Neural Networks from Scratch

This repository aims to derive and implement equations for training neural networks from scratch, which consist of an arbitrary number of fully connected, dense layers. First, we will attempt to derive equations for forward- as well as backward propagation 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 it to a matrix-based approach for processing `batch_size` examples at once. After implementing the necessary Python code, we will test the network's performance on the MNIST hand-written digits dataset and compare its performance with famous deep learning libraries such as TensorFlow.

Table of contents

Forward Propagation

In this section, we will start by explaining the forward pass, i.e. forward propagation, of a single training example. We will depict the computations in component form and then continue to explain how to vectorize the forward propagation for a single training example. We will show how the network generates predictions and how it evaluates the goodness of its predictions using a specific loss function. Finally, we will show a little "trick" to compute the forward propagation for `batch_size` training examples at once in a vectorized form.

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 $i = 1, 2, 3$. This notation allows 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 *weighted input* z_i^1 and a corresponding output, i.e. activation, a_i^1 for $i = 1, 2$. For example,

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

$$a_1^1 = f(z_1^1) \tag{2}$$

or more generally,

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

$$a_i^l = f(z_i^l), \tag{4}$$

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, b_i^l represents a bias term, and where $f(\cdot)$ represents an *activation function* that is applied to the weighted input in order to produce the output, i.e. activation a_i^l, of neuron i in layer l. The weights and biases are initialized with random values [1] and represent the parameters of the network, i.e. the parameters which the network *learns* during the training phase. The activation function $f(\cdot)$ is some non-linear transformation whose output should resemble the *firing* rate of neuron i in layer l. A detailed discussion of activation functions will be presented later, but for now, we will just assume that our activation function is always the sigmoid function which is defined as

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

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 usually (CITATION), it is 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 [2].

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

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

where the activation function $f(\cdot)$ is applied element wise such as

$$\begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_{n^l}^l \end{bmatrix} = \begin{bmatrix} f(z_1^l) \\ f(z_2^l) \\ \vdots \\ f(z_{n^l}^l) \end{bmatrix}. \tag{12}$$

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, the activation function $f(\cdot)$ is just applied to each element of $\mathbf{z}^l$ to produce the activation vector $\mathbf{a}^l$.

We can now apply equations (8) and (10) 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 (7).

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 a classification problem involving n^L classes, a good [3] choice for a loss function is the *cross entropy* which is calculated as follows

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

where $\mathbf{y}$ is the ground truth vector where element $y_i = 1$ if the current training example belongs to class i and $y_i = 0$ otherwise.

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 4 edge cases:

- Bad predictions
  - If $y_i = 1$ and $\hat{y} = 0$, then $y_i log(\hat{y}_i) = -\infty$ and $(1-y_i)log(1-\hat{y}_i) = 0$, so $L(y_i, \hat{y}_i) = \infty$
  - If $y_i = 0$ and $\hat{y} = 1$, then $y_i log(\hat{y}_i) = 0$ and $(1-y_i)log(1-\hat{y}_i) = -\infty$, so $L(y_i, \hat{y}_i) = \infty$
- Good predictions
  - If $y_i = 1$ and $\hat{y} = 1$, then $y_i log(\hat{y}_i) = 0$ and $(1-y_i)log(1-\hat{y}_i) = 0$, so $L(y_i, \hat{y}_i) = 0$
  - If $y_i = 0$ and $\hat{y} = 0$, then $y_i log(\hat{y}_i) = 0$ and $(1-y_i)log(1-\hat{y}_i) = 0$, so $L(y_i, \hat{y}_i) = 0$

in all of the 4 above cases, we get the desired result. Also, we need the loss function to be differentiable in order to compute gradients during back propagation and it should be derivable using probability theory [4].

