universität wien

MASTERARBEIT/MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

"Minimax problems and clustering for regression via neural networks"

verfasst von / submitted by

Daniel de Vicente Jiang, B.Sc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 821
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Mathematik
Betreut von / Supervisor:	emer. o. Univ.-Prof. Dr. Arnold Neumaier

Abstract

Artificial neural networks can approximate any continuous function over a compact set arbitrarily well. Therefore, in this thesis we have attempted to numerically solve the Poisson equation in two dimensions using artificial neural networks. In the literature, the neural network approach has always consisted of minimizing the quadratic loss function over the training set using gradient descent. Motivated by robust optimization, we propose a new approach which transforms the minimization into a minimax problem. At each iteration, a sequence of approximate minimax problems is solved. We examined where the approximation failed the most and clustered those values using the k-means algorithm. To select the new training set we randomly sampled in a neighbourhood of the centroids obtained by k-means. The final solution of the minimax problem yielded the weights of a neural network that approximated the exact solution of the Poisson equation with an error of around 10%.

Zusammenfassung

Jede stetige Funktion kann durch ein künstliches neuronales Netzwerk in einer kompakten Menge beliebig gut approximiert werden. Deswegen haben wir in dieser Thesis versucht, die Poisson-Gleichung numerisch in zwei Dimensionen mithilfe von künstlichen neuronalen Netzen zu lösen. In der Literatur bestand der neuronale Netzwerk Ansatz aus Minimierung der quadratischen Verlustfunktion über den Trainingset mit dem Gradientenverfahren. Motiviert durch die robuste Optimierung haben wir einen anderen Ansatz verwendet, der das Minimierungsproblem in ein Minimaxproblem umwandelt. In jeder Iteration wurde eine Folge von angenährten Minimaxproblemen gelöst. Wir haben Punkte gesucht, bei denen die Annäherung am schlechtesten war und diese mithilfe des k-Means-Algorithmuses geclustered. Außerdem haben wir weitere Punkte in einer geeigneten Umgebung der Centroide zufällig ausgewählt, um das nächste Trainingsset zu finden. Die Lösung des Minimaxproblems lieferte die Gewichte eines neuronalen Netzwerkes, das die exakte Lösung der Poisson-Gleichung mit einem Fehler von zirka 10%.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, emer.o. Univ.-Prof. Dr. Arnold Neumaier, for his incredible support throughout this project. His guidance, experience and overall insight in this field have been extremely helpful. The fact that we could meet every two weeks since the beginning of the project has helped me progress rapidly.
Next, I would like to thank my loving fiancée, Sara Riedel, MA for all her support and motivation in this crucial moment of my life. Thank you for being there for me.
Finally, I would like to thank my friend Ignacio García, BSc for the grammar review and my family, especially my mother and my grandfather for their continued love and support.

Contents

1. Introduction

The field of artificial intelligence has developed rapidly in the last decades. This has been driven on the one hand by an increase in computing power, and on the other hand by the large amounts of data enabled by technological advance. Since machine learning techniques are used in many different subjects there is also a need to develop techniques that are explainable and reliable. Due to their extraordinary flexibility, neural networks are one of the most popular approaches for approximating a desired function. In the past, they have been successfully used to even find the solution of partial differential equations (PDEs) numerically. Some authors claim that neural networks can overcome the curse of dimensionality for some problems see Grohs et al [7]. The goal of this thesis is to find the solution of a vanilla PDE: the Poisson equation with homogeneous Dirichlet boundary conditions. However, instead of solving the classical unconstrained minimization problem via gradient descent, we transform the minimization problem into a minimax problem. The solution of the minimax problem should return the weights of a neural network such that the neural network approximates the desired function in a least squares sense. In order to make the process more transparent we will only use shallow neural networks. We will combine minimax methods from robust optimization with machine learning methods such as regression, clustering and sorting to create a model that achieves a high accuracy on the test set.

1.1. Notation

We denote the derivative of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$ as ∇f. If the function f has two arguments, say x, z, then e.g. the derivative with respect to the first argument is denoted by $\nabla_x f$. The Hessian of a twice-differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is denoted by $\nabla^2 f$. We denote a shallow neural network by Φ with weights $x = (c, A, b, e)$ where $c \in \mathbb{R}^N$ is a row vector, $A \in \mathbb{R}^{N \times d}$ is a weight matrix, $b \in \mathbb{R}^N$ is a bias vector, and $e \in \mathbb{R}$ is a bias term. The number $N \in \mathbb{N}$ denotes the number of neurons and $d \in \mathbb{N}$ is the input dimension of an input vector z. The total number of parameters is given by $n = \dim x$. The function g refers to the loss function of the neural network.

1.2. Basic results from nonlinear optimization

This section draws heavily on Neumaier [16] and Ulbrich & Ulbrich [29] and covers methods for minimizing (twice-) continuously differentiable real functions $f : \mathbb{R}^n \to \mathbb{R}$ within a set $C \subseteq \mathbb{R}^n$ defined by equations and inequalities. We call the function f the **objective function** and the set C the **feasible domain**. Vectors $x \in C$ are called **feasible**. A **minimization problem** attempts to minimize a function f under the constraints $x \in C$ and is written as

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in C, \end{aligned} \tag{1.1}$$

where the abbreviation **s.t.** stands for **subject to**. Another equivalent, more concise formulation of the same minimization problem is given by

$$\min_{x \in C} f(x).$$

One could also consider the problem of maximizing the function f thus obtaining a **maximization problem**

$$\begin{aligned} \max \quad & f(x) \\ \text{s.t.} \quad & x \in C, \end{aligned}$$

However, since this problem is equivalent to a minimization problem

$$\begin{aligned} \min \quad & \tilde{f}(x) \\ \text{s.t.} \quad & x \in C, \end{aligned}$$

where

$$\tilde{f} := -f$$

we will focus only on minimization problems (1.1). A feasible vector $x^* \in C$ is called a **local minimizer** of (1.1) if there exists an $\varepsilon > 0$ with

$$f(x^*) \leqslant f(x) \quad \text{for all } x \in C \cap B_\varepsilon(x^*).$$

The set

$$B_\varepsilon(x^*) := \{x \in \mathbb{R}^n : \|x - x^*\| < \varepsilon\}$$

denotes the ε**-neighbourhood** of x^* and, unless explicitly stated, the norm

$$\|x\| := \sqrt{x^T x}$$

denotes the **Euclidean norm** on $\mathbb{R}^n$. A feasible vector $x \in C$ is called a **global minimizer** if

$$f(x^*) \leqslant f(x) \text{ for all } x \in C. \tag{1.2}$$

We can actually prove the existence of global solutions under certain conditions. The following theorem and its proof can be found on either Neumaier [16] or Ulbrich & Ulbrich [29] with minor differences between them.

Theorem 1.1 (Existence of global solutions)**.** *Let $f : C \to \mathbb{R}$ be continuous and assume that there exists an $\hat{x} \in C$ such that the* ***sublevel set***

$$C_f(\hat{x}) := \{x \in C : f(x) \leqslant f(\hat{x})\}$$

is compact. Then problem (1.1) *hat at least one global minimum.*

Proof. All global minima x^* are, if existing, elements of $C_f(\hat{x})$ since they must satisfy equation (1.2). The **extreme value theorem** states precisely that a continuous function over a compact set is bounded and attains a maximal and minimal value over that set. Hence, the function f attains its minimum at a point $x^* \in C_f(\hat{x})$. This point must also be a global minimum of f at C. □

All optimization problems can be divided into two different classes: unconstrained and constrained optimization problems. In the case of $C = \mathbb{R}^n$ one speaks of an **unconstrained optimization problem**

$$\min_{x \in \mathbb{R}^n} f(x). \tag{1.3}$$

In the case of $C \neq \mathbb{R}^n$ one deals with a **constrained optimization problem**. An unconstrained optimization problem is called convex if the objective function f is convex.
Unconstrained optimization is concerned with solving Problem (1.3). The techniques for solving unconstrained optimization problems can be classified in three main types depending on the information they require. **Zeroth-order methods** or black-box methods use only function

values, **first-order methods** employ function values and gradients and **second-order methods** make use of function values, gradients and Hessians. Amongst the three, second-order methods are the most accurate since they incorporate more information into the problem and thus perform better than the two other methods. However, Hessians are not so cheap to obtain as just gradients. In this section we discuss the gradient descent method, which only assumes that the objective function f is continuously differentiable. All theorems and proofs have been extracted and modified from Ulbrich & Ulbrich [29].

Definition 1.2 (Stationary point)**.** *Let $f : D \to \mathbb{R}$ be differentiable on an open set $D \subseteq \mathbb{R}^n$. A point $x^* \in D$ is called a* ***stationary point*** *if*

$$\nabla f(x^*) = 0.$$

As the following theorem shows, all extreme points are stationary points.

Theorem 1.3 (First-order necessary optimality condition)**.** *Let $f : D \to \mathbb{R}$ be differentiable on an open set $D \subseteq \mathbb{R}^n$. Let $x^* \in D$ be a local minimum of f. Then x^* is a stationary point.*

Proof. Consider the difference quotient

$$\frac{f(x^* + \alpha p) - f(x^*)}{\alpha}.$$

For any arbitrary **direction** $p \in \mathbb{R}^n$ a sufficiently small **step size** $\alpha > 0$ exists, such that the quotient is not zero. Taking the limit $\alpha \to 0^+$ we observe that

$$\lim_{\alpha \to 0^+} \frac{f(x^* + \alpha p) - f(x^*)}{\alpha} = \nabla f(x^*) p \geqslant 0.$$

Since this has to hold for any arbitrary p, we set

$$p := \nabla f(x^*).$$

Hence,

$$-p^T p \geqslant 0 \iff -\|p\|^2 \geqslant 0 \iff p = 0$$

Therefore, $\nabla f(x^*) = 0$ thus making x^* a stationary point. □

Even though every maximum or minimum is a stationary point., not all stationary points are a maximum or a minimum. Such a stationary point is called a **saddle point**. Figure 1.1 shows a stationary point of the function

$$f : \mathbb{R}^2 \to \mathbb{R}, f(x_1, x_2) = x_1^2 - x_2^2.$$

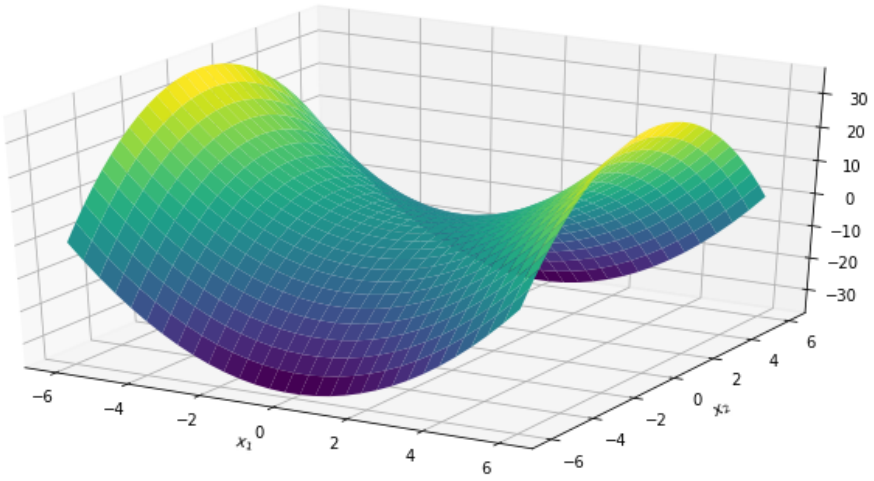


Figure 1.1.: Function f with saddle node at (0,0).

If we set the gradient $\nabla f(x_1, x_2) = 2(x_1, x_2)^T$ of this function f to be zero we will see that the function has only one stationary point at $x^* = (0, 0)$. As shown in Figure 1.1, x^* is neither a maximum nor a minimum since the function f is positive in the x_1-direction and negative in the x_2-direction.

Theorem 1.4 (Second-order necessary optimality condition)**.** *Let $f : D \to \mathbb{R}$ be twice continuously differentiable on an open set $D \subseteq \mathbb{R}^n$. Let $x^* \in D$ be a local minimum of f. Then we have*

(i) $\nabla f(x^*) = 0$

*(ii) The Hessian matrix $\nabla^2 f(x^*)$ is **positive semidefinite**:*

$$p^T \nabla^2 f(x^*) p \geqslant 0 \quad \text{for all } p \in \mathbb{R}^n.$$

Proof. Since *(i)* was already in Theorem 1.3. we will prove *(ii)* directly. Let $p \in \mathbb{R}^n$ be arbitrary. Let $\alpha > 0$ be sufficiently small. Then since x^* is a local minimum it satisfies, in an appropriate ϵ-neighbourhood of x^*:

$$f(x^*) \leqslant f(x^* + \alpha p).$$

Taylor expansion

$$f(x^* + \alpha p) = f(x^*) + \alpha \nabla f(x^*)^T p + \frac{\alpha^2}{2} p^T \nabla^2 f(x^*) p + o(\alpha^2)$$

together with the stationarity condition shows that

$$f(x^*) \leqslant f(x^*) + \frac{\alpha^2}{2} p^T \nabla^2 f(x^*) p + o(\alpha^2).$$

This implies that

$$p^T \nabla^2 f(x^*) p \geqslant -\frac{2o(\alpha^2)}{\alpha^2}.$$

The claim follows from the fact that

$$\lim_{\alpha \to 0} -\frac{2o(\alpha^2)}{\alpha^2} = 0.$$

□

Theorem 1.5 (Second-order sufficient optimality condition)**.** *Let $f : D \to \mathbb{R}$ be twice continuously differentiable on an open set $D \subset \mathbb{R}^n$. Let $x^* \in D$ be a point which satisfies*

(i) $\nabla f(x^*) = 0$

(ii) the Hessian matrix $\nabla^2 f(x^*)$ is ***positive definite****:*

$$p^T \nabla^2 f(x^*) p > 0 \quad \text{for all } p \in \mathbb{R}^n \backslash \{0\}.$$

Then x^* is a strict local minimum of f.

Proof. W.l.o.g. there exists a $\gamma > 0$ such that

$$p^T \nabla^2 f(x^*) p \geqslant \gamma \|p\|^2 \quad \text{for all } p \in \mathbb{R}^n.$$

We can find an $\epsilon > 0$ such that for all $p \in B_\epsilon(x^*)$ holds

$$f(x^* + p) - f(x^*) = \frac{1}{2} p^T \nabla^2 f(x^*) p + o(\|p\|^2) \geqslant \frac{\gamma}{2}\|p\|^2 + o(\|p\|^2) \geqslant \frac{\gamma}{4}\|p\|^2.$$

This implies that

$$f(x^*) + \frac{\gamma}{4}\|p\|^2 \leqslant f(x^* + p)$$

which given that

$$\gamma \|p\|^2 > 0$$

proves that x^* is a strict local minimum. □

Now we will prove another theorem which is central to optimization.

