NTNU
Norwegian University of
Science and Technology

Master's thesis

2020

Even Masdal

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Even Masdal

A spatial branch-and-bound method for ReLU network-constrained problems

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

A spatial branch-and-bound method for ReLU network-constrained problems

Even Masdal

Cybernetics and Robotics

Submission date:	June 2020
Supervisor:	Lars Imsland
Co-supervisor:	Bjarne Grimstad

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This thesis presents results of the masters degree course *TTK4900 - Engineering Cybernetics, Master's Thesis* at the Norwegian University of Science and Technology, NTNU. The work was performed over the course of six months between January and June of 2020.

All the work presented in the thesis was performed by me, with two exceptions. The implementations for constructing and performing bound tightening on MILPs representations of ReLU networks are closely derived from the implementations of Bjarne Grimstad in [23]. The original implementations were discarded due to a combination of low performance and issues stemming from a lack expandability.

I would like to thank both of my supervisors, Professor Lars Imsland and Bjarne Grimstad, for their patience and for sticking with me through periods when I lacked both motivation and progress. Not only helping providing guidance and discussion related to the thesis but giving great advice for life in general. Their guidance and encouragement having had a significant positive impact on the work presented.

I would also like to thank my friends and family, who helped me keep my spirits up and helped me get some much needed fresh air during the corona virus pandemic and subsequent closing of campus at NTNU. They are the the reason working, sleeping and eating all in the same room was bearable for a large part the duration of the thesis work.

Even Masdal
June 2020

Abstract

Motivated by the data-driven black-box modeling abilities of artificial neural networks and the ability of spatial branch-and-bound (sBB) methods to solve nonlinear programs (NLP), this thesis presents a method for incorporating ReLU network-constraints into an sBB solver. Other authors have previously employed mixed-integer linear programming (MILP) formulations of ReLU networks to embed ReLU network-constraints into larger MILP problems. Considering these networks to be nonlinear functions of the input variables instead of MILPs has the benefit of reducing the number of variables on which the output of the networks depend. To examine if the sBB approach could have performance benefits over the MILP approach an sBB solver was implemented and a series of tests devised. The results compare the performance of the sBB solver to a state of the art MILP solver, and show that the MILP solver has a performance well ahead of the sBB solver with very few exceptions.

Sammendrag

Motivert av evnen kunstige nevrale nettverk har til data-dreven black-box modellering og bruken av spatial branch-and-bound (sBB) algoritmer til å løse ulineære optimaliseringsproblemer presenterer denne opgaven en metode for å inkludere ReLU nettverk-beskrankninger i en sBB løser. Andre forfattere har tidligere brukt mixed-integer linear programming (MILP) formuleringer av slike nettverk for å bygge inn ReLU nettverk-beskrankninger i større MILP problemer. Å anse disse nettverkene for å være ulineære funksjoner av inngangsvariablene og ikke MILP formuleringer har den fordelen at antallet variabler som påvirker utgangsverdien reduseres. For å teste om en sBB-tilnærming kan være fordelaktig sammenlignet med en MILP-tilnærming ble en sBB-løser implementert og en rekke tester blitt konstruert. Resultatene sammenligner ytelsen til sBB-løseren med en toppmoderne MILP-løser og viser at MILP-løseren har betraktelig bedre ytelse enn sBB-løseren med veldig få unntak.

Table of contents

List of figures

List of tables

Nomenclature

Greek Symbols

$\boldsymbol{\theta}$	Parameter of a neural network

Acronyms / Abbreviations

ANN	Artificial Neural Network
BT	Bound tightening
FBBT	Feasibility-Based Bound tightening
IP	Integer Programming
LP	Linear Programming
MILP	Mixed-Integer Linear Programming
MIP	Mixed-Integer Programming
MLP	Multi-layer perceptron
NLP	Non-Linear Programming
OBBT	Optimality-Based Bound tightening
ReLU	Rectified Linear Unit
sBB	spatial branch-and-bound

Chapter 1

Introduction

1.1 Background and Motivation

The field of mathematical optimization is, in essence, the study of methods for finding the best solution to a given problem. The desire to find the best, or *optimal*, solution to a problem is innate in almost everything we do, be it save on grocery costs, minimize commute times or finding the perfect air conditioning setting. Mathematical programming as a tool has shown its benefits in many areas such as applied mathematics, engineering, medicine and economics [51]. These problems can range from finding optimal base station locations [2] to minimizing power generation cost given the unpredictable nature of windmill power generation [31].

A crucial aspect in finding optimal solutions to problems like these is being able to properly describe the problem at hand, a way of modeling it using mathematical constructs. For complex processes that are hard to directly incorporate into the optimization process, *surrogate models* are often used. Surrogate models can be used as substitutes for computationally expensive simulations or for systems where modeling the full behavior is not a feasible task. Another use of surrogate models is for modeling systems defined only by their inputs and outputs, known as "black-box" systems. The black-box construct can be a useful tool for abstraction, but also to describe and categorize systems for which the internal dynamics of a system are simply unknown.

A black-box system however, can often be sampled, generating a data set of corresponding inputs and outputs. As a result having a way to generate a model of a black-box system from sampled data would be immensely useful for a variety of tasks, including optimization. From simple interpolation of the data to utilizing classical methods from the field of machine learning, many methods for creating models from data have been explored, and been utilized in optimization [7].

With the increase in computational resources and data availability in recent times, the study and applications of artificial neural networks has experienced a boom. Artificial neural networks have shown to be diverse in their applications, from predicting electricity demand [43] to object detection [11]. They are able to deal with highly nonlinear phenomena and show robustness when trained on noisy and incomplete data [32]. This ability to learn complex behaviors only from data opens up possibilities for constructing models of black-box systems that would previously have been too difficult to model.

Using artificial neural networks as surrogate models is therefore an interesting area of research. If optimization problems with ANN surrogate models can be solved efficiently, that could mean easier optimization of complex and black box processes, with a limiting factor being data availability rather than modeling ability.

1.1.1 Previous work

For a class of artificial neural networks, *ReLU networks*, mathematical programming formulations, in the form of MILPs, have been constructed [17]. This formulation has been successfully used to analyze the robustness of networks and finding “adversarial examples”, inputs for where a small perturbation results in a large change in network prediction [52]. This formulation was also utilized in [23] where it was used to embed multiple ReLU networks into larger optimization problems. [23] also explored the effects of different bound tightening procedures and their influence on solution times for problems with ReLU networks embedded.

The output of a ReLU network is nonlinear with respect to the input. As a result optimization problems containing ReLU networks can be considered to be a sub class of nonlinear optimization. Nonlinear programs can be solved using spatial branch-and-bound methods like demonstrated in [50], solving several problems with nonlinear terms by reformulating them in to representations for which convex relaxations could be constructed. Similarly the MILP formulation of ReLU networks presented in [17] can be used to generate a convex relaxation of the ReLU networks.

1.2 Problem Formulation and objective

Having a class of problems that can be considered both MILP and NLP raises some interesting questions, since different approaches are used to solve MILPs and NLPs. The most interesting one might be whether one formulation has an advantage over the other, or gains an advantage as in certain conditions. In a broad sense this thesis aims to examine how solving a problem as NLP, using spatial branch-and-bound, compares to solving the equivalent MILP formulation and how each method scales with respect to different network properties. Gaining an understanding of how both methods scale could in turn give an indication of the viability of solving ReLU network-constrained problems using spacial branch-and-bound.

A simple hypothesis is formulated asserting that when given a problem with ReLU network-constraints, there should some ratio of total neurons to input neurons in the ReLU network, that if exceeded, will lead to the spatial branch-and-bound approach outperforming the MILP approach.

1.3 Contributions

In order to test the spatial branch-and-bound approach a solver capable of handling ReLU network-constraints had to be implemented. This thesis presents the methods used to implement such a solver together with a selection of test problems aimed at testing the performance of the solver with respect to different properties of the ReLU network-constraints.

1.4 Structure of report

This thesis begins with Chapter 2 introducing some theoretical background on the topics of artificial neural networks, mixed-integer programming, branch-and-bound algorithms and bound tightening. Also shown is how these topics combine to form MILP representations and bound tightening procedures for ReLU networks. Chapter 3 starts off with presenting the hypothesis in more detail and defining a problem class for the sBB solver implementation. Following that are details on the implementation of a few selected steps of the branch-and-bound algorithm. Chapter 4 presents the test problems used to gauge the performance of the solver as well as the reasoning behind their inclusion. Chapter 5 presents both the results of the training of the neural networks and the results of optimization, as the properties of the trained neural networks vary between tests. Chapter 6 contains a discussion into the performance of the branch-and-bound solver, how bound tightening procedures and network properties affect the performance and how the sBB approach compares to the MILP approach. Chapter 7 concludes the report by examining the accuracy of the hypothesis and discussing some of the directions further research could take.

Chapter 2

Theory

2.1 Artificial neural networks

Artificial neural networks, or ANNs, are computational graphs designed to loosely resemble the structure of biological neural networks, like the ones found in brains [42]. ANNs are generally used in machine learning as a tool to learn patterns or structures in a data set or environment. The training of neural networks typically involves a feedback loop that makes small adjustments to the sturcture of the network over and over. Depending on the type of network and the type of training this feedback loop can be constructed in different ways to allow ANNs to excel in a multitude of areas. A typical example of this feedback loop can be found in supervised learning, where a network tries to learn the structure of a dataset by comparing the output of the network to a true output value associated with a given input. Other examples of feedback loops are reward based loops, typically used in reinforcement learning, where the feedback is given depending on the success of an action chosen by the network [38] or unsupervised learning, like in Generative Adversarial Networks, where two ANNs compete with each other in a loop [22].

2.1.1 Supervised learning

Supervised learning is the process of training a machine learning algorithm to learn the patterns or structure in some data, given a set of inputs and outputs. A set like this is called the training set and consists of pairs of input and output data points. The basic process of training an ANN involves iterating over the training set multiple times while making small adjustments to the parameters of the network in a way that reduces the difference between the ground truth and the network output. The ultimate goal being to minimize the difference between the ground truth and the network outputs across all data points in the training set. This enables the inference of an underlying function, $y = f(x)$, from which some training data, $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ originated [16]. Assuming the process was successful the ANN could then be used to classify or predict data not in the training set with high accuracy.

2.1.2 Multilayer perceptron

One form of artificial neural networks is the multilayer perceptron or MLP. An MLP is a fully connected feedforward artificial neural network. The structure of a multilayer perceptron is shown in Figure 2.1. An MLP consists of three types of layers, an input layer, a number of intermediate layers, often called hidden layers, and an output layer. The computation in the MLP follows the paths of the arrows that can be seen in Figure 2.1. For a fully connected network like an MLP the input of a neuron in one layer consists of a combination of all the outputs of all the neurons in the previous layer. A useful property of multilayer perceptrons is that they are universal approximators. An MLP with one hidden layer and a sufficient number of neurons can approximate any smooth function on a compact domain to an arbirtrary degree of accuracy [29], meaning approximating a function to any degree of accuracy is a matter of having enough neurons.

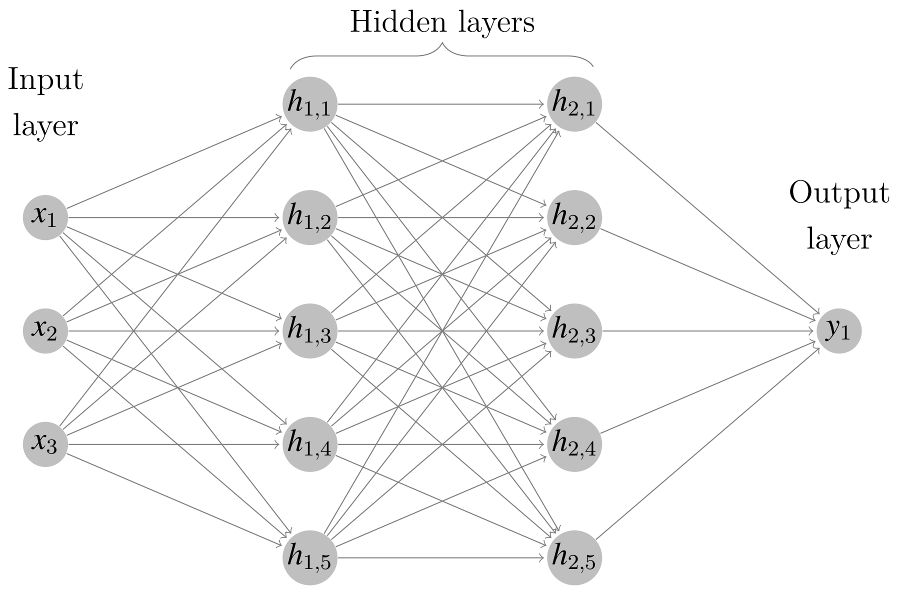

Fig. 2.1 The basic structure of a multilayer perceptron

While Figure 2.1 shows how the neurons in the network are structured in relation to each other, it does not show the inner workings of each neurons. Each edge in the network has an assigned weight which is multiplied with the value in the node the edge originated from. This results in the input to a neuron being the weighted sum of the values of the neurons in the previous layer. A bias parameter is then added to this sum and the resulting value is the input to what is known as the activation function. For a single neuron with activation function σ, the computation taking place can then be written as,

$$y = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right), \tag{2.1}$$

where $\mathbf{w}$ and $\mathbf{x}$ are vectors of edge weights and neuron values in the preceding layer. Typically the activation function in the output layer is linear so that the network can represent negative values.

While the universal approximator property of an MLP was originally shown for continuous, bounded and nonconstant activation functions [28], it has been shown that this property holds for any non-polynomial activation function [37]. It is important to note however that while this shows that an MLP can model arbitrary functions, it does not mean that training such an MLP is an easy task.

2.1.3 Objective functions and optimizers

The process of training ANNs is a form of optimization with the goal of minimizing or maximizing some objective. In ANN literature this objective is typically referred to as loss and for supervised learning it is a measure of the error between the output of the ANN and the true value for a given data point. The goal of the training is to minimize the average loss over all training samples. The output of an ANN can be written as a function,

$$\hat{y}_i = ANN(x_i; \boldsymbol{\theta}), \tag{2.2}$$

with inputs x_i and $\boldsymbol{\theta}$. Here x_i represents the input vector to the network and $\boldsymbol{\theta}$ represents the network parameters, the weights and biases of the network. Given a scalar function, ℓ, that assigns some error value for a given prediction, $\hat{y}_i$, and ground truth, y_i, the optimization problem that is training an ANN with n training samples can be written as,

$$\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, ANN(x_i; \boldsymbol{\theta})\right). \tag{2.3}$$

Typically ℓ will be a function like $\ell(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$, meaning the resulting loss will be the mean squared error.

The training process itself involves tweaking the parameters of the network, $\boldsymbol{\theta}$, in order to minimize the loss. This is done in an iterative process by using information about the gradient of the loss with respect to the network parameters. Through the process of backpropagation, this gradient information can be used to adjust the parameters of neurons in all layers [26]. Defining a function,

$$f_i\left(\boldsymbol{\theta}\right) = \ell\left(y_i, ANN(x_i; \boldsymbol{\theta})\right), \tag{2.4}$$

the gradient of the loss function with respect to the parameters, over all training samples, can be written as,

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta}). \tag{2.5}$$

The most fundamental approach to training the network is to use graident descent, repeatedly taking small steps in the direction of the gradient. Gradient descent can be formulated as,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \boldsymbol{\alpha} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_k), \tag{2.6}$$

where $\boldsymbol{\alpha}$ is a parameter called learning rate, which affects the magnitude of the steps taken. A variation on gradient descent is *Stochastic gradient descent*, which uses the assumption that the gradient of a randomly picked single sample, $\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})$, in general behaves like its expected value, the value of Equation 2.5, despite introducing noise [10]. An approach combining the ideas of stochastic gradient descent and gradient descent is mini-batch gradient descent, which uses the gradient of a randomly picked subset, or mini-batch, of the training data. For large data sets using the gradient of a subset of samples can improve training speed, as the number of samples evaluated for each iteration is small. Parameters like the learning rate and mini-batch size are commonly referred to as hyperparameters.

The method for which parameters are updated is typically referred to as an *optimizer*. As optimizers have matured new methods for achieving good convergence has emerged, including adaptive optimizers. These are optimizers that in essence modify the parameters of the optimizer during training. Adam, derived from Adaptive Moment Estimation, is an optimizer that computes individual adaptive learning rates for different parameters [34]. This can help ensure convergence for problems with large differences in scaling between input dimensions. Adam also incorporates a decaying average of past gradients, which acts like a sort of momentum, which can help the optimizer avoid local minima.

2.1.4 Generality, Regularization and Sparsity

For ANNs the concept of generality can be seen as the ability of an ANN to classify or predict unseen data with high accuracy [44]. Two ANNs that have the same performance on a given training set, are not guaranteed to predict new samples with same accuracy. The network that predicts new samples with the highest accuracy is said to have generalized better, as illustrated in Figure 2.2.