Forward Propagation for a Batch of Training Examples

We will conclude this section by showing how to compute the complete forward propagation for `batch_size` training examples at once. It starts by defining your input data matrix as follows

$$\mathbf{X} = \mathbf{A}^0, \tag{16}$$

where the feature vectors of each training examples are stacked horizontally next to each other in a column-wise fashion. Assuming that we have M training examples in our current batch and n^0 input features, equation (16) can be written out explicitly as

$$\begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^M \\ x_2^1 & x_2^2 & \dots & x_2^M \\ \vdots & \vdots & \ddots & \vdots \\ x_{n^0}^1 & x_{n^0}^2 & \dots & x_{n^0}^M \end{bmatrix} = \begin{bmatrix} a_1^{0,1} & a_1^{0,2} & \dots & a_1^{0,M} \\ a_2^{0,1} & a_2^{0,2} & \dots & a_2^{0,M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^0}^{0,1} & a_{n^0}^{0,2} & \dots & a_{n^0}^{0,M} \end{bmatrix}, \tag{17}$$

where $x_i^m = a_i^{0,m}$ both refer to the value of the i-th feature of the m-th training example.

Next, equation (9) becomes

$$\mathbf{Z}^l = \mathbf{W}^l\mathbf{A}^{l-1} + \mathbf{B}^l, \tag{18}$$

or written out explicitly

$$\begin{bmatrix} z_1^{l,1} & z_1^{l,2} & \dots & z_1^{l,M} \\ z_2^{l,1} & z_2^{l,2} & \dots & z_2^{l,M} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n^l}^{l,1} & z_{n^l}^{l,2} & \dots & z_{n^l}^{l,M} \end{bmatrix} = \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \dots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \dots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \dots & w_{n^l,n^{l-1}}^l \end{bmatrix} \begin{bmatrix} a_1^{l-1,1} & a_1^{l-1,2} & \dots & a_1^{l-1,M} \\ a_2^{l-1,1} & a_2^{l-1,2} & \dots & a_2^{l-1,M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^{l-1}}^{l-1,1} & a_{n^{l-1}}^{l-1,2} & \dots & a_{n^{l-1}}^{l-1,M} \end{bmatrix} + \begin{bmatrix} b_1^l & b_1^l & \dots & b_1^l \\ b_2^l & b_2^l & \dots & b_2^l \\ \vdots & \vdots & \ddots & \vdots \\ b_{n^l}^l & b_{n^l}^l & \dots & b_{n^l}^l \end{bmatrix}, \tag{19}$$

where $\mathbf{Z}^l$ simply contains M columns (one for each $\mathbf{z}^{l,m}$), $\mathbf{A}^{l-1}$ contains M columns (one for each $\mathbf{a}^{l-1}$), where $\mathbf{W}^l$ remains unchanged and where $\mathbf{B}^l$ needs to be repeated or *broadcasted* horizontally M times to make its addition conform. The dimensions of each component are as follows

- $\mathbf{Z}^l \rightarrow n^l \times M$
- $\mathbf{W}^l \rightarrow n^l \times n^{l-1}$
- $\mathbf{A}^{l-1} \rightarrow n^{l-1} \times M$
- $\mathbf{W}^l\mathbf{A}^{l-1} \rightarrow n^l \times M$
- $\mathbf{B}^l \rightarrow n^l \times M$

so the dimensions are conform.

Then, the activation function is applied element-wise as usual

$$\mathbf{A}^l = f(\mathbf{Z}^l), \tag{20}$$

which is equal to

$$\begin{bmatrix} a_1^{l,1} & a_1^{l,2} & \dots & a_1^{l,M} \\ a_2^{l,1} & a_2^{l,2} & \dots & a_2^{l,M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^l}^{l,1} & a_{n^l}^{l,2} & \dots & a_{n^l}^{l,M} \end{bmatrix} = \begin{bmatrix} f(z_1^{l,1}) & f(z_1^{l,2}) & \dots & f(z_1^{l,M}) \\ f(z_2^{l,1}) & f(z_2^{l,2}) & \dots & f(z_2^{l,M}) \\ \vdots & \vdots & \ddots & \vdots \\ f(z_{n^l}^{l,1}) & f(z_{n^l}^{l,2}) & \dots & f(z_{n^l}^{l,M}) \end{bmatrix}. \tag{21}$$

