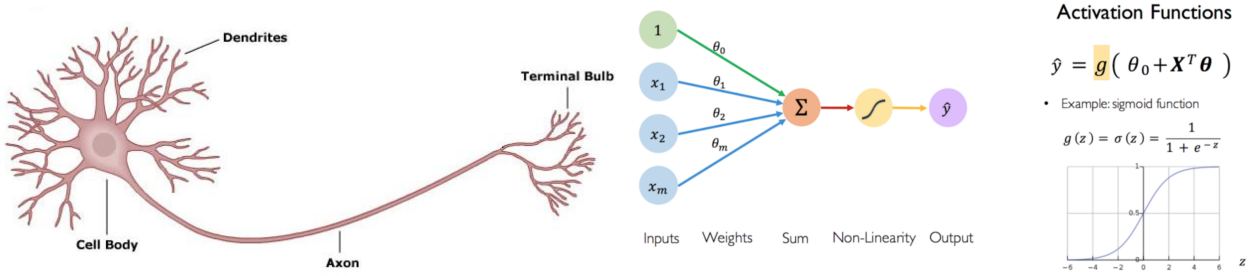
Multi-layer perceptron

1. The model

Neural networks are used to solve **supervised machine learning problems**. Here, we use the convention of having data as a collection of pairs $(\mathbf{x}, y)$ where the $\mathbf{x} \in \mathbb{R}^d$ is a vector characterizing input objects, with y being its associated target value. Neural networks can be used for classification or regression problems.[1]

In this document, we start describing a single artificial neuron, then we show how to put them in layers and finally how to build multi-layered network.

1.1 Artificial neuron



We start by looking at a single neuron. On the right, you see a computational device, known as an artificial neuron, inspired in its origin by biological brain neurons (on the left)[2].

The artificial neuron takes a vector of input values $x_1, .., x_d$ and combines it with some **weights** that are local to the neuron $(w_0, w_1, .., w_d)$ to compute a **net input** $w_0 + \sum_{i=1}^{d} w_i * x_i$. To compute its output, it then passes the net input through a possibly non-linear univariate activation function $g(\cdot)$. We will see in a bit common choices for activation function g. Note the special weight w_0 also known as **bias**. In order to treat all weights equally, we use the trick of creating an extra input variable x_0 with value always equal to 1, and so the function computed by a single artificial neuron (parameterized by its weights $\mathbf{w}$) is:

$$y(\mathbf{x}) = g\left(w_0 + \sum_{i=1}^{d} w_i\, x_i\right) = g\left(\sum_{i=0}^{d} w_i\, x_i\right) = g(\mathbf{w}^{\mathbf{T}}\mathbf{x})$$

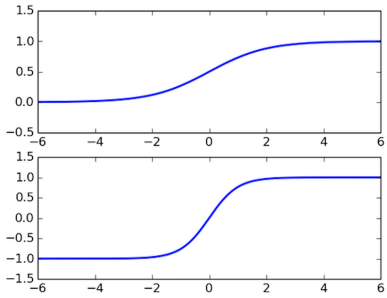
So, a single artificial neuron is used to model functions $\mathbb{R}^d \mapsto \mathbb{R}$. The weights $w_0, w_1, .., w_d$ are the **parameters** of the model. Different values will compute different functions. As usual, the learning task is to find suitable weights that allow neurons to model the data as accurately as possible.

1.2 Activation functions

Activation functions can be seen as a sort of *non-linear filters* that are applied to the *net inputs* of the neuron, that is, to the weighted linear combination of neuron inputs. Common choices are **sigmoidal functions**,

[1] We will also mention in a future session *autoencoders* which perform dimensionality reduction, a form of unsupervised learning.
[2] Note that in the picture the *weights* are depicted by greek letter θ however we use the convention of denoting weights with w

which are differentiable (and this is important for learning as we will see in future lectures), and have horizontal asymptotes in $\pm\infty$. Other choices such as the rectified linear function have become popular in the last years with the boom of deep learning.

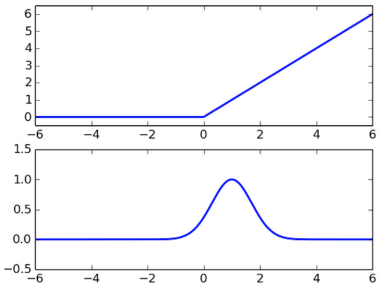


Sigmoid

$$\phi(z) = \frac{1}{1+e^{-z}}$$

Hyperbolic Tangent

$$\phi(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$



Rectified Linear

$$\phi(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

Radial Basis Function

$$\phi(z, c) = e^{-(\epsilon\|z-c\|)^2}$$

The most popular choice (at least in the context of classical multi-layer perceptrons) is the **sigmoid** function. In this case, **an artificial neuron is basically equivalent to a logistic regression model**. In fact, if we use the identity as activation function we end up with a linear regression model. In a way, an artificial neuron implements a generalized linear model.

1.3 One layer (stacking neurons *in parallel*)

We now generalize one artificial neuron in a straightforward way. What if we want to model functions with multivariate outputs? Well, we will just use a single neuron for each component in our output. Hence, we obtain *a layer* of neurons all working on the same input but each with **its own set of separate weights**.

Technically, a *layer* consists of a set of neurons sharing a common input vector but are not connected to each other. Typically, neurons in the same layer all use the same activation function.

So if our data consists now of pairs $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}^m$, we are going to use m neurons, each with its own set of weights $\mathbf{w}_k$ for $k = 1, .., m$. Remember, we follow the convention that $x_0 = 1$. And neurons will compute for each $k = 1, .., m$:

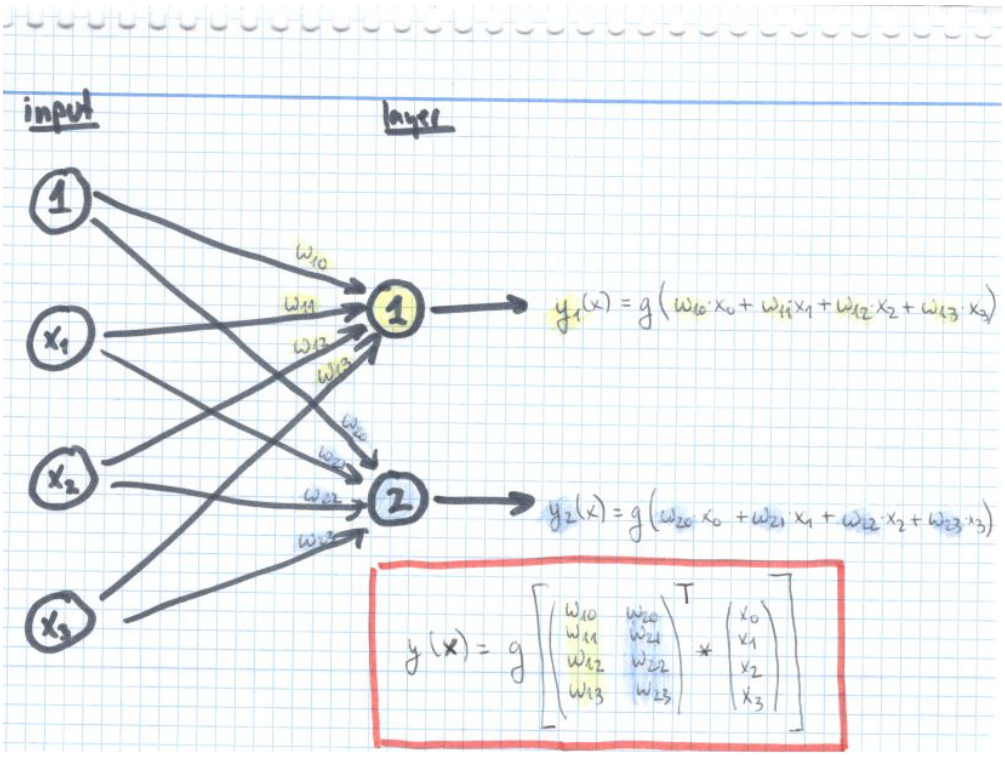
$$y_k(\mathbf{x}) = g\left(\sum_{i=0}^{d} w_{ki}\, x_i\right) = g(\mathbf{w}_k^T \mathbf{x})$$

We extend the notation for activation function $g[\cdot]$ to denote that g is applied to each component. For example, $g\left[\begin{pmatrix}5\\2\end{pmatrix}\right] = \begin{pmatrix}g(5)\\g(2)\end{pmatrix}$. With this notation, we can describe a **layer** of neurons in compact form using matrix notation as follows:

$$\mathbf{y}(\mathbf{x}) = g\left[W^T x\right]$$

where W is the matrix of size $(d+1) \times m$ containing the weights of all the neurons in the layer.

The following is an example with a 2-neuron layer on a three-dimensional input vector:



Notice that this continues to be a (generalized) linear model.

1.4 Multinomial logistic regression

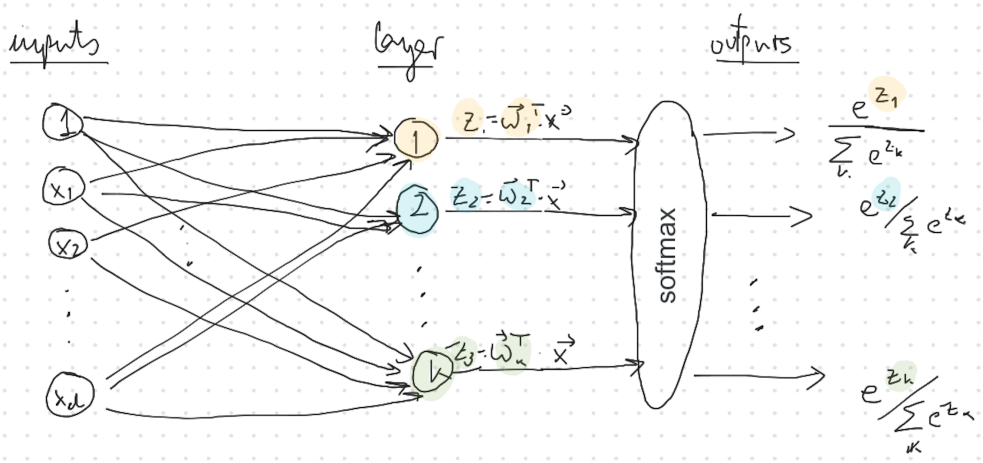
In the context of a multiclass classification problem with $K > 2$ classes, a popular model is multinomial logistic regression which we can achieve by having one neuron for each class and using the outputs from each neuron as *scores* that tell us how likely it is that the example belongs to each class.

