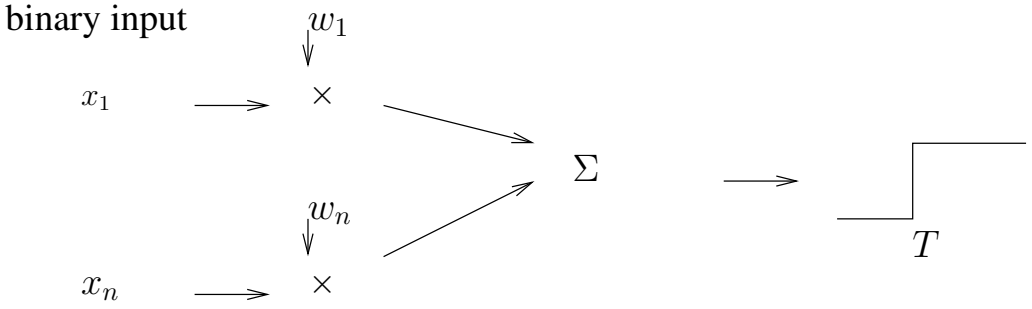
Let's look at the problem of recognising hand-written digits.
Good toy problem: practical, hard, simple.
It will take us the whole lecture to do a first draft on how to solve this problem.

Humans can do it, so maybe think about how we might do it.

Neural Networks

ANN

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



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

Let's take a quick detour into something that looks like a neural network, is very simple, but exhibits a different sort of complexity than our simple neuron.

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

We'll call $x_{j,i} = x_i^{(j)}$ 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:

v1

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

v2 ($d =$ desired, $y =$ observed) :

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

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.
Note we don't need a learning rate.

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.

Example: XOR is not linearly separable

Proof. If XOR is linearly separable, a perceptron can separate it. We will construct a two input, one output perceptron to compute XOR. We will show that this leads to a contradiction, and so our assumption that XOR is not linearly separable must be false.

Our perceptron's inputs are x_1 and x_2 and have associated weights w_1 and w_2 respectively. Then we have $y = w \cdot x + b$ and the output of the perceptron is

$$z = \begin{cases} 1 & \text{if } y = w \cdot x + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

Our training data is $D = \{(0,0),(0,1),(1,0),(1,1)\}$ and the desired output values are $d = \{0,1,1,0\}$.

Because I am lazy, I'd rather not learn the bias independently, so let $x_3 \equiv 1$ and when we set w_1 and w_2 to random values in $[0,1]$ to initialise learning, we also set w_3 to a similar random value. (We use random values because numerical analysts tell us that this provides better numerical stability.)

If we now agree that our vectors are now three-dimensional without changing their names, the perceptron's output may be written

$$z = \begin{cases} 1 & \text{if } y = w \cdot x > 0, \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We must now write $D = \{(0,0,1),(0,1,1),(1,0,1),(1,1,1)\}$.

Suppose that XOR is linearly separable. Then when we learn the final weights for our perceptron, we will have succeeded in separating the elements of D according to the values of d. Then we may write the four training cases of our simple perceptron (one line for each element of D and d respectively) thus:

$$\begin{aligned} z((0,0,1)) &= 0 \\ z((0,1,1)) &= 1 \\ z((1,0,1)) &= 1 \\ z((1,1,1)) &= 0 \end{aligned}$$

In other words, using (1), we can express the same system in terms of $w_1x_1 + w_2x_2 + w_3x_3$, and so we have

$$\begin{aligned} w_1 0 + w_2 0 + w_3 1 &\leqslant 0 \\ w_1 0 + w_2 1 + w_3 1 &> 0 \\ w_1 1 + w_2 0 + w_3 1 &> 0 \\ w_1 1 + w_2 1 + w_3 1 &\leqslant 0 \end{aligned}$$

Dropping the zero terms and remembering the multiplicative identity, this yields

$$w_3 \leqslant 0 \tag{2}$$
$$w_2 + w_3 > 0 \tag{3}$$
$$w_1 + w_3 > 0 \tag{4}$$
$$w_1 + w_2 + w_3 \leqslant 0 \tag{5}$$

Summing (3) and (4), we see that $(w_1 + w_2 + w_3) + w_3 > 0$. Since $w_3 \leqslant 0$ by (2), $-w_3 \geqslant 0$, and so we have

$$(w_1 + w_2 + w_3) > -w_3 \geqslant 0$$

We see then that (5) yields a contradiction, and so XOR is not linearly separable.

□

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. Separately, use it to classify a simple logic function. Push to your own git repo for the course.

Perceptron and MNIST

What other classifiers do we know?

