Business Analytics

– SEMINAR SUMMER SEMESTER 2014 –

Introduction to Neural Networks

– SEMINAR PAPER –

Submitted by:
Nicole Ludwig

Advisor:
Prof. Dr. Dirk Neumann

Contents

1 Introduction

Biology is a huge inspiration for technical models. Those models aim to exploit their capabilities by imitating their biological role model. The human brain, for example, can learn complex tasks easily and adapt the learned to new situations. These learning processes are achieved by the biological neurons through constant changing of their structure. This adaptability to change, is the main task a computer normally cannot perform. Although, the switching times in a computer are way higher than in the human brain, the computer cannot learn and is not capable of performing the complex tasks, that the biological neurons are able to perform. Thus, neural networks were invented as a statistical learning tool, which tries to imitate the brains learning process. Hereby, the network does not need to be programmed explicitly to a certain tasks, it can, if implemented correctly, learn from training data, either on its own (unsupervised learning), or with the help of a teacher (supervised learning). If the neural network is correctly specified it can solve new similar problems and develops a certain fault toleration, which is especially important with perturbed data. The implementation of neural networks is thus motivated by the relativeness to the human central nervous system. Research in the field of neural networks occurred in several 'waves', starting with the perceptron of McCulloch and Pitts (1943), and followed by Rosenblatt (1962). It first came to a halt when Minsky and Papert (1988) evaluated the perceptrons performance and pointed out its limitations, which could not be overcome due to the limited computational capacities. But the research was later revived with the development of the backpropagation algorithm for multilayer perceptrons and more recently deep learning. Neural networks are mainly used for regression and classification tasks, prominent tasks are the forecasting of stock market prices or pattern recognition.

This paper aims to introduce to artificial neural networks, that imitate biological neural networks as information processing systems, to solve learning tasks. It focuses on multilayer perceptrons, because those are applied mostly. Section 2 first explains the basic mathematics and structure of multilayer perceptrons. Then, basic learning algorithms and issues in training are discussed in Section 3. In the last section, the paper gives a short introduction to the Bayesian modelling of neural networks.

2 Multilayer Perceptrons

This section gives a definition of neural networks and a short overview of their structure. A *neural network* is, mathematically speaking, defined as a directed graph with the following properties (Müller et al. 1995):

- a state variable of each neuron
- a real-valued weight w_{ij} associated with each link (i,j) between two nodes i and j
- a real valued bias w_{0i} associated with each node i
- an activation function, for each node i.

The neurons or nodes, which imitate the biological neurons, represent small computational units. The links between those units represent the biological synapses. The information is processed through the network via those links, which are in the setting of a multilayer perceptron, weighted edges. Each weight can either have a strengthening or a weakening effect on the successive neuron. In the biological model the neurons only "fire" and become active if their input exceeds a certain activation potential. Otherwise they remain inactive. In the artificial neural network the activation potential is imitated through the activation function.

In general, they're is a distinction between two types of networks; a *feed-forward network*, is a network, whose topology has no closed paths, whereas a *recurrent network*, contains directed circles or loops (Kruse et al. 2013). In this paper, we focus on feed-forward networks, which are also called multilayer perceptrons.

A multilayer perceptron represents some function $f : X \mapsto Y$. Very complex functions can be calculated through the combination of many neurons. Those neurons are all connected and arranged in three (or more) layers. An *input layer*, that receives the input vector $\boldsymbol{x} = x_1, \dots, x_n$ and has no predecessor, one (or more) *hidden layers* which connect the input neurons with the output neurons, and an output layer, which has no successors. The output layer can consist of only one neuron or of several neurons. A scheme of a neural network is shown in Figure 1.

The computations in the network can be described as multiple linear combinations. At the hidden layer we compute N linear combinations of the input variables $\boldsymbol{x} = x_1, \dots, x_n$ in the form

$$a_j = \sum_{i=1}^{N} w_{0j} + w_{ij} x_i$$

where w_{0j} is the bias weight with $x_0 = 1$, x_i are the input variables and w_{ij} is the weight from neuron i to neuron j. The quantities a_j are known as *activations* of neuron j in the hidden layer. Each of them is transformed using a differentiable, non-linear *activation function* $\mathrm{A}(\cdot)$ to give

