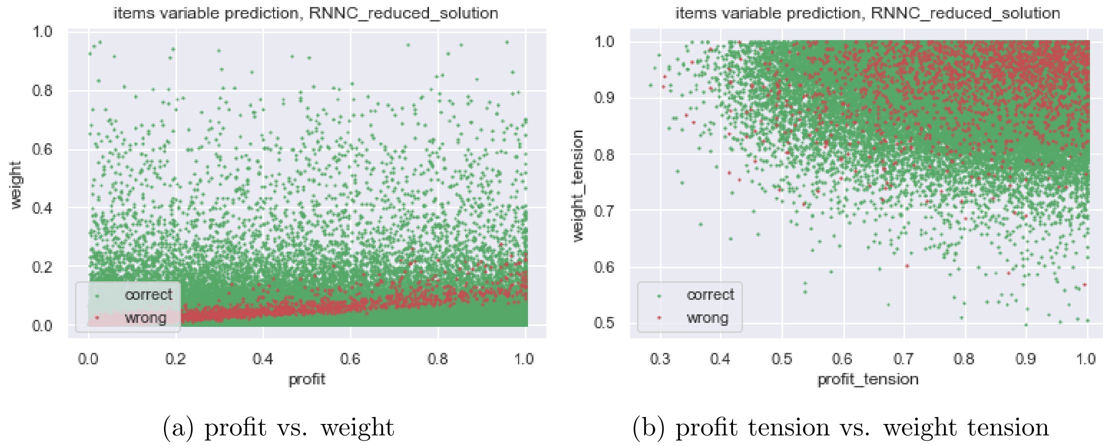
bit-error	0	1	2	3	4	5	6	8	mean	std	median	samples
greedy	420	0	173	227	127	41	11	1	1.81±0.107	1.72	2	1000
lp	0	234	370	284	97	15	0	0	2.29±0.061	0.98	2	1000
RNNC-red	68	305	373	189	59	6	0	0	1.88±0.063	1.02	2	1000

Table 5.3: RNNC-red; bit error

The values in columns of Table 5.3 which are labeled by numbers represent the number of solutions which have the respective bit-error when being compared to an optimal solution. In other words, this is a histogram of bit-errors. The mean, standard deviation, and median in each row aggregate the bit-errors over all solutions obtained by the respective method labeling the row.

First off, note that the evaluation of vector predictions adds a further layer of complication in comparison to merely evaluating label predictions. The solutions define ***bundles*** of $n = 50$ items which are evaluated above. We observe that a large proportion of instances is greedy optimal and unfortunately RNNC-reduced does not manage to predict many solutions (specific bundles of $n = 50$ items) perfectly. However the bit-errors on non-perfect predictions are comparably small. This motivated launching a neighborhood search on the set of predicted binary vectors which tries to push solutions towards feasibility and optimality.

Before doing so, let us put all ***items in a bag*** (i.e. multi-set) meaning we no longer consider their belonging to instances. In this sense, in the following we move on from an evaluation on the instance level to an evaluation on the item level by factoring out the $n = 50$ fold bundling.

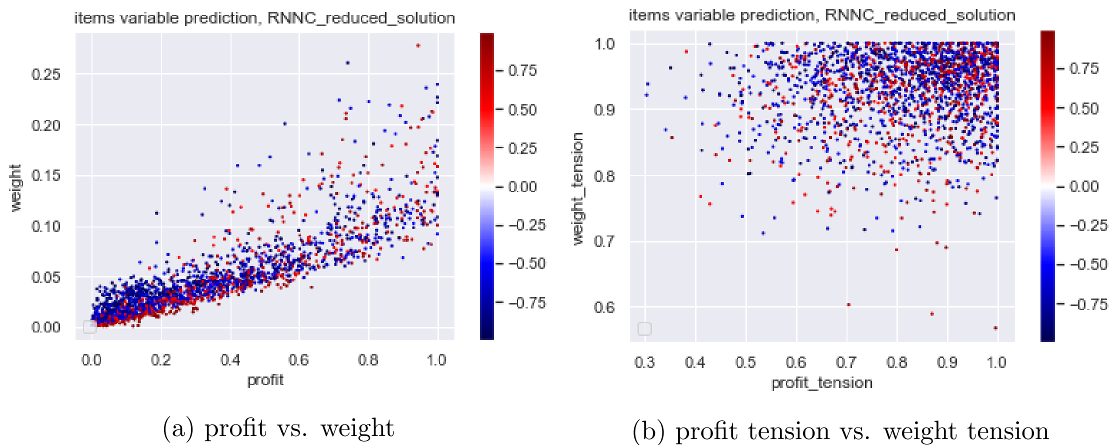


(a) profit vs. weight

(b) profit tension vs. weight tension

Figure 5.1: RNNC-red; item predictions

We plot items as points in a two-dimensional space. The coordinates are defined by the ***profit and weight*** (Figure 5.1 (a)) and respectively the corresponding ***profit tensions and weight tensions*** (Figure 5.1 (b)). A wrong prediction is indicated by a red dot while a correct prediction is indicated by a green dot. In Figure 5.1 (a) we clearly spot that wrongly predicted items satisfy a linear correlation of weight and profit. These items thus have a similar problematic benefit. Focusing on Figure 5.1 (b) there also is a tendency of wrongly predicting items if both tensions are large. Hence the difficulty of predicting a packing for RNNC-reduced lies in distinguishing similar items which are near a certain benefit.



(a) profit vs. weight

(b) profit tension vs. weight tension

Figure 5.2: RNNC-red; wrong item predictions

In Figure 5.2 we look more closely into the class of ***wrongly predicted items***. We again plot the items in the respective two dimensional spaces. This time we only plot items with a wrong prediction. If the bias error is negative we use a blue dot, otherwise we use a red dot. To be clear the bias error is given by $DNNC_{red}(i) - x_i^*$ where $DNNC_{red}(i) \in [0,1]$ is the predicted probability of being in a solution and $x_i^* \in \{0,1\}$ indicates if i actually was in an optimal solution (of some test instance we no longer factor in). By plotting only wrong predictions the bias error is in $[-1,-0.5] \cup [0.5,1]$. Negative values correspond to false negatives and positive values correspond to false positives.

In Figure 5.2 (a) it is visible that items with small profits lightly tend to be false positive if the weight is smaller and false negative if the weight gets larger. Items where both profit and weight are rather large do not show a clear tendency. In Figure 5.2 (b) we observe that by looking only at the respective tension of wrongly predicted items it is seemingly impossible to identify a pattern for false positives and false negatives. Considering all item predictions we achieve the following ***prediction scores on the item level***.

mcc	recall	precision	f-1-score	no-skill precision	no-skill f-1-score	samples
0.918923	0.960778	0.98023	0.970406	0.643	0.782715	50000

Table 5.4: RNNC-red; classification report

In addition to the the no-skill precision, also the following confusion matrix shows that the classes formed by item labels are imbalanced. We suggest looking at MCC which is well suited for assessing classifications of imbalanced test sets. We confirm a very strong correlation between predictions and the truth. In the ***confusion matrix*** below we observe that wrongly classifying positive items seems as likely as wrongly classifying negative ones.

true class	predicted class		
	positive	negative	total
positive	30889	1261	32150
negative	623	17227	17850

Table 5.5: RNNC-red; confusion matrix

Concluding, predicting whether an individual item can be found in an optimal solution or not can be done very reliably via RNNC-red. By using RNNC-red we could therefore preprocess instances by fixing a subset of items by thresholding according to the predicted probability. One perhaps would naturally fix items with either very large or very small probabilities (i.e. rather certain predictions) to the predicted label. Then an exact method like BB could be applied on the remaining unfixed part, again being an instance of KP but with fewer items. This could reduce the running time and perhaps yield (close to) optimal solutions depending on how conservative the threshold probability is chosen.

5.3.1 Local search heuristic

In order to leverage the relatively small average bit-error of the vector prediction coming from RNNC-reduced we employ a simple local search heuristic. From a neighborhood consisting of all feasible solutions which differ by at most ρ bits we choose one which maximizes the increase in profit. If no feasible solution is present in the neighborhood, we return the input. We perform r rounds and thus change up to $\rho \cdot r$ bits.

