Inhaltsverzeichnis

0.1 Ridge Functions

Ridge functions are a special class of multivariable functions that can be represented using a direction vector a and a univariate function g.

Definition 0.1.1. *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, $f(x) = g(x \cdot a)$.*

One of the prominent properties of ridge functions is summarized in the following lemma:

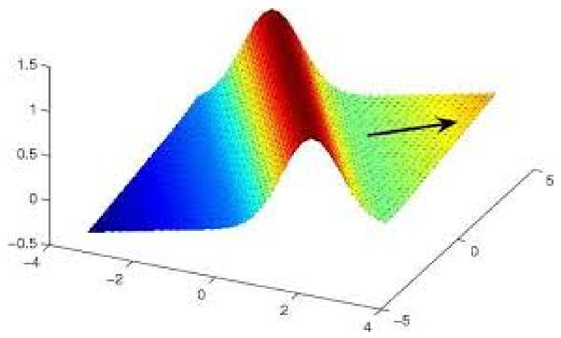
Lemma 0.1.1. *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a ridge function, $g : \mathbb{R} \to \mathbb{R}$, and $a \in \mathbb{R}^d$ a direction vector such that $f(x) = g(x \cdot a)$.*
Then the function value of f is constant over every hyperplane with $a \cdot x = c \ \ \forall c \in \mathbb{R}$.

Beweis. 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

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

$\square$

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



As we will see later, ridge functions possess some very good density properties in other function spaces.

0.1.1 Ridge Function Spaces

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

Definition 0.1.2 (Ridge Function Spaces)**.** *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., $a^r = \lambda a^{r-1}$. 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 $h_i : X \to \mathbb{R}$. 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 ANN functions.
A question that arises with the function spaces introduced above is how well arbitrary functions $f : \mathbb{R}^d \to \mathbb{R}$ can be approximated with them. 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 when an arbitrary function f lies in $\mathcal{R}(a^1, ..., a^r)$. A first result is provided by the following proposition:

Proposition 0.1.1. *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 $p(\frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_d})$. 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 0.1.2. *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.

0.1.2 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 functions over a set $X \subset \mathbb{R}^d$ and correspondingly the subfunction 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. The question now arises when $\mathcal{R}_b(a^1, ..., a^r) = B(X)$, $\mathcal{R}_c(a^1, ..., a^r) = C(X)$, or $\mathcal{R}(a^1, ..., a^r) = T(X)$?
To answer the first two questions, we first need the following set construction:

$$\tau_i(Z) = \{x \in Z : |p_i^{-1}(p_i(x)) \cap Z| \geq 2\} \tag{0.1}$$

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

Proposition 0.1.2 (Sternfeld)**.** *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 $C(\mathbb{R}^d)$, 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., $a \in S^{n-1}$. 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 0.1.3 (Cycle)**.** *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 0.1.3 (Sternfeld)**.** *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 0.1.3. *Let $X \subset \mathbb{R}^d$ be cycle-free, 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.*

Beweis. The forward direction is obvious, as the functional $G_{p,\lambda}$ annihilates all elements of $\mathcal{B}(h_1, ..., h_r; X)$. Now for the reverse direction, we introduce the following notation:

$$Y_i = h_i(X),\ i = 1, ..., r; \Omega = Y_1 \cup ... \cup Y_r,$$

and the set system

$$\mathcal{L} = \{Y = \{y_1, ..., y_r\} : \text{ if there exists } x \in X \text{ such that } h_i(x) = y_i,\ i = 1, ..., r\}.$$

Here, $\mathcal{L}$ is not a subset of Ω but a set of specific subsets of Ω.
Moving forward, all points x associated with Y according to the above definition are termed (*)-points of Y.
wird fortgesetzt □

Definition 0.1.4 (Minimal Cycles)**.** *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 0.1.3, 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 0.1.4. *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 0.1.5. *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 $T(X) = \mathcal{B}(h_1, ..., h_r; X)$.

Beweis. 1) The forward direction of the proof follows from the definition of $G_{p,\lambda}$. Now we proceed with proving the reverse direction:
For this, according to Lemma ..., it suffices to show that if $G_{p,\lambda}(f) = 0$ for every cycle-vector pair $\langle p, \lambda\rangle$ of X, then f lies in $\mathcal{B}(X)$. Consider an interval system $\{(a_i, b_i) \subset \mathbb{R}\}_{i=1}^r$ such that $(a_i, b_i) \cap (a_i, b_i) = \emptyset$ for all $i, j \in \{1, ..., r\}$, $i \neq j$. For $i = 1, ..., r$, let τ_i be the bijective map from $\mathbb{R}$ to (a_i, b_i). We now define the following functions on X:

$$h'_i(x) = \tau_i(h_i(x)), \quad i = 1, ..., r$$

Thus, every cycle with respect to $h_1, ..., h_r$ is also a cycle with respect to $h'_1, ..., h'_r$, and vice versa. Furthermore, $h'_i(X) \cap h'_j(X) = \emptyset$, for all $i, j \in \{1, ..., r\}$, $i \neq j$. According to Lemma..., we have:

$$f(x) = g'_1(h'_1(x)) + ... + g'_r(h'_r(x)),$$

where $g'_1, ..., g'_r$ are univariate functions depending on f. Thus, we obtain:

$$f(x) = g'_1(\tau_1(h_1(x))) + ... + g'_r(\tau_r(h_r(x)))$$

Therefore, $f \in \mathcal{B}(X)$.
2) Let $f : X \to \mathbb{R}$ be any function. First, assume that $h_i(X) \cap h_j(X) = \emptyset$ for all $i, j \in \{1, ..., r\}$, $i \neq j$.
wird fortgesetzt □

Furthermore, the following theorem holds:

Theorem 0.1.6. $\mathcal{B}(h_1, ..., h_r; X) = T(X)$ *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 0.1.6.1. *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 0.1.6.2. $\mathcal{R}(a^1, ..., a^r; X) = T(X)$ *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 0.1.5. *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 0.1.7. *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.*

0.1.3 Function Properties

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

$$f(x) = \sum_{i=1}^{r} g_i(a^i \cdot x) \tag{0.2}$$

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 $g_i \in W(\Delta_i)$.
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 inclusion $f \in B_{(\alpha,\beta)}$ and $r \in C(\alpha, \beta)$ and 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 0.1.8. *If a function f with $f \in C^k(\mathbb{R}^d)$, $k \in \mathbb{Z}^+$ has the form as in 0.2 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:

Theorem 0.1.9. *Let $X \subset \mathbb{R}$ be a non-empty, open set and W be one of the function classes D^k 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 0.2, 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 0.1.10. *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 0.2 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 0.1.11. *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 0.2 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 0.2 implies the local boundedness of g_i on Δ_i.

0.1.4 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 0.1.6 (Best Approximation)**.** *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 $n = 1, 2, ...$

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 $f \in L_2(\mathbb{B}^2)$.

Theorem 0.1.12. *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 0.1.13. *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.

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 $k_1, \dots, k_d$ be non-negative integers and $k = k_1 + \dots + k_d$ a positive integer. Then, the mixed differentiation operator applied to a function $f : \mathbb{R}^d \to \mathbb{R}$ is defined as

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

We now consider a class of specific Sobolev functions:

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

With these conditions, the following theorem holds:

Theorem 0.1.14. *Let $d \geq 2$, $k > 0$, and p, q be arbitrary numbers with $1 \leq p \leq q \leq \infty$. Then,*

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

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 0.1.15. *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^k 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 $C(\mathbb{I}_n)$.

Generalized Ridge Functions

One form of generalizing ridge functions is to replace direction vectors with matrices. A generalized ridge function then has the form $f(x) = g(Ax)$, where $f : \mathbb{R}^d \to \mathbb{R}$ and $A \in \mathbb{R}^{m \times d}$ with $1 \leq d < m$. We will assume that the matrix A always has rank m. Furthermore, we continue to consider f as a sum of the form

$$f(x) = \sum_{i=1}^r g_i(A^i x).$$

The matrices A^i are of the same size.

We now want to consider the same problem as above: If f belongs to a certain smoothness class $W(\mathbb{R}^d)$, under what conditions can we then assume that $g_i \in W(\mathbb{R}^m)$?

To formulate results for this, we need the function class B defined in the first section, or its multidimensional analogue. With B^m, we denote the set of all functions $f : \mathbb{R}^m \to \mathbb{R}$ that are invariant under the differential operator $\Delta_{\mathbf{h}} f(x) = f(x+\mathbf{h}) - f(x)$ for all $\mathbf{h} \in \mathbb{R}^m$. It also holds that if $f \in B^m$, $g \in C(\mathbb{R}^m)$, and the function $h := f - g$ satisfies the multidimensional Cauchy equation $h(x+y) = h(x) + h(y)$ for all $x, y \in \mathbb{R}^m$, then h is a linear function, meaning there exists $a \in \mathbb{R}^m$ such that $h(x) = a \cdot x$.

Now we can formulate the following theorems:

Theorem 0.1.16. *Let $f \in C^k(\mathbb{R}^d)$, $k \in \mathbb{Z}^+$ be a function of the form $f(x) = \sum_{i=1}^r g_i(A^i x)$, and let the $2m$ rows of matrices A^i and A^j be linearly independent for all pairs $i \neq j$. If the functions g_i belong to a class B^m, then $g_i \in C^k(\mathbb{R}^m)$ for all $i = 1, \ldots, r$.*

Theorem 0.1.17. *Let $f \in C^k(\mathbb{R}^d)$, $k \in \mathbb{Z}^+$ be a function of the form $f(x) = \sum_{i=1}^r g_i(A^i x)$ with $f \equiv 0$ over $\mathbb{R}^d$, and let the $2m$ rows of matrices A^i and A^j be linearly independent for all pairs $i \neq j$. If all functions g_i belong to a class B^m, then all functions g_i are polynomials, at most of degree $r-1$ in m variables.*

0.1.5 Uniqueness of Sum Representations by Ridge Functions

Now we want to address the question of how unique the ridge sum representations are. That is, if for a multivariate function f, we have

$$f(x) = \sum_{i=1}^r g_i(a^i \cdot x) = \sum_{j=1}^l h_j(b^j \cdot x),$$

what statements can we make about $\{g_i\}_{i=1}^r$ and $\{a^i\}_{i=1}^r$ in relation to $\{h_j\}_{j=1}^l$ and $\{b^j\}_{j=1}^l$? Due to linearity, the problem can be reduced as follows:

If it holds that

$$f(x) = \sum_{i=1}^r g_i(a^i \cdot x) = 0,$$

for all $x \in \mathbb{R}^d$, what statements can we make about $\{g_i\}_{i=1}^r$? A first answer to these questions is given by the following theorem:

Theorem 0.1.18. *Let $m \in \mathbb{N}$, the vectors a^i be linearly independent, and $g_i \in C(\mathbb{R})$, $i = 1, \ldots, m$. If*

$$f(x) = \sum_{i=1}^r g_i(a^i \cdot x) = 0,$$

for all $x \in \mathbb{R}^d$, then the g_i are polynomials, at most of degree $m-1$.