In this scenario of multiclass classification it is very common to apply the softmax function in order to obtain normalized probabilities accross each class.

The softmax function applied to input $\mathbf{z} = (z_1, \ldots, z_K)$ is defined as:

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

The use of the softmax makes it unnecessary to use any activation function after each linear combination $\mathbf{w}_k^T \mathbf{x}$.



So the function computed by this single-layered network is:

$$\mathbf{y}(\mathbf{x}) = \text{softmax}(W^T\mathbf{x})$$

with $y_k(\mathbf{x}) = \frac{\exp\{\mathbf{w}_k^T\mathbf{x}\}}{\sum_{k'}\exp\{\mathbf{w}_{k'}^T\mathbf{x}\}}$.

1.5 Multi-layer perceptrons (putting layers together *in sequence*)

So let us make our so-far linear model a little more powerful. In the context of linear regression, we already did this by replacing inputs $\mathbf{x}$ by non-linear functions of the inputs which we called *basis functions* and denoted them with $\phi_i(\mathbf{x})$. We saw, for example, that by using polinomials over the input variables as basis functions, we could go beyond linear modelling.

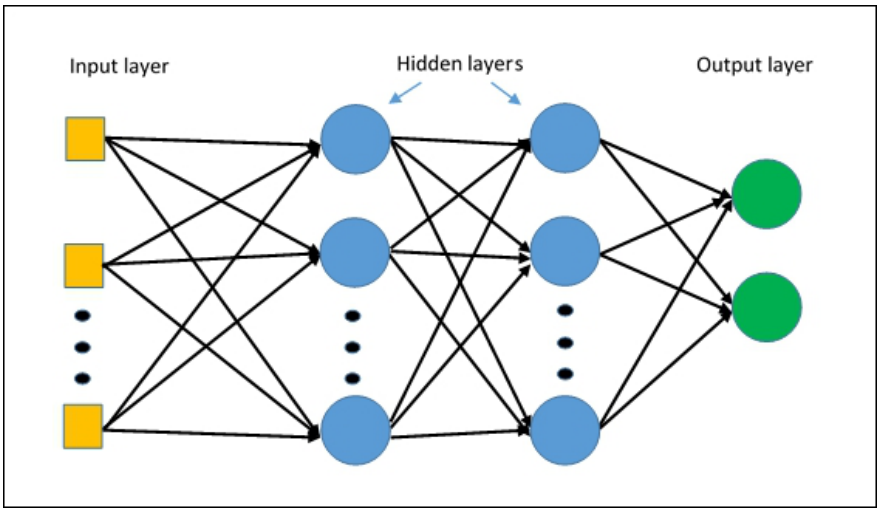
So let us do the same, but instead of using a predefined set of basis functions (which could be seen as a sort of fixed pre-processing of the data), we are going to use neurons as basis functions themselves. Their paramters will need to be adjusted as well, so in a way, we are letting the network do the pre-processing or **feature engineering** in a data-driven way, as part of the learning process. So, we are going to have several layers of neurons, each feeding their outputs as inputs to the next layers' neurons.

If we focus on one neuron in some layer, let us say neuron k, its output is now given by:

$$y_k(\mathbf{x}) = g\left(\sum_{i=0}^{h} w_{ki}\,\phi_i(x)\right)$$

where $\phi_i(\mathbf{x})$ is the output of the i'th neuron in the previous layer. Note also that the sum goes from $i = 0$ up to $i = h$, where h is now the size of the previous layer of neurons (as usual, we fix the first $\phi_0(\mathbf{x}) = 1$)).

So, MLP networks look like this:



In order to build a multi-layer network we are going to place layers in sequence, one layer's outputs being fed as inputs to the next one. The terminology regarding this is: the first (left-most) layer is the **input layer**, then the in-between layers come, known as **hidden layers**; finally the right-most layer is the **output layer**.

Of course, other architectures exist. In general, neural networks are directed graphs whose nodes are neurons. If the network has no cycles, then it is called a *feed-forward* network. Otherwise, it is called a *recurrent* network. Feed-forward networks represent functions, however recurrent networks represent dynamic systems and are therefore much more complex to study. MLPs are a special case of feed-forward networks, where the three following conditions hold:

- neurons are arranged in layers,
- there is at least one hidden layer,
- every layer is fully-connected to the next one, and
- no connections are allowed within layers.

The output layers' neurons are building a linear model on top of complex non-linear functions of the original input vector $\mathbf{x}$. This gives neural networks incredible flexibility and modelling strength[3]. By putting several neurons *in sequence*, we are injecting non-linearity into the model (as long as we use non-linear activation functions!).

1.6 Error functions

Regression. Let us start with a typical **regression** problem, having as input a collection of pairs $\{(\mathbf{x}_n, y_n)\}_{n=1,..,N}$ with $\mathbf{x}_n \in \mathbb{R}^d$ and $y_n \in \mathbb{R}$. As in ordinary least squares regression, we are going to penalize networks using *empirical mean square error*. In the following, we denote with $\mathbf{w}$ all the parameters used in the network we are evaluating. As you know, the weights used are grouped into one matrix for each layer, however we simplify the notation here and we put all weights from all layers into a vector $\mathbf{w}$. If we want to do regression with one single target variable, then our network will have one single neuron in its output layer, giving the prediction for a given input which we denote with $\hat{y}_{\mathbf{w}}(\mathbf{x})$ (of course, it could have many hidden layers with many neurons each). The error of the network is computed as usual:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y_n - \hat{y}_{\mathbf{w}}(\mathbf{x}_n))^2$$

and so the problem of learning will amount to chosing weights that minimize this function. Note that we could also minimize regularized versions of this error function, and this tends to help with overfitting, which is a serious problem given the flexibility of neural networks.

[3]This flexibility comes at a cost, as we will see, training neural networks is hard and the process can get stuck at bad solutions.

Binary classification. Assume now that the target values are of the form $y_n \in \{0, 1\}$ such that if $y_n = 1$, then $\mathbf{x}$ is classified as c_1 and if $y_n = 0$, then $\mathbf{x}$ is classified as c_2. We model

$$P(y|\mathbf{x}) = \begin{cases} \hat{y}_{\mathbf{w}}(\mathbf{x}), & \text{if } \mathbf{x} \text{ is in } c_1 \\ 1 - \hat{y}_{\mathbf{w}}(\mathbf{x}), & \text{if } \mathbf{x} \text{ is in } c_2 \end{cases}$$

which for $y \in \{0, 1\}$ is compactly expressed as:

$$P(y|\mathbf{x}) = \hat{y}_{\mathbf{w}}(\mathbf{x})^y (1 - \hat{y}_{\mathbf{w}}(\mathbf{x}))^{(1-y)}$$

Assuming input examples are i.i.d. then the **likelihood function** is:

$$\mathcal{L} = \prod_{n=1}^{N} \hat{y}_{\mathbf{w}}(\mathbf{x}_n)^{y_n} (1 - \hat{y}_{\mathbf{w}}(\mathbf{x}_n))^{(1-y_n)}$$

And so the negative log-likelihood gives us an error to minimize (known as **cross-entropy**):

$$E(\mathbf{w}) = -\ln \mathcal{L} = -\sum_{n=1}^{N} y_n \hat{y}_{\mathbf{w}}(\mathbf{x}_n) + (1 - y_n)(1 - \hat{y}_{\mathbf{w}}(\mathbf{x}_n))$$

Multi-class classification The case with $K > 2$ classes (multi-class) is modeled with a neural network with K neurons in the output layer. Each output neuron computes $\hat{y}_k(\mathbf{x})$ as the probability of $\mathbf{x}$ belonging to the k-th class[4]. In this case, target values are of the form $\mathbf{y} \in \{0, 1\}^K$ with the restriction that only one entry of $\mathbf{y}$ is set to 1, and all other are 0. So now $\mathbf{y}$ is a one-hot bit-vector indicating what class $\mathbf{x}$ belongs to. In this case, using the same log-likelihood argument as in the binary case, we end up with the **generalized cross-entropy** error function:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} y_{n,k}\, \hat{y}_k(\mathbf{x}_n)$$

As a final remark, we note that these error functions, in all three cases (regression, binary and multi-class classification) have very complex shapes – the deeper the networks, the more complex – and therefore finding weights minimizing the error becomes difficult. In most cases, some form of **gradient descent** is used and so a major concern is to compute gradients efficiently. This is the topic of the following chapter, where we cover the famous **backpropagation** algorithm for computing gradients.

[4]Notice that we have dropped the $\mathbf{w}$ subscript from the output neurons' predictions to avoid too much clutter in the notation.

2. Training the Multilayer Perceptron (backprop)

In this chapter we introduce the algorithm of **backpropagation** which allows us to compute gradients efficiently in the context of neural networks. In fact, this is more general algorithm which works for any **computational graph**. Feed-forward neural networks are indeed computational graphs, and so the Multilayer perceptron is one as well.

