1 A short view on the math and algorithmic steps for error back propagation on a MLP

Artificial Neural Networks [ANNs] can be trained for classification tasks. An analysis of simple Multi-Layer-Perceptrons [MLPs] shows:

> Optimizing the MLP's weights by training can be done iteratively for a whole set of training data. The expected correct classification results for all training samples have to be known in advance.
>
> During each iteration one first has to determine the **deviations** of the forward propagation results for the training data *from* the known correct results the MLP should produce. Then one has to perform a kind of backward directed propagation of error-related terms throughout the grid. The back-propagation gives you correction values for the weights at all nodes.

This method is called **"Error Back-Propagation"** [EBP]. It is a core ingredient of the so called "gradient descent" method applied to suitably defined *cost functions* for the MLP. The hope is that the named iterations sooner or later converge and that the step-wise applied corrections produce reasonable weight-values in the end.

The questions we are going to answer in this paper are:

- Why and in which form does the optimization of MLP-weights require EBP?
- What kind of matrix operations can we use to create an efficient EBP-algorithm?

1.1 Motivation for this paper

Beginners in Machine Learning [ML] – as myself – quickly grasp the idea how a MLP processes data. The mechanisms of the so called "*forward propagation*" through a network of multiple layers with nodes and the role of weights in this process are fairly easy to comprehend. We also understand that we somehow have to tune the parameters of the network, i.e. its weights, to get reasonable output or prediction values from the network.

Most introductory books on ML discuss the following standard method for weight optimization:

> Define a reasonable convex *cost function* depending indirectly on all weights and use the "***gradient descent***" method to find a global minimum of the cost function.

Later on in the study books we learn that gradient descent mathematically induces EBP in a special form - *somehow*. However, the details of this "*somehow*" often remain in the dark.

1.1.1 Cost functions and gradient descent

In the case of ANNs a virtual "**cost function**" weighs the deviations of the ANN's output from expected results after the propagation of training data. A properly defined cost function gives you scalar values. It thus defines a hyperplane in a multidimensional configuration space, whose (numerous) dimensions are defined by the weights; there is one axis per weight and one for the cost values.

The extrema of the cost-function's hyperplane can be investigated by step-wise exploring paths along the multidimensional surface. The direction to take at a specific surface point is indicated by the direction of the local ***gradient***.

The idea is based on the fact that the gradient on a multidimensional surface points into the direction of biggest local value change; the gradient has a direction perpendicular to local contour lines. Thus the name "gradient descent" as we are told to stepwise follow the gradient until we reach a (hopefully global) minimum of the surface.

Many books offer you nice pictures illustrating the principles of the gradient descent method, which we all understand intuitively. But then comes a major and laborious obstacle:

"Gradient descent" requires the calculation of gradients, i.e. partial derivatives. Here: Partial derivatives of the cost function with respect to all weights at *all* nodes on *all* MLP-layers. When you start doing the math for weights on some inner MLP-layers you soon find out that this is tedious work. And the results depend of course on details of the cost function and the activation function. The calculation is so laborious that we tend to just accept the end result, which tells us: perform EBP and leave the details to some clever ML-framework.

In this paper we shall not give up so fast: We shall discuss the mathematical origin of EBP step by step for a sufficiently general and complicated MLP-example. As said, **the discussion is intended for beginners** in Machine Learning and Artificial Neural Networks [ANNs] – as myself – who want to understand some of the basic math behind modern framework based ANN/MLP-codes.

In particular, we shall look at operations to deal with multiple training samples in parallel.

1.1.2 Why could a deeper understanding be useful?

Many introductory books on machine learning [ML] present the algorithmic *results* of the mathematical analysis behind EBP, but explain the math itself only briefly ─ if at all. Examples are the book of Rashka [1] and the book of A. Géron [2].

On the other hand side some elementary books as of T. Rashid [3] or M. Taylor [4] are focused on too specialized error functions with a simple quadratic form, or present a very specific discussion for a net with only *one* hidden layer.

Quite often also only one training sample is discussed, but not a whole set of samples, i.e. a so called (*mini-) batch of samples*. Such batches are typically handled in parallel by efficient code versions for MLPs. However, you learn in the books named above that dealing with "mini-batches" of multiple training data samples combines computational performance with some positive theoretical founded aspects during gradient descent. So dealing with the partial derivatives for *many* training samples is of major importance.

The relatively new book of M. Deru and A. Ndiaye [5] on "Deep Learning" stresses a purely algorithmic approach to all these questions in introductory chapters, because the authors feel it to be easier for SW-developers to access the topics in code or pseudo code language. Therefore, some results are directly presented in the form of Python code with matrix related functions of the Numpy library. Deeper theoretical or mathematical considerations are intentionally omitted. As EBP takes a

surprisingly simple form in the end such an approach is really tempting – not only for SW-developers.

In addition, available frameworks like Google's Tensorflow and frontends like Keras make it quite easy for ML-students to circumvent mathematical considerations. One just chooses from configuration options and lets the black-box algorithms work.

When authors as Rashka dive into Python based coding examples you stumble sooner or later across the use of the "**dot**(A,B)"-function of the Numpy library; it corresponds to a rather complicated matrix operation for two *multi*-dimensional arrays or matrices based on Linear Algebra fundamentals. It has to be distinguished from a more simple "(A*B)"-function between two multi-dimensional arrays of equal dimensions. But when looking at example codes for general MLPs with multiple hidden layers it is not trivial to see why EBP results in a certain succession of both kinds of operations.

Personally, I do not like math-free approaches to ANNs at all because they foster a dependency of a whole SW-developer generation on the knowledge of others, especially big global data companies. This hinders or even blocks out independent, fresh approaches and improvements on a basic theoretical level. We forget too often that the ANN methods and frameworks used today cover only a limited part of all real world problems which at least in principle could be formulated in terms of ANNs:

After having worked a bit in the field of supply chain problems I have understood that there are real world logistics problems which could be transformed into a form similar to the propagation through ANNs, but where conventional ANN-methods would have to be modified extensively to achieve appropriate and convincingly working optimization methods. This begins with the fact that cost functions in logistics would be defined for bunches of network nodes – and thus any kind of back propagation has to modified substantially. In addition, the occurrence and circumvention of local minima becomes a real challenge.

=> ML is too important to leave it to global firms or elite universities!

1.2 Approach and topics in this paper

My approach in this paper is to motivate theory based on an explicit example. To see the algorithmic patterns arising I shall discuss a four layer MLP as a study example.

Why **4** layers?

In my opinion, one needs at least ***two*** hidden layers to avoid simplifying too much when we try to pin down partial derivative terms. Only with at least two hidden layers you face the full challenge of the dependency of partial derivatives on *multiple different paths* from the output layer throughout the grid to nodes of inner layers.

We shall in addition look at the logarithmic "Log-Loss"-function, which is a bit more complicated than just a quadratic MSE-function used in some introductory books.

The derivative of the ***sigmoid*** activation function will be discussed explicitly.

Most important: We shall discuss mini-batches to see why and which matrix operations can support an efficient handling of multiple training samples in parallel. We shall learn how to formulate the EBP in a so called “vectorized” form for a simultaneous treatment of many training samples.

Requirements regarding the knowledge of the reader are:

- a basic understanding of MLPs,
- an understanding of the gradient descent method,
- a familiarity with the chain rule for differentiation
- and the will to become acquainted with some matrix operations.

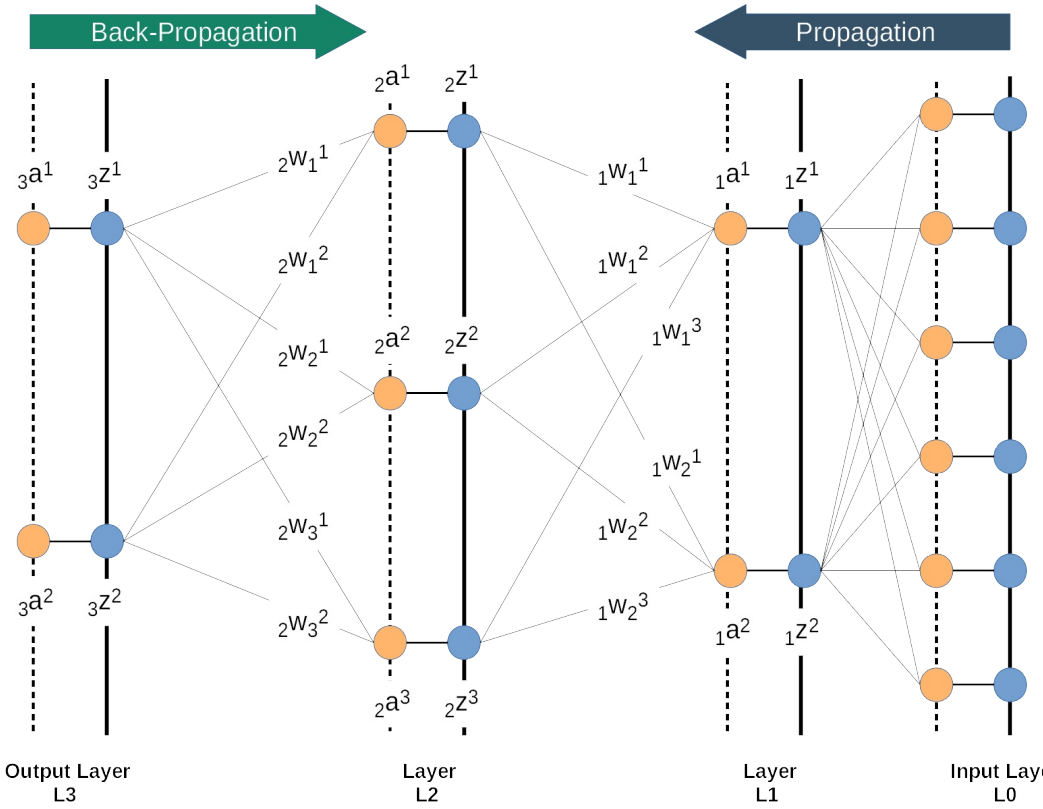
It is a long time ago that I went to University, but most of these topics were elements of basic introductory courses into analysis, linear algebra and vector analysis for undergraduate students of engineering, physics and informatics.