Theorem 1.6 (Local optimality implies global optimality for convex functions)**.** *Let $f : D \to \mathbb{R}$ be convex on a convex set $D \subset \mathbb{R}^n$. Then every local minimum of f in D is a global minimum of f in D.*

Proof. Suppose that x^* is a local minimum of f. Since the set D is convex we have that for any arbitrary $x \in D$ the **direction** $x - x^* \in D$. Since x^* is a local minimum, we can choose a small enough **step size** $\alpha > 0$ such that the point

$$\bar{x} := \alpha(x^* - x)$$

is in an ε-neighbourhood $B_\varepsilon(x^*)$ of x^* where $f(x^*)$ is minimal. This means that

$$f(x^*) \leqslant f(x^* - \alpha(x^* - x)) \tag{1.4}$$

or

$$f(x^*) \leqslant f(x^* + \alpha(x - x^*)).$$

By convexity of f we have that

$$f(x^* + \alpha(x - x^*)) = f(\alpha x + (1 - \alpha)x^*)) \leqslant \alpha f(x) + (1 - \alpha) f(x^*) \tag{1.5}$$

Putting (1.4)–(1.5) together we obtain

$$f(x^*) \leqslant \alpha f(x) + (1 - \alpha) f(x^*) \iff \alpha f(x^*) \leqslant \alpha f(x) \iff f(x^*) \leqslant f(x).$$

Since $x \in D$ was arbitrary it follows

$$f(x^*) \leqslant f(x) \text{ for all } x \in D$$

meaning that x^* must also be a global minimum. □

There are many algorithms for solving an unconstrained optimization problem (1.3) such as gradient methods, Newton methods, Newton-like methods, inexact Newton methods, quasi-Newton methods and trust-region methods. In this thesis only gradient methods will be discussed. We refer to Neumaier [16] and Ulbrich & Ulbrich [29] for a treatise on the rest of the mentioned methods.
A **line search method** is an iterative procedure to find a local minimum x^* of an objective function f. Such a method produces a **descent sequence**

$$f(x^{\ell+1}) < f(x^\ell) \quad \text{for } \ell = 0, 1, \dots \tag{1.6}$$

of feasible vectors $x^\ell \in C$. A line search consists of finding a suitable **step size** $\alpha_\ell > 0$ and a **search direction** $p^\ell \in \mathbb{R}^n$ in each iteration such that the next iterate

$$x^{\ell+1} := x^\ell + \alpha_\ell p^\ell \tag{1.7}$$

produces a descent sequence (1.6). Taylor expansion

$$f(x^\ell + \alpha_\ell p^\ell) = f(x^\ell) + \alpha_\ell \nabla f(x^\ell)^T p^\ell + o(\alpha_\ell)$$

shows that the **descent condition**

$$\nabla f(x^\ell)^T p^\ell < 0 \tag{1.8}$$

ensures a descent sequence (1.6) for sufficiently small $\alpha_\ell > 0$. Once a search direction p^ℓ has been decided, a line search must be found to compute the **step**

$$s^\ell := \alpha_\ell p^\ell$$

A line search which computes the step size α_ℓ as the global minimizer of $f(x^\ell + \alpha_\ell p^\ell)$, i.e. as

$$\alpha_\ell = \arg\min_\alpha \ f(x^\ell + \alpha p^\ell)$$

is called **exact**. Otherwise it is called **inexact**.

Example 1.7. *It makes sense to aim for an exact line search when the objective function f is quadratic and thus can be written as*

$$f(x + \alpha p) = f(x) + \alpha \nabla f(x)^T p + \frac{\alpha^2}{2} p^T \nabla^2 f(x) p.$$

In that case, the exact line search is reduced to solving the following quadratic optimization problem in one dimension:

$$\min_\alpha \ q(\alpha) := f(x) + \alpha \nabla f(x)^T p + \frac{\alpha^2}{2} p^T \nabla^2 f(x) p.$$

The function q has a minimum at an $\hat{\alpha} > 0$ satisfying

$$0 = q'(\alpha) = \nabla f(x)^T p + \alpha p^T \nabla^2 f(x) p.$$

Hence, the optimal $\hat{\alpha}$ is given by

$$\hat{\alpha} = -\frac{f(x)^T p}{p^T \nabla^2 f(x) p}.$$

The direction

$$p^\ell := -\nabla f(x^\ell)$$

is called the **steepest descent direction**. This direction satisfies the descent condition (1.8) if x^ℓ is not an equilibrium point since in this case

$$\nabla f(x^\ell)^T p^\ell = -\nabla f(x^\ell)^T \nabla f(x^\ell) = -\|\nabla f(x^\ell)\|^2 < 0.$$

Any descending method that updates according to

$$x^{\ell+1} = x^\ell - \alpha_\ell \nabla f(x^\ell) \quad \text{for } \ell = 0, 1, \dots \tag{1.9}$$

is called a **gradient descent method**. Now that the descent direction has been determined, we need to compute the step sizes α_ℓ. An easy rule is the **Armijo rule**.

Algorithm 1 : Armijo rule

Input: $\beta \in (0,1), \gamma \in (0,1)$
Output: Step size α_ℓ
Find the largest $\alpha_\ell \in \{1, \beta, \beta^2, \dots\}$ such that α_ℓ satisfies the **Armijo-Goldstein** condition:

$$f(x^\ell + \alpha_\ell p^\ell) - f(x^\ell) \leqslant \gamma \alpha_\ell \nabla f(x^\ell)^T p^\ell$$

Lemma 1.8. *Let $f : D \to \mathbb{R}$ be continuously differentiable on an open set $D \subset \mathbb{R}^n$. Let $\gamma \in (0,1)$ be fixed. Let $x \in \mathbb{R}^n$ and $p \in \mathbb{R}^n$ be a **descent direction** (i.e. it produces a descent sequence). Then an $\overline{\alpha} > 0$ exists with*

$$f(x + \alpha p) - f(x) \leqslant \gamma \alpha \nabla f(x)^T p \quad \text{for all } \alpha \in [0, \overline{\alpha}].$$

Proof. The case $\alpha = 0$ is trivial. Let $\alpha > 0$ be sufficiently small. Then

$$\begin{aligned} \lim_{\alpha \to 0^+} \frac{f(x+\alpha p) - f(x)}{\alpha} - \gamma \nabla f(x)^T p &= \nabla f(x)^T p - \gamma \nabla f(x)^T p \\ &= (1-\gamma) \nabla f(x)^T p < 0 \end{aligned}$$

hence, we can find an $\overline{\alpha} > 0$ such that the condition is fulfilled. □

We can now present the gradient descent method.

Algorithm 2 : Gradient descent

Input: $\beta \in (0,1), \gamma \in (0,1), x^0 \in \mathbb{R}^n$
Output: Stationary point x^*
$\ell \leftarrow 0$
while $\nabla f(x^\ell) \neq 0$ **do**
  $p^\ell \leftarrow -\nabla f(x^\ell)$
  Obtain $\alpha_\ell > 0$ from the Armijo rule, Algorithm 1
  $x^\ell \leftarrow x^\ell + \alpha_\ell p^\ell$
  $\ell \leftarrow \ell + 1$
end while

Now we can prove the convergence of the gradient descent algorithm.

Theorem 1.9 (Ulbrich & Ulbrich [29])**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable. Then Algorithm 2 either*

(i) terminates at a stationary point x^*, or

(ii) produces an infinite sequence $\{x^\ell\}_{\ell \in \mathbb{N}}$ such that
  (a) $f(x^{\ell+1}) < f(x^\ell)$ for all $\ell = 0, 1, \dots$
  (b) any accumulation point of $\{x^\ell\}_{\ell \in \mathbb{N}}$ is a stationary point of f.

Proof. (i) If Algorithm 2 terminates then it produces a stationary point.
(*ii.a*) We consider the case where Algorithm 2 does not terminate and produces an infinite sequence $\{x^\ell\}_{\ell\in\mathbb{N}}$. Because of Lemma 1.8:

$$f(x^{\ell+1}) - f(x^\ell) = f(x^\ell + \alpha_\ell p^\ell) - f(x^\ell) \leqslant \gamma\alpha_\ell \nabla f(x^\ell)^T p^\ell = -\gamma\alpha_\ell \|\nabla f(x^\ell)\|^2 < 0,$$

since $\gamma, \alpha_\ell > 0$. Hence

$$f(x^{\ell+1}) < f(x^\ell).$$

(*ii.b*) Let x^* be an accumulation point of $\{x^\ell\}_{\ell\in\mathbb{N}}$. Let $\{x^\ell\}_{\ell\in L}$ be a subsequence of $\{x^\ell\}_{\ell\in\mathbb{N}}$ where $L \subseteq \mathbb{N}$ and such that $\{x^\ell\}_{\ell\in L} \to x^*$. The sequence $\{f(x^\ell)\}_{\ell\in\mathbb{N}}$ is monotone decreasing by (*ii.a*) and has therefore a limit $\Phi \in \mathbb{R} \cup \{-\infty\}$. Hence $\{f(x^\ell)\}_{\ell\in\mathbb{N}} \to \Phi$. Due to the continuity of f and $\{x^\ell\}_{\ell\in L} \to x^*$ we have also $\{f(x^\ell)\}_{\ell\in L} \to f(x^*)$. Hence $\Phi = f(x^*)$ and

$$f(x^\ell) \to f(x^*) \quad \text{ as } \ell \to \infty.$$

Because of the Armijo rule we have by telescope sum

$$f(x^0) - f(x^*) = \sum_{\ell=0}^{\infty} f(x^\ell) - f(x^\ell + 1) \geqslant \gamma \sum_{\ell=0}^{\infty} \alpha_\ell \|\nabla f(x^\ell)\|^2.$$

Hence

$$\alpha_\ell \|\nabla f(x^\ell)\|^2 \to 0.$$

Now suppose we have $\nabla f(x^\ell) \neq 0$. Due to the continuity of ∇f and to $\{x^\ell\}_{\ell\in L} \to x^*$, there exists a $k \in L$ with

$$\|\nabla f(x^\ell)\| \geqslant \frac{\|\nabla f(x^*)\|}{2} > 0 \quad \text{ for all } \ell \in L \text{ with } \ell \geqslant k.$$

From

$$\alpha_\ell \|\nabla f(x^\ell)\|^2 \to 0$$

follows

$$\{\alpha_\ell\}_{\ell\in L} \to 0.$$

This means in particular, that there exists a $k' \in L$ with $k' \geqslant k$ such that $\alpha_\ell \leqslant \beta$ for all $\ell \in L$ with $\ell \geqslant k'$. By the Armijo rule:

$$f\Big(x^\ell + \frac{\alpha_\ell}{\beta} p^\ell\Big) - f(x^\ell) > \gamma \frac{\alpha_\ell}{\beta} \|\nabla f(x^\ell)\|^2 \quad \text{ for all } \ell \in L, \ell \geqslant k'.$$

Define

$$\{t_\ell\}_{\ell\in L} := \Big\{\frac{\alpha_\ell}{\beta}\Big\}_{\ell\in L}.$$

Then $\{t_\ell\}_{\ell\in L}$ is a null sequence. By the mean value theorem there exist $\tau_\ell \in [0, t_\ell]$ such that

$$\begin{aligned}\lim_{\ell\to\infty, \ell\in L} \frac{f(x^\ell + t_\ell p^\ell) - f(x^\ell)}{t_\ell} &= \lim_{\ell\to\infty, \ell\in L} \frac{t_\ell \nabla f(x^\ell + \tau_\ell p^\ell)^T p^\ell}{t_\ell} \\ &= -\lim_{\ell\to\infty, \ell\in L} \nabla f(x^\ell + \tau_\ell p^\ell)^T \nabla f(x^\ell) = -\|\nabla f(x^*)\|^2\end{aligned}$$

and

$$\lim_{\ell\to\infty, \ell\in L} \|\nabla f(x^\ell)\|^2 = \|\nabla f(x^*)\|^2.$$

Hence we arrive at the contradiction

$$0 < (1-\gamma)\|\nabla f(x^*)\|^2.$$

□

1.3. Organization

This thesis is organized as follows. The second chapter is an introduction to the theory of robust optimization. We explain that uncertainty can arise very frequently in real-life problems. Following Ben-Tal, El Ghaoui & Nemirovski [2] we introduce the chance constrained problem and the robustification problem. The minimax problems are introduced as an example of a robustification problem. We present the cutting plane method as a method to solve the robustification problem in a similar fashion as Pätzhold & Schöbel [20]. Minimax problems can be solved best when they are convex-concave by the gradient descent ascent algorithm, even in $\mathcal{O}(\varepsilon^{-1})$ iterations, see Nemorivski [15]. Nonconvex-concave problems can also be solved by Danskin's procedure [5]. However, there is no theory for solving nonconvex-nonconcave minimax problems.
In the third chapter, all needed tools from machine learning are explained. We present a convergence result for the k-means algorithm based on Shalev-Shwartz & Ben-David [26]. After that we briefly explain the regression problem in the context of machine learning. We explain automatic differentiation as exposed on Neumaier [18]. Lastly, we introduce shallow neural networks. We mention some activation functions and prove the universal approximation theorem in a similar way to Cybenko [4].
The fourth chapter is devoted to the computational experiments. We try to numerically solve the Poisson equation with homogeneous Dirichlet boundary conditions. First, following Chen [3], we prove that the Poisson equation has a unique solution. Motivated by the universal approximation theorem, there exists a neural network that approximates this solution arbitrarily well. We present an existing numerical solution method via neural networks due to Narain [13]. After that, we explain our newly developed method which involves solving a non-convex non-concave minimax problem as well as explain the intuition behind it in a simple example. Finally, we compare the performance of both approaches.
In the fifth and last chapter we summarize the obtained results and propose future research that can be performed. The employed Python code can be found in the Appendix.

2. Robust optimization

This chapter is mainly based on Ben-Tal, El Ghaoui & Nemirovski [2] for an introduction to robust optimization, on Adelhütte, Aßmann, González Grandón et al. [1] for probust constraints, on Pätzhold & Schöbel [20] for cutting plane methods and on Razaviyayn, Huang, Lu et al. [23] for minimax theory.
Uncertainty may arise in optimization problems due to multiple reasons, for example:

(i) **Measurement errors** make real-world data impossible to recover with perfect accuracy.

(ii) **Numerical errors** can add up in each iteration such that the final solution has nothing to due with the true solution.

(iii) In many applications, some of the data are not available prior to the optimization and must be estimated in advanced. This leads to **forecasting errors**.

There is a need to find a solution that remains feasible, while not departing from the true solution and that is **robust** to any realization of scenarios, i.e. **is immunized against uncertainty**.
In an optimization problem, uncertainty can affect the objective function and/or the constraints. By introducing an epigraphic variable, any uncertain optimization problem with uncertain objective function can be equivalently rewritten as an uncertain optimization problem with **certain** objective function and uncertain constraints. Hence, we can focus on such problems w.l.o.g. There are mainly two approaches to deal with uncertainty, namely **stochastic optimization** and **robust optimization**.
In stochastic optimization, uncertain numerical data ξ is assumed to be random and to follow a certain probability distribution P that can be (partially) known in advance. This assumption makes sense when historical data is available and the probability density function of the uncertain parameter ξ can be estimated with high accuracy. The **chance constrained problem**

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & \mathbb{P}_{\xi \sim P}[g(x,\xi) \geqslant 0] \geqslant 1-\varepsilon \end{aligned}$$

is perhaps the most studied one in stochastic optimization, where $f : \mathbb{R}^n \to \mathbb{R}$ denotes the objective function and $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^k$ the constraint mapping. The constraint

$$\mathbb{P}_{\xi \sim P}[g(x,\xi) \geqslant 0] \geqslant 1-\varepsilon$$

means that assuming that the random vector ξ obeys the probability law P, all constraints $g_1, \ldots, g_k$ have to be satisfied with at least a probability of $1-\varepsilon$, where $\varepsilon \in [0,1]$. The shortcut

$$g(x,\xi) \geqslant 0$$

stands for the k one-dimensional inequality constraints

$$\begin{aligned} g_1(x,\xi) &\geqslant 0, \\ &\vdots \\ g_k(x,\xi) &\geqslant 0 \end{aligned}$$

where $g_j : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ are scalar functions, for $j = 1, \ldots, k$. A vector $x \in \mathbb{R}^n$ that satisfies the previous constraint with the prescribed likelihood is called **feasible**.
When the distribution P is only partially known and all it is known is that the law P belongs to a given family $\mathcal{P}$ of probability distributions on the space of data (e.g. the Poisson or the Gaussian distribution), the above chance constraint problem is replaced by the **ambiguous chance constraint problem**

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \mathbb{P}_{\xi \sim P}[g(x, \xi) \geqslant 0] \geqslant 1 - \varepsilon \quad \text{for all } P \in \mathcal{P} \end{aligned}$$

As pointed out by Ben-Tal, El Ghaoui and Nemirovski [2] the stochastic optimization approach tends to be less conservative and to rely more on the law of large numbers. Stochastic optimization makes sense if all the following assumptions are met:

(i) The uncertain data is truly stochastic.

(ii) The family of distributions $\mathcal{P}$ can be narrowed down.

(iii) The decision-maker is willing to accept probabilistic guarantees.