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

$$L(\mathbf{Y}, \hat{\mathbf{Y}}) = \begin{bmatrix} L(\mathbf{y}^1, \hat{\mathbf{y}}^1) & L(\mathbf{y}^2, \hat{\mathbf{y}}^2) & \dots & L(\mathbf{y}^M, \hat{\mathbf{y}}^M) \end{bmatrix}, \tag{22}$$

where each element $m = 1, 2, \dots, M$ of the above loss vector represents the loss we have already defined in equation (15), i.e.

$$L(\mathbf{y}^m, \hat{\mathbf{y}}^m) = -\sum_{i=1}^{n^L} y_i^m log(\hat{y}_i^m) + (1 - y_i^m) log(1 - \hat{y}_i^m). \tag{23}$$

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

$$C = \frac{1}{M}\sum_{m=1}^{M} L(\mathbf{y}^m, \hat{\mathbf{y}}^m) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{i=1}^{n^L} y_i^m log(\hat{y}_i^m) + (1 - y_i^m) log(1 - \hat{y}_i^m), \tag{24}$$

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, because the average is independent of the `batch_size`. Also, the cost function may include a regularization term, which should be monotonically increasing in the number of parameters of the model, to account for overfitting.

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*. The next sections will start by describing the backward propagation for a single training example and then extend this procedure for `batch_size` training examples at once.

Backpropagation for a Single Training Example

The backpropagation algorithm works as follows. For any given layer l, the backpropagation algorithm computes an intermediate quantity, the so called *error* $\boldsymbol{\delta}^l$, and then computes the gradients 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:

- BP1.1: An equation for the error at the output layer, i.e. $\boldsymbol{\delta}^L$. Needed for initializing the backpropagation algorithm
- BP2.1: A recursive equation relating $\boldsymbol{\delta}^l$ to $\boldsymbol{\delta}^{l+1}$. Note the in the first iteration, $\boldsymbol{\delta}^{l+1} = \boldsymbol{\delta}^L$ which we computed in BP1. Needed for recursively calculating the error at each layer
- BP3.1: An equation relating $\boldsymbol{\delta}^l$ to the derivative of the cost function w.r.t. the weights in layer l, i.e. $\frac{\partial C}{\partial \mathbf{W}^l}$.
- BP4: An equation relating $\boldsymbol{\delta}^l$ to the derivative of the cost function w.r.t. the biases in layer l, i.e. $\frac{\partial C}{\partial \mathbf{b}^l}$.

Backpropagation for a Single Training Example

BP1.1

The error at the output layer $\boldsymbol{\delta}^L$, which represents $\frac{\partial L}{\partial \mathbf{z}^L}$, can be expressed as follows

$$\boldsymbol{\delta}^L := \frac{\partial L}{\partial \mathbf{z}^L} = \frac{\partial L}{\partial \mathbf{a}^L}\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}. \tag{25}$$

Remember that the derivative of a function yielding a scalar, e.g. the loss function, w.r.t. a vector is defined as a row vector and that the derivative of a function yielding a vector w.r.t. another vector is defined as a matrix, i.e. the *Jacobi* matrix. So, working out the above equation will produce the following result

$$\boldsymbol{\delta}^L = \begin{bmatrix} \frac{\partial L}{\partial a_1^L}, & \frac{\partial L}{\partial a_2^L}, & \dots, & \frac{\partial L}{\partial a_{n^L}^L} \end{bmatrix} \begin{bmatrix} \frac{\partial a_1^L}{\partial z_1^L}, & \frac{\partial a_1^L}{\partial z_2^L}, & \dots, & \frac{\partial a_1^L}{\partial z_{n^L}^L} \\ \frac{\partial a_2^L}{\partial z_1^L}, & \frac{\partial a_2^L}{\partial z_2^L}, & \dots, & \frac{\partial a_2^L}{\partial z_{n^L}^L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_{n^L}^L}{\partial z_1^L}, & \frac{\partial a_{n^L}^L}{\partial z_2^L}, & \dots, & \frac{\partial a_{n^L}^L}{\partial z_{n^L}^L} \end{bmatrix} \tag{26}$$

