ERASMUS UNIVERSITEIT ROTTERDAM

Erasmus School of Economics

Master Thesis Operations Research & Quantitative Logistics

Learning from K-adaptability to enhance partitioning in two-stage robust optimization problems

Author:
Esther Julien
Student ID:
546613

Supervisor:
Dr. K.S. Postek
Second Accessor:
Prof. dr. Ş.İ. Birbil

January 21, 2021

Abstract

Robust optimization is a field in optimization theory where problems affected by uncertainty are solved for the worst-case scenario. This solution is robust to all scenarios that can occur. Two-stage robust optimization problems are problems where we can separate the moment of making decisions in two stages. In the first stage, we do not know the uncertain scenario, but in the second stage we do. Therefore, the second stage can adapt to this scenario. We want to solve the problem for both the first- and second-stage at once. Such problems can be approximately solved with the K-adaptability algorithm, where the second stage can adapt in a total of K different ways to the scenarios. Here, the uncertainty set, a set that consists of all possible scenarios, is partitioned in K subsets. For each of these subsets, a corresponding second-stage decision is solved. The way an uncertainty set is partitioned greatly depends on the problem we are solving. The computational complexity for finding the optimal partition and its corresponding decisions is high. In this thesis, we wish to accelerate this algorithm by learning the structure of the partition made such that we can predict this. In a numerical experiment we show that by predicting the partition using neural networks, we quickly arrive at high quality solutions.

The content of this thesis is the sole responsibility of the author and does not reflect the view of the supervisor, second assessor, Erasmus School of Economics, or Erasmus University.

Acknowledgements

First of all, I would like to thank my supervisor Dr. (Krzysztof) Postek for his guidance and valuable feedback. Our many sparring sessions and discussions have led to where this thesis is today. I look forward to working on more similar projects together in my years at TU Delft. Prof. dr. (İlker) Birbil, thank you for being the co-reader of my thesis. I would also like to thank Ben Zwerink Arbonés, for his support, feedback, and being the friendly ear to my many thesis related monologues. I thank my family and friends for their support throughout this thesis and my master's program. I am particularly grateful for Rens Windeij, for helping me both on a theoretical and practical level in the parts of this thesis where machine learning methods were required.

Contents

1 | Introduction

Many problems in real life are subject to uncertainty. To find an optimal solution, it is important to account for this uncertainty. If we would not, the decisions made could result in worse solutions than anticipated. Moreover, the solution could become infeasible. We try to avoid this.

Robust optimization (Ben-Tal et al., 2009) is a method for dealing with uncertainty. Contrary to stochastic optimization (Schneider and Kirkpatrick, 2007; Charnes and Cooper, 1959), this method does not require a density over possible scenarios, merely a set containing these scenarios. This set is called the uncertainty set. Robust optimization solves problems for the worst-case scenario, such that the solution found is feasible for all other scenarios in the uncertainty set.

1.1 Two-stage robust optimization

Some problems have the characteristic that decisions have to be made both prior and subsequent to a realization of uncertainty. This is called two-stage robust optimization (Ben-Tal et al., 2004). The decisions that occur before the realization, are called the 'here-and-now', or the 'first-stage' decisions. The decisions that occur after the realization, are called the 'wait-and-see', or the 'second-stage' decisions. An example of this problem type is the facility location problem (Bertsimas and de Ruiter, 2016). Before the demand of customers is known, we decide on where to locate facilities. After the demand is known, facilities work together to satisfy the demand of the customers such that the costs are minimized.

We want to solve the problem for both the first- and second-stage decisions at once, before the realization has taken place. We solve the decisions of both stages at once since the decisions made in the first-stage, depend on the decisions that can be made in the second-stage. In order for both stages to obtain the optimal solution, the decisions of both stages have to be solved simultaneously. This two-stage robust optimization problem, where we use the notation of Subramanyam et al. (2019), is formulated as:

$$\min_{\boldsymbol{x}\in\mathcal{X}} \quad \max_{\boldsymbol{\xi}\in\Xi} \quad \min_{\boldsymbol{y}\in\mathcal{Y}} \left\{\boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y} : \boldsymbol{T}(\boldsymbol{\xi})\boldsymbol{x} + \boldsymbol{W}(\boldsymbol{\xi})\boldsymbol{y} \leq \boldsymbol{h}(\boldsymbol{\xi})\right\}, \tag{1.1}$$

where $\boldsymbol{x} \in \mathcal{X}$ are the first-stage decision variables, $\boldsymbol{y} \in \mathcal{Y}$ are the second-stage decision variables, and $\boldsymbol{\xi} \in \Xi$ are the uncertain scenarios. $\mathcal{X} \subseteq \mathbb{R}^{N_1}$, $\mathcal{Y} \subseteq \mathbb{R}^{N_2}$, and $\Xi \subseteq \mathbb{R}^{N_p}$ are non-empty and bounded mixed-integer linear programming (MILP) sets. These sets do not have to be convex. Furthermore, $\boldsymbol{c} \in \mathbb{R}^{N_1}$, and the functions $\boldsymbol{d} : \Xi \to \mathbb{R}^{N_2}$, $\boldsymbol{T} : \Xi \to \mathbb{R}^{L\times N_1}$, $\boldsymbol{W} : \Xi \to \mathbb{R}^{L\times N_2}$, and $\boldsymbol{h} : \Xi \to \mathbb{R}^{L}$ are affine. The two-stage robust optimization problem is illustrated in Figure 1.1.

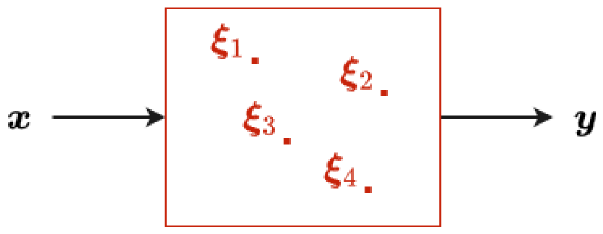


Figure 1.1: Two-stage robust optimization. The red square is the uncertainty set Ξ.

Ideally, the optimal solution of problem (1.1) is to assign to each scenario $\boldsymbol{\xi}$ an optimal second-stage decision $\boldsymbol{y}$. This results in a fully adaptive solution. However, since the uncertainty set is continuous, and therefore consists of infinitely many different scenarios, this problem becomes very large. Guslitzer (2002) proves that because of this issue, problem (1.1) is NP-hard.

Several methods to approximately solve problem (1.1) have been introduced (Ben-Tal et al., 2004; Bertsimas and Dunning, 2016; Postek and den Hertog, 2016). Some of these methods implicitly partition the uncertainty set Ξ into a number of subsets. The scenarios grouped together in a subset share the same robust second-stage decision. This is illustrated in Figure 1.2. By partitioning the uncertainty set, we restrict the second-stage decisions in being fully adaptive.

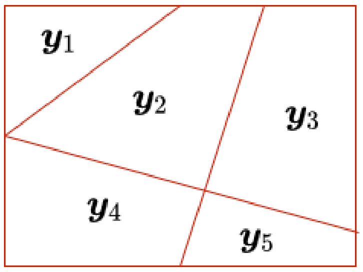


Figure 1.2: Partition of the uncertainty set Ξ. Each second-stage decision $\boldsymbol{y}_k$ is the decision of all scenarios in the k-th subset, for $k \in \{1, ..., 5\}$.

1.2 K-adaptability

In reality, one may want to restrict themselves to a fixed number of second-stage decisions for practical reasons. For instance, if the route planning problem (Hanasusanto et al., 2015) with uncertain path durations is solved for logistics companies, having many different decisions might be unfavorable for employees due to many possible last-minute changes in their schedules.

Bertsimas and Caramanis (2010) introduced the term of finite adaptability in two-stage robust optimization. This is also called K-adaptability, or K-partition, where the second-stage will adapt in a total of K different ways to the uncertainty set. The uncertainty set Ξ is then implicitly separated into K subsets, with each having a corresponding second-stage decision (Figure 1.2). The way the uncertainty set is partitioned, depends on the problem we are solving. The authors prove that making optimal partitions for $K \geq 2$ is NP-hard. The K-adaptability problem is formulated as follows:

$$\min_{\boldsymbol{x}\in\mathcal{X},\boldsymbol{y}\in\mathcal{Y}} \max_{\boldsymbol{\xi}\in\Xi} \min_{k\in\mathcal{K}} \left\{ \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y}_k : \boldsymbol{T}(\boldsymbol{\xi})\boldsymbol{x} + \boldsymbol{W}(\boldsymbol{\xi})\boldsymbol{y}_k \leq \boldsymbol{h}(\boldsymbol{\xi}) \right\}, \tag{1.2}$$

where $\mathcal{K} = \{1, ..., K\}$ and $\boldsymbol{y} = \{\boldsymbol{y}_1, ..., \boldsymbol{y}_K\}$. $\boldsymbol{y}_k$ is the second-stage decision when the realization of the uncertainty is in the k-th subset. The K-adaptability problem is the optimal solution to problem (1.1) if the value for K is large enough. Therefore, the optimal solution of problem (1.2) bounds the optimal solution of problem (1.1) from above.

Subramanyam et al. (2019) present a branch-and-bound algorithm that optimally solves the K-adaptability problem (1.2). This algorithm, compared to earlier presented approaches (Hanasusanto et al., 2015), can flexibly handle mixed continuous and integer first- and second-stage decisions, with or without deterministic $\boldsymbol{d}, \boldsymbol{T}$, and $\boldsymbol{W}$, for a fixed K.

1.3 Thesis goals

As mentioned before, Subramanyam et al. (2019) present an algorithm to solve the K-adaptability problem exactly. Due to the branch-and-bound structure of the algorithm, many mixed integer linear programming (MILP) problems have to be solved. This is computationally expensive. However, since this is the only algorithm that optimally solves (1.2) for a variety of different two-stage robust optimization problems, this algorithm is used as the starting point of this thesis. The K-adaptability algorithm will be described in detail in Section 4.1. In this thesis, this branch-and-bound algorithm will be referred to as K-B&B.

Research objective K-B&B is the current method to solve problem (1.2) exactly. However, this algorithm is computationally expensive. In this thesis, we propose an algorithm that first predicts the optimal partition of the uncertainty set using a neural network. This neural network is trained with the optimal solutions of the K-B&B algorithm. After the prediction of the partition, the algorithm will solve the first- and second-stage decisions that correspond to this predicted partition. This algorithm is called the K-adaptability partition prediction (K-PP) algorithm.

The following research questions are formulated for a better understanding of the problems we want to solve and focus on in this thesis:

1. How can we predict the partition of the uncertainty set of a two-stage robust optimization problem?
2. How will the objective and decision variables computed with K-PP differ from the ones computed with K-B&B?
3. How can the results of K-PP be used as initialization of K-B&B to derive exact solutions faster?
4. Which parameters of the problem are important for building the partitions?

The answers to these research questions will help create a better picture of the field of two- and multi-stage robust optimization from a theoretical point of view. From a practical point of view, there are many real-life problems that can be formulated as a two-stage robust optimization problem. Therefore, this research will benefit decision makers in various fields.

General approach This thesis will predominantly devote itself to answering research question 1. Our approach for answering this, is to use a neural network for predicting the partition of the uncertainty set of a two-stage robust optimization problem. The input of the neural network contains the only information available to us before we solve the K-adaptability problem, which is the collection of problem parameters of the two-stage robust optimization problem. The output of this neural network is the partitioned uncertainty set for which $\boldsymbol{x}$, and $\boldsymbol{y}_k$ for all $k \in \mathcal{K}$ are optimal. The dimension of the input and the output of this model have to be equal for different instances of two-stage robust optimization problems. If we would predict and train problems that differ in instance size, the dimension of the input and output per instance would differ. Therefore, we have chosen to study one problem, with one instance size, representative to two-stage robust optimization problems. This is the capital budgeting problem with loans (Subramanyam et al., 2019). This problem is representative due to the uncertainty element in both the objective function and the constraints.

For training the neural network, we will solve K-B&B for many instances. After analyzing the solutions of K-PP, we can answer research question 2.

Since the partition derived by the neural network is a prediction, the results of K-PP are not exact. However, the result of K-PP gives an upper bound for the K-adaptability problem. This upper bound can be used to initialize the K-B&B algorithm to run to optimality faster. In this thesis, we will explain how this initialization can be done in more detail, such that research question 3 can be answered.

The larger the instance of the problem, the larger the input of the model will be, which results in more neural network parameters. This will increase the training time. An attempt for limiting the input of the model is done by using a method to detect the importance of features of the neural network. Using this technique, we try to answer research question 4.

1.4 Thesis outline

This thesis combines two-stage robust optimization theory with neural network prediction. Chapter 2 lists literature of these research fields. In Chapter 3, the capital budgeting problem is defined. In Chapter 4, the K-PP algorithm is presented. In Chapter 5, the performance of this algorithm is studied on out-of-sample instances.

1.5 Notation

In this thesis, all lowercase characters are scalars, e.g. k. (Curly) capitals are used for sets, e.g. $\mathcal{K}$. Bold lowercase characters are vectors, e.g. $\boldsymbol{x}$. Bold capitals are used for matrices, e.g. $\boldsymbol{W}$. Moreover, $\boldsymbol{x}$ and $\boldsymbol{y}$ are used to describe the first- and second-stage decision variables in two-stage robust optimization problems, respectively. For neural networks, $\boldsymbol{x}_{NN}$ and $\boldsymbol{y}_{NN}$ are used to describe the input and output of this model, respectively.

2 | Literature review

This thesis combines both two-stage robust optimization theory and machine learning. Section 2.1 of this chapter contains literature needed for better understanding of the field of two-stage robust optimization. In Section 2.2, literature on neural networks is listed. Here we focus on methods that discover the importance that elements of the input in the neural network have on the output.

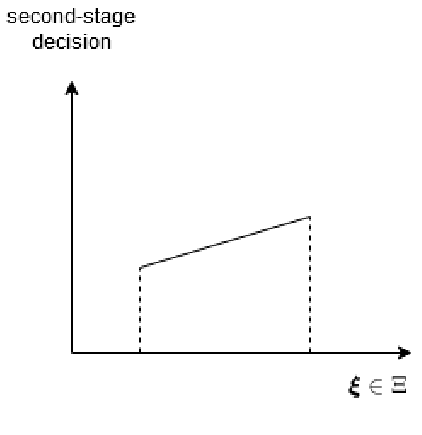
2.1 Two-stage robust optimization

As mentioned in the introduction, two-stage robust optimization problems are NP-hard. These problems are solved exactly if all scenarios correspond to their own optimal second-stage decision $\boldsymbol{y}$, to make the solution fully adaptive. For two-stage robust optimization problems with continuous two-stage second-stage decisions, Ben-Tal et al. (2004) introduce a simple method to approximately solve this problem such that the second-stage decisions can adapt uniquely to each scenario. This is done with the affine decision rule for $\boldsymbol{y}$, a function of scenario $\boldsymbol{\xi} \in \Xi$. This approximation of the second-stage decisions is described as:

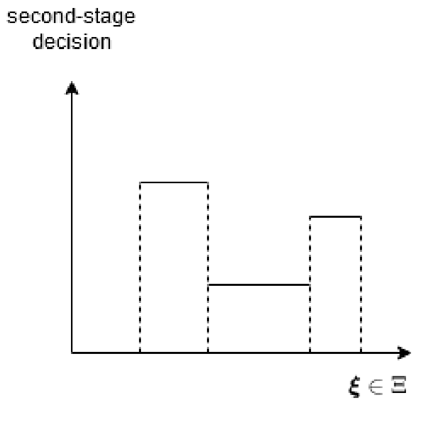
$$\boldsymbol{y}(\boldsymbol{\xi}) := \boldsymbol{z} + \boldsymbol{Z}\boldsymbol{\xi}, \tag{2.1}$$

where $\boldsymbol{z}$ and $\boldsymbol{Z}$ are simultaneously optimized together with the first-stage decision variable $\boldsymbol{x}$. No theorem has been found for establishing the optimality gap between solving problem (1.1) with this affine decision rule, or exactly. However, it is "experimentally proven" that it works very well for inventory problems (Ben-Tal et al., 2009).

Another method for solving problem (1.1) is to divide the uncertainty set into subsets. Each subset corresponds to an optimal robust second-stage decision. Hence, the second-stage decisions are not continuous, but piecewise constant as illustrated in Figure 2.1.



(a) Affine decision rule - linear



(b) Partitioning - piecewise constant

Figure 2.1: Difference between linear and piecewise constant decision rules.

Postek and den Hertog (2016) define an algorithm to split the uncertainty set using hyperplanes. The locations of these splits are based on duality information of the underlying problem.

The key is that so-called “active” scenarios need to be separated. The algorithm knows three stopping criteria: when the objective value is close to the lower bound, when the limit of total computational time is reached, or when the maximum number of splitting rounds is reached. Each stopping criterion is possible since the algorithm provides us with a robust solution after every iteration. In this thesis, we focus on the case where the number of subsets is fixed a priori, as is the case in Hanasusanto et al. (2015) and Subramanyam et al. (2019).

This thesis tries to approximately solve two-stage robust optimization problems (1.1) with the problem formulation of the K-adaptability problem (1.2). Hence, instead of obtaining a fully adaptive second-stage decision, we let this decision adapt in a total of K ways, hence finite. In this section we will discuss some theory on how well the finite adaptive approximation actually is compared to the fully adaptive solution.

Bertsimas et al. (2011) describe the conditions of the uncertainty set for which solving multi-stage optimization problems using finite adaptability give good approximations. Moreover, for the case where only the right-hand side of the constraints is uncertain, the cost of solving finitely adaptive (compared to fully adaptive) is lower than for the case the objective and the right-hand side of the constraints are uncertain. Hanasusanto et al. (2015) give a lower bound of K for a guarantee that K-adaptable two-stage binary robust optimization problems attain the same solution as solving the fully adaptable problem exact. For problems with uncertainty in the constraints, the authors claim that the K-adaptability problem may attain a strictly higher optimal value than solving this problem fully adaptive, for any number K smaller than the number of possible second-stage decisions. Note that this result holds for binary solutions, and therefore K does not have to be infinite. However, this result is still bad.

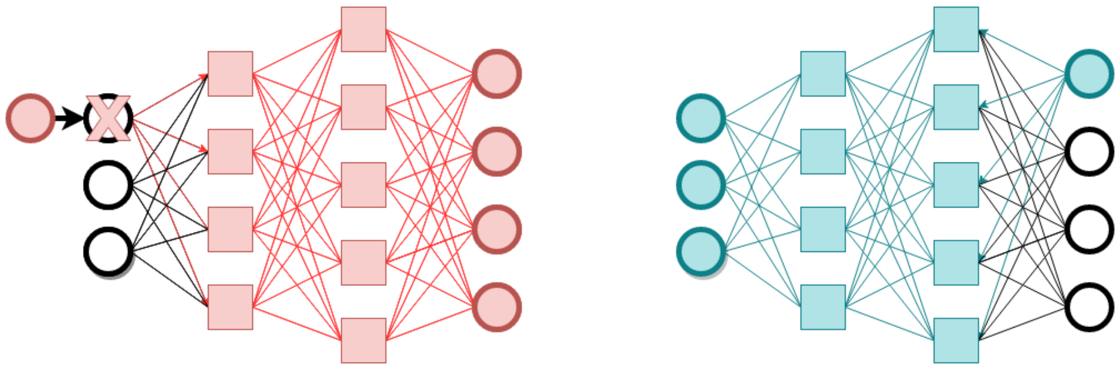
2.2 Neural network

In this thesis, we want to predict the optimal partition of the uncertainty set of a two-stage robust optimization problem. The model used for prediction is a neural network, to be more specific, a feedforward neural network. Feedforward neural networks are the first and simplest type of neural networks devised already in the 1960s (Ivakhnenko, 1968; Schmidhuber, 2015). In Section 4.4.1, a detailed explanation of this method is given.

Additionally, we want to obtain the importance of the features of this model, to detect which ones are important for prediction, and which ones can be ignored. There are multiple feature importance measures. We will group these into perturbation methods and gradient based methods.

Approaches that use perturbation change the input of the trained neural network and analyze what happens to the output and the later nodes as a result of this perturbation. There are different types of perturbations possible. Fisher et al. (2019) propose a permutation method based on the feature importance method introduced by Breiman (2001) for random forests. The authors switch the data of a feature from one observation with another observation. After switching, the expected loss of the model is calculated. The authors give an alternative to switching observations, i.e. add noise to the feature of interest such that it becomes completely uninformative to the output. However, here we need to be aware that the distribution of the feature remains the same. This alternative method is very similar to the "Perturb" method introduced in Gevrey et al. (2003). This algorithm changes the input values of one feature at a time while keeping the others equal to the original data.

Gradient based approaches propagate backwards instead of forwards. Figure 2.2 describes this process. Such methods give importance scores to all underlying nodes based on the output. Therefore, this method is more efficient than using perturbation since one backward pass assigns importance to all features. For perturbation of one feature, only importance to that feature is assigned (Shrikumar et al., 2017). Moreover, gradient based approaches represent pairwise interactions of features more clearly than perturbation methods.



(a) Perturbation methods **(b)** Gradient methods

Figure 2.2: Feature importance of a neural network with 3 features, 2 hidden layers, and 4 outputs. Subfigure 2.2a shows which nodes are affected by forward propagation perturbation methods. Subfigure 2.2b shows which nodes are affected by the backward propagation gradient methods.

The importance scores are based on the gradient of the output with respect to the underlying node. For convolution neural networks, widely used for image recognition, gradients are used in so called saliency maps that show the importance of pixels (Simonyan et al., 2013). Alternatives to the saliency map that also use gradients, are the deconvolution network introduced by Zeiler and Fergus (2014) and the guided back-propagation method introduced by Springenberg et al. (2014).

Shrikumar et al. (2017) introduce an algorithm called DeepLIFT that also uses gradients. This algorithm distinguishes itself from other methods by using certain “reference” points. The algorithm explains the difference from the reference value of the output in term of the difference from the reference value of the inputs. These reference points depend on the underlying problem. Reference points can be seen as default or neutral points. For image recognition problems, the reference point of the input would be all zeros since this represents a pixel with no color, or the background of the image.

In neural networks, (non-)linear activation functions on the nodes are used. A commonly used activation function is the rectified linear unit (ReLU), equal to max(0, *input*). The saliency map, deconvolution network, and guided back-propagation all have the disadvantage that for negative inputs, the importance signal of a node becomes zero when this node is activated with ReLU. Hence, most of the time only positive contributions to the output are highlighted. Moreover, only DeepLIFT can explain a strict pairwise interaction from features to the output, or as Shrikumar et al. (2017) likes to call it, saturation of multiple features.

