

Institute of Parallel and Distributed Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Master Thesis

# Bayesian Functional Optimization of Neural Network Activation Functions

Heiko Zimmermann

**Course of Study:** Informatik

**Examiner:** Prof. Dr. rer. nat. Mark Toussaint

**Supervisor:** Ph.D. Vien Ngo

**Commenced:** November 4, 2016

**Completed:** May 4, 2017

**CR-Classification:** G.1.6, I.2.6



## Abstract

In the past we have seen many great successes of Bayesian optimization as a black-box and hyperparameter optimization method in many applications of machine learning. Most existing approaches aim to optimize an unknown objective function by treating it as a random function and place a parametric prior over it. Recently an alternative approach was introduced which allows Bayesian optimization to work in nonparametric settings to optimize functionals (Bayesian functional optimization).

Another well recognized framework that powers some of today's most competitive machine learning algorithms are artificial neural networks which are state of the art tools to parameterize and train complex nonlinear models. However, while normally a lot of attention is paid to the network's layout and structure the neuron's nonlinear activation function is often still chosen from the set of commonly used function. While recent work addressing this problem mainly considers steepest-descent-based methods to jointly train individual neuron activation functions and the network parameters, we use Bayesian functional optimization to search for globally optimal shared activation functions. Therefore, we formulate the problem as a functional optimization problem and model the activation functions as elements in a reproducing kernel Hilbert space.

Our experiments have shown that Bayesian functional optimization outperforms a similar parametric approach using standard Bayesian optimization and works well for higher dimensional problems. Compared to the baseline models with fixed sigmoid and jointly trained shared activation function we achieved an improvement of the relative classification error over 39% and over 20%, respectively.

## Kurzfassung

In der Vergangenheit konnte *Bayesian Optimization* viele Erfolge als Black-Box und Hyperparameter-Optimierungsverfahren in vielen Anwendungen des maschinellen Lernens erzielen. Die meisten bestehenden Ansätze zielen auf die Optimierung einer unbekannteren unbekannteren Zielfunktion ab, indem sie diese als zufällige Funktion behandeln und einen parametrischen Ansatz wählen. Kürzlich wurde ein alternativer Ansatz vorgestellt, der es ermöglicht *Bayesian Optimization* in nicht parametrischen Szenarien zu verwenden (*Bayesian functional optimization*).

Eine weiteres viel beachtetes Framework, dass in viele wettbewerbsfähigen Algorithmen des maschinellen Lernens verwendet wird, sind künstliche neuronale Netze, die zu den state-of-the-art Werkzeugen zum Parametrisieren und Trainieren komplexer nicht linearer Modelle gehören. Während dem Layout und der Struktur des Netzwerks viel Aufmerksamkeit zukommt, wird die nichtlineare Aktivierung der Neuronen oftmals aus einer Menge von oft benutzten Aktivierungsfunktionen gewählt. Während bisherige Arbeiten vor allem Trainingsverfahren untersuchen, die individuelle Aktivierungsfunktionen zusammen mit den Netzwerkparametern optimieren, verwenden wir *Bayesian Function Optimization*, um gemeinsame global optimale Aktivierungsfunktionen zu suchen.

Wir formulieren das Problem als Optimierungsproblem für Funktionale und modellieren die Aktivierungsfunktion als Element in einem Hilbertraum mit reproduzierendem Kern. Unsere Experimente haben gezeigt, dass *Bayesian Function Optimization* einen ähnlichen parametrischen Ansatz mit Standard *Bayesian Optimization* schlägt und gut für höhere dimensionale Probleme funktioniert. Verglichen mit den zugrundegelegten Modellen mit fester Sigmoid-Aktivierungsfunktion und gemeinsam trainierten Aktivierungsfunktionen erzielen wir eine Verbesserung des relativen Klassifikationsfehlers von 39% beziehungsweise 20%.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Related Work</b>	<b>11</b>
<b>3</b>	<b>Background</b>	<b>15</b>
3.1	Reproducing Kernel Hilbert Spaces . . . . .	16
3.2	Gaussian Processes . . . . .	18
3.3	Bayesian Optimization . . . . .	21
3.4	Artificial Neural Networks . . . . .	23
<b>4</b>	<b>Methods</b>	<b>27</b>
4.1	Problem Statement . . . . .	28
4.2	The SoG activation function . . . . .	30
4.3	Bayesian Functional Optimization in Reproducing Kernel Hilbert Space using iGP-UCB . . . . .	31
4.4	Sparsification of Activation Functions . . . . .	35
<b>5</b>	<b>Evaluation</b>	<b>37</b>
5.1	MNIST Training with a Multilayer Perceptron . . . . .	38
<b>6</b>	<b>Discussion</b>	<b>45</b>
6.1	Results . . . . .	45
6.2	Possible Extensions and Limitations . . . . .	46
6.3	Conclusion . . . . .	48
	<b>Bibliography</b>	<b>49</b>



# List of Figures

3.1	Statistics of a Gaussian process prior and posterior distribution . . . . .	20
3.2	Evaluation of the MPI and UCB acquisition function . . . . .	22
3.3	Input-output mapping of an artificial neuron . . . . .	23
3.4	Forward propagation for a multilayer perceptron . . . . .	24
4.1	Approximation of the hyperbolic tangent function with a sum of Gaussians	31
5.1	Batch training of a multilayer perceptron for the MNIST data set . . . . .	38
5.2	Performance of PBO and BFO over 100 iterations . . . . .	41
5.3	Activation functions with 3 basis functions found by BFO . . . . .	42
5.4	Activation functions with 5 basis functions found by BFO . . . . .	43
5.5	Activation functions with 10 basis functions found by BFO . . . . .	44





# 1 Introduction

Artificial neural networks are used in a wide variety of fields, reaching from neuro science to image recognition, natural language processing, and the design of intelligent agents. In fact, neural networks are state of the art tools to parameterize and train complex non-linear models and power some of today’s most competitive algorithms in their respective fields. The nonlinearity of the neural network models come from the nonlinear activation function which is used in each artificial neuron. Different choices of these activation functions may lead to a very different behavior and performance of the network. However, while the structure of the network such as its depth, layer size, and type is generally considered to be crucial, often the activation functions are chosen from a small set of commonly used functions without further consideration.

In contrast, recent work by Agostinelli et al. [AHSB14] and Eisenach et al. [ELw17] showed that individual adaptive activation functions that are jointly trained with the network are clearly beneficial. However, they are still initialized with functions from the pool of commonly used activation functions. With steepest-descent-based training methods such as the standard backpropagation algorithm the resulting activation functions might be strongly related to their initialization as they get caught in near local minima. Turner and Miller [TM14] used an evolutionary algorithm that combines the strategy of selecting from previously defined activation functions and training of an additional scaling parameter. While both methods on their own were found to be beneficial, the combined strategy did not offer further advantages. However, they stated that the set of predefined activation functions and the range of the scaling parameter were very limited.

At the same time Bayesian optimization established itself as a main framework for hyperparameter optimization especially for objective functions that are expensive to evaluate. Bayesian optimization, however, does not scale well to higher dimensional problems. The number of samples needed to sufficiently cover the search space grows exponentially with its dimension. Wang et al. [WZH+13] (REMBO) and Djolonga et al. [DKC13] (SI-BO) address this problem by considering a lower dimensional embedding of the full search space that is expected to hold a good solution. Building on this idea, Ngo [Ngo16] (iGP-UCB) recently introduced the iGP-UCB algorithm for Bayesian functional optimization in possibly infinite dimensional reproducing kernel Hilbert spaces. The resulting search space does not rely on a set of predefined features or corresponding basis functions but adapts to the problems complexity.

Our work contributes to both of the presented research domains, as we aim to combine the search for optimal neural network activation functions with recent techniques in Bayesian optimization. More specifically, our goal is to find a near globally optimal activation function shared by all neurons for a given problem instance. Therefore, we model our activation functions as elements of a reproducing kernel Hilbert space and formulate the problem as a functional optimization problem. For finding optimal functions we use Bayesian functional optimization with Ngo’s iGP-UCB algorithm. The training method with Bayesian functional optimization outperforms standard parametric Bayesian optimization. The resulting models achieve a significant lower classification error compared to the jointly trained models and models with commonly used fixed activation functions.

## Outline

In the remainder, we first discuss related work in the field of neural networks and Bayesian optimization methods. In chapter 3 we introduce some basic theory and methods that will be required in the following chapters. In chapter 4 we first give a problem statement followed by the introduction of the sum of Gaussian activation function. Then we describe the Bayesian functional optimization framework with Ngo’s iGP-UCB algorithm and the kernel matching pursuit algorithm for sparsifying sum of Gaussian activation functions. In chapter 5 we present a detailed evaluation of the introduced training method for a multilayer perceptron that is trained on the MNIST data set.

## 2 Related Work

Research on neural network architecture that focused on the choice of the activation function goes back over twenty years. In 1996 Chen and Chang [CC96] used adaptive sigmoids that were trained by a steepest-descent-based autotuning algorithm. They found them to be beneficial compared to multilayer perceptrons with fixed sigmoid activation function. A different approach from Piazza et al. [PUZ92] investigated multilayer perceptrons that were using polynomial activation functions with adaptive coefficients. While this allowed them to reduce the size and complexity of the network, there were also drawbacks due to the unboundness of the activation function and global influence of the coefficients. As the coefficients have a global effect on the polynomial, changes that lead to locally better behavior in one part of the activation function may lead to worse behavior in other parts. Addressing these problems, Vecchi et al. [VPU98] and Guarnieri et al. [GPU99] used cubic splines with adaptive control points as activation functions. Those have the advantage that a change of one control point does not influence the activation function globally. To initialize the control points, uniform spaced samples from a sigmoid activation function were used. In recent work, Scardapane et al. [SSCU16] also used individual cubic spline activation functions for each neuron, but in a more efficient batch training setting. Additionally they introduced a novel regularization of control points to prevent overfitting. Here, the control points were initialized using samples from the hyperbolic tangent function with additional Gaussian noise. But also other than polynomial-type activation functions were studied. Agostinelli et al. [AHSB14] investigated the use of individual adaptive piecewise linear activation (APL) functions for deep neural network architectures. While the number of hinges of the activation functions was treated as a hyperparameter, the slopes of the single segments and the location of the hinges were trained jointly with the network by using standard gradient descent. Compared to a fixed rectifier linear activation function, they measured a relative improvement of 9.4% on the CIFAR-10 data set and 7.5% on the CIFAR-100 data set. They trained their network several times starting with randomly initialized activation functions. However, most of the learned activation functions were still very close to their initialization. Recently, another type of activation function was presented by Eisenach et al. [ELw17]. They were using a Fourier series basis expansion for nonparametric estimation of the activation functions for each neuron. Therefore, they presented a two-phase training procedure for convolutional neural networks which can be incorporated in the backpropagation framework. They initialized the activation functions of the fully connected layers to the Fourier series approximation of the hyperbolic tangent activation function. They achieved a relative

improvement of up to 15% on the MNIST and CIFAR-10 data set which gives further evidence for the potential of using problem specific trained activation functions.

While the stated results clearly show that adaptive activation functions can be beneficial and outperform commonly used fixed activation function, one crucial point remains the choice of their initialization. Especially when trained with steepest-descent-like algorithms, due to the local minima problem the initialization has great influence on the results. Therefore, the initialization with a specific function, e.g. the sigmoid or the hyperbolic tangent function, might force a strong prior over the potential space of activation functions. Early work that tried to explore the space of potentially very different types of activation functions was mainly focused on genetic and evolutionary algorithms that can choose from a predefined set of possible activation function as described by Yao [Yao99]. More recent work from Turner and Miller [TM14] combined the strategy of selecting from a predefined set of activation functions with the training of an additional scaling parameter for each neuron in the network. While both the strategies on their own were found to be beneficial, the combined strategy did not offer any additional advantage in their setting. However, they state that they only used a very limited set of activation functions to start with and optimized a single scaling parameter over a small range only.

Recently, we have also seen great successes of Bayesian optimization as a black-box and hyperparameter optimization method in many applications of Machine Learning. Especially for objective functions that are expensive to evaluate a good choice of evaluation points is crucial and the computational expense of Bayesian optimization becomes negligible. This makes Bayesian optimization a perfect fit for optimizing hyperparameters of neural networks as their training is time consuming and thus expensive. Snoek et al. [SLA12] used Bayesian optimization to optimize nine different hyperparameters of a three-layer convolutional neural network for the CIFAR-10 data set. The found hyperparameters were compared to hyperparameters that were hand tuned by experts to achieve state of the art performance and outperformed them by over 3%. However, standard Bayesian optimization does not scale well to higher dimensional search spaces. To converge to a globally optimal point, one needs samples that cover the search space sufficiently well. The amount of samples needed grows exponentially with the dimension of the search space. This was addressed by several papers in the past. Tyagi et al. [TGK14; TKGK16] presented an efficient sampling scheme for learning sparse additive models (SPAM) of cubic spline estimates. Their algorithm recovers a robust uniform approximation of the component functions using at most  $\mathcal{O}(d_{sparse}(\log d)^2)$  samples. Different approaches from Wang et al. [WHZ+16; WZH+13] (REMBO) or Djolonga et al. [DKC13] (SI-BO) assume that the possibly very high dimensional problem has only a small number of important dimensions which are dominating the problems solution. Once these dimensions are identified one can use Bayesian optimization in the lower dimensional embedding of the otherwise very high dimensional search space. While these approaches are concerned with the optimization of functions, Ngo [Ngo16] (iGP-UCB) proposed a framework for Bayesian functional optimization by defining the Bayesian optimization framework on a possibly infinite dimensional reproducing kernel Hilbert space. There-

fore, the used Gaussian process defines a distribution over functionals. The posterior belief over the loss functional is computed using a sparsified version of the previously found functions which keeps the computational cost under control. This results in a very flexible search space that does not rely on a set of predefined features or corresponding basis functions, but adapts to the problem's complexity.