In this document we describe backprop incrementally by using a simple example to illustrate the main ideas. Then, we describe backprop for the particular case of MLPs. The generalization to any directed graph whose nodes are differentiable is straightforward.

Backprop's two main characteristics are (1) its extensive use of the **chain rule** for computing derivatives, and (2) its efficiency achieved by storing intermediate results in a dynamic programming fashion.

2.1 Derivatives

But let us start with the definition of **derivative** of a real-valued function f at a given location a. As you may recall from your schooldays, this is

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

What this means is, if we were to change a slightly, how much (and how) will f change. In the case of multivariate f, we have a **partial derivative** for each input variable. Namely, the partial derivative of a function $f(x_1, .., x_m)$ in the direction x_i at the point $(a_1, ..., a_m)$ is defined as:

$$\frac{\partial f}{\partial x_i}(a_1, .., a_m) = \lim_{h \to 0} \frac{f(a_1, .., a_i + h, .., a_m)}{h}.$$

So we can define the **gradient** of a multi-variate function f at location $(a_1, .., a_m)$ as the following vector:

$$\nabla f(a_1, .., a_m) = \left(\frac{\partial f}{\partial x_1}(a_1, .., a_m), .., \frac{\partial f}{\partial x_m}(a_1, .., a_m) \right)$$

What does **backprop** do? Well, it computes the gradient $\nabla E(\mathbf{w})$ efficiently, where $E(\mathbf{w})$ is the error incurred by the network with weights $\mathbf{w}$. This is then used by gradient-based optimization techniques to update the current weights, e.g. $\mathbf{w} = \mathbf{w} - \alpha \nabla E(\mathbf{w})$ if we were using gradient descent with learning rate α. Since we have to compute $\nabla E(\mathbf{w})$ in each iteration of the optimization procedure, doing so efficiently is crucial, and this is why backprop has such importance in the context of neural networks.

2.2 The *chain rule* for taking derivatives of function compositions

Backprop uses the following two forms of the chain rule for real-valued functions:

1. Its simplest form, where we want to take the derivative of a composition of two functions:

   $$f(g(x))' = f'(g(x))g'(x)$$

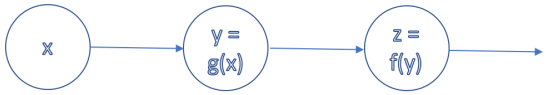
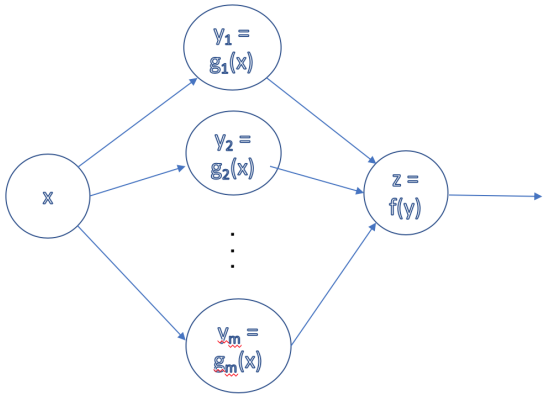
   Some of you may be more familiar with the notation (equivalent if $z = f(y)$ and $y = g(x)$):

   $$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

2. When the dependence is through more than one intermediate function (want to compute the derivative of $f(g_1(x), ..g_m(x))$ w.r.t. x): (so $z = f(y_1, ..., y_m)$ and $y_i = g_i(x)$), then:

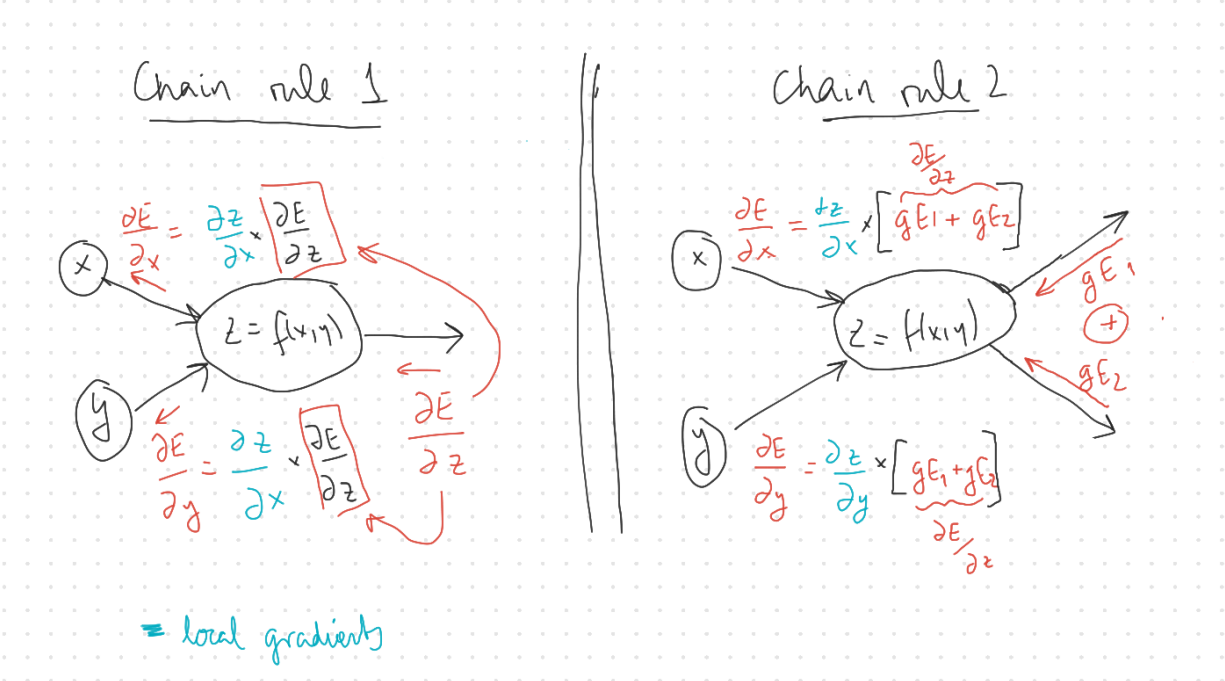
   $$\frac{dz}{dx} = \sum_{i=1}^{m} \frac{\partial z}{\partial y_i}\frac{dy_i}{dx}$$

These two situations correspond to the following computational graphs (or corresponding *function compositions*):

Chain rule 1	Chain rule 2
$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$	$\frac{dz}{dx} = \sum_{i=1}^{m} \frac{\partial z}{\partial y_i}\frac{dy_i}{dx}$
	

Simple example

In this first example, we will show how to apply the two chain rules in the context of a very simple example. We will start at the right-most node and work our way upwards "propagating" the gradient upstream making use of one of the two chain rules node by node.



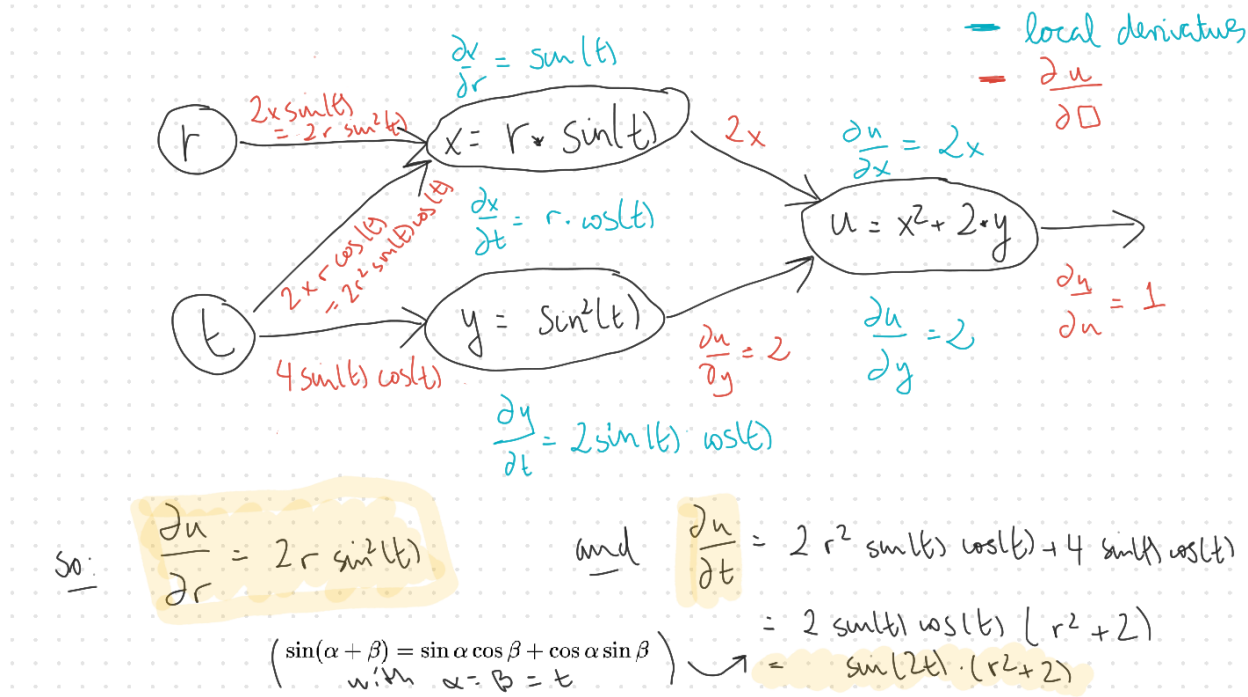
We will apply these two rules to the following example taken from Wikipedia:

Given $u(x, y) = x^2 + 2y$ where $x(r, t) = r\sin(t)$ and $y(r, t) = \sin^2(t)$, determine the value of $\frac{\partial u}{\partial r}$ and $\frac{\partial u}{\partial t}$

- $\frac{\partial u}{\partial r} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial r}$ since u depends on r through x and y, so $\frac{\partial u}{\partial r} = (2x)(\sin(t)) + (2)(0) = 2r\sin^2(t)$
- $\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial t}$ since u depends on t through x and y, so $\frac{\partial u}{\partial t} = (2x)(r\cos(t)) + (2)(2\sin(t)\cos(t)) = \ldots = (r^2+2)\sin(2t)$

One important thing to note in the example is that both partial derivatives need to compute $\frac{\partial u}{\partial x}$ and $\frac{\partial u}{\partial y}$, and so storing these values once computed will save doing it twice.