which we can multiply out as follows

$$\boldsymbol{\delta}^L = \left[\frac{\partial L}{\partial a_1^L}\frac{\partial a_1^L}{\partial z_1^L} + \frac{\partial L}{\partial a_2^L}\frac{\partial a_2^L}{\partial z_1^L} + \ldots + \frac{\partial L}{\partial a_{n^L}^L}\frac{\partial a_{n^L}^L}{\partial z_1^L}, \quad \frac{\partial L}{\partial a_1^L}\frac{\partial a_1^L}{\partial z_2^L} + \frac{\partial L}{\partial a_2^L}\frac{\partial a_2^L}{\partial z_2^L} + \ldots + \frac{\partial L}{\partial a_{n^L}^L}\frac{\partial a_{n^L}^L}{\partial z_2^L}, \quad \dots, \quad \frac{\partial L}{\partial a_1^L}\frac{\partial a_1^L}{\partial z_{n^L}^L} + \frac{\partial L}{\partial a_2^L}\frac{\partial a_2^L}{\partial z_{n^L}^L} + \ldots + \frac{\partial L}{\partial a_{n^L}^L}\frac{\partial a_{n^L}^L}{\partial z_{n^L}^L} \right] \quad \text{(}$$

and which can be simplified to

$$\boldsymbol{\delta}^L = \left[\sum_{j=1}^{n^L} \frac{\partial L}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_1^L}, \quad \sum_{j=1}^{n^L} \frac{\partial L}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_2^L}, \quad \dots, \quad \sum_{j=1}^{n^L} \frac{\partial L}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_{n^L}^L} \right]. \tag{28}$$

Notice that a_j^L is only a function of (i.e. it only depends on) z_j^L, so $\frac{\partial a_j^L}{\partial z_k^L} = 0$ if $j \neq k$. We can use that to simplify the above expression to

$$\boldsymbol{\delta}^L = \begin{bmatrix} \frac{\partial L}{\partial a_1^L}\frac{\partial a_1^L}{\partial z_1^L}, & \frac{\partial L}{\partial a_2^L}\frac{\partial a_2^L}{\partial z_2^L}, & \dots, & \frac{\partial L}{\partial a_{n^L}^L}\frac{\partial a_{n^L}^L}{\partial z_{n^L}^L} \end{bmatrix} \tag{29}$$

Finally, remember that $a_j^L = f(z_j^L)$, so $\frac{\partial a_j^L}{\partial z_j^L} = f'(z_j^L)$, so the above expression may be rewritten as

$$\boldsymbol{\delta}^L = \begin{bmatrix} \frac{\partial L}{\partial a_1^L} f'(z_1^L), & \frac{\partial L}{\partial a_2^L} f'(z_2^L), & \dots, & \frac{\partial L}{\partial a_{n^L}^L} f'(z_{n^L}^L) \end{bmatrix}. \tag{30}$$

When implementing everything in code, it will come in handy to keep every training example in a separate row, so we will re-define the result of the above equation as its transposition

$$\boldsymbol{\delta}^L := \begin{bmatrix} \frac{\partial L}{\partial a_1^L} f'(z_1^L) \\ \frac{\partial L}{\partial a_2^L} f'(z_2^L) \\ \vdots \\ \frac{\partial L}{\partial a_{n^L}^L} f'(z_{n^L}^L) \end{bmatrix}. \tag{31}$$

This is the equation for the error at the output layer, i.e. BP1.1.

Concrete Example

While BP1 will hold for any cost and any activation function, we will now provide a concrete example if the cost function is the categorical cross entropy and the activation function is the sigmoid function.