Another emerging question is about the relationships between functions that are differently smooth. Specifically, we want to analyze the following problem:

Let

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

with $f \in C^k(\mathbb{R}^d)$. Do there exist functions $\tilde{g}_i \in C^l(\mathbb{R})$ such that

$$f(x) = \sum_{i=1}^r \tilde{g}_i(a^i \cdot x)?$$

And what relationships then exist between k and l? This is answered by the following theorem:

Theorem 0.1.19. *Let f be given in the form $f(x) = \sum_{i=1}^{r} g_i(a^i \cdot x)$ and $f \in C^k(\mathbb{R}^d)$ for $k \geq m-1$. If in addition, $g_i \in L^1_{loc}(\mathbb{R})$ for all $i = 1, \dots, m$, then $g_i \in C^k(\mathbb{R})$.*

Beweis. We select a fixed $r \in \{1, \dots, m\}$ and define for each $j \in \{1, \dots, m\}, j \neq r$ $c^j \in \mathbb{R}^d$ such that

$$c^j \cdot a^j = 0 \text{ and } c^j \cdot a^r \neq 0$$

To be continued. □

0.2 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.

0.2.1 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 0.2.1. *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,

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

and

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

With wavelets, we can define the continuous wavelet transform.

Definition 0.2.2. *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:*

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

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:

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

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.

Definition 0.2.3 (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 $A = B$.

0.2.2 The Ridgelet Transform

Having described the basics of wavelets and the wavelet transform, we now move on to the ridgelet transform.

Definition 0.2.4. *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.

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, the 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. 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:

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

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 $\hat{\psi}(\lambda) = \chi_{1\le|\lambda|\le 2}(\lambda)$. 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:

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

Moreover, using Plancherel's theorem, we have:

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

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

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

Thus, the set

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

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

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

0.2.3 Orthonormal Ridgelets

unvollständig We will now construct a basis of orthonormal ridgelets, combining ridge functions and wavelets. This construction is more complex than the theory of ridge functions itself, but it is very useful for function approximation.

To begin, we require a specific class of wavelets known as Meyer wavelets.

Definition 0.2.5 (Meyer Wavelets). *The Meyer wavelet is an infinitely differentiable function defined in terms of the function ν in the frequency domain as follows:*

$$\Psi(\omega) = \begin{cases} \frac{1}{\sqrt{2\pi}} \sin\left(\frac{\pi}{2}\nu\left(\frac{3|\omega|}{2\pi} - 1\right)\right) e^{j\omega/2}, & \text{for } \frac{2\pi}{3} < |\omega| < \frac{4\pi}{3} \\ \frac{1}{\sqrt{2\pi}} \sin\left(\frac{\pi}{2}\nu\left(\frac{3|\omega|}{4\pi} - 1\right)\right) e^{j\omega/2}, & \text{for } \frac{4\pi}{3} < |\omega| < \frac{8\pi}{3} \\ 0, & \text{otherwise} \end{cases}$$

where

$$\nu(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } 0 < x < 1 \\ 1, & \text{for } x > 1 \end{cases}$$

We proceed by constructing a basis for $L^2(\mathbb{R}^2)$ using ridge functions and wavelets.

Let $\Psi_{j,k}(t) \equiv \Psi_{j,k}(t,m)(j,k \in \mathbb{Z})$ be an orthonormal basis of Meyer wavelets over the discrete circle with $-m/2 \leq t < m/2$ and m points. For indices, we have $J_0 \leq j < \log_2(m)$ and $0 \leq k < 2^j$. These wavelets are defined via inverse Fourier transformation:

$$\Psi_{j,k}(t) = \sum_{\omega=-m/2}^{m/2-1} c_\omega^{j,k} e^{(\frac{i2\pi}{m})\omega t}$$

with a predefined sequence $(c_\omega^{j,k})$. This formula holds for all $t \in \mathbb{R}$.

Next, we introduce the fractional-differentiated Meyer wavelets by defining

$$\delta_\omega = \begin{cases} \sqrt{\frac{2\omega}{m}}, & \text{for } \omega \neq 0 \\ \sqrt{\frac{1}{4m}}, & \text{for } \omega = 0 \end{cases}$$

and applying this as a multiplier to $\Psi_{j,k}(t)$, yielding

$$\tilde{\Psi}_{j,k}(t) = \sum_{\omega=-m/2}^{m/2-1} \delta_\omega \cdot c_\omega^{j,k} e^{(\frac{i2\pi}{m})\omega t}$$

This is called the fractional-differentiated or normalized Meyer wavelet.

We now define the following set:

$$\{w_{i_0,l}^0(\theta),\ l = 0, ..., 2^{i_0} - 1;\ \ w_{i,l}^1(\theta),\ i \geq i_0,\ l = 0, ..., 2^i - 1\}$$

consisting of periodized Lemariè scaling functions $\{w_{i_0,l}^0(\theta),\ l = 0, ..., 2^{i_0} - 1\}$ and periodized Meyer wavelets $\{w_{i,l}^1(\theta),\ i \geq i_0,\ l = 0, ..., 2^i - 1\}$. This set forms an orthonormal basis in $L^2[0, 2\pi)$.

Finally, we define the ridgelets $\rho_\lambda(x),\ \lambda = (j,k;i,l,\epsilon)$ for $x \in \mathbb{R}^2$ in the frequency domain:

$$\hat{\rho}_\lambda(\xi) = |\xi|^{-1/2} \left(\hat{\Psi}_{j,k}(|\xi|) w_{i,l}^\epsilon(\theta) + \hat{\Psi}_{j,k}(-|\xi|) w_{i,l}^\epsilon(\theta + \pi) \right) /2 \tag{0.3}$$

Through inverse Fourier transformation, we obtain the system $(\rho_\lambda)_{\lambda \in \Lambda}$, which forms a complete orthonormal basis in $L^2(\mathbb{R}^2)$.

The current representation of the functions ρ_λ does not yet clearly relate to ridgelets. However, they can also be defined differently. Let $\Psi_{j,k}^+(t)$ be defined as:

$$\Psi_{j,k}^+(t) = \frac{1}{2\pi} \int_{\mathbb{R}} |\omega|^{1/2} \hat{\Psi}_{j,k}(|\omega|) e^{i\omega t} d\omega$$

Then, for $x \in \mathbb{R}^2$, we have:

$$\rho_\lambda(x) = \frac{1}{4\pi} \int_0^{2\pi} \Psi_{j,k}^+(x_1 \cos(\theta) + x_2 \sin(\theta)) w_{i,l}^\epsilon(\theta) d\theta$$

Here, $\Psi_{j,k}^+(x_1 \cos(\theta) + x_2 \sin(\theta))$ is again a ridge function.

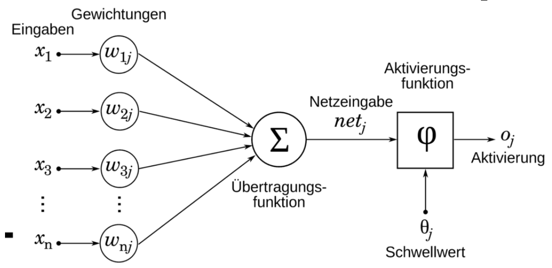
Insert further text here

0.3 Ridge Functions and Artificial Neural Networks

The theory of ridge functions can be effectively applied to artificial neural networks (ANNs) with specific layer architectures, as the neurons or groups of neurons in ANNs 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.

0.3.1 Artificial Neural Networks

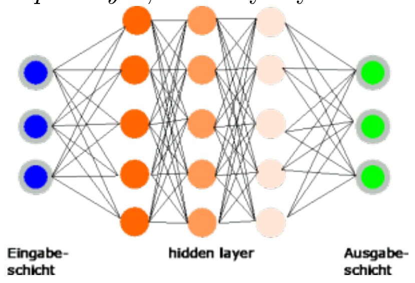
The basic building block of an ANN is the artificial neuron, essentially a mathematical abstraction of a biological neuron in the brain. This artificial neuron, as depicted in the following figure, takes as input a vector $x \in \mathbb{R}^n$. This input vector flows into a transfer function with corresponding weights $\{w_{i,j}\}_{i=1}^n$ for each x_i. Typically, this transfer function is the standard scalar product, so $net_j = \sum_{i=1}^n x_i w_{i,j}$.



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

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

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 (*layers*). The first layer is called the *input layer*, the last layer the *output layer*, and any layers in between are called *hidden layers*.



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 $o \in \mathbb{R}^n$ and a target value $t \in \mathbb{R}^n$, the MSE is defined as

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

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. 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.
These steps are repeated until a certain criterion is met, usually a desired level of accuracy in the mapping.

0.3.2 Ridge Functions in ANNs

As indicated at the beginning of this section, we now aim to apply some insights from the theory of ridge functions to ANNs. 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 $\mathcal{R}(a^1, ..., a^r; X)$ and formulate the following theorems that will assist us in the subsequent discussion.

Theorem 0.3.1. *Let $X \subset \mathbb{R}^d$ be a compact set, and define $\tau_i(X), i = 1, ..., r$ as in 0.1. 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, which we will need later.

Lemma 0.3.2. *If $\mathcal{R}(a^1, ..., a^r; X)$ is dense in $C(X)$, then X does not contain cycles with respect to the directions $a^1, ..., a^r$.*

Beweis. 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. $\square$

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 ANNs 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 0.3.3. *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.

Beweis. For (1): Let X be a compact subset of $\mathbb{R}^d$ with $\cap_{i=1,2,...}\tau_i(X) = \emptyset$. According to Theorem 0.3.1, $\mathcal{R}(a^1, ..., a^r)$ is dense in $C(X)$. Thus, for every $\epsilon > 0$, there exist continuous univariate functions g_i, $i = 1, ..., r$, such that

$$|f(x) - \sum_{i=1}^{r} g_i(a^i \cdot x)| < \frac{\epsilon}{r+1} \tag{0.4}$$

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

$$|g_i(y) - \sum_{j=1}^{m_i} c_{i,j}\sigma(y - \theta_{i,j})| < \frac{\epsilon}{r+1} \tag{0.5}$$

for all $y \in Y_i$, $i = 1, ..., r$. From 0.4 and 0.5, we obtain

$$||f(x) - \sum_{i=1}^{r}\sum_{j=1}^{m_i} c_{i,j}\sigma(a^i \cdot x - \theta_{i,j})||_{C(X)} < \epsilon \tag{0.6}$$

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 0.6 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 0.3.1, 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 0.3.1. *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 0.1.2 and0.1.4, 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 0.3.4. *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 $C(\mathbb{I}_n)$. In other words, for every $f \in C(\mathbb{I}_n)$ and every $\epsilon > 0$, there exists a sum $G(x) \in \mathcal{R}_r(\sigma)$ such that

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

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

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

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

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

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

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

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

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

Lemma 0.3.5. *Every bounded, measurable sigmoid function σ is discriminant. Specifically, every continuous sigmoid function is discriminant.*

Beweis. For $x, a \in \mathbb{I}_n$ 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,

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

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

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

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 $c(u)=\cos(m\cdot u)$,

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

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

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 $P_1,\dots,P_k$ and define a decision function f such that

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

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

Theorem 0.3.6. *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

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

Beweis. 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 $x\in D$,

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

$\square$

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 0.3.7. *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 0.3.8. *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 $D \subset \mathbb{I}_n$ such that $m(D) > 1 - \epsilon$ and

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

0.3.3 2-hidden-layer-KNN'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 KNNs with more hidden layers, which we now want to consider.