For anybody who needs a quick introduction to some of the math I strongly recommend to get a hand on the book “*Vector analysis and Cartesian tensors*” of D.E. Bourne and P.C. Kendall.

The discussion in this paper also supports the basic (Python based) coding example for a multilayer MLP which I present in an article series in my Linux blog for ML-beginners. See
https://linux-blog.anracom.com/2019/09/29/a-simple-program-for-an-ann-to-cover-the-mnist-dataset-i/
and the articles thereafter.

1.3 Our example MLP

We look at a 4-layer MLP. The network is displayed in the following sketch:



Some hints for a better understanding:

- All nodes of a layer L(n) are connected to all nodes of neighboring layers L(n+1) or L(n-1) in one or both horizontal directions.
- For our discussion of the math it is not important that the node distribution across the layers is a realistic one.
  Typically, the first hidden layer would contain more nodes than just two; a larger number of nodes is required to account for a sufficient representation of a reasonable clustering of data features. Remember, the input data space may have very many (hundreds to thousands) dimensions to account for all the features of the input data, and classification may not allow for a dramatic dimension reduction already on the second network layer.
- Information is transported through our net *from the right to the left*. Information – characterized by a numeric value of the quantity "*z*" entering a node is transformed into information leaving the node – this time characterized by a numerical value "*a*".
- The left sub-fix on the "*a*" and "*z*"-quantities represents the layer. The right super-fix of "*a*" and "*z*" represents the node. In our drawing we number nodes from the top to the bottom with rising index-numbers.
- The left sub-fix on the weights ${}_{n}w_{i}^{j}$ for a connection between two layers L(n) and L(n+1) represents the right origin layer L(n) in standard propagation direction. The right super-fix of "w" represents the target node on the right layer L(n+1), the right sub-fix instead represents the origin node on Layer L(n).

As mentioned, we have indicated the transition of input information "*z*" at a **node** into an output value "*a*" at the very same node by horizontal connector lines and a recoloring: Both a blue and a left neighboring yellow circle mean one and the same node on the layer – but in two different states regarding the information processed at this node.

In standard propagation direction ${}_{n}a^{i}$ at a node "*i*" is split into multiple *fragments*. The contribution of each fragment to ${}_{n+1}z^{j}$ is calculated by the multiplication of ${}_{n}a^{i}$ by a weight ${}_{n}w_{i}^{j}$ specific for the connection to a target node "*j*" on a neighboring layer to the left.

Each fragment is transported along its connection to a target node of the next layer. At each node incoming contributions to the "*z*"-value from different nodes of the previous layer are just summed up[1].

2 Forward propagation and multiple training samples

Let us first investigate how we deal with forward propagation between layer L1 and layer L2 – for one training sample *Sx*. To do this we need to talk a bit about matrix operations.

1 Note that this rule could at least in principle be replaced by a more complex one. Physicist may e.g. think about a rule describing the superposition of wave functions according to rules of quantum mechanics at some points of a grid. You may have nodes where wave functions just add up and others where detectors are placed and superpositions are turned into probabilities by evaluating quadratic amplitude values.

Let us assume we have two 2-dimensional matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ with the number of columns in matrix $\boldsymbol{X}$ being equal to the number of rows in matrix $\boldsymbol{Y}$ – as e.g. in

$$\boldsymbol{X} = \begin{bmatrix} x_1^1 & x_2^1 \\ x_1^2 & x_2^2 \\ x_1^3 & x_2^3 \end{bmatrix}, \quad \boldsymbol{Y} = \begin{bmatrix} y^1 \\ y^2 \end{bmatrix} \tag{1}$$

Then we can define the following operation $\boldsymbol{X} \odot \boldsymbol{Y}$ as

$$\boldsymbol{C} = \boldsymbol{X} \odot \boldsymbol{Y} := \begin{bmatrix} x_1^1 * y^1 + x_2^1 * y^2 \\ x_1^2 * y^1 + x_2^2 * y^2 \\ x_1^3 * y^1 + x_2^3 * y^2 \end{bmatrix} \tag{2}$$

We sum over products of *column* elements in $\boldsymbol{X}$ with corresponding *row* elements in $\boldsymbol{Y}$. It requires that the number of matrix elements in a row of $\boldsymbol{X}$ is equal to the number of elements in a column of $\boldsymbol{Y}$. The kind reader has of course already understood that our example above corresponds to a standard matrix operation on a vector – well known from Linear Algebra.

Let us now look at two other matrices, which comprise elements depicted in the drawing of our MLP-example; see layer L1:

$$_1\boldsymbol{W} = \begin{bmatrix} {}_1w_1^1 & {}_1w_2^1 \\ {}_1w_1^2 & {}_1w_2^2 \\ {}_1w_{12}^3 & {}_1w_2^3 \end{bmatrix}, \qquad {}_1\boldsymbol{A}_{\boldsymbol{S1}} = \begin{bmatrix} {}_1a_{\boldsymbol{S1}}^1 \\ {}_1a_{\boldsymbol{S1}}^2 \end{bmatrix} \tag{3}$$

We apply our $\odot$ -operation as $_2\boldsymbol{Z}_{\boldsymbol{S1}} = {}_1\boldsymbol{W} \odot {}_1\boldsymbol{A}_{\boldsymbol{S1}}$ and get the following components:

$$_2\boldsymbol{Z}_{\boldsymbol{S1}} = \begin{bmatrix} {}_2z_{\boldsymbol{S1}}^1 \\ {}_2z_{\boldsymbol{S1}}^2 \\ {}_2z_{\boldsymbol{S1}}^3 \end{bmatrix} = \begin{bmatrix} {}_1w_1^1 & {}_1w_2^1 \\ {}_1w_1^2 & {}_1w_2^2 \\ {}_1w_{12}^3 & {}_1w_2^3 \end{bmatrix} \odot \begin{bmatrix} {}_1a_{\boldsymbol{S1}}^1 \\ {}_1a_{\boldsymbol{S1}}^2 \end{bmatrix} = \begin{bmatrix} {}_1w_1^1 * {}_1a^1 + {}_1w_2^1 * {}_1a_{\boldsymbol{S1}}^2 \\ {}_1w_1^2 * {}_1a^1 + {}_1w_2^2 * {}_1a_{\boldsymbol{S1}}^2 \\ {}_1w_1^3 * {}_1a^1 + {}_1w_2^3 * {}_1a_{\boldsymbol{S1}}^2 \end{bmatrix} \tag{4}$$

In this case the number of columns in the first matrix **W** were equal to the number of rows in the second matrix **A**.

How can we define the meaning of the $\boldsymbol{X} \odot \boldsymbol{Y}$ - operation in a case where we have more than 2 dimensions in $\boldsymbol{X}$ and more than 1 in $\boldsymbol{Y}$?

In such a case, we interpret the operation in the sense of the "**dot** (X,Y)"-operation defined for Python Numpy arrays. Python programmers should feel comfortable now :-).

Let us look at this operation for a case with

- $\boldsymbol{X}$ having 3 dimensions indicated by indices *{i, j, u}* and
- $\boldsymbol{Y}$ having 3 dimensions indicated by indices *{k, v, m}*

Then, if the number of elements in dimension "*u*" in $\boldsymbol{X}$ is equal to the number of elements in dimension "*v*" of $\boldsymbol{Y}$, we define in Python notation:

```
dot(X,Y)[i,j,k,m] = sum(X[i,j,:] * Y[k,:,m]),
```

The numbers in the brackets identify the position of the element in the matrix. Meaning:

$$\boldsymbol{C}[i,j,k,m] = [\boldsymbol{X} \odot \boldsymbol{Y}][i,j,k,m] := \sum_{s} \left(\boldsymbol{X}[i,j,s] * \boldsymbol{Y}[k,s,m] \right) \tag{5}$$

(We have indicated by the brackets around $[\boldsymbol{X} \odot \boldsymbol{Y}]$ that the operation $\boldsymbol{X} \odot \boldsymbol{Y}$ results in a matrix in which elements are indexed by $[i,j,k,m]$.)

I.e., our generalized form of $\boldsymbol{X} \odot \boldsymbol{Y}$ means:

> We sum over products of corresponding elements along the last dimension of $\boldsymbol{X}$ and elements along the second to last dimension in $\boldsymbol{Y}$.

Note that in case of a 2-dimensional $\boldsymbol{X}$ the last dimension is normally given by columns, and in the case of a 2-dimensional $\boldsymbol{Y}$ the second to last dimension addresses rows. This particular case is relevant both for forward and back propagation in MLPs as well shall see in some minutes.

2.1 Applying weights for a mini-batch during forward propagation

Now, with our generalization of the $\odot$ -operation, we can write down a well defined matrix-operation for a whole "mini-batch" of training samples **{ s1, s2, s3,** …. **sn }**:

$$\begin{bmatrix} {}_2z^1_{S1}, & {}_2z^1_{S2}, & \ldots., & {}_2z^1_{Sn} \\ {}_2z^2_{S1}, & {}_2z^2_{S2}, & \ldots., & {}_2z^2_{Sn} \\ {}_2z^3_{S1}, & {}_2z^3_{S2}, & \ldots., & {}_2z^3_{Sn} \end{bmatrix} = \begin{bmatrix} {}_1w^1_1 & {}_1w^1_2 \\ {}_1w^2_1 & {}_1w^2_2 \\ {}_1w^3_1 & {}_1w^3_2 \end{bmatrix} \odot \begin{bmatrix} {}_1a^1_{S1}, & {}_1a^1_{S2}, & \ldots., & {}_1a^1_{Sn} \\ {}_1a^2_{S1}, & {}_1a^2_{S2}, & \ldots., & {}_1a^2_{Sn} \end{bmatrix} \tag{6}$$

We speak of a "*vectorized*" operation for all samples in parallel. Libraries in Python support and optimize such vectorized operations on modern CPUs.