From the categorical cross entropy loss function in equation (15), we can see that for $i = 1, 2, \ldots, n^L$,

$$\frac{\partial L}{\partial a_i^L} = -\left(\frac{y_i}{a_i^L} - \frac{1-y_i}{1-a_i^L}\right), \tag{32}$$

where we used the definition $\hat{y}_i := a_i^L$. We can now do a little algebra to simplify the above expression. First, we want to create a common denominator which we can achieve as follows

$$\frac{\partial L}{\partial a_i^L} = -\left(\frac{y_i}{a_i^L}\frac{1-a_i^L}{1-a_i^L} - \frac{1-y_i}{1-a_i^L}\frac{a_i^L}{a_i^L}\right) = -\left(\frac{y_i - a_i^L}{a^L(1-a_i^L)}\right). \tag{33}$$

Notice that $a_i^L = f(z_i^L)$ and that in the case of the sigmoid function, $f'(z_i^L) = f(z_i^L)(1 - f(z_i^L)) = a_i^L(1-a_i^L)$. We can use that, to simplify the above expression to

$$\frac{\partial L}{\partial a_i^L} = -\left(\frac{y_i - a_i^L}{f'(z_i^L)}\right). \tag{34}$$

Now finally, we plug this interim result back into each element of BP1.1 while noticing that the two $f'(z_i^L)$ terms will cancel out, yielding

$$\boldsymbol{\delta}^L = \left[-\left(y_1 - a_1^L\right), \quad -\left(y_2 - a_2^L\right), \quad \ldots, \quad -\left(y_{n^L} - a_{n^L}^L\right)\right] = -\left[\left(y_1 - a_1^L\right), \quad \left(y_2 - a_2^L\right), \quad \ldots, \quad \left(y_{n^L} - a_{n^L}^L\right)\right]. \tag{35}$$

The above expression has the particularly nice property that $\boldsymbol{\delta}^L$ is not dependent on $f'(z_i^L)$, which in case of the sigmoid function, may have caused a *learning slowdown*, because the derivative of the sigmoid function is very small for large inputs.

BP2.1

In order to represent the error of the previous layer $\boldsymbol{\delta}^{l-1}$ in terms of the error in the current layer $\boldsymbol{\delta}^l$, it helps to view the loss function as a nested functions 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

$$\boldsymbol{\delta}^{l-1} := \frac{\partial L}{\partial \mathbf{z}^{l-1}} = \frac{\partial L}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \mathbf{z}^{l-1}}. \tag{36}$$

Remembering from above that the derivative of a function yielding a scalar is defined as a row vector and that the derivative of a vector w.r.t. another vector is defined as the Jacobian matrix, we can rewrite the above expression as

$$\boldsymbol{\delta}^{l-1} = \left[\frac{\partial L}{\partial z_1^l}, \quad \frac{\partial L}{\partial z_2^l}, \quad \ldots, \quad \frac{\partial L}{\partial z_{n^l}^l}\right] \begin{bmatrix} \frac{\partial z_1^l}{\partial z_1^{l-1}}, & \frac{\partial z_1^l}{\partial z_2^{l-1}}, & \ldots, & \frac{\partial z_1^l}{\partial z_{n^{l-1}}^{l-1}} \\ \frac{\partial z_2^l}{\partial z_1^{l-1}}, & \frac{\partial z_2^l}{\partial z_2^{l-1}}, & \ldots, & \frac{\partial z_2^l}{\partial z_{n^{l-1}}^{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n^l}^l}{\partial z_1^{l-1}}, & \frac{\partial z_{n^l}^l}{\partial z_2^{l-1}}, & \ldots, & \frac{\partial z_{n^l}^l}{\partial z_{n^{l-1}}^{l-1}} \end{bmatrix}. \tag{37}$$

