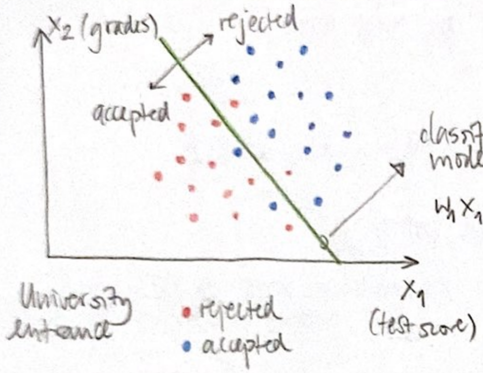
DEEP LEARNING NANODEGREE | Udacity

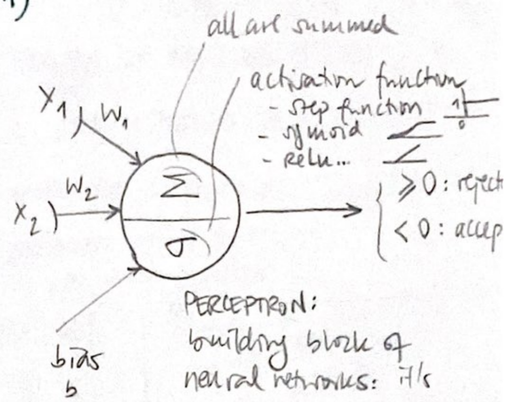
② NEURAL NETWORKS

INTRODUCTION TO NEURAL NETWORKS (LESSON 1)

CLASSIFICATION & PERCEPTRONS



$w_1x_1 + w_2x_2 + b = 0$



PERCEPTRON:
building block of neural networks: it's basically a classification model!

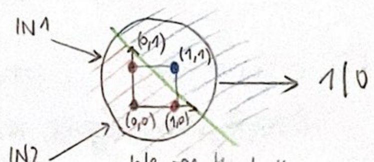
See CVND p. 16 - Perceptrons

Perceptrons and logical operators

- Logical operators can be modeled with perceptrons!

Example: AND

IN 1	IN 2	OUT
1	1	1 •
1	0	0 •
0	1	0 •
0	0	0 •



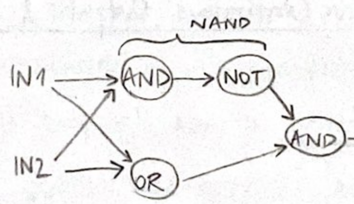
We see that there are actually many possible models...

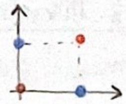
w_1	w_2	b
1	1	-1.5
1	1	-1.75
...		

- It's easy to create models for OR, NOT, AND
- For more complex operators, eg XOR, we combine other logical operators / perceptron classification models

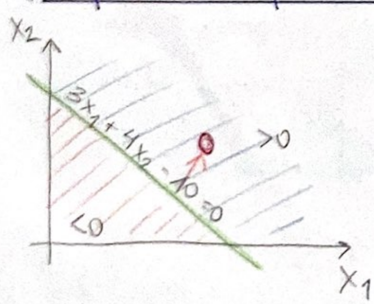
Example: XOR → only true if one is different

IN 1	IN2	OUT
1	1	0
1	0	1
0	1	1
0	0	0





Perceptron model optimization



$3x_1 + 4x_2 - 10 = 0$ model line

miss classified point O : (4,5)

↳ a fast method to move the model line towards the missclassified point consists in $\boxed{\text{substracting}}$ the point coords to the model params:

3	4	-10	model
− 4	5	1	point (1 for bias)
-1	-1	-11	new model

if move to ⊕ area, then ⊖, else ⊕

- But usually a learning rate is used for the point values to move slowly, eg:

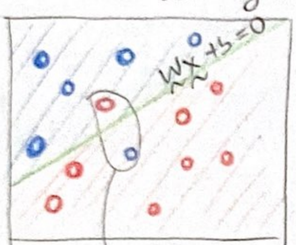
3	4	-10	model
− 0.4	0.5	0.1	point × 0.1 learning rate
2.6	3.5	-10.1]	⟶ new model, closer to missclasified point!

$2.6x_1 + 3.5x_2 - 10.1 = 0$

- If we want to move to negative area, we add instead of substracting!

