RWTH Aachen University
Chair of Computer Science 6
Prof. Dr.-Ing. Hermann Ney

Selected Topics in Human Language Technology and Pattern Recognition WS 13/14

Introduction to Neural Networks

David Stutz

February 10, 2014

Adviser: Pavel Golik

Contents

List of Tables

List of Figures

1 Abstract

In this seminar paper we study artificial neural networks, their training and application to pattern recognition. We start by giving a general definition of artificial neural networks and introduce both the single-layer and the multilayer perceptron. After considering several activation functions we discuss network topology and the expressive power of multilayer perceptrons. The second section introduces supervised network training. Therefore, we discuss gradient descent and Newton's method for parameter optimization. We derive the error backpropagation algorithm for evaluating the gradient of the error function and extend this approach to evaluate its Hessian. In addition, the concept of regularization will be introduced. The third section introduces pattern classification. Using maximum likelihood estimation we derive the cross-entropy error function. As application, we train a two-layer perceptron to recognize handwritten digits based on the MNIST dataset.

2 Motivation

Theoretically, the human brain has a very low rate of operations per second when compared to a state of the art computer [8, p. 28]. Nevertheless, all computers are still outraveled by the human brain considering an important factor: **learning**. The human brain is able to learn how to perform certain tasks based on experience and prior knowledge.

How to teach computers to learn? To clarify the term "learning" in respect to computers we assume a set of training data $T = \{(x_n, t_n) : 1 \leq n \leq N\}$ for some $N \in \mathbb{N}$ and an arbitrary target function g of which we know the target values $t_n := g(x_n)$. Our goal is to teach the computer a reasonably good approximation of $g(x)$ for x in the domain of g. Many classification[1] and regression[2] problems can be formulated this way. The target function may even be unknown.

Considering noise within the given training data such that we only have a value $t_n \approx g(x_n)$ we require an additional property that we call **ability for generalization**. Solving the problem by interpolation may result in exact values $g(x_n)$ but may be a very poor approximation of $g(x)$ in general. This phenomenon is called over-fitting of the underlying training data.

2.1 Historical Background and Bibliographical Notes

In 1943 McCulloch and Pitts introduced the first mathematical models concerning networks of neurons we call artificial neural networks. But this first step did not include any results on network training [6, p. 333-335].

The first work on how to train similar networks was Rosenblatt's perceptron in 1958 which we discuss in section 3.1 [19]. Only ten years later Minsky and Papert showed that Rosenblatt's perceptron hat many limitations as we see in section 3.6 and can only model linearly separable[3] problems of classification [6, p. 333-335].

In [20] Rumelhart, Hinton and Williams introduced the idea of error backpropagation for pattern recognition. In the late 80's it was shown that non linearly separable problems can be solved by multilayer perceptrons [10].

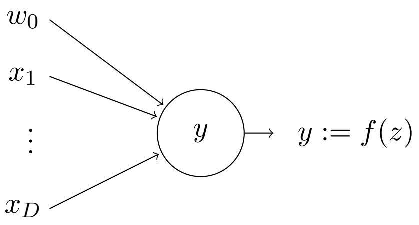
Weight decay was introduced in [9]. A diagonal approximation of the Hessian was introduced in [12]. A pruning method based on diagonal approximation of the Hessian is called Optimal Brain Damage and discussed in [16]. The exact evaluation of the Hessian is discussed in [2].

[1] The classification problem can be stated as follows: Given a D-dimensional input vector x assign it to one of C discrete classes (see section 5).

[2] The regression problem can be described as follows: Given a D-dimensional input vector x predict the value of C continuous target values.

[3] Considering a classification problem as introduced in section 5 we say a set of data points is not linearly separable if the classes can not be separated by a hyperplane [5, p. 179].

Figure 1: Single processing unit and its components. The activation function is denoted by f and applied on the actual input z of the unit to form its output $y = f(z)$. $x_1, \ldots, x_D$ represent input from other units within the network; w_0 is called bias and represents an external input to the unit. All inputs are mapped onto the actual input z using the propagation rule.



3 Neural Networks

An artificial neural network, also referred to as "neural network", is a set of interconnected processing units. A processing unit receives input from external sources and connected units and computes an output which may be propagated to other units. These units represent the neurons of the human brain which are interconnected by synapses [8, p. 23-24].

A processing unit consists of a propagation rule and an activation function. The propagation rule determines the actual input of the unit by mapping the output of all direct predecessors and additional external inputs onto a single input value. The activation function is then applied on the actual input and determines the output of the unit. The output of the processing unit is also called activation. This is illustrated by figure 1 showing a single processing unit where f denotes the activation function, z the actual input and y the output of the unit.

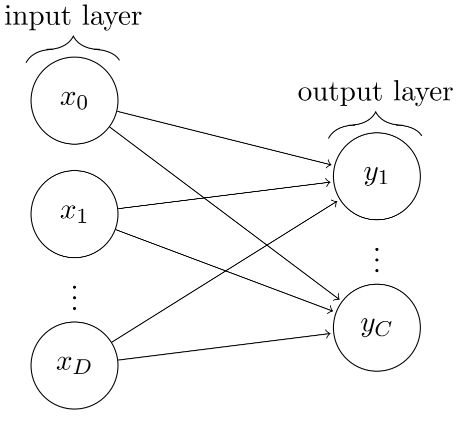
We distinguish input units and output units. Input units accept the input of the whole network and output units form the output of the network. Each input unit accepts a single input value x and we set the output to $y := x$. Altogether, a neural network models a function $y(x)$ which dimensions are determined by the number of input and output units.

As to visualize neural networks we use directed graphs which we call network graphs. As illustrated in figure 1, single processing units are represented by nodes and are interconnected by directed edges. The nodes of the graph are labeled according to the corresponding output.

3.1 The Perceptron

As example we discuss Rosenblatt's perceptron which was introduced in 1958 [6, p. 333-335]. The perceptron consists of D input units and C output units. Every input unit is

Figure 2: The perceptron consists of D input units and C output units. All units are labeled according to their output: $y_i = f(z_i)$ in the case of output units; x_i in the case of input units. The input values x_i are propagated to each output unit using the weighted sum propagation rule. The additional input value $x_0 := 1$ is used to include the biases as weights. As suggested in section 3.3 the units are assembled in layers.



connected to every output unit as shown in figure 2. For $1 \leq i \leq C$ the i^{th} output unit computes the output

$$y_i = f(z_i) \text{ with } z_i = \sum_{k=1}^{D} w_{ik}x_k + w_{i0} \tag{1}$$

where x_j is the input of the j^{th} input unit. In this case the propagation rule is the weighted sum over all inputs with weights w_{ik} and biases w_{0k}. The bias can be included as weight when considering an additional input $x_0 := 1$ such that the actual input z_i can be written as

$$z_i = \sum_{k=0}^{d} w_{ik}x_k. \tag{2}$$

Throughout this paper we use the weighted sum as propagation rule for all units except the input units while activation functions may vary according to the discussion of the next section.

3.2 Activation Functions

The activation function determines the output of the unit. Often, a threshold function as for example the heaviside function

$$h(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{3}$$

is used [8, p. 34-37]. But in general we want the activation function to have certain properties. Network training using error backpropagation as discussed in section 4.4 requires the activation function to be differentiable. In addition, we may want to use nonlinear activation functions as to increase the computational power of the network as discussed in section 3.6 [6, p. 307-308].

A sigmoid function is a commonly used s-shaped function. The logistic sigmoid is given by

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{4}$$

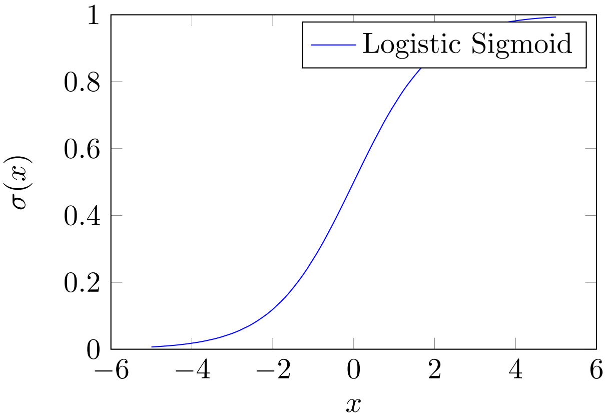


Figure 3: The logistic sigmoid is a commonly used s-shaped activation function. It is both smooth and monotonic and allows an probabilistic interpretation as its range is limited to $[0, 1]$.

and shown in figure 3. It can be considered as smooth version of the heaviside function. The softmax function is given by

$$\sigma(z, i) = \frac{\exp(z_i)}{\sum_{k=1}^{C} \exp(z_k)} \tag{5}$$

where C is the dimension of the vector z. Both functions are smooth and monotonic which means there are no additional local extrema. This is desirable for network training because multiple extrema within the activation function could cause additional extrema within the error surface [6, p. 307-308]. In addition they allow a probabilistic interpretation which we use in section 5. The derivatives of both the logistic sigmoid and the softmax function take preferable forms for implementation:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z)), \tag{6}$$

$$\frac{\partial \sigma(z, i)}{\partial z_j} = \sigma(z, i)\left(\delta(i, j) - \sigma(z, j)\right) \tag{7}$$

where δ denotes the Kronecker delta[4].