## 3 Background

This chapter will briefly introduce some of the concepts and techniques that are used in chapter 4. First, there will be a brief introduction to reproducing kernel Hilbert spaces and why they are useful for machine learning. We will then discuss Gaussian processes as statistical models that define distributions over functions and how they enable us to infer an unknown target function. Based on this we will introduce the Bayesian optimization framework using a Gaussian process. Last, we will give a brief introduction to artificial neural networks and how they are trained and evaluated.

### 3.1 Reproducing Kernel Hilbert Spaces

A Hilbert space  $\mathcal{H}$  is a possibly infinite dimensional inner product space that is complete and separable with respect to the norm defined by the inner product. These requirements will basically enable us to apply concepts of finite dimensional linear algebra to infinite dimensional function spaces. In the following, we assume  $\mathcal{X}$  be a nonempty compact set, e.g.  $\mathbb{R}^n$ , and  $\mathcal{H}$  a Hilbert space of real valued functions  $f : \mathcal{X} \rightarrow \mathbb{R}$  with domain  $\mathcal{X}$ . Probably the most common example for a Hilbert space is the space of square integrable functions  $L^2$  with an inner product

$$\langle f, g \rangle_{\mathcal{H}} = \int_{-\infty}^{\infty} f(x)g(x) dx$$

that contains all function  $f$  for which the integral of absolute square values is bounded

$$\int_{-\infty}^{\infty} \|f(x)\|_2^2 dx \leq \infty.$$

Now consider the function  $f'(x)$  that equals  $f(x)$  everywhere, but on a finite set of points  $\mathcal{X}'$ . As the resulting function

$$f'(x) = \begin{cases} c & \text{if } x \in \mathcal{X}' \\ f(x) & \text{otherwise} \end{cases}$$

only differs from  $f(x)$  on a finite number of points it is itself a squared integrable function. However, this also implies that  $\|f - f'\|_{\mathcal{H}} = 0$  although  $f(x) \neq f'(x) \forall x \in \mathcal{X}'$ . Thus, if we want the functions  $f$  and  $g$  to be pointwise close when they are close w.r.t. the norm  $\|\cdot\|_{\mathcal{H}}$ , the square integrable condition is not strong enough. This is, however, what we want for a lot of machine learning tasks where we aim to learn a model to predict the outcome of an unknown target. Therefore, functions which are similar to the target function should predict similar outcomes on every possible data point. Reproducing kernel Hilbert spaces do fulfill this requirement. The definitions in this section are based on the work by Gretton [Gre13].

**Definition 3.1** *A reproducing kernel Hilbert (RKHS) space is a Hilbert space of functions where all evaluation functionals  $e_x(f) : f \mapsto f(x)$ ,  $f \in \mathcal{H}$  are bounded*

$$|e_x f| \leq \lambda_x \|f\|_{\mathcal{H}}.$$

This implies that if two functions are converging w.r.t. the RKHS norm  $\|\cdot\|_{\mathcal{H}}$  they also need to converge pointwise. Equivalently, one might define a reproducing kernel Hilbert space over his reproducing kernel where it also gets its name from.

**Definition 3.2** *A reproducing kernel Hilbert space is a Hilbert space  $\mathcal{H}$  with a reproducing kernel  $k$  whose span  $\{k(x, \cdot) \mid x \in \mathcal{X}\}$  is dense in  $\mathcal{H}$ .*

To understand this definition we first define what we understand to be a kernels and the properties that make it a reproducing kernel.



**Definition 3.3** A kernel is a function  $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$  that fulfills the following conditions:

1.  $k(x, x') = k(x', x) \quad \forall x, x' \in \mathcal{X}$  (symmetry)
2.  $\sum_{i,j=1}^n \alpha_i \alpha_j k(x_i, x_j) \geq 0 \quad \forall x_1, \dots, x_n \in \mathcal{X}, \alpha_1, \dots, \alpha_n \in \mathbb{R}$ . (semi pos. def)

**Definition 3.4** A kernel  $k$  is a reproducing kernel of a Hilbert space  $\mathcal{H}$  if

1.  $\forall x \in \mathcal{X} : k(x, \cdot) \in \mathcal{H}$
2.  $\forall f \in \mathcal{H}, x \in \mathcal{X} : f(x) = \langle k(x, \cdot), f(\cdot) \rangle$ . (reproducing property)

As  $k(x, \cdot)$  is itself a function in  $\mathcal{H}$ , it must hold that there exists a function  $k(y, \cdot)$  such that  $k(x, y) = \langle k(x, \cdot), k(y, \cdot) \rangle$ . Let  $k$  be a reproducing kernel for a Hilbert space  $\mathcal{H}$  consisting of the span of  $\{k(x, \cdot) | x \in \mathcal{X}\}$  and its completion then

$$\begin{aligned}
 |e_x f| &= |f(x)| = |\langle k(\cdot, x), f(\cdot) \rangle| && \text{(Reproducing property)} \\
 &\leq \|k(\cdot, x)\|_{\mathcal{H}} \cdot \|f\|_{\mathcal{H}} && \text{(Cauchy-Schwarz inequality)} \\
 &= \langle k(\cdot, x), k(\cdot, x) \rangle^{1/2} \cdot \|f\|_{\mathcal{H}} \\
 &= \sqrt{k(x, x)} \|f\|_{\mathcal{H}}
 \end{aligned}$$

implies that all evaluation functionals  $e_x$  are bounded and thus  $\mathcal{H}$  is a RKHS. Moreover, as the reproducing kernel can be written in terms of an inner product, it must be a positive definite symmetric kernel. But interestingly, also the reverse holds. The Moore–Aronszajn theorem states that every positive definite symmetric kernel defines a unique RKHS with itself being the reproducing kernel.

Another point that make reproducing kernel Hilbert spaces especially useful for machine learning is the existence of several representer theorems. These theorems essentially state that the minimizer of an empirical regularized risk function defined on a RKHS can be represented as a finite linear combination

$$f^*(x) = \sum_{i=1}^n \alpha_i k(x_i, \cdot) = \sum_{i=1}^n \alpha_i \langle k(x_i, \cdot), k(x, \cdot) \rangle$$

of kernel products that only depend on the inputs of the training data. The theorem was first introduced by Kimeldorf and Wahba [KW71] for a squared error formulation and additional  $L^2$  regularization  $\|f\|_{\mathcal{H}}$ . A more general version by Schölkopf et al. [SHS01] extended the theorem to arbitrary cost functions and regularizations  $g(\|f\|)$ , where  $g$  is a strictly monotonically increasing real valued function. Ultimately, this means that despite the RKHS being an infinite dimensional space, it is sufficient to search in a finite dimensional subspace defined by the training data. Such finite dimensional optimization problems are well understood and can be solved computationally.

## 3.2 Gaussian Processes

A Gaussian process (GP)  $\mathcal{GP}(\mu, \sigma^2)$  is the generalization of a multivariate Gaussian distribution to infinite dimensions. It describes a distribution over an infinite amount of random variables, where any finite subset of these random variables is distributed jointly Gaussian. Like a Gaussian distribution that is specified by its mean and covariance, a GP is specified by its mean function  $\mu(\cdot)$  and covariance function  $k(\cdot, \cdot)$ .

$$P(f) = \mathcal{GP}(\mu(\cdot), k(\cdot, \cdot))$$

To gain some intuition, consider an infinite set of random variables  $\{f_x^* : x \in \mathbb{R}^n\}$  and an unknown smooth target function  $f^* : \mathbb{R}^d \rightarrow \mathbb{R}$ . Each variable  $f_x^*$  represents the belief over the value of  $f^*(x)$  at the corresponding evaluation point  $x^* \in \mathbb{R}^d$ . As we want to model a distribution over a continuous smooth function it seems natural to assume that function values of close evaluation points are more correlated than distant ones. Therefore it makes sense to define the covariance function  $\text{cov}(f_x^*, f_{x'}^*) = k(x, x')$  as some measure of similarity between the corresponding evaluation points. Moreover, we define  $\mu(x)$  to represent the mean of the random variable  $f_x^*$  that is corresponding to the evaluation point  $x$ . By construction there exists a random variable for any possible evaluation point in the domain of the target function. Therefore, we eventually described a distribution over functions. The function  $k$  is also referred to as the Gaussian process kernel and should be a positive definite symmetric kernel. A commonly used GP kernel for many standard problems is the squared exponential kernel  $k(x, x') = \exp(-\|x - x'\|^2 / 2l^2)$  with bandwidth  $l$ . However, in general, as the kernel function is crucial for the behavior and later performance of the Gaussian process it should be chosen regarding to the specific problem domain.

If we want to take samples from the GP prior for computational reasons, we can not sample full functions but have to choose a finite subset of random variables  $F = \{f_i^*\}_{i=1}^n$ . For this subset we define the vector  $f_{1:n}^* = (f_1^*, \dots, f_n^*)^\top$  with the corresponding matrix of evaluation points  $X^* = (x_i^*, \dots, x_n^*)^\top$ . Further we define the matrix of pairwise GP kernels as

$$K(X, X') = \begin{bmatrix} k(x_1, x'_1) & \dots & k(x_1, x'_t) \\ \vdots & \ddots & \vdots \\ k(x_n, x'_1) & \dots & k(x_n, x'_t) \end{bmatrix} \quad \text{for } X \in \mathbb{R}^{n \times d}, X' \in \mathbb{R}^{t \times d}.$$

The resulting joint normal distribution of the random variables in  $F$  can be written in terms of the  $n$ -dimensional mean vector  $m$  and the  $n \times n$ -dimensional covariance matrix  $K(X, X)$ . For simplicity we chose the prior mean  $m = 0_n$ .

$$P(f_{1:n}^*) = \mathcal{N}(m, K(X^*, X^*)),$$

Now consider that we additionally have access to data  $D = \{(y_i, x_i)\}_{i=1}^t$  sampled from  $f^*$  and want to incorporate these information to update our prior belief  $P(f_{1:n}^*)$ . Moreover, we assume that the samples  $y_i = f^*(x_i) + \epsilon$  suffer from some additive Gaussian noise  $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathcal{I})$ . Let  $y_{1:t} = (y_1, \dots, y_t)^\top$  be the vector of sampled values and  $X = (x_1, \dots, x_t)^\top$  be the matrix of corresponding evaluation points then the resulting joint normal distribution can be written as follows.

$$P\left(\begin{bmatrix} y_{1:t} \\ f_{1:n}^* \end{bmatrix}\right) = \mathcal{N}\left(0, \begin{bmatrix} K(X, X) + \sigma^2 \mathcal{I} & K(X, X^*) \\ K(X^*, X) & K(X^*, X^*) \end{bmatrix}\right)$$

For a general joint Gaussian distribution  $P(a, b)$  the conditional distribution  $P(a | b)$  is also distributed jointly Gaussian and takes the following form.

$$\begin{aligned} P(a, b) &= \mathcal{N}\left(\begin{bmatrix} m_a \\ m_b \end{bmatrix}, \begin{bmatrix} A & C \\ C^\top & B \end{bmatrix}\right) \\ \Rightarrow P(a | b) &= \mathcal{N}(m_a + CB^{-1}(b - m_b), A - CB^{-1}C^\top) \end{aligned}$$

Applying this to the joint prior  $P(y_{1:t}, f_{1:n}^*)$  results in the posterior distribution  $P(f_{1:n}^* | y_{1:t})$  with posterior mean  $\tilde{m}$  and posterior covariance matrix  $\tilde{K}$ . We also introduce the shorthand notation  $K_t = K(X, X)$  for the Gram matrix of kernels between the  $t$  samples from the target function.

$$\begin{aligned} P(f_{1:n}^* | y_{1:t}) &= \mathcal{N}(\tilde{m}, \tilde{K}) \\ \tilde{m} &= K(X^*, X)(K_t + \sigma_n^2 \mathcal{I})^{-1}y, \\ \tilde{K} &= K(X^*, X^*) - K(X^*, X)(K_t + \sigma_n^2 \mathcal{I})^{-1}K(X, X^*) \end{aligned}$$

However, in the end we are interested in the posterior mean function  $\tilde{\mu}(x)$  and the posterior covariance function  $\tilde{k}(x, x')$  of the GP. We can identify them by taking a look at the single entries  $\tilde{m}_i = \tilde{\mu}(x_i^*)$  and  $\tilde{K}_{ij} = \tilde{k}(x_i^*, x_j^*)$  and define them for arbitrary  $x, x' \in \mathbb{R}^d$ . To have a nice and compact notation we defined  $k_t(x) = (k(x, x_1), \dots, k(x, x_t))^\top$ .

$$\begin{aligned} \tilde{\mu}(x) &= k_t(x)(K_t + \sigma_n^2 \mathcal{I})^{-1}y \\ \tilde{k}(x, x') &= k(x, x') - k_t(x)^\top (K_t + \sigma_n^2 \mathcal{I})^{-1}k_t(x') \\ \tilde{\sigma}^2(x) &= \tilde{k}(x, x) \end{aligned}$$

The posterior mean function represents our best guess of the target function given the observed data. It is also interesting to note that the mean function is just a finite linear combination

$$\sum_{i=1}^t \alpha_i k(x_i, x) \quad \text{with} \quad \alpha = (K_t + \sigma_n^2 \mathcal{I})^{-1}y$$

of  $t$  kernels centered at the observation data point  $\{x_i\}_{i=1}^t$ , although the GP can represent any function in the infinite dimensional RKHS defined by the positive definite

symmetric kernel  $k(x, x')$ . This is in fact a manifestation of the representer theorem discussed in section 3.1 as the mean function is just essentially just a formulation of kernel ridge regression. As the computation of the posterior mean and covariance involves the inversion of the  $N \times N$  covariance matrix it scales cubically with the number of samples. In practice, this means that we can not incorporate arbitrary many samples, if we want to compute the GP update in reasonable time. Figure 3.1 shows the mean function, the standard deviation, and sample functions drawn from the prior distribution and the posterior distribution for a 1D GP with a squared exponential kernel. The contents of this section are manly founded on the work of Rasmussen [Ras06] and Brochu et al. [BCD10].