—▷ PERCEPTRON ALGORITHM: it's based on the previous trick

Given:
- m samples of dimension n each
- Start with random model weights $w_1, \dots, w_n$ and b
- learning rate α



missclassified points

for every missclassified point $\underset{\sim}{x}$

if $y = \underset{\sim}{W}\underset{\sim}{x} + b < 0$ / predicition was 1/⊕ and we want to move to 0/⊖

$w_i = w_i + \alpha x_i$

$b_i = b_i + \alpha$

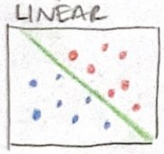
else if $y = \underset{\sim}{W}\underset{\sim}{x} + b < 0$ / predicition was 0/⊖ and we want to move to 1/⊕

$w_i = w_i - \alpha x_i$

$b_i = b_i - \alpha$

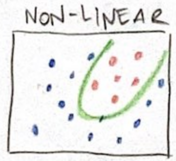
—▷ IMPORTANT: a perceptron is a linear classifier; if we want more complex classification regions, we need to use more perceptrons!

LINEAR



single neuron/perceptron

NON-LINEAR



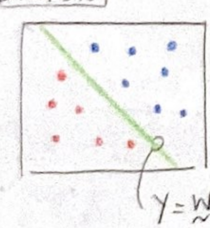
several neurons

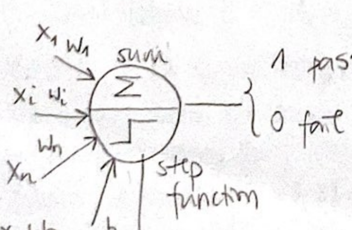
Error & Discrete vs Continuous Outputs & sigmoid function

—▷ $\sigma : \mathbb{R} \longrightarrow [0, 1]$ all real numbers mapped to $[0, 1]$

- Although classification is a DISCRETE problem (yes/no, dog/cat/rat), the error function is used to find the model parameters with gradient descend, and for that, we need CONTINUOUS outputs and error function: the sigmoid function achieves that.

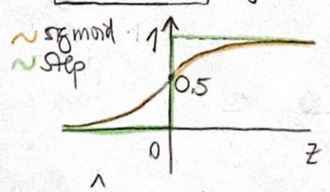
DISCRETE





$\hat{y} = \text{step}(\underset{\sim}{W}\underset{\sim}{x} + b)$

CONTINUOUS Sigmoid function



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$\hat{y} = \sigma(\underset{\sim}{W}\underset{\sim}{x} + b)$: $\hat{y}$ tells us the probability of pass (close to 1) or fail

Softmax function for 2+ classes

$$\begin{array}{l} \\ \text{class 1} \\ \text{class 2} \\ \text{class 3} \end{array} \begin{array}{c} \hat{y} \\ \begin{bmatrix} 2.6 \\ 0.5 \\ -3.7 \end{bmatrix} \end{array} \xrightarrow{\text{SOFTMAX}} \begin{array}{c} p \\ \underbrace{\begin{bmatrix} 0.87 \\ 0.10 \\ 0.03 \end{bmatrix}}_{\text{sum} = 1} \end{array}$$

$$\frac{e^{z_i}}{\sum e^{z_i}} : \text{soft max}$$

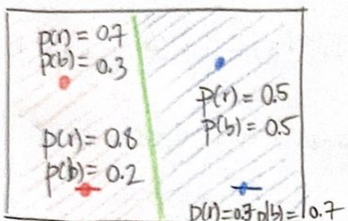
SOFTMAX converts the outputs in PROBABILITIES

Softmax for 2 classes is equivalent to the sigmoid!

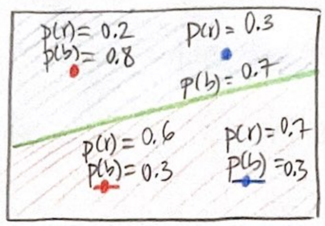
Note: usually the one-hot encoding is used for multi-class problems: for an input, the output has the size of the # classes.

Maximum Likelyhood

A



B



Model A is clearly better. We can check that numerically with the maximum likelyhood method

↓

We computing a model, we're finding the one that has the biggest maximum likelyhood.

$p(\text{red}) = 1 - p(\text{blue})$

MODEL A: maximum likelihood of all points ≐ probability of correct values

$p(\text{all}) = 0.7 \times 0.8 \times 0.5 \times 0.7$
$= 0.196$

MODEL B:

$p(\text{all}) = 0.2 \times 0.6 \times 0.7 \times 0.3$
$= 0.025$

→ the maximum likelihood of A is much higher ⇒ better model!