Definition 5.5. $\rho - r$ ***local search heuristic***
Given an instance of KP and a potentially infeasible integral solution $s \in \{0,1\}^n$. Let $\rho \in \mathbb{N}$ denote the radius of the neighborhood $\tilde{B}_\rho(s) := \{y \text{ feasible} \mid err_{bit}(s,y) \leq \rho\}$ and $r \in \mathbb{N}$ denote the number of repeats. We perform a best-improvement local search heuristic as follows.

Algorithm 9: $\rho - r$ local search heuristic

```
   Data: KP, binary vector s ∈ {0,1}^n, radius and number of repeats ρ, r ∈ ℕ
   Result: s_{ρ−r−heu} feasible for KP or s
 1 for r times do
 2     if B̃_ρ(s) ≠ ∅ then
 3         Find s̃ ∈ B̃_ρ(s) with maximal profit;
 4         Update s := s̃;
 5     else
 6         pass ;
 7 return s_{ρ−r−heu} := s and the associated profit c_{ρ−r−heu}.
```

Lemma 5.6. ***Properties of the*** $\rho - r$ ***local search heuristic***
Given an instance of KP and the optimal profit c^*. Consider a binary vector $s \in \{0,1\}^n$ and the $s_{\rho-r-heu}$ constructed according to Definition 5.5. Denote the profit associated to $s_{\rho-r-heu}$ by $c_{\rho-r-heu}$. Then for all $\rho, r \in \mathbb{N}$ the following statements hold.

(1) The $\rho - r$ local search heuristic has running time $O(rn^{\rho+1})$.

(2) If $\rho = r$ and $s_{1-r-heu}$ is feasible then $s_{\rho-1-heu}$ is feasible and $c_{\rho-1-heu} \geq c_{1-r-heu}$.
If $\rho = r$ and $s_{\rho-1-heu}$ is infeasible we have $s_{\rho-1-heu} = s_{1-r-heu} = s$.

(3) $s_{\rho-r-heu}$ is feasible if and only if $s_{\rho-(r+1)-heu}$ is feasible.
$s_{(\rho+1)-r-heu}$ is feasible if $s_{\rho-r-heu}$ is feasible.

(4) The profits $c_{\rho-(r+1)-heu} \geq c_{\rho-r-heu}$.
If $s_{\rho-r-heu}$ is feasible we have $c_{(\rho+1)-r-heu} \geq c_{\rho-r-heu}$.

(5) If $err_{bit}(s, s^*) \leq \rho$ then an optimal solution is found $c_{\rho-r-heu} = c^*$.

(6) $err_{bit}(s, s_{\rho-r-heu}) \leq r \cdot \rho$.

Proof.

Ad (1) By dropping the feasibility condition in $\tilde{B}_\rho(s)$ we estimate

$$|\tilde{B}_\rho(s)| \le |\dot\bigcup_{i=0}^{\rho}\{y \in \{0,1\}^n \mid err_{bit}(s,y) = i\}| = \sum_{i=0}^{\rho}\binom{n}{i} = O(n^\rho).$$

For each binary vector we requite $O(n)$ to compute the profit, the weight and to check feasibility. Hence, we can determine $\tilde{B}_\rho(s)$ and a maximizer $\tilde{s}$ in $O(n^{\rho+1})$. The maximum can be computed at run-time. By repeating this r for times we end up with a complete running-time of $O(rn^{\rho+1})$.

Ad (2) Let $\rho = r$. If we deploy the 1-r local search heuristic we update at most 1 bit in each of the r repeats and thus after r steps $err_{bit}(s_{1-r-heu}, s) \le r$. In the $\rho - 1$ local search heuristic we consider all solutions with at most ρ different bits, among which we also find $s_{1-r-heu}$ the output of the $1-r$ local search heuristic.

Hence, if $s_{1-r-heu} \in \tilde{B}_\rho(s) \neq \emptyset$ is feasible then the profit-maximizer $s_{\rho-1-heu}$ determined in the $\rho-1$ local search heuristic is a feasible solution with $c_{\rho-1-heu} \ge c_{1-r-heu}$. If $s_{\rho-1-heu}$ is infeasible there exists no feasible solution with a bit error of at most ρ since we exhaustively searched $\tilde{B}_\rho(s)$. Thus also $s_{1-r-heu}$ is infeasible and the algorithms both return s.

Ad (3) Since the $\rho - (r+1)$ local search heuristic also considers changing 0 bits in each round, we encounter $s_{\rho-r-heu}$. Hence, the feasibility of $s_{\rho-r-heu}$ implies the feasibility of $s_{\rho-(r+1)-heu}$. To prove the inverse, we recall that the algorithm returns the input if no feasible solution can be constructed hence the infeasibility is carried over in subsequent rounds.

Regarding the second claim, we similarly argue that $s_{\rho-r-heu}$ is considered in the $(\rho+1)-r$ local search heuristic. If $s_{\rho-r-heu}$ is feasible it appears as a candidate improvement in the $(\rho+1)-r$ local search thus we output some feasible $s_{(\rho-(r+1)-heu}$.

Ad (4) By (3) $s_{\rho-(r+1)-heu} = s_{\rho-r-heu}$ if both are infeasible. If both are feasible $s_{\rho-r-heu}$ appears as a candidate improvement in the $\rho-(r+1)$ local search. If $s_{\rho-r-heu}$ is feasible also $s_{(\rho+1)-r-heu}$ is feasible by (3) and it appears as a candidate improvement in the $(\rho+1)-r$ local search hence proving $c_{(\rho+1)-r-heu} \ge c_{\rho-r-heu}$.

Ad (5) In the $\rho - 1$ local search heuristic all elements with at most ρ different bits are encountered. Therefore also an optimal solution which has $err_{bit}(s, s^*) \le \rho$ appears as a candidate. Hence, we have a candidate with the optimal profit and by feasibility no larger-profit-candidate can appear. Thus $c_{\rho-1-heu} = c^*$. By (4) this holds true for all subsequent rounds thus $c_{\rho-r-heu} = c^*$

Ad (6) In each of the r round at most ρ bits are changed. The result of each round is used as input in the new round and again ρ bits may change. Thus in total at most $r \cdot \rho$ bits of the input s are changed.

□

Definition 5.7. $\rho - r$ ***local search hybrid heuristic***
Given a binary vector $s \in \{0,1\}$ for KP. Let c^{greedy} and x^{greedy} be the greedy profit, respectively the greedy solution of KP. Consider $s_{\rho-r-heu}$ the output of the $\rho - r$ local search heuristic according to Definition 5.5 when using s as input and let $c_{\rho-r-heu}$ be the corresponding profit. We define the hybrid solution $s_{\rho-r-hyb}$ by comparing it to the greedy as follows.

Algorithm 10: $\rho - r$ local search hybrid heuristic

Data: KP, x^{greedy} , $s_{\rho-r-heu}$
Result: $s_{\rho-r-heu}$ feasible for KP

```
1 if s_{ρ-r-heu} feasible and c_{ρ-r-heu} > c^{greedy} then
2 |   return s_{ρ-r-heu}
3 else
4 |   return x^{greedy}
```

Lemma 5.8.
Let $\rho, r \in \mathbb{N}$ and consider an instance of KP with optimal profit c^* and greedy profit c^{greedy}. Consider $s_{\rho-r-heu}$ and the associated profit $c_{\rho-r-heu}$ constructed according to Definition 5.5 and let $s_{\rho-r-hyb}$ and the associated profit $c_{\rho-r-hyb}$ be according to Definition 5.7. Then the following statements hold true.

(1) $s_{\rho-r-hyb}$ is feasible for KP.

(2) If already $s_{\rho-r-heu}$ is feasible $c_{\rho-r-hyb} \geq \max(c_{\rho-r-heu}, c^{greedy})$.

(3) If KP is greedy optimal $c_{\rho-r-hyb} = c^*$. If already $c_{\rho-r-heu} = c^*$ and $s_{\rho-r-heu}$ is feasible then still $c_{\rho-r-hyb} = c^*$.