We use the following abbreviations:

$${}_1\boldsymbol{A}_S = \begin{bmatrix} {}_1a^1_{S1}, & {}_1a^1_{S2}, & \ldots., & {}_1a^1_{Sn} \\ {}_1a^2_{S1}, & {}_1a^2_{S2}, & \ldots., & {}_1a^2_{Sn} \end{bmatrix}, \quad {}_2\boldsymbol{Z}_S = \begin{bmatrix} {}_2z^1_{S1}, & {}_2z^1_{S2}, & \ldots., & {}_2z^1_{Sn} \\ {}_2z^2_{S1}, & {}_2z^2_{S2}, & \ldots., & {}_2z^2_{Sn} \\ {}_2z^3_{S1}, & {}_2z^3_{S2}, & \ldots., & {}_2z^3_{Sn} \end{bmatrix}, \tag{7}$$

for matrices representing the "*a*"-values at all nodes in Layer L1 and the "*z*"-values at all nodes in layer L2, and

$$ {}_1\boldsymbol{W} \;=\; \begin{bmatrix} {}_1w_1^1 & {}_1w_2^1 \\ {}_1w_1^2 & {}_1w_2^2 \\ {}_1w_1^3 & {}_1w_2^3 \end{bmatrix} \tag{8} $$

for the weight matrix coupling layer L1 with layer L2. The index on the left side indicates the righter one of the two layers for which $ {}_1\boldsymbol{W} $ controls the information transport.

This notation makes us a bit independent of how many nodes we actually have on both layer L1 and layer L2; we can write in general

$$ {}_2\boldsymbol{Z}_S \;=\; {}_1\boldsymbol{W} \;\odot\; {}_1\boldsymbol{A}_S \;\; . \tag{9} $$

$ {}_1\boldsymbol{A}_S $ has as many rows as nodes on layer L1 and as many columns as there are samples in our mini-batch. $ {}_2\boldsymbol{Z}_S $ has as many rows as there are nodes on L2 and as many columns as there are samples.

Of course, we can write down analogous expressions for the other layers.

This is all consistent with the forward-propagagtion-operations discussed in my blog article

> **A simple program for an ANN to cover the MNIST dataset – III – forward propagation**
>
> (https://linux-blog.anracom.com/2019/10/07/a-simple-program-for-an-ann-to-cover-the-mnist-dataset-iii-forward-propagation/)

2.2 Element-wise matrix multiplications

The term $ \boldsymbol{X} \odot \boldsymbol{Y} $, which describes a rather complex operation between two matrices, has to be carefully distinguished from another operation: An element-wise multiplication of two matrices with identical dimensions. For 2-dimensional (n_i x m_j)-matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ we can define

$$ \boldsymbol{C} \;=\; \boldsymbol{X} \;*\; \boldsymbol{Y}, \quad \textit{with elements} \quad C_i^j \;=\; X_i^j \;*\; Y_i^j \tag{10} $$

"j" and "i" index the position of an element in the two distinguished dimensions, i.e. the row- and column-number, respectively.

Of course, with a bit different notation for element indexing, this can be generalized to multiple dimensions

$$ C[i,j,k,m,...] \;=\; X[i,j,k,m,...] \;*\; Y[i,j,k,m,...] \tag{11} $$

This makes only sense if the dimensions of the two matrices

$$ \left[\, n_i \;\times\; n_j \;\times\; n_k \;\times\; n_m \;\times\; .... \,\right] \tag{12} $$

are identical; the matrices must have the same number of elements in each dimension direction.

2.3 Applying an *activation function* during forward propagation

The transformation of the "*z*"-vector-data into "*a*"-vector data at a specific node means the application of an **activation function** ϕ to each of the elements of the ***Z***-matrix.

In case of the second layer L2 this element-wise operation gives us the values of ${}_2\boldsymbol{A_S}$:

$$ {}_2\boldsymbol{A_S} := \boldsymbol{\phi}\left({}_2\boldsymbol{Z_S}\right) = \begin{bmatrix} {}_2a^1_{S1}, & {}_2a^1_{S2}, & ...., & {}_2a^1_{Sn} \\ {}_2a^2_{S1}, & {}_2a^2_{S2}, & ...., & {}_2a^2_{Sn} \\ {}_2a^3_{S1}, & {}_2a^3_{S2}, & ...., & {}_2a^3_{Sn} \end{bmatrix} = \begin{bmatrix} \phi({}_2z^1_{S1}), & \phi({}_2z^1_{S2}), & ...., & \phi({}_2z^1_{Sn}) \\ \phi({}_2z^2_{S1}), & \phi({}_2z^2_{S2}), & ...., & \phi({}_2z^2_{Sn}) \\ \phi({}_2z^3_{S1}), & \phi({}_2z^3_{S2}), & ...., & \phi({}_2z^3_{Sn}) \end{bmatrix} . \quad (13) $$

For our MLP-example we use the **sigmoid** function – below generally defined for a node with index "*j*" on layer L(N):

$$ {}_N a^j_{Sx} = \phi\left({}_N z^j_{Sx}\right) := \frac{1}{1 - e^{-{}_N z^j_{Sx}}} . \quad (14) $$

"*Sx*" again refers to a specific training sample.

We assume below that the output function, i.e. the activation function on the output layer, is identical to the sigmoid function.

2.4 The derivative of the sigmoidal activation function

Our ultimate goal is to find a formula for the partial derivatives of a cost function defined for the MLP with respect to the variables of the MLP – i.e. the weights ${}_n w^j_i$ defined at each node of all layers.

Looking at our drawing gives us the clear impression that there are subsequent dependencies on variables which have to be resolved: We need to use the chain rule of calculus to get the partial derivatives.

It is clear that due to this rule we will sooner or later need the derivative of "*ϕ(z)*" with respect to "*z*". This derivative is given by:

$$ \frac{\partial \phi}{\partial z} = \frac{\partial}{\partial z}\left(\frac{1}{1 + e^{-z}}\right) = \frac{e^{-z}}{\left(1 + e^{-z}\right)^2} = \frac{\left(1 + e^{-z}\right)}{\left(1 + e^{-z}\right)^2} - \frac{1}{\left(1 + e^{-z}\right)^2} \quad (15) $$

A closer look reveals that we can conveniently rewrite this expression in terms of ϕ itself:

$$ \frac{\partial \phi(z)}{\partial z} = \phi(z) * \left(1 - \phi(z)\right) \quad (16) $$

We conclude that for each node "*j*" on a layer "*L(N)*" and a specific sample "*Sx*" we get the following relation:

$$\frac{\partial\ {}_N a^j_{Sx}}{\partial z} = {}_N a^j_{Sx} * \left(1 - {}_N a^j_{Sx}\right) \ . \tag{17}$$

3 “Log Loss” as the cost function of our MLP

At the output layer (here layer L3) we shall define a special cost function for our MLP[2]. We choose our cost function to be the so called “Log Loss”-function. (Note: We use the terms “cost function” and “loss function” as synonyms below.)

Let us assume that the correct value which the ANN should produce at a node “*j*” on the output layer (here at layer L3) is y^j_{Sx} for a certain test sample *Sx*.

Then we define a function “$\lambda({}_3 a^j)$” for every node of layer L3 and for a specific training sample “*Sx*”:

$$\lambda\left({}_3 a^j_{Sx}\right) := -\left[\ y^j_{Sx} \log\left({}_3 a^j_{Sx}\right) - \left(1 - y^j_{Sx}\right) \log\left(1 - {}_3 a^j_{Sx}\right)\ \right] \tag{18}$$

We next define a loss function for a full mini-batch – using the “λ “-term given above.

In a complete training set for an ANN we may have tenth or hundreds of thousands of samples. However, a so called mini-batch may comprise only hundreds or thousands of samples. A sub-fix “*k*” shall identify a special mini-batch “B_k”.

Our cost or loss function “**Λ**” for a mini-batch “B_k” would then be

$$\begin{aligned} \Lambda_{B_k} &= \frac{1}{n} \sum_{\forall Sx \in B_k} \ \sum_{\forall j\ on\ L3} \lambda\left({}_3 a^j_{Sx}\right) = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \ \sum_{\forall j\ on\ L3} \left[\ y^j_{Sx} \log\left({}_3 a^j_{Sx}\right) - \left(1 - y^j_{Sx}\right) \log\left(1 - {}_3 a^j_{Sx}\right)\ \right] \end{aligned} \tag{19}$$

I.e.: We sum up the “λ “-terms for all nodes of the output layer and all samples of the mini-batch.

Of course each of the cost functions for different mini-batches will show a different value for a *given* set of weight values. Also the correction values for the weights calculated for each mini-batch during an optimization iteration may jump a bit around. If each iteration for weight corrections were performed for all mini-batches and the costs for each step and for each batch were plotted we would get the impression that the cost function values vary a bit erratically around some mean average value. However, adjusting the weights over many iterations during many optimization epochs should lead to development with *all* of these cost functions systematically approaching a minimum at common values of the weights.

2 The fact that we define a cost function which depends on certain values given at output nodes, only, is indeed a simplification almost all teaching books follow – maybe unfortunately… But we also walk along this line in this paper.

Note: The cost function is a scalar. Not an array, not a vector. However, the partial derivatives with respect to all weights define a ***gradient*** of the cost function in a multidimensional vector space defined by

- the axes for real values of all the MLP's weights, each varying within $\Re$ (being the continuum space of real numbers)
- and one axis for the (real) values of "Λ" - changing with the values of all weights.

Thus, if there are "*m*" different weights in total then the cost function "Λ" defines a *m*-dimensional hyperplane in a $\Re^{(m+1)}$ space.

Note: The tangential gradient of "Λ" would show in a direction perpendicular to contour lines on the hyperplane – indicating the direction of maximum change. This is a fundamental result of multidimensional vector analysis.

4 Analysis of the partial derivatives of the summands in the cost function of our study MLP

Let us for a moment forget that we deal with a whole bunch of training data samples in a mini-batch. We shall later adjust for the necessary summations.

The "Log Loss" function looks complicated. But, actually, the derivative of each summand with respect to "$_3a^j$" takes a simple and useful form in our study case. We write it down for the case of just one training sample:

$$\frac{\partial \lambda\left({}_3a^j_{Sx}\right)}{\partial\left({}_3a^j_{Sx}\right)} = -\left[\frac{y^j_{Sx}}{{}_3a^j_{Sx}} - \left(1 - y^j_{Sx}\right)\frac{1}{1 - {}_3a^j_{Sx}}\right] = -\left(y^j_{Sx} - {}_3a^j_{Sx}\right)\frac{1}{{}_3a^j_{Sx} * \left(1 - {}_3a^j_{Sx}\right)} \quad . \quad (20)$$

Note the difference $\left({}_3a^j_{Sx} - y^j_{Sx}\right)$ of the correct value minus the calculated one (i.e. the propagated value) for the training sample and for the presently given weights. We call this difference the ***error*** which our MLP produces for a test sample at node "*j*" of the output layer.

