Neuronal Networks

Formulae Collection

Marvin Ritter
marvin.ritter@gmail.com

18th September 2015

Introduction

This is my personal collection of formulae in the field of neural networks. I started it in preparation to my exam on the neural nets at Karlsruhe Institute of Technology (KIT). Even so the content will be similar to the course it is neither limited to it nor linked with the course in any means. *This is not official script and may contain errors and lag completeness.*

Corrections, supplements (or wishes) and links to good sources/ papers are very welcome. Just mail me to marvin.ritter@gmail.com.

Contents

Chapter 1

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 ($N_k = \sum\limits_{C(i)=k} 1$)
- $d(x, y)$ 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 (1.3). 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.

1.1 Loss Functions for Clustering

- Intra-class scatter: $W(C) = \frac{1}{2} \sum\limits_{k=1}^{K} \sum\limits_{C(i)=k} \sum\limits_{C(j)=k} d(x_i, x_j)$

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

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

1.2 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

$$J = \sum_{k=1}^{K}\sum_{C(i)=k} d(x_i, \mu_k) \tag{1.1}$$

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 $W(C)$

$$W(C) = \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(j)=k} ||x_i - x_j||^2 = \sum_{k=1}^{K} N_k \sum_{C(i)=k} ||x_i - \mu_k||^2 \tag{1.2}$$

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:

$$\mu_k = \frac{\sum\limits_{C(i)=k} x_i}{N_k}$$

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.

1.3 Fuzzy k-Means

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:

$$J_m = \sum_{i=1}^{N} \sum_{k=1}^{K} u_{ik} d(x_i, \mu_k) \tag{1.3}$$

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.**

1.4 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:
$$\mu_1(0) = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{1.4}$$
$$t = 0 \tag{1.5}$$
2. **Double codebook**:
$$\mu_j(t+1) = (1+\epsilon)\, \mu_j(t) \quad \text{for } j = 1, \cdots, 2^t \tag{1.6}$$
$$\mu_{2*j+1}(t+1) = (1-\epsilon)\, \mu_j(t) \quad \text{for } j = 1, \cdots, 2^t \tag{1.7}$$
$$t = t + 1 \tag{1.8}$$
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 $\mu_j(\log_2(K))$.

1.5 Self-Organizing Maps

Sometimes also called self-organizing feature maps.

Chapter 2

Supervised Vector Quantization

2.1 Distortion Measures

TODO

Most common is the squared-error distortion:

$$d(x, \hat{x}) = \sum_{i=1}^{N} |x_i - \hat{x}_i|^2 \tag{2.1}$$

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

$$d(x, \hat{x}) = \left(\sum_{i=1}^{N} |x_i - \hat{x}_i|^\nu \right)^{\frac{1}{\nu}} = ||x - \hat{x}||_\nu \tag{2.2}$$

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

$$d(x, \hat{x}) = \sum_{i=1}^{N} |x_i - \hat{x}_i|^\nu \tag{2.3}$$

The holder Norm (2.2) is a distance and fulfills the triangle inequality[1], the ν^{th}-law distortion not.

All three and many others, as the weighted-squares distortion and the quadratic distortion, depend on the difference $x - \hat{x}$. We call them can be described as $d(x, \hat{x}) = L(x - \hat{x})$. A distortion not having this form is the one by Itakura, Saito and Chaffee,

$$d(x, \hat{x}) = (x - \hat{x})\, R(x)\, (x - \hat{x})^T \tag{2.4}$$

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

, where $R(x)$ is the autocorrelation matrix, see [Linde et al., 1980] for details.

2.2 Learning Vector Quantization

Literature: [Kohonen, 2001, 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

$$s(t) = \begin{cases} +1 & \text{if } x \text{ and } m_c \text{ belong to the same class} \\ -1 & \text{if } x \text{ and } m_c \text{ belong to different classes} \end{cases} \tag{2.5}$$

2.2.1 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

$$m_c(t+1) = m_c(t) + \eta(t)(x - m_c(t)) \tag{2.6}$$

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

$$m_c(t+1) = m_c(t) - \eta(t)(x - m_c(t)) \tag{2.7}$$

Leave the other codebook vectors unchanged

$$m_i(t+1) = m_i(t) \text{ for } i \neq c \tag{2.8}$$

Or in a compressed form:

$$m_i(t+1) = m_i(t) + \eta(t)s(t)\delta_{ci}(x(t) - m_i(t)) \tag{2.9}$$

2.2.2 OLVQ

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

$$\eta_c(t) = \frac{\eta_c(t-1)}{1 + s(t)\eta_c(t-1)} \tag{2.10}$$

Modify update rule from LVQ1.

