Neural networks (NN)

Consider $f(\vec{x};\vec{\theta}) = \vec{w}\cdot\vec{g}(\vec{x}) + b$, where $\vec{\theta} = (\vec{w}, b)$.

A natural idea is to construct features with their own parameters:

$$f(\vec{x};\vec{\theta}) = \vec{w}\cdot g(\vec{x};\vec{\theta}_2) + b, \text{ where}$$
$$\vec{\theta}_1 = (\vec{w}, b) \;\&\; \vec{\theta} = (\vec{\theta}_1, \vec{\theta}_2).$$

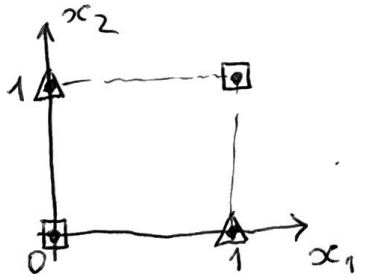
In deep neural networks (DNN), the functions are composed recursively:

	layer
$f_1(\vec{x};\vec{\theta}_1)$	1
$f_2(f_1(\vec{x};\vec{\theta}_1);\vec{\theta}_2)$	2
$\vdots$	$\vdots$
$f_\ell(f_{\ell-1}(\ldots);\vec{\theta}_\ell)$	ℓ

The XOR problem

x_1	x_2	y	
0	0	0	□
0	1	1	△
1	0	1	△
1	1	0	□

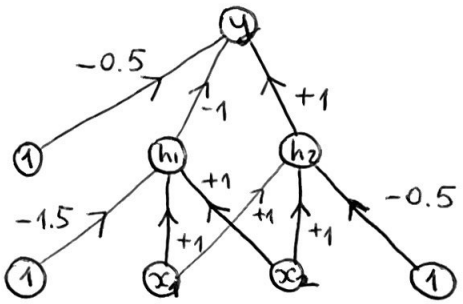
⇐ truth table



The XOR function is <u>not</u> linearly separable: no straight-line DB can separate △ from □.

However, we can use linear-DB functions to implement it anyway:

$$f(\vec{x};\vec{\theta}) = \mathbb{I}(\vec{w}\cdot\vec{x} + b \geq 0)$$ ← perceptron



] observed unit

] hidden units

] input

$h_1 = \mathbb{I}(x_1 + x_2 - 1.5 \geq 0)$ AND

If $x_1 = 1, x_2 = 1 \Rightarrow h_1 = 1$,
otherwise $h_1 = 0$.

$h_2 = \mathbb{I}(x_1 + x_2 - 0.5 \geq 0)$ OR

If $x_1 = 0, x_2 = 0 \Rightarrow h_2 = 0$,
otherwise $h_2 = 1$.

Finally, $y = \mathbb{I}(h_2 - h_1 - 0.5 \geq 0)$

x_1	x_2	h_1	h_2	y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

same as before (XOR)

Differentiable multi-layer perceptrons (MLP)

Define a differentiable activation function:

$$g: \mathbb{R} \to \mathbb{R} .$$

Make linear transformations of the inputs, or the outputs of the previous layer:

$$\vec{a}_\ell = \vec{b}_\ell + W_\ell \underbrace{\vec{z}_{\ell-1}}_{\text{or } \vec{x}}$$

(ℓ: layer index; dim of $\vec{a}_\ell$ = # nodes in layer ℓ)

Then apply the activation function to $\vec{a}_\ell$ element-wise:

$$g_\ell(\vec{a}_\ell) = g_\ell(\vec{b}_\ell + W_\ell \vec{z}_{\ell-1}) \equiv \vec{z}_\ell$$

($\vec{z}_\ell$: output of layer ℓ; same dim as $\vec{a}_\ell$)

In the final layer, the # nodes is dictated by the problem.

