Neural Networks
Deep Learning

Machine Learning course

Amaury Habrard & Marc Sebban

{amaury.habrard,marc.sebban}@univ-st-etienne.fr

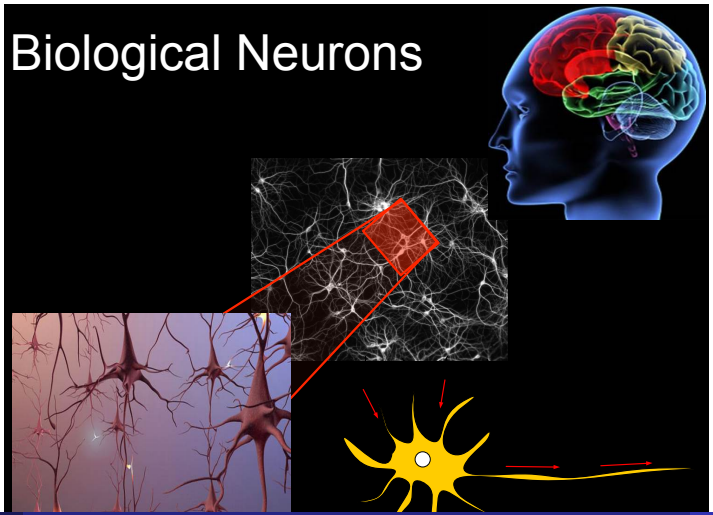
Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet Saint-Étienne

Semester 2

Sources

- These slides are mainly based on the course on Neural Networks from Hugo LaRochelle `http://info.usherbrooke.ca/hlarochelle/neural_networks/content.html`
- Some material are borrowed from Elisa Fromont or specific papers.
- Disclaimer: I am not an expert of the field, the objective of this class is to give the main ideas.

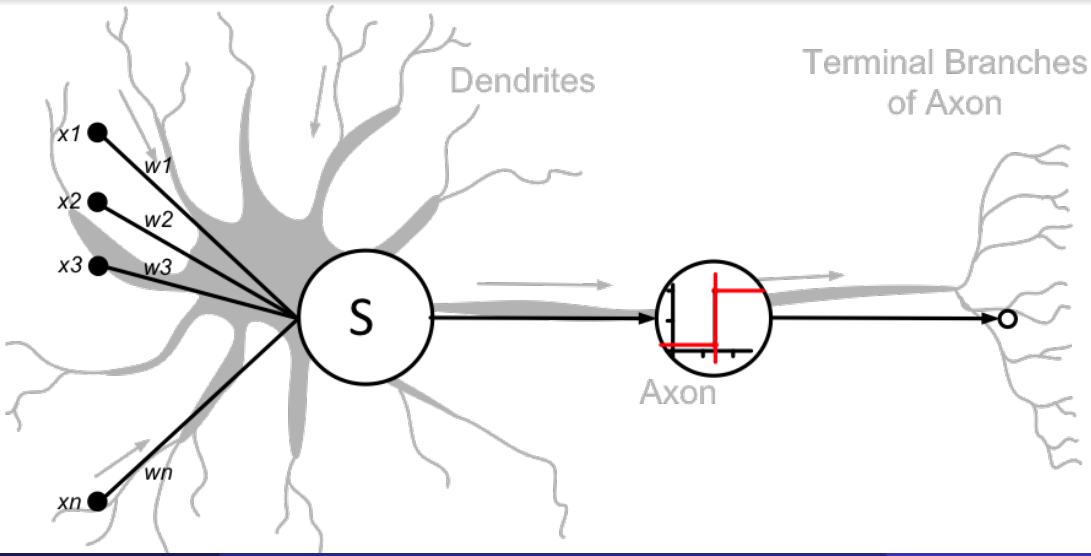
One of the most important (recent) breakthrough in machine learning



Motivation

Mimic biological neurons

- At "low level" the brain is composed of neurons
- A neuron receives an input from other neurons (maybe thousands) from its synapses
- Inputs are approximately summed
- When the input exceeds a threshold the neuron sends an electrical spike that travels from the body down to the axon to the next neuron(s)



A quick historical view

- History traces back to the 50's but became popular in the 80's with work by Rumelhart, Hinton, and Mclelland *A General Framework for Parallel Distributed Processing : explorations in the microstructure of cognition*
- Peaked in the 90' then "desert crossing".
- Today - A "revolution"
  - confusion btw machine learning and deep learning!
  - Hundreds of variants (thousands of research papers)
  - Less a model of the actual brain than a useful tool, but still some debate
- Numerous (successful) applications
  - Handwriting, face, speech recognition
  - Vehicles that drive themselves
  - Models of reading, sentence production, dreaming
- Debate for philosophers and cognitive scientists
  - Can human consciousness or cognitive abilities be explained by a connectionist model or does it require the manipulation of symbols/representation?

Outline

1. Feed-forward Neural Networks
   - Neural unit, capacity, multi-layer networks
2. Training neural network
   - Parameters, gradient, back-propagation
3. Regularization and tricks
4. Deep Learning

Feed-forward Neural Networks

Artificial Neuron/Perceptron

$\mathbf{x} \in \mathbf{R}^d$

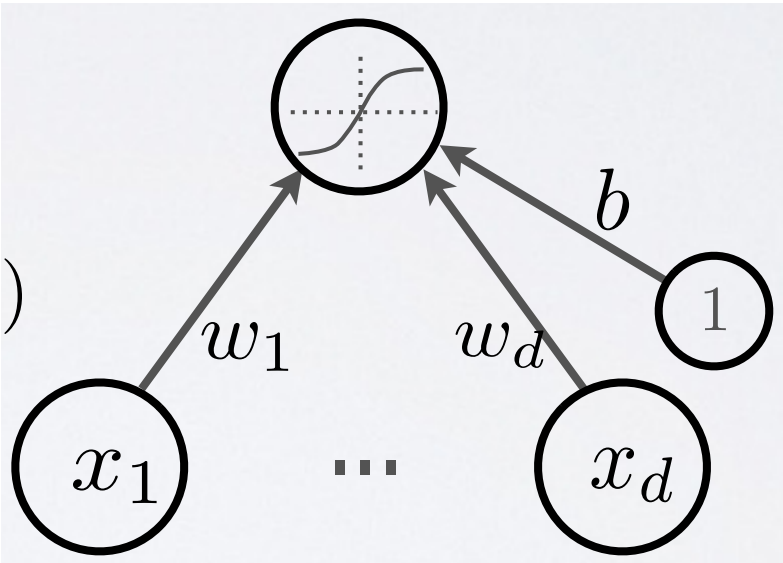
A linear machine (pre-activation or input activation)

$a(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b = \sum_{i=1}^{d} w_i x_i + b$

Neuron (output) activation

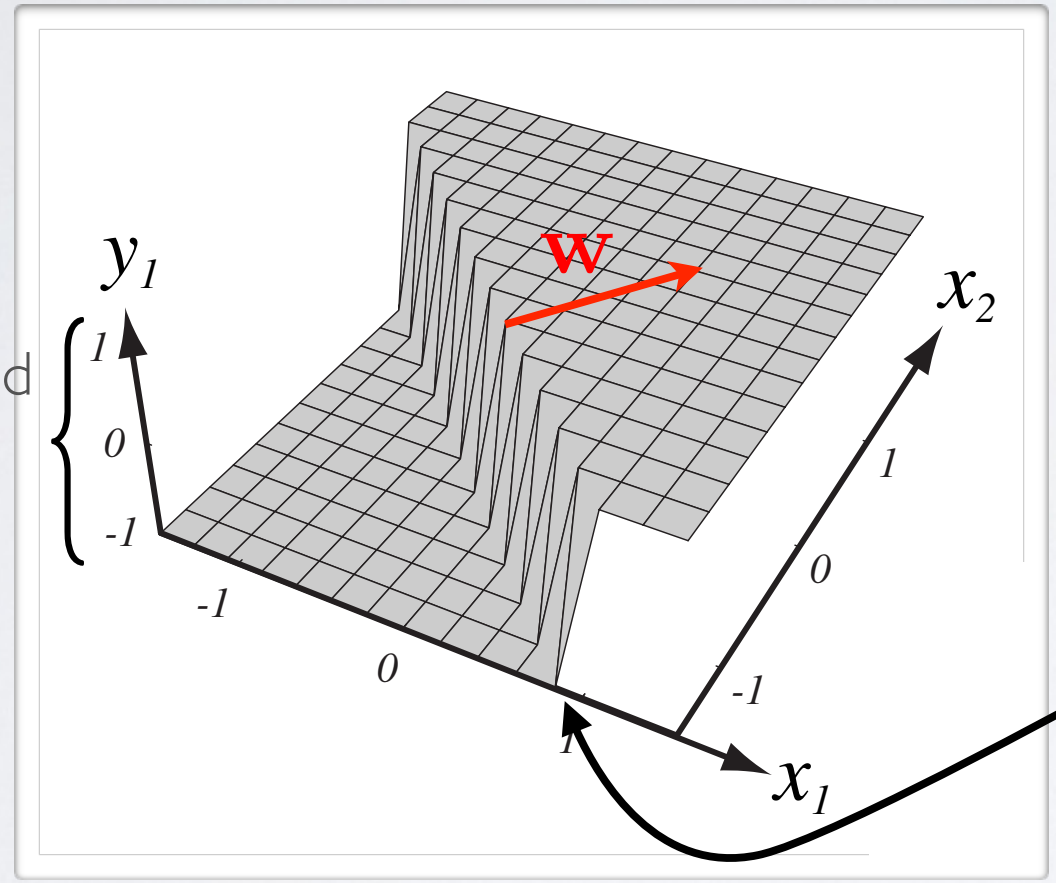
$h(\mathbf{x}) = g(a(\mathbf{x})) = g(\sum_i w_i x_i + b)$

- $\mathbf{w}$ are the connection weights
- b the neuron bias
- $g(\cdot)$ the activation function, $sign(.)$, $\tanh(.)$, sigmoid, ...



Artificial Neuron - activation

range determined by $g(\cdot)$

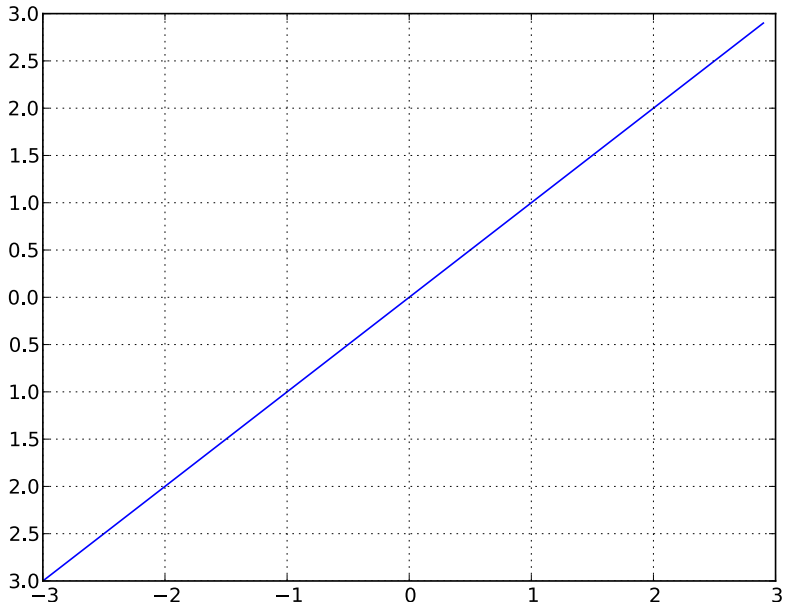


bias b only changes the position of the riff

Artificial Neuron - activation

Linear activation function $g(a) = a$

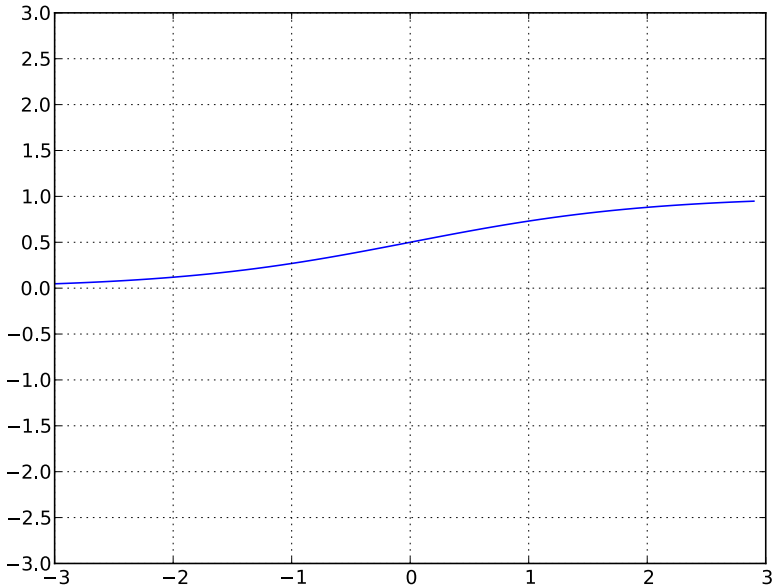
- No restriction on the values
- Not very interesting



Artificial Neuron - activation

sigmod activation function - $g(a) = sigmoid(a) = \frac{1}{1+exp(-a)}$

- Restrict neuron's activation in the interval $[0, 1]$
- Can be positive or negative
- Bounded
- Strictly increasing

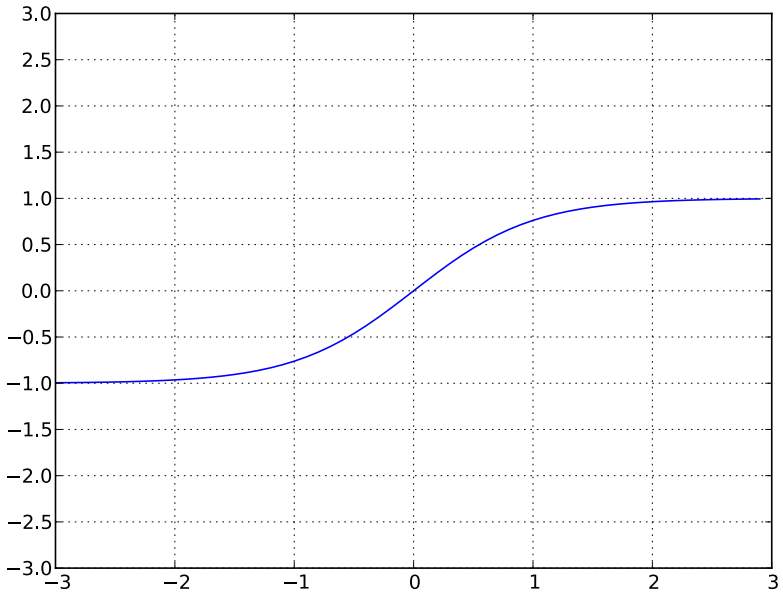


Artificial Neuron - activation

hyperbolic tangent activation function - $g(a) = tanh(a)$

- Restrict neuron's activation in the interval $[-1, 1]$
- Always positive, Bounded and Strictly increasing

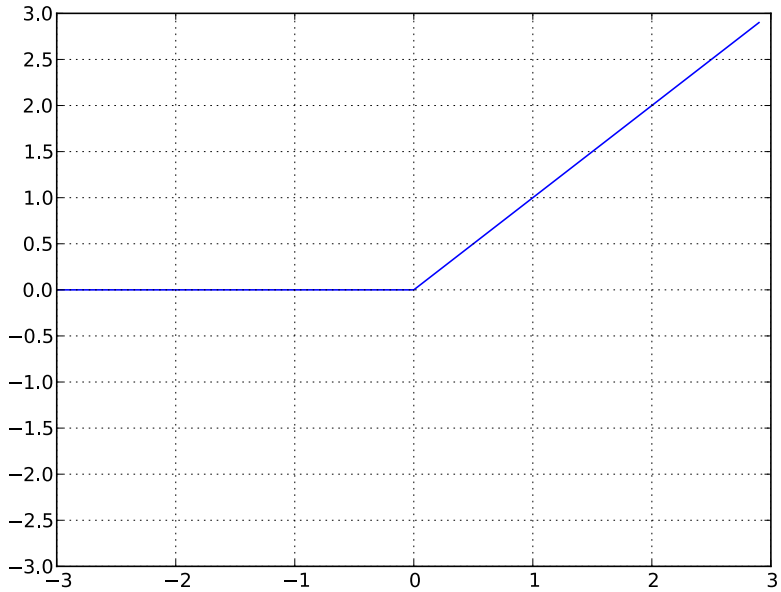
$$tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = \frac{\exp(2a)-1}{\exp(2a)+1}$$



Artificial Neuron - activation

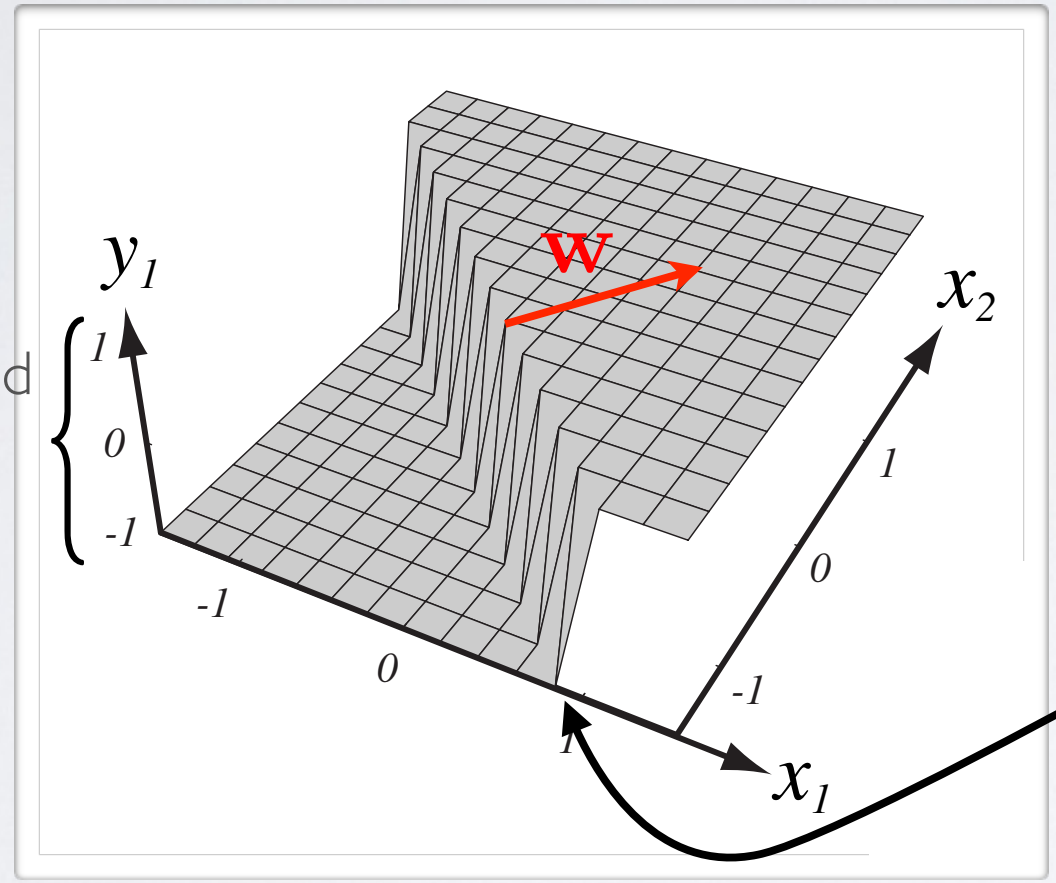
Rectified linear activation function - $g(a) = reclin(a) = max(1, 0)$