However, when historical data are not available, and therefore no probability distribution function can be estimated, **robust optimization** is resorted to. In this setting, the uncertain parameter z is assumed to belong to a possibly infinite uncertainty set $Z \subseteq \mathbb{R}^m$ so that we arrive at the optimization problem

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g(x, z) \geqslant 0 \quad \text{for all } z \in Z \end{aligned}$$

A third approach is the so-called **probust optimization** which considers a mixture between probabilistic and robust constraints. This can make sense when the uncertain parameter u can be separated into a stochastic and non-stochastic part

$$u = (\xi, z).$$

One practical application is gas transport optimization see Adelhütte, Aßmann, González Grandón et al. [1] where, e.g., on the one hand, historical data for the outgoing loads are available but, on the other hand, observations for the incoming loads are hardly accessible. Many problems can arise with this mixture, for example:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \mathbb{P}_{\xi \sim P}[g(x, \xi, z) \geqslant 0 \text{ for all } z \in Z] \geqslant 1 - \varepsilon \end{aligned}$$

and

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \mathbb{P}_{\xi \sim P}[g(x, \xi, z) \geqslant 0] \geqslant 1 - \varepsilon \quad \text{for all } z \in Z \end{aligned}$$

where the former is more restrictive than the latter. We will focus on robust optimization only, since stochastic optimization and probust optimization are beyond the scope of this thesis.
Following Pätzhold & Schöbel [20], consider an unconstrained optimization problem with uncertainty in the objective function. Let $\mathbb{R}^n$ be the **feasible set** and $Z \subseteq \mathbb{R}^d$ be the **uncertainty set** or the set or all possible scenarios. We call a $z \in Z$ a **scenario**. Let $g : \mathbb{R}^n \times Z \to \mathbb{R}$ be the **objective function**. An **uncertain optimization problem** $(P(z) : z \in Z)$ is a collection

$$\begin{aligned} \min_x \quad & g(x, z) \\ \text{s.t.} \quad & z \in Z \end{aligned}$$

of optimization problems of common structure with data z varying depending on a given uncertainty set Z. The goal is to find a solution $x(z)$ that minimizes the function $g(x, z)$ and also depends on the actually realized scenario z. A family of uncertain optimization problems is not associated with the concepts of optimal value and optimal solution. Given a candidate solution $x \in \mathbb{R}^n$, the **robust value** $\hat{t}(x)$ of the objective in $(P(z) : z \in Z)$ at x is the largest value of the objective over all realizations of the data from the uncertainty set:

$$\hat{t}(x) = \sup_{z \in Z} g(x, z).$$

The **robust counterpart** $\mathrm{Rob}(Z)$ of an uncertain optimization problem $(P(z) : z \in Z)$ is the optimization problem

$$\min_x \hat{t}(x) = \min_x \sup_{z \in Z} g(x, z)$$

of minimizing the robust value over all robust feasible solutions to the uncertain problem. A vector $x \in \mathbb{R}^n$ is called a **robust solution** if it is the optimal solution to $\mathrm{Rob}(Z)$.

2.1. The cutting plane method

A common solution algorithm for solving $\mathrm{Rob}(Z)$ is the **cutting plane approach**. In each iteration $\ell \in \mathbb{N}$ the set Z is approximated by a set Z^ℓ consisting of ℓ vectors $z^1, \ldots, z^\ell \in Z$. The set Z is approximated in the first iteration by its most likely scenario, the nominal scenario z^{nom}.
First, we solve the **robustification problem** $\mathrm{Rob}(Z^\ell)$ which is defined as

$$\min_x \sup_{z \in Z^\ell} g(x, z). \tag{2.1}$$

Observe that this problem is equivalent to the following problem

$$\begin{array}{ll} \min\limits_{x,\omega} & \omega \\ \text{s.t.} & g(x, z^i) \leqslant 0, \quad \text{for all } z^i \in Z^\ell \end{array}$$

By defining the functions

$$f(x, \omega) := \omega, \quad G(x, \omega) := \begin{pmatrix} g(x, z^1) - \omega \\ \vdots \\ g(x, z^\ell) - \omega \end{pmatrix}$$

we obtain the constrained optimization problem

$$\begin{array}{ll} \min\limits_{x,\omega} & f(x, \omega) \\ \text{s.t.} & G(x, \omega) \leqslant 0 \end{array}$$

which can be solved by **sequential quadratic programming** to obtain the optimal vector denoted by (x^ℓ, ω_ℓ).
Second, we solve the **pessimization problem** $\mathrm{Pes}(x^\ell)$

$$\sup_{z \in Z} g(x^\ell, z) \tag{2.2}$$

which is a constrained optimization problem that can be solved using the projected gradient descent to obtain the optimal vector denoted by z^ℓ and the optimal function value denoted by $\Omega_\ell = g(x^\ell, z^\ell)$.

Third, we update $Z^\ell \leftarrow Z^\ell \cup \{z^\ell\}, \ell \leftarrow \ell + 1$. We want the procedure to stop whenever

$$\Omega_\ell - \omega_\ell \leqslant \varepsilon,$$

where $\varepsilon > 0$ is a given degree of optimality since on the ℓth iteration we have

$$\min_x \sup_{z \in Z^\ell} g(x,z) \leqslant \min_x \sup_{z \in Z} g(x,z) \leqslant \sup_{z \in Z} g(x^\ell, z)$$

which by the discussion above is equivalent to

$$\omega_\ell \leqslant \min_x \sup_{z \in Z} g(x,z) \leqslant \Omega_\ell$$

see Pätzhold & Schöbel [20].

Algorithm 3 : Cutting plane

Input: set Z, nominal scenario $z^{nom} \in Z$, degree of optimality $\varepsilon > 0$
Output: robust vector $x^* \in \mathbb{R}^n$ which solves Rob(Z)
$\ell \leftarrow 1$; $Z^\ell \leftarrow \{z^{nom}\}$; $\omega_\ell \leftarrow -\infty$; $\Omega_\ell \leftarrow +\infty$;
while $\Omega_\ell - \omega_\ell > \varepsilon$ **do**
  Obtain x^ℓ, ω_ℓ from (2.1)
  Obtain z^ℓ, Ω_ℓ from (2.2)
  Update $Z^\ell \leftarrow Z^\ell \cup \{z^\ell\}$; $\ell \leftarrow \ell + 1$;
end while

2.2. Minimax problems

Assuming that the function g always attains its maximum with respect to z, the corresponding robust counterpart problem Rob(Z) becomes a minimax problem. A **minimax problem** is an optimization problem defined as

$$\min_{x \in X} \max_{z \in Z} g(x,z), \tag{2.3}$$

where the sets $X \subseteq \mathbb{R}^n$, and $Z \subseteq \mathbb{R}^d$ are called the **feasible sets** and $g : X \times Z \to \mathbb{R}$ is called the **objective function**. The problem

$$\max_{z \in Z} \min_{x \in X} g(x,z)$$

is called **maximin problem**. It is well known that the minimax problem is an upperbound for the maximin problem.

Lemma 2.1 (Max-min inequality)**.**

$$\max_{z \in Z} \min_{x \in X} g(x,z) \leqslant \min_{x \in X} \max_{z \in Z} g(x,z). \tag{2.4}$$

Proof. Suppose that the max-min inequality does not hold. This means

$$\max_{z \in Z} \min_{x \in X} g(x,z) > \min_{x \in X} \max_{z \in Z} g(x,z).$$

We will show that we arrive at a contradiction. For any arbitrary $x^* \in X, z^* \in Z$ it holds that

$$g(x^*, z^*) \geqslant \min_{x \in X} g(x, z^*).$$

Since this holds for any z^* we can choose z^* such that it maximizes the expression

$$\min_{x \in X} g(x, z^*).$$

This implies that

$$\min_{x \in X} g(x, z^*) = \max_{z \in Z} \min_{x \in X} g(x, z).$$

Now by assumption

$$\max_{z \in Z} \min_{x \in X} g(x, z^*) > \min_{x \in X} \max_{z \in Z} g(x, z).$$

We can choose x^* such that it minimizes the expression

$$\max_{z \in Z} g(x, z).$$

This implies

$$\min_{x \in X} \max_{z \in Z} g(x, z) = \max_{z \in Z} g(x^*, z).$$

Now in particular

$$\max_{z \in Z} g(x^*, z) \geqslant g(x^*, z^*)$$

for any point $z^* \in Z$. Putting everything together we have

$$g(x^*, z^*) > g(x^*, z^*).$$

□

By applying the transformations

$$\max f = -\min\{-f\}, \text{ and } \min f = -\max\{-f\},$$

the minimax problem (2.3) is equivalent to the **maximin problem**

$$-\max_{x \in X} \min_{z \in Z} -g(x, z).$$

Hence we can focus the discussion on minimax problems without loss of generality. The problem

$$h(x) := \max_{z \in Z} g(x, z) \tag{2.5}$$

is called the **inner maximization problem** and the problem

$$\min_{x \in X} h(x) \tag{2.6}$$

is the **outer minimization problem**. Minimax problems may arise naturally in any optimization problem containing uncertainty or adversariality. In such a set-up, the goal is to find a solution $x^* \in X$ that is robust against any uncertain scenario $z \in Z$. Minimax problems have been studied in different contexts since von Neumann [19] pioneered **game teory**. Some applications include: Marchi [10] to **zero-sum games**, Goodfellow, Pouget-Abadie, Mirza et al. [6] to **generative adversarial learning** and Madry, Makelov, Schmidt et al. [9] to **deep learning**.
The minimax problem (2.3) is also sometimes called **saddle point problem** because one is interested in finding a **saddle point** of g, i.e. a pair $(x^*, z^*) \in X \times Z$ that satisfies

$$g(x^*, z) \leqslant g(x^*, z^*) \leqslant g(x, z^*) \quad \text{for all } x \in X, z \in Z.$$

One of the simplest algorithm for finding such a saddle point is the **gradient descent ascent** (GDA) algorithm:

$$x^{\ell+1} = x^{\ell} - \gamma_1 \nabla_x g(x^{\ell}, z^{\ell}),$$
$$z^{\ell+1} = z^{\ell} + \gamma_2 \nabla_z g(x^{\ell}, z^{\ell}),$$

where gradient descent is performed on the outer variable x to find an outer minimizer and gradient ascent on the inner variable z to find an inner maximizer. GDA can solve problem (2.3) efficiently when the objective function g is **convex-concave**, i.e. when for a fixed $z \in Z,\ g(\cdot, z)$ is convex in x and for fixed $x \in X,\ g(x, \cdot)$ is convex in z:
Suppose that the function g in the minimax problem (2.3) is and also convex-concave continuously differentiable with derivatives denoted by g_x and g_z. Let both the sets X, Z be convex and compact, define $Y := X \times Z$ and set

$$F : Y \to \mathbb{R}^{n+d}, F(y) := F(x, z) = \begin{pmatrix} g_x(x, z) \\ -g_z(x, z) \end{pmatrix}.$$

In this case, the associated **variational inequality problem**, i.e.

$$\begin{array}{ll} \text{find} & y^* \in Y \\ \text{s.t.} & F(y^*)^T (y - y^*) \geqslant 0 \quad \text{for all } y \in Y \end{array} \tag{2.7}$$

is monotone. The **max-min inequality** is satisfied with equality whenever X, Z are convex and compact and g is convex-concave as a special case of Sion's minimax theorem [28] or Shiffman [27]. An ε-approximate saddle point can be found with rate of convergence $\mathcal{O}(\varepsilon^{-1})$ iterations, either by means of a prox-method see Nemorivski [15], or by subgradient methods see Nedić and Ozdaglar [14].
However, GDA can fail even for simple functions g that are nonconvex with respect to x. There is no general agreement on which algorithm to use. Razaviyayn, Huang, Lu et al. [23] collected many strategies that can be used in case of a nonconvex objective function g. To name only a few:

(i) Approximation of the function F in the variational inequality problem (2.7) by a sequence of strongly monotone mappings and solve a sequence of strongly monotone variational inequality problems.

(ii) An iterative procedure due to Danskin [5] (modified to the gradient descent algorithm):

$$z^{\ell+1} = \arg\max_{z \in Z} g(x^{\ell}, z), \tag{2.8}$$

$$x^{\ell+1} = x^{\ell} - \alpha_{\ell} \nabla_x g(x^{\ell}, z^{\ell+1}), \tag{2.9}$$

which implies computing the exact value $z^{\ell+1}$ at each iteration, which may even be impossible altogether. This procedure requires, however that:

- the objective function g is differentiable in x for all $x \in X$,
- the set Z is compact, and
- the objective function g is strongly concave in z.

3. Machine learning

Russel & Norvig [25] define **machine learning** as a subfield of **artificial intelligence** that studies computational algorithms and methods that gradually improve their performance based on past experiences or observations. Learning consists of making predictions from data that have not being explicitly programmed. A computer receives input data, builds a model based on this data and uses this model as a hypothesis to solve a given problem. Machine learning algorithms can be used to tackle a very broad set of problems, such as document classification, natural language processing and speech processing.

According to Mohri, Rostamizadeh & Talwalkar [12] there are many standard learning tasks. The two most common are **classification** and **regression**. Classification consists of assigning the label to an item, for example assigning a label "spam" or "not spam" to each received email. If there are only two possible labels, then we speak of **binary classification**. If there are more than two labels, it is called a **multi-label classification** problem. Regression focuses on predicting a value for an item, e.g. predicting the price of a house based on several parameters, such as the neighbourhood, the size, and the number of rooms. Other standard learning tasks include **ranking**, which tries to sort the items according a given rule and **dimensionality reduction**, which involves finding a lower-dimensional representation of a set of items while preserving the main features.

A distinction is often made between parameters and hyperparameters. **Parameters** are the variables that are inherent to the model and can be **learned**. **Hyperparameters** are variables that cannot be determined by the learning algorithm and must be specified as inputs to control the learning process. The **training sample** is a set of instances used to train the learning algorithm. The **validation sample** is a small set of instances used to tune the hyperparameters. In practice, a ratio 4:1 between a training set and a validation set is used. Finally, the **test sample** consists of examples for assessing the performance of the learning algorithm. This set should not be accessible in the learning stage.

3.1. Clustering

This section draws on Neumaier [18], Shalev-Shwartz & Ben-David [26] and Petersen [21]. **Unsupervised classification**, also known as **cluster analysis** or **clustering**, is an unsupervised learning technique that consists of dividing a set of m input vectors $S := \{z^1, \dots, z^m\} \subset Z$ into k categories or classes $\{C^1, \dots, C^k\}$, called **clusters**, which are disjoint and whose union is S in a way that "close" objects are grouped together.

Each clustering method compares the objects in a different way, usually using a potential. A **potential** is any function $V : Z \to \mathbb{R}_+$ bounded from below. Two examples of potentials are distance and similarity functions. A **distance function** is a symmetric function $d : Z \times Z \to \mathbb{R}_+$ that satisfies $d(z, z) = 0$, for all $z \in Z$ and also the triangle inequality. A **similarity function** is a symmetric function $s : Z \times Z \to [0, 1]$ that satisfies $s(z, z) = 1$, for all $z \in Z$.

There are two different types of classifiers: hard classifiers and soft classifiers. **Hard classifiers** are deterministic functions that return a label as an output. **Soft classifiers** can be thought of as the stochastic version of hard classifiers and instead return estimated class probabilities. Soft classifiers thus provide more information, but are also more expensive to obtain. This thesis will focus on hard classifiers.

There are many widely used clustering methods, such as **linkage-based clustering**, **spectral clustering**, and k**-means clustering**. In this thesis only k-means clustering will be discussed because it is a special case of the more general **expectation maximization algorithm** (EM-algorithm).
The EM-algorithm is an algorithm for unsupervised or semi-supervised classification problems of the form

$$\text{class}(z) = \underset{i=1,\dots,k}{\arg\min}\, V(z|\theta^i)$$

where $V(z|\theta^i)$ is a potential and θ^i are parameter vectors characterizing the classes. In the **E-step**, appropriate values for missing labels are estimated. In the **M-step**, it is assumed that all class labels are given, and the parameter vectors θ^i are computed using the maximum likelihood method. In the case that $V(z|\theta) = \|z-\theta\|^2$, the EM-algorithm is called k**-means algorithm**. In this case, the goal is to find k clusters and its centroids such that the elements of the clusters are closer to their centroid than to other centroids.
We define the **centroid** of a cluster C as

$$\mu(C) := \frac{1}{|C|}\sum_{z\in C} z,$$

and specifically for each cluster C^i we define

$$\mu^i := \mu(C^i) = \frac{1}{|C^i|}\sum_{z\in C^i} z.$$

We define the objective function to be minimized as

$$G(C^1,\dots,C^k) := \min_{\mu^1,\dots,\mu^k\in Z} \sum_{i=1}^{k}\sum_{z\in C^i} \left\|z-\mu^i\right\|_2^2.$$

Solving this problem can be shown to be equivalent to minimizing the **sum of squares within** which is NP-hard in general. Nevertheless, there exists a naive algorithm called Lloyd's algorithm. Let t be the iteration count and i the class. We define by $C^{i,t}$ the ith cluster and $\mu^{i,t}$ the ith centroid in iteration t.
In Lloyd's algorithm, the E-step consist of estimating the centroids $\mu^{i,t}$ in iteration t. In the M-step, a minimization problem is solved, and the clusters are thus created.