[Activation functions must be non-linear:]

if $g_\ell(a) = c_\ell a$,

$$f(\vec{x}; \vec{\theta}) = W_L C_L(W_{L-1} C_{L-1}(\dots (W_1 \vec{x})\dots) \sim$$

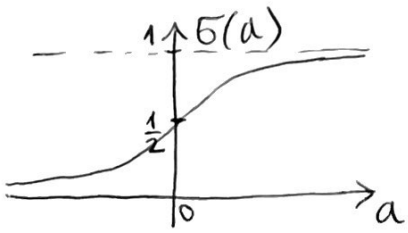
(L: total # layers)

$$\sim W_L W_{L-1} \dots W_1 \vec{x} \equiv \underbrace{W' \vec{x}}_{\text{linear model}}$$

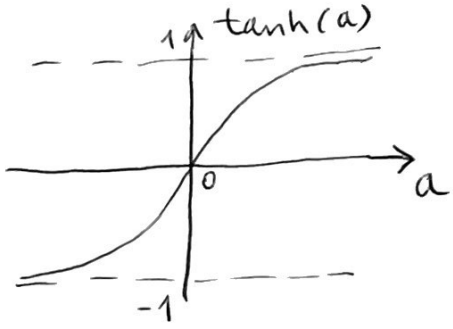
Activation functions

1. Sigmoid:

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



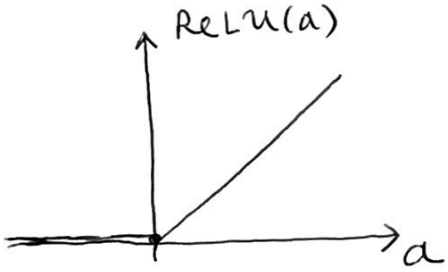
2. $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$



Both $\sigma(a)$ & $\tanh(a)$ saturate at $a \to +\infty$ and $a \to -\infty$ $\Rightarrow$ vanishing gradient problem

3. Rectified linear unit (ReLU):

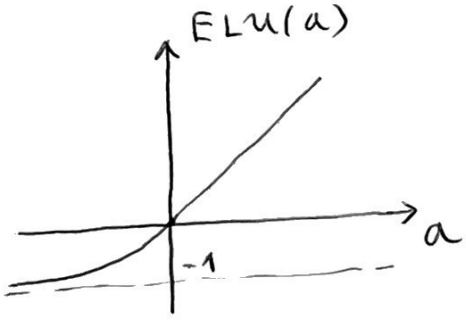
$$\mathrm{ReLU}(a) = \max(a, 0)$$



turns off negative inputs, but passes positive inputs unchanged

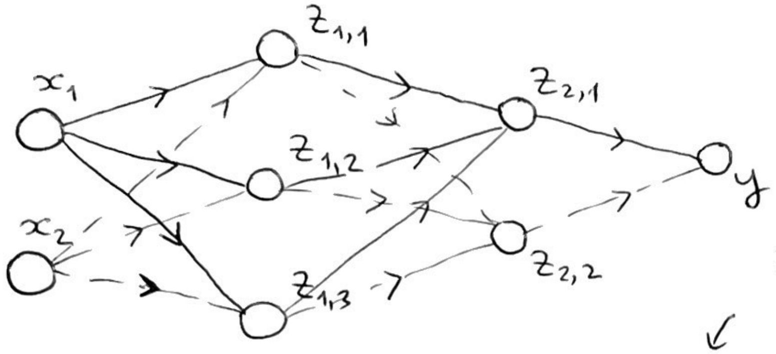
4. Exponential linear unit (ELU):

$$\mathrm{ELU}(a) = \begin{cases} a, & a \geq 0 \\ e^{a}-1, & a < 0 \end{cases}$$



Ex. $\vec{x} = (x_1, x_2)$ 2D input

output: prob. of class 1 (out of 2 classes) (possible)



[biases omitted for clarity]

interpreted as $p_1 = 1 - p_2$ - prob. of class 1

$\vec{x} \in R^2$ $\vec{z}_1 \in R^3$ $\vec{z}_2 \in R^2$ $y \in$ [redacted] $[0,1]$ with sigmoid activation function φ

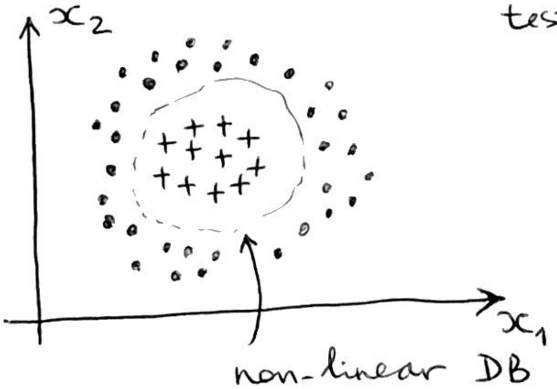
ℓ 1 2 3

Mathematically,

$$\begin{cases} \vec{z}_1 = \varphi(W_1 \vec{x} + \vec{b}_1), \\ \vec{z}_2 = \varphi(W_2 \vec{z}_1 + \vec{b}_2), \\ a_3 = \vec{w}_3 \cdot \vec{z}_2 + b_3 \end{cases} \Rightarrow p(y|\vec{x};\vec{\theta}) = \mathrm{Ber}(y|\sigma(a_3)).$$

$\vec{\theta} = (W_1, \vec{b}_1, W_2, \vec{b}_2, \vec{w}_3, b_3)$ need to be fitted to maximize $\log \mathcal{L}$.

