2 Introduction to Function Approximation and Regression

The first chapter of this thesis introduces Function Approximation (FA), also called *regression*, which is the basic feature required to learn sensorimotor mappings. A multitude of algorithm classes are introduced, including simple model fitting, interpolation, and advanced concepts such as Gaussian Processes and Artificial Neural Networks. The last section of this chapter discusses the applicability, but also questions the plausibility of such algorithms in the light of brain functionality.

2.1 Problem Statement

Continuous functions can be approximated with different methods depending on the type of function and the requirements to the resulting model. A function

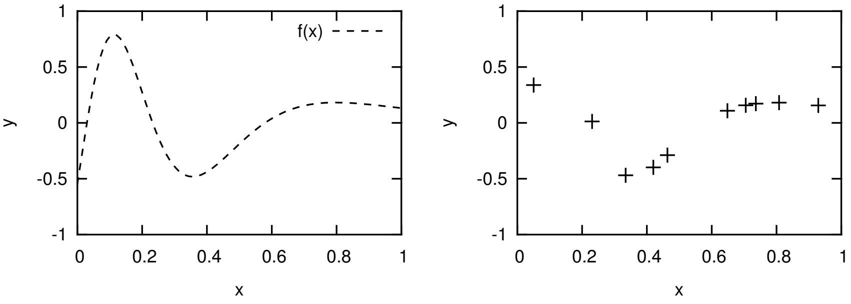
$$\begin{aligned} f : X &\rightarrow Y \\ f(x) &= y \end{aligned}$$

maps from $x \in X$ to $y \in Y$, where X is called the *input space* and Y is the *output space* of the function[1]. Generally, the data is available as samples (x, y), where the distribution of the inputs x and the function f are both unknown as illustrated in Figure 2.1. Eventually, the input space is multi-dimensional $\boldsymbol{X} \subseteq \mathbb{R}^n$. In this case, n-dimensional column vectors $\boldsymbol{x} \in \boldsymbol{X}$ are written as bold symbols.

The task is to find a model $h(\boldsymbol{x})$ that explains the underlying data, that is, $h(\boldsymbol{x}) \approx y$ for all samples $(\boldsymbol{x}, y)$. Ideally the model equals the underlying function: $h(\boldsymbol{x}) = f(x)$. However, the function f can have some unknown properties at a subspace where no samples are available. Here, Occam's razor can be applied: Given two models that both accurately explain the data, the simpler model should be preferred. For example, data from a quadratic function should not be explained with a fourth order polynomial.

An alternative term for FA is regression, which does essentially the same thing (create some kind of model from given data), but focuses more on statistical

[1]In mathematics literature X may be called domain and Y the co-domain.



(a) the true but unknown function **(b)** random samples

Figure 2.1: Illustration of a function approximation task. (a) The underlying function is usually unknown. (b) It can be challenging to accurately approximate a function surface from random samples. The data and its underlying function are specified in Appendix A.

properties such as expectation and variance. Usually regression approaches explicitly model noise or the variance of the prediction error in addition to the function surface. Input space X becomes the independent variable, where output space Y becomes the dependent variable. Throughout this work, the term *function approximation* is preferred, as noise poses an interesting challenge but need not be modeled explicitly and statistics are of lesser interest.

2.2 Measuring Quality

The goal of FA is to minimize the model (prediction, forecast) error. However, there are many ways to *measure* error that have only subtle differences. Given k samples (x, y) the quality of a model h can be assessed as follows. An intuitive measure is the Mean Absolute Error (MAE) computed as

$$\frac{1}{k}\sum_{j=1}^{k} |y_j - h(x_j)| \ , \tag{2.1}$$

which measures the average deviation from the true value.

The MAE does not account for the variance of the model error or, put differently, does not penalize outliers. Suppose model A accurately models 95% of the data but completely disagrees on the remaining five percent, while model B shows small errors over the full range of the data. They may well have the same MAE. When every data point has the same importance, model B may be preferable as it's errors have a lower variability. The amount of variability is taken into account by the following two measures.

Using the sum of *squared* errors instead of absolute errors yields the Mean Squared Error (MSE)

$$\frac{1}{k}\sum_{j=1}^{k}\left(y_j - h(x_j)\right)^2 , \tag{2.2}$$

where error variance also affects the resulting value. However, the MSE is not as easily interpreted as the MAE, e.g. because units are squared.

Taking the square root yields the Root Mean Squared Error (RMSE)

$$\sqrt{\frac{1}{k}\sum_{j=1}^{k}\left(y_j - h(x_j)\right)^2} , \tag{2.3}$$

which accounts for model deviation, error variance, but also more clearly relates to individual errors. When all errors have equal magnitude, then MAE and RMSE are equal as well; otherwise the RMSE is larger due to variability in the errors. When *accidental* outliers are known to be present, the RMSE may give a false impression of the model quality. Thus, the choice of quality measure depends on the actual application. Other quality measures exist, but those are usually tailored for a particular application. The MAE, MSE, and RMSE are the most common measures for FA.