$$z_j = \mathrm{A}(a_j).$$

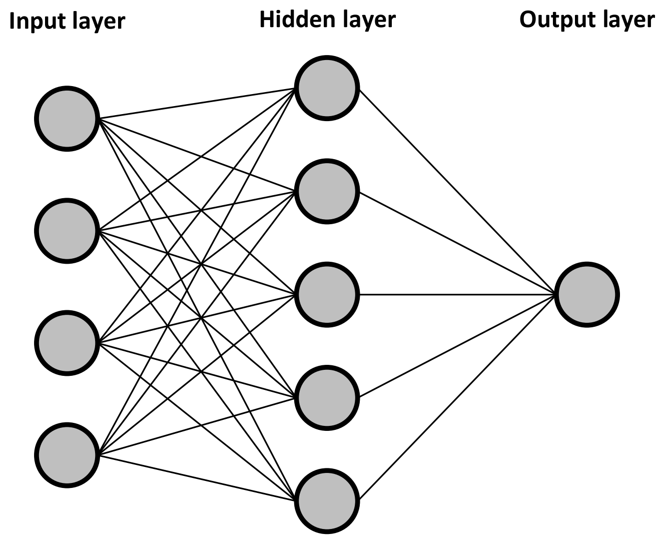

Figure 1: Schematic neural network with three layers and a single output neuron.

These values are again linearly combined to give the activation of neuron k in the output layer, with M being the number of neurons in the hidden layer

$$a_k = \sum_{j=1}^{M} w_{0k} + w_{jk} z_j.$$

This activation is again transformed by A to

$$y_k = \mathrm{A}(a_k).$$

The activation function is chosen according to the function the network is representing. In the first perceptron introduced by McCulloch and Pitts (1943) the activation function was a threshold-step function. This is closest to the biological model, where the neurons are only activated if they exceed a certain threshold. Thus they only have the two states active or inactive

$$f_{step}(z) = \begin{cases} 1, & \text{if } z \geq \theta, \\ 0, & \text{if } z < \theta, \end{cases}$$

where θ is the threshold, which needs to be exceeded, to activate the neuron. In the setting of multilayer perceptrons the activation function needs to be differentiable. Thus, the function is often chosen to be sigmoid. The sigmoid function is then a smooth, continuous approximation to the threshold-step function. Often used is the logistic function

$$f_{log}(z) = \frac{1}{1+e^{-z}},$$

or its linear transformation the hyperbolic tangent

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Neural networks, that use the tanh function, converge, according to Izenman (2008), faster than those using the logistic function. The hyperbolic tangent in comparison to a threshold-step function and a linear function are shown in Figure 2. The activation functions in the hidden and output layer, can either be the same or chosen to be different, depending on the function one wants to represent.

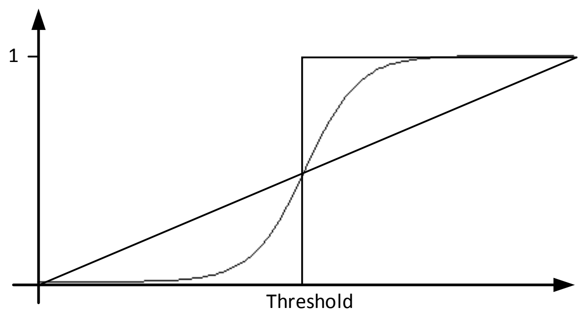

Figure 2: Comparison of a threshold-step, hyperbolic tangent and linear activation function.

We can combine the above stages, so that the total output of the network is then given as

$$y_k(\boldsymbol{x},\boldsymbol{w}) = \mathrm{A}_1\left(w_{0k} + \sum_{j=1}^{M} w_{jk}\cdot \mathrm{A}_2\left(\sum_{i=1}^{N} w_{0j} + w_{ij}x_i\right)\right),$$

where the set of all weights and bias parameters have been grouped together into a vector $\boldsymbol{w}$. Thus the neural network model is simply a non-linear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$, controlled by a vector $\boldsymbol{w}$ of adjustable parameters (Bishop 2006).