3.3 Layered Networks

Arranging the units in layers results in a layered network. Figure 2 already introduced the perceptron as layered network. In this case we have a layer of input units and a layer of output units. We may add additional units organized in so called hidden layers. These units are called hidden units as they are not visible from the outside [8, p. 43].

For counting the number of layers we skip the input layer because there is no real processing taking place and no variable weights [5, p. 229]. Thus, the perceptron can be considered a single-layer perceptron without any hidden layers.

3.4 Feed-Forward Networks

Since we can model every neural network in the means of a network graph we get new neural networks by considering more complex network topologies [5, p.229-231]. We distinguish two network topologies:

Feed-forward In a feed-forward topology we prohibit closed cycles within the network graph. This means that a unit in layer p may only propagate its output to a unit in layer l if $l > p$. Thus, the modeled function is deterministic.

Recurrent Recurrent networks allow closed cycles. A connection establishing a closed cycle within a network graph is called feedback connection.

In this paper we consider feed-forward networks only. As demonstrated in figure 2 the single-layer perceptron implements a feed-forward topology.

[4] The Kronecker delta $\delta(i, j)$ equals 1 if $j = i$ and is 0 otherwise.

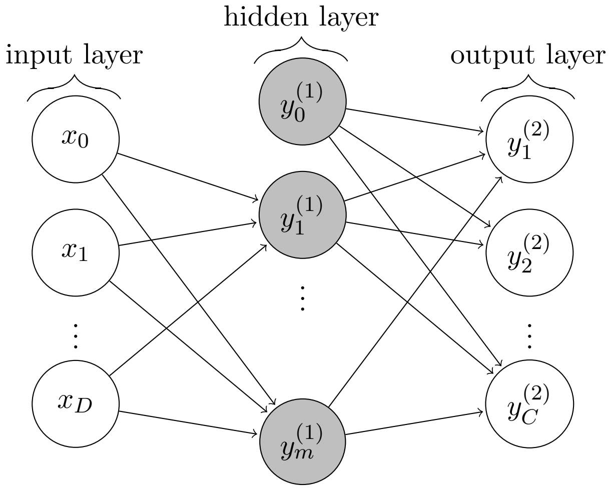


Figure 4: A two-layer perceptron with C input units, D output units and $m := m^{(1)}$ hidden units. Again, we introduced units $x_0 := 1$ and $y_0^{(1)} := 1$ to include the biases as weights. To distinguish units and weights with same indices in different layers, the number of the layer is written as superscript.

3.5 Multilayer Perceptrons

The multilayer perceptron has additional $L \geq 1$ hidden layers. The l^{th} hidden layer consists of $m^{(l)}$ hidden units. The output value $y_i^{(l)}$ of the i^{th} hidden unit within this layer is

$$y_i^{(l)} = f\left(\sum_{k=1}^{m^{(l-1)}} w_{ik}^{(l)} y_k^{(l-1)} + w_{i0}^{(l)}\right) \overset{y_0^{(l-1)}:=1}{=} f\left(\sum_{k=0}^{m^{(l-1)}} w_{ik}^{(l)} y_k^{(l-1)}\right). \tag{8}$$

Altogether, we count $(L+1)$ layers including the output layer where we set $D := m^{(0)}$ and $C := m^{(M+1)}$. Then $w_{ik}^{(l)}$ denotes the weighted connection between the k^{th} unit of layer $(l-1)$ and the i^{th} unit of layer l. Figure 4 shows a two-layer perceptron with $m := m^{(1)}$ units in its hidden layer. As previously mentioned, we added additional units $x_0 := y_0^{(1)} := 1$ to represent the biases.

In this paper we discuss the feed-forward multilayer perceptron as model for neural networks. The modeled function takes the form

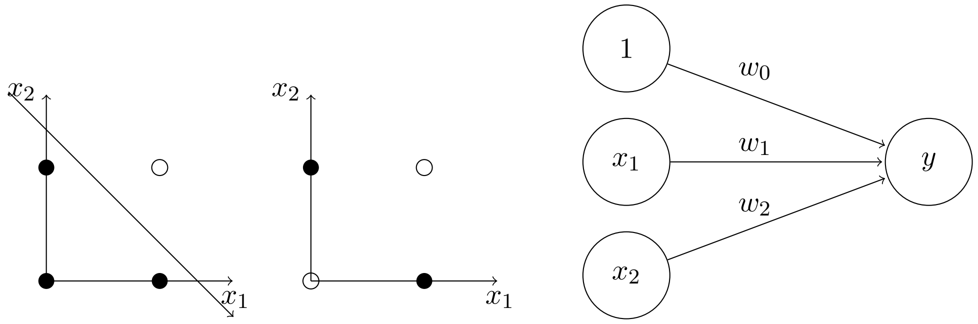
$$y(\cdot, w) : \mathbb{R}^D \to \mathbb{R}^C, x \mapsto y(x,w) = \begin{pmatrix} y_1(x,w) \\ \vdots \\ y_C(x,w) \end{pmatrix} \tag{9}$$

where w is the vector comprising of all weights and $y_i(x,w) := y_i^{(L+1)}(x,w)$ is the output of the i^{th} output unit.

3.6 Expressive Power

One of Minsky and Papert's results in 1969 showed that the single-layer perceptron has severe limitations one of which is called the Exclusive-OR (XOR) problem [6, p.333-335]. As introduction we consider the target function g, which we want to model, given by

$$g : \{0,1\}^2 \to \{0,1\}, x := (x_1, x_2) \mapsto \begin{cases} 1 & \text{if } x_1 = x_2 = 1 \\ 0 & \text{otherwise} \end{cases}. \tag{10}$$



(a) Boolean AND (left) and boolean XOR (right) represented as classification problems.

(b) Network graph of a single-layer perceptron modeling boolean AND.

Figure 5: Figure 5(a) shows both boolean AND and boolean XOR represented as classification problems. As we can see, boolean XOR is not linearly separable. A simple single-layer-perceptron capable of separating boolean AND is shown in figure 5(b).

Apparently, g describes boolean AND. With $D = 2$ input units and $C = 1$ output unit using the $sgn(x)$ activation function, g can be modeled by a single-layer perceptron:

$$y = sgn(z) = sgn(w_1 x_1 + w_2 x_2 + w_0). \tag{11}$$

Figure 5(b) illustrates the corresponding network graph. By setting $z = 0$ we can interpret equation (11) as straight line in two-dimensional space:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}. \tag{12}$$

A possible solution for modeling boolean AND is shown in figure 5(a). But, as we can see, boolean XOR cannot be modeled by this single-layer perceptron. We say boolean XOR is not linearly separable[5] [8, p. 197-200].

This limitation can be overcome by adding at least one additional hidden layer. Theoretically, a two-layer perceptron is capable of modeling any continuous function when using appropriate nonlinear activation functions. This is a direct consequence of Kolmogorov's theorem. It states that for every continuous function $f : [0,1]^D \to \mathbb{R}$ there exist continuous functions $\psi_j : [0,1] \to \mathbb{R}$ such that f can be written in the form

$$f(x_1, \ldots, x_D) = \sum_{j=0}^{2D} g_f \left(\sum_{k=1}^{D} w_k \psi_j(x_k) \right). \tag{13}$$

But as of [3, p. 137-140] the function g_f is dependent on f and the theorem is not applicable if the functions ψ_j are required to be smooth. In most applications we do not know the target function. And the theorem says nothing about how to find the functions ψ_j. It simply states the existence of such functions. Thus, the theorem is not of much practical use for network design [6, p. 287-288]. Nevertheless, the theorem is important as it states that theoretically we are able to find an optimal solution using one hidden layer. Otherwise, we could end up with a problem for which we cannot find a solution using a multilayer perceptron [8, p. 234-235].