Proof.
Since the greedy solution is always feasible, we force a feasible output. If $s_{\rho-r-heu}$ is feasible, $s_{\rho-r-heu}$ is the feasible solution with the larger profit by construction and not worse then the better one. In the case that the greedy solution is optimal, we obtain $c_{\rho-r-hyb} = c^*$. □

We predict solutions s by using RNNC-red from the reduced test-set as specified in Section 5.1. Furthermore, we consider the greedy-solutions x^{greedy} of all test-instances. We use both as respective input for the heuristics defined in Definition 5.5 and Definition 5.7 for all parameter choices $(\rho, r) \in [3]^2$. We launch the heuristics on all instances comprising the test set and display the bit-errors in the following table being composed analogous to Table 5.3. The prefix of row labels corresponds to the respective (ρ, r) and in case the hybrid Definition 5.7 is applied we append the syllable -hyb. The postfixes RNNC-red respectively greedy denote the inputs of the respective heuristics.

bit-error	0	1	2	3	4	5	6	7	8	mean	std	median	samples
RNNC-red	68	305	373	189	59	6	0	0	0	1.88±0.063	1.02	2.0	1000
greedy	420	0	173	227	127	41	11	0	1	1.81±0.107	1.72	2.0	1000
1-1-RNNC-red	373	92	122	259	114	36	4	0	0	1.77±0.1	1.62	2.0	1000
1-1-greedy	420	0	173	227	127	41	11	0	1	1.81±0.107	1.72	2.0	1000
1-1-hyb-RNNC-red	500	7	119	208	124	33	9	0	0	1.58±0.107	1.73	0.5	1000
1-2-RNNC-red	465	12	100	251	120	43	9	0	0	1.71±0.108	1.75	2.0	1000
1-2-greedy	420	0	173	227	127	41	11	0	1	1.81±0.107	1.72	2.0	1000
1-2-hyb-RNNC-red	524	1	105	205	122	34	9	0	0	1.54±0.108	1.74	0.0	1000
1-3-RNNC-red	477	0	100	248	123	43	8	1	0	1.71±0.109	1.76	2.0	1000
1-3-greedy	420	0	173	227	127	41	11	0	1	1.81±0.107	1.72	2.0	1000
1-3-hyb-RNNC-red	525	0	106	204	122	34	9	0	0	1.54±0.108	1.74	0.0	1000
2-1-RNNC-red	577	0	1	232	113	58	17	2	0	1.56±0.118	1.91	0.0	1000
2-1-greedy	593	0	2	207	115	60	21	1	1	1.53±0.12	1.94	0.0	1000
2-1-hyb-RNNC-red	623	0	9	186	116	51	15	0	0	1.38±0.115	1.86	0.0	1000
2-2-RNNC-red	578	0	0	232	114	57	15	3	1	1.56±0.119	1.92	0.0	1000
2-2-greedy	595	0	0	207	116	59	19	2	2	1.52±0.121	1.95	0.0	1000
2-2-hyb-RNNC-red	624	0	8	186	117	50	13	1	1	1.38±0.116	1.87	0.0	1000
2-3-RNNC-red	578	0	0	232	114	57	15	3	1	1.56±0.119	1.92	0.0	1000
2-3-greedy	595	0	0	207	116	59	19	2	2	1.52±0.121	1.95	0.0	1000
2-3-hyb-RNNC-red	624	0	8	186	117	50	13	1	1	1.38±0.116	1.87	0.0	1000
3-1-RNNC-red	810	2	6	48	64	58	10	1	1	0.78±0.104	1.67	0.0	1000
3-1-greedy	802	1	3	43	79	57	12	2	1	0.83±0.107	1.73	0.0	1000
3-1-hyb-RNNC-red	824	2	5	45	60	55	7	1	1	0.72±0.1	1.62	0.0	1000
3-2-RNNC-red	866	0	0	4	62	58	8	1	1	0.61±0.098	1.58	0.0	1000
3-2-greedy	849	0	2	8	75	52	11	2	1	0.68±0.102	1.64	0.0	1000
3-2-hyb-RNNC-red	875	0	1	4	58	54	6	1	1	0.57±0.095	1.53	0.0	1000
3-3-RNNC-red	870	0	0	1	61	58	8	1	1	0.6±0.097	1.57	0.0	1000
3-3-greedy	859	0	0	0	75	52	11	2	1	0.65±0.101	1.63	0.0	1000
3-3-hyb-RNNC-red	878	0	1	2	57	54	6	1	1	0.56±0.094	1.52	0.0	1000

Table 5.6: solution prediction heuristics, bit-errors

First of all, let us focus on the ***number of perfectly predicted solutions*** which we can find in the column labeled 0 in Table 5.6. By Lemma 5.8 the number of perfect predictions when applying the hybrid Definition 5.7 does not get smaller and merely adds the respective set of greedy-optimal instances to the set of perfect predictions. For this reason, the application of Definition 5.7 when using x^{greedy} as input does not have an effect, and hence, we did not consider it.

Now let us recall what effect one round of the heuristic (Definition 5.5) has. We simply add all false predictions having a bit-error of at most ρ to the set of perfect predictions as proven in Lemma 5.6. Solutions with a ***bit-error of at most*** ρ thus inevitably become optimal. In this way, the number of perfect predictions in each row can already be computed merely from the columns $1 \ldots \rho$ and an application of Definition 5.5 is not necessary in case we are only interested in the number of optimal predictions obtained by a one-time application of Definition 5.5. However the underlying optimal solution is only known after the actual application of Definition 5.5. Since the greedy-solution never admits a bit-error of 1 the application of the $1 - r$ heuristic never results in an optimal solution when using x^{greedy} as input.

Having discussed the behavior of the heuristic on instances with bit-errors of at most ρ let us move on and take a look at what happens to solutions s, x^{greedy} ***having a bit error larger than*** ρ. These solutions do not immediately become optimal (except maybe if the optimal solution is not unique) after one round of the heuristic
(Definition 5.5). By looking at the changing numbers of solutions having a bit error larger than a fixed ρ and when changing r it becomes evident that the application of rounds has a large effect on those distant solutions too. Typically there is a feasible and potentially better solution within a bit-error radius of ρ which becomes the new center of the neighborhood. However, it is not yet optimal. Moreover it very often is the case that the new center admits a smaller bit error and in particular many solutions now may admit a bit-error of at most ρ inevitably becoming optimal in the next round. The improvement with regard to bit-error is not to be taken for granted. For an update, we choose a feasible solution only having a maximal profit increase. So besides pushing almost-correct (up to ρ wrong bits) solutions towards optimality an application of Definition 5.5 literally ***draws solutions closer to the optimal one with respect to bit-error***. As long as solutions gravitate towards a smaller bit error an additional round of Definition 5.5 is promising. For $\rho = 1$ this seems to be the case for only two times $r = 2$. For $\rho \in \{2, 3\}$ this effect is exhausted even earlier. Perhaps if r is significantly larger this effect experiences a renaissance and a trade off between ρ and r has to be faced in any case.

In order to better grasp the application of the hybrid (Definition 5.7) and the ***relation of perfect predictions to greedy-optimal solutions***, we look at the sets of greedy-optimal solutions, the sets of perfectly predicted solutions, and their overlap. In other words we compare the predictive power of the greedy vector to the predictive power of the (heuristically improved) predictions using RNNC-red. We allow both being improved by the heuristic but only consider perfect predictions with respect to the individual methods.

