TU Bergakademie Freiberg · Institut für angewandte Analysis

Thesis for obtaining
the academic degree Dipl.-Math.

Ridge Functions and their Application to Machine Learning

Maximilian Schwabe

December 3, 2024

Supervisors:

Prof. Dr. Swanhild Bernstein
Dr. Dmitrii Legatiuk

Table of Contents

Thesis affidavit - Eidesstattliche Erklärung
Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Freiberg, der 03.12.2024 Unterschrift des Verfassers

1. Introduction

In today's fast-paced, technology-driven world, artificial intelligence (AI) plays a pivotal role across a variety of fields. From business ([5], [46]) and medicine ([3], [7], [38]) to transportation ([6]) and personal assistance, AI applications are transforming industries and daily life (see [18], [41]). Tools like ChatGPT from OpenAI, Gemini from Google, and Copilot from Microsoft are prime examples of how AI is seamlessly integrated into both personal and professional environments. These chatbots and assistants enhance productivity, streamline workflows, and facilitate more natural human-machine interactions, marking a significant evolution in how we live and work.

One of the fundamental technologies underlying these advancements is the artificial neural network (ANN). The concept of neural networks can be traced back to 1943, when Warren McCulloch and Walter Pitts introduced a theoretical model of the brain, capturing the essence of how neurons receive, process, and transmit information [29]. Their work laid the foundation for what would later become a central element of modern AI.

A major leap forward came in the late 1950s. Between 1957 and 1958, Frank Rosenblatt and Charles Wightman developed the Mark I Perceptron, the first working neural network computer capable of learning to recognize basic visual patterns, such as simple handwritten digits. Although the Mark I Perceptron was limited in its functionality, it was a key milestone in the development of machine learning and demonstrated the potential of neural networks to handle pattern recognition tasks.

Since the inception of the Mark I Perceptron, neural networks have evolved substantially, now powering some of the most sophisticated AI systems in use today. Early models were limited to simple pattern classification, but modern networks can perform complex operations, including image recognition, object detection, speech processing, and language understanding.

For example, in medical imaging, neural networks assist in tumor detection [37], helping doctors make faster and more accurate diagnoses [14]. In audio and video processing, these systems are used for speech recognition [45], real-time noise suppression [43], and image enhancement [39]. Neural networks have also become fundamental in self-driving vehicles, where they enable autonomous systems to interpret and respond to real-world environments [50]. Additionally, chatbots, voice assistants, and text generators, such as the aforementioned ChatGPT, are becoming indispensable, transforming customer service, writing assistance, and even creative fields.

Despite the increasing focus on high-profile AI applications, such as large language models and generative image models, the focus of this thesis is on the mathematical foundations that make these technologies possible. Specifically, this work aims to provide a detailed mathematical exploration of artificial neural networks, emphasizing their role as universal function approximators. We will analyze the underlying structure of neural networks, explaining how they approximate mathematical functions to model various relationships between inputs and outputs.

A key component of this analysis involves ridge functions, mathematical functions used to describe the way neural networks process information. However, these functions are not only used in neural networks, but also in areas such as statistics (e.g. [8], [15]) or computer tomography (see [19], [20], [26], [30]). But we only want to take a look at neural network applications and understanding these functions is essential for gaining deeper insights into how networks map input data to corresponding outputs, which forms the basis for many AI applications. This

thesis will explore the capabilities and limitations of neural networks in function approximation and examine the mathematical principles that govern their behavior.
A significant part of this thesis will focus on a specific type of ridge function known as the ridgelet. Ridgelets combine features of ridge functions with wavelet-like properties, making them particularly useful for image processing tasks. They are especially effective when integrated into convolutional layers of neural networks, where they enhance the network's ability to detect fine-grained patterns and structures within data.[49]
The use of ridgelet-based filters in image classification networks offers several advantages. Unlike conventional filters, ridgelets can better capture local and global patterns simultaneously, which improves the accuracy and efficiency of image recognition. This makes them particularly valuable in applications where detecting complex or subtle patterns — such as in medical imaging or satellite image analysis — is critical.
In addition to ridgelets, this thesis will explore quaternion convolution, a mathematical operation that extends complex numbers to four dimensions. Quaternions are highly effective for processing multi-channel data, such as color images, where each channel corresponds to a different color component. Compared to traditional convolutions, quaternion convolution can capture correlations across multiple channels more efficiently, making it a powerful tool for image classification.
A major contribution of this thesis will be the design and analysis of a novel network architecture that integrates two parallel branches:

1) The first branch will utilize ridgelet-based filters in its convolutional layers, leveraging their strong approximation properties.

2) The second branch will employ quaternion convolution, enabling the network to process multi-channel data more effectively.

[31] This dual-branch design offers a unique combination of ridgelet-based feature extraction and multi-channel processing through quaternions, which, to our knowledge, has not yet been explored in existing research. By combining these two advanced mathematical approaches, the proposed network aims to improve performance and efficiency in applications that require precise feature extraction and multi-dimensional data handling.
The goal of this thesis is to present and analyze a neural network architecture that combines ridgelet filters and quaternion convolution, a novel approach that bridges two mathematical techniques with distinct strengths. This hybrid architecture aims to improve the accuracy, efficiency, and flexibility of neural networks, particularly in tasks involving image classification and multi-channel data processing.
This work will not only contribute to the theoretical understanding of neural networks but also provide insights into the practical implications of using ridgelets and quaternions in network design. Furthermore, it seeks to explore new possibilities for AI research, offering a fresh perspective on how mathematical concepts can drive innovation in neural network architectures.
By the end of this thesis, we aim to:

- Demonstrate the approximation capabilities of ridge function based neural networks.
- Evaluate the effectiveness of ridgelet filtering in convolutional neural networks
- Develop and analyze a dual-branch network that leverages ridgelet and quaternion approaches to enhance performance.

This thesis thus combines theoretical exploration with practical innovation, offering a new perspective on neural network design and paving the way for future research in this area.

2. Ridge Functions

In the first chapter, we explore the fundamentals of ridge functions, beginning with their definition and key properties. We then examine specialized function spaces formed by sums of ridge functions, distinguishing them based on the direction vectors of the individual ridge functions and the univariate functions that define them. This leads us to consider the density of these spaces within well-established function spaces like $C(\mathbb{R})$, paving the way for deriving important application properties related to function approximation in neural networks, which we will delve into in later chapters. So in this chapter, we want to lay the foundation for the rest of the work The main results come from "Notes on Ridge Functions and Neural Networks" by Vugar Ismailov [16]. The interested reader can delve even deeper into the subject of ridge functions there; we will limit ourselves here to the most important results. Now we come to the ridge functions themselves.
Ridge functions are a special class of multivariate functions that can be represented using a direction vector a and a univariate function g.

Definition 2.0.1. *[16] A function $f : \mathbb{R}^d \to \mathbb{R}$ is called a ridge function if it can be represented as the composition of a univariate function $g : \mathbb{R} \to \mathbb{R}$ and an affine transformation with the direction vector $a \in \mathbb{R}^d$. That is, $f(x) = g(a \cdot x)$.*

It should be mentioned, that also functions of the form $f(x) = g(x \cdot a + b)$ with x, a, $b \in \mathbb{R}^d$ are considered as ridge functions. They also got the ridge profile, but are shifted by a vector b.
One of the prominent properties of ridge functions is summarized in the following lemma:

Lemma 2.0.1. *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a ridge function, $g : \mathbb{R} \to \mathbb{R}$, and $a \in \mathbb{R}^d$ a direction vector such that $f(x) = g(a \cdot x)$.*
Then the function value of f is constant over every hyperplane H with $a \cdot x = 0$ for $x \in H$.

Proof. Let $\boldsymbol{a} \in \mathbb{R}^d$ and $a_1, a_2, ..., a_{d-1} \in \mathbb{R}^d$ be linearly independent vectors orthogonal to $\boldsymbol{a}$. Then it holds that

$$f\left(x + \sum_{k=1}^{d-1} c_k a_k\right) = g\left(x \cdot \boldsymbol{a} + \sum_{k=1}^{d-1} c_k a_k \cdot \boldsymbol{a}\right) = g\left(x \cdot \boldsymbol{a} + \sum_{k=1}^{d-1} c_k 0\right) = g(x \cdot a) = f(x)$$

□

In summary, ridge functions are defined only by their profile given by g and the direction vector a, as shown in figure 2.1.

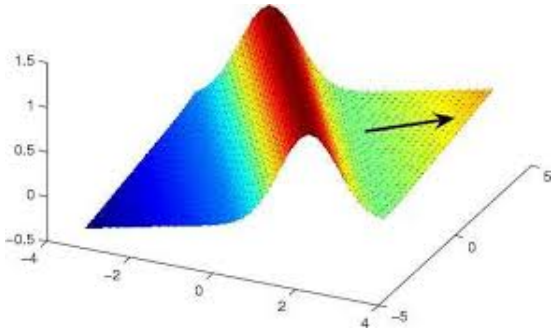


Figure 2.1.: Example of a ridge function with its direction vector
[44]

We will see that ridge functions have very good properties for function approximation. These result from density properties of function spaces constructed from them, which we will consider in the next chapters.

2.1. Ridge Function Spaces

We will now define some spaces for ridge functions, which we will investigate further.

Definition 2.1.1 (Ridge Function Spaces)**.** *[16] Let $a^1, ..., a^r \in \mathbb{R}^d$ be any but fixed direction vectors and $g_1, ..., g_r$ be real univariate functions. Then*

$$\mathcal{R}(a^1, ..., a^r) := \left\{ \sum_{i=1}^{r} g_i(a^i \cdot x) |\, g_i : \mathbb{R} \to \mathbb{R}, i = 1, ..., r \right\}$$

or

$$\mathcal{R}(a^1, ..., a^r; X) := \left\{ \sum_{i=1}^{r} g_i(a^i \cdot x) | x \in X \subset \mathbb{R}^d \, g_i : \mathbb{R} \to \mathbb{R}, i = 1, ..., r \right\}$$

is the space of linear combinations of ridge functions with fixed direction vectors, and

$$\mathcal{R}_r := \left\{ \sum_{i=1}^{r} g_i(a^i \cdot x) : a^i \in \mathbb{R}^d \backslash \{0\} : g_i : \mathbb{R} \to \mathbb{R}, i = 1, ..., r \right\}$$

is the corresponding space with variable direction vectors.

For the space $\mathcal{R}(a^1, ..., a^r)$, it should be noted that the sum can be reduced if two vectors from the set $\{a^1, ..., a^r\}$ are collinear, i.e., $a^r = \lambda a^{r-1}$, like it is shown in [22]. With the function representation $\tilde{g}_{r-1}(t) = g_{r-1}(t) + g_r(\lambda t)$, it holds that

$$\begin{aligned} \sum_{i=1}^{r} g_i(a^i \cdot x) &= \sum_{i=1}^{r-2} g_i(a^i \cdot x) + g_{r-1}(a^{r-1} \cdot x) + g_r(a^r \cdot x) \\ &= \sum_{i=1}^{r-2} g_i(a^i \cdot x) + g_{r-1}(a^{r-1} \cdot x) + g_r(\lambda a^{r-1} \cdot x) = \sum_{i=1}^{r-2} g_i(a^i \cdot x) + \tilde{g}_{r-1}(a^{r-1} \cdot x) \end{aligned}$$

Therefore, in the following, the vectors $\{a^1, ..., a^r\}$ are not collinear, exceptions will be mentioned.

The mentioned spaces can be generalized if the dot products $a^i \cdot x$ are replaced by functions $h_i : \mathbb{R}^d \to \mathbb{R}$ or $h_i : X \to \mathbb{R}$, $X \subset \mathbb{R}^d$. With these functions, we obtain

$$\mathcal{B}(X) := \mathcal{B}(h_1, ..., h_r; X) = \left\{ \sum_{i=1}^{r} g_i(h_i(x)) | x \in X \subset \mathbb{R}^d \, g_i : \mathbb{R} \to \mathbb{R}, i = 1, ..., r \right\}$$

For $h_i(x) = a^i \cdot x$, it holds that $\mathcal{B}(X) = \mathcal{R}(a^1, ..., a^r; X)$.
The last class of ridge functions we want to define is the so-called class of ridge functions with a fixed profile. It is defined as

$$\mathcal{R}(\sigma) := \mathcal{R}_r(\sigma) = \left\{ \sum_{i=1}^{r} c_i \sigma(a_i \cdot x + b_i) : c_i, b_i \in \mathbb{R}, \; a_i \in \mathbb{R}^d \right\}$$

We will see that this class of functions has a strong connection to artificial neural networks. Therefore, in applications, σ is often a sigmoid function, i.e., it is monotonically increasing and $\lim_{x \to -\infty} \sigma(x) = 0$ and $\lim_{x \to +\infty} \sigma(x) = 1$. If σ is a sigmoid function, $\mathcal{R}(\sigma)$ is also called the class of Artificial Neural Network functions and written as $\mathcal{M}(\sigma)$.
A question that arises with the function spaces introduced above is how dense they lie in well known function spaces like $C(\mathbb{R}^d)$, $C^k(\mathbb{R}^d)$, $B(\mathbb{R}^d)$ or $L^p(\mathbb{R}^d)$. Or in other words, how well can the elements of those function space be approximated by elements of the ridge function spaces. In the space $\mathcal{R}(a^1, ..., a^r)$ with fixed directional vectors, it quickly becomes clear that an arbitrarily good approximation will not be possible. However, we can make good statements about whether a given function f lies in $\mathcal{R}(a^1, ..., a^r)$ or not. A first result is provided by the following proposition:

Proposition 2.1.1. *[11] Let $a^1, ..., a^r \in \mathbb{R}^d$ be pairwise linearly independent vectors and let H^i be the hyperplane $\{c \in \mathbb{R}^d : c \cdot a^i = 0\}$. Then a function $f \in C^r(\mathbb{R}^d)$ can be represented as follows:*

$$f(x) = \sum_{i=1}^{r} g_i(a^i \cdot x) + P(x)$$

if

$$\prod_{i=1}^{r} \sum_{s=1}^{d} c_s^i \frac{\partial f}{\partial x_s} = 0$$

for all $c^i = (c_1^i, ..., c_d^i) \in H^i$. Here, $P(x)$ is a polynomial of at most degree r.

For a more general statement, we need a few more prerequisites.
First, we note that every polynomial $p(x_1, ..., x_d)$ defines a differential operator $p(\frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_d})$. For example, let $p(x_1, x_2) = x_1^2 + x_2^2$. Then $p(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}) = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$ defines the Laplace operator Δ.
Now let $P(a^1, ..., a^r)$ be the set of all polynomials that vanish over the lines $\{\lambda a^i, \; \lambda \in \mathbb{R}, \; i = 1, ..., r\}$. Here, $P(a^1, ..., a^r)$ is an ideal in the set of polynomials. Further, let Q be the set of polynomials $q = q(x_1, ..., x_d)$ such that $p(\frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_d})q = 0$ for all $p(x_1, ..., x_d) \in P(a^1, ..., a^r)$. Then the following theorem holds:

Theorem 2.1.1. *[25] Let $a^1, ..., a^r$ be pairwise linearly independent vectors in $\mathbb{R}^d$. A function $f \in C(\mathbb{R}^d)$ can be represented in the form*

$$f(x) = \sum_{i=1}^{r} g_i(a^i \cdot x)$$

if and only if f belongs to the closure of the linear span of Q.

2.2. Density Results

Now let us consider some results about the density of the above mentioned function spaces in the function spaces known to us, i.e. the $C(\mathbb{R}^d)$. But first consider the spaces $B(X)$, $C(X)$, and $T(X)$ of bounded, continuous, and real valued functions over a set $X \subset \mathbb{R}^d$ and correspondingly the sub spaces $\mathcal{R}_b(a^1, ..., a^r)$ and $\mathcal{R}_c(a^1, ..., a^r)$ of $\mathcal{R}(a^1, ..., a^r)$, which consist of the sums of bounded or continuous functions $g_i(a \cdot x)$ respectively, like they are defined in [2]. The question now arises when $\mathcal{R}_b(a^1, ..., a^r) = B(X)$, $\mathcal{R}_c(a^1, ..., a^r) = C(X)$, or $\mathcal{R}(a^1, ..., a^r) = T(X)$ is true. To answer the first two questions, we first need the following set construction:

$$\tau_i(Z) = \{x \in Z : |p_i^{-1}(p_i(x)) \cap Z| \geq 2\} \tag{2.1}$$

with $Z \subset X$ and $p_i(x) = a_i \cdot x$, $i = 1, ..., r$. Furthermore, we define $\tau(Z) = \bigcap_{i=1}^{r} \tau_i(Z)$ and $\tau^n(Z) = \tau(\tau^{n-1}(Z))$. With this, we can now formulate the following proposition:

Proposition 2.2.1 (Sternfeld [48]). *If for some $n \in \mathbb{N}$ it holds that $\tau^n(X) = \emptyset$, then $\mathcal{R}_b(a^1, ..., a^r) = B(X)$. Furthermore, if X is a compact subset of $\mathbb{R}^d$, then $\mathcal{R}_c(a^1, ..., a^r) = C(X)$.*

Since $\mathcal{R}_c(a^1, ..., a^r)$ is evidently not dense in $C(\mathbb{R}^d)$ ($\mathbb{R}^d$ is not a compact subset of itself), we will no longer fix the directional vectors and consider the space $\mathcal{R} = \text{span}\{g(a \cdot x) : g \in C(\mathbb{R}), a \in \mathbb{R}^d \backslash \{0\}\}$. Since all functions $g \in C(\mathbb{R})$ are considered, it does not matter whether the directional vector is $\mathbf{a}$ or $k\mathbf{a}$ with $k \in \mathbb{R}$. Therefore, only unit vectors will be used as directional vectors from now on, i.e., $a \in S^{d-1} = \{x \in \mathbb{R}^d | \ ||x|| = 1\}$. It will be shown that the space $\mathcal{R}(\mathcal{A}) = \text{span}\{g(a \cdot x) : g \in C(\mathbb{R}), a \in \mathcal{A} \subset \mathbb{R}^d \backslash \{0\}\}$ is dense in $C(\mathbb{R}^d)$.
To solve many of the mentioned problems, we need mathematical objects called cycles. These cycles depend on r functions $h_i : X \to \mathbb{R}, i = 1, ..., r$ or, in a more specific case, on r directional vectors $a_1, ..., a_r$. Additionally, let δ_A be the characteristic function defined as follows:

$$\delta_A(y) = \begin{cases} 1, & if\ y \in A \\ 0, & if\ y \notin A \end{cases}$$

Definition 2.2.1 (Cycle). *[16] Given $X \subset \mathbb{R}^d$ and functions $h_i : X \to \mathbb{R}, i = 1, ..., r$. A set of points $\{x_1, ..., x_n\} \subset X$ is called a cycle with respect to the functions $h_1, ..., h_r$, if there is a vector $\lambda = (\lambda_1, ..., \lambda_n)$ ($\lambda_i \in \mathbb{R} \backslash 0$) such that*

$$\sum_{j=1}^{n} \lambda_j \delta_{h_i(x_j)} = 0, \ i = 1, ..., r$$

For $h_i = a^i \cdot x$, $i = 1, ..., r$ with directional vectors $a^1, ..., a^r \in \mathbb{R}^d$, we call the cycle a linear cycle.

The following proposition now holds for cycles:

Proposition 2.2.2 (Sternfeld [48]). *For functions $h_i : X \to \mathbb{R}$, $i = 1, ..., r$, if every finite subset of X contains an $(r+1)$-cycle, then $\mathcal{R}_b(h_1, ..., h_r) = B(X)$. Furthermore, if X is a compact subset of $\mathbb{R}^d$, then $\mathcal{R}_c(h_1, ..., h_r) = C(X)$.*

Furthermore, we will need a specific functional. Let $\langle p, \lambda \rangle$ be a so-called cycle-vector pair, where $p = \{x_1, ..., x_n\}$ is a cycle in X and λ is the associated vector as defined above. We define the functional as follows:

$$G_{p,\lambda} : T(X) \to \mathbb{R}, \quad G_{p,\lambda}(f) = \sum_{j=1}^{n} \lambda_j f(x_j)$$

Here, $G_{p,\lambda}$ is linear, and $G_{p,\lambda}(f) = 0$ for $f \in \mathcal{B}(X)$.

Lemma 2.2.1. *[16] Let $X \subset \mathbb{R}^d$ have cycles and $h_i(X) \cap h_j(X) = \emptyset$ for $i, j \in \{1, ..., r\}$, $i \neq j$. Under these conditions, a function $f : X \to \mathbb{R}$ belongs to $\mathcal{B}(h_1, ..., h_r; X)$ if and only if $G_{p,\lambda}(f) = 0$ for every cycle-vector pair $\langle p, \lambda \rangle$ of X.*

Definition 2.2.2 (Minimal Cycles)**.** *[16] A cycle $p = \{x_1, ..., x_n\}$ is called minimal if it contains no proper cycle subset. It possesses the following properties:*

(a) The vector λ associated with p, as per Definition 2.2.1, is uniquely determined up to multiplication by a constant.

(b) If λ satisfies $\sum_{j=1}^{n} |\lambda_j| = 1$, then all λ_j are rational.

Thus, the minimal cycle p defines a unique (up to sign) functional:

$$G_p(f) = \sum_{j=1}^{n} \lambda_j f(x_j), \quad \sum_{j=1}^{n} |\lambda_j| = 1$$

The operator $G_{p,\lambda}$ and minimal cycles are related in a special way, as expressed by the following lemma:

Lemma 2.2.2. *[16] The functional $G_{p,\lambda}$ is a linear combination of functionals $G_{p_1}, ..., G_{p_k}$, where $p_1, ..., p_k$ are minimal cycles in p.*

The question of the relationship between $T(X)$ and $\mathcal{B}(h_1, ..., h_r; X)$ can now be answered by the following theorem:

Theorem 2.2.3. *[16] Let $X \subset \mathbb{R}^d$ and $h_1, ..., h_r$ be arbitrary but fixed real functions. Then:*

(1) If X has cycles with respect to functions $h_1, ..., h_r$, a function $f : X \to \mathbb{R}$ belongs to $\mathcal{B}(h_1, ..., h_r; X)$ if and only if $G_p(f) = 0$ for every minimal cycle $p \subset X$.

(2) If X has no cycles, then $T(X) = \mathcal{B}(h_1, ..., h_r; X)$.

Furthermore, the following theorem holds:

Theorem 2.2.4. *[16] $\mathcal{B}(h_1, ..., h_r; X) = T(X)$ if and only if X contains no cycles with respect to $h_1, ..., h_r$.*

These two theorems apply to general ridge functions, and naturally extend to the original problem involving vectors $a^1, ..., a^r$.

Corollary 2.2.4.1. *[16] Let $X \subset \mathbb{R}^d$ and $a^1, ..., a^r \in \mathbb{R}^d \backslash 0$. Then:*

(1) If X has cycles with respect to direction vectors $a^1, ..., a^r$, a function $f : X \to \mathbb{R}$ belongs to $\mathcal{R}(a^1, ..., a^r; X)$ if and only if $G_p(f) = 0$ for every minimal cycle $p \subset X$.

(2) If X has no cycles, then $T(X) = \mathcal{R}(a^1, ..., a^r; X)$.

Corollary 2.2.4.2. *[16] $\mathcal{R}(a^1, ..., a^r; X) = T(X)$ if and only if X contains no cycles with respect to vectors $a^1, ..., a^r$.*

Earlier, we saw that $\mathcal{R}_c(a^1, ..., a^r) = C(X)$ holds over a compact set $X \subset \mathbb{R}^d$. However, is it possible to find necessary and sufficient conditions for $\mathcal{R}(a^1, ..., a^r) = C(X)$? This problem is to be solved for $r = 2$. To do this, we take arbitrary but fixed continuous functions h_1 and h_2 over X. Furthermore, we define

$$H_1 = H_1(X) = \{g_1(h_1(x)) : \ g_1 \in C(h_1(X))\}$$
$$H_2 = H_2(X) = \{g_2(h_2(x)) : \ g_2 \in C(h_2(X))\}$$

With suitable choice of functions h_1 and h_2, $\mathcal{R}(a^1, a^2) = H_1 + H_2$. The aim is to show when $H_1(X) + H_2(X) = C(X)$ holds.
To solve this problem, we first need the concept of a *path*.

Definition 2.2.3. *[16] Let X be a compact subset of $\mathbb{R}^d$ and $h_i \in C(X), i = 1, 2$. A finite ordered subset $(p_1, p_2, ..., p_m)$ of X with $p_i \neq p_{i+1}$ $(i = 1, ..., m-1)$ and either $h_1(p_1) = h_1(p_2), h_2(p_2) = h_2(p_3), h_1(p_3) = h_1(p_4), ...$ or $h_2(p_1) = h_2(p_2), h_1(p_2) = h_1(p_3), h_2(p_3) = h_2(p_4), ...$ is called a path with respect to functions h_1 and h_2, or simply an $h_1 - h_2$ path.*

According to the above definition, if $p_1 = p_m$ and if $m - 1$ is even, then the $h_1 - h_2$ path $(p_1, p_2, ..., p_{m-1})$ is called *closed*. Furthermore, the relationship of two points belonging to the same $h_1 - h_2$ path is an equivalence relation. We write $a \approx b$. The equivalence classes of this relation are called *orbits*.

Theorem 2.2.5. *[16] Let X be a compact subset of $\mathbb{R}^d$, and let all orbits of X be closed. Then, $H_1(X) + H_2(X)$ is uniformly dense in $C(X)$ if and only if X contains no closed $h_1 - h_2$ path.*

We see, that we can make some statements about the density about the ridge function spaces in the function spaces, that are used in practice. Now we want to take a look at how the properties of a given function of the ridge function spaces transfer to the composition functions g_i.