- The main issue with the product of probabilities is that they become too small
- Solution: we use the logarithm: $\log(ab) = \log(a) + \log(b)$

$\ln(ab) = \ln(a) + \ln(b)$ ← the natural logarithm is usually used, but we could take $\log_{10}$ too

$\ln(0.7 \times 0.9) = \underbrace{-0.36 - 0.1}_{\text{logarithm is always} < 0 \text{ in } [0,1]}$

Cross Entropy

Negative sum of the logarithms of the probabilities:

$$C.E. : -\sum_{\text{samples}}^{m} \ln(p_j) \rightarrow \left\{ \begin{array}{l} \uparrow\uparrow \text{ BAD Model} \\ \downarrow\downarrow \text{ GOOD Model} \end{array} \right\} \rightarrow$$

Now, our goal has become to minimize the Cross Entropy instead of maximizing the p(all)

$$C.E. = -\sum_{\text{samples}}^{m} \underbrace{y_j}_{\text{Event}} \ln(\underbrace{p_j}_{p(\text{Event})}) + \underbrace{(1-y_j)}_{\text{complementary event}} \ln(\underbrace{1-p_j}_{p(\text{complementary event})}) \; ; \; Y = \left\{ \begin{array}{l} 1 \\ 0 \end{array} \right\} \rightarrow$$

Note that for binary classification one term cancels!

*** Multi-class cross-entropy:**

$$C.E. = -\sum_{i}^{n} \sum_{j}^{m} \underbrace{y_{ij}}_{} \ln(p_{ij})$$

probability of event y_{ij}

i: classes, j: samples

→ event: $y_{ij} = 1$ if sample j belongs to class i
$\emptyset$ otherwise

this formula is equivalent to the previous!
→ in the previous (binary classification), a term is always canceled!

Error Function.

- The error function is basically the averaged cross entropy:

Binary: $E = -\frac{1}{m}\sum_{j}^{m} y_j \ln(p_j) + (1-y_j)\ln(1-p_j)$

Multiclass: $E = -\frac{1}{m}\sum_{i}^{n}\sum_{j}^{m} y_{ij}\ln(p_{ij})$

Since: $p = \sigma(\underset{\sim}{W}\underset{\sim}{x}+b) \rightarrow E(\underset{\sim}{W},b) = -\frac{1}{m}\sum_{j}^{m} y_j \ln(\sigma(\underset{\sim}{W}\underset{\sim}{x}+b)) + (1-y_j)\ln(1-\sigma(\underset{\sim}{W}\underset{\sim}{x}+b))$

- We are going to find the $\underset{\sim}{W}$ & b values that minimize this error with gradient descend

Gradient Descend

$f \longrightarrow \underset{\sim}{\nabla} f = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, -, \frac{\partial f}{\partial x_n}]$: Gradient of f: direction of maximum increase!

$E(\underset{\sim}{W}, b) = E(W_1, W_2, \dots, W_n, b)$

$\underset{\sim}{\nabla} E = [\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2}, -, \frac{\partial E}{\partial W_n}, \frac{\partial E}{\partial b}]$

We update:

$W_i' = W_i - \alpha \frac{\partial E}{\partial W_i}$

$b' = b - \alpha \frac{\partial E}{\partial b}$

α learning rate: small to avoid large uncontrolled steps

Gradient descend: we initialize the $\underset{\sim}{W}$ & b with random values and iteratively change them with the gradient until the error is small enough / # iterations max.

In a binary classification problem with sigmoids and cross-entropy error:

$\frac{\partial E}{\partial W_j} = \dots = -\sum_{i}^{m}(y_i - \hat{y}_i)\cdot x_j \cdot \frac{1}{m}$ $\underset{\sim}{x} = [x_1, \dots, x_j, \dots, x_n]$

$\frac{\partial E}{\partial b} = \dots = -\sum_{i}^{m}(y_i - \hat{y}_i)\cdot\frac{1}{m}$

↳ sum over all m samples

$$\longrightarrow \nabla E = -\sum_{i}^{m}\underbrace{(y_i - \hat{y}_i)}_{\text{scalar: diff truth - prediction}} \cdot \underbrace{[x_1, x_2, -, x_j, -, x_n, \underset{\text{bias}}{1}]}_{\underset{\sim}{x}_i\text{: point with n features}} \cdot \underbrace{\frac{1}{m}}_{\text{average}}$$

y_i: ground truth

$\hat{y}_i$: prediction, $\hat{y} = \sigma(\underset{\sim}{W}\underset{\sim}{x}+b)$

i: samples

j: features of each sample

Perceptron Algorithm vs Gradient Descend Algorithm:

They are basically the same, but gradient descend repels the classification boundary if correct classification

Gradient Descend

$$w_j \leftarrow w_j + \alpha \frac{1}{m} \sum_i^m \underbrace{(y_i - \hat{y}_i)}\, x_{ij}$$

if correct classification, NOTHING is done in that term!