3 Training a Neural Network

We can train a neural network by updating the weights on the edges, that connect the nodes. We are searching for the combination of weights that minimizes the error $\mathrm{E}(\boldsymbol{w},\mathrm{D})$of the network. This error depends on the training Data D and the weight vector $\boldsymbol{w}$ by

$$\min_{w} \mathrm{E}(\boldsymbol{w},\mathrm{D}).$$

For the minimization approach we first calculate the error e_k of a single output layer node. It is given by the difference between the actual output y_k and the desired output d_k, i. e.

$$e_k = y_k - d_k.$$

The error of the whole network is thus the sum over all errors of the K output nodes. And the function that needs to be minimized is given by

$$\mathrm{E}(\mathrm{w}) = \frac{1}{2}\sum_{n=1}^{K} ||y(x_n,\boldsymbol{w}) - d_n||^2,$$

the mean squared error (MSE) of the network. Some authors minimize the residual sum of squares or the cross-entropy (e. g. Rojas 1993, Izenman 2008). The cross-entropy is especially useful for classification tasks (Bishop 2006). We will stick with the minimization of the MSE.

We stated that we minimize the error function by updating the weight vector. A change of the weights by a small step from $\boldsymbol{w}$ to $\boldsymbol{w}+\delta\boldsymbol{w}$, results in a change of the error function by

$$\delta\mathrm{E} \simeq \delta\boldsymbol{w}^T\nabla\mathrm{E}(\boldsymbol{w}),$$

where the gradient $\nabla\mathrm{E}(\boldsymbol{w})$ points in the direction of the error functions biggest increase. As $\mathrm{E}(\boldsymbol{w})$ is a continuous function the gradient is minimal for

$$\nabla\mathrm{E}(\boldsymbol{w}) = 0.$$

Depending on the starting point the gradient descent procedure can find several local minima. It does not seem to be necessary to find a global minimum, as this would probably overfit the data.

But it seems necessary to change the starting point several times and compare the outcome, to find the best possible solution. The update rule is given by

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} + \Delta\boldsymbol{w}^{\tau},$$

where τ labels the iteration step (Bishop 2006).

There are multiple ways to determine the update weight $\Delta\boldsymbol{w}^{(\tau)}$. The most popular supervised learning algorithm used, in training a feed-forward neural network, is the backpropagation algorithm, which computes the derivatives of an error function with respect to the network weights (Izenman 2008).

3.1 Backpropagation Algorithm

The backpropagation algorithm is some form of gradient descent and is a supervised learning algorithm. The algorithm can be divided into two parts. The first part being the *feed-forward* process through the network and the second being the *back-propagation* process. To simplify the calculation a network with only one hidden layer is considered and a schematic representation of the feed-forward process is shown in Figure 3. In the first part, the input information is processed through the network after initializing a weight vector $\boldsymbol{w} = w_1, \dots, w_n$. The activation at the jth hidden node is

$$z_j = A\left(w_{0j} + \sum_j w_{ij}x_i\right).$$

The information is then processed further, thus the output at the kth output node is given by

$$y_k = A\left(w_{0k} + \sum_k w_{jk}z_j\right).$$

The activation function A, can either be the same non-linear differentiable function for both layers, or it can be different for each layer. Depending on the function, the network should represent.

The second part of the algorithm goes backwards through the network. Starting at the kth output node, we calculate the derivation of the error function with respect to the weights that connect the output and hidden neuron. As the error function E only depends on w over the output y_k, we use the chain rule

$$\frac{\partial \mathrm{E}_n}{\partial w_{jk}} = \frac{\partial \mathrm{E}_n}{\partial e_k}\frac{\partial e_k}{\partial y_k}\frac{\partial y_k}{\partial w_{jk}}.$$

For the sake of clarity, the iteration superscripts τ are omitted. The above function can also be written as

$$\frac{\partial \mathrm{E}_n}{\partial w_{jk}} = -\delta_k z_{ij},$$

with the local gradient δ_k being

$$\delta_k := \frac{\partial \mathrm{E}_n}{\partial y_k}\frac{\partial y_k}{\partial a_k}.$$