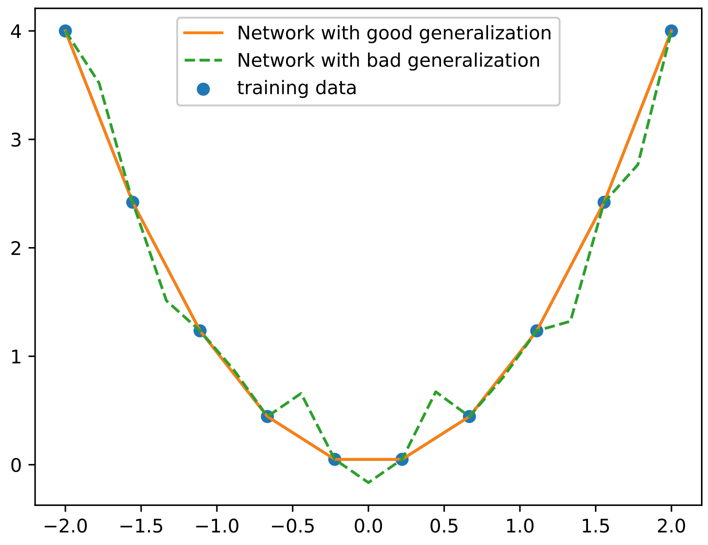

Fig. 2.2 An illustration of two ANNs that both fit the training data, but where the orange network generalizes better.

As ANNs train both the loss and the generality will improve initially, but as time goes on the generality could start to decrease, this is called *overfitting*. This is mostly a concern when the process from which the training data was sampled has some noise component. In that case the ANN could be trained to learn irrelevant patterns that are only a property of the chosen training data, like noise of the selected samples [16]. For ANNs that have more parameters than needed for a good representation of the data, this is a concern, as the extra parameters could allow the network to take on values that differ wildly from the underlying process between training samples. The idea of *sparsity* is the idea that for an ANN with too many parameters, some of the weights can be reduced to zero, essentially removing neurons from the network.

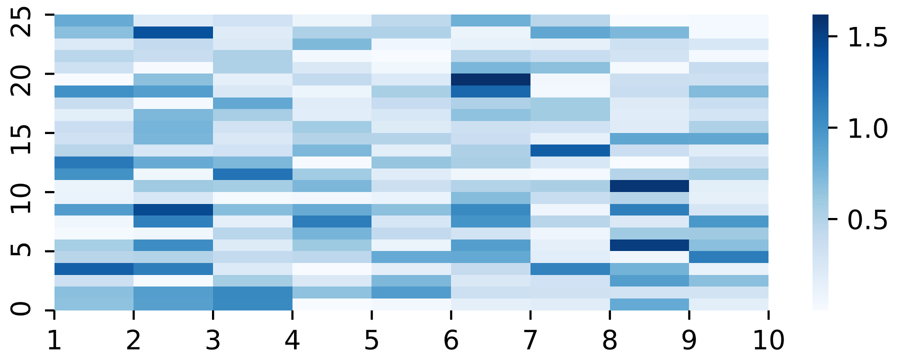

Fig. 2.3 Heatmap showing the absolute value of the weights of a network with a dense structure

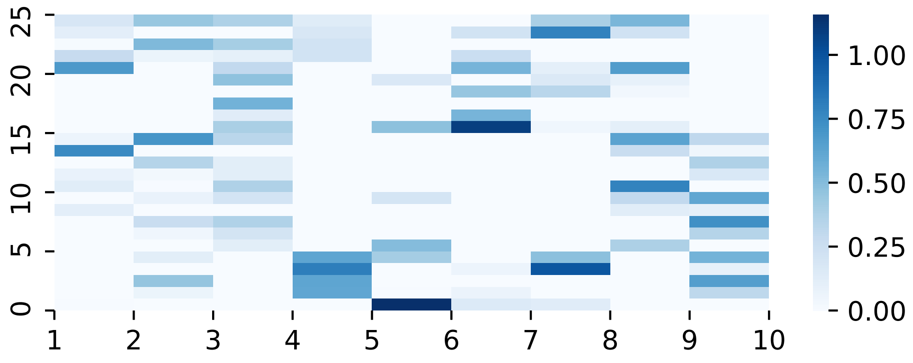

Fig. 2.4 Heatmap showing the absolute value of the weights of a network with a sparse structure

Figure 2.4 and Figure 2.3 show the absolute values of the weights for two neural networks trained on the same function. Both networks consist of 10 hidden layers, each with 5 neurons. Each column represents the weights between two hidden layers, with the weight matrix, W, being flattened into a vector. Figure 2.3 shows a dense network structure, while Figure 2.4 shows a comparatively sparse network structure. For the sparse network many of the weights are close to 0 in value.

Regularization is the name given to techniques that alter the training of neural networks with the goal of reducing overfitting and achieving better generality. One commonly used approach to regularization is norm regularization, usually in the form of L1 or L2 regularization. Norm regularization works by adding a term to the loss function that is dependent on the size of parameters of the network [40]. This term is then used to punish the optimizer for using large weights, hopefully leading to both a good fit and good generalization. The difference between L1 and L2 regularization is which norm is used on the parameters in the loss function. Changing the loss calculation in Equation 2.3 to include L2 regularization results in the following loss function,

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell \left(y_i, ANN(x_i; \boldsymbol{\theta}) \right) + \lambda_{L2} \sum \theta_j^2. \tag{2.7}$$

2.1.5 The Rectified Linear Unit

A commonly used activation function is the Rectified Linear Unit or ReLU. ReLU is a simple function, with the output being equal to the maximum of zero and the input, as shown in Equation 2.8. The derivative of the function is also exceedingly simple, simply being equal to 1 for positive inputs and 0 for negative inputs, shown in Equation 2.9. The derivative of ReLU is not defined at $x = 0$, but will typically either be implemented to be 0 or 1.

$$y = \sigma(x) = \max(0, x). \tag{2.8}$$

$$\sigma'(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x > 0 \end{cases} \tag{2.9}$$

A problem faced by many activation functions, is the problem of vanishing gradients [27]. The issue boils down to the activation functions having very small gradients for inputs with a large absolute value. As the networks use the gradient to update parameters this leads to slow learning. This problem is not faced by ReLU, as it has a constant gradient. Empirically ReLU has shown to work very well as an activation function, not only not facing the problem of vanishing gradients, but promoting sparsity [21].

Another property of the ReLU function is that while being a nonlinear function, ReLU consists of two linear segments, making it piecewise linear. As all the other computations in a neuron are linear, this leads to the output of a ReLU neuron being piecewise linear with respect to the input vector $\mathbf{x}$. As this holds for all the neurons in a ReLU network this means that the output of an MLP with only ReLU activations is piecewise linear with respect to the input [14].

2.1.6 Properties of deep networks

One of the benefits of neural networks is their ability to scale both to complex and high dimensional functions, it is a matter of adding more neurons to the network. The benefit of making neural networks deeper instead of wider is that they can require fewer neurons to represent a given function to the same degree of accuracy [13] as a shallow network. Another way to observe this benefit is to observe how the number of linear regions in the output increases with the number of neurons in a ReLU ANN. An upper bound on the number of linear regions for a ReLU network with an input dimension, m, and n hidden layers of width k, is $\mathcal{O}(k^{mn})$ [47]. The number of hidden neurons in that network would be $n \cdot k$. By keeping $n \cdot k$ constant and varying the amount of layers, n, the upper bound on the number of linear regions grows as n increases, meaning that, in a sense, the "resolution" of the network increases.

2.2 Mixed-Integer Programming

2.2.1 Integer Programming

Integer programming is a class of optimization where the problem variables are constrained to only take integer values. Unlike continuous variables, integer variables can model discrete states and as a result they can be used to model choices. An example of how integer variables can be used to model choices can be seen in the Knapsack problem, which can be formulated as,

$$\max \sum_{i=1}^{N} c_i \cdot x_i \tag{2.10a}$$

$$s.t. \quad a^T x \leq b \tag{2.10b}$$

$$x_i \in [0,1]. \tag{2.10c}$$

Mixed-integer programming, MIP, combines integer variables with continuous variables. This allows MIPs to model conditional relationships, like a set of constraints where only one constraint can be active at once. Mixed-integer linear programs, in turn, are mixed-integer programs where the objective and constraints are linear.

One challenge with mixed-integer programming is that it is an NP-Hard problem class [8]. An intuitive way to think of how the complexity of a mixed-integer program can grow, is that for each value every integer variable can take, a separate optimization problem can be constructed, where that variable is fixed. For 4 binary variables, that means a total of $2^4 = 16$ sub problems can be constructed. Depending on the problem many such combinations can be quickly ruled out, but there is no guarantee that there are simple ways to reduce the search space of a problem.

2.2.2 Mixed-integer representation of ReLU

The rectified linear unit is, as stated in Section 2.1.5, piecewise linear. As MILPs can model piecewise linear functions, there should be a way to represent the behavior of ReLU using mixed-integer programming. In fact a full representation of a network with ReLU activation functions can be constructed, as shown in [17]. Starting with the defenition of the ReLU function,

$$\sigma(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0, \end{cases} \tag{2.11}$$

it becomes apparent that $x = 0$ is the main point of interest. When x is below 0, the output is 0, and when x is above 0, the output is the value of x. Following [17], a step by step process of replicating this behaviour using mixed-integer linear programming begins with defining two new positive variables x^+ and x^-, where the difference $x^+ - x^-$ is defined to be equal to x. Representing this in equation form gives,

$$x^+ - x^- = x \tag{2.12a}$$

$$x^+, x^- \geq 0. \tag{2.12b}$$

As there are an infinite amount of x^+ and x^- pairs with a difference of x, additional constraints are needed to ensure x is uniquely represented by a single pair of x^+ and x^-. Adding constraints that only allow either x^+ or x^- to be positive at any given time ensures the existence of a unique pair, (x^+, x^-), for every x. This is done by introducing a binary variable, $z \in \{0, 1\}$, and two new constraints. By defining the constraints,

$$x^+ \leq U \cdot z, \tag{2.13a}$$

$$x^- \leq -L \cdot (1 - z), \tag{2.13b}$$

only either x^+ or x^- can be positive, depending on the value of z. U and L are upper bounds on x^+ and x^- respectively and represent the upper and lower bound on the input value, $x \in (L, U)$. Equation 2.13a and 2.13b are what is known as *big-M constraints*. The full formulation of the ReLU function then becomes,

$$x^+ - x^- = x \tag{2.14a}$$

$$x^+, x^- \geq 0 \tag{2.14b}$$

$$x^+ \leq U \cdot z \tag{2.14c}$$

$$x^- \leq -L \cdot (1 - z) \tag{2.14d}$$

$$z \in \{0, 1\}. \tag{2.14e}$$

In this formulation the value of x^+ is equal to the output of the ReLU function when the input is equal to $x^+ - x^-$.

2.2.3 ReLU network as MILP

Expanding on this representation to model a ReLU neuron as a component in a larger neural network formulation is a matter of changing the input value, previously referred to as x. The input to a neuron in a feed forward neural network is,

$$\mathbf{w}^{\intercal}\mathbf{x}_{prev} + b. \tag{2.15}$$

Using the variable names presented in [17] and replacing the input with Equation 2.15 gives the following formulation for a single ReLU neuron,

$$\begin{aligned} x - s &= \mathbf{w}^{\intercal}\mathbf{x}_{prev} + b && (2.16a)\\ x, s &\geq 0 && (2.16b)\\ x &\leq U \cdot z && (2.16c)\\ s &\leq -L \cdot (1 - z) && (2.16d)\\ z &\in \{0, 1\}\,. && (2.16e) \end{aligned}$$

Modeling a whole network comes down to connecting the neurons in one layer to the previous layer. For a network with K-1 hidden layers of n_k neurons, this leads to the following formulation,

Input layer, $k = 0$,

$$L^0 \leq x^0 \leq U^0. \tag{2.17}$$

Hidden layers, $k = 1, \ldots, K - 1$,

$$\begin{aligned} x^k - s^k &= W^k x^{k-1} + b^k && (2.18a)\\ x^k, s^k &\geq 0 && (2.18b)\\ x_j^k &\in \{0, 1\} && (2.18c)\\ x_j^k &\leq \quad U_j^k \cdot z_j^k && (2.18d)\\ s_j^k &\leq -L_j^k \cdot \left(1 - z_j^k\right) && (2.18e)\\ j &\in 1, ..., n_k. && (2.18f) \end{aligned}$$

Output layer, $k = K$,

$$w^K x^{K-1} + b^K = x^K \tag{2.19a}$$

$$L^K \leq x^K \leq U^K. \tag{2.19b}$$

2.2.4 The relaxation of an optimization problem

In mathematical programming an optimization problem can have what is called a relaxation. A relaxation is, in a sense, a version of the original problem that is easier to solve and is related to the original problem closely enough to where the solution to the relaxation can give useful information about the original problem. For any given problem many relaxations can exist and as such a definition of what makes a problem a relaxation is useful. Using the term relaxation as used in [18], gives the following definition of a relaxed problem. A relaxation of the problem,

$$z = \mathbf{min}\{f(x) : x \in \mathcal{X} \subseteq \mathbb{R}^n\}, \tag{2.20}$$

is another problem,

$$z^r = \mathbf{min}\{c(x) : x \in \mathcal{T} \subseteq \mathbb{R}^n\} \tag{2.21}$$

as long as the following conditions hold,

$$i) \quad \mathcal{X} \subseteq \mathcal{T} \tag{2.22a}$$

$$ii) \quad c(x) \leq f(x) \text{ for all } x \in \mathcal{X}. \tag{2.22b}$$

This definition of a relaxation is quite broad and does not guarantee a useful relaxation. As such the challenge of finding a relaxation is not as much finding a problem that fits this definition, but finding a problem that is both easier to solve and closely related to the original problem.

For a mixed-integer linear program a relaxation can be created by relaxing the integrality constraints. Changing all integer variables to continuous variables in a MILP means that any constraints containing integer variables become linear constraints. If the new continuous variables keep the same bounds as the integer variables they replaced, the original state space will be a subset of the relaxed state space. In addition, as the initial problem contained only linear and integer constraints the relaxed problem will, as a result, contain only linear constraints, meaning the relaxation of a MILP is a linear program.

For a minimization problem, a relaxation introduces a way to obtain a lower bound on the optimal value of the problem, as the solution to the relaxation will always be equal to, or lower, than the optimal value. For a linear program, a solution can be obtained in polynomial time [33]. Mixed-integer programs on the other hand are NP-hard. An LP relaxation of a MILP therefore allows a bound on the optimal value to be obtained in polynomial time.

2.3 Branch-and-bound methods

Branch-and-bound methods are methods for global optimization that take a divide and conquer approach to obtaining a solution. First presented in the 1960s [36], the idea behind branch-and-bound strategies is to recursively partition the feasible domain of an optimization problem into smaller regions, then calculating upper and lower bounds on the objective value in each partitioned region. The information these bounds give can then be used to discard partitions of the problem, if it can be shown that the partition can not contain the optimal solution.

2.3.1 The branch-and-bound tree

In order to illustrate how branch-and-bound strategies use a combination of partitioning and upper and lower bounds to find a global optima the branch-and-bound tree, as illustrated in Figure 2.5, is a useful construct.

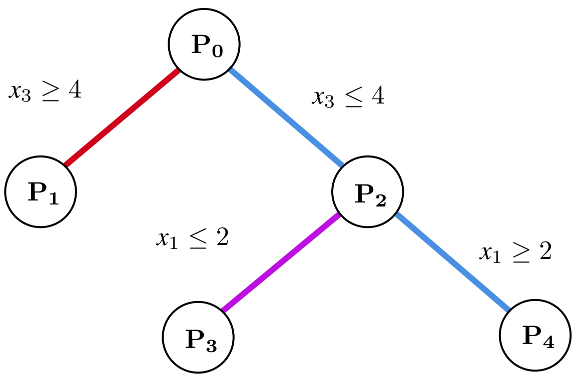

Fig. 2.5 Branch-and-bound tree

Beginning with a minimization problem, $\mathbf{P}_0$, an upper and lower bound on the optimal value of $\mathbf{P}_0$ is calculated, creating the root node of the branch-and-bound tree. Then $\mathbf{P}_0$ is divided into two separate sub problems, $\mathbf{P}_1$ and $\mathbf{P}_2$, by constraining the feasible set to where the *branching variable*, in this case x_3, is restricted to be greater than and less than the *branching point*, 4, for $\mathbf{P}_1$ and $\mathbf{P}_2$ respectively. In the branch-and-bound tree the sub problems, $\mathbf{P}_1$ and $\mathbf{P}_2$, become the child nodes of $\mathbf{P}_0$.