- Restrict neuron's activation in the interval $[-1, 1]$
- Always non-negative (bounded below by 0)
- Not upperbounded
- Tends to give neurons with sparse activities



Neuron Activation

range determined by $g(\cdot)$



bias b only changes the position of the riff

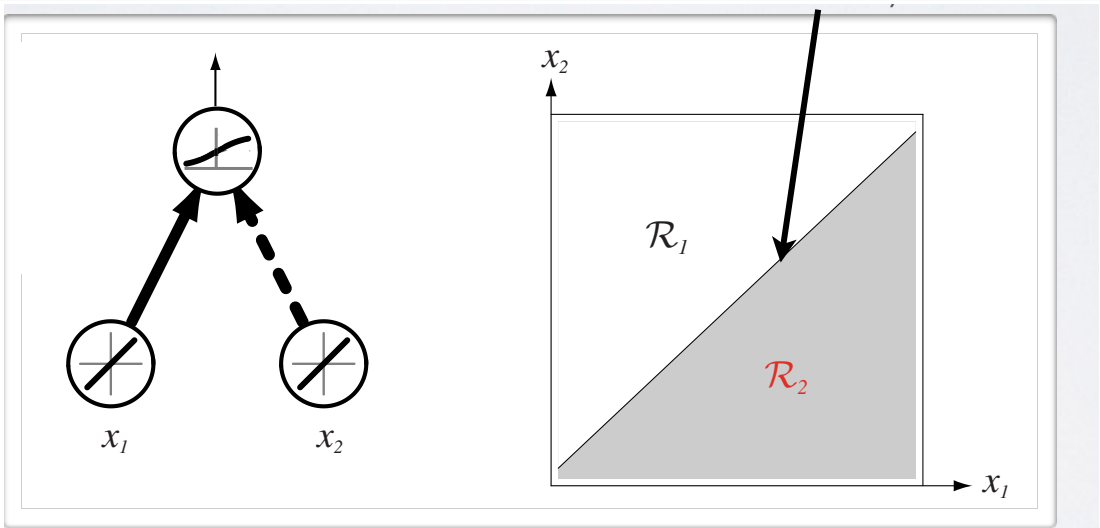
from Hugo Larochelle and Pascal Vincent

Capacity, decision frontier of one neuron

Can do binary classification

The sigmoid activation function can interpret neuron as estimating $p(y = 1|x)$

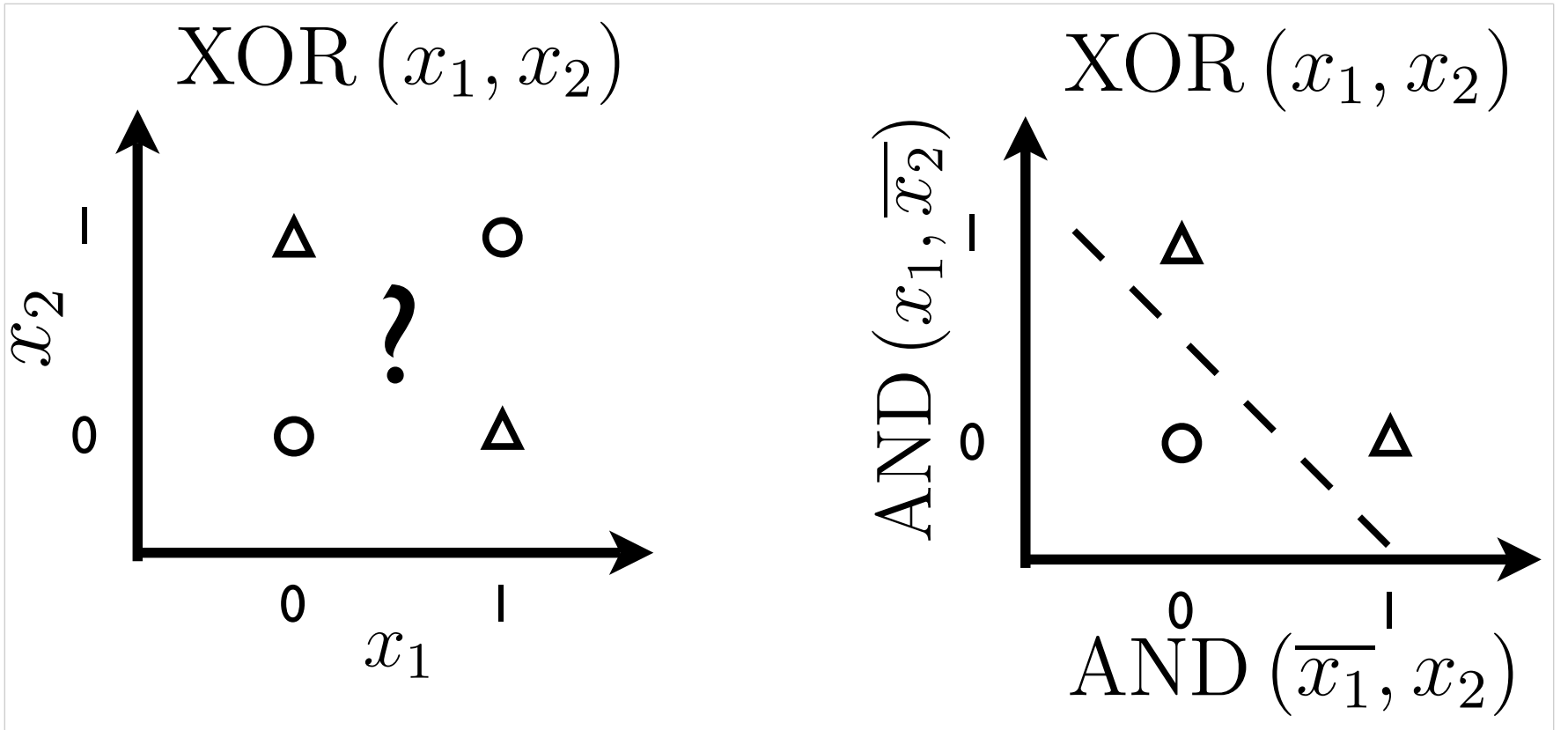
- Logistic regression classifier
- $\rightarrow$ if greater than 0.5 predict class 1
- $\rightarrow$ Otherwise, predict class 0.



from Hugo Larochelle and Pascal Vincent

Capacity of a single neuron

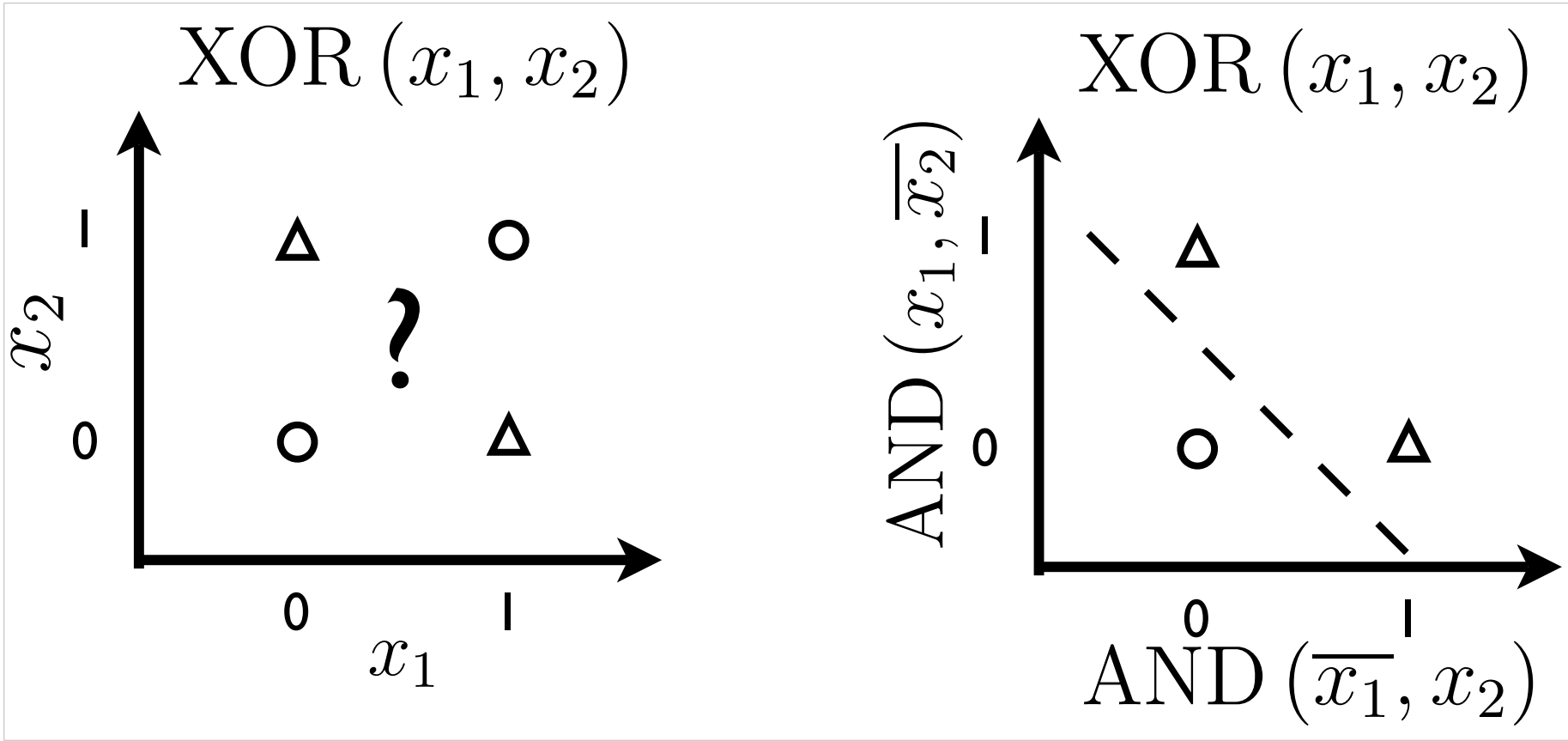
Can solve linearly separable problems



from Hugo Larochelle and Pascal Vincent

Capacity of a single neuron

But can't solve non linearly separable problems



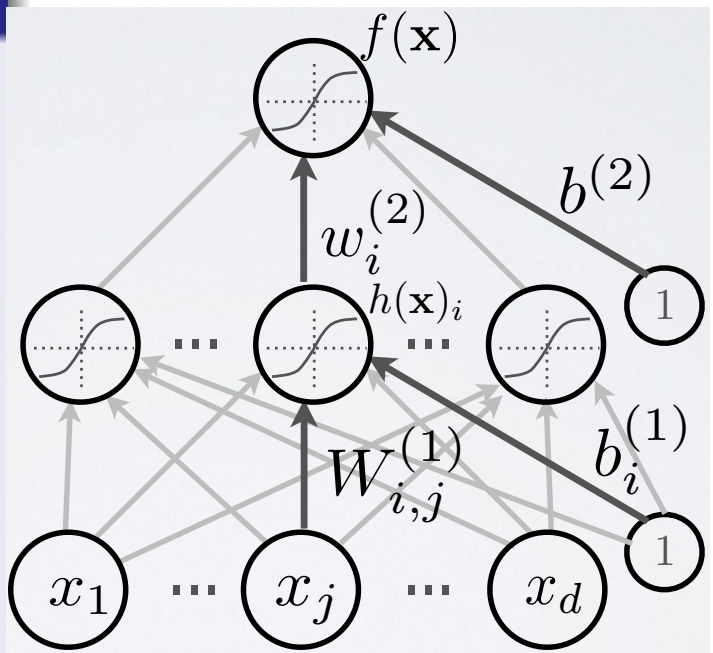
from Hugo Larochelle and Pascal Vincent

unless the input is transformed in a better representation.

More expressiveness: add a "hidden" layer

single hidden layer neural network

- Hidden layer pre-activation:
  $\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$
  $\left(\mathbf{a}(\mathbf{x})_i = \mathbf{b}_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j\right)$
- Hidden layer activation:
  $\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$
- Output layer activation:
  $f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)T}\mathbf{h}^{(1)}\mathbf{x}\right)$
  o is the output activation function



Softmax activation function

For multi-class classification

- Need of multiple outputs (1 per class)
- We would like to have $p(y = c|\mathbf{x})$

Softmax activation function at the output - for c classes

$$\mathbf{o}(\mathbf{a}) = softmax(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)}, \cdots, \frac{\exp(a_C)}{\sum_c \exp(a_c)}\right]$$

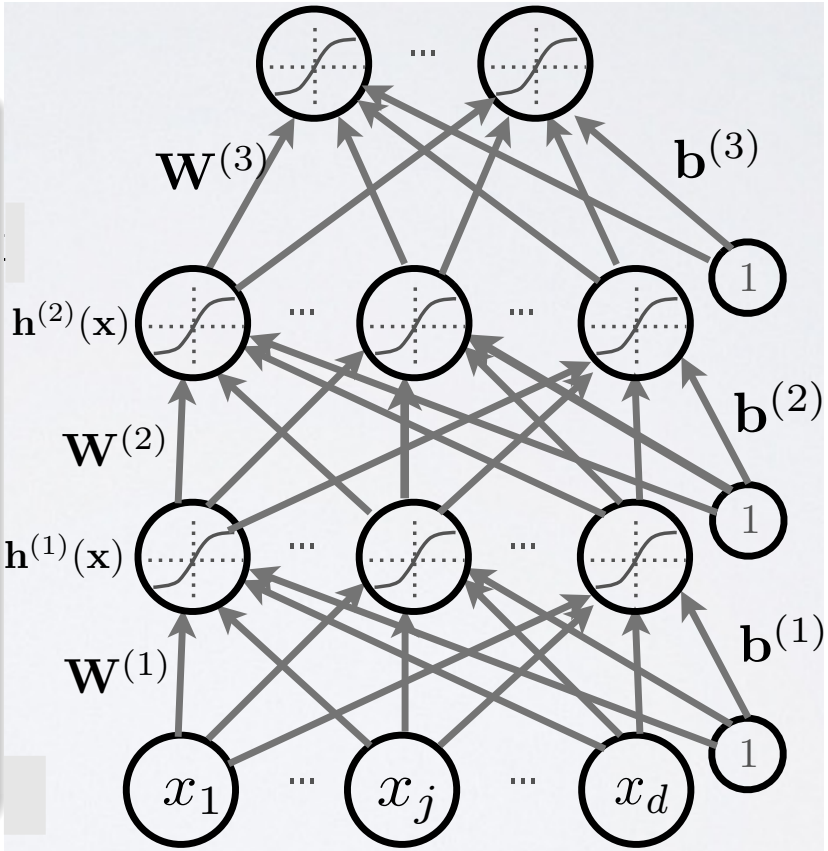
strictly positive and sums up to one

Predicted class is the one with highest estimated probability.

More expressiveness: multilayer neural network

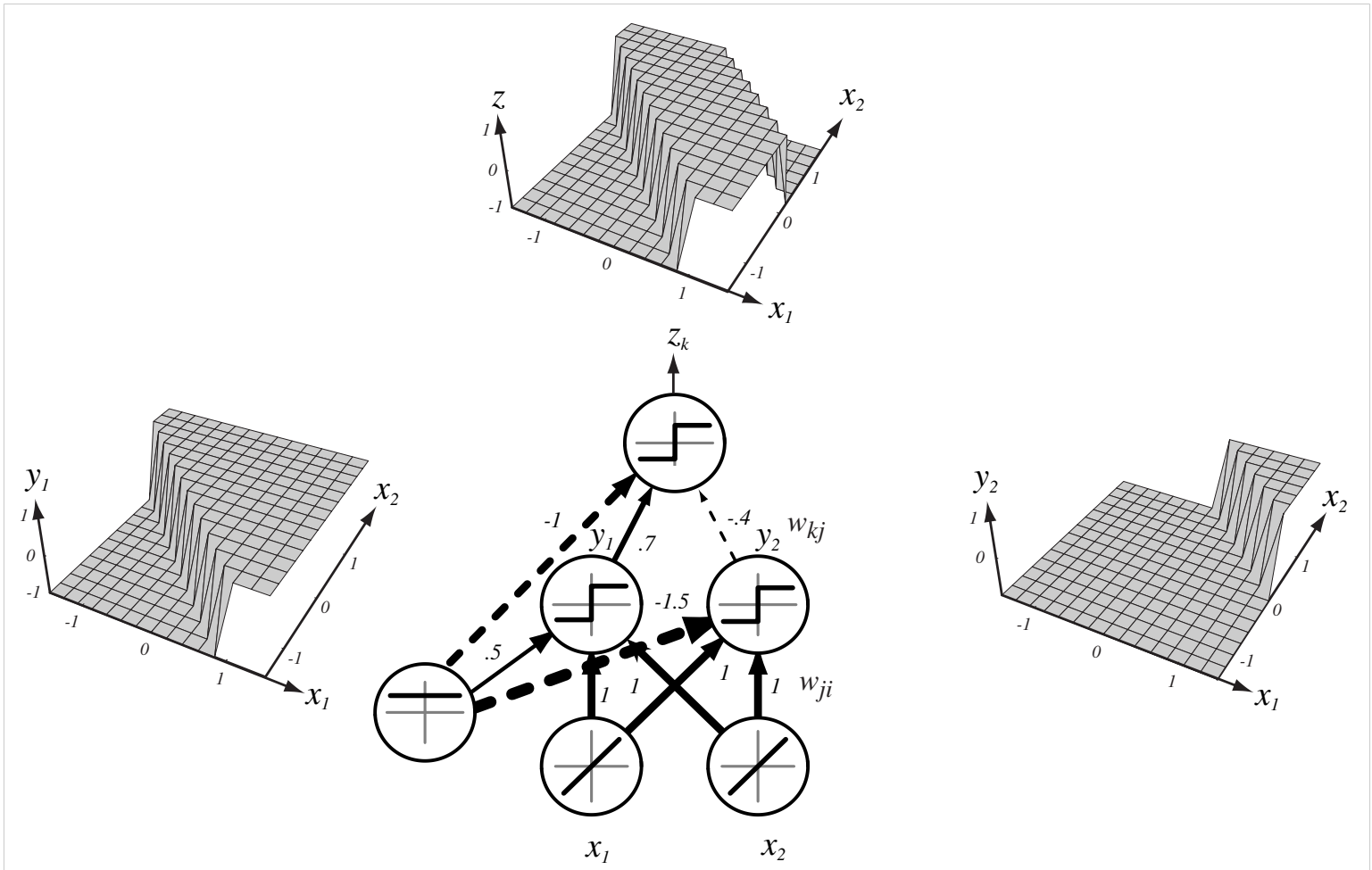
L hidden layers

- layer pre-activation for $k > 0$ ($\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$)
  $\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$
- hidden layer activation (k from 1 to L):
  $\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$
- output layer activation (k=L+1)
  $\mathbf{h}^{L+1}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{L+1}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$