Multiplying out the above equation yields the $1 \times n^{l-1}$ row vector

$$\boldsymbol{\delta}^{l-1} = \left[\frac{\partial L}{\partial z_1^l}\frac{\partial z_1^l}{\partial z_1^{l-1}} + \frac{\partial L}{\partial z_2^l}\frac{\partial z_2^l}{\partial z_1^{l-1}} + \ldots + \frac{\partial L}{\partial z_{n^l}^l}\frac{\partial z_{n^l}^l}{\partial z_1^{l-1}}, \quad \frac{\partial L}{\partial z_1^l}\frac{\partial z_1^l}{\partial z_2^{l-1}} + \frac{\partial L}{\partial z_2^l}\frac{\partial z_2^l}{\partial z_2^{l-1}} + \ldots + \frac{\partial L}{\partial z_{n^l}^l}\frac{\partial z_{n^l}^l}{\partial z_2^{l-1}}, \quad \frac{\partial L}{\partial z_1^l}\frac{\partial z_1^l}{\partial z_{n^{l-1}}^{l-1}} + \frac{\partial L}{\partial z_2^l}\frac{\partial z_2^l}{\partial z_{n^{l-1}}^{l-1}} + \ldots + \frac{\partial L}{\partial z_{n^l}^l}\frac{\partial z_{n^l}^l}{\partial z_{n^{l-1}}^{l-1}}\right], \quad ($$

which can be simplified to

$$\boldsymbol{\delta}^{l-1} = \left[\sum_{j=1}^{n^l}\frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial z_1^{l-1}}, \quad \sum_{j=1}^{n^l}\frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial z_2^{l-1}}, \quad \ldots, \quad \sum_{j=1}^{n^l}\frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial z_{n^{l-1}}^{l-1}}\right]. \tag{39}$$

Recall that $\frac{\partial L}{\partial z_j^l} := \delta_j^l$, so we can simplify the above equation to

$$\boldsymbol{\delta}^{l-1} = \left[\sum_{j=1}^{n^l}\delta_j^l\frac{\partial z_j^l}{\partial z_1^{l-1}}, \quad \sum_{j=1}^{n^l}\delta_j^l\frac{\partial z_j^l}{\partial z_2^{l-1}}, \quad \ldots, \quad \sum_{j=1}^{n^l}\delta_j^l\frac{\partial z_j^l}{\partial z_{n^{l-1}}^{l-1}}\right]. \tag{40}$$

Notice that

$$z_j^l = \sum_{k=1}^{n^{l-1}}\left(w_{j,k}^l\, a_k^{l-1}\right) + b_j^l = \sum_{k=1}^{n^{l-1}}\left(w_{j,k}^l\; f(z_k^{l-1})\right) + b_j^l \tag{41}$$

and therfore,

$$\frac{\partial z_j^l}{\partial z_k^{l-1}} = w_{j,k}^l\; f'(z_k^{l-1}). \tag{42}$$

Plugging (41) back into each element of (39) and rearranging a little bit yields

$$\boldsymbol{\delta}^{l-1} = \Big[f'(z_1^{l-1}) \textstyle\sum_{j=1}^{n^l} w_{j,1}^l \, \delta_j^l, \quad f'(z_2^{l-1}) \sum_{j=1}^{n^l} w_{j,2}^l \, \delta_j^l, \quad \ldots, \quad f'(z_{n^{l-1}}^{l-1}) \sum_{j=1}^{n^l} w_{j,n^{l-1}}^l \, \delta_j^l\Big], \tag{43}$$

which can be split into the element-by-element vector multiplication

$$\boldsymbol{\delta}^{l-1} = \big[f'(z_1^{l-1}), \quad f'(z_2^{l-1}), \quad \ldots, \quad f'(z_{n^{l-1}}^{l-1})\big] \odot \Big[\textstyle\sum_{j=1}^{n^l} w_{j,1}^l \, \delta_j^l, \quad \sum_{j=1}^{n^l} w_{j,2}^l \, \delta_j^l, \quad \ldots, \quad \sum_{j=1}^{n^l} w_{j,n^{l-1}}^l \, \delta_j^l\Big]. \tag{44}$$