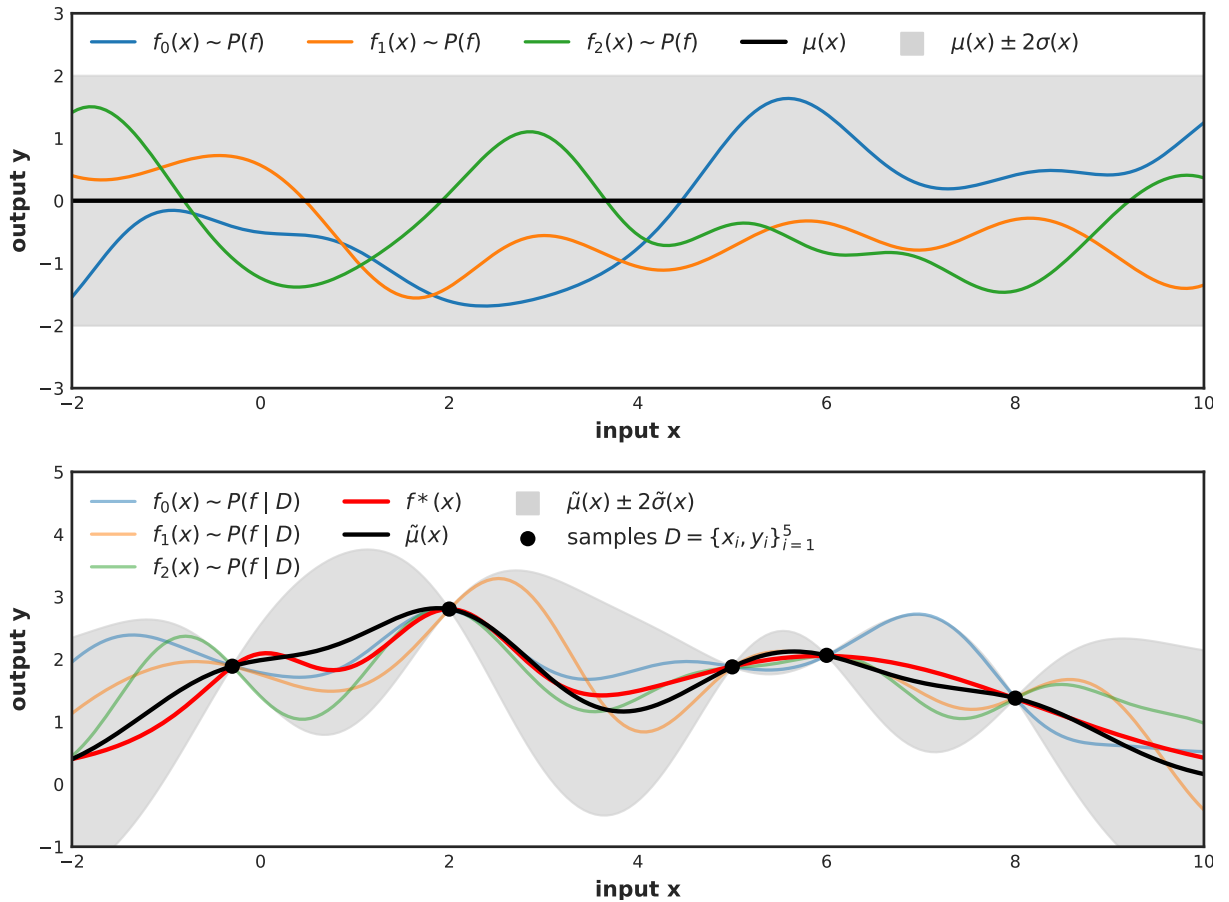


Figure 3.1: The mean, standard deviation, and samples from the GP prior (top) and GP posterior (bottom) for a 1D Gaussian process with squared exponential kernel.

## 3.3 Bayesian Optimization

---

**Algorithm 3.1** Generic Algorithm for Bayes Optimization

---

**Input** Objective  $f$ , prior belief  $P(f; \theta_0)$ , data  $D_0$

- 1: **repeat**
  - 2:   Compute posterior  $P(f \mid D_t; \theta_{t+1})$
  - 3:   Select  $x_{t+1} = \operatorname{argmax}_x u(x; \theta_{k+1})$
  - 4:   Sample  $y_{t+1} = f(x_{t+1}) + \epsilon$
  - 5:   Update data  $D_{t+1} = D_t \cup \{(x_{t+1}, y_{t+1})\}$
  - 6:   Tuning kernel hyperparameters
  - 7: **until** Convergence
- 

In the following, we consider maximization only, however, one can easily transform a given minimization problem into a maximization problem

$$\min_x g(x) = - \max_x (-g(x)).$$

Another assumption that is often made, is to require the objective function to be Lipschitz-continuous. This ensures that  $f(x)$  can not change arbitrarily when varying  $x$  but is bounded by a constant times the change. This is important as we want the samples we take to be locally representative for the values of the function. Without such assumptions we have no guarantees of finding a sufficiently good point in reasonable time. Therefore, in the following we assume sufficiently smooth objective functions.

Bayesian optimization is a sequential framework for global optimization and optimization of black-box functions. It typically assumes that the objective function  $f$  is sampled from a stochastic process and maintains a posterior distribution over the function as it samples more data over the course of the algorithm. When using a Gaussian process this results in updating the posterior mean and covariance functions based on samples  $D_t = \{(x_i, y_i)\}_{i=1}^t$  as described in section 3.2. In each iteration the current belief over  $f$  is used to determine the next sample point by maximizing an so-called acquisition or utility function  $u(x)$ . As indicated by the name, the acquisition function is a heuristic used to acquire the next evaluation point. Therefore, it rates potential evaluation points by their utility in finding the optimum. We usually want to select evaluation points that are expected to have a high value. On the other hand we also want to explore areas of high uncertainty which might lead to the discovery of even better locations for future function evaluations. The acquisition function balances exploration against exploitation and acts as a guide in the search for the optimum. The objective function is then evaluated at the selected position and the result is added to the data. Optionally, at the end of each iteration we can use the newly gained information to automatically tune the hyperparameters of our GP kernel, e.g. the bandwidth of the squared exponential kernel. This is normally done by selecting the parameters that maximize the log-likelihood on the data.

For the choice of the acquisition function  $u(x)$  there exist various options. Well known and often used heuristics are the Maximum Probability of Improvement (MPI), the Expected Improvement (EI) and the Upper Confidence Bound (UCB).

$$\text{MPI: } x_t = \operatorname{argmax}_x \int_{-\infty}^{y^*} \mathcal{N}(y \mid \mu(x), \sigma(x)) dx$$

$$\text{EI: } x_t = \operatorname{argmax}_x \int_{-\infty}^{y^*} \mathcal{N}(y \mid \mu(x), \sigma(x)) dx (y^* - y)$$

$$\text{UCB: } x_t = \operatorname{argmax}_x \mu(x) + \beta_t \sigma(x)$$

While the best choice of the acquisition function is arguably related to the specific problem, due to its simplicity and good performance UCB is often the default choice for many tasks. Furthermore, it was proven Srinivas et al. [SKKS09] that for an appropriate scheduling of parameter  $\beta_t$  the resulting heuristic (GP-UCB) with high probability has no regret. Still the maximization of a general acquisition function is a nonlinear nonconvex optimization problem. Thus, we need find a sufficiently good maximum to ensure that we take at least near optimal samples.

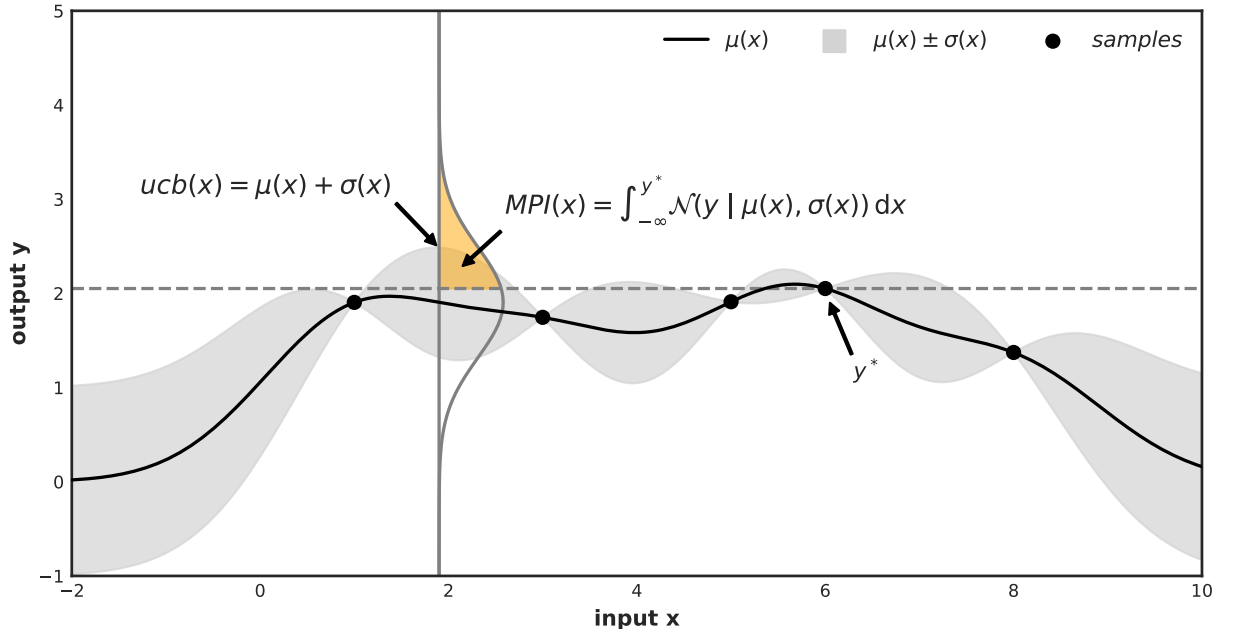


Figure 3.2: A 1D Gaussian process using a squared exponential kernel. The evaluation of the UCB and MPI acquisition functions at the next query point (selected by UCB) is shown graphically.

### 3.4 Artificial Neural Networks

For difficult supervised learning problems often complex and highly nonlinear models are needed to learn a sufficient mapping. Neural Networks offering a flexible framework to parameterize and train such models based on the combination of simple computational units called artificial neurons. The information processing is inspired by the way biological neural networks are working. Strongly simplified, neurons are connected through synapses that can transmit electrical signals. When the sum of input signals that a neuron receives surpasses a certain threshold the neuron itself starts sending signals to its peers. The networks working on this principle are typically referred to as multilayer perceptrons (MLP). Recent computational neural networks also involve advanced structures, e.g. convolutional layers and additional pooling layers. However, here we will focus on basic multilayer perceptrons. More formally such an artificial neuron consists of weights  $w_{1:n}$ , biases  $b_{1:n}$ , and a nonlinear activation function  $h$ . When inputs  $x_{1:n}$  are received the neuron calculates their weighted sum  $z = \sum_{i=1}^n w_i x_i + b$ . The sum is then used as an input to the activation function to compute the final output of the artificial neuron  $h(z)$  as shown in Figure 3.3. Common choices of the activation function are

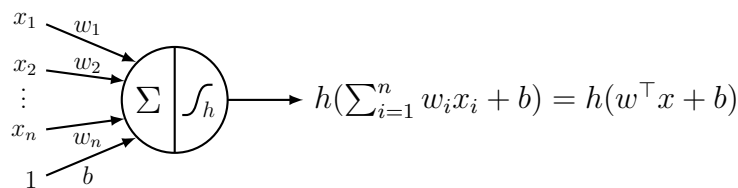
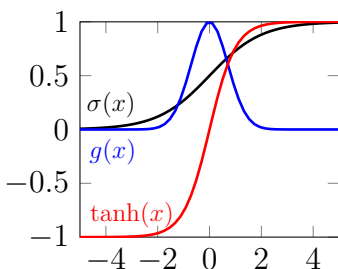


Figure 3.3: Input-output mapping of a single artificial neuron.

the logistic sigmoid, Gaussian, or hyperbolic tangent function stated in Equation 3.1. The choice of the activation function is important for several reasons, e.g. it might imply bounds for the neuron’s output by squashing the inputs into its co-domain. More importantly, it adds a layer of nonlinearity to the artificial neuron and to the resulting classifier or regression model. Otherwise the artificial neuron would just represent a linear model on the input data. One may notice that for choosing the activation function to be the logistic sigmoid the single neuron model corresponds to the class probability mapping of binary logistic regression. Also the performance and training of the later network can be strongly influenced by the choice of the actual activation function.



$$\begin{aligned}
 \sigma(x) &= \frac{1}{1 + \exp(-x)} \\
 \tanh(x) &= \frac{2}{1 + \exp(2x)} - 1 \\
 g(x) &= \exp\left(-\frac{(x - c)^2}{2l^2}\right)
 \end{aligned}
 \tag{3.1}$$

