Neural Networks from Scratch

Author	Mat-Nr.	University	Study Program	Course	Mentor
Kevin Südmersen	89966	Hochschule Albstadt-Sigmardingen	MSc Data Science	40200 Practical Work (Seminararbeit)	Prof. Dr. Knoblauch

Acknowledgments

The knowledge used in this repository stems from a variety of sources. The below is probably an incomplete list of helpful sources, but it is definitely a good start.

- Machine Learning course of Hochschule Albstadt-Sigmardingen (Prof. Dr. Knoblauch)
- Andrew Ng's Deep Learning Courses on Coursera
- Micheal Nielsons online Deep Learning book
- 3Blue1Brown

Table of contents

Introduction

This document aims to derive and implement equations for training Multi Layer Perceptrons (MLPs), i.e. Feed Forward Neural Networks consisting of a stack of dense layers, from scratch. These neural networks will consist of an arbitrary number of layers, each with an arbitrary number of neurons and arbitrary choice of activation function. At the moment, the following building blocks are supported, but may easily be extended:

- Dense layers with one of the following activation functions:
  - Sigmoid
  - Tanh
  - Linear
  - ReLU
  - Softmax
- Categorical Cross-Entropy losses

- Image data generators
- Stochastic Gradient Descent optimizers
- Following evaluation metrics:
    - Categorical Cross-Entropy
    - Accuracy (micro-averaged)
    - Precision (micro-averaged)
    - Recall (micro-averaged)

First, we will derive equations for the forward- as well as backward propagation algorithms in scalar form for a single training example. Then, we will extend these equations to a *matrix-based* approach for a single training example, and finally, we will extend them to a matrix-based approach for a batch of training examples.

Forward Propagation

The forward propagation algorithm propagates inputs through the layers of the network until they reach the output layer which generates the predictions. After all or a batch of input examples have been propagated through the network, the quality of these predictions are evaluated by calculating a certain cost function.

Forward Propagation for a Single Training Example

Suppose we wanted to decide whether or not to go to sports today and suppose that we had three types of information, i.e. *input features*, that can aid us making that decision: The weather temperature (in degree Celsius), whether or not we slept well last night (yes or no), and whether or not we have a lot of homework to do (yes or no). To answer the question whether we should go to sports tonight, we might construct a simple neural network consisting of an input layer, one hidden layer and an output layer that might look like this:

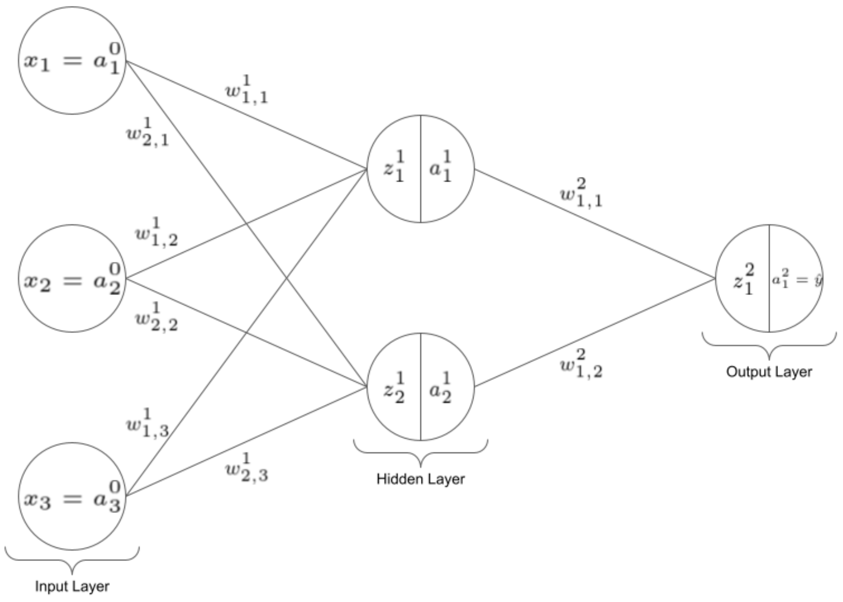

Figure 1: Example neural network

The input layer (layer index 0) consists of 3 neurons, the hidden layer (layer index 1) consists of 2 neurons and the output layer (layer index 2) consists of 1 neuron. The input layer has one neuron per input feature x_i which we will sometimes also refer to as the *activations* of the input layer, so that we may sometimes write $x_i = a_i^0$ for $i = 1, 2, 3$. Later on, this notation will allow us to represent the activations in layer l in terms of the activations in layer $l-1$ for all layers $l = 0, 1, \dots, L$.

The hidden layer consists of 2 neurons which are supposed to represent more complex, latent (not directly observable) features or combinations of features that the network *learns* by itself so that it can make better decisions whether or not we should go to sports today. For example, if we slept well last night and we have little homework to do, we might be in a very good *mood* today and we want to go to sports. So, some neuron in the hidden layer might be some sort of mood indicator (good or bad).

Each of these hidden neurons has a *dendritic potential* z_i^1 as input and a corresponding *firing rate*, i.e. activation, a_i^1 as output for $i = 1, 2$. For example,

$$z_1^1 = a_1^0 w_{1,1}^1 + a_2^0 w_{1,2}^1 + a_3^0 w_{1,3}^1 + b_1^1 \tag{1}$$

and

$$a_1^1 = \mathbf{f}(z_1^1, z_2^1, z_3^1)_1 \tag{2}$$

or more generally,

$$z_i^l = \sum_{k=1}^{n^{l-1}} \left(a_k^{l-1} w_{i,k}^l \right) + b_i^l \tag{3}$$

$$a_i^l = \mathbf{f}(z_1^l, z_2^l, \dots, z_i^l, \dots, z_{n^{l-1}}^l)_i = \mathbf{f}(\mathbf{z}^l)_i, \tag{4}$$

where n^{l-1} represents the number of neurons in layer $l-1$, $w_{i,k}^{l}$ the weight that connects a_k^{l-1} to a_i^l, and where b_i^l represents a bias term. Note that the weights and biases are initialized with random values and represent the parameters of the network, i.e. the parameters which the network *learns* and updates during the training phase.

$\mathbf{f}(\mathbf{z}^l)_i$ represents the i-th output element of the *activation function* $\mathbf{f}(\cdot)$ that is applied to the weighted inputs in layer l. Note that, in the most general case, $\mathbf{f}(\cdot)$ is a function that takes a vector as input and also outputs a vector. For now however, we will assume that the activation function is simply the scalar-valued sigmoid function, which is defined as follows:

$$f(z_i^l) = \frac{1}{1+e^{-z_i^l}}, \tag{5}$$

which has the desirable property that $0 < f(z_i^l) < 1$, so we can say that neuron a_i^l is firing if $f(z_i^l)$ is close to 1.

Then, in the output layer of our example network, we simply have one neuron that represents the probability whether or not we should go to sports, i.e.

$$a_1^2 = \hat{y}_i \tag{6}$$

or more generally,

$$a_i^L = \hat{y}_i \tag{7}$$

where L represents the final layer of the network and $\hat{y}_i$ the *probability* that we go to sports.

In our example network, there is no benefit for adding the neuron index i, but we still left it there to show that the output layer might consists of an arbitrary number of neurons, e.g. one for each category in our classification task. Also, since $\hat{y}_i$ is a probability, we know that the activation function of the output layer must return values between 0 and 1. To convert the predicted probability that we will go to sports into an actual decision, we will apply a threshold as follows

$$\text{prediction}_i = \begin{cases} 1, & \hat{y}_i > \text{threshold}_i \\ 0, & \hat{y}_i \leq \text{threshold}_i \end{cases}, \tag{8}$$

where 1 means that we will go to sports and 0 that we won't go to sports.

The threshold for category i may be chosen manually and fine tuned on a validation set, but for now, we will assume that $\text{threshold} = 0.5$. If you decide to increase the threshold, your model is likely to achieve a higher precision at the expense of recall and if you decide to decrease the threshold, your model is likely to achieve a higher recall at the expense of precision. Precision and recall will be defined more thoroughly later on.

Now, we want to introduce a matrix-based approach for forward propagating the input data to the output layer, because first, it will make the notation easier and second, it will make your code run faster when you actually need to implement it in Python, because vectorized operations are highly efficient and optimized. So first, we will rewrite equation (3) as

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{9}$$

or written out explicitly with all components

$$\begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_{n^l}^l \end{bmatrix} = \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \dots & w_{n^l,n^{l-1}}^l \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ \vdots \\ a_{n^{l-1}}^{l-1} \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_{n^l}^l \end{bmatrix} \tag{10}$$

and then, equation (4) can be rewritten as

$$\mathbf{a}^l = \mathbf{f}(\mathbf{z}^l), \tag{11}$$

or written out explicitly:

$$\begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_{n^l}^l \end{bmatrix} = \begin{bmatrix} \mathbf{f}(\mathbf{z}^l)_1 \\ \mathbf{f}(\mathbf{z}^l)_2 \\ \vdots \\ \mathbf{f}(\mathbf{z}^l)_{n^l} \end{bmatrix}. \tag{12}$$

So, we just stacked the weighted inputs, the activations and the biases of each layer into column vectors $\mathbf{z}^l$, $\mathbf{a}^{l-1}$, and $\mathbf{b}^l$. For each neuron in layer l, the weight matrix $\mathbf{W}^l$ contains one row and for each neuron in layer $l-1$, it contains one column, meaning that the dimensions of $\mathbf{W}^l$ are $n^l \times n^{l-1}$. Then finally, activation function $\mathbf{f}(\cdot)$ is just applied to each element of $\mathbf{z}^l$ to produce the activation vector $\mathbf{a}^l$.

We can now apply equations (9) and (11) recursively all the way to the output layer L, until we compute the predicted probabilities of the network as follows

$$\mathbf{a}^L = \hat{\mathbf{y}}, \tag{13}$$

or written out explicitly

$$\begin{bmatrix} a_1^L \\ a_2^L \\ \vdots \\ a_{n^L}^L \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_{n^L} \end{bmatrix}, \tag{14}$$

where each $\hat{y}_i$ is converted into an actual decision using (8).

Having computed $\mathbf{a}^L$, we can compute a certain *loss* which indicates how well or badly our model predicts for a *single* training example. For classification problems where *exactly* one of n^L classes must be predicted (i.e. a multi-class classification problem), a commonly used loss function is the *categorical cross entropy*, which is defined as follows

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -||\mathbf{y}\ log(\hat{\mathbf{y}})||^2 = -\sum_{i=1}^{n^L} y_i\ log(\hat{y}_i), \tag{15}$$

where $\mathbf{y}$ is the ground truth vector (containing the target values) where element $y_i = 1$ and all other elements are zero if the current training example belongs to class i. We may also say that $\mathbf{y}$ is *one-hot-encoded*.