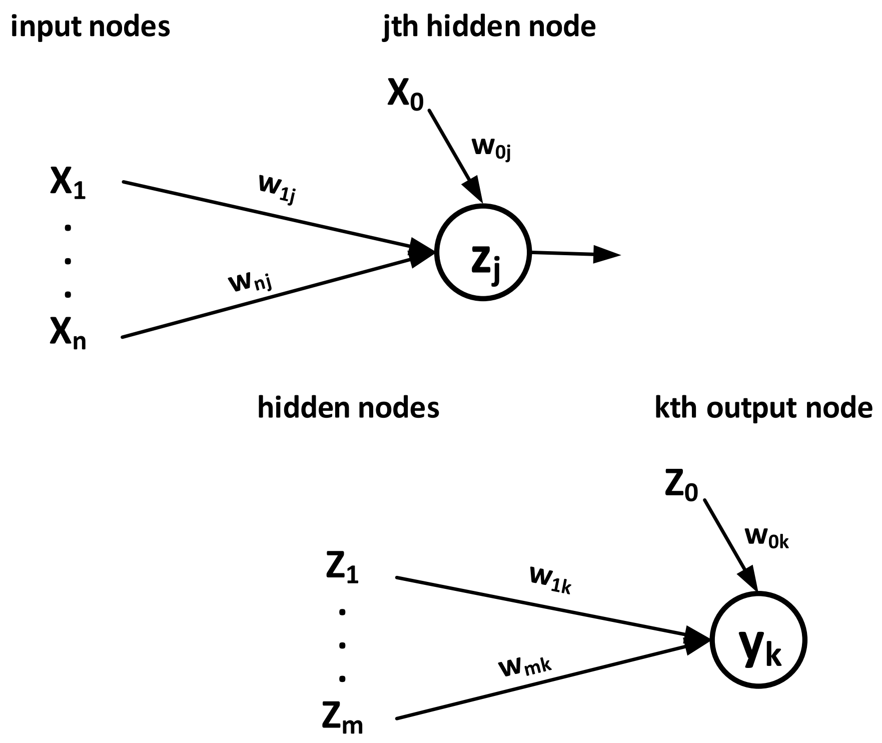

Figure 3: A schematic representation of the feed-forward step of the backpropagation algorithm with a single hidden layer. With the activation of node j and the output of node k being calculated according to the formulae in Section 3 (based on Izenman 2008).

The gradient descent update for the connection weight between the output and hidden layer is thus

$$w_{jk}{}^{\tau+1} = w_{jk}{}^{\tau} - \frac{\partial \mathrm{E}_n}{\partial w_{jk}{}^{\tau}} = w_{jk}{}^{\tau} + \delta_k z_{ij}.$$

Going further through the network, we can also calculate the update for the connection weight between the hidden and input layer applying the chain rule

$$\frac{\partial \mathrm{E}_n}{\partial w_{ij}} = \frac{\partial \mathrm{E}_n}{\partial z_j} \frac{\partial z_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}},$$

with the activation function a_j and

$$\frac{\partial \mathrm{E}_n}{\partial z_j} = \sum_k e_k \frac{a_k}{z_j} = -\sum_k \delta_k w_{jk}.$$

The local gradient δ_j can be calculated with

$$\delta_j = \mathrm{A}'(a_j) \sum_k \delta_k w_{jk},$$

resulting in a gradient descent update

$$w_{ij}{}^{\tau+1} = w_{ij}{}^{\tau} - \frac{\partial \mathrm{E}_n}{\partial w_{ij}{}^{\tau}} = w_{ij}{}^{\tau} + \delta_j x_i.$$

We can call the equation for δ_j the backpropagation formula. In general, the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network (Bishop 2006). A generic algorithm for learning by epoch is

Algorithm 1 Generic multilayer perceptron learning algorithm

```
choose an initial weight vector $\boldsymbol{w}$
while error did not converge do
    for all $\boldsymbol{x}$ do
        apply $\boldsymbol{x}$ to the network and calculate the output
        calculate $\delta$ for all weights
    end for
    calculate $\frac{\partial E}{\partial w_{ij}}$ for all weights summing over all training patterns
    perform one update step of the minimization approach
end while
```

The algorithm is that popular because it is simple and local and can be implemented easily on a parallel architecture computer (Hastie et al. 2009).

3.2 Other Training Algorithms

Although the backpropagation algorithm is simple and more or less straightforward, it does not need to converge. Or, it may take a lot of time until the algorithm converges. Thus, to accelerate the convergence speed, other algorithms have been introduced to find the appropriate update weights of the network.

Backpropagation with Variable Learning Rate

We can change the convergence speed of the algorithm by including a learning rate η. The gradient of the error function is then

$$\nabla E_i = \eta \frac{\partial E_i}{\partial w_{ij}},$$

and the weight at iteration τ is

$$\boldsymbol{w}^{\tau} = \boldsymbol{w}^{(\tau-1)} - \eta \nabla E.$$

The learning rate serves as the step width of the gradient descent. If the steps are too small, it will take a lot of computation time to find the local minimum. If the steps are too large, it is likely that one misses a minimum (Rojas 1993).