Finally, we want to vectorize the sum in the RHS of the above equation, so that the actual implementation in code can harness optimized matrix multiplication libraries. This can be achieved as follows

$$\boldsymbol{\delta}^{l-1} = \big[f'(z_1^{l-1}), \quad f'(z_2^{l-1}), \quad \ldots, \quad f'(z_{n^{l-1}}^{l-1})\big] \odot \big[\delta_1^l, \quad \delta_2^l, \quad \ldots, \delta_{n^l}^l\big] \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \cdots & w_{1,n^{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \cdots & w_{2,n^{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^l,1}^l & w_{n^l,2}^l & \cdots & w_{n^l,n^{l-1}}^l \end{bmatrix}, \tag{45}$$

or written more compactly

$$\boldsymbol{\delta}^{l-1} = \big(f('\mathbf{z}^{l-1})\big)^T \odot (\boldsymbol{\delta}^l)^T \mathbf{W}^l. \tag{46}$$

Again, later on, we want to keep every training example in a separate row, so we will redefine the above result to its transposition

$$\boldsymbol{\delta}^{l-1} := \big(f('\mathbf{z}^{l-1})\big) \odot (\mathbf{W}^l)^T \boldsymbol{\delta}^l. \tag{47}$$

Equation (46) is a representation of the error in the previous layer in terms of the error in the current layer, i.e. BP2.1.

BP3.1

After calculating the errors for each layer, we now want to relate them to the derivative of the loss w.r.t the weights (and later the biases as well). To find an expression for the derivative of the loss w.r.t. the weights in layer l, it might help to view weight matrix $\mathbf{W}^l$ as a long, flattened vector $\mathbf{w}^l$, whose, (i,j)-th element represents (or came from) the weight in the i-th row and j-th column of $\mathbf{W}^l$, such that $\mathbf{W}^l[i,j] = \mathbf{w}^l[(i,j)]$.

With that in mind, we can express the derivative of the loss w.r.t the weights in layer l as

$$\frac{\partial L}{\partial \mathbf{w}^l} = \frac{\partial L}{\partial \mathbf{z}^l} \frac{\partial \mathbf{z}^l}{\partial \mathbf{w}^l}. \tag{48}$$

We flattened the weight matrix into a long vector, because now, it is obvious that the term $\frac{\partial \mathbf{z}^l}{\partial \mathbf{w}^l}$, i.e. the derivative of a vector w.r.t another vector, is defined as the Jacobi matrix. Writing out the above equation explicitly yields

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}^l} &= \left[\frac{\partial L}{\partial z_1^l}, \quad \frac{\partial L}{\partial z_2^l}, \quad \ldots, \quad \frac{\partial L}{\partial z_{n^l}^l}\right] \\ &\begin{bmatrix} \frac{\partial z_1^l}{\partial w_{1,1}^l}, & \frac{\partial z_1^l}{\partial w_{1,2}^l}, & \ldots, & \frac{\partial z_1^l}{\partial w_{1,n^{l-1}}^l}, & \frac{\partial z_1^l}{\partial w_{2,1}^l}, & \frac{\partial z_1^l}{\partial w_{2,2}^l}, & \ldots, & \frac{\partial z_1^l}{\partial w_{2,n^{l-1}}^l}, & \ldots, & \frac{\partial z_1^l}{\partial w_{n^l,1}^l}, & \frac{\partial z_1^l}{\partial w_{n^l,2}^l}, & \ldots, & \frac{\partial z_1^l}{\partial w_{n^l,n^{l-1}}^l} \\ \frac{\partial z_2^l}{\partial w_{1,1}^l}, & \frac{\partial z_2^l}{\partial w_{1,2}^l}, & \ldots, & \frac{\partial z_2^l}{\partial w_{1,n^{l-1}}^l}, & \frac{\partial z_2^l}{\partial w_{2,1}^l}, & \frac{\partial z_2^l}{\partial w_{2,2}^l}, & \ldots, & \frac{\partial z_2^l}{\partial w_{2,n^{l-1}}^l}, & \ldots, & \frac{\partial z_2^l}{\partial w_{n^l,1}^l}, & \frac{\partial z_2^l}{\partial w_{n^l,2}^l}, & \ldots, & \frac{\partial z_2^l}{\partial w_{n^l,n^{l-1}}^l} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n^l}^l}{\partial w_{1,1}^l}, & \frac{\partial z_{n^l}^l}{\partial w_{1,2}^l}, & \ldots, & \frac{\partial z_{n^l}^l}{\partial w_{1,n^{l-1}}^l}, & \frac{\partial z_{n^l}^l}{\partial w_{2,1}^l}, & \frac{\partial z_{n^l}^l}{\partial w_{2,2}^l}, & \ldots, & \frac{\partial z_{n^l}^l}{\partial w_{2,n^{l-1}}^l}, & \ldots, & \frac{\partial z_{n^l}^l}{\partial w_{n^l,1}^l}, & \frac{\partial z_{n^l}^l}{\partial w_{n^l,2}^l}, & \ldots, & \frac{\partial z_{n^l}^l}{\partial w_{n^l,n^{l-1}}^l} \end{bmatrix}. \end{aligned} \tag{49}$$