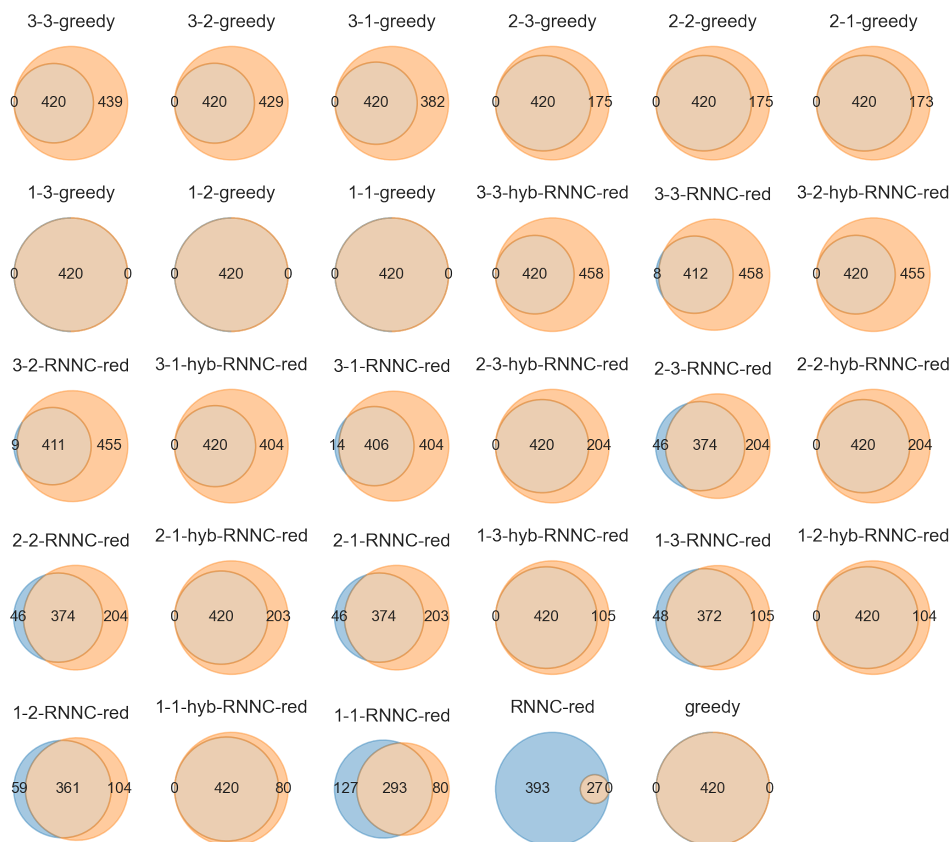


Figure 5.3: RNNC-red; sets of perf. pred.; Venn diagrams

Clearly using ***applying the hybrid*** (Definition 5.7) to an output of the heuristic (Definition 5.5) simply corresponds to adding the set of greedy-optimal solutions (blue, left hand circles in venn-diagrams in Figure 5.3) to the set of perfect predictions (orange, right hand circles in venn-diagrams in Figure 5.3). In this sense, we take the best of both worlds. Thus the more interesting aspect is looking at the relation between instances perfectly predicted by the greedy solution (i.e. greedy optimal) and instances which can be perfectly predicted by applying the plain (and not the hybrid) heuristic (Definition 5.5) on respective input vectors.

The very few ***perfect RNNC-red predictions***, denoted by s, without applying any neighborhood search are all achieved on greedy-optimal instances.
For $\rho = 1$ we observe that the resulting s_{1-r} do not only manage to perfectly predict more greedy-optimal instances but also some non-greedy-optimal instances.
When using $\rho = 2$ repetitions do not have any notable effect and the majority of greedy-optimal instances is perfectly predicted.

When using $\rho = 3$ the repetition seems to again be effective, and the set of perfectly predicted greedy-optimal instances is almost covered, already for $r = 1$.

When using the ***greedy solution*** x^{greedy} as input for the heuristic (Definition 5.5) we only observe an effect for $\rho \geq 2$ as discussed above. However in these cases for each r we observe only slightly weaker predictions as opposed to using s as input.

In conclusion, we recall that the $\rho - r$ local search heuristic has a ***running time*** of $O(rn^{\rho+1})$, see Lemma 5.6. While repetitions only increase the running time by a factor, their power seemingly is already exhausted after $r = 2$ steps regardless of the input x^{greedy} or s. Increasing the search radius ρ entails polynomial expense but seems very effective. When using x^{greedy} it is provably needed to use $\rho \geq 2$ to obtain an improvement. Thus when using x^{greedy} a minimal running time to achieve an improvement is cubic in n for fixed r. If we are willing to spend this much or even more computational cost we only achieve almost as good solutions as when using s as input. In this regard, however, the greedy solution catches up quite well.

So the ***leverage of this heuristic*** in combination with RNNC-red lies in using $\rho = 1$ and ideally using $r \geq 2$ repetitions. This provides quadratic running time for constant r. Computing the greedy can be done in $O(n)$ and thus applying the hybrid is a cheap way of getting many optimal heuristic solutions $s_{1-r-hyb}$ in $O(rn^2)$. The behavior of the heuristic (Definition 5.5) and the hybrid (Definition 5.7) with regard to the associated profit is discussed later.

5.4 Truncated branch-and-bound algorithm for KP

Any branch-and-bound algorithm can be used as a heuristic by interrupting (i.e. truncating) it at a suitable point. If we fetch the current incumbent solution we obtain a feasible solution which may or may not be optimal but is at least as good as the initial solution. We will apply this on BB and use a ML-aided node-selection for generating a heuristic solution for KP. The motivation for employing one of the learned node-selection rules is that the minimal exploration phase exactly comprises the PTO which we targeted during learning.

Definition 5.9. ***A truncated branch-and-bound algorithm (TBB)***
We use BB as defined in Section 2.2.3 with DFS-in-pto with restart as defined in Section 4.3. At the first time an active node becomes integral we ***break the while loop in BB*** and force the algorithm to terminate. Afterwards we compare the integral solution attained at this final node to the initial solution and take the more profitable of the two.

Remark 5.10.

(1) There are ***many potential events*** at which truncating BB would be natural. One such event is the time when the first incumbent update happens. In this setup we would however explore the entire branch-and-bound tree for greedy-optimal instances because we will never find a better solution. Another reasonable setup would

be to run BB until an integral node not worse than the initial solution is detected. In contrast to our rule, this would allow us to detect several integral nodes and perhaps dive through the initial struggle.

Recall the evaluation of the ML-aided node-selection rules (see Table 4.3) showed on the one hand we ***find an optimal node relatively quickly*** in a typical instance. On the other hand, there exist instances where the search for an optimal node is extensive even when using restarts. We hope to avoid these costly explorations by truncating at the first integral node which perhaps is found way earlier. In any case, we can not obtain a worse result than the greedy by comparing to the initial solution.

(2) Interestingly regarding this heuristic ***BESTFS is the worst node-selection among all optimal ones***. BESTFS processes all nodes of $T(c^*)$ before an integral node admitting c^* is found, see Lemma 2.29 and no optimal node-selection explores nodes outside of $T(c^*)$ by Theorem 2.36.

(3) A truncated branch-and-bound algorithm generally (if the last node is not the only active node left) does ***not provide a proof of optimality*** which would require moving through the entire validation phase. Thus even in case the result is an optimal solution TBB does not inform us whether the solution is optimal.

Having defined TBB let us evaluate the branch-and-bound trees on the test set as defined in Section 5.1 by reevaluating the instances. We look at the ***sizes of branch-and-bound trees*** $|T|$ which would have been obtained by BB if it was not truncated, the length of respective ***exploration phases*** $|N_E|$ and the time until a ***first integral node*** (i.e. prune by integrality or prune by new incumbent) was detected reflecting the duration of TBB. We perform this evaluation by using several node-selection rules and discuss their suitability below.

	all nodes			exploration phase			first integral			samples
node-selection	mean	std	med	mean	std	med	mean	std	med	
BESTFS	122.6±6.8	109.1	95	45.5±3.7	58.9	25.0	61.8±3.4	54.6	48	1000
DFS-exclusion	231.5±15.8	255.3	133	157.7±14.7	237.4	52.5	18.4±0.7	10.6	18	1000
DFS-exclusion-w-r	182.1±11.6	186.5	121	92.9±9.4	151.4	30.0	18.4±0.7	10.6	18	1000
DFS-in-pto	178.2±11.8	190.5	109	96.1±10.0	161.2	17.5	65.7±4.5	72.2	40	1000
DFS-in-pto-w-r	145.8±8.7	139.7	103	43.5±4.4	70.6	15.0	51.2±3.4	54.3	36	1000

Table 5.7: TBB, duration

First of all, we note that it can well be that the exploration phase is shorter than the time until an integral node is found. This is exactly the bias being introduced when having an optimal initial solution being the greedy in our case.

The duration of BB until an integral node is found is exactly the ***time at which TBB terminates***. In mean, both DFS-in-pto (with restart) are not significantly worse than

BESTFS and engaging the restart rule is recommended from a performance point of view. In particular, the restart rule is actually being applied when solving certain instances. This means before DFS-in-pto detects an integral node it detects leaves which are either infeasible or pruned by bound.

Regarding ***BESTFS*** it is visible that the proven formula $T_U = |\overline{T(c^*)}| = 2|T(c^*)|+1$ holds true. $|T(c^*)| + 1$ is exactly the time until the first integral node is detected and $T = T_U$ coincide when using BESTFS, see Section 2.2.5 for proofs and full details. Furthermore at time $|T(c^*)| + 1$ BESTFS is guaranteed to provide an optimal node, also proven in Section 2.2.5. Therefore, finding an integral node faster than BESTFS is a minimal requirement we demand from a node-selection rule in order to be used in the truncation heuristic (Definition 5.9).