A single neuron with fixed activation function  $h$  does not offer a rich model suited for complex supervised learning tasks. Therefore a MLP uses multiple connected artificial neurons that are organized in layers. In fact, such feed forward networks are capable of universal function approximation as shown by Hornik [Hor91]. The first layer of the MLP is referred to as the input layer. It takes a vector  $x$  and passes it into the network. The predicted outcome computed by the network is given by the output of the last layer which we call the output layer. The layers in between are referred to as the hidden layers as the states of their artificial neurons are normally not observed. Passing an input

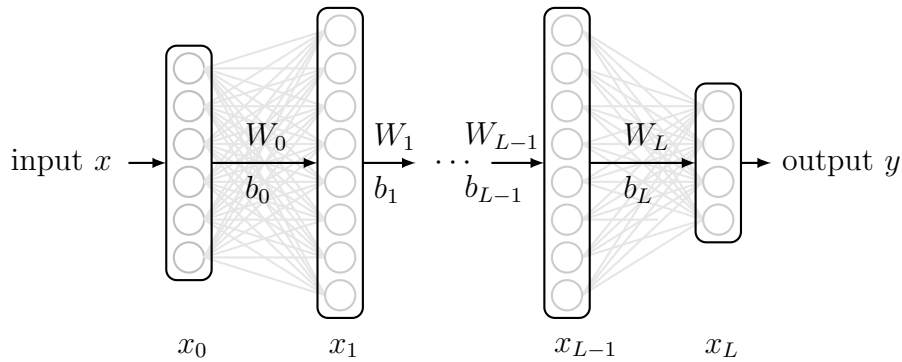


Figure 3.4: Forward propagation of an input vector through a multilayer perceptron with  $L$ -layers. The output layer gives out the computed output vector.

vector to the network and receiving a predicted outcome is called forward propagation. Rather than computing the outputs for each neuron individually we can express the computation for a full layer as a linear transformation of the layer's input vector followed by the element-wise evaluation of the activation function. The transformation matrix of the  $l$ -th layer  $W_l$  is a  $\dim(x_{l-1}) \times \dim(x_l)$  dimensional matrix where the each row represents the weight vector of the corresponding neuron in the same layer. Considering a network with  $L$  hidden layers the forward propagation of inputs can be written as stated in Equation 3.2.

$$\begin{aligned}
 x_0 &= x \\
 \forall l &= 1, \dots, L : \\
 z_l &= W_{l-1}x_{l-1} + b_{l-1} \\
 x_l &= h_l(z_l)
 \end{aligned} \tag{3.2}$$

In the case of a regression problem the activation function of the output layer is often chosen to be the identity such that only the linear transformation remains. Whereas in case of classification it is common to choose the softmax function to map the output to class probabilities.

Consider a supervised learning problem with the goal of learning a mapping  $f : \mathbb{R}^d \rightarrow \mathbb{R}^m$  given training data  $D = \{(x_i, y_i)\}_{i=1}^n$  and a loss function  $J$ . To find optimal parameters



for the neural network we are using the so-called backpropagation algorithm which is essentially a gradient descent method with recursive gradient computation. First, an input vector  $x$  from the training data is propagated forward through the network. The computed output  $y'$  is compared to the real output  $y$  from the training data by using the loss function  $J(W_{1:L}, b_{1:L}; x, y)$ . The computed error is then propagated back to the single neuron weights and biases. Finally, each parameter is updated w.r.t. its contribution to the overall error. In order to back-propagate the error and update the neuron's parameters one needs to compute the partial derivative w.r.t. the individual weights  $W_{l,i,j}$  and biases  $b_{l,i}$ . This is done recursively by computing the layer's derivatives w.r.t.  $z_l$  from back to front by using the chain rule as shown by Toussaint [Tou16].

$\forall L, \dots, 1 :$

$$\begin{aligned} \frac{dJ}{dz_l} &= \frac{dJ}{dz_{l+1}} \frac{\partial z_{l+1}}{\partial x_l} \frac{\partial x_l}{\partial z_l} \triangleq \delta_l \\ \frac{dJ}{dW_l} &= \delta_{l+1}^\top x_l^\top & \left( \frac{dJ}{dW_{l,i,j}} = \delta_{l+1,i} x_{l,j} \right) \\ \frac{dJ}{db_l} &= \delta_{l+1} & \left( \frac{dJ}{db_{l,i}} = \delta_{l+1,i} \right) \end{aligned}$$

When the derivatives w.r.t. the parameters are computed, we can update them by using standard gradient descent.

$$\begin{aligned} W_l &\leftarrow W_l - \alpha \frac{d}{dW_l} J(W_{1:L}, b_{1:L}) \\ b_l &\leftarrow b_l - \alpha \frac{d}{db_l} J(W_{1:L}, b_{1:L}) \end{aligned}$$

As we want to minimize the loss on the whole data set we do not want to calculate the gradient for an individual pair of training data only. The training inputs  $\{x_i\}_{i=1}^n$  are used to construct a matrix  $X = (x_1, \dots, x_n)^\top$ . Then forward propagating the whole matrix  $X$  results in an output matrix  $Y'$ . By extending the loss function to summing over the individual losses of all training examples it allows us to compute the gradient w.r.t. the full set of the training data. However, when working with large data sets computing the full gradient might be computational expensive and thus slow. Whereas the computation of gradient updates w.r.t. individual training samples is very cheap. Indeed, repeatedly doing gradient steps w.r.t. random samples of the training data leads to a stochastic approximation of standard gradient descent that is shown to almost certainly converge to a local minimum of the loss function with an appropriate decreasing schedule of the learning rate  $\alpha$  [Bot98]. This method of training is called stochastic gradient descent. On the other hand the update of the parameters suffers from a high variance as the gradients are computed w.r.t. individual training samples that may not agree on one particular

descent direction. This might lead to a much slower convergence compared to using the full gradient. Batch gradient descent tries to get the best out of both methods by considering small random batches of training data. This has the advantage of still being considerable fast while having a lower update variance compared to plain stochastic gradient descent. As the batches for training are selected randomly and each training session typically starts with parameters that are initialized with some degree of randomness, different training sessions lead to potentially very different network parameters. Therefore, multiple training sessions are launched and the best of the resulting models is selected.

To evaluate the resulting models we have to take data which was not used for training the network as we are interested in the model that generalized best to yet unseen data. The initial set  $D$  of training data is split into a training set  $D_{train}$ , validation set  $D_{val}$ , and test set  $D_{test}$ . Then the network is trained on  $D_{train}$  only while the error on the validation data set can be used to evaluate and tune the model's hyperparameters across training sessions or as a stopping criterion for the training algorithm to avoid overfitting. For estimation of the classification or regression error of the final model the test set is used. If one would use the test or validation set for this task the error estimate would be biased and therefore be too low as the test and validation set were also involved in the selection or even training of the final model.

At the end, as it is a currently very active field of research, we want to briefly discuss an example that should give some basic intuition on how neural networks are related to kernel methods and reproducing Kernel Hilbert spaces. Considering a 1D regression problem and a squared loss function the corresponding MLP model as discussed above looks like

$$\begin{aligned} f(x) &= w_L^T \underbrace{h_{L-1}(W_{L-1} h_{L-2}(\dots h_0(W_0 x + b_0) \dots) + b_{L-1})}_{\phi_L(x)} + b_L \\ &= w_L^T \phi_L(x) + b_L = w^T \phi(x). \end{aligned}$$

One interpretation is that the network is actually learning a feature map to represent the input data. The inner product of these feature maps implies a symmetric positive definite kernel  $k(x, x') = \phi(x)^\top \phi(x')$ . Due to this we also say the network is learning a kernel. This gets even more clear if we solve the linear regression problem and rewrite it by using the Woodbury identity (kernel trick).

$$w^* = (\Phi^\top \Phi)^{-1} \Phi^\top y = \Phi^\top (\Phi \Phi^\top)^{-1} y \quad \Rightarrow \quad f(x) = \underbrace{\phi(x)^\top \Phi^\top}_{k_t(x)^\top} \underbrace{(\Phi \Phi^\top)^{-1} y}_{K_t^{-1} y}.$$

This is a kernel regression formulation similar to the one of the mean function of Gaussian processes stated in section 3.2 but without regularization, where  $k_t$  the vector of kernels between the input and the training data points and  $K_t$  is the Gram matrix of pairwise kernels between training data points.

## 4 Methods

Instead of using predefined activation functions like the sigmoid or hyperbolic tangent activation function we want to find an activation function that is optimized for a given problem instance. This is done by optimizing an loss functional with methods from Bayesian optimization. To point this out we will refer to this method as Bayesian functional optimization (BFO). This section describes the steps that we have taken in detail starting with a problem statement. We will further introduce the Bayesian functional optimization framework and the sum of Gaussians (SoG) activation function. Last, we will discuss the sparsification of SoG activation functions using the kernel matching pursuit algorithm.

## 4.1 Problem Statement

In the following we consider a multilayer perceptron with  $L$  hidden layers. The corresponding model  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  is a function that maps a  $n$ -dimensional input vector  $x \in \mathbb{R}^n$  to a  $m$ -dimensional output vector  $y \in \mathbb{R}^m$  and is parameterized by weights  $W_{0:L}$ , biases  $b_{0:L}$  and an activation function  $h \in \mathcal{H}$ . While the activation function  $h$  is the same for all hidden layer neurons the neurons in the output layer may have an activation function  $g$  that is different from  $h$ . E.g. for classification, the output layer might use the softmax function in order to map the output to class probabilities. This results in the model

$$f(x; h, W_{0:L}, b_{0:L}) = g(W_L h(W_{L-1} h(\dots h(W_0 x + b_0) \dots) + b_{L-1}) + b_L).$$

Optimally we want to find parameters  $W_{0:L}^*, b_{0:L}^*$  and an activation function  $h^*$  that are minimizing the loss functional  $l$  over some distribution of data  $P(D)$ . For a general multilayer perceptron and loss functional  $l$  this is a nonlinear nonconvex optimization problem

$$\min_{h, W_{0:L}, b_{0:L}} l(h, W_{0:L}, b_{0:L}; X, Y) \quad (4.1)$$

$$= \min_h \left( \min_{W_{0:L}, b_{0:L}} l(W_{0:L}, b_{0:L}, h; X, Y) \right). \quad (4.2)$$

In general such optimization problems are hard to solve and may not have a closed form solution. Using first or second order gradient-descent-based methods does not guarantee to find a globally optimal solution. Thus, when jointly optimizing  $W_{0:L}$ ,  $b_{0:L}$ , and the parameterized activation function  $h$ , we observe that the resulting activation function  $h^*$  is strongly related to its initialization, since it is not able to escape all the local minima during training.

The problem is split into two coupled optimization problems as shown in Equation 4.2. In case of finding the globally optimal set of parameters, these two formulations are exactly the same. However, considering the local minima problem the separate formulation might be of advantage. The inner optimization problem is the training of the network for a fixed activation function  $h$  using gradient descent methods. Alternatively, for the training of the network, we can still use a joint training procedure that uses the selected activation function as an initialization only. The outer problem takes the loss of the trained network as a response to select a new activation function using Bayesian functional optimization. This allows a much better exploration of the space of possible activation functions. In theory, if we neglect the potentially suboptimal response of the inner problem, this will result in finding the globally optimal activation function  $h^*$  as the number of iterations converges to infinity. In practice, we do only have finite time and need to train the full network for a new activation function in every iteration. Therefore, fast convergence to a near global optimum must be ensured to be of practical use. Moreover, as we only have a limited amount of data the final training routine needs to prevent overfitting to ensure

good generalization to unseen data. This can be achieved by splitting the available data into a training set  $D_{train}$ , a validation data set  $D_{val}$ , and a test set  $D_{test}$ . While the network is trained on  $D_{train}$ , the error on the validation data set  $D_{val}$  is monitored and it is used as a stopping criterion. The final training of the network is shown in algorithm 4.1.

---

**Algorithm 4.1** Network Training

---

```

1: function TRAINNETWORK(activation function  $h$ )
2:   Initialize weights  $W_{0:L}^0$  and biases  $b_{0:L}^0$ 
3:   Initialize patience  $p$ , minimal loss  $l_{min} \leftarrow \infty$ 
4:   repeat
5:      $W_{0:L}^{k+1}, b_{0:L}^{k+1} \leftarrow \text{GD-Optimizer}(W_{0:L}^k, b_{0:L}^k, h, X_{train}, Y_{train})$ 
6:      $l_{k+1} = l(X_{val}, Y_{val}; W_{0:L}^{k+1}, b_{0:L}^{k+1}, h)$ 
7:
8:     if  $l_{k+1} < l_{min}$  then
9:        $l_{min} \leftarrow l_{k+1}$ 
10:    else  $p \leftarrow p - 1$ 
11:
12:   until  $p < 0$ 
13:   return  $l_{min}$ 
14: end function

```

---

The Bayesian functional optimization routine which takes the loss of the network training as an objective function is stated in algorithm 4.2. By considering the activation functions to have a fixed size parameterization, we can represent them as simple parameter vectors. Standard Bayesian optimization then works analogously to section 3.3. However, Bayesian functional optimization with Ngo’s [Ngo16] iGP-UCB algorithm is used which is described in detail in section 4.3.

