Preprint of "Many Regression Algorithms, One Unified Model – A Review"

Full citation: Freek Stulp and Olivier Sigaud. Many Regression Algorithms, One Unified Model – A Review. *Neural Networks*, 2015 Sep, Volume 69: 60-79

Digital Object Identifier: 10.1016/j.neunet.2015.05.005

Abstract: http://www.sciencedirect.com/science/article/pii/S0893608015001185

This is a preprint from 23.04.2015, and differs from the final published version.

Many Regression Algorithms, One Unified Model – A Review

Freek Stulp

École Nationale Supérieure de Techniques Avancées (ENSTA-ParisTech)
828, Boulevard des Marchaux, 91762 Palaiseau Cedex, France
FLOWERS Research Team, INRIA Bordeaux Sud-Ouest.
freek.stulp@ensta-paristech.fr

Olivier Sigaud

Sorbonne Universités, UPMC Univ Paris 06, UMR 7222, F-75005 Paris, France
olivier.sigaud@isir.upmc.fr *+33 (0) 1 44 27 88 53*

Abstract

Regression is the process of learning relationships between inputs and continuous outputs from example data, which enables predictions for novel inputs. The history of regression is closely related to the history of artificial neural networks since the seminal work of Rosenblatt (1958). The aims of this paper are to provide an overview of many regression algorithms, and to demonstrate how the function representation whose parameters they regress fall into two classes: a weighted sum of basis functions, or a mixture of linear models. Furthermore, we show that the former is a special case of the latter. Our ambition is thus to provide a deep understanding of the relationship between these algorithms, that, despite being derived from very different principles, use a function representation that can be captured within one unified model. Finally, step-by-step derivations of the algorithms from first principles and visualizations of their inner workings allow this article to be used as a tutorial for those new to regression.

Keywords: regression, locally weighted regression, Gaussian mixture regression, radial basis function networks, Gaussian process regression

1. Introduction

Regression is the process of learning relationships between inputs and continuous outputs from example data, which enables predictions for novel inputs. This relationship is represented as a function $f : X \to Y$, which predicts, for instance, a person's height from their age. Here, the input space X (age), is known as the dependent variable, and the output space Y (height) as the independent variable (Fisher, 1925). In the example, the training data consists of concrete age and height measurements for a set of people.

Regression is a form of supervised learning where the output space is continuous, i.e. $Y \subseteq \mathbb{R}^M$. In *parametric* regression, one assumes that the function f is well represented by a specific parameterized model, for instance a linear model $f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x}$. With a linear model, the model parameters are the slopes $\mathbf{a}$. The aim of parametric regression is to find the parameters of the model that minimize some error on the training examples.

Another example of a parameterized model is a Radial Basis Function Network (Park and Sandberg, 1993), where the function is modeled as a weighted sum of basis functions $f(\mathbf{x}) = \sum_{e=1}^{E} w_e \phi_e(\mathbf{x})$. If we assume that the basis functions $\phi_{e=1\ldots E}$ have pre-specified centers and widths, the model parameters that are to be determined through parametric regression are the weights $w_{e=1\ldots E}$, see Figure 1.

The generic scheme for parametric regression is depicted in Figure 2. The input to the regression algorithm is the training data and a set of algorithmic meta-parameters, including for instance learning rates. Each regression algorithm assumes a certain type of model, e.g. linear least squares assumes a linear model. The output of the algo-

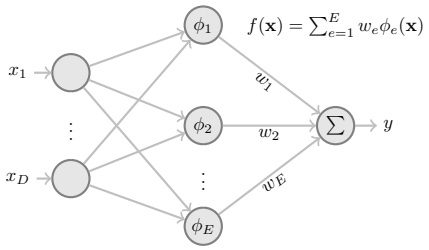

Figure 1: Radial Basis Function Network (RBFN) with D inputs and E radial basis functions. Unlabeled connections have weight 1.

rithm is a vector of model parameters, which are determined by minimizing an error measure on the training data. Evaluating the model to make predictions for novel inputs requires both the model (e.g. $f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x}$) and its model parameters (e.g. $a = [2\ 1]^\intercal$). A detailed discussion of the differences between model parameters and meta-parameters is given in Section 2.6.

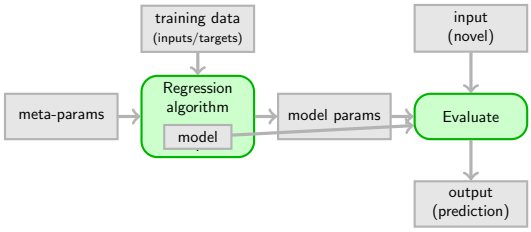

Figure 2: Generic flowchart for parametric regression.

In this article, we take a *model-centric* view on regression, which means that we classify and analyze algorithms based on the model they assume, rather than the algorithmic procedure that is used to optimize the parameters of this model. Our first contribution is to show that the models used in a wide variety of regression algorithms (listed in Table 1) fall into two main classes: a mixture of linear models or a weighted sum of basis functions.

Our second contribution is to demonstrate that the latter class of models (weighted sum of basis functions) is a special case of the former one (mixture of linear models). As a consequence, and rather strikingly, *all* the algorithms in Table 1 – despite having being derived from very different principles – use parameterized functions that can be described by one *unified model*. This has been visualized in Figure 3. Thus, these regression algorithms should not be thought of as using their own distinct model customized to the algorithmic procedure, but rather as using models that are special cases of the unified model. Such a perspective provides a deeper understanding of the relationship between these algorithms, and is a necessary step towards *model-based machine learning*, as proposed by Bishop (2013), i.e. the idea of the automated selection of the adequate machine learning algorithm given the formal description of a specific learning problem.

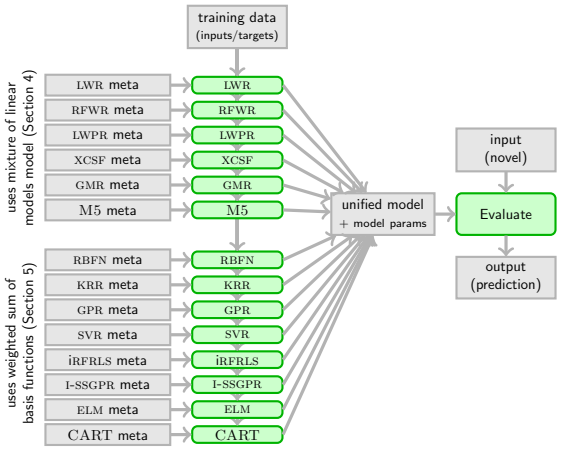

Figure 3: The algorithms described in this paper have a model that is either a mixture of linear models (upper block) or a weighted sum of basis functions (lower block). Since the former is a special case of the latter, the output of all algorithms can be described by one unified model. Full algorithm names are given in Table 1.

Despite our model-centric view, we do describe and explain the algorithmic procedures used in different regression algorithms, including (regularized) least squares, expectation-maximization, backpropagation, decision tree learning, and Gaussian process regression. This is necessary to understand why an algorithm assumes a certain type of model, and how that model relates to the unified model we propose. These explanations however, should not distract from the fact that our main interest is in the *underlying model* that the algorithms assume (linear model, RBFN, model tree, Gaussian mixture model, Gaussian process), and that all these models are special cases of the unified model.

Explaining the algorithms also allows this article to be used as a tutorial on regression; we provide an overview of many algorithms, show their derivations from first principles, visualize their inner workings so that novices may acquire an intuitive understanding, and provide network representations for readers with a background in artificial neural networks. Using one unified, easy to

Model: mixture of linear models	
LWR	Locally Weighted Regression (Atkeson and Schaal, 1995)
RFWR	Receptive Field Weighted Regression (Schaal and Atkeson, 1997)
LWPR	Locally Weighted Projection Regression (Vijayakumar and Schaal, 2000)
XCSF	. (Butz and Herbort, 2008)
GMR	Gaussian Mixture Regression (Hersch et al., 2008; Calinon, 2009)
M5	Model Trees (Quinlan, 1992)
Model: weighted sum of basis functions	
RBFNs	Radial Basis Function Networks (Park and Sandberg, 1993)
KRR	Kernel Ridge Regression (Saunders et al., 1998)
GPR	Gaussian Process Regression (Williams and Rasmussen, 2006)
SVR	Support Vector Regression (Vapnik, 1995)
iRFRLS	Incr. Random Features Regularized Least Squares (Gijsberts and Metta, 2011)
I-SSGPR	Incr. Sparse Spectrum Gaussian Process Regr. (Gijsberts and Metta, 2012)
CART	Regression Trees (Breiman et al., 1984)
ELM	Extreme Learning Machine (Huang et al., 2006b)
BPROP	Backpropagation (Werbos, 1974)

Table 1: List of algorithms presented in this article.

understand model highlights relationships between algorithms; a key to acquiring more global understanding of regression methods. It is not our aim to be exhaustive, in terms of presenting *all* regression algorithms and their variants. This would distract from our actual aim, which is to highlight the similarities and differences between those algorithms whose underlying model is a special case of the unified model. For further reading, we provide references to other tutorials and books, including the work of Smola and Schölkopf (2004), Bishop et al. (2006), Williams and Rasmussen (2006), and Schmidhuber (2014).

The rest of this article is structured as follows. In the next section, we present (regularized) *linear least squares*, the classical algorithm for linear regression. In Section 3, we describe how linear least squares has been extended in two different directions. These two directions have lead to two classes of algorithms, which generate either a mixture of linear models or a weighted sum of basis functions. In Section 3.3, we show that these two different classes of models can in fact be unified, the latter being a special case of the former. Algorithms that use these two classes of models are presented in Section 4 and 5 respectively. In Section 6, we discuss the impact of the unifying perspective, before concluding with Section 7.

2. Linear Least Squares

In this section, we present *linear least squares* (LLS) regression. We describe some variants of the basic algorithm, including different regularization methods, multivariate LLS, and recursive least squares. The aim of the following section – Section 3 – is then to show how *linear* least squares is used in the context of *non-linear* regression.

First, we formalize some of the concepts related to parametric regression, as presented in the introduction. The aim of supervised learning is to learn a function $f : X \to Y$ from N training examples[1] $\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$, where $\forall n, \mathbf{x}_n \in X \wedge \mathbf{y}_n \in Y$. In regression, the output space is continuous ($Y \subseteq \mathbb{R}^M$). Multi*variable* (resp. multi*variate*) regression refers to input spaces X (resp. output spaces Y) with a dimensionality of more than one[2].

In the case of *linear* regression, the family of model functions is linear, which means that $f(\mathbf{x})$ is represented as a line (1D), plane (2D) or hyperplane ($\geq$3D). Formally, if the data is assumed centered so that the hyperplane goes through the origin, the underlying model is

$$f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x}, \tag{1}$$

where $\mathbf{a}$ is a $D \times 1$ vector that contains the slopes of the linear function, as visualized in Figure 4.

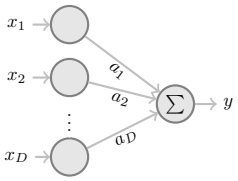

Figure 4: Network representation of a linear model.

A popular method for determining $\mathbf{a}$ is the linear least squares (LLS) algorithm. It uses the vertical

[1] Throughout the paper, we denote scalars as lowercase symbols (x), vectors as bold lowercase symbols ($\mathbf{x}$) and matrices as bold uppercase symbols ($\mathbf{X}$).

[2] To facilitate presentation and visualizations, we mostly use $Y \subseteq \mathbb{R}^1$ in this paper, i.e. the output is 1-dimensional, and the target values y_n are scalars rather than vectors. Multivariate regression is presented in Section 2.4.

distance between observed values y_n and the predictions $f(\mathbf{x}_n)$, which are known as the *residuals*:

$$r_n = y_n - f(\mathbf{x}_n) \tag{2}$$
$$= y_n - \mathbf{a}^\intercal \mathbf{x}_n. \quad \text{(for linear models)} \tag{3}$$

The residuals are visualized as vertical lines in Figure 5.

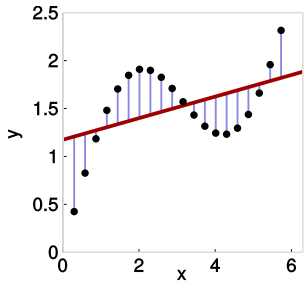

Figure 5: Illustration of least squares. Black dots represent the training examples, and the thick line is the learned function $f(\mathbf{x})$. Vertical lines represent residuals.

In the LLS case, the sum of the squared residuals

$$S(\mathbf{a}) = \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n))^2 \tag{4}$$
$$= \sum_{n=1}^{N} (y_n - \mathbf{a}^\intercal \mathbf{x}_n)^2 \tag{5}$$

is minimized, which in matrix form is

$$S(\mathbf{a}) = (\mathbf{y} - \mathbf{X}\mathbf{a})^\intercal (\mathbf{y} - \mathbf{X}\mathbf{a}), \tag{6}$$

with $\mathbf{X}$ being the $N \times D$ *design matrix* that combines all the input vectors, and the $N \times 1$ vector $\mathbf{y}$ combining all scalar target values:

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D} \end{bmatrix}, \ \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \tag{7}$$

where each row corresponds to one input/target example. The slopes of the linear model that minimize (6) are

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} S(\mathbf{a}) \tag{8}$$
$$= \arg\min_{\mathbf{a}} (\mathbf{y} - \mathbf{X}\mathbf{a})^\intercal (\mathbf{y} - \mathbf{X}\mathbf{a}), \tag{9}$$

which is a continuously differentiable unconstrained optimization problem. The solution of this problem is to determine when the derivative

$$S'(\mathbf{a}) = 2(\mathbf{a}(\mathbf{X}^\intercal \mathbf{X}) - \mathbf{X}^\intercal \mathbf{y}) \tag{10}$$

is 0, which is the case when

$$\mathbf{a}^* = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}. \tag{11}$$

Thus, given the input data in $\mathbf{X}$ and target data in $\mathbf{y}$, the linear function that best fits the data – in terms of minimizing the sum of squared residuals – is $f(\mathbf{x}) = \mathbf{a}^{*\intercal} \mathbf{x}$.

Before continuing, we want to draw explicit attention to the difference between the *model* used in this section – the linear model in (1) – and the *algorithm* used to determine the *model parameters* of the model – the computations in (11). Note that to make a prediction for a novel input $\mathbf{x}^{\text{new}}$, we need *only* to know the model (linear) and its parameters ($\mathbf{a}$), but not the algorithm used to compute $\mathbf{a}$.

2.1. Least Squares and Least Deviations

The L^p-norm $\|\mathbf{z}\|_p$ of a D-dimensional vector $\mathbf{z}$ is defined as

$$\|\mathbf{z}\|_p = \left(\sum_{d=1}^{D} |\mathbf{z}_d|^p \right)^{\frac{1}{p}}. \tag{12}$$

The L^2-norm is commonly known as the Euclidean distance, and the L^1-norm as the Manhattan distance.

As the name implies, linear least *squares* uses the sum of *squares* of the residuals as the error measure to minimize. This corresponds to the square of the L^2-norm of the residuals, and we may rewrite (9) as

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} \ (\|\mathbf{y} - \mathbf{X}^\intercal \mathbf{a}\|_2)^2 \tag{13}$$
$$= \arg\min_{\mathbf{a}} \left(\left(\sum_{n=1}^{N} |y_n - \mathbf{a}^\intercal \mathbf{x}_n|^2 \right)^{\frac{1}{2}} \right)^2 \tag{14}$$
$$= \arg\min_{\mathbf{a}} \left(\sum_{n=1}^{N} |y_n - \mathbf{a}^\intercal \mathbf{x}_n|^2 \right). \tag{15}$$

Alternatively, one may minimize the L^1-norm of the residuals

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} \ \|\mathbf{y} - \mathbf{X}^\intercal \mathbf{a}\|_1 \tag{16}$$
$$= \arg\min_{\mathbf{a}} \left(\sum_{n=1}^{N} |y_n - \mathbf{a}^\intercal \mathbf{x}_n| \right). \tag{17}$$

which is known as "least absolute deviations" (Bloomfield and Steiger, 1980). Regression with the L^1-norm is more robust, but a disadvantage is that (16) cannot be solved analytically, so an iterative optimization approach is required.