In general, we want a loss function which has high values for bad predictions, i.e. when $\hat{y}_i$ is far away from y_i, and low values for good predictions, i.e. when $\hat{y}_i$ is very close to y_i. Let's see if component i of (15) fulfills these requirements by considering the following example of a multi-class classificaton problem with 2 classes:

- Bad predictions
  - If $\mathbf{y} = [1, 0]^T$ and $\hat{\mathbf{y}} = [0, 1]^T$, then $L(y_i, \hat{y}_i) = -(1 \times log(0) + 0 \times log(1)) = -(-\infty + 0) = \infty$
  - If $\mathbf{y} = [0, 1]^T$ and $\hat{\mathbf{y}} = [1, 0]^T$, then $L(y_i, \hat{y}_i) = -(0 \times log(1) + 1 \times log(0)) = -(0 - \infty) = \infty$
- Good predictions
  - If $\mathbf{y} = [1, 1]^T$ and $\hat{\mathbf{y}} = [1, 1]^T$, then $L(y_i, \hat{y}_i) = -(1 \times log(1) + 1 \times log(1)) = -(0 + 0) = 0$
  - If $\mathbf{y} = [0, 0]^T$ and $\hat{\mathbf{y}} = [0, 0]^T$, then $L(y_i, \hat{y}_i) = -(0 \times log(0) + 0 \times log(0)) = -(0 + 0) = 0$

in all of the 4 above cases, we get the desired result.

Forward Propagation for a Batch of Training Examples

Assuming that we have M training examples in our current batch and n^0 input features, imagine a 3 dimensional (3D) matrix $\mathbf{X} = \mathbf{A}^0$, where each element in the depth dimension belongs to a different training example:

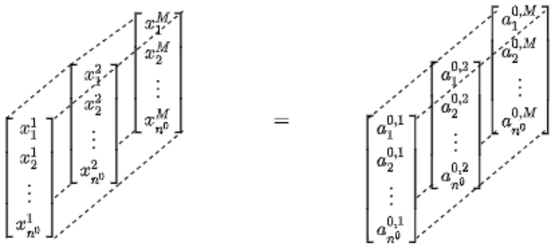

Figure 2

where $x_i^m = a_i^{l,m}$ represents the i-th activation in layer l of the m-th training example.

Next, equation (9) becomes

$$\mathbf{Z}^l = \mathbf{W}^l \mathbf{A}^{l-1} + \mathbf{B}^l, \tag{16}$$

or written out explicitly

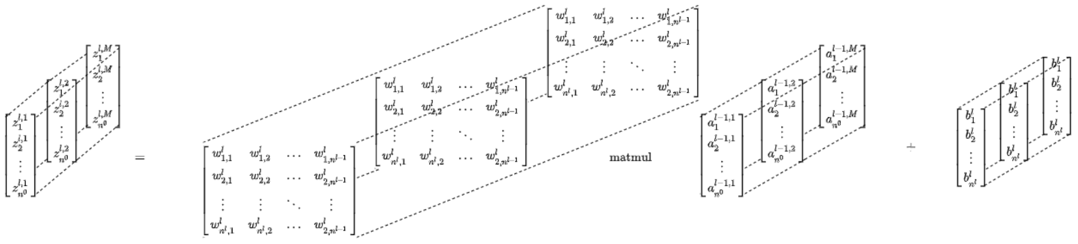

Figure 4

where the weight matrix $\mathbf{W}^l$ and the bias vector $\mathbf{B}^l$ have been broad-casted M times along the depth dimension in order to make the whole operation compatible.

Note that when implementing the forward propagation for a whole batch of training examples at once in NumPy, the batch-dimension must be placed at the first axis of any 3D array (`axis=0`), i.e. every training example must be placed in a separate row. However, in the figures above and the below figures to come, we placed each training example in the batch dimension, which is normally at `axis=2`. We chose to draw each training example in the depth dimension, because it is simply easier to draw that way. For the actual implementation in NumPy though, it's important to place each training example in a different row.

In the implementation with NumPy, the dimensions of each component of figure 4 are as follows:

- $\mathbf{Z}^l : M \times n^l \times 1$
- $\mathbf{W}^l : M \times n^l \times n^{l-1}$
- $\mathbf{A}^{l-1} : M \times n^{l-1} \times 1$
- $\mathbf{W}^l \mathbf{A}^{l-1} : M \times n^l \times 1$. Note here, that each of the M matrix multiplications is done independently and in parallel
- $\mathbf{B}^l : M \times n^l \times 1$

Then, like before, the activation function is applied independently to each training example

$$\mathbf{A}^l = \mathbf{f}(\mathbf{Z}^l), \tag{17}$$

which can be written out to

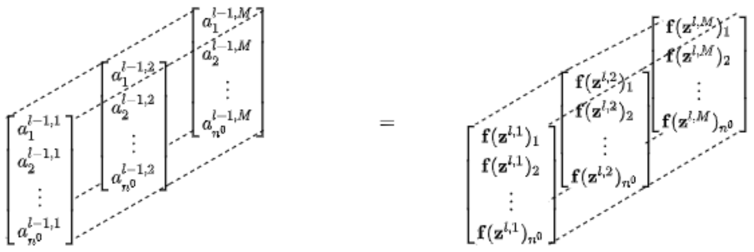
Figure 5

Like before, equations (19) and (21) are applied recursively to layer L, until we can compute all `batch_size` losses at once, yielding the following result

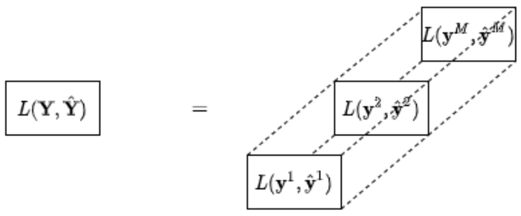
Figure 6

where each element of the above 3D loss array represents the loss we have already defined in equation (15), i.e.

$$L(\mathbf{y}^m, \hat{\mathbf{y}}^m) = -\sum_{i=1}^{n^L} y_i^m log(\hat{y}_i^m). \tag{18}$$

Having computed the loss array $L(\mathbf{Y}, \hat{\mathbf{Y}})$, we can now aggregate over all M training examples to compute a certain *cost*, which is just the average over all training examples in the current batch

$$C = \frac{1}{M}\sum_{m=1}^{M} L(\mathbf{y}^m, \hat{\mathbf{y}}^m) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{i=1}^{n^L} y_i^m log(\hat{y}_i^m), \tag{19}$$

Note that the loss function represents an error over a *single* training example, while the cost function is an aggregation of the loss over M training examples. When computing the cost for M training examples, it makes sense to choose the average as an aggregation method, because the average cost doesn't increase linearly with the `batch_size` (like e.g. the sum). Also, the cost function may include a regularization term, which should be monotonically increasing in the number of parameters of the model, to penalize models with lots of free parameters, leading to simpler models, and hence to fight over-fitting.

Backward Propagation

Neural networks learn by iteratively adjusting their weights and biases such that the cost decreases, i.e. such that the predictions become more accurate. This goal is achieved by (1) computing all partial derivatives of the cost w.r.t. the weights and biases in the network (the *gradient*) and (2) by updating the weights and biases using *gradient descent*. This section will describe how to calculate, the gradient using the backpropagation algorithm which is a very cost efficient algorithm. First, the backpropagation algorithm is explained for a single training example and then extended to a whole batch of training examples.

The backpropagation algorithm generally works as follows. For any given layer l, the backpropagation algorithm computes an intermediate quantity, the so called *error* at layer l, and then computes the gradients of the weights and biases in layer l using that error. Then, the error is propagated one layer backwards and the gradients are computed again. This process is repeated recursively until the gradients of the weights and biases in layer 1 (layer with index 1) are computed.

The backpropagation algorithm is based on 4 key equations which we will derive in detail in the following sections. The four key equations are as follows:

- BP1.x: An equation for the error at the output layer, needed for initializing the backpropagation algorithm
  - When considering a single training example, we will refer to this equation as $\boldsymbol{\delta}^L$ or BP1.1
  - When considering `batch_size` training examples, we will refer to this equation as $\boldsymbol{\Delta}^L$ or BP1.2.
- BP2.x: A recursive equation relating the error at layer $l+1$ to the error at layer l, needed for recursively calculating the error at each layer.
  - When considering a single training example, we will refer to this equation as $\boldsymbol{\delta}^l$ or BP2.1
  - When considering `batch_size` training examples, we will refer to this equation as $\boldsymbol{\Delta}^l$ or BP2.2.
  - Note that in the first iteration, we must set $\boldsymbol{\delta}^l = \boldsymbol{\delta}^L$ or $\boldsymbol{\Delta}^l = \boldsymbol{\Delta}^L$ which we already computed in BP1.1 and BP1.2 respectively. After that, we can recursively substitute the error all the way back to the input layer.
- BP3.x: An equation relating the error at layer l to:
  - The derivative of the *loss* function w.r.t the weights in layer l when considering a *single* training example, i.e. $\frac{\partial L}{\partial \mathbf{W}^l}$. We'll refer to this equation as BP3.1
  - The derivative of the *cost* function w.r.t. the weights in layer l when considering a *batch* of training examples, i.e. $\frac{\partial C}{\partial \mathbf{W}^l}$ .We'll refer to this equation as BP3.2
- BP4.x: An equation relating the error at layer l to:
  - The derivative of the *loss* function w.r.t the biases in layer l when considering a *single* training example, i.e. $\frac{\partial L}{\partial \mathbf{b}^l}$. We'll refer to this equation as BP4.1
  - The derivative of the *cost* function w.r.t. the biases in layer l when considering a *batch* of training examples, i.e. $\frac{\partial C}{\partial \mathbf{b}^l}$.We'll refer to this equation as BP4.2

Most of the work will go into deriving equations BP1.1-BP4.1 and applying these equations to `batch_size` training examples at once is just a little overhead but will save a lot of time when running the actual code.

Why Backpropagation

You might wonder why we should bother trying to derive a complicated algorithm and not use other seemingly simpler methods for computing all partial derivatives in the network. To motivate the need for the backpropagation algorithm, assume we simply wanted to compute the partial derivative of weight w_j as follows [1]

$$\frac{\partial L}{\partial w_j} = \lim_{\epsilon \to 0} \left(\frac{L(\mathbf{w} + \epsilon \mathbf{e}_j, \mathbf{b}) - L(\mathbf{w}, \mathbf{b})}{\epsilon} \right), \tag{20}$$

where $\mathbf{w}$ and $\mathbf{b}$ are flattened vectors containing all weights and biases of the network, where ϵ is a infinitesimal scalar and where $\mathbf{e}_j$ is the unit vector being 1 at position j and 0 elsewhere. Assuming that our network has one million parameters, we would need to calculate $L(\mathbf{w} + \epsilon \mathbf{e}_j, \mathbf{b})$ a million times (once for each *j*), and also, we would need to calculate $L(\mathbf{w}, \mathbf{b})$ once, summing up to a total of $1,000,001$ forward passes for just a *single* training example! As we will see in this section, the backpropagation algorithm let's us compute all partial derivatives of the network with just *one* forward- and one backward pass through the network, so the backpropagation algorithm is a very cost efficient way of computing the gradients.

