# 1 Subjects

- Backpropagation
- Deep Nets
- PCA
- Autoencoder

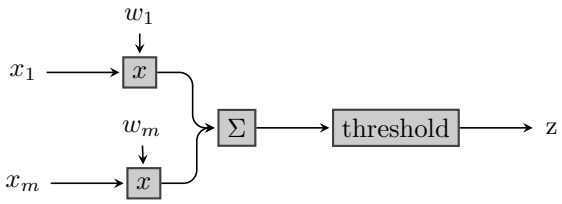
# 2 Notes

## 2.1 Neurons and background

Neurons consist of a dendritic tree (the input, which branches a lot), the axon (the output, which branches a little bit) and the cell body with nucleus.

Neurons work by having axons connected to dendritic trees, they give a binary (0 or 1) output, after an axon has sent a 1, it has to recharge.

In order to model this, we've got binary input $x_1, \ldots, x_n$ which is sent to the nucleus that sums the inputs together and outputs 1 if the input exceeds some threshold.




This models the following properties:

- All or non
- Cumulative influence
- Synaptic weight

But there are more properties that we might like to model, like:

- Refractory period, recovery time of each neuron.
- Axonal bifurcation, each pulse will either go down one branch of the axon or the other.
- Time patterns, we don't know if the timing of when impulses hit neurons matters.

So we actually don't know if what we are modelling is the essence of how neurons work or not. But we will start with the simple model.

If we look at what a neural net actually is, then it's a vector of input that goes through a "box" which uses some weights and threshold and then outputs some

vector $z$. Formally: $z = f(x, w, t)$. So the neural network is just the function $f$, when we train the neural net, all we need to do is change the weights and threshold. We can also think of the neural net as a "function approximator".

So, how do we measure the performance of the neural network? If we say the desired result of the neural net $d$ is the function $g$ on the input $x$, $d = g(x)$, then it would be natural to define the performance as: $P = \|d - z\|$, this turns out to be mathematically inconvenient so instead we will use the performance indicator: $P = \|d - z\|^2$.

What we want to do, is of course to maximize the performance. We can do this using gradient descent, for example for the weights $w_1$ and $w_2$:

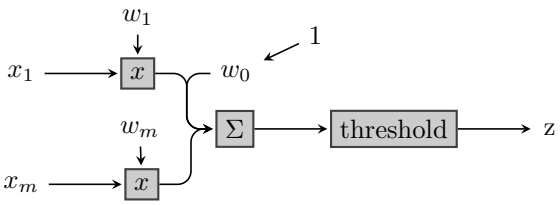
$$\Delta w = \eta \left( \frac{\partial P}{\partial w_1} i + \frac{\partial P}{\partial w_2} j \right)$$

Unfortunately, the function is not linear, and thus gradient descent doesn't really work. This was an issue for a long time untill Paul Werbos figure it out.

First of all, those thresholds are annoying as they are just extra baggage, so we would like to reduce $z$ to be a function of just the inputs and weights:

$$z = f(x, w)$$

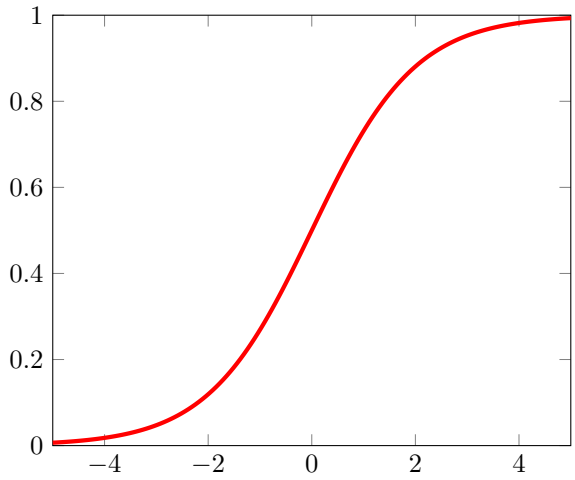
So what he proposed instead, was adding the bias input to each neuron $w_0$ which is always set to 1.



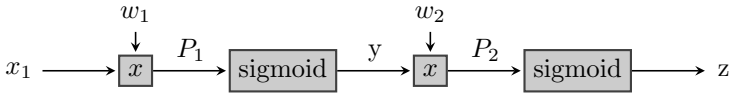

And then we let $w_0 =$ threshold, then the threshold is effectively 1.

revisit

Step two, is to smooth the threshold function, which we could do by applying the sigmoid function $\sigma(\alpha) = \frac{1}{1+e^{-\alpha}}$. Then it will be 1 if $\alpha$ is very big, and 0 if $\alpha$ is very small.




The generalized term "activation function" $\phi$ is in this case $\sigma$, our function is linear and we can take the partial derivatives to the function. Now, suppose we have a very simple neural network:




Now we want to re-write the partial derivative using the chain rule:

$$\frac{\partial P}{\partial w_2} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial p_2}\frac{\partial p_2}{\partial w_2}$$

We can do the same thing for $w_2$:

$$\frac{\partial P}{\partial w_1} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial p_2}\frac{\partial p_2}{\partial y}\frac{\partial y}{\partial p_1}\frac{\partial p_1}{\partial w_1}$$

We can rewrite these two products as:

$$\frac{\partial P}{\partial w_2} = \frac{\partial p_2}{\partial w_2}\frac{\partial z}{\partial p_2}\frac{\partial P}{\partial z}$$
$$\frac{\partial P}{\partial w_1} = \frac{\partial p_1}{\partial w_1}\frac{\partial y}{\partial p_1}\frac{\partial p_2}{\partial y}\frac{\partial z}{\partial p_2}\frac{\partial P}{\partial z}$$