2.2. Regularized Least Squares

Potential singularities in $(\mathbf{X}^\intercal\mathbf{X})$ may make it difficult to invert this matrix, which can result in very large values in the vector $\mathbf{a}$. A solution to this issue is to explicitly penalize large weights, which results in Regularized Linear Least Squares (RGLS), where (9) becomes

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} \left(\frac{\lambda}{2}\|\mathbf{a}\|^2 + \frac{1}{2}\|\mathbf{y} - \mathbf{X}^\intercal\mathbf{a}\|^2 \right), \quad (18)$$

where λ is a parameter that determines the trade-off between small values in $\mathbf{a}$ and small residuals. Using the L^2-norm for $\|\mathbf{a}\|$

$$\|\mathbf{a}\|_2 = \left(\sum_{d=1}^{D} |a_d|^2 \right)^{\frac{1}{2}} \quad (19)$$

is called Thikonov regularization. RGLS with Thikonov regularization is known as Ridge Regression (RR). Using the same derivation as for (11), the analytical solution to this least squares minimization problem is

$$\mathbf{a}^* = (\lambda\mathbf{I} + \mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}. \quad (20)$$

It is also possible to use the L^1-norm

$$\|\mathbf{a}\|_1 = \sum_{d=1}^{D} |a_d|, \quad (21)$$

which has the advantage that values in $\mathbf{a}$ will tend to go to zero. In the context of basis function networks (Section 5), the L^1 norm can thus be used as a feature selector. Here again, using the L^1-norm does not have an analytical solution as in (20). Several algorithms for finding solutions are described in (Tibshirani, 1996; Schmidt, 2005). In the context of RGLS, the L^1-norm is known as the "least absolute shrinkage and selection operator" (LASSO).

Another norm is used in Support Vector Regression (SVR) methods, which we discuss in Section 5.5. These three norms are visualized in Figure 6.

2.3. Adding offsets to the linear model

In the above, we have assumed that the data is centered, in which case a linear model $f(\mathbf{x}) = \mathbf{a}^\intercal\mathbf{x}$ suffices. If the data is not centered, an offset should be added so that the linear model becomes $f(\mathbf{x}) = \mathbf{a}^\intercal\mathbf{x} + b$, which we may rewrite as $f(\mathbf{x}) = \left[\begin{smallmatrix}\mathbf{a}\\ b\end{smallmatrix}\right]^\intercal \left[\begin{smallmatrix}\mathbf{x}\\ 1\end{smallmatrix}\right]$. In this case, the design matrix becomes

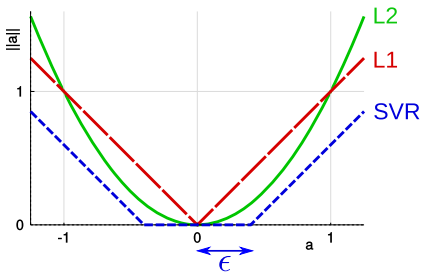

Figure 6: Visualization of the L^1, L^2, and SVR norm.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} & 1 \\ \vdots & \vdots & \ddots & \vdots & 1 \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D} & 1 \end{bmatrix} \quad (22)$$

The result of fitting a linear model with offsets to our example data – which are sampled from the function $y = 0.5x + \sin(x)$ – is depicted in Figure 5, where the estimated function is $f(x) = 0.11x + 1.18$.

2.4. Multivariate regression

In multivariate regression, the output space is multi-dimensional ($Y \subseteq \mathbb{R}^M$), where the output is thus a vector, $\mathbf{a}$ becomes a matrix $\mathbf{A}$, and the linear model is

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}^\intercal\mathbf{x}. \quad (23)$$

The design matrix of inputs is the same matrix $\mathbf{X}$, but the targets are now stored in a matrix $\mathbf{Y}$, and the least squares solution becomes

$$\mathbf{A}^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{Y}. \quad (24)$$

This solution is decoupled between the different output variables, and corresponds to M separate univariate least squares solutions:

$$\mathbf{a}_m^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}_m. \quad (25)$$

In the rest of this paper, we will mostly use a 1-D output, for ease of presentation and visualization. All examples are easily extended to the multivariate case by applying (24) or (25).

2.5. Batch vs. Incremental Learning

Equations (11) and (20) provide a *batch* method for finding the optimal weights $\mathbf{w}^*$, by considering

the training examples all at once. In contrast, *incremental* methods are able to update the weights (and other parameters of the model) incrementally, one training example at a time.

Throughout this paper, we present several models for which both batch and incremental algorithms exist. For instance, the weights of a single-layer feedforward network can be trained through Extreme Learning Machines (batch) or backpropagation (incremental). The weights of a mixture of linear models can be updated through locally weighted regression (batch) or receptive field weighted regression (incremental).

For least squares, the batch method in (11) and (20) requires the inversion of a matrix, which has complexity $\mathcal{O}(N^3)$. In fact, incremental methods also exist for linear least squares itself. The most well-known is the Recursive Least Squares (RLS, Plackett, 1950), which updates the linear model from each new example without recomputing (11) at each step.

Alternatively, one can use the Sherman-Morrison formula, which enables the inverse to be updated incrementally, reducing the inversion complexity to $\mathcal{O}(N^2)$. However, this method is sensitive to rounding errors; a numerically more stable option consists in updating the Cholesky factor of the matrix using the QR algorithm (Gijsberts and Metta, 2012).

Another incremental approach to solving these equations consists in optimizing the weights of a matrix that stands for $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ using gradient methods, as in the backpropagation algorithm for instance (see Section 5.10).

Note that the distinction between batch and incremental learning is related to, but not the same as the distinction between offline and online learning. A batch algorithm uses all the training data at once. An incremental algorithm updates an existing model *incrementally*, using one training example at a time. In offline learning, all examples are stored and accessible, whereas in online learning a training example is forgotten once it has been processed. Batch learning is always off-line, because it needs access to all the training examples at once, and they must thus be stored in memory. Online training always requires an incremental algorithm, because only one data example can be considered at a time, and the algorithm must thus be able to update the model incrementally. However, incremental learning can also be performed in an offline setting, by providing the stored examples to the algorithm one by one, but without deleting them from memory. The term 'online' is often (inaccurately) used to denote 'incremental' (Wilson and Martinez, 2003).

2.6. Model parameters and Meta-parameters

Figure 2 illustrates that regression algorithms assume a certain type of model. Each algorithm is designed to determine the optimal values of the parameters of this model, given an optimization criterion. Meta-parameters are algorithmic parameters that the user has to provide as an input to the algorithm. An understanding of linear least squares allows us to now give specific examples for an algorithm's model parameter and meta-parameters. This distinction is important for later sections.

The model used in linear least squares is a linear model, i.e. $f(\mathbf{x}) = \mathbf{a}^\intercal\mathbf{x}$. The model parameters are thus $\mathbf{a}$. If we use an offset $f(\mathbf{x}) = \mathbf{a}^\intercal\mathbf{x} + b$, the model parameters are $\mathbf{a}$ and b. Model parameters are *all* the parameters that are required to make a prediction for a novel output.

Linear least squares has no meta-parameters; there is no parameter the user has to tune to make the algorithm work. But in linear least squares with Thikonov regularization, the user has to tune the parameter λ, which determines the trade-off between having small values in $\mathbf{a}$ and having small residuals. The parameter λ is thus a meta-parameter of regularized linear least squares.

The important insight is that regularized LLS and 'standard' LLS use the exact same model, and thus have the same aim: optimize the values in the model parameter vector $\mathbf{a}$. 'Standard' LLS will always optimize $\mathbf{a}$ in the same way, and the values in $\mathbf{a}$ will always be the same for the same training data. With regularized LLS however, the resulting values in $\mathbf{a}$ depend not only on the training data, but also on the meta-parameter λ.

The main contribution of this article is to show that all the algorithms in Table 1 use the same unified model, and they thus all have the same aim: optimize the values in the 'unified model parameter vector'. They algorithms *only* differ in the procedure used to determine the values of the parameters of the unified model, and in the meta-parameters that the user must tune.

3. From Linear Least Squares to Non-linear Regression

Linear least squares (LLS) regression can be extended in two distinct ways to non-linear regression,

yielding two classes of algorithms:

- Algorithms that perform multiple weighted LLS regressions, using different input-dependent weighting functions. The resulting model is a mixture of linear models. This model is used in LWR, RFWR, LWPR, XCSF, GMR, and M5 (see Table 1 for full names).
- Algorithms that project the input space into a feature space using a set of non-linear basis functions, and performing one LLS regression in this projected feature space. The resulting model is a weighted sum of basis functions. This model is used in RBFNs, KRR, GPR, SVR, iRFRLS, I-SSGPR, CART, and ELM.

In this section, we first present two algorithms – Locally Weighted Regression (LWR) and regression with Radial Basis Function Networks (RBFN) – which are representative for each class. We have chosen these two algorithms as representatives of their class for pedagogical reasons, because they are the most straight-forward extensions to linear least squares, and thus require only little explanation beyond the previous section. Based on the models used in these illustratory algorithms, we then introduce the unified model. In Section 4 and 5, we go through the algorithms in Table 1 one by one, we highlight their relationships and demonstrate how they all use the same unified model to represent the parameterized function.

3.1. Locally Weighted Regression (LWR)

When performing the fitting in LLS, we may prefer to give certain data points more weight than others. Thus, for each training example $(\mathbf{x}_n, y_n)$, we define a corresponding weight w_n. With these weights, the sum of residuals is now computed as

$$S(\mathbf{a}) = \sum_{n=1}^{N} w_n (y_n - \mathbf{a}^\intercal \mathbf{x}_n)^2. \tag{26}$$

When minimizing $S(\mathbf{a})$ with this weighting, it is more important to minimize residuals that have large weights. For instance, examples $(\mathbf{x}_n, y_n)$ that have a weight of $w_n = 0$ do not contribute at all.

Defining $\mathbf{W}$ to be an $N \times N$ diagonal weight matrix with $\mathbf{W}_{nn} = w_n$, and rewriting (26) in matrix form

$$S(\mathbf{a}) = (\mathbf{y} - \mathbf{X}\mathbf{a})^\intercal \mathbf{W} (\mathbf{y} - \mathbf{X}\mathbf{a}), \tag{27}$$

the weighted linear least squares solution becomes

$$\mathbf{a}^* = (\mathbf{X}^\intercal \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{W} \mathbf{y}. \tag{28}$$

The weights for each sample are typically defined as a function of the input space through a function ϕ, parameterized with $\boldsymbol{\theta}$

$$w_n = \phi(\mathbf{x}_n, \boldsymbol{\theta}). \tag{29}$$

A commonly used weighting function is the multivariate Gaussian:

$$\phi(\mathbf{x}_n, \boldsymbol{\theta}) = g(\mathbf{x}_n, \mathbf{c}, \boldsymbol{\Sigma}) \text{ with } \boldsymbol{\theta} = (\mathbf{c}, \boldsymbol{\Sigma}) \tag{30}$$

$$g(\mathbf{x}, \mathbf{c}, \boldsymbol{\Sigma}) = \exp\left(-\tfrac{1}{2}(\mathbf{x} - \mathbf{c})^\intercal \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \mathbf{c})\right), \tag{31}$$

where $\boldsymbol{\theta}$ is a tuple containing the parameters of a multivariate Gaussian. An example of a Gaussian weighting function with $\mathbf{c} = \pi$ and $\boldsymbol{\Sigma} = 1.0$ is plotted in graph (B2) of Figure 7. In (B1), the weights w_n are visualized by the thickness of the vertical lines representing the residuals. When comparing (A1) and (B1), it becomes clear that weighting the examples can lead to a very different parameterization of the linear model.

Again, we draw explicit attention to the fact that the *model* in linear least squares and weighted linear least squares is exactly the same (1); only the *algorithm* used to determine the *model parameters* $\mathbf{a}$ is different, i.e. compare (11) and (28).

Locally Weighted Regression (LWR) is an extension of weighted linear least squares, in which E independent weighted regressions are performed on the same data (in the design matrix $\mathbf{X}$), but with E different weight matrices $\mathbf{W}_e$.

$$\texttt{for } e = 1 \ldots E$$
$$\mathbf{a}_e = (\mathbf{X}^\intercal \mathbf{W}_e \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{W}_e \mathbf{y}. \tag{32}$$

In practice, this usually means using the same type of weighting function, but with different parameter vectors $\boldsymbol{\theta}_{e=1\ldots E}$, for instance a set of E Gaussians with different centers $\mathbf{c}_{e=1\ldots E}$, as depicted in graph (C2) of Figure 7.

Performing weighted least squares E times for the different weighting functions leads to E local linear models $\mathbf{a}_e^\intercal \mathbf{x} + b_e$, which are visualized as dashed lines in (C1). By choosing E Gaussian weighting functions $\phi_e(\mathbf{x}) = g(\mathbf{x}, \mathbf{c}_e, \boldsymbol{\Sigma}_e)$ with different centers $\mathbf{c}_e$ in the input space, each linear model is thus responsible for fitting a different part of the data, as depicted in (C1) in Figure 7. This 'local responsibility' is the reason for the name *Locally* Weighted Regression.

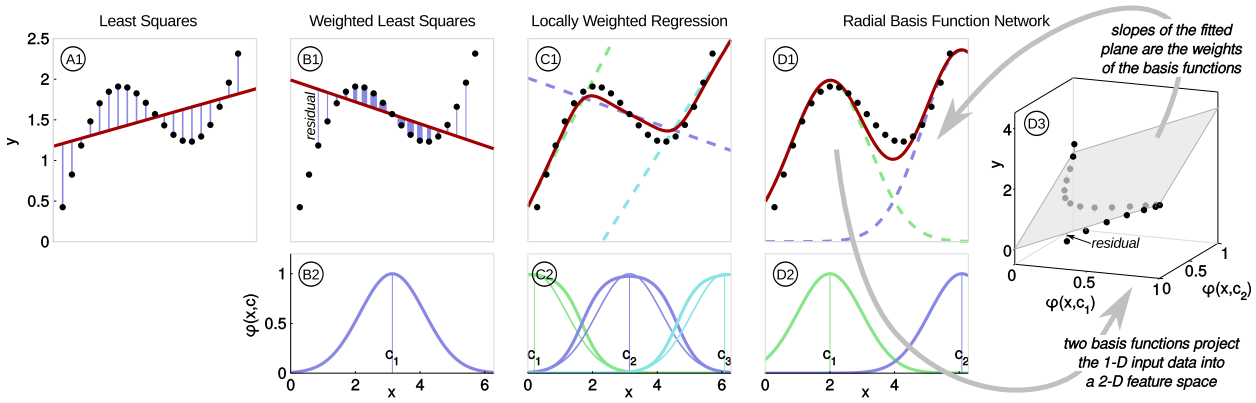

Figure 7: Illustration of four regression algorithms. Black dots represent the 20 training examples, and the thick (red) line is the learned function $f(\mathbf{x})$. Vertical lines represent residuals, where the thickness of these lines indicates their weight. Dashed lines represent local models. The lower row visualizes the basis functions, where for LWR the thin line represents the unnormalized basis functions (normalization is not necessary for the other algorithms). The right graph plots the feature space for RBFN. Each data point in graph D3 is acquired by computing the value of both $\phi(x, c_1)$ and $\phi(x, c_2)$ (which can be read from D2), and plotting y against these two values. Because the 2D feature space $\phi(x, c_1)$, $\phi(x, c_2)$ is of a higher dimension that the 1D input space x, we now have a 3D plot.