We give the right side of the above equation a new name:

$${}_{out}\delta^j_{Sx} := \left({}_3a^j_{Sx} - y^j_{Sx}\right) * \frac{1}{{}_3a^j_{Sx} * \left(1 - {}_3a^j_{Sx}\right)} \quad . \quad (21)$$

This expression will help us to denote the partial derivatives of the cost function in a systematic way later on. In our study case this is nothing else than

$${}_{out}\delta^j_{Sx} = \frac{\partial \lambda\left({}_3a^j_{Sx}\right)}{\partial\left({}_3a^j_{Sx}\right)} = \left({}_3a^j_{Sx} - y^j_{Sx}\right) * \left[\frac{\partial \phi\left({}_3a^j_{Sx}\right)}{\partial\left({}_3a^j_{Sx}\right)}\right]^{-1} \quad . \quad (22)$$

It is a good exercise to show that something analogous happens for the quadratic MSE loss function discussed in the literature listed in chapter 6.

4.1 Partial derivative of the cost function with respect to the weights of the second to last layer

The first interesting partial derivatives of the cost function would in our example be those which are derived with respect to weights ${}_2w_i^j$ on Layer L2.

By using the rules for derivatives of a sum over multiple summands ***and*** the chain rule for complex functions we derive the partial derivatives of Λ_{B_k}. Its summands define derivatives of the individual contributions to the cost function coming from *all* nodes of layer L3:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall j \text{ on } L3} \left[\frac{\partial \lambda \left({}_3a_{Sx}^j \right)}{\partial \left({}_3a_{Sx}^j \right)} * \frac{\partial \left({}_3a_{Sx}^j \right)}{\partial \left({}_3z_{Sx}^j \right)} * \frac{\partial \left({}_3z_{Sx}^j \right)}{\partial \left({}_2w_i^j \right)} \right] \tag{23}$$

We just used the chain rule on every summand on its constituting variable elements!

With our results from above this expression becomes:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall j \text{ on } L3} \left[{}_{out}\delta_{Sx}^j * {}_3a_{Sx}^j \left(1 - {}_3a_{Sx}^j \right) * \frac{\partial \left({}_3z_{Sx}^j \right)}{\partial \left({}_2w_i^j \right)} \right] \tag{24}$$

Note that in ${}_2w_i^j$ "*i*" indexes the source node on layer L2 and "*j*" the target node for forward propagation on layer L3!

If you look at the drawing in the introductory section you see that there is just *one* connection path to a node "*j*" on L3, where transport is influenced by ${}_2w_i^j$ for a specific "*i*" (in the set {1, 2, 3}).

So, actually the second sum is reduced to ***one*** term, only, for a given ${}_2w_i^j$:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^j * {}_3a_{Sx}^j \left(1 - {}_3a_{Sx}^j \right) * \frac{\partial \left({}_3z_{Sx}^j \right)}{\partial \left({}_2w_i^j \right)} \right] \tag{25}$$

Now, we define:

$${}_3\delta_{Sx}^j := {}_{out}\delta_{Sx}^j * {}_3a_{Sx}^j \left(1 - {}_3a_{Sx}^j \right) \tag{26}$$

Due to canceling of terms in our study case we get the simple relation

$${}_3\delta_{Sx}^j = \left({}_3a_{Sx}^j - y_{Sx}^j \right) , \tag{27}$$

which actually is the *error* our MLP produces for a training sample at a specific node of the output layer.

We shall see, however, that the notation including ${}_{out}\delta_{Sx}^{j}$ is very convenient, too.

Let us interpret the various ${}_{3}\delta_{Sx}^{j}$ for the nodes "*j*" on layer L3 as components of an array or vector including elements for all output nodes – i.e. the two output nodes in our example:

$$ {}_{3}\boldsymbol{\delta}_{Sx} = \begin{pmatrix} {}_{out}\delta_{Sx}^{1} * {}_{3}a_{Sx}^{1}\left(1 - {}_{3}a_{Sx}^{1}\right) \\ {}_{out}\delta_{Sx}^{2} * {}_{3}a_{Sx}^{2}\left(1 - {}_{3}a_{Sx}^{2}\right) \end{pmatrix} . \tag{28} $$

This will help us to formulate EBP in terms of matrix operations.

To achieve a working matrix operation we introduce the following notation, which formally is a bit dirty:

$$ {}_{3}\boldsymbol{D}_{Sx}^{\phi} := \frac{\partial^{c}\boldsymbol{\phi}\left({}_{3}\boldsymbol{z}_{Sx}\right)}{\partial^{c}\,{}_{3}\boldsymbol{z}_{Sx}} = \begin{bmatrix} \frac{\partial\,\phi\left({}_{3}z_{Sx}^{1}\right)}{\partial\,{}_{3}z_{Sx}^{1}} \\ \frac{\partial\,\phi\left({}_{3}z_{Sx}^{2}\right)}{\partial\,{}_{3}z_{Sx}^{2}} \end{bmatrix} $$

It denotes a component-wise differentiation of a vector resulting from applying function ϕ to the components of ${}_{3}\boldsymbol{z}_{Sx}$. We proceed with the following useful expression:

$$ {}_{3}\boldsymbol{\delta}_{Sx} := {}_{out}\boldsymbol{\delta}_{Sx} * {}_{3}\boldsymbol{D}_{Sx}^{\phi} = {}_{out}\boldsymbol{\delta}_{Sx} * \frac{\partial^{c}\boldsymbol{\phi}\left({}_{3}\boldsymbol{z}_{Sx}\right)}{\partial^{c}\,{}_{3}\boldsymbol{z}_{Sx}} = {}_{out}\boldsymbol{\delta}_{Sx} * \begin{bmatrix} \frac{\partial\,\phi\left({}_{3}z_{Sx}^{1}\right)}{\partial\,{}_{3}z_{Sx}^{1}} \\ \frac{\partial\,\phi\left({}_{3}z_{Sx}^{2}\right)}{\partial\,{}_{3}z_{Sx}^{2}} \end{bmatrix} . \tag{29} $$

Here, the "*" of course means an element-wise matrix multiplication.

Analogously, via an extension over all training samples the following expression has a well defined meaning, too:

$$ {}_{3}\boldsymbol{\delta}_{\boldsymbol{S}} := {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} * {}_{3}\boldsymbol{D}_{\boldsymbol{S}}^{\phi} := {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} * \frac{\partial^{c}\boldsymbol{\phi}\left({}_{3}\boldsymbol{Z}_{\boldsymbol{S}}\right)}{\partial^{c}\,{}_{3}\boldsymbol{Z}_{\boldsymbol{S}}} , \tag{30} $$

with

$$ {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} = \begin{pmatrix} {}_{out}\delta_{S1}^{1}, & {}_{out}\delta_{S2}^{1}, & \ldots\ldots, & {}_{out}\delta_{Sn}^{1} \\ {}_{out}\delta_{S1}^{'2}, & {}_{out}\delta_{S2}^{2}, & \ldots\ldots, & {}_{out}\delta_{Sn}^{2} \end{pmatrix} , \tag{31} $$

$$ {}_{3}\boldsymbol{Z}_{\boldsymbol{S}} = \begin{bmatrix} {}_{3}z_{\boldsymbol{S1}}^{1}, & {}_{3}z_{\boldsymbol{S2}}^{1}, & \ldots., & {}_{3}z_{\boldsymbol{Sn}}^{1} \\ {}_{3}z_{\boldsymbol{S1}}^{2}, & {}_{3}z_{\boldsymbol{S2}}^{2}, & \ldots., & {}_{3}z_{\boldsymbol{Sn}}^{2} \\ {}_{3}z_{\boldsymbol{S1}}^{3}, & {}_{3}z_{\boldsymbol{S2}}^{3}, & \ldots., & {}_{3}z_{\boldsymbol{Sn}}^{3} \end{bmatrix} , \tag{32} $$

and

$$ {}_3\boldsymbol{D}_{\boldsymbol{S}}^{\phi} := \frac{\partial^c \boldsymbol{\phi}\left({}_3\boldsymbol{Z}_{\boldsymbol{S}}\right)}{\partial^c {}_3\boldsymbol{z}_{\boldsymbol{S}}} := {}_3\left[\frac{\partial^c \boldsymbol{\phi}\left(\boldsymbol{Z}_{\boldsymbol{S}}\right)}{\partial^c \boldsymbol{z}_{\boldsymbol{S}}}\right] := \begin{bmatrix} \frac{\partial \phi\left({}_3z_{S1}^1\right)}{\partial {}_3z_{S1}^1}, & \frac{\partial \phi\left({}_3z_{S2}^1\right)}{\partial {}_3z_{S2}^1}, & \ldots., & \frac{\partial \phi\left({}_3z_{Sn}^1\right)}{\partial {}_3z_{Sn}^1} \\ \frac{\partial \phi\left({}_3z_{S1}^2\right)}{\partial {}_3z_{S1}^2}, & \frac{\partial \phi\left({}_3z_{S2}^2\right)}{\partial {}_3z_{S2}^2}, & \ldots., & \frac{\partial \phi\left({}_3z_{Sn}^2\right)}{\partial {}_3z_{Sn}^2} \end{bmatrix} \tag{33} $$

.

4.1.1 Elements of the gradient of the loss function with respect to weights of layer L2

For the gradient of the cost function we need to write down partial derivatives in an *ordered* form.

People trained in mathematics or physics would normally write a gradient in simple vector form, i.e. with just one column. But due to reasons which you will understand in a minute we write down the derivatives with respect to the weights of a given layer in a special matrix like form.

Let us look at those components of the gradient of "$\boldsymbol{\Lambda}$" which are related to the weights ${}_2w_i^j$ on layer L2. We order all the partial derivatives in the following way:

$$ {}_2\boldsymbol{\nabla}_{B_k} = \begin{bmatrix} \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_1^1\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_2^1\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_3^1\right)} \\ \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_1^2\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_2^2\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_3^2\right)} \end{bmatrix} . \tag{34} $$

Obviously, we distinguish the target nodes of layer L3 by rows and the source nodes on L2 by different columns.