Now, if we use the *computational graph* corresponding to the function u defined, we have:



There are a couple of things worth noting from this example:

1. We are able to **avoid unnecessary computations** by storing at each node both forward values, local derivatives, and backward gradients as we follow the computation graph topology. This is a key factor, crucial for making neural network learning computationally feasible.
2. In this example we computed the gradient **symbolically**, namely, we do not have concrete values for the inputs, so we are computing the gradient in the whole input space. Since we did this for general t and r, there is no forward pass in this example.
3. All we have done is apply one form of chain rule at each node. Namely, we applied chain rule 1 to every node except node t; since t is connected to more than one node downstream, we had to make use of chain rule 2 in this case. This comes down to **adding** the gradients that are flowing back into t from every connection it has to other nodes downstream.

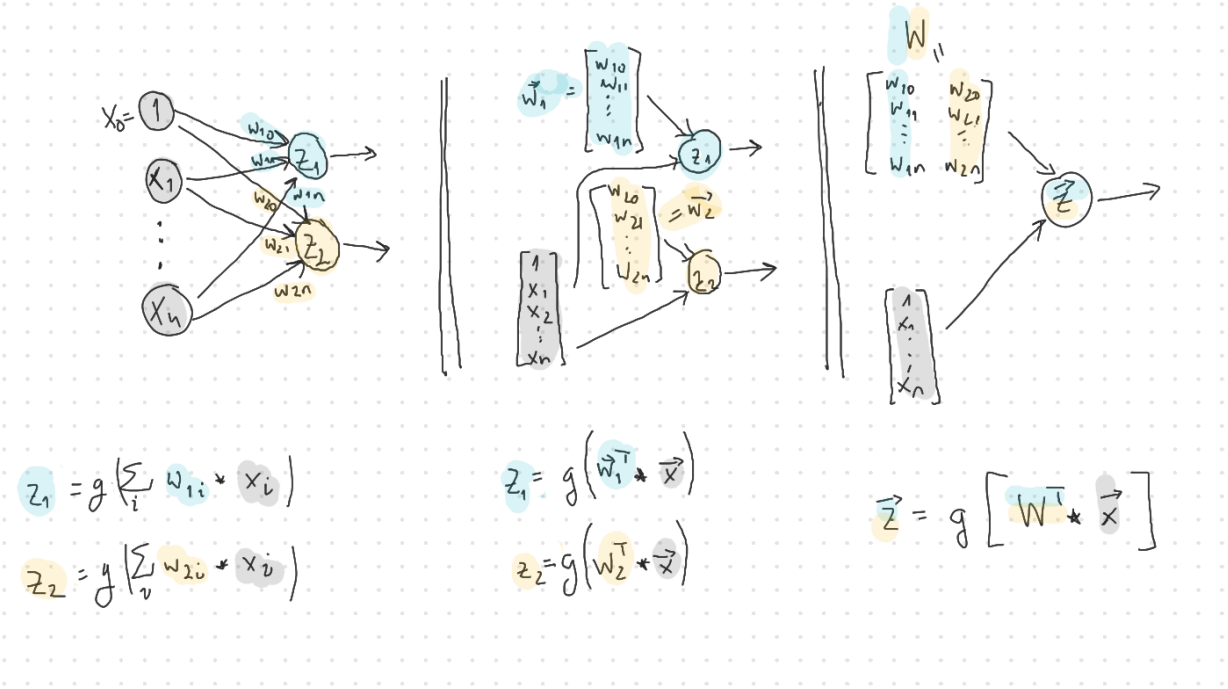
2.3 Introducing `backprop` through examples with incremental architecture complexity

2.3.1 Example with no hidden layers

So hopefully by now you get the general idea of the computation that we are performing. Another improvement comes from the fact that we will optimize parameters for neurons in the same layers "in parallel" by vectorizing computations. If your computer can perform such vectorized computations efficiently, you

should notice a **huge** improvement in performance. This is the reason for the high demand of GPUs when working with deep networks.

The network for this section is:



On the left, our graph in our usual "node-level" or "neuron level" notation. In the middle, you can see vectorized notation. Here, we have put together into vectors those parameters that correspond to each neuron. Thus, vector $\mathbf{w}_1$ is the column vector containing all paramters that are local to z_1 (neuron 1), and likewise with the second output neuron z_2:

$$\mathbf{w}_1 = \begin{pmatrix} w_{10} \\ w_{11} \\ \cdots \\ w_{1n} \end{pmatrix} \mathbf{w}_2 = \begin{pmatrix} w_{20} \\ w_{21} \\ \cdots \\ w_{2n} \end{pmatrix}$$

So, in this case the output value for neuron z_k (for $k = 1, 2$) is computed by performing a dot product between (local) weights $\mathbf{w}_k$ and input data $\mathbf{x}$ (and then applying the activation function g):

$$z_k = g(\mathbf{w}_k^T \mathbf{x})$$

On the right, we go one step further and put all weights into a single matrix W which contains all local neuron's weight vectors as columns. Namely: $W = (\mathbf{w}_1 \mathbf{w}_2)$. And so the output values can be computed as a matrix-vector multiplication, and then applying the component-wise activation function $g[\cdot]$ to the resulting vecotr. Thus:

$$\mathbf{z} = g[W^T \mathbf{x}] = g\left[\begin{pmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \end{pmatrix} \mathbf{x}\right] = g\left[\begin{pmatrix} \mathbf{w}_1^T \mathbf{x} \\ \mathbf{w}_2^T \mathbf{x} \end{pmatrix}\right] = \begin{pmatrix} g(\mathbf{w}_1^T \mathbf{x}) \\ g(\mathbf{w}_2^T \mathbf{x}) \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

Input nodes are shaded in grey, this is to signify that they are constant and therefore remain fixed throughout computation of gradients (for a given input vector $\mathbf{x}$).

This network contains $n + 1$ input nodes and 2 output nodes. In general, if we had m nodes in the output layer, the computation done in each output node would be $z_k = g\left(\sum_{i=0}^{n} w_{ki} x_i\right)$, and W would be of size

$((n+1) \times m)$, and $\mathbf{z}$ would be of size $(m \times 1)$. Let us assume from now on that, in fact, we have m nodes at the output, and do the computations in this general case.

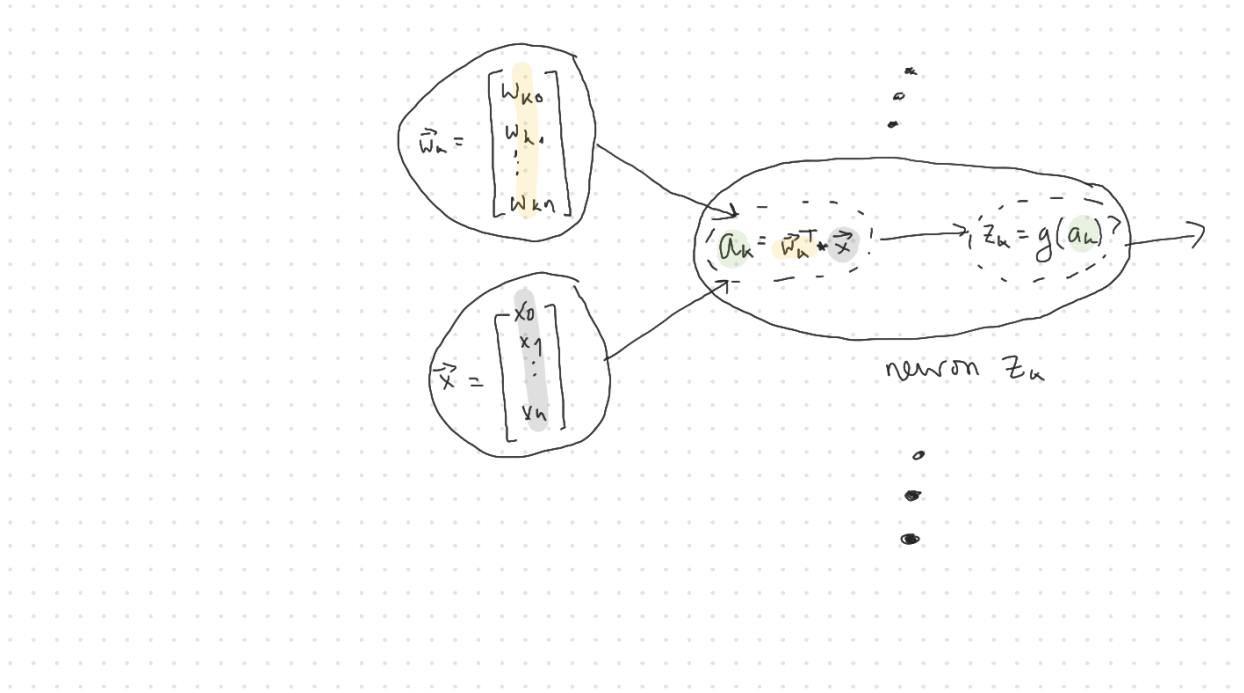
So, our aim is to compute gradient of the output of this network w.r.t. its weights W, namely $\frac{\partial \mathbf{z}}{\partial W}$, at location W and on input data $\mathbf{x}$. So, the first thing is to compute intermediate values of the network on a **forward pass**. All we do here, is to do the matrix multiplication as noted above. To make things easier, we introduce new variables $a_k = \mathbf{w}_k^T \mathbf{x}$ for each output neuron k; and so:

- $a_k := \mathbf{w}_k^T \mathbf{x} = \sum_{i=0}^{n} w_{ki} x_i$
- $\mathbf{a} = W^T \mathbf{x}$
- $z_k = g(a_k)$
- $\mathbf{z} = g[\mathbf{a}]$

These intermediate values a_k (before applying the activation) help us in the backward pass to make things one step at a time, and are sometimes known as the **net input** values of a neuron z_k.