The resulting model is a weighted sum of these linear models, where the weights are determined by the functions $\phi(\mathbf{x}, \boldsymbol{\theta}_e)$:

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot (\mathbf{a}_e^\intercal \mathbf{x}). \tag{33}$$

If we include the offsets b_e in the linear model and use the extended design matrix as explained in Section 2.3, we acquire the model

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot (\mathbf{a}_e^\intercal \mathbf{x} + b_e), \tag{34}$$

which is visualized in Figure 8.

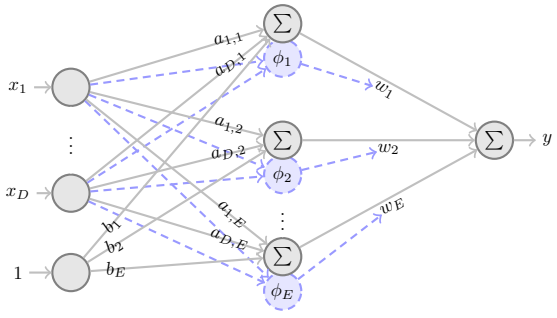

Figure 8: Function model in Locally Weighted Regression, represented as a feedforward neural network. The functions $\phi_e(\mathbf{x})$ generate the weights w_e from the hidden nodes – which contain linear sub-models ($\mathbf{a}_e^\intercal \mathbf{x} + b_e$) – to the output node. Here, ϕ_e is an abbreviation of $\phi(\mathbf{x}, \boldsymbol{\theta}_e)$

Because each of the E local least squares regressions is performed independently of the others, the summation in (33) and (34) requires the weighting functions to be normalized, i.e. $\sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) = 1$. For instance, the normalized Gaussian weighting function is:

$$\phi(\mathbf{x}, \boldsymbol{\theta}_e) = \frac{g(\mathbf{x}, \mathbf{c}_e, \boldsymbol{\Sigma}_e)}{\sum_{e'=1}^{E} g(\mathbf{x}, \mathbf{c}_{e'}, \boldsymbol{\Sigma}_{e'})} \text{ with } \boldsymbol{\theta}_e = (\mathbf{c}_e, \boldsymbol{\Sigma}_e). \tag{35}$$

In summary, the LWR algorithm performs several independent weighted linear least squares regressions on the same data, but each with a different weighting function, usually localized around a different part of the input space. The result is a weighted sum of linear models, where the *models* are linear, and where the *weights* are determined by the (normalized) weighting functions.

3.2. Regression with Radial Basis Function Networks (RBFNs)

Instead of differently weighting each input sample – as in weighted linear least squares and LWR – an alternative extension to LLS is to project each input sample into a higher-dimensional feature space using a set of basis functions. Thus, instead of a linear function model $f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x} = \sum_{d=1}^{D} a_d \cdot x_d$ we now have

$$f(\mathbf{x}) = \sum_{e=1}^{E} a_e \cdot \phi(\mathbf{x}, \boldsymbol{\theta}_e). \tag{36}$$

Here, $\mathbf{a}$ is no longer of size $D \times 1$ (with D the dimensionality of the input examples) but rather $E \times 1$ (with E the number of basis functions). In the literature, it is more customary to use $\mathbf{w}$ ('weights') instead of $\mathbf{a}$; a custom which we adopt in this section.

$$f(\mathbf{x}) = \sum_{e=1}^{E} w_e \cdot \phi(\mathbf{x}, \boldsymbol{\theta}_e) \tag{37}$$

This model is a weighted sum of basis functions. The term *basis function* is used because the sub-functions $\phi(\mathbf{x}, \boldsymbol{\theta}_e)$ are the 'building blocks', or 'basis', from which $f(\mathbf{x})$ is constructed.

When the basis functions are radial, it is called a Radial Basis Function Network (RBFN). This means that the output of the basis function depends only on the distance to a center, i.e. $\phi(||\mathbf{x} - \mathbf{c}||)$. A frequently used radial basis function is a Gaussian function, as in (31). Two Gaussian basis functions with centers c_1 and c_2 are plotted in (D2) in Figure 7.

In a linear model $f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x}$, the model is linear in the parameters $\mathbf{a}$ *and* linear in the input $\mathbf{x}$. In contrast, (37) is non-linear in the input $\mathbf{x}$, but it is still linear in the parameters $\mathbf{w}$. Thus, we can readily apply LLS to acquire the parameters $\mathbf{w}$, even if the resulting function $f(\mathbf{x})$ is not linear.

To apply LLS to optimize $\mathbf{w}$, we define the projected version of the design matrix – the feature matrix – which is $N \times E$ (one column for each basis function feature) instead of $N \times D$ as in (7)

$$\boldsymbol{\Theta} = \begin{bmatrix} \phi(\mathbf{x}_1, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_1, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_1, \boldsymbol{\theta}_E) \\ \phi(\mathbf{x}_2, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_2, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_2, \boldsymbol{\theta}_E) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\mathbf{x}_N, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_N, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_N, \boldsymbol{\theta}_E) \end{bmatrix} \tag{38}$$

and we get the least squares solution

$$\mathbf{w}^* = (\boldsymbol{\Theta}^\intercal \boldsymbol{\Theta})^{-1} \boldsymbol{\Theta}^\intercal \mathbf{y}. \tag{39}$$

This transforms non-linear regression in the input space into linear regression in a feature space, where the basis functions are responsible for projecting the input space into the feature space. The feature space is generally of higher dimensionality than the input space (i.e. $E > D$). In graph (D3) of Figure 7, we visualize the projection of the data from the 1D input space into a 2D feature space using the two Gaussian basis functions. The plane which minimizes the residuals is also shown. The slopes $\mathbf{w}$ of this plane correspond to the weights of the basis functions, and $w_1\phi(x, c_1)$ and $w_2\phi(x, c_2)$ are plotted in (D1).

Note that the function can be approximated relatively well with only these two basis functions; this is because their position has been manually tuned for this illustratory example. Several methods for automatically tuning the number of basis functions and their parameters are presented in Section 5.2.

In summary, non-linear regression can be achieved by using a set of basis functions to project the input space into a feature space, and to perform LLS in the feature space. The result is a weighted sum of linear models, where the *models* are the basis functions, and where the *weights* $\mathbf{w}$ are determined by a single linear regression. An example of such a model is a radial basis function network.

3.3. A Unified Model

When listing both models – a mixture of linear models (34) and a weighted sum of basis functions (37) – together

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot (b_e + \mathbf{a}_e^\intercal \mathbf{x}) \tag{40}$$

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot \, w_e, \tag{41}$$

it becomes clear that (41) is a special case of (40) with $\mathbf{a}_e = \mathbf{0}$ and $b_e \equiv w_e$. This is also illustrated in Figure 9.

Going from a mixture of linear models (top LWR network in Figure 9) to a sum of weighted basis functions (bottom RBFN network in Figure 9) leads to an interesting shift of interpretation. In an RBFN, the *sub-models* in the hidden nodes are *basis functions*, whereas in LWR, these *sub-models* are *linear models*. In an RBFN, the *weights* are the (degenerate) *lines* ($b_{e=1\ldots E}$), whereas in LWR, these *weights* are determined by the *basis functions*. Thus, the roles of the function implementing the sub-models/weights are switched between the representations. Due to this required shift in interpretation, the relationship between the two models may not be immediately obvious.

A closer inspection of the differences between the models also reveals why multiple LLS regressions are necessary in LWR, whereas only one LLS regression is required in RBFN. In (40), the model is not linear in the parameters b_e and $\mathbf{a}_e$, and LLS cannot be applied.

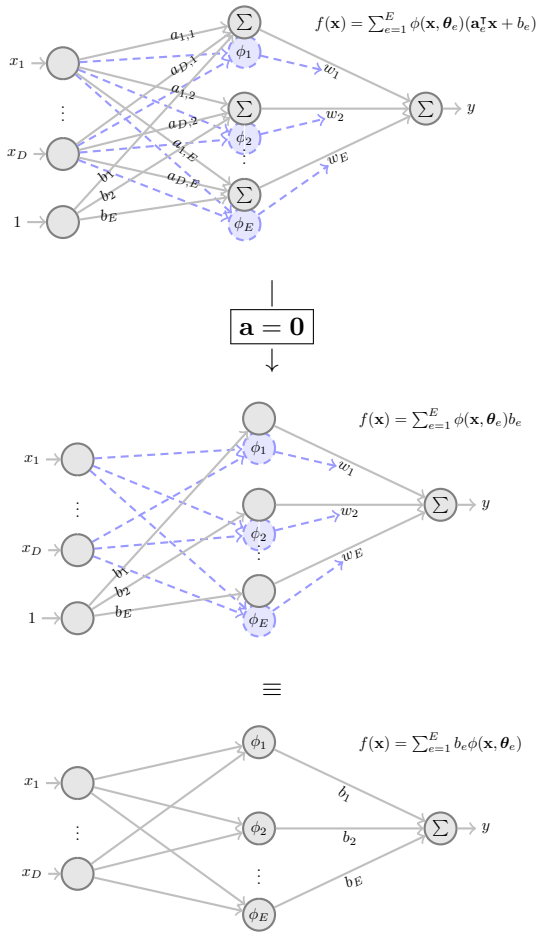

Figure 9: The RBFN model is a special case of the LWR model, with model parameters $\mathbf{a} = \mathbf{0}$. Here, ϕ_e is an abbreviation of $\phi(\mathbf{x}, \boldsymbol{\theta}_e)$

In summary, the model used in LWR is a mixture of linear models, and in RBFN it is is a weighted sum of basis functions, the latter being a special case of the former with $\mathbf{a} = \mathbf{0}$. In the next two sections, we list several algorithms that yield a mixture of linear models (Section 4) and a weighted sum of basis functions (Section 5). Throughout, it is important to keep in mind that the model of all these algorithms are special cases of the unified model in (40).

4. Model: Mixture of Linear Models

In Section 3.1 we summarized that LWR yields a function whose underlying model is a mixture of linear models, where the *sub-models* are linear, and where the *weights* are determined by the (normalized) weighting functions. In this section, we describe several algorithms which yield the same model, including Receptive Field Weighted Regression (RFWR), Locally Weighted Projection Regression (LWPR), XCSF, Gaussian Mixture Regression (GMR), and M5.

4.1. Algorithm: Locally Weighted Regression (revisited)

Before turning to other algorithms, we briefly revisit LWR to discuss its algorithmic meta-parameters. In the LWR algorithm (Atkeson and Schaal, 1995), the number of linear models, and the position and center of the weighting functions are fixed and must be specified by the user. Assuming diagonal covariance matrices for Gaussian weighting functions, this leads to $2 \cdot E \cdot D$ parameters to be tuned, i.e. a D-dimensional center and covariance matrix diagonal for each of the E weighting functions.

When the dimensionality of the input space is not too large, for instance when using LWR in the context of dynamical movement primitives (Ijspeert et al., 2013), it may suffice to specify the number of weighting functions per dimension (i.e. D integers), and space the centers equidistantly within the bounds of the input data. The diagonals of the covariance matrices can be determined automatically by specifying the height at which the (unnormalized) weighting functions should intersect, see Figure 10 for an illustration. This reduces the number of meta-parameters for LWR to $D+1$. Alternatively, cross-validation over the training set may be used to determine these parameters.

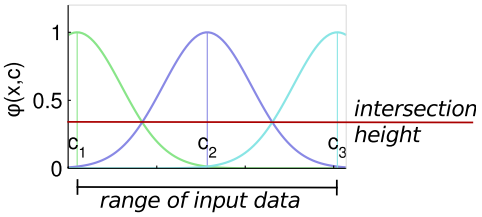

Figure 10: Meta-parameters for LWR: the number of weighting functions ($E = 3$ above) whose centers are spaced equidistantly in the range of the input data, and the intersection height of the unnormalized weighting functions (0.35 above).

Although proposed almost two decades ago, LWR is still a competitive, widely-used algorithm, due to its simplicity. Fitting results may, however, become brittle if the input space dimensionality is high and not enough data is available.

4.2. Algorithm: Receptive Field Weighted Regression

The RFWR algorithm is the incremental variant of LWR, and it automates the choice of several model parameters (Schaal and Atkeson, 1997). The main differences to LWR are:

- RFWR is incremental, rather than batch, and uses Recursive Least Squares (RLS) instead of a batch LS algorithm.
- New linear models and weighting functions – called receptive fields – are added automatically by the algorithm when needed, i.e. when some input point is not covered by any receptive field.
- The centers and covariance matrices of the receptive fields are also adapted.

This flexibility comes at the cost of having to tune several meta-parameters, such as thresholds for removing or adding receptive fields as well as first and/or second order learning rates. These meta-parameters control the dynamics of incremental learning; these dynamics are often difficult to predict, let alone the influence the meta-parameters have on the dynamics. Therefore, setting these meta-parameters is not always easy nor intuitive.

4.3. Algorithm: Locally Weighted Projection Regression

The LWPR algorithm (Vijayakumar and Schaal, 2000) is an extension of RFWR, in which a low-dimensional affine projection is realized before fitting the linear models. For 1-dimensional input spaces, no such low-dimensional projection is possible, and LWPR becomes equivalent to RFWR (Vijayakumar and Schaal, 2000). In LWPR, linear models are fitted using an iterative version of Partial Least Squares called NIPALS (Geladi and Kowalski, 1986).

As outlined by Sigaud et al. (2011), LWPR is particularly interesting to perform regression when the data lies in a limited domain - because it adds receptive fields only in this domain - in a space with a high dimensionality - because it uses NIPALS to infer reduced linear models. Therefore, it has often been used to learn mechanical models of robots along trajectories, see Sigaud et al. (2011) for a survey. LWPR has many of the same meta-parameters as RFWR, and thus the same difficulties in tuning these meta-parameters apply.

4.4. Algorithm: XCSF

The XCSF algorithm (Butz et al., 2009) also uses a mixture of linear models. The algorithm manages a population of *classifiers* that contain a *condition* part and a *prediction* part. The condition part is usually defined as a Gaussian function which determines the area of influence of the prediction part. The prediction part is usually a linear model. The population of classifiers is optimized using a genetic algorithm, and the corresponding local linear models are learned using RLS. Once the population of classifiers stabilizes, a condensation operator removes all the unnecessary classifiers.

A key feature of XCSF comes from the insight that the local linear models and corresponding Gaussian functions do not need to be defined over the full input space, an insight which can give rise to much more compact models, see Sigaud et al. (2011) for details.

Additionally, LWPR uses NIPALS instead of RLS whereas XCSF can build local linear models in a predetermined subdomain of the input space. As LWPR, XCSF suffers from the necessity to empirically tune several meta-parameters, though some of them can be set more intuitively in XCSF than in LWPR. See Sigaud et al. (2011) for further discussion and Droniou et al. (2012a) for an empirical comparison.

4.5. Algorithm: Gaussian Mixture Regression

The underlying assumption in Gaussian Mixture Regression (GMR) is that the data in the joint input×output space can be well represented by a set of Gaussians, which is known as a Gaussian Mixture Model (GMM). For an illustration of a GMM consisting of three Gaussians, see Figure 11.

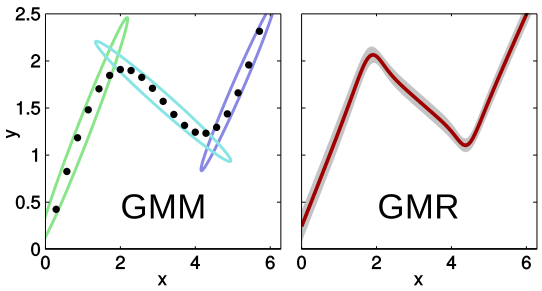

Figure 11: Left: Gaussian Mixture Model (GMM) with 3 Gaussians on the 20 example data points. This model is acquired through unsupervised learning on the joint input×output space. Right: Gaussian Mixture Regression, which yield the mean and variance in the output space Y when conditioning on the input space X.