Capacity of a NN

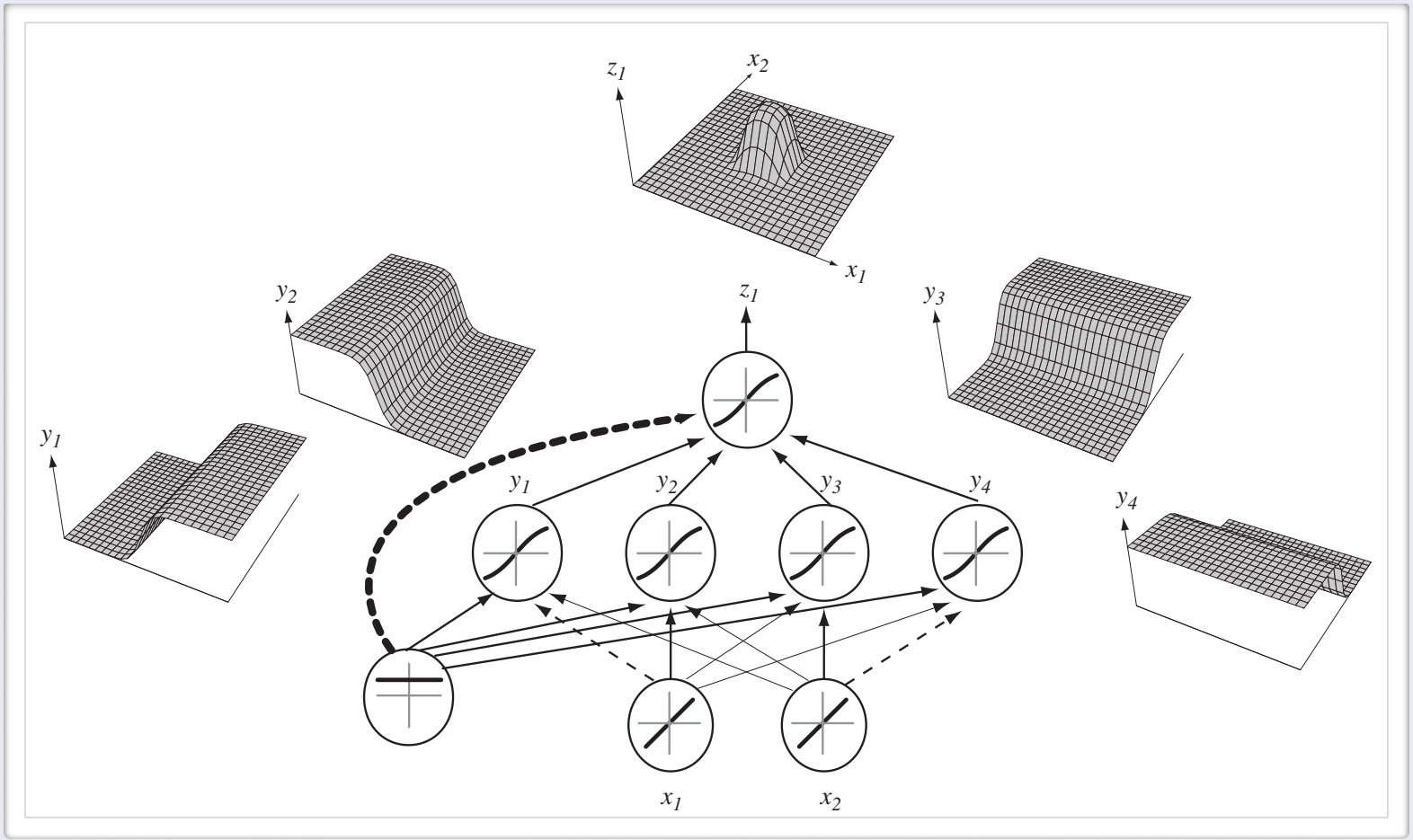
Single hidden layer Neural Network



from Hugo Larochelle and Pascal Vincent

Capacity of a NN

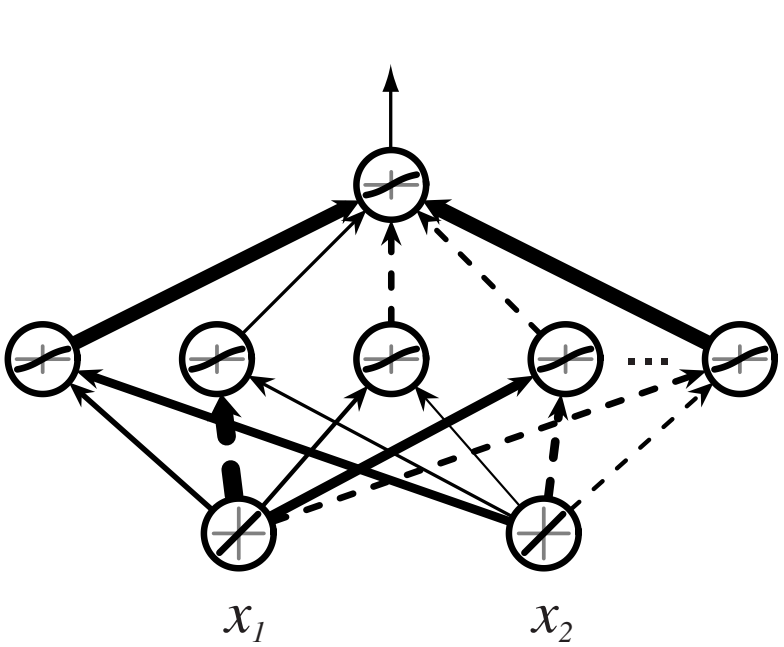
Single hidden layer Neural Network

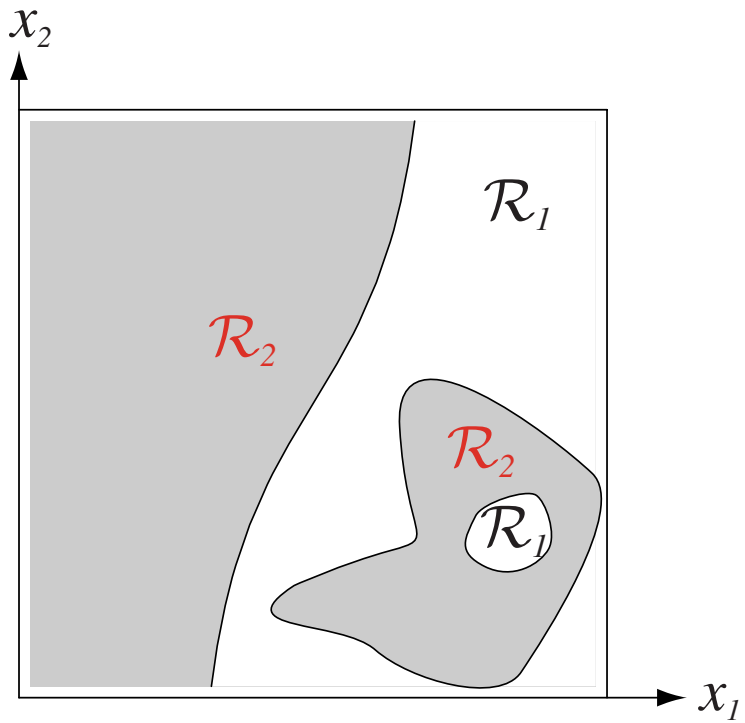


from Hugo Larochelle and Pascal Vincent

Capacity of a NN

Single hidden layer Neural Network





from Hugo Larochelle and Pascal Vincent

Capacity of Neural Network

Universal approximation

- Universal approximation theorem (Hornik, 1991):
  "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"
- The result applies for sigmoid, tanh and many other hidden layer activation functions
- This is a good result, but it does not mean that there is a learning algorithm that can find the necessary parameter values!

Learning Neural Network parameters

Empirical risk minimization

$\text{argmin}_\theta \frac{1}{T} \sum_i \ell(f(\mathbf{x}_i; \theta), y_i) + \lambda\Omega(\theta)$

- $\ell(f(\mathbf{x}_i; \theta), y_i)$ loss function w.r.t. to the model parameters θ
- $\Omega(\theta)$ a regularizer that penalizes certain values of θ

Learning is cast as optimization, the loss function is ideally smooth (think of a surrogate of the 0-1 classification error)

The approach used: Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent

Algorithm that performs updates after each example

- initialize θ ($\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
- For N iterations # make a training over **all** examples $= 1$ **epoch**
  - For each training example $(\mathbf{x}_i, y_i)$
    - $\Delta = -\nabla_{\theta} \ell(f(\mathbf{x}_i; \theta), y_i) - \lambda \nabla_{\theta} \Omega(\theta)$
    - $\theta = \theta + \alpha \Delta$

To apply this algorithm to neural network training, we need

- the loss function $\ell(f(\mathbf{x}_i; \theta), y_i)$
- the parameter gradients $\nabla_{\theta} \ell(f(\mathbf{x}_i; \theta), y_i)$
- the regularizer $\Omega(\theta)$ and its gradient $\nabla_{\theta} \Omega(\theta)$

Loss function

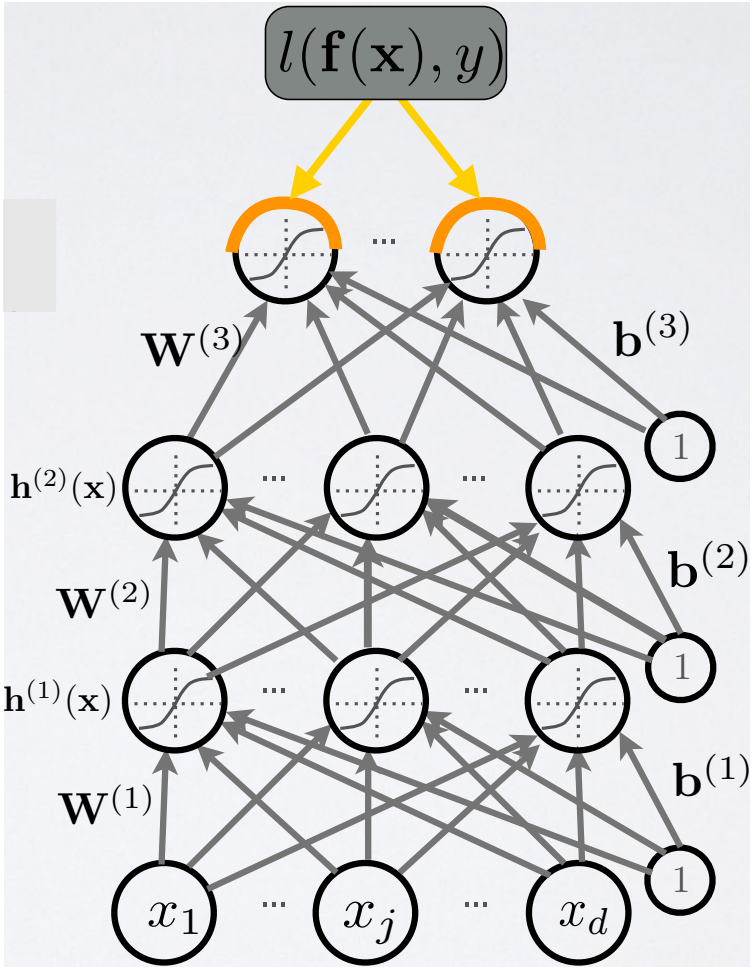
A particular Loss function for classification

- Neural network estimates $f(\mathbf{x})_c = p(y = c|\mathbf{x})$
  $\rightarrow$ we could maximize the probabilities y_i given $\mathbf{x}_i$ from training set
- Thie can be done by minimizing the log-likelihood
  $\ell(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{y=c} \log(f(\mathbf{x})_c) = -\log f(\mathbf{x})_y$
- This sometimes referred to as cross-entropy
- log allows a better numerical stability and math convenience

Loss gradient at output

- Partial derivative:
  $\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1_{y=c}}{f(\mathbf{x})_y}$
- Gradient:

$$\begin{aligned}
\nabla_{\mathbf{f}(\mathbf{x})} - \log f(\mathbf{x})_y & \\
&= \frac{-1}{f(\mathbf{x})_y} \begin{bmatrix} 1_{y=0} \\ \vdots \\ 1_{y=C-1} \end{bmatrix} \\
&= \frac{-\mathbf{e}(y)}{f(\mathbf{x})_y}
\end{aligned}$$



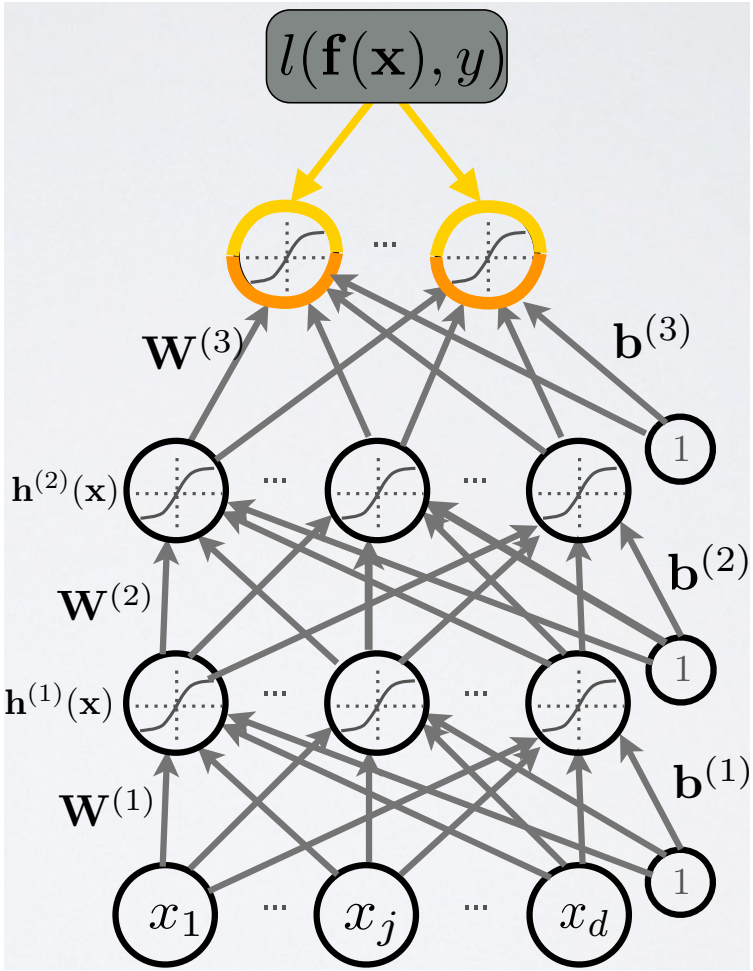
loss gradient at output pre-activation

- Partial derivative (see next slide)

$$\frac{\partial}{\partial \mathbf{a}^{L+1}(\mathbf{x})_c} - \log f(\mathbf{x})_y = -(1_{y=c} - f(\mathbf{x})_c)$$

- Gradient:

$$\nabla_{\mathbf{a}^{L+1}(\mathbf{x})_c}(-\log f(\mathbf{x})_y) = -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$



Derivative $\frac{\partial}{\partial \mathbf{a}^{L+1}(\mathbf{x})_c} - \log f(\mathbf{x})_y$

$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y$$

$$= \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} softmax(\mathbf{a}^{(L+1)}(\mathbf{x}))_y$$

$$= \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})}$$

$$= \frac{-1}{f(\mathbf{x})_y} \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \exp(a^{(L+1)}(\mathbf{x})_y) \times (\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})) \right.$$

$$\left. - \exp(a^{(L+1)}(\mathbf{x})_y) \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} (\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})) \right) /$$

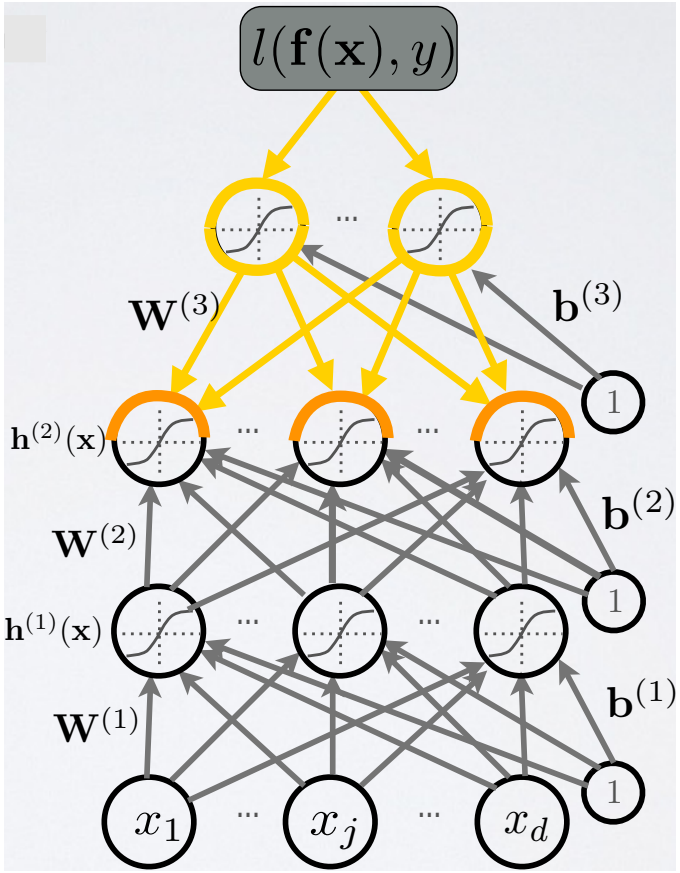
$$(\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}))^2$$