$$m_i(t+1) = [1 - \eta_i(t)s(t)\delta_{ci}]m_i(t) + \eta_i(t)s(t)\delta_{ci}x(t) \tag{2.11}$$

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

2.2.3 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.

$$\min\{\frac{d(x, m_i)}{d(x, m_j)}, \frac{d(x, m_j)}{d(x, m_i)}\} > \frac{1-w}{1+w} \tag{2.12}$$

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.

$$\begin{aligned} m_i(t+1) &= m_i(t) - \eta(t)[x(t) - m_i(t)] \\ m_j(t+1) &= m_j(t) + \eta(t)[x(t) - m_j(t)] \end{aligned} \tag{2.13}$$

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

2.2.4 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:

$$m_k(t+1) = m_k(t) + \epsilon\eta(t)[x(t) - m_k(t)] \quad fork \in i, j \tag{2.14}$$

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

Chapter 3

Multi-Layer Perceptrons

3.1 Backpropagation

3.1.1 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 $(1, \dots, L)$
- E: error function (e. g. $E_{\text{MSE}} = \frac{1}{2} \sum\limits_{i=1}^{k} (t_i - o_i^{(L)})^2$)
- $\phi(y)$: activation function (e. g. $\phi(y) = \sigma(y) = \frac{1}{1+e^{-y}}$)
- $x_{ij}^{(l)}$: input i of neuron j in layer l
- $w_{ij}^{(l)}$: weight of connection from neuron i in layer $l-1$ to neuron j in layer l
- $z_j^{(l)} = \sum\limits_i w_{ij}^{(l)} * a_i^{(l-1)}$ with $a_i^{(0)} = x$
- $a_j^{(l)} = \phi(z_j^{(l)})$
- $o = a^{(L)}$ is the output of the neural network
- η: learning rate (usually $\eta < 1$)
- $b^{(l)}$: bias unit layer l, either 1 if there is a weight for it, or without weight if $b^{(l)}$ is adjusted directly during training
- $\delta_j^{(l)}$: error of neuron j in layer l

3.1.2 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:

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

$$\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial \sum_i w_{ij}^{(L)} * x_i^{(L-1)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} \tag{3.1}$$

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial \phi(z_j^{(L)})}{\partial z_j^{(L)}} = (\phi(z_j^{(L)})(1 - \phi(z_j^{(L)})) = o_j(1 - o_j)$$

(sigmoid derivative, see 3.3.2)

$$\frac{\partial E}{\partial a_j^{(L)}} = \frac{\partial E}{\partial o_j} = \frac{\partial \frac{1}{2} \sum_{i=1}^{k} (t_i - o_i)^2}{\partial o_j} = (o_j - t_j) \qquad \text{(MSE derivative, see 3.2.1)}$$

putting everything together

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = (o_j - t_j) o_j (1 - o_j) a_i^{(L-1)} \tag{3.2}$$

We can now define the error for neuron j as

$$\begin{aligned} \delta_j^{(L)} &= \frac{\partial E}{z_j^{(L)}} \\ &= \frac{\partial E}{a_j^{(L)}} \frac{\partial a_j^{(L)}}{z_j^{(L)}} \\ &= (o_j - t_j) o_j (1 - o_j)) \end{aligned} \tag{3.3}$$

And our weight update $\Delta w_{ij}^{(L)}$ and new weight $\hat{w}_{ij}^{(L)}$

$$\Delta w_{ij}^{(L)} = -\eta \frac{\partial E}{\partial w_{ij}^{(L)}} = -\eta \delta_j^{(L)} a_i^{(L-1)} \tag{3.4}$$

$$\hat{w}_{ij}^{(L)} = w_{ij}^{(L)} + \Delta w_{ij}^{(L)} \tag{3.5}$$

3.1.3 Weight Outputs for Hidden Layers

For hidden layers we need to propagate the error back to the neuron j in layer l. Intuitively we put in the error in the output layer and use the weights to propagate it backwards.