Let us briefly discuss the seemingly well performing rules ***DFS-exclusion (with restart)***. It was shown in Chapter 2 that the node corresponding to the trajectory where all items are excluded provides the greedy solution. In particular this trajectory is traversed by DFS-exclusion (with restart) first. By breaking at this first integral node being detected, it is evident that these node-selection rules are not suited for this heuristic besides providing fast termination (of at most n steps).

Below in Figure 5.4 we look more closely into the ***choice of the truncation criterion***. We use BB with each node-selection rule of interest to solve the instances in the test set. For each instance we look at the sequence of incumbent updates. Each updated profit is normalized by the optimal profit of the respective instance. All sequences are padded with ones at the end in order to achieve uniform length. We then average over all instances the respective i-th normalized profit thus obtaining an ***average sequence of normalized profits*** for each respective node-selection rule. The padding of ones ensures that no bias towards instances with more incumbent updates is introduced when averaging. We then compute all ***mean hitting times*** being the average processing time until an i-th incumbent update applied. In Figure 5.4 we plot the relation between these two quantities and interpolate linearly.

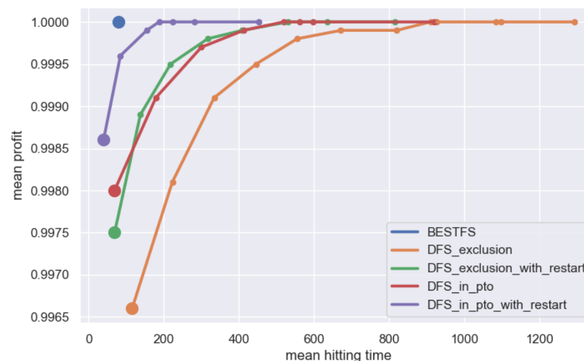


Figure 5.4: TBB; incumbent updates

We purposefully omit all nodes at which an integral node, but not an incumbent update was encountered. These nodes show that the mean bounds obtained at integral nodes fluctuate heavily over time and visual insight can hardly be deduced. Figure 5.4 motivates to use TBB in order to ***strictly improve the initial solution***.

Looking at Table 5.7 we can observe that DFS-exclusion (with restart) is able to find an integral node quickly but Figure 5.4 shows the time until a first incumbent update can be made is long. DFS-in-pto (with restart) is capable of both, finding an integral node quickly and it also finds a first improved integral solution quickly. Also note that the mean hitting times for DFS-in-pto with restart live on the most narrow and early time frame and DFS-exclusion again admits the exact opposite in this regard. However the ***interpretation of the time aspect*** shall be done with caution because we do not consider the number of instances which require many incumbent updates. So note that the large mean hitting times are obtained by (drastically) fewer instances.

Not only does the first incumbent update when using DFS-in-pto happen early, also the ***quality of the first new incumbent solution*** is on average the best besides BESTFS. While DFS-in-pto with restart provides the best mean incumbents throughout, DFS-in-pto without restart behaves slightly worse but similar to DFS-exclusion with restart in this regard. DFS-exclusion provides the weakest mean incumbent updates at each hitting time. The tendency of achieving large improvements early and running into a plateau later on is pronounced strongest by DFS-in-pto (with restart).

Focusing on BESTFS we are guaranteed to obtain an optimal solution at the first incumbent update (see Lemma 2.29) however still it does not provide a first incumbent update quickest. We therefore conclude that ***DFS-in-pto with restart is best applicable*** in TBB. ***BESTFS is also a good choice*** by the guarantee of an optimal solution but for an attempt in quickly improving the greedy, it is not as well suited. We finally note that all mean incumbent updates c^{inc} have an optimality gap $\frac{c^{inc}}{c^*}$ of less than 1%.

5.5 Evaluation of predicted profits

We dedicate this section to evaluating the profits which are associated with the predicted solutions. Again we use the same test set as defined in Section 5.1 throughout. We evaluate the individual techniques defined in this chapter and provide tables which hold the regarded quantities. We collect the data associate to the respective technique in a row and the columns reflect aggregates of the per data point bias error. The last column respectively reflects the number of instances which were predicted using the respective methods where we always fall back on the same 1000 instances from the test set.

Profit evaluation of DNN-red, DNN-full, RNN-red, RNN-full

First of all, let us evaluate the end-to-end methods which do not have a solution vector associated. The ***greedy-lp-mean*** is a fair baseline comparison for these methods since it also does not have a concrete solution associated.

bias-error	mean	std	median	max	min	samples
greedy-lp-mean	0.011±0.002	0.039	0.014	0.140	-0.195	1000
DNN-full	0.034±0.017	0.276	0.047	0.770	-1.741	1000
DNN-red	0.017±0.029	0.469	0.033	1.373	-2.095	1000
RNN-full	0.011±0.003	0.043	0.019	0.111	-0.218	1000
RNN-red	0.024±0.016	0.252	-0.005	1.263	-0.686	1000

Table 5.8: profit-prediction; DNN-full, DNN-red, RNN-full, RNN-red

It is visible in Table 5.8 that the only reasonable method among these is RNN-full. It achieves similar results as the greedy-lp-mean having only slightly more deviation and slightly less min bias error. DNN-full and RNN-red seem to have a similar performance and DNN-red is practically unusable by having extreme errors and a large deviation.

Profit evaluation of RNNC-red

We move on to RNNC-red which is a model which predicts a solution vector. Therefore the ***greedy profit*** may be considered as a fair baseline comparison.

bias-error	mean	std	median	max	min	samples
greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
RNNC-red	-0.189±0.016	0.264	-0.171	0.909	-1.102	1000

Table 5.9: profit-prediction; RNNC-red

Table 5.9 shows a performance similar to DNN-full or RNN-red which however do not provide an associated packing. In particular, an RNN seems to be equally well suited for profit prediction regardless if we demand the profit prediction stemming from a concrete solution or not.

Profit evaluation of the local search heuristic

We evaluate the heuristic solutions being generated according to the local search heuristics (with hybrid) (Definition 5.5, respectively Definition 5.7). In both cases we use the greedy-solution x^{greedy} and the RNNC-red prediction s as an input. Evaluating for pairs (rho, r) taking all combinations of numbers 1 to 3 provides us the following table which we subdivide according to different values for ρ and r. The row labels are of the form ρ-r-input consistent with Table 5.6.

bias-error	mean	std	median	max	min	samples
greedy-lp-mean	0.011±0.002	0.039	0.014	0.140	-0.195	1000
greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
1-1-RNNC-red	-0.053±0.004	0.070	-0.024	0.000	-0.376	1000
1-1-greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
1-1-hyb-RNNC-red	-0.035±0.004	0.057	-0.000	0.000	-0.341	1000
1-2-RNNC-red	-0.041±0.004	0.061	-0.007	0.000	-0.341	1000
1-2-greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
1-2-hyb-RNNC-red	-0.034±0.003	0.056	0.000	0.000	-0.341	1000
1-3-RNNC-red	-0.041±0.004	0.061	-0.006	0.000	-0.341	1000
1-3-greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
1-3-hyb-RNNC-red	-0.034±0.003	0.056	0.000	0.000	-0.341	1000
2-1-RNNC-red	-0.03±0.003	0.054	0.000	0.000	-0.335	1000
2-1-greedy	-0.03±0.003	0.056	0.000	0.000	-0.454	1000
2-1-hyb-RNNC-red	-0.025±0.003	0.048	0.000	0.000	-0.335	1000
2-2-RNNC-red	-0.03±0.003	0.054	0.000	0.000	-0.335	1000
2-2-greedy	-0.03±0.003	0.056	0.000	0.000	-0.454	1000
2-2-hyb-RNNC-red	-0.025±0.003	0.048	0.000	0.000	-0.335	1000
2-3-RNNC-red	-0.03±0.003	0.054	0.000	0.000	-0.335	1000
2-3-greedy	-0.03±0.003	0.056	0.000	0.000	-0.454	1000
2-3-hyb-RNNC-red	-0.025±0.003	0.048	0.000	0.000	-0.335	1000
3-1-RNNC-red	-0.008±0.001	0.024	0.000	0.000	-0.184	1000
3-1-greedy	-0.01±0.002	0.028	0.000	0.000	-0.225	1000
3-1-hyb-RNNC-red	-0.007±0.001	0.022	0.000	0.000	-0.157	1000
3-2-RNNC-red	-0.005±0.001	0.019	0.000	0.000	-0.173	1000
3-2-greedy	-0.006±0.001	0.021	0.000	0.000	-0.211	1000
3-2-hyb-RNNC-red	-0.005±0.001	0.018	0.000	0.000	-0.157	1000
3-3-RNNC-red	-0.005±0.001	0.018	0.000	0.000	-0.173	1000
3-3-greedy	-0.006±0.001	0.020	0.000	0.000	-0.211	1000
3-3-hyb-RNNC-red	-0.005±0.001	0.018	0.000	0.000	-0.157	1000

Table 5.10: profit-prediction; local search (hybrid)

We observe in Table 5.10 that choosing $\rho = 1$ is not sufficient to outperform the greedy-lp-mean. However, we already dominate the plain ***greedy profit*** which after all is a fairer benchmark if we ask for an associated solution. After one repetition $r = 1$ the input vector s from RNNC-red is worse than launching the heuristic on the greedy solution x^{greedy} which in considered case $\rho = 1$ only results in the input x^{greedy}. The hybrid is not worth being discussed separately as it simply takes the better solution of the two and naturally outperforms both.