Training: unsupervised learning. A notable feature of GMR is that the training phase consists of unsupervised learning. It is performed by fitting a Gaussian Mixture Model (GMM) to the data with the Expectation-Maximization (EM) (Dempster et al., 1977) algorithm. k-means clustering is commonly used to provide a first initialization of the centers of the Gaussians. Because EM is an unsupervised learning algorithm, there is no distinction between an input $\mathbf{x}_n$ and a target $\mathbf{y}_n$ example, and they are concatenated into one vector $\mathbf{z}_n = [\mathbf{x}_n^\intercal \ \mathbf{y}_n]^\intercal$. The GMM represents a model of the density of the vectors $\mathbf{z}_n$ as a weighted sum of E Gaussian functions:

$$p(\mathbf{z}_n) = \sum_{e=1}^{E} \pi_e \mathcal{N}(\mathbf{z}_n; \boldsymbol{\mu}_e, \boldsymbol{\Sigma}_e), \text{ with } \sum_{e=1}^{E} \pi_e = 1. \tag{42}$$

The EM algorithm adjusts the priors π_e and the parameters of the Gaussian functions (the means $\boldsymbol{\mu}_e$ and covariance matrices $\boldsymbol{\Sigma}_e$) that define this model, see Ghahramani and Jordan (1993) for details. The only meta-parameter of the training phase is E, the number of Gaussians.

Regression: conditioning on the input. As noted by Ghahramani and Jordan (1993), the learned density can be exploited in several ways, since we can estimate $\overline{\mathbf{y}} = f(\mathbf{x})$, $\overline{\mathbf{x}} = f^{-1}(\mathbf{y})$, or any other relation between two subsets of the elements of the concatenated vector $[\mathbf{x}_n^\intercal \ \mathbf{y}_n]^\intercal$.

In regression, we are interested in predicting $\overline{\mathbf{y}} = E(\mathbf{y}|\mathbf{x})$, the expectation of $\mathbf{y}$ given $\mathbf{x}$. To do so, $\boldsymbol{\mu}_e$ and $\boldsymbol{\Sigma}_e$ can be separated in input and output components as follows

$$\boldsymbol{\mu}_e = [\boldsymbol{\mu}_{e,X}^\intercal, \boldsymbol{\mu}_{e,Y}^\intercal]^\intercal \text{ and } \boldsymbol{\Sigma}_e = \begin{bmatrix} \boldsymbol{\Sigma}_{e,X} & \boldsymbol{\Sigma}_{e,XY} \\ \boldsymbol{\Sigma}_{e,YX} & \boldsymbol{\Sigma}_{e,Y} \end{bmatrix}, \tag{43}$$

For instance, if the dimensionality of the input space X is 2, and that of the output space Y is 1, then $\boldsymbol{\Sigma}_{e,X}$ is a 2×2 sub-matrix, and $\boldsymbol{\Sigma}_{e,Y}$ would be a scalar. The matrix $\boldsymbol{\Sigma}_e$ would then be of size 3×3, being the covariance matrix of the full input $\times$ output space.

Given the decomposition in (43), the expected output $\mathbf{y}$ given an input $\mathbf{x}$ is then computed as

$$\overline{\mathbf{y}} = \sum_{e=1}^{E} h_e(\mathbf{x})(\boldsymbol{\mu}_{e,Y} + \boldsymbol{\Sigma}_{e,YX}\boldsymbol{\Sigma}_{e,X}^{-1}(\mathbf{x} - \boldsymbol{\mu}_{e,X})), \tag{44}$$

with:

$$h_e(\mathbf{x}) = \frac{\pi_e \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_{e,X}, \boldsymbol{\Sigma}_{e,X})}{\sum_{l=1}^{E} \pi_l \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_{l,X}, \boldsymbol{\Sigma}_{l,X})}. \tag{45}$$

As a Bayesian method, an advantage of GMR over other LWR methods is that the variance can also be computed in the estimate of $\mathbf{y}$ with, see (Calinon et al., 2007):

$$var(\overline{\mathbf{y}}) = \sum_{e=1}^{E} h_e(\mathbf{x})^2 \left(\boldsymbol{\Sigma}_{e,Y} - \boldsymbol{\Sigma}_{e,YX}\boldsymbol{\Sigma}_{e,X}^{-1}\boldsymbol{\Sigma}_{e,YX}^\intercal\right). \tag{46}$$

Relationship to the unified model. When we rewrite (44) for the univariate case and set $\mathbf{a}_e^\intercal = \boldsymbol{\Sigma}_{e,YX}\boldsymbol{\Sigma}_{e,X}^{-1}$ and $b_e = \mu_{e,Y} - \boldsymbol{\Sigma}_{e,YX}\boldsymbol{\Sigma}_{e,X}^{-1}\boldsymbol{\mu}_{e,X}$ we get

$$\overline{y} = \sum_{e=1}^{E} \phi_e(\mathbf{x})(\mathbf{a}_e^\intercal \mathbf{x} + b_e), \tag{47}$$

where $h_e(\mathbf{x}) \equiv \phi(\mathbf{x}, \boldsymbol{\theta}_e)$ are normalized Gaussian functions

$$\phi(\mathbf{x}, \boldsymbol{\theta}_e) = \frac{\pi_e g(\mathbf{x}, \mathbf{c}_e, \boldsymbol{\Sigma}_e)}{\sum_{e'=1}^{E} \pi_{e'} g(\mathbf{x}, \mathbf{c}_{e'}, \boldsymbol{\Sigma}_{e'})} \tag{48}$$

as in (35), with the centers, widths and priors stored in the parameter vector $\boldsymbol{\theta}_e = (\mathbf{c}_e, \boldsymbol{\Sigma}_e, \pi_e)$. The model in (47) is equivalent to the unified model in (40). The relationship between the GMM and the unified model is illustrated in Figure 12.

Note that when representing GMR with the unified model, we can no longer compute the variance $var(\overline{\mathbf{y}})$ or the resulting confidence interval. This is because the matrices required to do so ($\boldsymbol{\Sigma}_{e,X}$, $\boldsymbol{\Sigma}_{e,X}$ and $\boldsymbol{\Sigma}_{e,YX}$) have been compiled into $\mathbf{a}_e$ and b_e. Thus, for performing the regression from $\mathbf{x}$ to $\overline{\mathbf{y}}$ the unified model is sufficient to capture the expected output of the GMM, but not to compute $var(\overline{\mathbf{y}})$.

Incremental GMR. Cederborg et al. (2010) present a fast and incremental version of GMR named ILO-GMR. They add to GMR a dedicated structure for storing all the data points so as to quickly access the points that are close to a query point. The idea consists in performing local regression at the query point on demand, using only a small set of close points to build a small GMR model with only 2 or 3 Gaussian features. The resulting algorithm, ILO-GMR, is as accurate as GMR yet performs faster.

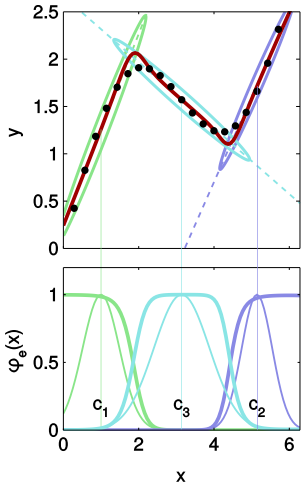

Figure 12: Relationship between GMM and the unified model. The basis functions are the (normalized) projections of the Gaussians in the GMM on the input space. The linear models have slopes $a_e = \Sigma_{e,YX}\Sigma_{e,Y}^{-1}$ and offsets $b_e = \mu_{e,Y} - \Sigma_{e,YX}\Sigma_{e,X}^{-1}\mu_{e,X}$.

4.6. Algorithm: M5 *(Model Tree Regression)*

Model trees are decision trees which do not have class labels in their leaves, but rather linear models (Quinlan, 1992; Wang and Witten, 1997). The classical algorithm for growing model trees is M5 (Quinlan, 1992). In each input dimension, it defines a decision boundary between each pair of adjacent input examples in that dimension. If there are N data points and the input dimensionality is D, there are $D(N-1)$ decision boundaries. Each decision boundary splits the original set S into two subsets T_1 and T_2. From these boundaries, the boundary that minimizes the standard deviation reduction is selected with

$$SDR = std(S) - \sum_i \frac{|T_i|}{|S|} \times std(T_i), \qquad (49)$$

where std denotes the standard deviation and $|...|$ denotes the cardinal of a set. Equation (49) is similar to the information gain criterion used to grow decision trees. This splitting procedure continues recursively until a given halting condition. After growing the tree, it may optionally be pruned to reduce its size. Each of E leaves in the tree is associated with a different disjoint subset $T_{e=1...E}$ of the original training set S, i.e. $S = \bigcup_{e=1}^{E} T_e$ and $\forall e T_e \subset S$.

Within our unified model, the weighting functions of a model tree are not Gaussian, but rather box-car functions, which are 1 if the condition in the leaf holds, and 0 otherwise. A visualization of the model tree function representation for the example data is shown in Figure 13.

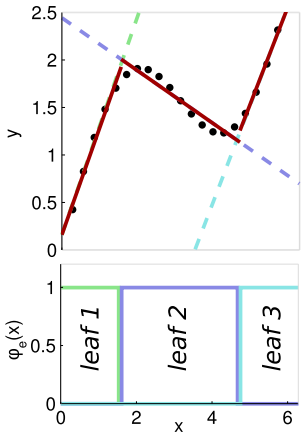

Figure 13: Model tree representation, with linear models (top), and non-overlapping box-car weighting functions (bottom), one for each leaf in the tree.

As Figure 13 illustrates, there may be discontinuities when going from one linear model to the next. This is a consequence of using discontinuous weighting functions, i.e. the box-car functions. To avoid discontinuities, a smoothing procedure is usually applied when querying a model tree (Quinlan, 1992; Wang and Witten, 1997).

One of the main advantages of a model tree is that its function representation is a tree, which can be easily visualized and interpreted. But this advantage holds mainly with trees of limited size, and with a large input space with many irrelevant features (Stulp and Beetz, 2008).

The main meta-parameters for growing model trees with M5 are the minimum number of training examples per leaf (n), a splitting threshold on the ratio of the standard deviation of the entire data set and the standard deviation of the leaf under consideration (σ), and whether the tree should be pruned or not (P). Finally, a smoothing parameter k determines how much the output is smoothed when querying the model tree.

4.7. Summary

A mixture of linear models is used in LWR, RFWR, LWPR, XCSF, GMR, and M5. Thus these algorithms only differ in the way they tune the weighting functions and the linear models, and in the meta-

parameters the user has to provide as an input to the algorithm.

In this family, LWR is at one extreme, as all the parameters of the weighting functions are predetermined by the user and the local linear models are learned with a one batch least squares method, whereas in LWPR and XCSF the number of linear models and their weighting functions are adaptive but constrained through algorithmic metaparameters.

5. Model: Basis Function Network

In Section 3, we described two directions in which standard least squares has been extended. One extension leads to a model that consists of a mixture of linear models; algorithms that yield this model, LWR amongst others, have been discussed in the previous section.

In this section we turn to algorithms that yield a weighted sum of basis functions. This model is a special case of a mixture of linear models with $\mathbf{a} = 0$, as demonstrated in Section 3.3 and repeated below in (50).

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot w_e, \tag{50}$$

In Radial Basis Function Networks (RBFNs), presented in Section 3.2, the basis function $\phi(\mathbf{x}, \boldsymbol{\theta}_e)$ is constrained to be radial, i.e. the output of ϕ depends only on the distance to a center: $\phi(||\mathbf{x} - \mathbf{c}||)$. We refer to the most generic case – when there are no constraints on ϕ – as a *Basis Function Network* (BFN). The different models presented in this section are all BFNs, but with different basis functions: RBFN uses radial basis functions, neural networks use mostly sigmoid or hyperbolic functions, iRFRLS uses cosine functions, KRLS and GPR use kernel functions, and regression trees use box-car functions.

Because the basis function weights in all these regression methods can be found through least squares, in Section 5.1 we first describe three types of design matrices used in the least squares solution. Treating them in a separate section avoids repeating these formulae for each of the algorithms; they are essentially the same for all.

5.1. Least Squares for Basis Function Networks

For all algorithms in this section, least squares regression can be used to learn the weight associated with each basis function. The standard and regularized solutions to the least square problem from Section 2 are repeated in (51) and (52):

$$\mathbf{w}^* = (\mathbf{Z}^\intercal \mathbf{Z})^{-1} \mathbf{Z}^\intercal \mathbf{y} \tag{51}$$

$$\mathbf{w}^* = (\lambda \mathbf{I} + \mathbf{Z}^\intercal \mathbf{Z})^{-1} \mathbf{Z}^\intercal \mathbf{y}. \quad \text{(regularized)} \tag{52}$$

We now describe three forms that $\mathbf{Z}$ may take: the design matrix, the feature matrix, or the Gram matrix.

5.1.1. Design Matrix

In linear least squares, the matrix $\mathbf{Z}$ is of size $N \times D$, i.e. each row contains one input example, with each column representing a different dimension of the input space. Informally, $\mathbf{Z}$ may be considered as containing the 'raw' input data.

$$\mathbf{Z} = \mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D}. \end{bmatrix} \tag{53}$$

5.1.2. Feature Matrix

As discussed in Section 3.2, projecting the input space into a higher dimensional feature space with E basis functions, yields an $N \times E$ feature matrix (one column for each basis function feature) to which Equations (51) or (52) are readily applicable.

$$\mathbf{Z} = \boldsymbol{\Theta}(\mathbf{X}) = \begin{bmatrix} \phi(\mathbf{x}_1, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_1, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_1, \boldsymbol{\theta}_E) \\ \phi(\mathbf{x}_2, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_2, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_2, \boldsymbol{\theta}_E) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\mathbf{x}_N, \boldsymbol{\theta}_1) & \phi(\mathbf{x}_N, \boldsymbol{\theta}_2) & \cdots & \phi(\mathbf{x}_N, \boldsymbol{\theta}_E) \end{bmatrix} \tag{54}$$

5.1.3. Gram Matrix (for Kernel Functions)

A kernel is a special type of basis function which relates two elements of the input space to each other, see Williams and Rasmussen (2006) for their mathematical properties. In this case, $\mathbf{Z} \equiv \mathbf{K}(\mathbf{X}, \mathbf{X})$ is of size $N \times N$, because the kernel is evaluated on all combinations of input examples. This matrix is known as the *Gram matrix*.

$$\mathbf{Z} = \mathbf{K}(\mathbf{X}, \mathbf{X}) = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & k(\mathbf{x}_N, \mathbf{x}_2) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}. \tag{55}$$

Note that the same kernel function k is used for all entries of the Gram matrix. For convenience, we therefore drop the kernel parameters $\boldsymbol{\theta}$.

Because the Gram matrix is symmetrical, (51) can be simplified as

$$\mathbf{w}^* = (\mathbf{K}^\intercal \mathbf{K})^{-1} \mathbf{K}^\intercal \mathbf{y} \tag{56}$$

$$= \mathbf{K}^{-1} \mathbf{y}, \tag{57}$$

where we use $\mathbf{K}$ as an abbreviation for $\mathbf{K}(\mathbf{X}, \mathbf{X})$. The same is possible for the regularized case, if we replace the general $\mathbf{Z}$ matrix with the kernel matrix $\mathbf{K}$ in (52), after some algebra we get

$$\mathbf{w}^* = (\lambda \mathbf{I} + \mathbf{K})^{-1} \mathbf{y}. \tag{58}$$

For more details about this computation, and in particular for a discussion about using the kernel trick (Saunders et al., 1998) in this context, see Williams and Rasmussen (2006).

5.1.4. Preview

Table 2 provides a preview of the algorithms presented in this section, listing which type of basis functions they use, the type of matrix used for linear least squares, and the least squares variant they use for batch regression. This table does not list incremental methods for learning the model parameters.