[5] When considered as classification problem (see section 5) we say the problem is not linearly separable if the classes can not be separated by a hyperplane [5, p. 179].

4 Network Training

Network training describes the problem of determining the parameters to model the target function. Therefore, we use a set of training data representing the initial knowledge of the target function. Often the target function is unknown and we only know the value of the function for specific data points.

Depending on the training set we consider three different learning paradigms [8, p. 85-88]:

Unsupervised learning The training set provides only input values. The neural network has to find similarities by itself.

Reinforcement learning After processing an input value of the training set the neural network gets feedback whether the result is considered good or bad.

Supervised learning The training set provides both input values and desired target values (labeled training set).

Although the human brain mostly learns by reinforcement learning (or even unsupervised learning), we discuss supervised learning only. Let $T := \{(x_n, t_n) : 1 \leq n \leq N\}$ be a training set where x_n are input values and t_n the corresponding target values. As for any approximation problem we can evaluate the performance of the neural network using some distance measure between approximation and target function.

4.1 Error Measures

We discuss mainly two error measures. The sum-of-squared error function takes the form

$$E(w) = \sum_{n=1}^{N} E_n(w) = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{C} (y_k(x_n, w) - t_{nk})^2 \tag{14}$$

where t_{nk} denotes the k^{th} entry of the n^{th} target value. The cross-entropy error function is given by

$$E(w) = \sum_{n=1}^{N} E_n(w) = -\sum_{n=1}^{N} \sum_{k=1}^{C} t_{nk} \log(y_k(x_n, w)). \tag{15}$$

For further discussion we note that by using $y_i(x_n, w) = f(z_i)$ the derivatives of both error functions with respect to the actual input $z_i^{(L+1)}$ of the i^{th} output unit take the form

$$\frac{\partial E_n}{\partial z_i^{(L+1)}} = \frac{\partial E_n}{\partial y_i^{(L+1)}} \frac{\partial y_i^{(L+1)}}{\partial z_i^{(L+1)}} = y_i(x_n, w) - t_{ni}. \tag{16}$$

Here we use the softmax activation function for the cross-entropy error function and the identity as activation function for the sum-of-squared error function [3, p. 230-231,236-240].

4.2 Training Approaches

Given an error function $E(w) = \sum_{n=1}^{N} E_n(w)$ we distinguish mainly two different training approaches [6, p. 293-295]:

Stochastic training Randomly choose an input value x_n and propagate it through the network. Update the weights based on the error $E_n(w)$.

Batch training Go through all input values x_n and compute $y(x_n, w)$. Update the weights based on the overall error $E(w) = \sum_{n=1}^{N} E_n(w)$.

Instead of choosing an input value x_n at random we could also select the input values in sequence leading to sequential training. As of [5, p. 240-241] stochastic training is considered to be faster especially on redundant training sets.

4.3 Parameter Optimization

The weight update in both approaches is determined by a parameter optimization algorithm which tries to minimize the error. The error $E(w)$ can be considered as error surface above the weight space. In general, the error surface is a nonlinear function of the weights and may have many local minima and maxima. We want to find the global minimum. The necessary criterion to find a minimum is

$$\nabla E(w) = 0 \tag{17}$$

where ∇E denotes the gradient of the error function E evaluated at point w.

As an analytical solution is usually not possible we use an iterative approach for minimizing the error. In each iteration step we choose an weight update $\Delta w[t]$ and set

$$w[t+1] = w[t] + \Delta w[t] \tag{18}$$

where $w[t]$ denotes the weight vector w in the t^{th} iteration and $w[0]$ is an appropriate starting vector. Optimization algorithms differ in choosing the weight update $\Delta w[t]$ [3, p. 254-256]. Before discussing two common optimization algorithms, we discuss the case of linear units and the problem of choosing a good starting vector $w[0]$.

4.3.1 Linear Units

In the case of a single-layer perceptron with linear output units and sum-of-squared error we obtain a linear problem. Thus, we can determine the weights exactly using singular value decomposition [3, p. 259-260].

When considering a multilayer perceptron with linear output units and nonlinear hidden units we can at least split up the problem of minimization. The weights in the output layer can be determined exactly using singular value decomposition whereas the weights in the hidden layers can be optimized using an iterative approach. Note that every time the weights of the nonlinear layers are updated the exact solution for the weights in the output layer has to be recomputed [3, p. 259-260].

4.3.2 Weight Initialization

Following [6, p. 311-312] we want to get a starting vector w such that we have fast and uniform learning, that is all components learn more or less equally fast. Assuming sigmoidal activation functions the actual input of a unit should neither be too large causing $\sigma'(z)$ to be very small nor be too small such that $\sigma(z)$ is approximately linear. Both cases will result in slow learning. For the logistic sigmoid an acceptable value for the unit input is of unity order [3, p. 260-262]. Thus, we choose all weights randomly from the same distribution such that

$$-\frac{1}{\sqrt{m^{(l-1)}}} < w_{ij}^{(l)} < \frac{1}{\sqrt{m^{(l-1)}}} \tag{19}$$

for the weights in layer l. Assuming a normal distribution with zero mean and unity variance for the inputs of each unit we get on average the desired actual input of unity order [6, p. 311-312].

4.3.3 Gradient Descent

Gradient descent is a basic first-order optimization algorithm. This means that in every iteration step we use information about the gradient ∇E at the current point. In iteration step $[t+1]$ the weight update $\Delta w[t]$ is determined by taking a step into the direction of the negative gradient at position $w[t]$ such that

$$\Delta w[t] = -\gamma \frac{\partial E}{\partial w[t]} \tag{20}$$

where γ is called learning rate. Gradient descent can be used both for batch training and stochastic training[5, p. 240-241]. In the case of stochastic training we choose

$$\Delta w[t] = -\gamma \frac{\partial E_n}{\partial w[t]} \tag{21}$$

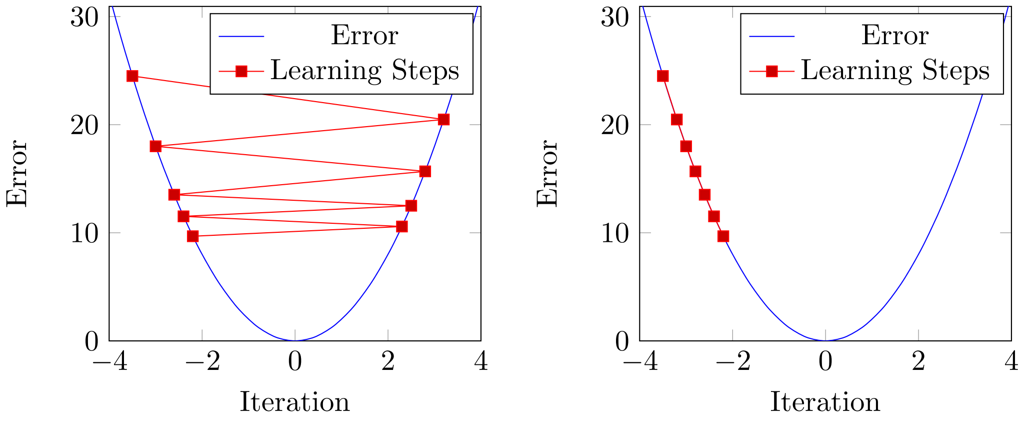
as weight update.

4.3.4 Momentum

The choice of the learning rate can be crucial. We want to choose the learning rate such that we have fast learning but avoid oscillation when choosing the learning rate too large [3, p. 267-268]. Both cases are illustrated by figure 6 where we seek to minimize a convex quadratic function. One way to reduce oscillation while using a high learning rate as described in [3, p. 267-268] is to add a momentum term. Then the change in weight is dependent on the previous change in weight such that

$$\Delta w[t] = -\gamma \frac{\partial E}{\partial w[t]} + \lambda \Delta w[t-1] \tag{22}$$

where λ is called momentum parameter.



(a) High learning rate.

(b) Low learning rate.