3 | Problem description

This thesis can be considered as preliminary research in the field of enhancing adaptive robust optimization (ARO) with machine learning. Therefore, we are starting with a problem that has the property of having a clear overview of the structure of the partitions.

3.1 Problem selection

Multiple two-stage robust optimization problems were considered to be studied in this thesis. Frequently applied problems in the ARO field are the facility location problem, the lot-sizing on a network problem (Bertsimas and de Ruiter, 2016), and the route planning problem (Hanasusanto et al., 2015). However, for all of these problems, the dimension of the uncertainty set depends on the instance size. If we want to study the partitions of the uncertainty set, a high dimensional uncertainty set is not able to give a clear overview of its structure. On the other hand, a low instance size would result in an unrealistic or too simple problem. For example, a route planning problem with only two roads, or a facility location problem with two locations would not be an interesting problem to solve with two-stage robust optimization solving methods.

Therefore, we wish to have a problem for which its instance size is independent of the uncertainty set dimension. In this way, we can force the dimension of the uncertainty set to be small while still having a realistic instance size. Thus, the problem does not become too simple and we have a clear overview of the structure of the partitioned uncertainty set. The capital budgeting problem with loans (Subramanyam et al., 2019) satisfies these demands. This problem will be introduced in Section 3.2. An additional interesting feature of this problem is the uncertainty element in both the objective function and in the constraints. It also has both integer and continuous first- and second-stage decision variables. These features make it a representative problem of two-stage robust optimization problems.

3.2 Capital budgeting with loans

We consider the capital budgeting with loans problem as defined in Subramanyam et al. (2019), where a company wishes to invest in a subset of N projects. Each project i has an uncertain cost $c_i(\boldsymbol{\xi})$ and an uncertain profit $r_i(\boldsymbol{\xi})$, defined as:

$$c_i(\boldsymbol{\xi}) = \big(1 + \boldsymbol{\Phi}_i^\intercal \boldsymbol{\xi}/2\big)c_i^0 \quad \text{and} \quad r_i(\boldsymbol{\xi}) = \big(1 + \boldsymbol{\Psi}_i^\intercal \boldsymbol{\xi}/2\big)r_i^0 \quad \forall i \in \{1, ..., N\},$$

where c_i^0 and r_i^0 represent the nominal cost and the nominal profit of project i, respectively. $\boldsymbol{\Phi}_i^\intercal$ and $\boldsymbol{\Psi}_i^\intercal$ represent the i^{th} row vectors of the sensitivity matrices $\boldsymbol{\Phi}, \boldsymbol{\Psi} \in \mathbb{R}^{N \times N_p}$. The realizations of the uncertain vector $\boldsymbol{\xi}$ belong to the uncertainty set $\Xi = [-1, 1]^{N_p}$, where N_p is the dimension of the uncertainty set. Originally, in Subramanyam et al. (2019), $N_p = 4$. However, to obtain the clear overview wished for, we choose $N_p = 2$.

The company can invest in a project either before or after observing the risk factor $\boldsymbol{\xi}$. In the latter case, the company generates only a fraction κ of the profit, which reflects to a penalty for postponement. However, the cost remains the same as in the case of an early investment. The company has a given budget B, which the company can increase by loaning from the bank at a unit cost of $\lambda > 0$, before the risk factors $\boldsymbol{\xi}$ are observed. A loan after the observation occurs has a unit cost of $\mu\lambda$, with $\mu > 1$. The objective of the capital budgeting problem is to maximize the total revenue subject to the budget.

K-adaptability formulation The capital budgeting problem with loans can be formulated as an instance of the K-adaptability problem (1.2) as follows:

$$\begin{aligned}
\max_{(x_0,\boldsymbol{x})\in\mathcal{X},(y_0,\boldsymbol{y})\in\mathcal{Y}} \ \min_{\boldsymbol{\xi}\in\Xi} \ \max_{k\in\mathcal{K}} \quad & \boldsymbol{r}(\boldsymbol{\xi})^\intercal \boldsymbol{x} - \lambda x_0 + \pi \\
\text{s.t.} \quad & \kappa \boldsymbol{r}(\boldsymbol{\xi})^\intercal \boldsymbol{y}_k - \lambda\mu y_0^k \geq \pi && \forall \boldsymbol{\xi} \in \Xi_k, \forall k \in \mathcal{K} \\
& \boldsymbol{x} + \boldsymbol{y}_k \leq \boldsymbol{e} && \forall \boldsymbol{\xi} \in \Xi_k, \forall k \in \mathcal{K} \\
& \boldsymbol{c}(\boldsymbol{\xi})^\intercal \boldsymbol{x} \leq B + x_0 && \forall \boldsymbol{\xi} \in \Xi_k, \forall k \in \mathcal{K} \\
& \boldsymbol{c}(\boldsymbol{\xi})^\intercal (\boldsymbol{x} + \boldsymbol{y}_k) \leq B + x_0 + y_0^k && \forall \boldsymbol{\xi} \in \Xi_k, \forall k \in \mathcal{K},
\end{aligned}$$

where $\mathcal{X} = \mathcal{Y} = \mathbb{R}_+ \times \{0,1\}^N$, $y_0 = \{y_0^1, ..., y_0^K\}$, and $\boldsymbol{y} = \{\boldsymbol{y}_1, ..., \boldsymbol{y}_K\}$. x_0 and y_0 are the loan amounts in the first- and second-stage, respectively. x_i and y_i are the binary variables that indicate whether we invest in the i-th project in the first- and second-stage, respectively. The constraint $\boldsymbol{c}(\boldsymbol{\xi})^\intercal \boldsymbol{x} \leq B + x_0$ ensures that for the first-stage, the expenditures are not more than the budget plus the loan taken before the realization of uncertainty.

Test case We now select the instance of the capital budgeting problem used in this thesis. To limit the total runtime needed for computing the observations of this problem, we choose $N = 10$. The nominal cost vector $\boldsymbol{c}^0$ is chosen uniformly at random from the set $[0, 10]^N$. Let $\boldsymbol{r}^0 = \boldsymbol{c}^0/5$, $B = \boldsymbol{e}^\intercal \boldsymbol{c}^0/2$, and $\kappa = 0.8$. the rows of the sensitivity matrices $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$ are sampled uniformly from the i-th row vector, which is sampled from $[0, 1]^{N_p}$, such that $\boldsymbol{\Phi}_i^\intercal \boldsymbol{e} = \boldsymbol{\Psi}_i^\intercal \boldsymbol{e} = 1$ for all $i \in \{1, ..., N\}$. This is also known as the unit simplex in $\mathbb{R}^{N_p}$. For determining the cost of the loans, we set $\lambda = 0.12$ and $\mu = 1.2$.

4 | Methodology

In this chapter, the methods used in the K-adaptability partition prediction (K-PP) algorithm are defined. In Section 4.1, we begin with explaining the K-adaptability branch-and-bound (K-B&B) algorithm since this is the starting point of K-PP. Then in Section 4.2, the basic steps of the K-PP algorithm will be introduced. In Section 4.3, we explain how we transform the input and the output of K-B&B such that it can be used in the neural network used in K-PP. In Section 4.4, we present this neural network and explain the method we use for detecting feature importance. Finally, in Section 4.5, we introduce the method used for using the imperfect results of K-PP as initial solutions of K-B&B to obtain optimal solution faster.

4.1 K-adaptability branch-and-bound algorithm (K-B&B)

In this thesis we want to solve the K-adaptability problem (1.2) with an alternative approach. As mentioned in the introduction, solving this problem entails that instead of giving a fully adaptive solution to the two-stage robust optimization problem (1.1), we give a finitely adaptive one. With K-adaptability, K different second-stage decisions are solved to adapt in K different ways to the uncertainty set Ξ. Therefore, Ξ will be implicitly partitioned into K subsets, such that for all the scenarios in subset k we have the corresponding second-stage decisions $\boldsymbol{y}_k$, for all $k \in \mathcal{K}$. This is illustrated in Figure 1.2.

Subramanyam et al. (2019) present a branch-and-bound algorithm (K-B&B) to solve the K-adaptability problem (1.2). This algorithm explicitly partitions the uncertainty set Ξ in K subsets $\Xi_1, ..., \Xi_K$. It does this by iteratively filling K initially empty subsets with scenarios $\boldsymbol{\xi} \in \Xi$. Each scenario found in an iteration of the algorithm can be added to any of the K subsets. To find the optimal allocation of found scenarios, the algorithm is structured as a branch-and-bound tree. This is illustrated in Figure 4.1.

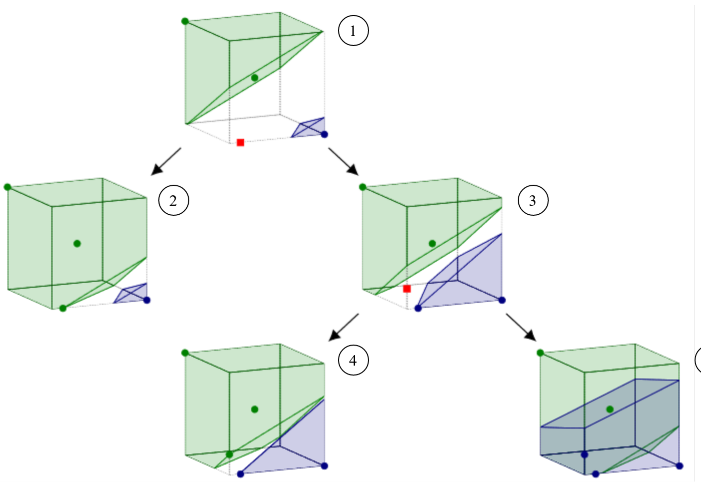


Figure 4.1: Branch-and-bound tree with $K = 2$ and $\Xi \subset \mathbb{R}^3$. The leaf nodes of the tree are numbered. **Source**: Subramanyam et al. (2019)

In this illustration we have a branch-and-bound tree for $K = 2$ and the uncertainty set

$\Xi \in \mathbb{R}^3$. The red dot is the newly found scenario. Then, in its two branches, the new scenario is added to each subset. The green and blue regions represent the subset Ξ_1 and Ξ_2 where for all the scenarios in these subsets, $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ are the second-stage decision variables, respectively. We distinguish the actual uncertainty set Ξ, and its subsets $\Xi_1, ..., \Xi_K$, with the subsets $\dot{\Xi}_1, ..., \dot{\Xi}_K$ found in the K-B&B algorithm. The actual uncertainty set and subsets consist of a continuous number of scenarios. The subsets found in K-B&B consist of a finite amount of scenarios. How we find the new scenario to be added to the subsets, and what the bounding criteria in the branch-and-bound tree are, will be explained via two problem formulations. First, for each leaf node, the two-stage robust optimization problem is solved based on the partition of that node. This problem is called the scenario-based K-adaptability problem.

Scenario-based K-adaptability problem This problem solves the underlying K-adaptability problem (1.2) with respect to the currently found scenarios $\boldsymbol{\xi}$ stored in the subsets $\dot{\Xi}_k$ for all $k \in \mathcal{K}$. Thus, for a collection $\dot{\Xi}_1, ..., \dot{\Xi}_K$ of a finite number of scenarios of the uncertainty set Ξ, the scenario-based K-adaptability problem is defined as follows:

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{y},\theta} \quad & \theta \\ \text{s.t.} \quad & \theta \in \mathbb{R}, \boldsymbol{x} \in \mathcal{X}, \boldsymbol{y} \in \mathcal{Y} \\ & \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} \leq \theta & \forall \boldsymbol{\xi} \in \dot{\Xi}_k, \forall k \in \mathcal{K} \\ & \boldsymbol{T}(\boldsymbol{\xi})\boldsymbol{x} + \boldsymbol{W}(\boldsymbol{\xi})\boldsymbol{y_k} \leq \boldsymbol{h}(\boldsymbol{\xi}) & \forall \boldsymbol{\xi} \in \dot{\Xi}_k, \forall k \in \mathcal{K}, \end{aligned} \tag{4.1}$$

where $\boldsymbol{y} = \{\boldsymbol{y}_1, ..., \boldsymbol{y}_K\}$.

The objective value θ and the first- and second-stage decisions $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, of the K-adaptability problem with respect to the partition of the current leaf node are now known. It might be the case that this solution is not feasible for a scenario not included in any of the subsets. The goal is to find this scenario and add it to any of the subsets. This is the red dot in node 1 and 3 in Figure 4.1. The scenario is chosen such that if we solve the scenario-based K-adaptability problem again, this new solution will be feasible for the found scenario. This scenario will be the solution to the separation problem.

Separation problem This problem tries to find the scenario for which the current solution $(\theta, \boldsymbol{x}, \boldsymbol{y})$ is most violated. For $(\theta, \boldsymbol{x}, \boldsymbol{y})$ found in problem (4.1), the separation problem is defined as follows:

$$\begin{aligned} \mathcal{S}(\theta, \boldsymbol{x}, \boldsymbol{y}) &= \max_{\boldsymbol{\xi} \in \Xi} S(\theta, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\xi}), \quad \text{where} \\ S(\theta, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\xi}) &= \min_{k \in \mathcal{K}} \max \left\{ \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - \theta, \max_{l \in \{1,...,L\}} \{ \boldsymbol{t_l}(\boldsymbol{\xi})^\intercal \boldsymbol{x} + \boldsymbol{w_l}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - h_l(\boldsymbol{\xi}) \} \right\}, \end{aligned} \tag{4.2}$$

This problem is equivalent to the MILP:

$$\begin{aligned} \max \quad & \zeta & & (4.3) \\ \text{s.t.} \quad & \zeta \in \mathbb{R}, \quad \boldsymbol{\xi} \in \Xi, \quad z_{kl} \in \{0,1\}, \quad (k,l) \in \mathcal{K} \times \{0,1,...,L\} & & (4.4) \\ & \sum_{l=0}^{L} z_{kl} = 1 & \forall k \in \mathcal{K} \quad & (4.5) \\ & z_{k0} = 1 \quad \Rightarrow \quad \zeta \leq \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - \theta & \forall k \in \mathcal{K} \quad & (4.6) \\ & z_{kl} = 1 \quad \Rightarrow \quad \zeta \leq \boldsymbol{t_l}(\boldsymbol{\xi})^\intercal \boldsymbol{x} + \boldsymbol{w_l}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - h_l(\boldsymbol{\xi}) & \forall l \in \{1,...,L\}, \forall k \in \mathcal{K} \quad & (4.7) \end{aligned}$$

The linear form of constraints (4.6) and (4.7) is formulated as follows:

$$\zeta + M z_{k0} \leq \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - \theta + M \qquad \forall k \in \mathcal{K} \tag{4.8}$$

$$\zeta + M z_{kl} \leq \boldsymbol{t_l}(\boldsymbol{\xi})^\intercal \boldsymbol{x} + \boldsymbol{w_l}(\boldsymbol{\xi})^\intercal \boldsymbol{y_k} - h_l(\boldsymbol{\xi}) + M \qquad \forall l \in \{1, ..., L\}, \forall k \in \mathcal{K}, \tag{4.9}$$

where M is a large number, which in practice is equal to the maximum of the upper bound of the right-hand side of (4.6) and the upper bound of the right-hand side of (4.7).

If $\zeta > 0$, we have found a scenario $\boldsymbol{\xi}$ such that the current solution $(\theta, \boldsymbol{x}, \boldsymbol{y})$ is violated, hence infeasible. We add K new branches to the tree in which the new scenario is added to $\dot{\Xi}_k$ in the k-th new branch for $k \in \mathcal{K}$. These new branches form for example nodes 2 and 3 in Figure 4.1. If $\zeta \leq 0$, no scenario is found for which the current solution is violated. Hence, no new scenario has to be added and the current partition is fully robust. This partition will be saved and $(\theta, \boldsymbol{x}, \boldsymbol{y})$ is the incumbent solution. In Figure 4.1, node 5 is the first incumbent solution found. In this node, for all scenarios $\boldsymbol{\xi}$ in the uncertainty set Ξ, any of the currently found second-stage decisions $\boldsymbol{y}_k$ for all $k \in \mathcal{K}$ are feasible. This is one way for bounding the tree. Another way for bounding is to compare the objective value θ of a leaf node with the one of the incumbent solution. If the objective of the current leaf node is worse (i.e. higher), we know we can bound this branch since adding more scenarios, resulting in adding more constraints to the MILP of the scenario-based K-adaptability problem, will only give a worse objective. This bounding method results in being able to skip solving the separation problem.

Now we will present the K-B&B algorithm. The branching and bounding techniques, scenario-based K-adaptability problem, and separation problem defined before are used in this formulation.

K-B&B algorithm

1. *Initialization:* Set the node set $\mathcal{N} \leftarrow \{\tau^0\}$, where $\tau^0 = (\dot{\Xi}_1^0, ..., \dot{\Xi}_K^0)$ with $\dot{\Xi}_k^0 = \emptyset$ for all $k \in \mathcal{K}$ is the root node. Set the incumbent solution to $(\theta^i, \boldsymbol{x}^i, \boldsymbol{y}^i) \leftarrow (+\infty, \emptyset, \emptyset)$.

2. *Check convergence:* If $\mathcal{N} = \emptyset$, then stop and declare infeasibility if $(\theta^i = +\infty)$, or report $(\boldsymbol{x}^i, \boldsymbol{y}^i)$ as an optimal solution to problem (1.2).

3. *Select node:* Select a node $\tau = (\dot{\Xi}_1, ..., \dot{\Xi}_K)$ from $\mathcal{N}$. Set $\mathcal{N} \leftarrow \mathcal{N} \setminus \{\tau\}$.

4. *Process node:* Let $(\theta, \boldsymbol{x}, \boldsymbol{y})$ be an optimal solution to the scenario-based K-adaptability problem (4.1). If $\theta \geq \theta^i$, go to Step 2.

5. *Check feasibility:* Let $(\zeta, \boldsymbol{\xi}, z)$ be an optimal solution to the separation problem (4.3). If $\zeta \leq 0$, then set $(\theta^i, \boldsymbol{x}^i, \boldsymbol{y}^i) \leftarrow (\theta, \boldsymbol{x}, \boldsymbol{y})$ and go to Step 2.

6. *Branch:* Make K new nodes $\tau_1, ..., \tau_K$, where $\tau_k = (\dot{\Xi}_1, ..., \dot{\Xi}_k \cup \{\boldsymbol{\xi}\}, ..., \dot{\Xi}_K)$ for each $k \in \mathcal{K}$. Set $\mathcal{N} \leftarrow \mathcal{N} \cup \{\tau_1, ..., \tau_K\}$ and go to Step 3.

Subramanyam et al. (2019) propose an improvement to the algorithm since problems (4.1) and (4.3) are identical across many nodes. Therefore, symmetry reducing methods are used in Step 6. Note that because of Step 5 in the algorithm, all incumbent solutions are robust. Hence, stopping the algorithm before completion will still give a robust solution. Moreover, in Step 6 the branching method becomes clear, i.e. breadth-first. Other branching techniques, such as depth-first, could be considered for faster convergence.

4.2 K-adaptability partition prediction algorithm (K-PP)

Solving K-B&B exactly can take a very long time. For the capital budgeting problem, with instance size $N = 10$, and four subsets ($K = 4$), running until completion takes more than 12 hours for many instances. The majority of this time is caused by solving many unnecessary MILPs in the branch-and-bound tree in the search for the optimal partition. In this section we present the K-adaptability partition prediction algorithm (K-PP). This method first predicts the partitioned uncertainty set with a neural network. Based on this prediction, the first- and second-stage decisions are solved. Predicting the partition using a neural network can significantly improve the speed of solving the K-adaptability problem. Because now instead of solving many MILPs, we only have to solve one.

In the K-PP algorithm, we first predict the partitioned uncertainty set. By prediction, we obtain the imperfect subsets $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$. The neural network used for predicting these imperfect subsets is trained based on the K-B&B subsets $\dot{\Xi}_1, ..., \dot{\Xi}_K$ of many instances. How this is done, will be explained in Sections 4.3 and 4.4.

After predicting the partition, we will solve the scenario-based K-adaptability problem (4.1) based on the predicted partition. Like in K-B&B, we have to add scenarios to the subsets $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$. To limit the computation time, we want to select as few as possible scenarios for each subset, such that we do not need many constraints in our MILP.

We noticed after solving many instances of the capital budgeting problem, that the subsets of the partitioned uncertainty set are convex, compact, and nonempty. Figure 4.2 illustrates the standard form of the partitioned uncertainty set of this problem.

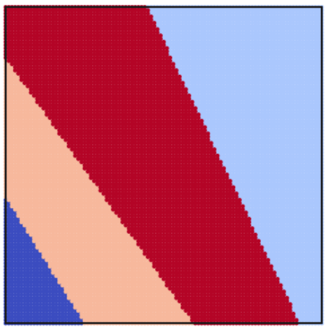


Figure 4.2: Example of a partitioned uncertainty set of the capital budgeting problem with loans with $K = 4$.

We want to leverage the convexity assumption such that we only use the extreme points of each subset $\tilde{\Xi}_k, \forall k \in \mathcal{K}$, for building the constraints of the scenario-based K-adaptability problem. An extreme point $\boldsymbol{\xi}_1$ in subset $\tilde{\Xi}_k$ for any $k \in \mathcal{K}$ has the following property:

$$\forall \boldsymbol{\xi}_2 \in \tilde{\Xi}_k, \nexists \boldsymbol{\xi}_3 \in \tilde{\Xi}_k, \boldsymbol{\xi}_2 \neq \boldsymbol{\xi}_3 : \boldsymbol{\xi}_1 = \gamma \boldsymbol{\xi}_2 + (1-\gamma)\boldsymbol{\xi}_3, 0 < \gamma < 1,$$

where $\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \boldsymbol{\xi}_3$ are different scenarios in subset $\tilde{\Xi}_k$. Moreover, we have that if the uncertainty subset Ξ_k for all $k \in \mathcal{K}$ is convex, compact, and nonempty, the following holds:

$$\left.\begin{array}{l}\boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y}_k \leq \theta \\ \boldsymbol{T}(\boldsymbol{\xi})\boldsymbol{x} + \boldsymbol{W}(\boldsymbol{\xi})\boldsymbol{y}_k \leq \boldsymbol{h}(\boldsymbol{\xi})\end{array}\right\} \quad \forall \boldsymbol{\xi} \in \tilde{\Xi}_k, \forall k \in \mathcal{K}$$

$$\Longleftrightarrow$$

$$\left.\begin{array}{l}\boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi})^\intercal \boldsymbol{y}_k \leq \theta \\ \boldsymbol{T}(\boldsymbol{\xi})\boldsymbol{x} + \boldsymbol{W}(\boldsymbol{\xi})\boldsymbol{y}_k \leq \boldsymbol{h}(\boldsymbol{\xi})\end{array}\right\} \quad \forall \boldsymbol{\xi} \in \text{ext}(\tilde{\Xi}_k), \forall k \in \mathcal{K},$$

where $\text{ext}(\tilde{\Xi}_k)$ is the set of extreme points of $\tilde{\Xi}_k$. Therefore, in the capital budgeting problem only the extreme points of these subsets are needed to solve the K-adaptability problem. We now present the K-adaptability partition prediction (K-PP) algorithm.