$$\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \tag{3.6}$$

$$\frac{\partial E}{\partial a_j^{(l)}} = \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \tag{3.7}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial \sum_i w_{ik}^{(l+1)} * a_i^{(l)}}{\partial a_j^{(l)}} \tag{3.8}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} w_{jk}^{(l+1)} \tag{3.9}$$

$$= \sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)} \tag{3.10}$$

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

$$\delta_j^{(l)} = a_j^{(l)}(1 - a_j^{(l)}) \sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)} \tag{3.11}$$

And finally our weight updates for hidden neurons

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E}{\partial w_{ij}^{(l)}} \tag{3.12}$$

$$= -\eta \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{3.13}$$

$$= -\eta\, \delta_j^{(l)} a_i^{(l-1)} \tag{3.14}$$

$$\hat{w}_{ij}^{(l)} = w_{ij}^{(l)} + \Delta w_{ij}^{(l)} \tag{3.15}$$

3.1.4 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

$$\begin{aligned}
A^{(0)} &= X && (3.16)\\
Z^{(l)} &= A^{(l-1)} * W^{(l)} && (3.17)\\
A^{(l)} &= \phi(Z^{(l)}) && (3.18)\\
O &= A^{(L)} && (3.19)\\
E &= \frac{1}{2}\sum(T-O)\circ(T-O) && (3.20)\\
\delta^{(L)} &= (O-T)\circ O\circ(1-O) && (3.21)\\
\delta^{(l)} &= A^{(l)}(1-A^{(l)})W^{(l+1)}\delta^{(l+1)}\Delta W^{(l)} \qquad = -\eta\,\delta^{(l)}A^{(l-1)} && (3.22)\\
\hat{W}^{(l)} &= W^{(l)} + \Delta W^{(l)} && (3.23)
\end{aligned}$$

3.2 Error Functions

Sometimes also called objective functions or loss-functions.

3.2.1 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.

$$E_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(t_i - o_i)^2 \tag{3.24}$$

This would derive to

$$\begin{aligned}
\frac{\partial E_{\text{MSE}}}{\partial o_j} &= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial (t_i-o_i)^2}{\partial o_j}\\
&= \frac{1}{n}\frac{\partial (t_j-o_j)^2}{\partial o_j}\\
&= \frac{1}{n}2(t_j-o_j)(-1)\\
&= \frac{2}{n}(o_j-t_j)
\end{aligned} \tag{3.25}$$

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

$$E_{\text{MSE}} = \frac{1}{2}\sum_{i=1}^{k}(t_i - o_i)^2 \tag{3.26}$$

$$\frac{\partial E_{\text{MSE}}}{\partial o_j} = -(t_j - o_j) = o_j - t_j \tag{3.27}$$

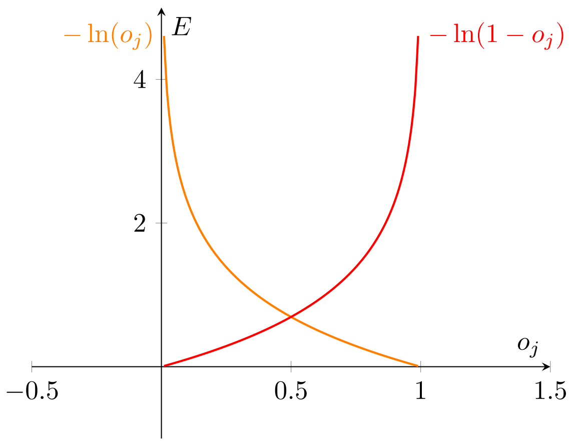

Figure 3.1: Error produced for $t_j = 1$ (orange) and for $t_j = 0$ (red). o_j must be in range $(0, 1]$, otherwise the logarithm can be complex or undefined or the error gets rediciously large.

3.2.2 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.

$$E_{\text{CE}} = -\sum_{i=1}^{k} (t_i \log(o_i) + (1 - t_i) \log(1 - o_i)) \tag{3.28}$$

This will derive to:

$$\begin{aligned}
\frac{\partial E_{\text{CE}}}{\partial o_j} &= -\frac{\partial}{\partial o_j} \sum_{i=1}^{k} (t_i \log(o_i) + (1 - t_i) \log(1 - o_i)) \\
&= -\frac{\partial}{\partial o_j} t_j \log(o_j) - \frac{\partial}{\partial o_j} (1 - t_j) \log(1 - o_j) \\
&= -\frac{t_j}{o_j} + \frac{1 - t_j}{1 - o_j} \\
&= \frac{1 - t_j}{1 - o_j} - \frac{t_j}{o_j}
\end{aligned} \tag{3.29}$$

3.3 Activation Functions

3.3.1 Step Function

$$\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{3.30}$$

The derivative is always 0.