Backpropagation for a Single Training Example

BP1.1

In order to conveniently represent the error at any layer l, will introduce the following notation

$$(\boldsymbol{\delta}^l)^T := \frac{\partial L}{\partial \mathbf{z}^l}. \tag{21}$$

Notice that the transposition T must be used, because the derivative of a scalar (L) w.r.t a vector ($\mathbf{z}^l$) is defined as a row vector [2].

The error at the output layer can be expressed as follows (remembering the chain rule from calculus)

$$(\boldsymbol{\delta}^L)^T := \frac{\partial L}{\partial \mathbf{z}^L} = \frac{\partial L}{\partial \mathbf{a}^L} \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \nabla L(\mathbf{a}^L)\, \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L). \tag{22}$$

The above equation shows us that the error at the output layer can be decomposed into the *gradient* $\nabla L(\mathbf{a}^L)$, and the *Jacobi* matrix $\mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L)$, from which the latter represents the derivative of a vector w.r.t. another vector. So, writing out every component of the above expression produces

$$\nabla L(\mathbf{a}^L)\, \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L) = \begin{bmatrix} \frac{\partial L}{\partial a_1^L} & \frac{\partial L}{\partial a_2^L} & \cdots & \frac{\partial L}{\partial a_{n^L}^L} \end{bmatrix} \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L} & \frac{\partial a_1^L}{\partial z_2^L} & \cdots & \frac{\partial a_1^L}{\partial z_{n^L}^L} \\ \frac{\partial a_2^L}{\partial z_1^L} & \frac{\partial a_2^L}{\partial z_2^L} & \cdots & \frac{\partial a_2^L}{\partial z_{n^L}^L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{n^L}^L}{\partial z_1^L} & \frac{\partial a_{n^L}^L}{\partial z_2^L} & \cdots & \frac{\partial a_{n^L}^L}{\partial z_{n^L}^L} \end{bmatrix}, \tag{23}$$

which ends up as a $1 \times n^L$ row vector. In its most general form, the above vector-matrix product represents **BP1.1**, so without making any assumptions about the loss function L and the activation function in the output layer, the above equation cannot be simplified any further. In the next section, we will show how to further specify this equation by choosing a specific loss and activation function.

Example

For multi-class classification problems, a common choice for the loss function is the categorical cross entropy (see equation 15) and a common choice for the activation function in the output layer is the *softmax* function, which unlike e.g. the sigmoid activation function, takes a vector as input and also outputs a vector, whose *j*-th component is defined as follows

$$a_j^l = \mathbf{f}([z_1^l, z_2^l, \dots, z_j^l, \dots, z_{n^l}^l])_j = \frac{e^{z_j^l}}{\sum_{k=1}^{n^l} e^{z_k^l}}. \tag{24}$$

First, we will try to find concrete expressions for each component of $\nabla L(\mathbf{a}^L)$ in equation (23). From the categorical cross entropy loss function in equation 15, we can derive that for $i = 1, 2, \dots, n^L$,

$$\frac{\partial L}{\partial a_j^L} = -\frac{y_j}{a_j^L}, \tag{25}$$

where we used the fact that $\hat{y}_j = a_j^L$.

Second, we want to find concrete expressions for each component of $\mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L)$ in (23). When taking the derivative of the Softmax function, we need to consider two cases. The first case is represented by $\frac{\partial a_j^L}{\partial z_k^L}$, if $j = k$, i.e. $\frac{\partial a_j^L}{\partial z_j^L}$.

$$\begin{aligned} \frac{\partial a_j^L}{\partial z_j^L} &= \frac{e^{z_j^L} \left(\sum_{k=1}^{n^L} e^{z_k^l} \right) - e^{z_j^L} e^{z_j^L}}{\left(\sum_{k=1}^{n^L} e^{z_k^L} \right)^2} \\ &= \frac{e^{z_j^L} \left(\left(\sum_{k=1}^{n^L} e^{z_k^L} \right) - e^{z_j^L} \right)}{\left(\sum_{k=1}^{n^L} e^{z_k^L} \right)^2} \\ &= \frac{e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} \frac{\left(\sum_{k=1}^{n^L} e^{z_k^L} \right) - e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} \\ &= \frac{e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} \left(\frac{\sum_{k=1}^{n^L} e^{z_k^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} - \frac{e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} \right) \end{aligned} \tag{26}$$

$$= \frac{e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}} \left(1 - \frac{e^{z_j^L}}{\sum_{k=1}^{n^L} e^{z_k^L}}\right),$$

where we can now use the definition of the Softmax function (equation 28) again to simplify further to

$$\frac{\partial a_j^L}{\partial z_j^L} = a_j^L(1 - a_j^L). \tag{27}$$

The second case is represented by $\frac{\partial a_j^L}{\partial z_k^L}$, where $k \neq j$, so that

$$\begin{aligned} \frac{\partial a_j^L}{\partial z_k^L} &= \frac{0 \times \left(\sum_{k=1}^{n^L} e^{z_k^L}\right) - e^{z_j^L} e^{z_k^L}}{\left(\sum_{k=1}^{n^L} e^{z_k^L}\right)^2} \\ &= \frac{-e^{z_j^L} e^{z_k^L}}{\left(\sum_{k=1}^{n^L} e^{z_k^L}\right)^2} \\ &= -\frac{e^{z_j^L}}{\left(\sum_{k=1}^{n^L} e^{z_k^L}\right)} \frac{e^{z_k^L}}{\left(\sum_{k=1}^{n^L} e^{z_k^L}\right)} \\ &= -a_j^L \, a_k^L. \end{aligned} \tag{28}$$

So, summarizing,

$$\frac{\partial a_j^L}{\partial z_k^L} = \begin{cases} a_j^L(1 - a_j^L) & \text{if} \quad j = k \\ -a_j^L \, a_k^L & \text{if} \quad j \neq k \end{cases} \tag{29}$$

Using (25) and (29), we can now fill in each component of (23) as follows

$$\nabla L(\mathbf{a}^L) \, \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L) = -\begin{bmatrix} \frac{y_1}{a_1^L} & \frac{y_2}{a_2^L} & \cdots & \frac{y_{n^L}}{a_{n^L}^L} \end{bmatrix} \begin{bmatrix} a_1^L(1 - a_1^L), & -a_1^L \, a_2^L & \dots & -a_1^L \, a_{n^L}^L \\ -a_2^L \, a_1^L, & a_2^L(1 - a_2^L) & \dots & -a_2^L \, a_{n^L}^L \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n^L}^L \, a_1^L, & -a_{n^L}^L \, a_2^L & \dots & a_{n^L}^L(1 - a_{n^L}^L) \end{bmatrix}, \tag{30}$$

which, when multiplied out, yields

$$\begin{gathered} \nabla L(\mathbf{a}^L) \, \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L) = \\ \left[-y_1(1 - a_1^L) + y_2 a_1^L + \ldots + y_{n^L} a_1^L \quad y_1 a_2^L - y_2(1 - a_2^L) + \ldots + y_{n^L} a_2^L \quad \ldots \quad y_1 a_{n^L}^L + y_2 a_{n^L}^L + \ldots + (-y_{n^L})(1 - a_{n^L}^L)\right] = \\ \left[-y_1 + a_1^L(y_1 + y_2 + \ldots + y_{n^L}) \quad -y_2 + a_2^L(y_1 + y_2 + \ldots + y_{n^L}) \quad \ldots \quad -y_{n^L} + a_{n^L}^L(y_1 + y_2 + \ldots + y_{n^L})\right]. \end{gathered} \tag{31}$$

Notice that $(y_1 + y_2 + \ldots + y_{n^L}) = 1$ due to the one hot encoded target vector $\mathbf{y}$. So, we can simplify the above expression to

$$\nabla L(\mathbf{a}^L) \, \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L) = -\left[(y_1 - a_1^L) \quad (y_2 - a_2^L) \quad \ldots \quad (y_{n^L} - a_{n^L}^L)\right]. \tag{32}$$

BP2.1

In order to represent the error of the previous layer $(\boldsymbol{\delta}^{l-1})^T$ in terms of the error in the current layer $(\boldsymbol{\delta}^l)^T$, it helps to view the loss function as a nested function of weighted input vectors, i.e. $L(\mathbf{z}^l(\mathbf{z}^{l-1}))$ which we want to derive w.r.t. $\mathbf{z}^{l-1}$. This can be done as follows

$$(\boldsymbol{\delta}^{l-1})^T := \frac{\partial L}{\partial \mathbf{z}^{l-1}} = \frac{\partial L}{\partial \mathbf{z}^l} \frac{\partial \mathbf{z}^l}{\partial \mathbf{z}^{l-1}} = \nabla L(\mathbf{z}^l) \, \mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}), \tag{33}$$

which can be written out explicitly as

$$\nabla L(\mathbf{z}^l) \, \mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = \begin{bmatrix} \frac{\partial L}{\partial z_1^l} & \frac{\partial L}{\partial z_2^l} & \cdots & \frac{\partial L}{\partial z_{n^l}^l} \end{bmatrix} \begin{bmatrix} \frac{\partial z_1^l}{\partial z_1^{l-1}} & \frac{\partial z_1^l}{\partial z_2^{l-1}} & \cdots & \frac{\partial z_1^l}{\partial z_{n^{l-1}}^{l-1}} \\ \frac{\partial z_2^l}{\partial z_1^{l-1}} & \frac{\partial z_2^l}{\partial z_2^{l-1}} & \cdots & \frac{\partial z_2^l}{\partial z_{n^{l-1}}^{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n^l}^l}{\partial z_1^{l-1}} & \frac{\partial z_{n^l}^l}{\partial z_2^{l-1}} & \cdots & \frac{\partial z_{n^l}^l}{\partial z_{n^{l-1}}^{l-1}} \end{bmatrix}. \tag{34}$$

In order to find an expression for every component of $\nabla L(\mathbf{z}^l)$, notice that by our definition in equation (21), we have defined every element as

$$\frac{\partial L}{\partial z_j^l} = \delta_j^l. \tag{35}$$

In order to find an expression for every component of $\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1})$, recall that $z_j^l = \sum_{k=1}^{n^{l-1}} \left(a_k^{l-1} w_{j,k}^l\right) + b_j^l$ and therefore,

$$\frac{\partial z_j^l}{\partial z_k^{l-1}} = \sum_{i=1}^{n^{l-1}} w_{j,i}^l \frac{\partial a_i^{l-1}}{\partial z_k^{l-1}}, \tag{36}$$

where we need to use the total differential. To add a little more intuition why the total differential must be used here, consider the following picture and assume that we wanted to determine $\frac{\partial z_1^l}{\partial z_2^{l-1}}$.