---

**Algorithm 4.2** Bayesian Functional Optimization for activation functions

---

```

1: function OPTIMIZEACTIVATION
2:   repeat
3:     Update posterior distribution.
4:     Select new activation function  $h_{t+1}$  (iGP-UCB)
5:      $l_{t+1} \leftarrow \text{trainNetwork}(h_{t+1})$ 
6:     Update data  $D_{t+1} = D_k \cup \{(h_{t+1}, l_{t+1})\}$ 
7:     Optimize hyperparameters
8:   until Convergence/Max number of iterations reached
9:   return  $h^*$ 
10: end function

```

---

## 4.2 The SoG activation function

We choose the activation function  $h$  to be a linear combination of Gaussian radial basis functions (RBF)  $k(x, x')$ . This is a very flexible model that offers a rich representation and makes it easy to approximate various functions, including commonly used functions such as the logistic sigmoid or hyperbolic tangent. Given any function  $f \in L^2\{\mathbb{R}\}$  an arbitrary good approximation by a finite linear combination of RBFs  $k_i$  with the same scale exists as shown by Park and Sandberg [PS91] in the context of RBF networks. In fact, the activation function  $h$  represents a RBF network with one hidden layer and  $N$  neurons and is thus an universal function approximator

$$k(x, x_i) = k_i(x) = \exp\left(\frac{-\|x - x_i\|^2}{2\sigma^2}\right)$$

$$h(x) = \sum_{i=1}^N \alpha_i \cdot k(x, x_i), \quad \alpha_i \in \mathbb{R}.$$

An activation function  $h$  consists of  $N$  RBFs and can be represented parametrically by centers  $x_{1:N}$ , weights  $\alpha_{1:N}$ , and a single scale  $\sigma$ . More importantly, this allows us to model the activation function in a reproducing kernel Hilbert space  $\mathcal{H}_k$  with reproducing kernel  $k(x, x')$ . This can be achieved for any positive definite kernel as described in section 3.1. The kernel scale  $\sigma$  is chosen heuristically as

$$\sigma = \gamma \frac{b_u - b_l}{N}$$

which offers good support on a desired interval with lower bound  $b_l$  and upper bound  $b_u$ . The size of the interval should be chosen w.r.t. the corresponding problem. If the input is normalized and the weights of the neural network are regularized one can normally guess an appropriate symmetric interval with  $b_l = -b_u$ . If needed, e.g. when considering outliers or unnormalized data with high variance, one might also chose  $h$  to keep up a constant level outside of the supported interval.

$$h(x) = \begin{cases} h(b_l) & x < b_l \\ \sum_{i=1}^N \alpha_i \cdot k(x, x_i) & x \in [b_l, b_u] \\ h(b_u) & x > b_u \end{cases}$$

The limiting factor of the representation capability of the activation function is the number  $N$  of used RBFs. However, smaller  $N$  speed up the learning process as they decrease the parameters and complexity of the activation function that is optimized by Bayesian functional optimization and used to train the network. Less complex activation functions might also be beneficial for the generalization capability of the resulting model. For example the approximation of the hyperbolic tangent on the interval  $[-7.5, 7.5]$ , a small  $N = 10$  is already enough to achieve a good approximation as shown in Figure 4.1.

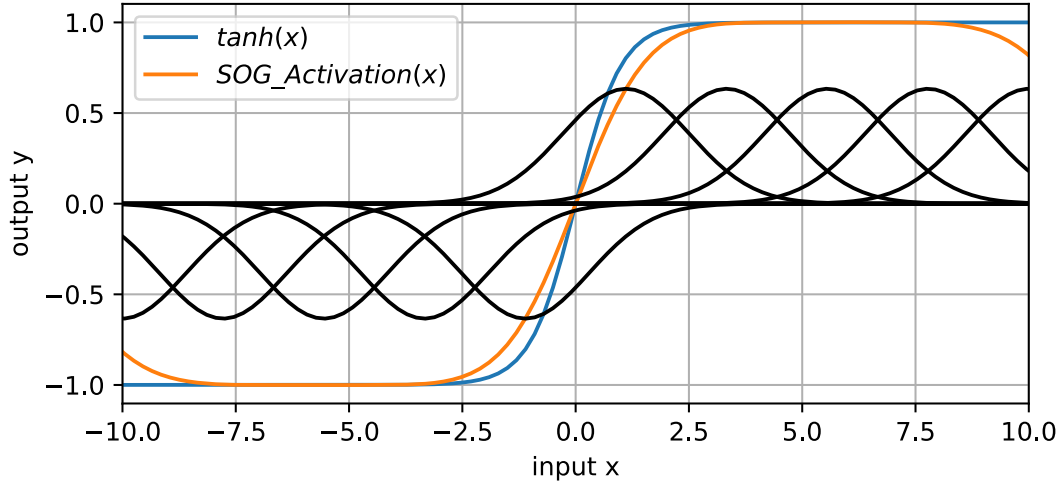


Figure 4.1: Approximation of the hyperbolic tangent function with a sum of Gaussians

### 4.3 Bayesian Functional Optimization in Reproducing Kernel Hilbert Space using iGP-UCB

Using standard Bayesian optimization for optimizing the activation functions in the probably high dimensional joint parameter space of weights  $\alpha_{1:N}$  and centers  $x_{1:N}$  has the disadvantage of not scaling well to higher dimensional problems ( $d > 10$ ). To converge to a globally optimal point in some bounded subset  $\mathcal{X} \subset \mathbb{R}^d$  one needs samples that cover  $\mathcal{X}$  sufficiently well. However, as the dimension  $d$  of the search space increases, the amount of samples needed to sufficiently cover  $\mathcal{X}$  grows exponentially. Also optimizing the typically nonconvex acquisition function for selecting the next query point gets increasingly challenging in higher dimensions. Finding appropriate step sizes and directions for steepest-descent-based optimization in the joint parameter is difficult. This is due to the space of weight and centers are different metric spaces and are influencing the activation functions in a very different ways. Even when fixing the centers  $x_i$  and only considering the weights  $\alpha_i$ , points that are close in weight space when measured with the euclidean distance, may not correspond to close functions in function space  $\mathcal{H}$ . The Gaussian processes allows to fix this with the use of an appropriate kernel that describes the connection between parameters and the corresponding functions in  $\mathcal{H}$ . Additionally, when optimizing the acquisition function we should take steps w.r.t. the correct metric of the underlying space  $\mathcal{H}$ .

Bayesian functional optimization (BFO) with iGP-UCB as described by Ngo [Ngo16] implicitly addresses these problems by defining the optimization problem over a reproducing kernel Hilbert space (RKHS). More specifically, it aims to optimize a loss functional  $l : \mathcal{H} \rightarrow \mathbb{R}$ , where the input space  $\mathcal{H}$  is a RKHS with reproducing kernel  $k$  as defined in section 3.1.

---

**Algorithm 4.3** Bayesian Functional Optimization with iGP-UCB
 

---

- 1: Initialize data  $D_0 = \emptyset$
  - 2: Initialize prior mean  $\mu_0 = 0$
  - 3: **repeat**
  - 4:   Select  $h_{t+1} = \operatorname{argmax}_{h \in \mathcal{H}} \operatorname{ucb}_t(h)$
  - 5:   Sparsify  $h_{t+1}$  to get a compact function  $\tilde{h}_{t+1}$
  - 6:   Sample  $y_{t+1} = f(\tilde{h}_{t+1}) + \epsilon_{t+1}$
  - 7:   Update data  $D_{t+1} = D_t \cup \left\{ \left( \tilde{h}_{t+1}, y_{t+1} \right) \right\}$
  - 8:   Compute posterior mean  $\mu_{t+1}$  and covariance  $K_{t+1}$
  - 9:   Tune kernel hyperparameter
  - 10: **until** Convergence
- 

Our Gaussian process kernel is defined as

$$K(h, g) = \exp \left( \frac{-\|g - h\|_{\mathcal{H}}^2}{2l^2} \right),$$

where  $\|\cdot\|_{\mathcal{H}}$  is the RKHS norm induced by the inner product of  $\mathcal{H}$ . As an activation function  $h \in \mathcal{H}$  is just a finite linear combination of basis functions we can express the squared distance between  $g$  and  $h$  in terms of the coefficients  $\alpha_{1:N}$ ,  $\beta_{1:M}$  and basis functions with corresponding centers  $x_{1:N}^{(g)}$ ,  $x_{1:M}^{(h)}$ . This results into

$$\begin{aligned} \|g - h\|_{\mathcal{H}}^2 &= \langle g - h \mid g - h \rangle \\ &= \sum_{i=1, j=1}^{N, N} \alpha_i \alpha_j k(x_i^{(h)}, x_j^{(h)}) \\ &\quad + \sum_{i=1, j=1}^{M, M} \beta_i \beta_j k(x_i^{(g)}, x_j^{(g)}) \\ &\quad - 2 \sum_{i=1, j=1}^{N, M} \alpha_i \beta_j k(x_i^{(h)}, x_j^{(g)}) \\ &= \alpha^\top K^{(gg)} \alpha + \beta^\top K^{(hh)} \beta - 2 \alpha^\top K^{(hg)}. \end{aligned}$$

The coefficient vectors  $\alpha$  and  $\beta$  can be represented in their joint basis  $\{k(x, \cdot) \mid x \in x_{1:N}^{(g)}\} \cup \{k(x, \cdot) \mid x \in x_{1:M}^{(h)}\}$  and states the distance as a single quadratic term

$$\|g - h\|_{\mathcal{H}}^2 = (\alpha - \beta)^\top K (\alpha - \beta).$$

The kernel matrix  $K$  consists of all pairwise inner products of these basis. The update



of the posterior mean and variance

$$\mu_t = k_t(h)^\top (G_t + \sigma_n^2 \mathcal{I})^{-1} y_t \quad (4.3)$$

$$\begin{aligned} K_t(h, h') &= K(h, h') - k_t(h)^\top (G_t + \sigma_n^2 \mathcal{I})^{-1} k_t(h') \\ \sigma_t^2(h) &= K_t(h, h) \end{aligned} \quad (4.4)$$

$$\begin{aligned} k_t(h) &= (K(h, h_1), K(h, h_2), \dots, K(h, h_t))^\top \\ G_{t,ij} &= K(h_i, h_j) \quad \forall i, j \in \{1, \dots, t\} \end{aligned}$$

are very similar to the standard GP versions.  $G_t$  is the  $t \times t$  dimensional Gram matrix of pairwise GP kernels between the stored functions. The  $t$ -dimensional vector  $k_t(h)$  contains the GP kernels between the input and the stored functions. Using the mean and variance as described in Equation 4.3 and Equation 4.4 lead to the expression for the UCB aquisition function

$$\text{ucb}(h) = \mu_t(h) + \beta_t(h) \sqrt{\sigma_t^2(h)}.$$

In order to find the function  $h^*$  that maximizes the aquisition function  $\text{ucb}(h)$  we compute its functional gradient w.r.t.  $h$ . Therefore, we first compute the gradients of the GP kernel

$$\frac{\partial}{\partial h} K(h, h') = (h' - h)/l^2 \exp\left(\frac{-\|h' - h\|^2}{2l^2}\right)$$

and the posterior mean and variance functions

$$\begin{aligned} \frac{\partial}{\partial h} \mu_t(h) &= \nabla_h k_t(h)^\top (G_t + \sigma_n^2 \mathcal{I})^{-1} y_t \\ &= \underbrace{((h - h_i)_t / l^2 * k_t(h))^\top}_{\nabla_h k_t(h)^\top} \underbrace{(G_t + \sigma_n^2 \mathcal{I})^{-1} y_t}_{\tilde{G}_t^{-1}} \end{aligned} \quad (4.5)$$

$$\frac{\partial}{\partial h} \sigma_t^2(h) = -2 \nabla_h k_t(h)^\top \tilde{G}_t^{-1} k_t(h).$$

Remark, the expansion of  $\nabla_h k_t(h)$  in Equation 4.5 which shows more intuitively that the functional gradient is just a linear combination of stored functions  $h_i$  and the input function  $h$ . Further  $(h - h_i)_t = (h - h_1, \dots, h - h_t)^\top$  and  $*$  denotes the element-wise multiplication. Hence, the overall UCB gradient is

$$\begin{aligned} \frac{\partial}{\partial h} \text{ucb}(h) &= \frac{\partial}{\partial h} \mu_t(h) + \underbrace{\beta_t}_{\frac{1}{\sqrt{\sigma_t^2(h)}}} \frac{1}{2} \frac{\partial}{\partial h} \sigma_t^2(h) \\ &= \nabla_h k_t(h)^\top \tilde{G}_t^{-1} (y_t - \tilde{\beta}_t k_t(h)). \end{aligned}$$

By applying the update rule

$$h \leftarrow h + \Lambda \left( (h - h_i)_t * k_t(h) \right)^\top \hat{G}_t^{-1} (y - \tilde{\beta}_t k_t(h)) \quad (4.6)$$

with learning rate  $\Lambda$ , the functional UCB gradient can be used to do gradient steps directly in RKHS. However, for computational reasons, the activation function is represented as a finite linear combination of basis functions  $k(x_i, x)$ . Then the update of the corresponding coefficients  $\alpha$  can be expressed as

$$\alpha \leftarrow \alpha + \Lambda (\alpha - \alpha_i)_{1:t} * k_t^T \hat{G}_t^{-1} (y - \tilde{\beta}_t k_t). \quad (4.7)$$

For the selection of an appropriate step size  $\Lambda$  a backtracking line search is used. Note that the resulting update is the same as starting directly in the space of coefficients and doing gradient steps w.r.t. the metric  $\|h\|_{\mathcal{H}}^2 = \alpha^T K \alpha$  given by the kernel matrix  $K$ .