We shall see that this rather special arrangement of terms will help us to deal with all the samples included in a mini-batch.

Let us now evaluate the last term in our general formula (25) for the partial derivatives:

$$ \frac{\partial \Lambda_{B_k}}{\partial\left({}_2w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^j * {}_3a_{Sx}^j \left(1 - {}_3a_{Sx}^j\right) * \frac{\partial\left({}_3z_{Sx}^j\right)}{\partial\left({}_2w_i^j\right)} \right] . \tag{35} $$

A look at our drawing shows that this simply is equal to

$$ \frac{\partial\left({}_3z_{Sx}^j\right)}{\partial\left({}_2w_i^j\right)} = {}_2a_{Sx}^i . \tag{36} $$

There is, understandably, no dependency on "*j*" but on "*i*"!

Using this result, we can easily write down some components of ${}_2\nabla_{B_k}$ in detail:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_1^1\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1\right) * {}_2a_{Sx}^1 \tag{37}$$

Note the significant difference in the layer indices for the different *a*-terms in this expression! The last "*a*" refers to layer L2, whilst the first two "*a*"s refer to layer L3!

Analogously we get:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_2^1\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1\right) * {}_2a_{Sx}^2 \ , \tag{38}$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_3^1\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1\right) * {}_2a_{Sx}^3 \tag{39}$$

and

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_1^2\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2\right) * {}_2a_{Sx}^1 \ , \tag{40}$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_2^2\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2\right) * {}_2a_{Sx}^2 \ , \tag{41}$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_2w_3^2\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2\right) * {}_2a_{Sx}^3 \ . \tag{42}$$

4.1.2 Representation of the partial derivatives with respect to weights on layer L2 via matrix operations

Can we represent the terms above via a matrix operation in the sense of the Numpy **dot**(*X*, *Y*)-operation? As we remember, such an operation includes a summation. Can we make use of it?

To achieve this we first order all the ${}_3\boldsymbol{\delta}_{Sx}$ within a matrix of the following form:

$${}_3\boldsymbol{\delta}_{\boldsymbol{S}} = \begin{pmatrix} {}_{out}\delta_{S1}^1 * {}_3a_{S1}^1\left(1 - {}_3a_{S1}^1\right), & {}_{out}\delta_{S2}^1 * {}_3a_{S2}^1\left(1 - {}_3a_{S2}^1\right), & \ldots\ldots, & {}_{out}\delta_{Sn}^1 * {}_3a_{Sn}^1\left(1 - {}_3a_{Sn}^1\right) \\ {}_{out}\delta_{S1}^2 * {}_3a_{S1}^2\left(1 - {}_3a_{S1}^2\right), & {}_{out}\delta_{S2}^2 * {}_3a_{S2}^2\left(1 - {}_3a_{S2}^2\right), & \ldots\ldots, & {}_{out}\delta_{Sn}^2 * {}_3a_{Sn}^2\left(1 - {}_3a_{Sn}^2\right) \end{pmatrix} \tag{43}$$

As we know this actually is equal to

$${}_3\boldsymbol{\delta}_{\boldsymbol{S}} = {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} * {}_3\boldsymbol{D}_{\boldsymbol{S}}^{\phi} \tag{44}$$

We transpose the ***A***-matrix for the layer L2

$$ {}_2\boldsymbol{A}_{\boldsymbol{S}} \;=\; \begin{bmatrix} {}_2a^1_{\boldsymbol{S1}},\; {}_2a^1_{\boldsymbol{S2}},\; \ldots.,\; {}_2a^1_{\boldsymbol{Sn}} \\ {}_2a^2_{\boldsymbol{S1}},\; {}_2a^2_{\boldsymbol{S2}},\; \ldots.,\; {}_2a^2_{\boldsymbol{Sn}} \\ {}_2a^3_{\boldsymbol{S1}},\; {}_2a^3_{\boldsymbol{S2}},\; \ldots.,\; {}_2a^3_{\boldsymbol{Sn}} \end{bmatrix} \tag{45} $$

to become

$$ {}_2\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}} \;=\; \begin{bmatrix} {}_2a^1_{\boldsymbol{S1}},\; {}_2a^2_{\boldsymbol{S1}},\; {}_2a^3_{\boldsymbol{S1}} \\ {}_2a^1_{\boldsymbol{S2}},\; {}_2a^2_{\boldsymbol{S2}},\; {}_2a^3_{\boldsymbol{S2}} \\ \ldots\ldots \\ {}_2a^1_{\boldsymbol{Sn}},\; {}_2a^2_{\boldsymbol{Sn}},\; {}_2a^3_{\boldsymbol{Sn}} \end{bmatrix} \tag{46} $$

Note that the column-dimension of $ {}_3\boldsymbol{\delta}_{\boldsymbol{S}} $ is identical to the row-dimension of $ {}_2\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}} $.

We quote from the Numpy documentation on the **dot**(A,B)-operation:

> "If *A* is an N-D array and *B* is an M-D array (where `M>=2`), it is a sum product over the last axis of *A* and the second-to-last axis of *B*: dot(A, B)[i,j,k,m] = sum(A[i,j,:] * B[k,:,m]) "

See https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html

Using this we get the first core result of our analysis:

$$ {}_2\boldsymbol{\nabla}_{\boldsymbol{B_k}} \;=\; {}_3\boldsymbol{\delta}_{\boldsymbol{S}} \;\odot\; {}_2\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}} \tag{47} $$

This really is an astonishingly simple mathematical expression – well suited to be directly coded in Python. Can we extend this result to other layers?

Note that we do not change anything if we formally define

$$ {}_3\boldsymbol{W} \;:=\; {}_3\boldsymbol{W}^{\boldsymbol{T}} \;=\; \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{48} $$

and write

$$ {}_2\boldsymbol{\nabla}_{\boldsymbol{B_k}} \;=\; \left[\left[{}_3\boldsymbol{W}^{\boldsymbol{T}} \odot {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} \right] * \frac{\partial^c \boldsymbol{\phi}\left({}_3\boldsymbol{Z}_{\boldsymbol{S}} \right)}{\partial^c \; {}_3\boldsymbol{z}_{\boldsymbol{S}}} \right] \;\odot\; {}_2\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}} \tag{49} $$

and remember that

$$ {}_3\boldsymbol{\delta}_{\boldsymbol{S}} \;=\; {}_{out}\boldsymbol{\delta}_{\boldsymbol{S}} \;*\; \frac{\partial^c \boldsymbol{\phi}\left({}_3\boldsymbol{Z}_{\boldsymbol{S}} \right)}{\partial^c \; {}_3\boldsymbol{z}_{\boldsymbol{S}}} \quad . \tag{50} $$

4.2 Partial derivatives of the cost function with respect to the weights of an inner layer

To see how the result for partial derivatives on the second to last layer (here L2) can be extended to other middle and hidden layers we now look at the partial derivatives with respect to weights on layer L1.

Also in this case we order the gradient components in a matrix like form – with rows indicating target nodes and columns source nodes connected by the weights. So, our objective is to fill the following matrix:

$$ {}_1\nabla_{B_k} = \begin{bmatrix} \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^1\right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^1\right)} \\ \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^2\right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^2\right)} \\ \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^3\right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^3\right)} \end{bmatrix} \tag{51} $$

4.2.1 Calculating the gradients components for weights of Layer L1

Let us look at a specific node indexed by "*i*" on layer L1 in our drawing. We are now confronted with the fact that there are multiple summands of the cost function which e.g. depend on the weight ${}_1w_i^j$. This is due to the fact that there are *multiple paths* (in our example 2) which lead from nodes on the output layer to the node of ${}_2z^j$.

We now apply the chain rule for the partial derivative with respect to ${}_1w_i^j$:

$$ \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall\, m\ on\ L3} \left[\frac{\partial \lambda\left({}_3a_{Sx}^m\right)}{\partial \left({}_3a_{Sx}^m\right)} * \frac{\partial \left({}_3a_{Sx}^m\right)}{\partial \left({}_3z_{Sx}^m\right)} * \frac{\partial \left({}_3z_{Sx}^m\right)}{\partial \left({}_2a_{Sx}^j\right)} * \frac{\partial \left({}_2a_{Sx}^j\right)}{\partial \left({}_2z_{Sx}^j\right)} * \frac{\partial \left({}_2z_{Sx}^j\right)}{\partial \left({}_1w_i^j\right)} \right] \tag{52} $$

In this case we can ***not*** ignore the sum over all nodes on layer L3. However, we can move the sum over all training samples to the left.

Evaluating the terms one by one we get

$$ \frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_i^j\right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall\, m\ on\ L3} \left[{}_{out}\delta_{Sx}^m * {}_3a_{Sx}^m \left(1 - {}_3a_{Sx}^m\right) * \frac{\partial \left({}_3z_{Sx}^m\right)}{\partial \left({}_2a_{Sx}^j\right)} * \frac{\partial \left({}_2a_{Sx}^j\right)}{\partial \left({}_2z_{Sx}^j\right)} * \frac{\partial \left({}_2z_{Sx}^j\right)}{\partial \left({}_1w_i^j\right)} \right] \tag{53} $$

and eventually

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_i^j \right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall m \text{ on } L3} \left[{}_{out}\delta_{Sx}^m * {}_3a_{Sx}^m \left(1 - {}_3a_{Sx}^m \right) * {}_2w_j^m * {}_2a_{Sx}^j \left(1 - {}_2a_{Sx}^j \right) * {}_1a_{Sx}^i \right] \tag{54}$$

Again, note the varying indices "*m*", "*j*" and "*i*" at the different "*a*"-factors!
"*m*" refers to nodes on layer L3, "*j*" to nodes on layer L2 and "*i*" to the present node on layer L1, at which there are multiple ${}_1w_i^j$ -values defined.