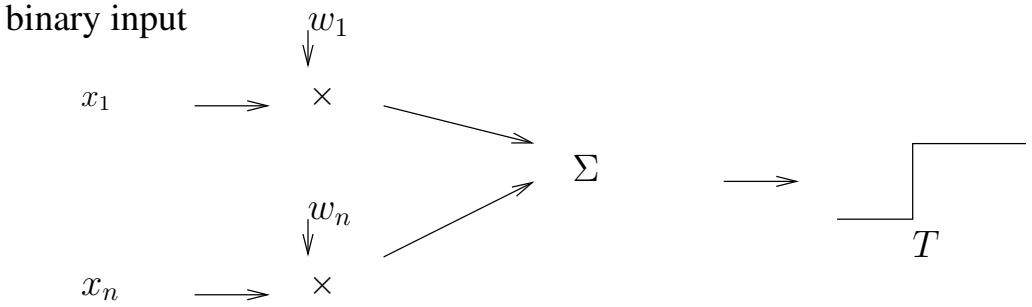
- How would we do this with logistic regression?
- How would we do this with CARTs ?

Talk about OvO, OvA.

This just introduces voting...

Note that many problems admit multiple solutions. The perceptron will pick one, but not necessarily the best. The "perceptron of optimum stability" (SVM) was designed to solve this problem.

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.)

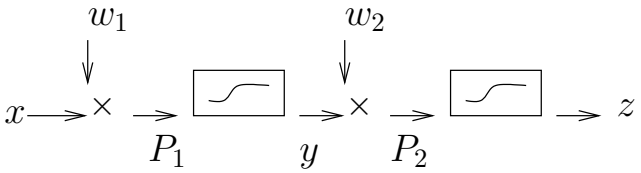
Summary: What have we done ?

- 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.



(Note that P_1 and P_2 are products.)

We talk about performance, loss, cost function:

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

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

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

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

How do we compute these things?

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

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

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

How do we compute the derivative of the sigmoid function?

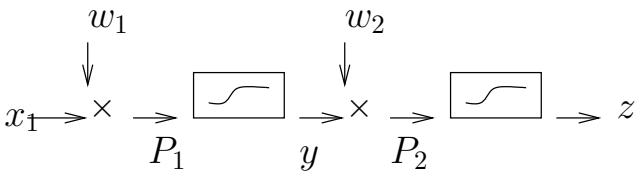
$$\begin{aligned}
\frac{\partial \beta}{\partial \alpha} &= \frac{\partial}{\partial \alpha}\left(\frac{1}{1+e^{-\alpha}}\right) \\
&= \frac{\partial}{\partial \alpha}\left(1+e^{-\alpha}\right)^{-1} \\
&= -\left(1+e^{-\alpha}\right)^{-2} e^{-\alpha}(-1) \\
&= \frac{e^{-\alpha}}{1+e^{-\alpha}} \cdot \frac{1}{1+e^{-\alpha}}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

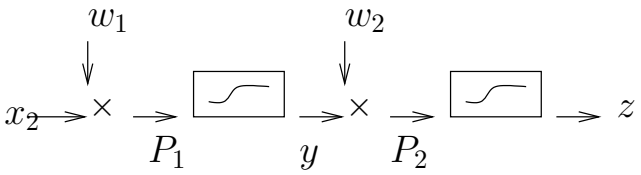
This is pretty cool: the derivative of the sigmoid function is independent of the input!

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.

MLP

This is our first step from 2 neurons to millions.

This is a real neural network. It's not a perceptron. Geoffrey Hinton apologised.

Back to computing hand-written digits.

8×8, then 16×16. Maybe 32×32, but let's do 28×28 for fun. We'll assume gray scale with activation in $[0, 1]$.

We'll use sigmoids to make intermediate activates also in $(0, 1)$.

Show hidden layers. Make 16 in each (somewhat arbitrary).

Then $784 \times 16 + 16 \times 16 + 16 \times 10 = 12960$ weights. Plus biases, so let's say 13K.

We might hope intermediate layers capture abstraction: say edges then loops. This time around, they don't.

Walk through backpropagation here. Note that ∇C tells us which changes matter the most.

Talk about inverse images of neuron outputs.

Talk about strange ways ANNs get things wrong (i.e., they don't recognise what we do, they can be confident about garbage).

Uber pedestrian accident

In March 2018 in Phoenix, an Uber self-driving car struck and killed a woman crossing the street with her bicycle.

- Radar detected woman 5.6 seconds before impact. SUV in far right lane, pedestrian crossing from the left. System classified as vehicle but didn't recognise that she was moving.
- Lidar pinged her several times over next few seconds, but classification repeatedly changed, so no continuity of recognition. It was a different object each time.
- At 1.5 seconds before impact, woman was partially in SUV lane, so system plotted course to steer around woman.
- Milliseconds later, identification as a bicycle on collision course. Abandons plan, which hadn't taken into account that bicycles move faster than pedestrians.
- Uber had disabled the emergency steering and braking systems because erratic. So SUV began gradual deceleration. At 1.2 seconds before impact, SUV was moving at 65 kph.
- One second later (0.2 seconds before impact), system alerted human safety driver that it had initated a controlled slowdown. Safety driver intervened, but 0.2 seconds is inadequate for a human. The SUV struck the woman, and then the safety driver engaged the brakes.
- She was not in a crosswalk, and the system had learned to identify pedestrians in crosswalks (a correlated feature).

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.