BP4.1

Backpropagation for batch_size Training Examples at once

BP1.2

BP2.2

BP3.2

BP4.2

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

$$\frac{\partial L}{\partial w_j} = \frac{L(\mathbf{w} + \epsilon \mathbf{e}_j, \mathbf{b}) - L(\mathbf{w}, \mathbf{b})}{\epsilon}, \tag{50}$$

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!

Gradient Descent

TODO

Loss and Activation Functions

Popular choices of activation functions in the hidden layers are the sigmoid (equation 3), ReLU (equation 4) and tanh (equation 5) functions. These functions and their corresponding derivatives are presented below

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{51}$$

TODO:

- Show equations for the activation function and their corresponding derivatives
- Show graphs of each activation function and their corresponding derivatives
- Discuss properties of the activation functions in the output layer

Ideally, $f(z)$ should be differentiable, non-linear, monotonically increasing and non-saturating. It should be differentiable, because in the back-propagation algorithm, we need to compute its derivative and it should be non-linear, because otherwise there is no benefit of introducing hidden layers. The latter follows from the fact that a chain of deeply nested linear transformations can be rewritten as merely another linear transformation (CITATION Studienbrief 3). Furthermore, it should be monotonically increasing so that as few as possible local minima are generated (GENERATED WHERE?) and it should be non-saturating to avoid the vanishing gradient problem. The vanishing gradient problem occurs when the norm of the gradient [2] is very small which happens when z is very large and $f'(z)$ is very small. Of course, the monotonically increasing and non-saturating properties could easily be achieved by setting $f(z)$ to the identity function, but then $f(z)$ would not be non-linear anymore, so there are some tradeoffs to be made and in practice, the ReLU is a good compromise between these tradeoffs. Note however, that the ReLU function is actually not defined at $z = 0$ so theoretically, it is not always differentiable. In practice however, the probability that z is *exactly* 0 is extremely small so that in code, it is often implemented that $f'(0) = 0$.

In the output layer, the activation may also be linear and it depends whether we're doing regression or classification (multi-class or multi-label)...

Implementation in Code

[1] Actually, only the weights must be initialized randomly and the biases may be initialized with zeros, because only the weights suffer from the *symmetry breaking problem* (CITATION) ↩

[2] Precision is the ratio of true positives divided by the sum of true positives and false positives while recall is the ratio of true positives divided by the sum of true positives and false negatives. ↩ ↩

[3] "good" in the sense that, unlike e.g. the squared error, it avoids a learning slowdown, because its gradient with respect to (w.r.t.) the weights and biases is independent of the derivative of the activation function in the output layer CITATION: http://neuralnetworksanddeeplearning.com/chap3.html#introducing_the_cross-entropy_cost_function ↩

[4] E.g. the loss function in linear regression, the squared error, is derivable by using the Maximum Likelihood theory assuming that the distribution of the training examples follows a Gaussian distribution (CITATION) ↩