Reducing problem bounds

For each of the sub problems an upper and a lower bound on the objective value can be calculated. The upper bound on the objective is simply any feasible solution found. In order to be useful the method used to calculate the lower bound has to satisfy two properties. First the lower bound of a node has to be nondecreasing with respect to its parents. Secondly the lower bound of a node should always be lower than the optimal solution of the node [35]. If the new bounds satisfy these properties the bounds on the root node, $\mathbf{P}_0$, can be updated. The lower bound on the original problem, $\mathbf{P}_0$, will now be the minimum of the lower bounds of its child nodes. Similarly the best upper bounds of the sub problems will now be the the new upper bound of $\mathbf{P}_0$. As new nodes and sub problems are created, bounds will propagate up the tree, converging on the optimal solution. The difference between the upper bound and the lower bound in the branch-and-bound tree is referred to as the *optimality gap*. The absolute optimality gap is simply the difference between the best known solution and the best lower bound, while the relative optimality gap is the absolute optimality gap divided by the best lower bound.

Cutting branches

In addition to shrinking the bounds of $\mathbf{P}_0$, the bounds of a child node can be used to discard nodes in the branch-and-bound tree that can be guaranteed not to contain the optimal solution. Any node with a lower bound greater than the current best upper bound problem can be discarded, as there is no point in exploring a node where even the best solution is greater than the current upper bound. This also holds for any nodes that can be shown to not have any feasible solution. When a node is discarded from the branch-and-bound tree it is said to be *fathomed*.

2.3.2 Branch-and-bound for mixed-integer linear programs

In order to create sub problems, new constraints are added to the original problem in a way that reduces the feasible region. For mixed-integer linear programs a common approach is first solving a linear relaxation of the original problem, then observing which variables in the relaxed solution do not conform to the integrality constraints. This accomplishes two things, generating a lower bound on the optimal solution and providing candidates for branching variables [9]. By selecting an integer variable, x_i, that takes on a non-integer value, b_i, in the relaxed solution, two sub problems can be created by adding the constraints $x_i \geq \lceil b_i \rceil$ and $x_i \leq \lfloor b_i \rfloor$, where $\lceil b_i \rceil$ and $\lfloor b_i \rfloor$ are the floor and ceil functions respectively. If any of the relaxed solutions happen to be a feasible solution to the original problem it will represent an upper bound on the optimal solution of the original problem.

2.3.3 Spatial branch-and-bound

Adapting the branch-and-bound method to allow for branching on non-integer variables allows the method to be used to solve nonlinear [15] and mixed-integer nonlinear programs. This results in what is commonly called a spatial branch-and-bound algorithm. Algorithm 1 shows a spatial Branch-and-bound algorithm based on ones described in [5] and [24]. It shows how the concepts described in Section 2.3.1 are used in order to obtain an optimal solution z^*.

2.4 Bound Tightening

One part of Algorithm 1 not yet introduced is step 6, bound tightening. When a sub problem is created, new and tighter variable bounds are imposed on the branching variable. These new bounds could, through shared constraints, imply new and tighter bounds on variables that did not have bounds explicitly tightened through branching. Bound tightening can improve the lower bound in branch-and-bound sub problems dramatically [3]. The process of bound tightening could prove a sub problem infeasible, in the case that the upper bound on a variable is shown to be lower than the lower bound. Illustrating how shared constraints can be used to propagate bounds can be done by considering two variables, x_1 and x_2, and a simple linear constraint.

Algorithm 1: Spatial Branch-and-bound algorithm

```
Result: The optimal value $z^*$ of an optimization problem $\mathbf{P}$
Define a set $L$: $L \leftarrow \{\mathbf{P}\}$
Define an upper bound on $\mathbf{P}$: $z^u \leftarrow \infty$
while $L \neq \emptyset$ do
    select a subproblem $\mathbf{P}_k \in L$
    remove $\mathbf{P}_k$ from $L$: $L \leftarrow L \setminus \{\mathbf{P}_k\}$
    tighten bounds of $\mathbf{P}_k$
    if bound tightening proved $\mathbf{P}_k$ infeasible then
        continue
    end
    Generate a convex relaxation $\mathbf{R}_k$ of $\mathbf{P}_k$
    Solve $\mathbf{R}_k$; let $\bar{x}^k$ be an optimum and $\bar{z}^k$ the corresponding objective value
    if $\bar{x}^k$ is a feasible solution to $\mathbf{R}_k$ then
        Let $z^u \leftarrow \min\{z^u, \bar{z}^u\}$
    end
    if $\bar{z}^k \geq z^u$ or $z^u - \bar{z}^k \leq \varepsilon$ or $\mathbf{R}_k$ is infeasible then
        continue
    else
        Obtain solution candidate $\hat{z}^k$ from $\mathbf{P}_k$
        Let $z^u \leftarrow \min\{z^u, \hat{z}^u\}$
        Choose a branching variable $x_i$ and branching point $x_i^b$
        Create subproblems:
            $\mathbf{P}_{k-} \leftarrow \mathbf{P}_k$ where $(x_i \leq x_i^b)$
            $\mathbf{P}_{k+} \leftarrow \mathbf{P}_k$ where $(x_i \geq x_i^b)$
        $L \leftarrow L \cup \{\mathbf{P}_{k-}, \mathbf{P}_{k+}\}$
    end
end
Result $z^* = z^u$
```

$$2x_1 \leq x_2 \tag{2.23a}$$
$$x_1 \in [3,5] \tag{2.23b}$$
$$x_2 \in [2,8] \tag{2.23c}$$

From Equation 2.23a and Equation 2.23c, it can be concluded that x_1 can, at most, be equal to 4 by inserting the upper bound of x_2 into the Equation 2.23a, $2x_1 \leq x_2 \leq x_2^{ub} = 8 \rightarrow x_1 \leq 4$. Similarly from Equation 2.23a and Equation 2.23b, it can be concluded that the value of x_2 can be no less than 6. The results in new bounds, $x_1 \in [3,4]$ and $x_2 \in [6,8]$.

2.4.1 Feasibility-based bound tightening

Feasibility-based bound tightening, FBBT, are bound tightening algorithms that use primal feasibility arguments to remove parts of the domain in which no feasible solutions are contained [19].

Consider a linear program with a set of constraints $Ax \leq b$. These constraints can be split into rows on the form, $a_i^T x \leq b_i$, where a_i is the ith row of A. This can then be written as a sum,

$$\sum_{j=0}^{n} a_{ij}x_j \leq b_i. \tag{2.24}$$

Separating out $a_{i0}x_0$ and rearranging the inequality gives

$$a_{i0}x_0 + \sum_{j=1}^{n} a_{ij}x_j \leq b_i. \tag{2.25a}$$
$$a_{i0}x_0 \leq b_i - \sum_{j=1}^{n} a_{ij}x_j \tag{2.25b}$$

An upper bound on $a_{i0}x_0$ can then be obtained minimizing the value of $a_{ij}x_j$ for each variable in the sum.

$$a_{i0}x_0 \leq b_i - \sum_{j=1}^{n} a_{ij}x_j \tag{2.26a}$$

$$\leq b_i - \sum_{j=1}^{n} \min(a_{ij}x_j^{LB}, a_{ij}x_j^{UB}) \tag{2.26b}$$

Depending on the sign of a_{ij}, feasibility-based bound tightening will generate one of the following bounds on x_j

$$x_0 \leq \frac{1}{a_{i0}} \left(b_i - \sum_{j=1}^{n} \min\left(a_{ij}x_j^{LB}, a_{ij}x_j^{UB}\right)\right), a_{ij} > 0 \tag{2.27a}$$

$$x_0 \geq \frac{1}{a_{i0}} \left(b_i - \sum_{j=1}^{n} \max\left(a_{ij}x_j^{LB}, a_{ij}x_j^{UB}\right)\right), a_{ij} < 0 \tag{2.27b}$$

Similarly the same argument can be used to infer bounds on $x_1, \ldots, x_n$. This method of computing bounds does not guarantee any or optimal bound tightening [39]. For problems with multiple constraints FBBT can improve bounds iteratively by running the bound tightening multiple times over all the constraints. This process could be repeated until the bounds stop improving, and the method reaches a fixed point, but it can not be guaranteed that the bounds will converge to such a point in finite time [4].

2.4.2 Optimality-based bound tightening

Optimality-based bound tightening, OBBT, generates bounds not using feasibility based arguments, but through, as the name implies, optimization. For a convex relaxation of a MINLP, OBBT computes the tightest bounds valid for all relaxation solutions by in turn minimizing and maximizing each variable [19]. For a problem with n variables this involves solving $2n$ optimization problems in order to generate bounds. While this can be much more computationally expensive than FBBT, OBBT can produce better bounds as all constraints are considered when generating bounds. Demonstrating this can be done with a simple set of two constraints,

$$x_1 + x_2 \leq 1 \tag{2.28a}$$
$$x_1 + x_2 \geq y \tag{2.28b}$$
$$x_1, x_2 \in [0, 1] \tag{2.28c}$$
$$y \in [0, 2]. \tag{2.28d}$$

For this problem using the FBBT approach shown in Section 2.4.1 to tighten the bounds of variables of one constraint at a time will not lead to a reduction in variable ranges. Using Equation 2.28a an upper bound on x_1 is $x_1 \leq 1 - x_2 \leq 1 - x_2^{LB} = 1$, and using Equation 2.28b a lower bound on, x_1 is, $x_1 \geq y - x_2 \geq y^{LB} - x_2^{UB} = -1$. As the problem is symmetric the same calculations hold true for x_2. Neither the upper or lower bounds calculated by FBBT improve upon the initial bounds and as such an upper bound on y, $y \leq x_1 + x_2 \leq x_1^{UB} + x_2^{UB} = 2$ yields no improvement.

If both constraints are considered simultaneously however, as is the case when using OBBT, it becomes immediately clear that $y \leq x_1 + x_2 \leq 1$, leading to tighter bounds on y.

2.4.3 Bounds for MILP representations of ReLU networks

The MILP representation of ReLU networks presented in Section 2.2.3 requires a set of bounds for each neuron in the network in order for the big-M constraints in Equation 2.13a and 2.13b, to be valid. In [23] different methods for producing bounds for MILP representations of ReLU networks were explored. Among those a feasibility based approach used to generate initial neuron bounds.

Feasibility-based bound generation

The input to each neuron in an MLP is a linear combination of the values of the neurons in the previous layer. For neuron j in layer k, this can be written as,

$$\sum_{i=1}^{n_{k-1}} w_{ji}^{k} x_i^{k-1} + b_j^k, \tag{2.29}$$

where w_{ji}^{k} is the weight in the i-th entry of the j-th row of the weight matrix between layer k-1 and k. U_j^k and L_j^k will be used to represent the upper and lower bounds of this sum. The value of the expression in Equation 2.29 can be maximized or minimized by maximizing and minimizing the product $w_{ji}^{k} x_i^{k-1}$ for each i. For the first hidden layer the bounds on x_i^{k-1} are $x_i^{k-1} \in (L_i^{k-1}, U_i^{k-1})$, as layer $k-1$ will be the input layer. This leads to the following expressions for the bounds of the first hidden layer,

$$\begin{aligned} U_j^k &= \sum_{i=1}^{n_{k-1}} \max\left\{w_{ji}^k U_j^{k-1}, w_{ji}^k L_j^{k-1}\right\} + b_j^k, \\ L_j^k &= \sum_{i=1}^{n_{k-1}} \min\left\{w_{ji}^k U_j^{k-1}, w_{ji}^k L_j^{k-1}\right\} + b_j^k. \end{aligned} \tag{2.30}$$

For subsequent layers however, due to the ReLU activation, the bounds on x_i^{k-1} are $\left(\max\left\{L_i^{k-1}, 0\right\}, \max\left\{U_i^{k-1}, 0\right\}\right)$. Leading to the expressions,

$$\begin{aligned} U_j^k &= \sum_{i=1}^{n_{k-1}} \max\left\{w_{ji}^k \max\left\{U_j^{k-1}, 0\right\}, w_{ji}^k \max\left\{L_j^{k-1}, 0\right\}\right\} + b_j^k, \\ L_j^k &= \sum_{i=1}^{n_{k-1}} \min\left\{w_{ji}^k \max\left\{U_j^{k-1}, 0\right\}, w_{ji}^k \max\left\{L_j^{k-1}, 0\right\}\right\} + b_j^k, \end{aligned} \tag{2.31}$$

for the upper and lower bounds.

Chapter 3

Method

3.1 Hypothesis

The complexity of a MILP representation of a ReLU network is exponential with respect to the number of neurons in the ReLU network being modeled. On the other hand the complexity of a spatial branch-and-bound method branching on the input variables of a ReLU network is exponential with respect to the number of inputs. As the complexity of the MILP representation increases with the number of hidden neurons in a network that should mean that for some ratio of hidden neurons to input neurons the spatial branch-and-bound strategy should outperform the MILP strategy. Assuming this scaling, given a problem with n inputs and m total neurons there should be a constant, K, where if $\frac{m}{n} > K$, the branch and bound approach will be faster than the MILP method.

3.2 Problem class

In order to test the performance of running spatial branch-and-bound on optimization problems containing ReLU networks as constraints, a basic branch-and-bound solver was implemented. The goal was a solver able to solve problems containing a mix of linear constraints and ReLU network constraints. Formulating a problem class from this goal results in the formulation,

$$z = \arg\min_{x} \sum_{i=0}^{M} x_i \cdot c_i, \tag{3.1a}$$

$$Ax \leq b, \tag{3.1b}$$

$$f_j(x;\theta_j) \leq x_j,\ \ j \in \mathcal{C}_v, \tag{3.1c}$$

$$f_k(x;\theta_k) \leq d_k,\ \ k \in \mathcal{C}_c, \tag{3.1d}$$

Set	Description
$\mathcal{C}_v$	Set of ReLU network constraints constrained by optimization variables.
$\mathcal{C}_c$	Set of ReLU network constraints constrained by constants.

where M is the number of variables, c_i is the cost of a variable x_i and $f_n(x, \theta_n)$ are ReLU networks with input x, that are parameterized by θ_n. One note on this notation is that while each network, $f(x, \theta)$ is written as if the input to the network is the full vector, x, only select entries of x have to be used.

Using this formulation the problem class describes a nonlinear program. Being a NLP it can be solved using the method described in Algorithm 1, assuming all the steps in the algorithm can be implemented. While many of the steps are fairly trivial, there are some tasks that, while being only a single step in the algorithm make up a large portion of the work performed by the spatial branch-and-bound algorithm. The tasks that in this case will be explained more in depth are, in the order they will be examined,

- Step 18: Obtaining a solution candidate,
- Step 10: Generating a convex relaxation,
- Step 6: Bound tightening.

3.3 Obtaining a solution candidate, upper bound

As the problem class described by Equation 3.1 is a minimization problem, any feasible solution to the problem will be a valid upper bound on the optimal value. For the upper bounding problem the NLP solver of choice was Ipopt, [53]. Ipopt is designed to solve large scale nonlinear programs to local optimality. In addition to solving for local optimality Ipopt will also report if a feasible solution to the problem was reached, even if the solution was not optimal. This allows upper bounds to be generated using Ipopt even when given a limited number of iterations to find a solution.

Ipopt takes problems on the form,

$$\min_{x\in\mathbb{R}^n} f(x) \tag{3.2a}$$
$$\text{s.t.} \quad g_L \leq g(x) \leq g_U \tag{3.2b}$$
$$x_L \leq x \leq x_U, \tag{3.2c}$$

where $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ are the constraint functions, for a problem with n variables and m constraints.

In order to adapt the formulation in Equation 3.1 to be on the form Ipopt uses some adaptations need to be made. By splitting $g(x)$ into separate functions, $g_{L_i} \leq g_i(x) \leq g_{U_i}$ for $i = 1, \ldots, m$ the two types of neural network constraints can be written as,

$$g_j(x) = f_j(x; \theta_j) - x_j, \quad j \in \mathcal{C}_v \tag{3.3a}$$
$$g_k(x) = f_k(x; \theta_j), \quad k \in \mathcal{C}_c. \tag{3.3b}$$

with an upper bound of 0 and a lower bound of negative infinity. The linear constraints are divided into one constraint per row, a_i, of the constraint matrix, A.

$$g_i(x) = a_i \cdot x \leq b_i \tag{3.4}$$

where b_i is the i-th element of b.

In order to work Ipopt requires the ability to evaluate the objective, f, and its gradient ∇f, as well as the constraint function, g, and the corresponding Jacobian, $\mathbf{J}_g$.