2.3. Function Properties

If we can represent a function f as a sum of ridge functions, i.e.,

$$f(x) = \sum_{i=1}^{r} g_i(a^i \cdot x) \tag{2.2}$$

the question arises as to what extent we can derive properties of the functions g_i from the properties of f.
To investigate this, let $X \subset \mathbb{R}^d$ for $d \geq 2$ and for $a = (a_1, ..., a_d)$, we define $\Delta(a) = \{a \cdot x : x \in X\}$, as well as $\Delta_i = \Delta(a_i)$. The function f, as represented above, belongs to a smoothness class $W(X)$. The question now is under what conditions we can guarantee that all functions g_i also belong to this class, i.e., when $g_i \in W(\Delta_i)$. The following results are mainly taken from the works of S. V. Konyagin, A. A. Kuleshov, and V. E. Maiorov [22]
For the first general and subsequent statements, we first describe a special class of functions B or $B_{(\alpha,\beta)}$ for an interval $(\alpha, \beta) \subset \mathbb{R}$ as follows:

1) The prerequisite $f \in B_{(\alpha,\beta)}$ and $r \in C(\alpha,\beta)$ as well as the fact that $f - r$ is additive on (α,β) implies that $f(x) - r(x) = cx$ for all $x \in (\alpha,\beta)$ and $c \in \mathbb{R}$.

2) For all $h \in \mathbb{R}$, $|h| < \beta - \alpha$, $f \in B_{(\alpha,\beta)}$ implies that $\Delta_h f \in B_J$, where $\Delta_h f(x) = f(x+h) - f(x)$ and $J = (\alpha,\beta) \cap (\alpha - h, \beta - h)$.

3) For all $(c,d) \subset (\alpha,\beta)$, $f \in B_{(\alpha,\beta)}$ implies that $f \in B_{(c,d)}$.

It has been shown that all properties 1)-3) are satisfied if $B_{(\alpha,\beta)}$ is assumed to be the union of locally bounded and Lebesgue-measurable functions on (α,β). With this, we obtain the first statements:

Theorem 2.3.1. *[36] If a function f with $f \in C^k(\mathbb{R}^d)$, $k \in \mathbb{Z}^+$ has the form as in 2.2 and all functions g_i belong to the class B, then $g_i \in C^k(\mathbb{R})$ for $i = 1, ..., r$.*

Another answer is provided by the following theorem, proven by S. V. Konyagin and A. A. Kuleshov [22]:

Theorem 2.3.2. *[24] Let $X \subset \mathbb{R}$ be a non-empty, open set and W be one of the function classes D^k (The set of all k-times differentiable functions with bounded support) or C^k with $k \in \mathbb{Z}^+$. Then the following statements are equivalent:*

1 The vectors a^i are linearly independent.

2 For every function f of the form 2.2, if $f \in W(X)$, then also $g_i \in W(\Delta_i)$ for $i = 1, ..., r$.

Additionally, the following theorem provides a statement for open, convex sets:

Theorem 2.3.3. *[22, 24] Let $X \subset \mathbb{R}$ be an open, convex set and W be one of the function classes D^k or C^k with $k \in \mathbb{Z}^+$. If a function $f \in W(X)$ has the form 2.2 and $g_i \in B_{\Delta_i}$ for $i = 1, ..., n$, then $g_i \in W(\Delta_i)$.*

For further statements about the properties of f and g_i, we need the concept of a convex body. This is defined for closed, convex sets $X \subset \mathbb{R}^d$ with $int(X) \neq \emptyset$.

Theorem 2.3.4. *[21] Let X be a convex body in $\mathbb{R}^d$, $X \neq \mathbb{R}^d$, and $r \geq 2$. Then the following statements are equivalent:*

1) The boundary of X is smooth.

2) For arbitrary vectors $a^1, ..., a^r$ and arbitrary functions $g_1, ..., g_r$ that are continuous in $int(\Delta_i)$ $(i = 1, .., r)$, the continuity of f of the form 2.2 implies the continuity of the functions g_i on Δ_i.

3) For arbitrary vectors $a^1, ..., a^r$ and arbitrary functions $g_1, ..., g_r$ that are continuous in $int(\Delta_i)$ $(i = 1, .., r)$, the continuity of f of the form 2.2 implies the local boundedness of g_i on Δ_i.

2.4. Approximation Theory of Ridge Functions

In the following, we will discuss how ridge functions can approximate other functions from specific classes. For this, we first need the concept of best approximation.

Definition 2.4.1 (Best Approximation)**.** *[22] For two subsets W and V of a Banach space X, the best approximation of W by V is defined as*

$$E(W,V)_X := \sup_{f\in W} E(f,V)_X$$

where $E(f,V)_X := \inf_{v\in V} ||f-v||_X$.

Furthermore, for two positive sequences $\{a_n\}_{n=1}^{\infty}$ and $\{b_n\}_{n=1}^{\infty}$, we write that $a_n \asymp b_n$ (a_n is equivalent to b_n) if there are two positive real numbers c_1, c_2 such that $c_1 \leq a_n/b_n \leq c_2$ for all $n = 1, 2, ...$

2.4.1. Approximation of Radial and Harmonic Functions

In the first section of the chapter, we will focus on the approximation theory of ridge functions concerning radial and harmonic functions over $\mathbb{R}^2$. Let f be a radial function with $f \in L_2(\mathbb{B}^2)$, where $\mathbb{B}^d$ is the d-dimensional unit ball defined as

$$\mathbb{B}^d := \{x \in \mathbb{R}^d : |x| \leq 1\}$$

Theorem 2.4.1. *[32, 33] The best estimates for approximating a radial function f by the sets $\mathcal{R}(a^1, ..., a^r)$ and $\mathcal{R}_r$ with fixed, uniformly distributed, or free direction vectors agree with the best approximation of f by polynomials of the space $\mathcal{P}_r^d$. That is,*

$$E(f,\mathcal{R}_r)_{L_2(\mathbb{B}^2)} \asymp E(f,\mathcal{R}(a^1,...,a^r))_{L_2(\mathbb{B}^2)} \asymp E(f,\mathcal{P}_r^d)_{L_2(\mathbb{B}^2)}$$

Theorem 2.4.2. *[33, 34] Let h be a harmonic function over $\mathbb{B}^2$, and the best approximation of h by polynomials of degree r satisfies the relation $E(h,\mathcal{P}_r)_{L_2(\mathbb{B}^2)} \asymp r^{-\alpha}$ for a positive α. Then:*

a) The best approximation of h by the set of ridge functions $\mathcal{R}_r$ satisfies the inequality

$$E(h,\mathcal{R}_r)_{L_2(\mathbb{B}^2)} \leq \frac{c}{n^{3\alpha/2-\epsilon}} \ \forall \epsilon > 0$$

b) There exists a harmonic function h^* over $\mathbb{B}^2$, such that

$$E(h^*,\mathcal{R}_r)_{L_2(\mathbb{B}^2)} \geq \frac{c'}{n^{3\alpha/2}}$$

where c and c' are constants that depend only on α and ϵ.

2.4.2. Approximation of Sobolev Functions by Ridge Functions

Now we want to consider the approximation of functions from the so-called Sobolev spaces. For this purpose, let $k_1, \ldots, k_d$ be non-negative integers and $k = k_1 + \ldots + k_d$ a positive integer. Then, the mixed differentiation operator applied to a function $f : \mathbb{R}^d \to \mathbb{R}$ is defined as

$$D^{(k_1,\ldots,k_d)} f := \frac{\partial^k f}{\partial x_1^{k_1} \ldots \partial x_d^{k_d}}.$$

We now consider a class of specific Sobolev functions:

$$W_p^k(\mathbb{B}^d) := \{f : \|f\|_{L_p} + \max_{k_1,\ldots,k_d} \|D^{(k_1,\ldots,k_d)} f\|_{L_p} \leq 1\}.$$

With these conditions, the following theorem holds:

Theorem 2.4.3. *[27] Let $d \geq 2$, $k > 0$, and p, q be arbitrary numbers with $1 \leq p \leq q \leq \infty$. Then,*

$$E(W_p^k, \mathcal{R}_r)_{L_p(\mathbb{B}^d)} \asymp E(W_p^k, \mathcal{R}(a^1, \ldots, a^r))_{L_p(\mathbb{B}^d)} \asymp r^{-k/(d-1)}$$

2.4.3. Approximation by Ridge Functions with Fixed Profile

To conclude this section, instead of using the previously used ridge function spaces, we return to the space of ridge functions with fixed profiles, which is particularly relevant in applications. As a reminder:

$$\mathcal{R}(\sigma) := \mathcal{R}_r(\sigma) = \left\{ \sum_{i=1}^{r} c_i \sigma(a_i \cdot x + b_i) : c_i, b_i \in \mathbb{R},\ a_i \in \mathbb{R}^d \right\}$$

For this space, in conjunction with the Sobolev class defined above, the following theorem holds:

Theorem 2.4.4. *[28] Let k, d be positive integers and $p \in [1, \infty]$. Then there exists a function $\sigma \in C^\infty(\mathbb{R})$, for which the best approximation of W_p^r by $\mathcal{R}_r(\sigma)$ in the L_p space satisfies the following relation:*

$$E(W_p^k, \mathcal{R}_r(\sigma))_{L_p(\mathbb{B}^d)} \asymp r^{-k/(d-1)}$$

In summary, this means that the approximation error between the function spaces W_p^k and $\mathcal{R}_r(\sigma)$ can be made arbitrarily small, depending on the number r of terms in the sum. However, W_p^k represents a very specific class of functions, and verifying if a function belongs to this class can be cumbersome in practice. More satisfactory results are provided later about the density of $\mathcal{R}_r(\sigma)$ in $C(\mathbb{I}_d)$.

So we see, that ridge functions got some very special properties in terms of function approximation. In this chapter, we focused on function spaces relevant for applications, particularly $C(\mathbb{R})$. While we have strong density results in the space of ridge functions with variable direction vectors, where we have flexibility in choosing the number of vectors, we note that in neural networks, direction vectors correspond directly to the weight vectors, which are finite. Although there are existing density results, such as those in proposition 2.2.1, these can be cumbersome and do not provide clear insights into the quality of approximation in practical applications. However, a slightly modified version of the space $\mathcal{R}_r$ offers strong density properties in $C(\mathbb{R})$, making it highly effective for neural network applications. In the next chapter, we turn our attention to this modified case.

3. Ridge Functions and Artificial Neural Networks

The theory of ridge functions can be effectively applied to artificial neural networks (ANN's) with specific layer architectures, as the neurons or groups of neurons in ANN's can be viewed as ridge functions or sums of ridge functions. Accordingly, it is natural for us to consider the space $\mathcal{R}(\sigma)$ in the following discussion, with an appropriate sigmoidal activation function $\sigma : \mathbb{R} \to \mathbb{R}$. However, before delving into this specific application, let's first examine some mathematical properties of neural networks.

3.1. Artificial Neural Networks

The basic building block of an ANN is the artificial neuron, essentially a mathematical abstraction of a biological neuron in the brain. This artificial neuron, as depicted in the following figure, takes as input a vector $x \in \mathbb{R}^d$. This input vector flows into a transfer function with corresponding weights $\{w_{i,j}\}_{i=1}^{d}$ for each x_i. Typically, this transfer function is the standard scalar product, so $net_j = \sum_{i=1}^{d} x_i w_{i,j}$. The resultant value, along with a threshold or bias θ_j, is fed into the activation function σ, which determines the output of the neuron, specifically

$$\sigma_j := \sigma(net_j - \theta_j)$$

The basic model of an artificial neuron is shown in figure 3.1.

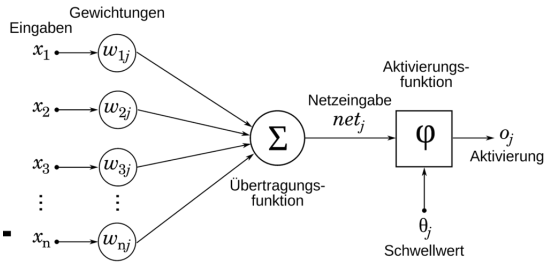


Figure 3.1.: Model of an artificial neuron
[51]

Neurons can now be arranged in various architectures. The most well-known is the *multilayer feed-forward model*, where neurons are arranged in layers of varying sizes. The first layer is called the *input layer*, the last layer the *output layer*, and any layers in between are called *hidden layers*. This is shown in figure 3.2.

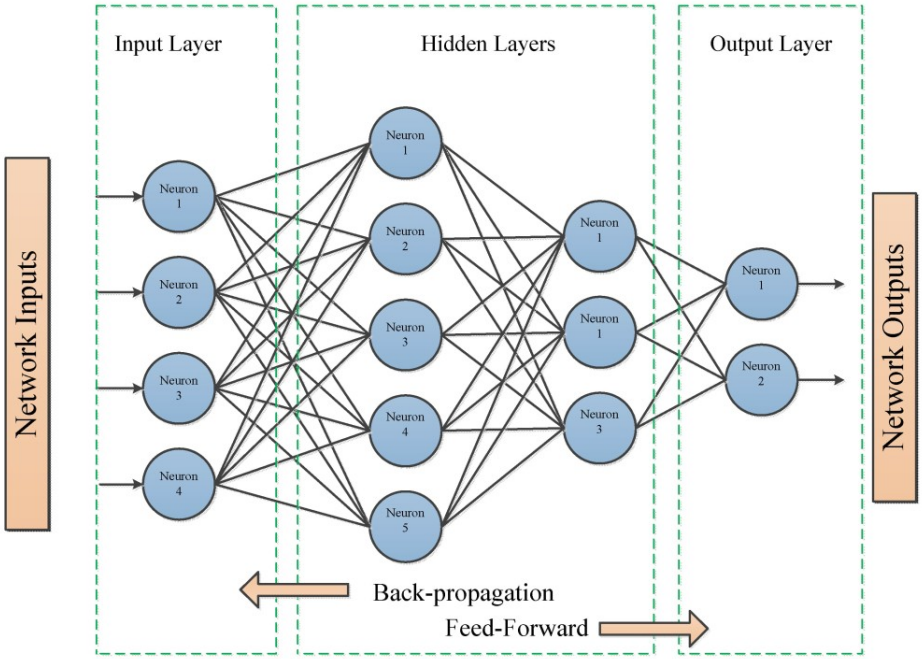


Figure 3.2.: Model of a feed forward network with hidden layers [12]

The input values are then passed through the individual neurons in the network as described above, up to the output layer. There, they are compared with a reference value using a loss function. There are various types of loss functions available, but the mean squared error (MSE) is commonly used. For an output value $o = (o_1, ..., o_d) \in \mathbb{R}^d$ and a target value $t = (t_1, ..., t_d) \in \mathbb{R}^d$, the MSE is defined as

$$MSE = \frac{1}{2}\sum_{i=1}^{d}(t_i - o_i)^2$$

The goal is to train the network to minimize this error. This means, for a given input, it should produce a desired output, such as classifying an image into a predefined class. At this point, the mathematical consideration of neural networks as universal function approximators also comes into play. For example, if we have an image with 10,000 pixels and want to assign it to one of ten classes, this is equivalent to searching for a function $f : \mathbb{Z}_{256}^{10000} \to [0,1]^{10}$. The aim is therefore to train the weights of the feed foward network in such a way that this function is approximated as well as possible or that the error between the network and the target function is as small as possible. Training the network involves adjusting the initially randomly chosen weights and thresholds so that such a mapping can occur. This is done using the principle of *backpropagation*. Initially, the gradient of MSE is computed layer by layer using the chain rule:

$$\frac{\partial MSE}{\partial w_{i,j}} = \frac{\partial MSE}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{i,j}}$$

The computed gradient can then be used to adjust the weights in a Newton-like update rule:

$$w_i = w_i - \alpha\frac{\partial MSE}{\partial w_i}$$

Here, α is the learning rate, which can be chosen arbitrarily but sensibly.
These steps are repeated until a certain criterion is met, usually a desired level of accuracy in the mapping or if a given number of epochs is reached.

3.2. Ridge Functions in Artificial Neural Networks

As indicated at the beginning of this chapter, we now aim to apply some insights from the theory of ridge functions to ANN's. To begin with, we revisit some more general results that are conceptually related to the first chapter. Since the number of weights required for computations in the neural network is finite, let us first revisit the space $\mathcal{R}(a^1, ..., a^r; X)$ and formulate the following theorems that will assist us in the subsequent discussion. These theorems are results of the work of Sproston and Strauss [47].

Theorem 3.2.1. *[17] Let $X \subset \mathbb{R}^d$ be a compact set, and define $\tau_i(X)$, $i = 1, ..., r$ as in 2.1. If $\cap_{i=1,2,...}\tau_i(X) = \emptyset$, then $\mathcal{R}(a^1, ..., a^r; X)$ is dense in $C(X)$.*

Additionally, the following lemma holds.

Lemma 3.2.2. *[17] If $\mathcal{R}(a^1, ..., a^r; X)$ is dense in $C(X)$, then X does not contain cycles with respect to the directions $a^1, ..., a^r$.*

Proof. Assume X contains cycles. Each cycle $l = (x_1, ..., x_n)$ and the corresponding vector $\lambda = (\lambda_1, ..., \lambda_n)$ generate the functional

$$G_{l,\lambda}(f) = \sum_{j=1}^{n} \lambda_j f(x_j), \;\; f \in C(X)$$

Here, $G_{l,\lambda}$ is linear and continuous, with a norm $\sum_{j=1}^{n} |\lambda_j|$. Moreover, for all $g \in \mathcal{R}(a^1, ..., a^r)$, $G_{l,\lambda}(g) = 0$. Now, let f_0 be a continuous function such that $f_0(x_j) = 1$ for $\lambda_j > 0$ and $f_0(x_j) = -1$ for $\lambda_j < 0$, $j = 1, ..., n$. For this function, $G_{l,\lambda}(f_0) \neq 0$. Hence, while f_0 lies in $C(X)$, it cannot be approximated continuously by functions from $\mathcal{R}(a^1, ..., a^r)$. Therefore, $\mathcal{R}(a^1, ..., a^r)$ is not dense in $C(X)$, proving the lemma. □

With these statements, we can now demonstrate properties specifically tailored to neural networks. Let $\sigma \in C(\mathbb{R})$ be a continuous activation function, and let $W \subset \mathbb{R}^d$. We define $\mathcal{M}(\sigma; W, \mathbb{R})$ as the set of ANN's with weights from W. That is,

$$\mathcal{M}(\sigma; W, \mathbb{R}) = \text{span}\{\sigma(w \cdot x - b) |\, w \in W, b \in \mathbb{R}\}$$

Theorem 3.2.3. *[16] Let $\sigma \in C(\mathbb{R}) \cap L_p(\mathbb{R})$ with $1 \leq p < \infty$, or a continuous, bounded, non-constant function with limits at $-\infty$ or ∞. Further, let $W = \{a^1, ..., a^r\} \subset \mathbb{R}^d$ be a set of given weights, and X a compact subset of $\mathbb{R}^d$. Then the following statements hold:*

(1) If $\cap_{i=1,2,...}\tau_i(X) = \emptyset$, then $\mathcal{M}_X(\sigma; W, \mathbb{R})$ is dense in $C(X)$.

(2) If $\mathcal{M}_X(\sigma; W, \mathbb{R})$ is dense in $C(X)$, then X does not contain cycles.

Proof. For (1): Let X be a compact subset of $\mathbb{R}^d$ with $\cap_{i=1,2,...}\tau_i(X) = \emptyset$. According to Theorem 3.2.1, $\mathcal{R}(a^1, ..., a^r)$ is dense in $C(X)$. Thus, for every $\epsilon > 0$, there exist continuous univariate functions g_i, $i = 1, ..., r$, such that

$$|f(x) - \sum_{i=1}^{r} g_i(a^i \cdot x)| < \frac{\epsilon}{r+1} \tag{3.1}$$

for every $x \in X$ and $f \in C(X)$. Since X is compact, the set $Y_i = \{a^i \cdot x |\, x \in X\}$, $i = 1, ..., k$, is also compact. For the next step, we need the concept of a *quasi-periodic function*:

A function $f \in C(\mathbb{R}^d)$ is said to be quasi-periodic if the set $span\{f(x-b) : b \in \mathbb{R}^d\}$ is not dense in $C(\mathbb{R}^d)$ in the sense of uniform convergence on compact sets.
It has been shown that p-times Lebesgue-integrable and continuous, univariate functions or continuous, bounded, non-constant functions with limits at $-\infty$ or ∞ are not quasi-periodic. It has been shown, moreover, that

$$span\{\sigma(y-\theta),\ \theta \in \mathbb{R}\}$$

is dense in $C(\mathbb{R})$ in the sense of uniform convergence on compact sets. This means that, for a given $\epsilon > 0$, there exist numbers $c_{i,j}, \theta_{i,j} \in \mathbb{R}$, $i = 1, ...r$, $j = 1, ..., m_i$ so that

$$|g_i(y) - \sum_{j=1}^{m_i} c_{i,j}\sigma(y - \theta_{i,j})| < \frac{\epsilon}{r+1} \tag{3.2}$$

for all $y \in Y_i$, $i = 1, ..., r$. From 3.1 and 3.2, we obtain

$$||f(x) - \sum_{i=1}^{r}\sum_{j=1}^{m_i} c_{i,j}\sigma(a^i \cdot x - \theta_{i,j})||_{C(X)} < \epsilon \tag{3.3}$$

Thus, $\overline{\mathcal{M}_X(\sigma; W, \mathbb{R})} = C(X)$
For (2): Let X be a compact subset of $\mathbb{R}^d$ and $\mathcal{M}_X(\sigma; W, \mathbb{R})$ be dense in $C(X)$. Then, for any $\epsilon > 0$, inequality 3.3 holds for certain $c_{i,j}, \theta_{i,j} \in \mathbb{R}$, $i = 1, ...r$, $j = 1, ..., m_i$. As for each $i = 1, .., k$, the function $\sum_{j=1}^{m_i} c_{i,j}\sigma(a^i \cdot x - \theta_{i,j})$ has the form $g_i(a^i \cdot x_i, \mathcal{R}_X(a^1, ..., a^r)$ is dense in $C(X)$. According to Lemma 3.2.1, X therefore has no cycles. □

In the following, let $\mathbb{I}^d = [0,1]^d$ denote the d-dimensional unit cube, and $M(\mathbb{I}^d)$ the space of finite, signed, regular Borel measures over $\mathbb{I}$. These are first applied in the concept of the *discriminant function.*

Definition 3.2.1. *A function $\sigma : \mathbb{R}^d \to \mathbb{R}$ is discriminant if for a measure $\mu \in M(\mathbb{I}^d)$, the condition*

$$\int_{\mathbb{I}^d} \sigma(a \cdot x + b)\, d\mu(x) = 0$$

implies that $\mu = 0$ for $a \in \mathbb{R}^d$, $b \in \mathbb{R}$.

Earlier in chapters 2.2 and2.4, we dealt with approximation and density properties of ridge functions. We now revisit this and observe that functions $\sigma \in \text{span}(\mathcal{R}_r(\sigma))$ possess particularly good properties in this regard.

Theorem 3.2.4. *[10] Let σ be a continuous discriminant function. Then $span(\mathcal{R}_r(\sigma))$, which is the set of all sums $G(x)$ of the form*

$$G(x) = \sum_{i=1}^{r} c_i\sigma(a_i \cdot x + b_i),$$

is dense in $C(\mathbb{I}_d)$. In other words, for every $f \in C(\mathbb{I}_d)$ and every $\epsilon > 0$, there exists a sum $G(x) \in \mathcal{R}_r(\sigma)$ such that

$$|G(x) - f(x)| < \epsilon \quad \forall x \in \mathbb{I}_d.$$

Proof. Let $S \subset C(\mathbb{I}_d)$ be the set of sums $G(x)$ as defined above. Clearly, S is a linear subspace of $C(\mathbb{I}_d)$. We now aim to show that the closure of S is exactly $C(\mathbb{I}_d)$.

Suppose the closure of S does not equal $C(\mathbb{I}_d)$. Then the closure of S, denoted by R, is a proper subspace of $C(\mathbb{I}_d)$. By the Hahn-Banach theorem, there exists a bounded linear functional L on $C(\mathbb{I}_d)$ such that $L \neq 0$, but $L(S) = L(R) = 0$.

According to the Riesz representation theorem, this functional takes the form

$$L(h) = \int_{\mathbb{I}_d} h(x)\, d\mu(x),$$

for some $\mu \in M(\mathbb{I}_d)$ and $h \in C(\mathbb{I}_d)$. Since $\sigma(a \cdot x + b)$ lies in R for all a and b, it must hold that

$$\int_{\mathbb{I}_d} \sigma(a \cdot x + b)\, d\mu(x) = 0.$$

However, we assumed σ to be a discriminant function, implying $\mu(x) = 0$. This contradicts our assumption, proving that S must be dense in $C(\mathbb{I}_d)$. □

Thus, we have established that for every discriminant function σ, $\mathcal{R}_r(\sigma)$ is dense in $C(\mathbb{I}_d)$. For applications, sigmoid functions are particularly relevant. What can be said about these? The following lemma provides an answer.

Lemma 3.2.5. *[10] Every bounded, measurable sigmoid function σ is discriminant. Specifically, every continuous sigmoid function is discriminant.*

Proof. For $x, a \in \mathbb{I}_d$ and $b, \phi, \lambda \in \mathbb{R}$, define $\sigma_\lambda(x) := \sigma(\lambda(a \cdot x + b) + \phi)$. For this function, we have:

$$\sigma_\lambda(x) \begin{cases} \to 1, & \text{if } a \cdot x + b > 0 \text{ as } \lambda \to +\infty, \\ \to 0, & \text{if } a \cdot x + b < 0 \text{ as } \lambda \to +\infty, \\ = 0, & \text{if } a \cdot x + b = 0 \text{ for all } \lambda \in \mathbb{R}. \end{cases}$$

Thus, σ_λ converges pointwise and bounded to

$$\gamma(x) = \begin{cases} = 1, & \text{if } a \cdot x + b > 0, \\ = 0, & \text{if } a \cdot x + b < 0, \\ = \sigma(\phi), & \text{if } a \cdot x + b = 0 \text{ for all } \lambda \in \mathbb{R} \text{ as } \lambda \to +\infty. \end{cases}$$

Let $\Pi_{\gamma,b}$ be the hyperplane $\{x | a \cdot x + b = 0\}$ and $H_{\gamma,b}$ the half-plane $\{x | a \cdot x + b > 0\}$. According to Lebesgue's theorem,

$$\begin{aligned} 0 &= \int_{\mathbb{I}_d} \sigma_\lambda(x)\, d\mu(x) \\ &= \int_{\mathbb{I}_d} \gamma(x)\, d\mu(x) \\ &= \sigma(\phi)\mu(\Pi_{\gamma,b}) + \mu(H_{\gamma,b}), \end{aligned}$$

for all ϕ, a, b.

