Neuronal Networks
Formulae Collection

Marvin Ritter
marvin.ritter@gmail.com

20th September 2015

About this document

This is my personal collection of formulae in the field of neural networks. Where I felt the need for it I also added explanations and derivations. I started it in preparation to my exam on the neural nets at Karlsruhe Institute of Technology (KIT). Even so the content is similar to the course topics it is neither limited to it nor guaranteed to cover it completely. *These are not official lecture notes and may contain errors and lag completeness.*

Corrections, supplements (or wishes for it) and links to good sources/papers are very welcome. Just mail me to marvin.ritter@gmail.com or create a ticket/pull request.

Contents

Chapter 1

Introduction

1.1 Notation

Throughout the literature (and history) in the big field of neural networks you will find a whole bunch of different notations and meanings of variable names. Especially when the meaning changes inside a document it can be very confusing for beginners. For this document I will try to stick with one notation. I found it to be very flexible and consistent.

Symbol	Meaning(s)
m	Number of features, size of input vector
k, K	Number of target classes/clusters, size of output vector
n, N	Number of training examples/observations, size of training set
η	Learning rate (see 4.4)
θ	Represents all model parameters
X	Training data
μ_j	Centroid of cluster j
$d(x, y)$	Dissimilarity measure, distance or distortion measure, if mention otherwise we use the squared Euclidean distance (see 3.1)
E, J	Error function (see 4.2), other names are: loss function, objective function or criterion function By default we will use the mean squared error.
$w_{ij}^{(l)}$	Weight of connection from neuron i in layer $(l-1)$ to neuron j in layer l, if the layer is omitted all variables in the equation are from the same layer
$\mathbf{W}$, $\mathbf{w}$	Weight matrix, weight vector
$b_j^{(l)}$	bias unit for neuron j in layer l For convenience we will use different meanings throughout the lecture notes: • Often the bias is not important for the understanding and left out, but keep in mind that neural network implementation nearly always require bias for good results. • b has an associated weight (usually w_{0j}), than the value of b is always 1 and we only train the weight • if there is no weight, we adjust b_j during training
Σ	Summation operator (e. g. $\sum_{i=1}^{n} x_i$), covariance matrix or the input function of a neuron
$\phi(x)$	Activation function (see 4.3)
$\sigma(x)$	Sigmoid activation function (see 4.3.2)
$z_j^{(l)}$	Total input of the neuron j in layer l, usually $z_j^{(l)} = b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)}$
$a_j^{(l)}$	Activation/Output of neuron j in layer l, usually computed from the total input $z_j^{(l)}$, $a_j = \phi(z_j)$
$v_j \in \{0, 1\}$	Activation of visible neuron/unit j in Boltzmann machine
$h_j \in \{0, 1\}$	Activation of hidden neuron/unit j in Boltzmann machine
$b_j (c_j)$	Bias of visible (hidden) neuron j in RBM
δ_{ij}	Kronecker delta, which is defined as $\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$

1.2 Other sources

Todo: Link some generally good sources (demos, tutorials, online courses, books).

Also have a look at the references at the end of the document.

1.3 Software

Todo: List with of good libraries for Machines Learning and Deep Neural Networks

Chapter 2

Unsupervised Vector Quantization

Also called *Clustering*.

Notation used in this chapter:

- N observations/training examples, $x_1, ..., x_N$
- fixed number of clusters/classes K
- $C(i) = K$ assigns observation x_i to a cluster k (typical each observation is only in one cluster)
- μ_k centroid of cluster k
- N_k number of observations that belong to cluster k ($N_k = \sum_{C(i)=k} 1$)
- $d(x, y)$ is a dissimilarity measure (usually the squared Euclidean distance[1]
- u_{ik} degree of membership of training example i to cluster k (if used in algorithm)
- $m \geq 1$ the 'fuzzifier' parameter for Fuzzy k-Means (2.3). Influences sharpness of the clusters, for $m \to \inf$ u_{ik} will be close to $\frac{1}{K}$ (not sharp). For m close to 1 the u_{ik}s will be close to 0 or 1. Typical values are between 1 and 2.5.

2.1 Loss Functions for Clustering

- Intra-class scatter: $W(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)=k} d(x_i, x_j)$
- Inter-class scatter: $B(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)\neq k} d(x_i, x_j)$
- Total scatter: $T(C) = W(C) + B(C) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} d(x_i, x_j)$
- Minimizing $W(C)$ is equivalent to maximizing $B(C)$

[1] Squared Euclidean distance $d(x, y) = ||x - y||^2 = (x - y)^T (x - y) = \sum_i (x_i - y_i)^2$

2.2 k-Means

Finding $C^*(i)$ by enumeration is too time-consuming. Instead use iterative greedy descent which leads to a local optima. The loss function is defined as

$$J = \sum_{k=1}^{K} \sum_{C(i)=k} d(x_i, \mu_k) \tag{2.1}$$

Using the squared Euclidean distance, $d(x_i, \mu_k) = ||x_i - \mu_k||^2$, will effectively assign each sample to the closet center.

A slightly different definition is to minimize our already defined $W(C)$

$$W(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)=k} ||x_i - x_j||^2 = \sum_{k=1}^{K} N_k \sum_{C(i)=k} ||x_i - \mu_k||^2 \tag{2.2}$$

Also called *Lloyd's Algorithm* we optimize for J in a greedy fashion.

1. **Classify**: Assign each observation i to the nearest centroid:
$$C(i) = \arg\min_{1 \leq k \leq K} ||x_i - \mu_k||^2$$
2. **Recenter**: For each class k, compute a new centroid as the mean of the updated class assignments:
$$\mu_k = \frac{\sum_{C(i)=k} x_i}{N_k}$$
3. **Repeat step 1 and 2 until stopping criteria fulfilled (e. g. centers stop moving/ C(i) unchanged).**

During the course of the k-means algorithm, the loss function monotonically decreases (in both steps) into a local minimum.

2.3 Fuzzy k-Means

Literature: [Introduction, 2000]

Observation does not belong strictly to one cluster, but has a member ship degree u_{ik} for each cluster k.

TODO: Do we need $\sum_{k=1}^{K} u_{ik} = 1$?

Our new loss function is:

$$J_m = \sum_{i=1}^{N} \sum_{k=1}^{K} u_{ik} d(x_i, \mu_k) \tag{2.3}$$

1. **Compute degree of membership:**
$$u_{ik} = \left[\sum_{j=1}^{K} \left(\frac{d(x_i, \mu_k)}{d(\mu_j, \mu_k)} \right)^{\frac{2}{m-1}} \right]^{-1}$$
2. **Recenter:**
$$\mu_k = \frac{\sum_{i=1}^{N} u_{ik}^m x_i}{\sum_{i=1}^{N} u_{ik}^m}$$
3. **Repeat step 1 and 2 until stopping criteria fulfilled.**

2.4 LBG-Algorithm

Original Paper: [Linde et al., 1980]
Demo: external website

The Linde, Buzo, Grey (LBG) Algorithm is an alternative method to design a K-vector codebook, where $K = 2^x$.
The algorithm takes a parameter a small $\epsilon > 0$ (e. g. $\epsilon = 0.001$).