Perceptron

$$w_j \leftarrow w_j + \alpha \frac{1}{m} \sum_i^m x_{ij}$$ if missclassified, nothing otherwise

the only thing missing is $(y_i - \hat{y}_i)$

which:

- $y_i - \hat{y}_i = 0$ if correct
- $y_i - \hat{y}_i = 1 / -1$

that's the only difference!

	Gradient descend	Perceptron
Correct classification	Repel further boundary	Do nothing
Miss classification	Attract boundary	Attract boundary

this makes sense: error becomes smaller and our region increases

See Jupyter Notebooks: Perceptrons, Gradient Descend, Student Admisions

Nonlinear Models: Multi-layer Perceptrons

See CVND p. 16

If we combine several neurons (Multi-layer perceptrons) we obtain NEURAL NETWORKS, which can handle NONLINEAR models.

We have

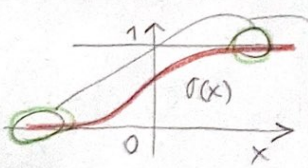
- input layer
- hidden layers: If we have several hidden layers, we have deep neural networks.
- output layer

Feed forward

Backpropagation

see p. 16 – 18, 19, 20 of the CVND.

IMPORTANT: keep in mind we are differentiating functions and applying the chain rule, thus, if a function has derivative almost $\emptyset$, the total error derivative will be $\emptyset$!
Therefore, we can easily run into numerical issues...



→ derivative is almost $\emptyset$ for large $|x|$ values!

This is called the VANISHING GRADIENT problem

IMPLEMENTING GRADIENT DESCEND (LESSON 2)

: This lesson is a recap + implementation in Numpy done step by step — so there's not much new in theory here, just the practical exercises + intuitive explanations

Different Error Functions

$$E_{\text{Cross-entropy}} = -\sum_{i}^{n}\sum_{j}^{m} y_{ij}\ln(\hat{y}_{ij})$$

i: classes, j: samples, y_{ij}: event: TRUE, $\hat{y}_{ij}$: output/probability of event: PREDICTION

$$\hat{y}_{ij} = P_{ij} = \sigma(\cdot) \in [0,1]$$

$$E_{MSE} = \underbrace{\frac{1}{m}}_{\text{mean}}\cdot\overbrace{\frac{1}{2}\sum_{i}^{n}\sum_{j}^{m}(y_{ij}-\hat{y}_{ij})^2}^{\text{Sum of Squared Errors}}$$

E_{MSE} = Mean Squared Error

i: classes, j: samples, y_{ij}: event: TRUE, $\hat{y}_{ij}$: prediction; probability PREDICTION

sample	y	class	$\hat{y}$
0	1	1	0.9
	0	2	0.1
	0	3	0
1	0	1	0.2
	1	2	0.6
	0	3	0.2
2		1	
		2	
		3	
⋮			

Gradient Descend (notes on the notebook for Student Admission)

$$\hat{y} = f(\underbrace{w, b}, x)$$

we are looking for the w & b values that minimize the error E

$$w_k \leftarrow w_k \boxed{-\eta\cdot\frac{\partial E}{\partial w_k}}$$

$$b \leftarrow b - \eta\frac{\partial E}{\partial b}$$

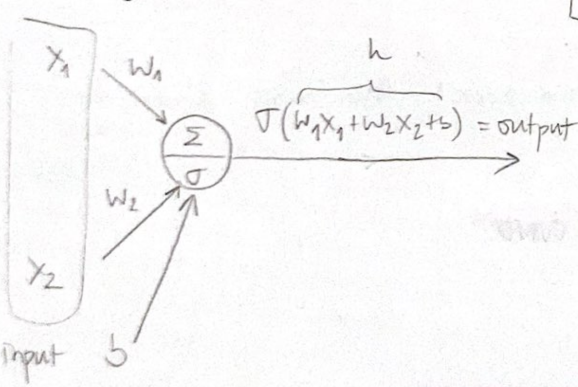
$$\Delta w_k = \eta\,\delta\,x_k \;:\; w_k \leftarrow w_k + \Delta w_k \quad \propto -\frac{\partial E}{\partial w_k} \text{ prop.}$$