We now show that if the measure μ over all half-spaces is 0, then μ itself is 0. Let a be arbitrary but fixed. For a bounded, measurable function h, define the linear functional F by

$$F(h) = \int_{\mathbb{I}_n} h(a \cdot x)\, d\mu(x).$$

Moreover, F is a bounded functional on $L^\infty(\mathbb{R})$, since μ is a finite, signed measure.

Let h be the indicator function on $[b, \infty)$, i.e., $h(x) = 1$ for $x \geq b$ and $h(x) = 0$ for $x < b$. Then

$$F(h) = \int_{\mathbb{I}_d} h(a \cdot x)\, d\mu(x) = \mu(\Pi_{\gamma,-b}) + \mu(H_{\gamma,-b}) = 0.$$

Similarly, $F(h) = 0$ if h is the indicator function over the open interval (b, ∞). Due to the linearity of F, $F(h) = 0$ for the indicator function of any interval, hence for every simple function. Since simple functions are dense in $L^\infty(\mathbb{R})$, we have $F = 0$.

Specifically, for the bounded and measurable functions $s(u) = \sin(m \cdot u)$ and $c(u) = \cos(m \cdot u)$,

$$F(s + ic) = \int_{\mathbb{I}_d} \cos(m \cdot x) + i \sin(m \cdot x)\, d\mu(x) = \int_{\mathbb{I}_d} e^{i(a \cdot x)}\, d\mu(x) = 0,$$

for all m. Since the Fourier transform of μ is 0, μ itself must be 0. Thus, σ is discriminant. □

The density properties of discriminant sigmoidal functions, as found in neural networks, can now be applied to the problem of mapping. That is, for example, we want to assign an image of an animal to a specific species (rabbit, fox, etc.) and train the network accordingly. To do this, we partition the unit cube $\mathbb{I}_d$ into disjoint, measurable partitions $P_1, \ldots, P_k$ and define a decision function f such that

$$f(x) = j \iff x \in P_j.$$

Now, the first question is to what extent we can approximate such a decision function using a 1-hidden-layer network.

Theorem 3.2.6. *[10] Let σ be a continuous sigmoidal function, f be a decision function for a measurable finite partition of $\mathbb{I}_d$, and m be the Lebesgue measure. For every $\epsilon > 0$, there exists a sum of the form*

$$G(x) = \sum_{i=1}^{r} c_i \sigma(a_i \cdot x + b_i)$$

and a set $D \subset \mathbb{I}_n$ such that $m(D) > 1 - \epsilon$ and

$$|G(x) - f(x)| < \epsilon \quad \forall x \in D.$$

Proof. By Lusin's theorem, there exists a continuous function h and a set D with $m(D) > 1 - \epsilon$ such that $h(x) = f(x)$ for $x \in D$. Since h is finite, according to the previous theorem, we can find a sum $G(x)$ such that $|G(x) - f(x)| < \epsilon$ for all $\mathbb{I}_n \in D$. Then for $x \in D$,

$$|G(x) - f(x)| = |G(x) - h(x)| < \epsilon.$$

□

Depending on the conditions imposed on the activation function, different density results can be derived for other spaces, as seen in the following theorems.

Theorem 3.2.7. *[10] Let σ be a bounded, measurable sigmoidal function. Then span$(\mathcal{R}_r(\sigma))$, which is the set of all sums $G(x)$ of the form*

$$G(x) = \sum_{i=1}^{r} c_i \sigma(a_i \cdot x + b_i)$$

is dense in $L^1(\mathbb{I}_n)$. In other words, for every $f \in L^1(\mathbb{I}_n)$ and every $\epsilon > 0$, there exists a sum $G(x) \in \mathcal{R}_r(\sigma)$ such that

$$|G(x) - f(x)| < \epsilon \quad \forall x \in \mathbb{I}_n.$$

Theorem 3.2.8. *[10] Let σ be a general sigmoidal function, f be a decision function for a measurable finite partition of $\mathbb{I}_d$, and m be the Lebesgue measure. For every $\epsilon > 0$, there exists a sum of the form*

$$G(x) = \sum_{i=1}^{r} c_i \sigma(a_i \cdot x + b_i)$$

and a set $D \subset \mathbb{I}_d$ such that $m(D) > 1 - \epsilon$ and

$$|G(x) - f(x)| < \epsilon \quad \forall x \in D.$$

3.3. 2-hidden-layer-ANN's

In the last chapter, it was shown that any function can be approximated arbitrarily well by a neural network with one hidden layer. The main result was that $\mathcal{M}_r(\sigma)$ is dense in $C(\mathbb{R})$. However, this is only the case when r can be chosen variably. For a fixed r, this statement does not hold, even if different univariate functions are used. We will see that this behavior does not transfer to ANNs with more hidden layers, which we now want to consider. The following results were also treated by Vugar E. Ismailov [16]

First, similar to $\mathcal{M}_r(\sigma)$, we want to describe a *two hidden layer* model. This results from the appropriate repeated application of the first model. Thus, the output of this model for r neurons in the first layer and s neurons in the second layer for an input $x \in \mathbb{R}^d$ is given by:

$$\sum_{i=1}^{r} d_i \sigma \left(\sum_{j=1}^{s} c_{ij} \sigma(w^{ij} \cdot x - \theta_{ij}) - \gamma_i \right)$$

Here, $c_{ij}, \theta_{ij}, d_i, \gamma_i \in \mathbb{R}$ and weights $w^{ij} \in \mathbb{R}^d$, along with an appropriate univariate function σ, which will again be sigmoidal. Before we proceed, here are some examples of such functions in practice:

- $\sigma(t) = \frac{1}{1+e^{-t}}$
- $\sigma(t) = \begin{cases} 0, & \text{if } t < -1 \\ \frac{t+1}{2}, & \text{if } -1 \leq t \leq 1 \\ 1, & \text{if } t > 1 \end{cases}$
- $\sigma(t) = \frac{1}{\pi} \arctan(t) + \frac{1}{2}$
- $\sigma(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} e^{-x^2/2}\, dx$

In this chapter, we will show that there exist 2-hidden-layer networks which can approximate any real function with d-dimensional input arbitrarily well, using d neurons in the first layer and $2d+2$ neurons in the second layer. The insights associated with this are based on Kolmogorov's superposition theorem.

Theorem 3.3.1 (Kolmogorov's superposition theorem)**.** *For the unit cube $\mathbb{I}^d = [0,1]^d$, where $d \geq 2$, there exist constants $\lambda_q > 0$, $q = 1, ..., d$, with $\sum_{q=1}^{d} \lambda_q = 1$, and non-decreasing continuous functions $\Phi_p : [0,1] \to [0,1]$, $p = 1, ..., 2d+1$, such that every function $f : \mathbb{I}^d \to \mathbb{R}$ can be represented as:*

$$f(x_1, ..., x_d) = \sum_{p=1}^{2d+1} g\Big(\sum_{q=1}^{d} \lambda_q \Phi_p(x_q) \Big) \tag{3.4}$$

Maiorov and Pinkus used this result to prove the following important theorem.

Theorem 3.3.2. *[28] There exists an activation function σ that is analytic, strictly increasing, sigmoidal, and satisfies the following properties: For every function $f \in C[0,1]^d$ and every $\epsilon > 0$, there exist constants d_i, c_{ij}, θ_{ij}, γ_i and vectors $w^{ij} \in \mathbb{R}^d$ such that*

$$\Big| f(x) - \sum_{i=1}^{6d+3} d_i \sigma\Big(\sum_{j=1}^{3d} c_{ij} \sigma(w^{ij} \cdot x - \theta_{ij}) - \gamma_i \Big) \Big| < \epsilon \tag{3.5}$$

for all $x \in [0,1]^d$.

In these results, the number of terms can be improved to $2d+2$ and d from $6d+3$ and $3d$, respectively, with slight relaxation of the conditions on σ.

Definition 3.3.1 (λ-Monotonicity)**.** *Let λ be a non-negative real number. A function $f : (a,b) \to \mathbb{R}$ is called λ-increasing (λ-decreasing) if there exists an increasing (decreasing) function $u : (a,b) \to \mathbb{R}$ such that $|f(x) - g(x)| < \epsilon$ for all $x \in (a,b)$. If u is strictly increasing (strictly decreasing), then f is called strictly λ-increasing (strictly λ-decreasing).*
The concept of 0-monotonicity corresponds to the standard concept of monotonicity.

Thus, we can formulate the following theorem:

Theorem 3.3.3. *[16] For all positive numbers α and λ, there exists a sigmoidal activation function $\sigma \in C^\infty(\mathbb{R})$ that is strictly increasing on $(-\infty, \alpha)$ and strictly λ-increasing on $[\alpha, +\infty)$, and satisfies the following properties:*
For every function $f \in C[0,1]^d$ and every $\epsilon > 0$, there exist constants d_p, c_{pq}, θ_{pq}, γ_p and vectors $w^{pq} \in \mathbb{R}^d$ such that

$$\Big| f(x) - \sum_{p=1}^{2d+2} d_p \sigma\Big(\sum_{q=1}^{d} c_{pq} \sigma(w^{pq} \cdot x - \theta_{pq}) - \gamma_p \Big) \Big| < \epsilon \tag{3.6}$$

for all $x \in [0,1]^d$.

Proof. Let α be a positive number. We now partition the interval $[\alpha, +\infty)$ into segments $[\alpha, 2\alpha], [2\alpha, 3\alpha], \ldots$. Let h be a strictly increasing infinitely differentiable function on $[\alpha, +\infty)$ satisfying the following properties:

1) $0 < h(t) < 1$ for all $t \in [\alpha, +\infty)$,

2) $1 - h(\alpha) \leq \lambda$,

3) $h(t) \to 1$ as $t \to +\infty$.

From conditions 1) to 3), it follows that any function f satisfying the condition $h(t) < f(t) < 1$ for all $t \in [\alpha, +\infty)$ is strictly λ-increasing, and $f(t) \to 1$ as $t \to +\infty$.

We will now construct σ step by step to achieve the required properties. Let $\{u_n(t)\}_{n=1}^{\infty}$ be the sequence of all polynomials with rational coefficients over $[0,1]$. First, we define σ on the closed intervals $[(2m-1)\alpha, 2m\alpha]$, $m = 1, 2, \ldots$, as follows:

$$\sigma(t) = a_m + b_m u_m \left(\frac{t}{\alpha} - 2m + 1 \right), \quad t \in [(2m-1)\alpha, 2m\alpha], \tag{3.7}$$

or equivalently,

$$\sigma(\alpha t + (2m-1)\alpha) = a_m + b_m u_m(t), \quad t \in [0,1], \tag{3.8}$$

where a_m and $b_m \neq 0$ are arbitrarily chosen constants. These constants are chosen such that the inequality

$$h(t) < \sigma(t) < 1, \quad t \in [(2m-1)\alpha, 2m\alpha], \tag{3.9}$$

holds. The constants can be determined straightforwardly. Suppose

$$M = \max_{t\in\mathbb{J}} h(t), \quad A_1 = \min_{t\in\mathbb{J}} u_m\left(\frac{t}{\alpha} - 2m + 1\right), \quad A_2 = \max_{t\in\mathbb{J}} u_m\left(\frac{t}{\alpha} - 2m + 1\right)$$

for $\mathbb{J} := [(2m-1)\alpha, 2m\alpha]$. Here, $M < 1$. In the case where $A_1 = A_2$ (i.e., u_m is constant on $[0,1]$), we define $\sigma(t) = (M+1)/2$ and find corresponding pairs a_m and b_m. If $A_1 \neq A_2$ and $y = a + bx$, $b \neq 0$, is a linear function that maps the interval $[A_1, A_2]$ to $(m, 1)$, then we can simply define $a_m = a$ and $b_m = b$.

Next, we define σ on the interval $[2m\alpha, (2m+1)\alpha]$ $(m = 1, 2, \ldots)$ such that it lies in $C^\infty(\mathbb{R})$ and satisfies condition 3.9. Additionally, we define σ on $(-\infty, \alpha)$ such that the function remains in $C^\infty(\mathbb{R})$, strictly monotonically increases, and $\lim_{t\to-\infty} \sigma(t) = 0$. From the conditions on h and 3.9, we obtain that σ is strictly λ-increasing on $[\alpha, +\infty)$ and $\lim_{t\to+\infty} \sigma(t) = 1$.

From the construction of σ in 3.8, it follows that for each $m = 1, 2, \ldots$, there exist numbers A_m, B_m, and r_m such that

$$u_m(t) = A_m \sigma(\alpha t - r_m) - B_m, \tag{3.10}$$

with $A_m \neq 0$.

Now, let f be a continuous function over the unit cube $[0,1]^d$. According to Kolmogorov's superposition principle, we can represent f as in 3.4. For the continuous, univariate function g used in this representation, for every $\epsilon > 0$, there exists a polynomial $u_m(t)$ such that

$$|g(t) - u_m(t)| < \frac{\epsilon}{2(2d+1)} \tag{3.11}$$

for all $t \in [0,1]$. Together with 3.10, we obtain

$$|g(t) - a\sigma(\alpha \cdot \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r) - b| < \frac{\epsilon}{2(2d+1)} \tag{3.12}$$

for some numbers a, b, and $r \in \mathbb{R}$ and $t \in [0,1]$.

Substituting this result into Kolmogorov's superposition principle in 3.4, we obtain

$$\left| f(x_1, \ldots, x_d) - \sum_{p=1}^{2d+1} \left(a\sigma\Big(\alpha \cdot \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r\Big) - b\right)\right| < \frac{\epsilon}{2} \tag{3.13}$$

for all $(x_1, \ldots, x_d) \in [0,1]^d$.

For each $p \in \{1, 2, \ldots, 2d+1\}$ and $\delta > 0$, there exist constants a_p, b_p, and r_p such that

$$|\Phi_p(x_q) - [a_p \sigma(\alpha x_q - r_p) - b_p]| < \delta, \tag{3.14}$$

for all $x_q \in [0,1]$. With $\lambda_q > 0$, $q = 1, \dots, d$, $\sum_{q=1}^{d} \lambda_q = 1$, from 3.14 we obtain

$$\Big| \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - \Big[\sum_{q=1}^{d} \lambda_q a_p \sigma(\alpha x_q - r_p) - b_p \Big] \Big| < \delta \tag{3.15}$$

for all $x_q \in [0,1]^d$.

Since the function $a\sigma(\alpha t - r)$ is uniformly continuous on each closed interval, δ can be chosen arbitrarily small. From 3.15, we obtain

$$\Big| \sum_{p=1}^{2d+1} a\sigma\Big(\alpha \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r\Big) - \sum_{p=1}^{2d+1} a\sigma\Big(\alpha\Big[\sum_{q=1}^{d} \lambda_q a_p \sigma(\alpha x_q - r_p) - b_p \Big] - r\Big) \Big| < \frac{\epsilon}{2} \tag{3.16}$$

This can be rewritten as

$$\Big| \sum_{p=1}^{2d+1} a\sigma\Big(\alpha \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r\Big) - \sum_{p=1}^{2d+1} d_p \sigma\Big(\sum_{q=1}^{d} c_{pq} \sigma(w^{pq} \cdot x - \theta_{pq}) - \gamma_p \Big) \Big| < \frac{\epsilon}{2} \tag{3.17}$$

From 3.13 and 3.17, it follows that

$$\Big| f(x) - \Big[\sum_{p=1}^{2d+1} d_p \sigma\Big(\sum_{q=1}^{d} c_{pq} \sigma(w^{pq} \cdot x - \theta_{pq}) - \gamma_p \Big) - s \Big] \Big| < \epsilon \tag{3.18}$$

where $s = (2d+1)b$.

As s can now be written in the form

$$s = d\sigma\Big(\sum_{q=1}^{d} c_q \sigma(w^q \cdot x - \theta_q) - \gamma \Big),$$

this verifies the correctness of 3.6. □

So we have seen in this chapter that the structure and mathematics of ridge functions are closely related to neural networks. Not only can neural networks be described well with them, but we can also transfer the approximation properties of ridge functions to the feed forward model. This means that we can approximate functions arbitrarily well with feed forward models. The most important statements on this were provided by the theorems 3.3.2 and 3.6, which even give us a certain number of neurons that a feed forward model must at least have in the hidden layers in order to approximate a function arbitrarily well. The problem here is that these theorems only state that a sigmoidal activation function exists with which such an approximation is possible, not what it looks like. In the next chapter, we will address the question of this special activation function and discuss its properties.

4. Construction of a Special Activation Function

In the previous chapter, we showed that under certain conditions, any continuous function over the unit cube (or even any compact subset of $\mathbb{R}^d$ can be approximated arbitrarily well by a two-hidden-layer neural network of the form

$$\sum_{i=1}^{s} d_i \sigma \left(\sum_{j=1}^{r} c_{ij} \sigma(w^{ij} \cdot x - \theta_{ij}) - \gamma_i \right).$$

However, we have only shown the existence of an appropriate activation function for this purpose, not how it looks in practice. We address this now and aim to construct such a function based on the work of Vugar Ismailov in [16, pages 198-202].

4.1. Construction Algorithm

For the construction of the activation function, we will use insights from the previous chapter. Let $[a, b]$ be a closed interval and λ be any small positive real number, and let $d = b - a$ denote the length of the interval.

Step 1: We define the function

$$h(x) := 1 - \frac{\min\{1/2, \lambda\}}{1 + \log(x - d + 1)}. \tag{4.1}$$

This function is strictly monotonically increasing over $\mathbb{R}$ and satisfies the following properties:

1) $0 < h(x) < 1 \quad \forall x \in [d, +\infty)$,

2) $1 - h(d) \leq \lambda$,

3) $h(x) \to 1$ as $x \to +\infty$.

We now aim to construct σ such that

$$h(x) < \sigma(x) < 1 \tag{4.2}$$

for $x \in [d, +\infty)$. Thus, σ approaches 1 as $x \to +\infty$ and satisfies

$$|\sigma(x) - h(x)| \leq \lambda, \tag{4.3}$$

meaning it is a λ-increasing function.

Step 2: Before proceeding with the construction of σ, we need to number the normalized (or monic) polynomials with rational coefficients. To do this, we use the *Calkin-Wilf* sequence.

Definition 4.1.1 (Calkin-Wilf Sequence)**.** *The Calkin-Wilf sequence is a sequence of rational numbers used to enumerate all positive rational numbers by natural numbers. The recursion formula is given by*

$$q_1 = 1, q_{k+1} = \frac{1}{2\lfloor q_k \rfloor - q_k + 1}.$$

The initial elements of the sequence are thus $\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}, \frac{1}{4}, \ldots$

Let $\{q_k\}_{k=1}^{\infty}$ denote the *Calkin-Wilf* sequence. Then, we can number all rational numbers as follows:

$$r_0 := 0, \quad r_{2n} := q_n, \quad r_{2n-1} := -q_n, \quad n = 1, 2, \ldots$$

Thus, every monic polynomial with rational coefficients can be uniquely represented in the form $x^l + r_{k_{l-1}} x^{l-1} + \ldots + r_{k_0}$.

Moreover, every positive rational number can be uniquely represented in the form of a continued fraction:

$$[m_0; m_1, ..., m_l] = m_0 + \cfrac{1}{m_1 + \cfrac{1}{m_2 + \cfrac{1}{\ddots + \cfrac{1}{m_l}}}} \tag{4.4}$$

Here, $m_0 \geq 0$, $m_1, ..., m_{l-1} \geq 1$, and $m_l \geq 2$. Now, we construct a bijection between all monic polynomials with rational coefficients and positive real numbers as follows:
Let $u_1(x) := 1$,

$$u_n(x) := r_{q_n - 2} + x$$

for $q_n \in \mathbb{Z}$,

$$u_n(x) := r_{m_0} + r_{m_1 - 2} x + x^2$$

for $q_n = [m_0; m_1]$, and

$$u_n(x) := r_{m_0} + r_{m_1 - 1} x + ... + r_{m_{l-2} - 1} x^{l-2} + r_{m_{l-1} - 2} x^{l-1} + x^l$$

for $q_n = [m_0; m_1, ..., m_l]$ with $l \geq 3$. Thus, the initial elements of the sequence are

$$1, \ x^2, \ x, \ x^2 - x, \ x^2 - 1, \ x^3, \ x - 1, \ x^2 + x, ...$$

Step 3: We will now construct σ first on the intervals $[(2n-1)d, 2nd]$, $n = 1, 2, ...$ Additionally, for each monic polynomial $u_n(x) = \alpha_0 + \alpha_1 x + ... + \alpha_{l-1} x^{l-1} + x^l$, we define the numbers

$$B_1 := \alpha_0 + \frac{\alpha_1 - |\alpha_1|}{2} + ... + \frac{\alpha_{l-1} - |\alpha_{l-1}|}{2}$$

and

$$B_2 := \alpha_0 + \frac{\alpha_1 + |\alpha_1|}{2} + ... + \frac{\alpha_{l-1} + |\alpha_{l-1}|}{2} + 1,$$

and the sequence

$$M_n := h((2n+1)d), \ n = 1, 2, ...$$

This sequence is strictly increasing and converges to 1.
Now, we define σ as follows:

$$\sigma(x) := a_n + b_n u_n(\frac{x}{d} - 2n + 1), \ \ x \in [(2n-1)d, 2nd] \tag{4.5}$$

where

$$a_1 := \frac{1}{2}, \quad b_1 = \frac{h(3d)}{2} \tag{4.6}$$

and

$$a_n := \frac{(1+2M_n)B_2 - (2+M_n)B_1}{3(B_2 - B_1)}, \quad b_n = \frac{1-M_n}{3(B_2-B_1)}, \ \ n = 2, 3, ... \tag{4.7}$$

The numbers a_n and b_n have the property that for $n > 2$, they are coefficients of the linear functions $y = a_n + b_n x$, which map the interval $[B_1, B_2]$ to the interval $[(1+2M_n)/3, (2+M_n)/3]$. Additionally, for $n = 1$, the function over the interval $[d, 2d]$ is defined as

$$\sigma(x) = \frac{1+M_1}{2}$$

Thus, we obtain

$$h(x) < M_n < \frac{1+2M_n}{3} \leq \sigma(x) \leq \frac{2+M_n}{3} < 1 \tag{4.8}$$

for all $x \in [(2n-1)d, 2nd]$, $n = 1, 2, ...$
Step 4: In this step, we construct the function on the interval $[2nd, (2n+1)d]$, $n = 1, 2, ...$ For this, we require the smooth-transition function. This is defined as

$$\beta_{a,b}(x) := \frac{\hat{\beta}(b-x)}{\hat{\beta}(b-x) + \hat{\beta}(x-a)},$$

where

$$\hat{\beta}(x) := \begin{cases} e^{-1/x}, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

Here, $\beta_{a,b}(x) = 1$ for $x \leq a$, $\beta_{a,b}(x) = 0$ for $x \geq b$, and $0 < \beta_{a,b}(x) < 1$ for $x \in (a, b)$.
Now, we define

$$K_n := \frac{\sigma(2nd) + \sigma((2n+1)d)}{2}, \ \ n = 1, 2, ... \tag{4.9}$$

It should be noted that the function values $\sigma(2nd)$ and $\sigma((2n+1)d)$ were defined in the previous step, and since these values lie in the interval $(M_n, 1)$, it follows that $K_n \in (M_n, 1)$.
Initially, we smooth σ onto the interval $[2nd, 2nd + d/2]$. For this, let $\epsilon := (1 - M_n)/6$. Furthermore, we choose $\delta \leq d/2$ such that

$$\left| a_n + b_n u_n(\frac{x}{d} - 2n + 1) - (a_n + b_n u_n(1)) \right| \leq \epsilon, \ \ x \in [2d, 2d + d/2] \tag{4.10}$$

δ can be chosen as

$$\delta := \min\left\{ \frac{\epsilon d}{b_n C}, \frac{d}{2} \right\}$$

where $C > 0$ satisfies the inequality $|u_n'(x)|$, $x \in (1,\ 1,5)$. Now, we define σ on $[2nd, 2nd + d/2]$ as

$$\sigma(x) := K_n - \beta_{2nd,2nd+\delta}(x) \cdot \Big(K_n - a_n - b_n u_n(\frac{x}{d} - 2n + 1) \Big), \ x \in [2nd, 2nd + d/2] \tag{4.11}$$

This function satisfies Condition 4.2:
For the case $x \in [2nd + \delta, 2nd + d/2]$, $\sigma(x) = K_n \in (M_n, 1)$, so there's nothing more to prove.
For $x \in [2nd, 2nd + \delta)$, $0 < \beta_{2nd,2nd+\delta}(x) \leq 1$. From 4.11, it follows that for every $x \in$

$[2nd, 2nd+\delta)$, $\sigma(x)$ lies between K_n and $A_n(x) := a_n + b_n u_n(\frac{x}{d} - 2n + 1)$. With 4.10, we also obtain

$$a_n + b_n u_n(1) - \epsilon \leq A_n \leq a_n + b_n u_n(1) + \epsilon \tag{4.12}$$

Together with 4.5 and 4.8, this gives us $A_n(x) \in \left[\frac{1+2M_n}{3} - \epsilon, \frac{2+M_n}{3} + \epsilon\right]$ for $x \in [2nd, 2nd+\delta)$. With $\epsilon = (1-M_n)/6$, it follows that $A_n(x) \in (M_n, 1)$. Now, since both K_n and $A_n(x)$ belong to $(M_n, 1)$, we have

$$h(x) < M_n < \sigma(x) < 1, \text{ for } x \in [2nd, 2nd + d/2].$$

The second half of the function is constructed similarly:

$$\sigma(x) := K_n - (1 - \beta_{(2n+1)d-\overline{\delta},(2n+1)d}(x)) \cdot \Big(K_n - a_{n+1} - b_{n+1}u_{n+1}(\frac{x}{d} - 2n - 1)\Big), x \in [2nd+d/2, (2n+1)d]$$

with

$$\delta := \min\Big\{\frac{\overline{\epsilon} d}{b_{n+1}\overline{C}}, \frac{d}{2}\Big\}, \quad \overline{\epsilon} := \frac{1 - M_{n+1}}{6}, \quad \overline{C} \geq \sup_{[-0.5,0]} |U'_{n+1}(x)|.$$

Thus, σ satisfies Condition 3.9 on $[2nd + d/2, 2nd + d]$ and

$$\sigma\Big(2nd + \frac{d}{2}\Big) = K_n, \quad \sigma^{(i)}\Big(2nd + \frac{d}{2}\Big) = 0, \; i = 1, 2, ...$$

Steps 3 and 4 construct σ on $[d, +\infty)$.
Step 5: On the remaining interval $(-\infty, d)$, we define the function as

$$\sigma(x) := \Big(1 - \hat{\beta}(d - x)\Big)\frac{1 + M_1}{2}, \; x \in (-\infty, d)$$

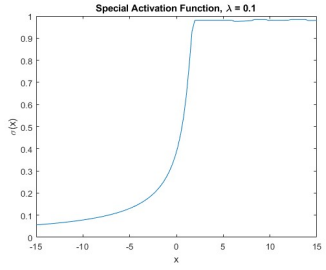
It can be easily seen that σ strictly increases and is smooth on $(-\infty, d)$.
Thus, the activation function is completely constructed.