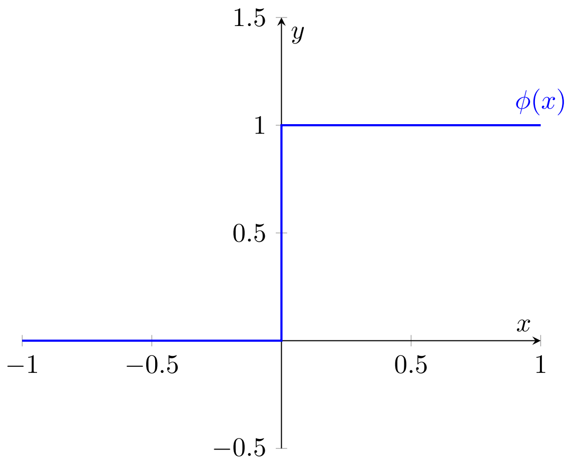

Figure 3.2: Step Function

3.3.2 Sigmoid

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

$$\sigma(x) = \frac{1}{1+e^{-\beta x}} \tag{3.31}$$

And it's derivative:

$$\begin{aligned}
\frac{d\,\sigma(x)}{d\,x} &= \frac{d}{d\,x}(1+e^{-\beta x})^{-1} \\
&= (-1)(1+e^{-\beta x})^{-2}(-\beta e^{-\beta x}) \\
&= \beta(1+e^{-\beta x})^{-2}e^{-\beta x} \\
&= \beta\frac{1}{1+e^{-\beta x}}\frac{e^{-\beta x}}{1+e^{-\beta x}} \\
&= \beta\sigma(x)\frac{e^{-\beta x}+1-1}{1+e^{-\beta x}} \\
&= \beta\sigma(x)(1-\frac{1}{1+e^{-\beta x}}) \\
&= \beta\sigma(x)(1-\sigma(x))
\end{aligned} \tag{3.32}$$

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

$$\begin{aligned}
\frac{d\,\sigma(x)(1-\sigma(x))}{d\,x} &= (1-\sigma(x))\frac{d\,\sigma(x)}{d\,x} + \sigma(x)\frac{d\,(1-\sigma(x))}{d\,x} \\
&= (1-\sigma(x))\sigma(x)(1-\sigma(x)) + \sigma(x)(-1)\sigma(x)(1-\sigma(x)) \\
&= \sigma(x)(1-\sigma(x))^2 - \sigma(x)^2(1-\sigma(x))
\end{aligned} \tag{3.33}$$

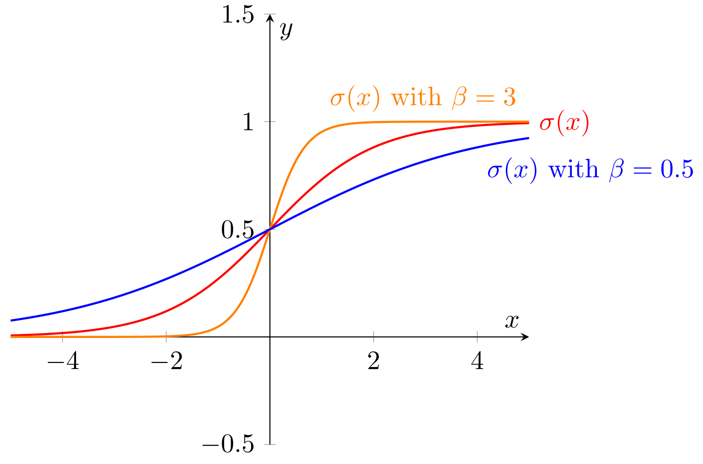

Figure 3.3: Sigmoid function with $\beta = 1$ (red), $\beta = 3$ (orange) and $\beta = 0.5$ (blue).