1. **Initialize** 1-vector starting codebook:

$$\mu_1(0) = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2.4}$$

$$t = 0 \tag{2.5}$$

2. **Double codebook**:

$$\mu_j(t+1) = (1+\epsilon)\,\mu_j(t) \quad \text{for } j = 1, \cdots, 2^t \tag{2.6}$$

$$\mu_{2*j+1}(t+1) = (1-\epsilon)\,\mu_j(t) \quad \text{for } j = 1, \cdots, 2^t \tag{2.7}$$

$$t = t+1 \tag{2.8}$$

3. **Run k-Means** Use k-Means to optimize to improve the $\mu_j(t)$.
4. **Repeat step 2 and 3** until desired number of codebook vectors is reached (for $t = 1, \cdots, \log_2(K) - 1$)

Our final codebook will be $\mu_j(\log_2(K))$.

2.5 Self-Organizing Maps

Sometimes also called self-organizing feature maps.

Chapter 3

Supervised Vector Quantization

3.1 Distortion Measures

Literature: [Linde et al., 1980]

Distortion is an nonnegative measure for an input vector x and a reproduction vector y. Distortion measures may not be true metrics, e. g. be unsymmetric or not fulfill the triangle inequality[1].

Most common is the squared-error distortion, better known as squared Euclidean distance:

$$d(x,y) = ||x-y||_2^2 = (x-y)^T(x-y) = \sum_{i=1}^{N}(x_i - y_i)^2 \tag{3.1}$$

Other common distortion measures are the l_ν, or Holder norm,

$$d(x,y) = \left(\sum_{i=1}^{N}|x_i - y_i|^\nu\right)^{\frac{1}{\nu}} = ||x-y||_\nu \tag{3.2}$$

and its ν^{th} power, the ν^{th}-law distortion:

$$d(x,y) = \sum_{i=1}^{N}|x_i - y_i|^\nu \tag{3.3}$$

The holder Norm (3.2) is a distance and fulfills the triangle inequality, the ν^{th}-law distortion not.

All three and many others, as the weighted-squares distortion and the quadratic distortion, only depend on the difference $(x-y)$ and can be described with $d(x,y) = L(x-y)$. A distortion not having this form is the one by Itakura, Saito and Chaffee,

$$d(x,y) = (x-y)\,R(x)\,(x-y)^T \tag{3.4}$$

, where $R(x)$ is the autocorrelation matrix.

[1] triangle inequality: $d(x,y) \leq d(x,y) + d(y,y)$, for all y

3.2 Learning Vector Quantization

Literature: [Kohonen, 2001, Chapter 6] and [Kohonen, 1990]

Learning vector quantization (LVQ) are used for statistical classification. Supervised learning is used to train multiple codebook vector per class. The class of new observations is then determined by the class of the closest codebook vector.

Kohonen published several versions of LVQ that only differ slightly an the conditions and formulae for updates during the training.

Notation:

- discrete time domain with $t = 0, 1, 2, \cdots$
- $x(t)$ is an input sample
- m_i our vector vectors and $m_i(t)$ their sequential values
- classes $S_1, \cdots, S_K$
- each codebook vectors belongs to exactly one class, but each class has multiple codebook vectors
- $d(x, y)$ is a distance measure (usually Euclidean distance is used)
- $c = \arg\min_i d(x, m_i)$ is the index of the closest codebook vector
- δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ for $i = j$, $\delta_{ij} = 0$ for $i \neq j$)
- $\eta(t)$ learning rate ate time t
- w window width (see LVQ2)

For convenience we also define $s(t)$ as

$$s(t) = \begin{cases} +1 & \text{if } x \text{ and } m_c \text{ belong to the same class} \\ -1 & \text{if } x \text{ and } m_c \text{ belong to different classes} \end{cases} \tag{3.5}$$

3.2.1 LVQ1

Update closest codebook vector m_c depending on whether it has the same class as x or not.

If x and $m_c(t)$ belong to the same class set

$$m_c(t+1) = m_c(t) + \eta(t)(x - m_c(t)) \tag{3.6}$$

If x and $m_c(t)$ belong to different classes

$$m_c(t+1) = m_c(t) - \eta(t)(x - m_c(t)) \tag{3.7}$$

Leave the other codebook vectors unchanged

$$m_i(t+1) = m_i(t) \text{ for } i \neq c \tag{3.8}$$

Or in a compressed form:

$$m_i(t+1) = m_i(t) + \eta(t)s(t)\delta_{ci}(x(t) - m_i(t)) \tag{3.9}$$

3.2.2 OLVQ

Determine the optimal learning rate $\eta_i(t)$ for each codebook vector m_i for fastest convergence.

$$\eta_c(t) = \frac{\eta_c(t-1)}{1 + s(t)\eta_c(t-1)} \tag{3.10}$$

Modify update rule from LVQ1.

$$m_i(t+1) = [1 - \eta_i(t)s(t)\delta_{ci}]m_i(t) + \eta_i(t)s(t)\delta_{ci}x(t) \tag{3.11}$$

This modification will not work for LVQ2, but could be applied to LVQ3.

3.2.3 LVQ2.1

m_i and m_j are the two closest codebook vectors to x. One of them must belong to the correct class x and the other to a different class. Moreover, x must be inside a 'window'. The window is defined around the mid-plane of m_i and m_j and depends on the window width w.

$$\min\{\frac{d(x,m_i)}{d(x,m_j)}, \frac{d(x,m_j)}{d(x,m_i)}\} > \frac{1-w}{1+w} \tag{3.12}$$

Good values for w are in $[0.2, 0.3]$.

Assume that m_j belongs to the same class as x ($\Rightarrow x$ and m_i belong to different class), then m_i and m_j are updated simultaneously and similar to LVQ1.

$$\begin{aligned} m_i(t+1) &= m_i(t) - \eta(t)[x(t) - m_i(t)] \\ m_j(t+1) &= m_j(t) + \eta(t)[x(t) - m_j(t)] \end{aligned} \tag{3.13}$$

In the original LVQ2 m_i had to be the closest codebook vector and belong to a different class.

3.2.4 LVQ3

LVQ3 has the same update rule as LVQ2.1, but in case that x, m_i and m_j all belong to the same class we will perform the following update:

$$m_k(t+1) = m_k(t) + \epsilon\eta(t)[x(t) - m_k(t)] \quad fork \in i, j \tag{3.14}$$

Applicable values for ϵ are between 0.1 and 0.5 and seem to depend on the window size.

Chapter 4

Multi-Layer Perceptrons

4.1 Backpropagation

Literature: [Mitchell, 1997, Chapter 4.5], [Duda et al., 2000, Chapter 6.3], [Patterson, 1997] and the original paper [Rumelhart et al., 1986]

4.1.1 Notation

- m inputs/features, $x \in \mathbb{R}^m$
- k target outputs, $t \in \mathbb{R}^k$
- n training examples of form $(x, t) \in \mathbb{R}^m \times \mathbb{R}^k$
- L layers $(1, \ldots, L)$
- E: error function (e. g. $E_{\text{MSE}} = \frac{1}{2} \sum_{i=1}^{k} (t_i - o_i^{(L)})^2$)
- $\phi(y)$: activation function (e. g. $\phi(y) = \sigma(y) = \frac{1}{1+e^{-y}}$)
- $\delta_j^{(l)}$: error of neuron j in layer l

For convenience we define $a_j^{(0)} \coloneqq x_j$ and $o_j \coloneqq a_j^{(L)}$.