Algorithm	Features?	Size of $\mathbf{Z}$?	Regularized?
RBFN	RBFs	FM ($E \times N$)	Yes
KRR	kernels	Gram ($N \times N$)	Yes
GPR	kernels	Gram ($N \times N$)	No
iRFRLS	cosine	FM ($E \times N$)	Yes
I-SSGPR	cosine	FM ($E \times N$)	Yes
ELM	sigmoid	FM ($E \times N$)	No

Table 2: Design of all weighted basis function algorithms (FM: feature matrix, RBFs: radial basis functions).

Meta-parameters and model complexity. The table above already gives some insight into the types of meta-parameters these algorithms have. For instance, in KRR and GPR each data point is the center of one kernel. Thus, these centers depend on the training data alone, and must not be set as meta-parameters. When applying linear least squares to RBFNs or in iRFRLS, I-SSGPR or ELM on the other hand, the placement of basis functions is arbitrary, and must thus be chosen in advance.

There are several strategies for choosing the centers of the basis functions: 1) setting them manually; 2) equidistantly spacing in the range of the input instances along each dimension; 3) randomly sampling from the input range. 4) randomly sampling from the input instances; 5) clustering the input examples and using the cluster centers as basis function centers; In the first strategy, the centers $\mathbf{c}_{e=1\ldots E}$ are part of the meta-parameters, and must be set by the user. In the other cases, only E is a meta-parameter, and the centers are determined automatically. Similar strategies exist for cosine basis functions (iRFRLS and I-SSGPR choose the phases randomly) and sigmoid basis functions (ELM chooses the bias randomly).

The choice of the number of basis function E affects the complexity of the model, and thus bias/variance trade-off. KRR and GPR take an extreme position on model complexity, by using all the training data as part of the model, thus letting the "data speak for itself". For this reason, such algorithms are known as *non-parametric regression* methods (Williams and Rasmussen, 2006). Although this article focuses on parametric regression, we have nevertheless included KRR and GPR because of their similarity to RBFNs and other basis function networks presented in this section. The disadvantage of this approach in GPR and KRR is that letting each data point speak requires patience when there is a lot of data; it involves the inversion of a $N \times N$ matrix, which has complexity $\mathcal{O}(N^3)$.

5.2. Algorithm: Regression with Radial Basis Function Networks (revisited)

Before turning to other algorithms, we briefly revisit regression with RBFNs to discuss its algorithmic meta-parameters. In RBFNs, the basis functions ϕ are chosen to be symmetric around some center $\mathbf{c}_e$, which is therefore always a parameter of ϕ, i.e. $\phi(\mathbf{x}, \mathbf{c}_e, \dots)$. Commonly used radial basis functions (RBFs) include the Gaussian (31). Different strategies for choosing these centers were presented in the previous section. Each of these versions yields a slightly different algorithm, and they do not all have the same meta-parameters. And although these algorithmic variations all use the exact same model, the algorithms may tune the values of the model parameters in different ways, as discussed in Section 2.6.

For the relationship between between RBFNs, regularization networks, generalized splines, and di-

mensionality reduction, we refer to the work by Poggio and Girosi (1990).

5.3. Algorithm: Kernel Ridge Regression

In Kernel Ridge Regression (KRR), also called Kernel Regularized Least Squares, the basis functions ϕ are generated from a kernel function $k(\mathbf{x}, \mathbf{x}')$, which takes two vectors from the input space as input. Kernel functions are such that their output is maximal when $\mathbf{x} = \mathbf{x}'$ and decreases as the distance $||\mathbf{x} - \mathbf{x}'||$ increases, which provides a locality property. An example of such a function is the Gaussian kernel.

Using these kernel functions, regression consists in finding the weights $\mathbf{w}$ of the function

$$f(\mathbf{x}) = \sum_{n=1}^{N} w_n \cdot k(\mathbf{x}, \mathbf{x}_n). \tag{59}$$

To perform the regression with this approach, we first compute the Gram matrix in (55), and then perform regularized ridge regression with (58).

How does (59) relate to the unified model in (41)? KRR turns out to be a special case where the number of basis function is identical to the number of data points, and where each data point is the center of a basis function. The corresponding network model is depicted in Figure 14. Thus, while RBFN allows arbitrarily placed basis function centers, KRR centers one basis function around each data point.

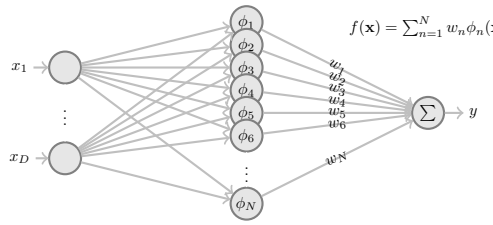

Figure 14: The function model used in KRR as a network.

5.4. Algorithm: Gaussian Process Regression

In Gaussian Process Regression (GPR), the key idea is that the output data $\mathbf{y} = \{y_1, \ldots, y_n\}$, or any subset of the output data, can be thought of as one sample from a multivariate (n-variate) Gaussian function (Williams and Rasmussen, 2006; Ebden, 2008). Often, it is assumed that the mean of this distribution is 0, i.e. the Gaussian Process is represented as

$$\mathbf{y} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}). \tag{60}$$

The covariance matrix $\mathbf{\Sigma}$ is determined by the covariance function or kernel. A typical choice is the Gaussian function

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\tfrac{1}{2}(\mathbf{x} - \mathbf{x}')^\intercal \mathbf{W}^{-1} (\mathbf{x} - \mathbf{x}')\right) \tag{61}$$

$$= \sigma_f^2 g(\mathbf{x}, \mathbf{x}', \mathbf{W}), \tag{62}$$

as for smooth functions, we expect in general higher covariance between points that are closer. Here, σ_f represents signal variance, and $\mathbf{W}$ the covariance matrix of the Gaussian kernel. The Gaussian function g, was previously defined in (31).

For a given covariance function k and N training points $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$ the corresponding Gaussian Process is

$$\mathbf{y} \sim \mathcal{N}(\mathbf{0}, \mathbf{K}(\mathbf{X}, \mathbf{X})), \tag{63}$$

where $\mathbf{K}$ is again the Gram matrix, as in (55).

Predicting y_q for a novel input $\mathbf{x}_q$ is done by assuming that the novel output y_q is also sampled from a multivariate Gaussian with

$$\left[\begin{smallmatrix} \mathbf{y} \\ y_q \end{smallmatrix}\right] \sim \mathcal{N}(\mathbf{0}, \left[\begin{smallmatrix} \mathbf{K}(\mathbf{X},\mathbf{X}) & \mathbf{k}(\mathbf{x}_q,\mathbf{X})^\intercal \\ \mathbf{k}(\mathbf{x}_q,\mathbf{X}) & k(\mathbf{x}_q,\mathbf{x}_q) \end{smallmatrix}\right]), \text{ and} \tag{64}$$

$$\mathbf{k}(\mathbf{x}_q, \mathbf{X}) = [k(\mathbf{x}_q, \mathbf{x}_1), \ldots, k(\mathbf{x}_q, \mathbf{x}_n)]. \tag{65}$$

Conditioning this joint distribution to predict y_q given $\mathbf{X}$, $\mathbf{y}$, and $\mathbf{x}_q$ yields another multivariate Gaussian (Ebden, 2008), which predicts the probability of observing a certain y_q:

$$y_q|\mathbf{X}, \mathbf{y}, \mathbf{x}_q \sim \mathcal{N}(\mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y}, \\ k(\mathbf{x}_q, \mathbf{x}_q) - \mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{k}(\mathbf{x}_q, \mathbf{X})^\intercal). \tag{66}$$

Thus, the best estimate for y_q is the mean, and the uncertainty in y_q is captured by the variance as

$$\overline{y}_q = \mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y} \tag{67}$$

$$var(y_q) = k(\mathbf{x}_q, \mathbf{x}_q) - \mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{k}(\mathbf{x}_q, \mathbf{X})^\intercal. \tag{68}$$

Figure 15 visualizes the mean $\overline{y}_q$ and variance $var(y_q)$ of the Gaussian process over the input domain, given 10 and 20 data points of our example training data. We see that for areas where no training data is available, the mean tends to go towards zero (i.e. to the mean of the Gaussian Process) and

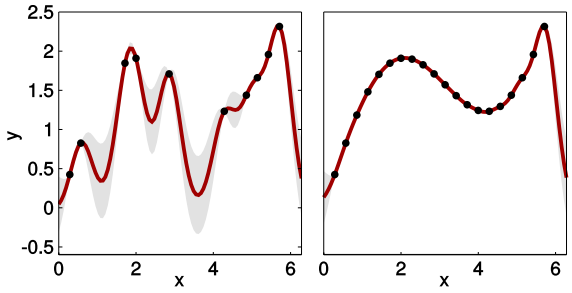

Figure 15: Mean and standard deviation ($\overline{y}_q \pm \sqrt{var(y_q)}$) of the Gaussian process given 10 (left) or 20 (right) example data points.

the variance goes up. When all 20 data points are known, the variance is very low.

The open parameters of a Gaussian Process, e.g. the parameters σ_f^2 and $\mathbf{W}$ in (62), are known as the *hyperparameters*. In our terminology, these correspond to the meta-parameters of the GPR algorithm for regression. Rather than tuning the hyperparameters by hand, it is customary to tune them automatically by minimizing their log likelihood on a training dataset, i.e. $\text{argmin}_\theta \ \log p(\mathbf{y}|\mathbf{X}, \theta)$, where θ contains all the hyperparameters. For a zero-mean Gaussian process with Gaussian kernel as in (61) for instance, θ contains σ_f^2 and $\mathbf{W}$. In this case, the only meta-parameters of the GPR algorithm are those related to the optimization of the hyperparameters of the Gaussian process.

Relation to the Unified Model. When computing the mean $\overline{y}_q$, $\mathbf{K}(\mathbf{X}, \mathbf{X})$ and $\mathbf{y}$ depend only on the training data, not the novel input $\mathbf{x}_q$. Therefore, $\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y}$ can be compacted in one weight vector, which does not depend on the query $\mathbf{x}_q$. This substitution of $\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y}$ with $\mathbf{w}^*$ corresponds exactly to the linear least squares solution in (57). The substitution yields

$$\begin{aligned}
\overline{y}_q &= \mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y} && (69)\\
&= \mathbf{k}(\mathbf{x}_q, \mathbf{X}) \cdot \mathbf{w}^* \quad \text{apply (57)} && (70)\\
&= [k(\mathbf{x}_q, \mathbf{x}_1) \dots k(\mathbf{x}_q, \mathbf{x}_N)] \cdot \mathbf{w}^* && (71)\\
&= \sum_{n=1}^{N} w_n \cdot k(\mathbf{x}_q, \mathbf{x}_n). && (72)
\end{aligned}$$

Thus, we see that the mean of the GPR is the same weighted sum of basis functions as in KRR (see (59)), and the same network representation as in Figure 14 applies. As KRR, GPR may thus be considered as a special case of RBFN, where each data point is the center of one basis function, and each basis function has the same width, determined by the covariance function. Such links have been previously made by Williams and Rasmussen (2006); Williams (1998); Neal (1996). Therefore, GPR is also a special case of the unified model in (41). From this perspective, the only difference between KRR and GPR is that the former uses regularized least squares, and the latter standard least squares (with $\lambda = 0$).

Figure 16 is a repetition of Figure 15, except that we now visualize the basis functions (bottom row) and weighted basis functions (top row), just as was done for RBFN in Figure 7 (D1) and (D2) .

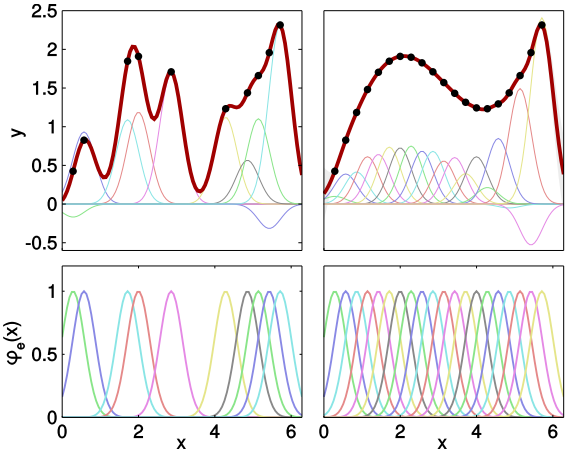

Figure 16: Repetition of Figure 15, this time emphasizing that the mean in GPR is also a weighted sum of basis functions.

As GPR is a Bayesian method, it provides an estimate of the variance $var(y_q)$. However, representing the GPR model with the unified model means that we can no longer compute the variance $var(\overline{y})$. This is because the matrices required to do so $\mathbf{K}(\mathbf{X}, \mathbf{X})$ and $\mathbf{y}$ have been reduced to the vector $\mathbf{w}^*$. Thus, for performing the regression from $\mathbf{x}$ to $\overline{y}$ the unified model is sufficient to capture the GPR model, but to compute $var(\overline{y})$ or the confidence interval, it is not. The same holds for GMR, see Section 4.5.

Non-zero mean function. Prior knowledge about a function may be introduced in the mean of the Gaussian process. For instance, if we know the average outside temperature at a specific location is 20° C, we may rather set the mean of the Gaussian process to $\mu = 20$. The mean itself may depend on the input, so we write $\mu(\mathbf{x})$, which leads to a

Gaussian Process:

$$\begin{bmatrix} \mathbf{y} \\ y_q \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(\mathbf{X}) \\ \mu(\mathbf{x}_q) \end{bmatrix}, \begin{bmatrix} \mathbf{K}(\mathbf{X},\mathbf{X}) & \mathbf{k}(\mathbf{x}_q,\mathbf{X})^\intercal \\ \mathbf{k}(\mathbf{x}_q,\mathbf{X}) & k(\mathbf{x}_q,\mathbf{x}_q) \end{bmatrix}\right), \tag{73}$$

which, when conditioned on $\mathbf{X}$, $\mathbf{y}$, and $\mathbf{x}_q$ yields (the variance estimate stays the same)

$$\overline{y}_q = \mu(\mathbf{x}_q) + \mathbf{k}(\mathbf{x}_q, \mathbf{X})\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}(\mathbf{y} - \mu(\mathbf{X})). \tag{74}$$

Note the similarity for computing the most likely estimate of in Gaussian mixture regression in (44). In the unified model, adding a non-zero mean to the Gaussian process thus means the weights $\mathbf{w}^*$ become $\mathbf{K}(\mathbf{X}, \mathbf{X})^{-1}(\mathbf{y} - \mu(\mathbf{X}))$, and a bias $\mu(\mathbf{x}_q)$ needs to be added.

Sparse and online versions of GPR. Given the cubic computation cost in the number of examples n due to the inversion of $\mathbf{K}(\mathbf{X}, \mathbf{X})$, GPR is often not applicable online. There are two approaches to avoiding the full inversion of $\mathbf{K}(\mathbf{X}, \mathbf{X})$. The *sparse* approach consists in limiting n by adequately choosing the points to remember and forgetting the rest. A good overview of this approach can be found in Quiñonero Candela and Rasmussen (2005). For instance, it has been recently used in *Sparse Online Gaussian Processes* (SOGP), in the context of learning control policies from demonstrations (Grollman and Jenkins, 2008).

The *mixture of experts* approach rather consists in splitting the global domain of the function into smaller regions, as done in the LWR family of methods. Such combination of ideas from both families are taken in Local Gaussian Process Regression (Nguyen-Tuong et al., 2009), see Sigaud et al. (2011) for details, and more recently in (Meier et al., 2014).