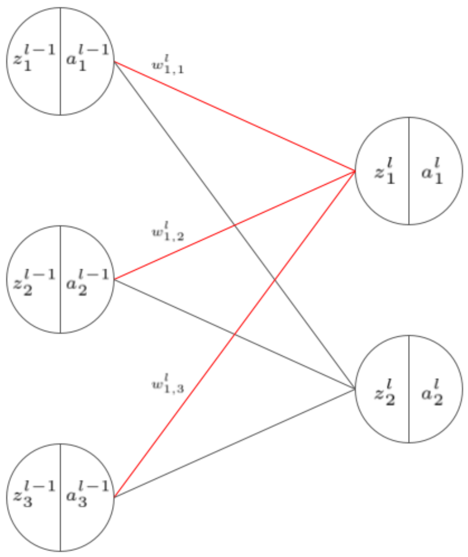

Figure 7

When determining $\frac{\partial z_1^l}{\partial z_2^{l-1}}$, we want to figure out how much z_1^l changes when z_2^{l-1} changes (that's how derivatives are defined). In order for z_1^l to change, there are 2 different paths how to achieve that:

1. Direct path: A change in z_2^{l-1} leads to a change a_2^{l-1}, which is amplified by $w_{1,2}^l$.
2. Indirect path: A change in z_2^{l-1} might cause a change in a_1^{l-1} and a_3^{l-1} (if e.g. the softmax activation function is used, since the activations of the softmax function alway sum up to 1), and each change in a_1^{l-1} and a_3^{l-1} is amplified by $w_{1,1}^l$ and $w_{1,3}^l$ respectively.

Using (35) and (36), we can fill in each component of (34) as follows

$$\nabla L(\mathbf{z}^l)\,\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = \begin{bmatrix}\delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} \sum_{i=1}^{n^{l-1}} w_{1,i}^l \frac{\partial a_i^{l-1}}{\partial z_1^{l-1}} & \sum_{i=1}^{n^{l-1}} w_{1,i}^l \frac{\partial a_i^{l-1}}{\partial z_2^{l-1}} & \dots & \sum_{i=1}^{n^{l-1}} w_{1,i}^l \frac{\partial a_i^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \\ \sum_{i=1}^{n^{l-1}} w_{2,i}^l \frac{\partial a_i^{l-1}}{\partial z_1^{l-1}} & \sum_{i=1}^{n^{l-1}} w_{2,i}^l \frac{\partial a_i^{l-1}}{\partial z_2^{l-1}} & \dots & \sum_{i=1}^{n^{l-1}} w_{2,i}^l \frac{\partial a_i^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n^{l-1}} w_{n^l,i}^l \frac{\partial a_i^{l-1}}{\partial z_1^{l-1}} & \sum_{i=1}^{n^{l-1}} w_{n^l,i}^l \frac{\partial a_i^{l-1}}{\partial z_2^{l-1}} & \dots & \sum_{i=1}^{n^{l-1}} w_{n^l,i}^l \frac{\partial a_i^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \end{bmatrix}, \tag{37}$$

which can be decomposed into

$$\nabla L(\mathbf{z}^l)\,\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = \begin{bmatrix}\delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \dots & w_{n^l,n^{l-1}}^l \end{bmatrix} \begin{bmatrix} \frac{\partial a_1^{l-1}}{\partial z_1^{l-1}} & \frac{\partial a_1^{l-1}}{\partial z_2^{l-1}} & \dots & \frac{\partial a_1^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \\ \frac{\partial a_2^{l-1}}{\partial z_1^{l-1}} & \frac{\partial a_2^{l-1}}{\partial z_2^{l-1}} & \dots & \frac{\partial a_2^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{n^{l-1}}^{l-1}}{\partial z_1^{l-1}} & \frac{\partial a_{n^{l-1}}^{l-1}}{\partial z_2^{l-1}} & \dots & \frac{\partial a_{n^{l-1}}^{l-1}}{\partial z_{n^{l-1}}^{l-1}} \end{bmatrix}, \tag{38}$$

or in short,

$$(\boldsymbol{\delta}^{l-1})^T = \nabla L(\mathbf{z}^l)\,\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = (\boldsymbol{\delta}^l)^T\,\mathbf{W}^l\,\mathbf{J}_{\mathbf{a}^{l-1}}(\mathbf{z}^{l-1}), \tag{39}$$

or similarly, after iterating one layer forward,

$$(\boldsymbol{\delta}^l)^T = \nabla L(\mathbf{z}^{l+1})\,\mathbf{J}_{\mathbf{z}^{l+1}}(\mathbf{z}^l) = (\boldsymbol{\delta}^{l+1})^T\,\mathbf{W}^{l+1}\,\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l) \tag{40}$$

The above equation represents **BP2.1** in its most general form. This is a nice result, because the term $\mathbf{J}_{\mathbf{a}^{l-1}}(\mathbf{z}^{l-1})$ already appeared in BP1.1, which shows us that for any activation function we want to use, we just need to implement its forward pass and its Jacobi matrix for the backpropagation algorithm. In other words, we regard any activation function as an interchangeable - *and completely independent* - component of our neural network.

Example

A common choice for activation functions in the hidden layers is the Sigmoid function (see equation 5), whose derivative is defined as follows

$$\frac{\partial a_j^l}{\partial z_k^l} = \begin{cases} a_j^l(1-a_k^l) & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{41}$$

Using the above result, we can write out (40) as follows

$$\nabla L(\mathbf{z}^l)\,\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = \\ \begin{bmatrix}\delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \dots & w_{n^l,n^{l-1}}^l \end{bmatrix} \begin{bmatrix} a_1^{l-1}(1-a_1^{l-1}) & 0 & \dots & 0 \\ 0 & a_2^{l-1}(1-a_2^{l-1}) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{n^{l-1}}^{l-1}(1-a_{n^{l-1}}^{l-1}) \end{bmatrix}, \tag{42}$$

which, in this specific example, reduces to

$$\nabla L(\mathbf{z}^l)\,\mathbf{J}_{\mathbf{z}^l}(\mathbf{z}^{l-1}) = \begin{bmatrix}\delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \dots & w_{n^l,n^{l-1}}^l \end{bmatrix} \odot \begin{bmatrix} a_1^{l-1}(1-a_1^{l-1}) & a_2^{l-1}(1-a_2^{l-1}) & \dots & a_{n^{l-1}}^{l-1}(1-a_{n^{l-1}}^{l-1})\end{bmatrix}. \tag{43}$$

because $\boldsymbol{\delta}^l\ \mathbf{W}^l$ yields a row vector, the Jacobian is a diagonal matrix, and a row vector multiplied with a diagonal matrix can be reformulated as the *Hadamard* product ($\odot$ operator) between the row vector and the elements on the main diagonal of the diagonal matrix.

BP3.1

After calculating the errors at a certain layer, we now want to relate them to the derivative of the loss w.r.t the weights in layer l, which we can be done as follows

$$\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \mathbf{W}^l} = \nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{W}^l), \tag{44}$$

which can be written out explicitly as

$$\nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{W}^l) = \begin{bmatrix}\frac{\partial L}{\partial z_1^l} & \frac{\partial L}{\partial z_2^l} & \cdots & \frac{\partial L}{\partial z_{n^l}^l}\end{bmatrix} \begin{bmatrix} \frac{\partial z_1^l}{\partial w_{1,1}^l} & \frac{\partial z_1^l}{\partial w_{1,2}^l} & \dots & \frac{\partial z_1^l}{\partial w_{1,n^{l-1}}^l} & \frac{\partial z_1^l}{\partial w_{2,1}^l} & \frac{\partial z_1^l}{\partial w_{2,2}^l} & \dots & \frac{\partial z_1^l}{\partial w_{2,n^{l-1}}^l} & \dots, & \frac{\partial z_1^l}{\partial w_{n^l,1}^l} & \frac{\partial z_1^l}{\partial w_{n^l,2}^l} & \dots & \frac{\partial z_1^l}{\partial w_{n^l,n}^l} \\ \frac{\partial z_2^l}{\partial w_{1,1}^l} & \frac{\partial z_2^l}{\partial w_{1,2}^l} & \dots & \frac{\partial z_2^l}{\partial w_{1,n^{l-1}}^l} & \frac{\partial z_2^l}{\partial w_{2,1}^l} & \frac{\partial z_2^l}{\partial w_{2,2}^l} & \dots & \frac{\partial z_2^l}{\partial w_{2,n^{l-1}}^l} & \dots & \frac{\partial z_2^l}{\partial w_{n^l,1}^l} & \frac{\partial z_2^l}{\partial w_{n^l,2}^l} & \dots & \frac{\partial z_2^l}{\partial w_{n^l,n}^l} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n^l}^l}{\partial w_{1,1}^l} & \frac{\partial z_{n^l}^l}{\partial w_{1,2}^l} & \dots & \frac{\partial z_{n^l}^l}{\partial w_{1,n^{l-1}}^l} & \frac{\partial z_{n^l}^l}{\partial w_{2,1}^l} & \frac{\partial z_{n^l}^l}{\partial w_{2,2}^l} & \dots & \frac{\partial z_{n^l}^l}{\partial w_{2,n^{l-1}}^l} & \dots & \frac{\partial z_{n^l}^l}{\partial w_{n^l,1}^l} & \frac{\partial z_{n^l}^l}{\partial w_{n^l,2}^l} & \dots & \frac{\partial z_{n}^l}{\partial w_{n^l,n}^l} \end{bmatrix}$$

where $J_{\mathbf{z}^l}(\mathbf{W}^l)$ is a $n^l \times (n^l \times n^{l-1})$ matrix, since there are n^l components in $\mathbf{z}^l$ and $n^l \times n^{l-1}$ components in $\mathbf{W}^l$.

Again, we will first find expressions for each component of $\nabla L(\mathbf{z}^l)$ and after that, for each component of $J_{\mathbf{z}^l}(\mathbf{W}^l)$. The components of $\nabla L(\mathbf{z}^l)$ are given by (21), and to derive each component of $J_{\mathbf{z}^l}(\mathbf{W}^l)$, we need to consider two cases again.