Let us also write down the partial derivative explicitly for ${}_1w_1^2$ to see the term pattern more clearly:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^2 \right)} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall m \text{ on } L3} \left[{}_{out}\delta_{Sx}^m * {}_3a_{Sx}^m \left(1 - {}_3a_{Sx}^m \right) * {}_2w_2^m * {}_2a_{Sx}^2 \left(1 - {}_2a_{Sx}^2 \right) * {}_1a_{Sx}^1 \right] \tag{55}$$

We reorder the summands a bit to get:

$$\begin{aligned} &\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^2 \right)} = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_2^1 * {}_2a_{Sx}^2 \left(1 - {}_2a_{Sx}^2 \right) * {}_1a_{Sx}^1 \right) + \\ &\qquad \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_2^2 * {}_2a_{Sx}^2 \left(1 - {}_2a_{Sx}^2 \right) * {}_1a_{Sx}^1 \right) \end{aligned} \tag{56}$$

Analogously:

$$\begin{aligned} &\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^1 \right)} = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) * {}_1a_{Sx}^1 \right) + \\ &\qquad \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) * {}_1a_{Sx}^1 \right) \end{aligned} \tag{57}$$

and

$$\begin{aligned} &\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^1 \right)} = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) * {}_1a_{Sx}^2 \right) + \\ &\qquad \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) * {}_1a_{Sx}^2 \right) \end{aligned} \text{ , } \tag{58}$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^3 \right)} =$$
$$\frac{1}{n} \sum_{\forall Sx \in B_k} \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_3^1 * {}_2a_{Sx}^3 \left(1 - {}_2a_{Sx}^3 \right) * {}_1a_{Sx}^2 \right) +$$
$$\left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_3^2 * {}_2a_{Sx}^3 \left(1 - {}_2a_{Sx}^3 \right) * {}_1a_{Sx}^2 \right) \tag{59}$$

4.2.2 Transition to a matrix version

To transfer these formulas into a matrix version, we first write down the weight matrices coupling the different layers:

$${}_2\boldsymbol{W} := \begin{bmatrix} {}_2w_1^1 & {}_2w_2^1 & {}_2w_3^1 \\ {}_2w_1^2 & {}_2w_2^2 & {}_2w_3^2 \end{bmatrix}, \qquad {}_2\boldsymbol{W^T} = \begin{bmatrix} {}_2w_1^1 & {}_2w_2^2 \\ {}_2w_2^1 & {}_2w_2^2 \\ {}_2w_3^1 & {}_2w_3^2 \end{bmatrix} \tag{60}$$

and

$${}_1\boldsymbol{W} := \begin{bmatrix} {}_1w_1^1 & {}_1w_2^1 \\ {}_1w_1^2 & {}_1w_2^2 \\ {}_1w_1^3 & {}_1w_2^3 \end{bmatrix}, \qquad {}_1\boldsymbol{W^T} := \begin{bmatrix} {}_1w_1^1, & {}_1w_1^2, & {}_1w_1^3 \\ {}_1w_2^1, & {}_1w_2^2, & {}_1w_2^3 \end{bmatrix}. \tag{61}$$

We remember that in our study case

$${}_3\boldsymbol{\delta_{Sx}} = \begin{bmatrix} {}_{out}\delta_{S1}^1 * {}_3a_{S1}^1 \left(1 - {}_3a_{S1}^1 \right) \\ {}_{out}\delta_{S1}^2 * {}_3a_{S1}^2 \left(1 - {}_3a_{S1}^2 \right) \end{bmatrix} \tag{62}$$

and find that

$$\left[{}_2\boldsymbol{W^T} \odot {}_3\boldsymbol{\delta}_{Sx} \right]$$

is a vector like matrix with 3 rows and one column. It has the following components:

$$\left[{}_2\boldsymbol{W^T} \odot {}_3\boldsymbol{\delta}_{Sx} \right] = \begin{bmatrix} \left[{}_2\boldsymbol{W^T} \odot {}_3\boldsymbol{\delta}_{Sx} \right]_1 \\ \left[{}_2\boldsymbol{W^T} \odot {}_3\boldsymbol{\delta}_{Sx} \right]_2 \\ \left[{}_2\boldsymbol{W^T} \odot {}_3\boldsymbol{\delta}_{Sx} \right]_3 \end{bmatrix} =$$
$$\begin{bmatrix} \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 \right) + \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 \right) \\ \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_2^1 \right) + \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_2^2 \right) \\ \left({}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_3^1 \right) + \left({}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_3^2 \right) \end{bmatrix} \tag{63}$$

From our formulas for the partial derivatives above we understand we need to multiply these components with some a-vector to make the matrix useful.

Analogously to formula (33)we define:

$$ {}_2\boldsymbol{D}^{\phi}_{Sx} \;:=\; {}_2\left[\frac{\partial^c \boldsymbol{\phi}\left(\boldsymbol{Z}_{Sx}\right)}{\partial^c \boldsymbol{z}_{Sx}}\right] \quad . \tag{64} $$

In our example this is given by:

$$ {}_2\boldsymbol{D}^{\phi}_{Sx} \;=\; \begin{bmatrix} {}_2a^1_{Sx}\left(1-{}_2a^1_{Sx}\right) \\ {}_2a^2_{Sx}\left(1-{}_2a^2_{Sx}\right) \\ {}_2a^3_{Sx}\left(1-{}_2a^3_{Sx}\right) \end{bmatrix} \quad . \tag{65} $$

The following vector-matrix will be quite useful:

$$ {}_2\boldsymbol{\delta}_{Sx} \;:=\; \left[\; {}_2\boldsymbol{W}^{\boldsymbol{T}} \;\odot\; {}_3\boldsymbol{\delta}_{Sx} \;\right] * {}_2\boldsymbol{D}^{\boldsymbol{\phi}}_{Sx} \tag{66} $$

In our example the 3 components of this matrix are:

$$ {}_2\boldsymbol{\delta}_{Sx} \;=\; \begin{bmatrix} {}_2\delta^1_{Sx} \\ {}_2\delta^2_{Sx} \\ {}_2\delta^3_{Sx} \end{bmatrix} \;=\; \left[\; {}_2\boldsymbol{W}^{\boldsymbol{T}} \;\odot\; {}_3\boldsymbol{\delta}_{Sx} \;\right] * {}_2[\boldsymbol{a}]^{\boldsymbol{\phi}}_{Sx} \;= $$

$$ \begin{bmatrix} \left({}_{out}\delta^1_{Sx} * {}_3a^1_{Sx}\left(1-{}_3a^1_{Sx}\right) * {}_2w^1_1 * {}_2a^1_{Sx}\left(1-{}_2a^1_{Sx}\right)\right) + \left({}_{out}\delta^2_{Sx} * {}_3a^2_{Sx}\left(1-{}_3a^2_{Sx}\right) * {}_2w^2_1 * {}_2a^1_{Sx}\left(1-{}_2a^1_{Sx}\right)\right) \\ \left({}_{out}\delta^1_{Sx} * {}_3a^1_{Sx}\left(1-{}_3a^1_{Sx}\right) * {}_2w^1_2 * {}_2a^2_{Sx}\left(1-{}_2a^2_{Sx}\right)\right) + \left({}_{out}\delta^2_{Sx} * {}_3a^2_{Sx}\left(1-{}_3a^2_{Sx}\right) * {}_2w^2_2 * {}_2a^2_{Sx}\left(1-{}_2a^2_{Sx}\right)\right) \\ \left({}_{out}\delta^1_{Sx} * {}_3a^1_{Sx}\left(1-{}_3a^1_{Sx}\right) * {}_2w^1_3 * {}_2a^3_{Sx}\left(1-{}_2a^3_{Sx}\right)\right) + \left({}_{out}\delta^2_{Sx} * {}_3a^2_{Sx}\left(1-{}_3a^2_{Sx}\right) * {}_2w^2_3 * {}_2a^3_{Sx}\left(1-{}_2a^3_{Sx}\right)\right) \end{bmatrix} \tag{67} $$

Remember that

$$ {}_1\boldsymbol{A}_{\boldsymbol{S}} \;=\; \begin{bmatrix} {}_1a^1_{\boldsymbol{S1}},\; {}_1a^1_{\boldsymbol{S2}},\; \ldots.,\; {}_1a^1_{\boldsymbol{Sn}} \\ {}_1a^2_{\boldsymbol{S1}},\; {}_1a^2_{\boldsymbol{S2}},\; \ldots.,\; {}_1a^2_{\boldsymbol{Sn}} \end{bmatrix} \quad , \qquad {}_1\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}} \;=\; \begin{bmatrix} {}_1a^1_{\boldsymbol{S1}},\; {}_1a^2_{\boldsymbol{S1}} \\ {}_1a^1_{\boldsymbol{S2}},\; {}_1a^2_{\boldsymbol{S2}} \\ \ldots\ldots \\ {}_1a^1_{\boldsymbol{Sn}},\; {}_1a^2_{\boldsymbol{Sn}} \end{bmatrix} \quad . \tag{68} $$

Now, if we had *just one Sx*, only, then ${}_1\boldsymbol{A}^{\boldsymbol{T}}_{\boldsymbol{S}}$ would have just one row. Remebering the meaning of the $X \odot Y$ operation we could write:

$$ {}_1\nabla_{B_k} = \begin{bmatrix} \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_1^1\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_2^1\right)} \\ \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_1^2\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_2^2\right)} \\ \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_1^3\right)}, & \frac{\partial \Lambda_{B_k}}{\partial\left({}_1w_2^3\right)} \end{bmatrix} = {}_2\boldsymbol{\delta}_{Sx} \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{T} $$

Compare the expressions given for the elements of ${}_1\nabla_{B_k}$ above.