So once we have done the forward pass, we have all intermediate values $\mathbf{a} = W^T \mathbf{x}$ and $\mathbf{z} = g[\mathbf{a}]$. Now, let us compute gradients as we backwards in the network (**backward pass**).

Since each neuron z_k is built from a sequential computation (first compute $a_k = \mathbf{w}_k^T \mathbf{x}$, then apply activation function $z_k = g(a_k)$) we look at what happens with an individual neuron:



So, we are going to compute $\frac{\partial z_k}{\partial \mathbf{w}_k}$. In fact, weights in the k-th column of W only affect the computation of neuron z_k (these are local parameters), so the gradients w.r.t. this k-th column only come from z_k. We need **local gradients** first. In the case of $z_k = g(a_k)$ its local gradient is simply

- $\frac{\partial z_k}{\partial a_k} = g'(a_k)$, where $g'(a_k)$ denotes the derivative of g w.r.t. a_k. Since we use *differentiable activation functions* this derivative always exists

The **local gradient** of the a_k node w.r.t. its i-th weight w_{ki} is: $\frac{\partial a_k}{\partial w_{ki}} = \frac{\partial \sum_{i'} w_{ki'} x_{i'}}{\partial w_{ki}} = x_i$. We can express this **local gradient** compactly for all its local weights $\mathbf{w}_k$ as:

- $\frac{\partial a_k}{\partial \mathbf{w}_k} = \mathbf{x}$

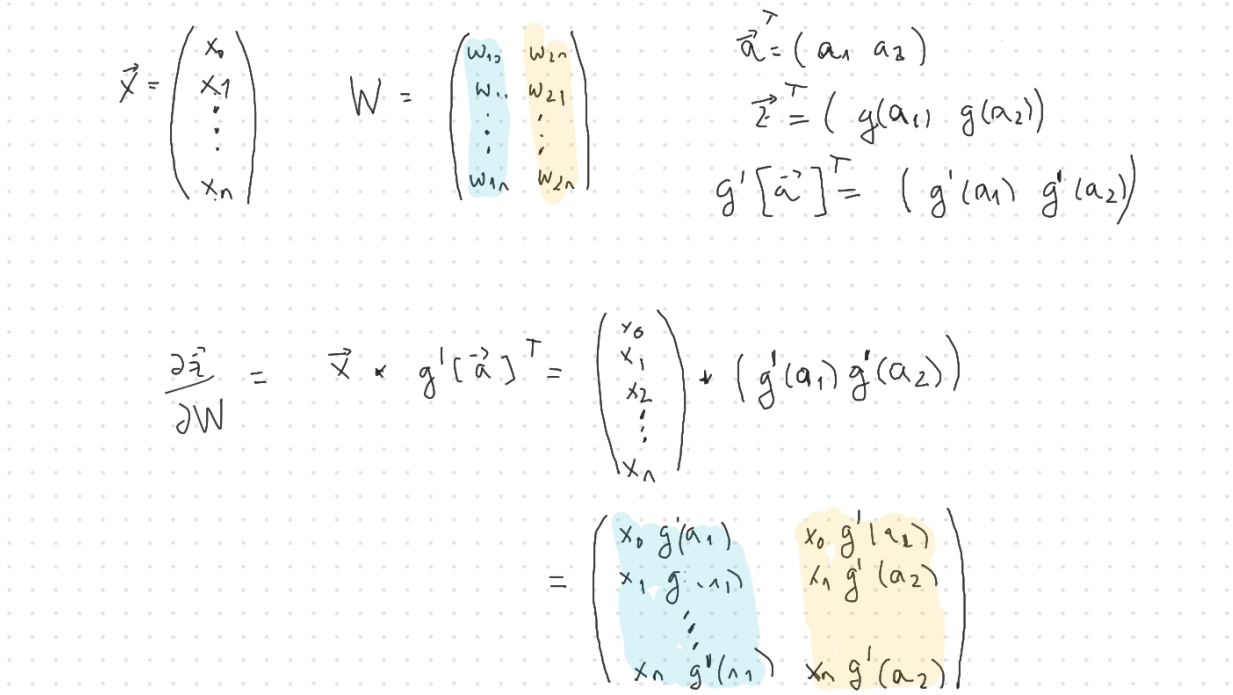
And finally, using the chain rule we have that:

$$\frac{\partial z_k}{\partial \mathbf{w}_k} = \frac{\partial z_k}{\partial a_k}\frac{\partial a_k}{\partial \mathbf{w}_k} = g'(a_k)\,\mathbf{x}$$

Since there is no interaction between output values z_k and local weights $\mathbf{w}_j$ from another neuron z_j with $j \neq k$, we can concatenate all gradients into a gradient matrix as follows:

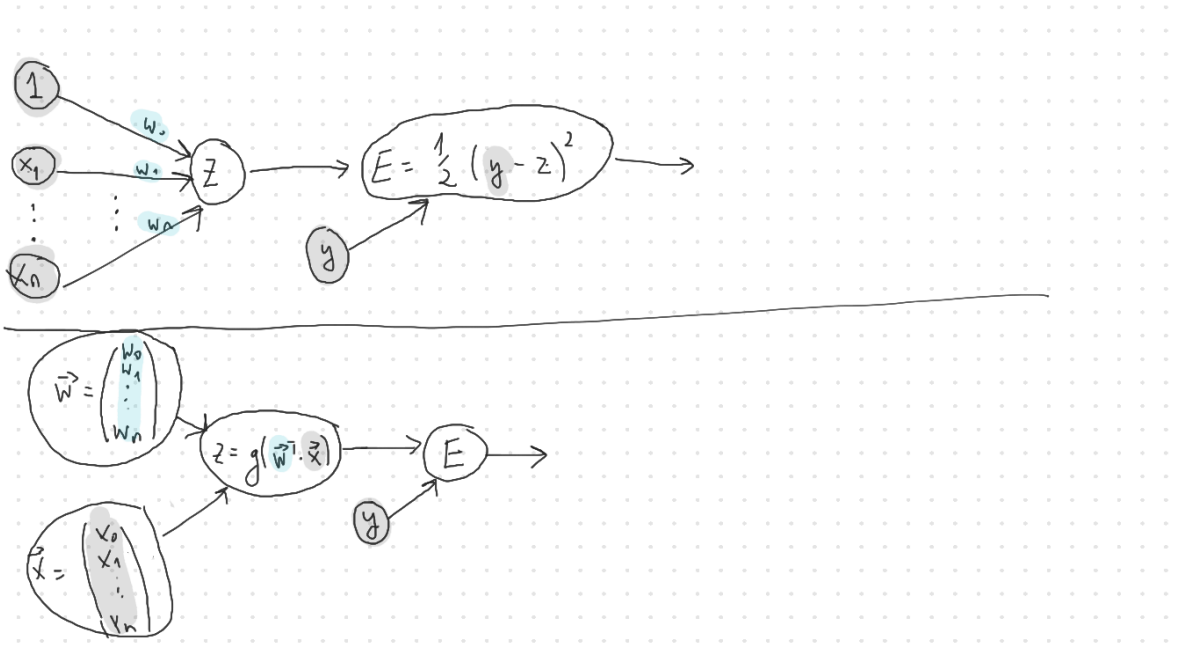
$$\frac{\partial \mathbf{z}}{\partial W} = \mathbf{x}\, g'[\mathbf{a}]^T$$

The following image shows this:

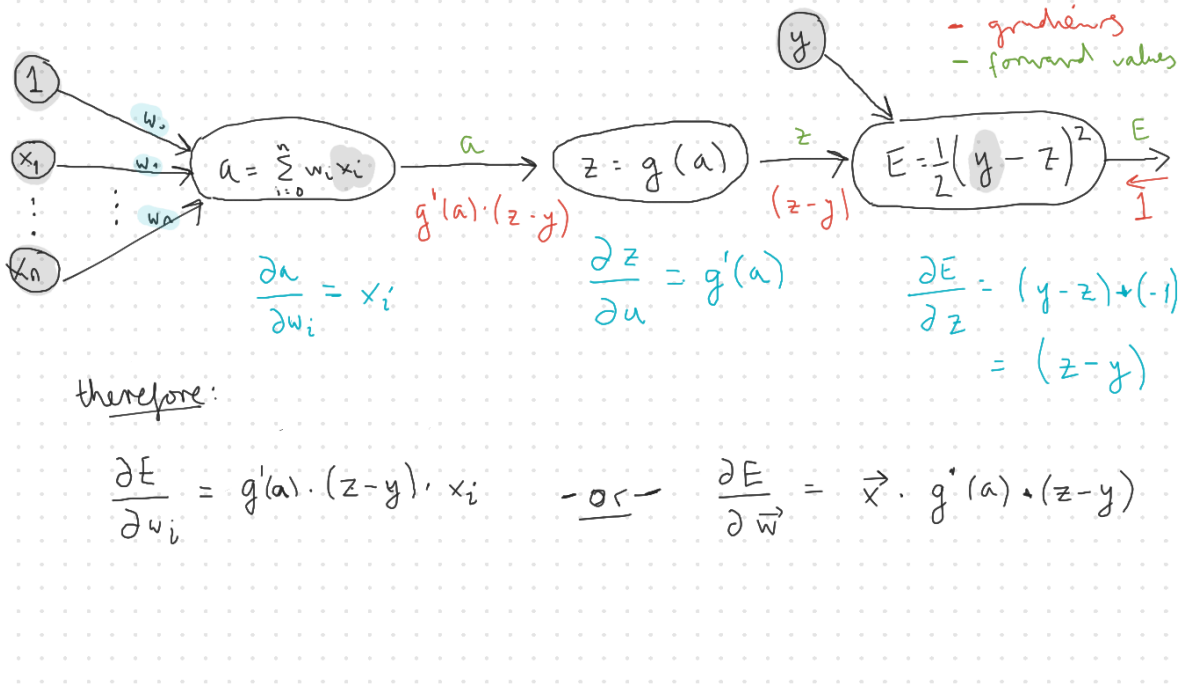


2.3.2 Regression example: neural network for single output regression using mean squared error

Notice that in the previous example there was no notion of quality of prediction, this next step will introduce an error in the context of regression. We will use a very simple model, equivalent to linear regression with one single prediction value. The picture now is:



Now the picture is slightly simpler than in the previous example because we only have one neuron at the output, but we have included a neuron downstream that computes the squared error of the output produced by the single neuron in the output layer. With only one neuron producing an output, our weight matrix contains a single column and it is thus a column vector $\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_n \end{pmatrix}$.



So, the final gradient is a column vector of partial derivatives (because we have a column vector of weights

as our parameters) and is given by:

$$\nabla E = \begin{pmatrix} x_0 g'(a)(z-y) \\ x_1 g'(a)(z-y) \\ \cdots \\ x_n g'(a)(z-y) \end{pmatrix} = \mathbf{x} g'(a)(z-y)$$

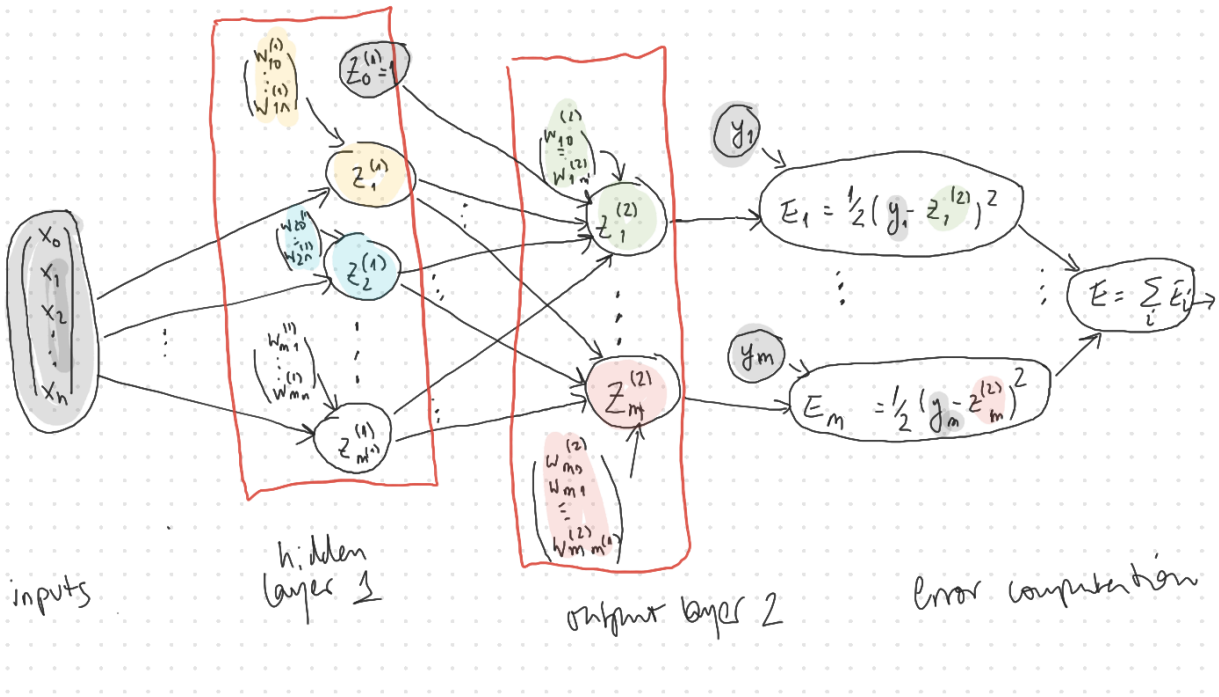
Notice the similarity with the previous example, where we multiplied $\mathbf{x}$ by $g'[\mathbf{a}]^T$ to obtain the gradient. In our current example, this follows the same shape, but since we only have one output neuron we multiply $\mathbf{x}$ by the scalar value $g'(a)$ and also take into account the error E measuring accuracy of prediction z.

This gradient gives now an extremely simple algorithm using batch gradient descent (with small enough learning rate α) for linear regression. The previous example shows how to compute the gradient of the error for a single input example $(\mathbf{x}, y)$, in general we have a set of such examples. But since the error is the sum of individual errors (for each input example pair), we just repeat the procedure for all examples and sum the individual gradients. That is why in the following code you see **gw**, this is the accumulator of individual example error gradients.

1. start with random weights $\mathbf{w}$
2. for a certain number of iterations or until convergence (i.e. $\mathbf{w}$ does not change):
   - set vector $\mathbf{gw}$ to all zeros *// $\mathbf{gw}$ accumulates error gradients over all examples*
   - for each input example pair $(\mathbf{x}, y)$:
     - compute forward values a, z (that depend on $\mathbf{x}$, y, and $\mathbf{w}$)
     - set $\mathbf{gw} = \mathbf{gw} + \mathbf{x} g'(a)(z-y)$
   - update weights $\mathbf{w} = \mathbf{w} - \alpha \mathbf{gw}$

2.3.3 Regression example: neural network with one hidden layer and multiple outputs

This example uses a network to model functions $\mathbb{R}^n \mapsto \mathbb{R}^m$. That is, we have n real-valued inputs and m real-valued inputs. Therefore our network contains m neurons in its output layer, each modelling one of the output dimensions of the function we are modelling. The input data pairs are therefore two vectors $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$. The network will use the *empirical mean squared error* as its error function, and will also add the contribution to this error from each output component. Also, to make things more interesting, we include one hidden layer with multiple neurons as the diagram suggests. Notice that this takes us out from the linear modelling realm, since introducing this non-linear hidden neuron will add non-linear modelling powers to our network.



We introduce some notation. We use the **superscript** $\cdot^{(l)}$ to denote layers, and use $m^{(l)}$ to denote the number of neurons at each layer l. Also, since we have m output neurons, $m^{(2)} = m$ in this case. Typically we say that a MLP has c hidden layers, so $1 \leq l \leq c+1$ (the output layer is layer $c+1$). In this example, $c = 1$. For weights:

- $w_{ki}^{(l)}$ refers to the i-th weight of neuron z_k of layer l. In this case, $1 \leq l \leq 2$, $1 \leq k \leq m^{(l)}$, and $0 \leq i \leq m^{(l-1)}$. For notational convenience, we let the inputs be referred to as the 0-th layer, and so: $z_i^{(0)} = x_i$, including $z_0^{(0)} = x_0$. Accordingly, $m^{(0)} = n$.
- $\mathbf{w}_k^{(l)}$ refers to the weight vector local to neuron z_k of layer l. Again, $l = 1, 2$ and $1 \leq k \leq m^{(l)}$
- $W^{(l)}$ is the concatenation of weight vectors of neurons at layer l; it is of size $(m^{(l-1)} + 1) \times m^{(l)}$
- $\mathbf{z}^{(l)}$ is now the vector including all outputs from neurons in layer l, including the extra "1" added $z_0^{(l)} = 1$. This is **not necessary** for the output layer, as downstream nodes do not need a bias.
- $\mathbf{a}^{(l)}$ are now the net input values for neurons at layer l, namely we have that $\mathbf{z}^{(l)} = g\left[\mathbf{a}^{(l)}\right]$

If you feel this is a lot of notation, I completely agree. However, we need to be able to refer to those quantities and we have many dimensions to consider: l, k, i.

So, we are given an input pair $(\mathbf{x}, \mathbf{y})$ and have some current values for all $W^{(l)}$. We use backprop to compute the gradient $\frac{\partial E}{\partial W^l}$ for all $1 \leq l \leq 2$.