4.1.2 Weight Updates in Output Layer

Backpropagation is generalization of the delta-rule. We minimize our error function E by 'going down' along the gradient. For a single training example (x, t), with x as input and t and desired output, E is calculated from o and t, where o is the result of a forward propagation using x and our weights $w_{ij}^{(l)}$. As the training example is fixed we can only adjust $w_{ij}^{(l)}$ to minimize E.

We start with the gradient in our output layer:

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = \frac{\partial E}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial E}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} \quad \text{(chain rule)}$$

$$\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial \sum_i w_{ij}^{(L)} * x_i^{(L-1)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} \tag{4.1}$$

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial \phi(z_j^{(L)})}{\partial z_j^{(L)}} = (\phi(z_j^{(L)})(1 - \phi(z_j^{(L)})) = o_j(1 - o_j) \quad \text{(sigmoid derivative, see 4.3.2)}$$

$$\frac{\partial E}{\partial a_j^{(L)}} = \frac{\partial E}{\partial o_j} = \frac{\partial \frac{1}{2} \sum_{i=1}^{k} (t_i - o_i)^2}{\partial o_j} = (o_j - t_j) \quad \text{(MSE derivative, see 4.2.1)}$$

putting everything together

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = (o_j - t_j) o_j (1 - o_j) a_i^{(L-1)} \tag{4.2}$$

We can now define the error for neuron j as

$$\begin{aligned} \delta_j^{(L)} &:= \frac{\partial E}{z_j^{(L)}} \\ &= \frac{\partial E}{a_j^{(L)}} \frac{\partial a_j^{(L)}}{z_j^{(L)}} \\ &= (o_j - t_j) o_j (1 - o_j)) \end{aligned} \tag{4.3}$$

, and our weight update

$$\Delta w_{ij}^{(L)} := -\eta \frac{\partial E}{\partial w_{ij}^{(L)}} = -\eta \, \delta_j^{(L)} a_i^{(L-1)} \tag{4.4}$$

$$w_{ij}^{(L)} \leftarrow w_{ij}^{(L)} + \Delta w_{ij}^{(L)} \tag{4.5}$$

.

4.1.3 Weight Outputs for Hidden Layers

For hidden layers we need to propagate the error back to the neuron j in layer l. Intuitively what are doing, is just inserting the error into the output layer and propagate backwards to the

input layer using.

$$\delta_j^{(l)} := \frac{\partial E}{\partial z_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \tag{4.6}$$

$$\frac{\partial E}{\partial a_j^{(l)}} = \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \tag{4.7}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial \sum_i w_{ik}^{(l+1)} * a_i^{(l)}}{\partial a_j^{(l)}} \tag{4.8}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} w_{jk}^{(l+1)} \tag{4.9}$$

$$= \sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)} \tag{4.10}$$

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = a_j^{(l)}(1 - a_j^{(l)}) \qquad \text{(sigmoid derivative)}$$

$$\delta_j^{(l)} = a_j^{(l)}(1 - a_j^{(l)}) \sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)} \tag{4.11}$$

This leads to our weight updates for hidden neurons.

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E}{\partial w_{ij}^{(l)}} \tag{4.12}$$

$$= -\eta \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{4.13}$$

$$= -\eta\, \delta_j^{(l)} a_i^{(l-1)} \tag{4.14}$$

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \Delta w_{ij}^{(l)} \tag{4.15}$$

4.1.4 Backpropagation using Matrix Notation

The algorithm can also be formulated using matrices.

- $X \in \mathbb{R}^{n \times m}$ input of training data
- $T \in \mathbb{R}^{n \times k}$ target output for training data
- $W^{(1)}, \ldots, W^{(L)}$ weight matrices
- $\delta^{(1)}, \ldots, \delta^{(L)}$ error matrices

TODO, the following might me incomplete and wrong

$$A^{(0)} = X \tag{4.16}$$

$$Z^{(l)} = A^{(l-1)} * W^{(l)} \tag{4.17}$$

$$A^{(l)} = \phi(Z^{(l)}) \tag{4.18}$$

$$O = A^{(L)} \tag{4.19}$$

$$E = \frac{1}{2} \sum (T - O) \circ (T - O) \tag{4.20}$$

$$\delta^{(L)} = (O - T) \circ O \circ (1 - O) \tag{4.21}$$

$$\delta^{(l)} = A^{(l)}(1 - A^{(l)}) W^{(l+1)} \delta^{(l+1)} \qquad \Delta W^{(l)} = -\eta\, \delta^{(l)} A^{(l-1)} \tag{4.22}$$

$$\hat{W}^{(l)} = W^{(l)} + \Delta W^{(l)} \tag{4.23}$$

4.2 Error Functions

Sometimes also called objective functions or loss-functions.

4.2.1 Mean-Squared Error

As the name suggests the mean-square error (MSE) is defined as the mean (over all training examples) of squared difference between the correct value t_i und the correct value o_i. This big errors are punished harder than small differences.

$$E_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(t_i - o_i)^2 \tag{4.24}$$

This would derive to

$$\begin{aligned}\frac{\partial E_{\text{MSE}}}{\partial o_j} &= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial (t_i - o_i)^2}{\partial o_j} \\ &= \frac{1}{n}\frac{\partial (t_j - o_j)^2}{\partial o_j} \\ &= \frac{1}{n}2(t_j - o_j)(-1) \\ &= \frac{2}{n}(o_j - t_j)\end{aligned} \tag{4.25}$$

which is technically fine, but as the fraction is only a constant factor, we will often use a a slightly different definition:

$$E_{\text{MSE}} = \frac{1}{2}\sum_{i=1}^{k}(t_i - o_i)^2 \tag{4.26}$$

$$\frac{\partial E_{\text{MSE}}}{\partial o_j} = -(t_j - o_j) = o_j - t_j \tag{4.27}$$

4.2.2 Cross Entropy Error

Cross-entropy (CE) works great well on classifications tasks. t_i is either 0 or 1 and o_i is the class probability computed by the network $\Rightarrow o_i \in (0, 1]$ as $log(0)$ is not defined.

$$E_{\text{CE}} = -\sum_{i=1}^{k}\left(t_i \log(o_i) + (1 - t_i)\log(1 - o_i)\right) \tag{4.28}$$

This will derive to:

$$\begin{aligned}\frac{\partial E_{\text{CE}}}{\partial o_j} &= -\frac{\partial}{\partial o_j}\sum_{i=1}^{k}\left(t_i \log(o_i) + (1 - t_i)\log(1 - o_i)\right) \\ &= -\frac{\partial}{\partial o_j}t_j \log(o_j) - \frac{\partial}{\partial o_j}(1 - t_j)\log(1 - o_j) \\ &= -\frac{t_j}{o_j} + \frac{1 - t_j}{1 - o_j} \\ &= \frac{1 - t_j}{1 - o_j} - \frac{t_j}{o_j}\end{aligned} \tag{4.29}$$