Figure 6: When using a large learning rate the weight updates in each iteration tend to overstep the minimum. This causes oscillation around the minimum. This observation is illustrated by figure 6(a). Choosing the learning rate too low will result in a slow convergence to the minimum as shown in figure 6(b). Both scenarios are visualized using a quadratic function we seek to minimize.

4.3.5 Enhanced Gradient Descent

There is still a serious problem with gradient descent: how to choose the learning rate (and the momentum parameter). Currently, we are required to choose the parameters manually. Because the optimal values are dependent on the given problem, an automatic choice is not possible.

We discuss a simple approach on choosing the learning rate automatically considering no momentum term. Let $\gamma[t]$ denote the learning rate in the t^{th} iteration. Then we have a simple criterion on whether the learning rate was chosen too large: If the error has increased after the weight update we may have overstepped the minimum. Then we can undo the weight update and choose a smaller learning rate until the error decreases [3, p. 268-272]. Thus, we have

$$\gamma[t+1] = \begin{cases} \rho \cdot \gamma[t] & \text{if } E(w[t+1]) < E(w[t]) \\ \mu \cdot \gamma[t] & \text{if } E(w[t+1]) > E(w[t]) \end{cases} \tag{23}$$

where the update parameters can be chosen as suggested in [3, p. 268-272]: set $\rho := 1.1$ to avoid frequent overstepping of the minimum; set $\mu := 0.5$ to ensure fast recovery to taking a step which minimizes the error.

4.3.6 Newton's Method

Although gradient descent has been improved in many ways it is considered a poor optimization algorithm as pointed out in [3, p. 268-269] and [5, p. 240-241]. But optimization is a widely studied area and, thus, there are more efficient algorithms. Conjugate gradients makes implicit use of second order information. Newton's method or so called quasi-Newton methods explicitly use the Hessian of the error function in each iteration step. Some of these algorithms are described in detail in [7, p. 274-290]. We discuss the general idea of Newton's method.

Newton's method is an iterative optimization algorithm based on a quadratic approximation of E using the Taylor series. Let $H(w[t])$ denote the Hessian of the error function evaluated at $w[t]$. Then, by taking the first three terms of the Taylor series[6], we obtain a quadratic approximation of E around the point $w[t]$ [7, p. 105-106]:

$$\tilde{E}(w[t] + \Delta w[t]) = E(w[t]) + \nabla E(w[t])^{tr} \Delta w[t] + \frac{1}{2}(\Delta w[t])^{tr} H(w[t]) \Delta w[t]. \quad (25)$$

We want to choose the weight update $\Delta w[t]$ such that the quadratic approximation $\tilde{E}$ is minimized. Therefore, the necessary criterion gives us:

$$\nabla \tilde{E}(w[t] + \Delta w[t]) = \nabla E(w[t]) + H(w[t]) \Delta w[t] \overset{!}{=} 0. \quad (26)$$

The solution is given by

$$\Delta w[t] = -\gamma H(w[t])^{-1} \nabla E(w[t]) \quad (27)$$

where $H(w[t])^{-1}$ is the inverse of the Hessian at point $w[t]$. Because we use a quadratic approximation, equation (27) needs to be applied iteratively. When choosing $w[0]$ sufficiently close to the global minimum of E, Newton's method converges quadratically [7, p. 105-106].

Nevertheless, Newton's method has several drawbacks. As we see later, the evaluation of the Hessian matrix and its inversion is rather expensive[7]. In addition, the Newton step in equation (27) may be a step in the direction of a local maximum or saddle point. This may happen if the Hessian matrix is not positive definite [3, p. 285-287].

4.4 Error Backpropagation

Error backpropagation describes an efficient algorithm to evaluate ∇E_n for multilayer perceptrons using differentiable activation functions.

Following [5, p. 241-245] we consider the i^{th} output unit first. The derivative of an arbitrary error function E_n with respect to the weight $w_{ij}^{(L+1)}$ is given by

$$\frac{\partial E_n}{\partial w_{ij}^{(L+1)}} \underset{rule}{\overset{\text{chain-}}{=}} \frac{\partial E_n}{\partial z_i^{(L+1)}} \frac{\partial z_i^{(L+1)}}{\partial w_{ij}^{(L+1)}} \quad (28)$$

where $z_i^{(L+1)}$ denotes the actual input of the i^{th} unit within the output layer. Using the chain rule, the first factor in equation (28) can be written as

$$\begin{aligned} \delta_i^{(L+1)} := \frac{\partial E_n}{\partial z_i^{(L+1)}} &\underset{rule}{\overset{\text{chain-}}{=}} \frac{\partial E_n}{\partial y_i^{(L+1)}} \frac{\partial y_i^{(L+1)}}{\partial z_i^{(L+1)}} \\ &= \frac{\partial E_n}{\partial y_i^{(L+1)}} f'\left(z_i^{(L+1)}\right) \end{aligned} \quad (29)$$

[6] In general, the Taylor series of a smooth function f at point x_0 is given by

$$T(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k. \quad (24)$$

[7] The inversion of a $n \times n$ matrix usually requires $\mathcal{O}(n^3)$ operations.

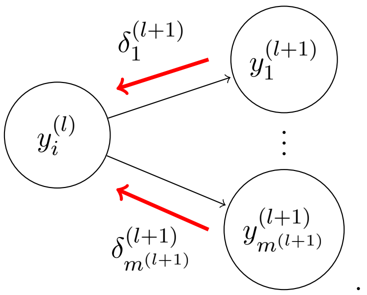


Figure 7: Once evaluated for all output units, the errors $\delta_i^{(L+1)}$ can be propagated backwards according to equation (34). The derivatives of all weights in layer l can then be determined using equation (31).

where $\delta_i^{(L+1)}$ is often called error and describes the influence of the i^{th} output unit on the total error E_n. The second factor of equation (28) takes the form

$$\begin{aligned}\frac{\partial z_i^{(L+1)}}{\partial w_{ij}^{(L+1)}} &= \frac{\partial}{\partial w_{ij}^{(L+1)}}\left[\sum_{k=0}^{m^{(L)}} w_{ik}^{(L+1)} y_k^{(L)}\right] \\ &= \frac{\partial}{\partial w_{ij}^{(L+1)}}\left[w_{ij}^{(L+1)} y_j^{(L+1)} + \underbrace{\sum_{k=0,k\neq i}^{m^{(L)}} w_{ik}^{(L+1)} y_k^{(L)}}_{const.}\right] \\ &= y_j^{(L)}. \end{aligned} \tag{30}$$

By substituting both factors (29) and (30) into equation (28) we get

$$\frac{\partial E_n}{\partial w_{ij}^{(L+1)}} = \delta_i^{(L+1)} y_j^{(L)}. \tag{31}$$

The errors $\delta_i^{(L+1)}$ are usually easy to compute, for example when choosing error function and output activation function as noted in section 4.1.

Considering the i^{th} hidden unit within an arbitrary hidden layer l we write

$$\frac{\partial E_n}{\partial w_{ij}^{(l)}} \overset{\text{chain-}}{\underset{rule}{=}} \frac{\partial E_n}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} \tag{32}$$

and notice that the second factor in equation (32) takes the same form as in equation (30):

$$\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}}\left[\sum_{k=0}^{m^{(l-1)}} w_{ik}^{(l)} y_k^{(l-1)}\right] = y_j^{(l-1)}. \tag{33}$$

Thus, only the error $\delta_i^{(l)}$ changes. Using the chain rule we get

$$\delta_i^{(l)} := \frac{\partial E_n}{\partial z_i^{(l)}} = \sum_{k=1}^{m^{(l+1)}} \frac{\partial E_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}. \tag{34}$$

In general, the sum runs over all units directly succeeding the i^{th} unit in layer l [5, p. 242-245]. But we assume that every unit in layer l is connected to every unit in layer $(l+1)$

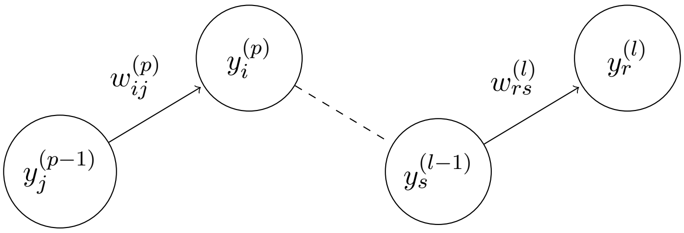


Figure 8: To derive an algorithm for evaluating the Hessian we consider two weights $w_{ij}^{(p)}$ and $w_{rs}^{(l)}$ where we assume that $p \leq l$. This figure illustrates the case when $p < (l-1)$.

where we allow the weights $w_{ki}^{(l+1)}$ to vanish. Altogether, by substituting the definition of $\delta_k^{(l+1)}$ from equation (34) we have

$$\delta_i^{(l)} = f'\left(z_i^{(l)}\right) \sum_{k=1}^{m^{(l+1)}} w_{ik}^{(l+1)} \delta_k^{(l+1)}. \tag{35}$$

In conclusion, we have found a recursive algorithm for calculating the gradient ∇E_n by propagating the errors $\delta_i^{(L+1)}$ back through the network using equation (35) and evaluating the derivatives $\frac{\partial E_n}{\partial w_{ij}^{(l)}}$ using equation (32).