First, we perform a **forward pass** to compute all values of intermediate nodes:

1. compute $\mathbf{a}^{(1)} = {W^{(1)}}^T \mathbf{x}$
2. compute $\mathbf{z}^{(1)} = g\left[\mathbf{a}^{(1)}\right]$
3. compute $\mathbf{a}^{(2)} = {W^{(2)}}^T \mathbf{z}^{(1)}$
4. compute $\mathbf{z}^{(2)} = g\left[\mathbf{a}^{(2)}\right]$
5. compute $E_k = \frac{1}{2}(y_k - z_k^{(2)})^2$ for $k = 1, .., m$
6. compute $E = \sum_k E_k$ as the total error

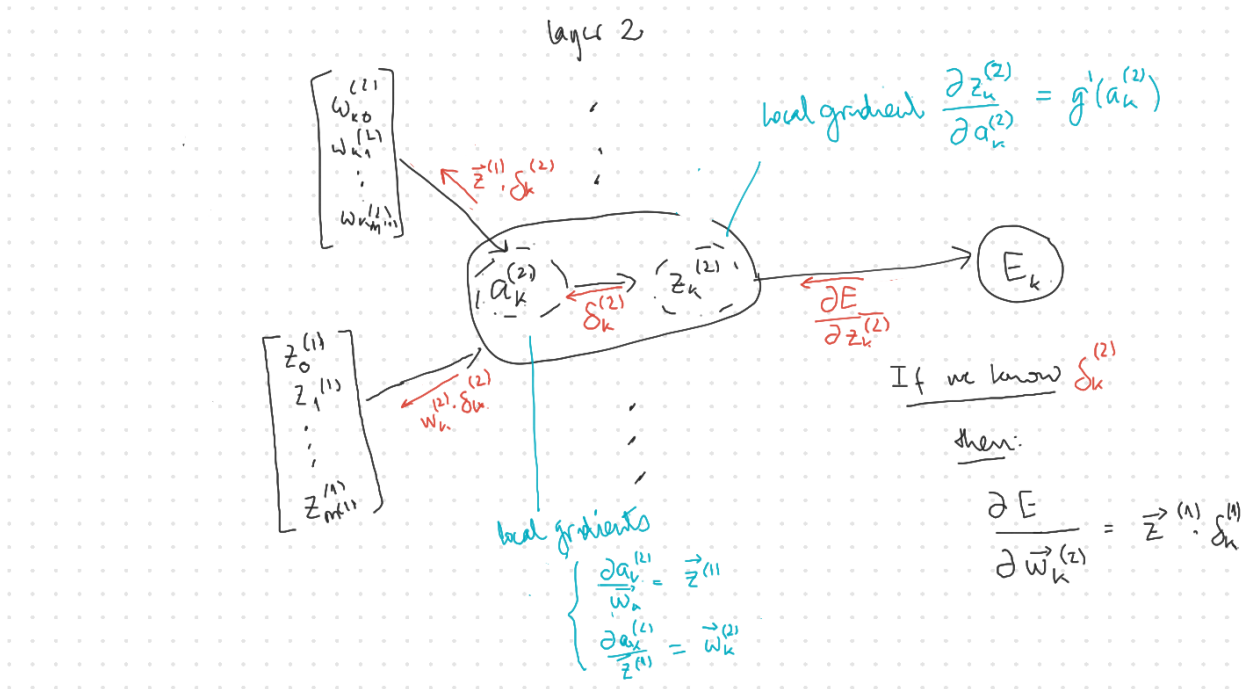
The error computation can be done using vectors as: $E = \frac{1}{2}\left[(\mathbf{y} - \mathbf{z}^{(2)})^2 * \begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}\right]$ where the substraction and squaring is done component-wise. The final multiplication by the all ones vector is to sum its components (step 6 in previous pseudocode).

If we had more layers, this forward pass would require looping from $l = 1$ up to $l = c + 1$ to compute $\mathbf{a}^{(l)}$ and $\mathbf{z}^{(l)}$. The error computation would then proceed as in this example using $\mathbf{z}^{(c+1)}$ values (from layer $c + 1$, i.e., the output layer).

Once we have all the output values, its time to to the **backward pass**. Let us start by computing:

- $\frac{\partial E}{\partial z_k^{(2)}} = \frac{\partial \sum_j E_j}{\partial z_k^{(2)}} = \frac{\partial E_k}{\partial z_k^{(2)}} = \frac{\partial \frac{1}{2}(y_k - z_k^{(2)})^2}{\partial z_k^{(2)}} = (z_k^{(2)} - y_k)$.
- $\frac{\partial E}{\partial \mathbf{z}^{(2)}} = (\mathbf{z}^{(2)} - \mathbf{y})$

We have the gradients at the outputs of the output layer, let us see the scenario to continue advancing backwards:



We focus on output neuron k and we open it up in order to make computations more explicit. We define a convenience variable $\delta_k^{(2)} := \frac{\partial E}{\partial a_k^{(2)}}$. Notice the similarity with a previous example where we have already computed this (with the only difference that in that example, instead of hidden layer's outputs we had the input $\mathbf{x}$), therefore:

Local gradients are:

$$\frac{\partial z_k}{\partial a_k} = g'(a_k) \qquad \frac{\partial a_k}{\partial \mathbf{w}_k} = \mathbf{z}^{(1)} \qquad \frac{\partial a_k}{\partial \mathbf{z}^{(1)}} = \mathbf{w}_k^{(2)}$$

So:

$$\delta_k^{(2)} := \frac{\partial E}{\partial a_k^{(2)}} = \frac{\partial E}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial a_k^{(2)}} = g'(a_k^{(2)})(z_k^{(2)} - y_k)$$

$$\frac{\partial E}{\partial \mathbf{w}_k^{(2)}} = \mathbf{z}^{(1)} \delta_k^{(2)}$$

Notice that we can vectorize this computation by noting that

$$\boldsymbol{\delta}^{(2)} = \begin{pmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \\ \vdots \\ \delta_m^{(2)} \end{pmatrix} = \begin{pmatrix} g'(a_1^{(2)})(z_1^{(2)} - y_1) \\ g'(a_2^{(2)})(z_2^{(2)} - y_2) \\ \vdots \\ g'(a_m^{(2)})(z_m^{(2)} - y_m) \end{pmatrix} = g'\left[\mathbf{a}^{(2)}\right] \odot (\mathbf{z}^{(2)} - \mathbf{y})$$

where $\odot$ stands for the component-wise or Hadamard product (this is **not** a dot product).

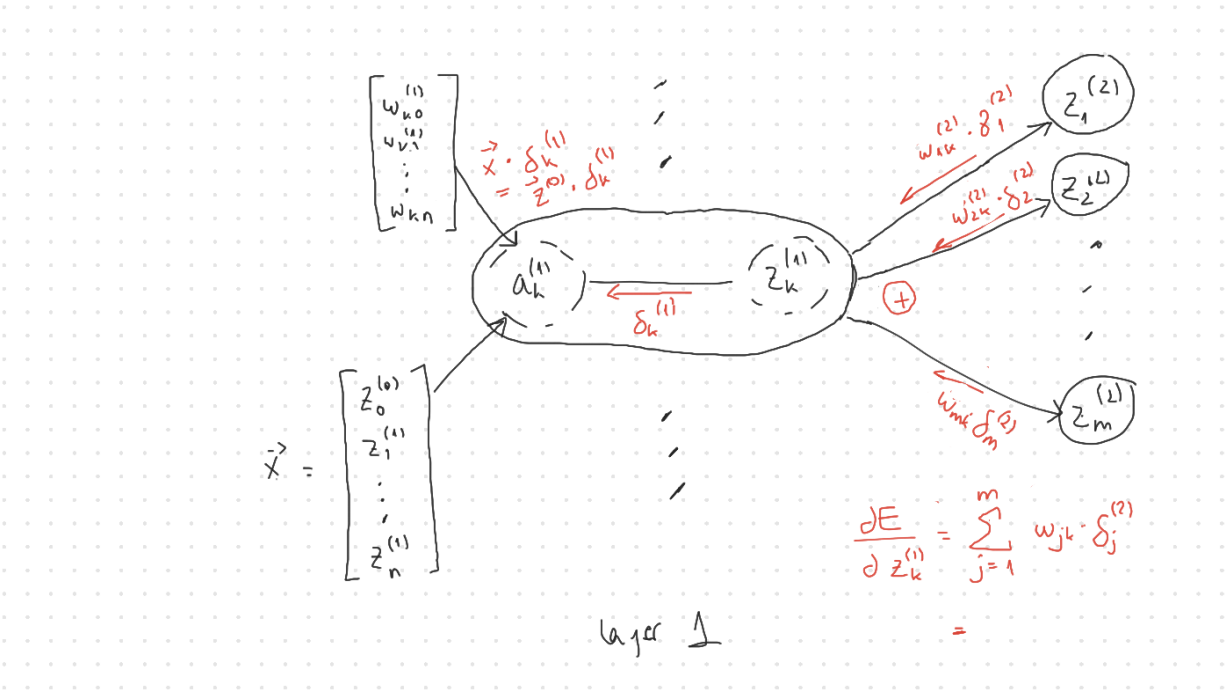
Since all the contribution of the gradient to neuron's k's weights is through $a_k^{(2)}$, this means that the gradient of the error w.r.t. $W^{(2)}$ is given by concatenating all gradient columns from all neurons:

$$\frac{\partial E}{\partial W^{(2)}} = \begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ \vdots \\ z_{m^{(1)}}^{(1)} \end{pmatrix} \left(\delta_1^{(2)} \delta_2^{(2)} \cdots \delta_m^{(2)}\right) = \left(\mathbf{z}^{(1)} \delta_1^{(2)} \quad \mathbf{z}^{(1)} \delta_2^{(2)} \quad \cdots \quad \mathbf{z}^{(1)} \delta_m^{(2)}\right) = \mathbf{z}^{(1)} \ {\boldsymbol{\delta}^{(2)}}^T$$

where now we are performing an outer product between $\mathbf{z}^{(1)}$ (of size $(m^{(1)} + 1) \times 1$) and $\boldsymbol{\delta}^{(2)}$ (of size $m \times 1$). So this operation returns a matrix of size $(m^{(1)} + 1) \times m$ which is precisely the size of $W^{(2)}$.

Notice that there is another interesting quantity that we can easily compute from $\delta_k^{(2)}$, which is the gradient that neuron k sends to neurons in the previous layer, which can be computed as $\mathbf{w}_k^{(2)} \delta_k^{(2)}$. Notice that this is a vector of size $(m^{(1)} + 1) \times 1$ so the sizes coincide as well.