K-PP algorithm

1. *Predict partition:* Predict the partitioned uncertainty set using a neural network with the problem parameters of the capital budgeting problem as inputs.

2. *Correct partitions:* It can happen that the prediction results in a partition with more subsets than intended. Check if this occurs in the partition prediction. If there are no such errors, go to Step 3. Otherwise, correct the partition and go to Step 3. The method used for correcting depends on the approach used for predicting the partition. This will both be explained in Section 4.3.2.

3. *Find extreme points:* For each subset only include the extreme points of these subsets. We have $\bar{\Xi}_k = \text{ext}(\tilde{\Xi}_k)$ for all $k \in \mathcal{K}$.

4. *Solve K-adaptability problem:* Obtain $(\tilde{\theta}, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$ by solving the scenario-based K-adaptability problem (4.1) based on the subsets $\bar{\Xi}_1, ..., \bar{\Xi}_K$ derived in Step 3.

Remark This algorithm can also be used for solving other two-stage robust optimization problems. If the subsets of the partitioned uncertainty set for these problems are not convex, Steps 3 and 4 should be replaced by the following steps:

3'. *Select scenarios:* Initialize $\{\bar{\Xi}_1^0, ..., \bar{\Xi}_K^0\}$, where for each initial subset $\bar{\Xi}_k^0, k \in \mathcal{K}$ a (random) set of scenarios that belongs to the k-th predicted subset $\tilde{\Xi}_k$ is included.

4'. *Solve K-adaptability problem:* Obtain $(\tilde{\theta}, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$ by solving the scenario-based K-adaptability problem (4.1).

5'. *Check robustness:* Let $(\zeta, \boldsymbol{\xi}, z)$ be the optimal solution to the separation problem (4.3). If $\zeta \leq 0$, then stop and return the solution obtained in Step 4'. Otherwise, check to which subset $\tilde{\Xi}_k$ the newly found scenario $\boldsymbol{\xi}$ belongs to, and add this scenario to $\bar{\Xi}_k$. Then go to Step 4'.

4.3 Prediction model setup

We use a neural network to map the features of a two-stage robust optimization problem to the uncertainty set subsets $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$. In Section 4.3.1, we explain how we transform the information of a given two-stage robust optimization problem into these features. In Section 4.3.2, we introduce the methods we use for transforming the results of K-B&B in such a way that it can be used as the output of the neural network in order to train the prediction model.

4.3.1 Prediction model input

We want to find the partitions of the uncertainty set of a problem, based on the problem parameters of that problem. For this thesis, we study the capital budgeting problem (Section 3.2). The nominal costs $\boldsymbol{c}^0 \in \mathbb{R}_+^N$ and the sensitivity matrices $\boldsymbol{\Phi}, \boldsymbol{\Psi} \in \mathbb{R}^{N \times N_p}$ are the only parameters of the problem that differentiate the instances. The other parameters are either fixed for the instances, or depend on the nominal cost. This is specified under the paragraph **Test case** in Section 3.2. Therefore, the nominal cost and sensitivity matrices are the input of the neural network.

Remark If the partition of the uncertainty set for a number of different two-stage robust optimization problems, e.g. both the capital budgeting problem and the facility location problem, are trained and predicted, we need a general formulation of these problems. This has the following form:

$$\begin{aligned}
\min_{\boldsymbol{x},\boldsymbol{y}} \quad & \tau \\
\text{s.t.} \quad & (\boldsymbol{c} + \boldsymbol{C}\boldsymbol{\xi})^\intercal \boldsymbol{x} + (\boldsymbol{d} + \boldsymbol{D}\boldsymbol{\xi})^\intercal \boldsymbol{y} \leq \tau && && \forall \boldsymbol{\xi} \in \Xi \\
& (\boldsymbol{a}^i + \boldsymbol{A}^i\boldsymbol{\xi})^\intercal \boldsymbol{x} \leq \boldsymbol{b}^i + \boldsymbol{B}^i\boldsymbol{\xi} && \forall i \in \{1, ..., m_1\}, && \forall \boldsymbol{\xi} \in \Xi \\
& (\boldsymbol{f}^i + \boldsymbol{F}^i\boldsymbol{\xi})^\intercal \boldsymbol{x} + (\boldsymbol{g}^i + \boldsymbol{G}^i\boldsymbol{\xi})^\intercal \boldsymbol{y} \leq \boldsymbol{h}^i + \boldsymbol{H}^i\boldsymbol{\xi} && \forall i \in \{1, ..., m_2\}, && \forall \boldsymbol{\xi} \in \Xi \\
& \boldsymbol{x} \in \mathcal{X}, \boldsymbol{y} \in \mathcal{Y},
\end{aligned} \tag{4.10}$$

where $\boldsymbol{x} \in \mathcal{X}$ are the first-stage decision variables, $\boldsymbol{y} \in \mathcal{Y}$ are the second-stage decision variables, and $\boldsymbol{\xi} \in \Xi$ are the uncertain scenarios. $\mathcal{X} \in \mathbb{R}^{N_1}$, $\mathcal{Y} \in \mathbb{R}^{N_2}$, $\Xi \in \mathbb{R}^{N_p}$ are the feasibility sets. The parameters $\boldsymbol{c} \in \mathbb{R}^{N_1}, \boldsymbol{C} \in \mathbb{R}^{N_1 \times N_p}$, $\boldsymbol{d} \in \mathbb{R}^{N_2}, \boldsymbol{D} \in \mathbb{R}^{N_2 \times N_p}$ are the parameters for the objective function. The parameters for the m_1 first-stage constraints are $\boldsymbol{a}^i \in \mathbb{R}^{N_1}, \boldsymbol{A}^i \in \mathbb{R}^{N_1 \times N_p}$, for all $i \in \{1, ..., m_1\}$. The parameters for the m_2 second-stage decision constraints are $\boldsymbol{f}^i \in \mathbb{R}^{N_1}, \boldsymbol{F}^i \in \mathbb{R}^{N_1 \times N_p}$, $\boldsymbol{g}^i \in \mathbb{R}^{N_2}, \boldsymbol{G}^i \in \mathbb{R}^{N_2 \times N_p}$ for all $i \in \{1, .., m_2\}$.

4.3.2 Prediction model output

This section is divided into three parts. The first part explains how we prepare the data to train the neural network. The second part explains how from the prediction, we can obtain the subsets $\bar{\Xi}_1, ..., \bar{\Xi}_K$ for the K-PP algorithm. And finally, the third part gives additional output methods for other two-stage robust optimization problems.

4.3.2.1 Preparation of training data

The output of the neural network is the optimal partition of the uncertainty set. To train and test the model, the two-stage robust optimization problem will be solved by K-B&B (Section 4.1). If the number of partitions becomes larger, the tree in the K-B&B algorithm becomes larger too. Therefore, the total runtime of the algorithm increases exponentially. When training a neural network, we need a big data sample. Therefore, we decide to limit the number of partitions to a maximum of 4. The output of the K-B&B algorithm contains the following:

- Optimal objective value θ.
- Solutions of the first-stage decisions $\boldsymbol{x}$.
- Solutions of the second-stage decisions $\boldsymbol{y}_k$ for all $k \in \mathcal{K}$.
- Subsets $\dot{\Xi}_k$ for all $k \in \mathcal{K}$. Note that not all scenarios $\boldsymbol{\xi} \in \Xi$ are contained in these subsets.

Due to the last item of the output, the structure of the partitioned uncertainty set is not yet clear. To obtain this structure, we need to derive the optimal partition using a large number of scenarios from the uncertainty set. For each scenario, we try to find the best subset it ‘wants’ to belong to, based on the results of K-B&B. Pseudo-code 1 gives the steps of this allocation method.

```
Pseudo-code 1: Allocation method
  Input          : First-stage decisions $\boldsymbol{x}$
                   Second-stage decisions $\boldsymbol{y}_k$ for all $k \in \mathcal{K}$
                   Set of scenarios $\hat{\Xi} \subset \Xi$
  Output         : Scenarios $\boldsymbol{\xi} \in \hat{\Xi}$ partitioned into $\hat{\Xi}_1, ..., \hat{\Xi}_K$
  Initialization: The set $\mathcal{S}^i_{\text{obj}} = \emptyset$ for all $i = 1, ..., |\hat{\Xi}|$ includes the objective values based
                   on scenario $\boldsymbol{\xi}_i$ and the decisions $\boldsymbol{x}$ and $\boldsymbol{y}_k$ for $k \in \mathcal{K}$ separately.
1 for $i \in \{1, ..., |\hat{\Xi}|\}$ do
2     for $k \in \mathcal{K}$ do
3         Check feasibility of $\boldsymbol{\xi}_i$ for decisions $\boldsymbol{x}$ and $\boldsymbol{y}_k$
4         if feasible then
5             Calculate the objective value based on scenario $\boldsymbol{\xi}_i$ and decisions $\boldsymbol{x}$ and $\boldsymbol{y}_k$
6             $\mathcal{S}^i_{\text{obj}} = \mathcal{S}^i_{\text{obj}} \cup \{\boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{\xi}_i)^\intercal \boldsymbol{y}_k\}$
7     Allocate $\boldsymbol{\xi}_i$ to the subset $\hat{\Xi}_k$ for which the decisions $\boldsymbol{x}$ and $\boldsymbol{y}_{\boldsymbol{k}}$ are feasible and give
      the best objective value.
8     $k^* = \arg\min_{k \in \mathcal{K}} \mathcal{S}^i_{\text{obj}}$
9     $\hat{\Xi}_{k^*} = \hat{\Xi}_{k^*} \cup \{\boldsymbol{\xi}_i\}$
```

If the number of scenarios in the set $\hat{\Xi}$ is large enough, we are able to find the structure of the partition. In Figure 4.3, two examples of this allocation method on two different instances are given. The squared instances are the instances found by K-B&B. These plots show that determining the structure of the subsets by only using for example the convex hull of the K-B&B scenarios, is neither sufficient nor accurate. Therefore, the allocation method is a useful tool for transforming the results of K-B&B in the actual partition.

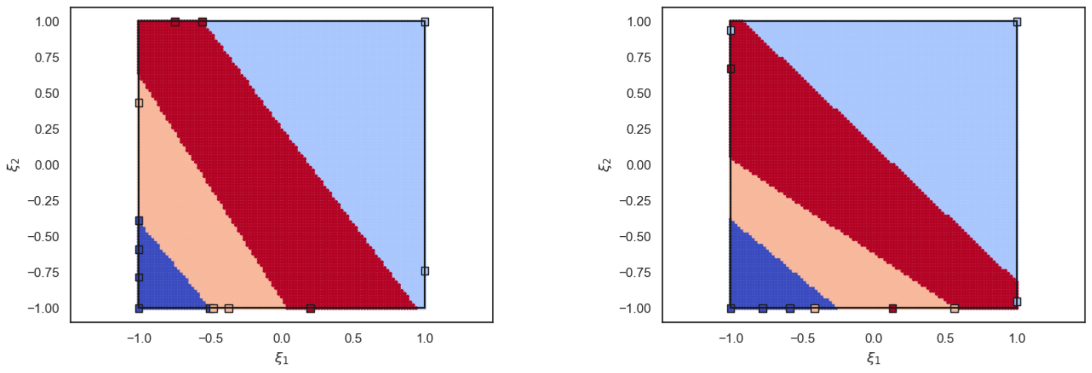


Figure 4.3: Partitioned uncertainty set of two instances of the capital budgeting problem for $K = 4$ with 10,200 scenarios. The squares are the scenarios found by K-B&B.

We need to find a way to summarize the partition such that it can be used as the output of the neural network. One can do this by defining hyperplanes that slice the uncertainty set. Partitioning the uncertainty set using hyperplanes is for example done by Vayanos et al. (2011) and Postek and den Hertog (2016). As mentioned in Section 4.2, we assume that the subsets of the partitioned uncertainty sets for the capital budgeting problem are convex, compact, and nonempty. Therefore, we can use the hyperplane technique for summarizing the partition.

Hyperplanes For the capital budgeting problem we noticed that the slices that split the subsets of the partition behave linearly, and do not cross each other. To obtain the linear line segments we can use the linear support vector machine (SVM) method.

SVM is a classification learning model. It uses a function to divide the data domain such that the data points belonging to the same class are grouped together (Suthaharan, 2016). The data domain in our case is $[-1, 1]^2$. We need this set to be divided into K subsets based on the data in $\hat{\Xi}$. In the case of linear SVM, the function that divides two classes is a hyperplane and is defined as

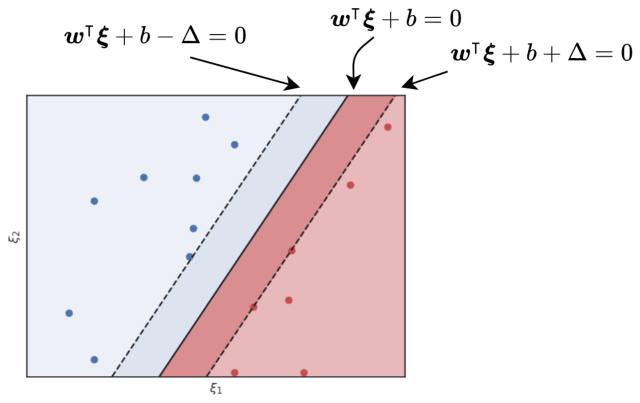
$$\boldsymbol{w}^\intercal \boldsymbol{\xi} + b = 0, \tag{4.11}$$

where $\boldsymbol{w} = (w_1, w_2)^\intercal \in \mathbb{R}^2$, $b \in \mathbb{R}$, and $\boldsymbol{\xi} \in \Xi = [-1, 1]^2$.

The goal of SVM is to find the values of $\boldsymbol{w}$, b, and Δ such that the distance between the two following hyperplanes is maximized:

$$\boldsymbol{w}^\intercal \boldsymbol{\xi} + b - \Delta = 0 \qquad \text{and} \qquad \boldsymbol{w}^\intercal \boldsymbol{\xi} + b + \Delta = 0, \tag{4.12}$$

Figure 4.4 illustrates this classification problem. To obtain these values, a quadratic problem is solved. The details of this problem can be found in Suthaharan (2016) and Bishop (2006).



$\boldsymbol{w}^\intercal\boldsymbol{\xi} + b - \Delta = 0$
$\boldsymbol{w}^\intercal\boldsymbol{\xi} + b = 0$
$\boldsymbol{w}^\intercal\boldsymbol{\xi} + b + \Delta = 0$
ξ_2
ξ_1

Figure 4.4: SVM illustration. The dashed lines are the parallel hyperplanes (4.12) to the hyperplane (4.11).

With the allocation method we obtain $\hat{\Xi}_1, ..., \hat{\Xi}_K$ that each consists of pre-selected scenarios $\boldsymbol{\xi} \in \hat{\Xi}$. With this information, we can classify each scenario into one of the K different classes. Hence, scenario $\boldsymbol{\xi}_i$ belongs to class $\mathcal{C}_i \in \{1, ..., K\}$ for $i \in \{1, ..., |\hat{\Xi}|\}$ if the following holds:

$$\mathcal{C}_i = k \iff \boldsymbol{\xi}_i \in \hat{\Xi}_k$$

Normally, the domain is not filled with as many classified data points as in our case. Then, we wish to predict the class of a non-classified new data point. However, SVM has a different application in this thesis. We summarize the partition with the values of $\boldsymbol{w}$ and b for each slice, obtained by SVM. In total we have $K - 1$ slices. We can reduce the number of hyperplane parameters to be predicted to $(\alpha^1, \beta^1), ..., (\alpha^{K-1}, \beta^{K-1})$. This has the following form:

$$\alpha^j = -\frac{w_1^j}{w_2^j} \quad \text{and} \quad \beta^j = -\frac{b_j}{w_2^j},$$

The subsets $\hat{\Xi}_1, ..., \hat{\Xi}_K$ are based on the order K-B&B placed the scenarios, which does not depend on the position of the subset in the uncertainty set. For predicting $(\alpha^1, \beta^1), ..., (\alpha^{K-1}, \beta^{K-1})$, it is more convenient if the hyperplanes, and therefore also subsets, are ordered based on its position. Hence, we order the line segments in the domain $\Xi = [-1, 1]^2$ where α^j is the coefficient and β^j the intersect of the j-th predicted hyperplane, for $j \in \{1, ..., K-1\}$, in the following way:

$$\alpha^1 \times -1 + \beta^1 \leq ... \leq \alpha^j \times -1 + \beta^j \leq ... \leq \alpha^{K-1} \times -1 + \beta^{K-1}$$

and

$$\alpha^1 \times 1 + \beta^1 \leq ... \leq \alpha^j \times 1 + \beta^j \leq ... \leq \alpha^{K-1} \times 1 + \beta^{K-1}$$

In other words, the line segments are ordered from the bottom left to the upper right corner of the uncertainty set. This means that the intersect of the line segment with $\boldsymbol{\xi_1} = -1$ and $\boldsymbol{\xi_1} = 1$ increases per ordered hyperplane. Figure 4.5 describes these hyperplanes that partition the uncertainty set.

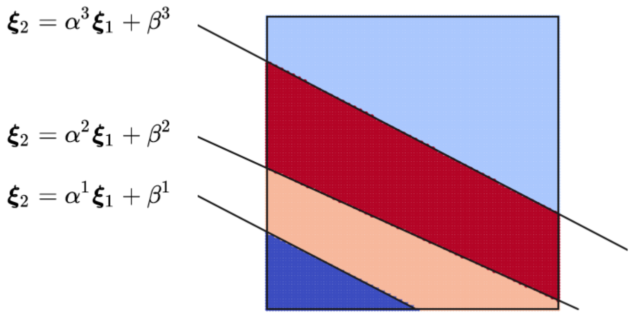

$\boldsymbol{\xi}_2 = \alpha^3\boldsymbol{\xi}_1 + \beta^3$
$\boldsymbol{\xi}_2 = \alpha^2\boldsymbol{\xi}_1 + \beta^2$
$\boldsymbol{\xi}_2 = \alpha^1\boldsymbol{\xi}_1 + \beta^1$

Figure 4.5: Partitioned uncertainty set with hyperplanes.

Figure 4.6 gives a summary of the methods used to prepare the data for the partition prediction model.

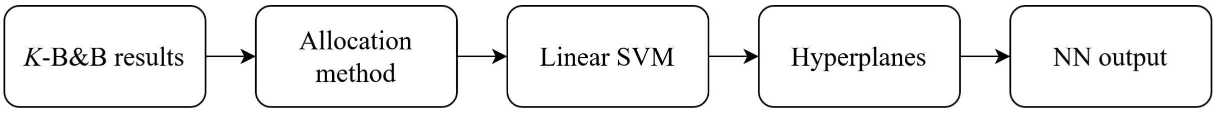


Figure 4.6: Methods to prepare for training.

4.3.2.2 Prediction to K-PP

In this section, we explain how to implement the prediction of the neural network into the K-PP algorithm.

As explained in Section 4.3.1, the problem parameters of the capital budgeting problem are used to predict the partition of the uncertainty set Ξ. The result of the neural network, based on the problem parameters, will consist of the $K-1$ hyperplanes. These hyperplanes slice the uncertainty set such that we have K subsets. However, if one of these hyperplanes intersects with another hyperplane in the domain of the uncertainty set Ξ, we would have an additional subset. The performance of K-PP would by definition be at least better than the one of K-B&B because the uncertainty set is partitioned in more parts. Hence, more second-stage decisions can be chosen to adapt to the uncertainty set. To prevent this from happening, we need to adjust these hyperplanes to ensure an equal amount of subsets for both algorithms. This is done in Step 2 of the K-PP algorithm. In this step, one of the two hyperplanes is changed such that the intersection is not inside $\Xi = [-1, 1]^2$, but on the edge of it. This is illustrated in Figure 4.7. This correction method was chosen to make sure the hyperplanes are as close to the prediction as possible.

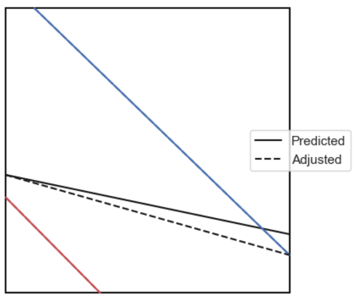


Figure 4.7: Adjusted hyperplanes in the domain $[-1, 1]^2$.

After a potential correction, we can obtain the subsets $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$ based on the hyperplanes. In Step 3, the extreme points of the subsets $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$ are selected and collected into the sets $\bar{\Xi}_1, ..., \bar{\Xi}_K$ to solve the K-adaptability problem in Step 4. Figure 4.8 gives a summary of the methods needed for the prediction output to become the data needed to solve K-PP.

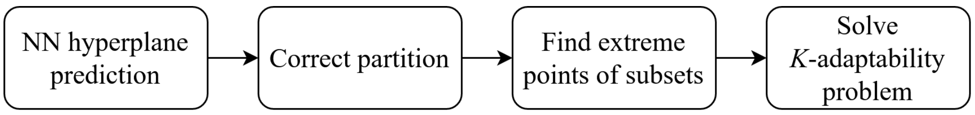


Figure 4.8: Methods used to transform the prediction of the hyperplanes such that it can be used in the K-PP algorithm.

4.3.2.3 Additional output methods

The disadvantage of the hyperplane summarizing technique is that it does not allow non-convex subsets. Hanasusanto et al. (2015) give an example of non-convex optimal subsets for problems with constraint uncertainty. For other problems than the capital budgeting problem, this non-convexity can still occur in its corresponding partitioned uncertainty set. For these problems we have other methods to summarize the partitions. In this section, two other techniques for summarizing are presented.