Derivative $\frac{\partial}{\partial \mathbf{a}^{L+1}(\mathbf{x})_c} - \log f(\mathbf{x})_y$ (continued)

$$
\begin{aligned}
&= \frac{-1}{f(\mathbf{x})_y}\left(\frac{1_{(y=c)}\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'}\exp(a^{(L+1)}(\mathbf{x})_{c'})} - \right. \\
&\qquad \left. \frac{\exp(a^{(L+1)}(\mathbf{x})_y)\exp(a^{(L+1)}(\mathbf{x})_c)}{(\sum_{c'}\exp(a^{(L+1)}(\mathbf{x})_{c'}))(\sum_{c'}\exp(a^{(L+1)}(\mathbf{x})_{c'}))}\right) \\
&= \frac{-1}{f(\mathbf{x})_y}\left(1_{(y=c)}softmax(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \right. \\
&\qquad \left. softmax(\mathbf{a}^{(L+1)}(\mathbf{x}))_y softmax(\mathbf{a}^{(L+1)}(\mathbf{x}))_c\right) \\
&= \frac{-1}{f(\mathbf{x})_y}(1_{(y=c)}f(\mathbf{x})_y - f(\mathbf{x})_y f(\mathbf{x})_c) \\
&= -(1_{(y=c)} - f(\mathbf{x})_c)
\end{aligned}
$$

Loss gradient at hidden layer

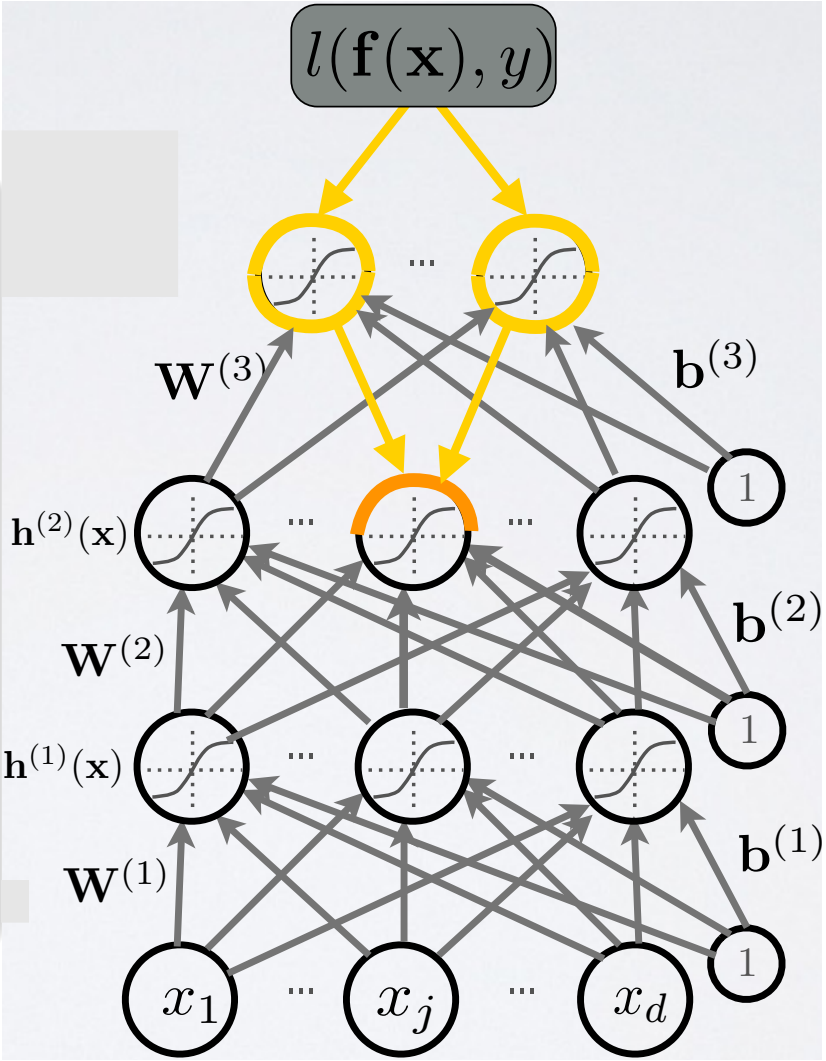
- ... more complicated



Loss gradient at hidden layer

The chain rule

- If a function $p(a)$ can be written as a function of intermediate results say $q_i(a)$, then we have:
  $\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$
- We can invoke it by setting
  - a to a unit layer
  - $q_i(a)$ a pre-activation in the layer above
  - $p(a)$ is the loss function



Loss gradient at hidden layer

- Partial derivative

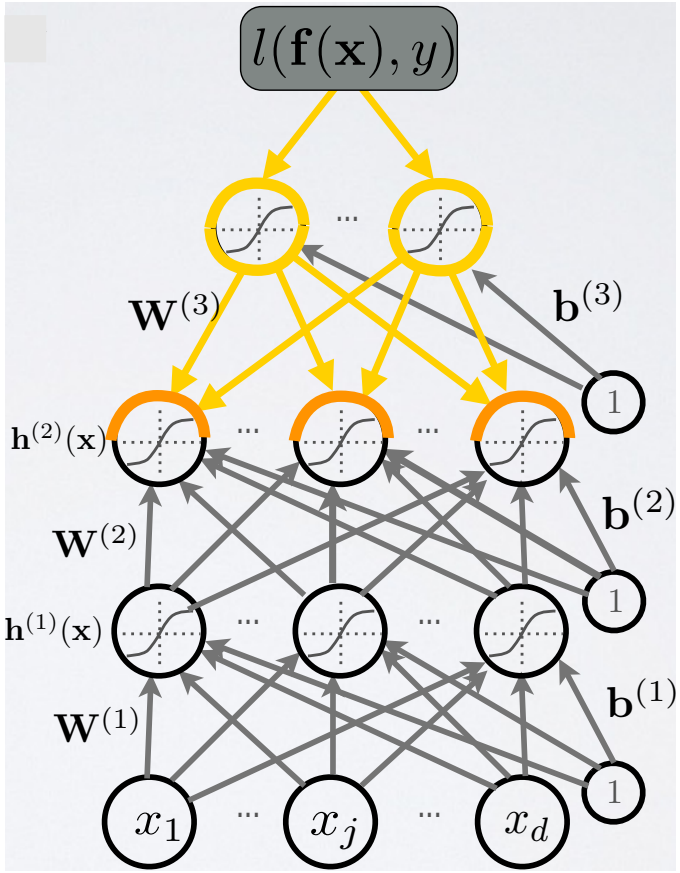
$$\frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j}$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W_{i,j}^{(k+1)}$$

$$= (\mathbf{W}_{\cdot,j}^{(k+1)})^\top (\nabla_{\mathbf{a}^{k+1}(\mathbf{x})}(-\log f(\mathbf{x})_y))$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

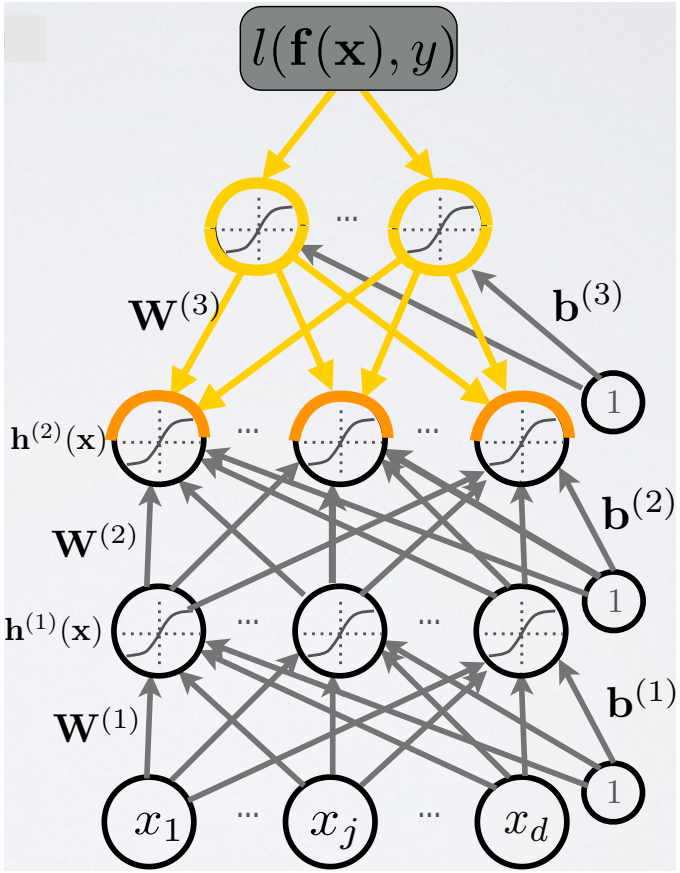


Loss gradient at hidden layer

- Gradient

$$\nabla_{h^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$
$$= \mathbf{W}^{(k+1)^\top} (\nabla_{\mathbf{a}^{k+1}(\mathbf{x})} (-\log f(\mathbf{x})_y))$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

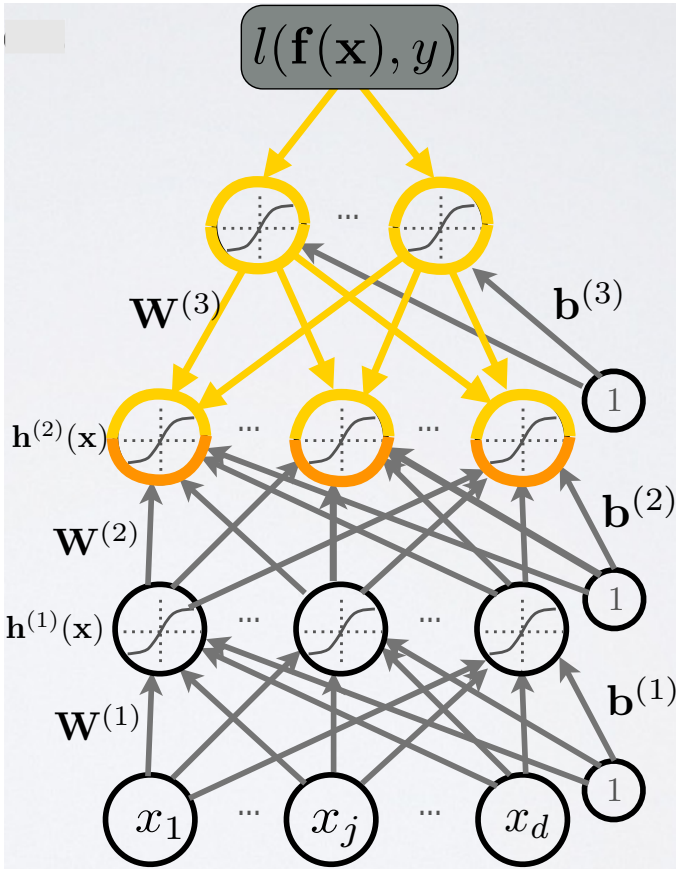


Loss gradient at hidden layer pre-activation

- Partial derivative

$$\frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j}$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j)$$

with: $h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$

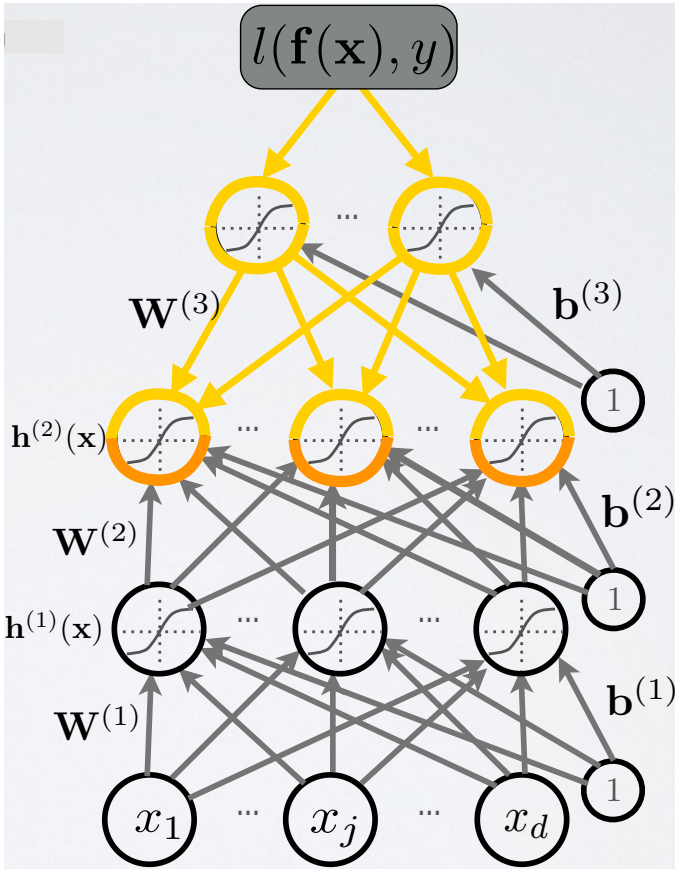


Loss gradient at hidden layer pre-activation

- Gradient

$$
\begin{aligned}
&\nabla_{a^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\
=&(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\
=&(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y) \circ \\
&\qquad [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots]
\end{aligned}
$$

with: $\circ$ the element-wise product and $h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$



Gradient of main activation functions

- Linear activation: $g(a) = a$
  $\rightarrow g'(a) = 1$
- sigmoid activation: $g(a) = sigm(a) = \frac{1}{1+exp(-a)}$
  $\rightarrow g'(a) = g(a)(1 - g(a))$
- tanh activation: $g(a) = tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = \frac{\exp(2a)-1}{\exp(2a)+1}$
  $\rightarrow g'(a) = 1 - g(a)^2$
- Reclin activation: $g(a) = reclin(a) = max(a, 0)$
  $\rightarrow g'(a) = 1_{a>0}$

Loss gradient of the parameters

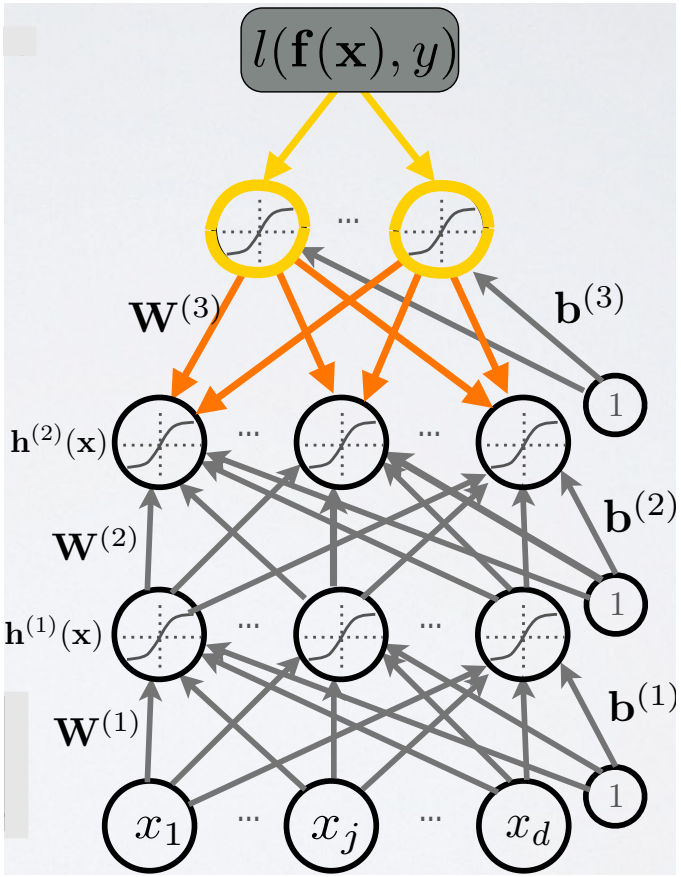
- Partial derivatives (weights)

$$\frac{\partial}{\partial W_{i,j}^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial W_{i,j}^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} h^{(k-1)}(\mathbf{x})_j$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

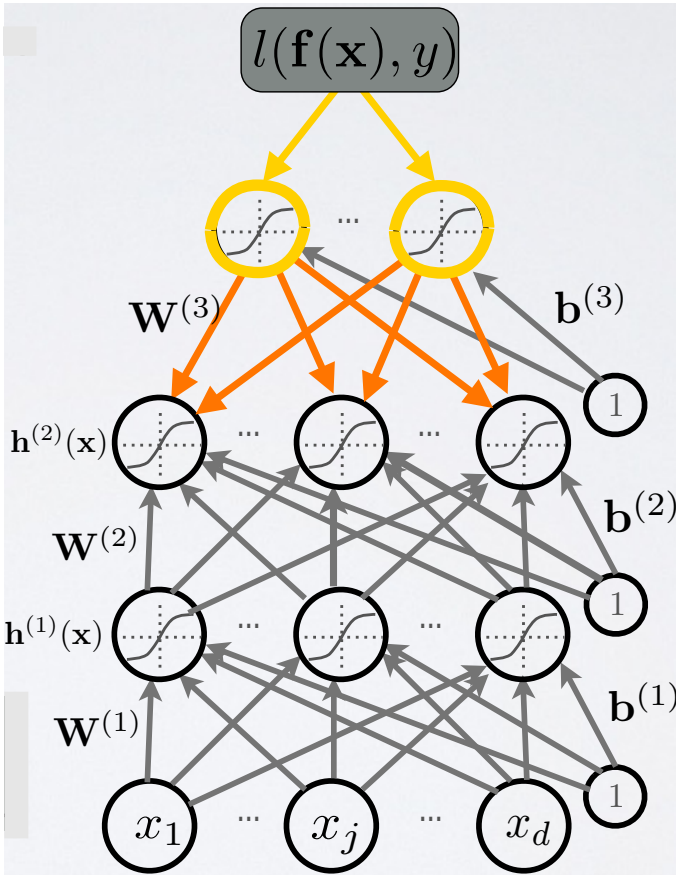


Loss gradient of the parameters

- Gradient (weights)

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y$$
$$=(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)\mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