Now we can compute the derivatives, in this example, the resulting performance $P$ is:

$$P = \frac{1}{2}(d-z)^2$$

And thus we can compute

$$\frac{\partial P}{\partial w_2} = \frac{\partial p_2}{\partial w_2}\frac{\partial z}{\partial p_2}(d-z)$$

$p_2$ is simply $p_2 w_2$ so we get:

$$\frac{\partial P}{\partial w_2} = y \frac{\partial z}{\partial p_2}(d - z)$$

Finally the derivative of the sigmoid function is simple:

$$\frac{\partial P}{\partial w_2} = y(1 - \sigma(\alpha))\sigma(\alpha)(d - z)$$

In this case the output of the $\sigma$ is $z$ so:

$$\frac{\partial P}{\partial w_2} = y(1 - z)z(d - z)$$

Now if we look at the derivative for $\frac{\partial P}{\partial w_1}$, it turns out that the last two elements we needed to compute, was the same as the last two elements in the computation of $\frac{\partial P}{\partial w_2}$! So if we make a neural network, where each "column of neurons" or layer is densely connected, i.e. each output from the previous layer connects to each input from the next layer, then even though the amount of connections increases exponentially, the computation of the derivatives do not!

The thing to note here, is that the output of layer $i$ has to go through layer $i + 1$ in order to affect the performance indicator. So the derivative for layer $i$ can re-use computation from the derivative of $i + 1$. So the amount of work we are gonna have to do will be:

- Linear in depth
- With respect to width it, it will be proportional to the number of connections and thus depends on $w^2$

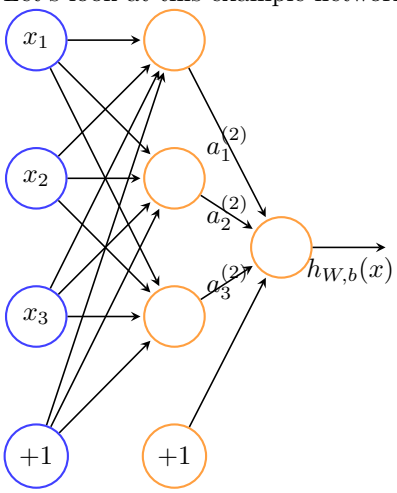
This is the foundation for the backpropagation algorithm, and why neural networks can efficiently learn.

## 2.2 Notation

- $n_l$ is the number of layers.
- $s_l$ is the number of nodes in layer $l$ (not counting the bias unit)
- $L_l$ is layer $l$.
- $L_0$ is the input layer, $L_{n_l}$ is the output layer
- $W_{ij}^{(l)}$ is the weight associated with the connection between unit $j$ in layer $l$ and unit $i$ in layer $l + 1$
- $b_i^{(l)}$ is the bias associated with unit $i$ in layer $l + 1$
- $a_i^{(l)}$ is the activation (or the output value) of unit $i$ in layer $l$. For $l = 1$ we use $a_i^{(1)} = x_i$

- $z_i^{(l)} = \sum_{j=1}^{n} W_{ij}^{(l)} x_j + b_i^{(l)}$ is a convenience notation such that $a_i^{(l)} = \phi(z_i^{(l)})$

Let's look at this example network:




This neural network represents the following computation:

$$
\begin{aligned}
a_1^{(2)} &= \phi\left(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}\right) \\
a_2^{(2)} &= \phi\left(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}\right) \\
a_3^{(2)} &= \phi\left(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}\right) \\
h_{W,b}(x) = a_1^{(3)} &= \phi\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)}\right)
\end{aligned}
$$

If we extend the activation function, to work on vectors such that $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$ then we can write the previous equation in a more compact fashion:

$$
\begin{aligned}
z^{(2)} &= W^{(1)} x + b^{(1)} \\
a^{(2)} &= \phi\left(z^{(2)}\right) \\
z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)} \\
h_{W,b}(x) = a^{(3)} &= \phi\left(z^{(3)}\right)
\end{aligned}
$$

Which can be generalized to:

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)} a^{(l)} + b^{(l)} \\
a^{(l+1)} &= \phi\left(z^{(l+1)}\right)
\end{aligned}
$$

The most common neural networks, are $n_l$-layered networks where $L_1$ is the input, $L_{n_l}$ the output and $L_i$ is densely connected to $L_{i+1}$. In order to compute

the output, we would simply have to compute the activation of $L_2$, $L_3$ etc. up to layer $L_{n_l}$, this is an example of a **feedforward** neural network, as there are no loops or cycles.

## 2.3 Backpropagation

Suppose we have a training set $D = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, we can then train our neural network with batch gradient descent, with the cost function for single sample as:

$$J(W, b; x, y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$

This is simply a (one-half) squared-error cost function. The overall cost function is then defined to be:

$$J(W, b) = \frac{1}{|D|}\sum_{i=1}^{|D|}\left(\frac{1}{2}\|h_{W,b}(x_i) - y_i\|^2\right)$$

If we add a weight decay term for regularization, then it becomes:

$$J(W, b) = \frac{1}{|D|}\sum_{i=1}^{|D|}\left(\frac{1}{2}\|h_{W,b}(x_i) - y_i\|^2\right) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

Our goal now, is to minimize $J(W, b)$. To train our neural network, we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero. If they are all the same value (e.g. 0) then they will end up learning the same function such that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \ldots$ for any input $x$.

Each iteration of gradient descent then updates the parameters $W, b$ as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \eta\frac{\partial}{\partial W_{ij}^{(l)}}J(W, b)$$

$$b_i^{(l)} = -\eta\frac{\partial}{\partial b_i^{(l)}}J(W, b)$$