Algorithm 4 : k-MEANS

Input: Set of data points to be clustered $S = \{z^1,\dots,z^m\}$ and number of desired clusters $k \in \{1,\dots,m\}$.
Output: Partition $\{C^1,\dots,C^k\}$ of S.
$t \leftarrow 1$; randomly choose initial centroids $\mu^{1,t},\dots,\mu^{k,t}$;
while not converged **do**
  for $i = 1 : k$ **do**
    $C^{i,t} \leftarrow \{z \in S : i = \underset{j\in\{1,\dots,k\}}{\arg\min} \left\|z-\mu^{j,t}\right\|_2^2\}$;
    $\mu^{i,t} \leftarrow \frac{1}{|C^{i,t}|}\sum_{z\in C^{i,t}} z$;
  end for
  $t \leftarrow t+1$;
end while
for $i = 1 : k$ **do**
  $C^i \leftarrow C^{i,t}$;
end for

Although it is not guaranteed that the output of this algorithm converges to the optimum, it can be proved that each iteration of Lloyd's algorithm does not increase the objective function G. The following theorem and its proof have been adapted from Shalev-Shwartz & Ben-David [26].

Theorem 3.1. *Each iteration of the k-means algorithm (Algorithm 4) does not increase the objective function G.*

Proof. Let $t \in \mathbb{N}$ be the iteration count. Let $C^{1,t}, \ldots, C^{k,t}$ be the partition of S and $\mu^{1,t}, \ldots, \mu^{k,t}$ the centroids produced at iteration t by Algorithm 4. Then, because $\mu^{1,t}, \ldots, \mu^{k,t}$ are the minimizers of $G(C^{1,t}, \ldots, C^{k,t})$ we have

$$G(C^{1,t}, \ldots, C^{k,t}) = \sum_{i=1}^{k} \sum_{z \in C^{i,t}} \left\| z - \mu^{i,t} \right\|_2^2 \leqslant \sum_{i=1}^{k} \sum_{z \in C^{i,t}} \left\| z - \mu^{i,t-1} \right\|_2^2$$

since any minimizer x^* of a function f satisfies $f(x^*) \leqslant f(x)$, for all other x. Moreover,

$$\sum_{i=1}^{k} \sum_{z \in C^{i,t}} \left\| z - \mu^{i,t-1} \right\|_2^2 \leqslant \sum_{i=1}^{k} \sum_{z \in C^{i,t-1}} \left\| z - \mu^{i,t-1} \right\|_2^2 .$$

This is because the $C^{i,t}$ are the minimizers of the expression

$$\sum_{i=1}^{k} \sum_{z \in C^{i}} \left\| z - \mu^{i,t-1} \right\|_2^2$$

over all possible partitions $C^1, \ldots, C^k$ of S due to the update in iteration t. Finally,

$$\sum_{i=1}^{k} \sum_{z \in C^{i,t-1}} \left\| z - \mu^{i,t-1} \right\|_2^2 = G(C^{1,t-1}, \ldots, C^{k,t-1}).$$

Putting it all together we obtain $G(C^{1,t}, \ldots, C^{k,t}) \leqslant G(C^{1,t-1}, \ldots, C^{k,t-1})$, as desired. □

3.2. Regression

Suppose that we want to approximate an unknown function

$$f : Z \to \mathbb{R}, \quad f(z) = y$$

that takes as input a vector $z \in Z$ from the **input space** $Z \subset \mathbb{R}^d$ and returns a **true output** $y \in \mathbb{R}$. All we know about this function is pairs of **training data** $(z^i, y^i)_{i=1}^m$. The **regression** task consists of finding a hypothesis set $H \subset \{Z \mapsto \mathbb{R}\}$ which produces a model

$$\Phi : X \times Z \to \mathbb{R}, \quad \Phi(x, z) = \hat{y}$$

that takes as input a **parameter vector** $x \in \mathbb{R}^n$ and an input vector z and yields a predicted output $\hat{y}$. To assess the quality of the approximation we can specify a **loss function** L that penalizes the differences between true and predicted output. The output of the loss function can be written in many different ways, all equivalent, depending on the context. If we want to signal the different arguments it can take, we write

$$L : \mathbb{R}^n \times Z \times \mathbb{R} \to \mathbb{R}_+, \ (x, z, y) \mapsto L(x, z, y).$$

If we want to remark the dependence with respect to model Φ, we write

$$L : \mathbb{R}^n \times Z \times \mathbb{R} \to \mathbb{R}_+, \ (x, z, y) \mapsto L(\Phi(x, z), y).$$

Finally, if we want to just compare the predicted versus the true output, we write

$$L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+, \ (\hat{y}, y) \mapsto L(\hat{y}, y).$$

Common loss functions include the **mean absolute error**

$$L(\Phi(x, z), y) := |\Phi(x, z) - y|$$

and the **mean squared error**

$$L(\Phi(x, z), y) := (\Phi(x, z) - y)^2.$$

We aim to find the optimal parameter configuration that minimizes the loss function over the training dataset. Therefore we want to solve the optimization problem

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{m} \sum_{i=1}^{m} L(x, z^i, y^i). \tag{3.1}$$

If the loss function L is differentiable with respect to x we can solve (3.1) via **gradient descent** and automatic differentiation.

3.3. Automatic differentiation

This section is largely based on Neumaier [18]. **Automatic differentiation** (AD) is a set of techniques for evaluating the derivatives of a continuously differentiable function f. It is based on the idea that all functions that can be programmed can also be written as a composition of elementary operations and elementary functions. The derivatives can be obtained by applying the chain rule several times and can be obtained with a small cost, multiple of the cost of evaluating the function f. In the context of machine learning, the function f is the real-valued cost function $f : \mathbb{R}^n \to \mathbb{R}$ and one wants to obtain the gradient ∇f of f at a point $x \in \mathbb{R}^n$ as cheaply as possible. Suppose we want to compute the derivative of a function $f : \mathbb{R}^n \to \mathbb{R}^p$ at a point $x \in \mathbb{R}^n$. Denote

$$y = f(x). \tag{3.2}$$

All intermediate results for computing y, which, as we said, are elementary functions and operations, will be stored at the vector z, which is a function that depends on x and also on itself

$$v := H(x, v). \tag{3.3}$$

Therefore, y can be sparsely recovered from v as

$$y = Pv \tag{3.4}$$

where P is a (0,1) matrix that has exactly one 1 in each row. Differentiating (3.3) with respect to x yields

$$\frac{\partial v}{\partial x} = H_x(x, v) + H_v(x, v)\frac{\partial v}{\partial x}$$

which can be rearranged as

$$\frac{\partial z}{\partial x} = (I - H_z(x, z))^{-1} H_x(x, z).$$

By construction, $v_j = H_j(x, v)$ depends only on x and on some of the v_i with $i < j$. Further, $H_v(x, v)$ is a square matrix, hence is $H_v(x, v)$ strictly unit lower triangular. This implies that $I - H_v(x, v)$ is unit lower triangular and, in particular, invertible. Moreover, observe that

$$\nabla f(x) = \frac{\partial y}{\partial x} = P\frac{\partial v}{\partial x} = P(I - H_v(x, v))^{-1}H_x(x, v). \tag{3.5}$$

Now there are two ways to proceed. In **forward AD**, one would solve

$$(I - H_v(x, v))\frac{\partial v}{\partial x} = H_x(x, v)$$

for $\dfrac{\partial v}{\partial x}$ by forward substitution and insert it in (3.5). Since $\dfrac{\partial v}{\partial x}$ and $H_x(x, v)$ are matrices with $n = \dim(x)$ columns, this is equivalent to solving a system

$$AX = B.$$

For each column X_i of X and B_i of B, one solves the linear system of equations

$$AX_i = B_i \quad \text{for } i = 1, \dots, n.$$

In **backward AD**, equation (3.5) is transposed (3.5):

$$\nabla f(x)^T = H_x(x, v)^T(I - H_v(x, v))^{-T}P^T.$$

The matrix $(I - H_v(x, v))^{-T}$ is now unit upper triangular, hence we can solve the following system for a new variable K by backwards substitution with $m = \dim(y)$ columns:

$$(I - H_v(x, v))^T K = P^T.$$

Then $\nabla f(x)$ is obtained as

$$\nabla f(x)^T = K^T H_x(x, v)^T.$$

As we can see, forward AD requires solving n system of equations, while backward AD requires solving p systems of equations. This implies that, on the one hand, forward AD is best suited for computing the derivative of functions $f : \mathbb{R}^n \to \mathbb{R}^p$ whose number of inputs n is less than the number of outputs p. On the other hand, when the number of inputs is smaller than the number of outputs, the backward AD should be chosen.

3.4. Shallow neural networks

We will approximate scalar functions $f : \mathbb{R}^d \to \mathbb{R}$ on a compact subset $Z \subset \mathbb{R}^d$ using shallow neural networks. A **shallow neural network** is a function

$$\Phi : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}, \Phi(x, z) = c\sigma(Az + b) + e = \sum_{i=1}^{N} c_i\sigma\Big(\sum_{j=1}^{d} A_{ij}z_j + b_i\Big) + e,$$

where

$$x = (c, A, b, e)$$

denotes the **parameters** of the neural network. The row vector $c \in \mathbb{R}^N$ and the matrix $A \in \mathbb{R}^{N\times d}$ are called the **weights**, and the column vector $b \in \mathbb{R}^N$ and the scalar $e \in \mathbb{R}$ are called the **biases** of the neural network. The vector $z \in \mathbb{R}^d$ is called the **input** of the neural network. The **number of parameters** is given by

$$n := N(d + 2) + 1,$$

where N is the **number of neurons** in the hidden layer. The following picture depicts a sketch of a shallow neural network.

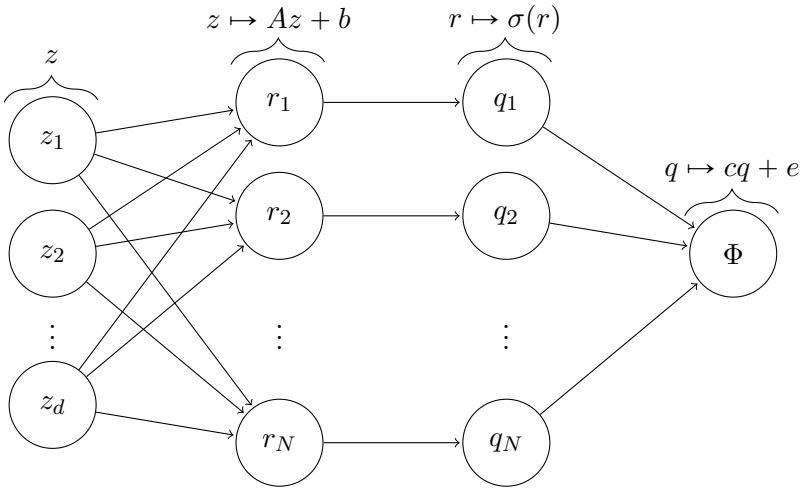


Figure 3.1.: Shallow neural network

Shallow neural networks are a special case of the so-called **multilayer perceptron**.

Definition 3.2 (Multilayer perceptron)**.** *Let $L, d, n_1, \dots, n_L \in \mathbb{N}$, define $d := n_0$ and let $\sigma : \mathbb{R} \to \mathbb{R}$ be a one-dimensional function which is applied coordinate-wise to multi-dimensional input. A multilayer perceptron with activation function σ input d and L layer short $MLP(\sigma, d, L)$ is a function*

$$\Phi : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^{n_L},$$
$$\Phi(z) := A_L \sigma(\dots \sigma(A_1 z + b_1) \dots) + b_L,$$

*where $A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, and $b_\ell \in \mathbb{R}^{n_\ell}$, are the **weight matrices** and **bias vectors**, for $\ell = 0, \dots, L$. The number of parameters is denoted*

$$n := \sum_{\ell=0}^{L} n_\ell n_{\ell-1} + n_\ell.$$

A shallow neural network is a multilayer perceptron with $L = 2$ number of layers. When the number of layers L is greater than 2, we speak about **deep neural networks**. Deep neural networks are a subject of study of **deep learning**.
The class of all multilayer perceptrons with activation function σ, input dimension d and number of layers L is denoted by $MLP(\sigma, d, L)$. Results for the $MLP(\sigma, d, L)$ can also be proved for multi-layer perceptrons with multi-dimensional output by techniques such as **parallelization with shared inputs**. While it is true that any function $\sigma : \mathbb{R} \to \mathbb{R}$ can be used as an activation function, there are some popular choices. Early neural nets such as those implemented by Mc Cullock and Pitts [11], used the **Heaviside** function

$$\sigma_{Heav}(r) := \mathbb{1}_{(0,\infty)}(r)$$

as activation function. Another commonly used activation function is the **rectified linear unit** (ReLU)

$$\sigma_{ReLU}(r) := \max\{0, r\}.$$

None of those two functions is differentiable. Differentiability is a necessary condition on the activation function in order to apply **backpropagation**. The simplest differentiable activation function is the **identity** function

$$\sigma_{Id}(r) := r.$$

However, this is a polynomial function. The approximation capability for polynomial activation functions of degree d is restricted to polynomial functions of degree less or equal than d. Polynomial functions of degree higher than d and nonpolynomial functions cannot be arbitrarily well approximated by a neural network with polynomial activation functions with fixed degree. Hence, we need activation functions that are differentiable and not a polynomial. The inverse tangent

$$\sigma_{arctan}(r) := \arctan r$$

is a function that is not a polynomial and differentiable. However, it is not sigmoidal.

Definition 3.3 (Sigmoidal function)**.** *A continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ is called **sigmoidal** if*

$$\sigma(r) \to \begin{cases} 1 & \text{if } r \to \infty \\ 0 & \text{if } r \to -\infty. \end{cases}$$

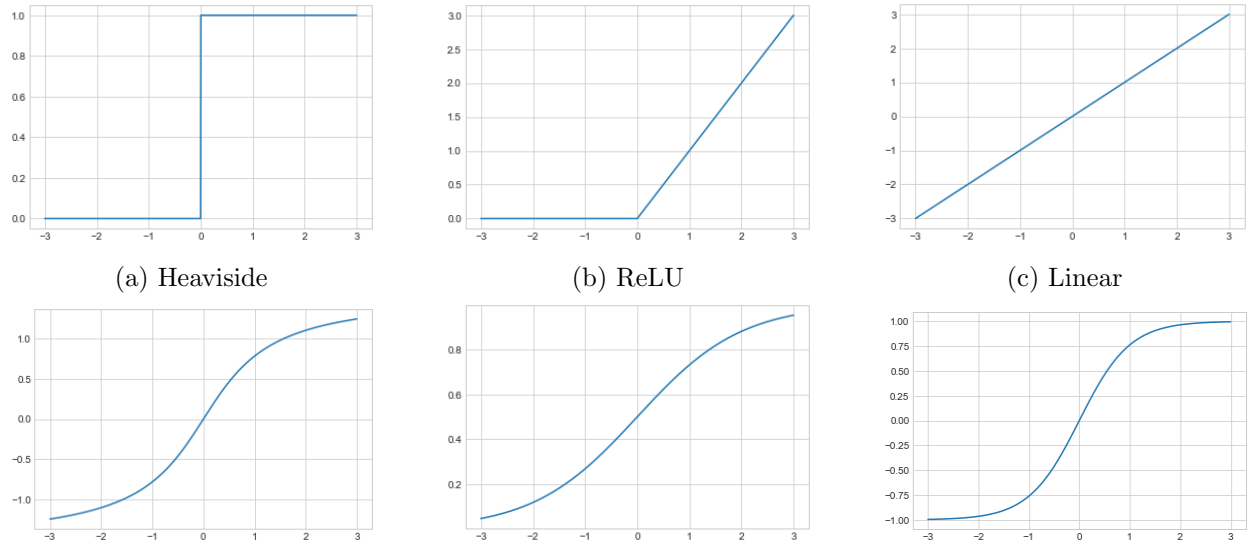
The **logistic** function

$$\sigma_{sigm}(r) := \frac{1}{1 + e^{-r}}.$$

is an example of a sigmoidal function. In some applications, the **hyperbolic tangent**

$$\sigma_{tanh}(r) := \tanh r = \frac{e^r - e^{-r}}{e^r + e^{-r}}$$

is preferred over the logistic function. Figure 3.2 shows the previously discussed activation functions.