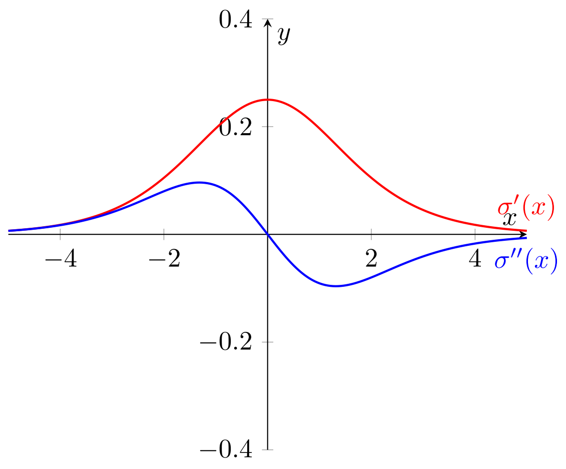

Figure 3.4: First and second derivative of the sigmoid function.

3.3.3 Softmax Function

In a classification problem we would like a_j to be a probability ($\Rightarrow \sum_i a_i = 1$).

$$a_i = \phi(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{3.34}$$

And it's derivative:

$$\begin{aligned}
\frac{\partial \phi(z_i)}{\partial z_i} &= \frac{\left(\frac{\partial}{\partial z_i} e^{z_i}\right)\left(\sum_j e^{z_j}\right) - e^{z_i}\left(\frac{\partial}{\partial z_i}\left(\sum_j e^{z_j}\right)\right)}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}\sum_j e^{y_j} - e^{y_i}e^{y_i}}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}\sum_j e^{y_j}}{\left(\sum_j e^{y_j}\right)^2} - \frac{e^{y_i}e^{y_i}}{\left(\sum_j e^{y_j}\right)^2} \\
&= \frac{e^{y_i}}{\sum_j e^{y_j}} - \left(\frac{e^{y_i}}{\sum_j e^{y_j}}\right)^2 \\
&= \phi(y_i) - \phi(y_i)^2 \\
&= \phi(y_i)(1 - \phi(y_i))
\end{aligned} \tag{3.35}$$

3.3.4 Hyperbolic Tangent Function

Similar to Sigmoid function, but if the input has a mean of 0 then so will the output.

$$\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.36}$$

And it's derivative:

$$\frac{d\phi(x)}{dx} = 1 - \tanh(x) = \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \tag{3.37}$$

3.3.5 Linear Function

$$\phi(x) = x \tag{3.38}$$

3.3.6 Rectified Linear Unit

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

$$\phi(x) = \max(0, x) \tag{3.39}$$

And it's derivative:

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

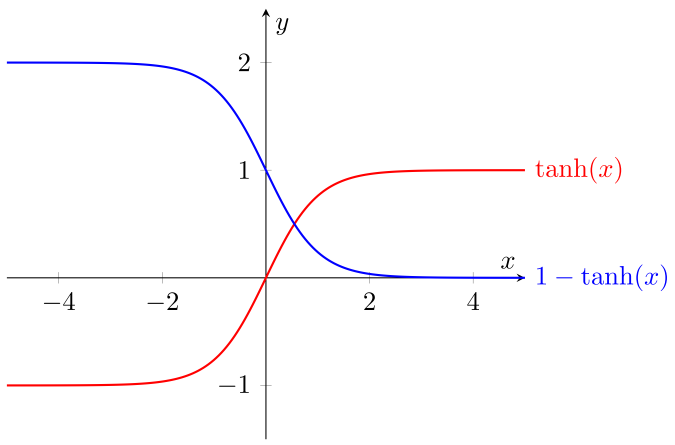

Figure 3.5: Hyperbolic Tangent Function (red) and its derivative (blue).