The objective function and its gradient are straightforward to compute, with the objective being the dot product of a the x vector and a cost vector, $\mathbf{c}$.

$$f(x) = \sum_{i=0}^{M} x_i \cdot c_i = \mathbf{c}^T \cdot \mathbf{x} \tag{3.5}$$

$$\nabla f(x) = \mathbf{c} \tag{3.6}$$

Evaluating $g(x)$ should return a vector of length m, with each element being on the form of either Equation 3.3 or Equation 3.4,

$$\begin{bmatrix} g_0(x) \\ g_1(x) \\ \vdots \\ g_m(x) \end{bmatrix}. \tag{3.7}$$

Evaluating the jacobian of $g(x)$ will require computing the gradient of the ReLU networks with respect to the inputs. As gradient computation is a large part of training ANNs, methods for computing the gradient of a neural network should be included out of the box for most neural network frameworks.

$$\begin{bmatrix} \nabla g_0(x) \\ \nabla g_1(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix}. \tag{3.8}$$

As not all constraints contain all variables it is important to ensure the ordering of the variables in the Jacobian is correct. Considering a problem with the following constraints

$$x_0 - x_1 \leq b \tag{3.9a}$$
$$f_0(x_1, x_2) \leq x_0 \tag{3.9b}$$
$$f_1(x_0, x_2) \leq d \tag{3.9c}$$

Then $g(x)$ becomes,

$$g(x) = \begin{bmatrix} x_0 - x_1 \\ f_0(x_1, x_2) - x_0 \\ f_1(x_0, x_2) \end{bmatrix} \tag{3.10}$$

And the Jacobian,

$$\mathbf{J}_g = \begin{bmatrix} 1 & -1 & 0 \\ -1 & \partial_{x_1} f_0(x_1, x_2) & \partial_{x_2} f_0(x_1, x_2) \\ \partial_{x_0} f_1(x_1, x_2) & 0 & \partial_{x_2} f_1(x_1, x_2) \end{bmatrix} \tag{3.11}$$

3.4 Generating a convex relaxation, Lower bound

In order to obtain a lower bound Algorithm 1 includes a step for generating a convex relaxation. Starting with the MILP formulation a relaxation can be generated by relaxing binary variable in Equation 2.14 to a continuous variable between 0 and 1. This leads to the LP relaxation,

$$x - s = x \tag{3.12a}$$

$$x, s \geq 0 \tag{3.12b}$$

$$x \leq U \cdot z \tag{3.12c}$$

$$s \leq -L \cdot (1 - z) \tag{3.12d}$$

$$z \in [0, 1], \tag{3.12e}$$

for a single ReLU neuron. As the problem class only has linear constraints and ReLU network constraints, relaxing all the neurons in all ReLU network constraints leads to a problem with only linear constraints, meaning the whole problem class can be relaxed to an LP. Being a linear program this also means the relaxation is convex, as any feasible set defined only by linear equalities and inequalities is a convex polytope [41].

The solver chosen for solving the relaxed problem is Gurobi, [25], a solver designed to solve linear, quadratic, mixed-integer linear and mixed-integer quadratic programs.

3.5 Bound Tightening

In the problem class presented there is, in a way, two different variable types, and it is useful to create a distinction. The first type of variable are the variables that appear both in the upper and lower bounding problems. These are the problem variables that are referred to in the problem class formulation in Equation 3.1 as x. In addition there are the variables used to represent neurons in the ReLU networks that only appear in the lower bounding problem. The latter will in this section be referred to as *internal* variables, as they are an internal part of the ReLU constraints.

The way the solver has been implemented bound tightening is in essence performed in two different steps of the spatial branch-and-bound algorithm. Bound tightening is performed on the inner variables when the linear relaxations of the ReLU constraints are constructed, and on the problem variables during the bound tightening step.

3.5.1 Linear constraints

Bound tightening on the linear constraints in the problem is only performed using FBBT methods. Using the method demonstrated in Section 2.4.1 the solver iterates over the linear constraints a set number of times when the bound tightening step is run to ensure bounds can get propagated through multiple constraints. Bound tightening on the linear constraints happens twice in each bound tightening step, once before and once after bound tightening is performed on the ReLU constraints, in order to propagate input bounds and output bounds respectively.

3.5.2 ReLU network constraints

During the bound tightening step the main concern of the bound tightening of the ReLU networks is to propagate bounds set on the problem variables of the input through the network and apply them to the problem variable of the output, thus allowing further propagation of bounds. This means that bound tightening will have to be performed on all the inner variables of each network for each node in the branch-and-bound tree. In [23] the bound tightening procedures were run once at the root node of the branch-and-bound tree, leaving further bound tightening up to the solver, due to the computational cost of running some of the procedures. As the bound tightening procedures will be performed at every node, only the two least computationally expensive methods will be implemented, both using the LP relaxation of the ReLU constraints.

FBBT

The first approach to both generate bounds and perform bound tightening is the FBBT method presented in Section 2.4.3. This method is fast but the speed comes at the expense of weaker bound tightening.

OBBT

The second approach to tighten bounds in the ReLU networks uses OBBT. Among the OBBT procedures presented in [23], this method is the least expensive and is referred to as RR. The method uses the LP relaxation of the ReLU networks and computes upper and lower bounds for each neuron in each layer, starting from the input layer. While this approach to bound tightening allows bounds on the output of the network to propagate backwards, this is not taken advantage of in the solver.

Mixed

A third approach to bound tightening involves a combination of the OBBT and FBBT methods. By running OBBT for the first few levels of the branch-and-bound tree then switching to FBBT the hope is that stronger bound tightening early on can lead to fathoming sub problems early, thus reducing the number of nodes that have to be explored to find an optimal solution. By then switching to FBBT deeper in the branch-and-bound tree the idea is that the cheaper computation of the FBBT method could allow the solver to explore the remaining branch-and-bound nodes quickly. In the implementation of this bound tightening procedure the solver switches from OBBT to FBBT at a depth of 10.

3.5.3 Caveats of switching bound tightening methods

When performing bound tightening it is important that internal bounds, the bounds on the internal variables, of the ReLU network constraints do not get relaxed between a parent and a child node, even if the bounds on the output problem variable are kept, as this will weaken the relaxation of the ReLU network-constraint. This is mostly evident when using the Mixed method as switching from OBBT to FBBT can result in the relaxation of the ReLU networks being weakened quite a bit. Giving some intuition to this can be done using a simple problem,

$$\begin{aligned}
\max \quad & x_1 + x_2 && (3.13a)\\
s.t. \quad & x_1 + x_2 \leq U && (3.13b)\\
& x_1, x_2 \in [0,1], && (3.13c)
\end{aligned}$$

with a single constraint limited by an upper bound, U, representing the internal dynamics and bounds of a ReLU network constraint. If the value of U increases between a parent and a child in the branch-and-bound tree, say from $U = 1$ to $U = 2$ the optimal value will increase accordingly, even if the bounds on x_1 and x_2, representing the problem variables, remain unchanged.

One approach to solving this is to separate the bound tightening used to tighten problem variable bounds and the bound tightening used to generate the relaxed problem, then only switching to FBBT for generating problem variable bounds. Another approach is to store not only the problem variable bounds, but also the internal bounds of the ReLU network constraints, for each node in the branch-and-bound tree. Of these the latter was chosen for performance reasons, as the speed benefit of switching to FBBT would be negated if OBBT would still have to be performed when generating the network bounds.

3.6 Solver implementation details

3.6.1 Obtaining an upper bound

When using a branch-and-bound algorithm it is not necessary to attempt to find a upper bound for every layer in the branch-and-bound tree. Only running upper bound problem at certain depths can improve the speed of the solver, especially when computing a lower bound is much less computationally expensive. In the solver the upper bounding problem was run for sub problems in the first two layers and in every fourth layer.

3.6.2 Node selection strategy

The node selection strategy implemented is a strategy of selecting the lowest bound first, often called a best-first search. This means selecting the node from $\mathcal{L}$ with the lowest lower bound, $\bar{z}^k$, on the optimal solution. While there are classes of problems where other strategies will outperform a best first approach [12], there are benefits to choosing best first. One benefit is that no node with a lower bound greater than the optimal value will be explored. In addition, as the node with the lowest bound is explored first, the optimality gap will increase for most iterations, which if the algorithm terminates due to time constraints, could give good bounds on the optimal solution and indicate how far the algorithm was from terminating.

3.6.3 Branching Strategy

When generating sub problems there are two choices that need to be made, which variable to branch on, and which value of that variable to branch on. All the variables in the problem class are continuous and as a result a simple variable selection strategy of choosing the branching variable with the largest bound range was implemented. For a branching variable, x_i, the midpoint, $x_i^m = (1/2)(x_i^{ub} + x_i^{lb})$ is chosen as the branching point.

3.6.4 Numerical precision

One challenge inherent to representing continous variables using computers is the issue of numerical precision. The tolerances of both Gurobi and Ipopt have been set to 10^{-6}, when not already the default. To ensure the solver remains consistent, any weights with an absolute value of less than 10^{-6} is set to 0 when the problem is constructed. Because of this a node is considered to be fathomed if the difference between the lower bound of the node and the upper bound is less than 10^{-6}.

3.7 Training and evaluating ANNs

Two different neural network frameworks were used for the training and evaluating of neural networks respectively. For training the ReLU networks TensorFlow [1] was used and for evaluating the neural networks and their gradients PyTorch [45] was used.

The ReLU networks were built with L2 regularization for both the weights and biases of the networks. The weights of the networks were initialized using a method proposed in [20], where samples are drawn from a normal distribution which is truncated depending on the number of neurons in the current and subsequent layer. The networks were all trained using the Adam optimizer. The parameters of the trained networks were stored in files to allow networks to be reused between tests and to allow the weights to be transferred to PyTorch.

When the optimization problems are constructed the weights and biases are loaded from the stored files, and the networks are reconstructed in PyTorch, the layout being inferred from the network parameters. To ensure the representation of the network in PyTorch is the same as the one used to generate the LP relaxations, the weights used to construct the LP are loaded from the PyTorch model. This means any weights smaller than 10^{-6} are set to 0 before being loaded into PyTorch. When Ipopt evaluates a constraint, the constraint will then call PyTorch which will evaluate the network for the given input, then return the output value as well as the gradient of the network at that point.

Chapter 4

Test problems

This section introduces the test problems that will be used to quantify the performance of the spatial branch-and-bound method for solving problems of the problem class presented in Section 3.2. The tests aim to give insight into how the performance of the branch-and-bound approach depends on and scales with network layout and depth, the number of network input variables and the shape and the properties of the underlying function generating the training data.

As the layouts of the neural networks will be changing between tests a shorthand notation is used to describe the layout of neurons. The shorthand will be on the form $a \times b$ where the first number in the shorthand, a, describes the number of hidden layers and the second number, b, describes the number of hidden neurons in each of those layers. For a network with 4 hidden layers with 20 neurons each the shorthand will then be, 4×20.

4.1 Scaling with respect to input dimension

The goal of this test is to give an idea of how the spatial branch-and-bound method scales as the number of input variables, and thus branching variables, increases when trying to optimize a problem with a single ReLU network constraint.

4.1.1 Rosenbrock function

The Rosenbrock function [48] is a non-convex function commonly used to test optimization algorithms and is defined as,

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2. \tag{4.1}$$

With a minimum of 0 at $(x,y) = (1,1)$. There exist multiple higher dimensional generatizations of the function, including,

$$f_N(x_1,...,x_N) = \sum_{i=1}^{N-1} \left[100(x_{i+1}-x_i^2)^2 + (x_i-1)^2\right], \tag{4.2}$$

which is defined for $N \geq 2$ [30].

For the cases of, $N = 2,3,4$, the global minima are $f(1,1) = 0$, $f(1,1,1) = 0$ and $f(1,1,1,1) = 0$ respecitvely. For the four dimensional case Equation 4.2 also has an additional local minimum [49].

This will be the function selected as to generate test problems of increasing dimensionality. A formulation of a test problem of dimension n is

$$\begin{aligned} z = \arg\min_{x,y} \ & y && \text{(4.3a)} \\ \text{s.t.} \quad & f_N(x_1,\dots,x_N) = y, && \text{(4.3b)} \\ & && \text{(4.3c)} \end{aligned}$$

where f is the n-dimensional Rosenbrock function.

4.1.2 The Rosenbrock test

In order to keep the weights of the trained networks reasonably small some tweaks were made to the function. The function that will be sampled is,

$$f_N(x_1,...,x_N) = \sum_{i=1}^{N-1} \left[3(x_{i+1}^2-x_i^2)^2 + (x_i^2-1)^2\right], \tag{4.4}$$

on the range $x \in [-2.048, 2.048]^N$.

In addition to vary the number of inputs, the layout of the ReLU network will vary to give an indication of how dependent solve times are on the depth and width of the networks. The number of inputs will vary from 2 to 4. The Rosenbrock function will be sampled uniformly in the region $x_i \in [-2.2, 2.2], i \in 1, \dots, N$. The number of samples will increase as input size increases, 2500 for N=2, 8000 for N=3 and 10000 for N=4. This results in 50, 20 and 10 sample points along each dimension respectively. Figure 4.1 shows the function that will be sampled in the 2d test case.

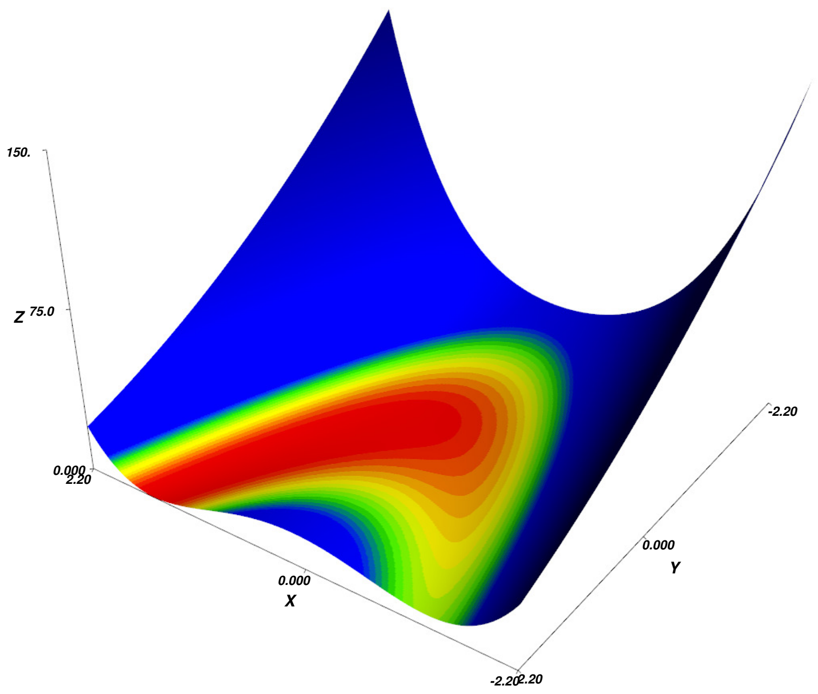

Fig. 4.1 The 2d Rosenbrock test function on $[-2.2, 2.2] \times [-2.2, 2.2]$

For each input dimension, N, four network layouts will be tested. Using the shorthand notation they are 4×20, 2×40, 6×20 and 2×60. These networks have enough neurons to allow a good fit in the 4 dimensional case and pairwise share the same number of neurons, but with one network being wide on one being deep.

4.2 Scaling with respect to the number of neurons

The problem class presented is not restricted in the amount of ReLU network-constraints it can contain. A problem with multiple ReLU network-constraints that share the same input variables could therefore be constructed. For a problem like this changing the layout of all the networks allows for large variations in the number of neurons, without affecting the size of the search space for spatial branch-and-bound. Which could benefit the spatial branch-and-bound approach.

Another property that has an impact on the amount of neurons in a problem is sparsity of the networks used. As mentioned in Section 3.6.4, any weight with an absolute value below 10^{-6} is set to zero to help numerical stability. If this happens with all the weights going in and out of a neuron, it is essentially removed from the network. The sparsity of a network could therefore have an impact on the *effective* number of neurons in the problem.

4.2.1 Test problem with quadratic network constraints

The test consists of four quadratic functions, all in two dimensions. They will all be dependent on the same variables x_1 and x_2. The functions will be numbered 1 to 4 and be defined as follows,

$$f_1(x_1, x_2) = (x_1 - 0.5)^2 + x_2^2 - 1, \tag{4.5a}$$
$$f_2(x_1, x_2) = (x_1 + 1)^2 + x_2^2, \tag{4.5b}$$
$$f_3(x_1, x_2) = x_1^2 + (x_2 + 1)^2, \tag{4.5c}$$
$$f_4(x_1, x_2) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2. \tag{4.5d}$$

Defined on $(x_1, x_2) \in [-2.5, 2.5] \times [-2.5, 2.5]$. The objective that will be minimized is f_1, while the other functions will be constrained by a constant, 1, giving the problem formulation,

$$z = \arg\min_{x,y} \; y \tag{4.6a}$$
$$\text{s.t.} \quad f_1(x_1, x_2) = y \tag{4.6b}$$
$$f_2(x_1, x_2) \leq 1, \tag{4.6c}$$
$$f_3(x_1, x_2) \leq 1, \tag{4.6d}$$
$$f_4(x_1, x_2) \leq 1. \tag{4.6e}$$

For each of the constraints the feasible region is a circle with unit radius centered at $(-1, 0)$, $(0, -1)$ and $(0.5, 0.5)$ for f_2, f_3 and f_4 respectively. These regions overlap and all contain the point $(0, 0)$, which is the optimal solution to the problem, with an objective value of $y = -0.75$.

4.2.2 Multi constraint test

To gauge the effects of the number of neurons in a problem this test will come in two variants, one using sparse networks and one using dense networks. The sparser networks will be trained over a large number of epochs using a low learning rate and the dense networks will be trained over a small number of epochs with a high learning rate.