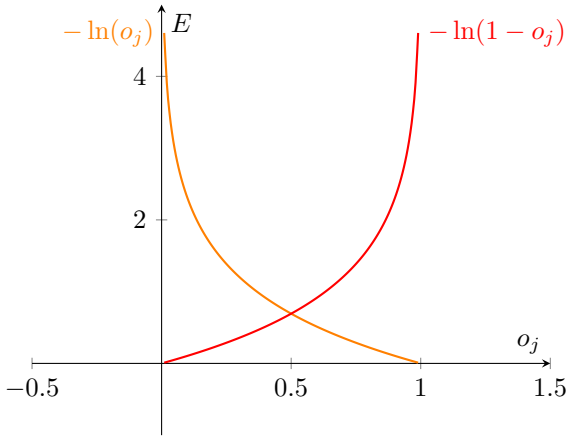

Figure 4.1: Error produced for $t_j = 1$ (orange) and for $t_j = 0$ (red). o_j must be in range $(0, 1]$, otherwise the logarithm can be complex or undefined or the error gets rediciously large.

4.2.3 L1 and L2 Regularization

Small weights tend to perform better and we can modify the error functions from above to penalize big weight by adding additional error terms.

- L1 norm: $||\mathbf{w}||_{L1} = \sum_j |\mathbf{w}_j|$
- L2 norm: $||\mathbf{w}||_{L2} = \sum_j \mathbf{w}_j^2$
- new error function $E' = E + \alpha_{L1}||\mathbf{w}||_{L1} + \alpha_{L2}||\mathbf{w}||_{L2}$
- new hyperparamaters α_{L1} and α_{L2}, optimize on development set

Todo: Explain advantages and disadvantages, see http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization

4.3 Activation Functions

4.3.1 Step Function

$$\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{4.30}$$

The derivative is always 0.

4.3.2 Sigmoid

Most common activation function, can saturate and is easy to dervice.

$$\sigma(x) = \frac{1}{1 + e^{-\beta x}} \tag{4.31}$$

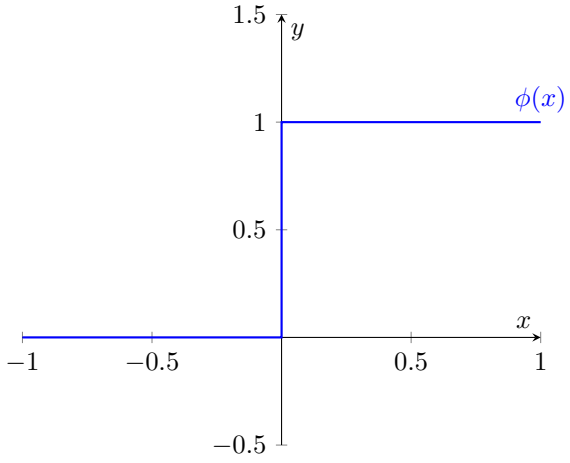

Figure 4.2: Step Function

And it's derivative:

$$\begin{aligned}
\frac{d\,\sigma(x)}{d\,x} &= \frac{d}{d\,x}(1+e^{-\beta x})^{-1} \\
&= (-1)(1+e^{-\beta x})^{-2}(-\beta e^{-\beta x}) \\
&= \beta(1+e^{-\beta x})^{-2}e^{-\beta x} \\
&= \beta\frac{1}{1+e^{-\beta x}}\frac{e^{-\beta x}}{1+e^{-\beta x}} \\
&= \beta\sigma(x)\frac{e^{-\beta x}+1-1}{1+e^{-\beta x}} \\
&= \beta\sigma(x)(1-\frac{1}{1+e^{-\beta x}}) \\
&= \beta\sigma(x)(1-\sigma(x))
\end{aligned} \tag{4.32}$$

And because there exists algorithm using this, you should have seen the second derivative as well ($\beta = 1$):

$$\begin{aligned}
\frac{d\,\sigma(x)(1-\sigma(x))}{d\,x} &= (1-\sigma(x))\frac{d\,\sigma(x)}{d\,x} + \sigma(x)\frac{d\,(1-\sigma(x))}{d\,x} \\
&= (1-\sigma(x))\sigma(x)(1-\sigma(x)) + \sigma(x)(-1)\sigma(x)(1-\sigma(x)) \\
&= \sigma(x)(1-\sigma(x))^2 - \sigma(x)^2(1-\sigma(x))
\end{aligned} \tag{4.33}$$

4.3.3 Softmax Function

In a classification problem we would like a_j to be a probability ($\Rightarrow \sum_i a_i = 1$). The softmax function will output a posteriori probability and the biggest value from the layer below is very likely to be close to 1.

$$a_i = \phi(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4.34}$$

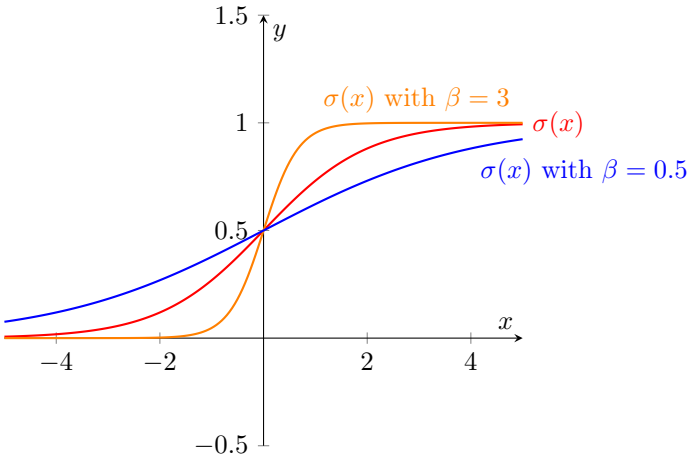

Figure 4.3: Sigmoid function with $\beta = 1$ (red), $\beta = 3$ (orange) and $\beta = 0.5$ (blue).

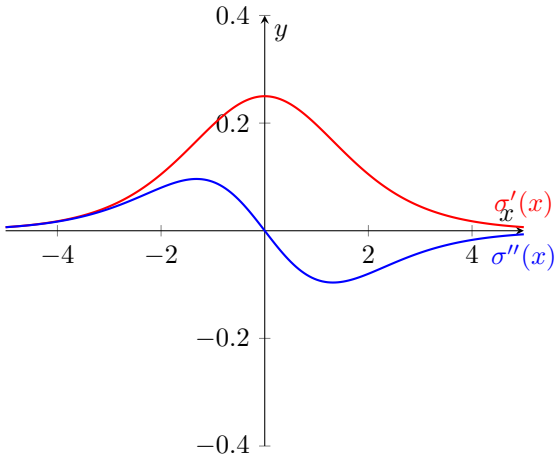

Figure 4.4: First and second derivative of the sigmoid function.