Backpropagation with Momentum

To prevent the algorithm from oscillating in deep valleys of the error function surface we can add a momentum to the update function. The momentum gives more information of the weight space, leading the function faster to the center of the 'valley'. Furthermore, it can accelerate the minimization approach in flat areas of the error function. The weight update is then given by

$$\Delta w_{\tau} = -\eta \nabla E + \alpha \Delta w_{\tau-1},$$

where α is the momentum, which is derived empirically (Henseler 1995; Rojas 1993). Rumelhart et al. (1988) achieved fast convergence with $\alpha \approx 0.9$.

Quickprop

Using large data sets the backpropagation algorithm might converge really slow. Therefore, to increase the convergence speed, Fahlman (1988) introduced another algorithm, the quickprop. This algorithm approximates the error function by a parabola and tries to 'jump' directly to the point where the gradient is zero, by using information from the previous and current gradient (Kruse et al. 2013). Resulting in an update rule of

$$\Delta w_i{}^{\tau} := \frac{-\nabla E^{\tau}}{\nabla E^{\tau} - \nabla E^{\tau-1}} \left(w_i{}^{\tau} - w_i{}^{\tau-1} \right).$$

Large steps can be avoided by bounding the weight change upwards. According to Kruse et al. (2013), the algorithm is really quick, if the parameters are largely independent and we carry out batch learning.

3.3 Regularization in Training Neural Networks

In the previous sections, we have discovered algorithms to choose the weights and thresholds of the neurons. However, those algorithms do not help choosing the network topology and initial weights. The main problems while fitting a neural network are over fitting and stopping at a rather bad local minimum. If we fit the network too closely on the training data, we are not able do generalize well. Thus, we need the model to be relatively simple (cf. Ockham's razor). If the initial weights are chosen badly the minimum where the algorithm stops does not reduce the error function sufficiently. Hence, in this section we consider several methods to regularize the networks performance and prevent over fitting.

Early Stopping

Early stopping describes a method to regularize the complexity of the network. Although the error function of the network on the training set is constantly decreasing, the error on a validation set is normally not. As the network starts to over fit, the error on the validation set starts to increase, as can be seen in Figure 4. Thus, to have the best performance for generalization, the training can be stopped at the minimum of the error on the validation data.

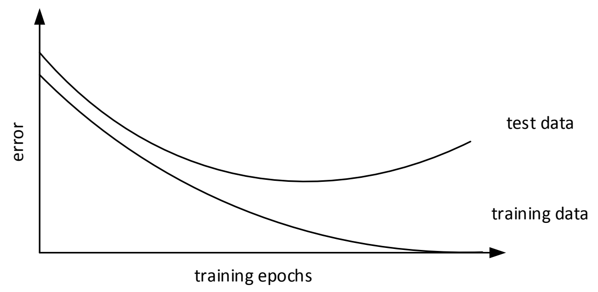

Figure 4: Error function on the test and the validation set

Weight Decay

In general, it is not desirable to have large values for the connection weights, because of two main reasons. The first is that with greater weight the gradient vanishes and the training process is really slow or even comes to a halt. Secondly, if too much weight is given to information in the training process, the network is more likely to be over-fitted and thus perform badly when generalized. *Weight decay* is thus a method that prevents the weights from increasing too much by adding a penalty term to the weights in each step.

Optimal Number of Hidden Layers and Neurons

If we train a network we have to decide how many hidden neurons we include in how many hidden layers. As this is not an easy decision, one of the best methods is just to use trial and error. For forecasting it has been shown that a single hidden layer performs quite well. Nevertheless, we can also choose the optimal network size using *cross-validation* or *bootstrapping*. In both methods the test and training sample out of the given data set are changed several times. We can then choose the weights and number of layers that performed best or take an average over the models.

Ensemble Learning

The basic idea for ensemble learning methods is that it is more likely for a single expert to fail than for a group of experts. Therefore, those methods try to 'ask' a committee of experts to choose the appropriate network. Two ensemble learning methods are *bagging* and *boosting*. For *bagging*, one builds a committee by training several multilayer perceptrons on bootstrap samples of the training data. Afterwards, we calculate the average output of all those multilayer perceptrons. In *boosting*, the multilayer perceptrons are not trained independently, but the second multilayer perceptron is learned on the training data that was not well learned by the first multilayer perceptron, the third multilayer perceptron is learned on the training data that was not well learned by the first and second multilayer perceptron, and so on. Thus, we get better committees on the training set. Those methods are described in detail in e. g. Hastie et al. (2009) and Murphy (2012).