Increasing r for $\rho = 1$ changes the preferred input and s is preferably used. At this point we can also outperform the (unfair competitor) ***greedy-lp-mean*** at least in median. For $\rho = 2$ we are even closer to greedy-lp-mean but still have more deviation in the predicted profits. At $\rho = 3$ we finally also beat the greedy lp-mean in all categories and repetitions hardly have an influence on the profit at this point.

Profit evaluation of TBB

Now let us discuss the predictions coming from (TBB) as defined in Definition 5.9. Again we have associated a packing and thus the ***greedy shall be our primary baseline***. Recall that TBB yields results at least as good as the greedy by construction and we merely ask how much we can improve.

bias-error	mean	std	median	max	min	samples
greedy-lp-mean	0.011±0.002	0.039	0.014	0.140	-0.195	1000
greedy	-0.049±0.004	0.071	-0.014	0.000	-0.454	1000
DFS-in-pto-tbb	-0.019±0.003	0.041	-0.000	0.000	-0.318	1000
DFS-in-pto-wr-tbb	-0.013±0.002	0.034	-0.000	0.000	-0.271	1000

Table 5.11: profit-prediction, TBB

We observe in Table 5.11 we are able to outperform the the greedy profits in a many cases and a restart (DFS-in-pto-wr-tbb) is preferable.

Taking the somewhat unfair ***greedy-lp-mean as comparison*** we achieve a smaller median error and less deviation. Also when regarding the span of the error (max minus min) the method TBB dominates the greedy-lp-mean and the average bias error is quite close. Note that the unfairness does not only arise by demanding an associated packing but also has to do with having feasible packing as it is the case for TBB. The predictions from TBB are in $[c^{greedy}, c^*]$ while the greedy-lp-mean can deviate in $[c^{greedy}, c^{lp}]$ around c^* and we introduce a bias in this way. The mean bias error when using TBB will only be zero if all predictions are perfect, and the median is only zero if more than half of the predictions are perfect.

If one would again be only interested in a ***profit estimation*** one could empirically evaluate the expected error of TBB and subtract it. In this way, we enjoy the smaller deviation in contrast to the greedy-lp-mean and get closer to the optimal profit on average. The TBB however is not guaranteed to terminate in polynomial time but perhaps yields faster performance than BB and more accurate estimates than the greedy-lp-mean which in turn can be computed in $O(n)$.

Chapter 6

Implementation

In order to obtain the evaluations presented in chapters 5 and 4, we implement BB as presented in Definition 2.7. We use the programming language ***Python*** (v.3.8.0 64-bit) where many convenient packages for setting up ML-tasks are available.

We use the IDEs ***Atom*** (with suitable extensions), ***Spyder*** and ***Jupyter Notebook*** for writing and executing the code. For data science related tasks it is highly recommended to use some sort of interactive IDE like these. In this way, one can store intermediate results and data in memory and directly access it for plotting and data manipulation without having to reload it repeatedly.

The packages are managed using ***Anaconda***. We use Google's ***Tensorflow*** and in particular the sublibrary ***Keras*** for creating, fitting and evaluating the Neural Networks. For assessing the fit of models, we use the respective metrics which are implemented in the library ***scikit-learn***. In order to keep track of the data we use the library ***pandas*** which offers SQL-like abilities for processing tabular data. All data is stored using the package ***pickle*** being well integrated to pandas. For storing the models we use Keras. The data-visualization is done by using ***matplotlib*** and ***seaborn***.

All computations are performed on the CPU of a ***standard Home PC*** using Windows 10 64-bit, an Intel Xeon CPU (E3-1231-v3, 3.40GHz, 4 cores, 8 logical cores), an Nvidia GPU (GeForce GTX 970) and 16GB of RAM.

6.1 Setting up a branch-and-bound algorithm

Both ***items and nodes*** of KP are represented as classes in the sense of object oriented programming. The list of items defining an instance of KP serves as input for BB, and we can omit the capacity by assuming $W = 1$ as random instances meet the Permanent Assumption 1. A notable detail is that we also normalize in particular the benefits by their maximum which is good practice for inputs of a neural network. The algorithm BB is not influenced by this. At runtime the nodes are instances of a node class having attributes such as the fixed weight and profit, the unfixed items, the critical item, the lp and greedy bounds, and the current incumbent profit.

Furthermore BB takes as input a ***node-selection rule*** which is an instance of a node-selection class. Both a potential machine-learning model and a function which selects nodes from a set of active nodes are stored as attributes. In this way we only need to load an ML-model before launching BB (potentially many times for benchmarking) and gain efficiency.

The ***machine learning model*** is stored as an instance of an ML model class. It first and foremost comprises the neural network being used and a getter function which fetches the correct columns (features) as inputs from a given node. In case an ML-node-selection is used, we also store the predicted label, the associated estimated probability, and the sibling similarity (if the node is not the root) in the respective instance of the node-class. Of course the prediction can only be performed when all quantities of the model input are defined. In particular, the root and leaves are not predicted. In this case, we artificially set the PTO-probability to 1 thus preferring leaves (they may offer a new incumbent) and the root is in PTO anyways.

The ***output of BB*** is a pandas dataframe (similar to a MS-Excel table) harboring the nodes and a dataframe which returns the items being used also in form of a dataframe. In case we are interested in the actual optimal solution and not only the optimal profit, we transform the node data frame into a tree where a tree-search algorithm using the networkx library returns the PTO. In this way we can reconstruct the instance via the fixed items and again solve the lp in linear time for obtaining the optimal solution. We explicitly do not store solutions associated with nodes as this substantially adds to the memory being used.

6.2 Setting up the ML-tasks

In this section we focus on notable details when targeting the ML-tasks of chapters 4 and 5. By using random instances as defined in Definition 4.1 and by scaling the benefit according to the Permanent Assumption 3 all input features except for the depth used for DNNC-pto are in $[0,1]$. Scaling and general preprocessing can greatly influence the performance of ML models and in our case these preprocessing steps turned out to be adequate.

6.2.1 Learning profits and solutions

The respective architecture of the models ***DNN-red, DNN-full, RNN-red, RNN-full*** is already defined in Chapter 5. Thus we clarify details about the input-data and the learning procedure. All inputs are associated with items and we fix an ordering of the components by benefits as specified in the Permanent Assumption 1. Recall that in Chapter 5 we only consider instances of size $n = 50$.

When using ***DNN*** we require the ***input in form of a vector***, and therefore, simply concatenate the profits and weights $x = (c_1, \dots, c_n, w_1, \dots w_n) \in \mathbb{R}^{100}$. When adding the lp and greedy bounds as input for DNN we construct input vectors of the form $x = (c_1, \dots, c_n, w_1, \dots w_n, c^{greedy}, c^{lp}) \in \mathbb{R}^{102}$.

RNN however takes ***time series as input***, and we define a matrix $T \in \mathbb{R}^{50\times 2}$ column-wise by $T_1 = (c_1, \dots c_n)$ and $T_1 = (w_1, \dots w_n)$. Each row $x^{(t)} = (c_t, w_t)$ represents the input at a time $1 \le t \le \tau = 50$. In order to add the greedy and lp-bound to the input we compose $T \in \mathbb{R}^{50\times 4}$ by using $x^{(t)} = (c_t, w_t, c^{greedy}, c^{lp}), 1 \le t \le \tau = 50$ as inputs throughout time. In this way at any time the bounds are known to RNN and offer a look ahead and declares what profit range is being targeted.

