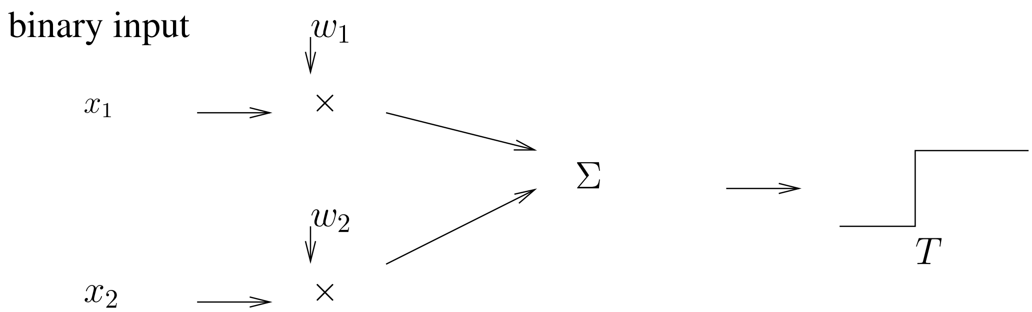
Neural Networks

ANN

- Long a curiosity
- 2012 paper, Hinton, image classification, 1000 categories, 60 million parameters
- neurones : axone, terminaison de l'axone, noyau, dendrites



What are we modeling?

1. all or none
2. cumulative influence
3. synaptic weight
4. (not) refractory period *(période réfractaire)*
5. (not) axonal bifurcation
6. (not) time patterns

So we have a model of a neuron (a collection of weights and thresholds).
But what about collections of neurons?

$$\text{inputs} \rightarrow (w, t) \rightarrow \text{outputs}$$

So we want $z = f(x, w, t)$. Training means adjusting w, t.
An ANN is a function approximator.

We want some desired function, $d = g(x)$.

Too simple example: perceptron

Perceptron is a supervised algorithm. Given inputs $D = \{(x_j, d_j)\}$, want classifier f.

We'll call $x_{j,i}$ the value of feature i in training datum j. We set $x_{j,0} \equiv 1$. This way we don't have to think about bias.

Perceptron algorithm:

$$\text{At each output, } f(x; w, b) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Initialise weights randomly.
- $\forall x_j \in$ inputs, compute output: $y_j(t) = f(w(t) \cdot x_j)$
- Update weights

A bit more detail on weight update:

$\boxed{\text{v1}}$

$$w(t+1) = \begin{cases} w(t) - r(t) & \text{if false positive} \\ w(t) & \text{if no error} \\ w(t) + r(t) & \text{if fail to recognise} \end{cases}$$

$\boxed{\text{v2}}$ ($d =$ desired, $y =$ observed) :

$$d(t) - y(t) = \begin{cases} -1 & \text{if false positive} \\ 0 & \text{if no error} \\ 1 & \text{if fail to recognise} \end{cases}$$

$\boxed{\text{v3}}$ $w_i(t+1) = w_i(t) + r(t) \cdot (d_j - y_j(t))x_{j,i}$ for all features i

The weight are updated at the last step after each training sample.

Converges if separable.

A perceptron with two inputs looks like this:

$$w_1 x_1 + w_2 x_2 = T$$

(where T is a threshold). So this is a line.

Show linear separability of logical operations except xor.

For xor:

$$\begin{cases} w_1 x_1 + w_2 x_2 & = w_1 0 + w_2 0 < T \\ w_1 x_1 + w_2 x_2 & = w_1 0 + w_2 1 \geqslant T \\ w_1 x_1 + w_2 x_2 & = w_1 1 + w_2 0 \geqslant T \\ w_1 x_1 + w_2 x_2 & = w_1 1 + w_2 1 < T \end{cases}$$

So

$$\begin{cases} w_2 1 & \geqslant T \\ w_1 1 & \geqslant T \\ w_1 + w_2 & < T \end{cases}$$

which is a contradiction.

We also call the separating plane (hyperplane) a decision boundary.
The function we also call a discriminant or decision function.
At the boundary, the decision function is zero.

Recall geometry: decision boundary is perpendicular to weight vector.

Proof. Consider x_1 and x_2 on the decision boundary. Then $f(x_1; w, b) = 0$ and $f(x_2; w, b) = 0$. So $w \cdot (x_1 - x_2) = 0$. □

> **Exercise:** Compute the weights for a perceptron that computes logical AND. If it doesn't feel easy, compute logical OR as well.

> **Exercise:** In python, implement (from scratch) a perceptron algorithm with (at least) two inputs. It should accept as input a matrix of training data (numpy `ndarray` is your friend) and an array of supervision classes. Push to your own git repo for the course.

Perceptron and MNIST

What other classifiers do we know?

- How would we do this with logistic regression?
- How would we do this with CARTs ?

Talk about OvO, OvA.

This just introduces voting…

Discuss: Why is it important that weights are non-binary here?

Training a single neuron

Let's start with a performance function $P = \|d - z\|$.
Why don't we like that?

So instead use $P = \|d - z\|^2 = (d - z)^2$.