2.3 Function Fitting or Parametric Regression

In the simplest form of FA the type of function is known and only a finite set of parameters has to be estimated. Put differently, the function is *fitted* to the data, e.g. a linear function or a polynomial. The extra assumption about the type of function not only improves, but also simplifies the approximation process. However, that assumption is a strong one to make and often the true underlying function is unknown. Thus, the chosen function type is a best guess and making wrong assumptions may result in poor quality models.

2.3.1 Linear Models with Ordinary Least Squares

Linear regression approximates the data with a linear function of two parameters $h(x) = \alpha + \beta x$. Intercept α and gradient β can be easily computed using the method of least squares [57, 29], where the squared error (thus, the RMSE as well) is minimized. Without particular assumptions on the distribution of noise (if there is any), the so called Ordinary Least Squares (OLS) method [53] can be applied to k data points:

$$\beta = \frac{\sum_{j=1}^{k}(x_j - \overline{x})(y_j - \overline{y})}{\sum_{j=1}^{k} x_j^2}\,, \quad \alpha = \overline{y} - \beta\,\overline{x} \tag{2.4}$$

where $\overline{x} = 1/k \sum_j x_j$ is the mean of the inputs and analogously $\overline{y}$ the mean of the function values. The above equation refers to one-dimensional inputs x.

The generalization to n dimensions is best described with a matrix formulation. Therefore, let $\boldsymbol{X}$ the $k \times n$ matrix of all inputs and $\boldsymbol{y}$ the vector of function values such that each row represents a sample $(\boldsymbol{x}_j, y_j)$. A linear mapping of all samples can be written as

$$\boldsymbol{X}\boldsymbol{\beta} = \boldsymbol{y} \tag{2.5}$$

where the linear weights $\boldsymbol{\beta}$ are unknown. If the underlying data is truly linear, then there exists a solution to the above stated system of equations by inversion of the matrix $\boldsymbol{X}$ written as

$$\boldsymbol{\beta} = \boldsymbol{X}^{-1}\boldsymbol{y}\,. \tag{2.6}$$

However, if the data is non-linear or noisy an approximation is required. The OLS estimator is calculated as

$$\boldsymbol{\beta} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}\,, \tag{2.7}$$

where $\cdot^T$ is the transpose operator. The term $(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ actually computes the so called Pseudoinverse matrix[2] which is the closest solution to a matrix inversion, if the regular inverse X^{-1} does not exist. The Pseudoinverse is equivalent to the regular inverse, if it exists.

The above formula assumes that the function values y have zero mean, that is, no intercept α is required. If this is not the case, the function values can be zero-centered by subtraction of the mean $y_i - \overline{y}$. Alternatively, a simple trick allows to include a non-zero intercept: The matrix $\boldsymbol{X}$ is extended with another column of ones, that is, $\boldsymbol{X}_{n+1} = 1$. The corresponding entry β_{n+1} then represents the intercept of the least squares hyperplane such that

$$h(x_1, \ldots, x_n) = \underbrace{\beta_1 x_1 + \ldots + \beta_n x_n}_{\text{linear model}} + \underbrace{\beta_{n+1}}_{\text{intercept}} \approx y\,. \tag{2.8}$$

[2]Details about computation of the Pseudoinverse, also known as Moore-Penrose matrix can be found in Section 7.3

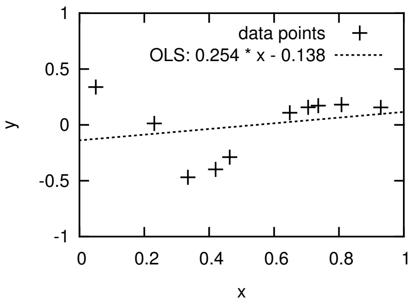


Figure 2.2: The linear OLS solution does not fit well to the underlying non-linear data. Furthermore, it is difficult to guess the true, underlying function type.

This simple OLS computation produces a linear model with minimal RMSE, even if the underlying data is *not* linear as exemplified in Figure 2.2. However, it requires all data points to be known in advance. Put differently, when data arrives iteratively the model must be recomputed at every step and complexity increases steadily with the number k of data points. Fortunately, there are iterative least squares versions for linear approximation.

2.3.2 Online Approximation with Recursive Least Squares

The iterative version of OLS is the so called Recursive Least Squares (RLS) method [3, 41], where the underlying model is updated for every sample individually, but without explicitly storing all data points. This way estimates are already available for few data points and the model continuously improves with incorporation of more samples.