(a) Heaviside (b) ReLU (c) Linear

(d) Inverse tangent (e) Sigmoid (f) Hyperbolic tangent

Figure 3.2.: Different activation functions

Sigmoidal functions are particularly interesting, since it can be shown that they are also discriminatory.

Definition 3.4 (Discriminatory function)**.** *Let $d \in \mathbb{N}, K \subset \mathbb{R}^d$ be a compact set. A continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ is called **discriminatory** if the only measure $\mu \in \mathcal{M}(K)$ such that*

$$\int_K \sigma(ax - b)d\mu(x) = 0, \quad \text{for all } a \in \mathbb{R}^d, b \in \mathbb{R}$$

is the zero measure $\mu = 0$.

The hyperbolic tangent is also discriminatory, since it is a shifted version of the logistic function. Once $\sigma_{tanh}(r)$ is already computed, its derivatives are cheap to compute.

Lemma 3.5. *The hyperbolic tangent is arbitrarily often continuously differentiable and its derivatives satisfy:*

(i) $\tanh'(r) = 1 - \tanh^2(r)$,

(ii) $\tanh''(r) = -2\tanh'(r)\tanh(r)$,

(iii) $\tanh'''(r) = -2(\tanh''(r)\tanh(r) + \tanh'(r)^2)$.

Proof. Omitted. Definition of the hyperbolic tangent and chain rule. □

Given that the hyperbolic tangent is arbitrarily often continuously differentiable, bounded, cheap to compute, and discriminatory it will be used as activation function.
Many approximation results exist for the $MLP(\sigma, d, 2)$. In any of them, the approximation function has to satisfy some property. Perhaps the most known result is the **universal approximation theorem**. This theorem was first proved by Hornik, Stinchcombe, & White [8] and Cybenko [4]. It says that every continuous function can be arbitrarily well approximated by a shallow neural network, provided that the activation function is discriminatory.

Theorem 3.6 (Universal approximation theorem)**.** *Let $d \in \mathbb{N}, K \subset \mathbb{R}^d$ compact, and $\sigma : \mathbb{R} \to \mathbb{R}$ be discriminatory. Then* $\mathrm{MLP}(\sigma, d, 2)$ *is **universal**, i.e. is dense in C(K).*

We will present a proof of Theorem 3.6. Before we do that, we need some definitions and results from functional analysis that will be stated without proof. Let $K \subset \mathbb{R}^d$ be a compact set. We define the space

$$C(K) := \{f : K \to \mathbb{R} : f \text{ is continuous}\}$$

with the uniform norm

$$\|f\|_\infty := \sup_{x \in K} |f(x)|.$$

The topological dual space of $C(K)$ is given by

$$\mathcal{M}(K) := \{ \text{ space of all finite, signed nonzero Borel measures on } K\}.$$

Theorem 3.7 (Hahn-Banach)**.** *Let X be a real vector space and $\varphi : X \to \mathbb{R}$ be a convex functional. Let ℓ be a linear functional defined on a subspace Y of X with*

$$\ell(y) \leqslant \varphi(y), \text{ for all } y \in Y.$$

Then there exists an extension $\tilde{\ell}$ to all of X satisfying

$$\ell(y) = \tilde{\ell}, \text{ for all } y \in Y$$

and

$$\tilde{\ell}(y) \leqslant \varphi(y), \text{ for all } y \in X.$$

Proof. Omitted. We refer to Rudin [24]. □

Theorem 3.8 (Riesz representation theorem)**.** *Let ℓ be a continuous linear functional on $C(K)$, K compact Hausdorff. Then there exists a unique $\mu \in M(K)$ such that*

$$\ell(f) = \int_K f d\mu, \text{ for all } f \in C(K).$$

Proof. Omitted. We refer to Rudin [24]. □

Now, we will prove Theorem 3.6. The proof is based on Petersen [22].

Proof of Theorem 3.6. By construction, $MLP(\sigma, d, 2) \subset C(K)$. Let us assume for the sake of contradiction that $MLP(\sigma, d, 2)$ is not dense in $C(K)$. Then, the closure of $MLP(\sigma, d, 2)$ is not $C(K)$, i.e.,

$$\overline{MLP(\sigma, d, 2)} \neq C(K).$$

By the Hahn-Banach theorem, a nonzero linear functional $\ell \in C(K)$ exists such that

$$\ell(MLP(\sigma, d, 2)) = \overline{\ell(MLP(\sigma, d, 2))} = 0.$$

By the Riesz-Representation theorem, ℓ is of the form

$$\ell(f) = \int_K f d\mu,$$

for some $\mu \in M(K)$, for all $f \in C(K)$. In particular, since $f(x) := \sigma(ax - b) \in C(K)$, for all $a \in \mathbb{R}^d, b \in \mathbb{R}$, we have:

$$0 = \ell(\sigma(ax - b)) = \int_K \sigma(ax - b) d\mu(x) \quad \text{for all } a \in \mathbb{R}^d, b \in \mathbb{R}$$

where in the first equality we used Theorem 3.7 and in the second equality we used Theorem 3.8. This is a contradiction to σ being discriminatory. □

As a consequence of the Universal Approximation Theorem, we can approximate any continuous function on a compact interval with a neural network with hyperbolic tangent as activation function. The only problem is that this theorem is not constructive, i.e., it does not give a recipe on **how** to build such a neural network.
The gradient of the loss function $L(\Phi(x, z), y)$ can be found via automatic differentiation. The forward evaluation of the loss function $L(\Phi(x, z), y)$ consists of the four steps

$$r := Az + b, \quad q := \sigma(r), \quad \Phi := cq + e, \quad h := L(\Phi, y)$$

(i) The change dh of h under an $\mathcal{O}(\varepsilon)$ change $d\Phi$ of Φ is

$$dh = L(\Phi + d\Phi, y) - L(\Phi, y) = \nabla_\Phi L(\Phi, y) d\Phi + o(\varepsilon) = \alpha d\Phi + o(\varepsilon)$$

with the scalar $\alpha \in \mathbb{R}$, defined as

$$\alpha := \nabla_\Phi L(\Phi, y). \tag{3.6}$$

Recall that the loss function L is a univariate function $L : \mathbb{R} \to \mathbb{R}$ hence $\alpha \in \mathbb{R}$.

(ii) The change $d\Phi$ of Φ under $\mathcal{O}(\varepsilon)$ changes dc of c, dq of q and de of e is

$$\begin{aligned} d\Phi &= (c+dc)(q+dq)+(e+de)-Cq-e \\ &= cq+cdq+dcq+dcdq+e+de-cq-e = cdq+dcq+dcdq+de \\ &= cdq+dcq+de+o(\varepsilon) \end{aligned}$$

hence,

$$dh = \alpha(cdq+dcq+de)+o(\varepsilon) = adq+\alpha dcq+\alpha de+o(\varepsilon)$$

with the row vector $a \in \mathbb{R}^m$ defined as

$$a := \alpha c. \tag{3.7}$$

(iii) The change dq of q under an $\mathcal{O}(\varepsilon)$ change dr of r is

$$dq = \sigma(r+dr)-\sigma(r)+o(\varepsilon) = \sigma'(r)dr+o(\varepsilon)$$

hence,

$$dh = a\sigma'(r)dr+\alpha dcq+\alpha de+o(\varepsilon) = a'dr+\alpha dcq+\alpha de+o(\varepsilon)$$

with the row vector $a' \in \mathbb{R}^m$ defined as

$$a' := a\sigma'(r). \tag{3.8}$$

(iv) Finally, the change dr of r under $\mathcal{O}(\varepsilon)$ changes dA of A, db of b and dz of z is

$$\begin{aligned} dr &= (A+dA)(z+dz)+(b+db)-Az-b \\ &= Az+Adz+dAz+dAdz+b+db-Az-b = Adz+dAz+db+o(\varepsilon). \end{aligned}$$

Define the row vector $a'' \in \mathbb{R}^d$ as

$$a'' := a'A. \tag{3.9}$$

Then

$$\begin{aligned} dh &= a'(Adz+dAz+db)+\alpha dcq+\alpha de+o(\varepsilon) \\ &= a''dz+a'dAz+a'db+\alpha dcq+\alpha de+o(\varepsilon) \end{aligned} \tag{3.10}$$

From (3.10) we can read off the values of the partial derivatives

$$\frac{dh}{dz_j} = a''_j, \quad \frac{dh}{dc_i} = \alpha q_i, \quad \frac{dh}{de} = \alpha, \quad \frac{dh}{dA_{ij}} = a'_i z_j, \quad \frac{dh}{db_i} = a'_i \tag{3.11}$$

where $i = 1,\dots,m$ and $j = 1,\dots,d$. The reverse sweep consists of computing the adjoint information (3.6)–(3.9) and assembling the gradient from it using (3.11).

Summing up: The regression task via shallow neural networks is solved by applying gradient descent to the unconstrained optimization problem (3.1) to find the optimal parameters $x^* \in \mathbb{R}^n$. The gradient of the loss function L is computed via automatic differentiation.

4. Computational experiments

In this chapter we will try machine learning methods to find the solution of a **partial differential equation** (PDE). Traditional methods for numerically solving PDEs are:

(i) **Finite difference methods**. These are the easiest of all methods. The idea is to discretize the domain Ω into a rectangular grid and to approximate the derivatives at each point of the grid by difference quotients.

(ii) **Finite element methods**. Here the domain Ω is discretized into a mesh and piecewise polynomial functions are fitted through the points. For a complete exposition of finite element methods we refer to Neumaier [16].

(iii) **Scattered data interpolation** using the regression techniques from Section 3.2.

Modern techniques try to approximate the solution of a PDE by neural networks. We know from the universal approximation theorem that a neural network can approximate any continuous function on a compact set arbitrarily well. Moreover, neural networks can overcome the **curse of dimensionality** (i.e. the increase of difficulty with increasing dimension d) for some problems see Grohs et al. [7].

4.1. The Poisson equation

This section is based on both Neumaier [17] and Chen [3] for an introduction to PDEs. Let $\Omega \subset \mathbb{R}^d$ be a **domain**, i.e. a nonempty, open and bounded subset of $\mathbb{R}^d$. We denote by $d \in \mathbb{N}$ the **dimension** and by $|\Omega| < \infty$ the **volume** of the domain Ω. We consider multivariate functions u of d variables $z = (z_1, \dots, z_d)$ ranging over the **closure** $\overline{\Omega}$ of the domain Ω with function values $u(z) = u(z_1, \dots, z_d) \in \mathbb{R}$. We denote by $C^k(\Omega)$ the set of all k **times continuously differentiable real-valued functions** on Ω. The **Laplacian operator** is a **second-order linear differential operator** defined for twice-continuously differentiable functions:

$$\Delta : C^k(\Omega) \to C^{k-2}(\Omega), \quad \Delta u = \sum_{j=1}^{d} \frac{\partial u}{\partial z_j^2}$$

where

$$\frac{\partial u}{\partial z_j}$$

denotes the jth partial derivative of u. The **Laplace equation**

$$\Delta u = 0 \quad \text{on } \Omega \tag{4.1}$$

is the simplest **elliptic partial differential equation** (PDE). Any twice-continuously differentiable function $u \in C^2(\Omega)$ satisfying the Laplace equation (4.1) is called a **harmonic function**. The Laplace equation is a simplified version of the so-called **Poisson equation**

$$\Delta u = f \quad \text{on } \Omega \tag{4.2}$$

which is also an elliptic PDE. The Poisson equation can be used to model the temperature field u in a space Ω when an external heat source f is applied. In order to solve the Poisson equation, we have to prescribe a condition on the boundary of the following form:

$$\alpha u + \beta \frac{\partial u}{\partial n} = g \quad \text{on } \partial\Omega. \tag{4.3}$$

The problem (4.2) combined with a condition (4.3) is called a **boundary value problem**. The term

$$\frac{\partial u}{\partial n}$$

represents the directional derivative of u in direction n, the direction normal to the **boundary** $\partial\Omega$. The choice of scalars $\alpha, \beta \in \mathbb{R}$ gives rise to four main boundary conditions.

(i) **Dirichlet boundary conditions** appear in case $\alpha \neq 0, \beta = 0$:

$$u = g \quad \text{on } \partial\Omega. \tag{4.4}$$

The function g encaptures the constant $\alpha \neq 0$. As we will see later, the boundary problem (4.2) with Dirichlet boundary conditions (4.4) is well-posed when the function g is continuous on the boundary.

(ii) In case $\alpha = 0, \beta \neq 0$ we have **Neumann boundary conditions**

$$\frac{\partial u}{\partial n} = g \quad \text{on } \partial\Omega.$$

(iii) A third option is given when $\alpha \neq 0, \beta \neq 0$. Boundary conditions of this type are called **Robin boundary conditions**

$$\alpha u + \beta \frac{\partial u}{\partial n} = g \quad \text{on } \partial\Omega..$$

(iv) Suppose that $\alpha \neq 0, \beta \neq 0$ and also that the boundary $\partial\Omega$ consists of two disjoint parts Γ_1 and Γ_2, such that

$$\partial\Omega = \Gamma_1 \cup \Gamma_2.$$

Then we may have **Dirichlet boundary conditions** on part of the boundary

$$u = u_0 \quad \text{on } \partial\Gamma_1$$

and **Neumann boundary conditions** on the other part of the boundary

$$\frac{\partial u}{\partial n} = g \quad \text{on } \partial\Gamma_2.$$

For reasons of space, in this thesis we will restrict to the Poisson equation (4.2) with **homogeneous Dirichlet boundary conditions**, namely

$$u = 0 \quad \text{on } \partial\Omega. \tag{4.5}$$

The following theorems and proofs for the Poisson equation (4.2) with Dirichlet boundary conditions (4.4) have been extracted and modified from Chen [3]. The mean value theorem states that the value $u(z)$ of a harmonic function u evaluated at a point $z \in \mathbb{R}^d$ equals its average value over balls $B_\varepsilon(z)$ of radius $\varepsilon > 0$ centered at that point.

Theorem 4.1 (Mean value property)**.** *Let $u \in C^2(\Omega)$ be harmonic. Then for any ball $B_\varepsilon(z) \subset \Omega$ the function u satisfies the* ***mean value property****:*

$$u(z) = \frac{1}{|\partial B_\varepsilon|} \int_{\partial B_\varepsilon(z)} u(y) dS(y) = \frac{1}{|\partial B_\varepsilon|} \int_{B_\varepsilon(z)} u(y) S(y). \tag{4.6}$$

Proof. The proof consists of two parts. In the first part, we will prove the first equality in (4.6) and in the second part we will prove the second equality.
(i) Define an ε-independent domain of integration

$$w(\varepsilon) := \frac{1}{|\partial B_\varepsilon|} \int_{\partial B_\varepsilon(z)} u(y) dS(y) = \frac{1}{|\partial B_1|} \int_{B_1(0)} u(z + \varepsilon r) dS(r)$$

where we used **integration by substitution**

$$\int_{\varphi(U)} f(v) dv = \int_U f(\varphi(u)) |\det \varphi'(u)| du.$$

Differentiating gives

$$w'(\varepsilon) = \frac{1}{|\partial B_1|} \int_{B_1(0)} \nabla u(z + \varepsilon r) \cdot r dS(r).$$

By applying the transformation

$$y = z + \varepsilon r \iff r = \frac{y - z}{\varepsilon}$$

we obtain

$$\frac{1}{|\partial B_1|} \int_{\partial B_1(0)} \nabla u(z + \varepsilon r) \cdot r dS(r) = \frac{1}{|\partial B_\varepsilon|} \int_{\partial B_\varepsilon(z)} \nabla u(y) \cdot \left(\frac{y - z}{\varepsilon} \right) dS(y).$$

Now using the **divergence theorem for balls**

$$\int_{\partial U} F \cdot n dS = \int_U \nabla \cdot F dy$$

we obtain

$$\frac{1}{|\partial B_\varepsilon|} \int_{\partial B_\varepsilon(z)} \nabla u(y) \cdot \left(\frac{y - z}{\varepsilon} \right) dS(y) = \frac{1}{|B_\varepsilon|} \int_{B_\varepsilon(z)} \nabla \cdot \nabla u(y) dy.$$

This in turn implies that the function w must be constant, since

$$\frac{1}{|B_\varepsilon|} \int_{B_\varepsilon(z)} \nabla \cdot \nabla u(y) dy = \frac{1}{|B_\varepsilon|} \int_{B_\varepsilon(z)} \Delta u(y) dy = \frac{1}{|B_\varepsilon|} \int_{B_\varepsilon(z)} 0 \; dy = 0$$

because u is harmonic. Now observe

$$|w(0) - u(z)| = \lim_{\varepsilon \to 0} |w(\varepsilon) - u(z)| = \lim_{\varepsilon \to 0} \frac{1}{|\partial B_\varepsilon|} \left| \int_{\partial B_\varepsilon(z)} u(y) dS(y) - u(z) \right|.$$

Bounding this expression from above two times we can actually prove that $w = u(z)$:

$$\lim_{\varepsilon \to 0} \frac{1}{|\partial B_\varepsilon|} \left| \int_{\partial B_\varepsilon(z)} u(y) dS(y) - u(z) \right| \leqslant \lim_{\varepsilon \to 0} \frac{1}{|\partial B_\varepsilon|} \int_{\partial B_\varepsilon(z)} |u(y) - u(z)| dS(y).$$

and,

$$\lim_{\varepsilon\to 0}\frac{1}{\partial B_\varepsilon|}\int_{\partial B_\varepsilon(z)}|u(y)-u(z)|dS(y)\leqslant\sup_{y\in B_\varepsilon(z)}|u(y)-u(z)|\to 0.$$

(*ii*) Observe

$$\int_{B_\varepsilon(z)}u(y)dy=\int_0^\varepsilon\left(\int_{\partial B_r(z)}udS\right)dr$$

where this is precisely the formula for **integration in spherical polar coordinates**. Now using the 1st equation

$$\int_0^\varepsilon\left(\int_{\partial B_r(z)}udS\right)dr=\int_0^\varepsilon u(z)|\partial B_r|dr.$$

Since $u(z)$ is independent from r, we can take it outside the integral

$$\int_0^\varepsilon u(z)|\partial B_r|dr=u(z)\int_0^\varepsilon|\partial B_r|dr.$$

Now we can apply again the formula for integration in spherical polar coordinates

$$u(z)\int_0^\varepsilon|\partial B_r|dr=u(z)\int_{B_\varepsilon(z)}dy=u(z)|B_\varepsilon|.$$

□

The following theorem shows that the mean value property characterizes harmonic functions up to a certain degree.