5.5. Algorithm: Support Vector Regression (SVR)

The L^1 and L^2 loss functions penalize any residual that is larger than 0. In Support Vector Regression (SVR) a more tolerant approach is taken. Any residual $r_n = y_n - f(\mathbf{x}_n)$ that is smaller than a given threshold $|r_n| < \epsilon$ is not penalized. The region in which the residuals are zero, visualized in Figure 17, is called the *ϵ-insensitive tube*. This leads to the *ϵ-insensitive loss function*

$$L_\epsilon(r) = \begin{cases} 0 & \text{if } r \leq \epsilon \\ |r| - \epsilon & \text{if } r > \epsilon \end{cases} \tag{75}$$

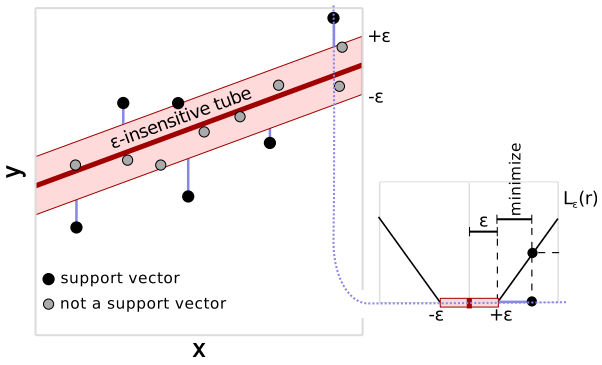

Figure 17: Linear Support Vector Regression. Data points within the ϵ-insensitive tube have zero penalty, and are gray. Those outside are the support vectors, which are black. Adapted from (Smola and Schölkopf, 2004).

This loss function cannot be minimized analytically, and the problem is rather stated as a constrained optimization problem (Smola and Schölkopf, 2004)

$$\text{minimize } \tfrac{1}{2}\|\mathbf{a}\|^2 + C\sum_{n=1}^{N}(\xi_n^+ + \xi_n^-), \tag{76}$$

$$\text{subject to } \begin{cases} r_n & \leq \epsilon + \xi_n^+ \\ -r_n & \leq \epsilon + \xi_n^- \\ \xi_n^+, \xi_n^- & \geq 0 \end{cases} \tag{77}$$

which uses the slack variables[3] ξ_n^+ and ξ_n^-, which represent the distance above or below the ϵ-insensitive tube, cf. Figure 17. Informally, this problem can be formulated as "Try to find small slopes $\mathbf{a}$, such that the distances above ξ_n^+ or below ξ_n^- the tube are minimal; ideally points lie within the tube, because this leads to $\xi_n^+ = 0$ and $\xi_n^- = 0$, and thus zero penalty for that point." C plays the same role as λ in Thikonov regularization; it determines the trade-off between minimizing the residuals and minimizing the norm of $\mathbf{a}$.

This constrained optimization problem is solved by defining a Lagrange function that incorporates both the term to be minimized and the constraints[3]. The final result for linear SVR, in which the under-

[3]The full understanding of the solution requires knowledge of "slack variables" and "Lagrangian multipliers". Explaining these concepts is beyond the scope of this article, and we refer to the tutorial by (Smola and Schölkopf, 2004) for the full derivation.

lying model is $f(\mathbf{x}) = \mathbf{a}^\intercal \mathbf{x} + b$, is:

$$\mathbf{a} = \sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\mathbf{x}_n, \tag{78}$$

where α_n^+ and α_n^- are the Lagrangian multipliers in the Lagrange function for the first two constraints in (77). With this solution, the linear function becomes:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{a}^\intercal \mathbf{x} + b & (79)\\ &= \left(\sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\mathbf{x}_n \right)^\intercal \mathbf{x} + b & (80)\\ &= \sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\langle \mathbf{x}_n, \mathbf{x} \rangle + b. & (81) \end{aligned}$$

The slopes $\mathbf{a}$ are thus represented[4] as a linear combination of the training examples $\mathbf{x}_{n=1\ldots N}$.

Note that the data points inside the ϵ-insensitive tube do not contribute to the computation of (80), because α_n^+ and α_n^- are 0 for these points, and $(0-0)\mathbf{x}_n$ will thus always be 0, independent of the values in the vector $\mathbf{x}_n$. Since only the examples outside the tube contribute to (81), i.e. only the example input vectors *support* it, they are known as *support vectors*.

Using the L^2 norm in KRR and GPR leads to a dense representation, in which all example data points are required to make a prediction, and must be stored in the model parameters. Using the L_ϵ norm leads to a sparse representation, in which only a limited number of data points – the support vectors – must be stored. Thus, only the support vectors are part of the model parameters.

Linear least squares is readily extended to nonlinear regression by projecting the input space into a feature space, as explained in Section 3.2. When applying this approach to SVR by using a basis function $\phi_{k=1\ldots K}(\mathbf{x})$ to project in the feature space (and using $\mathbf{w}$ rather than $\mathbf{a}$ to denote weights instead of slopes), the solution becomes

$$\begin{aligned} w_k &= \sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\phi_k(\mathbf{x}_n), & (82)\\ f(\mathbf{x}) &= \sum_{k=1}^{K} \left(\sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\phi_k(\mathbf{x}_n) \right) \phi_k(\mathbf{x}) + b & (83)\\ &= \sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)\langle \boldsymbol{\phi}(\mathbf{x}_n), \boldsymbol{\phi}(\mathbf{x}) \rangle + b & (84)\\ &= \sum_{n=1}^{N} (\alpha_n^+ - \alpha_n^-)k(\mathbf{x}_n, \mathbf{x}) + b. & (85) \end{aligned}$$

This last step is the "kernel trick", which is based on the following insights: 1) SVR only requires dot products in the feature space $\langle \boldsymbol{\phi}(\mathbf{x}_n), \boldsymbol{\phi}(\mathbf{x}) \rangle$. 2) certain types of kernels $k(\mathbf{x}, \mathbf{x}')$ – see Smola and Schölkopf (2004, Section 2.3) for their properties – correspond to a dot product in a feature space. 3) $\langle \boldsymbol{\phi}(\mathbf{x}_n), \boldsymbol{\phi}(\mathbf{x}) \rangle$ may be replaced with $k(\mathbf{x}, \mathbf{x}')$, which avoids computing the potentially high-dimensional feature vector $\boldsymbol{\phi}(\mathbf{x})$ explicitly.

The representation in (85) is compatible with the unified model in (37) with $E = N$ (every data point $\mathbf{x}_n$ is the center of a basis function $\phi_{n \equiv e}$) and $w_e = (\alpha_e^+ - \alpha_e^-)$. The offset b can be pulled into the weight vector by defining $w_0 = b$ and $k_0(.) = 1$.

The meta-parameters of SVR are the tolerance parameter ϵ and the trade-off parameter C. An additional parameter ν can be introduced to deal with the trade-off between the value of ϵ and model complexity, giving rise to the ν-SVR algorithm (Schölkopf et al., 2000).

*5.6. Algorithm: i*RFRLS

To circumvent the kernel expansion problem present in KRR and GPR, Rahimi and Recht (2007) propose to approximate the Gram matrix $\mathbf{K}$ with a set of random features, giving rise to *Random Features Regularized Least Squares* (RFRLS).

Gijsberts and Metta (2011) present an incremental version of RFRLS, named iRFRLS hereafter and I-RFRR in (Gijsberts and Metta, 2012). Though the authors present their algorithm starting from KRR, from the perspective of this paper the relationship to RBFNs is stronger. Indeed, with respect to RBFNs, the key idea behind iRFRLS is that any function can be approximated arbitrarily well as a weighted sum of cosine functions, as outlined by the theory underlying the Fourier Transform (see

[4] Methods for computing b are presented in (Smola and Schölkopf, 2004).

Droniou et al. (2012b) for a presentation of iRFRLS based on its relationship to Fourier Transform). Thus, instead of using Gaussian kernels as in RBFNs, $f(\mathbf{x})$ is as a set of E cosine functions

$$f(\mathbf{x}) = \sum_{e=1}^{E} w_e \cdot \phi(\mathbf{x}, \boldsymbol{\omega}_e, \psi_e) \tag{86}$$

$$= \sum_{e=1}^{E} w_e \cdot \cos(\boldsymbol{\omega}_e^\intercal \mathbf{x} + \psi_e). \tag{87}$$

The basis functions are defined by randomly drawing their multidimensional frequency $\boldsymbol{\omega}$ and phase ψ using $\boldsymbol{\omega} \sim \mathcal{N}(0, 2\gamma I)$ and $\psi \sim \mathcal{U}(0, 2\pi)$.

As in all other RBFN-like methods, the main meta-parameter in iRFRLS is the number of features E, that is easy to tune considering a trade-off between the model accuracy and the time complexity. The other two meta-parameters of iRFRLS are λ, the regularization parameter and γ, the variance over the period of the random cosine features. Experiments on several databases confirm that RFRLS and iRFRLS can be set close to KRR with a small enough processing time. Indeed, iRFRLS shows interesting practical performances (Droniou et al., 2012a) and its computation time is independent of the learning data.

To summarize, iRFRLS is a basis function network approach that uses randomly sampled features, as ELM and certain variants of RBFN do. But it uses cosine features instead of radial or sigmoid features.

5.7. Algorithm: I-SSGPR

In order to approximate a function, KRR uses (59) whereas GPR uses (67). A quick way to describe I-SSGPR is to say that it applies to GPR the same idea that iRFRLS applies to KRR: instead of using one kernel per data point, it draws a collection of random cosine feature functions and then it solves the resulting – non-regularized – linear least square problem into the projected domain. Actually, I-SSGPR also inherits from the GPR framework the capability to optimize most of meta-parameters using the marginal likelihood (Williams and Rasmussen, 2006), so the only meta-parameter that remains to be tuned is E.

5.8. Algorithm: CART *(Regression Trees)*

Regression trees (Breiman et al., 1984) are a special case of model trees for which the slopes of the linear model are zero ($\mathbf{a} = 0$), i.e. the leaf contains only one continuous value (b). Thus model trees and regression trees follow our classification between a mixture of linear models (model trees), and a weighted sum of basis functions (regression trees), the latter being a special case of the former with degenerate linear models with $\mathbf{a} = 0$.

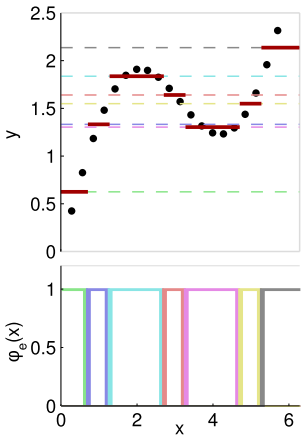

Figure 18: Regression tree model, with degenerate linear models (top), and box-car basis functions (bottom).

Because model trees and regression trees are grown using very similar algorithms (M5 and "CART learning", respectively), and have essentially the same advantages and disadvantages, we refer to Section 4.6 for the full discussion.

Orr et al. (2000) use regression trees to initialize a radial basis function network, thus highlighting the similarity between these two representations.

*5.9. Algorithm: Extreme Learning Machine (*ELM*)*

The Extreme Learning Machine (ELM) is a batch regression algorithm for training the weights of a Single-Hidden Layer Feedforward Network (SLFN) (Huang et al., 2006b, 2011). An SLFN is an Artificial Neural Network (ANN) that has one input layer, one hidden layer, and one output layer, as depicted in Figure 19. The representation[5] of an

[5]We follow the convention by Bishop (1995), where w_{ji} denotes the weight from layer i to layer j. For consistency with the rest of the paper, we use d for the input layer, e for the hidden layer, and use only one output node for ease of presentation. Furthermore, we use different symbols for the weights in the different layers (v for input→hidden, w for hidden→output) to facilitate the explanation of applying least squares to learning $\mathbf{w}$.

SLFN is

$$f(\mathbf{x}) = \sum_{e=1}^{E} w_e \psi \left(\sum_{d=1}^{D} v_{ed} x_d \right), \tag{88}$$

$$= \sum_{e=1}^{E} w_e \psi \left(z_e \right), \tag{89}$$

where $z_e = \sum_{d=1}^{D} v_{ed} x_d$ is the input to the activation function ψ, which is often chosen to be sigmoid ($\psi(z_e) = (1+e^{-z_e})^{-1}$) or hyperbolic ($\psi(z_e) = \tanh(z_e)$). The model is easily extended to have multiple output nodes (Bishop, 1995).

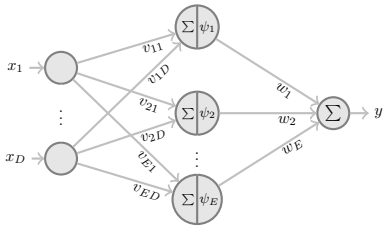

Figure 19: Single-Hidden Layer Feedforward Network with D inputs and E hidden nodes. A bias may be introduced by adding an input neuron with input 1, but it has been left out for clarity.

The ELM algorithm for determining the weights of the SLFN is essentially the same as when using RBFNs for regression, i.e. they both perform batch linear least squares on a projected feature space. In the ELM approach, the basis function parameters $\mathbf{v}_{e=1\ldots E}$ are first set randomly, and are then used to generate the feature matrix $\mathbf{\Theta}(\mathbf{X})$ (Huang et al., 2006b). Least squares (39) is then used to determine the parameters $\mathbf{w}$.

Some extensions to ELMs include growing the number of hidden nodes incrementally (Huang et al., 2006a) – whereby hidden nodes may be preselected from a random set (Huang and Chen, 2008) – or extending them to the complex domain (Huang et al., 2008).

The SLFN model (88) used in ELMs is readily made compatible with the unified model by pulling the weights from the input layer to the output layer into the activation function, i.e.

$$f(\mathbf{x}) = \sum_{e=1}^{E} w_e \phi(\mathbf{x}, \mathbf{v}_e). \tag{90}$$

Thus, from the model-centric perspective taken in this paper, a SLFN is a special case of a basis function network[6] (and thus the unified model), where the basis function must have the form

$$\phi(\mathbf{x}, \mathbf{v}_e) = \psi \left(\sum_{d=1}^{D} v_{ed} x_d \right) = \psi \left(\langle \mathbf{v}_e, \mathbf{x} \rangle \right), \tag{91}$$

where $\langle \mathbf{v}_e, \mathbf{x} \rangle$ denotes the inner product between $\mathbf{v}_e$ and $\mathbf{x}$.

5.10. Algorithm: Backpropagation

An incremental approach to optimizing the connection weights in an artificial neural network is the backward propagation of errors (Werbos, 1974), or simply *backpropagation*. This algorithm first propagates an input $\mathbf{x}_n$ *forwards* through the network (from the input to the output layer), and computes the error $\frac{1}{2}(y_n - f(\mathbf{x}_n))^2$ at the output node. It then propagates this error *backwards* through the network, and updates the weights in each layer through gradient descent. Although backpropagation is usually associated with artificial neural networks that use sigmoid or hyperbolic activation functions, the algorithm is also applicable to radial basis function networks (Poggio and Girosi, 1990; Schwenker et al., 2001a).

In Figure 20, we depict one neuron of an ANN, agnostic of whether it is in a hidden or an output layer. The local rule for updating one weight of one such neuron is:

$$w_{ji} \leftarrow w_{ji} + \alpha \delta_j u_i \tag{92}$$

where δ_j is the error between the actual output of the neuron and the target output, u_i is the input from neuron i to neuron j, and α is the learning rate. This rule is acquired[7] by computing the partial derivative of the error of the neuron w.r.t. the weight $\frac{\partial E}{\partial w_{ji}}$.

If the node is an output node, the error δ_j depends on the residual $r_n = y_n - f(\mathbf{x})$, i.e.

$$\delta_j = \psi_j'(z_j)(y_n - f(\mathbf{x}_n)). \tag{93}$$

If the node is a hidden node, δ_j is defined in terms of errors δ_k of neurons in the subsequent layer, i.e.

$$\delta_j = \psi_j'(z_j) \sum_{k=1}^{K} \delta_k w_{kj}. \tag{94}$$

[6] The relationship between *radial* basis function networks and *multi-layer* feedforward networks is further discussed in Section 5.10.4.