First, consider $\frac{\partial z_j^l}{\partial w_{i,k}^l}$ if $j = i$, i.e. $\frac{\partial z_j^l}{\partial w_{j,k}^l}$. Remember that $z_j^l = \sum_{k=1}^{n^{l-1}} w_{j,k}^l\ a_k^{l-1} + b_j^l$, so

$$\frac{\partial z_j^l}{\partial w_{j,k}^l} = a_k^{l-1}. \tag{46}$$

Next, consider $\frac{\partial z_j^l}{\partial w_{i,k}^l}$ if $j \neq i$. In that case, the weight $w_{i,k}^l$ is not connected to neuron j in layer l, so z_j^l will never change if $w_{i,k}^l$ changes and therefore,

$$\frac{\partial z_j^l}{\partial w_{i,k}^l} = 0. \tag{47}$$

Summarizing, we have that

$$\frac{\partial z_j^l}{\partial w_{i,k}^l} = \begin{cases} a_k^{l-1} & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases} \tag{48}$$

Using (21) and (48), we can now fill in each value of (45) as follows

$$\nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{W}^l) = \begin{bmatrix}\delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} a_1^{l-1} & a_2^{l-1} & \dots & a_{n^{l-1}}^{l-1} & 0 & 0 & \dots & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & a_1^{l-1} & a_2^{l-1} & \dots & a_{n^{l-1}}^{l-1} & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & \dots & a_1^{l-1} & a_2^{l-1} & \dots & a_{n^{l-1}}^{l-1} \end{bmatrix}. \tag{49}$$

Multiplying out the above expression yields the following $1 \times (n^l \times n^{l-1})$ row vector

$$\nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{W}^l) = \begin{bmatrix}\delta_1^l\ a_1^{l-1} & \delta_1^l\ a_2^{l-1} & \dots & \delta_1^l\ a_{n^{l-1}}^{l-1} & \delta_2^l\ a_1^{l-1} & \delta_2^l\ a_2^{l-1} & \dots & \delta_2^l\ a_{n^{l-1}}^{l-1} & \dots & \delta_{n^l}^l\ a_1^{l-1} & \delta_{n^l}^l\ a_2^{l-1} & \dots & \delta_{n^l}^l\ a_{n^{l-1}}^{l-1}\end{bmatrix}, \tag{50}$$

which we want to stack as

$$\nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{w}^l) = \begin{bmatrix} \delta_1^l a_1^{l-1} & \delta_1^l a_2^{l-1} & \dots & \delta_1^l a_{n^{l-1}}^{l-1} \\ \delta_2^l a_1^{l-1} & \delta_2^l a_2^{l-1} & \dots & \delta_2^l a_{n^{l-1}}^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{n^l}^l a_1^{l-1} & \delta_{n^l}^l a_2^{l-1} & \dots & \delta_{n^l}^l a_{n^{l-1}}^{l-1} \end{bmatrix}, \tag{51}$$

because now, we can decompose the above equation into two quantities we have already computed, i.e.

$$\nabla L(\mathbf{z}^l)\ J_{\mathbf{z}^l}(\mathbf{W}^l) = \begin{bmatrix}\delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_{n^l}^l\end{bmatrix} \begin{bmatrix} a_1^{l-1} & a_2^{l-1} & \dots & a_{n^{l-1}}^{l-1}\end{bmatrix} = \boldsymbol{\delta}^l\ (\mathbf{a}^{l-1})^T, \tag{52}$$

which represents **BP3.1**.

BP4.1

Now, we want to relate the errors of each layer to the derivative of the loss w.r.t. the biases. The derivative of the loss w.r.t the biases can be expressed as follows

$$\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \mathbf{b}^l} = \nabla L(\mathbf{z}^l)\; J_{\mathbf{z}^l}(\mathbf{b}^l), \tag{53}$$

which can be written out explicitly as

$$\nabla L(\mathbf{z}^l)\; J_{\mathbf{z}^l}(\mathbf{w}^l) = \begin{bmatrix} \frac{\partial L}{\partial z_1^l} & \frac{\partial L}{\partial z_2^l} & \dots & \frac{\partial L}{\partial z_{n^l}^l} \end{bmatrix} \begin{bmatrix} \frac{\partial z_1^l}{\partial b_1^l} & \frac{\partial z_1^l}{\partial b_2^l} & \dots & \frac{\partial z_1^l}{\partial b_{n^l}^l} \\ \frac{\partial z_2^l}{\partial b_1^l} & \frac{\partial z_2^l}{\partial b_2^l} & \dots & \frac{\partial z_2^l}{\partial b_{n^l}^l} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n^l}^l}{\partial b_1^l} & \frac{\partial z_{n^l}^l}{\partial b_2^l} & \dots & \frac{\partial z_{n^l}^l}{\partial b_{n^l}^l} \end{bmatrix} \tag{54}$$

Again, the components of $\nabla L(\mathbf{z}^l)$ are given by (21), and to derive each component of $J_{\mathbf{z}^l}(\mathbf{b}^l)$, we can easily see from (3) that

$$\frac{\partial z_j^l}{\partial b_k^l} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{55}$$

So, using (21) and (55), we can re-write equation (54) as follows

$$\nabla L(\mathbf{z}^l)\; J_{\mathbf{z}^l}(\mathbf{w}^l) = \begin{bmatrix} \delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}, \tag{56}$$

which simply means that

$$\frac{\partial L}{\partial \mathbf{b}^l} = \nabla L(\mathbf{z}^l)\; J_{\mathbf{z}^l}(\mathbf{w}^l) = (\boldsymbol{\delta}^l)^T. \tag{57}$$

The above equation represents **BP4.1**.

Backpropagation for a Batch of Training Examples

In the previous section, we have derived equations which can help us to compute the gradients of a *single* training example. Computing these expressions separately for each training example will take a tremendous amount of time, so in this section, we aim to extend these equations so that we can compute the gradient for `batch_size` training examples at once, harnessing already optimized and extremely efficient matrix multiplication libraries such as `NumPy`.

BP1.2

Recall from BP1.1 that

$$(\boldsymbol{\delta}^L)^T = \nabla L(\mathbf{a}^L)\; \mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L). \tag{58}$$

In order to remain conform with the notation we used when describing the forward propagation for `batch_size` training examples at once, we will first transpose both sides of the above equation to

$$\boldsymbol{\delta}^L = (\mathbf{J}_{\mathbf{a}^L}(\mathbf{z}^L))^T\; (\nabla L(\mathbf{a}^L))^T = \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L} & \frac{\partial a_2^L}{\partial z_1^L} & \dots & \frac{\partial a_{n^L}^L}{\partial z_1^L} \\ \frac{\partial a_1^L}{\partial z_2^L} & \frac{\partial a_2^L}{\partial z_2^L} & \dots & \frac{\partial a_{n^L}^L}{\partial z_2^L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_1^L}{\partial z_{n^L}^L} & \frac{\partial a_2^L}{\partial z_{n^L}^L} & \dots & \frac{\partial a_{n^L}^L}{\partial z_{n^L}^L} \end{bmatrix} \begin{bmatrix} \frac{\partial L}{\partial a_1^L} \\ \frac{\partial L}{\partial a_2^L} \\ \vdots \\ \frac{\partial L}{\partial a_{n^L}^L} \end{bmatrix}, \tag{59}$$

Next, for each training example m, we will simply stack the Jacobian $(\mathbf{J}_{\mathbf{a}^{L,m}}(\mathbf{z}^{L,m}))^T$ and the gradient of each training example $(\nabla L(\mathbf{a}^{L,m}))^T$ for all $m = 1, 2, \dots, M$ along the first axis (`axis=0`), which again, we chose to draw in the depth dimension, such that

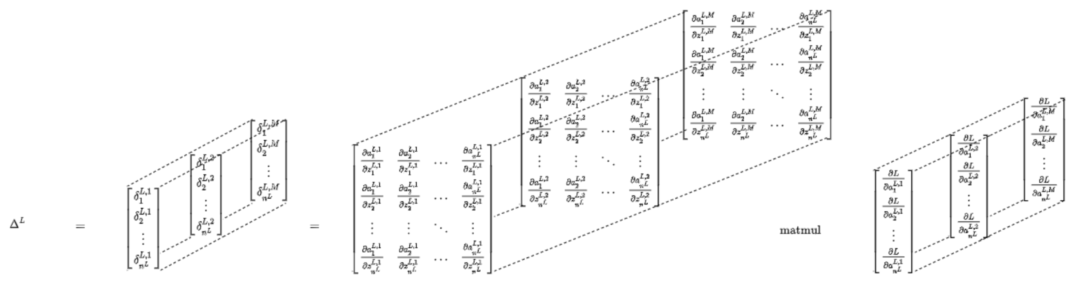

Figure 8

where $\mathbf{\Delta}^L$ is an $M \times n^L \times 1$ dimensional array. Notice that the matrix-vector multiplications for each training example, i.e. each element in the depth dimension or where `axis=0`, is done independently and in parallel. In its most general form, the above equation represents **BP1.2**.

Example

Assuming we are using the categorical cross entropy cost function from equation (19) and the Softmax activation function in the output layer, we can proceed similarly as above and first transpose both sides of (32) and then stack the error of each training example in a a separate element of the depth dimension. Having done so, we will end up with the following expression

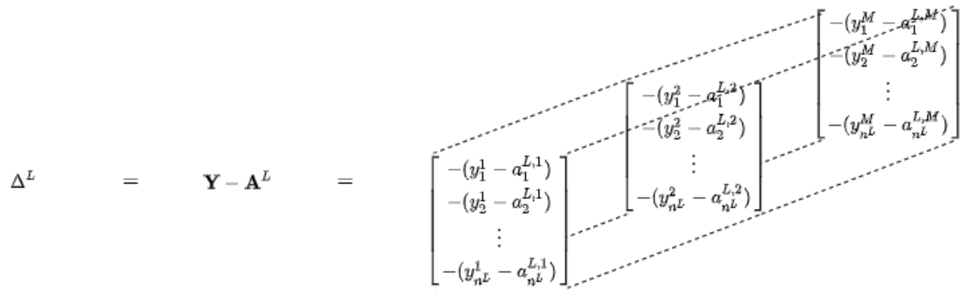

Figure 9

which is easily computed, because we are given $\mathbf{Y}$ (since we are talking about a supervised learning problem here) and we already computed $\mathbf{A}^L$ during the forward propagation.

BP2.2

Recall from BP2.1 that

$$(\boldsymbol{\delta}^{l-1})^T = (\boldsymbol{\delta}^l)^T \, \mathbf{W}^l \, \mathbf{J}_{\mathbf{a}^{l-1}}(\mathbf{z}^{l-1}). \tag{60}$$

Again, we will first transpose both sides of the above equation giving us

$$\boldsymbol{\delta}^{l-1} = (\mathbf{J}_{\mathbf{a}^{l-1}}(\mathbf{z}^{l-1}))^T \, (\mathbf{W}^l)^T \, \boldsymbol{\delta}^l. \tag{61}$$

As before, we want to stack each error $\boldsymbol{\delta}^{l,m}$ and each Jacobian $\left(\mathbf{J}_{\mathbf{a}^{l-1,m}}(\mathbf{z}^{l-1,m})\right)^T$ along `axis and broadcast each weight matrix $(\mathbf{W}^l)^T$ for all $m = 1, 2, \ldots, M$ training examples in each element of the depth dimension, such that

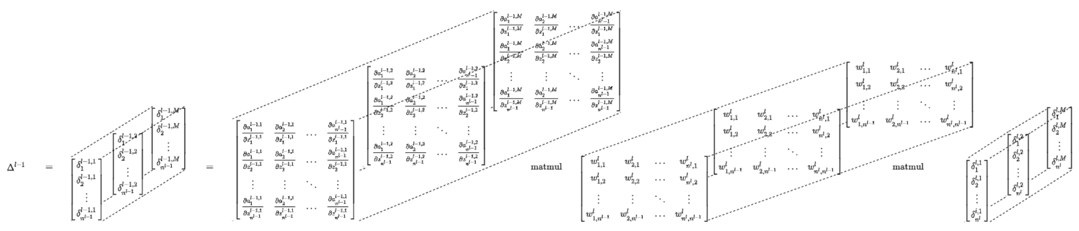

Figure 10

where Δ^{l-1} is a $M \times n^{l-1} \times 1$ dimensional array. Again, the matrix-matrix-vector multiplication for each training example is done independently. In its most general form, Figure 10 represents **BP2.2**.