4 A Bayesian View on Neural Networks

In this last section, we take a look at the Bayesian modelling of neural networks. The difference to the approach before is that we assume the connection weights $\boldsymbol{w}$ of our network to be random with a prior distribution $p(\boldsymbol{w})$. Afterwards, we update our expectations according to the Bayes theorem to the posterior distribution $p(\boldsymbol{w}|D)$, where D is a vector of target variables t (Hippert and Taylor 2010). In this paper, we restrict ourselves to the discussion of a regression example. We take a look at the *Laplace method*, where predictions can be made by estimating the maximum of the posterior distribution (MAP estimation) (Izenman 2008).

Following Murphy (2012) and Bishop (2006), it is the aim to predict a single continuous target variable t, from an input vector $\boldsymbol{x}$. We use the conditional distribution

$$p(t|\boldsymbol{x},\boldsymbol{w},\beta) = \mathcal{N}(t|y(\boldsymbol{x},\boldsymbol{w}),\beta^{-1})$$

and the prior $p(\boldsymbol{w}|\alpha) = \mathcal{N}(\boldsymbol{w}|0,\alpha^{-1}\boldsymbol{I})$, with $\boldsymbol{w}$ being the combined weights and the precision of the noise $\beta = \frac{1}{\sigma^2}$.

The non-Gaussian posterior distribution is given by

$$p(\boldsymbol{w}|D,\alpha,\beta) \propto p(\boldsymbol{w}|\alpha)p(D|\boldsymbol{w},\beta),$$

where $p(D|\boldsymbol{w},\beta)$ is the likelihood function. To find a Gaussian approximation, we can estimate the maximum posterior, denoted by w_{MAP} and get the Hessian of the data error

$$\boldsymbol{H} = \nabla^2 E_D(w_{\text{MAP}}).$$

The Hessian M of the network error is then given by

$$\boldsymbol{M} = \beta\boldsymbol{H} + \alpha\boldsymbol{I}.$$

Thus, the resulting Gaussian posterior is provided by the equation

$$p(\boldsymbol{w}|\alpha,\beta,D) \approx \mathcal{N}(\boldsymbol{w}|w_{\text{MAP}},\boldsymbol{M}^{-1}).$$

Marginalizing with respect to the posterior distribution leads to the predictive distribution

$$p(t|x,D,\alpha,\beta) = \int \mathcal{N}(y|y(x,w),\beta^{-1})\mathcal{N}(\boldsymbol{w}|w_{\text{MAP}},\boldsymbol{M}^{-1})d\boldsymbol{w}.$$

This is not computable, thus we can use a Taylor series around the mode w_{MAP} to get a linear Gaussian model with Gaussian prior. The resulting marginal distribution of t is then provided by

$$p(t|x,D,\alpha,\beta) \approx \mathcal{N}(t|y(x,w_{\text{MAP}},\sigma^2(x)),$$

where the predictive variance depends on the noise on the target variable and the uncertainty arising in the interpolant due to the uncertain model parameters $\boldsymbol{w}$

$$\sigma^2(x) = \beta^{-1} + g^T\boldsymbol{M}^{-1}g.$$

With g being the gradient of the output function at the point $w = w_{\text{MAP}}$.

The Bayesian model approach can be used to determine the optimal number of hidden units, where it is much faster than e. g. cross-validation. And we can model uncertainty for tasks like active learning or risk-averse decision making (Murphy 2012).

5 Conclusion

This paper has introduced to the concepts of neural networks as statistical learning tools. We have seen, how neural networks compute their output, and can be trained on a given data set, to be able to predict future values. Interestingly, they are in fact *"multiple layers of logistic regression models with continuous nonlinearities"* (Bishop 2006). Nevertheless, neural networks are very likely to overfit, or find no global minimum. This occasionally makes them inferior to other machine learning tasks i. e. Support Vector Machines (Olson and Delen 2008). But, as mentioned above, there are several regularization methods possible to improve their overall performance. A further motivation to use neural networks is given by the *universal approximation theorem*. It proves that there exists, under some assumptions, for any continuous function in $\mathbb{R}^n$ an approximation with a single-layer hidden network and a finite number of neurons in the hidden layer. Additionally, the Bayesian approach to neural networks gives new possibilities to model uncertainty in the parameters.