For each variant the layout of the networks will be varied, with the same network layout being used for each constraint, f_n. The layouts will again be scaled with increasing network depth, following 2×20, 4×20, 6×20 and 8×20, as well as with increasing network width, following 2×20, 2×40, 2×60 and 2×80. All of the functions will be sampled from the same region of $[-2.5, 2.5] \times [-2.5, 2.5]$ using a grid of equally spaced sample points in a 50×50 grid.

$$
\begin{aligned}
z = \arg\min_{x,y} \; & y && (4.7a)\\
\text{s.t.} \quad & f_1\left(x_1, x_2; \boldsymbol{\theta}_1\right) \leq y && (4.7b)\\
& f_2\left(x_1, x_2; \boldsymbol{\theta}_2\right) \leq 1, && (4.7c)\\
& f_3\left(x_1, x_2; \boldsymbol{\theta}_3\right) \leq 1, && (4.7d)\\
& f_4\left(x_1, x_2; \boldsymbol{\theta}_4\right) \leq 1, && (4.7e)
\end{aligned}
$$

where $\boldsymbol{\theta}_n$ represents the parameters of the network trained on f_n.

4.3 Scaling for non-convex functions

One property of the test problems introduced until now is that the functions being sampled are invex, meaning a local minimum is also a global minimum [6]. The exception being the Rosenbrock function in 4 dimensions, which also has a local minimum. This test aims to test how a function with many local minima affects the performance of the spatial branch-and-bound methods. The idea being that for a problem with many local minima relatively close together in objective value, it could be difficult excluding the parts of the feasible region that contain minima through bounding early in the branch-and-bound tree.

4.3.1 Rastrigin function

The Rastrigin function [46], is a function with many local minima. The function is defined as,

$$f(x_1, x_2) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i)), \tag{4.8}$$

and is usually evaluated in $x_i \in [-5.12, 5.12]$. It has a global minimum in the origin as well as several local minima periodically placed as you move away from the origin.

4.3.2 The Rastrigin test

The Rastrigin function, with its many local minima, is quite a bit more complex than the previous functions. In order to ensure that the neural networks capture the shape of the function. without having to resort to using huge networks, the input range of the network is reduced to $x_i \in [-2.2, 2, 2]$. In addition the function uses 5 as the coefficient instead of 10 leading to the function,

$$f(x_1, x_2) = 5n + \sum_{i=1}^{n}(x_i^2 - 5\cos(2\pi x_i)), \tag{4.9}$$

with $x_i \in [-2.2, 2, 2]$. The resulting surface is shown in Figure 4.2

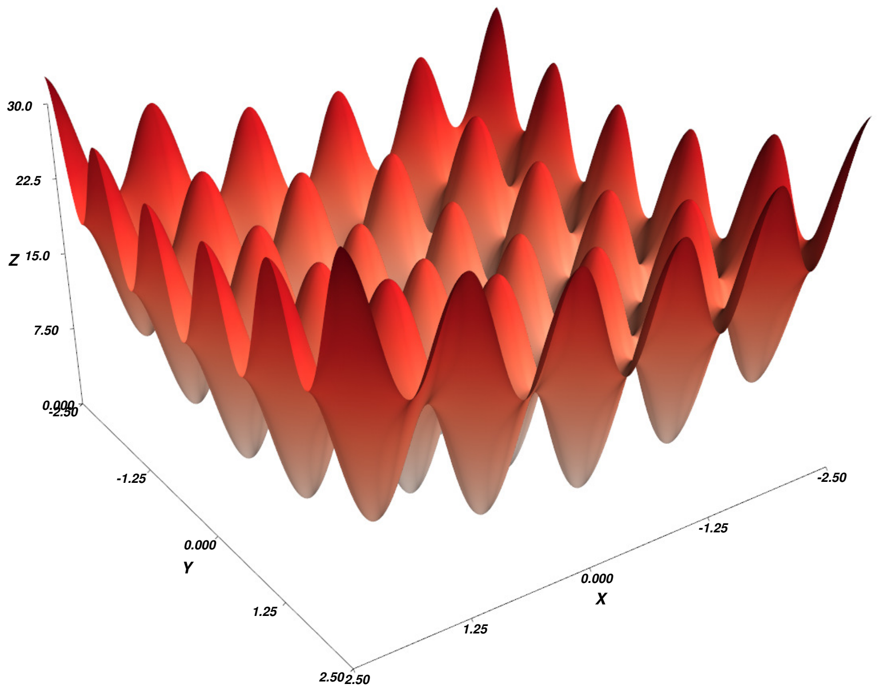

Fig. 4.2 Rastrigin function on the domain

The networks for this test are 4×20, 5×20 and 6×20 for testing scaling as depths increases, and 2×40, 2×50 and 2×60, for testing scaling as the network width increases.

4.4 Production optimization case

In order to test the solver on a problem closer to a practical setting a simplification of the *Oil production optimization case* presented in [23] is used. Using the utility sets,

Set	Description
N	Set of nodes in the network.
N^w	Set of well (source) nodes in the network. $N^w \subset N$.
N^m	Set of manifold nodes in the network. $N^m \subset N$.
N^s	Set of separator (sink) nodes in the network. $N^s \subset N$.
E	Set of edges in the network. An edge $e = (i; j)$ connects node i to node j, where $i, j \in N$.
E^d	Set of discrete edges that can be open or closed. $E^d \subset E$.
E^r	Set of riser edges. $E^r \subset E$.
E_i^{in}	Set of edges entering node i, i.e. $E_i^{in} = \{e : e = (j, i) \in E\}$.
E_i^{out}	Set of edges leaving node i, i.e. $E_i^{in} = \{e : e = (i, j) \in E\}$.
C	$C = \{\text{oil; gas; wat}\}$, denoting the flow rate of oil, gas, and water, respectively.

the original problem proposed was presented as,

$$\max_{y,q,p} \; z = \sum_{e \in E^r} q_{e,oil} \tag{4.10a}$$

$$\text{s.t.} \sum_{e \in E_i^{in}} q_{e,c} = \sum_{e \in E_i^{out}} q_{e,c}, \qquad \forall c \in C, i \in N^m \tag{4.10b}$$

$$p_j = g_e\left(q_{e,oil}, q_{e,gas}, q_{e,wat}, p_i\right), \qquad \forall e \in E^r \tag{4.10c}$$

$$(-p_j^U + p_i^L)(1 - y_e) \leq p_i - p_j \leq (p_i^U + p_j^L)(1 - y_e), \qquad \forall e \in E^d \tag{4.10d}$$

$$\sum_{e \in E_i^{out}} y_e \leq 1, \qquad \forall i \in N^w \tag{4.10e}$$

$$y_e q_{e,c}^L \leq q_{e,c} \leq y_e q_{e,c}^U, \qquad \forall c \in C, e \in E^d \tag{4.10f}$$

$$p_i^L \leq p_i \leq p_i^U, \qquad \forall i \in N \tag{4.10g}$$

$$\sum_{e \in E_i^{out}} q_{e,oil} = f_i\left(p_i\right), \qquad \forall i \in N^w \tag{4.10h}$$

$$\sum_{e \in E_i^{out}} q_{e,gas} = c_{e,gor} \sum_{e \in E_i^{out}} q_{e,oil}, \qquad \forall i \in N^w \tag{4.10i}$$

$$\sum_{e \in E_i^{out}} q_{e,wat} = c_{e,wor} \sum_{e \in E_i^{out}} q_{e,oil}, \qquad \forall i \in N^w \tag{4.10j}$$

$$p_i = p_i^s, \qquad \forall i \in N^s \tag{4.10k}$$

$$y_e \in \{0, 1\}. \qquad \forall e \in E^d \tag{4.10l}$$

The functions f_i in Equation 4.10h and g_e in Equation 4.10c are modeled using ReLU networks. This problem is a routing problem and includes binary variables. The binary variables, y_e, indicate whether or not flow is allowed through an edge.

As the problem class in Equation 3.1 does not include binary variables a simplified version of the problem using fixed routing was constructed. This means the values of y_e will be predefined and fixed. The routing implemented for testing is shown in Table 4.1.

well	1	2	3	4	5	6	7	8
riser	9	10	9	9	9	10	10	10

Table 4.1 The predefined routing used for the oil production optimization case

This test will come in two variants, one using deep networks and one using shallow networks. For flow line networks, g_e, the shallow case and deep case use the network shapes 2×50 and 5×20 respectively. For the well networks, f_i, the network shapes are 2×20 and 4×10.

Chapter 5

Numerical Results

For each test presented in Chapter 4, four different methods for solving the problem were tested. Three of the approaches were variations of the spatial branch-and-bound algorithm, each variation using one of the ReLU bound tightening methods presented in Section 3.5.2. These will be referred to as SBB-FBBT, SBB-OBBT and SBB-Mixed. As a point of comparison all the problems were also constructed as MILPs and solved using Gurobi.

All the tests were given 1800 seconds, or 30 minutes, to complete. In the cases where the the solver did not find an optimal solution within the 30 minute time frame the absolute optimality gap was used as an indicator of how well the solver performed. For cases when the solver is restricted by time, the true scaling for that problem is not known. To distinguish these cases from true growth they are represented using dotted lines when graphed.

The hyperparameters used to train the ReLU networks used in each test will be presented along with resulting error measures. In addition a rudimentary measure of sparsity is included, which will be referred to as sparsity percentage or SP. SP is the percentage of trained network weights with an absolute value below 10^{-6}, meaning the percentage of weights ignored when the problems are constructed.

5.1 Rosenbrock test

The hyperparameters used to train the networks in the Rosenbrock test are shown Table 5.1 and the resulting networks and their properties are shown in Table 5.2.

Dimension	Epochs	batch size	learning rate	L2 regularization
2	1000	32	0.0001	1e-6
3	1000	32	0.0001	1e-6
4	1000	32	0.001	1e-6

Table 5.1 Hyperparameters used to train the networks in the Rosenbrock test.

Dimension, n	Layout	MSE	MAE	SP
2	4×20	0.0357	0.1469	11.90
2	6×20	0.0566	0.1982	29.27
2	2×40	0.1238	0.2180	5.76
2	2×60	0.0102	0.0789	17.22
3	4×20	0.4049	0.4560	7.89
3	6×20	0.2872	0.4149	20.96
3	2×40	0.3888	0.4836	5.23
3	2×60	0.1707	0.3086	1.59
4	4×20	1.4888	0.9540	12.15
4	6×20	0.8874	0.7308	14.81
4	2×40	0.4678	0.5424	3.17
4	2×60	0.2273	0.3718	1.92

Table 5.2 Properties of the trained networks in the rosenbrock test.

Figures 5.1 through 5.4 show how the run times change as the number of inputs increase for each method, with each figure representing the scaling for one network layout. As run times can vary quite drastically the y-axis is logarithmic.

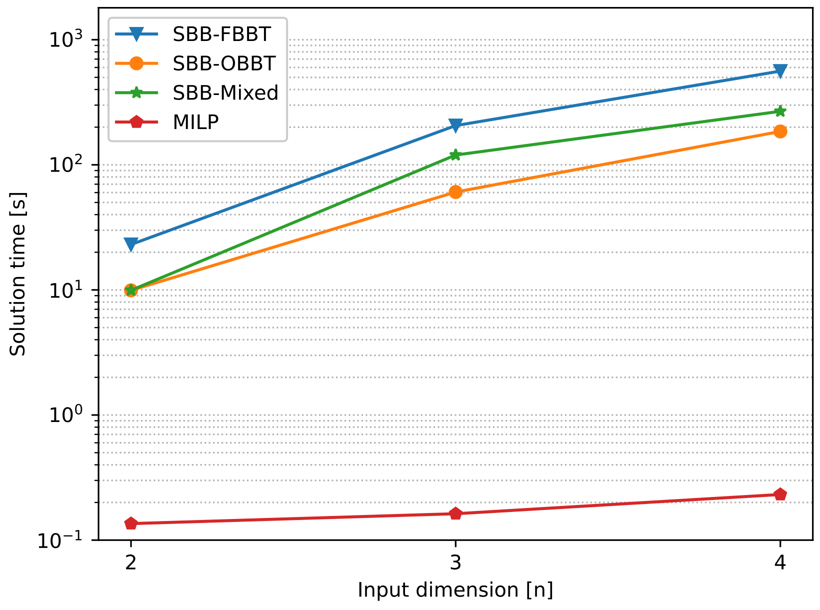

Fig. 5.1 Run times for the Rosenbrock test with a 2×40 network, 2 hidden layers with 40 neurons.

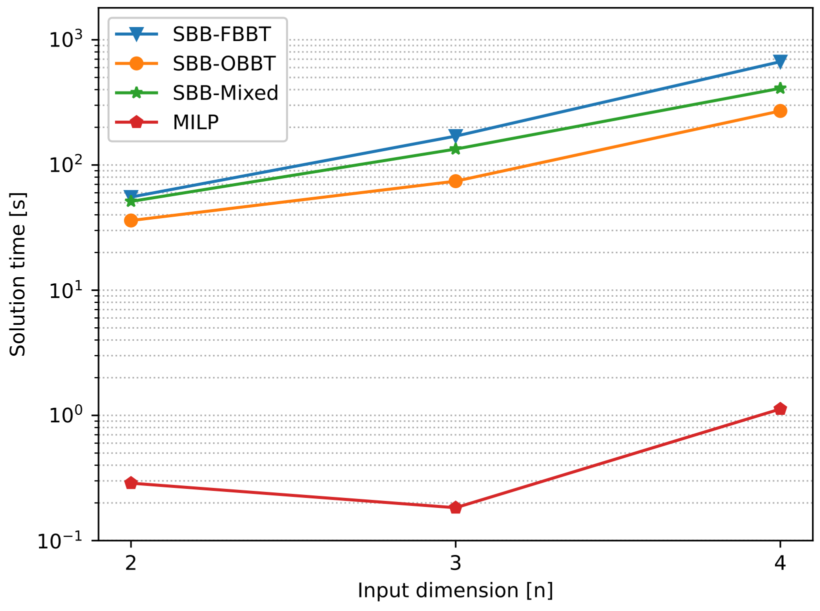

Fig. 5.2 Run times for the Rosenbrock test with a 2×60 network, 2 hidden layers with 40 neurons..

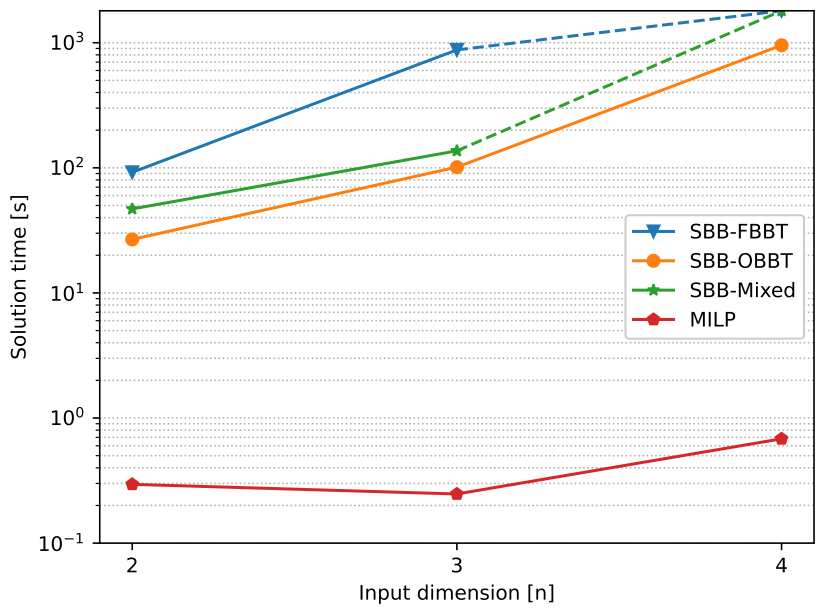

Fig. 5.3 Run times for the Rosenbrock test with a 4×20 network, 2 hidden layers with 40 neurons..

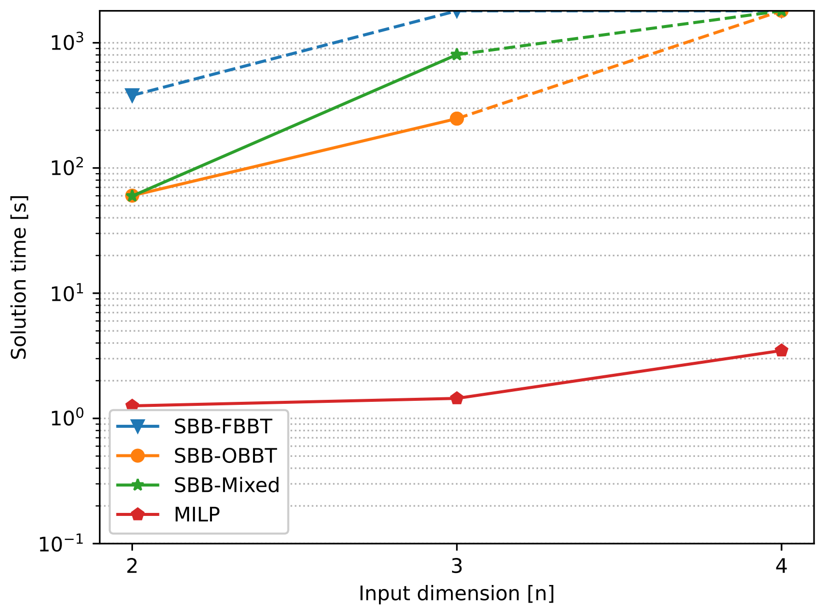

Fig. 5.4 Run times for the Rosenbrock test with a 6×20 network, 2 hidden layers with 40 neurons..