Example

Using the sigmoid activation function in layer $l-1$, figure 10 can be specified as follows

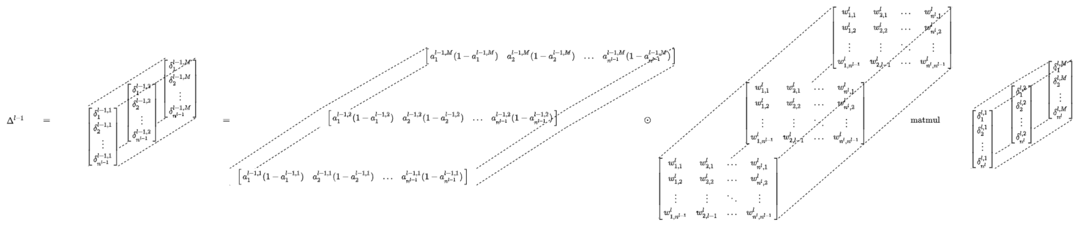

Figure 11

BP3.2

Remember that the real quantities of interest during backpropagation are the gradients of the *cost* function w.r.t. the weights and biases, because we need those to adjust the weights and biases into the direction so that the overall cost decreases. Also, recall that the cost is just the averaged loss over M training examples, i.e.

$$\frac{\partial C}{\partial \mathbf{W}^l} = \frac{1}{M}\sum_{m=1}^{M} \frac{\partial L^m}{\partial \mathbf{W}^l}, \tag{62}$$

where L^m is the loss associated with the m-th training example.

From BP3.1, we know that

$$\frac{\partial L}{\partial \mathbf{W}^l} = \begin{bmatrix} \delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_{n^l}^l \end{bmatrix} \begin{bmatrix} a_1^{l-1} & a_2^{l-1} & \dots & a_{n^{l-1}}^{l-1} \end{bmatrix}, \tag{63}$$

so, using that, we can rewrite (62) as follows

$$\frac{\partial C}{\partial \mathbf{W}^l} = \frac{1}{M}\sum_{m=1}^{M} \begin{bmatrix} \delta_1^{l,m} \\ \delta_2^{l,m} \\ \vdots \\ \delta_{n^l}^{l,m} \end{bmatrix} \begin{bmatrix} a_1^{l-1,m} & a_2^{l-1,m} & \dots & a_{n^{l-1}}^{l-1,m} \end{bmatrix}. \tag{64}$$

Using the same notation as before, we can represent the above equation such that each training example belongs to a different element of the depth dimension

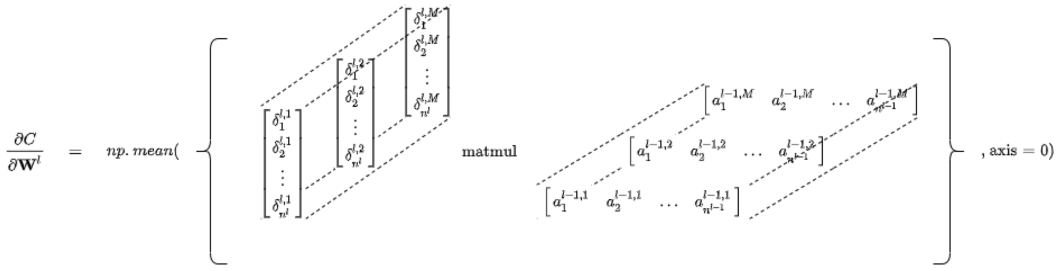

Figure 12

where $np.mean$ refers to the `mean` function of `NumPy` . Figure 12 can be multiplied out as

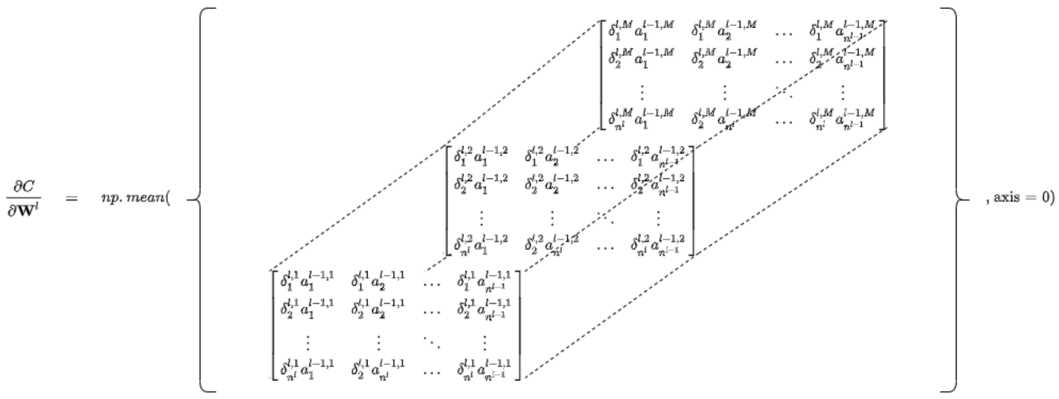

Figure 13

Note that the average is taken along `axis=0`, i.e. we take the average across all training examples of each element of the matrix resulting from $\boldsymbol{\delta}^l(\mathbf{a}^{l-1})^T$. Figure 13 represents **BP3.2**, where $\frac{\partial C}{\partial \mathbf{W}^l}$ is an $M \times n^l \times n^{l-1}$ dimensional array.

BP4.2

Finally, the other real quantity of interest is the gradient of the cost function w.r.t. the biases. Again, using the fact that the cost is an average over M training examples, we can deduce that

$$\frac{\partial C}{\partial \mathbf{b}^l} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial L^m}{\partial \mathbf{b}^l}. \tag{65}$$

Remember from BP4.1 that

$$\frac{\partial L}{\partial \mathbf{b}^l} = \begin{bmatrix} \delta_1^l & \delta_2^l & \dots & \delta_{n^l}^l \end{bmatrix}. \tag{66}$$

First of all, we will transpose both sides of the above equation in order to remain conform with the notations used from BP1.2 to BP3.2, i.e.

$$\left(\frac{\partial L}{\partial \mathbf{b}^l}\right)^T = \begin{bmatrix} \delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_{n^l}^l \end{bmatrix}. \tag{67}$$

From here on out, it is really straight forward. Transpose both sides of (65) and use (67), yielding

$$\left(\frac{\partial C}{\partial \mathbf{b}^l}\right)^T = \frac{1}{M} \sum_{m=1}^{M} \begin{bmatrix} \delta_1^{l,m} \\ \delta_2^{l,m} \\ \vdots \\ \delta_{n^l}^{l,m} \end{bmatrix}. \tag{68}$$

Representing (68) such that each training example refers to a separate element of the depth dimension, we will get

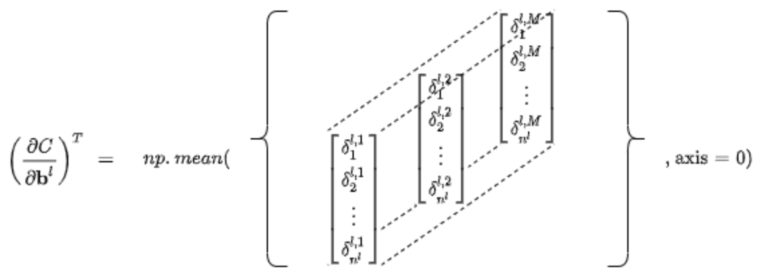
$\left(\frac{\partial C}{\partial \mathbf{b}^l}\right)^T = np.mean($
$\delta_1^{l,1}$ $\delta_2^{l,1}$ $\vdots$ $\delta_{n^l}^{l,1}$
$\delta_1^{l,2}$ $\delta_2^{l,2}$ $\vdots$ $\delta_{n^l}^{l,2}$
$\delta_1^{l,M}$ $\delta_2^{l,M}$ $\vdots$ $\delta_{n^l}^{l,M}$
$, \text{axis} = 0)$

Figure 14

where again, we take the average across all training examples of each element of $\boldsymbol{\delta}^l$. Figure 14 represents **BP4.2**, where $\left(\frac{\partial C}{\partial \mathbf{b}^l}\right)^T$ is an $M \times n^l \times 1$ dimensional array.

Gradient Descent

In the previous section, we described how to compute the gradients, which mathematically speaking, point into the *direction* of the steepest ascent of the cost function. In this section, we will describe how to use the gradients in order to *update* the weights and biases such that the cost decreases.

We will describe a very simple way of updating the weights and biases which is called *Stochastic Gradient Descent* (SGD). Assuming that we have calculated BP3.2 and BP4.2 for all layers, we can perform the weight updates as

$$\mathbf{W}_s^l = \mathbf{W}_{s-1}^l - \lambda \left(\frac{\partial C}{\partial \mathbf{W}^l}\right)_{s-1}, \tag{69}$$

and similarly, the bias updates as

$$\mathbf{b}_s^l = \mathbf{b}_{s-1}^l - \lambda \left(\frac{\partial C}{\partial \mathbf{b}^l}\right)_{s-1}^T, \tag{70}$$

for update steps $i = 1, 2, \dots, S$. Notice that $\mathbf{w}_{s=0}^l$ and $\mathbf{b}_{s=0}^l$ are initialized randomly, S represents the number of update steps and λ represents the *learning rate* controlling the step size toward the local (and hopefully global) minimum of the cost function.

Assuming that we divided our dataset into batches with at most M training examples each, we will end up with S batches, where S is computed as

$$S = \text{round\_up}(N/M), \tag{71}$$

where round_up is a function that always rounds a floating point number *up* to the nearest integer and where N represents the number of all training examples in total. Notice that S always needs to be rounded up in order to make sure that during one *epoch* [3], all training examples have been forward- and backward propagated through the network. If we rounded down, some training examples might be skipped.

Note that, before updating the weights and biases of each layer, we must have calculated the errors of *all* layers beforehand. Calculating the gradients and updating the weights and biases for each layer simultaneously will yield incorrect gradients, because in BP2.1 and BP2.2, we can see that the error of layer l is also a function of the weights in layer $l+1$. So, if you were to calculate the error of layer $l+1$ and then immediately update the weights in layer $l+1$, you will get a wrong error for layer l in the next iteration. To get the correct errors of each layer, you need to keep the weights fixed. Only after having calculated the errors of each layer, you may update the weights and biases.