4.2. Properties of the Activation Function

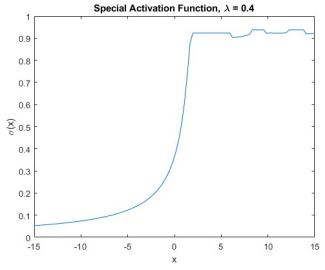
The activation function constructed above possesses the following properties, shown in [16]:

1) σ is a sigmoid function.
2) $\sigma \in C^\infty(\mathbb{R})$.
3) σ is strictly monotonically increasing on $(-\infty, d)$ and strictly λ-increasing on $[d, +\infty)$.

We see the plot of the special activation function in figure **??**.



(a) Plot of σ for $\lambda = 0.1$



(b) Plot of σ for $\lambda = 0.4$

Please note that the function was calculated numerically, which leads to irregularities in the representation, which do not correspond to the good properties from above.
Besides the approximation properties for a 2-hidden-layer neural network, the special activation function is also used in another important theorem:

Theorem 4.2.1. *[16] Let f be a continuous function over a compact interval $[a, b] \subset \mathbb{R}$, and let σ be the activation function constructed in 4.1. Then, for any arbitrarily small $\epsilon > 0$ and constants $c_1, c_2, \theta_1, \theta_2$:*

$$|f(x) - c_1\sigma(x - \theta_1) - c_2\sigma(x - \theta_2)| < \epsilon$$

for all $x \in [a, b]$.

Proof. Let $d := b - a$. We divide the interval $[d, +\infty)$ into segments $[d, 2d], [2d, 3d], \ldots$. From 4.5, it follows that

$$\sigma(dx + (2n - 1)d) = a_n + b_n u_n(x), \quad x \in [0, 1] \tag{4.13}$$

for $n = 1, 2, \ldots$. Here, a_n and b_n are computed according to 4.6 and 4.7, respectively, for $n = 1$ and $n > 1$.

From 4.13, it follows for each $n = 1, 2, \ldots$

$$u_n(x) = \frac{1}{b_n}\sigma(dx + (2n - 1)d)\frac{a_n}{b_n} \tag{4.14}$$

Now, g is an arbitrary continuous function on the interval $[0, 1]$. The set of polynomials with rational coefficients is dense in the space of continuous functions over a compact subset of $\mathbb{R}$. This implies that for every $\epsilon > 0$, there exists such a polynomial p such that

$$|g(x) - p(x)| < \epsilon$$

for all $x \in [0, 1]$. Let p_0 denote the leading coefficient of p. If $p_0 \neq 0$ (or $p \not\equiv 0$), we define $u_n := p(x)/p_0$; otherwise, $u_n := 1$. In both cases,

$$|g(x) - p_0 u_n(x)| < \epsilon$$

Together with 4.14, this means

$$|g(x) - c_1\sigma(dx - s_1) - c_0| < \epsilon$$

for certain $c_0, c_1, s_1 \in \mathbb{R}$ and all $x \in [0, 1]$. Specifically, $c_1 = p_0/b_n$, $s_1 = d - 2nd$, and $c_0 = p_0 a_n/b_n$. Thus, we can also write $c_0 = c_2(dx - s_1)$, where $c_2 := 2c_0/(1 + h(3d))$ and $s_2 := -d$. Therefore,

$$|g(x) - c_1\sigma(dx - s_1) - c_2\sigma(dx - s_2)| < \epsilon \tag{4.15}$$

This statement holds on the unit interval $[0, 1]$. Through appropriate linear transformations, it can be extended to any compact interval $[a, b]$. Thus, let $f \in C[a, b]$, σ constructed as above, and $\epsilon > 0$ arbitrarily small. The transformed function $g(x) = f(a + (b - a)x)$ is well-defined over $[0, 1]$, and we can apply 4.15. Now, using the variable transformation $x = (t - a)/(b - a)$, we can write

$$|f(t) - c_1\sigma(t - \theta_1) - c_2\sigma(t - \theta_2)| < \epsilon$$

for all $t \in [a, b]$, where $\theta_1 = a + s_1$ and $\theta_2 = a + s_2$. Thus, the proof is complete. □

4.3. Application of the special activation function for function approximation

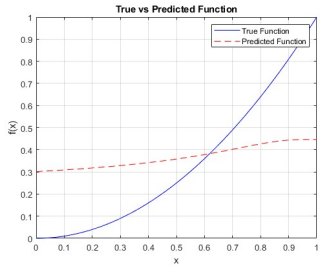
Now we will apply the won knowledge to a real problem, in this case the approximation of continuous functions on $[0,1]^d$. We will restrict ourselves here to the cases $d = 1$ and $d = 2$. The aim is to show that the function approximation using this minimal network also succeeds in the application. We only want to look at the results of the approximation and leave out a performance comparison with other approximation methods, as the computing time alone is quite high due to the complexity of the activation function. However, we will show that the above approximation theorem is valid in real neural networks for functions over $[0,1]^d$.
First, let's look at the setup in which we work on this problem. In this and all other applications of this work, we use Matlab version R2023b and the Deep Learning Toolkit for the construction of neural networks. In our current example, the activation function is integrated in a custom layer, in front of which so-called fully connected layers are built, which contain the weights and biases. Since we are dealing here with the approximation of functions over $[0,1]$ and $[0,1]^2$, we need one or two input neurons and one output neuron. Here we see the setup for the network in Matlab code:

```
layers = [
    featureInputLayer(1)
    fullyConnectedLayer(d)
    CustomSigmaLayer('CustomSigma1')
    fullyConnectedLayer(2*d+2)
    CustomSigmaLayer('CustomSigma2')
    fullyConnectedLayer(1)
    regressionLayer
];
```

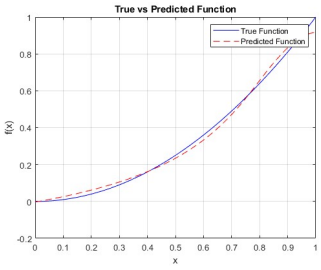
In our first example, we want to approximate the function $f : [0,1] \to \mathbb{R}$, $x \mapsto f(x) = x^2$. To do this, we use the layer setup above and the following options for the neural network:

```
options = trainingOptions('adam', ...
    'MaxEpochs', 1000, ... % Increase epochs
    'InitialLearnRate', 0.001, ... % Adjust learning rate
    'GradientThreshold', Inf, ...
    'MiniBatchSize', 50, ...
    'Verbose', true, ...
    'Plots', 'training-progress');
];
```

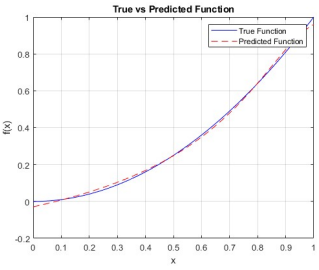
With this setup we obtain the following approximation and learning curve:



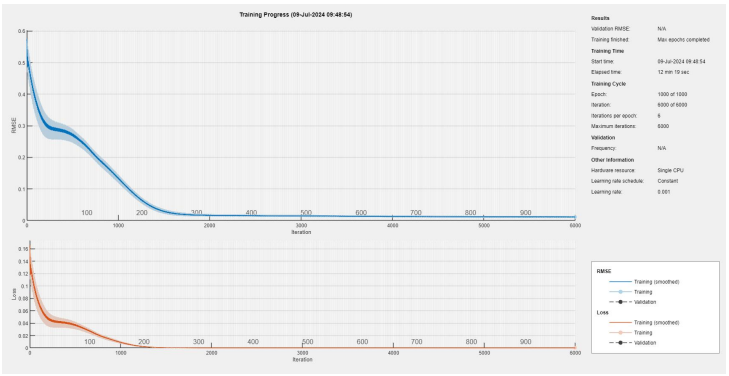
(a) 200 Epochs



(b) 600 Epochs



(c) 1000 Epochs



(d) Training progress within 1000 Epochs

Figure 4.2.: Function approximation of $f(x) = x^2$ on $[0, 1]$

We see that we can get approximation of any accuracy also in real neural networks. The error function used in this example is the Rooted Mean Squared Error (RMSE). It is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

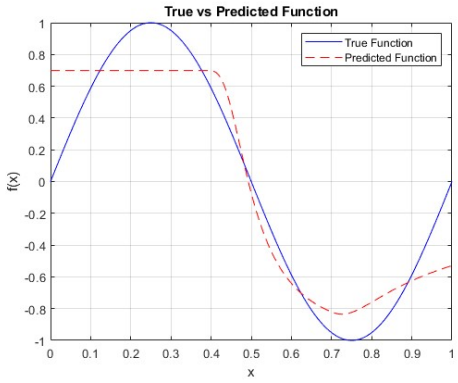
where:

- n is the number of data points,
- y_i is the observed value,
- $\hat{y}_i$ is the predicted value.

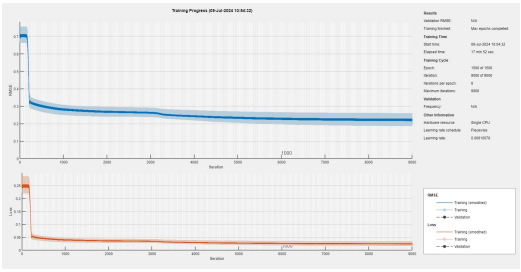
The advantage we see is that we can approximate the function arbitrarily well with a relatively small network (in this case only 5 hidden neurons). The disadvantage, like said in the beginning of the chapter, is that learning takes quite a long time in contrast to conventional architectures. Nevertheless, the correctness of the above theorems could thus be demonstrated. Now we want to approximate a somewhat more complicated function, namely $f(x) = \sin(2\pi x)$. For this we have the following training options:

```
options = trainingOptions('adam', ...
    'MaxEpochs', 1500, ... % Increase epochs
    'InitialLearnRate', 0.01, ... % Adjust learning rate
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.6, ...
    'LearnRateDropPeriod', 150, ...
    'GradientThreshold', 1.5, ...
    'MiniBatchSize', 50, ...
    'Verbose', true, ...
    'Plots', 'training-progress');
];
```

Because of some numerical issues we have to decrease the learn rate over time and apply a gradient threshold. After the learning we get the following approximation:



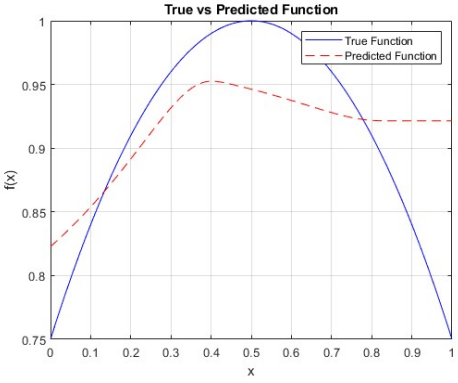
(a) Function Approximation



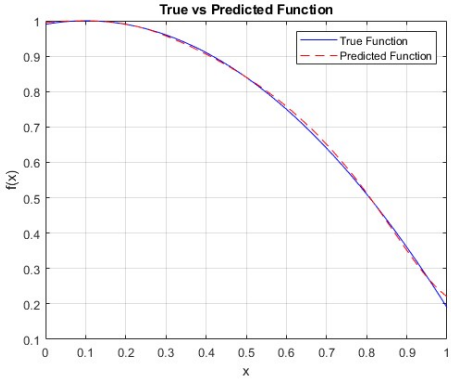
(b) Learning Progress

Figure 4.3.: Function approximation of $f(x) = \sin(2\pi x)$ on $[0, 1]$

We see, that we also get a good approximation, but it is not as accurate as the first example, even after 1500 epochs. So a slightly more complex function leads to big performance problems, but the theorem still works.
However, there are functions for which the approximation by this network does not work so well. An example of this is $-(x - 0.5)^2 + 1$ on the interval $[0, 1]$:



(a) Bad Function Approximation



(b) Successful Function Approximation

Figure 4.4.: Function approximation depending on translation

With the interval-symmetric variant, the network adjusts to a certain configuration after a certain time, from which it no longer deviates. Mathematically speaking, a local minimum is

reached there for the loss function in terms of the weights, which is no longer left. We can see the result at 4.4a. However, if we shift the function a little bit less, we obtain an arbitrarily good approximation again through the network, as can be seen at 4.4b.
Now we want to apply all of this to $\mathbb{R}^2 \to \mathbb{R}$ functions. For this, we need to make small changes to the network. First, we now have two input neurons because of the two dimensional input. Second, our number of hidden neurons changes depending on the dimension, in this case $d = 2$. First we want to approximate the relatively simple function $(x + y)^2$. The outcome after 450 epochs is the following:

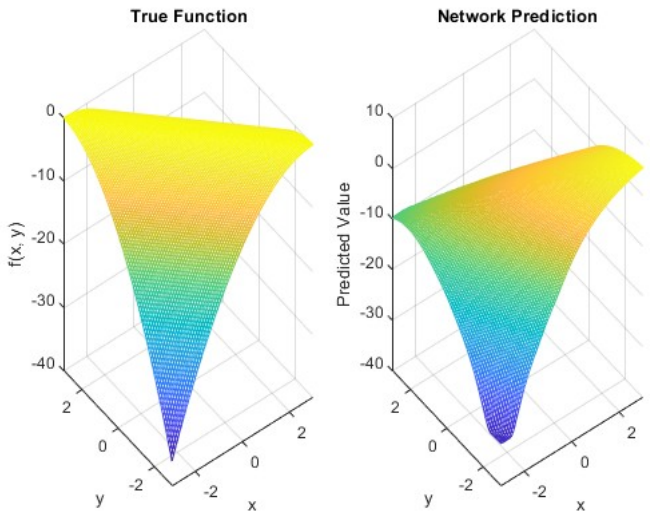
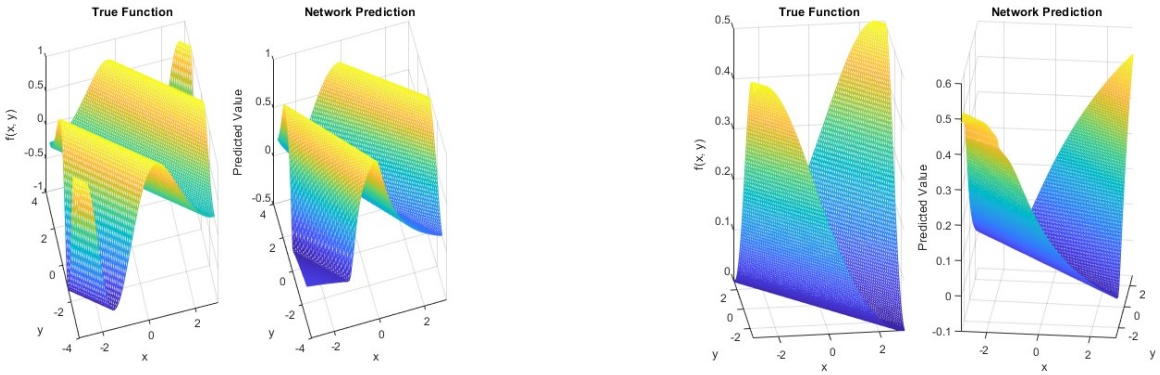


Figure 4.5.: Approximation of $f(x, y) = (x + y)^2$

The result is not as accurate as wanted, but here too the network quickly loses itself in local minima. But after all it is good enough for presentation. Here are two more functions we want to approximate. Here we use our first function in a scaled sine function:



(a) Approx. $f(x, y) = \sin(0.25 * (x + y)^2)$

(b) Approx. $f(x, y) = 0.5 * \sin(0.05 * (x + y)^2)$

Figure 4.6.: Function approximation of sine functions over $\mathbb{R}^2$

Now the approximations looks a bit better as in the first example.
What we have thus shown is that the modification of Kolmogorov's approximation theorem is indeed confirmed in real neural networks. However, the knowledge of the existence of parameters that optimize the neural network with respect to function approximations is no guarantee that these parameters will be found during learning, as we have shown using the example of 4.4a. Furthermore, due to the complex activation function, a longer computing time must be planned than would be the case with conventional networks. The only advantage of the approach shown so far is the very small number of neurons in the network. Another problem is, that in more complex function examples, the network often gets stucked in local minima, which could be solved by special weight initializations, learn rate adjustments or choosing different optimizers, but it is hard to achieve good results in such cases.

5. Ridgelets and Ridgelet Filtering

Now we want to go one step further and combine the theory of ridge functions with wavelets. Wavelets are a special class of functions with a characteristic wave-like character, certain decay behavior and finite energy. The study of wavelets is relatively new in mathematical history and is mainly used in signal analysis. In the 1980s, they were first used in a generalization of the short time Fourier transform, which reveals the frequency behaviour of a signal at certain points in time. They can also be used to implement the wavelet transform, which we will discuss later. The theory of wavelets can thus be combined very well with that of ridge functions because, as we will see later, ridge functions with a wavelet profile are wonderfully suitable for edge detection in images. To do this, it is necessary to understand those edges as signals along lines whose frequencies are analyzed by a wavelet. But before we get to that, let us look at the fundamentals of wavelet and ridgelet theory.

5.1. Ridgelets

After exploring the theory of ridge functions and their approximation properties for arbitrary functions, we now turn to another approach that combines wavelet theory with ridge functions, known as ridgelets. To delve into this, we first need to build upon the theory of wavelets.

5.1.1. Fundamentals

Wavelets are a specific class of functions primarily used in signal processing within the wavelet transform, a specialized form of the short-time Fourier transform enabling time-frequency analysis of a signal.

Definition 5.1.1. *A function $\psi \in L^2(\mathbb{R})$ is called a wavelet if it satisfies the following admissibility condition:*

$$\int_{-\infty}^{\infty} \frac{|\hat{\psi}(\omega)|}{|\omega|} d\omega < \infty$$

where $\hat{\psi}$ denotes the Fourier transform of ψ.
Additionally, for a wavelet ψ the following conditions must hold:

$$\hat{\psi}(0) = 0$$

and

$$\int_{-\infty}^{\infty} \psi(t)\, dt = 0$$

With wavelets, we can define the continuous wavelet transform.

Definition 5.1.2. *Let $\psi \in L^2(\mathbb{R})$ be an admissible wavelet. Then, the wavelet transform $\mathcal{W}(f)$ for a function $f \in L^2(\mathbb{R})$ and parameters $a, b \in \mathbb{R}$ is defined as:*

$$\mathcal{W}_\psi f(a,b) := \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} f(t) \overline{\psi\left(\frac{t-b}{a}\right)} dt$$

Here, a is the scaling parameter and b is the translation parameter. The derived wavelet family from the mother wavelet,

$$\psi_{ab} = \frac{1}{a}\psi\left(\frac{t-b}{a}\right)$$

allows the transformation to be expressed as an inner product:

$$\mathcal{W}_\psi f(a,b) = \langle f, \psi_{ab} \rangle$$

The theory of wavelets is extensive, but for our purposes, this brief definition of wavelets and the wavelet transform suffices.

Another important concept is that of frames, which we will need later.

Definition 5.1.3 (Frames)**.** *Let $\mathcal{H}$ be a separable Hilbert space with inner product $\langle \cdot, \cdot \rangle$ and induced norm $|| \cdot || = \sqrt{\langle \cdot, \cdot \rangle}$. A sequence $\{x_k\}_{k \in \mathbb{Z}} \subset \mathcal{H}$ is called a frame if there exist constants $A, B > 0$ such that for all $y \in \mathcal{H}$,*

$$A||y||^2 \leq \sum_{k \in \mathbb{Z}} |\langle y, x_k \rangle|^2 \leq B||y||^2$$

Here, A and B are the frame bounds. A frame is called tight if $A = B$.

5.1.2. Ridgelet Applications in Function Approximation

As we saw in the last chapters, ridge functions are very suitable for describing the approximation of certain functions by artificial neural networks. However, it has been shown that not only simple ridge functions are suitable for such tasks, but also the ridgelets described earlier, which we will revisit here. As a reminder:

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a function such that it fulfills the admissibility condition

$$K_\psi = \int_{\mathbb{R}} \frac{|\hat{\psi}(\xi)|^2}{|\xi|^d} d\xi < \infty.$$

Then we call the functions $\psi_\tau = a^{-1/2}\psi\left(\frac{u \cdot x - b}{a}\right)$, derived from ψ, *ridgelets* or the family of ridgelets derived from ψ. Here, τ belongs to the set of neurons $\Gamma = \{\tau = (a, u, b) : a, b \in \mathbb{R}, a > 0, u \in \mathcal{S}^{d-1}, ||u|| = 1\}$. Here, a indicates the scaling, u the direction, and b the position of the corresponding ridgelet. Additionally, we define the surface area of the d-dimensional unit sphere $\mathcal{S}^{d-1}$ as σ_d. Then we can define the measure over $d\tau$ as $\mu(d\tau) = \sigma_d \frac{da}{a^{d+1}} dudb$. Thus, any multi-dimensional function $f \in L^1 \cap L^2(\mathbb{R}^d)$ can be described as a superposition of ridgelets, like it is shown in [54]

$$f = c_\psi \int \langle f, \psi_\tau \rangle \psi_\tau \mu(d\tau) \tag{5.1}$$

$$= \pi(2\pi)^{-d} K_\psi^{-1} \int \langle f, \psi_\tau \rangle \psi_\tau \sigma_d \frac{da}{a^{d+1}} dudb \tag{5.2}$$

In neural networks, the central goal is to approximate a function $Y = f(x) : \mathbb{R}^d \to \mathbb{R}^m$. This is achieved by approximating m mappings $y_i = f_i(x) : \mathbb{R}^d \to \mathbb{R}^m$. With the corresponding ridgelet approximation, we obtain

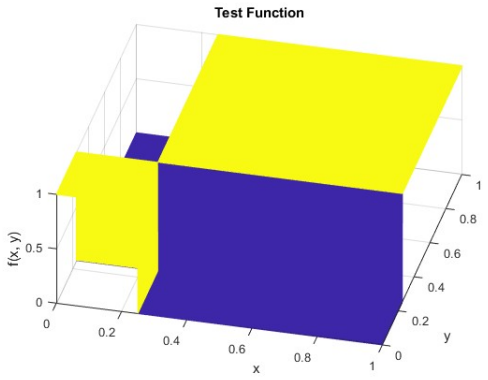
$$\tilde{y}_i = \sum_{j=1}^{N} c_{ij} \psi_\tau \left(\frac{u_j \cdot x - b_j}{a_j} \right) \tag{5.3}$$

where $\tilde{Y} = (\tilde{y}_1, \ldots, \tilde{y}_m)$, $||u_j|| = 1$, $u_j, x \in \mathbb{R}^d$. Here, c_{ij} is the corresponding ridgelet coefficient. If we use ridgelets as activation functions in a neural network, this is the output we get in one layer. So we can use ridgelets to approximate any given of the space $L^1 \cap L^2(\mathbb{R}^d)$ in neural networks. This way of using ridgelets does not only bring us pure function approximation. It is particularly suitable for approximating functions with d-1 dimensional singularities, which we will look at in more detail in the next section.

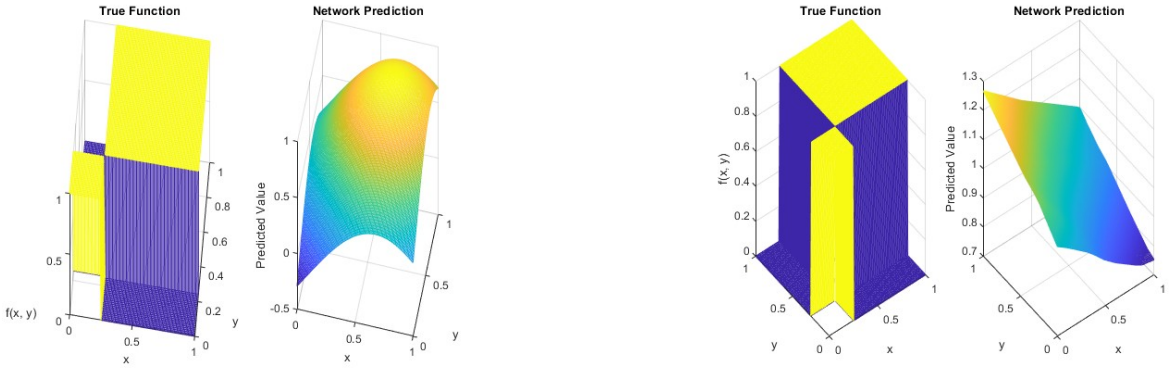
5.1.3. Application example for Ridgelet Neural Networks

In this section, we will look at what ridgelet neural networks can be used for. According to the work of Dr. Jiao Licheng [53], these networks are particularly suitable for approximating functions with (d-1)-dimensional singularities and other discontinuous functions. In application, this means that they are suitable for approximating functions with large gradients and for pattern recognition in images. To test our network, we will approximate the function

$$f(x, y) = \begin{cases} 1 & (x - 1/4)(y - 1/4) > 0 \\ 0 & \text{else} \end{cases}$$



First we will try to approximate the function by a pure ridgelet network. For this we use 30 neurons in the hidden layer. We see the result in figure 5.1.

