1 Subjects

- Backpropagation
- Deep Nets
- PCA
- Autoencoder

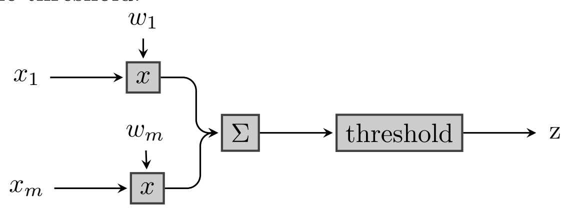
2 Notes

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 $x_1, \dots, x_n$ 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:

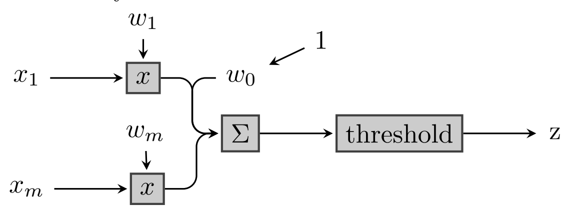
$$\Delta w = \eta \left(\frac{\partial P}{\partial w_1} i + \frac{\partial P}{\partial w_2} j \right)$$

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:

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

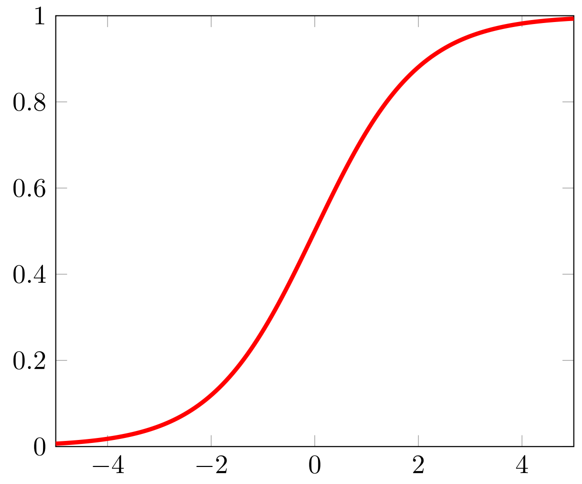
So what he proposed instead, was adding the bias input to each neuron w_0 which is always set to 1.



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 α is very big, and 0 if α is very small.



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

$$\begin{tikzcd}
 & w_1 \arrow[d] & & w_2 \arrow[d] & & \\
x_1 \arrow[r] & \boxed{x} \arrow[r, "P_1"] & \boxed{\text{sigmoid}} \arrow[r, "\text{y}"] & \boxed{x} \arrow[r, "P_2"] & \boxed{\text{sigmoid}} \arrow[r] & \text{z}
\end{tikzcd}$$

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 σ 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

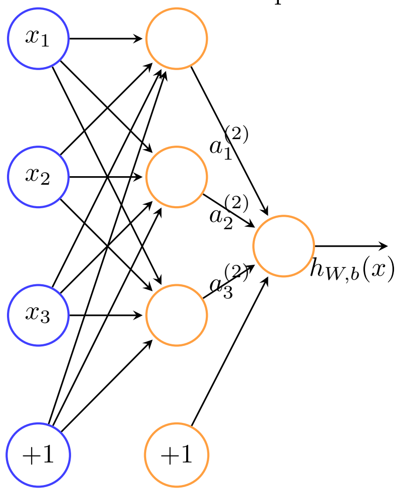
This is the foundation for the backpropagation algorithm, and why neural networks can efficiently learn.

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 $D = \{(x_1, y_1), \dots, (x_m, y_m)\}$, 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 $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \dots$ for any input x.

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

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

Seems simple right? But how do we do this efficiently? The backpropagation algorithm is the key. The intuition is that given some training example (x, y), we want to do a "forward pass" which computes all the activations throughout the network, including the output value. Then for each node i in layer l, we compute an "error term" $\delta_i^{(l)}$ which measures how much that node was "responsible" for any errors in the output.

For $\delta_i^{(nl)}$ we can simply compute the difference between the activation and the true target label y. For hidden units, we don't have the "correct answer" so instead we compute $\delta_i^{(l)}$ based on weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as input. So in detail:

1. Perform a feedforward pass, computing the activations for layers $L_2, L_3, \ldots, L_{n_l}$
2. For each output unit i in layer n_l, set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\|y - h_{W,b}(x)\|^2 = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\left(\sqrt{(y_1 - z_1^{(n_l)})^2 + \cdots + (y_i - z_i^{(n_l)})^2 + \cdots + (y_{s_l} - z_{s_l}^{(n_l)})^2}\right)^2$$
$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot \phi'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, \ldots, 2$ For each node i in L_l set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \phi'(z_i^{(l)})\right)$$

4. We can now compute the desired partial derivatives, as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

We can then compute the derivatives with respect to the whole data-set, for the overall cost function, as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[\frac{1}{|D|} \sum_{i=1}^{|D|} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x_i, y_i)\right] + \lambda W_{ij}^{(l)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x_i, y_i)$$

Numerical differentiation takes one evaluation of the neural net per weight ($\mathcal{O}\left((\#weights)^2\right)$), now with back propagation, we simply use each edge in each pass $\mathcal{O}(\#weights)$.