Recent research is especially interested in *deep learning* with neural networks, since the neocortex of the brain is believed to work in hierarchical levels. Those levels, which are associated with an increase in abstraction, are represented by the depth of the network, where the depth refers to the number of levels of non-linear functions in the task learned.

To conclude, neural networks are a helpful statistical learning tool, that has been implemented successfully in economics, e. g. for the prediction of stock market prices. Their successors, the deep learning networks, are supposed to further improve the artificial intelligence capabilities of computers.

A References

BISHOP, C. M. (2006). *Pattern recognition and machine learning*. eng. Information science and statistics. New York, NY: Springer, p. 738. ISBN: 0-387-31073-8, 978-0-387-31073-2.

FAHLMAN, S. E. (1988). *An empirical study of learning speed in back-propagation networks*. In:

HASTIE, T. J., R. J. TIBSHIRANI, and J. H. FRIEDMAN (2009). *The elements of statistical learning : data mining, inference, and prediction*. eng. 2. ed. Springer series in statistics. New York, NY: Springer, p. 745. ISBN: 978-0-387-84857-0.

HENSELER, J. (1995). *Back Propagation*. In: *Artificial Neural Networks*. Ed. by P. BRASPENNING, F. THUIJSMAN, and A. WEIJTERS. Vol. 931. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 37–66. ISBN: 978-3-540-59488-8.

HIPPERT, H. S. and J. W. TAYLOR (2010). *An evaluation of Bayesian techniques for controlling model complexity and selecting inputs in a neural network for short-term load forecasting*. In: *Neural Networks*, Vol. 23, No. 3, pp. 386 –395. ISSN: 0893-6080.

IZENMAN, A. J. (2008). *Modern multivariate statistical techniques : regression, classification, and manifold learning*. eng. Springer texts in statistics. New York: Springer, p. 731. ISBN: 978-0-387-78188-4, 0-387-78188-9.

KRUSE, R. et al. (2013). *Computational Intelligence A Methodological Introduction*. eng. Texts in Computer Science, SpringerLink : Buecher. London: Springer, Online–Ressource (XI, 488 p. 234 illus, digital). ISBN: 978-1-447-15013-8.

MCCULLOCH, W. and W. PITTS (1943). *A logical calculus of the ideas immanent in nervous activity*. English. In: *The bulletin of mathematical biophysics*, Vol. 5, No. 4, pp. 115–133. ISSN: 0007-4985.

MINSKY, M. L. and S. PAPERT (1988). *Perceptrons : an introduction to computational geometry*. eng. Expanded ed., 3. print. Cambridge, Mass. [u.a.]: MIT Pr., p. 292. ISBN: 0-262-63111-3.

MÜLLER, B., J. REINHARDT, and M. T. STRICKLAND (1995). *Neural networks : an introduction*. eng. 2., upd. and corr. ed. Physics of neural networks. Berlin ; Heidelberg [u.a.]: Springer, p. 329. ISBN: 3-540-60207-0.

MURPHY, K. P. (2012). *Machine learning : a probabilistic perspective*. eng. Adaptive computation and machine learning series. Cambridge, Mass. [u.a.]: MIT Press, p. 1067. ISBN: 978-0-262-01802-9, 0-262-01802-0.

OLSON, D. L. and D. DELEN (2008). *Advanced data mining techniques*. eng. Berlin ; Heidelberg [u.a.]: Springer, p. 180. ISBN: 978-3-540-76916-3.

ROJAS, R. (1993). *Theorie der neuronalen Netze : eine systematische Einführung*. ger. Springer-Lehrbuch. Berlin ; Heidelberg [u.a.]: Springer, p. 446. ISBN: 3-540-56353-9.

ROSENBLATT, F. (1962). *Principles of neurodynamics*. In:

RUMELHART, D. E., G. E. HINTON, and R. J. WILLIAMS (1988). *Readings in cognitive science : a perspective from psychology and artificial intelligence*. English. In: ed. by A. M. COLLINS and E. E. SMITH. San Mateo, Calif.: Morgan Kaufmann. Chap. Learning internal representations by error propagation. Pp. 399–421.

B List of Figures