Now we apply the recipe we already used for layer L3, and define

$$ {}_2\boldsymbol{D}_{\boldsymbol{S}}^{\boldsymbol{\phi}} := \begin{bmatrix} {}_2a_{S1}^1\left(1-{}_2a_{S1}^1\right), & {}_2a_{S2}^1\left(1-{}_2a_{S2}^1\right), & \ldots, & {}_2a_{Sn}^1\left(1-{}_2a_{Sn}^1\right) \\ {}_2a_{S1}^2\left(1-{}_2a_{S1}^2\right), & {}_2a_{S2}^2\left(1-{}_2a_{S2}^2\right), & \ldots, & {}_2a_{Sn}^2\left(1-{}_2a_{Sn}^2\right) \\ {}_2a_{S1}^3\left(1-{}_2a_{S1}^3\right), & {}_2a_{S2}^3\left(1-{}_2a_{S2}^3\right), & \ldots, & {}_2a_{Sn}^3\left(1-{}_2a_{Sn}^3\right) \end{bmatrix} . \tag{69} $$

Thus we get:

$$ {}_2\boldsymbol{\delta}_{\boldsymbol{S}} := \begin{bmatrix} {}_2\boldsymbol{\delta}_{S1}^1, & {}_2\boldsymbol{\delta}_{S2}^1, & \ldots, & {}_2\boldsymbol{\delta}_{Sn}^1 \\ {}_2\boldsymbol{\delta}_{S1}^2, & {}_2\boldsymbol{\delta}_{S2}^2, & \ldots., & {}_2\boldsymbol{\delta}_{Sn}^2 \\ {}_2\boldsymbol{\delta}_{S1}^3, & {}_2\boldsymbol{\delta}_{S2}^3, & \ldots, & {}_2\boldsymbol{\delta}_{Sx}^3 \end{bmatrix} = \left[{}_2\boldsymbol{W}^{\boldsymbol{T}} \odot {}_3\boldsymbol{\delta}_{\boldsymbol{S}} \right] * {}_2\boldsymbol{D}_{\boldsymbol{S}}^{\boldsymbol{\phi}} . \tag{70} $$

Our hope is that the summation over *Sx* can be represented by a matrix operation as on layer L2.

Therefore, let us try and write down some components of the matrix $\left[{}_2\boldsymbol{\delta}_{\boldsymbol{S}} \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{\boldsymbol{T}} \right]$ explicitly:

$$ \begin{aligned} &\left[{}_2\boldsymbol{\delta}_{\boldsymbol{S}} \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{T} \right]_1^1 = \left[\left[\left[{}_2\boldsymbol{W}^{\boldsymbol{T}} \odot {}_3\boldsymbol{\delta}_{\boldsymbol{S}} \right] * {}_2\boldsymbol{D}_{\boldsymbol{S}}^{\boldsymbol{\phi}} \right] \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{\boldsymbol{T}} \right]_1^1 = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1\left(1-{}_3a_{Sx}^1\right) * {}_2w_1^1 * {}_2a_{Sx}^1\left(1-{}_2a_{Sx}^1\right) + \right. \\ &\qquad \left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2\left(1-{}_3a_{Sx}^2\right) * {}_2w_1^2 * {}_2a_{Sx}^1\left(1-{}_2a_{Sx}^1\right) \right] * {}_1a_{\boldsymbol{Sx}}^1 \end{aligned} \tag{71} $$

The super-sub-fix combination here indicates the (1,1) component (1st row, 1st column) of the resulting matrix.

Analogously, we get for the (1,2) element (first row, second column) :

$$ \begin{aligned} &\left[{}_2\boldsymbol{\delta}_{\boldsymbol{S}} \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{T} \right]_2^1 = \left[\left[\left[{}_2\boldsymbol{W}_{\boldsymbol{S}}^{\boldsymbol{T}} \odot {}_3\boldsymbol{\delta}_{\boldsymbol{S}} \right] * {}_2\boldsymbol{D}_{\boldsymbol{S}}^{\boldsymbol{\phi}} \right] \odot {}_1\boldsymbol{A}_{\boldsymbol{S}}^{\boldsymbol{T}} \right]_2^1 = \\ &\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1\left(1-{}_3a_{Sx}^1\right) * {}_2w_1^1 * {}_2a_{Sx}^1\left(1-{}_2a_{Sx}^1\right) + \right. \\ &\qquad \left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2\left(1-{}_3a_{Sx}^2\right) * {}_2w_1^2 * {}_2a_{Sx}^1\left(1-{}_2a_{Sx}^1\right) \right] * {}_1a_{\boldsymbol{Sx}}^2 \end{aligned} \tag{72} $$

And for the (3,2) element (third row, second column):

$$\left[{}_2\boldsymbol{\delta}_S \odot {}_1\boldsymbol{A}_S^T \right]_2^1 = \left[\left[\left[{}_2^3\boldsymbol{W}_S^T \odot {}_3\boldsymbol{\delta}_S \right] * {}_2\boldsymbol{D}_S^{\boldsymbol{\phi}} \right] \odot {}_1\boldsymbol{A}_S^T \right]_2^1 =$$
$$\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) + \right. \tag{73}$$
$$\left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) \right] * {}_1a_{\mathbf{Sx}}^2$$

Compare this with a little rearranged presentation of our gradient components:

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_1^1 \right)} =$$
$$\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) + \right. \quad , \tag{74}$$
$$\left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) \right] * {}_1a_{Sx}^1$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^1 \right)} =$$
$$\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_1^1 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) + \right. \quad , \tag{75}$$
$$\left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_1^2 * {}_2a_{Sx}^1 \left(1 - {}_2a_{Sx}^1 \right) \right] * {}_1a_{Sx}^2$$

$$\frac{\partial \Lambda_{B_k}}{\partial \left({}_1w_2^3 \right)} =$$
$$\frac{1}{n} \sum_{\forall Sx \in B_k} \left[{}_{out}\delta_{Sx}^1 * {}_3a_{Sx}^1 \left(1 - {}_3a_{Sx}^1 \right) * {}_2w_3^1 * {}_2a_{Sx}^3 \left(1 - {}_2a_{Sx}^3 \right) + \right. \quad . \tag{76}$$
$$\left. {}_{out}\delta_{Sx}^2 * {}_3a_{Sx}^2 \left(1 - {}_3a_{Sx}^2 \right) * {}_2w_3^2 * {}_2a_{Sx}^3 \left(1 - {}_2a_{Sx}^3 \right) \right] * {}_1a_{Sx}^2$$

Heureka! We can indeed transfer the results for layer L2 to layer L1 and get our second core result:

$${}_1\nabla_{B_k} = \left[{}_2\boldsymbol{\delta}_S \odot {}_1\boldsymbol{A}_S^T \right] \tag{77}$$

5 Generalization of the required steps for EBP

After this tour de force through derivative evaluation and sample related matrix operations for our study case MLP, we can generalize and summarize the steps we identified. What is it that we have to "*generalize*" in comparison to the example discussed above?

Well, the careful reader has noticed that we used an *explicit* number of nodes on each layer. We give up this restriction now. We also need to generalize with respect to the number of layers.

5.1 Step 0

The outermost layer on the left side of an MLP may have index "*E*". On a layer L(N) we may find $m_N \geq 1$ nodes.

We prepare the following matrices for all layers L(N), with N in [1, E-1] :

$$ {}_N\boldsymbol{W} := \begin{bmatrix} {}_N w_1^1, & {}_N w_2^1, & \ldots., & {}_N w_{m_N}^1 \\ {}_N w_1^2, & {}_N w_2^2, & \ldots., & {}_N w_{m_N}^2 \\ \ldots\ldots\ldots., & \ldots\ldots\ldots., & \ldots., & \ldots\ldots\ldots \\ {}_N w_1^{m_{N+1}}, & {}_N w_2^{m_{N+1}}, & \ldots., & {}_N w_{m_N}^{m_{N+1}} \end{bmatrix} \tag{78} $$

and with N in [0, E-1]

$$ {}_N\boldsymbol{A_S} = \begin{bmatrix} {}_N a_{S1}^1, & {}_N a_{S2}^1, & \ldots., & {}_N a_{Sn}^1 \\ {}_N a_{S1}^2, & {}_N a_{S2}^2, & \ldots., & {}_N a_{Sn}^2 \\ \ldots., & \ldots., & \ldots., & \ldots. \\ {}_N a_{S1}^{m_N}, & {}_N a_{S2}^{m_N}, & \ldots., & {}_N a_{Sn}^{m_N} \end{bmatrix} . \tag{79} $$

We then transpose these matrices to ${}_N\boldsymbol{W^T}$ and ${}_N\boldsymbol{A_S^T}$. As this is a standard operation we leave the details to the reader.

Furthermore we create the following matrices:

$$ {}_N\boldsymbol{D_S^{\phi}} := \frac{\partial^c \boldsymbol{\phi}\left({}_N\boldsymbol{Z_S}\right)}{\partial^c {}_N\boldsymbol{z_S}} = $$

$$ {}_N\left[\frac{\partial^c \boldsymbol{\phi}\left(\boldsymbol{Z_S}\right)}{\partial^c \boldsymbol{z_S}}\right] := \begin{bmatrix} \frac{\partial \phi\left({}_N z_{S1}^1\right)}{\partial {}_N z_{S1}^1}, & \frac{\partial \phi\left({}_N z_{S2}^1\right)}{\partial {}_N z_{S2}^1}, & \ldots., & \frac{\partial \phi\left({}_N z_{Sn}^1\right)}{\partial {}_N z_{Sn}^1} \\ \frac{\partial \phi\left({}_N z_{S1}^2\right)}{\partial {}_N z_{S1}^2}, & \frac{\partial \phi\left({}_N z_{S2}^2\right)}{\partial {}_N z_{S2}^2}, & \ldots., & \frac{\partial \phi\left({}_N z_{Sn}^2\right)}{\partial {}_N z_{Sn}^2} \\ \ldots\ldots\ldots, & \ldots\ldots\ldots, & \ldots., & \ldots\ldots\ldots \\ \frac{\partial \phi\left({}_N z_{S1}^{m_N}\right)}{\partial {}_N z_{S1}^{m_N}}, & \frac{\partial \phi\left({}_N z_{S2}^{m_N}\right)}{\partial {}_N z_{S2}^{m_N}}, & \ldots., & \frac{\partial \phi\left({}_N z_{Sn}^{m_N}\right)}{\partial {}_N z_{Sn}^{m_N}} \end{bmatrix} \tag{80} $$

In our example a component of such a matrix for layer L(N) is given by

$$ {}_N D_{j,\,Sx}^{\phi} := \phi\left({}_N z_{Sx}^j\right) * \left(1 - \phi\left({}_N z_{Sx}^j\right)\right) $$

Our objective is to determine the components of ${}_{N}\nabla_{B_k}$ with N in [0, E-1]

$$ {}_{N}\nabla_{B_k} = \begin{bmatrix} \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_1^1 \right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_2^1 \right)}, & \ldots\ldots, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_{m_N}^1 \right)} \\ \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_1^2 \right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_2^2 \right)}, & \ldots\ldots, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_{m_N}^2 \right)} \\ \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_1^{m_{N+1}} \right)}, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_2^{m_{N+1}} \right)}, & \ldots\ldots, & \frac{\partial \Lambda_{B_k}}{\partial \left({}_{N}w_{m_N}^{m_{N+1}} \right)} \end{bmatrix} \tag{81} $$

by a sequence of generalized steps.