Theorem 4.2 (Converse to the mean value property)**.** *Let $u\in C^2(\Omega)$ be such that*

$$u(z)=\frac{1}{|\partial B_\varepsilon|}\int_{\partial B_\varepsilon(z)}u(y)dS(y)$$

for all balls $B_\varepsilon(z)\subset\Omega$. Then u is harmonic in Ω.

Proof. Assume that u is not harmonic, i.e. $\Delta u\neq 0$. Then there exists a ball $B_\varepsilon(z)\subset\Omega$, such that $\Delta u>0$ in $B_\varepsilon(z)$. On the other hand,

$$0=w'(\varepsilon)=\frac{1}{|B_\varepsilon|}\int_{B_\varepsilon(z)}\Delta u(y)\cdot\left(\frac{y-z}{\varepsilon}\right)dy>0$$

which is a contradiction. □

The strong maximum principle states that extrema of a solution of an elliptic equation can be attained in the interior if and only if the function is a constant.

Theorem 4.3 (Strong maximum principle)**.** *Let $u\in C^2(\Omega)\cap C(\overline{\Omega})$ be harmonic in Ω and let Ω be connected. Suppose there exists a $z_0\in\Omega$ such that*

$$u(z_0)=\max_\Omega u(z).$$

Then u is constant within Ω.

Proof. Define

$$M := u(z_0) = \max_{\Omega} u(z).$$

Then for all $0 < \varepsilon < \operatorname{dist}(z_0, \partial\Omega)$ we have by the mean value property, since u is harmonic:

$$M = u(z_0) = \max_{\Omega} u(z) = \frac{1}{|B_\varepsilon|} \int_{B_\varepsilon(z_0)} u(y)dy \leqslant M.$$

hence, u is constant within B_ε. Therefore, the set

$$\Omega_M := \{z \in \Omega : u(z) = M\}$$

is both open and closed in Ω. Since Ω is connected we have $\Omega_M = \Omega$. □

As a corollary, we obtain the uniqueness of solutions of the Dirichlet boundary value problem thereby showing that this problem is **well-posed**.

Corollary 4.4 (Uniqueness)**.** *The Dirichlet boundary value problem* (4.2), (4.4) *has at most one* $C^2(\Omega) \cap C(\overline{\Omega})$ *solution.*

Now we will present the Poisson equation that we want to solve. Let

$$Z := \overline{\Omega} := [0,1]^2$$

$$f(z) := -2\pi^2 \sin \pi z_1 \sin \pi z_2$$

$$g(z) := 0.$$

Explicitly, our Poisson equation with homogeneous Dirichlet boundary conditions reads

$$\Delta u(z) = -2\pi^2 \sin \pi z_1 \sin \pi z_2 \qquad \text{on } z \in Z^\circ \tag{4.7}$$

$$u(z) = 0 \qquad \text{on } z \in \partial Z. \tag{4.8}$$

Solving the Poisson equation consists of finding a sufficiently smooth function u that satisfies (4.7)–(4.8). Thanks to Corollary 4.4, this problem has at most one $C^2((0,1)) \cap C([0,1])$ solution. We claim that this solution is given precisely by the function

$$u^{\text{exact}} : \mathbb{R}^2 \to \mathbb{R},\ u^{\text{exact}}(z) := \sin \pi z_1 \sin \pi z_2.$$

Indeed, u^{exact} is twice continuously differentiable with partial derivatives

$$\frac{\partial u^{\text{exact}}}{\partial z_1}(z) = \pi \cos \pi z_1 \sin \pi z_2,$$

$$\frac{\partial u^{\text{exact}}}{\partial z_2}(z) = \pi \sin \pi z_1 \cos \pi z_2,$$

and

$$\frac{\partial^2 u^{\text{exact}}}{\partial z_1^2}(z) = -\pi^2 \sin \pi z_1 \sin \pi z_2 = \frac{\partial^2 u^{\text{exact}}}{\partial z_2^2}(z).$$

The Laplacian operator of u^{exact} satisfies

$$\Delta u^{\text{exact}}(z) = \sum_{j=1}^{2} \frac{\partial^2 u^{\text{exact}}}{\partial z_j^2}(z) = -2\pi^2 \sin \pi z_1 \sin \pi z_2 = f(z).$$

The boundary conditions are also fulfilled:

$$u^{\text{exact}}(0, z_2) = u^{\text{exact}}(1, z_2) = u^{\text{exact}}(z_1, 0) = u^{\text{exact}}(z_1, 1) = 0.$$

Therefore, u^{exact} is the unique solution to (4.7)–(4.8). In the following figure, we can see a plot of the exact solution u^{exact}.

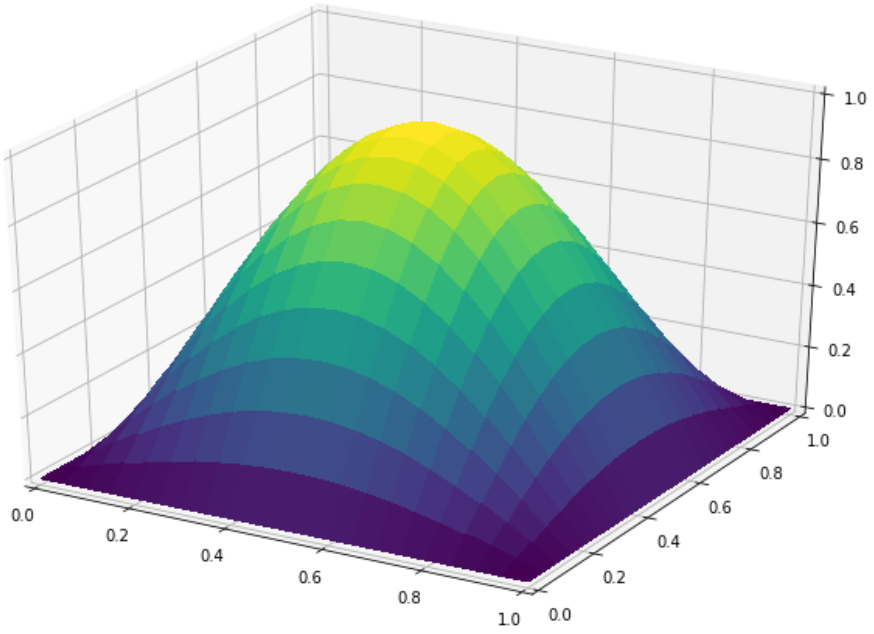


Figure 4.1.: u^{exact}

4.2. A solution method via neural networks

This section is based on Narain [13] and on its programming code available at `https://github.com/narain42/Poission-s-Equation-Solver-Using-ML`.
In his paper, Narain presented in 2021 a method to solve the Poisson equation in $d = 2$ for diverse combinations of boundary conditions including non-homogeneous Dirichlet boundary conditions and mixed homogeneous Dirichlet boundary conditions in some edges and non-homogeneous boundary conditions either Dirichlet or Neumann in the other edges. His method is compared to already existing finite difference methods with excellent results. Narain uses shallow neural networks with zero biases:

$$\Phi(x, z) = c\sigma(Az)$$

as well as the sigmoid as activation function. Further, the function which should solve the Poisson equation is not the neural network Φ but a function

$$\Psi(x, z) = B(z)\Phi(x, z) + C(z).$$

An appropriate choice of the functions $B(z)$ and $C(z)$ makes that the function Ψ satisfies the boundary conditions. Narain defined the function Ψ for this case as

$$\Psi(x, z) := z_1(1 - z_1)z_2(1 - z_2)\Phi(x, z).$$

This has the benefit that the loss function must not be defined with distinction of cases. Narain defined the loss function g as

$$g(x,z) = (\Delta_z \Psi(x,z) - f(z))^2$$

and solved the unconstrained optimization problem

$$\min_{x \in \mathbb{R}^n} \sum_{i=1}^{m} g(x, z^i) \tag{4.9}$$

via gradient descent to find the optimal parameter vector x for the dataset where the derivatives $\nabla_x g(x, z^i)$ were computed via the automatic differentiation library Autograd within Python.

4.3. Our proposed minimax method

We want to solve the Poisson equation (4.7)–(4.8) by a shallow neural network with the hyperbolic tangent as activation function. Hence, we define in a similar manner as Narain,

$$\sigma(r) := \tanh r \tag{4.10}$$

$$\Phi(x,z) := c\sigma(Az + b) + e \quad \text{with } x = (c, A, b, e) \tag{4.11}$$

$$\Psi(x,z) := z_1(1 - z_1)z_2(1 - z_2)\Phi(x,z) \tag{4.12}$$

$$g(x,z) := (\Delta_z \Psi(x,z) - f(z))^2. \tag{4.13}$$

We want that the loss function g is approximately zero on Z, i.e.

$$g(x,z) \approx 0 \quad \text{for all } z \in Z.$$

However, this will not be achieved by solving an unconstrained optimization problem (4.9) but by solving the minimax problem

$$\min_{x \in \mathbb{R}^n} \max_{z \in Z} g(x,z). \tag{4.14}$$

It can be checked that the objective function g is neither convex on x nor concave on z. We are dealing with a non-convex non-concave minimax problem. Based on suggestions with my supervisor, the proposed procedure to solve this non-convex non-concave minimax problem consists of iteratively sampling a finite set of points $S^\ell := \{z^{1,\ell}, \ldots, z^{m,\ell}\}$ from the set Z and restricting the maximization to $z \in S^\ell$. This yields the approximate minimax problem

$$\min_{x \in \mathbb{R}^n} \max_{z \in S^\ell} g(x,z).$$

Motivated by Danskin [5], we will compute

$$z^{\ell+1} = \operatorname*{argmax}_{z \in S^\ell} g(x^\ell, z) \tag{4.15}$$

which can be done through simple enumeration as the following algorithm shows.

Algorithm 5 : FindArgMax

Input: vector x, current set $S = \{z^1, \ldots, z^m\}$
Output: vector z^{max} which maximizes $g(x,z)$ over $z \in S^\ell$
$g^* \leftarrow -\infty$;
for $i = 1 : m$ **do**
  if $g(x, z^i) \geqslant g^*$ **then**
    $g^* \leftarrow g(x, z^i)$; $\quad z^{max} \leftarrow z^i$;
  end if
end for

Then we will compute the gradient

$$\nabla_x g(x^\ell, z^{\ell+1})$$

via automatic differentiation to perform an iteration of gradient descent, Algorithm 2, on x:

$$x^{\ell+1} = x^\ell - \alpha_\ell \nabla_x g(x^\ell, z^{\ell+1}). \tag{4.16}$$

Finally, we obtain the next set $S^{\ell+1}$ from the ResampleS procedure:

$$S^{\ell+1} = \text{ResampleS}(x^{\ell+1}, S^\ell). \tag{4.17}$$

Algorithm 6 : ResampleS

Input: vector x, current set $S = \{z^1, \ldots, z^m\}$, quantile q, number of clusters k
Output: new set S'
1. Sort S from highest value of $g(x, z^i)$ to lowest.
2. Keep the q quantile and delete the rest. We call the remaining set S'.
3. Cluster S' using Algorithm 4, k-Means, to find k centroids.
4. For each point in S' sample more points z and append them to S' so that we have the same points as in the beginning.

The idea of Algorithm 6 is to look for values of z whose function value at the optimizer x^ℓ is in a high quantile. Those values are clustered using k-means to find regions of local maxima with respect to z, since those points are where the approximation fails the most and is where we need more points. The new sample $S^{\ell+1}$ in a cubic neighbourhood around those points. We will illustrate this idea with the following example.

Example 4.5 (Step 3)**.** *Let $x^{\ell+1}$ be fixed and define the residual*

$$r(z) := \nabla_z \Psi(x^{\ell+1}, z) - f(z).$$

It is clear that

$$g(x^{\ell+1}, z) = r^2(z).$$

The function $\nabla_z \Psi(x^{\ell+1}, z)$ can be a good approximation of $f(z)$ for some of the points $z \in Z$ and bad for others. For the sake of illustration, suppose that the residual r has the simple form

$$r(z) = \sin 4\pi z.$$

Figure 4.2 shows the function r as a dashed orange line versus the function r^2 as a solid blue line on the interval $[0, 1]$.

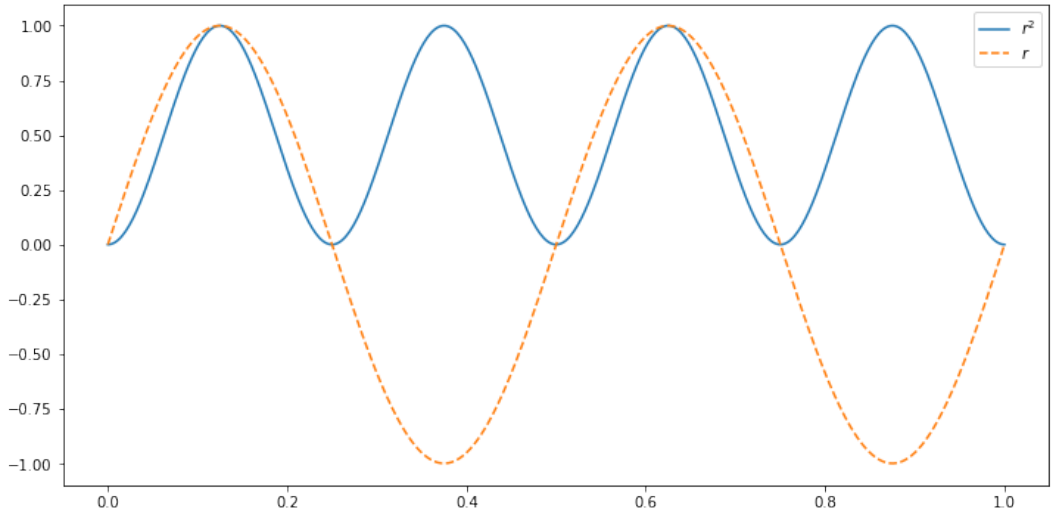


Figure 4.2.: Functions r vs r^2 on [0,1].

As can be seen in Figure 4.2, there are values of z near

$$z = \frac{j}{4}, \quad \text{for } j = 0, \dots, 4$$

where both functions $r(z)$ and $r^2(z)$ are 0. This means that the function $\nabla_z \Psi(x^{\ell+1}, z)$ more or less approximates the function $f(z)$ at those points. However, there are values of z near

$$z = \frac{2j+1}{8}, \quad \text{for } j = 0, \dots, 3$$

where regression fails the most. These points correspond to extreme points of r or maximal points of r^2. Let the set S consist of the points $z^1, \dots, z^m$. In the following figure, the function r^2 is depicted again as a solid blue line together with the evaluations $r^2(z^i)$, for $i = 1, \dots, m$ marked as crosses.