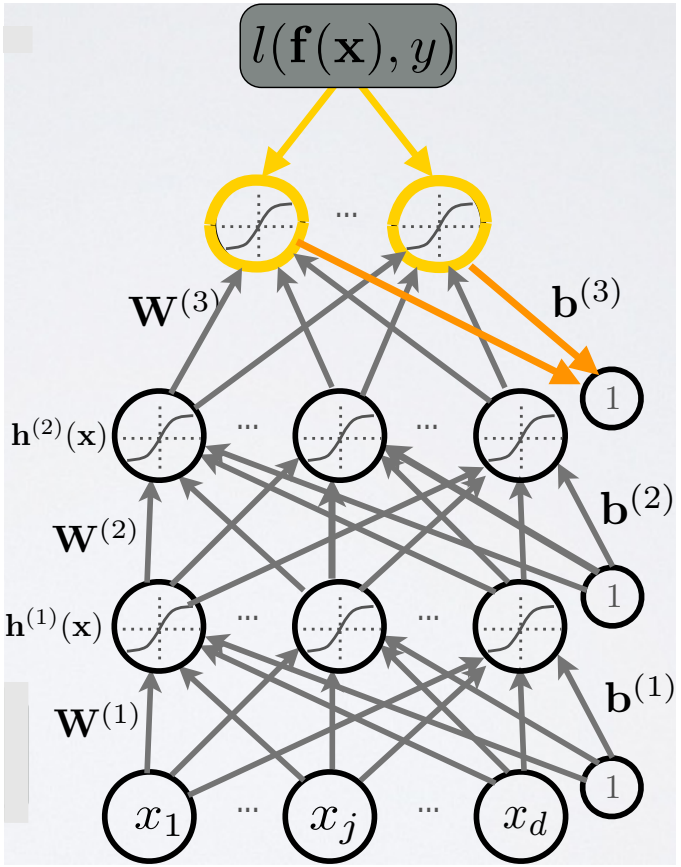


Loss gradient of the parameters

- Partial derivatives (biases)

$$\frac{\partial}{\partial b_i^{(k)}} - \log f(\mathbf{x})_y$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial b_i^{(k)}}$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i}$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

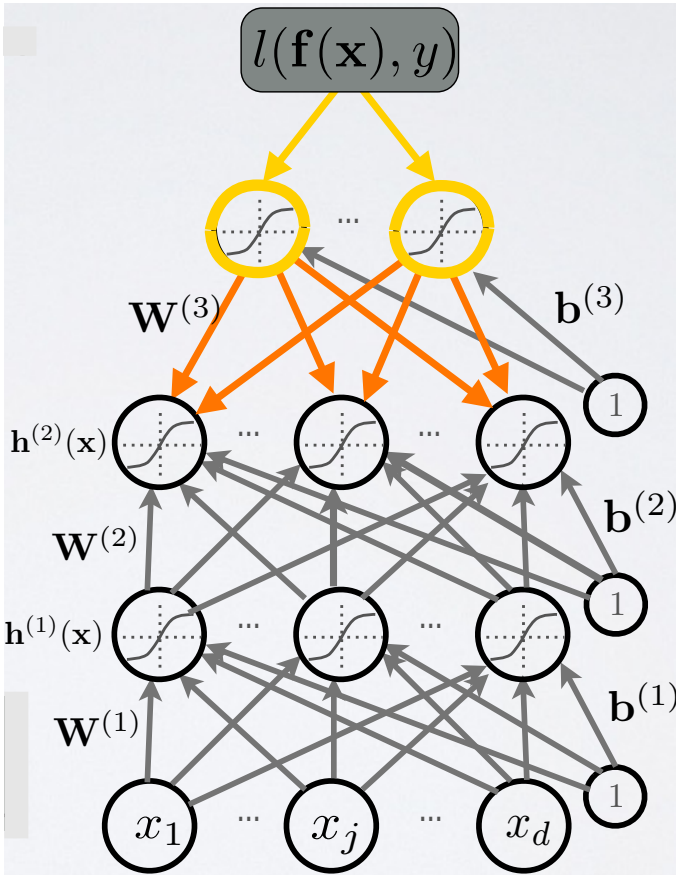


Loss gradient of the parameters

- Gradient (weights)

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y$$
$$= \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

with:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



Backpropagation algorithm

We assume a forward propagation has been made before (to evaluate the loss)

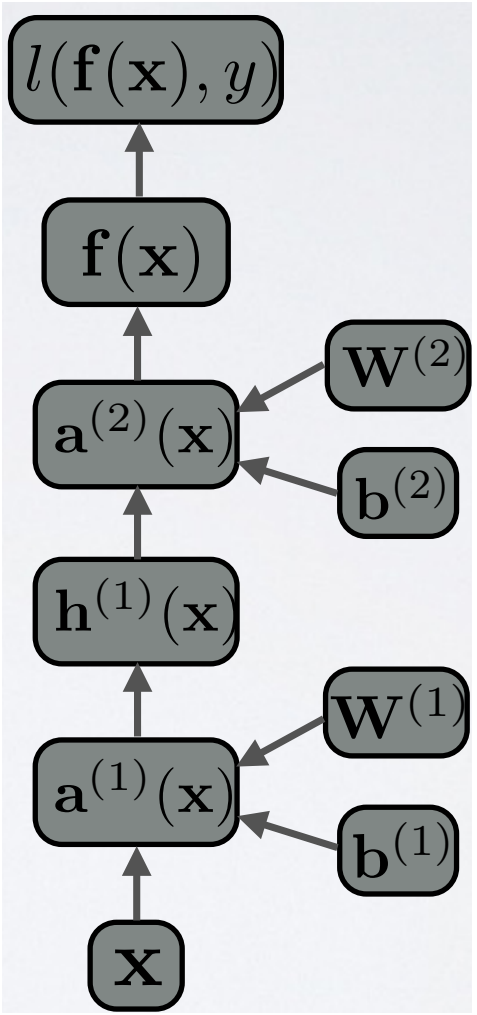
Algorithm

- Compute output gradient (before activation)
- For k from $L+1$ to 1
  - compute gradients of hidden layer parameter
    $\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \leftarrow (\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)\mathbf{h}^{(k-1)}(\mathbf{x})^\top$
    $\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \leftarrow \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$
  - compute gradient of hidden layer below
    $\nabla_{h^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \leftarrow \mathbf{W}^{(k+1)^\top}(\nabla_{a^{k+1}(\mathbf{x})}(-\log f(\mathbf{x})_y))$
  - compute gradient of hidden layer below (before activation)
    $\nabla_{a^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \leftarrow (\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y) \circ [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots]$

Flow graph

Flow graph

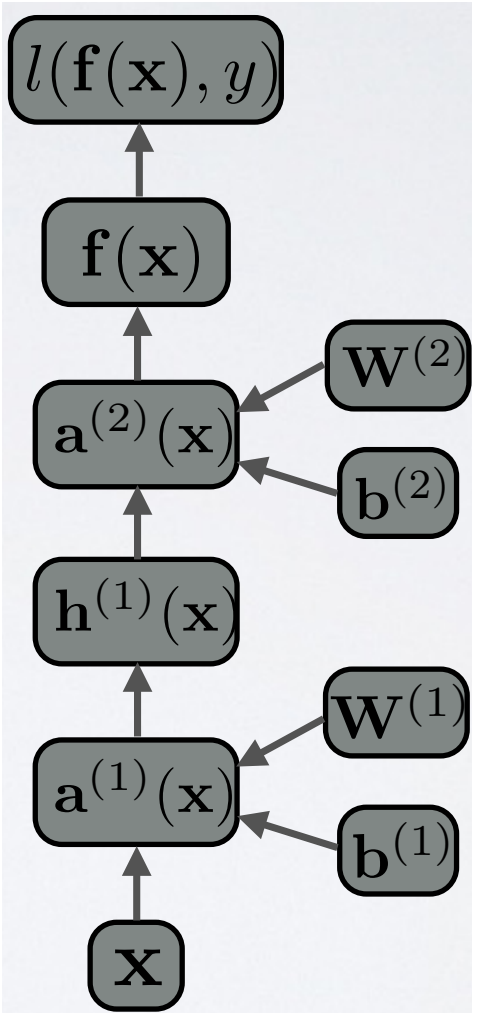
- Forward propagation can be represented as an acyclic flow graph
- It's a nice way of implementing foward propagation in a modular way
  - each box could be an object with a fprop method that computes the value of the box given its children
  - calling the fprop method of each box in the right order yields forward propagation



Flow graph

Automatic differentiation

- Each object has also a bprop method
  - It computes the gradient of the loss with respect to each children
  - fprop depends on the fprop of a box's children while bprop depends on the bprop of a box's parents
- By calling bprop in the reverse order, we get backpropagation.
  (only need to reach the parameters)



About regularization

L2 regularization

$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k \|\mathbf{W}^{(k)}\|_F^2$

Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = 2\mathbf{W}^{(k)}$

Only applied on weights, not on biases (weight decay)

Can be interpreted as having Gaussian prior over weights

L1 regularization

$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left|W_{i,j}^{(k)}\right|$

Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = sign(\mathbf{W}^{(k)})$, with

$sign(\mathbf{W}^{(k)})_{i,j} = 1_{W_{i,j}^{(k)} > 0} - 1_{W_{i,j}^{(k)} < 0}$

Also only applied on weights

Can be interpreted as having a Laplacian prior over weights, certain weights will be exactly zero.

About initialization

- For biases: intialize to 0 (or 1)
- For weights
  - Can't initialize weights to 0 with tanh activation (we can show that all gradients would tend to 0)
  - Can't initialize all weights to the same value - we can show that all hidden units in a layer will always behave the same, need to break symmetry.
- Recipe: sample $\mathbf{W}_{i,j}^{(k)}$ from $Uniform[-b, b]$ where $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$

  where H_k si the size of $\mathbf{h}^{(k)}(\mathbf{x})$.
  Sample aroud zero but break symmetry, other values of b could work well (not an exact science) [Glorot& Bengio, 2010]

Model validation

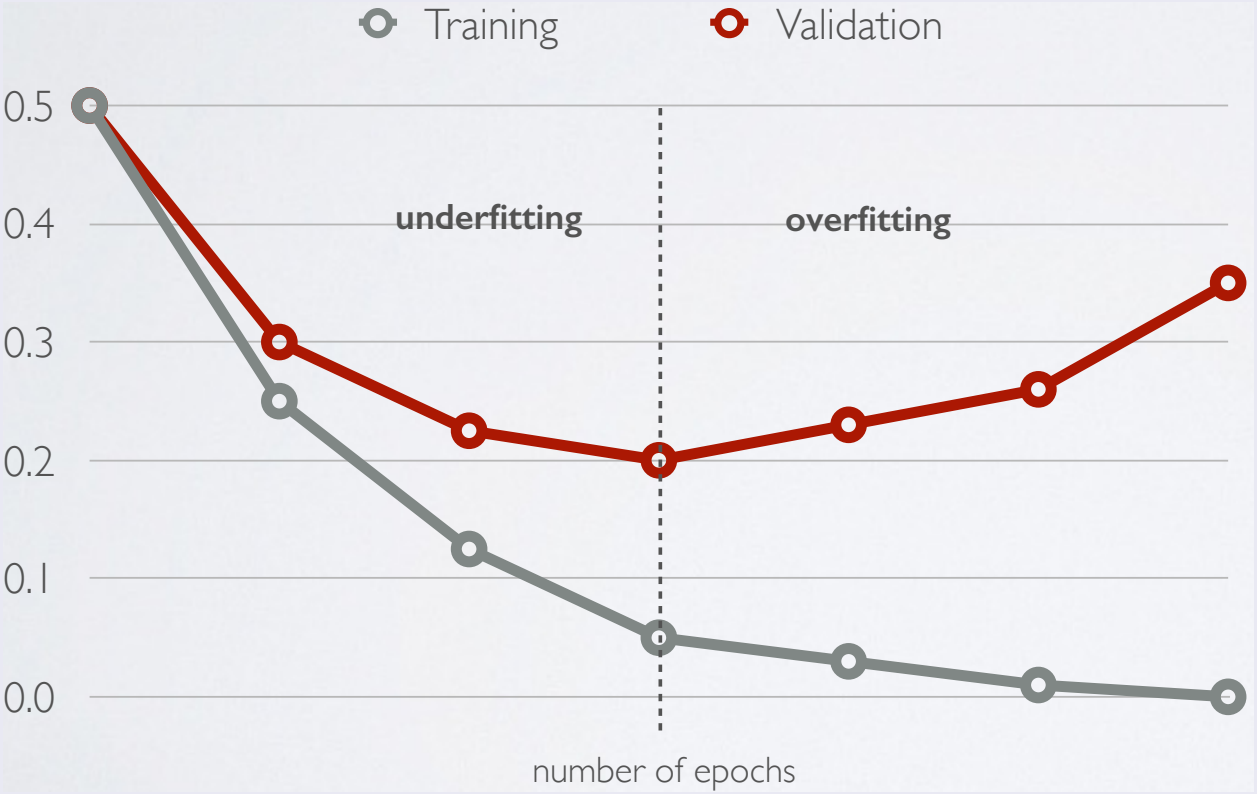
Model selection

- A **training** set to learn the model; A **validation** set to select hyper-parameters (hidden layer size(s), learning rate, number of iterations, epochs, ...); and of course a **test** set.
- Parameter selection
  - Grid search: set of values to test for each hyper-parameter, try all possible configurations
  - Random search: draw parameters according to a distribution over the values of each hyper-parameters (uniform)
  - Other more complex strategies
  - You can go back and refine the grid/distributions if needed.

Early Stopping

Early stopping

- Early stopping - to select the number of epochs: stop training when validation set error increases.



Other tricks

- Normalize data (subtract each feature by mean and divide by standard deviation (and dimension))
- Decaying the learning rate: as getting closer to optimum
  Start with large value (*e.g.* 0.1), maintain until validation error stops decreasing, divide learning by 2 and go back to previous step
- Mini-batch optimization: updates based on a mini-batch instead of only 1 example
- Adaptive learning rates
- Gradient checking: To debug you can compare with a finite difference of gradients
  $\frac{\partial f(x)}{\partial x} \underset{\sim}{} \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$
  f the loss, x a parameter, ϵ the approximation
- To debug: your model should overfit on a small dataset ($\sim$50 examples)
  If not: scale down initialization for units saturated, normalize inputs, decrease learning rate if training error bounces up and down

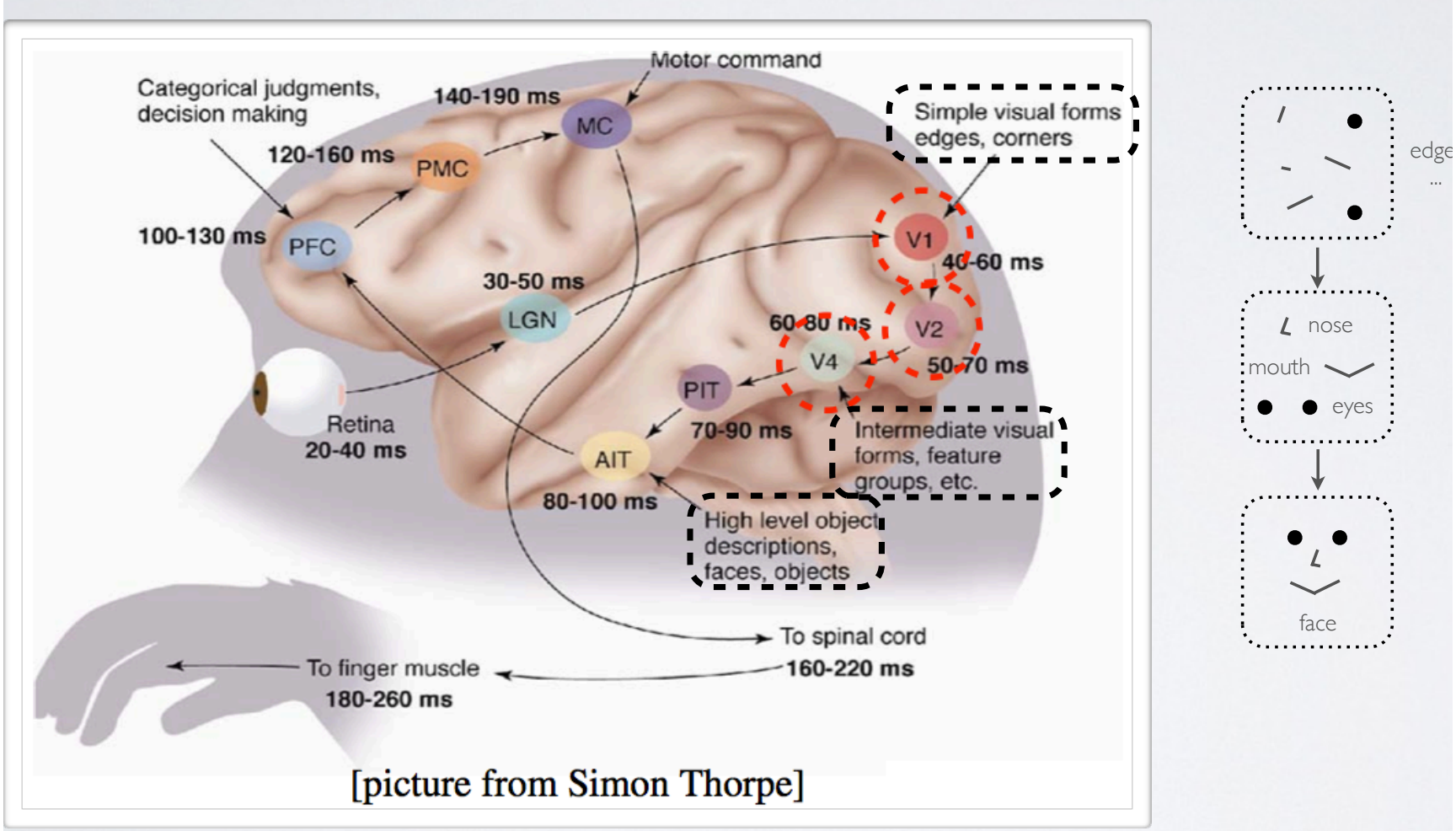
Deep Learning

Deep Learning

What is deep learning

- Deep learning refers to learning (complex) functions with multilayer representations (this includes multi-layer neural networks, but this refers also to other types of hierarchical models)
- Each layer can be interpreted as an (intermediate) representation $\rightarrow$ representation learning