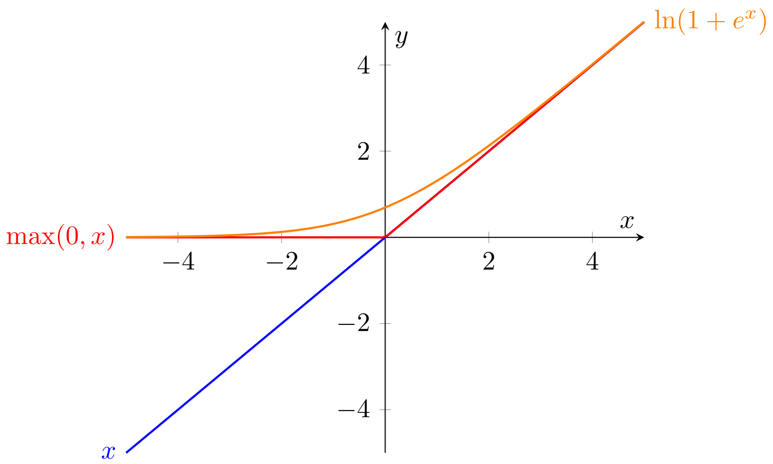

Figure 3.6: Linear Function (blue), Rectified Linear Unit (red), Softplus (orange)

3.3.7 Softplus

Smoothed version of ReLU.

$$\phi(x) = \ln(1 + e^x) \tag{3.40}$$

And it's derivative:

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

3.3.8 Maxout

Outputs the maximum of its inputs

$$a_j^{(l)} = \max_i z_i^{(l)}, \quad (j-1)g + 1 \leq i \leq jg \tag{3.41}$$

$$z_i^{(l)} = \sum_k x_k^{(l-1)} w_{ki}^{(l)} + b^{(l)} \tag{3.42}$$

Chapter 4

Boltzmann Machines

4.1 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.

4.1.1 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

$$\phi(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{4.1}$$

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.

4.1.2 Energy function

We define the energy in our Hopfield net as:

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} x_i x_j w_{ij} \tag{4.2}$$

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.

4.1.3 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

$$\begin{align} E_j &= -\frac{1}{2} \sum_i x_i x_j w_{ij} \tag{4.3} \\ \Delta E &= E_j(t+1) - E_j(t) \tag{4.4} \\ &= -\frac{1}{2} \left[x_j(t+1) \sum_i x_i w_{ij} - x_j(t) \sum_i x_i w_{ij} \right] \tag{4.5} \\ &= -\frac{1}{2} \Delta x_j \sum_i x_i w_{ij} \quad \text{with} \quad \Delta x_j = x_j(t+1) - x_j(t) \tag{4.6} \end{align}$$

Change from 0 to 1

$$\Delta x_j = 1, \sum_i x_i w_{ij} > 0 \Rightarrow \Delta E_j \leq 0 \tag{4.7}$$

Change from 1 to 0

$$\Delta x_j = -1, \sum_i x_i w_{ij} \leq 0 \Rightarrow \Delta E_j \leq 0 \tag{4.8}$$

4.1.4 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.

4.1.5 Training

A simple non-incremental learning rule is:

$$w_{ij} = \sum_{x \in X} (2x_i - 1)(2x_j - 1) \tag{4.9}$$

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

$$\frac{m}{4\ln(m)} < C < \frac{m}{2\ln m} \tag{4.10}$$

, where m is the number of neurons.

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

4.1.6 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

4.2 Boltzmann Machines

Boltzmann machines are stochastic recurrent neural networks.

Boltzmann machines work similar to Hopfield nets. They have a binary state vector ($x_j \in 0, 1$), but the new state a neuron x_j is determined stochastically (based on the total input z_j) on each update of x_j.

$$z_j = b_j + \sum_i x_i w_{ij} \tag{4.11}$$

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

$$p(x_j = 1) = \sigma(z_j) = \frac{1}{1 + e^{-z_j}} \tag{4.12}$$

, otherwise the activation of x_j becomes 0.

In general Boltzmann machines are unrestricted (see 4.3 for restricted Boltzmann machine (RBM)), but have the properties:

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

One can define input neurons that have the fixed input value during evaluation.

4.2.1 Energy

The energy of a state x is defined by

$$E(x) = -\sum_i x_i b_i - \frac{1}{2} \sum_{i<j} x_i x_j w_{ij} \tag{4.13}$$

With this we can calculate the probability of a state x

$$p(x) = \frac{e^{-E(x)}}{\sum_y e^{-E(y)}} \tag{4.14}$$

In average updating the neurons will decrease the energy in the network.

4.2.2 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 $\beta = \frac{1}{T}$.

$$p(x_j = 1) = \frac{1}{1 + e^{\frac{-z_j}{T}}} \tag{4.15}$$

Figure 4.1 shows curves with different temperature.