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 0.3.9. *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:*

$$f(x_1, ..., x_d) = \sum_{p=1}^{2d+1} g\Big(\sum_{q=1}^{d} \lambda_q \Phi_p(x_q) \Big) \tag{0.7}$$

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

Theorem 0.3.10. *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*

$$\Big| f(x) - \sum_{i=1}^{6d+3} d_i \sigma\Big(\sum_{j=1}^{3d} c_{ij} \sigma(w^{ij} \cdot x - \theta_{ij}) - \gamma_i \Big) \Big| < \epsilon \tag{0.8}$$

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 0.3.2 (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 0.3.11. *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

$$\Big| f(x) - \sum_{p=1}^{2d+2} d_p \sigma\Big(\sum_{q=1}^{d} c_{pq} \sigma(w^{pq} \cdot x - \theta_{pq}) - \gamma_p \Big) \Big| < \epsilon \tag{0.9}$$

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

Beweis. Let α be a positive number. We now partition the interval $[\alpha, +\infty)$ into segments $[\alpha, 2\alpha]$, $[2\alpha, 3\alpha]$, ... 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:

$$\sigma(t) = a_m + b_m u_m \left(\frac{t}{\alpha} - 2m + 1 \right), \quad t \in [(2m-1)\alpha, 2m\alpha], \tag{0.10}$$

or equivalently,

$$\sigma(\alpha t + (2m-1)\alpha) = a_m + b_m u_m(t), \quad t \in [0,1], \tag{0.11}$$

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

$$h(t) < \sigma(t) < 1, \quad t \in [(2m-1)\alpha, 2m\alpha], \tag{0.12}$$

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 0.12. 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 0.12, 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 0.11, it follows that for each $m = 1, 2, \ldots$, there exist numbers A_m, B_m, and r_m such that

$$u_m(t) = A_m \sigma(\alpha t - r_m) - B_m, \tag{0.13}$$

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 0.7. For the continuous, univariate function g used in this representation, for every $\epsilon > 0$, there exists a polynomial $u_m(t)$ such that

$$|g(t) - u_m(t)| < \frac{\epsilon}{2(2d+1)} \tag{0.14}$$

for all $t \in [0,1]$. Together with 0.13, we obtain

$$|g(t) - a\sigma(\alpha \cdot \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r) - b| < \frac{\epsilon}{2(2d+1)} \tag{0.15}$$

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

Substituting this result into Kolmogorov's superposition principle in 0.7, we obtain

$$\Big| f(x_1, \ldots, x_d) - \sum_{p=1}^{2d+1} \Big(a\sigma\Big(\alpha \cdot \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - r \Big) - b \Big) \Big| < \frac{\epsilon}{2} \tag{0.16}$$

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

$$|\Phi_p(x_q) - [a_p \sigma(\alpha x_q - r_p) - b_p]| < \delta, \tag{0.17}$$

for all $x_q \in [0,1]$. With $\lambda_q > 0$, $q = 1, \ldots, d$, $\sum_{q=1}^{d} \lambda_q = 1$, from 0.17 we obtain

$$\Big| \sum_{q=1}^{d} \lambda_q \Phi_p(x_q) - \Big[\sum_{q=1}^{d} \lambda_q a_p \sigma(\alpha x_q - r_p) - b_p \Big] \Big| < \delta \tag{0.18}$$

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 0.18, we obtain

$$\Big|\sum_{p=1}^{2d+1} a\sigma\Big(\alpha\sum_{q=1}^{d}\lambda_q\Phi_p(x_q) - r\Big) - \sum_{p=1}^{2d+1} a\sigma\Big(\alpha\Big[\sum_{q=1}^{d}\lambda_q a_p\sigma(\alpha x_q - r_p) - b_p\Big] - r\Big)\Big| < \frac{\epsilon}{2} \tag{0.19}$$

This can be rewritten as

$$\Big|\sum_{p=1}^{2d+1} a\sigma\Big(\alpha\sum_{q=1}^{d}\lambda_q\Phi_p(x_q) - r\Big) - \sum_{p=1}^{2d+1} d_p\sigma\Big(\sum_{q=1}^{d} c_{pq}\sigma(w^{pq}\cdot x - \theta_{pq}) - \gamma_p\Big)\Big| < \frac{\epsilon}{2} \tag{0.20}$$

From 0.16 and 0.20, it follows that

$$\Big|f(x) - \Big[\sum_{p=1}^{2d+1} d_p\sigma\Big(\sum_{q=1}^{d} c_{pq}\sigma(w^{pq}\cdot x - \theta_{pq}) - \gamma_p\Big) - s\Big]\Big| < \epsilon \tag{0.21}$$

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 0.9. □

0.4 Construction of a Special Activation Function

In the previous section, 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 (KNN) 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 only demonstrated 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.

0.4.1 Construction Algorithm

For the construction of the activation function, we will use insights from the previous section. 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

$$h(x) := 1 - \frac{\min\{1/2,\lambda\}}{1+\log(x-d+1)}. \tag{0.22}$$

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

$$h(x) < \sigma(x) < 1 \tag{0.23}$$

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

$$|\sigma(x) - h(x)| \leq \lambda, \tag{0.24}$$

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.

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:

$$[m_0; m_1, ..., m_l] = m_0 + \cfrac{1}{m_1 + \cfrac{1}{m_2 + \cfrac{1}{\ddots + \cfrac{1}{m_l}}}} \tag{0.25}$$

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:

$$\sigma(x) := a_n + b_n u_n(\frac{x}{d} - 2n + 1),\ x \in [(2n-1)d, 2nd] \tag{0.26}$$

where

$$a_1 := \frac{1}{2},\quad b_1 = \frac{h(3d)}{2} \tag{0.27}$$

and

$$a_n := \frac{(1+2M_n)B_2 - (2+M_n)B_1}{3(B_2 - B_1)},\quad b_n = \frac{1 - M_n}{3(B_2 - B_1)},\ n = 2, 3, ... \tag{0.28}$$

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

$$h(x) < M_n < \frac{1+2M_n}{3} \leq \sigma(x) \leq \frac{2+M_n}{3} < 1 \tag{0.29}$$

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

$$K_n := \frac{\sigma(2nd) + \sigma((2n+1)d)}{2},\ n = 1, 2, ... \tag{0.30}$$

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

$$\left| a_n + b_n u_n(\frac{x}{d} - 2n + 1) - (a_n + b_n u_n(1)) \right| \leq \epsilon,\ x \in [2d, 2d + d/2] \tag{0.31}$$

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

$$\sigma(x) := K_n - \beta_{2nd,2nd+\delta}(x) \cdot \Big(K_n - a_n - b_n u_n(\frac{x}{d} - 2n + 1)\Big),\ x \in [2nd, 2nd + d/2] \tag{0.32}$$

This function satisfies Condition 0.23:
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 $x \in [2nd, 2nd + \delta)$, $0 < \beta_{2nd,2nd+\delta}(x) \le 1$. From 0.32, 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 0.31, we also obtain

$$a_n + b_n u_n(1) - \epsilon \le A_n \le a_n + b_n u_n(1) + \epsilon \tag{0.33}$$

Together with 0.26 and 0.29, 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:

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

with

$$\delta := \min\left\{\frac{\bar{\epsilon} d}{b_{n+1}\overline{C}}, \frac{d}{2}\right\},\quad \bar{\epsilon} := \frac{1 - M_{n+1}}{6},\quad \overline{C} \ge \sup_{[-0.5,0]} |U_{n+1}'(x)|.$$

Thus, σ satisfies Condition 0.12 on $[2nd + d/2, 2nd + d]$ and

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

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

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

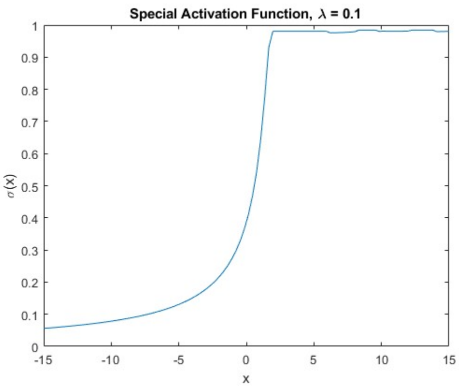
It can be easily seen that σ strictly increases and is smooth on $(-\infty, d)$.
Thus, the activation function is completely constructed.

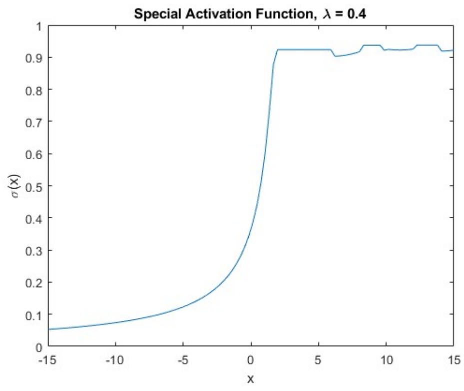
0.4.2 Properties of the Activation Function

The activation function constructed above possesses the following properties:

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 $[d, +\infty)$.

The special activation function looks like this:





Besides the approximation properties for a 2 hidden layer neural network, the special activation function is also used in another important theorem:

Theorem 0.4.1. *Let f be a continuous function over a compact interval $[a, b] \subset \mathbb{R}$, and let σ be the activation function constructed in* **??***. 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 $x \in [a, b]$.

Beweis. Let $d := b - a$. We divide the interval $[d, +\infty)$ into segments $[d, 2d], [2d, 3d], \ldots$. From 0.26, it follows that

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

for $n = 1, 2, \ldots$. Here, a_n and b_n are computed according to 0.27 and 0.28, respectively, for $n = 1$ and $n > 1$.