[7] For a full derivation of the backpropagation rules, we refer to (Bishop, 1995). Here, we provide only a brief summary of the resulting rules.

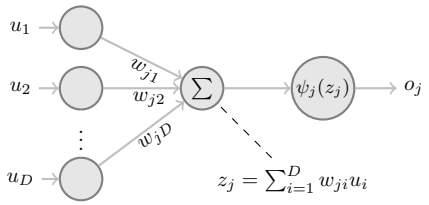

Figure 20: Detailed view of one node in a artificial neural network. If the neuron is in a hidden layer, the output of the neuron o_j will be the input for another neuron.

This recursive rule for computing δ is what propagates errors from the back (where the errors correspond to the residuals at the output neuron) to the front of the network (the input layer).

So far, we have presented backpropagation as an incremental algorithm, i.e. it is able to update the weights incrementally with (92), one training example at a time. However, a batch version exists also, which updates the weights at once, given all the available data:

$$w_{ji} \leftarrow w_{ji} + \alpha \sum_{n=1}^{N} \delta_{n,j} u_{n,i} \quad \text{(batch version).} \tag{95}$$

In *mini-batch* learning, the weights are updated with (95), but using sub-batches of size $K < N$ (Wilson and Martinez, 2003). Incremental and standard batch training can thus be seen as special cases with $K = 1$ and $K = N$ respectively. Wilson and Martinez (2003) show that using batch or mini-batch for backpropagation on a speech recognition task provides no advantage over incremental learning in terms of learning accuracy or speed.

5.10.1. Backpropagation for SLFN*s*

In the SLFN depicted in Figure 19, there is only one hidden layer, and the output node does not have an activation function (it is identity, and thus $\psi'(z_e) = 1$). When applying the local update rule (92) to update the weight w_e, we have $\delta = y_n - f(\mathbf{x}_n)$ from (93), and $u_e = \sum_{d=1}^{D} v_{ed} x_d$, which results in

$$w_e \leftarrow w_e + \alpha(y_n - f(\mathbf{x})) \sum_{d=1}^{D} v_{ed} x_d. \tag{96}$$

If we keep the weights $\mathbf{v}_{e=1\ldots E}$ fixed, and only the weights to the output are thus to be trained, (96) is an incremental gradient-based alternative to the batch least squares used in ELM.

For the weights from the input layer to the hidden layer, $\delta_d = \psi'(z_d)\delta_e w_e$ and $u_d = x_d$, which yields the update rule

$$v_{ed} \leftarrow v_{ed} + \alpha \psi'(z_d) \delta_e w_e x_d. \tag{97}$$

Comparison of ELM *and Backpropagation.* Huang et al. (2011) show that training ELMs with random features and least squares is much faster than with backpropagation. It has lower computational cost than using gradient back-propagation, despite the cubic cost of computing the inverse $(\mathbf{X}^\intercal \mathbf{X})^{-1}$.

Rahimi and Recht (2008) analyze the advantages of using random features instead of backpropagation. They show that, although being much easier to obtain, random features can generate close to optimal solutions if numerous enough. This analysis is also at the heart of the design of iRFRLS and I-SSGPR (see Sections 5.6 and 5.7). A complementary analysis comes from Widrow et al. (2013), who examine under which conditions training a network benefits from backpropagation or not. Schwenker et al. (2001a) use a combination of both least squares batch learning and incremental backpropagation.

References to empirical comparisons between ELM, backpropagation, and other algorithms are given in Section 6.2.

*5.10.2. Backpropagation for Multi-Layer Feedforward Networks (*MLFF*s)*

The network visualized in Figure 19 is easily extended to incorporate more than one hidden layer. Following the notation by Poggio and Girosi (1990), the function representation of such multi-layer feedforward networks (MLFFs) is

$$f(\mathbf{x}) = \sum_{e=1}^{E} w_e \psi \left(\sum_{d=1}^{D} v_{ed} \psi \left(\ldots \psi \left(\sum_{c=1}^{C} u_{dc} x_c \right) \ldots \right) \right). \tag{98}$$

An advantage of the backpropagation algorithm is that it optimizes *all* of the weights in the network from output to input with the generic update rule (92), and thus readily extends to multi-layer networks. In the multivariate, high-dimensional case, Friedman and Stuetzle (1981) propose an advanced gradient descent algorithm for networks with several layers. In contrast, the least squares approach used in ELMs can only be used to tune the weights from the final hidden layer to the output layer, because these are the only weights which are linear with respect to the output.

Deep Neural Networks (DNN) (Bengio, 2009) are networks, typically feedforward, that have many hidden layers. This allows more abstract features to be represented in the hidden layers. Gradient backpropagation is also used for tuning the weights of the network, though a first stage of unsupervised learning is commonly used to preshape those features (Schmidhuber, 2014; Bengio et al., 2013), to avoid the vanishing gradient problem (Hochreiter et al., 2001). DNNs are more commonly used for classification, where they outperform their shallow counterpart. Whether this superiority extends to regression is still an open question. Even for classification, understanding the reasons for the superiority of DNNs is an active topic of investigation. Researchers who address these questions tend to use elementary basis functions that are piecewise linear rather than nonlinear, because it provides a better grip on their composition (Pascanu et al., 2013). These investigations are too recent to be properly covered in this review, so the reader is referred to Pascanu et al. (2013); Alain and Bengio (2013); Dauphin et al. (2014) for more references to this quickly evolving domain.

Our unified model can in principle be used to represent a DNN, if we consider all the layers up to the final layer of a DNN to be the implementation of a rather complex basis function (consisting of weighted compositions of the basis functions in preceding layers). However, this shallow perspective does not capture the dynamics of learning the weights in preceding layers through backpropagation, nor does it provide insight in the relations to other types of regression algorithms.

*5.10.3. Backpropagation for Recurrent Neural Networks (*RNN*s)*

In feedforward networks, only connections to nodes in subsequent layers are allowed. *Recurrent* neural networks do not have this constraint, and connections to nodes in previous layers or nodes within a layer are possible. This leads RNNs to have an internal state, which changes over time even if the input does not change. *Backpropagation through time* is a generalization of backpropagation, enabling this gradient-based method to be applied to RNNs (Werbos, 1988). Due to the cycles in RNNs, and the internal state that arises from them, RNNs cannot be captured by the unified model.

5.10.4. Comparison of RBFN*s and* MLFF*s*

Bishop (1995, Section 5.8) summarizes some of the differences between RBFNs and MLFFs. RBFNs use radial basis functions, which take the distance to a center as an input. Each basis function is thus a localized representation, and only a few basis functions are active for a given input. In contrast, an MLFF is a distributed representation, because the sigmoid function – which splits the input space into an active and inactive spaces – activates many of the hidden units. Furthermore, RBFNs typically have only one layer of hidden units, whereas MLFFs often have many hidden layers (Bishop, 1995). Finally, RBFNs weights are typically trained in two phases – randomly setting or preshaping features and then performing batch least squares – although backpropagation has been applied also (Schwenker et al., 2001a; Poggio and Girosi, 1990). On the other hand, MLFFs are typically trained with backpropagation, because least squares can only be applied to the last layer of MLFFs. Dorffner (1994) discusses these differences in greater details, before introducing a more general class of conic basis functions that encompasses radial basis functions and sigmoid activation functions. The paper also discusses the interest of combining both families of functions in the same approximator, as in (Cohen and Intrator, 2002).

Apart from standard activation functions such as Gaussian, hyperbolic or sigmoidal functions, many other families of basis functions can be used. One can even evolve the activation function of neurons with evolutionary techniques, sometimes leading to higher performance than sigmoidal activation functions (Augusteijn and Harrington, 2004). However, as noted by Dorffner (1994), finding the right class of activation functions might be very problem dependent or give rise to the curse of dimensionality. One can also use neural units whose activation function is of higher order than a sigmoidal function applied to a weighted sum of inputs. The representational power and learning complexity of several instances of this family of networks, such as product unit networks, is studied in (Schmitt, 2002).

5.11. Summary

All the algorithms listed in this section use the same function model

$$f(\mathbf{x}) = \sum_{e=1}^{E} \phi(\mathbf{x}, \boldsymbol{\theta}_e) \cdot w_e, \tag{99}$$

which is a special case of the unified model in (41) with $\mathbf{a} = \mathbf{0}$. However, different algorithms use different instantiations of (99), as listed in Table 2.

In KRR and GPR, the number of basis functions is taken to be N whereas in all other algorithms this is an arbitrary parameter E. Additionally, in GPR and KRR, the basis functions are centered on the data points whereas in RBFNs, they can be either regularly placed to pave the input space or drawn randomly. In all models but ANNs, iRFRLS and I-SSGPR, it is standard to use Gaussian basis functions (that combine the kernel and radial basis function properties), whereas iRFRLS and I-SSGPR use cosine basis functions and ANNs more often use sigmoids or hyperbolic functions.

Some types of ANNs are special cases of the unified model (SLFN) and some are not (RNNs). The backpropagation algorithms is applicable to all ANNs, whereas least squares is only applicable to SLFNs. The latter case, known as the ELM, is very similar to training RBFNs. For further connections between these methods, the reader is referred to Smola and Schölkopf (2004), who examine the relationship between SVR and ANNs.

6. The Unified Model Revisited

All the algorithms discussed in this paper use the same generic function model in (41), but different algorithms use special cases of this generic model. In this section, we reconsider the relationships between the two main classes of models. Furthermore, we list, for the different algorithms, which model parameters must be pre-specified by the user, and which are determined by the algorithm itself.

6.1. Classification based on Special Cases of the Unified Model

Figure 21 presents a classification of the regression algorithms presented in this paper, based on what type of special case of the unified model they use. Note that this perspective considers only the model used to represent the function, and treats the algorithmic process as a black-box. For instance, least squares and weighted least squares output the exact same representation (one linear model), even if they use different methods to compute it, and may thus yield different parameterizations of this model.

This model-based perspective on the algorithm leads to several observations, some straightforward, and some rather intriguing.

- Least squares is special case of weighted least squares, with $\mathbf{w} = 1$
- Weighted least squares is a special case of Locally Weighted Regression with $E = 1$
- Gaussian process regression, which uses standard least squares, is the Bayesian interpretation of KRR, which rather uses regularized least squares.
- The model used in Gaussian process regression is a radial basis function network with a potentially infinite number of hidden basis functions (Williams, 1998; Neal, 1996).
- iRFRLS is a basis function network that uses (non-radial) randomly generated cosine basis function. Its excellent performance in practice raises interesting questions, because iRFRLS differs from all other algorithms in this paper but artificial neural networks in two important ways: it uses global (cosine) rather than local (radial) basis functions, and it generates their parameters randomly, rather than from the data.
- regression trees are to RBFNs (both using a weighted sum of basis functions) what model trees are to LWR (both using a mixture of linear models), except that the trees use box-car basis functions. Thus, even though regression and model trees are learned using the same algorithm, the underlying model they learn belongs to a different class.

6.2. Meta-Parameters and Empirical Comparisons

Regression algorithms differ in the number of meta-parameters that need to be set. For instance, for LWR, the number of basis functions, as well as their centers and widths, must be specified by the user. For GMR, GPR and I-SSGPR, on the other hand, only the number of basis functions must be set. For completeness, Table 3 summarizes the meta-parameters for the different algorithms. The first column lists the regression algorithms, and the second column the underlying algorithm used for the linear model estimation(s). Column 3-5 lists the model-parameters related to the number, position, and size of the basis functions. Model-parameters that must be pre-specified by the user – and are thus also meta-parameters of the algorithm – are highlighted in gray. Model-parameters that are automatically determined by the algorithm are left white. For example, in ELM the number of basis

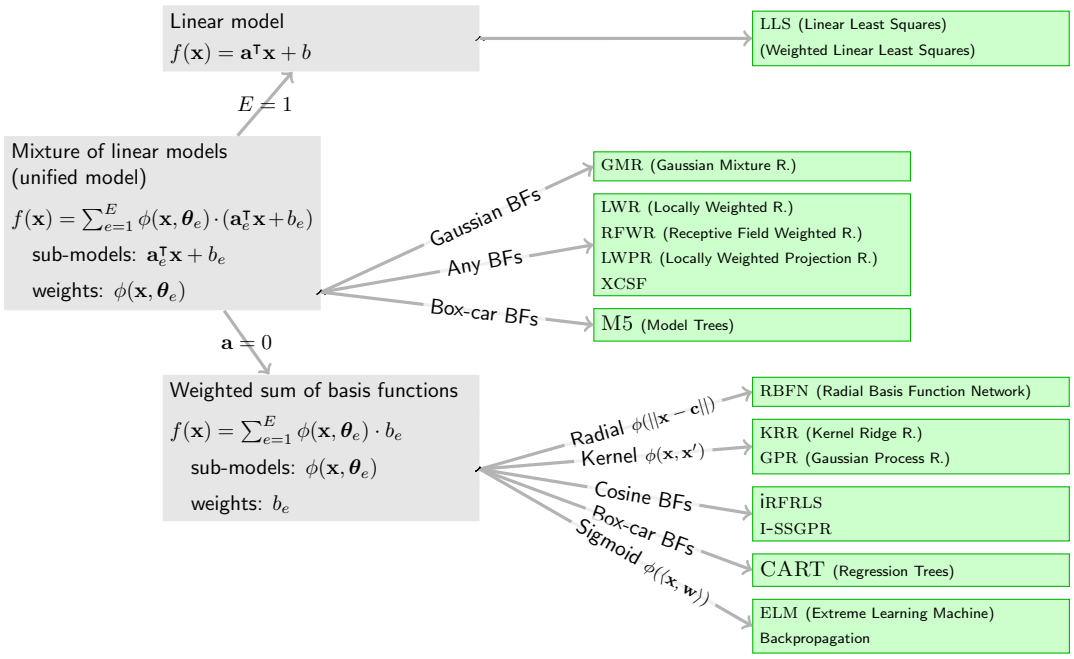

Figure 21: Classification of regression algorithms, based only on the model used to represent the underlying function.

functions is a meta-parameter that must be specified by the user (gray cell), whereas the centers of the basis functions are determined automatically through random sampling (white cell). The final column lists any further meta-parameters which are used for training, but not stored in the model parameters.

The settings of these meta-parameters can greatly influence the result of fitting, and they often depend on properties of the data. Exhaustive empirical comparisons are beyond the scope of this paper; our aim is to show that a wide variety of algorithms use different instances of the same unified model. In Table 4, we provide references to papers in which comparisons between the algorithms presented in this article are made.

7. Conclusion

In this article, we have described two main classes of models that regression algorithms use – a weighted sum of basis functions and a mixture of linear models – and shown that the former is a special case of the latter. Therefore, rather than viewing every algorithm as using a different model, we see them as variations on a common theme, whereby these variations include the use of different basis functions, or using degenerate linear models with $\mathbf{a} = 0$. This view thus forms a strong bridge between seemingly disparate regression algorithms in the field of machine learning, among which many ANN models.

Going from a weighted sum of basis functions and a mixture of linear models requires a shift of interpretation. In the former, the sub-models are basis functions, and the weights are (degenerate) lines. But in a mixture of linear models, the sub-models are linear models, and the weights are determined by the basis functions. Due to this shift in interpretation, it may not be immediately obvious that the two models are the same.

A further insight is that, in a mixture of linear models, both the basis functions $\phi(\mathbf{x}_n, \boldsymbol{\theta}_e)$ and the linear model $\mathbf{a}_e^\intercal \mathbf{x} + b_e$ depend on $\mathbf{x}$. Since we cannot regress against both at the same time. This is why LWR must perform multiple weighted linear regressions. By contrast, with a sum of weighted basis functions, the weights are linear with respect to the output of the basis functions, and can thus be learned in one least squares regression.