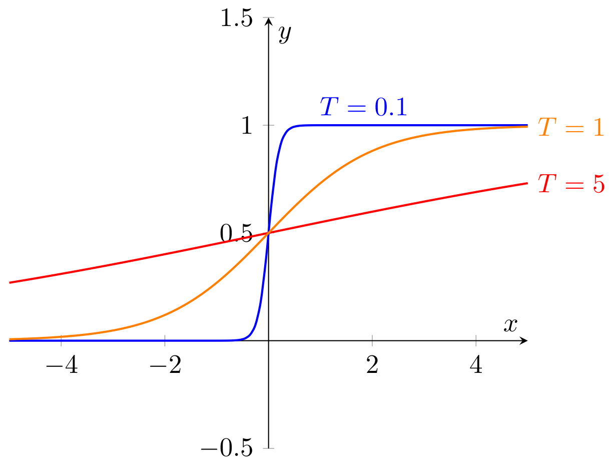

Figure 4.1: Sigmoid function (orange), with high temperature (red) and low temperature (blue).

4.2.3 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.

4.3 Restricted Boltzmann Machines

For an efficient training a Boltzmann machines can be restricted to have a bipartite graph with one set of visible neurons and on set of hidden neurons. An shown in figure 4.2 there are only visible-hidden and hidden-visible connection (still symmetric).

4.4 Stacking RBMs

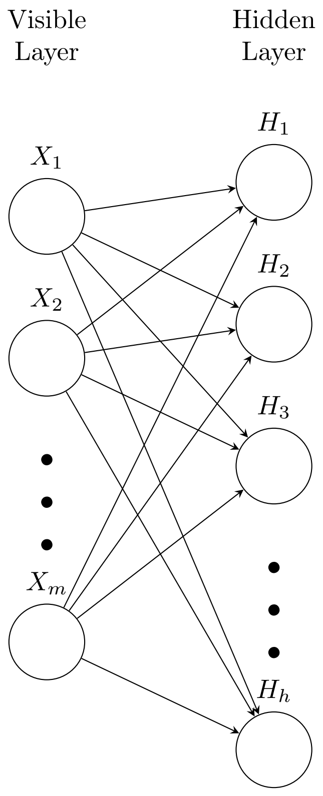

Figure 4.2: Architecture of a RBM

Chapter 5

Autoencoders

5.1 Training

5.2 Spares Autoencoders

5.3 Denoising Autoencoders

5.4 Stacking Autoencoders

Chapter 6

Practical Tricks

6.1 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),

$$\hat{x}_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{6.1}$$

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

$$\hat{x}_i = \frac{x_i - \text{mean}(x)}{\text{std}(x)} = \frac{x_i - \mu}{\sigma} \tag{6.2}$$

, 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.

6.2 Bottleneck-Features

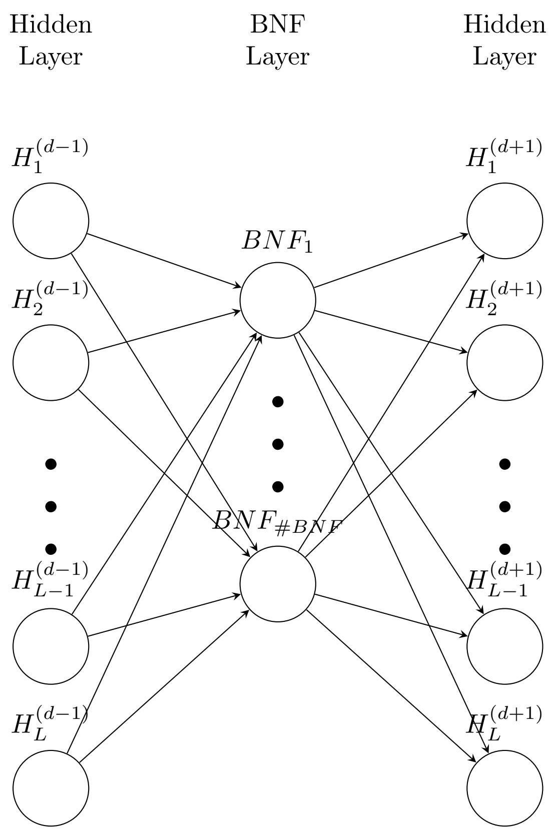

Figure 6.1: Neural network with Bottleneck features

Bibliography

[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.

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

[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.

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