4.4.1 Efficiency

The number of operations required for error backpropagation scales with the total number of weights W and the number of units. Considering a sufficiently large multilayer perceptron such that every unit in layer l is connected to every unit in layer $(l+1)$ the cost of propagating an input value through the network is dominated by W. Here we neglect the evaluation cost of the activation functions. Because of the weighted sum in equation (34), backpropagating the error is dominated by W, as well. This leads to an overall effort linear in the total number of weights: $\mathcal{O}(W)$ [5, p. 246-247].

4.5 The Hessian

Following [2] we derive an algorithm which allows us to evaluate the Hessian exactly by forward and backward propagation. Again, we assume differentiable activation functions within a multilayer perceptron.

We start by picking two weights $w_{ij}^{(p)}$ and $w_{rs}^{(l)}$ where we assume the layer p to come before layer l, that is $p \leq l$. The case where $p > l$ will follow by the symmetry of the Hessian [2]. Figure 8 illustrates this setting for $p < (l-1)$. As the error E_n is only depending on $w_{ij}^{(p)}$ through the actual input $z_i^{(p)}$ of the i^{th} unit in layer p we start by writing

$$\begin{aligned} \frac{\partial^2 E_n}{\partial w_{ij}^{(p)} \partial w_{rs}^{(l)}} &\overset{\text{chain-}}{\underset{\textit{rule}}{=}} \frac{\partial z_i^{(p)}}{\partial w_{ij}^{(p)}} \frac{\partial}{\partial z_i^{(p)}} \left(\frac{\partial E_n}{\partial w_{rs}^{(l)}} \right) \\ &\overset{(30)}{=} y_j^{(p-1)} \frac{\partial}{\partial z_i^{(p)}} \left(\frac{\partial E_n}{\partial w_{rs}^{(l)}} \right). \end{aligned} \tag{36}$$

Now we remember the definition of $\delta_r^{(l)}$ in equation (34) and add two more definitions:

$$g_{si}^{(l-1,p)} := \frac{\partial z_s^{(l-1)}}{\partial z_i^{(p)}} \text{ and } b_{ri}^{(l,p)} := \frac{\partial \delta_r^{(l)}}{\partial z_i^{(p)}}. \tag{37}$$

As E_n is depending on $w_{rs}^{(l)}$ only through the value of $z_r^{(l)}$ we use the chain rule and write

$$\begin{aligned}\frac{\partial^2 E_n}{\partial w_{ij}^{(p)} \partial w_{rs}^{(l)}} &= \frac{\partial}{\partial z_i^{(p)}} \left[y_j^{(p-1)} \frac{\partial z_r^{(l)}}{\partial w_{rs}^{(l)}} \frac{\partial E_n}{\partial z_r^{(l)}} \right] \\ &= \frac{\partial}{\partial z_i^{(p)}} \left[y_j^{(p-1)} y_s^{(l-1)} \frac{\partial E_n}{\partial z_r^{(l)}} \right] \\ &\overset{(34)}{=} \frac{\partial}{\partial z_i^{(p)}} \left[y_j^{(p-1)} y_s^{(l-1)} \delta_r^{(l)} \right] . \end{aligned} \tag{38}$$

Because the output $y_j^{(p-1)}$ of the j^{th} unit in layer $(p-1)$ is not depending on $z_i^{(p)}$, we can use the product rule and plug in the definitions from equation (37):

$$\begin{aligned}\frac{\partial^2 E_n}{\partial w_{ij}^{(p)} \partial w_{rs}^{(l)}} &\underset{\textit{rule}}{\overset{\text{product-}}{=}} y_j^{(p-1)} f'\left(z_s^{(l-1)}\right) \frac{\partial z_s^{(l-1)}}{\partial z_i^{(p)}} + y_j^{(p-1)} y_s^{(l-1)} \frac{\partial \delta_r^{(l)}}{\partial z_i^{(p)}} \\ &\overset{(37)}{=} y_j^{(p-1)} f'\left(z_s^{(l-1)}\right) g_{si}^{(l-1,p)} + y_j^{(p-1)} y_s^{(l-1)} b_{pi}^{(l,p)} . \end{aligned} \tag{39}$$

Following this derivation, Bishop suggests a simple algorithm to evaluate the $g_{si}^{(l-1,p)}$ and the $b_{ri}^{(l,p)}$ by forward and backward propagation. Using the chain rule we can write

$$\begin{aligned} g_{si}^{(l-1,p)} \overset{(37)}{=} \frac{\partial z_s^{(l-1)}}{\partial z_i^{(p)}} &\underset{\textit{rule}}{\overset{\text{chain-}}{=}} \sum_{k=1}^{m^{(l-2)}} \frac{\partial z_s^{(l-1)}}{\partial z_k^{(l-2)}} \frac{\partial z_k^{(l-2)}}{\partial z_i^{(p)}} \\ &= \sum_{k=1}^{m^{(l-2)}} f'\left(z_k^{(l-2)}\right) w_{sk}^{(l-1)} g_{ki}^{(l-2)} . \end{aligned} \tag{40}$$

Initially we set $g_{ii}^{p,p} = 1$ and $g_{si}^{l,p} = 0$ for $l \leq p$. Thus, the $g_{si}^{(l-1,p)}$ can be obtained by forward propagating them through the network. Using the definition of $\delta_r^{(l)}$ from equation (34), the product rule and the definitions from equation (37) we write

$$\begin{aligned} b_{ri}^{(l,p)} &\overset{(37)}{=} \frac{\partial \delta_r^{(l)}}{\partial z_i^{(p)}} \overset{(34)}{=} \frac{\partial f'\left(z_r^{(l)}\right)\left(\sum_{k=1}^{m^{(l+1)}} w_{kr}^{(l+1)} \delta_k^{(l+1)}\right)}{\partial z_i^{(p)}} \\ &\underset{\textit{rule}}{\overset{\text{product-}}{=}} f''\left(z_r^{(l)}\right) \frac{\partial z_r^{(l)}}{\partial z_i^{(p)}} \left[\sum_{k=1}^{m^{(l+1)}} w_{kr}^{(l+1)} \delta_k^{(l+1)} \right] + f'\left(z_r^{(l)}\right) \left[\sum_{k=1}^{m^{(l+1)}} w_{kr}^{(l+1)} \frac{\partial \delta_k^{(l+1)}}{\partial z_i^{(p)}} \right] \\ &\overset{(37)}{=} f''\left(z_r^{(l)}\right) g_{ri}^{(l,p)} \left[\sum_{k=1}^{m^{(l+1)}} w_{kr}^{(l+1)} \delta_k^{(l+1)} \right] + f'\left(z_r^{(l)}\right) \left[\sum_{k=1}^{m^{(l+1)}} w_{kr}^{(l+1)} b_{ki}^{(l+1,p)} \right] . \end{aligned} \tag{41}$$

Using equation (41) we can backpropagate the $b_{ri}^{(l,p)}$ after evaluating $b_{ki}^{(L+1,p)}$ for all output units k:

$$\begin{aligned}
b_{ki}^{(L+1,p)} &\overset{(37)}{=} \frac{\partial \delta_k^{(L+1)}}{\partial z_i^{(p)}} \overset{(34)}{=} \frac{\partial \left(\frac{\partial E_n}{\partial z_k^{(L+1)}} \right)}{\partial z_i^{(p)}} \\
&= \frac{\partial^2 E_n}{\partial^2 z_k^{(L+1)}} \frac{\partial z_k^{(L+1)}}{\partial z_i^{(p)}} = \frac{\partial^2 E_n}{\partial^2 z_k^{(L+1)}} g_{ki}^{(L+1,p)} \\
&= g_{ki}^{(L+1),p)} \left[f'' \left(z_k^{(L+1)} \right) \frac{\partial E_n}{\partial y_k^{(L+1)}} + f' \left(z_k^{(L+1)} \right) \frac{\partial^2 E_n}{\partial \left(y_k^{(L+1)} \right)^2} \right].
\end{aligned} \tag{42}$$

The errors $\delta_r^{(l)}$ can be evaluated as seen in section 4.4 using error backpropagation.