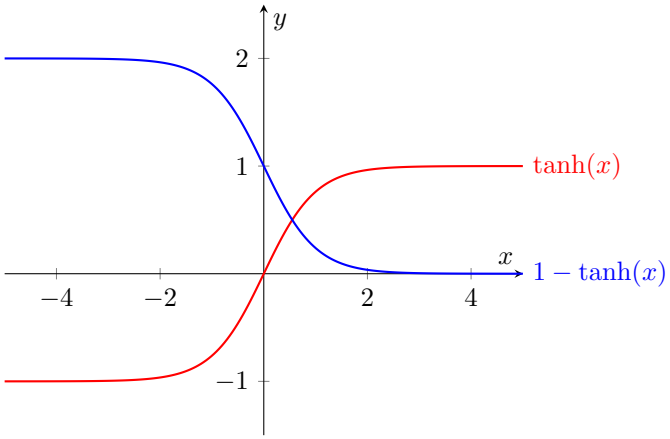

Figure 4.5: Hyperbolic Tangent Function (red) and its derivative (blue).

And it's derivative:

$$\begin{aligned}
\frac{\partial \phi(z_i)}{\partial z_i} &= \frac{\left(\frac{\partial}{\partial z_i} e^{z_i}\right)\left(\sum_j e^{z_j}\right) - e^{z_i}\left(\frac{\partial}{\partial z_i}\left(\sum_j e^{z_j}\right)\right)}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}\sum_j e^{y_j} - e^{y_i} e^{y_i}}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}\sum_j e^{y_j}}{\left(\sum_j e^{y_j}\right)^2} - \frac{e^{y_i} e^{y_i}}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}}{\sum_j e^{y_j}} - \left(\frac{e^{y_i}}{\sum_j e^{y_j}}\right)^2 \\
&= \phi(y_i) - \phi(y_i)^2 \\
&= \phi(y_i)(1 - \phi(y_i))
\end{aligned} \tag{4.35}$$

4.3.4 Hyperbolic Tangent Function

Similar to Sigmoid function, but with outputs in $[-1, 1]$. Also not that if the input has a mean of 0 then so will the output.

Todo: Why is tanh worth the computational overhead over sigmoid?

$$\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.36}$$

And it's derivative:

$$\frac{d\phi(x)}{dx} = 1 - \tanh(x) = \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \tag{4.37}$$

4.3.5 Linear Function

Very simple, but with two big downsides. The gradient is always 1 and multi-layers using it effectively only do a linear combination and therefore can be left out.

$$\phi(x) = x$$

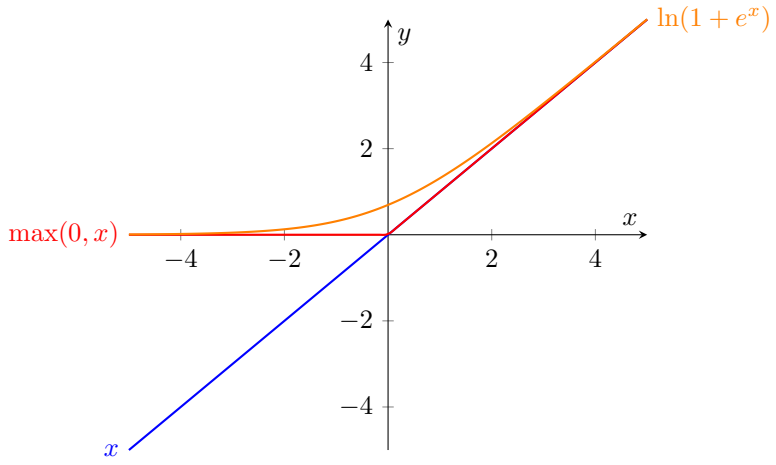

Figure 4.6: Linear Function (blue), Rectified Linear Unit (red), Softplus (orange)

4.3.6 Rectified Linear Unit

The Rectified Linear Unit (ReLU) is more biologically plausible.

$$\phi(x) = \max(0, x) \tag{4.38}$$

And it's derivative

$$\frac{d\phi(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{4.39}$$

4.3.7 Softplus

Smoothed version of ReLU.

$$\phi(x) = \ln(1 + e^x) \tag{4.40}$$

And it's derivative

$$\frac{d\phi(x)}{dx} = \frac{e^x}{1 + e^x} = \frac{1}{e^{-x} + 1} \tag{4.41}$$

4.3.8 Maxout

Literature: [Goodfellow et al., 2013]

Outputs the maximum of its inputs

$$a_j^{(l)} = \max_i z_i^{(l)}, \quad (j-1)g + 1 \leq i \leq jg \tag{4.42}$$

where $z_i^{(l)} = b^{(l)} + \sum_k w_{ki}^{(l)} x_k^{(l-1)}$ is the summed input and g the size of the group on which we take the maximum.

4.4 Learning Rate

Literature: [Duda et al., 2000, Chapter 6.8]

We use the learning rate η in our weight updates:

$$w_{ij} \leftarrow w_{ij} + \eta\, \delta_j a_i$$

There are several problems with a constant learning rate:

- large learning rate
  - constantly jump past the minimum ($\Rightarrow$ no convergence)
  - neurons may saturate
- small learning rate
  - slow training
  - may get stuck in local minimum

And of course several ways to deat with it them:

- Predetermined piecewise constant learning rate
  Use a predetermined sequence of η_i, increment i after every iteration (epoch/batch)
- Exponentially decaying learning rate
- Performance Scheduling
- Weight Dependent Learning Rate Methods (e. g. Momentum, AdaGrad)

4.4.1 Exponential Decaying Learning Rate

We calculate a new learning rate $\eta(t)$ after each iteration (epoch/batch).

$$\eta(t) = \alpha\eta(t-1) = \alpha^t\eta(0) \tag{4.43}$$

With the parameter $\alpha \in (0, 1)$ and we still have to define an initial learning rate $\eta(0)$.

4.4.2 Performance Scheduling

Measure the cross validation error and decrease the learning rate when the error stops improving.

4.4.3 Momentum or Rprop

Literature: [Riedmiller and Braun, 1994]

Similar to a rock rolling down a hill we let our gradient get some momentum to speed up training while the direction (sign) of the gradient stays the same. We do so by modifying our weight update rule to,

$$w_{ij} \leftarrow w_{ij} + v_{ij}(t) \tag{4.44}$$

$$v_{ij}(t) \coloneqq \alpha\, v_{ij}(t-1) - \eta\, \delta_j a_i^{(l-1)} \tag{4.45}$$

, with α as momentum factor.

Resilient Propagation (Rprop)

Todo: Verify that *Rprop* and *Momentum* are really the same or add differences

4.4.4 AdaGrad

Literature: [Zeiler, 2012, Duchi et al., 2011, Dyer,]

AdaGrad alters the weight update to adapt based on historical information, so that frequently occurring features in the gradients get small learning rates and infrequent features get higher ones.

$$g_{ij} = \frac{\partial E}{\partial w_{ij}} \tag{4.46}$$

$$\eta_{ij}(t) = \frac{\eta_0}{\sqrt{\sum_{\tau=1}^{t} g_{ij}(\tau)^2}}\, g_{ij}(t) \tag{4.47}$$

$$w_{ij}(t) \leftarrow w_{ij}(t-1) - \eta_{ij}(t)\, g_{ij}(t) \tag{4.48}$$

This eliminates the hyperparameter η, but introduces an initial learning rate η_0 and has some more downsides [Zeiler, 2012]. One way to improve it is the limit the window for the sum to size w.

$$\eta_{ij}(t) = \frac{\eta_0}{\sqrt{\sum_{\tau=1}^{w} g_{ij}(t-\tau)^2}}\, g_{ij}(t) \tag{4.49}$$

AdaDelta contains further improvements.