Now, we need to go one layer to the left in order to compute the missing gradient $\frac{\partial E}{\partial \mathbf{W}^{(1)}}$. Now the situation is:



This is very similar to our previous output layer, however now the k-th neuron from layer 1 is receiving gradient from all neurons in the following layer, and se we will have to sum these gradients up to account for the total gradient coming into $z_k^{(1)}$. As before, an important quantity to compute is $\delta_k^{(1)} := \frac{\partial E}{\partial a_k^{(1)}}$. So now:

$$\delta_k^{(1)} = g'(a_k^{(1)}) \sum_{j=1}^{m^{(2)}} w_{jk}^{(2)} \delta_j^{(2)}$$

This expression can be vectorized by noting that the sum is computing a dot product between the *deltas* from the previous layer and the k-th **row** from the weight matrix $W^{(2)}$. Since $\mathbf{w}_k$ is taken as the k-th column of the matrix we are going to use $W_{k\star}^{(2)}$ to denote its k-th row. Therefore, $\delta_k^{(1)} = g'(a_k^{(1)}) W_{k\star}^{(2)} \boldsymbol{\delta}^{(2)}$. Now to compute the full $\boldsymbol{\delta}^{(1)}$ vector:

$$\boldsymbol{\delta}^{(1)} = \begin{pmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \vdots \\ \delta_{m^{(1)}}^{(1)} \end{pmatrix} = \begin{pmatrix} g'(a_1^{(1)}) W_{1\star}^{(2)} \boldsymbol{\delta}^{(2)} \\ g'(a_2^{(1)}) W_{2\star}^{(2)} \boldsymbol{\delta}^{(2)} \\ \vdots \\ g'(a_m^{(1)}) W_{m^{(1)}\star}^{(2)} \boldsymbol{\delta}^{(2)} \end{pmatrix} = g'\left[\mathbf{a}^{(1)}\right] \odot (\tilde{W}^{(2)} \boldsymbol{\delta}^{(2)})$$

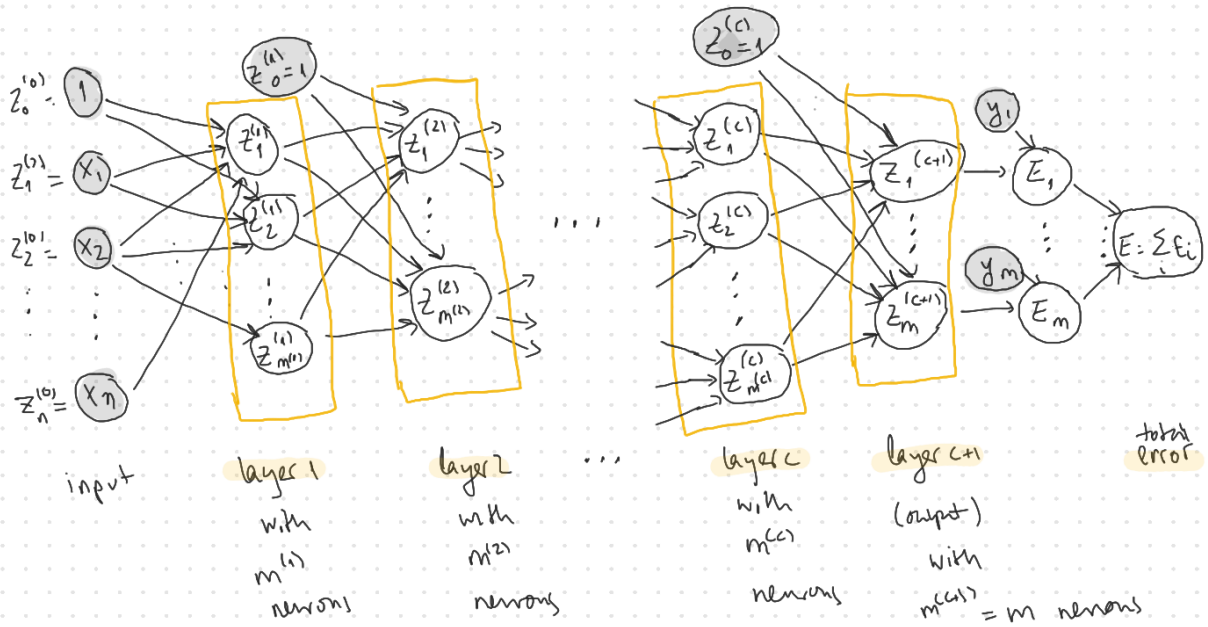
where $\tilde{W}^{(2)}$ stands for the matrix $W^{(2)}$ without the first row of biases.

Now that we have the *deltas* for this layer, we can compute the gradient of the error w.r.t. weight matrix $W^{(1)}$ as an outer product:

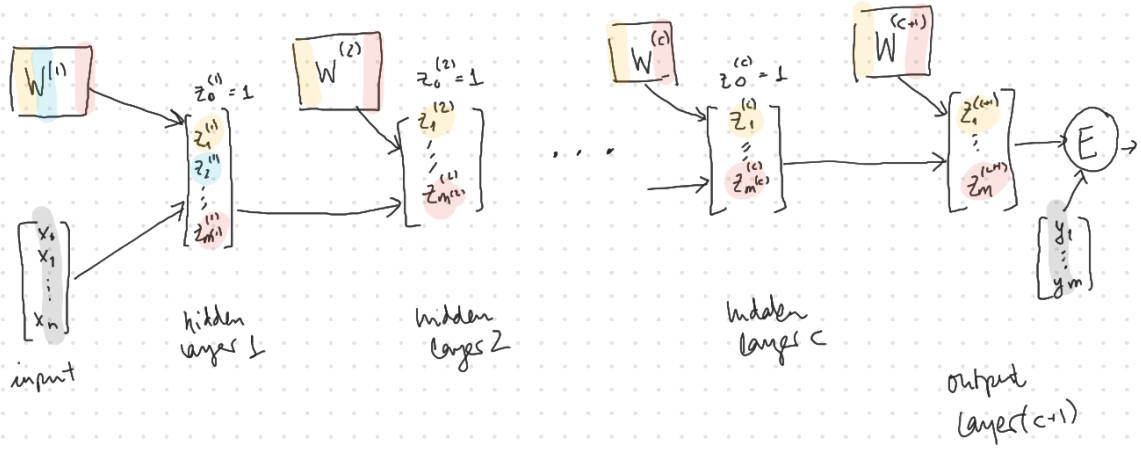
$$\frac{\partial E}{\partial W^{(1)}} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \left(\delta_1^{(1)} \delta_2^{(1)} \cdots \delta_{m^{(1)}}^{(1)} \right) = \mathbf{x}\, \boldsymbol{\delta}^{(1)^T}$$

At this point, we have finished and have all the required gradients.

2.3.4 Regression example: general MLP with multiple hidden layers



The previous example was complete in the sense that it shows the mechanisms for computing the gradient from layer l to previous layer $l-1$, so by some sort of induction we can introduce the pseudocode for the backpropagation algorithm. We use the vectorized representation to keep things compact.



We depart from any general MLP with $c \geq 0$ hidden layers. Each layer l contains $m^{(l)}$ neurons and we use the convention of making our input layer be the 0-th layer. Again, we use the network for regression of multiple outputs. Each layer has its local weight matrix $W^{(l)}$ of size $(m^{(l-1)}+1) \times m^{(l)}$, $\tilde{W}^{(l)}$ stands for the

local weight matrix at layer l without the first row corresponding to biases. At the output we have $m^{(c+1)}$ neurons.

1. **forward pass**
   - for layers $l = 1$ up to $l = c + 1$:
     - $\mathbf{a}^{(l)} = {W^{(l)}}^T \mathbf{z}^{(l-1)}$, and
     - $\mathbf{z}^{(l)} = g\left[\mathbf{a}^{(l)}\right]$
2. **backward pass**
   - $\boldsymbol{\delta}^{(c+1)} = g'\left[\mathbf{a}^{(c+1)}\right] \odot (\mathbf{z}^{(c+1)} - \mathbf{y})$
   - $\frac{\partial E}{\partial W^{(c+1)}} = \mathbf{z}^{(c)}\ {\boldsymbol{\delta}^{(c+1)}}^T$
   - for $l = c$ down to $l = 1$:
     - $\boldsymbol{\delta}^{(l)} = g'\left[\mathbf{a}^{(l)}\right] \odot (\tilde{W}^{(l+1)}\boldsymbol{\delta}^{(l+1)})$
     - $\frac{\partial E}{\partial W^{(l)}} = \mathbf{z}^{(l-1)}\ {\boldsymbol{\delta}^{(l)}}^T$

It should be straightforward to convert this pseudocode into code in any language that supports matrix and vector operations. Before doing that, we should specify of course which activation functions we want to use, so that we can substitute g and g' by the suitable expressions. The following and last section covers two options for g.

2.4 Differentiation of activation function g

In previous examples whenever we needed to differentiate g, we left it as g'. Namely: $\frac{\partial z_i}{\partial a_i} = \frac{\partial g(a_i)}{\partial a_i} = g'(a_i)$.

Logistic $g(x) = \sigma(x)$ Let us instantiate now g with the *logistic function* $\sigma(x) = \frac{1}{1+e^{-x}}$

$$\begin{aligned}
\frac{d\sigma(x)}{dx} &= \frac{d}{dx}(1+e^{-x})^{-1} \\
&= (-1)(1+e^{-x})^{-2}e^{-x}(-1) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \cdots \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)\,(1-\sigma(x))
\end{aligned}$$

In our context, typically $z_i^{(l)} = g(a_i^{(l)})$. Therefore, $g'(a_i^{(l)}) = g(a_i^{(l)})\left(1 - g(a_i^{(l)})\right) = z_i^{(l)}(1 - z_i^{(l)})$.

Hyperbolic tangent $g(x) = \tanh(x)$ Since $(\tanh(x))' = 1 - (\tanh(x))^2$, in this case

$$g'(a_i^{(l)}) = 1 - \left(g(a_i^{(l)})\right)^2 = 1 - \left(z_i^{(l)}\right)^2.$$