(Draw w_1-w_2 plot with isoclines. Show four candidate steps. Mention 60 million parameters in Hinton's model. Talk about exponential explosion with number of weights.)

So we can follow the gradient:

$$\frac{\partial P}{w_1} \quad , \quad \frac{\partial P}{w_2}$$

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

What goes wrong?

Computer scientists were stuck on this for a quarter century. Then Paul Werbos showed in his 1974 PhD dissertation how to train neural networks using back-propagation of errors.

Two problems:

1. Thresholds are annoying. We don't want $z = f(x, w, T)$ but $z = f(x, w)$.

So let's add an extra weight w_0 and input $x_0 = -1$. Then set $w_0 \equiv T$.

(Show that this moves the threshold step to 0. So we don't have to pay attention to it anymore.)

2. We're using a step function, which isn't continuous.

Let's smooth it out. Use sigmoid function.

$$\beta = \frac{1}{1 + e^{-\alpha}}$$

(Walk through how to graph this.)

Summary: What have we done ?

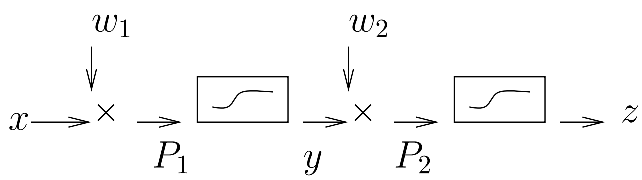
- Built a neuron
- Described how to do gradient descent
- Modified our activation function to be compatible with gradient descent

Training two neurons

- 1 neuron isn't a net

- 2 neurons is a net

This is a simplest possible neural network.



$$\text{Then performance function is} \quad P = \frac{1}{2}(d - z)^2$$

So we want to do gradient descent, this means computing partial derivatives.

$$\frac{\partial P}{\partial w_2} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial P_2}\frac{\partial P_2}{\partial w_2}$$

$$\frac{\partial P}{\partial w_1} = \frac{\partial P}{\partial z}\frac{\partial z}{\partial P_2}\frac{\partial P_2}{\partial y}\frac{\partial y}{\partial P_1}\frac{\partial P_1}{\partial w_1}$$

How do we compute these things?

$$\frac{\partial P_2}{\partial w_2} = y$$

$$\frac{\partial z}{\partial P_2} = z(1 - z) \quad \text{(come back to this)}$$

$$\frac{\partial P}{\partial z} = d - z$$

How do we compute the derivative of the sigmoid function?

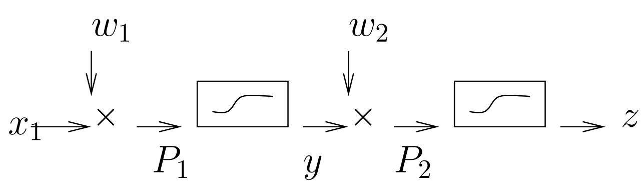
$$
\begin{aligned}
\frac{\partial \beta}{\partial \alpha} &= \frac{\partial}{\partial \alpha}\left(\frac{1}{1+e^{-\alpha}}\right) \\
&= \frac{\partial}{\partial \alpha}\left(1+e^{-\alpha}\right)^{-1} \\
&= -\left(1+e^{-\alpha}\right)^{-2} e^{-\alpha}(-1) \\
&= \frac{e^{-\alpha}}{1+e^{-\alpha}} \cdot \frac{1}{1+e^{-\alpha}} \\
&= \frac{\mathbf{1}+e^{-\alpha}-\mathbf{1}}{1+e^{-\alpha}} \cdot \frac{1}{1+e^{-\alpha}} \\
&= \left(\frac{1+e^{-\alpha}}{1+e^{-\alpha}} - \frac{1}{1+e^{-\alpha}}\right) \cdot \frac{1}{1+e^{-\alpha}} \\
&= (1-\beta)\beta
\end{aligned}
$$

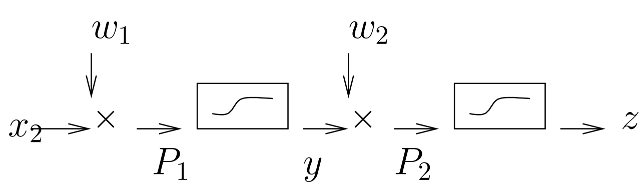
This is pretty cool: the derivative of the sigmoid function is independent of the input!

Note : at each phase, we only need to compute the last three factors.

Towards real life…

In real life, we have more than one of these going on at once. Sort of like this:





And then there are cross-overs with more weights, more multipliers. So we are again at risk of exponential blow-up: there are an exponential number of paths through the network.

But note that the dependencies are only by column, so lots of things are already computed.

In other words,

- Linear in depth
- Quadratic in width

> **Exercise:** Compute all of the partial derivatives in the 2×2 network above, assuming the interconnections we've shown.

Deep networks

We want to go from 2 parameters to millions.

- convolution
- pooling — max pooling
- kernel *(taking some neurons and running them across the image)*
- multiple kernels
- softmax
- dropout

We don't really know why any of this works.

Talk about auto-encoders.