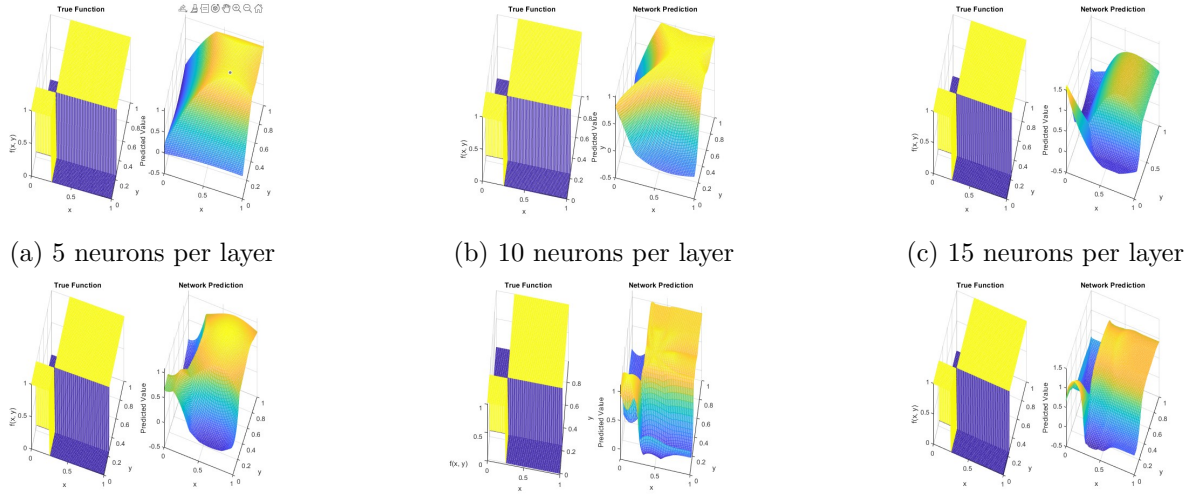


(a) Approximation with ridgelet layer

(b) Approximation with relu layer

Figure 5.1.: Approximation of a discontinous function over $[0, 1]^2$

A certain similarity could be assumed in figure 5.1a, but the resulting approximation is not very good, to say the least. On the other hand, the network does not perform any better if we only equip it with the ReLu activation function. One could therefore speculate that the ridgelet networks are not so well suited to discontinuous functions after all. This is not quite true, as it has been shown that these networks unfold their full potential especially in combination with other network architectures and layers. We will now see what happens when we combine the ridgelet and ReLu networks. For this, we just put a fully connected layer with the ReLu activation function and the same number of neurons as the ridgelet layer behind the ridgelet layer. The results with different numbers of neurons and epochs we see in figure 5.2:



(a) 5 neurons per layer

(b) 10 neurons per layer

(c) 15 neurons per layer

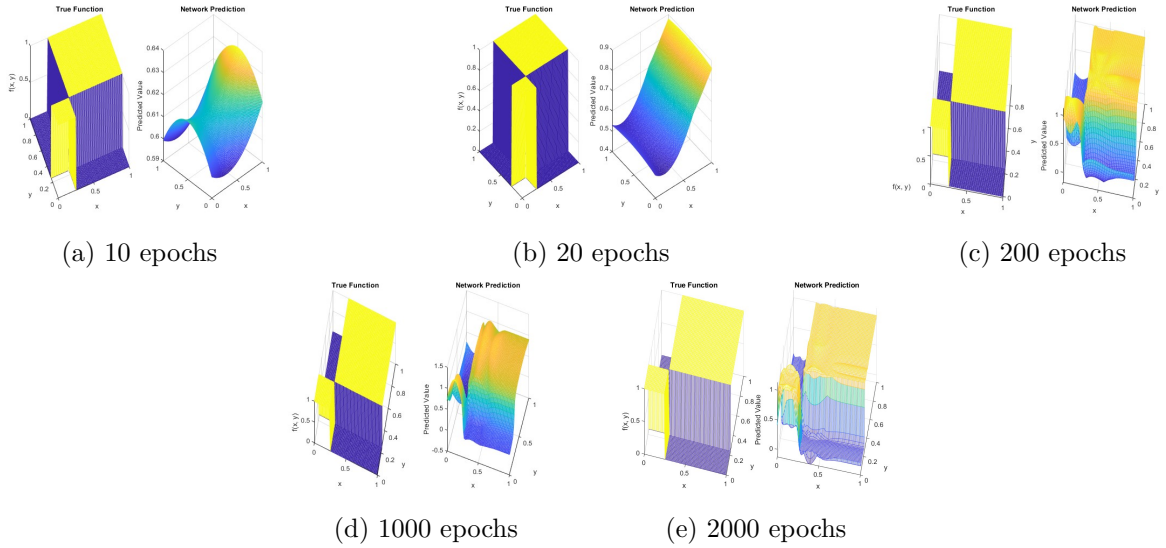
(d) 20 neurons per layer

(e) 30 neurons per layer

(f) 40 neurons per layer

Figure 5.2.: Function Approximation with different neuron number after 200 epochs

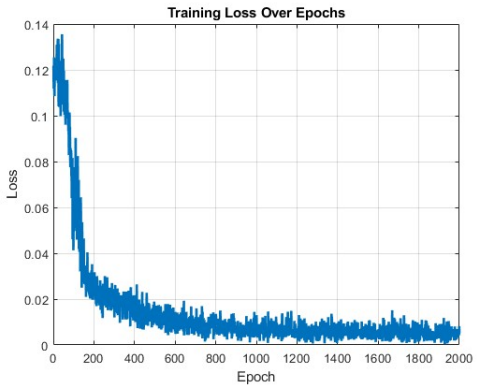
We see, that we not only get a good approximation of the function, but the accuracy also rises with the number of neurons, reaching a good peak at 30 neurons with a MSE of ≈ 0.021. After this, the loss will not get any better. So now we fix the number of neurons at 30 and look, how the accuracy develops over the epochs:
Here, as expected, we see an increase in accuracy across the epochs. However, we need at least about 200 epochs to achieve satisfactory accuracy. The following graph shows the development of the loss function over the epochs:



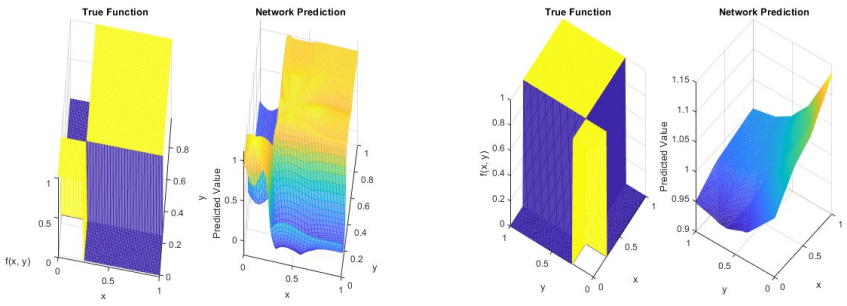
(a) 10 epochs (b) 20 epochs (c) 200 epochs

(d) 1000 epochs (e) 2000 epochs

Figure 5.3.: Function Approximation with 30 Neurons



In this graph we can even see that it takes about 1000 epochs for the loss function to stabilize. As we can see, ridgelet networks are very well suited for approximating such functions. This is particularly evident in comparison with other conventional networks of a comparable size. The following graphic shows the approximation of the discontinuous function, once with the ridgelet network with 30 neurons and once with a two hidden layer ReLu network with 30 neurons in each of the hidden layers:



(a) Ridgelet Network (b) Relu Network

Figure 5.4.: Function Approximation with different Networks

We see that the ReLu network performs much worse compared to the ridgelet network. This means that, in contrast to conventional networks, ridgelet networks are particularly effective for functions with singularities and discontinuities.

5.2. The Ridgelet Transform

Having described the basics of wavelets and the wavelet transform, we now move on to the ridgelet transform. Ridgelets are mainly used for recognizing lines and edges in mathematical image processing. This concept is particularly effective when we apply it to image classification in convolutional neural networks, which we will discuss in later chapters. For now, we will focus on the construction of the ridgelet transform with respect to the goal of edge detection.
If we have an image that we want to analyze with respect to lines and edges is the use of the Radon Transform on this image. If we interpret the image as a function $f : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto f(x, y)$, then the Radon Transform shows the integral or energy of f over a given line in the $\mathbb{R}^2$ plane. The parametrization of such a line can be written as

$$(x(t), y(t)) = \big((s \sin\theta + t\cos\theta), (s\sin\theta - t\cos\theta)\big)$$

Here, θ is the angle of the direction vector along the line with $\theta \in [0, \pi)$ and s is the distance of the line from the point $(0, 0)$. So the vector (θ, s) can be interpreted as the unique coordinate of every line in the $\mathbb{R}^2$ plane. We should note that there are different definitions of the line depending on the literature: some use the direction vector of the line, others the normal vector. We will use the definition with the direction vector. This given, we can define the radon transform.

Definition 5.2.1 (Radon Transform)**.** *Let $f \in C(\mathbb{R}^2)$ and let f fulfill the following conditions:*

1) $\int_{\mathbb{R}^2} \frac{|f(x,y)|}{\sqrt{x^2+y^2}}\, d(x, y) < \infty$

2) $\lim_{t\to\infty} \int_0^{2\pi} f(x + r\cos\phi, y + r\sin\phi)\, d\phi = 0$

Then the Radon Transform of f is defined as

$$\begin{aligned} R_f(\theta, s) &:= \int_{-\infty}^{\infty} f(x(t), y(t))\, dt \\ &= \int_{-\infty}^{\infty} f((s\sin\theta + t\cos\theta), (s\sin\theta - t\cos\theta))\, dt \end{aligned}$$

The data we get from our image or function after the Radon Transform is called sinogram. In figure 5.5 we see the sinogram of the famous cameraman picture.

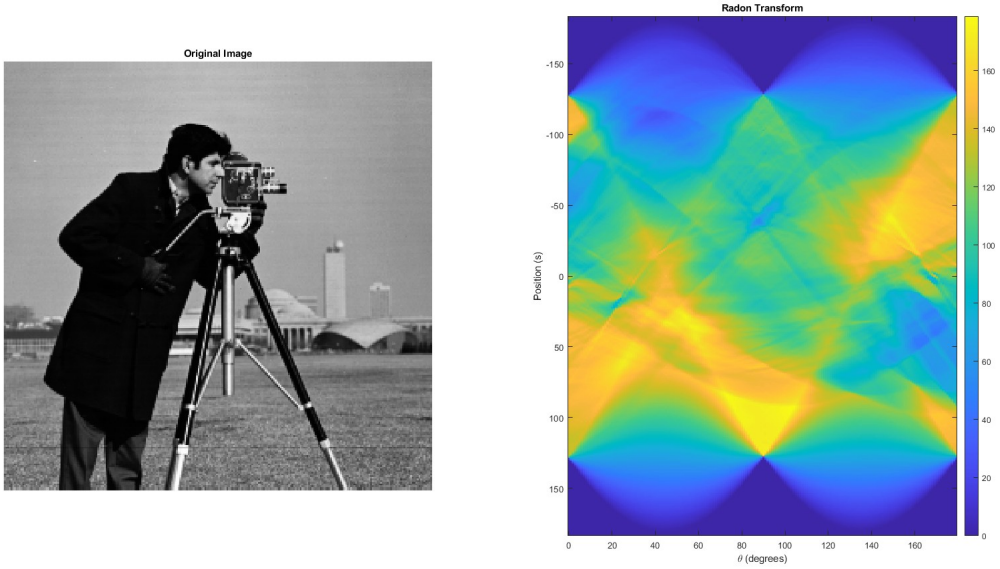


Figure 5.5.: Image of the cameraman with sinogram

What we see at each point are the summed up values over the different lines defined by the distance s and the angle θ. For example, there is a bright peak at $\theta \approx 90°$ and $s \approx 125$. This is the vertical line that goes through the leg of the cameraman. There is also a peak at $\theta \approx 60°$ and $s \approx 25$, which is the line that goes through the cameraman. To see the effects of the Radon Transform better, we take a look at a constructed image, which is black at the main diagonal line and gets brighter to the corners.

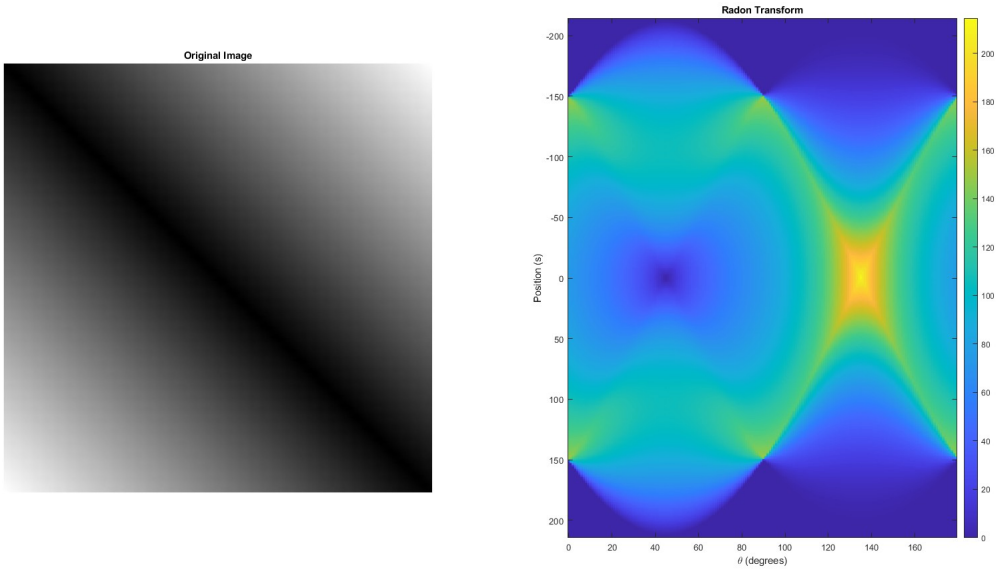


Figure 5.6.: Example image with sinogram

We see the main peak at 135° and distance zero, because this represents the summed up values on the main diagonal line with the highest values. This fades away if we rise the distance. On the other hand there is a minimum at the same distance at 45°, because here are the smallest

values summed up.
So with the Radon Transform we get information about over which lines an image has the highest values. This gives us some special information but does not help to find edges. For this, we need to find changes or strong oszillations along the distance at the same angle. In other words, we need to find at which angles along the distance s stronger or weaker frequencies does appear, because this gives us information whether there are edges perpendicular to this line or not.
To give us such information, we use the wavelet transform introduced in 5.1.2 along the s-axis.

$$\mathcal{W}_\psi R_f(\theta, a, b) := \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} R_f(\theta, s)\psi\left(\frac{s-b}{a}\right) ds \tag{5.4}$$

Now we can substitute $s = x_1 \cos\theta + x_2 \sin\theta$. With this, we put the Integration over the different lines inside the wavelet definition and can rewrite the double integral as

$$\mathcal{R}_f(a, b, \theta) = \int_{\mathbb{R}^2} f(x)\psi_{a,b,\theta}(x)\, dx$$

With this, we get the full definition of the ridgelet transform.

Definition 5.2.2. *Let $\psi : \mathbb{R} \to \mathbb{R}$ be a smooth, univariate function with rapid decay and vanishing mean, i.e., $\int_{\mathbb{R}} \psi(\lambda)\, d\lambda = 0$. For each $a > 0$, $b \in \mathbb{R}$, and $\theta \in [0, 2\pi)$, we define the bivariate function $\psi_{a,b,\theta} : \mathbb{R}^2 \to \mathbb{R}$ by*

$$\psi_{a,b,\theta}(x) = \frac{1}{\sqrt{a}} \cdot \psi\left(\frac{\cos(\theta)x_1 + \sin(\theta)x_2 - b}{a}\right)$$

This function represents a ridge function with $(\cos(\theta)x_1 + \sin(\theta)x_2 = const.)$, where the profile is a wavelet. For a bivariate function f, the continuous ridgelet transform is defined as:

$$\mathcal{R}_f(a, b, \theta) = \int_{\mathbb{R}^2} f(x)\overline{\psi}_{a,b,\theta}(x)\, dx$$

The conditions ensure that $K_\psi := \int_{\mathbb{R}} \frac{|\hat{\psi}(\lambda)|^2}{\lambda^2}\, d\lambda = \int_{\mathbb{R}} \frac{|\psi(x)|^2}{x^2}\, dx < \infty$. Furthermore, ψ is normalized such that $K_\psi = 1$. In the case where $K_\psi < \infty$, ψ is termed admissible.
Now we want to take a look at the properties of the ridgelet transform.

Inverse Transformation:

For integrable or square-integrable functions, f can be uniquely recovered from its ridgelet transform:

$$f(x) = \int_0^{2\pi} \int_{-\infty}^{\infty} \int_0^{\infty} \mathcal{R}_f(a, b, \theta)\psi_{a,b,\theta}(x) \frac{da}{a^3}\, db \frac{d\theta}{4\pi}$$

Additionally, Parseval's theorem holds:

$$\int_{\mathbb{R}^2} |f(x)|^2\, dx = \int_0^{2\pi} \int_{-\infty}^{\infty} \int_0^{\infty} |\mathcal{R}_f(a, b, \theta)|^2 \frac{da}{a^3}\, db \frac{d\theta}{4\pi}$$

Discrete Ridgelet Transform:

For practical applications, a discrete form of the ridgelet transform is often needed. In the most papers, the way to discretize the ridgelet transform goes over the representation of the ridgelet transform shown in 5.4 and discretizing the radon and wavelet transform. We tak an other way and use the discretizaion made by Emmanuel Candès and David Donoho [9]. We derive this

from the continuous ridgelet transform in terms of the Fourier transform. Let $\hat{f}$ and $\hat{\psi}$ be the Fourier transforms of f and ψ, respectively. Then, the ridgelet transform can be represented as:

$$\mathcal{R}_f(a,b,\theta) = \frac{1}{2\pi}\int_{\mathbb{R}^2} \hat{f}(\xi)\overline{\hat{\psi}}_{a,b,\theta}(\xi)\,d\xi$$

We represent ξ in polar coordinates, $\xi(\lambda,\theta) = (\lambda\cos(\theta), \lambda\sin(\theta))$, where $\lambda \in \mathbb{R}$ and $\theta \in [0, 2\pi)$. Thus, we have:

$$\mathcal{R}_f(a,b,\theta) = \frac{1}{2\pi}\int_{\mathbb{R}} \sqrt{a}\overline{\hat{\psi}}_{a,b,\theta}(a\lambda)e^{-i\lambda b}\hat{f}(\xi(\lambda,\theta))\,d\lambda$$

This says, that the ridgelet transform can also be written as the integral of the weighted fourier transform of f as $w_{a,b}\hat{f}(\xi)$ with $w_{a,b} = a^{1/2}\overline{\hat{\psi}}_{a,b,\theta}(a|\xi|)e^{-i\lambda b}$. Now if we restrict the ridgelet transform to a function of b as $\rho_{a,\theta}(b) = \mathcal{R}_f(a,b,\theta)$ it holds that

$$\rho_{a,\theta}(b) = \mathcal{F}^{-1}(\hat{\rho}_{a,\theta}(\lambda)$$

where $\mathcal{F}$ is the 1-d inverse fourier transform and

$$\hat{\rho}_{a,\theta}(\lambda) = a^{1/2}\hat{\psi}_{a,b,\theta}(a\lambda)\hat{f} \quad \lambda \in \mathbb{R}$$

is the restriction of $w_{a,0}\hat{f}(\xi)$ to the radial line. The continuous ridgelet transform at a particular scale a and angle θ can be computed as follows:

- Perform 2D Fourier transform on $\hat{f}(\xi)$
- Compute the radial cut $w_{a,0}(\xi)\hat{f}(\xi)$
- Perform inverse 1D Fourier transform along the radial line to obtain $\rho_{a\theta}(b)$ for all $b \in \mathbb{R}$

Next, we discretize $(a_j, b_{j,k}, \theta_{j,l})$ such that we obtain frame bounds, ensuring:

$$\sum_{j,k,l} |\mathcal{R}_f(a_j, b_{j,k}, \theta_{j,l})|^2 \asymp \int\int\int |\mathcal{R}_f(a,b,\theta)|^2 \frac{da}{a^3}\,db\,d\theta$$

For simplicity, we assume $\hat{\psi}(\lambda) = \chi_{1\leq|\lambda|\leq 2}(\lambda)$. It has been shown that a dyadic discretization for the scaling constant a and the translation parameter b, i.e., $a_j = a_0 2^j$ and $b_{j,k} = 2\pi k 2^{-j}$, offers an effective method. With this discretization, ridgelet coefficients can be written as:

$$\mathcal{R}_f(a_j, b_{j,k}, \theta) = \frac{1}{2\pi}2^{-j/2}\int_{2^j\leq|\lambda|\leq 2^{j+1}} e^{-i\lambda 2\pi k 2^{-j}}\hat{f}(\xi(\lambda,\theta))\,d\lambda$$

Moreover, using Plancherel's theorem, we have:

$$\sum_k |\mathcal{R}_f(a_j, b_{j,k}, \theta)|^2 = \frac{1}{\sqrt{2\pi}}\int_{2^j\leq|\lambda|\leq 2^{j+1}} |w_{2^j,0}|^2|\hat{f}(\xi(\lambda,\theta))|^2\,d\lambda$$

Similarly, a discretization in the angle parameter θ is meaningful, i.e.,

$$\theta_{j,l} = 2\pi l 2^{-j}$$

Thus, the set

$$\{2^{j/2}\psi(2^j(x_1\cos(\theta_{j,l}) + x_2\sin(\theta_{j,l}) - 2\pi k 2^{-j}))\}_{j\geq j_0,k,l}$$

forms a frame on the unit disk. Hence, for any function f with finite L^2-norm,

$$\sum_{j,k,l} |\langle\psi_{(a_j,b_{j,k},\theta_{j,l})}, f\rangle|^2 \asymp ||f||_2^2$$

5.2.1. Ridgelet Filter

In chapter 5.1.2 we saw that ridgelets can be used as activation functions for neural networks to approximate a given function with singularities. However, the use of ridgelets does not end there. We can also use ridgelets to define filters that we can apply in a convolutional layer like it is shown in the work of Sahul Mohan Tarachandy and Aravinth J [49]. The idea behind this is the application of the ridgelet transform mentioned in chapter 5.2 to detect the presence of edges or edge-like structure in the area that the filter is covering. So we transfer the idea of edge detection for a continuous function to a discrete function, or in this case, an image. What is the scalar product of function and ridgelet in the continuous case corresponds here to a convolution of image section and ridgelet filter at a certain position in the image. The aim is then not to train each individual filter pixel or matrix entry, as is the case in the normal CNN, but to adjust the ridgelet parameters a, b and θ. For this adaptation in the backpropagation of the neural network, we use the Newton method or the steepest gradient descent method.
For this, we take a look at the definition of the ridgelet transform as the scalar product of function and ridgelet, like in chapter 5.2.

$$\mathcal{R}_f(a,b,\theta) = \int_{\mathbb{R}^2} f(x)\psi_{a,b,\theta}(x)\,dx \tag{5.5}$$

Where we have

$$\psi_{a,b,\theta}(x) = a^{-\frac{1}{2}}\psi\Big(\frac{x\cdot\Theta - b}{a}\Big) \tag{5.6}$$

With ψ as an one dimensional wavelet and $\Theta = (\cos\theta, \sin\theta)$.
The calculation of the gradients as a function of the parameters a, b and θ is then carried out using the well known chain rule.

Chain rule for scaling parameter a
The gradient of a with respect to the loss function L is

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial \mathcal{R}_f}\cdot\frac{\partial \mathcal{R}_f}{\partial a} \tag{5.7}$$

Here $\frac{\partial L}{\partial \mathcal{R}_f}$ is the gradient of the loss with respect to the ridgelet coefficient and $\frac{\partial \mathcal{R}_f}{\partial a}$ can be computed via multiplication and chain rule as follows

$$\frac{\partial \mathcal{R}_f}{\partial a} = -\frac{1}{2}a^{-\frac{3}{2}}\int_{\mathbb{R}^2} f(x)\psi\Big(\frac{x\cdot\Theta - b}{a}\Big)\,dx + a^{-\frac{1}{2}}\int_{\mathbb{R}^2} f(x)\psi'\Big(\frac{x\cdot\Theta - b}{a}\Big)\cdot\frac{-(x\cdot\Theta - b)}{a^2}\,dx$$

Chain rule for position parameter b
For b we have the chain rule

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \mathcal{R}_f}\cdot\frac{\partial \mathcal{R}_f}{\partial b} \tag{5.8}$$

With

$$\frac{\partial \mathcal{R}_f}{\partial b} = -a^{-\frac{3}{2}}\int_{\mathbb{R}^2} f(x)\psi'\Big(\frac{x\cdot\Theta - b}{a}\Big)\,dx$$

Chain rule for angle parameter θ
For θ we have

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \mathcal{R}_f} \cdot \frac{\partial \mathcal{R}_f}{\partial \theta} \tag{5.9}$$

With

$$\frac{\partial \mathcal{R}_f}{\partial \theta} = a^{-\frac{1}{2}} \int_{\mathbb{R}^2} f(x) \psi' \Big(\frac{(x_1 \cos\theta + x_2 \sin\theta) - b}{a} \Big) \cdot \frac{-x_1 \sin\theta + x_2 \cos\theta}{a} \, dx$$

For computer applications, we have to discretize these equations. For this we need the discretized ridgelet transform

$$\mathcal{R}_f(a,b,\theta) \approx \mathcal{R}_f^d(a,b,\theta) = \sum_{i,j \in \mathbb{Z}^2} f[i,j] \psi_{a,b,\theta}(i,j)$$

With this we get the discrete gradient calculations

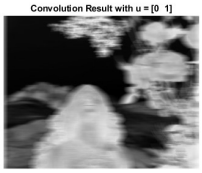
$$\begin{aligned} \frac{\partial \mathcal{R}_f}{\partial a} \approx & -\frac{1}{2} a^{-\frac{3}{2}} \sum_{i,j \in \mathbb{Z}^2} f[i,j] \psi \left(\frac{(i,j) \cdot \Theta - b}{a} \right) \\ & + a^{-\frac{1}{2}} \sum_{i,j \in \mathbb{Z}^2} f[i,j] \psi' \left(\frac{(i,j) \cdot \Theta - b}{a} \right) \cdot \frac{-((i,j) \cdot \Theta - b)}{a^2}, \end{aligned} \tag{5.10}$$

$$\frac{\partial \mathcal{R}_f}{\partial b} \approx -a^{-\frac{3}{2}} \sum_{i,j \in \mathbb{Z}^2} f[i,j] \psi' \left(\frac{(i,j) \cdot \Theta - b}{a} \right), \tag{5.11}$$

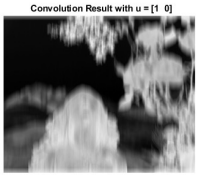
$$\frac{\partial \mathcal{R}_f}{\partial \theta} \approx a^{-\frac{1}{2}} \sum_{i,j \in \mathbb{Z}^2} f[i,j] \psi' \left(\frac{(i \cos\theta + j \sin\theta) - b}{a} \right) \cdot \frac{-i \sin\theta + j \cos\theta}{a}. \tag{5.12}$$

The advantage here is that we do not have to train every entry of the kernel, but only the angle, scaling and position of the ridgelet, regardless of the kernel size. For example, we want to show the effects of ridgelet convolution with different rotation, scale and position parameters. Our test image from the flowers data set [13] has a size of 494×349 and we apply kernels with the size 16×16. Let us take a look at different convolutions with different scales and positions:

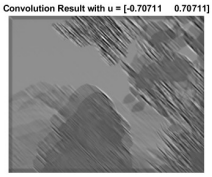
(a) Original Image



(b) $u = [0, 1]$



(c) $u = [1, 0]$



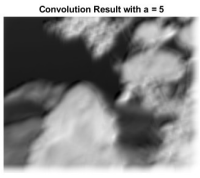
(d) $u = [-1/\sqrt{2}, 1/\sqrt{2}]$

(e) $u = [1/\sqrt{2}, 1/\sqrt{2}]$

Figure 5.7.: Ridgelet Convoluted Image with different Ridgelet Directions

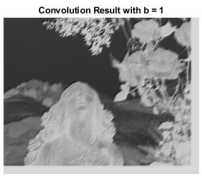
We can clearly see a slight blurring in the direction of the corresponding direction vector in the images and more ore less sharp contrasts, depending on the direction. Now let us take a look at how the scaling and position parameters affect the convolution:

(a) Original Image



(b) $a = 5$

(c) $a = 0.2$



(d) $b = 1$

(e) $b = -1$

Figure 5.8.: Ridgelet Convoluted Image with different Ridgelet scales and positions

We can observe the greatest effect with the scaling parameter. For a larger a, the ridgelet stretches further in the direction u. This makes the resulting filtered image more blurred, while it becomes sharper for a smaller a.