Centroids of subsets For this method, the centroid of each subset needs to be found and selected. This is illustrated in Figure 4.9. To obtain subsets of the uncertainty set, one could use the K-nearest neighborhood method, where K is the number of data points considered in this method, and not the number of subsets the uncertainty set is partitioned in. Since only one data point per class is identified, i.e. the centroid, this will be a 1-nearest neighborhood method.

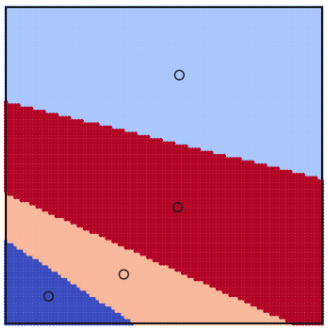


Figure 4.9: Partitioned uncertainty set with outlined centroids.

Support vectors With this method, the scenarios that lie around the edges of the subsets are selected. This is illustrated in Figure 4.10. These scenarios can be found by using support vector machines (SVM). The data points that lie closest to the function that split the domain are called the support vectors. These vectors are used to form the function. Instead of using the linear hyperplanes to split the uncertainty set, a non-linear function can be used (Bishop, 2006).

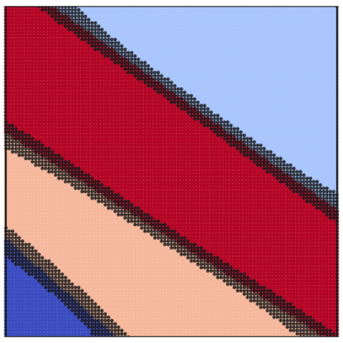
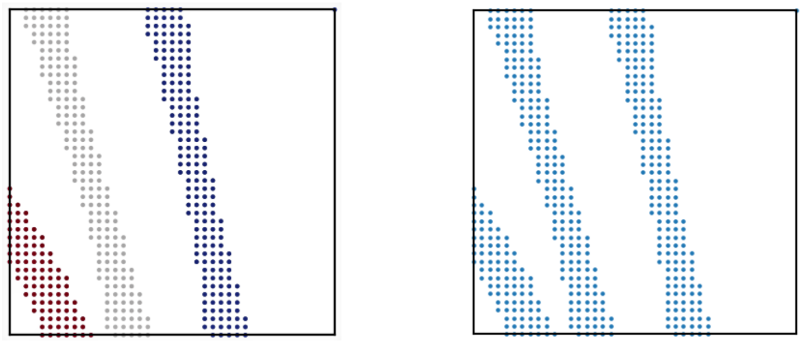


Figure 4.10: Partitioned uncertainty set with outlined support vectors.

The outlined scenarios in Figure 4.10 are the support vectors that summarize the uncertainty set partitions. For the capital budgeting problem, for K subsets constructed by K-B&B, we have $K-1$ groups of support vectors. With these support vector groups we can formulate two different methods. The first method is to define a classification problem by classifying each support vector group separately, plus classifying the scenarios not categorized as support vector, which sums up to K classes in total. This problem can be formulated as a K multi-class classification problem. The second method is to formulate the problem as a binary classification problem, where all support vectors belong to one class, and all scenarios not categorized as support vector to the other class. Figure 4.11 illustrates both methods.



(a) K multi-class classification **(b)** Binary classification

Figure 4.11: Classification methods for support vector groups.

Note that by summarizing the partitioned uncertainty set with support vectors, the dimension of the output of the prediction model is equal to the number of scenarios selected. If there are many scenarios, the prediction model becomes larger which increases the training time.

4.4 Neural network

Since no literature can be found that uses machine learning tools for enhancing solution methods for two-stage robust optimization problems, we have no pre-knowledge of a favourable prediction model. Therefore, we need an extensive model, such that the risk of not finding complex patterns is low. For this reason, we chose a fully connected feedforward neural network. Such neural networks are easily implementable and adjustable via publicly available packages such as Keras (Chollet et al., 2015), which we use. In Section 4.4.1 we give an explanation of feedforward neural networks (FNN). The method for training these models is backward propagation. This method is discussed in Section 4.4.2. Finally, in Section 4.4.3, the method for feature importance detection in FNNs is explained. Throughout the first two sections, the theory explained is mainly based on Goodfellow et al. (2016) and the lecture notes of Birbil (2019).

4.4.1 Feedforward neural network

The goal of feedforward neural networks (FNN) is to estimate an input to output mapping $\boldsymbol{y}_{NN} = f(\boldsymbol{x}_{NN})$ based on input features $\boldsymbol{x}_{NN}$ and the output $\boldsymbol{y}_{NN}$. Here we added the subscript *NN* to not get confused with the first- and second-stage decision variables of the two-stage robust optimization problem. The FNN is a fully connected layered network of nodes, or neurons, that form a directed acyclic graph. The first layer is the input layer, the last layer is the output layer, and the layers in between are called the hidden layers. Figure 4.12 gives an example of an FNN with two hidden layers.

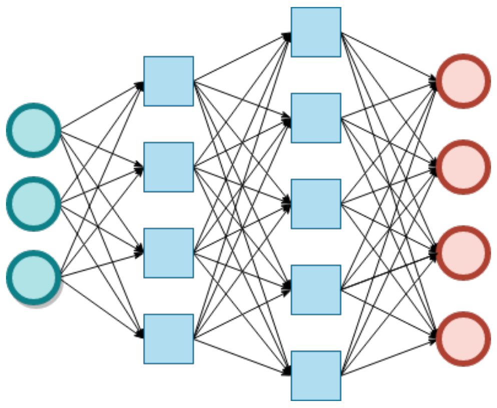


Figure 4.12: Feedforward neural network framework with two hidden layers, 3 features, and 4 outputs.

Every arc of the graph has a weight, and each node has a bias. These weights and biases describe the activation of a certain node, and are optimized such that the error of the prediction of the neural network and the ground truth is minimized. To be more precise, each node $j \in \{1, ..., n_l\}$ in layer $l \in \{1, ..., L, L+1\}$ has a (non-)linear activation $z_j^{(l)} = f^{(l)}(a_j^{(l)})$. In this formulation, $l = 0$ is the input layer, and $l = L+1$ is the output layer. The input of the activation of connection (i, j) between layer $l-1$ and l is equal to $a_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} z_j^{(l-1)} + w_{j0}^{(l)}$. Here, the weights between i and j are denoted as $w_{ji}^{(l)}$ and the bias of node j is described as $w_{j0}^{(l)}$. Using all this information, we can formulate the function of output $\boldsymbol{y}_{NN}$ in FNN:

$$\boldsymbol{y}_{NN} = f(\boldsymbol{x}_{NN}) = f^{(L+1)}(f^{(L)}(f^{(L-1)}(...(f^{(1)}(\boldsymbol{x}_{NN})...)))$$

Note that in this formulation, $z_j^{(0)} = x_{NN}^j$.

4.4.2 Backward propagation

To train neural networks, or any other supervised machine learning model, we need an input, output, and a cost function to evaluate prediction performance. The goal is to minimize the cost, or equivalently, to maximize the (log-)likelihood. Due to the non-linear nature of neural networks, the loss function (of the entire neural network) becomes non-convex. This means that we cannot easily obtain the global minimum loss. Therefore, we need to train the neural network iteratively. This is done with gradient-based optimization methods. Using such methods, the loss function is minimized. However, it cannot be guaranteed that a global optimum will be reached.

The only elements of the FNN than can be influenced and changed, are the weights w_{ji}^{l} and biases w_{j0}^{l} of each node j in layer l. All weights and biases are collected in a vector of parameters $\boldsymbol{\phi}$. The first step of training is to initialize these parameters to be small random values. Then, in order to obtain an FNN with minimal prediction error, we use the commonly used backward propagation method.

The process of training an FNN with the backward propagation method can be divided in two parts which alternate one after another: the forward pass and back-propagation. After initializing the model weights and biases in $\boldsymbol{\phi}$, a forward pass is performed. Here, we predict output $\hat{\boldsymbol{y}}_{NN}^{(m)}$ based on input $\boldsymbol{x}_{NN}^{(m)}$ for each training observation m. This produces scalar cost $C_m(\boldsymbol{\phi})$. Back-propagation determines the derivatives between the loss value and all weights and biases within the FNN. These derivatives give information on how to change these parameters such that the next forward pass will result in lower cost.

For illustration, imagine we have a loss function $C(w)$ with only one parameter w and we want to find the minimum value. We move in the direction in which we have the steepest descent. Hence, for a 1-dimensional function, we move in the direction for which the slope

$$\frac{dC(w)}{dw}$$

is lowest. This is described in Figure 4.13. The black square is the initial weight, which has a corresponding cost. The weight will move in the direction of steepest descent, hence along the green arrow. Note that the initial value of the weight is important for finding the global minimum value of the cost.

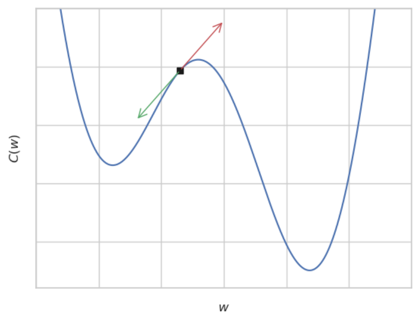


Figure 4.13: Gradient descent for a one dimensional loss function.

In general, FNNs have a lot more parameters. For example, the total number of parameters of the very small FNN in Figure 4.12 already has $3\times4+4\times5+5\times4 = 52$ weights and $4+5+4 = 13$

biases. Which results in 65 parameters in total. The loss function of this model will therefore be 65-dimensional. Thus, finding the direction in which we want to move to will be more complex.

For FNN, gradient descent requires computing the derivative of the cost with respect to all parameters of the model, using all training data $m \in \{1, ..., M\}$, i.e.

$$\Delta_{\phi} C(\phi) = \frac{1}{M} \sum_{m=1}^{M} \Delta_{\phi} C_m(\phi) \tag{4.13}$$

These gradients $\Delta_{\phi} C_m(\phi)$ are computed using the chain rule. Figure 4.14 illustrates the dependencies in an FNN with only one node per layer.

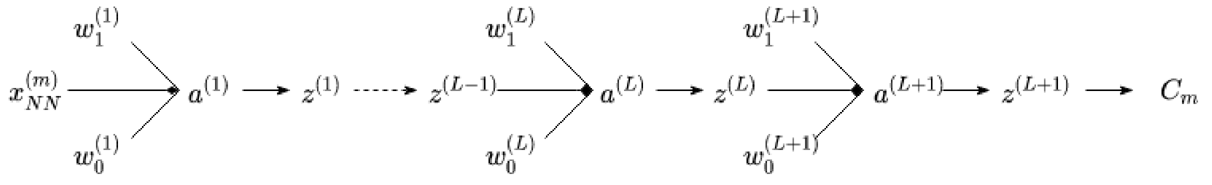


Figure 4.14: FNN dependencies with one node per layer

It is computationally inefficient to train the FNN based on all data available, so for all $m \in \{1, ..., M\}$. For example, with many observations, the computer can run out of memory. Therefore, to train the network more efficient, batch gradient descent is used. Sequentially a random batch of training data is selected to train. After each batch finishes training, another batch of the training data will be selected. This continues until all training data has been trained. The process in which all data is used once, is called an epoch. The batch size is typically between one and a few hundred. If the size of the batch is one, this method is called stochastic gradient descent. Contrary to normal gradient descent, batch gradient descent converges to the minimum cost faster, but with more shocks. The insight of batch gradient descent is that the change the FNN parameters ϕ need to make, can be approximated using less observations per forward and backward pass. This is faster since using smaller data sets results in a more efficient mathematical program. Batch gradient descent has more shocks since by not using all the data, we can move in a wrong direction easily. By increasing the batch size, we have less shocks.

The number of epochs used to train an FNN can be predetermined, or can be decided during training the FNN based on reaching a certain level of the cost. If too many epochs are performed, the neural network can overfit the data. Overfitting the data means that the model follows the training data too closely, such that it does not perform well on other data. It can happen that instead of learning from the data, the model will simulate it. Overfitting can also happen when there are too many parameters in the model, hence, when it is too complex. Instead of restricting the number of epochs, one can also use regularization techniques to obtain a better performance for testing data.

4.4.3 Feature importance

By design, FNNs are non-linear and consist of many different parameters that contribute to the performance of the model. Interpreting the actions neural networks take is therefore difficult. Most people do not bother about the interpretation, but in some fields this is very important. For example, in the field of medicine, neural networks can be used as a decision making tool for cancer treatment (Lisboa and Taktak, 2006). Here, the cost of bad decisions can be very high. Therefore, being able to know why a certain decision is made, is important.

For this thesis, we are predicting the partition of the uncertainty set to solve the K-adaptability problem (1.2) faster. The goal of detecting feature importance for us is to become aware of the elements in the two-stage robust optimization problem that have an effect on the partitions, and which do not. Also, due to to the long training time of FNNs with many parameters, feature importance can help us deleting irrelevant features.

One could argue that instead of using a machine learning model difficult to interpret, we should use one capable of this. Such as classification and regression trees. However, this is often at the cost of prediction performance.

4.4.3.1 DeepLIFT

The DeepLIFT (Deep Learning Important FeaTures) method tries to find the importance of the features of a trained neural network (Shrikumar et al., 2017). This method uses a back-propagation approach to assign importance scores to the nodes in the network based on a given input $\boldsymbol{x}_{NN}^m$ and its prediction $\hat{\boldsymbol{y}}_{NN}(\boldsymbol{x}_{NN}^m, \boldsymbol{\phi})$ of observation m. DeepLIFT uses a difference-from-reference approach to detect the importance of every node based on its reference. The reference point can be seen as the default or neutral point. Therefore, this reference is problem specific. As explained in Section 4.3.1, the features of the FNN are the nominal costs ($\boldsymbol{c}^0 \in [0,10]^N$) and the sensitivity matrices ($\boldsymbol{\Phi}, \boldsymbol{\Psi} \in [0,1]^{N \times N_p}$). The reference chosen for each of these features is the lower bound of its domain, and therefore 0. Assigning different values as reference instead of the lower bound can also be done, e.g. the average or a random value in the domain of the corresponding feature.

For a full overview of DeepLIFT, we point to Shrikumar et al. (2017). In this thesis, only a brief explanation will be given, in which we abuse the formulation such that it is understandable for our application. We can do this since the references in the nodes are 0.

Let t be the target node of interest in layer l. $\{x_1, ..., x_{n_{l-1}}\}$ are the inputs of t in layer $l-1$. Then $CS_{x_i,t}$ is the contribution score of input x_i onto target t, for any $i \in \{1, ..., n_{l-1}\}$, such that:

$$\sum_{i=1}^{n_{l-1}} CS_{x_i,t} = t$$

Hence, the value of t is divided into the n_{l-1} inputs with respect to their contribution to t. The multiplier $m_{x_i,t}$ is the contribution of x_i to t divided by x_i:

$$m_{x_i,t} = \frac{CS_{x_i,t}}{x_i} \tag{4.14}$$

With the chain rule and back-propagation, we can calculate the feature importance score for all features and output combinations. This is formulated as:

$$CS_{x_{NN}^j, y_{NN}^k} = m_{x_{NN}^j, y_{NN}^k} x_{NN}^j,$$

for all $j \in \{1, ..., n_0\}$ and for all $k \in \{1, ..., n_{L+1}\}$. How the multipliers are calculated is explained in the aforementioned DeepLIFT paper. In short, the nodes are divided into a positive and negative component, such that negative contributions of features to the output are assigned more clearly.

4.5 K-PP for initialization of K-B&B

The results of K-PP are based on a prediction of the partition, and therefore not exact. However, we can use the results of K-PP to initialize K-B&B such that it converges to optimality in less iterations. We call the modified version of the K-B&B algorithm in which we use elements of K-PP, the Enhanced K-B&B algorithm. The solutions of K-PP contain the following items:

- Objective value $\tilde{\theta}$.
- Solutions of the first-stage decisions $\tilde{\boldsymbol{x}}$.
- Solutions of the second-stage decisions $\tilde{\boldsymbol{y}}_k$ for all $k \in \mathcal{K}$.
- Subsets $\tilde{\Xi}_k$ for all $k \in \mathcal{K}$.

The objective value $\tilde{\theta}$ of K-PP is used as an upper bound for the Enhanced K-B&B algorithm. In the initialization step (Step 1) of the Enhanced K-B&B algorithm, we set $\theta^i \leftarrow \tilde{\theta}$. Selecting $\tilde{\theta}$ as the initial value of θ^i instead of $+\infty$ causes that in Step 5, many branches are cut. This results in skipping Steps 4 and 5 often, such that we are creating less nodes in the tree, and therefore have to solve less unnecessary MILPs. However, it can take a longer time in the beginning of the algorithm to obtain an incumbent solution.

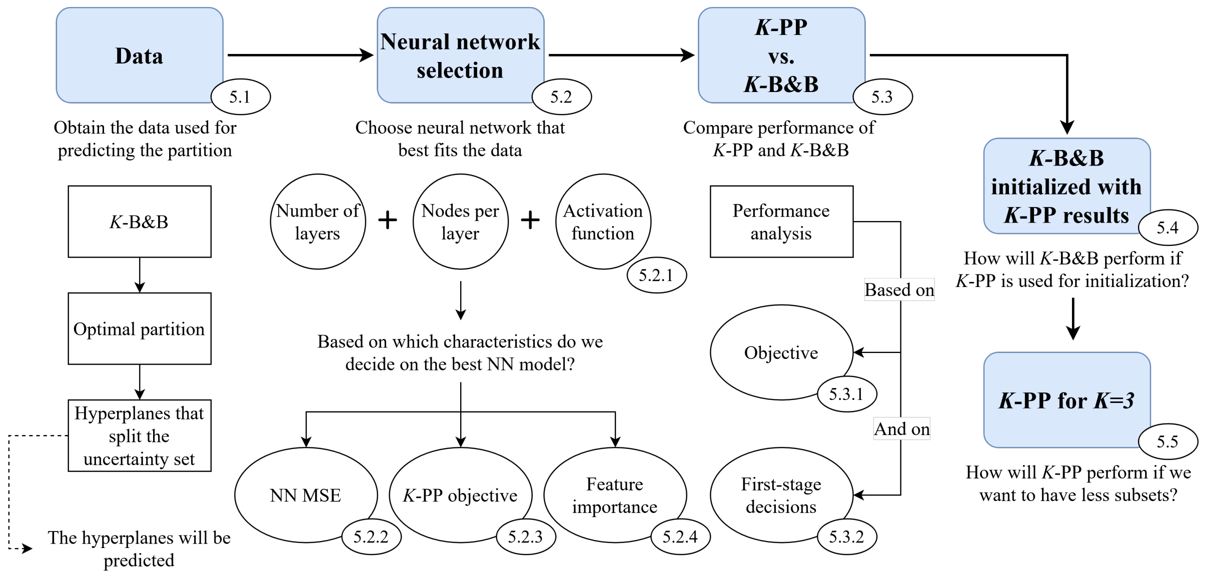
When the Enhanced K-B&B algorithm is not run to optimality, but stopped due to some maximum running time criterion, initializing the incumbent decisions with the K-PP decisions $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{y}}$ is essential. Due to the longer time for the first incumbent solution to be found, it can happen that within the maximum runtime, no single incumbent solution was found. If this happens, the decisions of K-PP will be the solution to the Enhanced K-B&B algorithm.

With the predicted subsets $\tilde{\Xi}$, we can help the K-B&B algorithm during its rough start by assigning the centroids of the predicted subsets $\tilde{\Xi}_k$ to the subsets $\dot{\Xi}_k$, for $k \in \{1, ..., K\}$. This way not only the objective found by K-PP, but also information about the partitions is given to the algorithm. Therfore, in Step 1, the set with initial scenarios τ^0 is filled with the centroids of $\tilde{\Xi}_1, ..., \tilde{\Xi}_K$. The extreme points of $\tilde{\Xi}_k$ are not included because this leaves the K-B&B algorithm without space improve. This selection of scenarios results in a fully robust solution, equal to the one of K-PP. In Step 4, $\zeta < 0$. This results in skipping Step 5. In this step, the new nodes are made. Therefore, if we skip this step immediately in the beginning of the Enhanced K-B&B algorithm, the algorithm would stop.

Remark The subsets of the uncertainty set in the capital budgeting problem all behave in the same way. Therefore, selecting the centroids of the subsets as initial scenarios will not cause harm. For problems where the partition of the uncertainty set do not behave similarly for different instances, it is better to not initialize the scenarios and only use $\tilde{\theta}$ for initialization of the Enhanced K-B&B algorithm.

5 | Numerical experiments

In this chapter, we test the methodology as described in the previous chapter (Chapter 4) on the capital budgeting problem with loans (Section 3.2). An illustrative overview of this chapter is given in Figure 5.1.



Data
5.1
Obtain the data used for predicting the partition
Neural network selection
5.2
Choose neural network that best fits the data
K-PP vs. K-B&B
5.3
Compare performance of K-PP and K-B&B
K-B&B initialized with K-PP results
5.4
How will K-B&B perform if K-PP is used for initialization?
K-PP for K=3
5.5
How will K-PP perform if we want to have less subsets?
K-B&B
Optimal partition
Hyperplanes that split the uncertainty set
The hyperplanes will be predicted
Number of layers
Nodes per layer
Activation function
5.2.1
Based on which characteristics do we decide on the best NN model?
NN MSE
5.2.2
K-PP objective
5.2.3
Feature importance
5.2.4
Performance analysis
Based on
Objective
5.3.1
And on
First-stage decisions
5.3.2

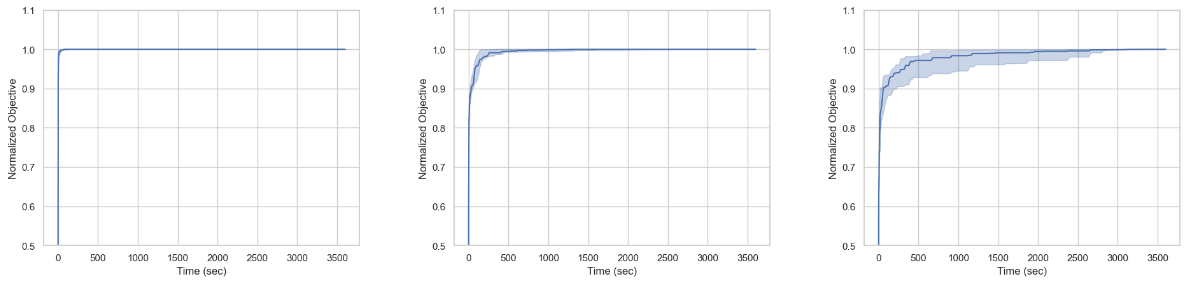
Figure 5.1: Overview of the steps taken in the numerical experiments chapter.