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

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

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 **??**, 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,

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

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 **??**. 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. □

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 continous functions on $[0, 1]^d$. We will restrict ourselves here to the cases $d = 1$ and $d = 2$. Unfortunately we won't win any new realizations for real world applications. This is due to the complexity of the activation function, which takes much longer to calculate than the usual activation functions, making it unusable for real applications. 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

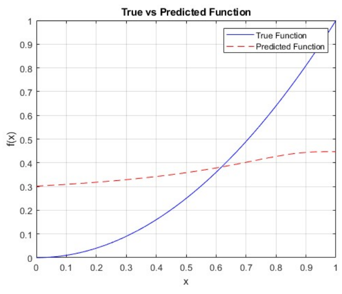
neural networks. In our current example, the activation function is integrated in a custom layer, in front of which so-called fully connected layers are built, which contain the weights and biases. Since we are dealing here with the approximation of functions over $[0,1]$ and $[0,1]^2$, we need one or two input neurons and an output neuron. The combination of the layers in Matlab looks as follows:

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

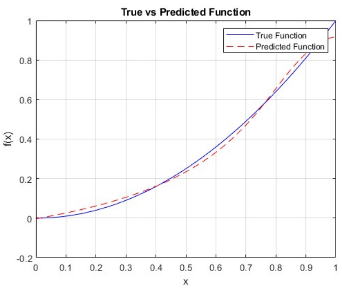
In our first example, we want to approximate the function $f:[0,1]\to\mathbb{R}$, $x\mapsto f(x)=x^2$. To do this, we use the layer setup above and the following options for the neural network:

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

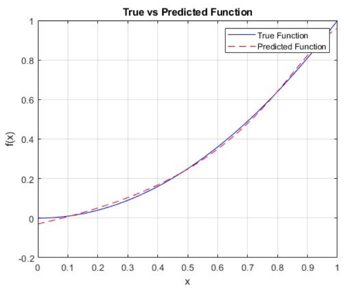
With this setup we obtain the following approximation and learning curve: We see that we can get any



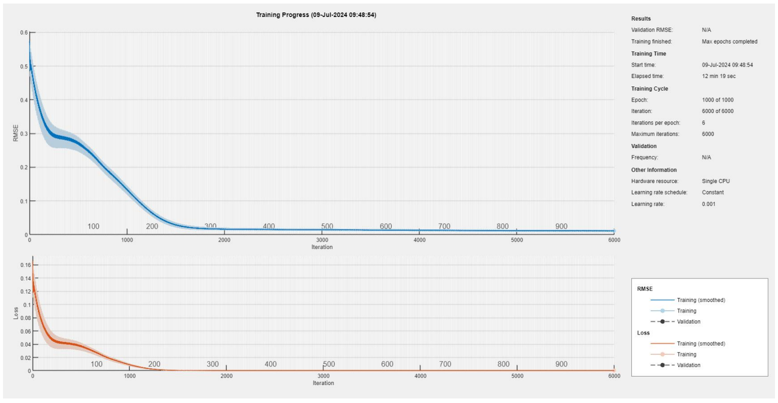
(a) 200 Epochs



(b) 600 Epochs



(c) 1000 Epochs



```
Results
Validation RMSE:            N/A
Training finished:          Max epochs completed
Training Time
Start time:                 09-Jul-2024 09:48:54
Elapsed time:               12 min 19 sec
Training Cycle
Epoch:                      1000 of 1000
Iteration:                  6000 of 6000
Iterations per epoch:       6
Maximum iterations:         6000
Validation
Frequency:                  N/A
Other Information
Hardware resource:          Single CPU
Learning rate schedule:     Constant
Learning rate:              0.001
```

(d) Training progress within 1000 Epochs

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

desired approximation also in real neural networks. The used error function is the Root Mean Squared Error (RMSE). It is defined as:

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

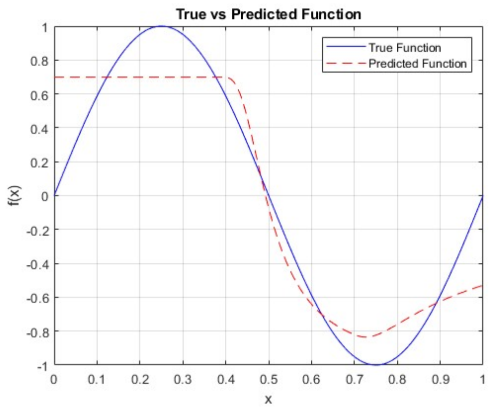
where:

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

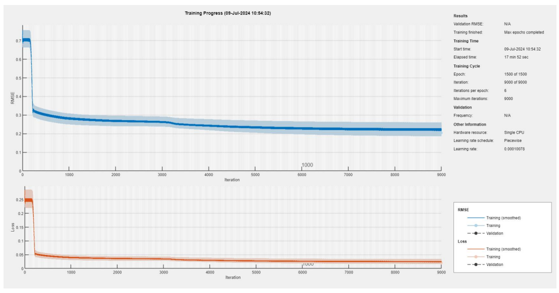
The advantage we see is that we can approximate the function arbitrarily well with a relatively small network (in this case only 5 hidden neurons). The disadvantage is that learning takes quite a long time in contrast to conventional architectures. As I said, it makes the previous approach unusable in real applications, but shows the correctness of the above theorems. Now we want to approximate a somewhat more complicated function, namely $f(x) = sin(2\pi x)$. For this we have the follwing 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

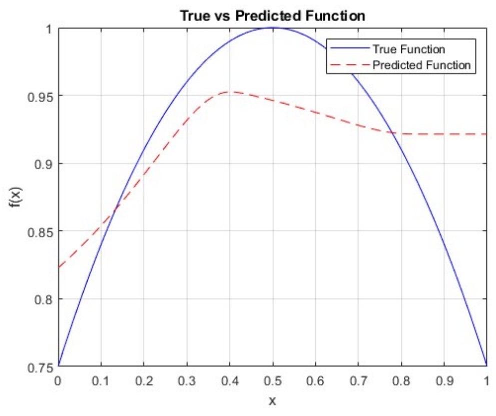

Training Progress (09-Jul-2024 10:54:32)

Results
Validation RMSE: N/A
Training finished: Max epochs completed
Training Time
Start time: 09-Jul-2024 10:54:32
Elapsed time: 17 min 52 sec
Training Cycle
Epoch: 1500 of 1500
Iteration: 9000 of 9000
Iterations per epoch: 6
Maximum iterations: 9000
Validation
Frequency: N/A
Other Information
Hardware resource: Single CPU
Learning rate schedule: Piecewise
Learning rate: 0.00010078

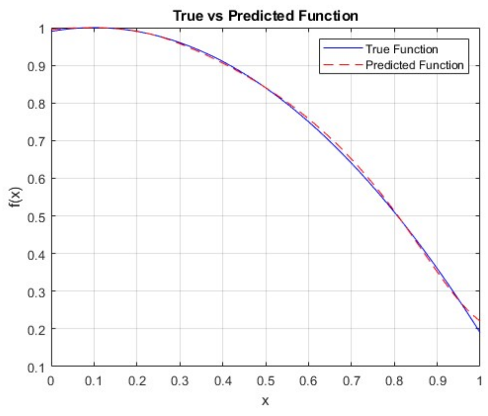
(b) Learning Progress

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

We see, that we also get some kind of good approximation, but it's 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 $[0, 1]$:
With the interval-symmetric variant, the network adjusts to a certain configuration after a certain



(a) Bad Function Approximation



(b) Successful Function Approximation

Abbildung 3: Function approximation depending on translation

time, from which it no longer deviates. Mathematically speaking, a local minimum is reached there for the loss function, which is no longer left. We can see the result at 3a. 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 3b.
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:

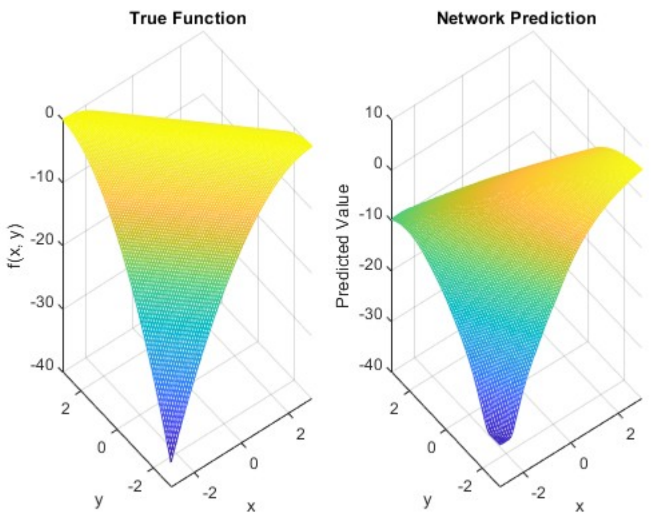
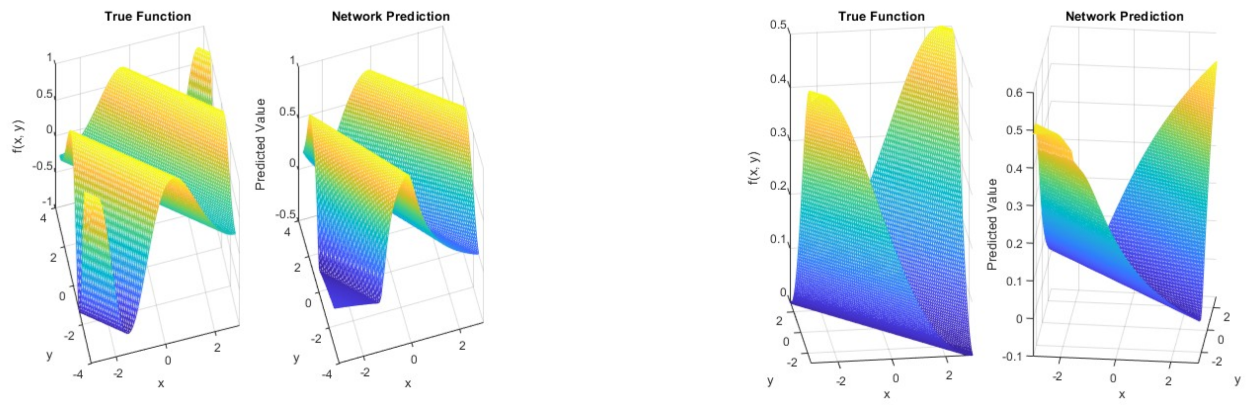


Abbildung 4: Approximation of $f(x, y) = (x + y)^2$

The result is not as accurate as we might wish, 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:
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 ... ... 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. However, this method is useless for real applications.