It is important that the basis which spans the search space is chosen large enough and well distributed to ensure a rich function representation in the supported domain of the activation function. From Equation 4.6 and Equation 4.7 we can observe that this finite basis consists of the basis functions which represent the sampled functions  $h_1, \dots, h_t$  and the function  $h_0$  that is used to initialize the gradient descent optimizer. Thus, if the joint basis representation of  $h_{0:t}$  is large it will result in a large search space. On the other hand functions with a large basis representation may slow down the GP update as the computation of the GP kernel scales quadratically with the number of basis functions.

iGP-UCB sparsifies the function  $h^*$  obtained from gradient descent and stores only the sparse approximation  $\tilde{h}^*$  consisting of the most significant basis functions. However, to guarantee a rich representation of the search space, the initial function  $h_0$  for gradient descent is chosen to have a large enough and well distributed basis. Therefore, we uniformly sample the centers  $x_i$  that define the basis functions  $k(x_i, \cdot)$  from the bounded domain of the activation function. This choice implicitly fulfills the upper and lower bound constraints on the domain of centers. Constraints on the coefficients are handled in a soft manner by regularization during sparsification.

The resulting sparse function  $\tilde{h}^*$  is used to evaluate the objective function. Here the objective function is the loss on the validation data for the neural network that was trained using  $\tilde{h}^*$ . We then append the sparse activation function and the returned loss to the data set  $D_t$ . This results in an incrementally extending search space that is spanned by the significant basis functions from previous iterations and randomly sampled candidate basis functions.

## 4.4 Sparsification of Activation Functions

We sparsify our activation functions using the kernel matching pursuit algorithm with pre-fitting by Vincent and Bengio [VB02]. It takes data  $\{(x_i, y_i)\}_{i=1}^l$  sampled from a function  $h$  and a dictionary of basis functions  $\mathcal{D} = \{k_i\}_{i=1}^M$  and computes a sparse approximation  $\tilde{h}$  consisting of  $N$  of these basis function. This is achieved by solving  $N$  times the optimization problem

$$\min_{k_{n+1}, \alpha_{1:n+1}} \left\| \left( \sum_{i=1}^n \alpha_i \vec{k}_i \right) + \alpha_{n+1} \vec{k}_{n+1} - \vec{y} \right\|^2 \quad \forall n = 1, \dots, N$$

with  $\vec{y} = (y_1, \dots, y_M)^\top$ ,  $\vec{k}_i = (k(x_i, x_1), \dots, k(x_i, x_l))^\top$ .

In contrast to the back-fitting approach which selects a new basis function and then optimizes the coefficients accordingly, the pre-fitting approach jointly optimizes the next basis function and coefficients. In each iteration it selects new optimal coefficients and an additional basis function that expands the so far solution. Following the notation of Vincent and Bengio,  $\vec{y}$  denotes the vector of evaluations of the function  $h$  at  $x_{1:l}$  while  $\vec{k}_i$  is the vector of evaluations of the  $i$ -th basis function at these points. The actual kernel matching pursuit algorithm solves the optimization problem above very efficiently by making use of orthogonality properties. It only takes two passes over the dictionary of basis functions in each iteration. This results in an overall algorithm with time complexity  $\mathcal{O}(NML)$ . For a detailed algorithmic description and additional explanation regarding the kernel matching pursuit algorithm we refer to the work by Vincent and Bengio [VB02].

Sparsifying a SoG activation function  $h$  with  $M$  basis functions  $k_{1:M}$  as described in section 4.2 is a special case of the algorithm, as we already know a good set of dictionary functions and evaluation points. We select  $\mathcal{D}$  exactly to contain the  $M$  basis function of  $h$  and the evaluation points  $x_{1:l}$  ( $l = M$ ) to be the centers corresponding to these basis functions. Therefore,  $\vec{k}_i$  is exactly the  $i$ -th row of the kernel matrix  $K$  with  $K_{ij} = k(x_i, x_j)$ . Additionally, after the algorithm we do one more iteration of back-fitting with regularization on the coefficients  $\alpha$ . This regression problem can be solved analytically and gives the final coefficients

$$\min_{\alpha_{1:N}} \left\| \underbrace{\left( \sum_{i=1}^N \alpha_i \vec{k}_i \right)}_{K\alpha} - \vec{y} \right\|^2 + \|\alpha\|^2 \quad \Rightarrow \quad \alpha^* = (KK + \lambda\mathcal{I})^{-1} K\vec{y}.$$



## 5 Evaluation

In this chapter we state our evaluation results of the presented methods for the MNIST data set. Therefore, we first evaluated the performance of commonly used fixed activation functions and the SoG activation function that was trained jointly with the network's parameters. Next, we evaluated the separate training procedure using Bayesian functional optimization and compared it to standard parametric Bayesian optimization.

## 5.1 MNIST Training with a Multilayer Perceptron

The MNIST database consists of labeled 28x28 pixel greyscale images of handwritten digits. It contains a test data set of 10,000 data tuples and a training data set of 60,000 data tuples. From the training data set we use 5,000 data tuples as a validation data set. Each data tuple consists of the vector representation of an image  $x \in [0, 1]^{784}$  and a corresponding one-hot-encoded label  $y \in \{0, 1\}^{10}$  with  $\sum_i^{10} y_i = 1$ . We train a multilayer perceptron with 2 hidden layers containing 500 and 300 neurons as depicted in Figure 5.1. Each hidden layer neuron is using the SoG activation function described in section 4.2 while the neurons in the output layer are using the softmax function to map to final class probabilities. The network is trained using the cross entropy loss and stochastic batch gradient descent with batches of size 100. The multilayer perceptron and the training procedure were implemented with the free machine learning library TensorFlow. For stochastic batch gradient descent we used TensorFlow's implementation of the adaptive moment estimation optimizer (ADAM) by Kingma and Ba [KB14]. ADAM estimates the mean and variance of past exponentially decayed gradients to balance new gradients and computes adaptive learning rates for each parameter.

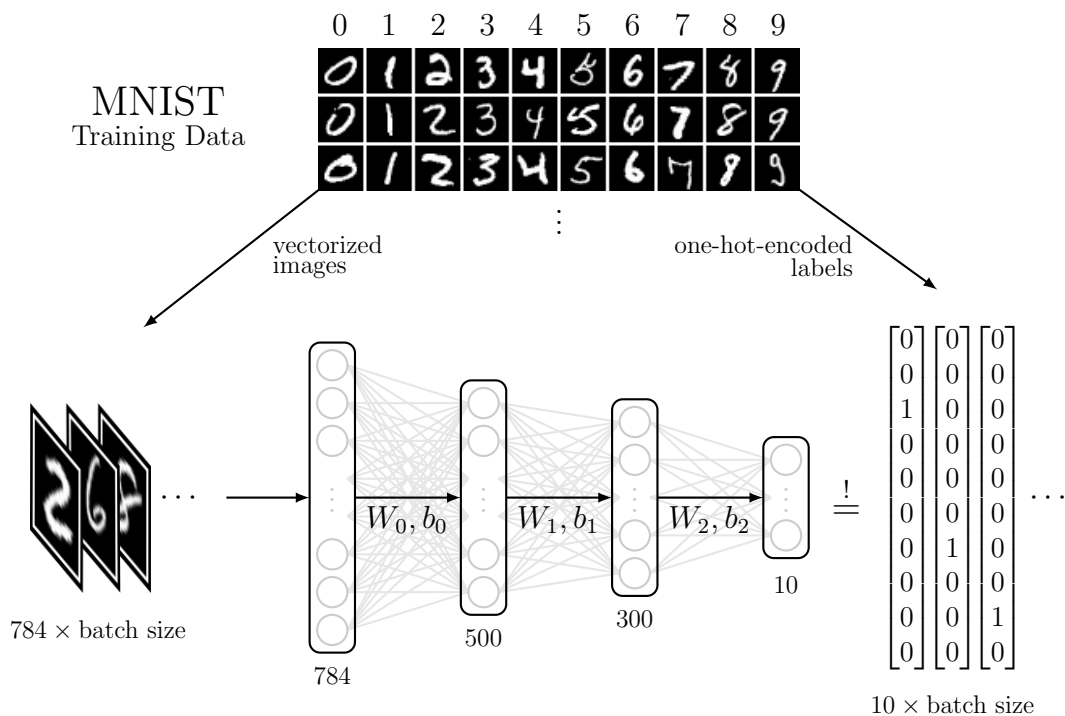


Figure 5.1: Batch training of a MLP with 2 hidden layers for the MNIST data set. The batches of the vectorized images and corresponding one-hot-encoded labels are sampled randomly from the training data set.

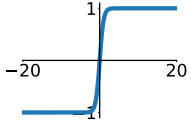
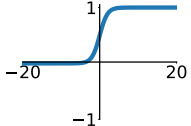
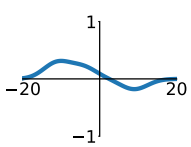
Training Method	Mean CE Val. Data	Std. CE Val. Data	Mean Error Val. Data	Test CE Best Model	Test Error Best Model	Activation Best Model
Fixed Tanh AF	0.2682	0.0131	7.55%	0.2360	6.67%	
Fixed Sigmoid AF	0.1124	0.0066	3.36%	0.0977	2.94%	
Joint Training SoG AF (3 BFs)	0.0938	0.0075	2.67%	0.0773	2.23%	

Table 5.1: MNIST results for the fixed sigmoid, fixed hyperbolic tangent, and the SoG activation function that was jointly trained with the network. Each version was run for 100 times

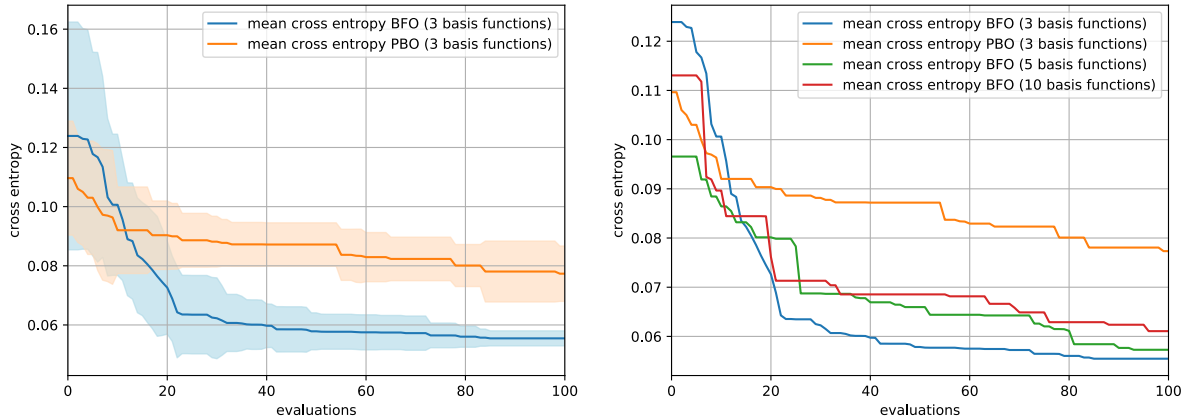
To have a baseline we first evaluate 100 training sessions for each, the sigmoid and the hyperbolic tangent activation function. As a stopping criterion for the network training we used the monitored validation error with a high patience  $p = 2000$ . This is to ensure convergence of the stochastic batch gradient descent as, due to its high update variance, it often overcomes small local minima and jumps to better ones. For the presented MLP the sigmoid activation function performs better than the hyperbolic tangent function and achieves a classification error of 2.94% on the test data. Next, we evaluate the performance of the SoG activation function with 3 basis functions, whose centers and weights are trained jointly with the network parameters by stochastic batch gradient descent. We randomly initialized the basis function centers from the interval  $[-5, 5]$  and the corresponding weights from the interval  $[-1, 1]$ . As described in section 4.2 we chose the bandwidth  $\sigma = 3.5 \approx 5 + 5/3$ . Compared to the models with fixed sigmoid activation function the SoG activation function achieves a relative improvement of over 16% of the mean validation cross entropy and an improvement of over 24% of the relative test classification error. The high standard deviation of the cross entropy and the fact that the resulting activation functions greatly differ in their parameters and shape, indicate that we have found different local minima of the loss function. This is due to the stochasticity in the initialization of the network parameters and the randomly selected batches for stochastic gradient descent. Most commonly, the resulting activation functions took a Gaussian or sigmoid-like shape on the supported interval. The results for the different activation functions are stated in Table 5.1. Based on this we can evaluate our separate training procedure with Bayesian functional optimization as described in chapter 4.

Training Method	Mean CE Val. Data	Std. CE Val. Data	Best CE Val. Data	Test CE Best Model	Test Error Best Model	Activation Best Model
PBO (3 BFs) (7 runs)	0.0764	0.0107	0.0583	0.0548	1.92%	
BFO (3 BFs) (10 runs)	0.0555	0.0025	0.0524	0.0532	1.78%	
BFO (5 BFs) (3 runs)	0.0573	0.0011	0.0560	0.0639	1.95%	
BFO (10 BFs) (3 runs)	0.0611	0.0008	0.0600	0.0664	2.17%	

Table 5.2: MNIST results for the PBO and BFO training procedure.