For the tests that did not find an optimal solution withing the 30 minute time limit, Table 5.3 shows the bounds of the branch-and-bound algorithm at the time of termination, the optimality gap and the optimal solution to the problem.

Dim, n	Method	Layout	Optimum	Upper bound	Lower bound	Gap
3	fbbt	6×20	0.3919	0.4749	-5.316	5.791
4	fbbt	4×20	-0.8282	1.8321	-88.837	90.669
4	mixed	4×20	-0.8282	-0.4354	-8.964	8.528
4	fbbt	6×20	1.0398	1.3232	-176.182	177.505
4	obbt	6×20	1.0398	1.1635	-1.795	2.958
4	mixed	6×20	1.0398	1.3208	-9.982	11.303

Table 5.3 Final bounds of the tests that did not find an optimal solution after 30 minutes.

5.2 Multi constraint test

5.2.1 Sparse multi constraint test

For this test all the networks were trained using the same hyperparameters, shown in Table 5.4. Table 5.5 shows the resulting MSE, MAE and sparsity percentage for the networks trained on the objective function, f_1. The corresponding properties for the variable constrained functions, f_2, f_3 and f_4 are available in Table B.1, Table B.2 and Table B.3 respectively.

Epochs	batch size	learning rate	L2 regularization
1000	32	0.001	1e-6

Table 5.4 Hyperparameters used to train the networks in the Multi constraint test.

Function	Layout	MSE	MAE	SP
f1	2×20	0.00098	0.02482	12.17
f1	4×20	0.00147	0.02961	28.97
f1	6×20	0.00100	0.02259	48.88
f1	2×40	0.00048	0.01692	32.91
f1	2×60	0.00071	0.02107	38.81
f1	8×20	0.00149	0.03010	54.65
f1	2×80	0.00058	0.01894	55.29

Table 5.5 Properties of the networks trained on the f_1 function in the multi constraint test.

Figures 5.5 and 5.6 show the solution times for each method as the number of neurons in the the ReLU networks increase, with the x-axis being the number of neurons per network. As the problem has four ReLU network-constraints, the total number of neurons in the problem is four times this number. Figure 5.5 shows how the run times change for the network layouts, 2×20, 2×40, 2×60 and 2×80, where the width of each layer increases by 20 neurons for each test. Similarly Figure 5.6 shows how run times change for the network layouts, 2×20, 4×20, 6×20 and 8×20, where two additional layers of 20 neurons are added for each test.

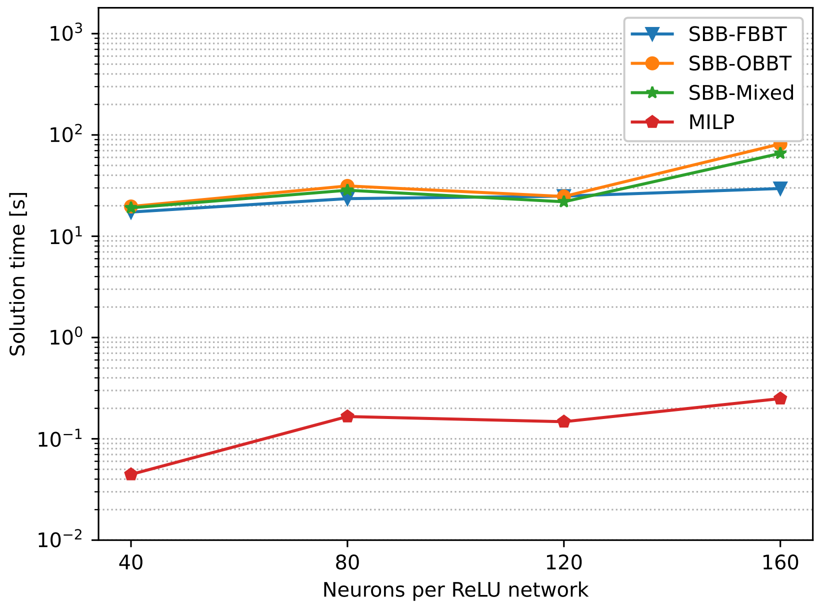

Fig. 5.5 Run time as a function of the total number of neurons as the width of the networks increase from two layers of 20 neurons to two layers of 80 neurons.

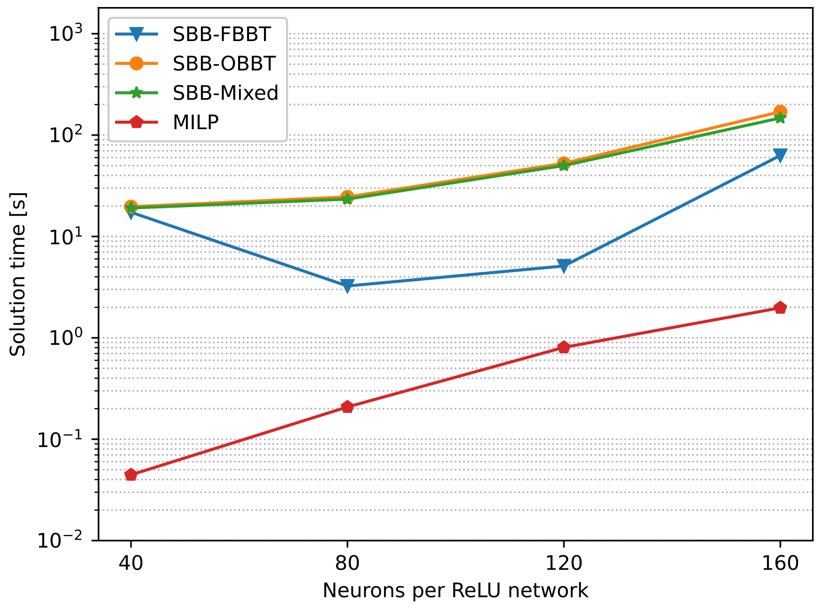

Fig. 5.6 Run time as a function of the total number of neurons as the depth of the networks increase from two layers of 20 neurons to eight layers of 20 neurons.

5.2.2 Dense multi constraint test

All the networks in this test were again trained using the same hyperparameters, but with a higher learning rate over fewer epochs when compared to the sparse test. The hyperparameters are shown in Table 5.6.

Epochs	batch size	learning rate	L2 regularization
50	32	0.01	1e-6

Table 5.6 Hyperparameters used to train the networks in the Multi constraint test.

The resulting network properties for the function f_1 are shown in Table 5.7. The corresponding tables for the other networks, f_2, f_3 and f_4 are shown in Table B.4, Table B.5 and Table B.6 respectively.

Function	Layout	MSE	MAE	SP
f1	2×20	0.02612	0.12881	0.00
f1	4×20	0.00211	0.03619	1.35
f1	6×20	0.00438	0.05054	10.19
f1	2×40	0.00246	0.03762	2.56
f1	2×60	0.00105	0.02278	0.50
f1	8×20	0.00462	0.05398	11.47
f1	2×80	0.00187	0.03174	2.97

Table 5.7 Properties of the networks trained on the f_1 function in the multi constraint test.

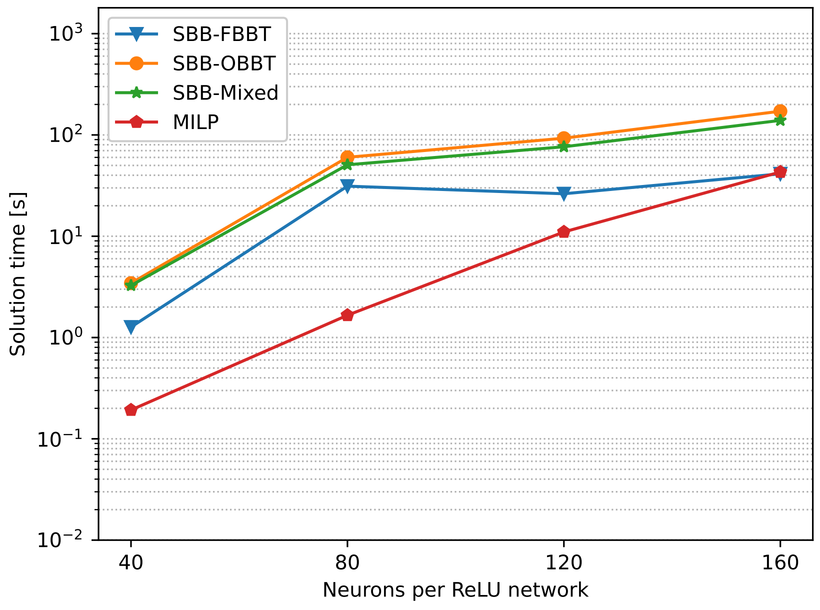

Fig. 5.7 Run time as a function of the total number of neurons as the width of the networks increase from two layers of 20 neurons to two layers of 80 neurons.

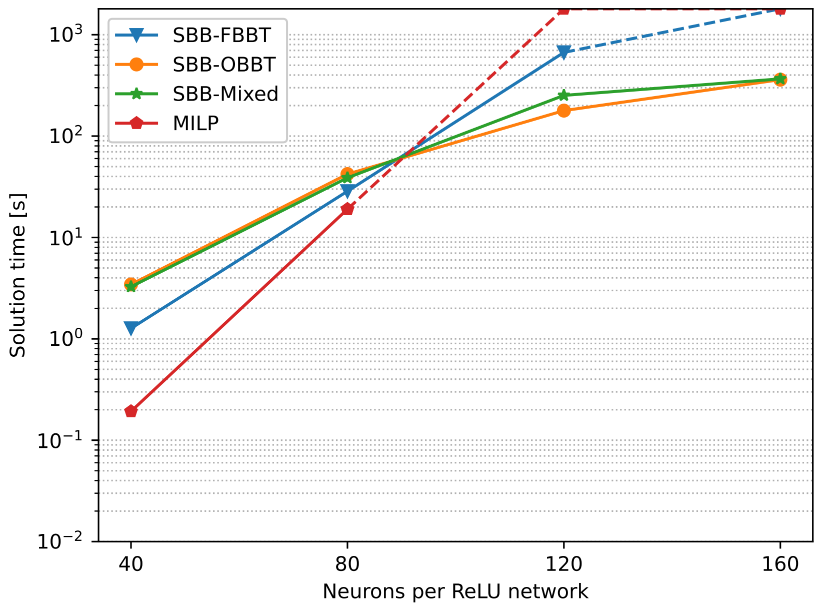

Fig. 5.8 Run time as a function of the total number of neurons as the depth of the networks increase from two layers of 20 neurons to eight layers of 20 neurons.

For the tests that did not find an optimal solution given 30 minutes Table 5.8 shows the bounds of the branch-and-bound algorithm at the time of termination, the optimality gap and the optimal solution to the problem.

Method	Layout	Time	Gap	UB	LB
MILP	6×20	1800.106	inf	inf	-0.94213
SBB-FBBT	8×20	1801.984	0.00306	-0.68182	-0.68488
MILP	8×20	1800.046	inf	inf	-1.96273

Table 5.8 Final bounds of the tests that did not find an optimal solution to the dense multi constraint test after 30 minutes.

5.3 Rastrigin test

The hyperparameters used to train the networks for the Rastrigin test are shown in Table 5.9. Table 5.10 show the resulting mean squared error and mean absolute error.

Epochs	batch size	learning rate	L2 regularization
1000	32	0.001	1e-10

Table 5.9 Hyperparameters used to train the networks in the Rastrigin test.

Figure 5.9 shows the methods scale as the width of the network increases, following 2×40, 2×50 and 2×60. Figure 5.10 shows the methods scale as the depth of the network increases, following the layouts 4×20, 5×20 and 6×20. The bounds of the tests that did not reach an optimal value in 1800 seconds are shown in Table 5.11.

Layout	MSE	MAE	SP
4×20	0.54204	0.58967	0.00
5×20	0.37771	0.48860	33.25
6×20	0.19595	0.34428	37.28
2×40	0.28014	0.41932	30.81
2×50	0.39417	0.49741	35.28
2×60	0.28412	0.42420	37.43

Table 5.10 Properties of the networks trained for the Rastrigin test.

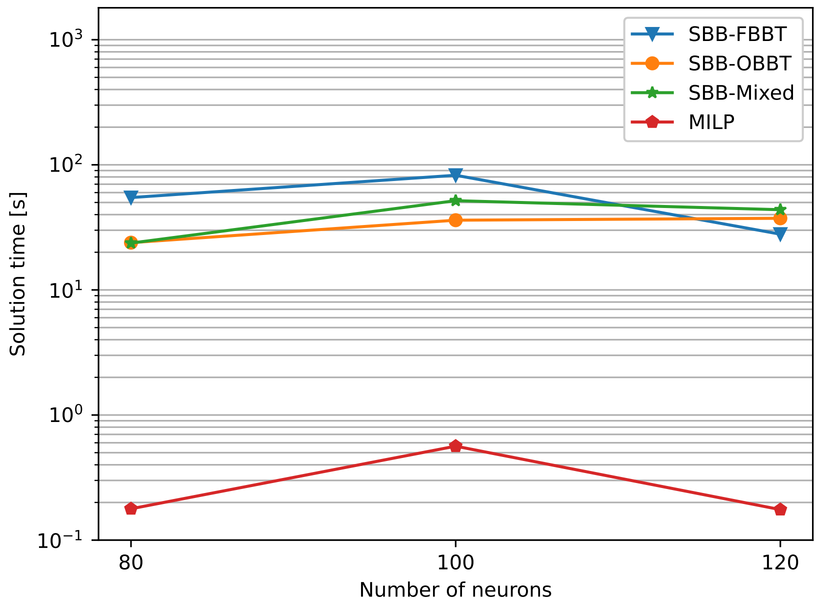

Fig. 5.9 Run times as a function of the total number of neurons in the problem as the width of the networks increase. The layouts used are 2×40, 2×50 and 2×60.

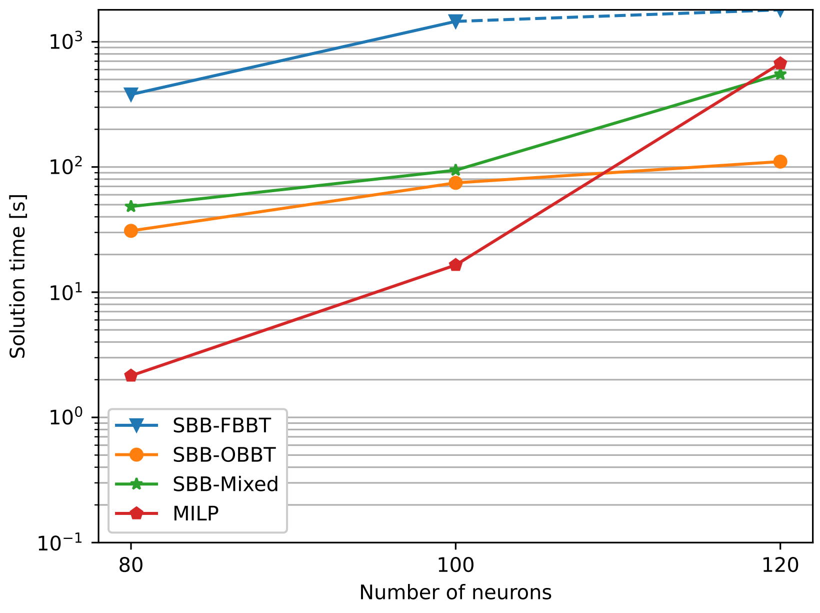

Fig. 5.10 Run times as a function of the total number of neurons in the problem as the depth of the networks increase. The layouts used are 4×20, 5×20 and 6×20.

Method	Layout	Optimum	Upper bound	Lower bound	Gap
SBB-FBBT	2×60	0.154958	0.26300	-12.7918	13.055

Table 5.11 Final bounds of the tests that did not find an optimal solution to the Rastrigin test after 30 minutes.

5.4 Oil Production optimization case

The training data for the well networks, f_i, consists of around 20 data points per well. The well networks were trained over 10000 epochs with a batch size of 10. The training results are shown in Table 5.12.

Well number	Network type	MSE	MAE	SP
1	deep	4.7755e-07	5.6862e-04	0.00
2	deep	7.8293e-07	7.2136e-04	0.00
3	deep	4.9021e-06	1.4810e-03	0.00
4	deep	4.3038e-07	5.2477e-04	0.00
5	deep	9.7869e-07	7.5680e-04	0.00
6	deep	5.4099e-06	1.6059e-03	0.00
7	deep	3.7397e-06	1.5525e-03	0.00
8	deep	4.7427e-07	5.4746e-04	0.00
1	shallow	1.4991e-06	8.7823e-04	0.00
2	shallow	6.9680e-06	1.8215e-03	0.00
3	shallow	3.1112e-06	1.3694e-03	0.00
4	shallow	6.6996e-08	2.2250e-04	0.00
5	shallow	4.0347e-07	5.1624e-04	0.00
6	shallow	5.2044e-06	1.4153e-03	0.00
7	shallow	2.7713e-06	1.4667e-03	0.00
8	shallow	2.5244e-05	3.5042e-03	0.00

Table 5.12 Well network training results

The training data for the flow line networks, g_e, consisted of around 3700 data points. While there are two instances of g_e in the problem, they use the same model. The flow line network was trained over 2000 epochs with a batch size of 32, the resulting network losses are shown in Table 5.13.