5.2.2. Application of the Ridgelet Convolution in CNN's

Now that we have looked at the effect of ridgelet filters, let us apply them to CNNs. As mentioned at the beginning, ridgelet filters in neural networks have the advantage that, regardless of their size, only the parameters u, a and b need to be adapted or learned. On the other hand, however, it is also necessary to choose the filters as large as possible, otherwise the ridgelet profile will be lost. For example, this comes out better in a 30x30 kernel than in one with a size of 3x3. For comparison, we will look at the following table which shows the learning duration and accuracies for different kernel sizes. The network itself classifies images of the NWPU-RESISC45 data set, which consists of 45 image classes of 256×256 RGB images. These are, for example, vehicles, buildings or landscapes. For network learning, these images are grayscaled and downsized to 64×64. The network, which learns on 14 cores in parallel, looks as follows:

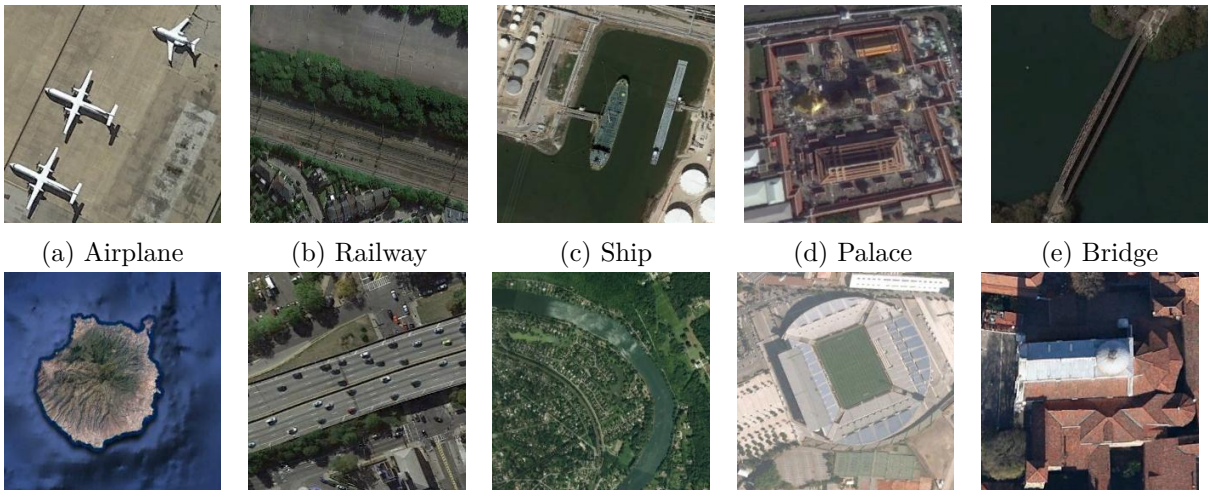
```
    imageInputLayer([imsize imsize 1])
%Changing Ridgelet and Convolutional Layers with comments
%RidgeletConvLayer([28 28],10, 'ridglet_conv')
convolution2dLayer(28, 10, 'Padding', 'same')

batchNormalizationLayer
reluLayer

maxPooling2dLayer(2, 'Stride', 2)

fullyConnectedLayer(32)
reluLayer

fullyConnectedLayer(45)        %Output for 45 classes
softmaxLayer
```

For testing we will activate and deactivate the ridgelet and convolutional layer and let the network learn for 20 epochs with a batch size of 128. The results looks as follows:

Kernel Size		CNN	RCNN
3	Time	2:24 min	3:25 min
	Accuracy	0.406	0.367
14	Time	3:25 min	4:34 min
	Accuracy	0.419	0.406
28	Time	6:26 min	4:55 min
	Accuracy	0.393	0.422

Table 5.1.: Comparison of CNN and RCNN in terms of time and accuracy

We can see that with a small kernel size, the convolutional layer has an advantage in terms of learning time and accuracy. However, with a larger kernel size, the ridgelet layer quickly becomes better, especially when we look at the learning time. The advantage here is that the ridgelet kernel only has three learnable variables, whereas the convolutional layer has 28x28. To make this advantage even clearer, we scale the images to 128x128 and enlarge the kernel to 56x56. With this setup, the standard CNN needs 1 hour and 30 minutes for 20 Epochs, while the RCNN needs only 22 minutes. So the RCNN is more than three times faster than the CNN for large kernel sizes. This makes ridgelet filtering very useful for the classification of images with high resolution, but for images with a small number of pixels, the standard convolution is still the best.
So far we have only looked at RCNN's in terms of learning speed, the accuracys remained pretty much the same, mainly because we only looked at 5 of the 45 classes in the NWPU-RESISC45 data set [35]. Now we want to see how well ridgelet layers perform compared to conventional convolutional layers in terms of accuracy. For this purpose and for further network tests, we now use 10 specially selected classes from the NWPU data set. Furthermore, we are now switching to network implementations in PyTorch, as the networks can be represented particularly well here. We will look at further advantages of Pytorch later in other application examples. All images are always scaled down to a resolution of 128x128 pixels, deviations from this are mentioned.
Here we see sample images of the selected classes:



(a) Airplane (b) Railway (c) Ship (d) Palace (e) Bridge

(f) Island (g) Freeway (h) River (i) Stadium (j) Church

Figure 5.9.: Example Images of the 10 used Classes

These classes were chosen because distinguishing between some classes is a challenge for the network due to their similarity. For example, railways, freeways and bridges can look quite similar. Buildings can also be easily confused with each other due to a poorly classifying network.
The network with which we now want to classify the images consists of a Ridgelet Layer with 15 kernels of size 50×50, followed by a Batch Normalization Layer, a ReLu Layer, a MaxPooling Layer and two fully connected layers at the end. The structure of the network is therefore relatively simple and the implementation in Pytorch is as follows:

```
class PureRidgeletNetwork(nn.Module):
    def __init__(self, num_classes=10, num_kernels=15,
                         kernel_size=30):
        super(PureRidgeletNetwork, self).__init__()

        # Ridgelet Layer
        self.ridgelet = RidgeletLayer(in_channels=3,
        out_channels=32,
        kernel_size=50, num_kernels=15)

        # Batch Normalization Layer
        self.bn = nn.BatchNorm2d(32)

        # ReLU Activation Layer
        self.relu = nn.ReLU()

        # MaxPooling Layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully Connected Layers
        # Adjust input size according to the output shape
        self.fc1 = nn.Linear(131072, 256)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self, x):
        # Apply Ridgelet Layer
        x = self.ridgelet(x)

        # Apply Batch Normalization
        x = self.bn(x)

        # Apply ReLU Activation
        x = self.relu(x)

        # Apply MaxPooling
        x = self.pool(x)

        # Flatten the tensor for fully connected layers
        x = x.view(x.size(0), -1)  # Flatten the output

        # Apply Fully Connected Layers
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
```

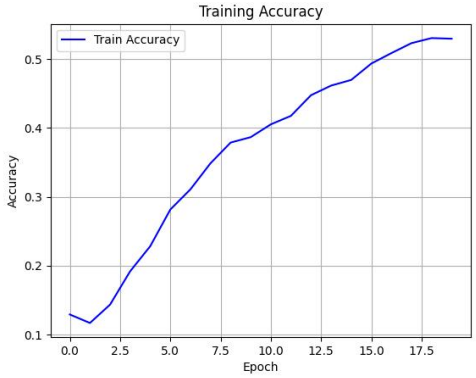
```
        return x
```

To compare the performance, we will use a conventional convolutional network. This initially consists of three convolutional layers, followed by a MaxPooling layer and two fully connected layers. The implementation looks like this:

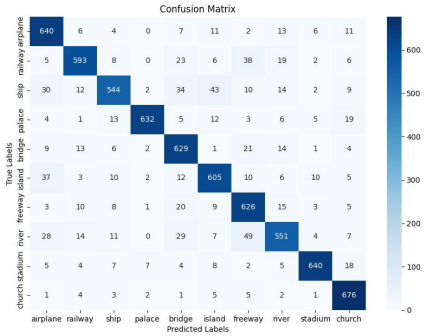
```
        class StandardCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(StandardCNN, self).__init__()

        self.conv1 =
        nn.Conv2d(in_channels=3, out_channels=32,
        kernel_size=3, padding=1)
        self.conv2 =
        nn.Conv2d(in_channels=32, out_channels=64,
        kernel_size=3, padding=1)
        self.conv3 =
        nn.Conv2d(in_channels=64, out_channels=128,
        kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Compute the size of the flattened input to the first
        fully connected layer
        self.fc1_input_size = 128 * (high // 8) * (high // 8)

        self.fc1 = nn.Linear(self.fc1_input_size, 512)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(x.size(0), -1)  # Flatten the tensor

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Let's look at the learning curves and the confusion matrices again.

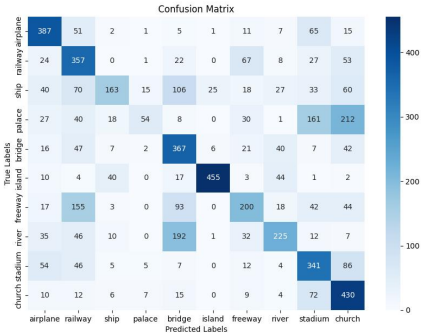
(a) Accuracy over the Epochs, RCNN



(b) Accuracy over the Epochs, CNN



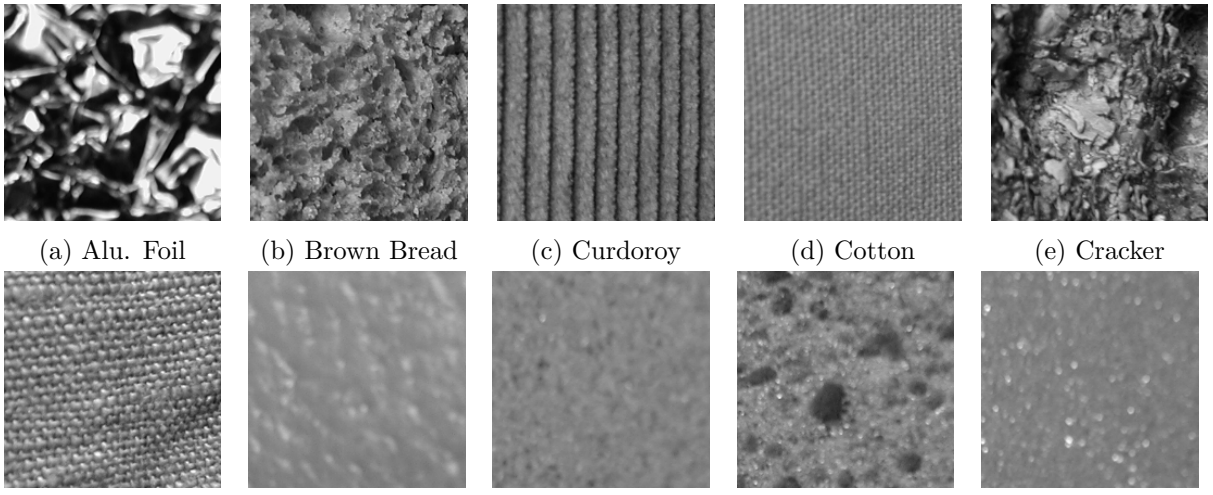
(c) Confusion Matrix, RCNN



(d) Confusion Matrix, CNN

Figure 5.10.: Performance of the RCNN applied on the KTH TIPS dataset

We can see that the performance of the RCNN far exceeds that of the CNN. Not only does the accuracy increase much faster and end up higher, the learning process itself is also much more stable. Especially in the CNN's confusion matrix, the pitfalls of the selected classes become apparent: railways were often mistaken for freeways, bridges for rivers and ships, and stadiums and churches were confused with palaces. The RCNN, on the other hand, coped very well with these difficulties. The majority of the images were correctly assigned, which is due to the good edge detection behavior of the ridgelets. However, it should also be mentioned that the runtimes differ greatly. While an epoch of the CNN took about 17 seconds on average, the RCNN took about 170 seconds, i.e. ten times as long. However, this is still manageable and the accuracy after 20 epochs speaks for itself.
Now that we have tested the RCNN on a very classic image data set, where its true strength, namely edge detection, comes into its own. For this purpose, we use the KTH-TIPS data set [23], which consists of 200×200 gray scale images of 10 different materials. We can look at some examples of these images here:

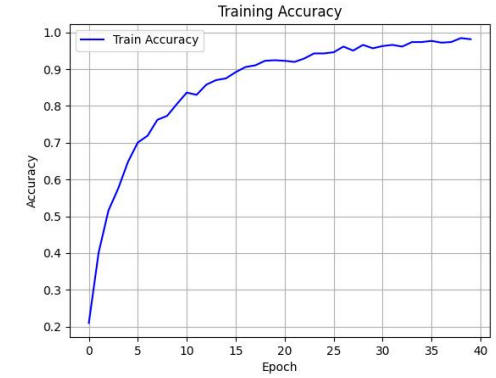


(a) Alu. Foil (b) Brown Bread (c) Curdoroy (d) Cotton (e) Cracker

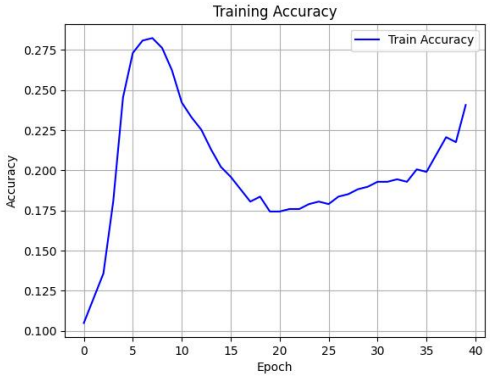
(f) Linen (g) Orange Peel (h) Sandpaper (i) Sponge (j) Styrofoam

Figure 5.11.: Example Images of the 10 used Material Classes

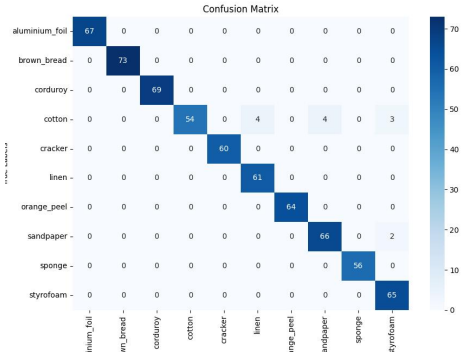
We can see immediately that this is a fairly sophisticated data set. Some of the classes, such as styrofoam and sandpaper, look quite similar. This can therefore be a challenge for a neural network. Ridgelet convolution is very promising in this case, as the ridgelet filter can recognize edges of different thicknesses and orientations very well. The network we use remains the same and for comparison we take again the CNN defined above. For faster testing, the images are scaled down to a resolution of 100×100. Because the data set is harder to classify, we will let run both networks over 40 epochs. Here are the results:



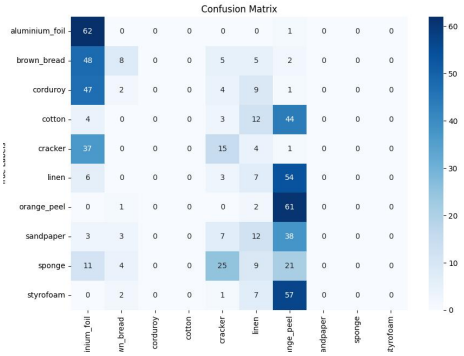
(a) Accuracy over the Epochs, RCNN



(b) Accuracy over the Epochs, CNN



(c) Confusion Matrix, RCNN



(d) Confusion Matrix, CNN

Figure 5.12.: Performance of the RCNN and CNN applied on the KTH TIPS dataset

We observe an amazingly good result with the RCNN. Not only does it learn more stably compared to the CNN, but its accuracy results far surpass those of the CNN. Even with this challenging data set, the RCNN achieves an accuracy of over 98%. In contrast, the CNN was hardly able to perform at all.
In conclusion, we can say that the RCNN works exceptionally well, particularly with high-resolution, edge-rich images, such as the material surfaces shown above. While the conventional CNN still has its relevance, especially since it is significantly superior to the RCNN in terms of runtime, it should be applied to less complex data sets with lower resolution.
Thus, we see that both network architectures have their own merits and applications. This raises the idea of whether the two architectures could be combined to further improve image classification and it has been shown that this is indeed possible. However, before we go into this topic, in the next chapter we want to take a step away from ridgelet networks and instead look at a network architecture that is not as widely used, but provides a new, effective approach to classifying color images, the so-called quaternion convolutional neural networks.

6. Quaternion Neural Networks

So far, artificial neural networks, despite their comprehensive treatment using ridge functions and ridgelets, have only dealt with real numbers in this work. However, some considerations for neural network applications extend beyond this approach and involve the use of quaternions in these networks. In particular, we will focus on the quaternion convolutional neural network, which is based on the work of Xuanyu Zhu, Yi Xu, Hongteng Xu, and Changjian Chen [55]. The basic idea of the network is based on interpreting the pixels of an image as a color vector and rotating it, so that not all three color values are learned per pixel, but only the scaling and the rotation angle of the vector. This rotation can be realized very well with the special algebra of quaternions. We will look at this and the structure of the quaternion network in the following chapters.

6.1. Fundamentals of Quaternions

Quaternions extend the number system beyond real and complex numbers by introducing an imaginary part consisting of three components, represented by the imaginary units i, j, and k. A quaternion $q \in \mathbb{H}$ can be uniquely represented in the form

$$q = a + bi + cj + dk,$$

where $a, b, c, d \in \mathbb{R}$. The elements $1, i, j, k$ thus form a basis for quaternions. The imaginary units satisfy

$$i^2 = j^2 = k^2 = ijk = -1,$$

and the following multiplication rules apply:

- $ij = k$, $jk = i$, $ki = j$
- $ji = -k$, $kj = -i$, $ik = -j$

It is evident from these rules that $\mathbb{H}$ is not a field due to the lack of commutativity,so it is only a division ring. However, it is a skew field. Division in quaternions does not use a fraction bar because the direction of multiplication is not uniquely defined.
The following example shows that multiplication with quaternions is somewhat more complicated than with real or complex numbers:
Given two quaternions $q_1 = a_1 + b_1 i + c_1 j + d_1 k$ and $q_2 = a_2 + b_2 i + c_2 j + d_2 k$, their product is

$$\begin{aligned} q_1 q_2 &= (a_1 + b_1 i + c_1 j + d_1 k)(a_2 + b_2 i + c_2 j + d_2 k) \\ &= a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2 \\ &+ (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i \\ &+ (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j \\ &+ (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k \end{aligned}$$

We see that in practice we have to implement the multiplication of quaternions carefully. The division of quaternions also causes problems, as the notation with a fraction bar is not unique due to the lack of commutativity. For this reason, $\frac{q_1}{q_2}$ is written as either $q_1 q_2^{-1}$ or $q_2^{-1} q_1$ with

$$q^{-1} = \frac{1}{(q\overline{q})^2}\overline{q}.$$

Here $\overline{q}$ is the conjugate of q. Equally to the complex conjugate, it negates the imaginary part of a quaternion, so that $\overline{q} = a - bi - cj - dk$.
A special feature of quaternions, which is also used in practice, is that quaternion operations can be used to describe rotations in $\mathbb{R}^3$. For this, let $x = (x_1, x_2, x_3)$ be a vector, which is to be rotated around a rotation vector $w \in \mathbb{R}^3$, $|w| = 1$ with a certain angle $\theta \in [0, 2\pi)$. This leads to a vector $y = (y_1, y_2, y_3)$. Now we can represent the rotation with the following quaternion multiplication:

$$y^* = w^* x^* \overline{w^*} \tag{6.1}$$

with the quaternion representation of the vectors $x^* = 0 + x_1 i + x_2 j + x_3 k$, $y^* = 0 + y_1 i + y_2 j + y_3 k$ and $w^* = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(w_1 i + w_2 j + w_3 k)$.

6.2. The Quaternion Convolutional Layer

Quaternions are a frequently considered research object in many parts of analysis, for example Fourier analysis ([42]) or wavelets([4], [52]). The use of quaternions in the context of neural networks has also been an area of active research in recent years. Many studies have shown that quaternions perform better than their conventional counterparts in many areas of application. We now want to take a look at quaternion convolutional neural networks (QCNN for short), which we want to use to classify color images. To do this, we must first convert the given image into a quaternion representation. Our three clour channel image $A \in \mathbb{R}^{n \times n \times 3}$ is now to be represented as a pure quaternion matrix:

$$A^* = [a_{ij}^*] \in \mathbb{H}^{n \times n}$$

with

$$A^* = 0 + Ri + Gj + Bk, \;\; R, G, B \in \mathbb{R}^{n \times n}$$

Here the matrices R, G and B are the corresponding red, green and blue channels of the image. Now we want to define a convolution for such a quaternion matrix.
For this, we take a convolution kernel $W^* = [w_{ij}^*] \in \mathbb{H}^{L \times L}$. We have two requirements for this matrix. First, it should apply rotations and scalings to color vectors in order to find the best representation in the whole color space and second, it should work as a normal convolution for gray scale pictures. To achieve this, we use the rotation formular from 6.1. So we define the entries of the convolution kernel as

$$w_{ij}^* = s_{ij}(\cos\frac{\theta_{ij}}{2} + \sin\frac{\theta_{ij}}{2}\mu)$$

with $\theta_{ij} \in [0, 2\pi)$, $s_{ij} \in \mathbb{R}$ and μ is a fixed unit rotation axis. Now we can perform a convolution like

$$A^* \circledast W^* = F^* = [f_{kl}^*] \in \mathbb{H}^{n-L+1 \times n-L+1}$$

with

$$f_{kl}^* = \sum_{i=1}^{L}\sum_{j=1}^{L} \frac{1}{s_{ij}} w_{ij}^* a_{i+l,k+j}^* \overline{w_{ij}^*}$$

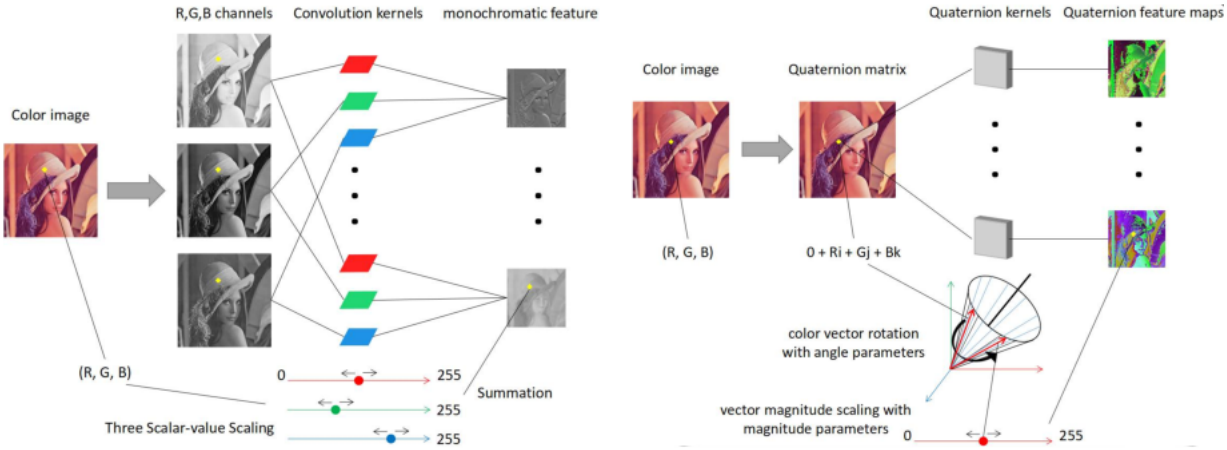
We can also calculate this quaternion operation with a matrix representation as follows:

$$f_{kl} = \sum_{i=1}^{L}\sum_{j=1}^{L} s_{ij} \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix} a_{i+k,j+l}$$

Here, f_{kl} is the vectorized representation of f^*_{kl} and

$$f_1 = \frac{1}{3} + \frac{2}{3}\cos\theta_{ij}, \quad f_2 = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} - \frac{\pi}{3}), \quad f_3 = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} + \frac{\pi}{3})ta_{ij},$$

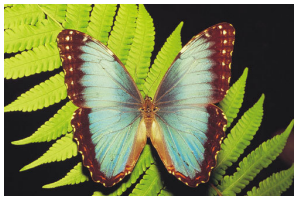
Fig. 6.1 shows how quaternion convolution works. In contrast to conventional convolution, in which the weights are adjusted for each color channel, what makes three learnable variables per pixel, in quaternion convolution the color vector of a pixel is moved over the surface of a cone around the vector $\frac{1}{\sqrt{3}}(1,1,1)^T$. This requires only two learnable variables per pixel, namely the scaling and the rotation angle. Although this reduces the range in which the color vector moves, it also reduces overfitting.



(a) Real-valued CNN (b) Quaternion CNN

Figure 6.1.: Example of image processing in real and quaternion CNN's [55]

We will now use an image of a butterfly taken from the butterfly dataset [1] to see how quaternion folding affects an RGB image:



(a) Original Butterfly Image

(b) Kernel size=1 × 1, $\theta = -\pi/2$, scale=1

(c) Kernel size=1×1, $\theta = \pi/2$, scale=1

(d) Kernel size=1 × 1, $\theta = \pi/2$, scale=2

(e) Kernel size=3×3, $\theta = -\pi/2$, scale=0.1

(f) Kernel size=3×3, $\theta \in [-\pi/2, \pi/2]$, scale $\in [0, 0.1]$ (random)

Figure 6.2.: Convoluted Image with different Kernels

We see that different kernel sizes, scales and angles have different influences on the color aspects of the images. This can emphasize or attenuate certain areas in the image, which helps to classify the images.

6.3. The Quaternion fully connected Layer

Now that we have dealt with the convolutional layer, which can process quaternions, we turn to the most well-known structure of a neural network, the fully connected layer.
In real-valued CNN's we take the dot product of the input and weight vector. This can also be interpreted as a convolution of the input with a kernel of the same size. We want to apply this concept to a quaternion valued input $a^* = [a_i^*]_{i=1}^N \in \mathbb{H}^N$. For this we have M kernels $w^* = [w_j^*]_{j=1}^M \in \mathbb{H}^M$ as defined above. Then we get the output $b^* = [b_j^*]_{j=1}^M \in \mathbb{H}^M$ as follows:

$$b_j^* = \sum_{i=1}^{N} \frac{1}{s_i} w_i^* a_i^* \overline{w_i^*}$$

with $s_i = |w_i^*|$.
As in the convolutional layer, we do not have three weights per pixel for the color channels, but only rotation angle and scaling, which have to be trained. What we get is a rotated color vector for each pixel, like in the figures 6.2b and 6.2c. These are then added for the neuron input.
The definition of activation functions for the fully connected layers of a QCNN is relatively intuitive. In most cases, the function is simply applied component by component. For example, let σ be the activation function of a neuron and $q = a + bi + cj + dk$ the quaternion-valued input. Then the output results in

$$Y(q) = \sigma(a) + \sigma(b)i + \sigma(c)j + \sigma(d)k$$

6.4. Backpropagation in QCNN's

The last problem you have to deal with with QCNNs is backpropagation. It works again according to the chain rule already mentioned, but unlike conventional CNNs, it is not based on the scalar product of input and weight vector, but on quaternion multiplication of pure quaternions. In this case, let $q^* = ai + bj + ck$ and $p^* = xi + yj + zk$ with $p^* = w^* q^* \overline{w^*}$. Also this can be represented in matrix representation for better understanding. For this, let $q = [a, b, c]^T$ and $p = [x, y, z]^T$ be the corresponding vector representation of the above defined quaterions p^* and q^* and let L be the loss function. Then we get

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial q}, \quad \frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial \theta}, \quad \frac{\partial L}{\partial s} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial s}$$

For this equation we have

$$\frac{\partial p}{\partial q} = s\begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix}, \quad \frac{\partial p}{\partial \theta} = s\begin{pmatrix} f_1' & f_2' & f_3' \\ f_3' & f_1' & f_2' \\ f_2' & f_3' & f_1' \end{pmatrix} \quad \frac{\partial p}{\partial s} = \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

6.5. Benefits of a QCNN

We have seen that the definition and implementation of a QCNN is relatively complex. This does not even apply to the implementation of the quaternion convolution itself, but the back-propagation in particular is complex and, if it is not optimized, correspondingly slow. However, if you have a functioning QCNN in front of you, you will quickly gain many advantages for learning. On the one hand, fewer variables have to be trained in the network, as each color value of a pixel is no longer trained, but the color pixel with rotation angle and scaling as a

whole. Furthermore, the reduced space in which the vector can move ensures that overfitting is avoided. This means that a QCNN learns fairly quickly and effectively, making it ideal for use in the field of image classification or image denoising, like it is stated in [55]. There the QCNN shows better results compared to a CNN, depending on the used data set.

7. The Splitted CNN

In the last section we want to take a look at a very special CNN architecture, which splits the input into a low and high resolution part, which are processed parallel. The aim of using this network architecture is to classify high-resolution images as accurately as possible. For the low resolution part, we will use standard convolutional layers, because they are very effective for this task. The problem with using convolutional layers in the high-resolution part is that the computing effort and the associated computing time increase rapidly with the size of the image. Therefore, we want to use a special type of convolutional layer for this part, which is based on so-called ridgelet filters, which we will now consider. The idea of this splitted CNN (SCNN) is based on the work of T Gladima Nisia and S Rajesh [31].

7.1. The SCNN in Application

Like said at the beginning, the idea is to split the input image in two branches. On the one hand, the high-resolution image remains. This is then processed in a ridgelet layer as shown in the last section. On the other hand, we create a version of the image with a lower resolution. To simplify matters, we will not divide the image into four low-resolution images, like it is done in the paper, but only use one image with half the edge length. This is then passed on to a standard convolutional layer. The two resulting feature maps are then merged again via concatenation and processed further. All of this is illustrated in figure 7.1.

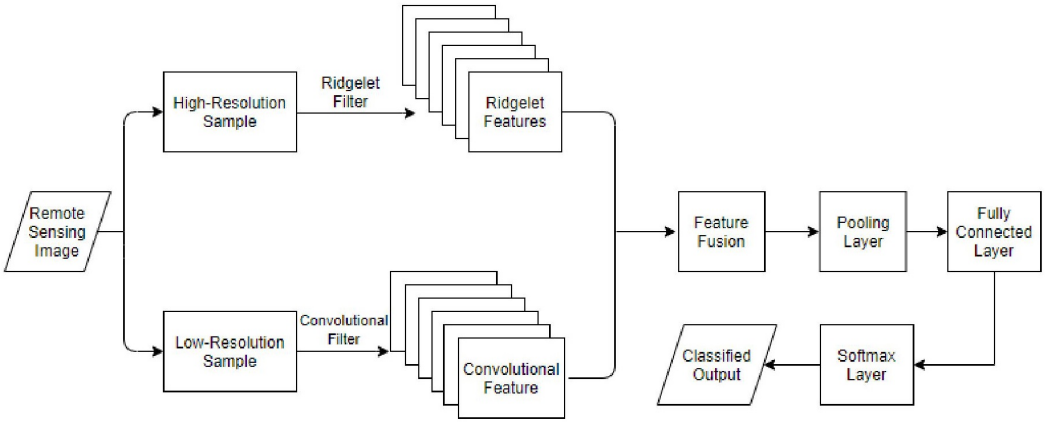


Figure 7.1.: Basic structure of an SCNN
[31]

In contrast to the last section, we will no longer use the mexican hat wavelet for ridgelet filtering, but the so-called DOG (difference of gaussian) wavelet for the ridgelet. This is defined as

$$\psi(x) = e^{-x^2/2} - \frac{1}{2}e^{-x^2/8}$$

Furthermore, we no longer want to set the entire direction vector u as learnable, but, since it is only a two-dimensional unit vector, the angle of rotation θ. So the ridgelet itself looks as follows:

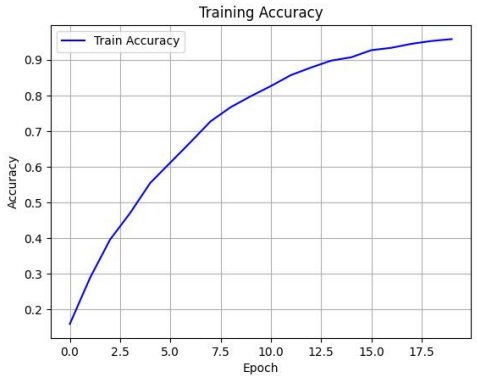
$$\Psi(x) = a^{-1/2}\psi(\frac{x_1 \cos\theta + x_2 \sin\theta - b}{a})$$

Because the data set is split, processed and merged again in the process, we will now use Python instead of Matlab for the implementation of the network. Especially we will use PyTorch, because the realization of the splitting is really simple there. So, mathematically expressed, the split network part does the following:

$$F_{low} = g(f(X_{low} * K + b))$$

$$F_{high} = g(f(X_{high} * R))$$

Here F is the output, X the high and low resolution input, K the convolution kernel, R the ridgelet kernel, f the activation function and g the pooling operator. Afterwards the two separated outputs are concatenated. The merged output then goes in a pooling layer, followed by a fully connected layer and finally put into a softmax layer for classification. One can make this network as complex as wanted, but for testing this will be our minimal example.
Like said, this network will be implemented in PyTorch because of the easy splitting and fusion of the high and low resolution branches. For testing, we take 10 classes of the NWPU-RESISC45, namely *airplane, railway, ship, palace, bridge, island, freeway, river, stadium* and *church* as showed above in figure **??**.
Let's take a look at its training progress. For the training, the images were scaled down to a size of 128x128 and for the low-resolution branch we use an image size of 32x32. The ridgelet layer has 8 kernels with a kernel size of 30x30. Training is performed with the SGD optimizer with a learning rate of 0.001 over 20 epochs. For comparison, we take a conventional CNN of similar size, in which the split part was replaced by two standard convolutional layers in a row. We can see the results here:

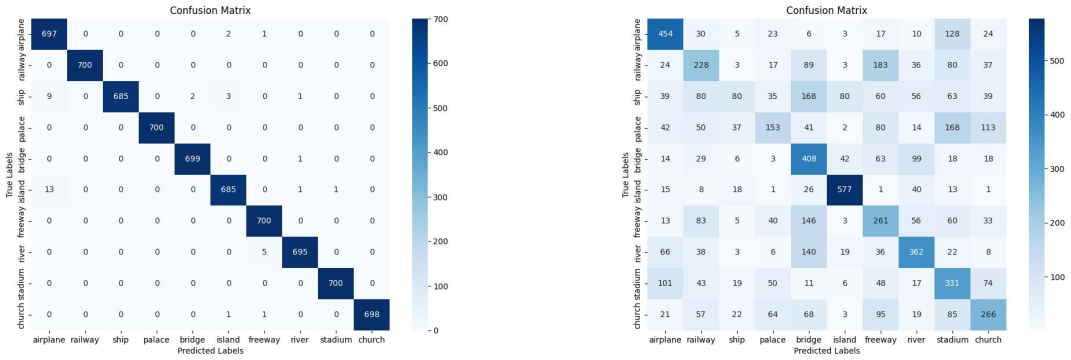


(a) Learning Progress of SCNN

(b) Learning Progress of CNN

Figure 7.2.: Learning Progress of SCNN and CNN in Comparison, 20 Epochs

We see, that the accuracy of the conventional CNN is much worse after 20 epochs. It doesn't even reach half of the SCNN accuracy. More interesting is a look at the confusion matrices:



(a) Confusion Matrix of SCNN

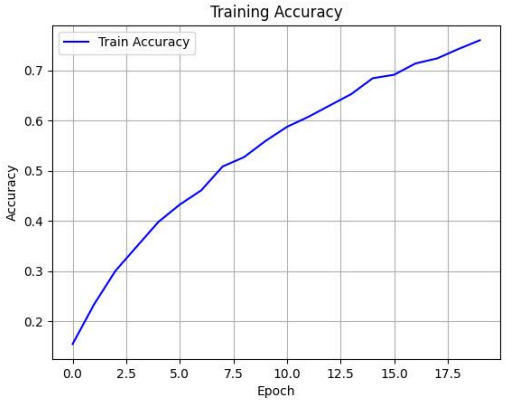
(b) Confusion Matrix of CNN

Figure 7.3.: Confusion Matrices of SCNN and CNN after 20 Epochs

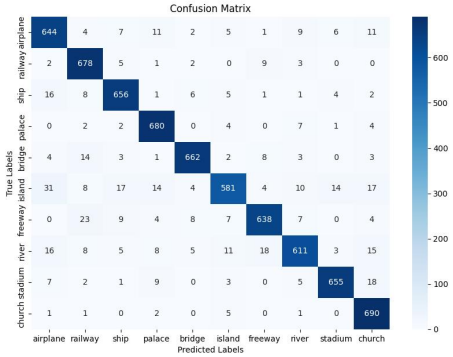
We can see that the SCNN assigns the given images extremely well to the respective classes. With the CNN, on the other hand, the high error rate was already apparent in the training trial. What is exciting, however, is that the problem mentioned at the beginning is proving to be true here. Bridges in particular were often incorrectly interpreted as a river or freeway and the freeway was often mistaken for a railway. Similarly, stadium and church were often mistaken for a palace. So we can see that the SCNN works extremely well.
We have thus seen that the performance of the SCNN not only greatly exceeds that of the conventional CNN, but that it performs even better than the RCNN presented in the last section. This is due to the fact that an additional evaluation is performed in the low resolution convolutional layers, which increases the accuracy of the classification. We now want to go one step further and see how the performance changes if we replace the convolutional layers with the quaternion convolutional layers presented in section 6.2.

7.2. The Fusion of Ridgelet and Quaternion Filtering

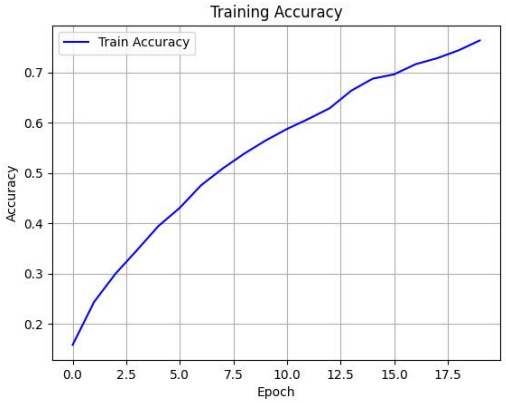
As we saw in the last chapter, the split ridgelet network proves to be extremely efficient in classifying high-resolution color images. Now we want to go one step further and implement the previously introduced quaternion convolution instead of the normal convolution in the low-resolution part of the network. Since the quaternion convolution requires a high computing power and thus a very high computing time, we will reduce the high-resolution part of the network to a size of 64x64 and the low-resolution part to a size of 16x16 pixels. However, the basic structure of the network will remain the same and we will continue to let it learn over 20 epochs with a batch size of 64 on the NWPU dataset. The Code for the quaternion convolution was provided by Johanna Richter. It was created as part of her diploma thesis [40]. For comparison, we will also reduce the previously used SCNN to the same resolution. Here we see the outcome after 18.2 hours:



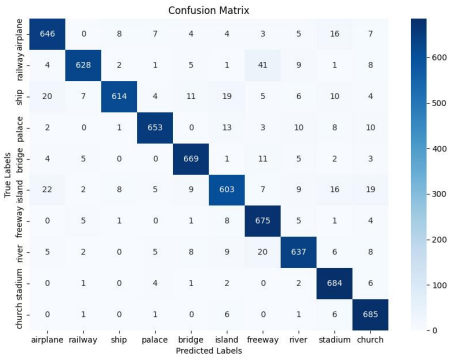
(a) Accuracy over Epochs, QRCNN



(b) Confusion Matrix of QRCNN



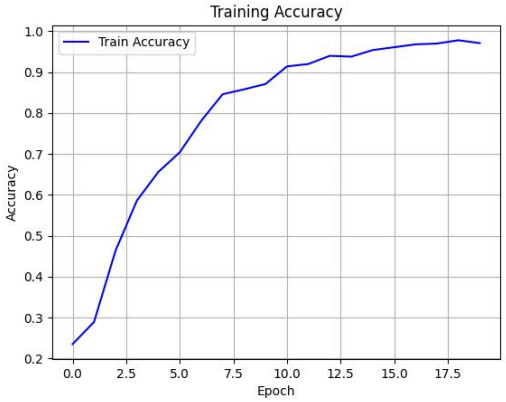
(c) Accuracy over Epochs, SCNN, Image Size 64 × 64



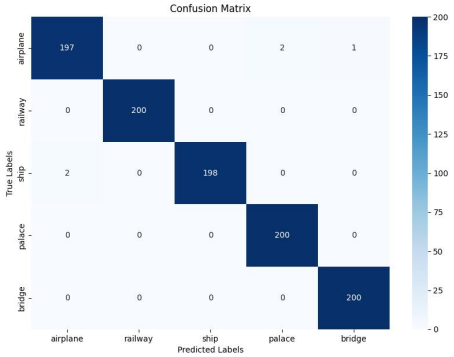
(d) Confusion Matrix of SCNN, Image Size 64 × 64

Figure 7.4.: Learning Comparison of QRCNN and SCNN

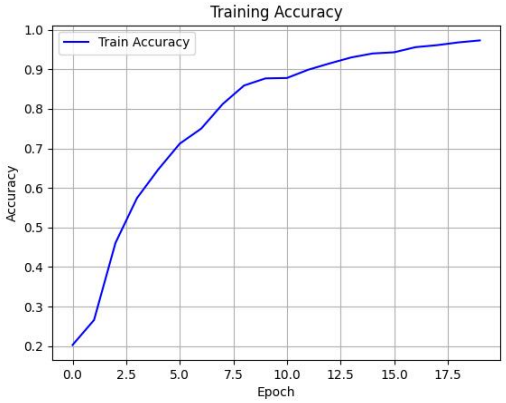
We can see the functionality of the QRCNN in this example. Unfortunately, we cannot yet observe any improvement in the performance of the network, it works just as well as the previously used SCNN. One reason for this could be the low image resolution in the quaternion part of the network. With an image size of 16x16, this will not really come into play. However, as tests on this data set with a higher resolution would take too long due to the complexity, we will adapt the data set. We now consider only the five classes 'airplane', 'railway', 'ship', 'palace' and 'bridge' and consider only 200 images per class. Certainly too few for a network in use, but sufficient for testing. We can therefore increase the resolution to 128x128 in the ridgelet part and 32x32 in the quaternion part and obtain the following results:



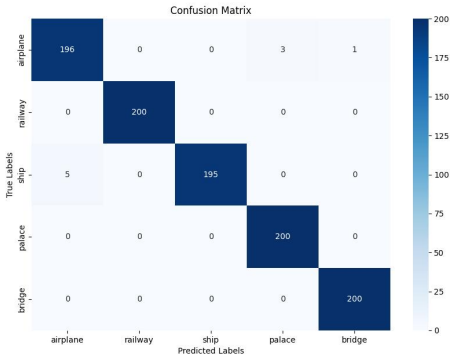
(a) Accuracy over Epochs, QRCNN



(b) Confusion Matrix of QRCNN



(c) Accuracy over Epochs, SCNN, Image Size 128×128



(d) Confusion Matrix of SCNN, Image Size 128×128

Figure 7.5.: Learning Comparison of QRCNN and SCNN

In summary, the results indicate that the SCNN and QRCNN achieve comparable performance. Specifically, the QRCNN's accuracy after 20 epochs is only marginally higher by approximately 0.3%, which is negligible in practical terms. However, a significant difference is observed in computational effort: while the SCNN required 25 minutes for 20 epochs, the QRCNN took 12 hours under the same conditions. This disparity highlights the complexity of quaternion-valued calculations, which demand extensive fine-tuning. Although further experiments were constrained by the high computational cost, the approach shows promise. With future optimizations, the QRCNN could become a valuable tool for specialized image recognition tasks.

8. Conclusion

This thesis has explored a wide range of practical applications and theoretical aspects of ridge functions, with a focus on their role in function theory and artificial neural networks. At the heart of the work was the exploration of the fundamental properties of ridge functions, specifically their approximation capabilities, given by Vugar Ismailov [16] and S.V. Konyagin [22]. One of the significant outcomes of this research was the analysis of a special activation function derived from ridge functions, which enables the definition of neural networks with a minimal number of neurons while still achieving arbitrary precision in approximating functions. This result is particularly relevant in the context of deep learning, where the efficiency of neural network architectures is often a crucial concern.
However, while the theoretical properties of this activation function hold promise, practical challenges arise when it comes to its real-world application. The main difficulty lies in the computational complexity involved in evaluating the special activation function. This makes it less attractive in practice when compared to more commonly used activation functions, which are often simpler to compute. Furthermore, the idealized notion that a network can approximate a given function arbitrarily well, as suggested by the theory, does not always hold true in practice. One of the primary reasons for this discrepancy is the occurrence of local minima during the training process. These local minima can prevent the network from fully approximating the target function, resulting in only partial success in function approximation like we have seen in figure 4.4a in chapter 4.3.
Among the various ridge functions discussed in this work, ridgelets, ridge functions with a wavelet profile along the direction vector, emerged as a particularly noteworthy subclass. Ridgelets have distinct advantages when it comes to approximating functions with $d-1$ dimensional singularities. This capability was demonstrated both theoretically and through experiments presented in chapter 5.1.3. Despite these promising theoretical results, the practical application of ridgelets in neural networks goes beyond their approximation abilities. Specifically, their use as filters in convolutional neural networks (CNNs) presents a significant advantage. By utilizing the ridgelet transformation, which combines the Radon and wavelet transformations, these filters are highly effective at detecting edges and corners in data. This is because the ridgelet transformation captures important geometric features of the data, such as sharp edges, with great precision.
One of the advantages of ridgelet filters over traditional CNN filters is their efficiency in learning. The ridgelet transformation only requires three parameters, namely scaling, shifting, and rotation, to be learned for each filter kernel, which significantly reduces the complexity compared to standard convolutional layers, especially for larger filters, like it is shown in tabular 5.1. Through experimentation, it was shown that ridgelet filters outperform traditional CNN filters, especially when applied to edge-rich and highly detailed images. This was evident in datasets such as the KTH-TIPS, where the ridgelet-based CNN demonstrated superior performance.
Despite their strengths, ridgelet-based CNNs (RCNNs) do face limitations. While RCNNs excel with high-resolution images, where larger filter kernels can be used without a significant loss in computational efficiency, they perform less efficiently with low-resolution images. In these cases, traditional CNNs remain more computationally efficient and can yield better results. This observation led to the development of a hybrid architecture that combines the strengths of both approaches. In chapter 7, we presented a network that processes high-resolution images using

ridgelet filtering, while simultaneously processing low-resolution versions of the same images using standard convolutional layers. This approach originated from work of T Gladima Nisia and S Rajesh [31] and allows the network to recognize both large and small structures in the images, yielding outstanding results, particularly in tasks that require detailed image analysis. Building upon this foundation, this thesis further explored the potential of quaternion convolution as a method for enhancing neural network capabilities. Quaternions, as an extension of complex numbers, provide a natural way to represent rotations in three-dimensional space. This feature was exploited to design a convolution operation that can process color images more efficiently by treating each pixel as a color vector and performing rotations on it. The use of quaternion convolution allows for a more natural representation of color information, which is essential in many image recognition tasks. This work builds on previous studies, including the work of Johanna Richter, who provided the code for quaternion convolution that was used in our experiments.

The combination of ridgelet filtering and quaternion convolution in a single network allowed us to design a system capable of both edge detection and color-based pattern recognition. This dual approach proved successful in classifying complex datasets, demonstrating the networks robustness. However, due to the computational complexity involved in quaternion convolution, further testing was limited. Nevertheless, this technique holds significant promise, and future research could focus on optimizing the network architecture to reduce computational overhead and improve performance further.

In conclusion, this thesis makes a significant contribution to the theoretical understanding of ridge functions and their application in neural networks. It also introduces novel neural network architectures that combine ridgelet filtering and quaternion convolution, opening new possibilities for image recognition tasks. These findings lay the groundwork for future advancements in specialized neural network design, particularly in the fields of high-resolution image processing and color image analysis. Future research may focus on optimizing these networks further, addressing the computational challenges, and exploring additional applications in areas such as computer vision and medical imaging.

Acknowledgement - Danksagung

Mit dieser Danksagung möchte ich all jenen meinen tief empfundenen Dank aussprechen, die mich während der Erstellung dieser Arbeit unterstützt haben.

Mein besonderer Dank gilt Frau Prof. Swanhild Bernstein und Herrn Dr. Dmitrii Legatiuk für ihre engagierte Betreuung meiner Diplomarbeit. Ihre wertvolle Unterstützung, ihre hilfreichen Ratschläge sowie die stets offene Tür für meine Anliegen haben maßgeblich zum Erfolg dieser Arbeit beigetragen.

Herzlich bedanken möchte ich mich auch bei Herrn Prof. Sprungk, Herrn Matthias Werner, Herrn Volker Göhler und Herrn Dr. Martin Reinhardt. Ihre fachkundige Beratung, ihre konstruktiven Anregungen und ihre Geduld bei der Beantwortung meiner Fragen waren von unschätzbarem Wert und haben mir stets geholfen, die richtigen Lösungsansätze zu finden.

Ein weiterer Dank gilt Frau Johanna Luise Richter, die mir ihren Code für das Quaternion-Deep-Learning-Netzwerk zur Verfügung gestellt hat. Durch ihren Beitrag konnte ich mich vertieft mit der Fusion von Ridgelet- und Quaternion-Netzwerken auseinandersetzen und diesen Aspekt in meine Arbeit einfließen lassen.

Abschließend möchte ich meinen Eltern von Herzen danken. Durch ihre bedingungslose Unterstützung und ihren Rückhalt haben sie mir mein Studium in dieser Form überhaupt erst ermöglicht. Ohne sie wäre dieser Weg nicht denkbar gewesen.

List of Figures

Bibliography

[1] Ailurophile. *Butterfly Dataset*. 10.01.2020. URL: https://www.kaggle.com/datasets/veeralakrishna/butterfly-dataset.

[2] Rashid A. Aliev and Vugar E. Ismailov. "A representation problem for smooth sums of ridge functions". In: *Journal of Approximation Theory* 257 (2020), p. 105448. ISSN: 00219045. DOI: 10.1016/j.jat.2020.105448.

[3] Shuroug A. Alowais et al. "Revolutionizing healthcare: the role of artificial intelligence in clinical practice". In: *BMC medical education* 23.1 (2023), p. 689. DOI: 10.1186/s12909-023-04698-z.

[4] Eduardo Bayro-Corrochano. "The Theory and Use of the Quaternion Wavelet Transform". In: *Journal of Mathematical Imaging and Vision* 24.1 (2006), pp. 19–35. ISSN: 0924-9907. DOI: 10.1007/s10851-005-3605-3.

[5] Jasmin Bharadiya. "Artificial Intelligence in Transportation Systems A Critical Review". In: *American Journal of Computing and Engineering* 6.1 (2023), pp. 34–45. DOI: 10.47672/ajce.1487.

[6] Jasmin Praful Bharadiya, Reji Kurien Thomas, and Farhan Ahmed. "Rise of Artificial Intelligence in Business and Industry". In: *Journal of Engineering Research and Reports* 25.3 (2023), pp. 85–103. DOI: 10.9734/JERR/2023/v25i3893.

[7] Varun H. Buch, Irfan Ahmed, and Mahiben Maruthappu. "Artificial intelligence in medicine: current trends and future possibilities". In: *The British journal of general practice : the journal of the Royal College of General Practitioners* 68.668 (2018), pp. 143–144. DOI: 10.3399/bjgp18X695213.

[8] Emmanuel J. Candès. "Ridgelets: estimating with ridge functions". In: *The Annals of Statistics* 31.5 (2003). ISSN: 0090-5364. DOI: 10.1214/aos/1065705119.

[9] Emmanuel J. Candès and David L. Donoho. "Ridgelets: a key to higher-dimensional intermittency?" In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 357.1760 (1999), pp. 2495–2509. ISSN: 1364-503X. DOI: 10.1098/rsta.1999.0444.

[10] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314. ISSN: 0932-4194. DOI: 10.1007/BF02551274.

[11] Persi Diaconis and Mehrdad Shahshahani. "On Nonlinear Functions of Linear Combinations". In: *SIAM Journal on Scientific and Statistical Computing* 5.1 (1984), pp. 175–191. ISSN: 0196-5204. DOI: 10.1137/0905013.

[12] *Feedforward Deep Learning Models · AFIT Data Science Lab R Programming Guide*. 19.06.2018. URL: https://afit-r.github.io/feedforward_DNN.

[13] formerly DPhi. *Data Sprint #25: Flower Recognition | AI Planet (formerly DPhi)*. 28.11.2024. URL: https://aiplanet.com/challenges/61/data-sprint-25-flower-recognition-61/overview/about.

[14] Dóra Göndöcs and Viktor Dörfler. "AI in medical diagnosis: AI prediction & human judgment". In: *Artificial intelligence in medicine* 149 (2024), p. 102769. DOI: 10.1016/j.artmed.2024.102769.

[15] Peter J. Huber. "Projection Pursuit". In: *The Annals of Statistics* 13.2 (1985). ISSN: 0090-5364. DOI: 10.1214/aos/1176349519.

[16] Vugar Ismailov. *Notes on ridge functions and neural networks.* 2020. DOI: 10.48550/arXiv.2005.14125.

[17] Vugar E. Ismailov. "On the representation by linear superpositions". In: *Journal of Approximation Theory* 151.2 (2008), pp. 113–125. ISSN: 00219045. DOI: 10.1016/j.jat.2007.09.003.

[18] Maria Kasinidou, Styliani Kleanthous, and Jahna Otterbacher. "Artificial Intelligence in Everyday Life: Educating the Public Through an Open, Distance-learning Course". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1.* Ed. by Mikko-Jussi Laakso et al. New York, NY, USA: ACM, 2023, pp. 306–312. ISBN: 9798400701382. DOI: 10.1145/3587102.3588784.

[19] I. G. Kazantsev. "Tomographic reconstruction from arbitrary directions using ridge functions". In: *Inverse Problems* 14.3 (1998), pp. 635–645. ISSN: 0266-5611. DOI: 10.1088/0266-5611/14/3/014.

[20] I. G. Kazantsev and I. Lemahieu. "Reconstruction of elongated structures using ridge functions and natural pixels". In: *Inverse Problems* 16.2 (2000), pp. 505–517. ISSN: 0266-5611. DOI: 10.1088/0266-5611/16/2/317.

[21] S. V. Konyagin and A. A. Kuleshov. "On the continuity of finite sums of ridge functions". In: *Mathematical Notes* 98.1-2 (2015), pp. 336–338. ISSN: 0001-4346. DOI: 10.1134/S0001434615070378.

[22] S. V. Konyagin, A. A. Kuleshov, and V. E. Maiorov. "Some Problems in the Theory of Ridge Functions". In: *Proceedings of the Steklov Institute of Mathematics* 301.1 (2018), pp. 144–169. ISSN: 0081-5438. DOI: 10.1134/S0081543818040120.

[23] *KTH-TIPS image databases homepage.* 9.06.2006. URL: https://www.csc.kth.se/cvap/databases/kth-tips/index.html.

[24] A. A. Kuleshov. "On some properties of smooth sums of ridge functions". In: *Proceedings of the Steklov Institute of Mathematics* 294.1 (2016), pp. 89–94. ISSN: 0081-5438. DOI: 10.1134/S0081543816060067.

[25] V. Y. Lin and A. Pinkus. "Fundamentality of Ridge Functions". In: *Journal of Approximation Theory* 75.3 (1993), pp. 295–311. ISSN: 00219045. DOI: 10.1006/jath.1993.1104.

[26] B. F. Logan and L. A. Shepp. "Optimal reconstruction of a function from its projections". In: *Duke Mathematical Journal* 42.4 (1975). ISSN: 0012-7094. DOI: 10.1215/S0012-7094-75-04256-8.

[27] V. E. Maiorov. "Best approximation by ridge functions in L p -spaces". In: *Ukrainian Mathematical Journal* 62.3 (2010), pp. 452–466. ISSN: 0041-5995. DOI: 10.1007/s11253-010-0364-0.

[28] Vitaly Maiorov and Allan Pinkus. "Lower bounds for approximation by MLP neural networks". In: *Neurocomputing* 25.1-3 (1999), pp. 81–91. ISSN: 09252312. DOI: 10.1016/S0925-2312(98)00111-8.

[29] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. ISSN: 0007-4985. DOI: 10.1007/BF02478259.

[30] F. Natterer. *The Mathematics of Computerized Tomography*. Society for Industrial and Applied Mathematics, 2001. ISBN: 978-0-89871-493-7. DOI: 10.1137/1.9780898719284.

[31] T. Gladima Nisia and S. Rajesh. "Extraction of High-level and Low-level feature for classification of Image using Ridgelet and CNN based Image Classification". In: *Journal of Physics: Conference Series* 1911.1 (2021), p. 012019. ISSN: 1742-6588. DOI: 10.1088/1742-6596/1911/1/012019.

[32] K. I. Oskolkov, ed. *Approximation theory. Harmonic analysis. Collected papers dedicated to the memory of Professor Sergei Borisovich Stechkin: Ridge approximation, Chebyshev-Fourier analysis and optimal quadrature formulas*. 1997.

[33] K. I. Oskolkov. "Linear and nonlinear methods of relief approximation". In: *Journal of Mathematical Sciences* 155.1 (2008), pp. 129–152. ISSN: 1072-3374. DOI: 10.1007/s10958-008-9212-2.

[34] K. I. Oskolkov. *Ridge approximation, Chebyshev-Fourier analysis and optimal quadrature formulas*. Moscow, 1997.

[35] *Papers with Code - RESISC45 Dataset*. 28.11.2024. URL: https://paperswithcode.com/dataset/resisc45.

[36] Allan Pinkus. *Ridge Functions*. Cambridge University Press, 2015. ISBN: 9781107124394. DOI: 10.1017/CBO9781316408124.

[37] U. Raghavendra et al. "Brain tumor detection and screening using artificial intelligence techniques: Current trends and future perspectives". In: *Computers in biology and medicine* 163 (2023), p. 107063. DOI: 10.1016/j.compbiomed.2023.107063.

[38] Pranav Rajpurkar et al. "AI in health and medicine". In: *Nature medicine* 28.1 (2022), pp. 31–38. DOI: 10.1038/s41591-021-01614-0.

[39] Milind Rane et al. "AI Algorithm for Image Enhancement". In: *2023 Third International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*. IEEE, 2023, pp. 1–4. ISBN: 978-1-6654-9400-7. DOI: 10.1109/ICAECT57570.2023.10117819.

[40] Johanna Luise Richter. "Quaternion Deep Learning". Diploma Thesis. TU Bergakademie Freiberg, 2022.

[41] Robert Kinzler. *AI AND YOU: HOW ARTIFICIAL INTELLIGENCE SHAPES OUR DAILY LIVES*. 2023. DOI: 10.13140/RG.2.2.26650.82881.

[42] S. J. Sangwine. "Fourier transforms of colour images using quaternion or hypercomplex, numbers". In: *Electronics Letters* 32.21 (1996), p. 1979. ISSN: 00135194. DOI: 10.1049/el:19961331.

[43] Pawankumar Sharma. "Active Noise Cancellation in Microsoft Teams Using AI & NLP Powered Algorithms". In: *International Journal of Computer Science and Information Technology* 15.1 (2023), pp. 31–42. ISSN: 09754660. DOI: 10.5121/ijcsit.2023.15103.

[44] Amith Singhee and Rob A. Rutenbar. *Novel Algorithms for Fast Statistical Analysis of Scaled Circuits*. Vol. 46. Dordrecht: Springer Netherlands, 2009. ISBN: 978-90-481-3099-3. DOI: 10.1007/978-90-481-3100-6.

[45] Takialddin Al Smadi et al. "Artificial Intelligence for Speech Recognition Based on Neural Networks". In: *Journal of Signal and Information Processing* 06.02 (2015), pp. 66–72. ISSN: 2159-4465. DOI: 10.4236/jsip.2015.62006.

[46] Neha Soni et al. "Artificial Intelligence in Business: From Research and Innovation to Market Deployment". In: *Procedia Computer Science* 167 (2020), pp. 2200–2210. ISSN: 18770509. DOI: 10.1016/j.procs.2020.03.272.

[47] J. P. Sproston and D. Strauss. “Sums of Subalgebras of C (X)”. In: *Journal of the London Mathematical Society* s2-45.2 (1992), pp. 265–278. ISSN: 00246107. DOI: 10.1112/jlms/s2-45.2.265.

[48] Y. Sternfeld. “Uniformly separating families of functions”. In: *Israel Journal of Mathematics* 29.1 (1978), pp. 61–91. ISSN: 0021-2172. DOI: 10.1007/BF02760402.

[49] Sahul Mohan Tarachandy and Aravinth J. “Enhanced Local Features using Ridgelet Filters for Traffic Sign Detection and Recognition”. In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. IEEE, 2021, pp. 1150–1156. ISBN: 978-1-6654-2867-5. DOI: 10.1109/ICESC51422.2021.9532967.

[50] Hiral Thadeshwar et al. “Artificial Intelligence based Self-Driving Car”. In: *2020 4th International Conference on Computer, Communication and Signal Processing (ICCCSP)*. IEEE, 2020, pp. 1–5. ISBN: 978-1-7281-6509-7. DOI: 10.1109/ICCCSP49186.2020.9315223.

[51] Wikipedia, ed. *Artificial neuron*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Artificial_neuron&oldid=1251139734.

[52] Yi Xu et al. “QWT: Retrospective and New Applications”. In: *Geometric Algebra Computing*. Ed. by Eduardo Bayro-Corrochano and Gerik Scheuermann. London: Springer London, 2010, pp. 249–273. ISBN: 978-1-84996-107-3. DOI: 10.1007/978-1-84996-108-0{\textunderscore}13.

[53] Shuyuan Yang, Min Wang, and Licheng Jiao. “A New Adaptive Ridgelet Neural Network”. In: *Advances in Neural Networks — ISNN 2005*. Ed. by Jun Wang, Xiaofeng Liao, and Zhang Yi. Vol. 3496. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 385–390. ISBN: 978-3-540-25912-1. DOI: 10.1007/11427391{\textunderscore}61.

[54] Shuyuan Yang, Min Wang, and Licheng Jiao. “Incremental constructive ridgelet neural network”. In: *Neurocomputing* 72.1-3 (2008), pp. 367–377. ISSN: 09252312. DOI: 10.1016/j.neucom.2008.01.001.

[55] Xuanyu Zhu et al. *Quaternion Convolutional Neural Networks*. URL: http://arxiv.org/pdf/1903.00658v1.

A. Matlab Code for the special Activation Function