We selected the objective functional for Bayesian functional optimization as the cross entropy of the validation data set obtained by training the MLP model with the input activation function. We described this in more detail in section 4.1. To compare the performance of Bayesian functional optimization (BFO) with iGP-UCB we additionally evaluate the parametric formulation with standard parametric Bayesian optimization (PBO). PBO works in the joint parameter space of centers and corresponding weights of the parameterized activation function. It uses a parameterized version of the squared exponential inner product kernel that is used by BFO. For the evaluation we consider 100 iterations of BFO and PBO with the UCB acquisition function and a fixed schedule for  $\beta_t = 1$ . Beforehand we also tried to set the schedule according to GP-UCB but received empirically better results for the constant  $\beta_t$ . For the optimization of the acquisition function PBO uses LBFGS while BFO uses functional gradient descent with a backtracking line search. To speed up the network training we chose a smaller patience  $p = 500$ . At the end we used the activation functions of the best model w.r.t. the loss on the validation data to train a final model with patience  $p = 2000$ . The final model was used to obtain the final test cross entropy and relative test classification error. The evaluation considers 10 runs of BFO and 7 runs of PBO for activation functions consisting of 3 basis functions resulting in 6 overall parameters for PBO. We additionally evaluated 3 runs of BFO for activation functions consisting of 5 and 10 basis functions. However, we not evaluated PBO for activation functions with more than 3 basis functions as the performance drastically





(a) Mean and standard deviation for BO and BFO with 3 basis functions (b) Means for BO with 3 basis functions and BFO with 3, 5, and 10 basis functions

Figure 5.2: Performance of PBO and BFO measured over 100 iterations

decreases. Again, we chose the supported interval of the activation functions as  $[-5, 5]$ . For BFO and PBO with 3 basis functions we chose the bandwidth  $\sigma = 3.5 \approx 5 + 5/3$ . However, to compare the performance of BFO for different numbers of basis functions and see if they converge to the same optima we also used the same bandwidth  $\sigma = 3.5$  for the evaluation of BFO with 5 and 10 basis functions. Compared to the joint training procedure the best model of BFO for 3 basis functions achieves an improvement of the mean cross entropy on the validation error of over 40%. Compared to PBO we observe an improvement of over 27%. All results can be found in Table 5.2. We also observe that PBO has difficulties to sufficiently explore the space and to eventually converge to some good minima. The cross entropy and form of the resulting activation functions varies greatly across the different runs as indicated by the high standard deviation. However, there is one outlier in the PBO runs with a very low cross entropy compared to the mean. On the other hand we observe that all versions of BFO converge much faster to better solution. The low standard deviation of the cross entropy and the similar shapes (neglecting symmetries) of the resulting activation functions indicate that we might have found a near globally optimal activation function for the given problem and basis function bandwidth  $\sigma = 3.5$ . Moreover, the outlier activation function found by PBO has a cross entropy value and shape similar to the activation functions found by BFO. 5.2a depicts the mean cross entropy and the corresponding standard deviation of BFO and PBO for 3 basis functions over the course of the algorithm. As mentioned earlier we not further evaluated PBO for more than 3 basis functions as the performance decreases heavily and we were not able to receive usable results. BFO however still performs good for 5 and even 10 basis functions which correspond to 20 parameters in a parametric setting. 5.2b depicts the mean for all evaluated versions. Plots of all activation functions computed by BFO can be found on the following pages in Figure 5.3, Figure 5.4, and Figure 5.5.

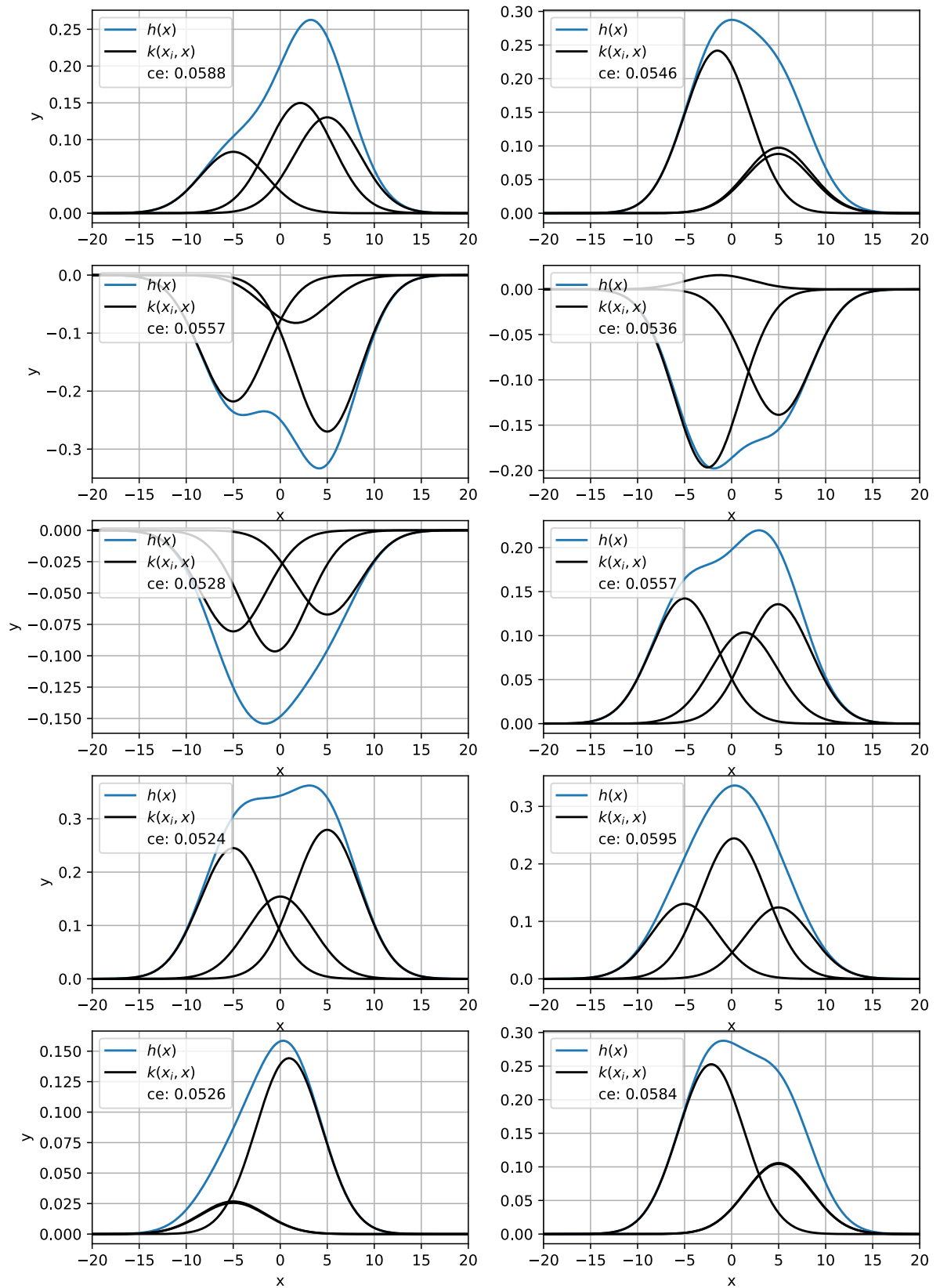


Figure 5.3: Activation functions with 3 basis functions found by BFO

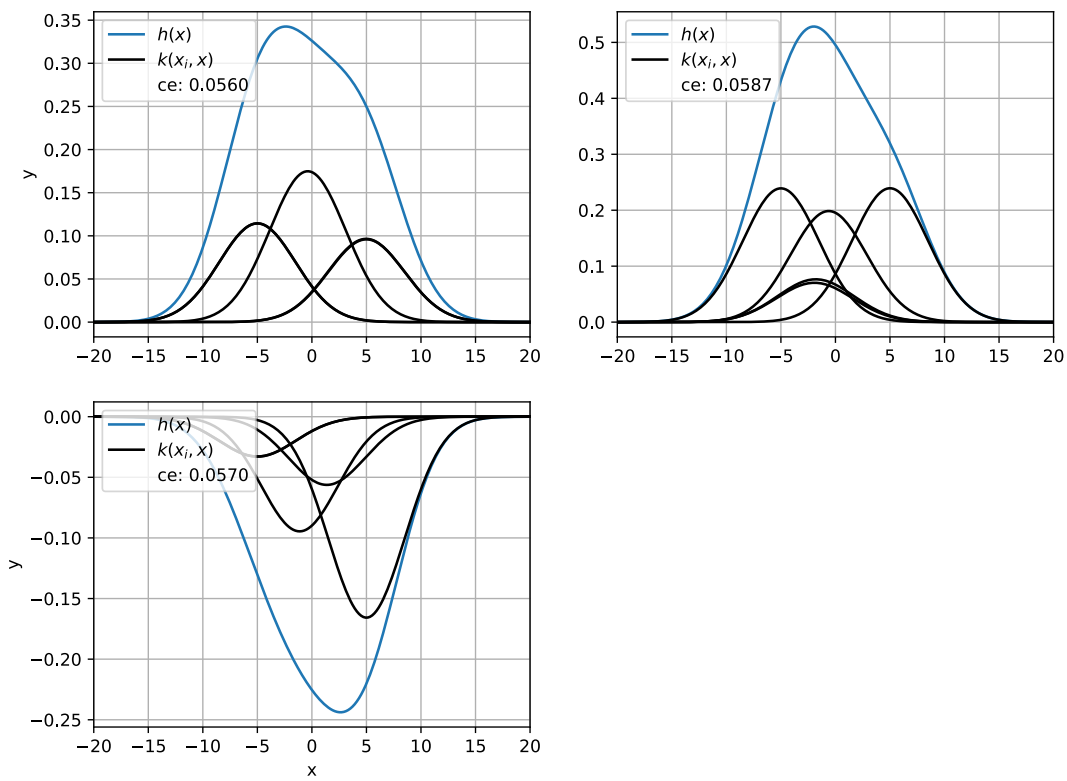


Figure 5.4: Activation functions with 5 basis functions found by BFO

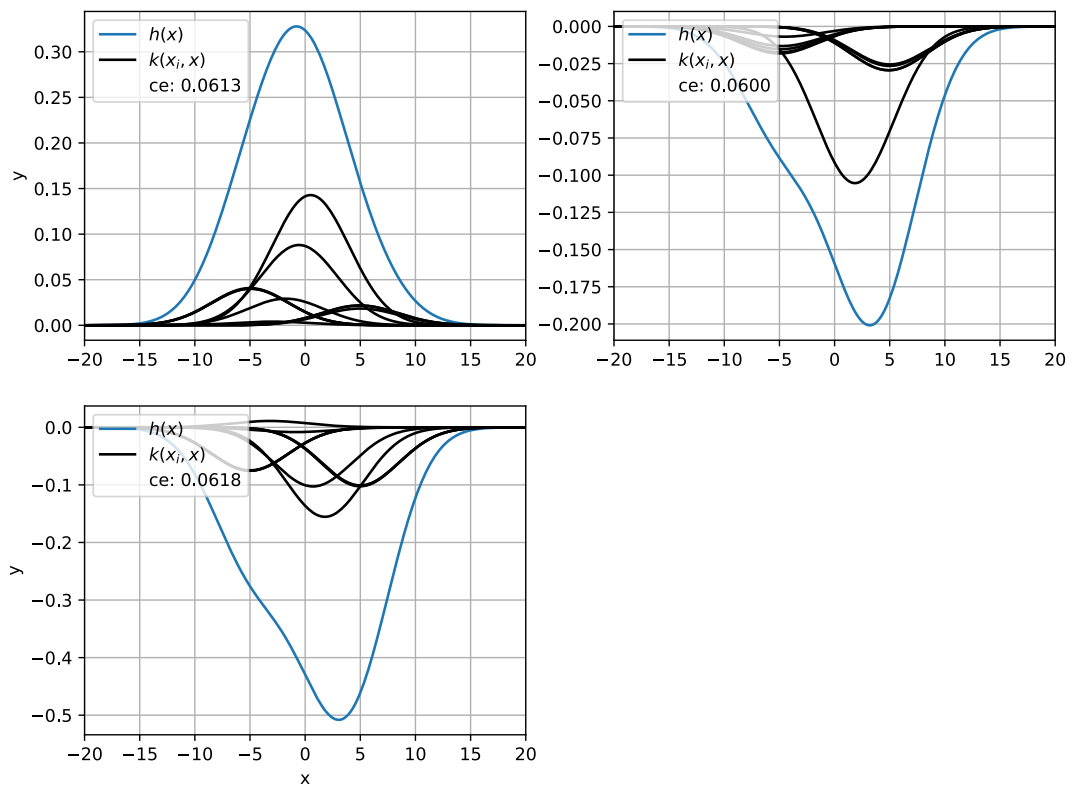


Figure 5.5: Activation functions with 10 basis functions found by BFO

# 6 Discussion

## 6.1 Results

Our evaluation showed that using a shared adaptive SoG activation function for our multilayer perceptron is clearly beneficial compared to commonly used fixed activation functions. However, the training method has a large impact on the quality of the resulting models. As expected, the joint training of the parameters of the activation function and the parameters of the network resulted in models with a lower cross entropy and classification error. This can be explained by the additional degrees of freedom in the nonlinear activation function which can be better adapted to the inputs. Due to the high update variance, stochastic batch gradient descent is able to overcome small local minima, but eventually still suffers from the problem of getting caught in local minima. This is also indicated by the high standard deviation of the cross entropy of the resulting models and the very different shapes of the corresponding activation functions. Despite of performing clearly better than the fixed sigmoid activation function, the joint training method had difficulties to fully explore the space of possible SoG activation functions as it suffers from the local minima problem.

On the other hand, the separate training variants that are using Bayesian functional optimization and standard parametric Bayesian optimization do not suffer from the local minima problem directly. Therefore, they are better able to explore the space of possible activation functions. However, the objective functional or function involves the training of a neural network model which still uses gradient descent methods and therefore still suffers from the local minima problem. BFO and PBO try to account for the local minima problem and the stochasticity which is involved in the training of the network by modeling it as additional Gaussian noise.