Result:



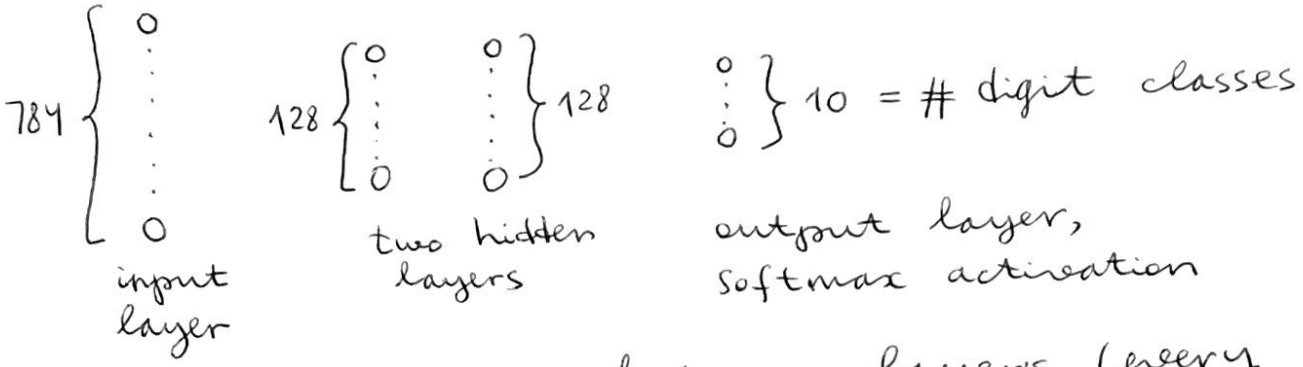
test set

MLP for image classification

Consider MNIST database of digits:

$\Rightarrow$ flatten into a vector with $D = 28^2 = 784$.

prms 100480 16512 1290



Dense connections between layers (every node in layer l connected to every node in layer $l+1$).

This model can reach test accuracy of ~97% after training for 2 epochs.

MLP for text classification

Idea: convert a variable-length sequence of words $\vec{v}_1, \ldots, \vec{v}_T$ into a fixed-size vector. (T ← length of the document)

$\vec{v}_j$ = 'one-hot' vector of length V,

V = vocabulary size.

The simplest way to interpret a text is to treat it as a 'bag of words', i.e. ignore word order.

(1.) Map each word to a <u>token</u> via preprocessing steps:

- drop punctuation
- convert all words to lowercase
- drop "a", "the", "and", etc. (stop word removal)
- replace conjugations with base forms, e.g. "running" → "run" (word stemming)

Tokens come from a vocabulary.

Now, consider x_t = token at location t in the document

Compute $\tilde{x}_v = \sum_{t=1}^{T} \mathbb{I}(x_t = v)$ ← token v

Now we have $\vec{\tilde{x}}$ = D-dim vector of token counts

Note that $D \leq V$ (vocab. size)

$\vec{\tilde{x}}$ is the <u>vector space model</u> of the text.

With N texts, we can construct a $D \times N$ (or $V \times N$ / $D_{max} \times N$, where D_{max} is the union of all D's) <u>term frequency matrix</u> $TF_{D \times N}$, where TF_{ij} = freq. of term i in document j

$$\sum_{i=1}^{D} TF_{ij} = 1 \quad , \forall j$$ each column is normalized

what to do with common (i.e., high-freq.) words? Define $\log(TF_{ij}+1)$ - log frequencies.

Also, define <u>inverse document frequency</u>:

$$IDF_i = \log \frac{N}{1+DF_i}, \text{ where } 1 \leq DF_i \leq N$$ ← in the Dmax scheme

is the number of documents with term i.

$$\begin{cases} DF_i = 1 \Rightarrow IDF_i = \log\left(\frac{N}{2}\right) \\ DF_i = N \Rightarrow IDF_i = \log\left(\frac{N}{1+N}\right) \to 0 \text{ as } N \to \infty \end{cases}$$

Thus, IDF_i can be used to penalize common words:

compute $\underbrace{TFIDF_{ij}}_{\text{TF-IDF matrix}} = \log(TF_{ij}+1) \times IDF_i$

The TF-IDF transform does not place semantically similar words together in vector space.