(a) Approx. $f(x,y) = \sin(0.25 * (x+y)^2)$ (b) Approx. $f(x,y) = 0.5 * \sin(0.05 * (x+y)^2)$

Abbildung 5: Function approximation of sine functions over $\mathbb{R}^2$

0.4.3 Ridgelet Applications

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

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

Then we call the functions $\psi_\tau = a^{-1/2}\psi\left(\frac{u \cdot x - b}{a}\right)$, derived from ψ, *ridgelets*. 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 $\mu(d\tau) = \sigma_d \frac{da}{a^{d+1}} du db$. Thus, any multi-dimensional function $f \in L^1 \cap L^2(\mathbb{R}^d)$ can be described as a superposition of ridge functions:

$$f = c_\psi \int \langle f, \psi_\tau \rangle \psi_\tau \mu(d\tau) \tag{0.34}$$

$$= \pi(2\pi)^{-d} K_\psi^{-1} \int \langle f, \psi_\tau \rangle \psi_\tau \sigma_d \frac{da}{a^{d+1}} du db \tag{0.35}$$

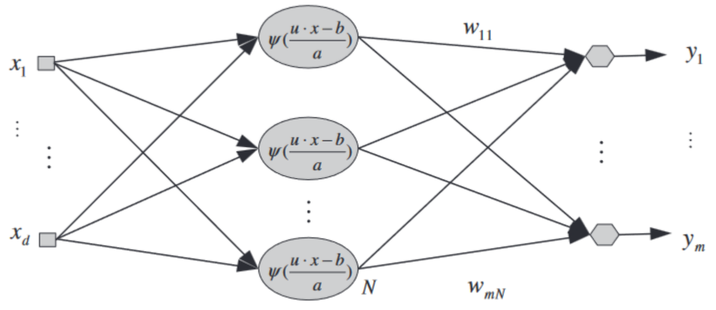
In neural networks, the central goal is to approximate a function $Y = f(x) : \mathbb{R}^d \to \mathbb{R}^m$. This is achieved by approximating m mappings $y_i = f_i(x) : \mathbb{R}^d \to \mathbb{R}^m$. With the corresponding ridgelet approximation, we obtain

$$\tilde{y}_i = \sum_{j=1}^{N} c_{ij} \psi_\tau \left(\frac{u_j \cdot x - b_j}{a_j} \right) \tag{0.36}$$

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.

Application to a Single-Hidden-Layer Network

So far, we have chosen sigmoid functions as activation functions for neurons in artificial neural networks. This is still mostly the case in practice. Now, however, we want to deviate from this approach and use ridgelets as activation functions based on their approximation properties. The basic model looks as follows:


x_1
x_d
$\psi(\frac{u\cdot x-b}{a})$
$\psi(\frac{u\cdot x-b}{a})$
$\psi(\frac{u\cdot x-b}{a})$
N
w_{11}
w_{mN}
y_1
y_m

For the model, we take a set of test data $\boldsymbol{S} = \{(x_1, y_1), ..., (x_P, y_P)\}$ (where $x_i = (x_{1i}, ..., x_{di})$ and $y_i = (y_{1i}, ..., y_{mi})$, $i = 1, ..., P$). Here, x_i is the input and y_i is the corresponding output. Furthermore, w_{ij} denotes the weight connecting the i-th output neuron to the j-th hidden layer neuron. The weights between the input and hidden layer are all constant 1, making the weight matrix between these layers the identity matrix. The weight matrix between the last two layers is given by

$$W = \begin{pmatrix} w_{11} & \dots & w_{1N} \\ \vdots & \ddots & \vdots \\ w_{m1} & \dots & w_{mN} \end{pmatrix}.$$

Due to the ridgelet activation functions $\psi((u_j \cdot x_i - b_j)/a_j)$ in the neurons, we not only have weights w_{ij} that need to be trained, but also direction vectors u_j, as well as scaling and translation parameters a_j and b_j. These are summarized in matrix form as follows:

$$U_{N\times d} = \begin{pmatrix} u_{11} & \dots & u_{1d} \\ \vdots & \ddots & \vdots \\ u_{N1} & \dots & u_{Nd} \end{pmatrix}, \quad A_{N\times P} = \begin{pmatrix} a_1 & a_1 & \dots & a_1 \\ \vdots & \vdots & \dots & \vdots \\ a_N & a_N & \dots & a_N \end{pmatrix}, \quad B_{N\times P} = \begin{pmatrix} b_1 & b_1 & \dots & b_1 \\ \vdots & \vdots & \dots & \vdots \\ b_N & b_N & \dots & b_N \end{pmatrix}.$$

The hidden layer output is thus

$$\Psi = \psi((UX - B)/A)$$

or equivalently,

$$\Psi = (\psi_1, \dots, \psi_P),$$

where

$$\psi_k = (\psi(x_k, u_1, a_1, b_1), \psi(x_k, u_2, a_2, b_2), \dots, \psi(x_k, u_P, a_P, b_P))^T.$$

Therefore, the final output of the network is

$$\tilde{Y} = W\Psi.$$

We define the performance of the network as

$$\text{Perf} = \frac{1}{2P}\sum_{i=1}^{P}(\tilde{y}_i - y_i)^T(\tilde{y}_i - y_i) = \frac{1}{2P}\text{tr}((\tilde{Y}_i - Y_i)^T(\tilde{Y}_i - Y_i)).$$

As in all other networks, the goal now is to minimize this cost function. For linear weights w_{ij}, most classical neural networks use a gradient-based method such as stochastic gradient descent (SGD).

This works well for linear weights, but in our case, we also need to optimize the scaling, rotation, and position parameters a_j, u_j, and b_j of the ridgelets $\psi((u_j \cdot x - b_j)/a_j)$, which unfortunately are not linear. To optimize all parameters in the network, we separate the weights from the ridgelet parameters and handle the latter with particle swarm optimization (PSO). This method is inspired by the natural behavior of bird or fish swarms, where a swarm of candidate solutions explores the space to minimize the objective function. Here is how the algorithm works:

First, for each candidate i ($i = 1, ..., N$), choose an initial position $\overrightarrow{p}_{i,0}$ randomly distributed over the area of interest. Each candidate also receives a random initial velocity vector $\overrightarrow{v}_{i,0}$. We also define the following predefined parameters:

- Inertia of movement ω
- Memory of the best coordinates achieved by each candidate, $\overrightarrow{p}_{i,\text{best}}$
- Knowledge of the coordinates of the global best value $\overrightarrow{g}_{\text{best}}$
- Cognitive weight factor c_k
- Social weight factor c_s

In each step, a new velocity vector is calculated from the above values, weighted by random values r_i, and from this, a new position is computed for each candidate as follows:

$$\overrightarrow{v}_{i,k+1} = \omega \cdot \overrightarrow{v}_{i,k} + c_k \cdot r_1(\overrightarrow{p}_{i,\text{best}} - \overrightarrow{p}_{i,k}) + c_s \cdot r_2(\overrightarrow{g}_{\text{best}} - \overrightarrow{p}_{i,k})$$

The new positions of the candidates are then

$$\overrightarrow{p}_{i,k+1} = \overrightarrow{p}_{i,k} + \overrightarrow{v}_{i,k+1}.$$

For the following applications, we will go a step further and use an optimized form of PSO. This includes the density ρ of swarm points around a found best value. The goal of the method is to prevent the algorithm from terminating when the swarm points gather in a local minimum, aiming to find a global minimum. First, let's define the density of a swarm.

Definition 0.4.1. *Let $P_k = \{\overrightarrow{p}_{1,k}, \overrightarrow{p}_{2,k}, ..., \overrightarrow{p}_{N,k}\}$ be a swarm of candidates in the k-th iteration step of PSO, and let $f : \mathbb{R}^d \to \mathbb{R}$ be the function to be minimized. We define the average fitness value of candidate points as $fit^k_{avg} := (f(\overrightarrow{p}_{1,k}) + \ldots + f(\overrightarrow{p}_{N,k}))/N$, and the best value of the points as $fit^k_{gbest} := f(\overrightarrow{g}_{best})$. Then, the density ρ_k of the swarm P_k is defined as*

$$\rho_k := e^{(fit^k_{avg} - fit^k_{gbest})^2/\sigma^2},$$

where σ is a measure of the spread of swarm points.

The density of points becomes relevant when the points are too concentrated in a local minimum, surpassing a specified ρ_{min}. In such cases, the points should move away from each other. This scenario is built into the inertia constant ω:

$$\omega = \omega_{max} - iter \cdot (\omega_{max} - \omega_{min})/iter_{max} + (\rho_{min} \leq \rho) \cdot \rho \cdot \omega_{dis} \tag{0.37}$$

Here, ω_{max} and ω_{min} are the prescribed maximum and minimum values of ω, ω_{dis} is the disturbance constant, and $iter$ and $iter_{max}$ are the current iteration and the maximum number of iterations, respectively.

As mentioned earlier, we will optimize the nonlinear ridgelet parameters a, u, b using the so-called ρ-PSO. The linear weights w will be optimized using the standard Newton's method as follows:

$$w^{k+1}_{ij} = w^k_{ij} - \alpha \cdot \nabla Perf \tag{0.38}$$

where

$$\nabla Perf = \frac{\partial Perf}{\partial w_{ij}^k} = \frac{1}{P}\left(\sum_{i=1}^{P}(\tilde{y}_i - y_i)\right)\psi\left(\frac{u_{j0}x_{0i} - b_j}{a_j}\right). \tag{0.39}$$

For the optimization of parameters, the position p_k of the particle is defined by concatenating individual vectors:

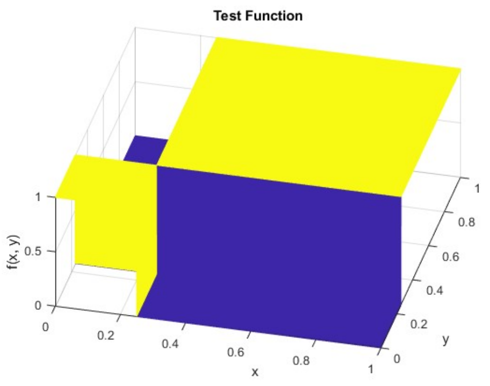
$$p_k = (a_1, \ldots, a_N, u_{11}, \ldots, u_{Nd}, b_1, \ldots, b_N).$$

The algorithm proceeds as follows: **Step 1:** Initialize all position and velocity vectors (randomly), as well as weights, swarm size, learning rate, etc.
Step 2: Compute fitness of all candidates and their average. Select the best individual and store its position. If termination criterion is met, stop training.
Step 3: Update linear weights using formulas 0.38 and 0.39.
Step 4: Calculate swarm density and inertia using formula 0.37.
Step 5: Update position and velocity vectors, return to Step 2.