Loss functions

This section will show how all relevant loss functions and their gradients $\nabla L(\mathbf{a}^L) = \frac{\partial L}{\partial \mathbf{a}^L}$, which are needed as an interim quantity to initialize the error at the output layer (see BP1.1 and BP1.2).

Categorical Crossentropy

Recall from (15) that

$$L = -\sum_{i=1}^{n^L} y_i \; log(a_i^L), \tag{72}$$

where we used that $\hat{\mathbf{y}}^m = \mathbf{a}^{L,m}$. Its gradient is defined as

$$\frac{\partial L}{\partial \mathbf{a}^L} = -\frac{\mathbf{y}}{\mathbf{a}^L} = -\begin{bmatrix} \frac{y_1}{a_1^L} & \frac{y_2}{a_2^L} & \dots & \frac{y_{n^L}}{a_{n^L}^L} \end{bmatrix} \tag{73}$$

Sum of Squared Errors

The Sum of Squared Errors loss is defined as

$$L = \frac{1}{2}\sum_{i=1}^{n^L} \left(y_i - a_i^L\right)^2 \tag{74}$$

and its gradient is

$$\frac{\partial L}{\partial \mathbf{a}^L} = -(\mathbf{y} - \mathbf{a}^L) = -\begin{bmatrix} (y_1 - a_1^L) & (y_2 - a_2^L) & \dots & (y_{n^L} - a_{n^L}^L) \end{bmatrix} \tag{75}$$

Activation Functions

This section will show all relevant activations functions and their corresponding Jacobians $\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l)$ which are needed as an interim quantity when initializing the error at the output layer (see BP1.1 and BP1.2) as well as an interim quantity when backpropagating the error from layer to layer (see BP2.1 and BP2.2). Notice that for all activation functions shown here, the Jacobians are always symmetric matrices, so in the actual Python implementation of the activation functions, we may use that $(\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l))^T = \mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l)$.

Sigmoid

Recall from (5) that

$$a_i^l = f(z_i^l) = \frac{1}{1+e^{-z_i^l}}. \tag{76}$$

From (45) and 46), we can infer that the Jacobian of the Sigmoid function is

$$\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l) = \begin{bmatrix} a_1^l(1-a_1^l) & 0 & \dots & 0 \\ 0 & a_2^l(1-a_2^l) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{n^l}^l(1-a_{n^l}^l) \end{bmatrix}. \tag{77}$$

Numerical Stability Amendment

Imagine the input to the sigmoid function z_i^l is largely negative. In that case, the exponent of e would be extremely large and in that case, $e^{z_i^l}$ would converge to infinity very quickly. If you were to implement the above version of the sigmoid function Python, you will most likely encounter a situation where at some point, $e^{z_i^l} = \infty$ and if ∞ is used in subsequent computations, you will get some error. To avoid this error, we will use the following, equivalent version of the sigmoid function for any negative z_i^l.

$$a_i^l = f(z_i^l) = \frac{e^{z_i^l}}{1+e^{z_i^l}} \tag{78}$$

To see how the above amendment equals the original sigmoid function, see the following proof.

$$\begin{gathered} \frac{e^{z_i^l}}{1+e^{z_i^l}} = \\ \frac{e^{z_i^l}}{1+e^{z_i^l}} \frac{e^{-z_i^l}}{e^{-z_i^l}} = \\ \frac{1}{e^{-z_i^l}+e^{z_i^l}e^{-z_i^l}} = \\ \frac{1}{1+e^{-z_i^l}} \end{gathered} \tag{79}$$

For large positive z_i^l, $e^{-z_i^l}$ will convert to 0, in which case we can use the original version of the sigmoid function in (78).

ReLU

The Rectified Linear Unit (ReLU) is defined as

$$a_i^l = f(z_i^l) = max(0, z_i^l). \tag{80}$$

Its derivative is actually not defined for $z = 0$, but in practice, the probability that $z = 0.000000000...$ exactly is infinitesimal. So, we will define the derivative of the ReLU function as

$$\frac{\partial a_i^l}{} \quad \begin{cases} 1 & \text{if} \quad z_i^l > 0 \end{cases}$$

$$\frac{\partial a_i^l}{\partial z_i^l} = f'(z_i^l) = \begin{cases} 1 & \text{if} \quad z_i^l > 0 \\ 0 & \text{if} \quad z_i^l \leq 0 \end{cases}. \tag{81}$$

Since like the Sigmoid function, ReLU function also just depends on a single scalar, we know that

$$\frac{\partial a_i^l}{\partial z_j^l} = \begin{cases} f'(z_i^l) & \text{if } i = j \\ 0 & \text{if } i \neq k \end{cases} \tag{82}$$

where $f'(z_i^l)$ was already defined in (81). Using (82), we can construct Jacobian of the ReLU as follows

$$\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l) = \begin{bmatrix} f'(z_1^l) & 0 & \dots & 0 \\ 0 & f'(z_2^l) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & f'(z_{n^l}^l) \end{bmatrix}. \tag{83}$$

tanh

The tanh function is defined as follows

$$a_i^l = f(z_i^L) = \frac{e^{z_i^l} - e^{-z_i^l}}{e^{z_i^l} + e^{-z_i^l}} \tag{84}$$

and one can show that its derivative is

$$\frac{\partial a_i^l}{\partial z_j^l} = \begin{cases} 1 - (a_i^l)^2 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}. \tag{85}$$

Hence, its Jacobian is

$$\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l) = \begin{bmatrix} 1 - (a_1^l)^2 & 0 & \dots & 0 \\ 0 & 1 - (a_2^l)^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 - (a_{n^l}^l)^2 \end{bmatrix}. \tag{86}$$

Numerical Stability Amendment

Unlike the sigmoid function, the tanh function will suffer from numerical instability if both $z_i^l \to \infty$ and $z_i^l \to -\infty$, because once the exponent is positive and once it is negative in (84). To avoid numerical instability for large positive z_i^l, we will introduce the following equivalent version:

$$a_i^l = f(z_i^l) = \frac{1 - e^{-2z_i^l}}{1 + e^{-2z_i^l}} \tag{87}$$

Proof:

$$\begin{gathered} \frac{1 - e^{-2z_i^l}}{1 + e^{-2z_i^l}} = \\ \frac{1 - e^{-2z_i^l}}{1 + e^{-2z_i^l}} \frac{e^{z_i^l}}{e^{z_i^l}} = \\ \frac{e^{z_i^l} - e^{z_i^l} e^{-2z_i^l}}{e^{z_i^l} + e^{z_i^l} e^{-2z_i^l}} = \\ \frac{e^{z_i^l} - e^{-z_i^l}}{e^{z_i^l} + e^{-z_i^l}}, \end{gathered} \tag{88}$$

Now, for large z_i^l, $e^{-2z_i^l}$ will approach 0.

For large negative inputs, we will implement the following, equivalent version:

$$a_i^l = f(z_i^l) = \frac{e^{2z_i^l} - 1}{e^{2z_i^l} + 1} \tag{89}$$

Proof:

$$\begin{gathered} \frac{e^{2z_i^l} - 1}{e^{2z_i^l} + 1} = \\ \frac{e^{2z_i^l} - 1}{e^{2z_i^l} + 1} \frac{e^{-z_i^l}}{e^{-z_i^l}} = \\ \frac{e^{2z_i^l} e^{-z_i^l} - e^{-z_i^l}}{e^{2z_i^l} e^{-z_i^l} + e^{-z_i^l}} = \\ \frac{e^{z_i^l} - e^{-z_i^l}}{e^{z_i^l} + e^{-z_i^l}}, \end{gathered} \tag{90}$$

Now, for large negative z_i^l, $e^{2z_i^l}$ will also approach 0.

Softmax

Unlike all other activation functions discussed so far, the Softmax function is a vector valued function which receives a vector of length n as input and also outputs a vector of length n, whose elements sum up to 1. Recall from (24) that the Softmax function is defined as

$$a_i^l = \mathbf{f}([z_1^l, z_2^l, \dots, z_i^l, \dots, z_{n^l}^l])_i = \frac{e^{z_i^l}}{\sum_{k=1}^{n^l} e^{z_k^l}} \tag{91}$$

and recall from (30), that its Jacobian is

$$\mathbf{J}_{\mathbf{a}^l}(\mathbf{z}^l) = \begin{bmatrix} a_1^l(1-a_1^l), & -a_1^l\, a_2^l & \dots & -a_1^l\, a_{n^l}^l \\ -a_2^l\, a_1^l, & a_2^l(1-a_2^l) & \dots & -a_2^l\, a_{n^l}^l \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n^l}^l\, a_1^l, & -a_{n^l}^l\, a_2^l & \dots & a_{n^l}^l(1-a_{n^l}^l) \end{bmatrix}. \tag{92}$$

Numerical Stability Amendment

Because the exponent $e^{z_i^l}$ is positive, we might run into numerical overflow problems if $z_i^l \to \infty$. So, we should try to reduce $e^{z_i^l}$ somehow without changing the result of (91). The following implementation will do just that:

$$\begin{gathered} a_i^l = \mathbf{f}([z_1^l, z_2^l, \dots, z_i^l, \dots, z_{n^l}^l])_i = \\ \mathbf{f}(\mathbf{z}^l)_i = \frac{e^{z_i^l - \max(\mathbf{z}^l)}}{\sum_j^{n^l} e^{z_j^l - \max(\mathbf{z}^l)}} \end{gathered} \tag{93}$$

Proof:

$$\begin{gathered} \frac{e^{z_i^l - \max(\mathbf{z}^l)}}{\sum_j^{n^l} e^{z_j^l - \max(\mathbf{z}^l)}} = \\ \frac{e^{z_i^l}\, e^{-\max(\mathbf{z}^l)}}{\sum_j^{n^l} e^{z_j^l}\, e^{-\max(\mathbf{z}^l)}} = \\ \frac{e^{z_i^l}}{\sum_{j=1}^{n^l} e^{z_j^l}} \frac{e^{-\max(\mathbf{z}^l)}}{e^{-\max(\mathbf{z}^l)}} = \\ \frac{e^{z_i^l}}{\sum_{j=1}^{n^l} e^{z_j^l}}, \end{gathered} \tag{94}$$

where $\max(\mathbf{z}^l)$ represents the maximum of $\mathbf{z}^l$, which could actually be replaced by any other constant. However, $\max(\mathbf{z}^l)$ seems like a good choice because by definition, it is a large value and hence keeps the exponent small. Notice that we can take $e^{-\max(\mathbf{z}^l)}$ out of the summation, because it will reduce to just some constant.