In Section 5.1, we first discuss how we obtain the data for the neural networks. When the uncertainty set is partitioned into a larger number of parts, the second-stage can adapt more often to the uncertainty. Therefore, a better solution of the problem is found. This is the reason for predominantly focusing on the case where K=4. Except for the last section of this chapter, in which we will use the methods on the case where K=3. In Section 5.2, we select the best performing feedforward neural network (FNN). First, in Section 5.2.1 we select some design parameters of the FNN commonly used for the kind of problem we are solving. Then, in the following sections, other design parameters of the FNN are selected based on the performance this prediction method has on three different elements. The first one is the performance the FNN has based on the prediction accuracy (Section 5.2.2). The second measure on which we select the FNN, is the performance it has when used in the K-adaptability partition prediction (K-PP) algorithm (Section 5.2.3). The third measure on which we decide the FNN, is feature importance (Section 5.2.4). In this section we try to find the features that most and least effect the partitions. Based on these observations, we can change the features used for predicting the partitions to limit training time. After the FNN has been selected, in Section 5.3 we study the performance the K-PP algorithm has when this specific FNN is used. The performance is studied by comparing the objective values and first-stage decisions found by K-PP and by K-B&B. In Section 5.4, we analyze how K-B&B performs when the results of K-PP are used as initial solutions. And finally, in Section 5.5, we look at the performance the K-PP algorithm has for K=3.

5.1 Data

To train and test the different FNNs, the K-B&B algorithm is solved for around 5000 instances of the capital budgeting problem with loans. 4000 instances are used for training and 1000 for testing. Implementations are done in Python 3.6.6. The K-B&B algorithm is solved with the Gurobi Optimizer version 8.0.1 (Gurobi Optimization, 2020). We use SURFsara's Lisa clustering system to run all these jobs in parallel. This cluster uses Intel Xeon Gold 6130 Processor, 2.10 GHz with 16 cores. Running all the problem instances to optimality can sometimes take more than 12 hours. Therefore, we have to choose a time limit. Due to the steps of K-B&B, the best robust solution found so far, i.e. the incumbent solution, is selected after the maximum runtime.

For the capital budgeting problem we decided on a maximum runtime of 20 minutes for the algorithm to converge for $K \in \{2, 3, 4\}$. In Figure 5.2, the normalized objective (with respect to the last θ found) over computation time is illustrated for 7 instances that were stopped after one hour.



(a) $K = 2$ **(b)** $K = 3$ **(c)** $K = 4$

Figure 5.2: Normalized incumbent best objective values over computation time.

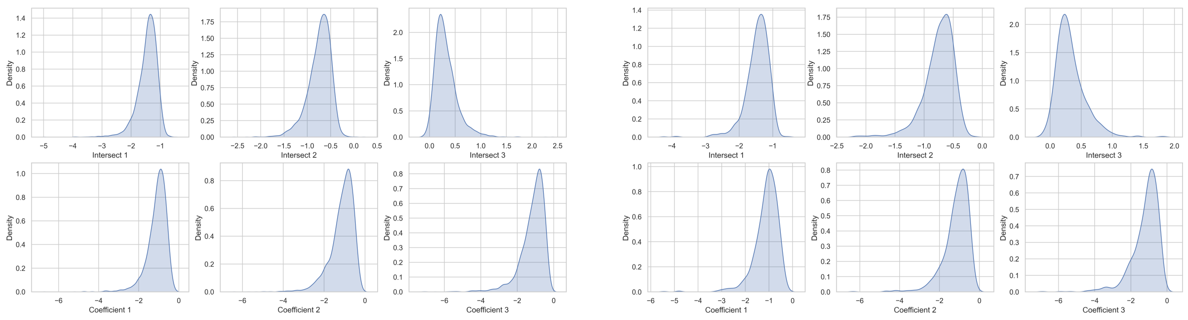
Table 5.1 shows for $K = 2, 3, 4$ the K-B&B performance of the instances of the capital budgeting problem with loans. For K=2, all instances are run to optimality within a reasonable computation time. For K=3, some instances are run until completion of the algorithm. For K=4, no instances are exactly solved.

Table 5.1: K-B&B performance of the data used in the neural network. *5%* is the 5% quantile, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 5% quantile of the runtime of the K-B&B algorithm. *Exact* is the percentage of instances run until completion of the algorithm.

K	Runtime K-B&B				Exact (%)
	5%	10%	Median	90%	
2	65.2539	89.811	186.994	307.484	100
3	924.853	1200.02	1200.27	1200.63	7.35
4	1200.01	1200.02	1200.12	1200.36	0

The method used for transforming the capital budgeting problem parameters into the input is given in Section 4.3.1. In Section 4.3.2, we show how the results of K-B&B for a given capital budgeting problem are transformed into subsets $\hat{\Xi}_1, ..., \hat{\Xi}_K$. $K - 1$ hyperplanes are selected that slice the uncertainty set that form these subsets. These hyperplanes are ordered from left to right, for all instances.

For neural networks, it is important that the distributions of the input for the testing and training data are identical. Since we generate the problem parameters ourselves, where for each instance the problem parameters are drawn from the same distribution, this requirement holds. The output distributions of the test and training data are also similar. This is illustrated in the plots in Figure 5.3 for $K = 4$.



(a) Training Data **(b)** Testing Data

Figure 5.3: Distributions of the coefficients and intersections of the three ordered hyperplanes that split the uncertainty set into four parts.

5.2 Neural network selection

As discussed in Section 4.4.1, the model used for predicting the partitioned uncertainty set of the capital budgeting problem, is a feedforward neural network (FNN). FNNs have many design parameters, e.g. the number of layers, activation functions, and cost functions the FNN has. In this section, we will select these design parameters. In Section 5.2.1, we already select some design parameters, and introduce the selection of other parameters that have to be chosen based on the performance of the FNN. In Section 5.2.2 we discuss the results of the different FNNs. In Section 5.2.3 we select the FNN based on the performance the K-PP algorithm has when this FNN is used. To obtain more information on the features, we perform the DeepLIFT method on the selected FNN in Section 5.2.4.

5.2.1 NN design parameters

In Section 4.4.1, we introduced the FNN together with a number of model design parameters. In this section we elaborate on these parameters.

First, we discuss the output unit of the FNN. The output unit is the activation of the last layer $(L + 1)$. Using the notation from before, we now decide on $f^{(L+1)}$. The role of this layer is to give the final transformation of the data. The choice of the output unit is tightly linked to the distribution of the output. Moreover, it is also linked to the choice of the cost function of the model. The most common three output units are the linear, sigmoid, and softmax units. These output units are formulated as:

$$\sigma(a_j) = f^{(L+1)}(a_j) = a_j \qquad \text{(linear)}$$

$$\sigma(a_j) = f^{(L+1)}(a_j) = \frac{1}{1 + \exp(-a_j)} \qquad \text{(sigmoid)}$$

$$\sigma(a_j) = f^{(L+1)}(a_j) = \frac{\exp(a_j)}{\sum_{i=1}^{n_{(L+1)}} \exp(a_i)}, \qquad \text{(softmax)}$$

for all $j \in \{1, ..., n_{(L+1)}\}$. These units are suited for Gaussian, Bernoulli, and Multi-Bernoulli output distributions, respectively. When using the hyperplane output, we wish to predict the intersections and coefficients of the partition slices. Hence, we need a regression model. Figure 5.3 shows the densities of the different outputs of this FNN for 4 partitions. Thus, we have $n_{(L+1)} = 2(K-1)$. The distributions of all outputs behave normal-like. Therefore, it is accepted to use the linear output unit for this FNN.

When training an FNN, we want to define a distribution $p(\boldsymbol{y}_{NN}|\boldsymbol{x}_{NN}; \boldsymbol{\phi})$. To test if this distribution is accurate, the principle of maximum likelihood is used. In practice, this means that the cross-entropy between the training data and the prediction is the cost function of the model. We can also take a simpler approach. Instead of predicting the complete distribution, we use some statistic of $\boldsymbol{y}_{NN}$ conditioned on $\boldsymbol{x}_{NN}$.

Maximizing the log-likelihood for normal distributed outputs, with a linear output unit, is equivalent to minimizing the mean squared error. Therefore, we chose for the sum of squared errors (SSE) as cost function. The SSE is an example of one of these statistics to use instead of cross-entropy as cost function of the model. The SSE function is defined as follows:

$$C_m(\boldsymbol{\phi}) = \frac{1}{2} \sum_{j=1}^{n_{(L+1)}} (\hat{y}^j_{NN}(\boldsymbol{x}^m_{NN}, \boldsymbol{\phi}) - y^{m,j}_{NN})^2$$

Subsequently, we have to decide on the activation function for the hidden units. Traditionally, the sigmoid or the closely related hyperbolic tangent activation functions are used in neural networks. However, these activation functions saturate across most of their domain. Saturation entails that for high values of a, $f(a)$ will go to the upper bound of the function, and for low values of a, $f(a)$ will go to the lower bound. Furthermore, the functions are only very sensitive to the input when it is near 0. This saturation makes training the FNN using gradient descent very difficult. Therefore, the rectified linear unit, also known as ReLU, is used as activation function in our model. ReLU is formulated as:

$$f(a) = \max\{0, a\}$$

ReLU behaves linear in half of its domain. For optimizing a model, it is easier and faster if the behavior is close to linear.

We now have to decide on the layers, the number of epochs and the batch size. Deciding on any of these specifications is very problem and data specific. In order to find an FNN architecture that suits the data best, we train a total of 24 different FNNs with each having different layers. The selection of layers is based on two criteria. The first one is depth, and the second one is width. The depth of an FNN describes the number of layers. In this thesis, *Depth* is used to describe the number of hidden layers per FNN architecture, excluding the input and output layers. The width of an FNN describes the number of nodes in the layers. Due to the large number of possibilities for different architectures, we have limited the FNNs to having the same amount of nodes per hidden layer. In this thesis, *Width* describes the number of nodes per layer. The batch size is chosen using a trial and error approach, and fixed for all architectures. For this data, with a batch size of 30, the error decreased quickly.

An overview of all combinations of design parameters not yet decided on, is given in Table 5.2. In the following sections, we will chose one of these combinations based on the performance with respect to the data set we have.

Table 5.2: FNN design parameters. The FNN is trained for each combination of the *Width*, *Depth*, and *Epochs*.

Width	Depth	Epochs
10	1	10
50	3	50
100	5	100
200	7	200
500		500
1000		1000
		1500
		2000

Remark In Section 4.3.2.3, two other output methods are described. For the centroid output, we want to predict the coordinates of the centroid of each subset. Thus, a regression model is required. The same cost function and output unit can be used as for the hyperplanes output. For the support vector output, these design parameters need to change. The support vector output is a classification problem. This method has two alternative approaches. The first one is where all support vectors are 1, and all non-support vectors are 0. This is binary classification, therefore Bernoulli distributed. This means that the output unit is the sigmoid function. For these functions, the maximum likelihood is the preferred approach for training. Hence, cross-entropy is the cost function. The cross-entropy function for binary classification is formulated as follows:

$$C_m(\boldsymbol{\phi}) = -\sum_{j=1}^{n_{(L+1)}} \left(y_{NN}^{m,j} \ln(\hat{y}_{NN}^{j}(\boldsymbol{x}_{NN}^{m}, \boldsymbol{\phi})) + (1 - y_{NN}^{m,j}) \ln(1 - \hat{y}_{NN}^{j}(\boldsymbol{x}_{NN}^{m}, \boldsymbol{\phi})\right)$$

The second approach is to group the support vectors in so-called support vector groups. Here we have multiple classes. Thus, a model for multi-class classification is needed. This output data is Multi-Bernoulli distributed, which requires the softmax output unit. For this FNN, we also use cross-entropy as cost function. The cross-entropy for multi-class classification is formulated as follows:

$$C_m(\boldsymbol{\phi}) = -\sum_{j=1}^{n_{(L+1)}} y_{NN}^{m,j} \ln(\hat{y}_{NN}^{j}(\boldsymbol{x}_{NN}^{m}, \boldsymbol{\phi})$$

5.2.2 NN results

The FNN is trained for all combinations of the *Width*, *Depth* and epoch sizes listed in Table 5.2. The full overview of results for these FNNs can be found in Table B.1. Keras Chollet et al. (2015) is used for implementation of these neural networks. The models are trained with Python 3.7, on a computer with Intel Core i7-10510U Processor, with 1.80 GHz and 4 cores. The FNNs predict the intersections and the coefficients of the hyperplanes that slice the uncertainty set. An example is given in Figure 5.4.

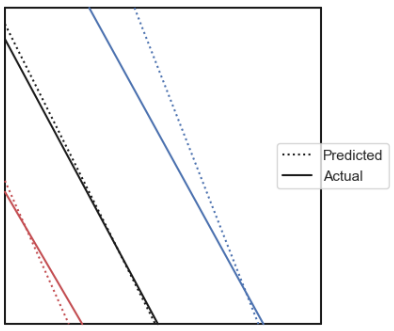


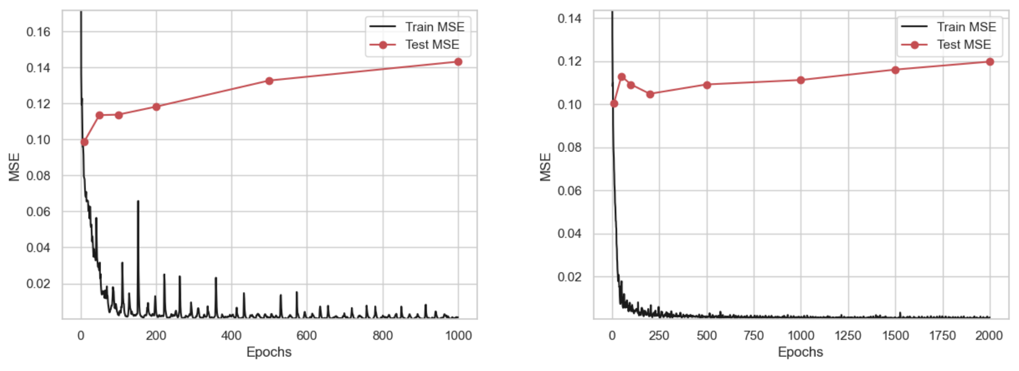
Figure 5.4: Example of FNN predictions where the actual hyperplane is found by using K-B&B, the allocation method, and linear SVM. The predicted line is found by predicting with an FNN with *Width* 1000 and *Depth* 7.

Out of all FNNs that were trained, we want to select the best FNN for this data. The five best FNN architectures selected based on their test MSE, are shown in Table 5.3.

Table 5.3: Five best FNN frameworks based on test MSE. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* are the number of parameters in the FNN.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
1000	7	6,063,006	10	0.0988	0.0787	122.5035
1000	3	2,059,006	10	0.1003	0.0751	43.0980
500	7	1,531,506	10	0.1012	0.0750	81.5342
500	5	1,030,506	10	0.1014	0.0783	54.0095
1000	3	2,059,006	200	0.1048	0.0018	858.2739

This table shows that neural networks with more parameters are favored for this data, but with a small number of epochs. So we can say that underfitting only happens when we have a small FNN, whereas overfitting happens very quickly when the epoch size increases. This observation becomes clear in Figure 5.5. The training and testing MSE per epoch are shown for the two best FNN architectures. We see that the test MSE of both FNNs increase after a small number of epochs.



(a) *Width* = 1000, *Depth* = 7 **(b)** *Width* = 1000, *Depth* = 3

Figure 5.5: Test and train MSE of the two best FNN architectures based on test MSE.

5.2.3 K-PP objective

In this section we study the performances of the trained FNNs in the K-PP algorithm. The data we have given the model is not based on exact solutions. Therefore, the test MSE would punish predictions that result in a better objective than the ones of K-B&B. To our surprise, this actually happens quite often. In Table C.1, a full overview of the results of the K-PP algorithm is given for all trained FNNs. In Table 5.4, the five best FNN architectures are given based on the median of the normalized objective. The normalized objective is defined as:

$$\textit{Normalized Objective} = \frac{\textit{Objective K-PP}}{\textit{Objective K-B\&B}} \tag{5.1}$$

Table 5.4: K-PP results that use the five best FNN frameworks. The K-PP algorithm is tested on 1000 test instances. The FNNs are selected based on the median value of the *Normalized Objective* (5.1). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the normalized objective of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
1000	7	6,063,006	100	0.9730	1.0061	1.0349	0.0894	0
1000	7	6,063,006	200	0.9740	1.0058	1.0377	0.0943	0
1000	5	4,061,006	10	0.9761	1.0055	1.0374	0.0777	0
100	1	5,706	200	0.9678	1.0045	1.0392	0.1144	3
1000	7	6,063,006	10	0.9752	1.0043	1.0330	0.0832	0

The five best selected FNNs based on the K-PP objective are different than the ones based on the test MSE. Thus, our prediction that test MSE punishes FNNs that result in a better performance of K-PP than K-B&B holds. This is also shown in Figure 5.6. This plot shows that choosing the FNN based on the minimal MSE often gives smaller K-PP normalized objective values.

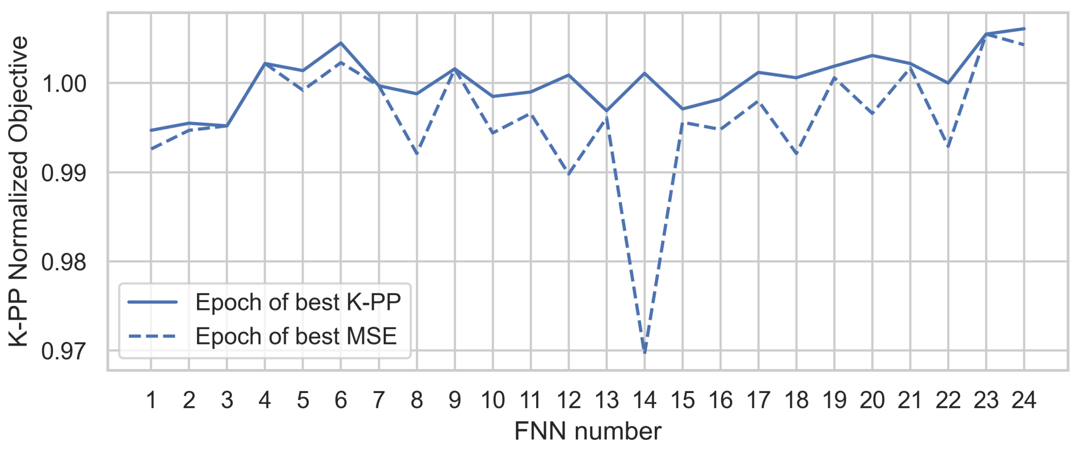


Figure 5.6: The K-PP median *Normalized Objective* values are shown per FNN architecture. Per architecture, the epoch size is chosen based on the lowest MSE and the highest *Normalized Objective*. This plot shows the values of the *Normalized Objective* when the FNN is chosen for the highest value of the *Normalized Objective* (continuous line), or when it is chosen based on the lowest MSE (dotted line). On the x-axis, the FNNs are ordered based on their number of parameters. In Table A.1, the FNN number with its corresponding architecture is shown.

We want the K-PP to perform in such a way that it also does not underperform compared to K-B&B. Thus, for selecting the best FNN, we have the following performance measure:

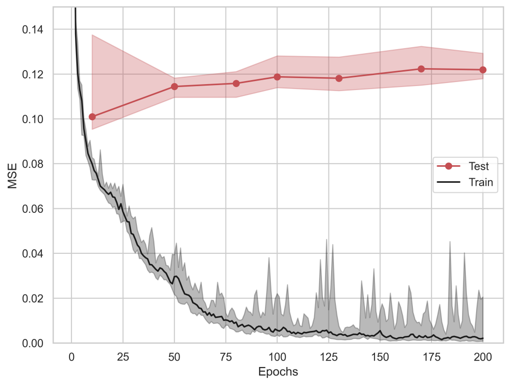
$$K\text{-}PP\ Performance\ Measure = \frac{Norm.\ Obj.\ Median}{1 - Norm\ Obj.\ 10\% + (1 - Norm\ Obj.\ 90\%)^{+}} \tag{5.2}$$

So, if the median of the *Normalized Objective* (i.e. *Norm. Obj. Median*) is large, and the 10% and 90% quantiles of this value (i.e. *Norm. Obj. 10%* and *Norm. Obj. 90%*) do not deviate much from 1, the *K-PP Performance Measure* becomes large. We have that *Norm. Obj. 10%* < 1 for all trained FNNs. Therefore the denominator cannot become 0. The 90% quantile is often more than 1. Therefore, this term is corrected to avoid assigning a lower performance measure to the FNN that gives good results for K-PP. Table 5.5 shows the five best FNN architectures based on this performance measure.

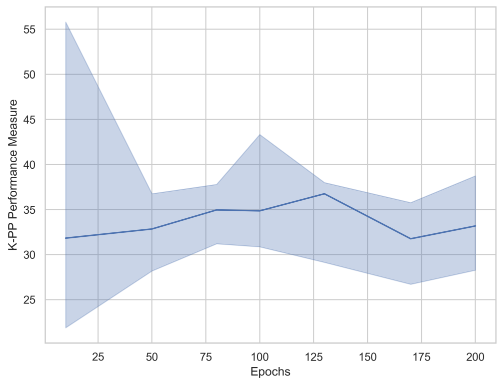
Table 5.5: K-PP results that use the five best FNN frameworks based on the performance measure (5.2).

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
1000	5	4,061,006	10	0.9761	1.0055	1.0374	0.0777	0
1000	7	6,063,006	10	0.9752	1.0043	1.0330	0.0832	0
1000	7	6,063,006	200	0.9740	1.0058	1.0377	0.0943	0
1000	7	6,063,006	50	0.9739	1.0023	1.0329	0.0833	0
500	7	1,531,506	10	0.9736	1.0017	1.0298	0.0673	0

Similar architectures as before are favorable, but with less epochs. The problem with having only a small number of epochs is that if we train the FNN again, the weights and biases of the FNN can differ much, which results in a different prediction model. This is a problem since we want the methodology to be repeatable for others. There are a lot of random factors in training FNNs. In the beginning, the weights and biases are assigned random values. Moreover, in every epoch, different random batch selections are chosen. If we have a small number of epochs, the parameters chosen in the FNN may depend on this. After a larger number of epochs, these effects disappear. This is shown in Figure 5.7. The same FNN was trained ten different times. Particularly the *K-PP Performance Measure* shows that for different training trials, the parameters of the FNN differ much, which results in a broad spectrum of K-PP performances.