Application in computer vision



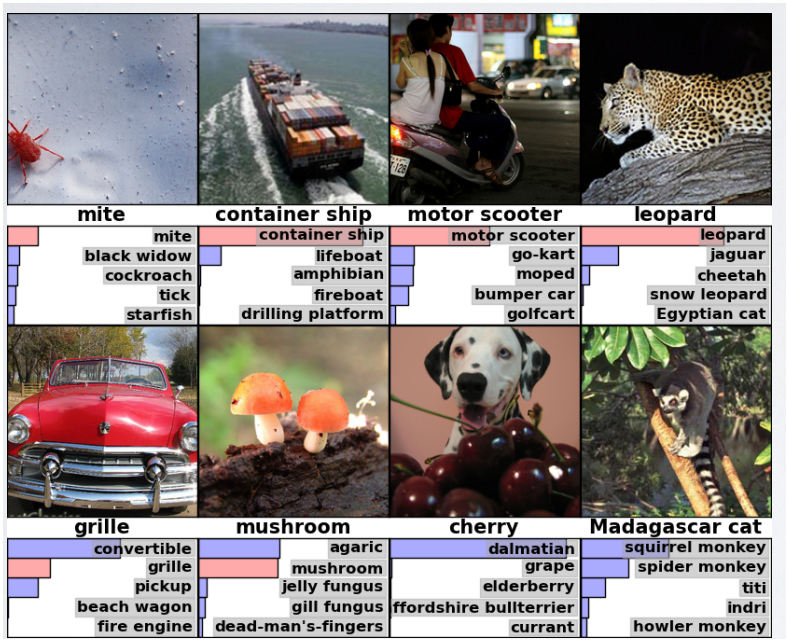
Deep Learning

Some Theoretical features

- A deep architecture can represent certains functions (that may require an exponential representation) more compactly
- Example: Boolean functions
  - A boolean circuit can be seen as a kind of neural network where hidden units are logic gates (AND; OR, NOT, ...)
  - any boolean function can be represented by a single hidden layer boolean circuit
    But it might require an exponential number of units
  - It can be shown that there are Boolean functions that require an expontial number of children with a single layer model but need only a polynomial number of hidden units if we have more layers
    (See 'Exploring Strategies for Training Deep Neural Networks' for more)

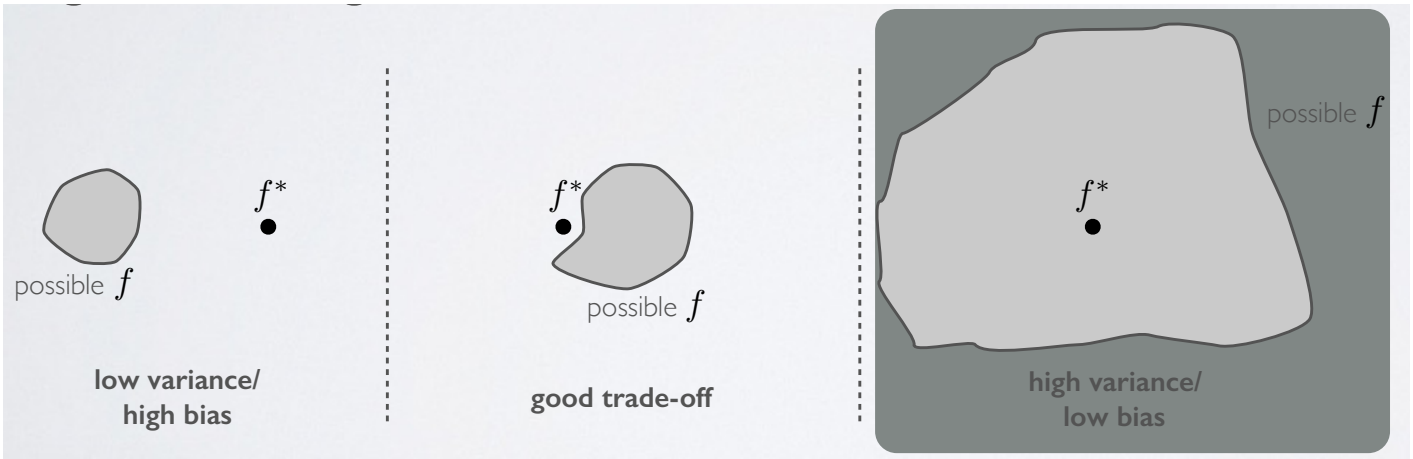
Deep learning - success story

- Computer vision
- Speech recognition
- Natural language processing (even though models are not so deep)
- Automatic driving, Go, google translate, Poker?, ...



Deep learning - Training is hard!

- First hypothesis: optimization is hard (under-fitting)
  vanishing gradient problem, saturated units block gradient propagation
  Well known problem in recurrent neural networks
- Second hypothesis: overfitting
  we explore a space of complex functions, deep nets have usually a lot of parameters
  Might be in a high variance/low bias problem.

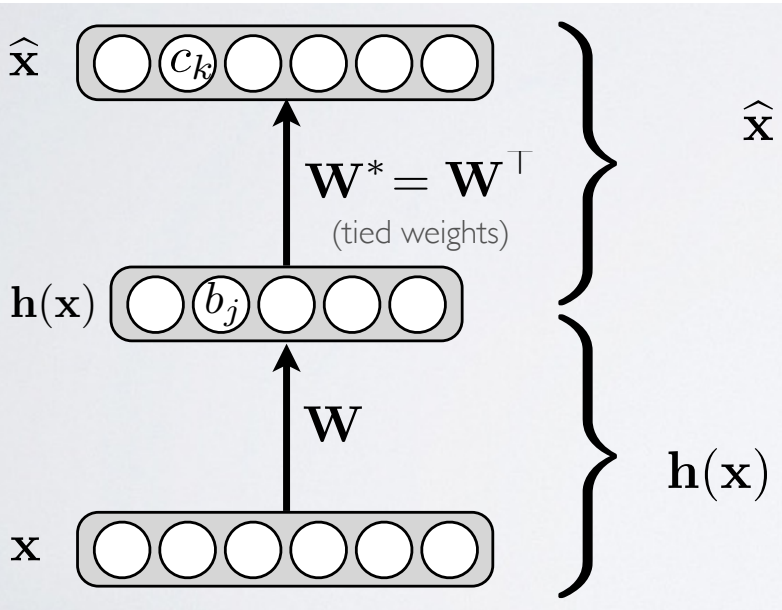


Deep learning - Training is hard!

- Depending on the problem, one or the other situation will dominate
- If first hypothesis (underfitting): better optimize
  better optimization method, use GPUs
- If second hypothesis (overfitting)
  unsupervised learning, stochastic dropout training

Autoencoders

- Feed-forward neural network trained to reproduce its input at the output layer
- Intermediate layer must small/ sparse (can be forced by using L_1 regularization)

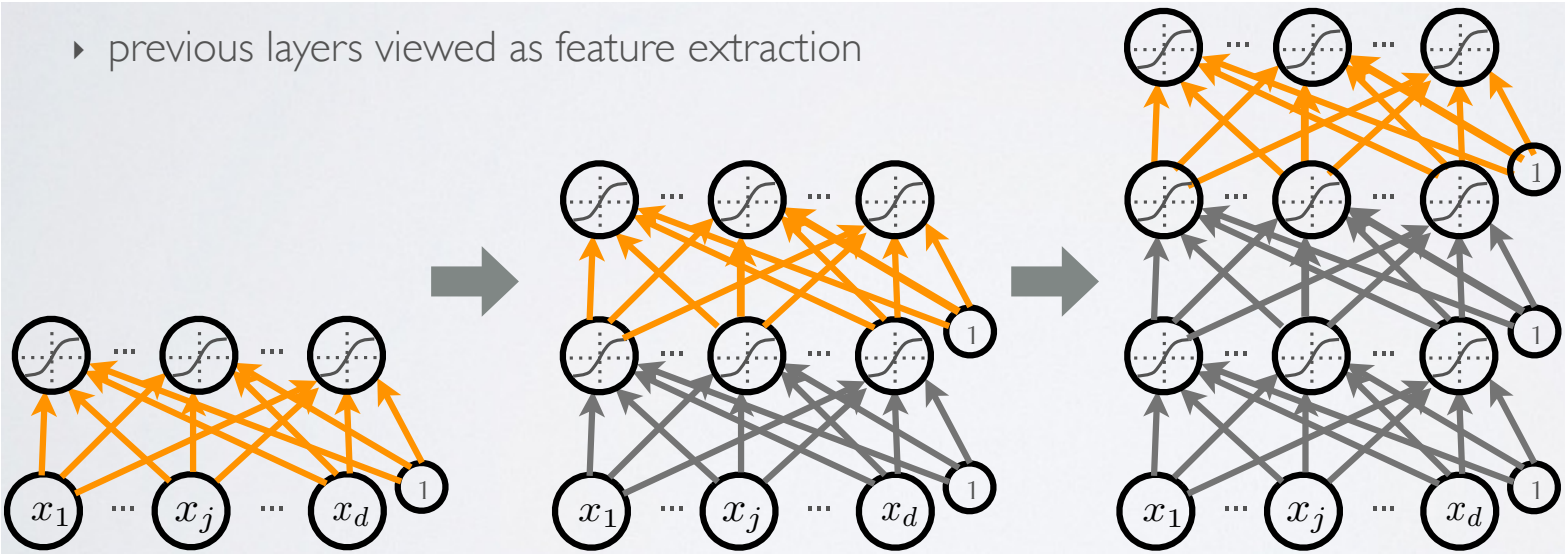


- Decoder (for binary inputs): $\hat{\mathbf{x}} = o(\hat{\mathbf{a}}(\mathbf{x})) = sigm(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))$
- Encoder $\mathbf{h}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x})) = sigm(\mathbf{b} + \mathbf{W}\mathbf{x})$
- Loss function (binary) $-\sum_k (x_k \log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k))$
- Loss function for real-valued inputs $\frac{1}{2}\sum_k (\hat{x}_k - x_k)^2$

Unsupervised pre-training

Greedy, layer-wise procedure

- train one layer at a time, from 1st to last, with unsupervised criterion
- Fix the parameters of previous hidden layers, these layers are seen as **feature extraction**



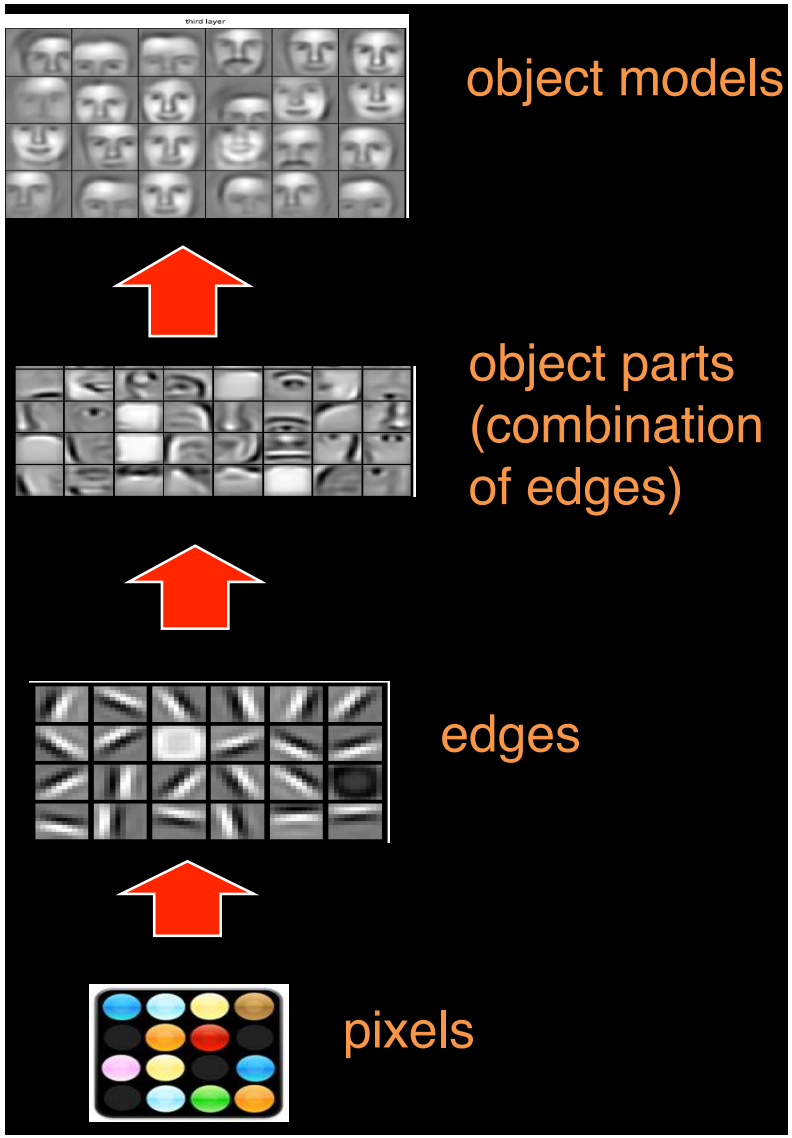
Unsupervised pre-training

This procedure is called unsupervised pre-training

- **first layer**: find hidden unit features that are more common in training inputs than random hidden unit features
- **second layer**: find combinations of hidden unit features that are more common than random unit features
- **third layer**: fond combinations of combinations ...
- etc.

Pre-training initializes the parameters in a region such that teh near local optimal overfit less the data

Application in computer vision



Fine-Tuning

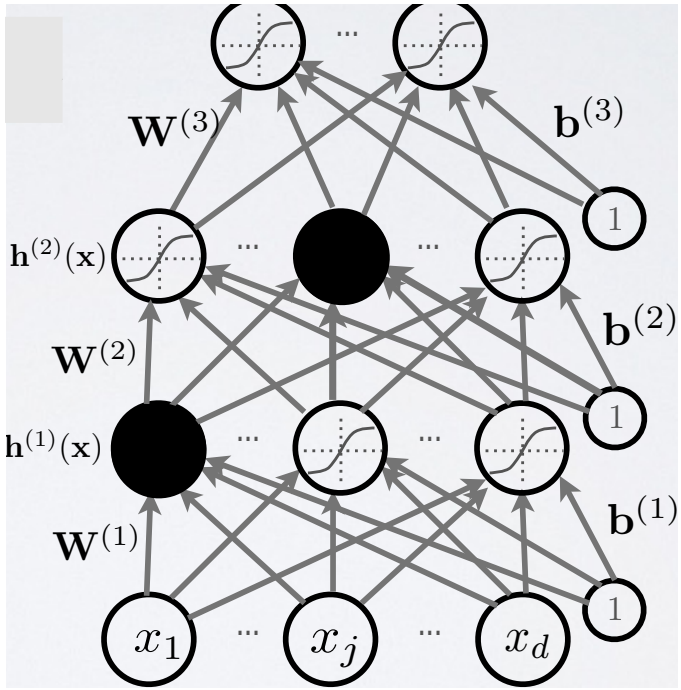
- Once all layers are pre-trained
  add output layer and train the while network using supervised learning
- Supervised learning is performed as seen previously (forward propagation, back propagation, updates, ...)
- This last phase is called **fine-tuning**
  All parameters are "tuned" for the supervised task at hand
  representation is adjusted to be more discriminative

Pseudo-code

- for $l = 1$ to L - (pre-training)
  - Build unsupervised training set $S = \{\mathbf{h}^{(l-1)}(\mathbf{x}^{(i)})\}_{i=1}^{T}$, $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$
  - train "greedy module" (autoencode, Restricted Boltzman Machine, ...) on S.
  - parameters found are used to initialized the networks parameters $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$
- Initialize $\mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}$ randomly
- Train the while network with supervised stochastic gradient descent (with backprop) - this is fine tuning.

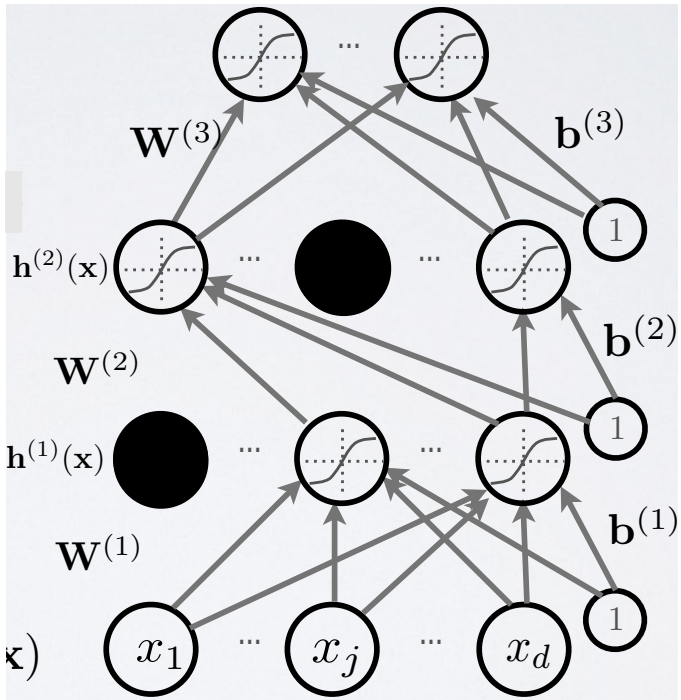
Dropout

- Idea: "cripple" neural network by removing hidden units stochastically
  - each hidden unit is set to 0 with prob 0.5
  - hidden unit must be more generally useful
- Could use a different dropout probability, but 0.5 usually works well.



Dropout

- In practice: use random binary masks $m^{(k)}$ and plug it into backprop
  $\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \circ m^{(k)}$
- At test time, masks are replaced by their expectation (constant vector 0.5 is dropout proba is 0.5)
- Can be combined with unsupervised pre-training.



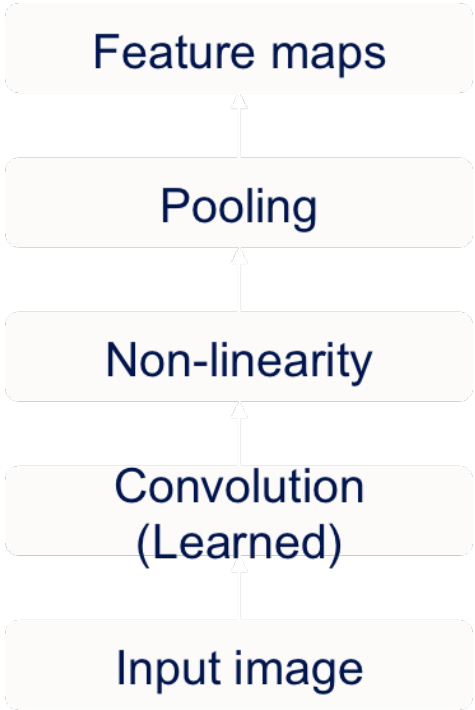
Other idea

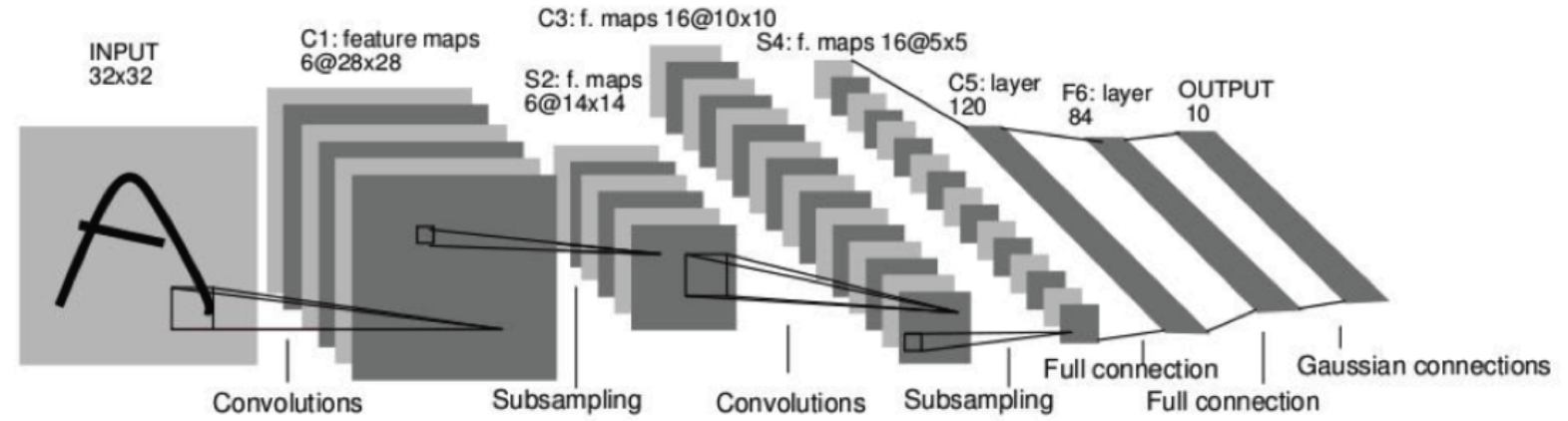
Batch normalization

- Take mean and variance over a mini-batch
  $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$,
  $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$
  $\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$,
  output $y_i = \gamma \hat{x}_i + \beta$
- Help to avoid saturated activations
- Adds stochasticity to training which might regularize
- At test time, can use global mean and stddev.