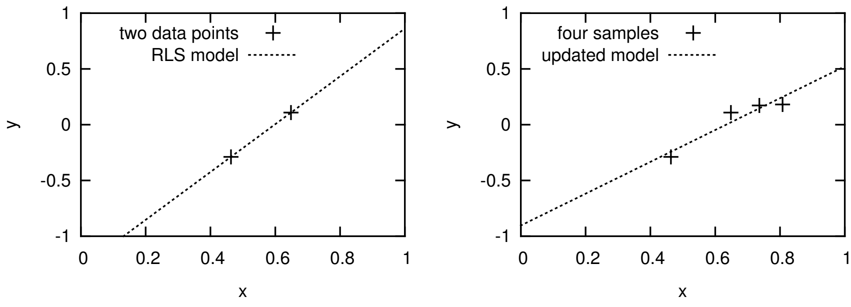
Again, a linear model is considered for now, though OLS and RLS can be applied to other models as well. Rewriting Equation (2.8) into vector notation yields an n-dimensional linear model

$$h(\boldsymbol{x}) = \boldsymbol{\beta}^T \boldsymbol{x}\,, \tag{2.9}$$

where $x_{n+1} = 1$ and β_{n+1} is the intercept. The approximation goal is to minimize the sum of squared errors over all samples $(\boldsymbol{x}, y)_{1 \leq j \leq k}$, that is, minimization of

$$\sum_{j=1}^{k} \lambda^{k-j}\, \varepsilon_j^2, \tag{2.10}$$

where $\varepsilon = y - h(\boldsymbol{x})$ is the model error for a single sample and $0 \ll \lambda \leq 1$ is a forgetting factor that assigns exponentially less weight to older samples. The parameter λ can be ignored for now, that is, set to one.



(a) two data points **(b)** two more points

Figure 2.3: The RLS algorithm iteratively includes the samples. (a) Two data points can be perfectly fitted with a straight line. (b) Adding two more points reveals that the underlying function is non-linear. The linear RLS model incorporates the data piece by piece.

Without prior knowledge, gradient and intercept are initialized to zero. When a new sample arrives, the gradient must be corrected to account for the new data. This can be seen as a *recursive* process, based on the model at iteration $k-1$. Generally, model correction can be written as $\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta}_{\text{old}} + \varepsilon \boldsymbol{g}$, where $\varepsilon = y - h(\boldsymbol{x})$ is the a priori model error for the current sample and the so called *gain* $\boldsymbol{g}$ specifies how the model is adapted. The key is to compute a suitable gain such that the model converges to the optimal gradient as exemplified in Figure 2.3. The next paragraph roughly sketches the idea in RLS before details of the math are given.

In order to determine a suitable gain factor $\boldsymbol{g}$, RLS stores an estimate of the *inverse covariance matrix* $\boldsymbol{P}$ that essentially defines the relevance of inputs for model updates. Initially, any sample should have a strong impact on the model correction and therefore this matrix is initialized as $\sigma \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix and a large σ defines the update rate during the first iterations. When many samples have been seen in some direction, the matrix $\boldsymbol{P}$ is shrunk in that direction. Thus, overrepresented regions produces small model corrections (low information gain) while rarely sampled regions have a stronger impact (high gain) on the model update. Since the gain is multiplied with the error, the other factor is prediction accuracy. When the model is accurate anyway, no model correction is required. Large errors, on the other hand, result in strong updates.

Detailed derivation of the presented formulas can be found in the literature [3, 41]. A brief summary of the math can be given in three steps. Let $(\boldsymbol{x}, y)$ be the current sample. If an intercept is desired, the input is augmented by a constant entry $x_{n+1} = 1$. First, the gain factor $\boldsymbol{g}$ is computed as

$$\boldsymbol{g} = \frac{1}{\lambda + \boldsymbol{x}^T \boldsymbol{P} \boldsymbol{x}} \boldsymbol{P} \boldsymbol{x}\,, \tag{2.11}$$

where $\boldsymbol{x}^T \boldsymbol{P} \boldsymbol{x}$ defines the overall relevance of the current input $\boldsymbol{x}$ while $\boldsymbol{P}\boldsymbol{x}$ defines the relevance for individual weights β_i, that is, the relevance for each dimension i of the linear model. The computed gain vector $\boldsymbol{g}$ is used in conjunction with the a priori error $\varepsilon = y - h(\boldsymbol{x}) = y - \boldsymbol{\beta}^T \boldsymbol{x}$ to update the linear weights

$$\boldsymbol{\beta} = \boldsymbol{\beta}_{\text{old}} + \varepsilon\, \boldsymbol{g}\,. \tag{2.12}$$

Updating the inverse covariance matrix

$$\boldsymbol{P} = \frac{1}{\lambda} \left(\boldsymbol{P}_{\text{old}} - \boldsymbol{g} \boldsymbol{x}^T \boldsymbol{P}_{\text{old}} \right) \tag{2.13}$$

is the last step which incorporates knowledge about the distribution of inputs seen so far.