η: learning rate, aka α

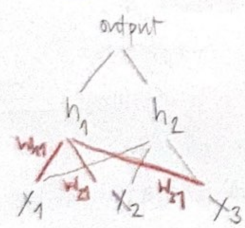
x_k: value of feature k

error term: $\delta = (y-\hat{y})\underbrace{f'(\sum w_k x_k)}$: Note: it contains already the '−' sign!!!

$f(h) = \sigma(h)$, $h = \sum w_k x_k$ — activation function



Multi-Layer Perceptron see OWND 16-19



$$\begin{array}{c|cc} & h_1 & h_2 \\ \hline x_1 & w_{11} & w_{12} \\ x_2 & w_{21} & w_{22} \\ x_3 & w_{31} & w_{32} \end{array}$$

$$f_j = \sigma(\overbrace{\sum_i w_{ij}x_i}^{h_j})$$

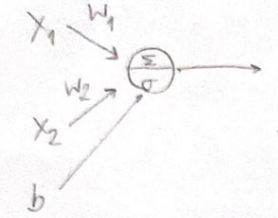
$$\sigma(\overbrace{W^T X}^{h}) = f$$

$$\sigma\left(\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32}\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right) = \sigma\left(\begin{bmatrix}h_1\\h_2\end{bmatrix}\right) = \begin{bmatrix}f_1\\f_2\end{bmatrix}$$

Note: Numpy

- Transpose of a matrix (defined as array): arr.T
- BUT, for 1-D arrays, we arr[:, None]

Backpropagation : a nice and intuitive explanation is intended here



① Example with a unique perceptron

$$\hat{y} = \overbrace{\sigma(\underbrace{w_1x_1 + w_2x_2 + b}_{h})}^{f(x)}$$

$$E = \frac{1}{2m}\sum_j (y_j - \hat{y}_j)^2 \quad \text{(MSE)}$$

→ Error term = difference prediction output multiplied by the derivative of the output function

$$\frac{\partial E}{\partial w_i} = -\frac{1}{m}\sum_j (y_j - \hat{y}_j) \cdot \underbrace{f'(\underset{\sim}{x})}_{\hat{y}_j'}$$

because $\frac{1}{m}$ is already in δ

(note: we're using MSE and sigmoid)

→ ERROR TERM : $\delta = (y - \hat{y}) \cdot \boxed{f'(\underset{\sim}{x})}$

learning rate, aka η

WEIGHT UPDATE: $w_i \leftarrow w_i + \frac{\alpha}{m}\sum_j \delta$ → average across all samples j of all error terms

NOTE: we are with a single perceptron neuron here...

$$f'(\underset{\sim}{x}) = \frac{\partial h}{\partial w_i} = \frac{\partial \hat{y}}{\partial w_i} = \frac{\partial \sigma(h(\underset{\sim}{w}, \underset{\sim}{x}))}{\partial w_i} =$$

$$= \frac{\partial \sigma}{\partial h} \cdot \frac{\partial h}{\partial w_i} = \sigma(h)(1-\sigma(h)) \cdot \frac{\partial h}{\partial w_i} = \underbrace{\sigma(\underset{\sim}{x})(1-\sigma(\underset{\sim}{x}))}_{\sigma'} \cdot x_i$$

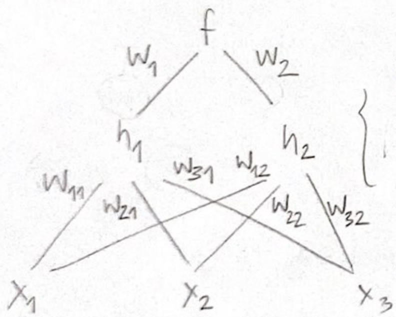
value of $\underset{\sim}{x}$ only in dimension i

$$h(\underset{\sim}{w}, \underset{\sim}{x}) = w_1x_1 + w_2x_2 + b = \underset{\sim}{w}^T \underset{\sim}{x} + b$$

σ': if we use another activation function, then this term is different.

② Example with a Multi-Layer Perceptron network:

Feedforward:

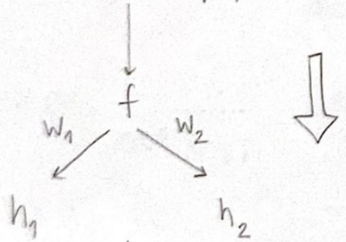


$$\hat{y} = f(h_1, h_2) = \sigma(w_1\sigma(h_1) + w_2\sigma(h_2))$$

$$= \sigma\left(w_1\sigma(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2\sigma(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)\right)$$

$$\begin{cases} h_1 = (w_{11}x_1 + w_{21}x_2 + w_{31}x_3) \longrightarrow \sigma(h_1) \\ h_2 = (w_{12}x_1 + w_{22}x_2 + w_{32}x_3) \longrightarrow \sigma(h_2) \end{cases}$$

Errors: $\delta = (y - \hat{y}) \cdot f'(\cdot)$, f' ≡ output function derivation of the node wrt weight we want to update



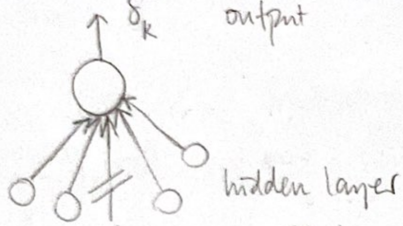
$$\delta_1 = \underbrace{\delta}_{\text{previous error}} \cdot \underbrace{w_1}_{\text{weight}} \underbrace{f'(h_1)}_{\text{derivative}} \qquad \delta_2 = \delta \cdot w_2 f'(h_2)$$

Therefore:
1) Bigger weights, bigger errors
2) Derivative must be considered carefully : not → ∞, not → 0

Backpropagation propagates the error backwards in the network! (analogous to propagating inputs forwards); Now the error is the input!

→ the output errors are propagate by being multiplied by: (1) weights (2) function derivative

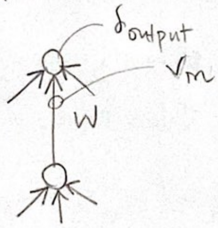
- For training/learning, we want to update the weights so that the error is minimized
- Backpropagation is gradient descend with the chain rule and it turns out it's like applying the weights backwards to the final error — hence, the name.
- We take the final error and basically scale it through the network multiplying it by the weights along the way: this gives the portion/contribution of each weight to the final error ⇒ the update each weight needs.



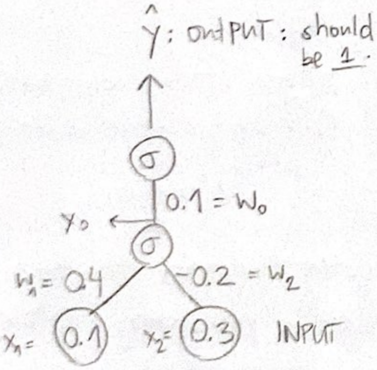
$$\delta_j^h = \sum \underbrace{W_{jk}}_{\text{weights between output and hidden layer}} \delta_k^o f'(h_j) \longrightarrow \underbrace{\Delta W_{ij}}_{\text{weights between inputs and hidden layer } h} = \alpha \underbrace{\delta_j^h}_{\text{error of hidden unit } j} \underbrace{x_i}_{\text{input unit values}}$$

- In general, for any level/layer:

$$\Delta W = \alpha\, \underbrace{\delta_{output}}_{\text{output error in node W points to}} \cdot \underbrace{V_{in}}_{\text{value through link with weight W}}$$



(3) Example exercise: compute ΔW for a simple network manually:



output
$$h = \sum W_i x_i = 0.4 \cdot 0.1 - 0.2 \cdot 0.3 = -0.02$$
$$f(h) = \sigma(h) = \frac{1}{1 + e^{+0.02}} = 0.495 = x_0$$
$$\hat{y} = f(W_0 x_0) = \sigma(W_0 x_0) = \frac{1}{1 + e^{-0.0495}} = 0.512$$

Error in output
$$\delta^o = (y - \hat{y}) \cdot f'(W_0 x_0)$$
$$= (y - \hat{y}) \cdot f(W_0 x_0)(1 - f(W_0 x_0)) =$$
$$= (1 - 0.512) \cdot 0.512 (1 - 0.512) = 0.122$$