Altogether, we are able to evaluate the Hessian of E_n by forward propagating the $g_{si}^{(l-1,p)}$ and backward propagating the $b_{ri}^{(l,p)}$ as well as the errors $\delta_r^{(l)}$ [2].

4.5.1 Efficiency

We require one step of forward propagation and backward propagation per unit within the network. Again, assuming a sufficiently large network such that the number of weights W dominates the network, the evaluation of equation (39) requires $\mathcal{O}(W^2)$ operations.

4.6 Regularization

Now we focus on the ability of generalization. We saw that theoretically a neural network with a sufficiently large number of hidden units can model every continuous target function. The ability of a network to generalize can be observed using unseen test data not used for training, usually called a validation set. If the trained approximation of the target function works well for the validation set the network is said to have good generalization capabilities [8, p. 227-228]. Overtraining of the network may result in over-fitting of the training set, that is the network memorizes the training set but performs very poorly on the validation set [8, p. 227-228].

Regularization tries to avoid over-fitting and to give a better generalization performance. This can be achieved by controlling the complexity of the network which is mainly determined by the number of hidden units. The simplest form of regularization is adding a regulizer to the error function such that

$$\hat{E}(w) = E(w) + \eta P(w) \tag{43}$$

where $P(w)$ is a penalty term which tries to influence the form and complexity of the solution [3, p. 338].

4.6.1 L2-Regularization

Large weights result in an approximation with poor generalization capabilities. Therefore we try to penalize large weights such that we have [3, p. 338-343]:

$$\hat{E}(w) = E(w) + \eta w^T w. \tag{44}$$

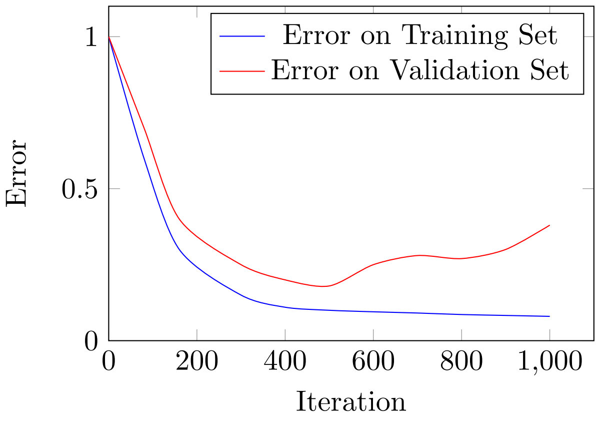


Figure 9: The error on the validation set (red) is usually getting larger when the network begins to overfit the training set. Thus, although the error on the training set (blue) is decreasing monotonically, we may want to stop training early to avoid overfitting.

This regulizer is also referred to as weight decay. For understanding why this regulizer is called weight decay, we consider the derivative of the error function $\hat{E}$ which is given by

$$\nabla \hat{E} = \nabla E + \eta w. \tag{45}$$

Neglecting ∇E and considering gradient descent learning the change in w with respect to the iteration step t can be described as

$$\frac{\partial}{\partial t} w[t] = -\gamma \eta w[t]. \tag{46}$$

Equation (46) has the solution $w[t] = w[0]e^{-\gamma\eta t}$ such that the weights tend exponentially to zero – giving the method its name [3, p. 338-340]. While weight decay represents L2-regularization we could also consider L1-regularization.

4.6.2 Early Stopping

Early stopping describes the approach to stop training before gradient descent has finished. The error on the training set is usually monotonically decreasing with respect to the iteration number. For unseen data the error is usually much higher and not monotonically decreasing. It tends to get larger when we reach a state of over-fitting. Therefore, it seems reasonable to stop training at a point where the error on the validation set reaches a minimum [3, p. 343-345]. This is illustrated in figure 9.

5 Pattern Classification

In this section we have a closer look on pattern classification. The classification problem can be defined as follows: Given a D-dimensional input vector x assign it to one of C discrete classes. We refer to a given class by its number c which lies in the range $1 \leq c \leq C$. The input vector x is also called pattern or observation.

5.1 Statistical Background

Using a statistical approach we assume the pattern x and the class c to be random variables. Then $p(x)$ is the probability that we observe the pattern x and $p(c)$ is the probability that we observe a pattern belonging to class c [17, p. 108-109].

The classification task is to assign a pattern x to its corresponding class. This can be accomplished by considering the class-conditional probability $p(c|x)$, that is the probability of pattern x belonging to class c [17, p. 108-109]. By applying Bayes' theorem we can rewrite the class-conditional probability to give

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}. \tag{47}$$

Then the class-conditional probability can be interpreted as posterior probability where $p(c)$ is the prior probability. The prior probability represents the probability of class c before making an observation. The posterior probability describes the probability of class c after observing the pattern x. [5, p. 38-39].

5.2 Bayes' Decision Rule

Given the posterior probabilities $p(c|x)$ for all classes c we can determine the class of x using a decision rule. But we want to avoid decision errors. A decision error occurs if an observation vector x is assigned to the wrong class. Bayes' decision rule given by

$$c : \mathbb{R}^D \rightarrow \{1, \ldots, C\}, x \mapsto \underset{1 \leq c \leq C}{argmax} \{p(c|x)\} \tag{48}$$

results in a minimum number of decision errors [17, p. 109]. But it assumes the true posterior probabilities to be known.

In practice the posterior probabilities $p(c|x)$ are unknown. Therefore, we may model the probability $p(c|x)$ directly (or indirectly by modeling $p(x|c)$ first) in the means of a so called model distribution $q_\theta(c|x)$ which depends on some parameters θ [18, p. 637-638]. Then we apply the model-based decision rule given by

$$c_\theta : \mathbb{R}^D \rightarrow \{1, \ldots, C\}, x \mapsto \underset{1 \leq c \leq C}{argmax} \{q_\theta(c|x)\}. \tag{49}$$

Here we use the model distribution and, thus, the decision rule is fully dependent on the parameters θ [18, p. 638].

5.3 Maximum Likelihood Estimation

Using a maximum likelihood approach we can estimate the unknown parameters of the model distribution. Therefore, we assume that the posterior probabilities are given by the model distribution $q_\theta(c|x)$ and the patterns x_n are drawn independently from the distribution $q_\theta(c|x)$. We say the data points x_n are independent and identically distributed [6, p. 85-88]. Then the likelihood function is given by

$$\mathcal{L}(\theta) = \prod_{n=1}^{N} q_\theta(c_n|x_n). \tag{50}$$

where c_n denotes the corresponding class of pattern x_n. We then want to maximize $\mathcal{L}(\theta)$. This is equivalent to minimizing the negative log-likelihood function which is given by

$$-\log(\mathcal{L}(\theta)) = -\sum_{n=1}^{N} \log(q_\theta(x_n|c)). \tag{51}$$

Then, we can use the negative log-likelihood function as error function to train a neural network. The sum-of-squared error function and the cross-entropy error function can both be motivated using a maximum likelihood approach. We derive only the cross-entropy error function in the next section.

5.3.1 Derivation of Cross-Entropy

We follow [5, p. 232-236] and consider the multiclass[8] classification problem. The target values t_n follow the 1–of–C encoding scheme, that is the i^{th} entry of t_n equals 1 iff the pattern x_n belongs to class i.