<u>Idea</u>: use <u>word embeddings</u>

$\vec{v}_t$ = one-hot vector of the token at position t in the document

$\dim \vec{V}$

is mapped into

$\vec{e}_t \in R^K$ = lower-dimensional dense vector ($K \ll V$)

$\dim K$

$$\vec{e}_t = \underbrace{E_{K \times V}}_{\text{embedding matrix, places semantically similar words together}} \vec{v}_t$$

With N documents, we have $\vec{e}_{nt} \in R^K$, can compute $\vec{e}_n = \sum_{t=1}^{T} \vec{e}_{nt}$

(n: document index, t: token index)

↑ bag of word embeddings, can be used as input to a NN classifier (document)

Word embeddings should be similar for semantically similar words.

Assumption: two words are similar if they occur in similar contexts (the distributional hypothesis).

Thus, it makes sense to learn the word's embedding vector from its context.

→ Latent semantic indexing (LSI)

Compute C_{ij} = # times token i occurs in document j ($j = 1, \ldots, N$)

$C_{M \times N}$ = count matrix

↑ vocab. size

Approximate C with rank K (truncated) SVD approximation:

$$C_{ij} \simeq \sum_{k=1}^{K} U_{ik} \sigma_k V_{jk}$$

Then $\vec{u}_i$ = embedding for word $i = 1, \ldots, M$
(length K)

$\vec{w}_j = \vec{s} \times \vec{v}_j$ = embedding for document $j = 1, \ldots, N$
($\times$: element-wise)

One can compute a 'bag of words' vector for each document:

$$\vec{q}_j = \frac{1}{T_j} \sum_{t=1}^{T_j} \vec{u}_t$$

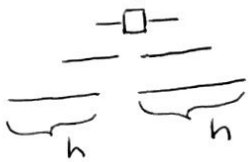
(T_j: # terms in document j)

and use cosine similarity to compare $\vec{q}_j$ with $\vec{w}_{j'}$:

$$\frac{\vec{q}_j \cdot \vec{w}_{j'}}{|\vec{q}_j||\vec{w}_{j'}|}$$

→ Latent semantic analysis (LSA)

Do the same SVD analysis as in LSI, but for each word i, define local neighborhoods:



} set of neighborhoods

Thus, C_{ij} = # times token i occurs in neighborhood j

h is a hyperparameter.

→ positive pointwise MI (PPMI)

Use $PPMI(i,j) = \max(PMI(i,j), 0)$,
(↑ removes negative correlations)

where $PMI(i,j) = \log \frac{p(i,j)}{p(i)\,p(j)}$

Do k-rank SVD decomposition on $PPMI(i,j)$ instead of C_{ij}.

→ word2vec

Finally, use word embedding as part of the NN architecture itself:

input = unordered bag of words $\vec{v}_1, \ldots, \vec{v}_T$ ('one-hot', $E \times V$, T ↖ document length)

(1) Embedding: $\vec{e}_t = \underbrace{W_1}_{\sim}\, \vec{v}_t \quad t = 1, \ldots, T$ 'do all words simultaneously'

(2) global average pooling: $\langle \vec{e} \rangle = \frac{1}{T} \sum_{t=1}^{T} \vec{e}_t$

(3) Hidden layer: $\vec{h} = g(\underbrace{W_2}_{H \times E} \langle \vec{e} \rangle + \vec{b}_2)$

(4) Binary classification: $p(y \mid \vec{v}_1, \ldots, \vec{v}_T) = \mathrm{Ber}(y \mid \sigma(\vec{w}_3 \cdot \vec{h} + b_3))$

This can be used to e.g. classify movie reviews into pos. & neg.

Ex. $V = 10^4$ words, $\underbrace{E}_{\text{embedding size}} = 16$, $\underbrace{\vec{H}}_{\text{hidden layer size}} = 16$

Layer	# prms	
1	16×10^4	} 160,289 in total
2	0	
3	$16 \times 16 + \overbrace{16}^{\text{biases}} = 272$	
4	$16 + \underbrace{1}_{\text{bias}} = 17$	

However, W_1 can be obtained ('pre-trained') separately using the techniques described above, to reduce the # prms & prevent overfitting.