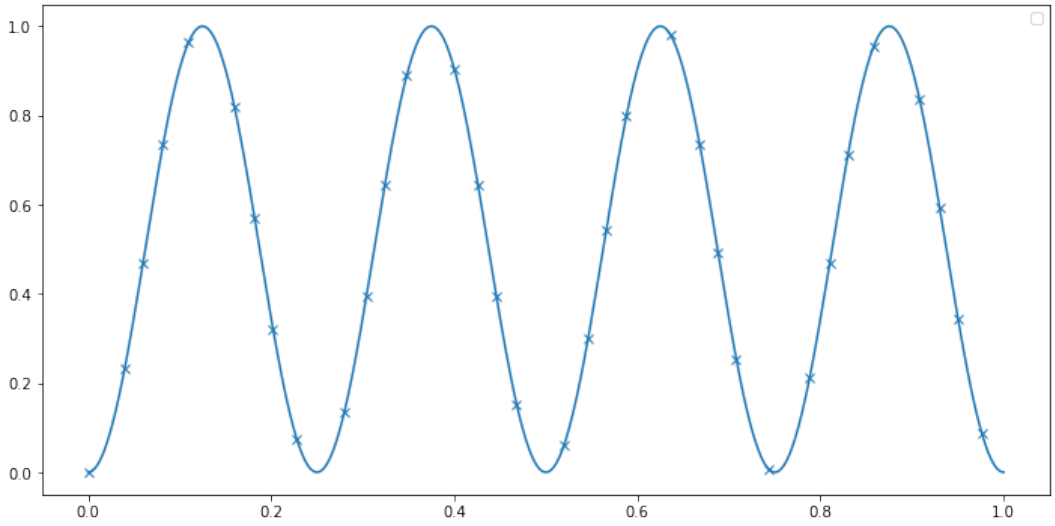


Figure 4.3.: Function r^2 and evaluations $r^2(z^i)$, for $i = 1, \dots, m$ on [0,1].

The aim now is to find the points where the function r^2 is maximal. Hence, the pairs $(z^i, r^2(z^i)), i = 0, \dots, m$ can be sorted in descending order using any sorting algorithm, e.g.

merge sort. *Merge sort is a divide-and-conquer sorting algorithm that has a worst-case performance of $\mathcal{O}(m \log m)$ iterations. A threshold has to be defined to keep all values above the threshold. This value is a* ***hyperparameter****. In this example the 85% quantile was used. The next figure shows a dashed orange line that represents the 85% quantile.*

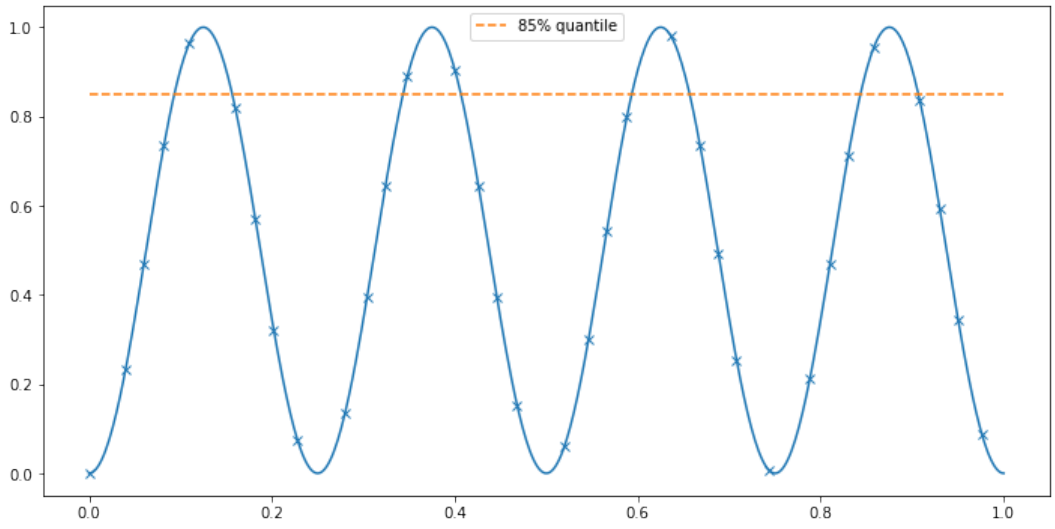


Figure 4.4.: Function r^2 and 85% quantile on [0,1].

All values z^i with an evaluation $r^2(z^i)$ below the threshold will be discarded. All remaining values will be clustered aiming to find regions of truly maxima of r^2. The correct choice for the number of clusters (in this case, 4) is also a ***hyperparameter****. The following figure shows four clusters highlighted with circles (which look like ellipses, since the axes are stretched) around the 5 best points. The centroids of the clusters are also determined.*

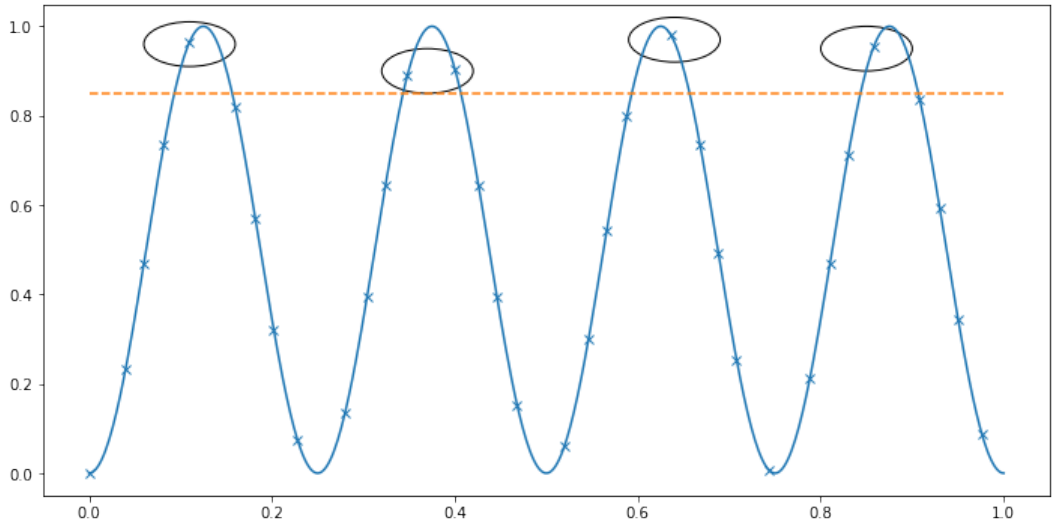


Figure 4.5.: Best points are clustered.

For each centroid, more points will be sampled randomly from a neighbourhood of it, e.g., according to the uniform distribution in a cube (in this case an interval) of size smaller than the radius of the cluster. This will yield the next set S.

The complete procedure is gathered in the following algorithm.

Algorithm 7 : MAIN

Input: learning rate α, number of neurons N, number of iterations τ
Output: optimal weights x^* s.t. the function $\Psi(x^*, z)$ solves the PDE (4.7)– (4.8)
Choose $x^0 \sim \mathcal{N}(0, I), S^0 \sim \mathcal{U}_{[0,1]^d}$ random
for $\ell = 1 : \tau$ **do**
  Use Algorithm 5 to obtain $z^{\ell+1}$.
  Compute $\nabla_x g(x^\ell, z^{\ell+1})$ via automatic differentiation.
  Perform one iteration of gradient descent $x^{\ell+1} \leftarrow x^\ell - \alpha \nabla_x g(x^\ell, z^{\ell+1})$.
  Use Algorithm 6 to obtain $S^{\ell+1}$.
end for

4.4. Results

In this section, we solved the Poisson equation (4.7)– (4.8) using the Narain approach from Section 4.2 and the minimax approach from 4.3. We adapted the Narain approach to our problem without modifying the core ideas. For both approaches, we used the functions defined in (4.10)–(4.13). Since $Z = [0, 1]^2$, we tested three different stepsizes $\alpha \in \{\frac{1}{1.000}, \frac{1}{10.000}\frac{1}{100.000}\}$. Moreover, we used for both approaches a number of neurons $N = 10$, which resulted in $n = 41$ parameters. As a testset, we used the rectangular grid resulting from dividing each axis into 21 equally distanced points. This gave us 441 many test points $(z^{test,i}, y^{test,i})_{i=1}^{441}$.

The Narain method with stepsize $\alpha = \frac{1}{1.000}$ did not converge. In the following figure, we see that the plots of the loss function g with different values of α as the iterations increase. The solid line represents a value of $\alpha = \frac{1}{10.000}$ and the dashed line represents a value of $\alpha = \frac{1}{100.000}$. The loss function for $\alpha = \frac{1}{1.000}$ is much higher and is therefore not displayed in the figure.

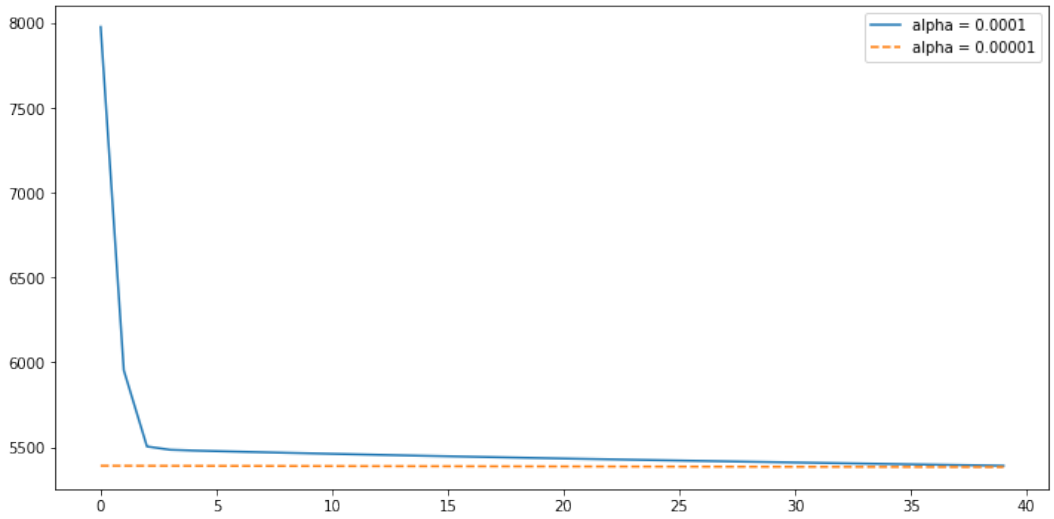
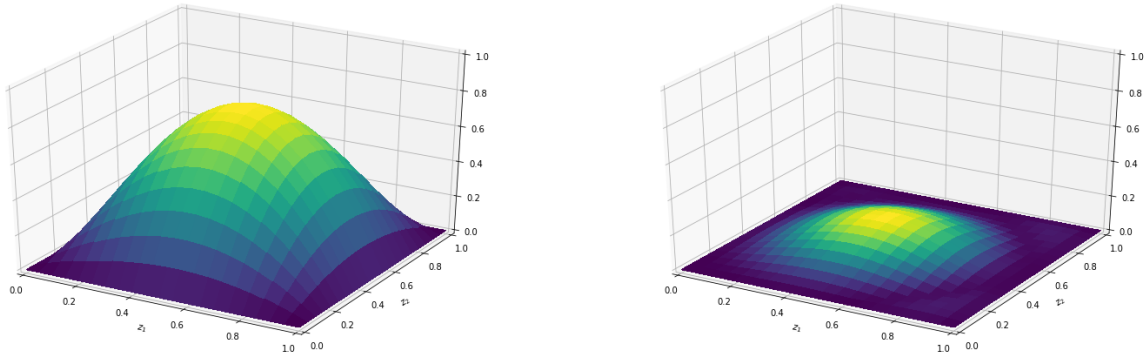


Figure 4.6.: Loss function for different α vs iterations.

Observe that after the second iteration, the progress is minimal. However, both step sizes returned a maximal error of 18.49% which is surprisingly high, since Narain [13] used the same training set as the testset. In Figure 4.7, we can see on the left the solution provided by the Narain method after 40 iterations and on the right the surface of the error between reconstructed and exact solution.



(a) Reconstructed function (b) Error surface

Figure 4.7.: Narain method

The Narain method required approximately 40 iterations to converge. Each iteration ran for approximately 3 seconds and at each iteration, the gradient over the whole dataset was computed.

For the minimax method, the loss function also decreased each iteration. Since this method only computed the gradient at one point in each iteration, it needed more iterations. To produce good results, the algorithm needed 300 iterations and run for about 7 min. This means that each iteration needed about 1 min 20 s, which is surprising fast, since it performed k-means and resampled within that time. In Figure 4.8, we can see a plot of the loss functions for different values of α. The solid line represents a value of $\alpha = \frac{1}{10.000}$, the dashed line represents a value of $\alpha = \frac{1}{100.000}$ and the dotted line represents a value of $\alpha = \frac{1}{1.000}$.

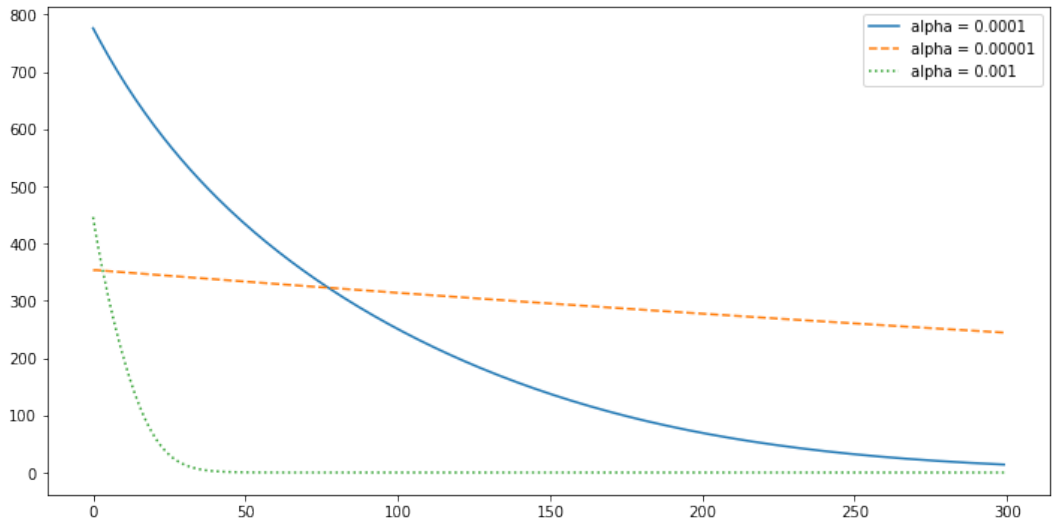
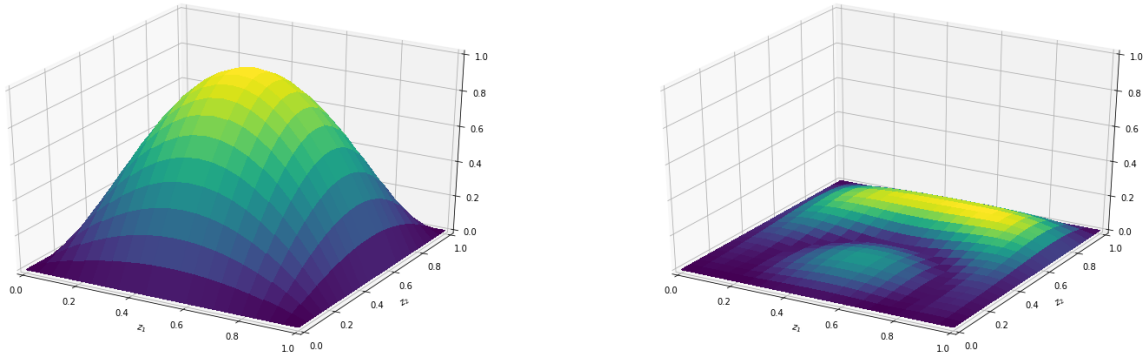


Figure 4.8.: Loss function for different α vs iterations

The step size $\alpha = \frac{1}{10.000}$ was again the best, producing a maximal error of 10.74%. In the next figure, we see the reconstructed function and the error surface for the minimax method.