4.4.5 Newbob Scheduling

Combination of exponential decaying learning rate and performance scheduling. Easy to implement and solid results.

Phase 1 Start with constant learning rate $\eta(0)$

Phase 2 Once the cross-validation error stops decreasing switch to exponential decaying learning rate

Terminate End training after cross-validation error stops decreasing again

Chapter 5

Boltzmann Machines

5.1 Hopfield Nets

Literature: [Patterson, 1997, Chapter 5.5], [Storkey, 1997]

In the following we will only consider binary Hopfield nets, in which the neurons are limited to states 0 (or $-/-1$) and 1 (or $+/+1$) (sometimes also denoted as $+$ and $-$ or -1 and $+1$). Hopfield nets are not very efficient, but good to show the principle of neural nets and a good introduction to Boltzmann machines.

Hopfield nets consist of a single layer of neurons, that is fully connected and acts both as input and output layer. Weights are stored an a weight matrix W, where the entry w_{ij} correspondents to the connection weight form neuron i to neuron j. We will not allow self loops ($w_{ii} = 0$) and the connections will be symmetric ($w_{ij} = w_{ji}$). There exists variations without those limitations.

5.1.1 Update Procedure

1. Set state x to our input
2. Update neuron state using

$$x_j = \phi\left(\sum_i w_{ij} x_i\right)$$

   where $\phi(z)$ is the step function

$$\phi(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{5.1}$$

   The updates can be performed in different ways:

   - synchronous: all neurons get the same input (parallel, we cannot guarantee that the energy function E, see next section, decreases)
   - asynchronous (sequential): one neuron at the time (next neuron gets updated state), order can be fixed or random (random order is closest to biological neural nets)
   - combination are possible (group of neurons in parallel, similar to mini-batch training)

3. Repeat step 2 until x stabilizes.
4. State x is our output

x is now a state that minimizes the energy function E (see next section). If x is a *retrieval state* is matches exactly one of our training patterns, otherwise it is a *spurious state* and can not be recognized.
If we want to remember more that the state vector (e. g. class of the pattern) we have to store these ourself during training and use x to look them up.

5.1.2 Energy function

We define the energy in our Hopfield net as:

$$E = -\frac{1}{2}\sum_i \sum_{j \neq i} x_i x_j w_{ij} \tag{5.2}$$

E is also the objective function that we try to minimize during the update procedure.
A state x that is a local minimum of E is also called attractor.

5.1.3 Convergence

If only one neuron j is updated at the time, the update will always lead to the same or lower energy.

- $x_j(t+1) = x_j(t) \Rightarrow$ state did not change, energy stays the same
- $x_j(t+1) = 1 - x_j(t) \Rightarrow$ state did changed, compute the energy change

$$E_j = -\frac{1}{2}\sum_i x_i x_j w_{ij} \tag{5.3}$$

$$\Delta E = E_j(t+1) - E_j(t) \tag{5.4}$$

$$= -\frac{1}{2}\left[x_j(t+1)\sum_i x_i w_{ij} - x_j(t)\sum_i x_i w_{ij}\right] \tag{5.5}$$

$$= -\frac{1}{2}\Delta x_j \sum_i x_i w_{ij} \quad \text{with} \quad \Delta x_j = x_j(t+1) - x_j(t) \tag{5.6}$$

Change from 0 to 1

$$\Delta x_j = 1, \sum_i x_i w_{ij} > 0 \Rightarrow \Delta E_j \leq 0 \tag{5.7}$$

Change from 1 to 0

$$\Delta x_j = -1, \sum_i x_i w_{ij} \leq 0 \Rightarrow \Delta E_j \leq 0 \tag{5.8}$$

5.1.4 Associative Memory

An associative memory can store information, e. g. a binary vector representing the state of a Hopfield network, and retrieve it from only partial information. The number of patterns that can be learned by a Hopfield net is called the capacity C and depends on the number of neurons m and the used training algorithm.

If we want the link additional information to a state vector (e. g. the class of it) we have to store this ourself.

5.1.5 Training

A simple non-incremental learning rule is:

$$w_{ij} = \sum_{x \in X} (2x_i - 1)(2x_j - 1) \tag{5.9}$$

Assuming uncorrelated patterns, a net trained with this method has can store up to C patterns (capacity). C is bound by

$$\frac{m}{4\ln(m)} < C < \frac{m}{2\ln m} \tag{5.10}$$

, where m is the number of neurons.

See [Storkey, 1997] for more on different learning rules and their effect on the capacity.

5.1.6 Limitations

- Found stable state is not guaranteed the most similar pattern to the input pattern
- Not all memories are remembered with same emphasis (attractors regions have different sizes)
- Spurious states can occur
- Efficiency is not good

5.2 Boltzmann Machines

Literature: [Bengio, 2009]

Boltzmann machines are stochastic recurrent neural networks. They work similar *Hopfield nets* in that they have a binary state vector ($s_j \in 0, 1$), which is sometimes divided in visible states (neurons) v_j and hidden neurons h_j. But while the activation calculation in *Hopfield nets* is deterministically, it is stochastic for Boltzmann machine (BM).

$$z_j = b_j + \sum_i v_i w_{ij} \tag{5.11}$$

The probability that the activation of neuron s_j is set to 1 is calculated using the sigmoid function,

$$p(s_j = 1) = \sigma(v_j) = \frac{1}{1 + e^{-z_j}} \tag{5.12}$$

, otherwise the activation of s_j becomes 0.

Further we define the network architecture as follows

- Network is fully connected
- No self connections, $w_{ii} = 0$
- Undirected/symmetric, $w_{ij} = w_{ji}$

5.2.1 Energy-based models

Todo: Derivations of energy-based models, see [Bengio, 2009]

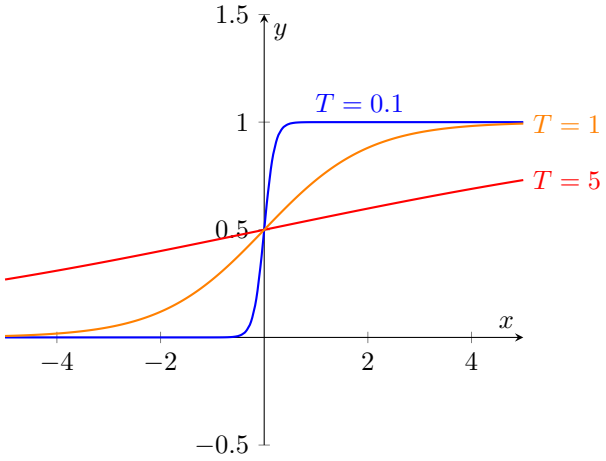

Figure 5.1: Sigmoid function (orange), with high temperature (red) and low temperature (blue).