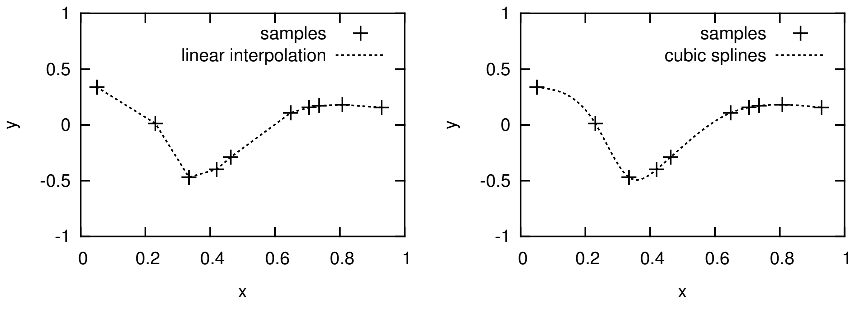
The RLS algorithm expects zero mean inputs, that is, $\sum x_j = 0$, as the covariance matrix is a zero centered geometric representation. Furthermore, the noise on function values – if any – is assumed to be zero mean. Finally, the function is assumed to be *static*, that is, the underlying function is not changing dynamically over time. If the function is *dynamic*, a forget rate $\lambda < 1$ helps to continuously adapt to a varying environment. When specific assumptions about the noise can be made or a forward model of the function dynamics are known, an advanced approximation method such as Kalman filtering [51, 3, 41], which is an extension of RLS, should be applied instead.

2.4 Non-Parametric Regression

Instead of fitting data to a predefined function, arbitrary models can be built from the data in various ways. While learning takes considerably longer, the function does not need to be specified in advance. Therefore *interpolation* approaches, in particular Gaussian Processes for regression are described. Artificial Neural Networks are another member of non-parametric regression methods and inherently provide a plausible way to describe brain functionality.

2.4.1 Interpolation and Extrapolation

A rather simple way to approximate a function surface from a *finite* set of samples is interpolation, where predictions are computed as a function of adjacent



(a) linear interpolation **(b)** cubic spline interpolation

Figure 2.4: Interpolation is an intuitive and simple way to produce a function surface within available samples. (a) Linear interpolation connects each sample with a straight line and produces a jagged surface. (b) Smooth interpolation via cubic splines yields a two-times differentiable surface.

data points. For example, if adjacent samples are connected with straight lines (see Figure 2.4a), the model is not only simple but also provides a good intuition of the underlying function. However, the resulting surface is not smooth.

When samples are instead connected with low order polynomials, the surface becomes smooth as illustrated in Figure 2.4b. Here the so called *cubic splines* [6], that is, third order polynomials, are applied. For $k+1$ samples, the $4k$ unknowns of the k cubic polynomials are specified by the following conditions: First, each polynomial h_i must pass through the two adjacent data points y_i and y_{i+1}, that is,

$$h_i(x_i) = y_i \tag{2.14}$$

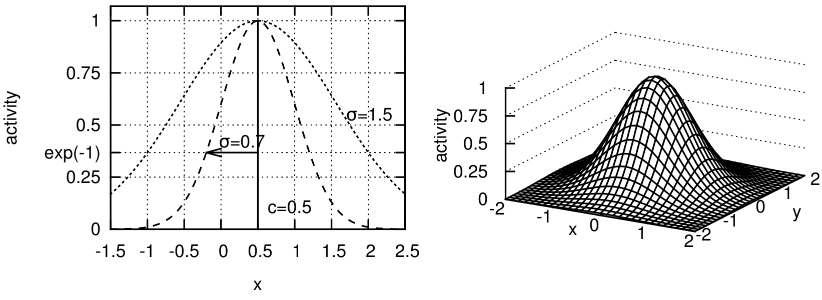
$$h_i(x_{i+1}) = y_{i+1}\,, \tag{2.15}$$

which yields $2k$ equations. Next, the first and second derivatives must match for interior points,

$$h_i'(x_{i+1}) = h_{i+1}'(x_{i+1}) \tag{2.16}$$

$$h_i''(x_{i+1}) = h_{i+1}''(x_{i+1})\,, \tag{2.17}$$

which yields another $2(k-1)$ equations, that is, a total of $4k-2$ equations. Two more conditions are required and typically it is required that the second derivatives are zero at the endpoints. Those equations can be solved for the four polynomial parameters resulting in a symmetric, tridiagonal system of equations that yields a unique solution for the cubic splines.



(a) 1D Gaussian kernel **(b)** 2D variant

Figure 2.5: Illustration of Gaussian kernels used as a distance metric. (a) The activity of two kernels with $\sigma = 0.7$ and $\sigma = 1.5$ exponentially decreases from the center c, while the maximum activity of one is excited at the center. The width σ defines the distance from center to the inflection point of the activity function, which occurs at a height of $\exp(-1)$. (b) A two-dimensional kernel with $\sigma = 1$ centered on the origin.

2.4.2 Gaussian Process Regression

Another very powerful approach is Gaussian Process Regression (GPR) [67], where not only adjacent data points contribute to the estimated surface, but instead all known samples contribute to the prediction. The rough idea is that samples close to a target should have a stronger impact (of course), but only in the light of all other samples the true trend can be seen.