2.4 Deep Nets

Deep nets are when we have many layers of neurons. Each layer transforms the input into a "better" representation, and we will just repeat this procedure, and keep getting "better" representations. Historically, this has seemed to failed, this could either be because of a bad hypothesis or an issue with the learning algorithm.

One of the approaches to fixing this is better architecture, for example the convolutional network. Others are much more data, or computing power or do find better weights before we do backpropagation or maybe the field just needs to mature.

2.4.1 Convolutional networks

These are useful for, for example image recognition. The building blocks of a convolutional network are as follow:

1. Convolution
2. Non Linearity
3. Pooling
4. Classification (fully connected layer)

2.4.2 Convolution

First of all, convolution is like sliding a window over the image and applying some filter to it. For example we could have a simple "filter", "kernel" or "feature detector" which is simply a $m \times n$ matrix which slides over the image. For example the matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

might slide over the input:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 3 \end{pmatrix}$$

Since the feature detector is 2×2, it will look at first top-left 4 numbers, then it will slide 1 pixel right and look at the top-right numbers, then the bottom-left and then the bottom right. Producing the following "feature map":

$$\begin{pmatrix} 1*1+0*0+0*0+1*2 & 1*0+0*1+0*2+1*0 \\ 1*0+0*2+0*1+1*0 & 1*2+0*1+0*1+1*3 \end{pmatrix} = \begin{pmatrix} 3 & 0 \\ 0 & 5 \end{pmatrix}$$

The size of the "feature map" is controlled by three parameers:

- Depth corresponding to the number of filters
- Stride is the number of pixels or indices we move the feature detector with. In the above example it was 1
- Zero-padding, we might zero-pad in order to be able to slide the window around the edges of the matrix as well. Zero-padded convolution is also called *wide convolution* while non-zero-padded is called *narrow convolution*

2.4.3 Non-linearity

The non-linearity operation aims to introduce some non-linearity into the data, as it real world data is often non-linear. An example of such an operation is the ReLU opeartion, which simply sets any negative elements to 0. The result is the Rectified Feature Map

2.4.4 Pooling

Pooling reduces the dimensionality of each feature map but retains the most important information. Pooling can be done in a number of different way like Max, Average, Sum etc. It runs a window over the rectified feature map, and runs some function on the entries in the window. It might, for example, output the maximum value in the window, resulting in a smaller feature map.

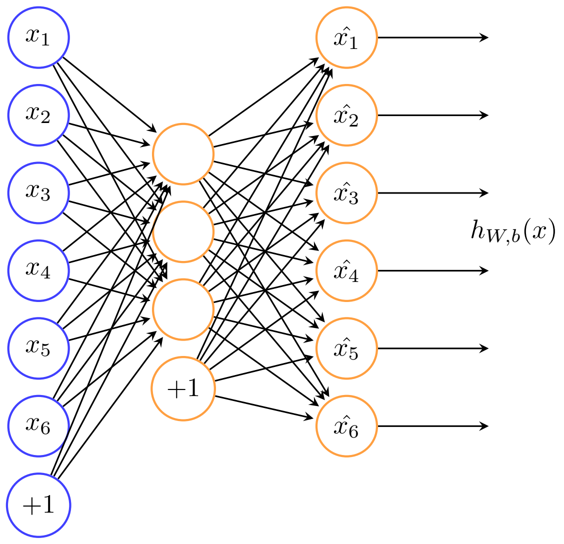
2.4.5 Fully connected layer

The result of one or more convolution and poolings, can then be put through the fully connected layer which will try to use the information to perform e.g. classification, as usual.

2.5 PCA

PCA tries to take some data in dimension d in to some dimension k. A neural net autoencoder can come up with something very similar to this.

2.6 Autoencoder



This is an auto-encoder: The autoencoder tries to learn a function $h_{W,b} \approx x$, i.e. it takes the input and tries to re-create it. This can be useful, as it effectively tries to learn what features in the data is important if I have to know the difference between two inputs but I can only do it in e.g. half the space. So if there some of the input features are correlated, then this algorithm will be able to discover some of those correlations. In fact it often ends up learning a low-dimensional representation very similar to PCA's.

2.7 Regularization

We introduced weight decay regularization already, another good way to regularize neural networks is the simple “dropout” technique, which simply means that we ignore the output of some random neurons. This forces the hidden neurons to be robust and approcimates averaging of exponentially many models.

2.8 Finding a good local minimum

Our neural network is riddled with local minimum, so in order to find a good local minimum, we will simply start the gradient descent from many different starting points, so we end up in different local minimums and choose the best one, hoping that it is a global minimum.