Our evaluation also compared the performance of BFO and PBO. Bayesian functional optimization far outperformed standard parametric Bayesian optimization and works well even for higher dimensional problems. As they are both using the same GP kernel, it must be that the optimization of the parametric acquisition function did not find good optima. This is due to the fact that PBO is working in the joint parameter space of the two potentially very different metric spaces of coefficients (weights) and centers. In contrast BFO only considers the coefficients of a fixed set of basis functions in every iteration and computes the gradient and step size w.r.t. the underlying norm of the reproducing kernel Hilbert space. The selection of these basis functions is handled separately and

consists of the most significant basis function from previous iterations and randomly sampled ones. Despite of working in a potentially much higher dimensional search space, this results in the selection of better evaluation points. The similar shape of the found activation functions and low variance of the cross entropy of their corresponding models indicates that BFO might have found a near globally optimal activation function for the given problem and kernel bandwidth. The results also showed that 3 basis functions with bandwidth  $\sigma = 3.5$  are seemingly enough to represent a good activation function for the MNIST data set and our MLP on the chosen interval  $[-5, 5]$ . When observing the resulting activation functions from BFO with 5 and 10 basis functions that used the same bandwidth, we see that their representation basically collapsed to 3 or at most 4 significant basis functions. In our evaluation, the simple heuristic for selecting the bandwidth worked well for the given interval and 3 basis functions. However, for higher numbers of basis functions with heuristically selected bandwidths, e.g. 5 basis functions with bandwidth  $\sigma = 2$ , we experienced much worse outcomes. Eventually, fewer basis functions seem to work better and the resulting bandwidth should be taken as a first estimate only.

In the end, our results give two core insights. First, the selection of a problem specific activation function shared by all hidden layer neurons, can have a significant impact on the resulting models and their test error. For the presented setting our training method using BFO was able to find such problem specific activation functions that might be nearly globally optimal. Second, for the given problem, Bayesian functional optimization far outperformed standard parametric Bayesian optimization and was able to perform well even for high dimensional problems.

## 6.2 Possible Extensions and Limitations

One can think of several extensions of the presented methods. In our work we chose the loss functional to be the cross entropy of the validation data set. However, one may use the loss functional to encode more wanted characteristics of the resulting activation functions. For example, one might want to additionally penalize the time that was spend on training or evaluation of the network to encourage activation functions that provide a good computational performance. But while one can design arbitrarily complex loss functionals with many penalizers, this might come with the drawback of needing more samples to converge to a sufficiently good minima. While we only investigated the use the Gaussian RBF kernel, one might also use different kernels that are better suited for certain problems. For example, it might be interesting to use periodic kernels as they do not suffer from vanishing gradients.

Another possible extension considers the construction of the search space to give more control and guarantees for how well it covers the problem domain. This could be done by introducing a schedule for the random sampling of basis function centers which are

used to initialize the gradient descent optimizer. Therefore, in the early stages of the algorithm, sampled centers that are close to a center of a basis function which is already contained in the search space are discarded and resampled. As the algorithm continues, the schedule gradually decreases the minimal allowed distance between the centers. This leads to a better coverage of the problem domain especially in the early stages of the algorithm. Moreover, such schedules might provide bounds for the sample density of centers in the problem domain and state how good the search covers the current space of possible solutions.

We are also aware that MLPs are not the state of the art network type for many problems. However, we are confident that due to the very generic framework of Bayesian functional optimization, our method can also be applied to different architectures like convolutional neural networks (CNNs) or recurrent neural networks (RNNs). For CNNs, one could simply start by using Bayesian functional optimization to optimize the activation functions of the fully connected layers only.

In chapter 2 we presented related work that uses joint steepest-descent-based training methods to adapt each neurons activation function individually, but is likely to suffer from the local minima problem. While our method does not directly suffer from the local minima problem, it is also less flexible as each neuron shares the same activation function. In general, there is no reason to believe that a good activation function is the same for all hidden layer neurons. However, one may use the found activation functions as an initialization for joint training methods that work on a per neuron level. The idea is, that starting from a better initial activation function that is more intrinsic to the problem might result in better per neuron activation functions and lower error models. Of course we still have the local minima problem, but we expect that per neuron activation functions that vary around a good shared activation function to be better than activation functions that vary around some commonly used initialization. Further, instead of training the model that is used in Bayesian functional optimization for a fixed activation function, one could also use a joint per neuron training procedure and directly optimize to find the best initialization function. However, these are just hypothesis and need further research and evaluation.

Probably the biggest limitation of our method is that it has to train the network several times before coming up with a good activation function and model. Therefore, for complex networks that are time consuming to train, the number of samples that are needed for a sufficiently good result might be too high. A different approach is to use more sophisticated training methods to overcome the local minima problem in the joint training setting. For example, Lo et al. [LGP12] and Lo et al. [LGP13] (NRAE, NRAE-MSE) presented a method that gradually convexifies the error surface of the mean squared error loss for MLP training. Thereby, it creates shortcuts that can be used by gradient-descent-based methods to overcome local minima. While the first introduced Normalized Risk-Averting Error (NRAE) training method had an overall unsatisfying success rate, the later proposed NRAE-MSE method reached a success rate of 100% in their numerical

experiments with a fixed hyperbolic tangent activation function. However, this method was designed for optimizing the weights of a multilayer perceptron for a fixed choice of the activation function only. It is therefore unclear how well it can be extended to jointly train the weights and activation function parameters, as the joint problem might result in completely different error surfaces.

## 6.3 Conclusion

In this work we presented a training method for shared activation functions for multilayer perceptrons. Therefore, we formulated the problem of finding an optimal shared activation function as a functional optimization problem. We then used Bayesian functional optimization with iGP-UCB to search for activation functions that we modeled as elements of a reproducing kernel Hilbert space. In contrast to training methods that jointly train the activation functions parameters together with the network parameters, our method does not suffer from the local minima problem. Our evaluation showed that Bayesian functional optimization far outperforms the parametric approach with standard Bayesian optimization and works well even for higher dimensional problems. Moreover, the resulting activation functions have a significant lower test classification error compared to their jointly trained variants and the commonly used fixed activation functions. The similar shape of the found activation functions and low variance of the cross entropy of their corresponding models are indications that we might have found near globally optimal activation functions. Compared our baseline models with fixed sigmoid activation function and jointly trained SoG activation function, we were able to reduce the relative classification error on the test data by over 39% and over 20% respectively.



# Bibliography

- [AHSB14] F. Agostinelli, M. D. Hoffman, P. J. Sadowski, and P. Baldi. “Learning Activation Functions to Improve Deep Neural Networks.” In: *CoRR* abs/1412.6830 (2014). URL: <http://arxiv.org/abs/1412.6830> (cit. on pp. 9, 11).
- [BCD10] E. Brochu, V. M. Cora, and N. De Freitas. “A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.” In: *arXiv preprint arXiv:1012.2599* (2010) (cit. on p. 20).
- [Bot98] L. Bottou. “On-line Learning in Neural Networks.” In: (1998). Ed. by D. Saad, pp. 9–42. URL: <http://dl.acm.org/citation.cfm?id=304710.304720> (cit. on p. 25).
- [CC96] C.-T. Chen and W.-D. Chang. “A Feedforward Neural Network with Function Shape Autotuning.” In: *Neural networks* 9.4 (1996), pp. 627–641. ISSN: 0893-6080. DOI: 10.1016/0893-6080(96)00006-8. URL: [http://dx.doi.org/10.1016/0893-6080\(96\)00006-8](http://dx.doi.org/10.1016/0893-6080(96)00006-8) (cit. on p. 11).
- [DKC13] J. Djolonga, A. Krause, and V. Cevher. “High-Dimensional Gaussian Process Bandits.” In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Curran Associates, Inc., 2013, pp. 1025–1033. URL: <http://papers.nips.cc/paper/5152-high-dimensional-gaussian-process-bandits.pdf> (cit. on pp. 9, 12).
- [ELw17] C. Eisenach, H. Liu, and Z. W. wang. “Nonparametrically Learning Activation Functions in Deep Neural Nets.” 2017 (cit. on pp. 9, 11).
- [GPU99] S. Guarnieri, F. Piazza, and A. Uncini. “Multilayer feedforward networks with adaptive spline activation function.” In: *IEEE Transactions on Neural Networks* 10.3 (1999), pp. 672–683. ISSN: 1045-9227. DOI: 10.1109/72.761726 (cit. on p. 11).
- [Gre13] A. Gretton. “Introduction to RKHS, and some simple kernel algorithms.” In: *Adv. Top. Mach. Learn. Lecture Conducted from University College London* (2013) (cit. on p. 16).

- [Hor91] K. Hornik. “Approximation Capabilities of Multilayer Feedforward Networks.” In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T. URL: [http://dx.doi.org/10.1016/0893-6080\(91\)90009-T](http://dx.doi.org/10.1016/0893-6080(91)90009-T) (cit. on p. 24).
- [KB14] D. Kingma and J. Ba. “Adam: A method for stochastic optimization.” In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 38).
- [KW71] G. Kimeldorf and G. Wahba. “Some results on Tchebycheffian spline functions.” In: *Journal of mathematical analysis and applications* 33.1 (1971), pp. 82–95 (cit. on p. 17).
- [LGP12] J. Lo, Y. Gui, and Y. Peng. “Overcoming the local-minimum problem in training multilayer perceptrons with the NRAE training method.” In: *Advances in Neural Networks–ISNN 2012* (2012), pp. 440–447 (cit. on p. 47).
- [LGP13] J. T.-H. Lo, Y. Gui, and Y. Peng. “Overcoming the local-minimum problem in training multilayer perceptrons with the NRAE-MSE training method.” In: *International Symposium on Neural Networks*. Springer. 2013, pp. 83–90 (cit. on p. 47).
- [Ngo16] V. Ngo. “Bayesian Optimization in Reproducing Kernel Hilbert space and Application for Direct Policy Search.” 2016 (cit. on pp. 9, 10, 12, 29, 31).
- [PS91] J. Park and I. W. Sandberg. “Universal Approximation Using Radial-Basis-Function Networks.” In: *Neural Computation* 3.2 (1991), pp. 246–257. ISSN: 0899-7667. DOI: 10.1162/neco.1991.3.2.246 (cit. on p. 30).
- [PUZ92] F. Piazza, A. Uncini, and M. Zenobi. “Artificial neural networks with adaptive polynomial activation function.” In: (1992) (cit. on p. 11).
- [Ras06] C. E. Rasmussen. “Gaussian processes for machine learning.” In: (2006) (cit. on p. 20).
- [SHS01] B. Schölkopf, R. Herbrich, and A. J. Smola. “A Generalized Representer Theorem.” In: *Computational Learning Theory: 14th Annual Conference on Computational Learning Theory, COLT 2001 and 5th European Conference on Computational Learning Theory, EuroCOLT 2001 Amsterdam, The Netherlands, July 16–19, 2001 Proceedings*. Ed. by D. Helmbold and B. Williamson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 416–426. ISBN: 978-3-540-44581-4. DOI: 10.1007/3-540-44581-1\_27. URL: [http://dx.doi.org/10.1007/3-540-44581-1\\_27](http://dx.doi.org/10.1007/3-540-44581-1_27) (cit. on p. 17).
- [SKKS09] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger. “Gaussian process optimization in the bandit setting: No regret and experimental design.” In: *arXiv preprint arXiv:0912.3995* (2009) (cit. on p. 22).

- [SLA12] J. Snoek, H. Larochelle, and R. P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms.” In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 2951–2959. URL: <http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf> (cit. on p. 12).
- [SSCU16] S. Scardapane, M. Scarpiniti, D. Comminiello, and A. Uncini. “Learning activation functions from data using cubic spline interpolation.” In: *arXiv preprint arXiv:1605.05509* (2016) (cit. on p. 11).
- [TGK14] H. Tyagi, B. Gärtner, and A. Krause. “Efficient Sampling for Learning Sparse Additive Models in High Dimensions.” In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 514–522. URL: <http://papers.nips.cc/paper/5466-efficient-sampling-for-learning-sparse-additive-models-in-high-dimensions.pdf> (cit. on p. 12).
- [TKGK16] H. Tyagi, A. Kyrillidis, B. Gärtner, and A. Krause. “Learning sparse additive models with interactions in high dimensions.” In: *proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2016 (cit. on p. 12).
- [TM14] A. J. Turner and J. F. Miller. “NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks.” In: *Evolutionary Intelligence* 7.3 (2014), pp. 135–154. ISSN: 1864-5917. DOI: 10.1007/s12065-014-0115-5. URL: <http://dx.doi.org/10.1007/s12065-014-0115-5> (cit. on pp. 9, 12).
- [Tou16] M. Toussaint. “Introduction to Machine Learning.” University Lecture. 2016 (cit. on p. 25).
- [VB02] P. Vincent and Y. Bengio. “Kernel Matching Pursuit.” In: *Machine Learning* 48.1 (2002), pp. 165–187. ISSN: 1573-0565. DOI: 10.1023/A:1013955821559. URL: <http://dx.doi.org/10.1023/A:1013955821559> (cit. on p. 35).
- [VPU98] L. Vecchi, F. Piazza, and A. Uncini. “Learning and Approximation Capabilities of Adaptive Spline Activation Function Neural Networks.” In: *Neural Networks* 11.2 (1998), pp. 259–270. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/S0893-6080\(97\)00118-4](https://doi.org/10.1016/S0893-6080(97)00118-4). URL: <http://www.sciencedirect.com/science/article/pii/S0893608097001184> (cit. on p. 11).
- [WHZ+16] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. De Freitas. “Bayesian Optimization in a Billion Dimensions via Random Embeddings.” In: *J. Artif. Int. Res.* 55.1 (2016), pp. 361–387. ISSN: 1076-9757. URL: <http://dl.acm.org/citation.cfm?id=3013558.3013569> (cit. on p. 12).

- [WZH+13] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, N. Freitas, et al. “Bayesian optimization in high dimensions via random embeddings.” In: AAAI Press/International Joint Conferences on Artificial Intelligence. 2013 (cit. on pp. 9, 12).
- [Yao99] X. Yao. “Evolving artificial neural networks.” In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447. ISSN: 0018-9219. DOI: 10.1109/5.784219 (cit. on p. 12).

## **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature