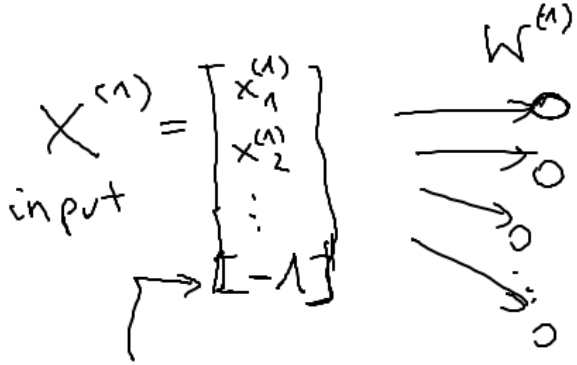


$W^{(2)}$ ... $W^{(n)}$ — weights of the layers

(we append -1 to encode the bias)

rows correspond to the neurons of the j-th layer

$$net^{(j)} = W^{(j)} \cdot X^{(j)} = \begin{bmatrix} W^{(j)} \\ \cdots \end{bmatrix} \begin{bmatrix} x^{(j)} \\ \cdots \end{bmatrix} = \begin{bmatrix} \\ \end{bmatrix}$$

— net output of the j-th layer, $j = 1, 2, \ldots, N$

$$Y^{(j)} = f[net^{(j)}] = \begin{bmatrix} f(net_1^{(j)}) \\ f(net_2^{(j)}) \\ \vdots \end{bmatrix}$$

— output of the j-th layer, $j = 1, \ldots, N$ (f — activation function)

$$X^{(j+1)} = \begin{bmatrix} Y^{(j)} \\ -1 \end{bmatrix}$$

— input of the (j+1)-th layer, $j = 1, 2, \ldots, N-1$

$\longrightarrow$ $Y^{(N)}$ = output of the network

Back-propagation phase:

We have $x^{(j)}$, $net^{(j)}$, $y^{(j)}$ from the forward pass.

the output was $y^{(N)} = \begin{bmatrix} \vdots \end{bmatrix}$, let us suppose that we expected $t = \begin{bmatrix} \vdots \end{bmatrix}$

$b^{(N)} = t - y^{(N)} = \begin{bmatrix} \vdots \end{bmatrix}$ - error of the Nth layer

—

$$\delta^{(j)} = \begin{bmatrix} b_1^{(j)} \cdot f'(net_1^{(j)}) \\ b_2^{(j)} \cdot f'(net_2^{(j)}) \\ \dots \end{bmatrix} \quad , j = N, N-1, \dots, 1$$

— ~~error~~ „delta signal"

$$\tilde{W}^{(j)} = W^{(j)} + c \begin{bmatrix} \vdots \end{bmatrix} \cdot \delta^{(j)} \cdot \left(x^{(j)}\right)^T \quad , j = N, N-1, \dots, 1$$

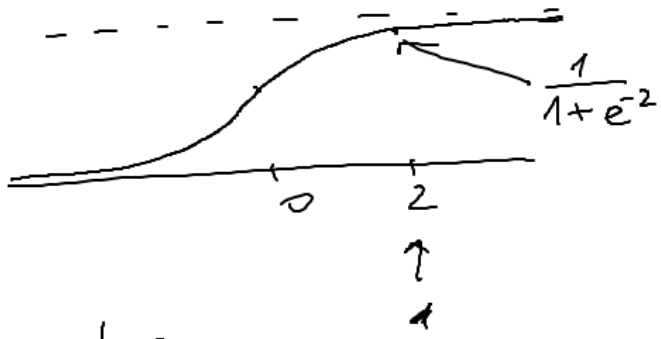
$\tilde{W}^{(j)}$ — weights after the adjustment

$$\tilde{b}^{(j)} = \left(W^{(j+1)}\right)^T \cdot \delta^{(j+1)} \quad , j = N-1, \dots, 1$$

$b^{(j)} = \tilde{b}^{(j)}$ with last (bottom) entry removed , $j = N-1, \dots, 1$

Say $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

$\sigma'(x) = \sigma(x)(1-\sigma(x))$



Suppose $net_1^{(j)}$ has a large absolute value

Then $\delta^{(j)} = \begin{bmatrix} b_1^{(j)} \cdot f'(net_1^{(j)}) \\ \vdots \end{bmatrix}$ — close to 0

$\tilde{W}^{(j)} = W^{(j)} + c \begin{bmatrix} b_1^{(j)} f'(net_1^{(j)}) \\ \dots \end{bmatrix} \cdot \begin{bmatrix} x_1^{(j)} & x_2^{(j)} & \dots \end{bmatrix} =$

$= W^{(j)} + c \begin{bmatrix} \text{will be close to 0} \\ \text{these are the adjustments for the weights} \\ \text{of the 1st neron} \\ \text{in the } j\text{-th layer} \\ \dots \end{bmatrix}$

This is called 'saturation'.

the choice of initial weights

- we know that we shouldn't choose the same weights for the neurons in the same layer
- so let us take (i.i.d) random weights, e.g. $w_{ij}^{(1)} \sim N(0, \sigma^2)$

then the net output of the first neuron is

$$net_1^{(1)} = \sum_j w_{1j}^{(1)} x_0^{(1)}$$

e.g. MNIST: $x_j^{(1)} \in [0,1]$, $j = 1, \dots, 28^2 = 784$

$x_{785}^{(1)} = -1$ $\quad N = 785$

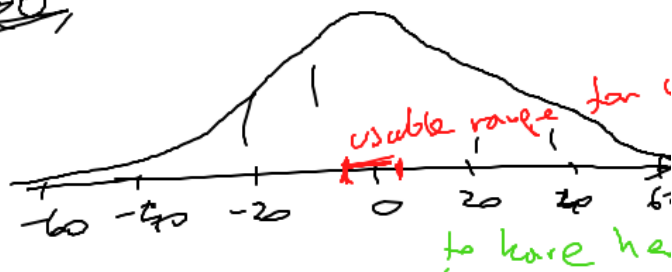
suppose half of inpts is 0 and half is 1

then $net_1^{(1)}$ = sum of more-less $\frac{N}{2}$ ranibles $\sim N(0, \sigma^2)$

$$\sim N\left(0, \frac{N}{2}\sigma^2\right) \approx (20\sigma)^2$$

If we tok $\sigma = 1$, then $net_1^{(1)} \sim N(0, 20^2)$

$\varphi'(net_1^{(1)})$ is likely

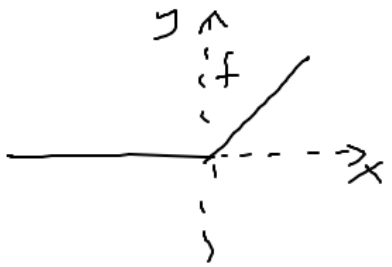


$\frac{N}{2}\sigma^2 = 1$ we would like $N(0,1)$ to have he

... or take different activation function (apart from the last layer?)

$f(x) = \max(x, 0)$

(ReLU)



Training process:

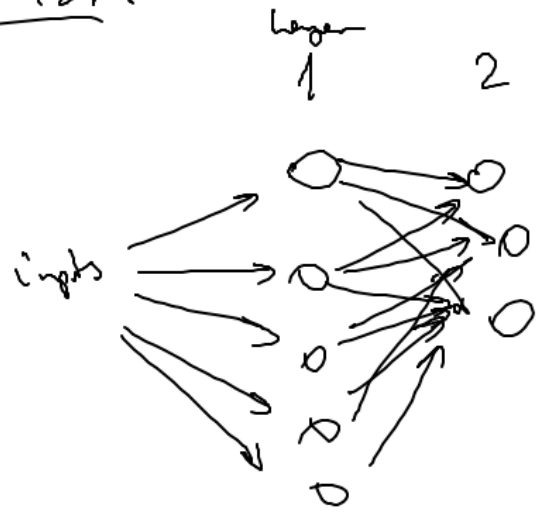
- take one sample, compute output, compare with the target value, use backpropagation to adjust the weight

- batch-learning:

  take N samples, compute outputs, compare with the target values, use backpropagation to compute the adjustments but do not change the weights

  afterwards adjust the weights by the sum/mean of the

So far:



dense layers

Linear Regression (supervised, continuous)

$f : \mathbb{R}^n \to \mathbb{R}$

$$f_\vartheta(x_i) = \begin{bmatrix} 1 & x_{i1} & \dots & x_{in} \end{bmatrix} \begin{bmatrix} \vartheta_0 \\ \vartheta_1 \\ \vdots \\ \vartheta_n \end{bmatrix} = \vartheta_0 + \sum_{j=1}^{n} \vartheta_j x_{ij}$$

↖ affine function ("linear")

model:

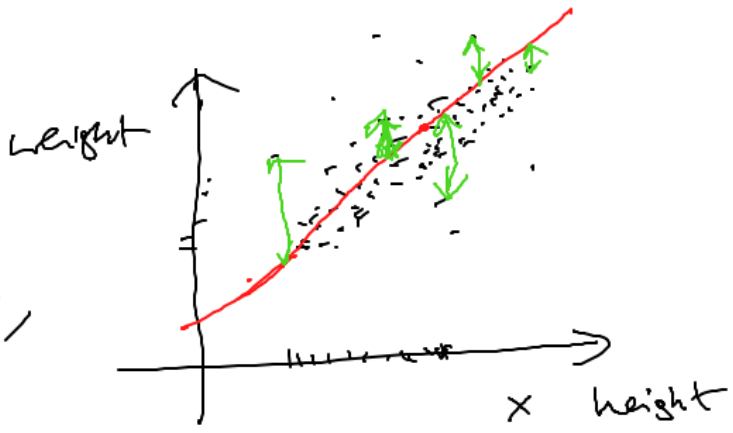
$$f(x_i) = f_\vartheta(x_i) + e_i \quad , i = 1, \dots$$

↑ error, e_i iid, mean 0, variance σ^2

$\vartheta_0, \dots, \vartheta_n$ – coefficients

given data (x_i, y_i), $i = 1, \dots$

x_i: inputs $\in \mathbb{R}^n$; y_i: output $\in \mathbb{R}$; $y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$



we want to find „the best" coefficients ϑ, so that the formula

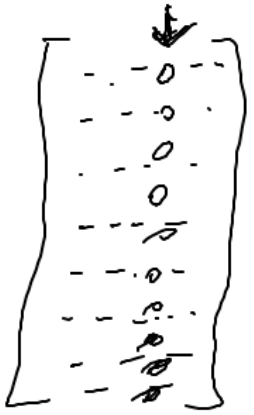
$$\hat{y} = f_\vartheta(x)$$

gives „the best" outputs.

One of the possible choices: minimise the square error:

$$L(y, \hat{y}) = \|y - \hat{y}\|_2^2 = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

When $X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \cdots & x_{2n} \\ \vdots & & \\ x_{N1} & \cdots & x_{Nn} \end{bmatrix}$ is of full rank



then there is a unique $\hat{\theta}$ minimising L:

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

(note: there are better ways to compute $\hat{\theta}$ numerically)

use scikit-learn.org (sklearn)

$$\begin{bmatrix} \cdots \\ \cdots \end{bmatrix} \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix}$$

To make the model more flexible, one may add some terms, e.g.

$$\underbrace{x_{ij} x_{ik}}_{\text{cross corr. terms}}, \quad x_{ij}^2, \quad x_{ij}^3, \ldots$$

For one variable: numpy : polyfit

$$P_\vartheta(x) = \vartheta_0 + \vartheta_1 \underbrace{x_{i,1}}_{\tilde{x}_{i,1}} + \vartheta_2 \underbrace{x_{i,1}^2}_{\tilde{x}_{i,2}} + \dots + \vartheta_n \underbrace{x_{i,1}^n}_{\tilde{x}_{i,n}}$$