Therefore, GPR uses a distance metric (also known as kernel function or covariance function) that specifies the relation between samples. Commonly, *Gaussian* kernels are used, that is, an exponential distance metric

$$\phi(\boldsymbol{x}, \boldsymbol{c}) = \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{c}\|^2}{\sigma^2}\right) \tag{2.18}$$

from input $\boldsymbol{x}$ to another point $\boldsymbol{c}$, where the quadratic distance is scaled by the width σ of the kernel and the exponential of the negative term yields a Gaussian bump as illustrated in Figure 2.5. Thus, samples far away from a point of interest $\boldsymbol{c}$ have a low influence, while closer samples strongly affect the outcome.

Given k samples $(\boldsymbol{x}_j, y_j)$, the prediction at query point $\boldsymbol{x}_*$ can be formulated as a linear combination of kernels as

$$h(\boldsymbol{x}_*) = \sum_{j=1}^{k} \alpha_j \phi(\boldsymbol{x}_j, \boldsymbol{x}_*)\,. \tag{2.19}$$

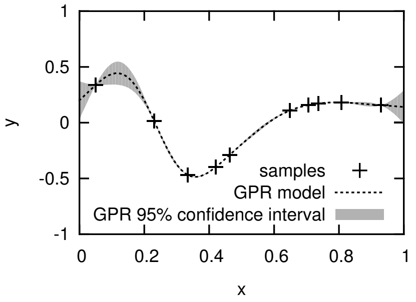


Figure 2.6: GPR provides a good model, even with few samples. Furthermore, the confidence interval clearly indicates points of uncertainty.

The weights α_j are computed in an OLS fashion as

$$\boldsymbol{\alpha} = (\Sigma + \gamma^2 I)^{-1} \boldsymbol{y} \,, \tag{2.20}$$

where the $k \times k$ covariance matrix has entries $\Sigma_{ij} = \phi(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and γ^2 is the expected independent and identically distributed noise.

The algebraic path sketched above fits well into this thesis, but is rarely found in the GPR literature. Instead, a probabilistic view is preferred: In addition to the predicted (mean) value a variance is computed. The prediction then becomes a normal distribution with mean

$$\boldsymbol{q}^T \left[\Sigma + \gamma^2 I\right]^{-1} \boldsymbol{y} \,, \tag{2.21}$$

where $q_j = \phi(\boldsymbol{x}_j, \boldsymbol{x}_*)$ is the covariance between the j-th sample and the query point. The variance of the predictive distribution is given as

$$\phi(\boldsymbol{x}_*, \boldsymbol{x}_*) - \boldsymbol{q}^T \left[\Sigma + \gamma^2 I\right]^{-1} \boldsymbol{q} \,. \tag{2.22}$$

The resulting GPR model for the data set used so far is visualized in Figure 2.6. GPR allows for sophisticated statistical inference – at the cost of an inversion of the $k \times k$ covariance matrix, which grows quadratically in the number of samples. A more detailed introduction into GPR goes beyond the scope of this thesis and the interested reader is referred to [67].

2.4.3 Artificial Neural Networks

Inspired by the capabilities of the human brain, interest in simulated neurons dates back to 1943 [61]. A single neuron represents a kind of function, as it receives inputs and produces some kind of output. Thus, a network of neurons, so called Artificial Neural Network (ANN), is clearly related to FA as well. Furthermore, biological plausibility comes for free with such models, when necessary simplifications to simulate neural activity in a computer are ignored.

The Cybenko theorem [23] states that any continuous function can be approximated with arbitrary precision on a (compact) subset of $\mathbb{R}^n$ with a particularly simple type of ANN. On first sight, this theorem might make ANN the first choice for the present thesis. However, arbitrary precision may come at the cost of arbitrary computational resources and the theorem is not constructive in that it does not provide the optimal number of neurons for a given task. Furthermore, the mere existence of an *optimal* network does not imply that a learning algorithm converges to that solution on a complex problem.

There is a vast number of different network types including, but not limited to, simple feed forward networks [61], recurrent neural networks [36], and spiking neural networks [32]. The most attractive type for FA in the present work, tough, are Radial Basis Function Networks (RBFNs) [40]: They fall into the Cybenko category of ANNs and arbitrary approximation precision is possible. However, in contrast to other networks, such as multilayer perceptrons, the *globally optimal* solution for a given data set can be computed in a simple fashion.

Radial Basis Function Networks

RBFNs [40] belong to the family of so called feed forward neural networks that do not have cyclic connections. They can be described as a three layer architecture composed of an input layer, a hidden layer, and an output layer. Inputs are propagated to the hidden layer, where a non-linear activation function determines the activity of the hidden neurons. Every hidden neuron has a simple model and votes according to its activity and model. The votes are aggregated and provide the estimated function value for the given input as depicted in Figure 2.7.

Formally, the prediction with m hidden neurons is a weighted sum

$$h(\boldsymbol{x}) = \frac{\sum_{r=1}^{m} \phi_r(\boldsymbol{x})\lambda_r(\boldsymbol{x})}{\sum_{r=1}^{m} \phi_r(\boldsymbol{x})}\,, \tag{2.23}$$

where ϕ_r represents the r'th radial basis function and λ_r are local models, be it a plain weight or a linear model. Here, the normalized version is considered, that is, the weighted vote from all models is divided by the sum of all weights.