Network type	MSE	MAE	SP
deep	7.7698e-06	2.0833e-03	0.00
shallow	7.6071e-06	2.1268e-03	0.00

Table 5.13 Flowline network training results

Table 5.14 shows the resulting run times for the different methods for the shallow and deep test case.

Network type	Method	Time	GAP	Lower Bound	Upper bound
shallow	SBB-FBBT	396.306	2.088e-05	-1.294258	-1.294237
shallow	SBB-OBBT	13.191	1.748e-05	-1.294255	-1.294237
shallow	SBB-Mixed	13.362	1.748e-05	-1.294255	-1.294237
shallow	MILP	0.710	0.000e+00	-1.294237	-1.294237
deep	SBB-FBBT	1800.503	3.369e-01	-1.618563	-1.281682
deep	SBB-OBBT	48.429	5.840e-06	-1.281688	-1.281682
deep	SBB-Mixed	52.852	3.099e-06	-1.281685	-1.281682
deep	MILP	33.575	0.000e+00	-1.281682	-1.281682

Table 5.14 Productioin optimisation results

Chapter 6

Discussion

6.1 Spatial Branch-and-bound performance

6.1.1 Rosenbrock test

The Rosenbrock test was constructed to give an indication of how the spatial branch-and-bound methods scale as the number of inputs to the ReLU network-constraints increase. An increase in the amount of inputs means an increase in the number of branching variables for spatial branch-and-bound. As a result the expected increase is solution times should be exponential, which in a log plot should look like linear growth. For the wide networks tested in Figure 5.1 and Figure 5.2 show that this seems to be the case. The same conclusion can not really reached be for the deep networks from Figure 5.3 and Figure 5.4 however, as a many of the tests did not find an optimal solution in time. SBB-OBBT being the only method to reach an optimum for the 4 dimensional case for the deep network methods.

The order of the bound tightening methods is the same for all tests and network layouts. SBB-OBBT is the fastest, followed by SBB-Mixed with SBB-FBBT being the slowest. For the wide tests all BT procedures seem to scale at close to the same rate. For the deep network it is again difficult to reach a clear conclusion, but it does seem like SBB-FBBT scales worse than SBB-OBBT when considering Figure 5.3 and Figure 5.4 in combination with the results in Table 5.3, where the bounds of SBB-FBBT are much weaker than those of SBB-OBBT and SBB-Mixed.

6.1.2 Multi constraint test

Moving on to the multiple constraint test an interesting observation is that the order in which the methods finish is reversed for quite a few test cases with SBB-FBBT performing better than both SBB-OBBT and SBB-Mixed. This is the case for all the tests in both Figure 5.6 and Figure 5.7.

In the multiple constraint test the performance of SBB-OBBT and SBB-Mixed is close to identical for all tests, the methods following each other much more closely than in the Rosenbrock test. The scaling of these two methods also seem to be fairly predictable, with more neurons giving longer run times. The performance and scaling of SBB-FBBT is a bit more unpredictable with solution times not following clear trends.

6.1.3 Rastrigin test

The results in the Rastrigin test show quite a bit of variation both between width and depth scaling and between the performance of the methods. For the width scaling case in Figure 5.9 all the methods reach a solution in about the same time, independent of the number of neurons, with the order of the methods switching and SBB-FBBT beating out the other methods for the 120 neuron case.

In Figure 5.10, when the depth of the networks increase, these similarities dissappear. In contrast to the wide network case SBB-FBBT is almost an order of magnitude slower than the other methods for the 80 neuron case. For the 100 neuron case the relative performance is even worse and the method does not obtain a solution in the 120 neuron case. In the 80 and 100 neuron case SBB-OBBT and SBB-Mixed scale at about the same rate with comparable performance, but in the 120 neuron case SBB-OBBT is faster by a large margin.

6.1.4 Oil Production optimization Test

The oil production optimization test differs from the other test problems in a few ways, among those having many more problem variables and including linear constraints. Another large difference is that the inputs to the riser networks, g_e, are not all branching variables. Some of the variables are instead the outputs of the well networks, f_i. For this test all problem variables were also given an initial set of bounds.

Looking at Table 5.14 the performance of the SBB-FBBT method is the most interesting aspect. It performs much worse in the shallow network case and does not finish in the deep network case. The suspected reason for this is the combination of weak bound tightening and the inputs to the riser networks being dependent on the outputs of the well networks. With weak bound tightening the output bounds of the well networks are weak. If they are weaker than the initial bounds of the output variable no bound tightening takes place, meaning some of the input bounds to the riser networks staying unchanged. As a result SBB-FBBT has to reduce the input bounds on the well networks significantly before seeing any major bound tightening in the riser networks, leading to slow convergence.

6.1.5 Comparing wide and deep networks

One factor present in all the tests is that they were run with a variety of neural network layouts, typically starting with one base case then scaling the networks either in depth or in width. This results in pairs of tests that have the same number of neurons, but where one network is wide and the other is deep. In general it seems that across all tests and bound tightening methods the spatial branch-and-bound approach scales better if networks increase in width rather than depth.

6.2 Comparing bound tightening methods

This section aims to discuss the performance of each bound tightening method across the different tests and gain some insight what would make one method preferrable over another.

6.2.1 SBB-FBBT

Of the spatial branch-and-bound methods tested SBB-FBBT has the most inconsistent performance, being much faster for some tests than others. Despite by far being the fastest bound tightening procedure to compute, the bounds produced are usually quite weak. The one exception to this being the bounds produced by FBBT on the network approximations of the quadratic functions in the multiple constraint test. The lower bounds FBBT generates for these functions are quite tight from the first iteration and comparable to those of OBBT, leading to branch-and-bound nodes being fathomed much earlier for this test than is usually seen for SBB-FBBT.

For many of the other test problems however FBBT does not produce tight enough bounds to allow nodes to be fathomed early on, leading to large branch-and-bound trees. Ipopt will, as a result of weak bounding, run on sub problems that would have been fathomed using stronger bound tightening. The combination of FBBT being so fast and few nodes being fathomed means that a large portion of the run time is taken up by Ipopt and the upper bound problem.

SBB-FBBT seems to scale pretty similarly to SBB-OBBT and SBB-Mixed with respect to network width increase, but worse with respect to network depth. This might be most evident in the Rastrigin test when comparing Figure 5.9 and Figure 5.10. In addition SBB-FBBT does not scale that well for higher dimensional inputs, Figure 5.2 showing that SBB-FBBT was the only method unable to solve the 3 dimensional case of the Rosenbrock problem for a network size of 6×20.

6.2.2 SBB-OBBT

SBB-OBBT was the best performing method in the Rosenbrock tests and seems to be the method with the most consistent scaling with respect to both network width and depth. Between the Rosenbrock tests and the oil production optimization test it also looks to be the preferred bound tightening method for networks with higher dimensional inputs, SBB-OBBT being the only spatial branch-and-bound method to solve the four dimensional case for the 4 network in Figure 5.3.

6.2.3 SBB-Mixed

In general SBB-Mixed seems to perform in between the SBB-FBBT and SBB-OBBT methods, which does make some sense intuitively. For problems where SBB-OBBT is faster, switching bound tightening to the FBBT leads to an increase in run times over OBBT. Similarly for the case where FBBT is faster, switching will increase the speed at which the method converges compared to pure OBBT. This method was introduced with the hope that it could benefit from OBBT cutting nodes early, then the speed of FBBT could quickly find the solution among the remaining nodes. However it seems that when switching is beneficial the benefit of SBB-Mixed over SBB-OBBT is small. On the other hand, when SBB-FBBT is the slower method and switching is not beneficial, the mixed approach can be quite a lot slower than SBB-OBBT. Figure 5.4 and Figure 5.10 being examples of this.

6.2.4 Choosing a bound tightening strategy

For most cases it seems like SBB-OBBT would be the preferred spatial branch-and-bound method, mostly as a result of being the method with the most predictable scaling. While it is slower than SBB-FBBT in select cases, SBB-FBBT sees comparatively large increases in run times for select problems, especially for deeper networks with higher dimensional inputs. While SBB-Mixed is generally close to SBB-OBBT in the tests presented, it does to some extent experience the same scaling issues that SBB-FBBT faces.

In addition, in the tests where SBB-FBBT performed comparatively well, it had both the benefit of networks for which it could produce strong bounds and of the problems having a comparatively high number of neurons. The run time of a single SBB-FBBT iteration is much lower than that of than an SBB-OBBT iteration, with the difference only increasing as the number of neurons increases.

6.3 MILP performance

For almost every test problem the MILP approach to solving beat the spatial branch-and-bound methods by a large margin, in some cases up to 3 orders of magnitude. There are only four cases where the MILP approach was not the fastest and only one where it can be determined to be the slowest. In the hypothesis presented in Section 3.1, the predicted behavior of the MILP approach was that it would scale exponentially with the number of neurons in an optimization problem. This prediction however did not capture the true performance of the method for all tests.

In both the sparse and dense variants of the multiple constraint test, and for both width and depth increase, the MILP method seems to approximate a line quite closely. This does lend some credibility to the prediction of exponential growth with respect to the number of neurons. At the same time however, the scaling in the Rastrigin test shows another story entirely. The width scaling in Figure 5.9 shows the MILP method stay fast for all network sizes, but the depth scaling in Figure 5.10 shows a huge increase in run time for the same number of neurons.

Like with the spatial branch-and-bound methods, the MILP methods seems to scale better with increased width than increased depth. This can be seen both in the multiple constraint test and the Rastrigin test.

6.4 Effects of network sparsity

One network property that seems to impact performance of the different methods is how sparse the network is. The assumption being that as the number of weights in a network with a value of 0 increases, the number of neurons that have an effect on the output of the network is reduced and, in essence, reduce the size and number of neurons in the network.

Comparing the sparse and dense versions of the multiple constraint test indicate that all the methods are affected by network sparsity to some extent, but that sparsity has the biggest impact on the MILP method. This could indicate sparsity being a significant factor in determining the speed of the MILP method. Looking at the results of other tests however indicates that sparsity does not adequately describe the performance of the MILP method by itself. The 6×20 Rastrigin test has a sparsity percentage of 37.28% and takes just under 700 seconds to complete. The 2 dimensional 6×20 Rosenbrock test has the same network layout and despite having a lower sparsity percentage of 29.27% only takes around a second to solve.

6.5 Implementation related performance factors

6.5.1 Tunable parameters

In addition to the performance effects different types of network introduce, there are also tunable factors in the spatial branch-and-bound solver that could have an impact on both the absolute and relative performance of the different methods. One example is the depth at which SBB-Mixed switches from OBBT to FBBT. As mentioned a relatively large part of the run time of SBB-FBBT is taken up by Ipopt. It could be that reducing the frequency of which Ipopt is run could be beneficial in for SBB-FBBT as more of the runtime would be spent tightening the lower bounds, which could lead to more nodes being fathomed between the layers at which Ipopt is run.

6.5.2 Branch-and-bound implementation

The node and branching point selection implemented in the solver are quite rudimentary. It could be that more advanced strategies for node and branching point selection, like the ones presented in [5], could lead to increased performance.

6.5.3 Choice of programming language

The main purpose of the tests performed was to get an indication of how the spatial branch-and-bound approach scales with respect to different factors, not necessarily achieving the best possible performance. That however does not mean that a more performant implementation of the methods presented could have benefits. As a starting point the current solver was implemented Python, which is an interpreted language and as a result includes some overhead. Using a compiled language, like C++, could therefore prove beneficial.

A more sophisticated implementation could probably also take better advantage of the computer resources available, while Gurobi is able to take full advantage of the CPU and parallel processing, most of the time spent in the spatial branch-and-bound solver is either in Ipopt or constructing sub problems in Python.

Chapter 7

Conclusion

7.1 Hypothesis

The main takeaway from the results presented in Chapter 5 is that the hypothesis, as formulated in Section 3.1, might be too simple to describe the relative scaling of a sBB approach and a MILP approach. The hypothesis stated that for a problem with n inputs and m neurons, a constant, K, should exist such that when $\frac{m}{n} > K$, sBB will outperform MILP.

The scaling however is not as straight forward as more neurons resulting in longer solution times. For starters there are multiple ways in which more neurons can be added to a network and both MILP and sBB seems to scale differently depending on whether networks are made wider or deeper. When adding more neurons both sBB and MILP seems to scale better with wider rather than deeper networks. For the tests where sBB was beat MILP the intersection points did not stay consistent, meaning a different value would be obtained for K if one were to calculate it for each intersection point.

The tests performed also show that there are other factors that can play a significant role in the run times. Depending on the underlying function from which the neural networks were trained the run times can vary significantly, even for problems with the same number of variables and neurons. The sparsity of the networks used can also have a significant impact on performance, however sparsity alone does not necessarily predict performance of either method.

It seems that the performance of either the sBB or MILP approaches is difficult to predict sufficiently from only a single factor that the issue of scaling is more complex than relying just on the ratio of neurons to inputs. There could however still be factors not considered in this thesis that are good indicators of performance.

7.2 Viability of spatial branch-and-bound

From the results presented it is difficult to conclude that the sBB approach is a viable option compared to the MILP approach. Out of only four tests where MILP was not the fastest methods, three were the result of networks being deliberately trained to not achieve sparseness, a property that is usually sought after when training neural networks.

7.3 Further work

While it will not necessarily give better insights into scaling, one exciting direction to take further research would be to expand upon the solver and problem class to include binary variables. This would allow the solver to handle a subset of MINLPs, allowing the solver to work on a larger set of problems with more complex dynamics. One example being the full oil production optimization case instead of the simplified case with fixed routing.

It could also be interesting to implement more advanced branching techniques to see if any large benefits can be gained from more intelligent partitioning of the feasible domain. Similarly exploring different node and branching variable selection strategies could yield interesting results.

This thesis presented a variety of tests for designed to look for scaling with respect to a variety of factors. A more focused study looking at fewer scaling factors might find stronger correlations between certain factors and scaling than found in this thesis.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Amaldi, E., Capone, A., and Malucelli, F. (2003). Planning umts base station location: Optimization models with power control and algorithms. *IEEE Transactions on wireless Communications*, 2(5):939–952.

[3] Belotti, P. (2013). Bound reduction using pairs of linear inequalities. *Journal of Global Optimization*, 56(3):787–819.

[4] Belotti, P., Cafieri, S., Lee, J., and Liberti, L. (2010). Feasibility-based bounds tightening via fixed points. In *International Conference on Combinatorial Optimization and Applications*, pages 65–76. Springer.

[5] Belotti, P., Lee, J., Liberti, L., Margot, F., and Wächter, A. (2009). Branching and bounds tighteningtechniques for non-convex minlp. *Optimization Methods & Software*, 24(4-5):597–634.

[6] Ben-Israel, A. and Mond, B. (1986). What is invexity? *The Journal of the Australian Mathematical Society. Series B. Applied Mathematics*, 28(1):1–9.

[7] Bhosekar, A. and Ierapetritou, M. (2018). Advances in surrogate based modeling, feasibility analysis, and optimization: A review. *Computers & Chemical Engineering*, 108:250 – 267.

[8] Bonami, P., Kilinç, M., and Linderoth, J. (2012). Algorithms and software for convex mixed integer nonlinear programs. In Lee, J. and Leyffer, S., editors, *Mixed Integer Nonlinear Programming*, pages 1–39, New York, NY. Springer New York.

[9] Borchers, B. and Mitchell, J. E. (1994). An improved branch and bound algorithm for mixed integer nonlinear programs. *Computers & Operations Research*, 21(4):359–367.

[10] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

[11] Cai, Z., Fan, Q., Feris, R. S., and Vasconcelos, N. (2016). A unified multi-scale deep convolutional neural network for fast object detection. In *European conference on computer vision*, pages 354–370. Springer.

[12] Clausen, J. and Perregaard, M. (1999). On the best search strategy in parallel branch-and-bound: Best-first search versus lazy depth-first search. *Annals of Operations Research*, 90:1–17.

[13] Delalleau, O. and Bengio, Y. (2011). Shallow vs. deep sum-product networks. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24*, pages 666–674. Curran Associates, Inc.

[14] Eckle, K. and Schmidt-Hieber, J. (2019). A comparison of deep networks with relu activation function and linear spline-type methods. *Neural Networks*, 110:232–242.

[15] Falk, J. E. and Soland, R. M. (1969). An algorithm for separable nonconvex programming problems. *Management science*, 15(9):550–569.

[16] Figueiredo, M. A. (2003). Adaptive sparseness for supervised learning. *IEEE transactions on pattern analysis and machine intelligence*, 25(9):1150–1159.

[17] Fischetti, M. and Jo, J. (2017). Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. *arXiv preprint arXiv:1712.06174*.

[18] Geoffrion, A. M. (1974). Lagrangean relaxation for integer programming. In *Approaches to integer programming*, pages 82–114. Springer.

[19] Gleixner, A. M., Berthold, T., Müller, B., and Weltge, S. (2017). Three enhancements for optimization-based bound tightening. *Journal of Global Optimization*, 67(4):731–757.

[20] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

[21] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

[22] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

[23] Grimstad, B. and Andersson, H. (2019). Relu networks as surrogate models in mixed-integer linear programs. *Computers & Chemical Engineering*, 131:106580.

[24] Grimstad, B. and Sandnes, A. (2016). Global optimization with spline constraints: a new branch-and-bound method based on b-splines. *Journal of Global Optimization*, 65(3):401–439.

[25] Gurobi Optimization, L. (2020). Gurobi optimizer reference manual.

[26] Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier.

[27] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.

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

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

[30] Jamil, M. and Yang, X.-S. (2013). A literature survey of benchmark functions for global optimization problems. *arXiv preprint arXiv:1308.4008*.

[31] Jiang, R., Zhang, M., Li, G., and Guan, Y. (2010). Two-stage robust power grid optimization problem. *submitted to Journal of Operations Research*, pages 1–34.

[32] Kalogirou, S. A. (2000). Applications of artificial neural-networks for energy systems. *Applied energy*, 67(1-2):17–35.

[33] Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311.

[34] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[35] Lai, T.-H. and Sprague, A. (1985). Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, 100(10):962–964.

[36] Land, A. H. and Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520.

[37] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867.

[38] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[39] Neumaier, A. (2004). Complete search in continuous global optimization and constraint satisfaction. *Acta numerica*, 13:271–369.

[40] Ng, A. Y. (2004). Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78.

[41] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization.* Springer, New York, NY, USA, second edition.

[42] Pal, S. K. and Mitra, S. (1992). Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks*, 3(5):683–697.

[43] Park, D. C., El-Sharkawi, M. A., Marks, R. J., Atlas, L. E., and Damborg, M. J. (1991). Electric load forecasting using an artificial neural network. *IEEE Transactions on Power Systems*, 6(2):442–449.

[44] Partridge, D. (1996). Network generalization differences quantified. *Neural Networks*, 9(2):263–271.

[45] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[46] Pohlheim, H. (2007). Examples of objective functions. *Retrieved*, 4(10):2012.

[47] Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Dickstein, J. S. (2017). On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2847–2854. JMLR. org.

[48] Rosenbrock, H. (1960). An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184.

[49] Shang, Y.-W. and Qiu, Y.-H. (2006). A note on the extended rosenbrock function. *Evolutionary Computation*, 14(1):119–126.

[50] Smith, E. and Pantelides, C. (1999). A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex minlps. *Computers & Chemical Engineering*, 23(4):457 – 478.

[51] Snyman, J. A. (2005). *Practical mathematical optimization.* Springer.

[52] Tjeng, V., Xiao, K., and Tedrake, R. (2017). Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356.*

[53] Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57.

Appendix A

Solver results

A.1 Rosenbrock test

Dimension	Method	Layout	Time	Gap	UB	LB
2	SBB-FBBT	4×20	92.094	0.00000	-0.17321	-0.17321
2	SBB-OBBT	4×20	26.775	0.00000	-0.17321	-0.17321
2	SBB-Mixed	4×20	46.908	0.00000	-0.17321	-0.17321
2	MILP	4×20	0.295	0.00000	-0.17321	-0.17321
2	SBB-FBBT	6×20	380.274	0.00000	0.00600	0.00599
2	SBB-OBBT	6×20	60.137	0.00000	0.00599	0.00599
2	SBB-Mixed	6×20	59.722	0.00000	0.00599	0.00599
2	MILP	6×20	1.260	0.00000	0.00599	0.00599
2	SBB-FBBT	2×40	23.074	0.00000	-0.05936	-0.05936
2	SBB-OBBT	2×40	9.918	0.00000	-0.05936	-0.05936
2	SBB-Mixed	2×40	9.933	0.00000	-0.05936	-0.05936
2	MILP	2×40	0.136	0.00000	-0.05936	-0.05936
2	SBB-FBBT	2×60	55.446	0.00000	-0.09512	-0.09512
2	SBB-OBBT	2×60	36.029	0.00000	-0.09512	-0.09512
2	SBB-Mixed	2×60	51.200	0.00000	-0.09512	-0.09512
2	MILP	2×60	0.288	0.00000	-0.09512	-0.09512

Dimension	Method	Layout	Time	Gap	UB	LB
3	SBB-FBBT	4×20	876.003	0.00000	0.20296	0.20296
3	SBB-OBBT	4×20	101.266	0.00000	0.20296	0.20296
3	SBB-Mixed	4×20	136.682	0.00000	0.20296	0.20296
3	MILP	4×20	0.247	0.00000	0.20296	0.20296
3	SBB-FBBT	6×20	1800.025	5.79105	0.47496	-5.31609
3	SBB-OBBT	6×20	247.903	0.00000	0.39194	0.39194
3	SBB-Mixed	6×20	804.016	0.00000	0.39194	0.39194
3	MILP	6×20	1.445	0.00000	0.39194	0.39194
3	SBB-FBBT	2×40	205.878	0.00000	-0.83282	-0.83282
3	SBB-OBBT	2×40	60.602	0.00000	-0.83281	-0.83281
3	SBB-Mixed	2×40	119.770	0.00000	-0.83282	-0.83282
3	MILP	2×40	0.163	0.00000	-0.83281	-0.83281
3	SBB-FBBT	2×60	170.006	0.00000	-0.71651	-0.71651
3	SBB-OBBT	2×60	74.259	0.00000	-0.71651	-0.71651
3	SBB-Mixed	2×60	133.985	0.00000	-0.71651	-0.71651
3	MILP	2×60	0.183	0.00000	-0.71651	-0.71651

Dimension	Method	Layout	Time	Gap	UB	LB
4	SBB-FBBT	4×20	1800.020	90.66903	1.83214	-88.83689
4	SBB-OBBT	4×20	954.261	0.00000	-0.82823	-0.82823
4	SBB-Mixed	4×20	1800.525	8.52840	-0.43549	-8.96390
4	MILP	4×20	0.682	0.00000	-0.82823	-0.82823
4	SBB-FBBT	6×20	1800.222	177.50545	1.32322	-176.18224
4	SBB-OBBT	6×20	1800.068	2.95823	1.16354	-1.79469
4	SBB-Mixed	6×20	1800.267	11.30325	1.32086	-9.98239
4	MILP	6×20	3.483	0.00000	1.03983	1.03983
4	SBB-FBBT	2×40	561.306	0.00000	-3.90748	-3.90748
4	SBB-OBBT	2×40	184.790	0.00000	-3.90748	-3.90748
4	SBB-Mixed	2×40	267.736	0.00000	-3.90748	-3.90748
4	MILP	2×40	0.232	0.00000	-3.90748	-3.90748
4	SBB-FBBT	2×60	668.375	0.00001	-1.40353	-1.40354
4	SBB-OBBT	2×60	270.138	0.00000	-1.40354	-1.40354
4	SBB-Mixed	2×60	409.581	0.00001	-1.40353	-1.40354
4	MILP	2×60	1.125	0.00000	-1.40354	-1.40354

A.2 Multi constraint test

A.2.1 Sparse networks

Method	Layout	Time	Gap	UB	LB
SBB-FBBT	2×20	17.326	0.00000	-0.84450	-0.84450
SBB-OBBT	2×20	19.671	0.00000	-0.84450	-0.84450
SBB-Mixed	2×20	19.080	0.00000	-0.84450	-0.84450
MILP	2×20	0.045	0.00000	-0.84450	-0.84450
SBB-FBBT	2×40	23.506	0.00000	-0.74951	-0.74951
SBB-OBBT	2×40	31.383	0.00000	-0.74951	-0.74951
SBB-Mixed	2×40	28.454	0.00000	-0.74951	-0.74951
MILP	2×40	0.166	0.00000	-0.74951	-0.74951
SBB-FBBT	2×60	24.875	0.00000	-0.73884	-0.73884
SBB-OBBT	2×60	24.685	0.00000	-0.73884	-0.73884
SBB-Mixed	2×60	21.889	0.00000	-0.73884	-0.73884
MILP	2×60	0.148	0.00000	-0.73884	-0.73884
SBB-FBBT	2×80	29.589	0.00000	-0.72206	-0.72206
SBB-OBBT	2×80	81.546	0.00000	-0.72206	-0.72206
SBB-Mixed	2×80	65.767	0.00000	-0.72206	-0.72206
MILP	2×80	0.250	0.00000	-0.72206	-0.72206
SBB-FBBT	4×20	3.246	0.00025	-0.78115	-0.78140
SBB-OBBT	4×20	24.546	0.00025	-0.78115	-0.78140
SBB-Mixed	4×20	23.350	0.00025	-0.78115	-0.78140
MILP	4×20	0.208	0.00000	-0.78115	-0.78115
SBB-FBBT	6×20	5.113	0.00000	-0.79080	-0.79080
SBB-OBBT	6×20	52.633	0.00022	-0.79080	-0.79101
SBB-Mixed	6×20	50.047	0.00000	-0.79080	-0.79080
MILP	6×20	0.806	0.00000	-0.79080	-0.79080
SBB-FBBT	8×20	62.959	0.00000	-0.83866	-0.83866
SBB-OBBT	8×20	169.853	0.00000	-0.83866	-0.83866
SBB-Mixed	8×20	147.507	0.00000	-0.83866	-0.83866
MILP	8×20	1.978	0.00000	-0.83866	-0.83866

A.2.2 Dense networks

Method	Layout	Time	Gap	UB	LB
SBB-FBBT	2×20	1.268	0.00006	-0.81472	-0.81478
SBB-OBBT	2×20	3.440	0.00013	-0.81472	-0.81485
SBB-Mixed	2×20	3.265	0.00003	-0.81472	-0.81475
MILP	2×20	0.193	0.00000	-0.81472	-0.81472
SBB-FBBT	2×40	31.216	0.00000	-0.79019	-0.79019
SBB-OBBT	2×40	60.133	0.00000	-0.79019	-0.79019
SBB-Mixed	2×40	50.736	0.00000	-0.79019	-0.79019
MILP	2×40	1.660	0.00000	-0.79019	-0.79019
SBB-FBBT	2×60	26.264	0.00000	-0.86351	-0.86351
SBB-OBBT	2×60	92.871	0.00000	-0.86351	-0.86351
SBB-Mixed	2×60	76.434	0.00000	-0.86351	-0.86351
MILP	2×60	11.026	0.00000	-0.86351	-0.86351
SBB-FBBT	2×80	41.305	0.00000	-0.81623	-0.81623
SBB-OBBT	2×80	171.422	0.00000	-0.81623	-0.81623
SBB-Mixed	2×80	139.158	0.00000	-0.81623	-0.81623
MILP	2×80	43.074	0.00000	-0.81623	-0.81623
SBB-FBBT	4×20	28.570	0.00000	-0.76896	-0.76896
SBB-OBBT	4×20	42.163	0.00001	-0.76896	-0.76896
SBB-Mixed	4×20	38.764	0.00000	-0.76896	-0.76896
MILP	4×20	19.001	0.00000	-0.76896	-0.76896
SBB-FBBT	6×20	668.753	0.00000	-0.79920	-0.79920
SBB-OBBT	6×20	178.374	0.00000	-0.79920	-0.79920
SBB-Mixed	6×20	252.031	0.00000	-0.79920	-0.79920
MILP	6×20	1800.106	inf	inf	-0.94213
SBB-FBBT	8×20	1801.984	0.00306	-0.68182	-0.68488
SBB-OBBT	8×20	360.324	0.00000	-0.68187	-0.68187
SBB-Mixed	8×20	366.730	0.00000	-0.68187	-0.68187
MILP	8×20	1800.046	inf	inf	-1.96273

A.3 Rastrigin test

Method	Layout	Time	Gap	UB	LB
SBB-FBBT	2×40	54.733	0.00000	-0.34515	-0.34515
SBB-OBBT	2×40	23.869	0.00000	-0.34515	-0.34515
SBB-Mixed	2×40	23.674	0.00000	-0.34515	-0.34515
MILP	2×40	0.178	0.00000	-0.34515	-0.34515
SBB-FBBT	2×50	82.519	0.00000	-0.72492	-0.72492
SBB-OBBT	2×50	36.089	0.00000	-0.72492	-0.72492
SBB-Mixed	2×50	51.684	0.00000	-0.72492	-0.72492
MILP	2×50	0.564	0.00000	-0.72492	-0.72492
SBB-FBBT	2×60	27.898	0.00000	-1.60732	-1.60732
SBB-OBBT	2×60	37.376	0.00000	-1.60732	-1.60732
SBB-Mixed	2×60	43.767	0.00000	-1.60732	-1.60732
MILP	2×60	0.175	0.00000	-1.60732	-1.60732
SBB-FBBT	4×20	379.406	0.00000	-1.44406	-1.44406
SBB-OBBT	4×20	30.904	0.00000	-1.44406	-1.44406
SBB-Mixed	4×20	48.347	0.00000	-1.44406	-1.44406
MILP	4×20	2.152	0.00000	-1.44406	-1.44406
SBB-FBBT	5×20	1455.043	0.00000	-0.52882	-0.52882
SBB-OBBT	5×20	74.692	0.00000	-0.52882	-0.52882
SBB-Mixed	5×20	94.245	0.00000	-0.52882	-0.52882
MILP	5×20	16.473	0.00000	-0.52882	-0.52882
SBB-FBBT	6×20	1800.030	13.05482	0.26300	-12.79182
SBB-OBBT	6×20	110.406	0.00000	0.15496	0.15496
SBB-Mixed	6×20	550.976	0.00000	0.15496	0.15496
MILP	6×20	671.834	0.00000	0.15496	0.15496

Appendix B

Network training results

B.1 Multi constraint test

B.1.1 Sparse networks

Table B.1 Properties of the networks trained on the f2 function in the sparse multi constraint test.

Function	Layout	MSE	MAE	SP
f2	2×20	0.00164	0.03195	34.35
f2	4×20	0.00190	0.03609	37.06
f2	6×20	0.00510	0.05193	47.52
f2	8×20	0.00201	0.03695	55.03
f2	2×40	0.00049	0.01728	30.99
f2	2×60	0.00146	0.02995	37.38
f2	2×80	0.00054	0.01583	55.15

Table B.2 Properties of the networks trained on the f3 function in the sparse multi constraint test.

Function	Layout	MSE	MAE	SP
f3	2×20	0.00226	0.03701	24.35
f3	4×20	0.00184	0.03219	39.84
f3	6×20	0.00408	0.05037	47.14
f3	8×20	0.00079	0.01989	57.48
f3	2×40	0.00080	0.02290	32.73
f3	2×60	0.00072	0.01877	44.31
f3	2×80	0.00040	0.01520	47.38

Table B.3 Properties of the networks trained on the f4 function in the sparse multi constraint test.

Function	Layout	MSE	MAE	SP
f4	2×20	0.00260	0.04243	19.35
f4	4×20	0.00148	0.02821	38.65
f4	6×20	0.00051	0.01762	53.98
f4	8×20	0.00142	0.03013	55.84
f4	2×40	0.00057	0.01842	28.72
f4	2×60	0.00052	0.01785	49.42
f4	2×80	0.00112	0.02613	53.96

B.1.2 Dense networks

Table B.4 Properties of the networks trained on the f2 function in the dense multi constraint test.

Function	Layout	MSE	MAE	SP
f2	2×20	0.03587	0.13680	0.43
f2	4×20	0.00272	0.04045	4.44
f2	6×20	0.01285	0.08711	3.98
f2	8×20	0.00327	0.04657	7.27
f2	2×40	0.00614	0.06140	2.50
f2	2×60	0.00427	0.05299	4.02
f2	2×80	0.00623	0.06735	2.80

Table B.5 Properties of the networks trained on the f3 function in the dense multi constraint test.

Function	Layout	MSE	MAE	SP
f3	2×20	0.01093	0.08372	0.00
f3	4×20	0.00346	0.04473	5.48
f3	6×20	0.00415	0.04664	9.61
f3	8×20	0.00175	0.03232	12.34
f3	2×40	0.00481	0.05313	0.29
f3	2×60	0.00453	0.04851	1.93
f3	2×80	0.00167	0.03126	0.21

Table B.6 Properties of the networks trained on the f4 function in the dense multi constraint test.

Function	Layout	MSE	MAE	SP
f4	2×20	0.00525	0.05455	0.00
f4	4×20	0.00259	0.04036	4.60
f4	6×20	0.00193	0.03399	8.69
f4	8×20	0.00250	0.03842	16.71
f4	2×40	0.00490	0.05384	0.70
f4	2×60	0.00200	0.03519	1.40
f4	2×80	0.00299	0.04298	3.51