Error in first layer:

and, the more we move backwards, the smaller they become

$$\delta_j^h = \sum_K W_{jK}\delta_K^o f'(h_j) = W_0\,\delta^o f'(h) = 0.1\cdot 0.122\cdot 0.495(1-0.495) = 0.003 = \delta^h$$

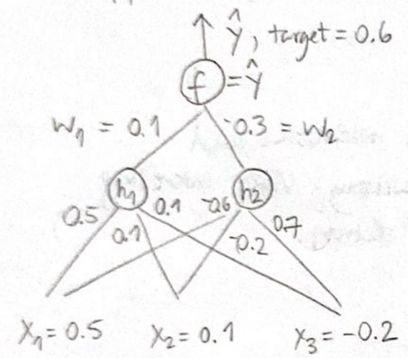
$$\Delta W_0 = \alpha\,\delta^o\cdot X_0 = 0.5\times 0.122\cdot 0.495 = 0.0302$$
$$\Delta W_1 = \alpha\,\delta^h\cdot X_1 = 0.5\times 0.003\times 0.1 = 0.00015$$
$$\Delta W_2 = \alpha\,\delta^h\cdot X_2 = 0.5\times 0.003\times 0.3 = 0.00045$$

we see we have VERY SMALL steps! That's related to the VANISHING GRADIENT problem, here fueled by the sigmoid function

weights in same layer have same error term!

other activation functions (eg. ReLU) alleviate this issue

④ Numpy exercise



$$\delta^o = (y-\hat{y})\cdot f',\quad f' = f(1-f) = \hat{y}(1-\hat{y})$$
$$\delta^h = w_1\cdot\delta^o\cdot f'(h_1) + w_2\,\delta^o f'(h_2) =$$
$$= w_1\,\delta^o f(h_1)(1-f(h_1)) + w_2\,\delta^o f(h_2)(1-f(h_2))$$

⑤ Backpropagation algorithm

- Error term in output layer: $\delta_K^o = (y_K - \hat{y}_K) f'(a_K)$
- Error term in hidden layer: $\delta_j = \sum [w_{ij}\delta_K]\cdot f'(h_j)$

Set
- input to hidden weights: $\Delta w_{ij} = 0$
- hidden to output weights: $\Delta W_j = 0$

repeat for e epochs

for each traing sample (a total of m samples):

$\hat{y} = \text{forward}(x)$

error term of output unit: $\delta^o = (y-\hat{y}) f'(z)$, $z = \sum_j W_j a_j$ ← input of the output unit

propagate errors to hidden layers:

$$\delta_j^h = \delta^o W_j\cdot f'(h_j)$$

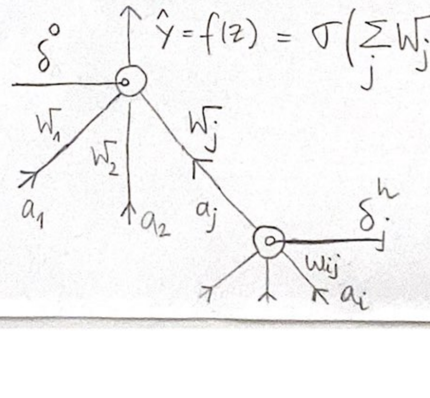
update weight steps

$$\Delta W_j = \Delta W_j + \delta^o a_j$$
$$\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$$

then:

$$W_j = W_j + \frac{\alpha}{m}\Delta W_j$$
$$w_{ij} = w_{ij} + (\alpha/m)\cdot\Delta w_{ij}$$

$$\hat{y} = f(z) = \sigma\Big(\underbrace{\sum_j W_j a_j}_{z}\Big)$$



TRAINING NEURAL NETWORKS ⟼ it's 1:1 the same as in the CVND: p. 20-24

SENTIMENT ANALYSIS by Andrew Trask: NLP PhD student at Oxford
author of Grokking Deep Learning

- We get a human generated text and want to know whether the content +/–.

→ Go to the repository and work there: deep-learning-v2-pytorch/sentiment-analysis-network
The section is divided in 6 mini-projects

25,000 movie reviews labelled as positive/negative sentiment

See Jupyter notebooks and repository!

Very interesting lesson: a neural network class is implemented from-the-scratch using numpy for sentiment analysis.
A basic workflow for NLP/text processing is shown.

DEEP LEARNING WITH PYTORCH ⟼ see the notebooks and the repository. VERY interesting lesson/section.

PROJECT: PREDICTING BIKE-SHARING PATTERNS