Typically a Gaussian kernel (cf. Equation (2.18) and Figure 2.5) is used as the radial basis function

$$\phi_r(\boldsymbol{x}) = \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{c}_r\|^2}{\sigma^2}\right)\,, \tag{2.24}$$

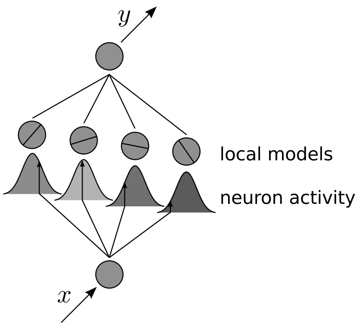


Figure 2.7: Each neuron of an RBFN receives the input x and excites a certain activity. The contribution of a hidden neuron to the output is defined by its local model which is, in the simplest case, just a single weight. The final output is a activity-weighted sum of the individual models.

where the width of all kernels in a RBFN is usually a fixed, global value σ, while the centers $\boldsymbol{c}_r$ are distributed according to the problem at hand. Non-linearity of the activation function is an important property here, as the *linear* combination of *linear* models collapses to a single linear model.

Learning, which takes place at the level of local models λ_r, often referred to as *weights*. The simplest RBFN approach uses a single weight $\lambda_r(\boldsymbol{x}) = \alpha_r$ independent of the actual input and Equation (2.23) becomes

$$h(\boldsymbol{x}) = \frac{\sum_{r=1}^{m} \phi_r(\boldsymbol{x})\, \alpha_r}{\sum_{r=1}^{m} \phi_r(\boldsymbol{x})}\,, \tag{2.25}$$

Those weights can then be optimized with respect to the squared error. In contrast to other ANNs, where the popular *back-propagation* algorithm computes suitable weights, the *globally optimal* weights for an RBFN can be found by solving a linear system of equations. Thus, this is closely related to OLS, where linear weights were computed for a linear model. Here, linear weights are computed as well – but the non-linear activation function allows for non-linearity in the final RBFN model.

Given a data set of k samples $(\boldsymbol{x}, y)$ with $\boldsymbol{x} \in \mathbb{R}^n$, the model for all samples can be written in matrix notation as

$$\boldsymbol{H}\boldsymbol{\alpha} = \begin{pmatrix} H_{11} & \dots & H_{1m} \\ \vdots & \ddots & \vdots \\ H_{k1} & \dots & H_{km} \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} = \boldsymbol{y}\,, \tag{2.26}$$

where the matrix entry at index j, q is the normalized activity of the q'th kernel on the j'th sample

$$H_{jq} = \frac{\phi_q(\boldsymbol{x}_j)}{\sum_{r=1}^{m} \phi_r(\boldsymbol{x}_j)} \tag{2.27}$$

The former Equation (2.26) is equivalent to an ideal, that is, zero-error model evaluated at all inputs. Thus, the optimal weights $\boldsymbol{\alpha}$ can be derived by rewriting the above equation into $\boldsymbol{\alpha} = \boldsymbol{H}^{\dagger}\boldsymbol{y}$, where $\boldsymbol{H}^{\dagger} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ denotes the Pseudoinverse matrix. This is the same strategy as used for the OLS solution in Equation (2.7) or for GPR in Equation (2.20), and therefore another example for using OLS with a non-linear model.

Importantly, computing the weights this way does not necessarily result in a zero-error model, but instead minimizes the MSE for the given RBFN. Placing one kernel on top of each sample is a simplified from of GPR. However, analogously to OLS and GPR this approach requires all samples in advance and is not suited for online approximation on infinite data streams.

Iterative Update Rules for RBFNs

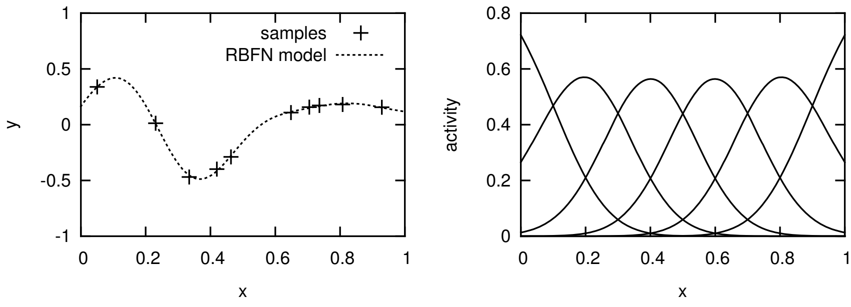
An alternative to the batch-training is a gradient descent that continuously adapts the model with a certain learning rate. Given sample $(\boldsymbol{x}, y)$, a simple update rule for one neuron is

$$\alpha = \alpha_{\text{old}} + \delta \underbrace{\phi(\boldsymbol{x})}_{\text{activity}} \underbrace{(y - h(\boldsymbol{x}))}_{\text{error}}, \tag{2.28}$$

where $0 < \delta < 1$ defines the learning rate. Training an RBFN with six kernels and a learning rate of $\delta = 0.5$ over 15000 iterations on the given data set (see Figure 2.8) yields satisfactory results.