What are the implications of having a unified model for all these algorithms? First of all, it allows us to clearly separate questions about models from those about algorithms. As to the models, is it better to use global (e.g. sigmoid, cosine) or local (e.g. radial) features? When is it better to assume local

Algorithm	Linear model estim.	Number of BFs (E)	Position of BFs ($\mathbf{c}_e$)	Size of BFs ($\mathbf{\Sigma}_e$)	Algorithmic parameters
Model: Mixture of Linear Models					
LWR	LS	fixed	fixed	fixed	
RFWR	RLS	adap.	indiv.	adap.	many
LWPR	NIPALS	adap.	indiv.	adap.	many
XCSF	RLS	adap.	adap.	adap.	many
GMR	EM	fixed	adap.	adap.	
M5	LS	adap.	adap.	adap.	n, σ, P, k
Model: Weighted Sum of Basis Functions					
RBFN	LS	fixed	fixed	fixed	
KRR	(RG)LS	indiv.	indiv.	fixed	λ
GPR	LS	indiv.	indiv.	fixed	$k(.)$
SVR	CO	adap.	indiv.	fixed	ϵ
iRFRLS	RLS	fixed	rand.	rand.	γ, λ
I-SSGPR	RLS	fixed	rand.	rand.	
CART	LS	adap.	adap.	adap.	n, σ, P, k
ELM	LS	fixed	rand.	rand.	
BPROP	N/A	fixed	adap.	adap.	α

Table 3: Gray: meta-parameters of the algorithms (partial model specifications in columns 3-5 and further algorithmic parameters in column 6) which the user must set. White: model parameters that the algorithm determines itself. Abbreviations: adap.=adapted, indiv.=from individual training examples, rand.=randomly sampled, CO=constrained optimization.

linearity – and thus use a model based on locally weighted lines – or use weighted basis functions? Which types of models are easier to optimize in online learning, e.g. through backpropagation or reinforcement learning? What is the effect of mixing different types of basis functions (Augusteijn and Harrington, 2004)? Orthogonal question related only to algorithms are: How should features be generated (Rahimi and Recht, 2008) – should they be randomly generated, preshaped through unsupervised learning, or optimized through constrained optimization or backpropagation? In which context should we use which norm? This work provides a conceptual framework in which to address these questions more clearly.

From our perspective, research topics whose full potential has not yet been explored include

- using weighted linear models with global features for weighting,
- deriving a locally weighted regression variant of Support Vector Regression based on its specific norm, which does not seem to exist,
- mixing initial batch learning with subsequent incremental learning, as done by (Schwenker et al., 2001b),
- and investigating which types of regression algorithms yield models instances that are easily optimized in a reinforcement learning context (Munzer et al., 2014).

This latter topic is highly relevant to robotics, where skill learning is often split in two phases: 1) imitation: initializing a model of human motion from observations of human movement (Calinon, 2009; Stulp et al., 2012) through offline batch regression 2) reinforcement learning: optimize the model (incrementally or with mini-batch) with respect to a cost function through trial and error exploration (Stulp et al., 2012; Marin et al., 2011).

A very pragmatic advantage of a unified model arose whilst implementing several of the algorithms presented here. The classes representing the models of the different algorithms became so similar that it was an obvious step to implement a routine for casting them all to a unified model, which greatly simplified the code, and made it easier to exchange algorithms. This is not merely a low-level implementation advantage though. On a more abstract level, such modularity is an important pre-condition for realizing the vision of model-based machine learning (Bishop, 2013).

Reference	**Algorithms compared**
(Atkeson and Schaal, 1995)	LWR, MLFFS
(Schaal and Atkeson, 1997)	RFWR, MLFFS, LS
(Vijayakumar and Schaal, 2000)	LWPR, RFWR
(Schwenker et al., 2001a)	ELM, BPROP, SVR
(Williams and Rasmussen, 2006)	LWPR, GPR, LS
(Grollman and Jenkins, 2008)	SOGP, LWPR
(Nguyen-Tuong et al., 2009)	LGP, LWPR, SVR, GPR
(Cederborg et al., 2010)	LGP, LWPR, GMR, SVR, GPR
(Lammert et al., 2010)	LWR, MLFFS
(Huang et al., 2011)	ELM, BPROP, SVR
(Gijsberts and Metta, 2012)	GPR, LWPR, I-SSGPR
(Droniou et al., 2012a)	XCSF, LWPR, IRFRLS
(Munzer et al., 2014)	LWR, GMR, RBFN, IRFRLS

Table 4: List of articles, and the algorithms between which they make empirical comparisons.

Acknowledgments

This work was partially supported by the European Commission, within the CoDyCo project (FP7-ICT-2011-9, No.600716), and has received funding from the European Unions Horizon 2020

research and innovation programme under grant agreement No 640891. We thank Vincent Padois, Nicolas Perrin, Ryan Lober and Mathieu Lefort for proofreading earlier versions of this article.

References

Alain, G., Bengio, Y., 2013. What Regularized Auto-Encoders Learn from the Data Generating Distribution. arXiv 1211.4246v4.

Atkeson, C. G., Schaal, S., 1995. Memory-based neural networks for robot learning. Neurocomputing 9 (3), 243–269.

Augusteijn, M. F., Harrington, T. P., 2004. Evolving transfer functions for artificial neural networks. Neural Computing & Applications 13 (1), 38–46.

Bengio, Y., 2009. Learning deep architectures for AI. Foundations and Trends in Machine Learning 2 (1), 1–127.

Bengio, Y., Courville, A., Vincent, P., 2013. Representation learning: A review and new perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence 35 (8), 1798–1828.

Bishop, C. M., 1995. Neural Networks for Pattern Recognition. Oxford University Press.

Bishop, C. M., 2013. Model-based machine learning. Philosophical Transactions of the Royal Society A 371, 1–17.

Bishop, C. M., et al., 2006. Pattern recognition and machine learning. Vol. 1. springer New York.

Bloomfield, P., Steiger, W., 1980. Least absolute deviations curve-fitting. SIAM Journal on Scientific Computing 1 (2), 290–301.

Breiman, L., Friedman, J., Olshen, R., Stone, C., 1984. Classification and Regression Trees. Wadsworth and Brooks, Monterey, CA.

Butz, M. V., Herbort, O., 2008. Context-dependent predictions and cognitive arm control with XCSF. In: Conference on Genetic and Evolutionary Computation. ACM, pp. 1357–1364.

Butz, M. V., Pedersen, G. K. M., Stalph, P. O., 2009. Learning sensorimotor control structures with XCSF: redundancy exploitation and dynamic control. In: Conference on Genetic and Evolutionary Computation. ACM, pp. 1171–1178.

Calinon, S., 2009. Robot programming by demonstration. EPFL/CRC Press.

Calinon, S., Guenter, F., Billard, A., 2007. On learning, representing, and generalizing a task in a humanoid robot. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 37 (2), 286–298.

Cederborg, T., Li, M., Baranes, A., Oudeyer, P.-Y., 2010. Incremental local online Gaussian mixture regression for imitation learning of multiple tasks. In: IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 267–274.

Cohen, S., Intrator, N., 2002. A hybrid projection-based and radial basis function architecture: initial values and global optimisation. Pattern Analysis & Applications 5 (2), 113–120.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio, Y., 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in Neural Information Processing Systems. pp. 2933–2941.

Dempster, A. P., Laird, N. M., Rubin, D. B., 1977. Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society. Series B (Methodological), 1–38.

Dorffner, G., 1994. A unified framework for MLPs and RBFNs: Introducing conic section function networks. Cybernetics and Systems 25 (4), 511–554.

Droniou, A., Ivaldi, S., Padois, V., Sigaud, O., 2012a. Autonomous online learning of velocity kinematics on the icub: A comparative study. In: Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on. pp. 3577–3582.

Droniou, A., Ivaldi, S., Stalph, P., Butz, M., Sigaud, O., 2012b. Learning velocity kinematics: Experimental comparison of on-line regression algorithms. In: Proceedings Robotica. pp. 15–20.

Ebden, M., 2008. Gaussian processes for regression: A quick introduction. Tech. rep., Department on Engineering Science, University of Oxford.

Fisher, R., 1925. Statistical methods for research workers. Oliver & Boyd.

Friedman, J. H., Stuetzle, W., 1981. Projection pursuit regression. Journal of the American statistical Association 76 (376), 817–823.

Geladi, P., Kowalski, B., 1986. Partial least squares regression: A tutorial. Analytica Chimica Acta 185, 1–17.

Ghahramani, Z., Jordan, M. I., 1993. Supervised learning from incomplete data via an EM approach. In: Advances in Neural Information Processing Systems 6. pp. 120–127.

Gijsberts, A., Metta, G., 2011. Incremental learning of robot dynamics using random features. In: IEEE International Conference on Robotics and Automation. pp. 951–956.

Gijsberts, A., Metta, G., 2012. Real-time model learning using incremental sparse spectrum gaussian process regression. Neural Networks.

Grollman, D., Jenkins, O. C., 2008. Sparse incremental learning for interactive robot control policy estimation. In: IEEE International Conference on Robotics and Automation. pp. 3315–3320.

Hersch, M., Guenter, F., Calinon, S., Billard, A., 2008. Dynamical system modulation for robot learning via kinesthetic demonstrations. IEEE Transactions on Robotics 24 (6), 1463–1467.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: Kremer, Kolen (Eds.), A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press.

Huang, G.-B., Chen, L., 2008. Enhanced random search based incremental extreme learning machine. Neurocomputing 71 (16-18), 3460–3468.

Huang, G.-B., Chen, L., Siew, C. K., 2006a. Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Transactions on Neural Networks 17 (4), 879–892.

Huang, G.-B., Li, M.-B., Chen, L., Siew, C. K., 2008. Incremental extreme learning machine with fully complex hidden nodes. Neurocomputing 71 (4-6), 576–583.

Huang, G.-B., Wang, D. H., Lan, Y., 2011. Extreme learning machines: a survey. International Journal of Machine Learning and Cybernetics 2 (2), 107–122.

Huang, G.-B., Zhu, Q.-Y., Siew, C.-K., 2006b. Extreme learning machine: theory and applications. Neurocomputing 70 (1), 489–501.

Ijspeert, A. J., Nakanishi, J., Hoffmann, H., Pastor, P.,

Schaal, S., 2013. Dynamical movement primitives: learning attractor models for motor behaviors. Neural computation 25 (2), 328–373.

Lammert, A. C., Goldstein, L., Iskarous, K., 2010. Locally-weighted regression for estimating the forward kinematics of a geometric vocal tract model. In: INTERSPEECH. pp. 1604–1607.

Marin, D., Decock, J., Rigoux, L., Sigaud, O., 2011. Learning cost-efficient control policies with xcsf: generalization capabilities and further improvement. In: Proceedings of the 13th annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1235–1242.

Meier, F., Hennig, P., Schaal, S., 2014. Local gaussian regression. arXiv preprint arXiv:1402.0645.

Munzer, T., Stulp, F., Sigaud, O., 2014. Non-linear regression algorithms for motor skill acquisition: a comparison. In: Proceedings JFPDA. pp. 1–16.

Neal, R. M., 1996. Bayesian Learning for Neural Networks. Springer-Verlag.

Nguyen-Tuong, D., Seeger, M., Peters, J., 2009. Model learning with local gaussian process regression. Advanced Robotics 23 (15), 2015–2034.

Orr, M. J. L., Hallam, J., Takezawa, K., Murray, A. F., Ninomiya, S., Oide, M., Leonard, T., 2000. Combining regression trees and radial basis function networks. Int. J. Neural Syst. 10 (6), 453–465.

Park, J., Sandberg, I. W., 1993. Approximation and radial-basis-function networks. Neural computation 5 (2), 305–316.

Pascanu, R., Montúfar, G., Bengio, Y., 2013. On the number of inference regions of deep feed forward networks with piece-wise linear activations. arXiv preprint arXiv:1312.6098.

Plackett, R. L., 1950. Some theorems in least squares. Biometrika 37, 149–157.

Poggio, T., Girosi, F., 1990. Networks for approximation and learning. Proceedings of the IEEE 78 (9).

Quiñonero Candela, J., Rasmussen, C. E., 2005. A unifying view of sparse approximate Gaussian process regression. Journal of Machine Learning Research 6, 1939–1959.

Quinlan, R. J., 1992. Learning with continuous classes. In: 5th Australian Joint Conference on Artificial Intelligence. World Scientific, Singapore, pp. 343–348.

Rahimi, A., Recht, B., 2007. Random features for large-scale kernel machines. In: Advances in neural information processing systems. pp. 1177–1184.

Rahimi, A., Recht, B., 2008. Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning. In: Advances in Neural Information Processing Systems 21. pp. 1313–1320.

Rosenblatt, F., 1958. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological review 65 (6), 386–408.

Saunders, C., Gammerman, A., Vovk, V., 1998. Ridge regression learning algorithm in dual variables. In: (ICML-1998) Proceedings of the 15th International Conference on Machine Learning. Morgan Kaufmann, pp. 515–521.

Schaal, S., Atkeson, C. G., 1997. Receptive field weighted regression. Tech. Rep. TR-H-209, ATR Human Information Processing Laboratories.

Schmidhuber, J., 2014. Deep learning in neural networks: An overview. arXiv preprint arXiv:1404.7828.

Schmidt, M., 2005. Least squares optimization with l1-norm regularization. Tech. rep., CS542B Project Report.

Schmitt, M., 2002. On the complexity of computing and learning with multiplicative neural networks. Neural Computation 14 (2), 241–301.

Schölkopf, B., Smola, A., Williamson, R., Bartlett, P., 2000. New support vector algorithms. Neural computation 12 (5), 1207–1245.

Schwenker, F., Kestler, H. A., Palm, G., May 2001a. Three learning phases for radial-basis-function networks. Neural Netw. 14 (4-5), 439–458.

Schwenker, F., Kestler, H. A., Palm, G., 2001b. Three learning phases for radial-basis-function networks. Neural networks 14 (4), 439–458.

Sigaud, O., Salaün, C., Padois, V., 2011. On-line regression algorithms for learning mechanical models of robots: a survey. Robotics and Autonomous Systems 51, 1117–1125.

Smola, A. J., Schölkopf, B., 2004. A tutorial on support vector regression. Statistics and computing 14 (3), 199–222.

Stulp, F., Beetz, M., June 2008. Refining the execution of abstract actions with learned action models. Journal of Artificial Intelligence Research (JAIR) 32, 487–523.

Stulp, F., Theodorou, E., Schaal, S., 2012. Reinforcement learning with sequences of motion primitives for robust manipulation. IEEE Transactions on Robotics 28 (6), 1360–1370, King-Sun Fu Best Paper Award of the IEEE Transactions on Robotics for the year 2012.

Tibshirani, R., 1996. Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society (Series B) 58, 267–288.

Vapnik, V., 1995. The Nature of Statistical Learning Theory. Springer, New York,.

Vijayakumar, S., Schaal, S., 2000. Locally weighted projection regression: An O(n) algorithm for incremental real time learning in high dimensional space. In: Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000). Vol. 1. pp. 288–293.

Wang, Y., Witten, I. H., 1997. Induction of model trees for predicting continuous classes. In: Poster papers of the 9th European Conference on Machine Learning. Springer.

Werbos, P. J., 1974. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. thesis, Harvard University.

Werbos, P. J., 1988. Generalization of backpropagation with application to a recurrent gas market model. Neural Networks 1 (4), 339–356.

Widrow, B., Greenblatt, A., Kim, Y., Park, D., 2013. The no-prop algorithm: A new learning algorithm for multilayer neural networks. Neural Networks 37, 182–188.

Williams, C. K., Rasmussen, C. E., 2006. Gaussian processes for machine learning. MIT Press.

Williams, C. K. I., Jul. 1998. Computation with infinite neural networks. Neural Comput. 10 (5), 1203–1216.

Wilson, D. R., Martinez, T. R., 2003. The general inefficiency of batch training for gradient descent learning. Neural Networks 16 (10), 1429–1451.