(a) FNN MSE



(b) *K-PP Performance Measure*

Figure 5.7: FNN MSE and *K-PP Performance Measure* results for 10 different trained FNNs each having *Width* 1000 and *Depth* 7 over 200 epochs. The bold line represents the median of the respective value, and the shaded area around it represents the minimum and maximum value of the other trials.

In Subfigure 5.7b, we see that for 80 epochs, the median of the *K-PP Performance Measure* (5.2) is relatively high. More importantly, the *K-PP Performance Measure* of the different training trials do not deviate much from each other.

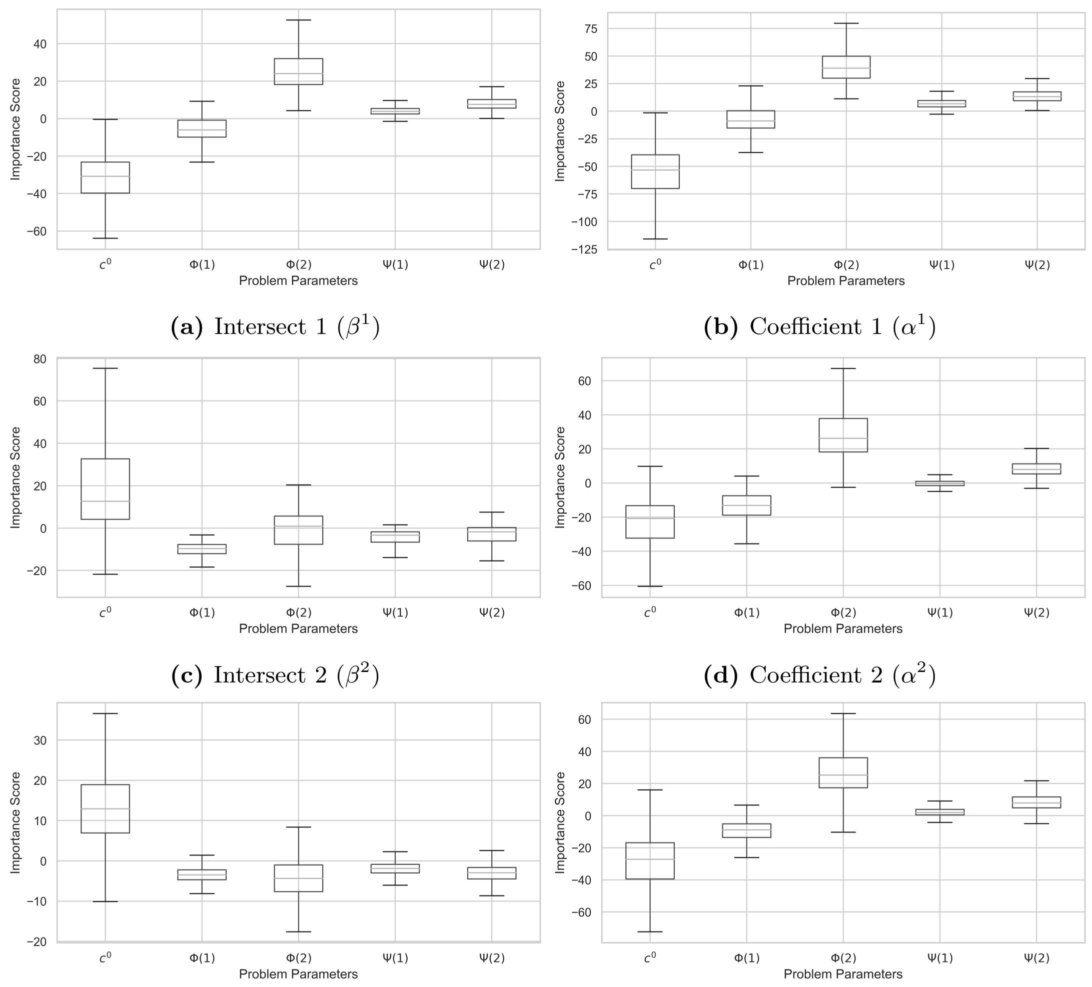
We conclude this section by selecting the best FNN architecture that will be used in the following sections for further performance studies. Based on the *K-PP Performance Measure*, and the reliability of the epoch size studied in Figure 5.7, the FNN that has *Width* 1000, *Depth* 7, and 80 epochs is chosen. Out of the ten trained FNNs with these characteristics, we will select the one for which its *K-PP Performance Measure* is closest to the median.

5.2.4 Feature importance

In this section we discuss the results the DeepLIFT method has on the features of the capital budgeting problem with loans described in Section 3.2. The FNN analyzed in this section has *Width* 1000, *Depth* 7, and 80 epochs, as selected before.

As a reminder, the features of the capital budgeting problem used for predicting the hyperplanes are the nominal costs $\boldsymbol{c}^0 \in [0, 10]^N$, the sensitivity matrix of the cost $\boldsymbol{\Phi} \in [0, 1]^{N\times 2}$, and the sensitivity matrix of the revenue $\boldsymbol{\Psi} \in [0, 1]^{N\times 2}$. N is the number of projects to invest in. In this thesis, this value is equal to 10 for all instances.

Figure 5.8 shows the contribution scores $CS_{x_{NN},y_{NN}}$ of all features x_{NN} on all outputs y_{NN}. Note that the contribution scores of $\boldsymbol{c}^0$ and $\boldsymbol{\Phi}$ are largest, whereas the contribution scores of $\boldsymbol{\Psi}$ are close to 0 for all outputs. This indicates that the sensitivity matrix of the revenues are not important for the partition strategies of problem instances. Moreover, the plots indicate that for high costs, the first subset, i.e. left bottom, and the last subset, i.e. right upper corner, are smaller. Next to that, the values of the second dimension of $\mathbf{\Psi}$ are more important than the ones of the first dimension. This could be explained by the fact that the hyperplane is a function of ξ_2, and therefore the values of this dimension are also more important for predicting the hyperplanes.



(a) Intersect 1 (β^1) **(b)** Coefficient 1 (α^1)

(c) Intersect 2 (β^2) **(d)** Coefficient 2 (α^2)

(e) Intersect 3 (β^3) **(f)** Coefficient 3 (α^3)

Figure 5.8: Boxplots of DeepLIFT contribution score results per ordered hyperplane parameter output for all features of the FNN with *Width* 1000, *Depth* 7, and 80 epochs. The sensitivity matrices $\mathbf{\Phi}$ and $\mathbf{\Psi}$ are divided based on the dimension of the uncertainty set, i.e. $\Phi(1)$, $\Phi(2)$, $\Psi(1)$, $\Psi(2) \in [0,1]^N$ since we have two dimensions in the uncertainty set. The importance scores are based on the predictions made from testing data, since these predictions of the partitions perform well in K-PP.

To test if $\mathbf{\Psi}$ is unimportant for the partition strategy, the same FNN is trained three more times, where separately $\boldsymbol{c}^0$, $\mathbf{\Phi}$, and $\mathbf{\Psi}$ are deleted from the features. The results are given in Table 5.6. This table shows that the K-PP performance when only $\mathbf{\Psi}$ is deleted, does not differ much from the case when no features are deleted. On the contrary, when either $\boldsymbol{c}^0$ or $\mathbf{\Phi}$ are deleted from the features, the K-PP performance based on these FNNs is much worse. Thus, we can conclude that the sensitivity matrix of the revenues $\mathbf{\Psi}$ do not effect the partitions made for instances of the capital budgeting problem. The training duration has not improved with deleting features. This will however be more visible when more features can be deleted.

Table 5.6: Results of K-PP in which the FNN with *Width* 1000, *Depth* 7, and 80 epochs, for different selections of features is used. *NN Features* indicates which features were deleted. *K-PP Perf.* is the *K-PP Performance Measure* (5.2).

NN Features	NN Parameters	K-PP Normalized Objective			K-PP Perf.	NN Train Dur. (s)
		10%	Median	90%		
All	6,053,016	0.9717	1.0058	1.0368	35.5406	1015.29
No $\mathbf{\Psi}$	6,043,006	0.9715	1.0051	1.0351	35.2667	994.24
No $\mathbf{\Phi}$	6,043,006	0.9205	0.9799	1.0165	12.3258	1040.36
No $\boldsymbol{c}^0$	6,053,006	0.9559	0.9935	1.0263	22.5283	995.759

Because the training time has not improved noticeably, and the K-PP algorithm still performs best when all features are used, we will continue using the FNN with *Width* 1000, *Depth* 7, and 80 epochs, trained with all features, in the following sections.

5.3 K-PP performance

In Section 5.3.1, we will compare the performance of the K-PP algorithm with the performance of the K-B&B algorithm based on the objective value over computation time. In Section 5.3.2, we will compare the solutions of the first-stage decisions obtained by both algorithms.

5.3.1 Objective

In this section, we compare the performance of the K-PP algorithm with the one of K-B&B, based on the objective value. As an extra check, we introduce a very simple method that approximately solves the K-adaptability problem. We will compare the performance this method has with the performance of K-PP and K-B&B.

In Figure 5.3, the distribution of the coefficients and intersections of the hyperplanes are shown. For each of these hyperplane parameters, the distribution is tightly concentrated around the median. In the simple method we take as the hyperplanes that split the uncertainty set, the median hyperplanes of the training data, for each test instance. Hence, the median values of the coefficients and intersections per slice are taken. This results in the partition illustrated in Figure 5.9. Based on this partition, the K-adaptability problem for each testing instance will be solved. Comparing the performance of K-PP with this simple method is fair since both methods are constructed from the information of the training data.

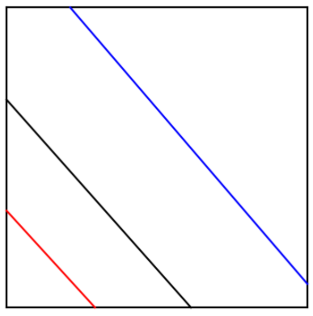


Figure 5.9: Partition used in the simple method based on median hyperplane values of training data.

In Section 5.2.3, the FNN that performs best in the K-PP algorithm was selected. This FNN has *Width* 1000, *Depth* 7, and was trained with 80 epochs. Hence, the FNN has 7 hidden layers, each consisting of 1000 nodes. The performance over computation time of the K-PP algorithm compared to the performance of the K-B&B algorithm, and the simple method, are shown in Figure 5.10. The computation time of the K-PP method and the simple method are both less than a second. So in reality, the line should have stopped here. However, for illustration, the objective result is shown over the whole runtime horizon of the K-B&B algorithm.

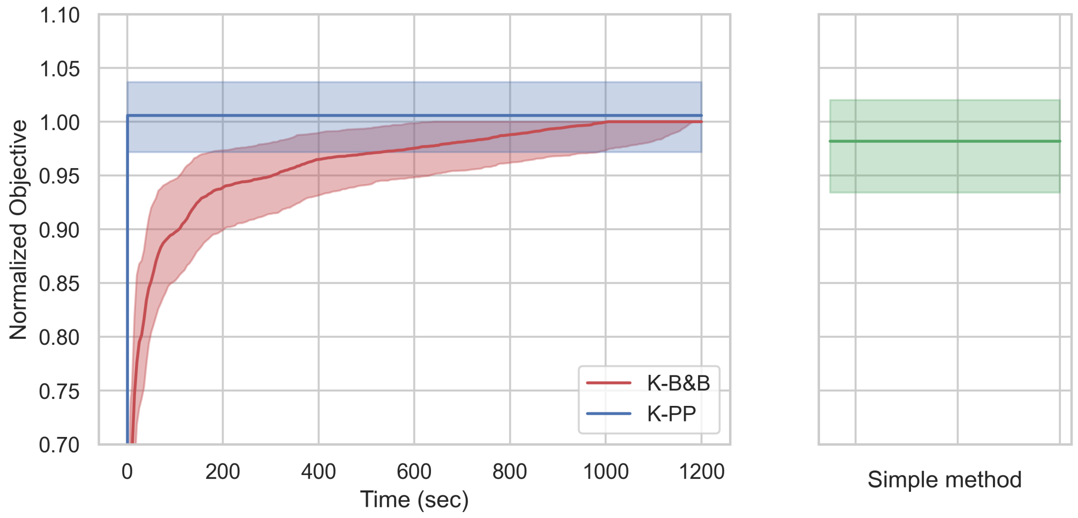


Figure 5.10: Performance of K-PP versus K-B&B versus the simple method over computation time. In the left plot, the incumbent objective results are shown of the K-PP algorithm and of the K-B&B algorithm. In the right plot, the objective of the simple method is shown. Both plots share the same y-axis. All (incumbent) objective values are normalized based on the K-B&B objective value found after 20 minutes. The algorithms are performed on the instances of the testing data set. The bold line represents the median of the respective value, and the shaded area around it is the 80% confidence interval. This K-PP algorithm uses an FNN with *Width* 1000 and *Depth* 7 with 80 epochs.

This figure shows that the K-PP algorithm is computationally very efficient and gives high quality solutions. The objective value of many instances is even better than the one obtained by K-B&B after 20 minutes. The simple method is also capable of finding good solutions. However, it does underperform compared to K-B&B more often than K-PP.

The time it takes to train the FNN used in K-PP is omitted from the computation time since it only has to be done once for all testing instances. However, this can still take a long time. For the FNN used in Figure 5.10, the training time was 16.5 minutes. In Figure 5.11, we again compare the performance of the K-PP algorithm with the K-B&B algorithm and the simple method. Here, the FNN used in the K-PP algorithm is the FNN that performs best based on both the *K-PP Performance Measure* (5.2) and its training duration. This results in the following performance measure:

$$K\text{-}PP\ Speed\ Performance\ Measure = \frac{K\text{-}PP\ Performance\ Measure}{FNN\ Train\ Duration} \tag{5.3}$$

This measure punishes FNNs that have a high training time compared to the performance it has in the K-PP algorithm. Out of all FNNs trained, the FNN with *Width* 50, *Depth* 1, and 10 epochs, has the highest *K-PP Speed Performance Measure* (5.3) value.

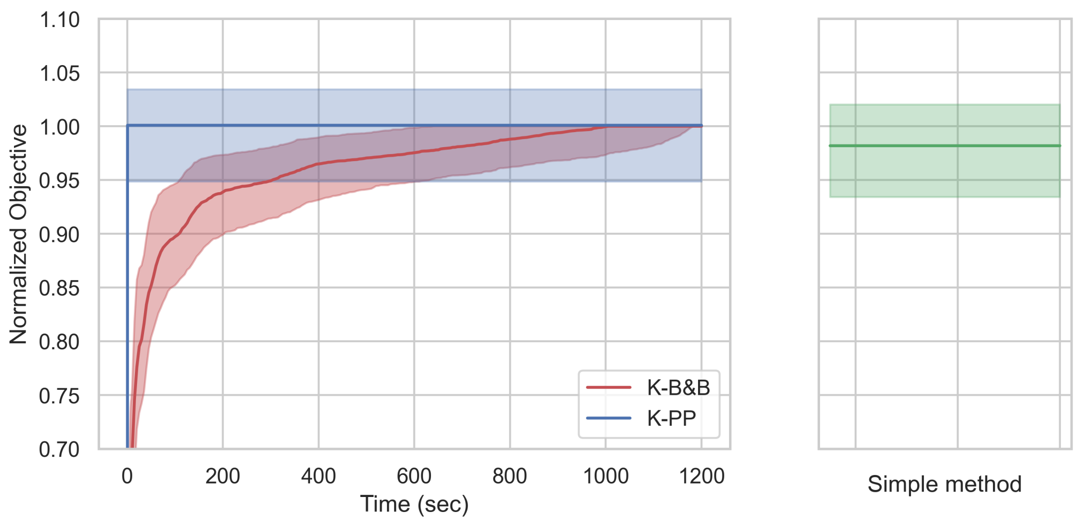


Figure 5.11: Performance of K-PP versus K-B&B versus the simple method over computation time. This K-PP uses an FNN selected based on the *K-PP Speed Performance Measure* (5.3). This FNN has *Width* 50, *Depth* 1, and was trained with 10 epochs.

The K-PP algorithm with this FNN still outperforms the K-B&B algorithm and the simple method on average. The training time of the FNN is 1.13 seconds, which is considerably less than the FNN used in Figure 5.10. However, as discussed in Section 5.2.3, FNNs trained with only a few epochs can give considerably different results when trained again. For this problem, training the FNN with a large epoch size overfits the data easily. Therefore, if it is important to keep training time low, selecting a low epoch size is a good approach since this often gives good results.

We also compare the performance of K-PP with K-B&B and the simple algorithm, where K-PP uses the FNN that has the lowest *K-PP Performance Measure* (5.2). This is the FNN with *Width* 1000, *Depth* 1, and 1500 epochs.

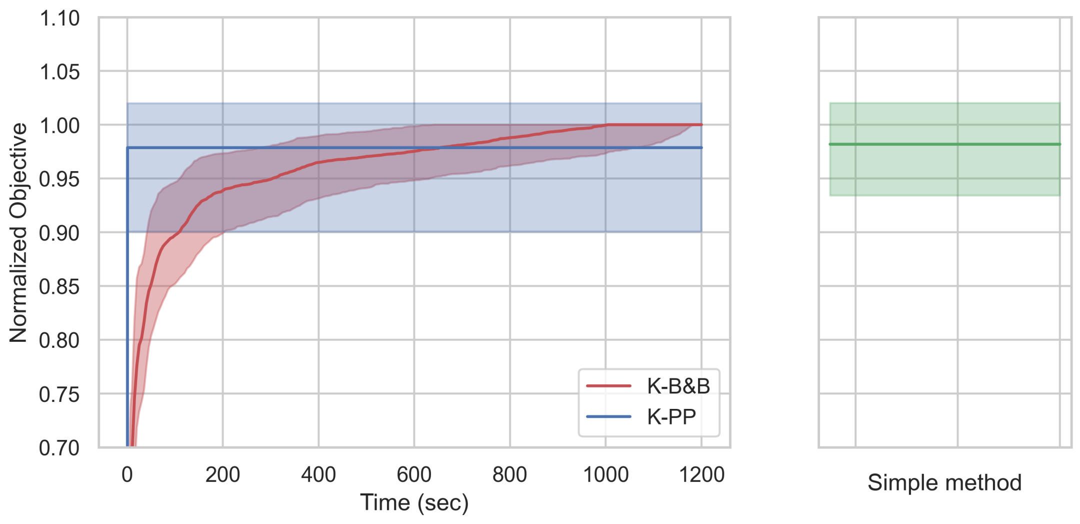


Figure 5.12: Performance of K-PP versus K-B&B versus the simple method over computation time. This K-PP uses an FNN selected based on the lowest *K-PP Performance Measure* (5.3). This FNN has *Width* 1000, *Depth* 1, and was trained with 1500 epochs.

The performance of the K-PP algorithm that uses this FNN is even worse than the performance of the simple algorithm. Hence, the FNN design parameters have to be selected with caution.

We conclude that the K-PP algorithm performs surprisingly well compared to the K-B&B algorithm. This is especially interesting since the prediction method used in the K-PP algorithm is trained on data that has been build similarly as the testing data. So, one would expect that K-PP would at most perform as good as K-B&B, not better. However, we have to admit that the simple algorithm also often outperforms the K-B&B algorithm, but not as well.

5.3.2 First-stage decisions

For the two stages in which decisions need to be made, the first-stage decisions are most important. These are the decisions made closest in time from now. In practice, the purpose of deciding on the second-stage decisions is to find first-stage decisions that are feasible and optimal for what happens in the future. Moreover, if we are in the period after the uncertain scenario has occurred, so in the second-stage, the decision maker may always fully adapt the second-stage decision to the actual scenario. Solving this problem is straightforward. All elements that made the problem difficult to solve, the uncertainty and the combination of first- and second-stage, are resolved. Therefore, the solution of the first-stage is the most important solution of the two-stage robust optimization problem.

In this section we analyze the first-stage decisions found by K-PP and by K-B&B. This is done since we want to discover how much the first-stage decisions depend on the partitions made. It could be the case that multiple partition strategies give the same decisions. If this is the case, then finding one best optimal partition may not be the most important goal for solving two-stage robust optimization problems. Therefore, we study the (dis)similarities of the first-stage decisions of the two methods.

As a preliminary check, we analyze the first-stage decisions found by the K-B&B algorithm over its computation time. This is done by comparing the first-stage decision found after the maximum runtime with the first-stage decisions found in the incumbent solutions. Each incumbent solution relates to a partition that has been rejected by a better partition in the algorithm. Table 5.7 shows after what computation time the first-stage decision that has been selected, has been found in the algorithm. It also shows the number of times the selected first-stage decision was found in the incumbent solutions. The first-stage decisions of the capital budgeting problem consist of two elements. The first one is $x_0 \in \mathbb{R}$, the decision on the loan amount in the first-stage. The second one is $\boldsymbol{x} \in \{0,1\}^N$, the decision in which project i to invest in, for $i \in \{1, ..., N\}$. We noticed that more often than both x_0 and $\boldsymbol{x}$, solely $\boldsymbol{x}$ has already been found in any of the incumbent solutions of the algorithm. Therefore we separated these two cases. Moreover, the decisions $\boldsymbol{x}$ are more important than x_0, since the loan can be seen as a side-issue.

Table 5.7: K-B&B analysis on incumbent solutions. *First Time Found* is the computation time after which the first-stage decisions selected, have been found for the first time in the K-B&B algorithm. *Num. Times Found* is the number of times this first-stage decision was found any of the incumbent solutions. *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the respective value of the 1000 test instances.

	First Time Found (s)			Num. Times Found		
	10%	Median	90%	10%	Median	90%
$(x_0, \boldsymbol{x})$	400.722	961.649	1177.4	1	1	2
$\boldsymbol{x}$	190.293	905.998	1171.92	1	1	3

This table shows that $\boldsymbol{x}$ is found earlier in the algorithm, and also more often than both $(x_0, \boldsymbol{x})$. However, the values of *First Time Found* are distributed spaciously over the runtime horizon (i.e. 1200 seconds). Hence, we cannot say anything about the moment the first-stage decisions remain unchanged. *Num. Times Found* tells us that the partition is important for the first-stage decisions since its values do not often differ from 1. Hence for many incumbent solutions, i.e. other partitions, other first-stage decisions are optimal.