Let us now briefly talk about ***fitting the models***.
For the classifier RNNC-red we used the (sparse) categorical cross entropy loss, see below. For the other four regression models for predicting the profit we used the mean squared error loss function. The models were fitted by using Adam with its default parameters. We only adapt the learning rate ϵ for training the particular models.
For RNNC-red, RNN-red and RNN-full we used $\epsilon = 0.01$ and for DNN-red, DNN-full we used $\epsilon = 0.001$. In each respective training session, we used 50 epochs which in all cases provided small enough losses to declare convergence. Keras offers a feature which allows us to track various metrics such as accuracy or mean errors throughout training on the partly fitted model. Also based on these measures we observed satisfying convergence behavior.

The ***input for the models*** when using Keras is given as a sequence which contains all the training data points and is identified as a matrix (a 2d numpy array) in case a data points is a vector (DNN) and as a tensor (a 3d numpy array) in case a data point is a matrix (RNN). When performing a ***sequence-to-1 prediction*** using RNN as the case for profit predictions, we feed the output of the LSTM-layers in what is called a Dense layer in Keras. In this way, only the last hidden state of the LSTM layer is propagated. When performing a ***sequence-to-sequence task*** with RNN like we do when learning solutions, we need to propagate the current hidden state in each time step when using subsequent layers. In this case, the Keras Dense layer is wrapped in what Keras calls a Time Distributed layer.

Obeying these pitfalls regarding the input format and the ***stacking of layers***, we can quite simply define models within Keras Sequential framework which simply lets us stack layers. In sequence-to-sequence learning we shall not be confused by the use of only two output neurons (instead of maybe 50 being the other logical choice in our case) when performing classification. Keras interprets the data points being fed through the network in an online fashion perfectly corresponding to the mathematical framework. However, the output of said model is a matrix containing 2-fold rows harboring the estimated posterior probabilities of each item.

Luckily for ***classification***, Keras performs a one-hot encoding for evaluating the binary cross entropy loss automatically when using the sparse categorical cross entropy loss, and the targets can simply be given as a matrix holding all solutions as rows. When learning the profits of instances, we simply only need to give a vector harboring the targeted respective profits.

6.2.2 Learning node-selection rules

Concerning the model DNNC-pto used for node-selection in Chapter 4 we refer to the above section where we already discussed how we address null-values in the inputs. The ***data format*** required for the input format of a DNN type model in Keras is already covered above. We again used Keras and the Adam optimizer to fit the model according to the sparse categorical cross entropy. We trained for 50 epochs using a learning rate of $\epsilon = 0.01$.

6.3 Real time component

One general remark to be made concerns the real time required to set up and perform the learning tasks and the evaluations. In order to generate the training data it is required to solve the respective number of instances using BB.

Typically we use ***BESTFS for generating training/test data*** by proven minimality. As proven when using BESTFS we actually could truncate BB at the time the first integral node is found and obtain a proven optimal solution. This would omit the validation phase which makes up about half of the branch-and-bound tree, see Section 2.2.6. However, for the sake of flexibility of BB we did not do that.

We achieved real time ***performances*** for BB of about (25,12), (50,3), (75,2), (100,1) given in pairs (instance size, solutions per second). In this way generating the nodes for training DNNC-pto takes about half an hour (4000 instances), respectively the test set (1000 instances) is generated in about 7.5 minutes. In order to generate what is called the full and reduced training set in Chapter 5 we of course only solved the 9000 respectively 1000 instances once and take the according slice of the dataframe when needed. The training set was generated in about 50 minutes and the test set is computed in about 6 minutes.

Please note that the path when doing an ML task is potentially accompanied by many ***dead ends***, and we tried various sizes of training and test sets ranging up to 100.000 instances. In this regard incorporating 9000 instances seemed to be a critical number of instances which still provided sufficient generality (i.e. approximation accuracy of p_{data}) to learn the task. In turn 1000 test instances were necessary to achieve a certain level of significance (i.e. approximation accuracy of $\widehat{p_{data}}$ and similarity to p_{data}).

The real time used for solving KP with BB when ***using other node selection*** than BESTFS can be deduced from Table 4.3. The proportions reflect in real time since evaluating the relatively small model DNNC-pto is done in practically no time. By using a node-selection class we need to load the model once before solving many instances which is a notable contributing factor. Loading the model with Keras can take a few milliseconds up to seconds for large models.

The real time component is also a limiting factor to the size and types of models which we used outside of BB. Using Keras and Adam ***training the DNN*** models on the respective sets takes tens of seconds while ***training the RNN*** models takes hundreds

of seconds. Note that generally RNNs suffer from extensive training costs because by unfolding the networks when using BPTT, we can end up with very deep networks.In addition, when using LSTM cells the number of parameters increases by a factor.

6.4 Heuristics

The heuristics ***local search heuristic*** defined in Section 5.3.1 is implemented directly as in the definition. The ***hybrid*** can be applied by comparing the associated weight and profits of the solution in a large dataframe which collects all relevant quantities. Similarly, we do not implement the branch-and-bound heuristic ***TBB*** defined in Section 5.4 explicitly but rather compute the entire tree by using BB as mentioned in Section 6.2.2. We then only a posteriori determine the first integral node and compare it to the greedy solution being stored in the very first node. The reason being that we aim to show how much computational costs we save in contrast to fully executing BB which we can only tell by fully evaluating it. Needless to say, in a real application one only profits by performing an actual truncation.

Chapter 7

Conclusion

Let us briefly recapitulate the contents and motivate further steps which can be taken in various directions.

Recapitulate Chapter 2

In Chapter 2 we formally introduced the 0-1-fractional knapsack problem. We explained the setup for a basic branch-and-bound algorithm BB for generating exact solutions. We showed that the ***BESTFS node-selection strategy is optimal*** with respect to the size of resulting branch-and-bound trees if a standard set of parameters is fixed.

By subdividing the branch-and-bound algorithm BB in phases before and after an optimal node is found we have available a finer scope for looking at the performance. In this regard, we could show results which give ***insight into what happens in the individual phases***. We showed an estimation of the duration the validation phase can be made on the basis of an unavoidable subtree and by considering the tree being generated until an optimal node is found. In case of BESTFS we showed that the unique minimal branch-and-bound tree can be given explicitly.

Possible refinements and other fields of application

In a subsequent elaboration one could extend this notion of ***optimality of node-selection rules*** to more general combinatorial optimization problems. In order to do so meaningfully, one requires well defined branch-and-bound trees or at least a well defined measurement for their size. Similarly, one could observe whether branch-and-bound trees all grow within a big maximal tree as in Section 2.2.5.

In regard to ***bounding the size of branch-and-bound trees***, see Section 2.2.6, we state that one can easily improve the estimation $|T| \leq 2t^* + 2|\mathcal{N}(c^*)| - 1$ (Theorem 2.36) by properly using inclusion-exclusion as opposed to estimating the sizes of the union by the sizes of the individual sets. Moreover the subtree containment $T \subseteq \overline{T(c^*)} \cup \overline{T_E}$ gives rise to finer estimations of the upper bound $\overline{T(c^*)} \cup \overline{T_E}$ (w.r.t. $\subseteq$). Note that proving the inverse inclusion does not work by potentially encountering (avoidable) nodes which

can be pruned by bound if a good enough incumbent is available. One could therefore refine $\overline{T(c^*)} \cup \overline{T_E}$ perhaps by partitioning T_E into subtrees where the same incumbent is available and estimate the individual sizes of sets of these avoidable nodes to obtain a tighter bound.

Recapitulate Chapter 3

In Chapter 3 we introduced the machine learning techniques used for the modeling in this elaboration. We exclusively referred to the literature in order to offer a smooth introduction. We recalled ***classical techniques for regression and classification*** and presented more modern concepts involving ***neural networks***. In order to get an understanding of the modelling capabilities of neural networks, we presented some ***universal approximation results*** (Hornik (et al.), [17], [18], Maiorov et al. [26], Ismailov [19], Cybenko [8] and Schäfer et al. [32]).