We interpret the output of the neural network as the posterior probabilities

$$p(c|x) = y_c(x, w). \tag{52}$$

The posterior probability for class c can then be written as

$$p(c|x) = \frac{p(x|c)p(c)}{\sum_{k=1}^{C} p(x|k)p(k)}. \tag{53}$$

As output activation function we use the softmax function such that we can rewrite equation (53) to give

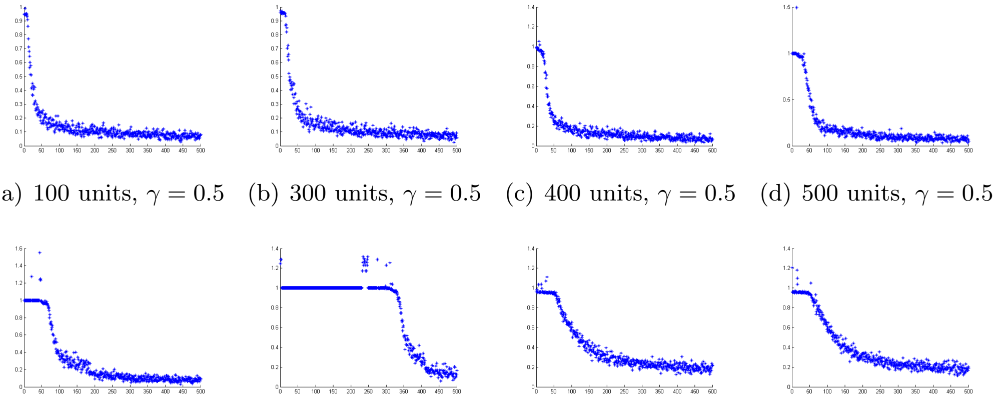
$$p(c|x) = \frac{\exp(z_c)}{\sum_{k=1}^{c} \exp(z_k)} \text{ with } z_k = \log(p(k|x)p(k)). \tag{54}$$

Here we model the posteriors in the means of the network output. This means that the parameters of the model distribution are given by the weights of the network. Given the probabilities $p(c|x)$ the maximum likelihood function takes the form

$$\mathcal{L}(w) = \prod_{n=1}^{N} \prod_{c}^{C} p(c|x_n)^{t_{nc}} = \prod_{n=1}^{N} \prod_{c=1}^{C} y_c(x_n, w)^{t_{nc}} \tag{55}$$

and we can derive the error function $E(w)$ which is given by the cross-entropy error function already introduced in section 4.1:

$$E(w) = -\log(\mathcal{L}(w)) = -\sum_{n=1}^{N} \sum_{c=1}^{C} t_{nc} ln(y_c(w, x_n)). \tag{56}$$



(a) 100 units, $\gamma = 0.5$ (b) 300 units, $\gamma = 0.5$ (c) 400 units, $\gamma = 0.5$ (d) 500 units, $\gamma = 0.5$

(e) 600 units, $\gamma = 0.5$ (f) 700 units, $\gamma = 0.5$ (g) 750 units, $\gamma = 0.1$ (h) 800 units, $\gamma = 0.1$

Figure 10: The error during training for different learning rates γ and numbers of hidden units. The two-layer perceptron was trained with a batchsize of 100 randomly chosen images iterated 500 times. The error was measured using the sum-of-squared error function and plotted after each iteration.

5.4 Application: Recognizing Handwritten Digits

Based on the MNIST dataset (available online at http://yann.lecun.com/exdb/mnist/) we want to train a two-layer perceptron to recognize handwritten digits. The dataset provides a training set of $60,000$ images and a validation set of $10,000$ images. The images have 28×28 pixels of size and, thus, we may write an image as vector with $28 \cdot 28 = 784$ dimensions. The neural network will be trained using gradient descent for parameter optimization and the sum-of-squared error function. The training procedure is implemented in MatLab.

5.4.1 Matrix Notation

For implementing error backpropagation in MatLab it is convenient to introduce a matrix notation. We discuss the general case of a multilayer perceptron with L layers. The output of all units in layer l and their actual input can both be combined in vectors:

$$y^{(l)} := \left(y_i^{(l)}\right)_{i=1}^{m^{(l)}} \text{ and } z^{(l)} := \left(z_i^{(l)}\right)_{i=1}^{m^{(l)}}. \tag{57}$$

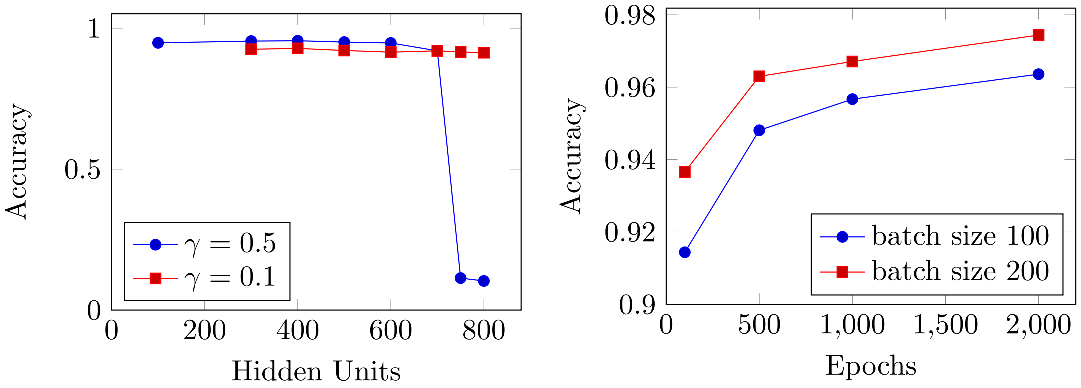
When combining all weights $w_{ij}^{(l)}$ of layer l in a single weight matrix we can express the propagation of an input vector through the network as several vector matrix products. The weight matrix $W^{(l)}$ is given by

$$W^{(l)} := \left(w_{ij}^{(l)}\right)_{i,j=1}^{m^{(l)},m^{(l-1)}}. \tag{58}$$

Propagating the output vector of layer $(l-1)$ denoted by $y^{(l-1)}$ through layer l can then be written as

$$y^{(l)} = f\left(W^{(l)}y^{(l-1)}\right) \tag{59}$$

[8] Generally, we distinguish the binary classification problem in the case of $C = 2$ and the multiclass classification problem for $C > 2$.



(a) 500 epochs with batch size 100.

(b) 500 epochs with learning rate $\gamma = 0.5$.

Figure 11: Figure 11(a) plots the achieved accuracy depending on the number of hidden units for a fixed learning rate $\gamma = 0.5$. For 750 or more units we need to adjust the learning rate in order to ensure convergence. But this may result in slow convergence such that we get worse results for few hidden units. Figure 11(b) shows the impact of increasing the number of epochs. As we would expect, the accuracy increases with rising training set.

where f operates element by element. Given the error vector $\delta^{(l+1)}$ defined as

$$\delta^{(l+1)} := \left(\delta_i^{(l+1)}\right)_{i=1}^{m^{(l+1)}} \tag{60}$$

for layer $(l+1)$ we can use the transpose of the weight matrix to give

$$\delta^{(l)} = f'(z^{(l)}) \bullet \left(W^{(l)}\right)^{tr} \delta^{(l+1)} \tag{61}$$

where f' operates element by element and $\bullet$ denotes the component-wise multiplication.

5.4.2 Implementation and Results

The MNIST dataset is provided in the IDX file format. To get the images as vector with 784 dimensions we use two functions `loadMNISTImages` and `loadMNISTLabels` which can be found online at http://ufldl.stanford.edu/.

Appendix A.2 shows the implementation of the training procedure. The activation function and its derivative are passed as parameters. We use the logistic sigmoid, which is shown in appendix A.1, as activation function for both the hidden and the output layer. The number of hidden units and the learning rate are adjustable. The weight matrices for both layers are initialized randomly and normalized afterwards (lines 29 to 35). We apply stochastic training in so called epochs, that is for each epoch we train the network on randomly selected input values (lines 42 to 61). The number of randomly selected input values to use for training is defined by the parameter `batchSize`. After each epoch we plot the current error on the same input values as the network was trained on (lines 63 to 73).

For comparing different configurations we use the validation set. The validation procedure, which counts the number of correctly classified images, is shown in appendix A.3. The accuracy is measured as the quotient of correctly classified images and the total number of images. Figure 11 shows some of the results.

6 Conclusion

In the course of this paper we introduced the general concept of artificial neural networks and had a closer look at the multilayer perceptron. We discussed several activation functions and network topologies and explored the expressive power of neural networks.

Section 4 introduced the basic notions of network training. We focussed on supervised training. Therefore, we introduced two different error measure to evaluate the performance of the network in respect to function approximation. To train the network we introduced gradient descent as iterative parameter optimization algorithm as well as Newton's method as second-order optimization algorithm. The weights of the network are adjusted iteratively to minimize the chosen error measure. To evaluate the gradient of the error measure we introduced the error backpropagation algorithm. We extended this algorithm to allow the exact evaluation of the Hessian, as well.