Implementation

The actual Python implementation can be found in the `src/lib` directory which contains the following packages:

- `activation_functions` : Contains separate modules (i.e. python files) for each activation function which all have the same signature so that they can be used interchangeably. For each activation function, the forward and backward pass are implemented as described in the Activation Functions section
- `data_generators` : Contains data generators yielding batches of training data for supervised learning problems. Each type of data generator must implement the `train`, `val`, and `test` method so that they can be used interchangeably. Currently, an image data generator has been implemented.
- `layers` : Contains different layer types responsible for forward- and backpropagation as well as gradient computations. Each layer type must implement the `init_parameters`, `forward_propagate`, `backward_propagate`, `compute_weight_gradients`, and `compute_bias_gradients` methods.
- `losses` : Implements the loss as well as cost computation and initializes backpropagation. Each loss must implement `compute_losses`, `compute_cost`, and `init_error` methods. We chose to initialize the backpropagation in this class, because initializing backpropagation is dependent on the choice of loss/cost function.
- `metrics` : Implements different metrics for evaluating performance during or after training. Each metric must implement the `update_state`, `result`, and `reset_state` methods.
- `optimizers` : Implements the weights and biases updates. Each optimizer must implement the `update_parameters` method and at the moment, Stochastic Gradient Descent is supported.
- `models` : Implements different types of models in which all of the above building blocks come together and are executed in a specific way. Each type of model must implement the `train_step`, `fit`, `predict`, `val_step`, and `evaluate` methods. Currently sequential models are supported which consist of a stack of layers executed sequentially.

Tests

A lot of tests can be found in the `tests` folder which generally contains one module per package described above. The most interesting tests are contained in `tests/test_model.py` which takes a simple example neural network and tests that the forward pass produces the expected dendritic potentials, activations, losses and cost and that the backward pass produces the expected gradients. The expected results were computed manually and the actual results were computed using the above described algorithms for forward- and backward propagation.

The architecture the example network is as follows:

- Input layer: 3 neurons
- Hidden layer: 2 neurons with sigmoid activation function
- Output layer: 2 neurons with softmax activation function and Categorical Cross -ntropy loss

As the forward pass computations are relatively trivial and have already been explained in the Forward Propagation section, we will now show how the gradients of each layer are computed manually.

Since our example network has two layers with trainable parameters, i.e. the hidden and output layer, what we are most interested in are $\frac{\partial L}{\partial \mathbf{W}^1}$ and $\frac{\partial L}{\partial \mathbf{b}^1}$ as well as $\frac{\partial L}{\partial \mathbf{W}^2}$ and $\frac{\partial L}{\partial \mathbf{b}^2}$. Each of these terms can be expanded using the chain rule. We will start with the gradients in the hidden layer:

$$\frac{\partial L}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}\frac{\partial \mathbf{z}^2}{\partial \mathbf{W}^2}, \tag{95}$$

where

$$\frac{\partial L}{\partial \mathbf{a}^2} = \begin{bmatrix} \frac{\partial L}{\partial a_1^2} & \frac{\partial L}{\partial a_2^2} \end{bmatrix} = -\begin{bmatrix} \frac{y_1}{a_1^2} & \frac{y_2}{a_2^2} \end{bmatrix}, \tag{96}$$

(gradient of the categorical cross entropy loss)

$$\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \begin{bmatrix} \frac{\partial a_1^2}{\partial z_1^2} & \frac{\partial a_1^2}{\partial z_2^2} \\ \frac{\partial a_2^2}{\partial z_1^2} & \frac{\partial a_2^2}{\partial z_2^2} \end{bmatrix} = \begin{bmatrix} a_1^2(1-a_1^2) & -a_1^2 a_2^2 \\ -a_2^2 a_1^2 & a_2^2(1-a_2^2) \end{bmatrix}, \tag{97}$$

(Jacobian of the softmax activation function)

$$\frac{\partial \mathbf{z}^2}{\partial \mathbf{W}^2} = \begin{bmatrix} \frac{\partial z_1^2}{\partial w_{1,1}^2} & \frac{\partial z_1^2}{\partial w_{1,2}^2} & \frac{\partial z_1^2}{\partial w_{2,1}^2} & \frac{\partial z_1^2}{\partial w_{2,2}^2} \\ \frac{\partial z_2^2}{\partial w_{1,1}^2} & \frac{\partial z_2^2}{\partial w_{1,2}^2} & \frac{\partial z_2^2}{\partial w_{2,1}^2} & \frac{\partial z_2^2}{\partial w_{2,2}^2} \end{bmatrix} = \begin{bmatrix} a_1^1 & a_2^1 & 0 & 0 \\ 0 & 0 & a_1^1 & a_2^1 \end{bmatrix}, \tag{98}$$

(see equation (3))

which, after forward propagation, can be filled in with exact values. As a sanity check, note that (95) will yield a 1×4 row vector (or stacked as a 2×2 matrix if you will), i.e. it has 4 elements, just as many elements as $\mathbf{W}^2$ should have.

Having calculated $\frac{\partial L}{\partial \mathbf{W}^2}$, calculating $\frac{\partial L}{\partial \mathbf{b}^2}$ is straight forward, because we already know some intermediate quantities:

$$\frac{\partial L}{\partial \mathbf{b}^1} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}\frac{\partial \mathbf{z}^2}{\partial \mathbf{b}^2}, \tag{99}$$

where the new quantity

$$\frac{\partial \mathbf{z}^2}{\partial \mathbf{b}^2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{100}$$

(see equation (3)), which will yield a 1×2 row vector, as expected.

Now, let's move to the weights in the first layer:

$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}\frac{\partial \mathbf{z}^2}{\partial \mathbf{a}^1}\frac{\partial \mathbf{a}^1}{\partial \mathbf{z}^1}\frac{\partial \mathbf{z}^1}{\partial \mathbf{W}^1}, \tag{101}$$

where the new quantities are

$$\frac{\partial \mathbf{z}^2}{\partial \mathbf{a}^1} = \begin{bmatrix} \frac{\partial z_1^2}{\partial a_1^1} & \frac{\partial z_1^2}{\partial a_2^1} \\ \frac{\partial z_2^2}{\partial a_1^1} & \frac{\partial z_2^2}{\partial a_2^1} \end{bmatrix} = \begin{bmatrix} w_{1,1}^2 & w_{1,2}^2 \\ w_{2,1}^2 & w_{2,2}^2 \end{bmatrix}, \tag{102}$$

(see equation (3))

$$\frac{\partial \mathbf{a}^1}{\partial \mathbf{z}^1} = \begin{bmatrix} \frac{\partial a_1^1}{\partial z_1^1} & \frac{\partial a_1^1}{\partial z_2^1} \\ \frac{\partial a_2^1}{\partial z_1^1} & \frac{\partial a_2^1}{\partial z_2^1} \end{bmatrix} = \begin{bmatrix} a_1^1(1-a_1^1) & 0 \\ 0 & a_2^1(1-a_2^1) \end{bmatrix}, \tag{103}$$

(Jacobian of the sigmoid function)

$$\frac{\partial \mathbf{z}^1}{\partial \mathbf{W}^1} = \begin{bmatrix} \frac{\partial z_1^1}{\partial w_{1,1}^1} & \frac{\partial z_1^1}{\partial w_{1,2}^1} & \frac{\partial z_1^1}{\partial w_{1,3}^1} & \frac{\partial z_1^1}{\partial w_{2,1}^1} & \frac{\partial z_1^1}{\partial w_{2,2}^1} & \frac{\partial z_1^1}{\partial w_{2,3}^1} \\ \frac{\partial z_2^1}{\partial w_{1,1}^1} & \frac{\partial z_2^1}{\partial w_{1,2}^1} & \frac{\partial z_2^1}{\partial w_{1,3}^1} & \frac{\partial z_2^1}{\partial w_{2,1}^1} & \frac{\partial z_2^1}{\partial w_{2,2}^1} & \frac{\partial z_2^1}{\partial w_{2,3}^1} \end{bmatrix} = \begin{bmatrix} a_1^0 & a_2^0 & a_3^0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_1^0 & a_2^0 & a_3^0 \end{bmatrix} \tag{104}$$

(see equation (3)).

Finally, the computation of the bias gradients of layer 1 is very similar to the one of the weight gradients of layer 1, i.e.

$$\frac{\partial L}{\partial \mathbf{b}^1} = \frac{\partial L}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}\frac{\partial \mathbf{z}^2}{\partial \mathbf{a}^1}\frac{\partial \mathbf{a}^1}{\partial \mathbf{z}^1}\frac{\partial \mathbf{z}^1}{\partial \mathbf{b}^1}, \tag{105}$$

where the new quantity is

$$\frac{\partial \mathbf{z}^1}{\partial \mathbf{b}^1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{106}$$

(see equation (3)).

All of the above gradients can be computed after randomly initializing the weights and biases of the network and after providing some random input vector $\mathbf{a}^0$. To see the actual implementation of this test, I suggest to view the code in `tests/test_model.py`.

Empirical results

The implemented neural network was tested on the MNIST dataset containing hand-written digits from 0 to 9, which can be downloaded from here. The dataset contains $60,000$ training and $10,000$ testing images with a height and width of 28 pixels each. We tested a couple of different model architectures and noticed that, in our implementation, deep neural networks suffered from the vanishing gradient problem very quickly, so we tried to keep the model architecture fairly simple.

So, we chose a model with an input layer of $28 \times 28 = 784$ neurons (which was simply given by the image size), 32 neurons in the first hidden layer, 16 neurons in the second hidden layer and 10 neurons in the output layer. The two hidden layers both used the tanh activation function and the output layer used the softmax activation function while the weight updates were performed using Stochastic Gradient Descent. Finally, we compared the results of our model with the same network architecture implemented in Tensorflow.

Below you can see a summary of the results and the settings that we used:

Library	Epochs	Batch Size	Learning Rate	Categorical Cross-Entropy	Accuracy	Precision	Recall	Training Time	Processing Unit
Our own	5	32	0.1	0.166	0.990	0.950	0.950	17.95 min	CPU
Tensorflow	5	32	0.1	0.290	0.984	0.941	0.895	6.64 min	GPU

Note that all metric values of categorical cross-entropy, accuracy, precision and recall were computed on the test set. Also notice that accuracy, precision and recall were computed using *micro averaging* which means that - independently of class - we count all instances of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) and once that is done, accuracy, precision and recall are calculated as usual, i.e.:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{107}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{108}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{109}$$

1. For simplicity reasons we left out the layer and column indices of the weight matrices, because this has no influence on the point we want to make. ↩

2. Notice that some authors define the derivative of a scalar w.r.t. a vector as a column vector. No matter which notation is used, the results of one notation should be equal to the transposition of the results using the other notation. ↩

3. During one epoch, all training examples have been forward- and backward propagated through the network. Usually, neural networks will need many (50-100) of such epochs to accurately predict the target values. Notice, that during each epoch, S gradient descent update steps are performed. ↩