While the OLS solution depicted above optimizes all weights globally, the gradient descent method in Equation (2.28) trains each neuron locally – not taking information from neighboring neurons into account. Put differently, here each neuron minimizes its own error while the OLS method minimizes the sum of (squared) errors from *all* neurons. The global optimization naturally produces a more accurate model, while gradient descent is applicable to infinite data streams.

Up to now, nothing has been said about the shape of the radial basis functions, that is centers $\boldsymbol{c}_r$ and width σ, and this is the crux of RBFN. On a finite data set one kernel can be set on top of every data point, which is a basic form of interpolation with GPR. Alternatively a reduced number of centers can be chosen randomly among inputs, distributed uniformly, or even completely random in the input space.



(a) RBFN model **(b)** kernel activity

Figure 2.8: A simple six-neuron RBFN is trained for 15000 iterations. (a) The function structure is well approximated. (b) Six Gaussian kernels are uniformly distributed with $\sigma = 0.2$. Normalized activity is shown here.

While this might give acceptable results, the intuition is that centers and corresponding widths should be tuned as well. In this case, the optimization becomes a two-step procedure: a) choosing centers and widths, b) training. The so called *meta* learning is repeated until performance is satisfactory. The following section introduces two algorithms that interweave the optimization of shape, that is, clustering and training of the local models.

2.5 Local Learning Algorithms

RBFNs fit quite well to the requirements for this thesis stated in Section 1.4. It is based on a population of neurons that cooperatively model a non-linear function with several local, simple sub-models. Two crucial factors affect the performance of RBFN *jointly*: the shape of the radial basis functions and the weights of the local models. The latter can be trained in a rather short time scale. However, to assess (or optimize) the shape of the neuron activation functions, the weights have to be trained as well – therefore optimization of the shape requires considerably more effort.

Algorithms that train several simple, parametric models in a certain locality *and* learn the locality – that is, a clustering of the input space – are henceforth called *local learning* algorithms. Where RBFNs adapts its weights based on the global prediction error (average from all neurons), a local learner updates it's parametric models on their individual errors. The most demanding requirement for this class of algorithms is *online* functionality, that is, function approximation on a continuous stream of samples. For finite data sets, closed form solutions such as GPR can provide high accuracy. Iterative solutions, on the other hand, can hardly reach the accuracy of a closed form solution and, additionally, require exhaustive training. In the following, two such algorithms are briefly introduced.

2.5.1 Locally Weighted Projection Regression

Similar to RBFN the so called Locally Weighted Projection Regression (LWPR) [75, 91] approximates a non-linear function surface by several locally linear models. LWPR also applies Gaussian kernels to cluster the input space. However, the shape of the kernels is optimized concurrently during training of the linear models. Since this is done iteratively, on a stream of samples, the approach becomes rather complex compared to RBFNs. The ideas are outlined briefly.

First, the local models applied in LWPR are linear, but neither trained by RLS nor by a simple gradient descent as done in Equation (2.28). Instead, the so called Partial Least Squares (PLS) method is applied, which does not regress the full n-dimensional input against the output, but instead projects the full input space onto the $l < n$ most relevant dimensions. This reduced input space is then used to estimate a linear relation to the output. While it allows to treat high-dimensional, partially irrelevant data, PLS also requires another parameter: the desired dimension l of the reduced input space. In LWPR this parameter is incremented until the performance gain can be neglected (which turns out to be another parameter).

The major difference to RBFNs, though, is the on-demand creation of kernels and the optimization of kernel *shape* and *size* during training. When a new sample is fed into LWPR and no kernel elicits an activation greater than some threshold, a new kernel centered on the new input is created. The shape of kernels is then optimized due to a cost function close to the least squares sense.

However, when the shape is optimized to minimize the squared error, then the obvious solution are very small kernels, so small that one kernel is responsible for one data point. With an increasing amount of data, an ever increasing number of very small kernels is required and consequently a slightly different approach is taken. Let ϕ be a single kernel with its local model h to approximate samples $(\boldsymbol{x}, y)_i$. Instead of minimizing the sum of *all* squared errors

$$\frac{\sum \phi(\boldsymbol{x}_i)(y_i - h(\boldsymbol{x}_i))^2}{\sum \phi(\boldsymbol{x}_i)}\,, \tag{2.29}$$

the leave one out cross validation [75]

$$\frac{\sum \phi(\boldsymbol{x}_i)(y_i - h(\boldsymbol{x}_{i,-i}))^2}{\sum \phi(\boldsymbol{x}_i)} \tag{2.30}$$

is used, where the index $i, -i$ indicates that the i-th sample is trained on all other data points $j \neq i$. Still, this yields infinitely small kernels and therefore a shrinkage penalty is added

$$\frac{\sum \phi(\boldsymbol{x}_i)(y_i - h(\boldsymbol{x}_{i,-i}))^2}{\sum \phi(\boldsymbol{x}_i)} + \gamma \sum D_{jk}\,, \tag{2.31}$$

where D_{jk} is one entry of the kernel matrix and a smaller entry refers to a greater radius. Minimizing the latter term yields infinitely large kernels, minimizing the former results in infinitely small kernels, and the parameter γ defines the balance. Minimizing both means reduction of the squared error with reasonably sized kernels. Details of the incremental update rules that approximate this optimization can be found in [75, 91].