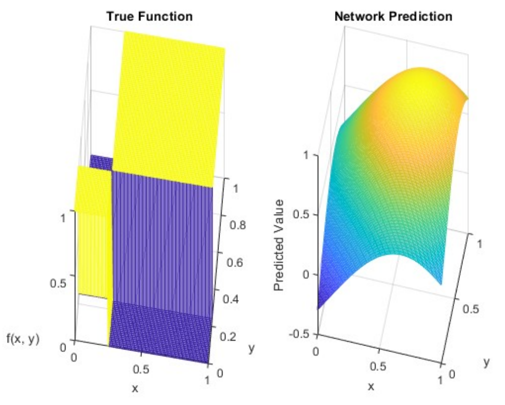
0.4.4 Application example for Ridgelet Neural Networks

In this section, we will look at what ridgelet neural networks can be used for. According to the work of Dr. Jiao Licheng, 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

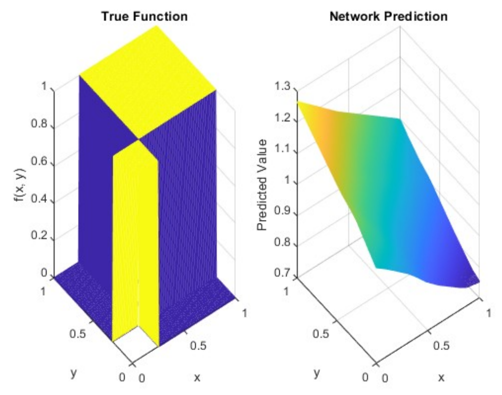
$$f(x, y) = \begin{cases} 1 & (x - 1/4)(y - 1/4) > 0 \\ 0 & \text{sonst} \end{cases}$$



First we will try to approximate the function by a pure ridgelet network. For this we use 30 neurons in the hidden layer. The result looks like this:
You can guess a certain similarity in 6a, 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 result we get when we combine the ridgelet and relu networks. For this, we just put a fully connected relu layer with 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 the following graphics:
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 won't get any better. So now we fix the number of neurons at 30 and look, how the accuracy develops over the epochs:

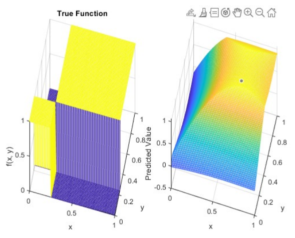


(a) Approximation with ridgelet layer

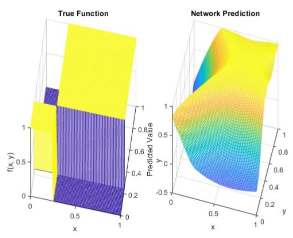


(b) Approximation with relu layer

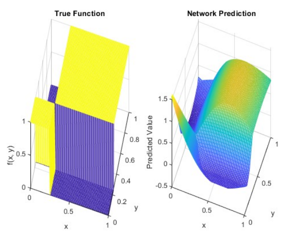
Abbildung 6: Approximation of a discontinous function over $[0, 1]^2$



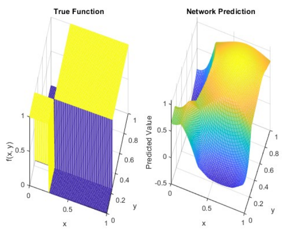
(a) 5 neurons per layer



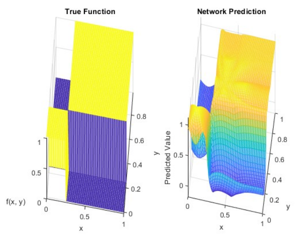
(b) 10 neurons per layer



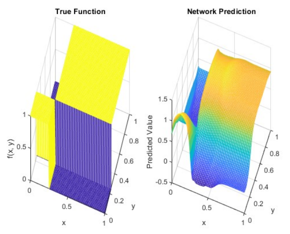
(c) 15 neurons per layer



(d) 20 neurons per layer



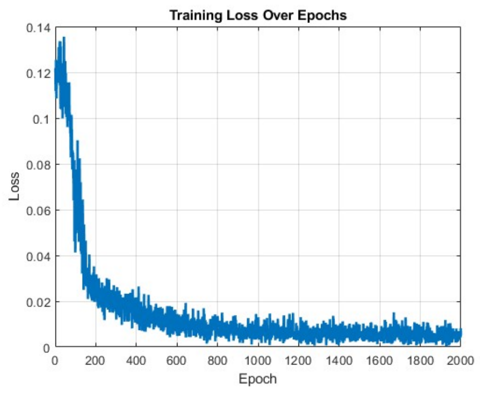
(e) 30 neurons per layer



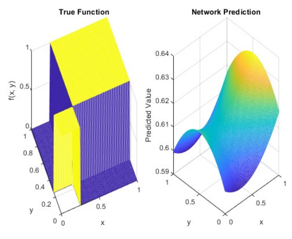
(f) 40 neurons per layer

Abbildung 7: Function Approximation with different neuron number after 200 epochs

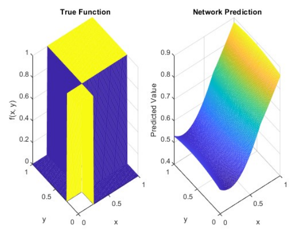
Here, as expected, we see an increase in accuracy across the epochs. However, we need at least about 200 epochs to achieve satisfactory accuracy. The following graph shows the development of the loss function over the epochs:



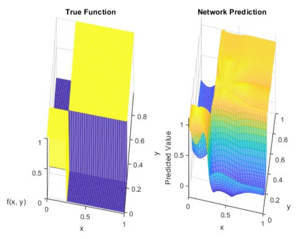
In this graph we can even see that it takes about 1000 epochs for the loss function to stabilize. As we can see, ridgelet networks are very well suited for approximating such functions. This is particularly evident in comparison with other conventional networks of a comparable size. The following graphic 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:



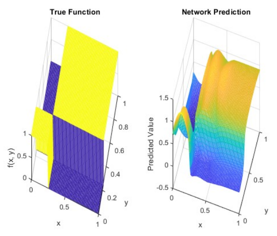
(a) 10 epochs



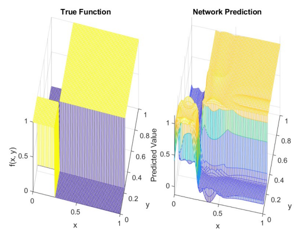
(b) 20 epochs



(c) 200 epochs

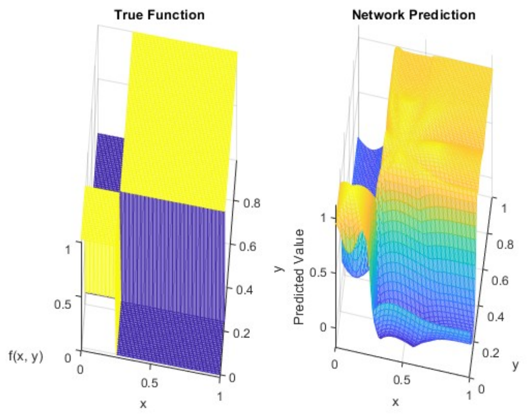


(d) 1000 epochs

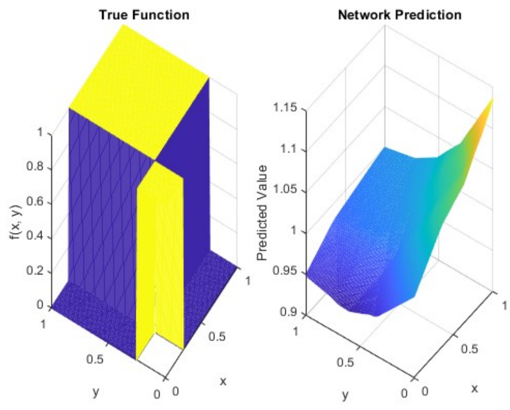


(e) 2000 epochs

Abbildung 8: Function Approximation with 30 Neurons



(a) Ridgelet Network



(b) Relu Network

Abbildung 9: 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. But where are they used in practice? One area of application of the ridgelet neural network is the analysis and classification of textures, as they contain many discontinuities that can be easily processed by a ridgelet layer. For the following test we use the KTH-TIPS data set, which contains images of surfaces of 10 different materials. We will now test how an image classification network with and without a ridgelet layer handles the classification of the images. The network looks as follows:

```
layers = [
    imageInputLayer(inputSize, 'Name', 'input')

    % Custom Ridgelet Layer

    %RidgeletLayer('ridgelet2', randn([1 1 inputSize(3)]), 0, 1)
    % Simpler architecture
    convolution2dLayer(3, 8, 'Padding', 'same', 'Name', 'conv1')
    batchNormalizationLayer('Name', 'bn1')
    reluLayer('Name', 'relu1')
    maxPooling2dLayer(2, 'Stride', 2, 'Name', 'maxpool1')
    %RidgeletLayer('ridgelet1', randn([1 1 inputSize(3)]), 0, 1)
    convolution2dLayer(3, 16, 'Padding', 'same', 'Name', 'conv2')
    batchNormalizationLayer('Name', 'bn2')
    reluLayer('Name', 'relu2')
    maxPooling2dLayer(2, 'Stride', 2, 'Name', 'maxpool2')
    %RidgeletLayer('ridgelet2', randn([1 1 inputSize(3)]), 0, 1)
    fullyConnectedLayer(32, 'Name', 'fc1')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(numel(unique(imds.Labels)), 'Name', 'fc2')
    softmaxLayer('Name', 'softmax')
    classificationLayer('Name', 'output')
];
```

We see a compilation of different convolution, pooling and batchnormalization layers. These are preceded by the ridgelet layer, which we now switch on and off in order to compare the accuracy results of the network.

0.5 Quaternion Neural Networks

So far, artificial neural networks, despite their comprehensive treatment using ridge functions and ridgelets, have only dealt with real numbers. However, some considerations for neural network applications extend beyond this approach and involve the use of quaternions in these networks.

0.5.1 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 commutative field due to the lack of commutativity. 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 $q_1 = a_1 + b_1i + c_1j + d_1k$ and $q_2 = a_2 + b_2i + c_2j + d_2k$, their product is

$$\begin{aligned} q_1q_2 &= (a_1 + b_1i + c_1j + d_1k)(a_2 + b_2i + c_2j + d_2k) \\ &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ &= +(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i \\ &= +(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j \\ &= +(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

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:

$$y^* = w^* x^* \overline{w^*} \tag{0.40}$$

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 $w^* = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(w_1 i + w_2 j + w_3 k)$.