Other promising techniques

For further application of ML in a combinatorial optimization context it is highly recommended to additionally look into ***pointer networks*** which were already successfully applied for instance to the TSP or to convex hull computations. Furthermore it seems promising to get familiar with ***ensemble methods*** which combine multiple models to reduce the variance or bias to fine tune a model to a given situation.

Recapitulate Chapters 4, 5

In chapters 4 and 5 we presented various ways of applying ML to the knapsack problem. The major concern in the context of ML-aided combinatorial optimization perhaps is ***finding a suitable framework*** within which ML models are applied (end-to-end, finding useful properties, or within an algorithm, see Bengio et al. [5]).

Regarding the ***end-to-end methods***, we pointed out two distinct entry points which are already widely used. One is via directly predicting optimal profits and the second one is via predicting an associated solution. The basic ideas concerning the profit prediction are adaptations of Martinis work [29]. In our scenario, the approach of learning solutions seemed to be preferable. It not only admits profit predictions of equal quality but straightforward local search heuristics allow to further improve predicted solutions. Furthermore we analyzed the troublesome items in predicting packings and found they tend to share a similar benefit and tend to admit both large profit and weight tensions.

Optimizing a node-selection rule in the context of branch-and-bound for the knapsack is only interesting from a heuristic point of view (Section 5.4). Still we managed to get close to an optimal node-selection rule and on a typical instance obtained a reduction of the duration of the exploration phase.

An application of an ML-model for ***aiding a heuristic*** is a valid approach already mentioned by Bengio et al. [5]. We followed this approach and set up a specific truncated branch-and-bound algorithm TBB which is guided by a ML-node-selection rule. In this regard we noticed that there is room for improving BESTFS by trying to move along PTO in the exploration phase. In the greedy-optimal case DFS-exclusion rules behave optimally in this regard. In this setting ML-aided node-selection rules seem promising to manage the trade off between quick termination and good quality of heuristic solutions.

Further applications of ML on end-to-end methods

The variant of directly applying ML to an instance of KP (***end-to-end method***) can certainly be executed more carefully and larger computational power would perhaps help too. For instance, Martini conducted this approach more diligently for the knapsack problem with $n = 200$ and showed that indeed satisfactory end-to-end modeling is possible [29]. As we have seen most instances of size $n = 50$ are greedy optimal (and thus solvable in linear time) and the mean of greedy and lp profit is a very good estimator for such a small problem size. Perhaps at this stage using ML-techniques does not yet pay off and one is motivated to expand to larger instances. Still the fundamental ideas apply and can be expanded to other problems at hand.

The ***learning of optimal solutions*** can potentially be improved and fully utilized (in particular on a packing-level as opposed to the item-level) by using different models and moving onto larger instances. The application of a local search strategy may no longer be necessary for outperforming reasonable baselines. Still the application of a local search can be used to improve a prediction, in particular if it is close to optimal. It may also be interesting to experiment with variable neighborhoods throughout search. We observed that the leverage of constantly using large neighborhoods is exhausted quickly. It might be preferable to cover many solutions in the first round by using a large neighborhood and then switching to a smaller neighborhood allowing for more repetitions to be performed within the same time.

Further variants of applying ML within a branch-and-bound algorithm

Sticking with ***learning node-selection rules*** it may be a good leverage point to use ensemble methods which can compensate the downsides of individual models. In our case we observed that our single model for predicting the direction of trajectories traversed in a DFS-fashion is troubled close to the root. One may be able to find a more specialized model for these nodes and then fade to another method deeper in the tree or use more advanced ensemble learning schemata.

Moving to other areas of application within a branch-and-bound algorithm is conceivable that instead of evaluating lp, one works with estimated profits (perhaps by using one of the presented end-to-end profit predictions) and obtains ***heuristic pruning by bound***. One could also heuristically fix a part of the solution (perhaps by using a model similar to the presented end-to-end solution prediction). This approach would correspond to

applying BB only on a subtree of a branch-and-bound tree and thus limiting the search space by learning a promising root of the subtree. In case of having multiple fractional variables, ***learning effective variable selections*** is an influential leverage point (see Balcan et al. [2]). Furthermore, the commonly used ***branch-and-cut method*** can be regarded. At a node, the associated upper bound is improved by restraining the feasible set by inserting hyperplanes which do not cut off an optimal solution. Computing suitable cutting planes can be expensive and perhaps an ML model may yield good heuristic planes. In the context of knapsack this could for instance be realized by predicting unfixed items at a node $\mathrm{KP}_{I,\theta}$ which most likely are not in a subsequent optimal solution. In fact this is corresponds to (heuristically) skipping nodes along a trajectory and jumping to a subsequent subtree. Thus we would obtain a local version of excluding items in a preprocessing step as suggested above (i.e. starting at a subtree).

Applications of ML in other heuristics

Besides adapting the branch-and-bound framework by introducing heuristically learned rules, further variants of heuristic applications may be to take metaheuristics like ***simulated annealing*** as a basis and for instance guide a cooling sequence by an ML-model. When using ***tabu search*** one could model tabu durations and when using ***hill climbing algorithms*** one may model a good reentry point. More generally the ***selection of new centers in predefined neighborhoods*** in respective meta-heuristics may promise increased performance with respect to both computation cost and quality of the solution.

Bibliography

[1] E. Alpaydin. *Introduction to Machine Learning.* MIT Press, 3 edition, 2014.

[2] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. volume 80, pages 344–353. Proceedings of the 35th International Conference on Machine Learning, PMLR, 2018.

[3] A. Banerjee. An analysis of logistic models: Exponential family connections and online performance. Proceedings of the 7th SIAM International Conference on Data Mining, 2007.

[4] T. Bayes. LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S. *Philosophical Transactions of the Royal Society of London*, pages 370–418, 1764.

[5] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv*, 1811.06128 [cs.LG], 2018.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer, 2007.

[7] D. Chicco and G. Jurman. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(6), 2020.

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[9] P. K. Dunn and G. K. Smyth. *Generalized Linear Models With Examples in R.* Springer, 2018.

[10] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[11] P. Flach and M. Kull. *Precision-Recall-Gain Curves: PR Analysis Done Right.* Advances in Neural Information Processing Systems 28. Curran Associates, Inc., 2015.

[12] D. George B. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):266–288, 1957.

[13] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[14] H. He, H. Daume III, and J. M. Eisner. *Learning to Search in Branch and Bound Algorithms*. Advances in Neural Information Processing Systems 27. Curran Associates, Inc., 2014.

[15] J. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Master's thesis, Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Brauer, 1991.

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

[17] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[18] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[19] V. E. Ismailov. On the approximation by neural networks with bounded number of neurons in hidden layers. *Journal of Mathematical Analysis and Applications*, 417(2):963–969, 2014.

[20] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer, 2013.

[21] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 40:85–103, 1972.

[22] H. Kellerer, U. Pferschy, and P. David. *Knapsack Problems*. Springer, 2004.

[23] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[24] R. Kolpakov and M. Posypkin. Upper and lower bounds for the complexity of the branch and bound method for the knapsack problem. *Discrete Mathematics and Applications*, 20:95–112, 2010.

[25] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 5th edition, 2012.

[26] V. Maiorov and A. Pinkus. Lower Bounds for Approximation by MLP Neural Networks. *Neurocomputing*, 25:81–91, 1999.

[27] S. Martello, D. Pisinger, and P. Toth. Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem. *Management Science*, 45(3):414–424, 1999.

[28] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations.* John Wiley & Sons, Inc., 1990.

[29] D. Martini. Application of neural network for the knapsack problem. `http://tesi.cab.unipd.it/62965/1/Application_of_NN_for_the_KP.pdf`, 2019. [Online; accessed 17th October 2020].

[30] A. Neves, I. Gonzalez, J. Leander, and R. Karoumi. A new approach to damage detection in bridges using machine learning. *International Conference on Experimental Vibration Analysis for Civil Engineering Structures*, pages 73–84, 2018.

[31] T. Poggio and Q. Liao. Theory I: Deep networks and the curse of dimensionality. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, pages 761–773, 2018.

[32] A. Schäfer and H. Zimmermann. Recurrent neural networks are universal approximators. volume 4131 of *Proceedings of the 16th International Conference on Artificial Neural Networks*, pages 632–640. Springer, 2006.

[33] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. volume 2 of *NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems*, pages 2692–2700. MIT Press, 2015.

[34] A. Zheng and A. Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists.* O'Reilly Media, Inc., 1st edition, 2018.