2.5.2 XCSF – a Learning Classifier System

An alternative route is taken by the XCSF [98, 99] algorithm. The naming conventions are – again – different but the idea remains the same. For now, only a brief introduction is given, as the algorithm is explained in depth later.

XCSF evolves a population of rules, where each rule specifies a kernel accompanied by a local model. The rules jointly approximate a function surface. Moreover, it works online, that is, iteratively. Local models are typically linear and trained via RLS; the kernels can be described by arbitrary distance metrics.

In contrast to LWPR however, the shape and size of kernels is not adapted by a stochastic gradient descent, but instead by a Genetic Algorithm (GA). Therefore, the cost function is termed fitness function and its explicit derivative is not required, as a GA does not directly climb the gradient but instead samples some individuals around the current point and picks its path based on the evaluated surrounding. This allows for more complicated cost (fitness) functions, where explicit derivatives are not easily available.

Instead of introducing a penalty term, the error is not minimized to zero but to a certain target error. This gives a clear idea of the desired accuracy and resolves the balancing issue of parameter γ in Equation (2.31) for LWPR. However, other tricks are required to achieve a suitable population of kernels, including a *crowding* factor that prevents that the majority of kernels hog to simple regions of the function surface but instead spread over the full input space. A detailed algorithmic description is given later.

2.6 Discussion: Applicability and Plausibility

The different approaches to FA introduced in the previous sections are now inspected for their *applicability* to the problem statement of this thesis as well as for the *biological plausibility.* Lets recall the requirements for FA stated in Section 1.4:

- online learning,
- with feasible computational complexity,
- using some neuron-like structures in a population code,
- without putting too much prior information into the system.

When the underlying function type is known (e.g. linear, polynomial, exponential, etc.), only the proper parameters of that function (e.g. $ax^2 + bx + c$) have to be found. The so called *curve fitting*, *model fitting*, or *parametric regression* methods find those parameters a, b, c as outlined in Section 2.3. Iterative variants with acceptable computational complexity are available as well, e.g. RLS. Furthermore, a neural implementation of RLS is possible [95]. However, when the function type is hard-coded into the learner – for example, that would be a combination of multiple sine waves for a kinematic chain – the biological plausibility is questionable. Humans can not only work with kinematic chains they are born with, e.g. arms and legs, but also learn to use new, complex tools that have effects quite different from sine waves: for example, driving a car.

An alternative, rather simple approach to FA without particular assumptions on the function type is *interpolation.* Even simple spline-based models can result in suitable models of highly non-linear function surfaces. GPR is the statistically sound variant that shows high accuracy even for small sample size at the cost of a quadratic computational complexity in the number of samples. However, all such approaches require to store the full data set and therefore apply to finite data sets only. Even if only a subset of a continuous data stream (that a brain receives every day) is stored, the biological plausibility is questionable: Are kinematic models stored in an neural look-up table that is queried for movement execution? It is reasonable to assume a higher level of storage abstraction for population codes, where each neuron is responsible for a certain surrounding not just a single point in space.

As ANNs are inspired by the brain structure, they are inherently plausible in the biological sense. In particular, the RBFN is a suitable candidate for the present work: A population of neurons jointly models a non-linear function surface. Each neuron responds to a certain area of the input space defined by a kernel function. In its area of responsibility a simple local model is learned by an iterative gradient descent. Thus the method works online and, furthermore, with a low computational demand. The major drawback of RBFNs is the missing training possibilities for the kernels.

This issue is addressed by two *local learning* algorithms, namely LWPR and XCSF. Similar to RBFNs those algorithms iteratively model non-linear functions with several local models whose area of responsibility is defined by a kernel. This can be seen as a population code, a population of neurons that learns a particular mapping from one space onto another. The computational demand highly depends on the number of neurons or kernels, which does not matter in a parallelized environment such as the brain, where neurons fire asynchronously, independently of each other.

There are many other FA techniques available, and some of them may be extended or modified to fit the desired framework outlined here. However, the simple RBFN and its more sophisticated brothers LWPR and XCSF naturally fit the needs for a biologically plausible learning framework. The next chapter analyzes the common features of those algorithms in depth to not only foster a better understanding but eventually to lay the foundation for new Machine Learning (ML) algorithms that are well-suited to explain cognitive functionality.