Convolutional Neural Networks (CNN)

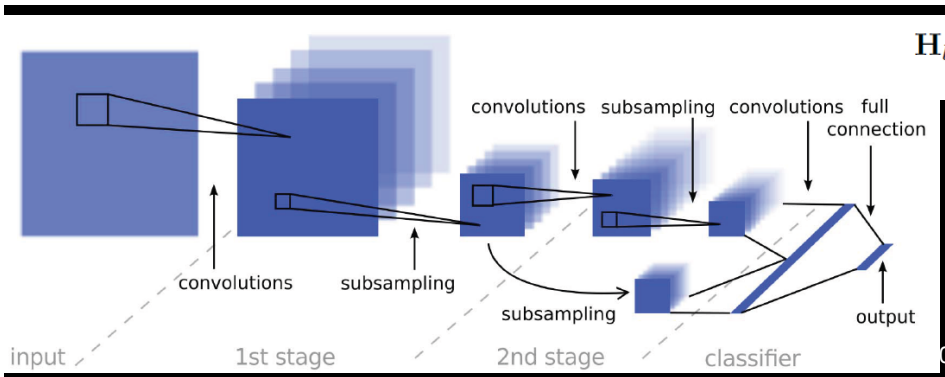
- A special kind of multi-layer neural network
- Implicitly extract relevant features.
- $\rightarrow$ a feed-forward network that can extract topological properties from an image.
- trained with a version of the back propagation algorithm.
- Suitable for signal processing applications (computer vision, speech recognition)





CNN: overview

- NN with specialized connectivity structure
- Feed-forward: Convolve input, non linearity (Reclin), pooling.



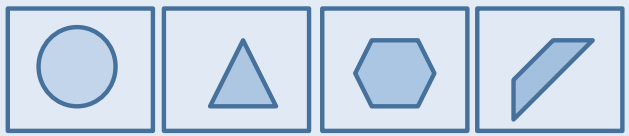
$$\mathbf{H}_{lp} = \text{pool}\left(\tanh\left(b_{lp} + \sum_{q \in \text{parents}(p)} \mathbf{w}_{lpq} * \mathbf{H}_{l-1,q}\right)\right)$$

012]

- A CNN consists of three modules: Convolution Squashing Pooling

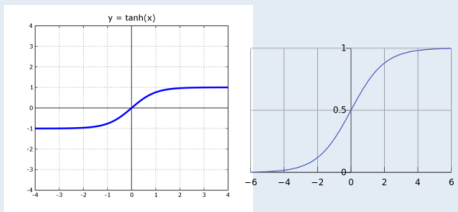
Convolution

- Multiple convolution filters acts as pattern detectors



Activation Function

- Typically tanh(.) or sig(.)

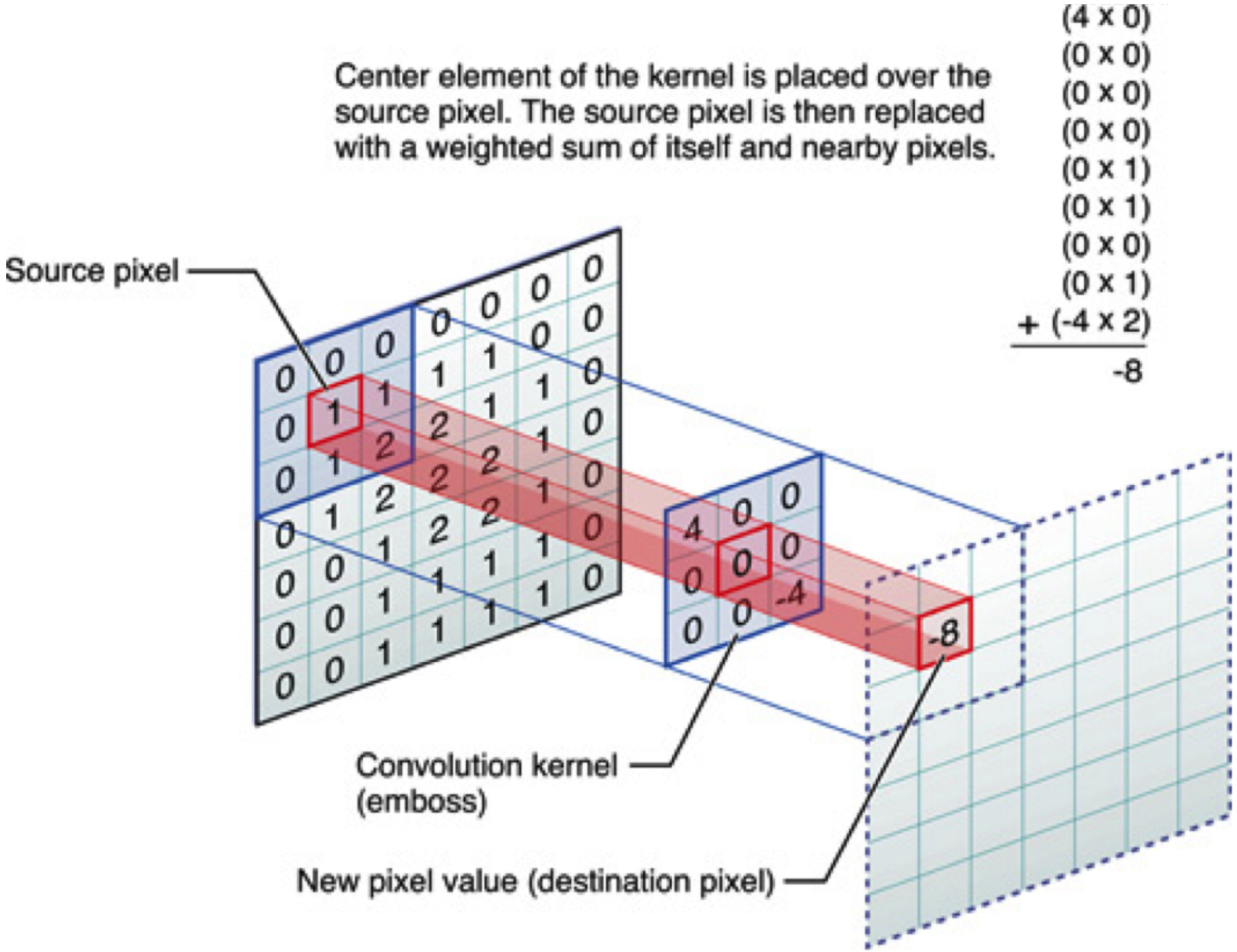


Pooling

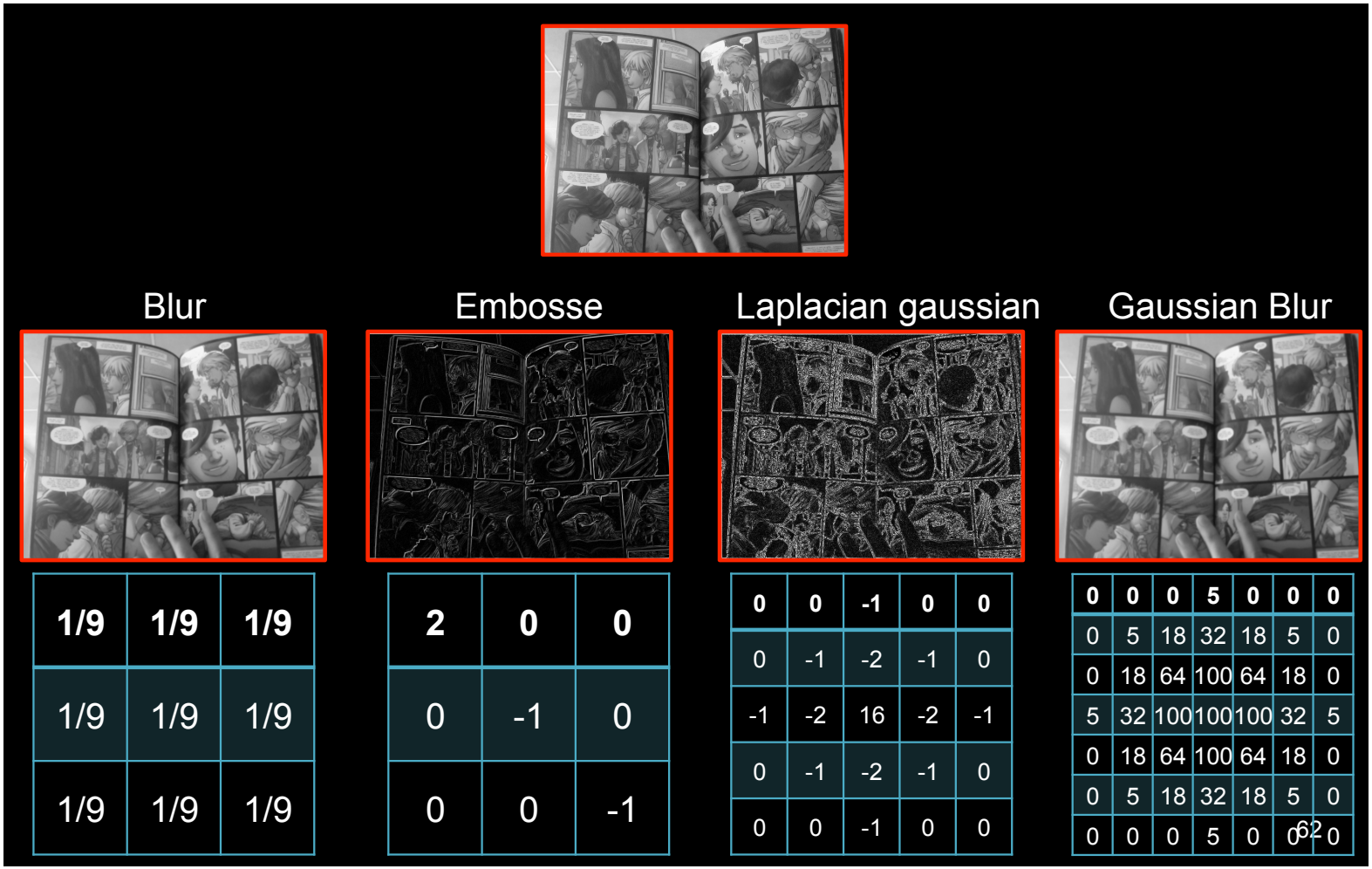
- Invariance to feature locations.

average pooling
max pooling
LP pooling

Discrete convolution



Effect of the convolution mask



Blur

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Embosse

2	0	0
0	-1	0
0	0	-1

Laplacian gaussian

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

Gaussian Blur

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

62

Effect of the different average filter sizes



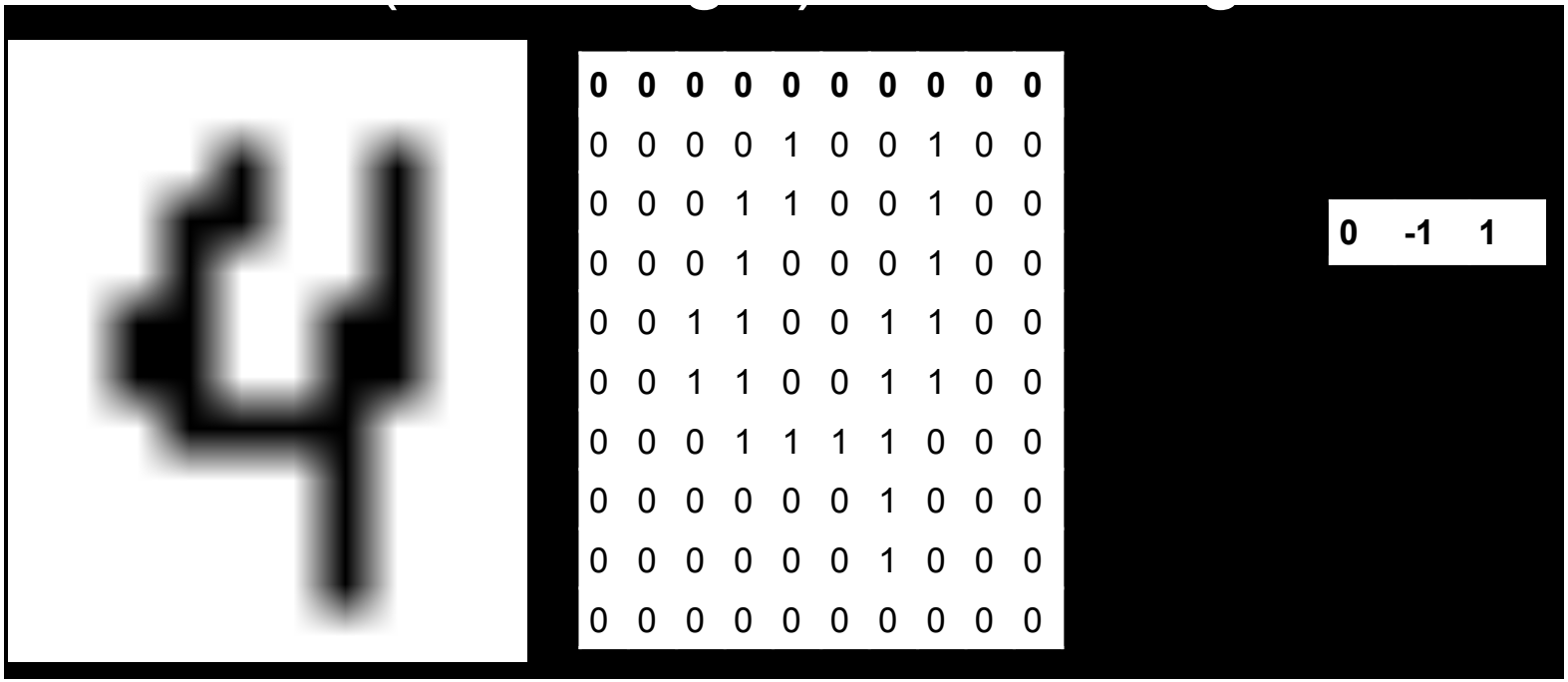
1/9	**1/9**	**1/9**
1/9	1/9	1/9
1/9	1/9	1/9

3X3 5X5 7X7 Average Filter



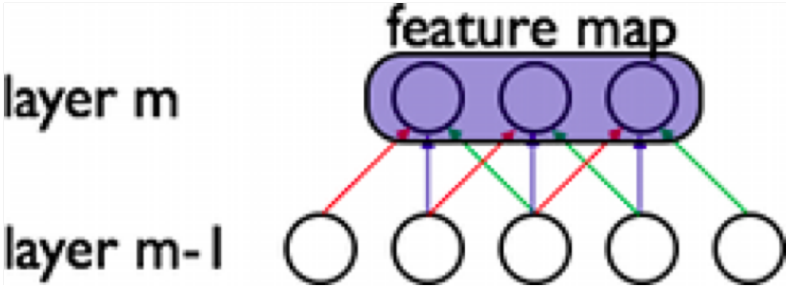
3X3 5X5 7X7 Average Filter

Try with a 1×3 filter



http://beej.us/blog/data/convolution-image-processing/

About Learning Convolutions

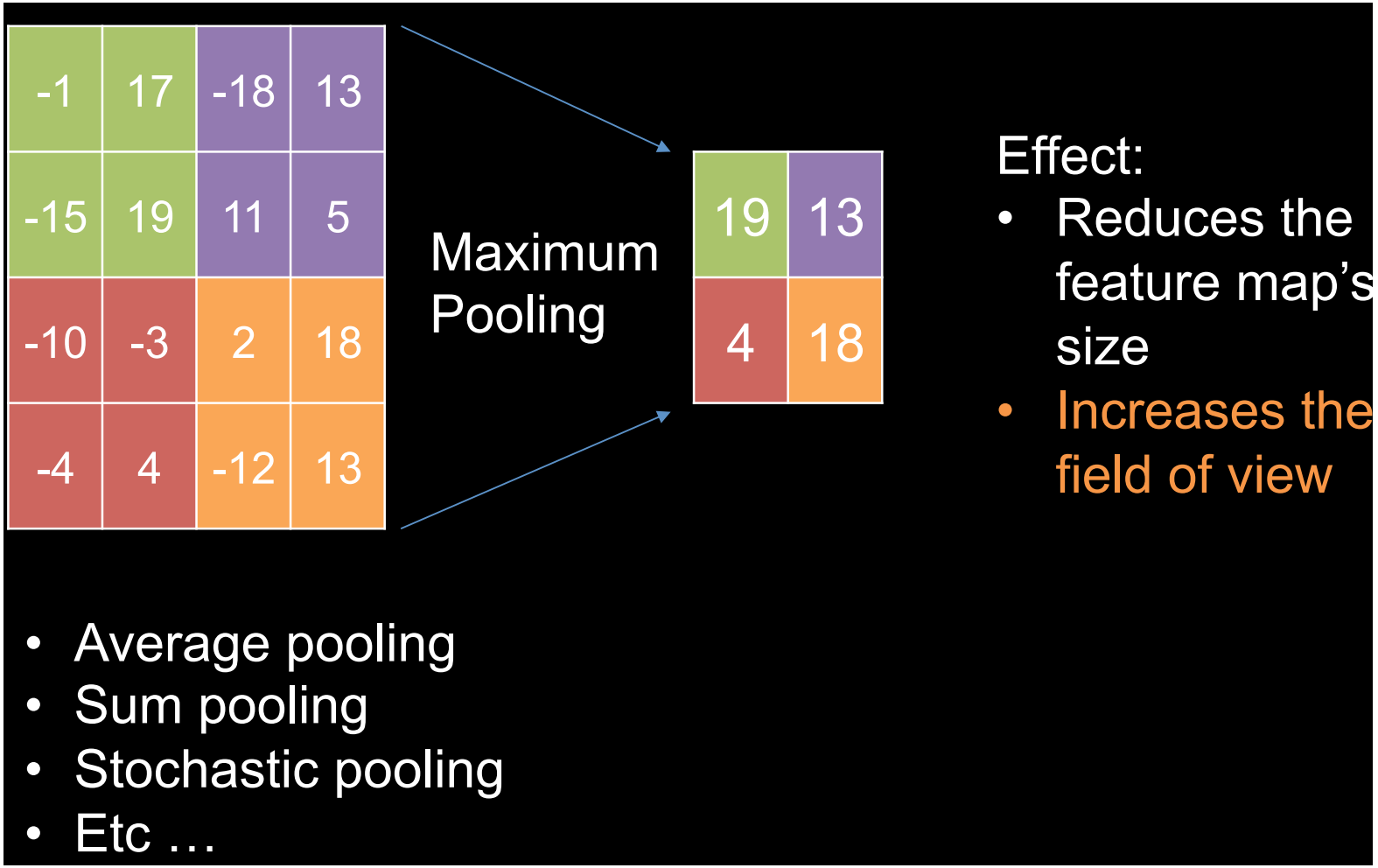


- The size of the convolution kernel is fixed depending on the application. The parameters of the kernel (the weights = the type of filter) correspond to the connexions from one layer to another (this is what is learned !). The resulted convolved image is called a feature map.
- The same convolution kernel is slided over the entire image so the weights to construct each convoluted pixels are shared ((colors "red, green, blue") in the above image)
- The smaller the convolution kernel, the less parameters to learn
- Depending of the size of the filter, the image size can be reduced ((dim image – dim kernel) / (stepsize for sliding)) + 1