The Quaternion Convolutional Layer

Quaternions are a frequently considered research object in many parts of analysis, for example Fourier analysis or wavelets. Much research has also been done on quaternions in the field of neural networks. 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 0.40. So we define our 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

$$f^*_{kl} = \sum_{i=1}^{L}\sum_{j=1}^{L} \frac{1}{s_{ij}} w^*_{ij} a^*_{i+l,k+j} \overline{w^*_{ij}}$$

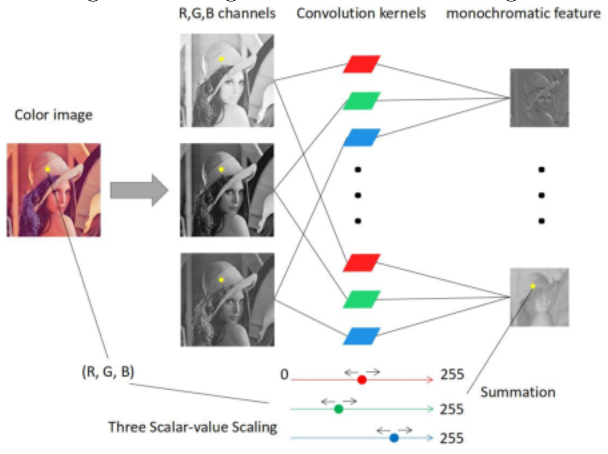
We can also represent this quaternion operation as a matrix operation. This looks as follows:

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

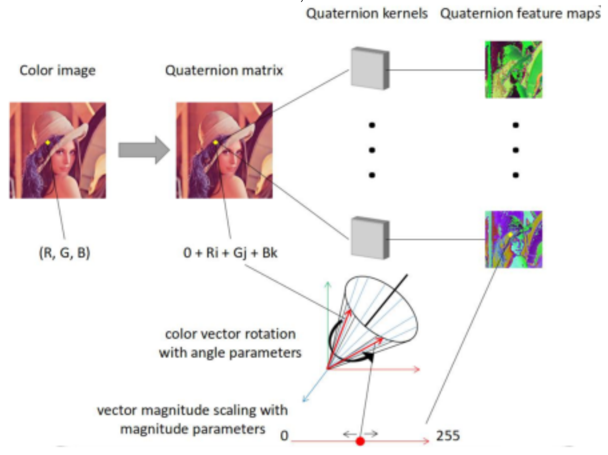
Here, f_{kl} is the vectorized representation of f^*_{kl} and

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

Picture 0.5.1 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.

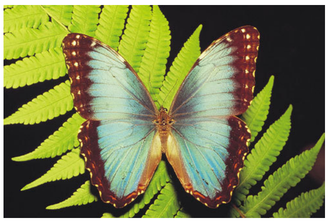


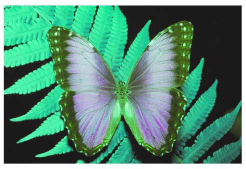
(a) Real-valued CNN

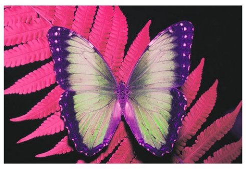


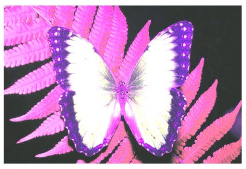
(b) Quaternion CNN

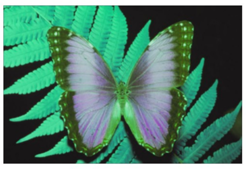
We will now use an image of a butterfly to see how quaternion folding affects an RGB image:


(a) Original Butterfly Image


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


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


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


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

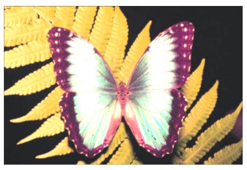

(f) Kernel size=3 × 3, angle∈ $[-\pi/2, \pi/2]$, scale∈ $[0, 0.1]$

Abbildung 10: Convoluted Image with different Kernels

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 10b and 10c. 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

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

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

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

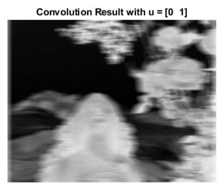
Benefits of a QCNN

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 backpropagation 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.

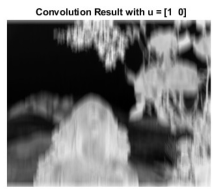
0.5.2 Ridgelet Filter

In chapter ... we saw that ridgelets can be used wonderfully as activation functions for neural networks to classify images. However, the use of ridgelets does not end there. As with QCNNs, we can also use ridgelets to define filters that we can apply in a convolutional layer. 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 has a size of 494×349 and we apply

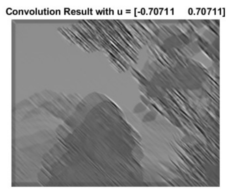
(a) Original Image



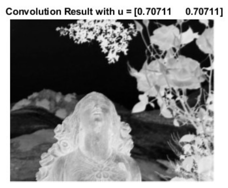
(b) $u = [0, 1]$



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



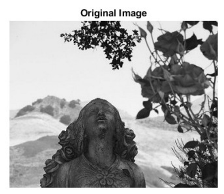
(d) $u = [-1/\sqrt{2}, 1/\sqrt{2}]$



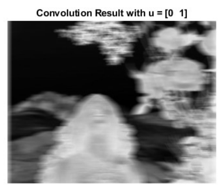
(e) $u = [1/\sqrt{2}, 1/\sqrt{2}]$

Abbildung 11: Ridgelet Convoluted Image with different Ridgelet Directions

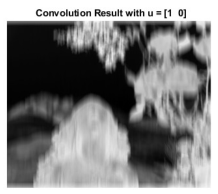
kernels with the size 16×16:
Now we want to take a look at different convolutions with different scales and positions:



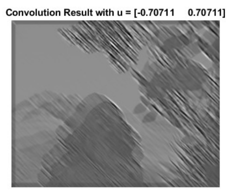
(a) Original Image



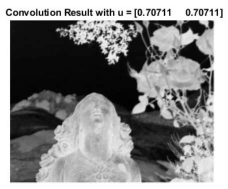
(b) $u = [0, 1]$



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



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



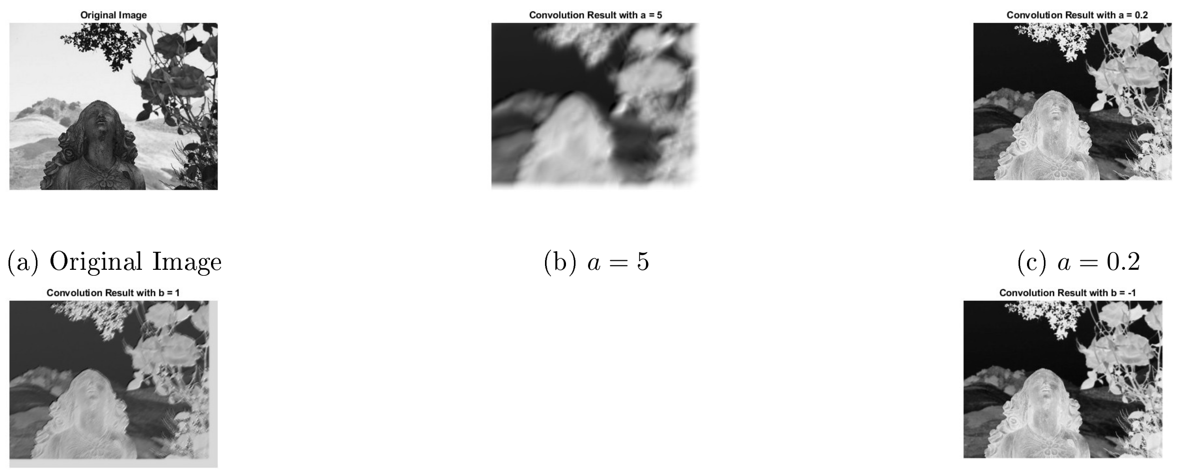
(e) $u = [1/\sqrt{2}, 1/\sqrt{2}]$

Abbildung 12: 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.
Now let's take a look at how the scaling and position parameters affect the convolution:
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.

Application of the Ridgelet Convolution in CNN's

Now that we've looked at the effect of ridgelet filters, let's 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


Original Image
Convolution Result with a = 5
Convolution Result with a = 0.2
Convolution Result with b = 1
Convolution Result with b = -1

(a) Original Image (b) $a = 5$ (c) $a = 0.2$

(d) $b = 1$ (e) $b = -1$

Abbildung 13: Ridgelet Convoluted Image with different Ridgelet scales and positions

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:

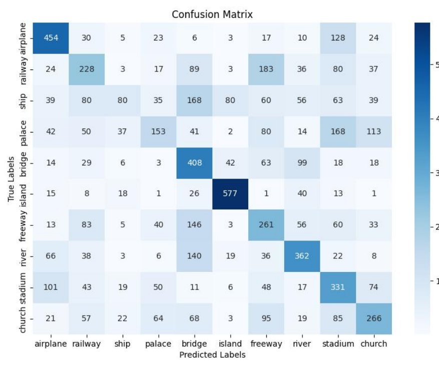
```
    imageInputLayer([imsize imsize 1])

%RidgeletConvLayer([28 56],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: We can see that with a small

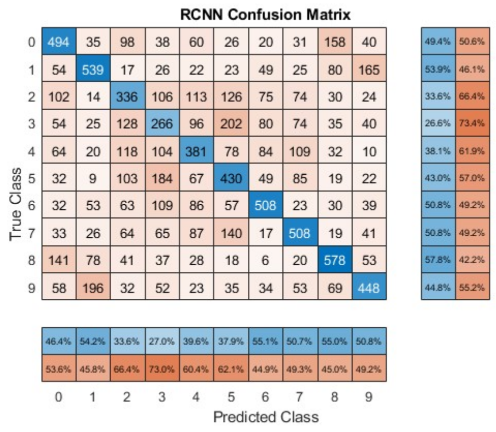
Kernel Size		CNN	RCNN
3	Time	2:24 min	3:25 min
	Accuracy	0.406	0.367
14	Time	3:25 min	4:34 min
	Accuracy	0.419	0.406
28	Time	6:26 min	4:55 min
	Accuracy	0.393	0.422

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 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 dataset. Now we want to see how well ridgelet layers perform compared to conventional convolutional layers in terms of accuracy. For this purpose, we use the well-known CIFAR10 dataset, which is gray-scaled for testing purposes. This should be classified in the same network as above, but with a kernel size of 14x14. To compare the performances, let's take a look at the confusion matrices of the two neural networks. These show how often an image was assigned to a certain class on the basis of the trained network. The higher the values on the main diagonal, the better the network. In this case, both networks were trained for 50 epochs, which is certainly not enough for perfect training, but for a comparison.



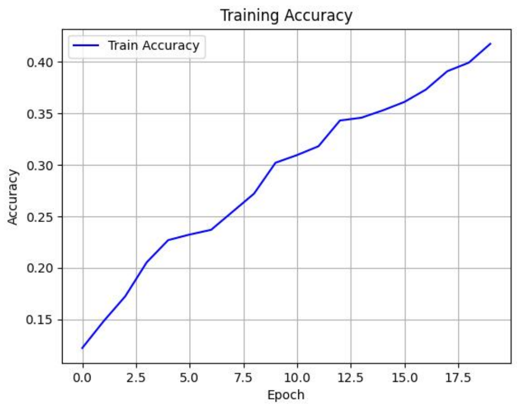
(a) Confusion Matrix of CNN



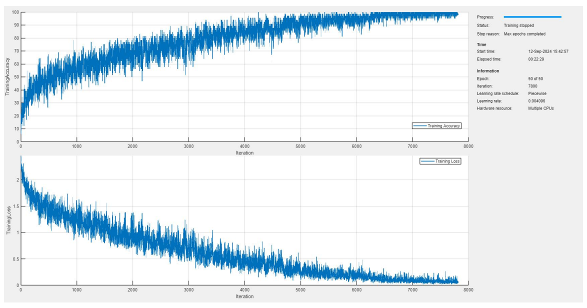
(b) Confusion Matrix of RCNN

Abbildung 14: Confusion Matrices in Comparison

Number	1	2	3	4	5	6	7	8	9	10
Class	Airplane	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck



(a) Learning Progress of CNN



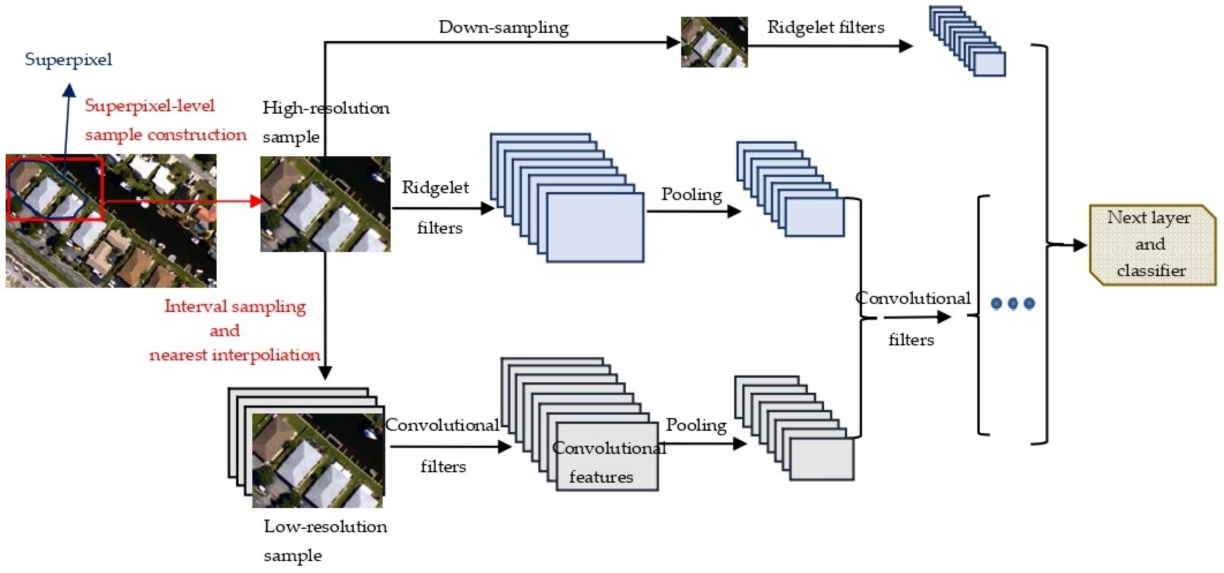
(b) Learning Progress of RCNN

Abbildung 15: Learning Progress in Comparison

In terms of accuracy, we can see that there are no major differences. Some classes are registered better, but on average the accuracy is around 0.43. This is not much, but it is something that can certainly be improved with more epochs. More interesting for this example is the time that the network needs for 50 epochs. The results in the table above already suggest this result, but it is particularly clear here that the RCNN takes a good deal longer with this kernel size. In the end, this underlines that both networks work well, but are intended for different types of tasks: CNN's for low resolution images and RCNN's for high resolution images.

0.6 The splitted CNN

In the last section we want to fusionate the results of the last chapters. To be precise, we want to use both ridgelet and quaternion filtering in a network. This idea is based on the work of Zhifeng Zheng and Jiannong Cao, who used standard and ridgelet convolutional layers in a network. The idea is to split the input image. On the one hand, the high-resolution image remains. This is then processed in a ridgelet layer as shown in the last chapter. On the other hand, we create a version of the image with a lower resolution. This is then passed on to a standard convolutional layer. The two resulting feature maps are then merged again and processed further. All of this is illustrated once again in the following figure.



In contrast to the last chapter, 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:

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

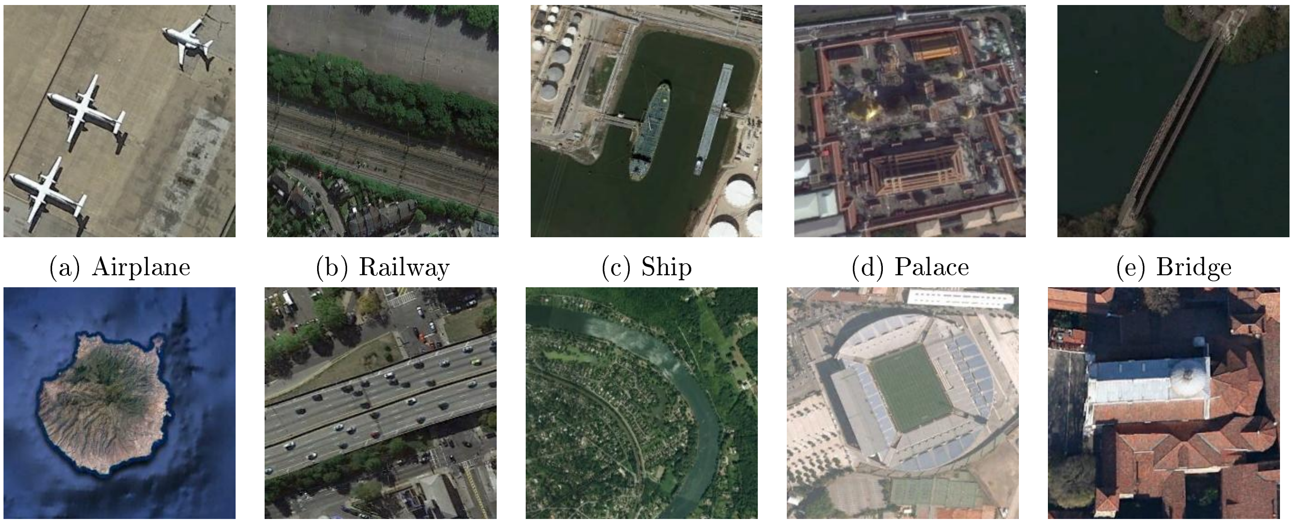
Because the data set is split, processed and fusionated again in the process, we will now use Python instead of Matlab for the implementation of the network. Especially we will use PyTorch, because the realization of the splitting is really simple there. So, mathematically expressed, the splitted network 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 seperated outputs are concatenated. The fusionated uput 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 spliiting 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*. We can see examples of the given images here:

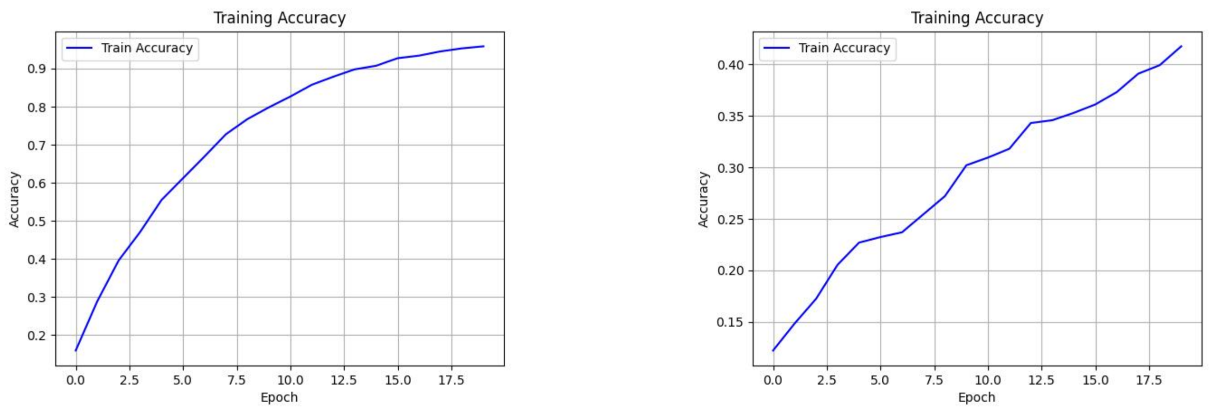


(a) Airplane (b) Railway (c) Ship (d) Palace (e) Bridge

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

Abbildung 16: Example Images of the 10 used Classes

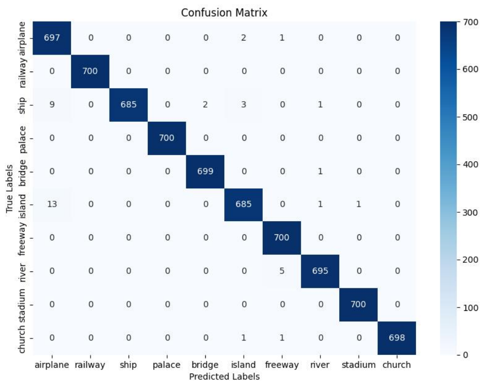
We can see that the difficulty with the selected classes lies in the similarity of some of them. Especially railway, bridge, freeway and river sometimes look quite the same, as they are long constructs that run through a landscape. The palace and church also look quite similar. However, we will see that the split network copes well with such things. 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:



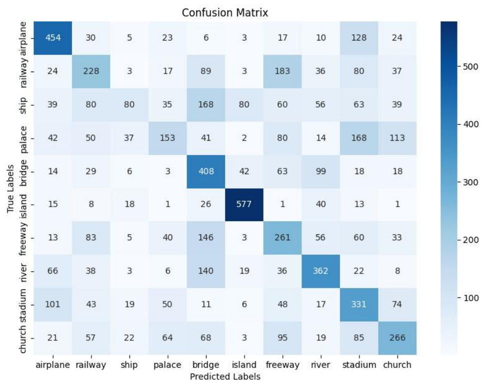
(a) Learning Progress of SCNN (b) Learning Progress of CNN

Abbildung 17: 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

Abbildung 18: 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.

0.6.1 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. Although this is more computationally expensive than standard convolution, it shows better results in terms of accuracy and stability. Furthermore, the computational effort should not be so significant, as the images to be processed do not have a high resolution, as mentioned at the beginning. For test purposes, the convolutional layer is simply replaced by a quaternion convolutional layer.