(a) Reconstructed function

(b) Error surface

Figure 4.9.: Minimax method

The real number of classes k is unknown. There are many "rules of thumb" for selecting the most appropriate number of classes. They are based on **hierarchical clustering** i.e. iteratively perform clustering on a set with an increasing number of classes k. Then using one or different plotting methods (by averaging them or by using majority rule) obtain the optimal k. Those methods include the **elbow method** and the **silhouette method**. For simplicity we used a quantile of 90% and clustered according to a number of clusters $k = 20$ which is roughly half the number of parameters.

5. Conclusion and future research

The initial goal of this thesis was to investigate if a minimax approach could be used to find the solution of a PDE via neural networks. We believe that this is a more than valid approach to tackle such problems. So far, the results have been surprisingly good. Our method achieved around around 90% accuracy on the test set without hyperparameter tuning. The shallow neural network was thus able to generalize well. While we only tested the method on one of the simplest PDEs possible, we believe that this approach can be extrapolated to other types of PDEs, even in higher dimensions, with similar results. Since regression problems are easier than PDEs because they do not involve higher derivatives, such kind of problems could also be solved with this minimax approach.

There are many directions this approach could evolve and refine. In the future we can try to solve the Poisson equation for $d = 3$ and also slightly more complicated PDEs like the heat equation with this method. We did not try different boundary conditions so this is something we could do in the future. There are many hyperparameters that can be investigated. Regarding the neural network, the most obvious hyperparameter is the number of neurons. For more flexibility, we could also try deep neural networks or even other types of neural networks such as convolutional neural networks to try to solve the Poisson equation in high dimensions. For the clustering process, we could try other clustering algorithms such as linkage-based algorithms or hierarchical clustering. New points could be sampled from around a unit cube instead of a unit ball whose side is also a hyperparameter. The number of clusters and the choice of the quantile were hyperparameters that were not exhaustively investigated. Finally, the last hyperparameter is the learning rate of the gradient descent algorithm. The learning rate could change each iteration instead of being constant. Furthermore, instead of the gradient descent, we could use better algorithms such as Newton methods or ADAM.

A. Implementation

This chapter contains the code used in Sections 4.2 and 4.3 in the programming language Python. First, we import all required packages and prepare the plot functions.

```
from matplotlib import pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from time import process_time
import autograd.numpy.random as npr
import autograd.numpy as np
from  autograd import grad, jacobian
from sklearn.cluster import KMeans
from sklearn import metrics
import warnings
warnings.filterwarnings("ignore")

# Setting up the rectangular grid
nz1 = 21
nz2 = 21
z1, z2 = np.linspace(0, 1, nz1), np.linspace(0, 1, nz2)
dz1, dz2 = z1[1]-z1[0], z2[1]-z2[0]
Z1, Z2 = np.meshgrid(z1, z2, indexing='ij')
z1_space = np.linspace(0, 1, nz1)
z2_space = np.linspace(0, 1, nz2)
np.random.seed(99)

def plot2D(z1,z2,label2D):
  plt.figure(figsize = (12,6))
  plt.plot(z1, z2, label=label2D)
  plt.legend()
  plt.show()

def plot3D(Z,label3D):
  print(label3D, ":")
  fig = plt.figure(figsize=(12,8))
  ax = fig.gca(projection='3d')
  Z1, Z2 = np.meshgrid(z1_space, z2_space)
  surf = ax.plot_surface(Z1, Z2, Z, rstride=1, cstride=1, cmap=cm.viridis,
                         linewidth=0, antialiased=False)
  ax.set_xlim3d(0, 1)
  ax.set_ylim3d(0, 1)
  ax.set_zlim3d(0, 1)
  ax.set_xlabel('$z_1$')
  ax.set_ylabel('$z_2$')
  plt.show()
```

Then we define functions common to both methods.

```
def u_exact(z1,z2):
  return np.sin(np.pi * z1) * np.sin(np.pi * z2)

def f(z):
  return -2. * np.pi **2 * np.sin(np.pi * z[0]) * np.sin(np.pi * z[1])

def tanh(r):
  return np.tanh(r)

def phi(x, z):
  q = tanh(np.dot(z,x[1])) + x[2]
  return np.dot(q,x[0]) + x[3]

def psy(z, net_out):
  return  z[0] * (1 - z[0]) * z[1] * (1 - z[1]) * net_out

def surfaces(x,method):
  surface_method = np.zeros((nz1, nz2))
  surface_err = np.zeros((nz1, nz2))
  max_test_err = -1
  for i, z1 in enumerate(z1_space):
    for j, z2 in enumerate(z2_space):
      net_out = phi(x,[z1, z2])[0]
      surface_method[j][i] = psy([z1, z2], net_out)
      surface_err[j][i] = abs(u_exact(z1,z2) - psy([z1, z2], net_out))
      if surface_err[j][i] > max_test_err:
        max_test_err = surface_err[j][i]
  print("The maximum error for the ",method," method is ", max_test_err)
  return surface_method, surface_err
```

Now we define the functions needed for Narain's method.

```
def loss_func(x, z1, z2):
  loss_sum = 0
  for z2_i in z2:
    for z1_i in z1:
      z = np.array([z1_i, z2_i])
      net_out = phi(x, z)[0]
      psy_hess = jacobian(jacobian(psy))(z, net_out)
      err_sqr = (np.trace(psy_hess) - f(z))**2
      loss_sum += err_sqr
  return loss_sum

def method_Narain(alpha,n_neur,n_iter):
  tic = process_time()
  loss_Narain = np.empty(n_iter)
  iterations = list(range(n_iter))
  for i in iterations:
    loss_grad =  grad(loss_func)(x, z1_space, z2_space)
```

```
    x[0] = x[0] - alpha * loss_grad[0]
    x[1] = x[1] - alpha * loss_grad[1]
    x[2] = x[2] - alpha * loss_grad[2]
    x[3] = x[3] - alpha * loss_grad[3]
    loss_Narain[i] = loss_func(x, z1_space, z2_space)
  toc = process_time()
  el_time = toc - tic
  print("The elapsed time for the Narain method is ",el_time)
  return x,loss_Narain
```

We define the functions for the minimax method.

```
def g(x,z):
  net_out = phi(x,z)[0]
  psy_hess = jacobian(jacobian(psy))(z, net_out)
  return (np.trace(psy_hess) - f(z))**2

def find_argmax(x,S):
  (m,d) = np.shape(S)
  tmp = 0
  z_max = np.empty(d)
  for i in range(m):
    z = S[i]
    g_value = g(x,z)
    if g_value >= tmp:
      tmp = g_value
      z_max = z
  return z_max

def resample_S(x,S):
  (m,d) = S.shape
  # a.) Ordering of the rows of S.
  #     The rows of S are ordered from highest value of g(x,S[i]) to lowest. The ordered matrix
  #     S is saved as S_1.
  tmp = np.empty([m])
  m_div_10 = int(m/10)
  for i in range(m):
    tmp[i] = g(x,S[i])
  tmp2 = np.argsort(tmp)
  S_1 = S[tmp2]
  # b.) Keep the "quant"0% percentile and delete the rest. A value quant = 9 means that we want
  #     to keep the 90% percentile. Since we will delete the lower 90% percentile, later we will
  #     need to create more  points. The value kehrwert says the multiple of the points that we
  #     need. Example: if we deleted 90% of the data we need again 9 times as many new points.
  #     Possible combinations are: quant, kehrwert = (5,1) , (8,4) , (9,9). The pairs satisfy:
  #     (10 - quant)(kehrwert + 1) = 10.
  quant = 9
  kehrwert = 9
  n_clusters = 20
  S_2 = S_1[quant*m_div_10:m]
  tmp3 = tmp[tmp2]
```

```
  g_values = tmp3[quant*m_div_10:m]
  num_points_left = len(g_values)
  # c.) Cluster the vectors into n_clusters.
  #     Centroids is a matrix whose rows are the centroids of kmeans. The vector max_radius
  #     stores the largest distance from all points within a cluster and the centroid, measured
  #     in the Euclidean norm. The distance matrix is an m x n matrix where m = num_points_left
  #     and n = n_clusters.
  kmeans = KMeans(n_clusters).fit(S_2)
  centr = kmeans.cluster_centers_
  lab = kmeans.labels_
  max_rad = np.zeros(n_clusters)
  distance_matrix = kmeans.transform(S_2)
  for i in range(num_points_left):
    if distance_matrix[i,lab[i]] > max_rad[lab[i]]:
      max_rad[lab[i]] = distance_matrix[i,lab[i]]
  # d.) Resample
  #     Using the uniform distribution, sample "kehrwert" more points for each point within the
  #     centroids with radius equal to the maximal radius computed before.
  new_vec = np.empty((1,d))
  for i in range(num_points_left):
    for j in range(kehrwert):
      for k in range(d):
        new_vec[0][k] = npr.uniform(centr[lab[i]][k] -max_rad[lab[k]],\
                                    centr[lab[i]][k] +max_rad[lab[k]])
        if new_vec[0][k] > 1:
          new_vec[0][k] = 1
        elif new_vec[0][k] < 0:
          new_vec[0][k] = 0
      S_2 = np.concatenate((S_2, new_vec), axis=0)
  return S_2

def method_minimax(alpha,n_neur,n_iter):
  tic = process_time()
  # a.) Initialize x and S
  x = [npr.randn(n_neur), npr.randn(2, n_neur), npr.randn(n_neur), npr.randn(1)]
  S = npr.uniform(low=0.0, high=1.0, size=(400,2))
  loss_minimax = np.empty(n_iter)
  iterations = list(range(n_iter))
  for i in iterations:
    # b.) Look for the vector z in S which maximizes g(x,z)
    z = find_argmax(x,S)
    # c.) Find derivatives via automatic differentiation
    loss_grad =  grad(g)(x,z)
    # d.) Gradient descent update
    x[0] = x[0] - alpha * loss_grad[0]
    x[1] = x[1] - alpha * loss_grad[1]
    x[2] = x[2] - alpha * loss_grad[2]
    x[3] = x[3] - alpha * loss_grad[3]
    loss_minimax[i] = g(x,z)
    # e.) Resample S
    S = resample_S(x,S)
```

```
  toc = process_time()
  el_time = toc - tic
  print("The elapsed time for the Minimax method is ",el_time)
  return x,loss_minimax
```

Both approaches can be tested using the following code.

```
n_neur = 10 #dim x = n_neur(d+2) + 1 = 41
x = [npr.randn(n_neur), npr.randn(2, n_neur), npr.randn(n_neur), npr.randn(1)]
    # a.) x = (c,A,b,e)
    # b.) dim x = n_neur(d+2) + 1 = 41
    # c.) For better compatibility with Python, some components of the neural
    #     network are transposed. In particular: c is a row vector, A is a (d x m) matrix, b is
    #     a row vector, e is a scalar, z is a row vector
alpha1 = 0.0001
alpha2 = 0.00001
alpha3 = 0.001  #this produces very bad results for Narain

n_iter_Nar = 40    # This method computes the gradient at all points and needs less iterations to
                   # converge.
iter_Nar = list(range(n_iter_Nar))
n_iter_min = 300   # This method computes the gradient only at one point hence needs more
                   # iterations to converge.
iter_min = list(range(n_iter_min))

print("------             Narain method            ------")

x_Nar1,loss_Nar1 = method_Narain(alpha1,n_neur,n_iter_Nar)
x_Nar2,loss_Nar2 = method_Narain(alpha2,n_neur,n_iter_Nar)

plt.figure(figsize = (12,6))
plt.plot(iter_Nar,loss_Nar1,label="alpha = 0.0001")
plt.plot(iter_Nar,loss_Nar2,label="alpha = 0.00001",linestyle='--')
plt.legend()
plt.show()

surf_N1, surf_errN1 = surfaces(x_Nar1,"Narain1")
plot3D(surf_N1,"Narain method1")
plot3D(surf_errN1,"Error surface for the Narain method1")
surf_N2, surf_errN2 = surfaces(x_Nar2,"Narain2")
plot3D(surf_N2,"Narain method2")
plot3D(surf_errN2,"Error surface for the Narain method2")


print("------             Minimax method            ------")

x_min1,loss_min1 = method_minimax(alpha1,n_neur,n_iter_min)
x_min2,loss_min2 = method_minimax(alpha2,n_neur,n_iter_min)
x_min3,loss_min3 = method_minimax(alpha3,n_neur,n_iter_min)

plt.figure(figsize = (12,6))
```

```
plt.plot(iter_min, loss_min1, label="alpha = 0.0001")
plt.plot(iter_min, loss_min2, label="alpha = 0.00001",linestyle='--')
plt.plot(iter_min, loss_min3, label="alpha = 0.001",linestyle=':')
plt.legend()
plt.show()

surface_method_M1, surface_errM1 = surfaces(x_min1,"Minimax1")
plot3D(surface_method_M1,"Minimax method1")
plot3D(surface_errM1,"Error surface for the Minimax method1")
surface_method_M2, surface_errM2 = surfaces(x_min2,"Minimax2")
plot3D(surface_method_M2,"Minimax method2")
plot3D(surface_errM2,"Error surface for the Minimax method2")
surface_method_M3, surface_errM3 = surfaces(x_min3,"Minimax3")
plot3D(surface_method_M3,"Minimax method3")
plot3D(surface_errM3,"Error surface for the Minimax method3")
```

Bibliography

[1] D. Adelhütte, D. Aßmann, T. González Grandón, M. Gugat, H. Heitsch, R. Henrion, F. Liers, S. Nitsche, R. Schultz, M. Stingl, et al. Joint model of probabilistic-robust (probust) constraints applied to gas network optimization. *Vietnam Journal of Mathematics*, 49(4):1097–1130, 2021.

[2] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust optimization*, volume 28. Princeton university press, 2009.

[3] L. Chen. *Lecture notes for Introduction to PDEs*. University of Mannheim, Winter 2019/2020.

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

[5] J. Danskin. The theory of max-min, with applications. *SIAM Journal on Applied Mathematics*, 14(4):641–664, 1966.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[7] P. Grohs, F. Hornung, A. Jentzen, and P. von Wurstemberger. A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black-scholes partial differential equations. *arXiv preprint arXiv:1809.02362*, 2018.

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

[9] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[10] E. Marchi. On the minimax theorem of the theory of game. *Annali di Matematica Pura ed Applicata*, 77(1):207–282, 1967.

[11] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[12] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT Press, Cambridge, 2018.

[13] J. P. Narain. Solution of poisson's equation using artificial neural networks. *Computer Science and Engineering*, 11(1):17–21, 2021.

[14] A. Nedić and A. Ozdaglar. Subgradient methods for saddle-point problems. *Journal of optimization theory and applications*, 142(1):205–228, 2009.

[15] A. Nemirovski. Prox-method with rate of convergence o (1/t) for variational inequalities with lipschitz continuous monotone operators and smooth convex-concave saddle point problems. *SIAM Journal on Optimization*, 15(1):229–251, 2004.

[16] A. Neumaier. *Lecture notes in Nonlinear Optimization.* University of Vienna, Winter 2020/2021.

[17] A. Neumaier. *Lecture notes in Advanced Numerical Analysis.* University of Vienna, Summer 2021.

[18] A. Neumaier. *Lecture notes in Artificial Intelligence.* University of Vienna, Summer 2022.

[19] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[20] J. Pätzold and A. Schöbel. Approximate cutting plane approaches for exact solutions to robust optimization problems. *European Journal of Operational Research*, 284(1):20–30, 2020.

[21] P. Petersen. *Lecture notes in Mathematics of Machine Learning Neural Network Theory.* University of Vienna, Summer 2021.

[22] P. Petersen. *Lecture notes in Neural Network Theory.* University of Vienna, Winter 2021/2022.

[23] M. Razaviyayn, T. Huang, S. Lu, M. Nouiehed, M. Sanjabi, and M. Hong. Nonconvex min-max optimization: Applications, challenges, and recent theoretical advances. *IEEE Signal Processing Magazine*, 37(5):55–66, 2020.

[24] W. Rudin. *Functional analysis.* McGraw-Hill, New York, 1973.

[25] S. Russel and P. Norvig. *Artificial intelligence: a modern approach.* Prentice Hall, Englewood Cliffs, 2003.

[26] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge University Press, Cambridge, 2014.

[27] M. Shiffman. On the equality min max= max min, and the theory of games. *RAND Report RM-243*, 1949.

[28] M. Sion. On general minimax theorems. *Pacific Journal of mathematics*, 8(1):171–176, 1958.

[29] M. Ulbrich and S. Ulbrich. *Nichtlineare Optimierung.* Springer-Verlag, 2012.

Index