5.2.2 Energy of Boltzmann Machines

$$E(s) = -\sum_i b_i s_i - \sum_{i,j} s_i w_{ij} s_j \tag{5.13}$$

$$E(v,h) = -\sum_i b_i v_i - \sum_i c_i h_i - \sum_{i,j} v_i w_{ij}^{vh} h_j - \sum_{i,j} v_i w_{ij}^{vv} v_j - \sum_{i,j} h_i w_{ij}^{hh} h_j \tag{5.14}$$

5.2.3 Training

Training is done by calculating the gradient of the log-likelihood derived by the the parameter set θ.

$$\frac{\partial \log p(v)}{\partial \theta} \tag{5.15}$$

Todo: Training of BMs

5.2.4 Simulated Annealing

Use temperature T to allow more changes in the beginning (e. g. jumps out of bad local minima) by using the sigmoid function with $\beta = \frac{1}{T}$.

$$p(s_j = 1) = \frac{1}{1 + e^{\frac{-z_j}{T}}} \tag{5.16}$$

Figure 5.1 shows curves with different temperature.

5.2.5 Why to restrict BMs

Unrestricted Boltzmann machines are very powerful and can compute every function. But due the complex net structure the training is very slow and computational expensive. RBMs applying constraints to the network architecture, but can still compute any function.

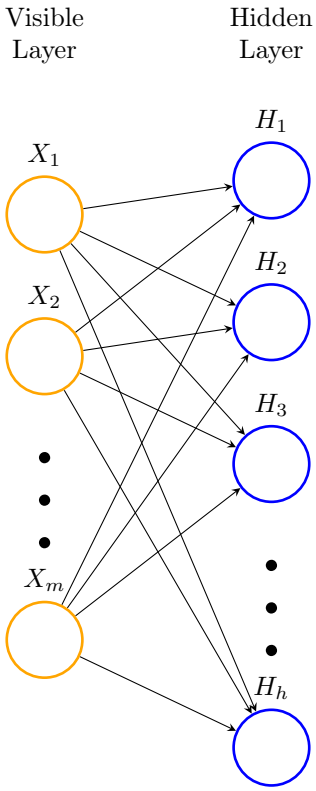

Figure 5.2: Architecture of a RBM

5.3 Restricted Boltzmann Machines

Literature: [Bengio, 2009]

For an efficient training Boltzmann machines can be restricted to have a bipartite graph with one set of visible neurons and on set of hidden neurons. As shown in figure 5.2 there are only visible-hidden and hidden-visible connection (still symmetric). Therefore hidden units h_j only depend on the visible units v_j and vice versa.

$$p(h_j = 1|v) = \sigma(c_j + \sum_i w_{ij} v_i) \tag{5.17}$$

$$p(v_j = 1|h) = \sigma(b_j + \sum_i w_{ij} h_i) \tag{5.18}$$

With b_j as the biases for the visible units and c_j for the hidden units.

5.3.1 Energy

$$E(v, h) = -\sum_{i,j} w_{ij} v_i h_j - \sum_i a_i v_i - \sum_i b_i h_i \tag{5.19}$$

5.4 Training

Literature: [Hinton et al., 2012]

The most common algorithm used is contrastive divergence (CD) used inside a gradient-descent and preforming Gibbs Sampling. A single-step contrastive divergence (CD-1) procedure for a single training example can be summarized as follows:

1. Sample hidden units $\mathbf{h}$ from training example $\mathbf{v}$
2. Sample reconstruction $\mathbf{v}'$ of visible units using $\mathbf{h}$ and then resample $\mathbf{h}'$ from it. (Gibbs sampling step)
3. $w_{ij} \leftarrow w_{ij} - \eta\left(\mathbf{v}\mathbf{h}^T - \mathbf{v}'\mathbf{h}'^T\right)$

5.5 Stacking RBMs

Chapter 6

Autoencoders

- Hidden layer can be any size
- Output layer must have size of input layer
- Weight matrix $\mathbf{W}$ and bias $\mathbf{b}$ can be learning using backpropagation
- $\mathbf{W}^{(2)} = (\mathbf{W}^{(1)})^T$
- Non-linear activation function is required, otherwise autoencoder might learn pass through and hidden layer could be similar to principal component analysis (PCA)

6.1 Spares Autoencoders

Literature: [Ng, 2011]

Favor sparse weight matrices by forcing the average neuron activation $\hat{\rho}_j$ to be close to ρ 0.2.

$$\hat{\rho}_j = \frac{1}{X} \sum_{x \in X} \mathbf{w}_j \mathbf{x} \tag{6.1}$$

This can be done by adding the Kullback–Leibler divergence,

$$KL(\rho || \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j} \tag{6.2}$$

, to the error function E,

$$E_{\text{sparse}} = E + \sum_{j \in \text{hidden}} KL(\rho || \hat{\rho}_j) \tag{6.3}$$

Figure 6.2 shows the error introduced by the average neuron activation of a single hidden neuron.

6.2 Denoising Autoencoders

- Create corrupted X' of training data X, e. g. using Gaussian noise or by setting values to zero
- During training use X' as input, but target X as output
- Resulting autoencoder finds good features for noisy input

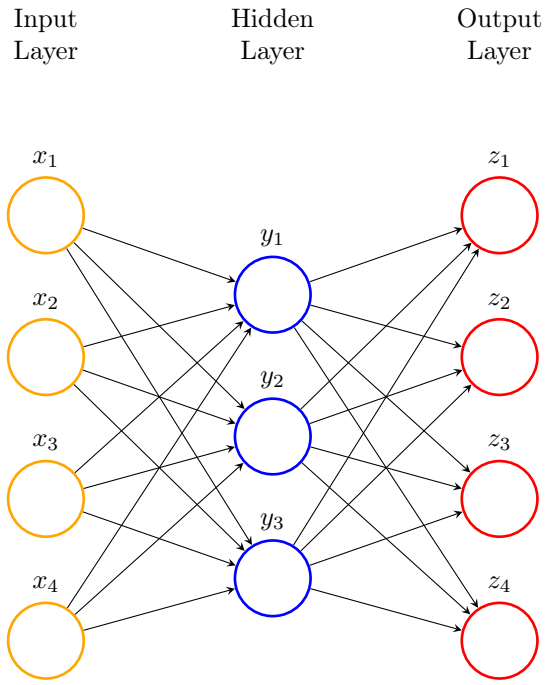

Figure 6.1: Simple Autoencoder

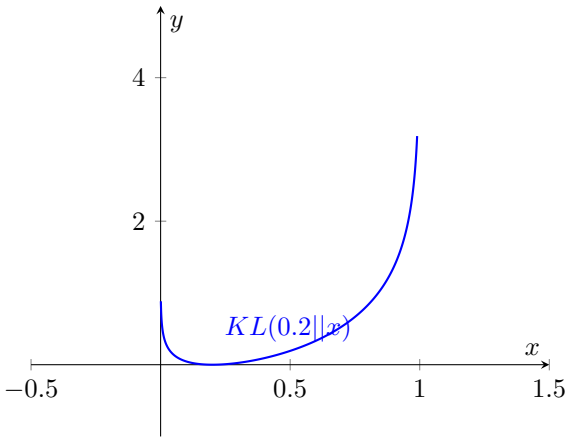

Figure 6.2: Kullback–Leibler divergence $KL(\rho||\hat{\rho}_j)$ with $\rho = 0.2$