Now, we compare the first-stage decisions found by K-PP and by K-B&B for the test instances. Table 5.8 shows the percentage of instances where both x_0 and $\boldsymbol{x}$ are equal for K-PP and K-B&B. It also shows the cases where only $\boldsymbol{x}$ is the same for both algorithms. The results are shown separately for the instances for which its K-PP objective value was at least better than that of K-B&B, and for the instances K-PP performed worse. Hence, this is a separation between instances for which K-PP found better and worse partitions than K-B&B, respectively.

Table 5.8: Number of instances that for both K-B&B and K-PP have the same first-stage decisions. *K-PP Normalized Obj. > 1* are the instances for which the objective found by K-PP is better than found by K-B&B. *K-PP Normalized Obj. < 1* are the instances for which the K-PP objective is worse than that of K-B&B.

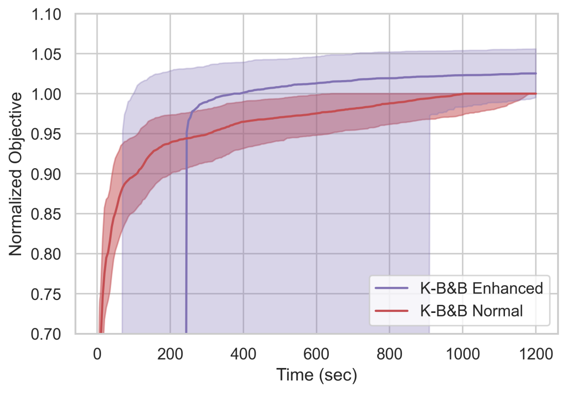
	Total Instances (%)	Same $(x_0, \boldsymbol{x})$ (%)	Same $\boldsymbol{x}$ (%)
All	100	0.6993	4.995
K-PP Normalized Obj. > 1	59.4406	0.6723	4.7059
K-PP Normalized Obj. < 1	40.5594	0.7389	5.4187

Table 5.8 shows that for a very small number of instances the first-stage decisions of K-PP are equal to the ones of K-B&B. For instances that perform better with K-PP, the first-stage decisions are less often equal than for the instances that perform worse. However, this statement is weak since this difference is caused by a few instances only.

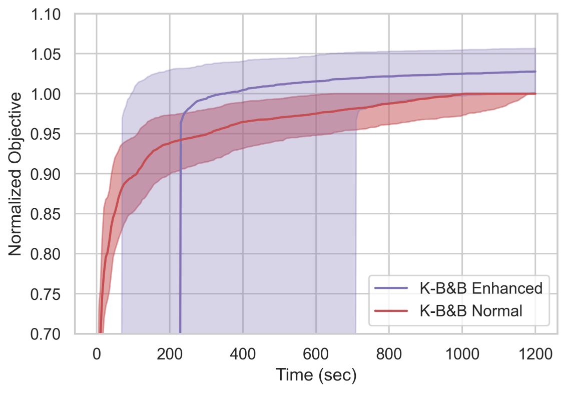
We can conclude that the first-stage decisions change often for different partitions. Better partitions found by K-B&B do not guarantee a higher possibility of the first-stage decisions to remain unchanged once the partition improves more.

5.4 K-B&B with K-PP initialization

In this section we discuss the performance of the K-B&B algorithm when we initialize with K-PP solutions, as introduced in Section 4.5. The FNN used in this section is the FNN with *Width* 1000, *Depth* 7, and 80 epochs. This is the FNN that was selected in Section 5.2.3. We call the K-B&B method, where we use K-PP as initialization, the Enhanced K-B&B algorithm. Figure 5.13 shows the performance of the Enhanced K-B&B algorithm and the performance of the normal K-B&B algorithm, for the 1000 test instances. All incumbent objective values are normalized based on the objective value found with the normal K-B&B algorithm after 20 minutes. As to be expected, the figures show that the Enhanced K-B&B converges faster to optimality than the normal K-B&B algorithm. The figures also show that for the Enhanced K-B&B algorithm, the time until the first incumbent solution is found, can take a long time. For 6.3% of the instances, no incumbent solution was found within the maximum computation time of 20 minutes. For these instances, the solution of the Enhanced K-B&B algorithm is the solution of K-PP.



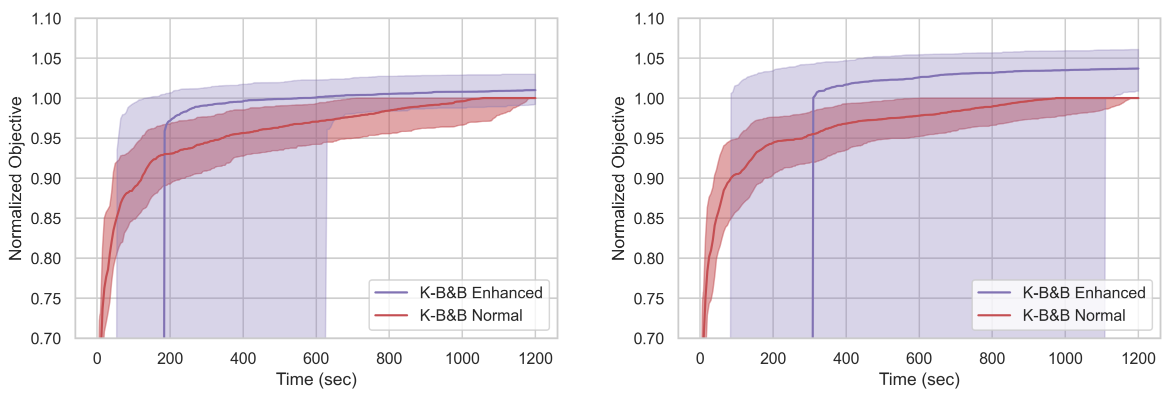
(a) All instances (100%)



(b) Instances with incumbent solutions (93.7%)

Figure 5.13: Performance of Enhanced K-B&B versus the normal K-B&B algorithm over computation time. In Subfigure 5.13a, the performance of all data instances are shown. In 5.13b, only the instances for which the Enhanced K-B&B algorithm obtains any incumbent solutions within 20 minutes are shown. All incumbent objective values are normalized based on the normal K-B&B objective value found after 20 minutes. The algorithms are performed on the instances of the testing data set. The bold line represents the median of the respective value. The shaded area around this line is the 80% confidence interval.

Finding the first incumbent solution can take a very long time since many branches are cut, which would have been an incumbent solution in the normal K-B&B algorithm. This is particularly the case when the solution of K-PP is better than that of K-B&B. Figure 5.14 shows the performance of the Enhanced K-B&B algorithm where the instances are separated based on their K-PP performance. These figures show that for the instances where the K-PP normalized objective is smaller than 1, an incumbent solution is found sooner than when this value is larger than 1.



(a) K-PP Normalized Objective < 1 **(b)** K-PP Normalized Objective > 1

Figure 5.14: Performance of Enhanced K-B&B versus the normal K-B&B algorithm over computation time. For the instances in Subfigure 5.14a, the objective value of the normal K-B&B is better than that of K-PP. For the instances in Subfigure 5.14b, the objective of K-PP is better.

With these extra cuts, the Enhanced algorithm solves less unnecessary MILPs since we are earlier steered towards the correct nodes, and therefore partitions. Thus, within the same computation time, we can get deeper in the tree. This is measured by the number of scenarios $\boldsymbol{\xi}$ allocated in any of the subsets $\dot{\Xi}_1, ..., \dot{\Xi}_K$. Table 5.9, gives an overview of the results. This table shows that indeed, the Enhanced K-B&B algorithm is further down in the tree than the normal K-B&B algorithm. Therefore the Enhanced algorithm will reach optimality sooner.

Table 5.9: The depth of the branch-and-bound tree measured with the number of scenarios allocated in the subsets by the K-B&B algorithm and by the Enhanced K-B&B algorithm.

	Num. Scenarios K-B&B			Num. Scenarios Enh. K-B&B		
	10%	Median	90%	10%	Median	90%
All	10	11	13	13	16	20
K-PP Norm. Obj. > 1	10	11	12	13	16	19
K-PP Norm. Obj. < 1	10	12	14	14	16	21.3

5.5 K-PP performance for K=3

In this section, we will conduct analysis for the case of K=3. Hence, now the uncertainty set of the capital budgeting problem is divided into 3 parts, instead of 4. Figure 5.2 shows that for K=3, the K-B&B algorithm converges faster to optimality than in the case of K=4. Moreover, Table 5.1 shows that for K=3, we have instances solved exactly within 20 minutes. Therefore, the data used for training the prediction model used in K-PP, is based on more accurate optimality data than the data we used for K=4.

Figure 5.15 shows the distributions of the parameters of the two hyperplanes that split the uncertainty set. These distributions behave similarly as the ones for K=4 (Figure 5.3). Therefore, the same selection of design parameters of the FNNs are chosen again. Thus, the different FNNs listed in Table 5.2 are trained.

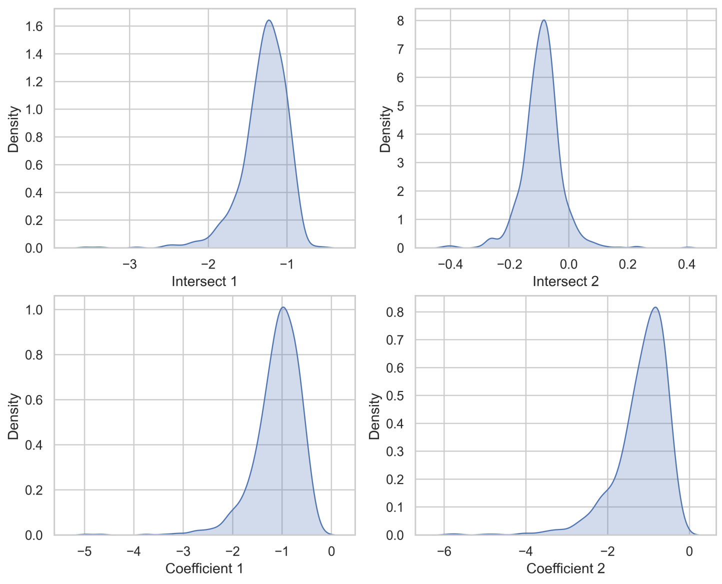


Figure 5.15: Distributions of the coefficients and intersections of the two ordered hyperplanes that split the uncertainty set into three parts.

The best FNN for K-PP is selected based on the *K-PP Performance Measure* (5.2). The best five FNNs are listed in Table 5.10. Based on this information, the FNN with *Width* 500, *Depth* 7, and 100 epochs is selected. This table also shows that the *Normalized Objective* (5.1) is lower than the *Normalized Objective* for K=4.

Table 5.10: Results of K-PP that use the five best FNN frameworks based on the *K-PP Performance Measure* (5.2) for K=3. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the normalized objective of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
500	7	1530504	100	0.9621	0.9860	1.0000	0.3144	0
500	3	528504	500	0.9621	0.9860	1.0003	0.2905	0
1000	3	2057004	500	0.9621	0.9857	1.0000	0.3204	0
500	7	1530504	200	0.9617	0.9858	1.0000	0.3241	0
500	3	528504	1000	0.9614	0.9853	1.0003	0.2930	0

For analyzing the performance of the K-PP algorithm, we use the K-B&B performance and the simple method. The simple method is the same one as illustrated in Section 5.3.1. The median coefficients and intersections of the hyperplanes that split the uncertainty set of the training instances are used, for the partitions of the testing instances. This results in the partition illustrated in Figure 5.16.

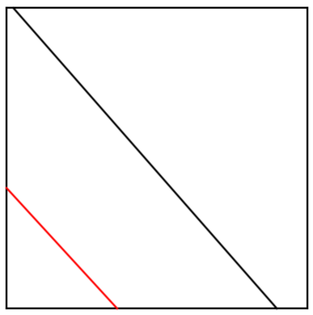


Figure 5.16: Partition used in the simple method based on median hyperplane values of training data for K=3.

Figure 5.17 shows the performance of the K-PP algorithm using the FNN selected based on the performance measure. As comparison, the K-B&B performance and the performance of the simple method are given.

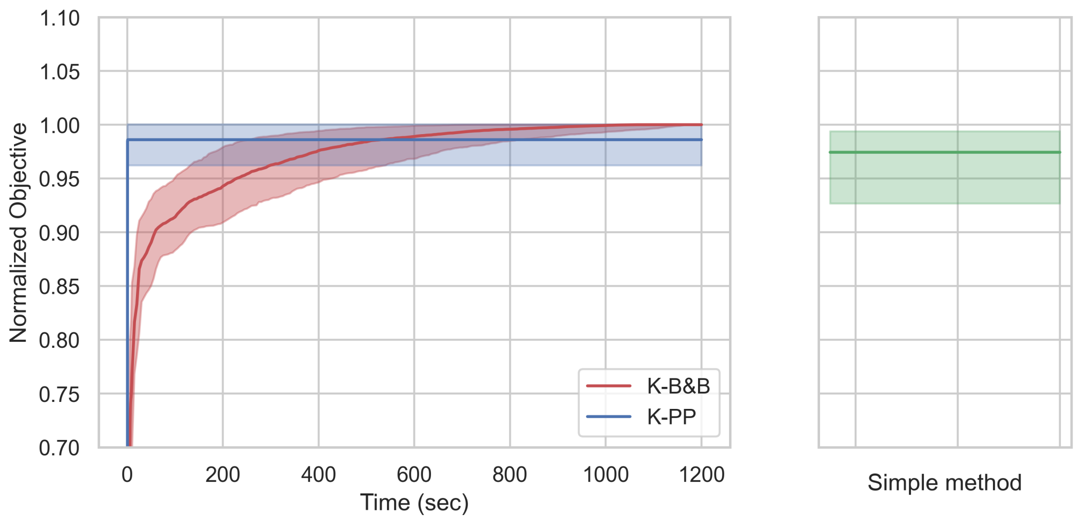


Figure 5.17: Performance of K-PP versus K-B&B versus the simple method over computation time for K=3. In the left plot, the incumbent objective results are shown of the K-PP algorithm and of the K-B&B algorithm. In the right plot, the objective of the simple method is shown. Both plots share the same y-axis. All (incumbent) objective values are normalized based on the K-B&B objective value found after 20 minutes. The algorithms are performed on the instances of the testing data set. The bold line represents the median of the respective value, and the shaded area around it is the 80% confidence interval. This K-PP uses an FNN selected based on the *K-PP Performance Measure* (5.2). This FNN has *Width* 500, *Depth* 7, and was trained with 100 epochs.

Figure 5.17 shows that K-PP performs better than the simple method since the 80% confidence interval of the simple method is wider and lies lower than that of K-PP. Compared to the case of K=4, for K=3, the K-B&B algorithm more often gives better objective values. This is to be expected since the K-B&B algorithm is closer to optimality within 20 minutes. However, K-PP still gives a good approximation and is computationally much more efficient than K-B&B.

6 | Conclusion

In this thesis, an approach to approximately solve the K-adaptability problem of two-stage robust optimization problems is introduced. With K-adaptability, the uncertainty set of the problem is partitioned into K subsets. For each of these subsets, a corresponding second-stage decision is solved. Our approach is called the K-adaptability partition prediction (K-PP) algorithm. This method uses a neural network to predict the partitioned uncertainty set of the respective two-stage robust optimization problem. Based on this partition, the first- and second-stage decisions of the two-stage robust optimization problem are solved. This method is computationally very efficient, i.e. the runtime is less than a second. The data used for training the neural network is obtained by solving many instances of the problem with the K-adaptability branch-and-bound (K-B&B) algorithm (Subramanyam et al., 2019). In this thesis, we focused on the capital budgeting problem with loans (Section 3.2). This problem is a representative problem for other two-stage robust optimization problems due to the uncertain element in the objective and in the constraints. The dimension of the uncertainty set for this problem is moreover independent of the instance size. This is a favorable feature since this allows us to keep a clear overview of the partitioned uncertainty set.

In the introduction, 4 research questions were introduced. The answers to these questions are the following:

1. *How can we predict the partition of the uncertainty set of a two-stage robust optimization problem?*: For the capital budgeting problem, the subsets of the uncertainty set are separated by hyperplanes. Therefore, with a neural network, the parameters of the hyperplanes are predicted. For training the neural network, many instances of the capital budgeting problem were solved. This is the output of the prediction model. The input of this model is the collection of elements in the capital budgeting problem that differ from one instance to another. These are for example the costs of the instances. Section 4.3 details the methods we used for selecting the input and the output of the model. Section 4.4.1 shows which neural network is used.

2. *How will the objective and decision variables computed with K-PP differ from the ones computed with K-B&B?*: For obtaining training data for the neural network, the K-B&B algorithm was stopped after 20 minutes. After this computation time, the objective values are converging, but have not reached optimality yet. To our surprise, the K-PP objective values for $K{=}4$ are on average better than the ones found by K-B&B. This means that with the neural network, trained with non-optimal K-B&B results, better partitions are found than with K-B&B itself. In Section 5.3.1, these results are shown. Section 5.3.2 shows that the first-stage decisions found by K-PP and K-B&B are rarely equal. We have not researched the similarity of second-stage decisions, however, as explained in Section 5.3.2, first-stage decisions are of higher importance in two-stage robust optimization.

3. *How can the results of K-PP be used as initialization of K-B&B to derive exact solutions faster?*: The objective value found by K-PP is used to cut branches of the branch-and-bound tree. The centroids of the predicted partitions are used as initial scenarios in the subsets. In Section 5.4, we show that it can take a longer time for the K-B&B with initialization to find an incumbent solution. However, within 20 minutes, the algorithm has reached deeper into the tree. It will therefore run to optimality faster than the normal K-B&B algorithm.

4. *Which parameters of the problem are important for building the partitions?*: This question is answered with the results of the feature importance method DeepLIFT (Shrikumar et al., 2017). Section 5.2.4 shows that for the capital budgeting problem, $\mathbf{\Psi}$ is least important for forming the hyperplanes that split the uncertainty set. When training the same neural network again, but without $\mathbf{\Psi}$, the performance of K-PP was a bit worse. However, when we trained the neural network again, but then instead of $\mathbf{\Psi}$ missing, other parameters were taken out, the K-PP performs considerably worse. Hence, this indicates that $\mathbf{\Psi}$ is indeed less important for the partitioning technique.

A few issues remain to be resolved in future research. The biggest issue is how to find a generalized form of the prediction model, that finds the partition of the uncertainty set of any two-stage robust optimization problem, for any instance size of the problem, and for any dimension of the uncertainty set. This requires that the dimensions of the input and the output of the neural network change per instance.

Moreover, in reality it is difficult to train the prediction model. The K-B&B model is solved many times to obtain the data needed for prediction, which is computationally very expensive. Other methods for obtaining training data need to be discovered. As we saw in Section 5.3.1, for data not based on optimal solutions, we often find better partitions. Perhaps this is also the case if we use heuristics instead of exact algorithms to solve two-stage robust optimization problems. For inventory problems of companies with a long history of available data, we could gather information on the decisions made, and how good or bad these were at that time.

Finally, feature importance detection will especially be interesting if the prediction model deals with multiple different two-stage robust optimization problems. These problems can then be written in the general form given in the remark of Section 4.3.1. Feature importance methods can tell us which elements of the two-stage robust optimization problem do have an effect on the partition, and which ones do not.

We conclude this thesis by stating that still many elements can be improved in this research. However, this thesis shows the advantages of using machine learning techniques to enhance solution algorithms to solve two-stage robust optimization problems.

Bibliography

Ben-Tal, A., El Ghaoui, L., and Nemirovski, A. (2009). *Robust Optimization*, volume 28. Princeton University Press.

Ben-Tal, A., Goryashko, A., Guslitzer, E., and Nemirovski, A. (2004). Adjustable robust solutions of uncertain linear programs. *Mathematical Programming*, 99(2):351–376.

Bertsimas, D. and Caramanis, C. (2010). Finite adaptability in multistage linear optimization. *IEEE Transactions on Automatic Control*, 55(12):2751–2766.

Bertsimas, D. and de Ruiter, F. J. (2016). Duality in two-stage adaptive linear optimization: Faster computation and stronger bounds. *INFORMS Journal on Computing*, 28(3):500–511.

Bertsimas, D. and Dunning, I. (2016). Multistage robust mixed-integer optimization with adaptive partitions. *Operations Research*, 64(4):980–998.

Bertsimas, D., Goyal, V., and Sun, X. A. (2011). A geometric characterization of the power of finite adaptability in multistage stochastic and adaptive optimization. *Mathematics of Operations Research*, 36(1):24–54.

Birbil, Ş. İ. (2019). Erasmus School of Economics. Lecture Notes: Machine Learning, Neural Networks.

Bishop, C. M. (2006). *Pattern recognition and machine learning.* Springer.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Charnes, A. and Cooper, W. W. (1959). Chance-constrained programming. *Management Science*, 6(1):73–79.

Chollet, F. et al. (2015). Keras.

Fisher, A., Rudin, C., and Dominici, F. (2019). All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research*, 20(177):1–81.

Gevrey, M., Dimopoulos, I., and Lek, S. (2003). Review and comparison of methods to study the contribution of variables in artificial neural network models. *Ecological Modelling*, 160(3):249–264.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press.

Gurobi Optimization, L. (2020). Gurobi optimizer reference manual.

Guslitzer, E. (2002). Uncertainty-immunized solutions in linear programming. Master's thesis, Technion, Israeli Institute of Technology.

Hanasusanto, G. A., Kuhn, D., and Wiesemann, W. (2015). *K*-adaptability in two-stage robust binary programming. *Operations Research*, 63(4):877–891.

Ivakhnenko, A. G. (1968). The group method of data of handling; a rival of the method of stochastic approximation. *Soviet Automatic Control*, 13:43–55.

Lisboa, P. J. and Taktak, A. F. (2006). The use of artificial neural networks in decision support in cancer: a systematic review. *Neural Networks*, 19(4):408–415.

Postek, K. and den Hertog, D. (2016). Multistage adjustable robust mixed-integer optimization via iterative splitting of the uncertainty set. *INFORMS Journal on Computing*, 28(3):553–574.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Schneider, J. and Kirkpatrick, S. (2007). *Stochastic Optimization*. Springer Science & Business Media.

Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. *arXiv preprint arXiv:1704.02685*.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Subramanyam, A., Gounaris, C. E., and Wiesemann, W. (2019). K-adaptability in two-stage mixed-integer robust optimization. *Mathematical Programming Computation*, pages 1–32.