Thereafter, we discussed the classification problem using a statistical approach. Based on maximum likelihood estimation we derived the cross-entropy error measure for multiclass classification. As application we considered digit recognition based on the MNIST dataset using a two-layer perceptron.

Unfortunately this paper is far too short to cover the extensive topic of neural networks and their application in pattern recognition. Especially for network training there are more advanced techniques available (see for example [3]). Instead of the exact evaluation of the Hessian described in [2], approximations to the Hessian or its inverse are more efficient to compute (for example described in [3]). Based on a diagonal approximation of the Hessian there are advanced regularization methods as for example the Optimal Brain Damage algorithm [16].

In conclusion, artificial neural networks are a powerful tool applicable to a wide range of problems especially in the domain of pattern recognition.

Appendices

A MatLab Implementation

A.1 Logistic Sigmoid and its Derivative

```
function y = logisticSigmoid(x)
% simpleLogisticSigmoid Logistic sigmoid activation function
%
% INPUT:
% x     : Input vector.
%
% OUTPUT:
% y     : Output vector where the logistic sigmoid was applied element by
% element.
%

    y = 1./(1 + exp(-x));
end
```

```
function y = dLogisticSigmoid(x)
% dLogisticSigmoid Derivative of the logistic sigmoid.
%
% INPUT:
% x     : Input vector.
%
% OUTPUT:
% y     : Output vector where the derivative of the logistic sigmoid was
% applied element by element.
%
    y = logisticSigmoid(x).*(1 - logisticSigmoid(x));
end
```

A.2 Training Procedure

```
function [hiddenWeights, outputWeights, error] =
    trainStochasticSquaredErrorTwoLayerPerceptron(activationFunction,
    dActivationFunction, numberOfHiddenUnits, inputValues, targetValues, epochs,
    batchSize, learningRate)
% trainStochasticSquaredErrorTwoLayerPerceptron Creates a two-layer perceptron
% and trains it on the MNIST dataset.
%
% INPUT:
% activationFunction          : Activation function used in both layers.
% dActivationFunction         : Derivative of the activation
% function used in both layers.
% numberOfHiddenUnits         : Number of hidden units.
% inputValues                 : Input values for training (784 x 60000)
% targetValues                : Target values for training (1 x 60000)
% epochs                      : Number of epochs to train.
% batchSize                   : Plot error after batchSize images.
% learningRate                : Learning rate to apply.
%
% OUTPUT:
% hiddenWeights               : Weights of the hidden layer.
% outputWeights               : Weights of the output layer.
%

```

```
    % The number of training vectors.
    trainingSetSize = size(inputValues, 2);

    % Input vector has 784 dimensions.
    inputDimensions = size(inputValues, 1);
    % We have to distinguish 10 digits.
    outputDimensions = size(targetValues, 1);

    % Initialize the weights for the hidden layer and the output layer.
    hiddenWeights = rand(numberOfHiddenUnits, inputDimensions);
    outputWeights = rand(outputDimensions, numberOfHiddenUnits);

    hiddenWeights = hiddenWeights./size(hiddenWeights, 2);
    outputWeights = outputWeights./size(outputWeights, 2);

    n = zeros(batchSize);

    figure; hold on;

    for t = 1: epochs
        for k = 1: batchSize
            % Select which input vector to train on.
            n(k) = floor(rand(1)*trainingSetSize + 1);

            % Propagate the input vector through the network.
            inputVector = inputValues(:, n(k));
            hiddenActualInput = hiddenWeights*inputVector;
            hiddenOutputVector = activationFunction(hiddenActualInput);
            outputActualInput = outputWeights*hiddenOutputVector;
            outputVector = activationFunction(outputActualInput);

            targetVector = targetValues(:, n(k));

            % Backpropagate the errors.
            outputDelta = dActivationFunction(outputActualInput).*(outputVector -
                targetVector);
            hiddenDelta =
                dActivationFunction(hiddenActualInput).*(outputWeights'*outputDelta);

            outputWeights = outputWeights - learningRate.*outputDelta*hiddenOutputVector';
            hiddenWeights = hiddenWeights - learningRate.*hiddenDelta*inputVector';
        end;

        % Calculate the error for plotting.
        error = 0;
        for k = 1: batchSize
            inputVector = inputValues(:, n(k));
            targetVector = targetValues(:, n(k));

            error = error +
                norm(activationFunction(outputWeights*activationFunction(hiddenWeights*inputVector))
                - targetVector, 2);
        end;
        error = error/batchSize;

        plot(t, error,'*');
    end;
end
```

A.3 Validation Procedure

```
function [correctlyClassified, classificationErrors] =
    validateTwoLayerPerceptron(activationFunction, hiddenWeights, outputWeights,
    inputValues, labels)
% validateTwoLayerPerceptron Validate the twolayer perceptron using the
% validation set.
%
% INPUT:
% activationFunction          : Activation function used in both layers.
% hiddenWeights               : Weights of the hidden layer.
% outputWeights               : Weights of the output layer.
% inputValues                 : Input values for training (784 x 10000).
% labels                      : Labels for validation (1 x 10000).
%
% OUTPUT:
% correctlyClassified         : Number of correctly classified values.
% classificationErrors        : Number of classification errors.
%

    testSetSize = size(inputValues, 2);
    classificationErrors = 0;
    correctlyClassified = 0;

    for n = 1: testSetSize
        inputVector = inputValues(:, n);
        outputVector = evaluateTwoLayerPerceptron(activationFunction, hiddenWeights,
            outputWeights, inputVector);

        class = decisionRule(outputVector);
        if class == labels(n) + 1
            correctlyClassified = correctlyClassified + 1;
        else
            classificationErrors = classificationErrors + 1;
        end;
    end;
end

function class = decisionRule(outputVector)
% decisionRule Model based decision rule.
%
% INPUT:
% outputVector    : Output vector of the network.
%
% OUTPUT:
% class           : Class the vector is assigned to.
%

    max = 0;
    class = 1;
    for i = 1: size(outputVector, 1)
        if outputVector(i) > max
            max = outputVector(i);
            class = i;
        end;
    end;
end

function outputVector = evaluateTwoLayerPerceptron(activationFunction, hiddenWeights,
    outputWeights, inputVector)
% evaluateTwoLayerPerceptron Evaluate two-layer perceptron given by the
% weights using the given activation function.
%
% INPUT:
% activationFunction          : Activation function used in both layers.
% hiddenWeights               : Weights of hidden layer.
% outputWeights               : Weights for output layer.
% inputVector                 : Input vector to evaluate.
%
% OUTPUT:
% outputVector                : Output of the perceptron.
%

    outputVector =
        activationFunction(outputWeights*activationFunction(hiddenWeights*inputVector));
end
```

References

[1] Sue Becker and Yann LeCun. Improving the convergence of back-propagation learning with second order methods. Technical report, University of Toronto, Toronto, 1988.

[2] Christopher M. Bishop. Exact calculation of the hessian matrix for the multi-layer perceptron. *Neural Computation*, 4, 1992.

[3] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[4] Christopher M. Bishop. Neural Networks: A Pattern Recognition Perspective. Technical report, Aston University, Birmingham, UK, 1996.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer Verlag, New York, 2006.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* Wiley-Interscience Publication, New York, 2001.

[7] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical optimization.* Academic Press, London, 1981.

[8] Simon Haykin. *Neural Networks A Comprehensive Foundation.* Pearson Education, New Delhi, 2005.

[9] Geoffrey E. Hinton. Learning distributed representations of concepts. 1986.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 1989.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3, 1990.

[12] Yann LeCun. *Modeles connexionistes de l'apprentissage.* PhD thesis, 1987.

[13] Yann LeCun. A theoretical framework for back-propagation. 1988.

[14] Yann LeCun. Generalization and network design strategies. Technical report, University of Toronto, Toronto, 1989.

[15] Yann LeCun, Léon Buttou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition, 1998.

[16] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage, 1990.

[17] Hermann Ney. On the probabilistic interpretation of neural network classifiers and discriminative training criteria. *IEEE Transactions on Pattern Alaysis and Machine Intelligence*, 17, 1995.

[18] Hermann Ney. On the relationship between classification error bounds and training criteria in statistical pattern recognition. In *Iberian Conference on Pattern Recognition and Image Analysis*, pages 636–645, Puerto de Andratx, Spain, 2003.

[19] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1958.

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323, 1986.