```
function z = sigma(x)
    %% Initialization of the output variable and parameters
    z = 0;
    a = -1;
    b = 1;
    d = b - a;
    lam = 0.4;
    n = 1000;
    C = 10;
    x = single(x);

    %% Helper functions

    % Helper function for boundary control
    function y = h(z)
        y = 1 - min(0.5, lam) / (1 + log(z - d + 1));
    end

    % Calkin-Wilf sequence, computes rational number on position
       k
    function y = CW(k)
        y = 1;
        k = k - 1;
        [N, D] = rat(y);
        for i = 1:k
            frac = 1 / (2 * floor(y) - y + 1);
            [N, D] = rat(frac);
            y = N / D;
        end
    end

    % Function for rational number enumeration
    function y = r(k)
        if k == 0
            y = 0;
        elseif mod(k, 2) == 0
            y = CW(round(k / 2));
        else
            y = -CW(round((k + 1) / 2));
        end
    end
```

```
% Enumeration of all monic polynomials with rational
   coefficients
function y = u(k)
    p = 1;
    m = cfrac(CW(k), 100);
    if k == 1
        p = 1;
    elseif m(1) == fix(m(1)) && (m(1) ~= 1)
        p = poly([1, r(CW(k) - 2)]);
    elseif length(m) == 2
        p = poly([1, r(m(2) - 2), r(m(1))]);
    elseif length(m) > 2
        plist = [r(m(1))];
        for i = 2:(length(m) - 1)
            plist = [plist, r(m(i) - 1)];
        end
        plist = [plist, r(m(end) - 2), 1];
        plist = fliplr(plist);
        p = poly(plist);
    end
    y = p;
end

% Calculation of B1
function B1 = B1(lst)
    B1 = lst(1);
    for i = 1:length(lst)
        B1 = B1 + (lst(i) - abs(lst(i))) / 2;
    end
end

% Calculation of B2
function B2 = B2(lst)
    B2 = lst(1) + 1;
    for i = 1:length(lst)
        B2 = B2 + (lst(i) + abs(lst(i))) / 2;
    end
end

% Explicit Calculation of the sequence M_k
function y = M(k)
    y = h((2 * k + 1) * d);
end

% Explicit Calculation of the sequence a_k
function y = af(k)
    if k == 1
        y = 1 / 2;
    else
        y = ((1 + 2 * M(k)) * B2(u(k)) - (2 + M(k)) * B1(u(k)
           )) / (3 * (B2(u(k)) - B1(u(k))));
```

```
    end
end

% Explicit Calculation of the sequence b_k
function y = bf(k)
    if k == 1
        y = h(3 * d) / 2;
    else
        y = (1 - M(k)) / (3 * (B2(u(k)) - B1(u(k))));
    end
end

% Calculation for x in [(2k-1)d,2kd]
function y = presigma(k, x)
    p = u(k);
    y = af(k) + bf(k) * polyval(p, x / d - 2 * k + 1);
end

% Helper function for the smooth transition function
function y = betad(x)
    if x > 0
        y = exp(-1 / x);
    else
        y = 0;
    end
end

% Smooth transition function
function y = beta(a, b, x)
    y = betad(b - x) / (betad(b - x) + betad(x - a));
end

% Explicit Calculation of the sequence K_k
function y = K(k)
    y = (presigma(k, 2 * k * d) + presigma(k, (2 * k + 1) * d
       )) / 2;
end

%% Main Calculation

% Defining the Intervals based on n
N =3*n;
A=zeros(N,2);
for l = 1:n

        starti=1+3*(l-1);
        A(starti,1)=2*l-1;
        A(starti+1,1)=2*l;
        A(starti+2,1)=2*l+0.5;
        A(starti,2)=2*l;
        A(starti+1,2)=2*l+0.5;
```

```
            A(starti+2,2)=2*l+1;

    end

    % Stretch the Intervals with a given d
    A=d.*A;

    % Find the interval where x lies in
    Akl = find(A(:,1)<=x);
    Akr = find(A(:,2)>x);
    kl = max(Akl);
    kr = min(Akr);



    % Determine z based on interval
    if isempty(kl) || isempty(kr)
        z = (1 - betad(d - x)) * (1 + M(1)) / 2;
    else
        intervalType = mod(kl, 3);
        k = ceil(kl / 3);
        if intervalType == 2 % presigma interval
            z = presigma(k, x);
        elseif intervalType == 1 % beta interval 1
            eps = (1 - M(k)) / 6;
            delta = min(eps * d / (bf(k) * C), d / 2);
            p = u(k);
            z = K(k) - beta(2 * k * d, 2 * k * d + delta, x) * (K
               (k) - af(k) - bf(k) * polyval(p, x / d - 2 * k +
               1));
        elseif intervalType == 0 % beta interval 2
            p = u(k);
            epsq = (1 - M(k + 1)) / 6;
            deltaq = min(epsq * d / (bf(k + 1) * C), d / 2);
            z = K(k) - (1 - beta((2 * k + 1) * d - deltaq, (2 * k
                + 1) * d, x)) * (K(k) - af(k) - bf(k) * polyval(p
               , x / d - 2 * k - 1));
        else
            z = (1 - betad(d - x)) * (1 + M(1)) / 2;
        end
    end
    % Make z single precision for network calculations
    z = single(z);
end
```

B. Python Code for the splitted Network

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from torch.cuda.amp import GradScaler, autocast
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import os
import math
import gc
import time
import seaborn as sns
from sklearn import metrics
from torchsummary import summary

high = 64
low = 16

# Custom Dataset
class NWPU_RESISC45(Dataset):
    def __init__(self, root_dir, classes, transform=None):
        self.root_dir = root_dir
        self.classes = classes
        self.transform = transform
        self.img_paths = []
        self.labels = []

        for class_idx, class_name in enumerate(classes):
            class_dir = os.path.join(root_dir, class_name)
            for img_name in os.listdir(class_dir):
                self.img_paths.append(os.path.join(class_dir,
                img_name))
                self.labels.append(class_idx)

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        img_path = self.img_paths[idx]
        label = self.labels[idx]
        image = Image.open(img_path).convert('RGB')
```

```
        if self.transform:
            image = self.transform(image)

        return image, label
# Ridgelet Layer
class RidgeletLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size,
                         num_kernels=1):
        super(RidgeletLayer, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.num_kernels = num_kernels

        # Initialize parameters for ridgelet kernels
        self.scales =
        nn.Parameter(torch.ones(out_channels * num_kernels))
        self.directions =
        nn.Parameter(torch.linspace(0, math.pi, out_channels *
        num_kernels))
        self.positions =
        nn.Parameter(torch.zeros(out_channels * num_kernels))

    def ridgelet_function(self, x1, x2, direction, scale, position):
        transformed_coordinate = (x1 * torch.cos(direction)
        + x2 * torch.sin(direction) - position) / scale
        ridgelet_value = torch.exp(-transformed_coordinate ** 2 / 2)
        - 0.5 * torch.exp(-transformed_coordinate ** 2 / 8)
        return ridgelet_value

    def create_ridgelet_kernel(self, device):
        kernel = torch.zeros((self.out_channels, self.in_channels,
        self.kernel_size, self.kernel_size), device=device)
        center = self.kernel_size // 2
        x1, x2 = torch.meshgrid(torch.arange(self.kernel_size)
        - center,
        torch.arange(self.kernel_size) - center)
        x1, x2 = x1.float().to(device), x2.float().to(device)

        for out_ch in range(self.out_channels):
            for k in range(self.num_kernels):
                idx = out_ch * self.num_kernels + k
                direction = self.directions[idx]
                scale = self.scales[idx]
                position = self.positions[idx]

                ridgelet_kernel = self.ridgelet_function(x1, x2,
                direction, scale, position)
                kernel[out_ch, :, :, :] += ridgelet_kernel
```

```
        return kernel


    def forward(self, x):
        device = x.device
        kernel = self.create_ridgelet_kernel(device)
        x = F.conv2d(x, kernel, padding=self.kernel_size // 2)
        return x

# Mixed Resolution CNN

class MixedResolutionCNN(nn.Module):
    def __init__(self, num_classes=10, num_kernels=1):
        super(MixedResolutionCNN, self).__init__()
        self.num_classes = num_classes

        # Low-resolution processing
        self.low_res_conv = nn.Conv2d(3, 16, kernel_size=3, stride=1,
        padding=1)
        self.low_res_pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # High-resolution processing
        self.high_res_ridgelet = RidgeletLayer(in_channels=3,
        out_channels=16, kernel_size=25, num_kernels=10)
        self.high_res_pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Calculate the output size after processing
        self._conv_output_size = self._get_conv_output_size()
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(self._conv_output_size, 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, self.num_classes)

    def _get_conv_output_size(self):
        # Dummy inputs to compute the output shape
        # Low-resolution dummy input
        dummy_low_res = torch.zeros(1, 3, low, low)
        # High-resolution dummy input
        # High-resolution dummy input
        dummy_high_res = torch.zeros(1, 3, high, high)

        # Low-resolution feature extraction
        x_low = F.relu(self.low_res_conv(dummy_low_res))
        x_low = self.low_res_pool(x_low)
        x_low = x_low.view(x_low.size(0), -1)

        # High-resolution feature extraction
        x_high = F.relu(self.high_res_ridgelet(dummy_high_res))
        x_high = self.high_res_pool(x_high)
```

```
        x_high = x_high.view(x_high.size(0), -1)

        # Return the total number of features after concatenating
        return x_low.size(1) + x_high.size(1)

    def forward(self, x):
        # Low-resolution branch
        x_low = F.interpolate(x, size=(low, low), mode='bilinear',
        align_corners=False)
        x_low = F.relu(self.low_res_conv(x_low))
        x_low = self.low_res_pool(x_low)
        x_low = x_low.view(x_low.size(0), -1)  # Flatten the tensor

        # High-resolution branch
        x_high = F.relu(self.high_res_ridgelet(x))
        x_high = self.high_res_pool(x_high)
        # Flatten the tensor
        x_high = x_high.view(x_high.size(0), -1)

        # Concatenate both branches
        x_fused = torch.cat((x_low, x_high), dim=1)

        # Fully connected layers
        x_fused = F.relu(self.dropout(self.fc1(x_fused)))
        x_fused = F.relu(self.fc2(x_fused))
        x_fused = self.fc3(x_fused)

        return x_fused


def print_model_params(model):
    print("Model parameters:")
    for name, param in model.named_parameters():
        if param.requires_grad:
            print(f"Name: {name}, Shape: {param.shape}")
# Function for model training
def train_model(trainloader, device, epochs=25,
        accumulation_steps=4):
    model = MixedResolutionCNN(num_classes=10).to(device)
    optimizer = optim.SGD(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
    patience=3, factor=0.5)
    model.train()
    scaler = GradScaler()

    train_losses = []
    train_accuracies = []
    start_time = time.time()

    for epoch in range(epochs):
```

```
        print(f"\nEpoch {epoch+1}/{epochs}")
        epoch_start_time = time.time()  # Start time for the epoch
        running_loss = 0.0
        correct_predictions = 0
        total_samples = 0
        optimizer.zero_grad()

        for batch_idx, (inputs, labels) in enumerate(trainloader):
            inputs, labels = inputs.to(device), labels.to(device)

            with autocast():
                outputs = model(inputs)
                loss =
                criterion(outputs, labels) / accumulation_steps

            scaler.scale(loss).backward()

            if (batch_idx + 1) % accumulation_steps == 0:
                scaler.step(optimizer)
                scaler.update()
                optimizer.zero_grad()

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total_samples += labels.size(0)
            correct_predictions += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(trainloader.dataset)
        epoch_accuracy = correct_predictions / total_samples
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_accuracy)

        scheduler.step(epoch_loss)  # Use loss for scheduler step
        torch.cuda.empty_cache()
        gc.collect()

        # Measure and print the epoch duration
        epoch_duration = time.time() - epoch_start_time
        print(f"Epoch [{epoch+1}/{epochs}] - Loss: {epoch_loss:.4f},
        Accuracy: {epoch_accuracy:.4f}, Duration: {epoch_duration:.2f}
        seconds")


    # Measure and print the total training time
    total_training_time = time.time() - start_time
    print(f"Total training time: {total_training_time:.2f}
    seconds")

    return model, train_losses, train_accuracies
```

```
# Function to plot confusion matrix
def plot_confusion_matrix(model, dataloader, classes, device):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    cf_matrix = metrics.confusion_matrix(all_labels, all_preds)

    plt.figure(figsize=(10, 7))
    ax = sns.heatmap(cf_matrix, annot=True, cmap='Blues', fmt='g',
    xticklabels=classes, yticklabels=classes)
    ax.set_title('Confusion Matrix')
    ax.set_xlabel('Predicted Labels')
    ax.set_ylabel('True Labels')
    plt.show()

def main():
    # Device setup
    device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")

    # Define transformations for data augmentation
    transform = transforms.Compose([
        # Resize to a fixed size
        transforms.Resize((high, high)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # Directory for NWPU-RESISC45 dataset
    # Path to where the dataset is located
    root_dir = r"C:\Users\MaxSc\Desktop\dataset\NWPU-RESISC45"

    # Define the classes to use
    classes = ['airplane', 'railway', 'ship', 'palace', 'bridge',
    'island', 'freeway', 'river', 'stadium', 'church']
    # Load the dataset
    dataset = NWPU_RESISC45(root_dir=root_dir, classes=classes,
    transform=transform)

    # Create DataLoader
    trainloader = DataLoader(dataset, batch_size=64, shuffle=True,
    num_workers=4)
```

```
    # Training the model
    num_kernels = 8  # Adjust the number of kernels here
    model, train_losses, train_accuracies = train_model(trainloader,
    device, epochs=20, accumulation_steps=4)
    #summary(model, input_size=(3, high, high), device=str(device))
    # Plot training results
    plt.figure()
    plt.plot(train_accuracies, label='Train Accuracy', color='b')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot confusion matrix after training
    plot_confusion_matrix(model, trainloader, classes, device)

if __name__ == "__main__":
    main()
```