Suthaharan, S. (2016). Support vector machine. In *Machine learning models and algorithms for big data classification*, pages 207–235. Springer.

Vayanos, P., Kuhn, D., and Rustem, B. (2011). Decision rules for information discovery in multi-stage stochastic programming. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 7368–7373. IEEE.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer.

A | FNN numbers

Table A.1: FNN numbers with their respective architectures used in Figure 5.6. The FNN architectures are ordered based on their amount of parameters *Pars.*.

FNN number	Width	Depth	Pars.
1	10	1	576
2	10	3	796
3	10	5	1016
4	10	7	1236
5	50	1	2856
6	100	1	5706
7	50	3	7956
8	200	1	11406
9	50	5	13056
10	50	7	18156
11	100	3	25906
12	500	1	28506
13	100	5	46106
14	1000	1	57006
15	100	7	66306
16	200	3	91806
17	200	5	172206
18	200	7	252606
19	500	3	529506
20	500	5	1030506
21	500	7	1531506
22	1000	3	2059006
23	1000	5	4061006
24	1000	7	6063006

B | Trained neural networks

Table B.1: All trained feedforward neural networks for different architectures that predict the hyperplanes for four partitions (K=4). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* is the number of parameters in the FNN. For FNNs with more than 5 hidden layers, the model was trained up to 1000 epochs due to the long runtime of models with many parameters.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
10	1	576	10	0.1641	0.1643	0.9907
			50	0.1362	0.1229	3.7528
			100	0.1361	0.1158	7.6811
			200	0.1313	0.1073	17.3126
			500	0.1319	0.1050	44.1419
			1000	0.1346	0.1049	88.2485
			1500	0.1342	0.1049	136.2694
			2000	0.1375	0.1045	182.8587
10	3	796	10	0.1391	0.1292	1.5039
			50	0.1276	0.1076	6.8943
			100	0.1223	0.0997	12.8211
			200	0.1248	0.0934	24.9110
			500	0.1288	0.0879	59.9835
			1000	0.1298	0.0858	115.9884
			1500	0.1326	0.0846	173.6836
			2000	0.1377	0.0834	239.1201
10	5	1016	10	0.1320	0.1211	2.0942
			50	0.1234	0.1009	11.2693
			100	0.1234	0.0914	20.4884
			200	0.1272	0.0842	37.9750
			500	0.1340	0.0793	89.3295
			1000	0.1550	0.0754	176.4505
10	7	1236	10	0.1360	0.1192	2.8551
			50	0.1208	0.1030	14.7521
			100	0.1170	0.0924	26.2353
			200	0.1276	0.0818	50.3218
			500	0.1415	0.0728	121.5394
			1000	0.1393	0.0689	239.2660
50	1	2856	10	0.1398	0.1283	1.1312
			50	0.1364	0.1059	4.8672
			100	0.1229	0.0895	9.8277
			200	0.1113	0.0749	20.1343
			500	0.1077	0.0625	51.1342
			1000	0.1122	0.0592	103.0620
			1500	0.1216	0.0573	155.6066

Continued on next page

Table B.1: All trained feedforward neural networks for different architectures that predict the hyperplanes for four partitions (K=4). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* is the number of parameters in the FNN. For FNNs with more than 5 hidden layers, the model was trained up to 1000 epochs due to the long runtime of models with many parameters.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
			2000	0.1108	0.0559	208.5223
50	3	7956	10	0.1312	0.1190	1.6535
			50	0.1090	0.0690	7.8910
			100	0.1093	0.0509	15.1384
			200	0.1314	0.0366	29.4246
			500	0.1434	0.0262	72.4078
			1000	0.1564	0.0211	144.4761
			1500	0.1566	0.0191	217.0863
			2000	0.1572	0.0196	289.8535
50	5	13056	10	0.1234	0.1047	2.1045
			50	0.1120	0.0522	10.7377
			100	0.1209	0.0359	19.9273
			200	0.1353	0.0217	38.3777
			500	0.1446	0.0152	93.4349
			1000	0.1525	0.0112	187.2637
50	7	18156	10	0.1237	0.1035	2.8355
			50	0.1125	0.0549	14.2921
			100	0.1251	0.0337	25.5463
			200	0.1363	0.0217	47.7849
			500	0.1461	0.0124	113.6499
			1000	0.1478	0.0096	222.8682
100	1	5706	10	0.1454	0.1230	2.2834
			50	0.1120	0.0750	12.1683
			100	0.1128	0.0619	22.2474
			200	0.1272	0.0528	42.3711
			500	0.1395	0.0423	102.8051
			1000	0.1551	0.0389	202.2680
			1500	0.1689	0.0376	301.8631
			2000	0.1783	0.0370	400.9644
100	3	25906	10	0.1213	0.0914	3.5374
			50	0.1199	0.0379	17.9490
			100	0.1268	0.0207	32.0564
			200	0.1321	0.0120	60.4542
			500	0.1411	0.0067	146.3315
			1000	0.1464	0.0048	288.5546
			1500	0.1524	0.0048	433.4165
			2000	0.1514	0.0045	576.1043
100	5	46106	10	0.1366	0.0916	4.3616
			50	0.1228	0.0259	23.3216
			100	0.1295	0.0128	42.4563
			200	0.1274	0.0081	80.1491

Continued on next page

Table B.1: All trained feedforward neural networks for different architectures that predict the hyperplanes for four partitions (K=4). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* is the number of parameters in the FNN. For FNNs with more than 5 hidden layers, the model was trained up to 1000 epochs due to the long runtime of models with many parameters.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
			500	0.1327	0.0033	192.7747
			1000	0.1322	0.0026	377.9640
100	7	66306	10	0.1141	0.0897	6.1556
			50	0.1228	0.0308	32.9244
			100	0.1314	0.0103	59.7943
			200	0.1257	0.0073	113.4094
			500	0.1244	0.0036	270.6758
			1000	0.1223	0.0017	533.3798
200	1	11406	10	0.1302	0.1104	4.0428
			50	0.1108	0.0608	21.0759
			100	0.1206	0.0466	38.0864
			200	0.1294	0.0354	72.1392
			500	0.1499	0.0278	174.2236
			1000	0.1612	0.0240	342.5780
			1500	0.1743	0.0229	511.9418
			2000	0.1773	0.0223	682.5223
200	3	91806	10	0.1121	0.0827	12.1046
			50	0.1150	0.0173	61.5675
			100	0.1174	0.0083	116.9390
			200	0.1217	0.0057	226.7858
			500	0.1171	0.0016	559.1711
			1000	0.1166	0.0014	1113.3787
			1500	0.1161	0.0013	1666.2155
			2000	0.1154	0.0008	2221.6160
200	5	172206	10	0.1053	0.0809	23.4375
			50	0.1128	0.0157	122.6109
			100	0.1126	0.0093	237.2962
			200	0.1128	0.0040	464.7284
			500	0.1099	0.0021	1152.2380
			1000	0.1112	0.0017	2295.7450
200	7	252606	10	0.1068	0.0841	34.7863
			50	0.1208	0.0146	181.0250
			100	0.1182	0.0083	353.9400
			200	0.1212	0.0077	699.1839
			500	0.1200	0.0017	1732.8545
			1000	0.1238	0.0010	3454.6105
500	1	28506	10	0.1110	0.0907	10.7701
			50	0.1185	0.0443	54.7463
			100	0.1197	0.0284	102.9426
			200	0.1358	0.0165	198.6092
			500	0.1508	0.0097	487.0060

Continued on next page

Table B.1: All trained feedforward neural networks for different architectures that predict the hyperplanes for four partitions (K=4). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* is the number of parameters in the FNN. For FNNs with more than 5 hidden layers, the model was trained up to 1000 epochs due to the long runtime of models with many parameters.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
			1000	0.1615	0.0069	967.3379
			1500	0.1702	0.0071	1449.0688
			2000	0.1718	0.0067	1929.0509
500	3	529506	10	0.1067	0.0720	33.8441
			50	0.1169	0.0115	173.7767
			100	0.1061	0.0046	338.8918
			200	0.1071	0.0042	668.8068
			500	0.1074	0.0018	1658.2271
			1000	0.1088	0.0013	3315.5244
			1500	0.1130	0.0012	4970.5509
			2000	0.1141	0.0007	6626.3518
500	5	1030506	10	0.1014	0.0783	54.0095
			50	0.1173	0.0100	274.3427
			100	0.1132	0.0043	537.4205
			200	0.1144	0.0016	1061.1723
			500	0.1212	0.0009	2658.6254
			1000	0.1259	0.0009	5280.3918
500	7	1531506	10	0.1012	0.0750	81.5342
			50	0.1141	0.0136	424.8081
			100	0.1245	0.0044	833.1662
			200	0.1189	0.0012	1647.3677
			500	0.1249	0.0004	4089.9076
			1000	0.1358	0.0009	8173.1844
1000	1	57006	10	0.1093	0.0857	18.6521
			50	0.1178	0.0329	96.5533
			100	0.1322	0.0165	183.4581
			200	0.1425	0.0087	356.7093
			500	0.1479	0.0062	876.8253
			1000	0.1544	0.0042	1749.1374
			1500	0.1570	0.0038	2685.2984
			2000	0.1589	0.0034	3619.6982
1000	3	2059006	10	0.1003	0.0751	43.0980
			50	0.1130	0.0168	218.0598
			100	0.1092	0.0055	429.8577
			200	0.1048	0.0018	858.2739
			500	0.1091	0.0014	2082.9280
			1000	0.1112	0.0010	4095.7939
			1500	0.1160	0.0009	6085.9303
			2000	0.1197	0.0010	8065.7573
1000	5	4061006	10	0.1055	0.0740	79.0602
			50	0.1106	0.0086	394.2136

Continued on next page

Table B.1: All trained feedforward neural networks for different architectures that predict the hyperplanes for four partitions (K=4). *Width* is the number of nodes per layer. *Depth* is the number of hidden layers, excluding the input and output layer. *Pars.* is the number of parameters in the FNN. For FNNs with more than 5 hidden layers, the model was trained up to 1000 epochs due to the long runtime of models with many parameters.

Width	Depth	Pars.	Epochs	Test MSE	Train MSE	Train Duration (s)
			100	0.1119	0.0062	784.9651
			200	0.1161	0.0018	1608.2491
			500	0.1288	0.0015	4011.2452
			1000	0.1344	0.0009	7965.8147
1000	7	6063006	10	0.0988	0.0787	122.5035
			50	0.1135	0.0272	609.2876
			100	0.1138	0.0027	1215.1179
			200	0.1182	0.0059	2451.8224
			500	0.1327	0.0011	5924.9844
			1000	0.1432	0.0012	11580.5000

C | K-PP results

Table C.1: K-PP results based on all trained feedforward neural networks for different architectures for K=4. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the *Normalized Objective* of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
10	1	576	10	0.9107	0.9770	1.0232	0.0556	0
			50	0.9553	0.9891	1.0212	0.0564	0
			100	0.9570	0.9912	1.0251	0.0578	0
			200	0.9598	0.9926	1.0244	0.0567	0
			500	0.9635	0.9941	1.0273	0.0579	0
			1000	0.9629	0.9947	1.0276	0.0595	0
			1500	0.9604	0.9918	1.0228	0.0616	0
			2000	0.9611	0.9932	1.0244	0.0601	0
10	3	796	10	0.9346	0.9866	1.0284	0.0597	1
			50	0.9580	0.9949	1.0296	0.0601	0
			100	0.9645	0.9947	1.0279	0.0593	0
			200	0.9649	0.9952	1.0282	0.0596	0
			500	0.9625	0.9950	1.0268	0.0599	0
			1000	0.9606	0.9923	1.0246	0.0594	0
			1500	0.9626	0.9955	1.0272	0.0633	0
			2000	0.9619	0.9947	1.0265	0.0651	0
10	5	1016	10	0.9446	0.9891	1.0241	0.0681	0
			50	0.9595	0.9952	1.0254	0.0624	0
			100	0.9585	0.9912	1.0236	0.0651	0
			200	0.9660	0.9947	1.0280	0.0730	0
			500	0.9619	0.9936	1.0238	0.0719	0
			1000	0.9623	0.9942	1.0262	0.0647	0
10	7	1236	10	0.9541	0.9940	1.0318	0.0710	0
			50	0.9535	0.9856	1.0176	0.0636	0
			100	0.9701	1.0022	1.0329	0.0702	0
			200	0.9703	1.0017	1.0328	0.0719	0
			500	0.9624	0.9941	1.0257	0.0707	0
			1000	0.9629	0.9937	1.0255	0.0692	1
50	1	2856	10	0.9485	1.0008	1.0342	0.0775	0
			50	0.9471	0.9834	1.0182	0.0690	4
			100	0.9556	0.9898	1.0227	0.0728	1
			200	0.9680	1.0014	1.0324	0.0723	1
			500	0.9650	0.9992	1.0303	0.0694	2

Continued on next page

Table C.1: *K*-PP results based on all trained feedforward neural networks for different architectures for K=4. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the *Normalized Objective* of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
			1000	0.9676	0.9983	1.0290	0.0704	1
			1500	0.9587	0.9937	1.0263	0.0666	1
			2000	0.9697	1.0002	1.0317	0.0819	2
50	3	7956	10	0.9366	0.9857	1.0266	0.0822	0
			50	0.9651	0.9997	1.0316	0.0755	0
			100	0.9671	0.9970	1.0279	0.0673	0
			200	0.9595	0.9990	1.0290	0.0688	3
			500	0.9608	0.9961	1.0258	0.0697	4
			1000	0.9550	0.9933	1.0227	0.0702	10
			1500	0.9539	0.9909	1.0211	0.0725	15
			2000	0.9558	0.9970	1.0271	0.0747	15
50	5	13056	10	0.9495	0.9885	1.0259	0.0770	0
			50	0.9680	1.0016	1.0330	0.0833	0
			100	0.9702	1.0010	1.0316	0.0832	2
			200	0.9604	0.9932	1.0260	0.0853	1
			500	0.9525	0.9950	1.0274	0.0808	10
			1000	0.9488	0.9922	1.0266	0.0849	9
50	7	18156	10	0.9603	0.9940	1.0250	0.0881	0
			50	0.9621	0.9944	1.0250	0.1039	0
			100	0.9650	0.9985	1.0318	0.1019	0
			200	0.9619	0.9972	1.0274	0.1003	1
			500	0.9517	0.9885	1.0214	0.1082	1
			1000	0.9521	0.9920	1.0261	0.1089	2
100	1	5706	10	0.9352	0.9866	1.0253	0.1138	0
			50	0.9668	1.0023	1.0365	0.1192	0
			100	0.9594	0.9929	1.0274	0.1139	1
			200	0.9678	1.0045	1.0392	0.1144	3
			500	0.9610	0.9990	1.0308	0.1216	4
			1000	0.9567	0.9947	1.0256	0.1127	18
			1500	0.9594	0.9947	1.0266	0.1186	14
			2000	0.9503	0.9891	1.0223	0.1185	17
100	3	25906	10	0.9309	0.9735	1.0138	0.1214	0
			50	0.9618	0.9966	1.0286	0.1214	1
			100	0.9603	0.9958	1.0260	0.1218	5
			200	0.9615	0.9990	1.0294	0.1217	8
			500	0.9410	0.9883	1.0237	0.1215	15
			1000	0.9381	0.9899	1.0251	0.1269	18
			1500	0.9294	0.9856	1.0214	0.1207	16
			2000	0.9306	0.9851	1.0202	0.1266	17

Continued on next page

Table C.1: *K*-PP results based on all trained feedforward neural networks for different architectures for K=4. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the *Normalized Objective* of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
100	5	46106	10	0.9528	0.9865	1.0235	0.1503	0
			50	0.9645	0.9961	1.0288	0.1507	0
			100	0.9564	0.9924	1.0257	0.1608	0
			200	0.9587	0.9969	1.0294	0.1642	0
			500	0.9505	0.9895	1.0267	0.1657	1
			1000	0.9490	0.9902	1.0257	0.1754	1
100	7	66306	10	0.9640	0.9956	1.0284	0.1783	0
			50	0.9621	0.9951	1.0258	0.1837	0
			100	0.9558	0.9905	1.0242	0.1690	1
			200	0.9598	0.9951	1.0283	0.1791	0
			500	0.9591	0.9970	1.0300	0.1853	0
			1000	0.9609	0.9971	1.0304	0.1917	0
200	1	11406	10	0.9368	0.9898	1.0294	0.2089	1
			50	0.9558	0.9921	1.0252	0.2119	2
			100	0.9563	0.9938	1.0266	0.1956	5
			200	0.9611	0.9988	1.0301	0.1940	7
			500	0.9438	0.9905	1.0223	0.1920	30
			1000	0.9352	0.9942	1.0307	0.1944	25
			1500	0.9132	0.9826	1.0159	0.1801	55
			2000	0.9242	0.9919	1.0253	0.1898	40
200	3	91806	10	0.9574	0.9948	1.0315	0.2163	0
			50	0.9578	0.9946	1.0285	0.2031	1
			100	0.9590	0.9982	1.0324	0.2128	2
			200	0.9466	0.9884	1.0244	0.2144	4
			500	0.9480	0.9899	1.0246	0.2137	3
			1000	0.9491	0.9909	1.0283	0.2295	1
			1500	0.9492	0.9884	1.0256	0.2320	1
			2000	0.9525	0.9928	1.0270	0.2446	2
200	5	172206	10	0.9641	0.9980	1.0291	0.2375	0
			50	0.9638	0.9953	1.0272	0.2418	0
			100	0.9612	0.9973	1.0284	0.2416	1
			200	0.9622	0.9980	1.0305	0.2619	2
			500	0.9669	1.0012	1.0341	0.2622	0
			1000	0.9639	0.9981	1.0308	0.2702	0
200	7	252606	10	0.9608	0.9921	1.0223	0.2598	0
			50	0.9630	0.9946	1.0249	0.2605	0
			100	0.9615	0.9956	1.0275	0.2702	1
			200	0.9630	0.9981	1.0302	0.2677	1
			500	0.9642	1.0006	1.0330	0.2904	0

Continued on next page

Table C.1: K-PP results based on all trained feedforward neural networks for different architectures for K=4. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the *Normalized Objective* of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
			1000	0.9650	0.9992	1.0309	0.2739	0
500	1	28506	10	0.9460	0.9898	1.0272	0.2774	2
			50	0.9449	0.9879	1.0203	0.2851	2
			100	0.9597	1.0009	1.0327	0.2745	4
			200	0.9393	0.9924	1.0261	0.2629	21
			500	0.9247	0.9885	1.0240	0.2635	50
			1000	0.9168	0.9864	1.0218	0.0702	44
			1500	0.9061	0.9820	1.0210	0.0837	67
			2000	0.9083	0.9829	1.0214	0.0828	68
500	3	529506	10	0.9662	1.0014	1.0353	0.0566	0
			50	0.9582	0.9941	1.0258	0.0562	1
			100	0.9619	1.0006	1.0339	0.0590	1
			200	0.9659	1.0011	1.0338	0.0579	0
			500	0.9687	1.0019	1.0348	0.0596	0
			1000	0.9643	1.0010	1.0347	0.0603	0
			1500	0.9631	0.9985	1.0319	0.0601	0
			2000	0.9598	0.9990	1.0323	0.0606	0
500	5	1030506	10	0.9680	0.9966	1.0282	0.0608	0
			50	0.9636	0.9961	1.0273	0.0612	0
			100	0.9706	1.0031	1.0348	0.0632	0
			200	0.9677	1.0008	1.0346	0.0637	0
			500	0.9640	0.9980	1.0317	0.0646	0
			1000	0.9584	0.9906	1.0264	0.0642	0
500	7	1531506	10	0.9736	1.0017	1.0298	0.0673	0
			50	0.9673	0.9994	1.0287	0.0646	0
			100	0.9686	1.0022	1.0349	0.0690	0
			200	0.9671	1.0020	1.0333	0.0690	0
			500	0.9654	1.0011	1.0338	0.0692	0
			1000	0.9587	0.9951	1.0290	0.0674	1
1000	1	57006	10	0.9313	0.9696	1.0092	0.0648	3
			50	0.9550	1.0011	1.0330	0.0691	6
			100	0.9463	0.9901	1.0248	0.0718	14
			200	0.9371	0.9937	1.0249	0.0742	21
			500	0.9236	0.9838	1.0214	0.0804	33
			1000	0.9093	0.9825	1.0214	0.0873	47
			1500	0.9006	0.9786	1.0199	0.0882	48
			2000	0.9048	0.9793	1.0204	0.0883	53
1000	3	2059006	10	0.9600	0.9929	1.0245	0.0669	0
			50	0.9599	0.9940	1.0288	0.0651	0

Continued on next page

Table C.1: K-PP results based on all trained feedforward neural networks for different architectures for K=4. *Width* is the number of nodes per layer. *Depth* is the number of hidden layers. *Pars.* is the number of parameters in the FNN. *Normalized Objective* is the quantity given in (5.1). Here, *10%* is the 10% quantile, *Median* is the median, and *90%* is the 90% quantile of the *Normalized Objective* of the 1000 test instances. *Adjusted* is the number of instances that needed to be adjusted due to intersections of the predicted hyperplanes.

Width	Depth	Pars.	Epochs	Normalized Objective			Duration (s)	Adjusted
				10%	Median	90%		
			100	0.9667	0.9974	1.0323	0.0659	0
			200	0.9690	0.9995	1.0330	0.0662	0
			500	0.9675	1.0000	1.0318	0.0662	0
			1000	0.9644	0.9993	1.0307	0.0717	0
			1500	0.9661	0.9992	1.0319	0.0733	0
			2000	0.9607	0.9962	1.0311	0.0738	0
1000	5	4061006	10	0.9761	1.0055	1.0374	0.0777	0
			50	0.9705	1.0043	1.0358	0.0765	0
			100	0.9723	1.0029	1.0337	0.0781	0
			200	0.9698	1.0005	1.0351	0.0771	0
			500	0.9607	0.9978	1.0305	0.0774	1
			1000	0.9554	0.9910	1.0269	0.0781	0
1000	7	6063006	10	0.9752	1.0043	1.0330	0.0832	0
			50	0.9739	1.0023	1.0329	0.0833	0
			100	0.9730	1.0061	1.0349	0.0894	0
			200	0.9740	1.0058	1.0377	0.0943	0
			500	0.9691	1.0029	1.0343	0.0930	0
			1000	0.9598	0.9941	1.0296	0.0904	0