5.2 Step 1

In step 1 we take care of the outermost layer to the left:

We define an "***error***"-value at each node "*j*" of the left outermost layer "LE" for each of the training samples "*Sx*". y_{Sx}^{j} is the known correct value for this sample at the "*j*"-node.

$$ {}_{E}\theta_{Sx}^{j} = \left({}_{3}a_{Sx}^{j} - y_{Sx}^{j} \right) \tag{82} $$

We define the cost function to be a sum of node-wise contributions of an error at the outermost layer to the left.

$$ \Lambda_{B_k} = \frac{1}{n} \sum_{\forall Sx \in B_k} \sum_{\forall j \text{ on LE}} \lambda\left({}_{E}a_{Sx}^{j}, y_{Sx}^{j} \right) \tag{83} $$

We take a suitable form of λ, such that the derivative results in a function which depends on the error:

$$ {}_{out}\delta_{Sx}^{j} = \frac{\partial \lambda\left({}_{E}a_{Sx}^{j} \right)}{\partial \left({}_{E}a_{Sx}^{j} \right)} = \Psi\left(\left(y_{Sx}^{j} - {}_{E}a_{Sx}^{j} \right) \right) = \Psi\left({}_{E}\theta_{Sx}^{j} \right) \ . \tag{84} $$

Then we evaluate all D-components

$$ {}_{E}D_{j,\,Sx}^{\phi} = \frac{\partial \phi\left({}_{E}z_{Sx}^{j} \right)}{\partial\, {}_{E}z_{Sx}^{j}} \tag{85} $$

and define

$$ {}_{E}\delta_{Sx}^{j} := {}_{out}\delta_{Sx}^{j} * {}_{E}D_{j,\,Sx}^{\phi} \tag{86} $$

Note: In our study example it holds that

$$ {}_{E}D_{j,\,Sx}^{\phi} = \phi\left({}_{E}z_{Sx}^{j}\right) * \left(1-\phi\left({}_{E}z_{Sx}^{j}\right)\right) $$

and

$$ {}_{out}\delta_{Sx}^{j} = \Psi\left({}_{E}\theta_{Sx}^{j}\right) = {}_{E}\theta_{Sx}^{j} * \left[\,{}_{E}D_{j,\,Sx}^{\phi}\,\right]^{-1} \quad . $$

So,

$$ {}_{\boldsymbol{E}}\delta_{Sx}^{j} = {}_{E}\theta_{Sx}^{j} = \left({}_{3}a_{Sx}^{j} - y_{Sx}^{j}\right) \ , $$

due to a very special form of the cost function and the activation function.

We now account for the bunch of training samples called mini-batch.

Assuming that we have “n” samples and $m_{\boldsymbol{E}}$ nodes on layer “L_E” we build the following matrices:

$$ {}_{\boldsymbol{out}}\boldsymbol{\delta_S} = \begin{bmatrix} {}_{out}\delta_{S1}^{1}, & {}_{out}\delta_{S2}^{1}, & \ldots\ldots, & {}_{out}\delta_{Sn}^{1} \\ {}_{out}\delta_{S1}^{'2}, & {}_{out}\delta_{S2}^{2}, & \ldots\ldots, & {}_{out}\delta_{Sn}^{2} \\ & \ldots\ldots & & \\ {}_{out}\delta_{S1}^{m_E}, & {}_{out}\delta_{S2}^{m_E}, & \ldots\ldots, & {}_{out}\delta_{Sn}^{m_E} \end{bmatrix} \tag{87} $$

and

$$ {}_{\boldsymbol{E}}\boldsymbol{\delta_S} = \begin{bmatrix} {}_{E}\delta_{S1}^{1}, & {}_{E}\delta_{S2}^{1}, & \ldots\ldots, & {}_{E}\delta_{Sn}^{1} \\ {}_{E}\delta_{S1}^{'2}, & {}_{E}\delta_{S2}^{2}, & \ldots\ldots, & {}_{E}\delta_{Sn}^{2} \\ & \ldots\ldots & & \\ {}_{E}\delta_{S1}^{m_E}, & {}_{E}\delta_{S2}^{m_E}, & \ldots\ldots, & {}_{E}\delta_{Sn}^{m_E} \end{bmatrix} = {}_{\boldsymbol{out}}\boldsymbol{\delta_S} * {}_{\boldsymbol{E}}\boldsymbol{D_S^{\phi}} \tag{88} $$

We then calculate the matrix of the first partial derivatives with respect to the weights on layer L_{E-1}:

$$ {}_{\boldsymbol{E-1}}\boldsymbol{\nabla}_{\boldsymbol{B_k}} = {}_{\boldsymbol{E}}\boldsymbol{\delta_S} \odot {}_{\boldsymbol{E-1}}\boldsymbol{A_S^T} \tag{89} $$

5.3 Step 2

We take the matrix of weights ${}_{\boldsymbol{E-1}}\boldsymbol{W}$ connecting the $m_{\boldsymbol{z-1}}$ nodes on layer “L_(z-1)”

and transpose it to ${}_{\boldsymbol{E-1}}\boldsymbol{W^T}$.

Then we need to calculate

$$ {}_{E-1}\boldsymbol{\delta}_S \;:=\; \left[\; {}_{E-1}\boldsymbol{W}^T \odot {}_{E}\boldsymbol{\delta}_S \;\right] * {}_{E-1}\boldsymbol{D}_S^{\phi} \quad . \tag{90} $$

We know that the components of ${}_{E-1}\boldsymbol{D}_S^{\phi}$ may take a simple form in case of the “sigmoid”-function for ϕ ; however, we leave it in the given form to account for more general activation functions.

Eventually, we get:

$$ {}_{E-2}\nabla_{B_k} \;=\; {}_{E-1}\boldsymbol{\delta}_S \;\odot\; {}_{E-2}\boldsymbol{A}_S^T \tag{91} $$

5.4 General step N

Following the steps of our study case we now calculate in the general case

$$ {}_{N}\boldsymbol{\delta}_S \;:=\; \left[\; {}_{N}\boldsymbol{W}^T \odot {}_{N+1}\boldsymbol{\delta}_S \;\right] * {}_{N}\boldsymbol{D}_S^{\phi} \quad . \tag{92} $$

and derive

$$ {}_{N-1}\nabla_{B_k} \;=\; {}_{N}\boldsymbol{\delta}_S \;\odot\; {}_{N-1}\boldsymbol{A}_S^T \tag{93} $$

This is a fairly simple result, which we have found for the required gradients of the cost function with respect to the weights on the layers of the MLP.

Note, that the actual weight correction for a weight ${}_{N}w_i^j$ is typically calculated by multiplying the corresponding gradient component with a small factor ϵ (called step-size):

$$ \Delta\left({}_{N}w_i^j\right) \;=\; \epsilon * \frac{\partial \Lambda_{B_k}}{\partial\left({}_{N}w_i^j\right)} \tag{94} $$

5.5 Presuppositions?

Although we have already generalized quite a lot, let us think a bit about the assumptions and preconditions which made the whole process work.

Supposition 1: The whole step sequence is based on the assumption that the cost function uses information at the outermost layer and its nodes, only. This is a rather severe restriction. If we had special contributions to the cost function from nodes on inner layers we would face at least one consequence which would break our simple algorithm:

We could not just back-propagate ${}_{N}\boldsymbol{\delta}_{S}$ terms between layers. We would have to add node-specific ${}_{N}\lambda^{j}_{Sx}$ -values at every layer and differentiate with respect to local ${}_{N}a^{j}_{Sx}$ values before we propagate the resulting terms in addition.

Supposition 2: Regarding the cost functions we should be aware of the following aspects:

- In our study example we employed a cost function at each node whose derivative contained the error ${}_{\boldsymbol{E}}\theta^{j}_{Sx}$ as a separable factor.
- In our special case we could even cancel the derivative of the activation function when calculating the various ${}_{\boldsymbol{E}}\delta^{j}_{Sx}$.

The first point restricts the form of the cost function. Another cost function where this point holds is an expression proportional or otherwise dependent on $\left({}_{\boldsymbol{E}}\theta^{j}_{Sx}\right)^{2}$. Such an approach is typical for MSE-dependent cost functions.

Supposition 3: Do not forget that we assumed the output function at the outer layer to be equal to the general activation function

$${}_{out}\phi\left({}_{E}z^{j}_{Sx}\right) = \phi\left({}_{E}z^{j}_{Sx}\right) \tag{95}$$

used elsewhere in the grid. This may not be the case in more general MLPs – and then the derivative

$${}_{\boldsymbol{E}}D^{\phi}_{j,\,Sx} := \frac{\partial_{out}\phi\left({}_{E}z^{j}_{Sx}\right)}{\partial_{E}z^{j}_{Sx}} \tag{96}$$

will have to be analyzed separately.

In my blog I shall sooner or later discuss cases which break the standard EBP due to cost functions whose contributions stem from contributions of clusters of nodes distributed over multiple inner layers. Have fun with ML and stay tuned …. .

6 Literature

[1] *"Python Machine Learning"*, Sebastian Rashka, 2016, Packt Publishing Limited, Birmingham, UK

[2] *"Machine Learning with SciKit-Learn and Tensor-Flow: Concepts, Tools , and Techniques to Build Intelligent Systems"*, Aurélien Géron, 2017, O'Reilly Media Inc.
I use the German version: "*Praxiseinstieg Machine Learning mit SciKit-Learn & TensorFlow*", Aurélien Géron, 2018, dpunkt.verlag GmbH, Heidelberg, Germany

[3] *"Neuronale Netze selbst programmieren"*, Tariq Rashid, 2017, dpunkt.verlag GmbH, Heidelberg, Germany

[4] *"Neural Networks – A Visual Introdution For Beginners"*, Michael Taylor, 2017, Blue Windmill Media, Canada

[5] *"Deep Learning mit TensorFlow, Keras und TensorFlow.js"*, Matthieu Deru, Alassane Ndiaye, 2019, Rheinwerk Verlag, Bonn, Germany