6.3 Stacking Autoencoders

- Train first autoencoder AE_1 using the original training data X
- Input X into AE_1 and use output as X'
- Train second autoencoder AE_2 using X'
- Use AE_2 to transform X' to X''
- Continue to create $\mathrm{EA}_3, \mathrm{EA}_4, \cdots$
- Use weight matrices and biases from autoencoders to stack a deep neural network (DNN)

Chapter 7

Practical Tricks

7.1 Data Normalization

Numerical needs to be normalized because neural network (NN) function best with inputs between in range $[0, 1]$ or $[-1, +1]$. There are two common normalizations, the min-max normalization (sometimes called feature scaling),

$$\hat{x}_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{7.1}$$

, which will transform the smallest value to 0 and the biggest to 1 and everything else linearly in between, and gaussian normalization,

$$\hat{x}_i = \frac{x_i - \text{mean}(x)}{\text{std}(x)} = \frac{x_i - \mu}{\sigma} \tag{7.2}$$

, will transform x to have zero mean and a standard deviation of one. Other names for int are standard score or z-scores.

Those are the two most common methods, but depending on the input there might be more.

7.2 Bottleneck-Features

7.3 Weight Decay

Literature: [Connect et al., 1992] and [Duda et al., 2000, Chapter 6.8]

To help generalizing by simplifying a network and to avoid overfitting one can impose a heuristic that the weights stay small. This may not always lead to improved network performance, but experience showed that it helps in many cases and even can be used together with momentum (see 4.4.3).

It is also every simple, after each weight update we simply "decay" the weights with a parameter $\epsilon \in (0, 1)$.

$$w_{ij} \leftarrow w_{ij}(1 - \epsilon) \tag{7.3}$$

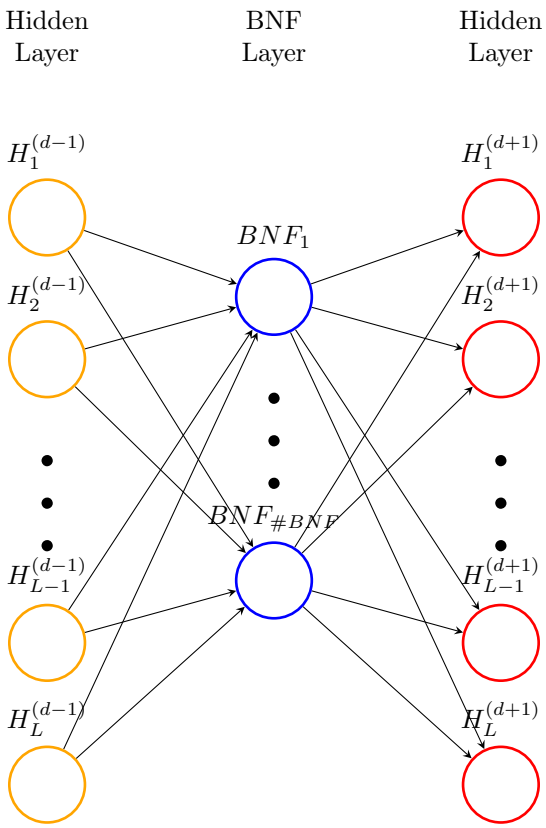

Figure 7.1: Neural network with Bottleneck features

combined with the weight update we get

$$w_{ij} \leftarrow (w_{ij} - \eta \frac{\partial E}{\partial w_{ij}})(1-\epsilon) \tag{7.4}$$

$$\Delta w_{ij} = -\epsilon\, w_{ij} - (1-\epsilon)\eta \frac{\partial E}{\partial w_{ij}} \tag{7.5}$$

$$= -\epsilon\, w_{ij} - \eta' \frac{\partial E}{\partial w_{ij}} \tag{7.6}$$

$$= -\epsilon\, w_{ij} - \eta' \frac{\partial E}{\partial w_{ij}} \tag{7.7}$$

Those weights that are need to solve the problem will increase enough with the weight update to compensate the small decrease, but others will get smaller and smaller until they can be eliminated completely. It can be shown that the above is equivalent to gradient descent we a modified error function:

$$E_{\text{weight decay}} = E + \frac{2\epsilon}{\eta}\mathbf{w}^T\mathbf{w} \tag{7.8}$$

7.4 Dropout

Literature: [Hinton et al., 2012, Hinton, 2014, Wang and Manning, 2013]

- Randomly drop out hidden units (set to zero) and input features during training.
- Each time a different model is trained and the final model is the mean of all trained models.
- Prevents feature co-adaptation detectors and overfitting.

Bibliography

[Bengio, 2009] Bengio, Y. (2009). Learning deep architectures for ai.

[Connect et al., 1992] Connect, a. K., Krogh, a., and Hertz, J. a. (1992). A simple weight decay can improve generalization. *Advances in Neural Information Processing Systems*, 4:950–957.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

[Duda et al., 2000] Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification*. JOHN WlLEY & SONS, INC.

[Dyer,] Dyer, C. Notes on adagrad. *Ark.Cs.Cmu.Edu*, pages 1–3.

[Goodfellow et al., 2013] Goodfellow, I. J., Warde-farley, D., and Courville, A. (2013). Maxout networks.

[Hinton, 2014] Hinton, G. (2014). Dropout : A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958.

[Hinton et al., 2012] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. pages 1–18.

[Introduction, 2000] Introduction, I. (2000). A fuzzy clustering model of data and fuzzy. pages 302–307.

[Kohonen, 1990] Kohonen, T. (1990). Improved versions of learning vector quantization. *International Joint Conference on Neural Networks*.

[Kohonen, 2001] Kohonen, T. (2001). *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer.

[Linde et al., 1980] Linde, Y., Buzo, A., and Gray, R. (1980). An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28(1):84–95.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*, volume 4 of *McGraw-Hill Series in Computer Science*. McGraw-Hill.

[Ng, 2011] Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes*, pages 1–19.

[Patterson, 1997] Patterson, D. W. (1997). *Kuenstliche neuronale Netze - Das Lehrbuch*. Prentice Hall, Muenchen [u.a.].

[Riedmiller and Braun, 1994] Riedmiller, M. and Braun, H. (1994). Rprop - description and implementation details. *Neural Networks*, (January):1–2.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362.

[Storkey, 1997] Storkey, A. (1997). Increasing the capacity of a hopfield network without sacrificing functionality. In *Artificial Neural Networks-ICANN'97*, pages 451–456. Springer.

[Wang and Manning, 2013] Wang, S. I. and Manning, C. D. (2013). Fast dropout training. *Proceedings of the 30th International Conference on Machine Learning*, 28:118–126.

[Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method. page 6.

Acronyms

CE cross-entropy. 13

DNN deep neural network. 9, 19

KIT Karlsruhe Institute of Technology. 3

LVQ learning vector quantization. 8

MSE mean-square error. 13

NN neural network. 19

Todo list