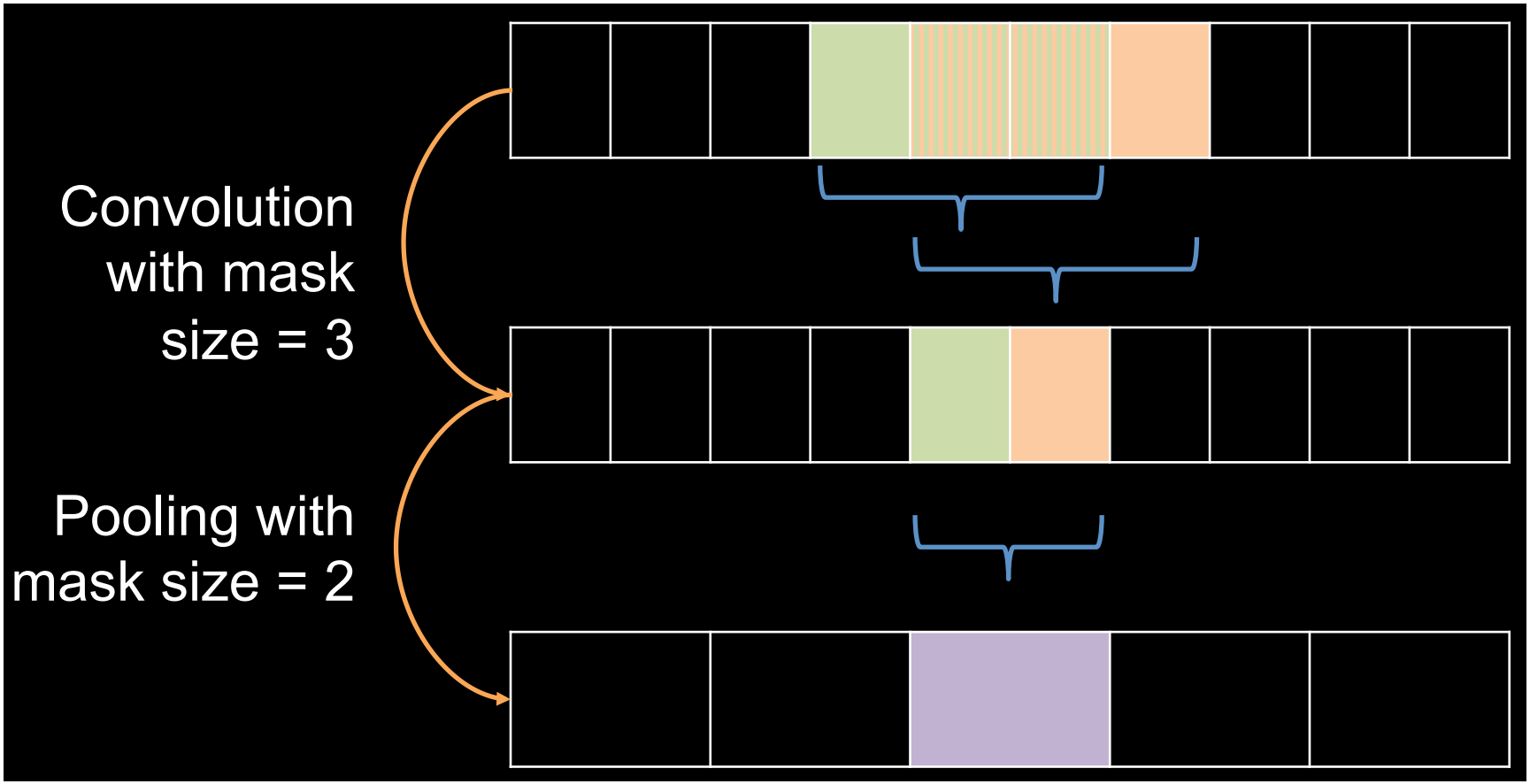
Pooling

- In general terms, the objective of pooling is to transform the joint feature representation into a new, more usable one that preserves important information while discarding irrelevant detail, the crux of the matter being to determine what falls in which category.
- Achieving invariance to changes in position or lighting conditions, robustness to clutter, and compactness of representation, are all common goals of pooling.
- Speed up the process (less parameters)

Example of Pooling

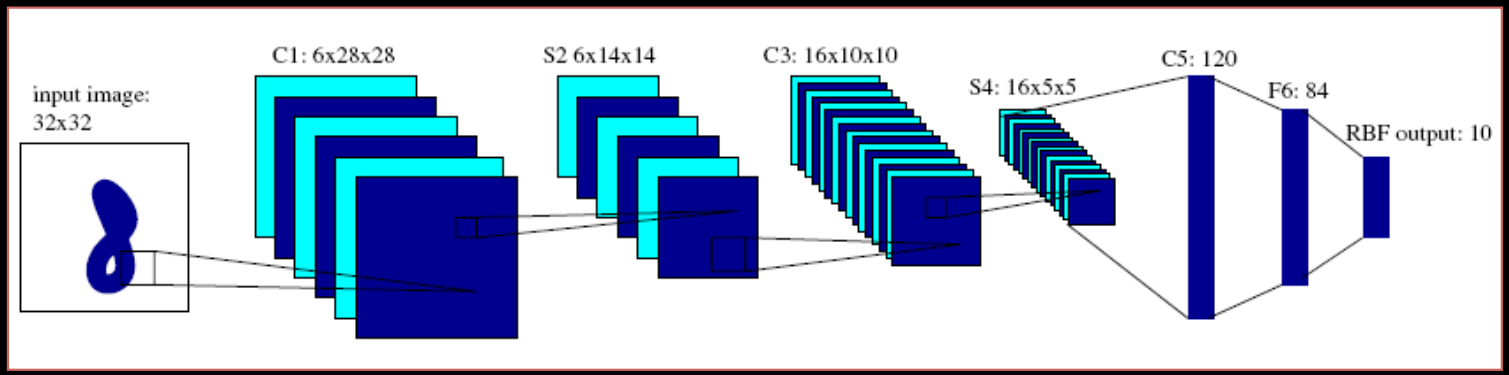


Field of View

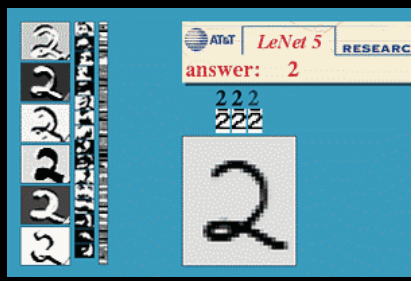


Field of View

Example: LeNet5



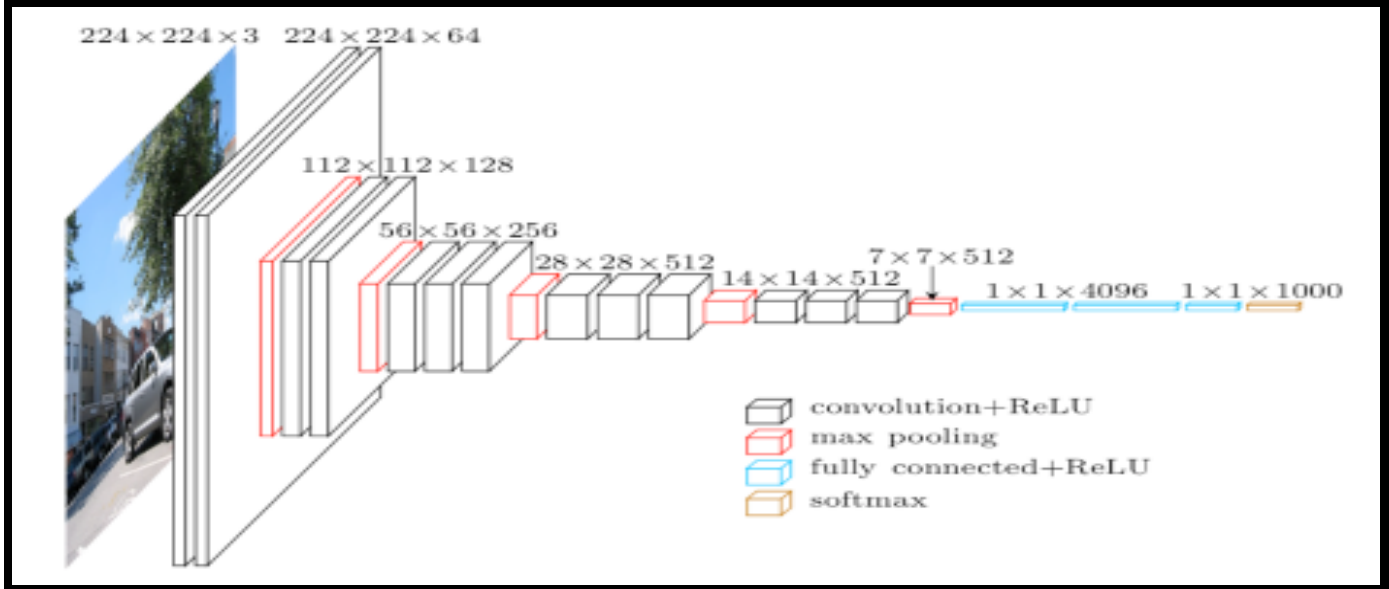
- C1,C3,C5 : Convolutional layers (**5 × 5 × ?** convolution kernels (2D size given)
- S2 , S4 : Subsampling layer. (by factor 2)
- F6 : Fully connected layer.
- Nb of feature maps (6, 16, 120 and then 84) is given



Try it out: http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

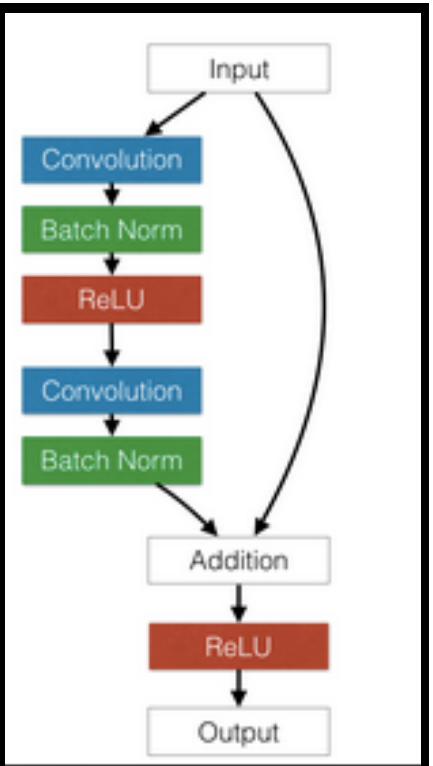
73

Classic current architectures



VGG16 (16 conv layers)
https://arxiv.org/abs/1409.1556

Winner Imagenet Challenge 2014 (localisation + classificaiton)



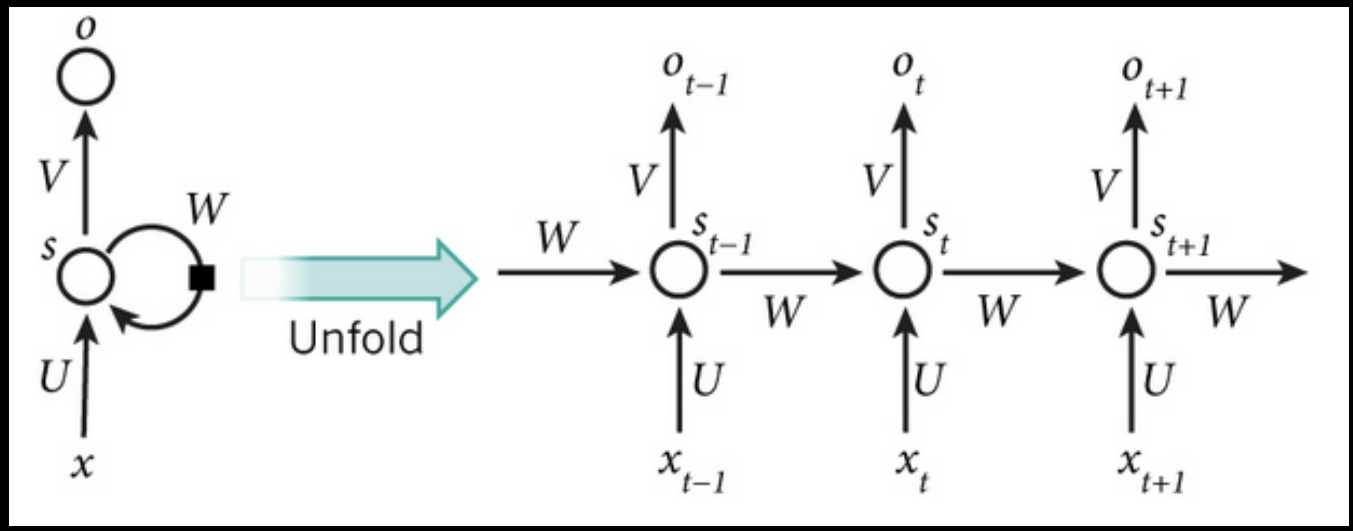
Deep Residual Learning for Image Recognition

CVPR 2016

Recurrent Neural Networks

Recurrent networks

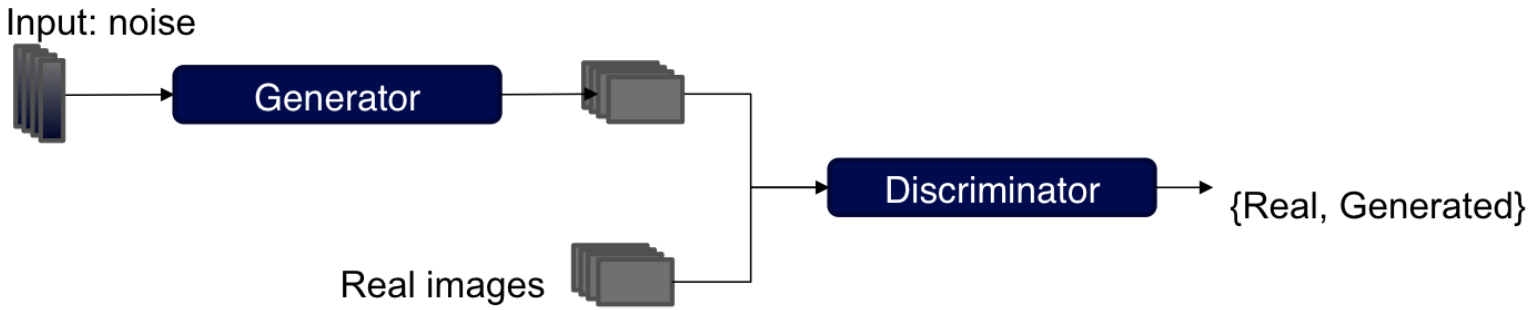
Source: Nature

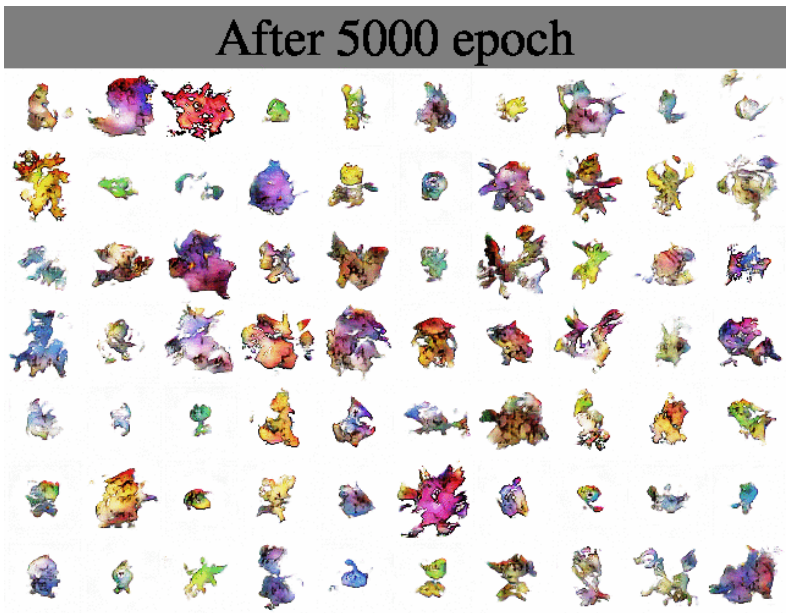


A recurrent neural network and the unfolding in time of the computation involved in its forward computation (S_t = « memory of the network »). $s_t = f(Ux_t + Ws_{t-1})$

Perform the same task for each element of a sequence with the output being depended on the previous computations... they have a "memory" which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps (ex: LSTM are popular RNN see `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

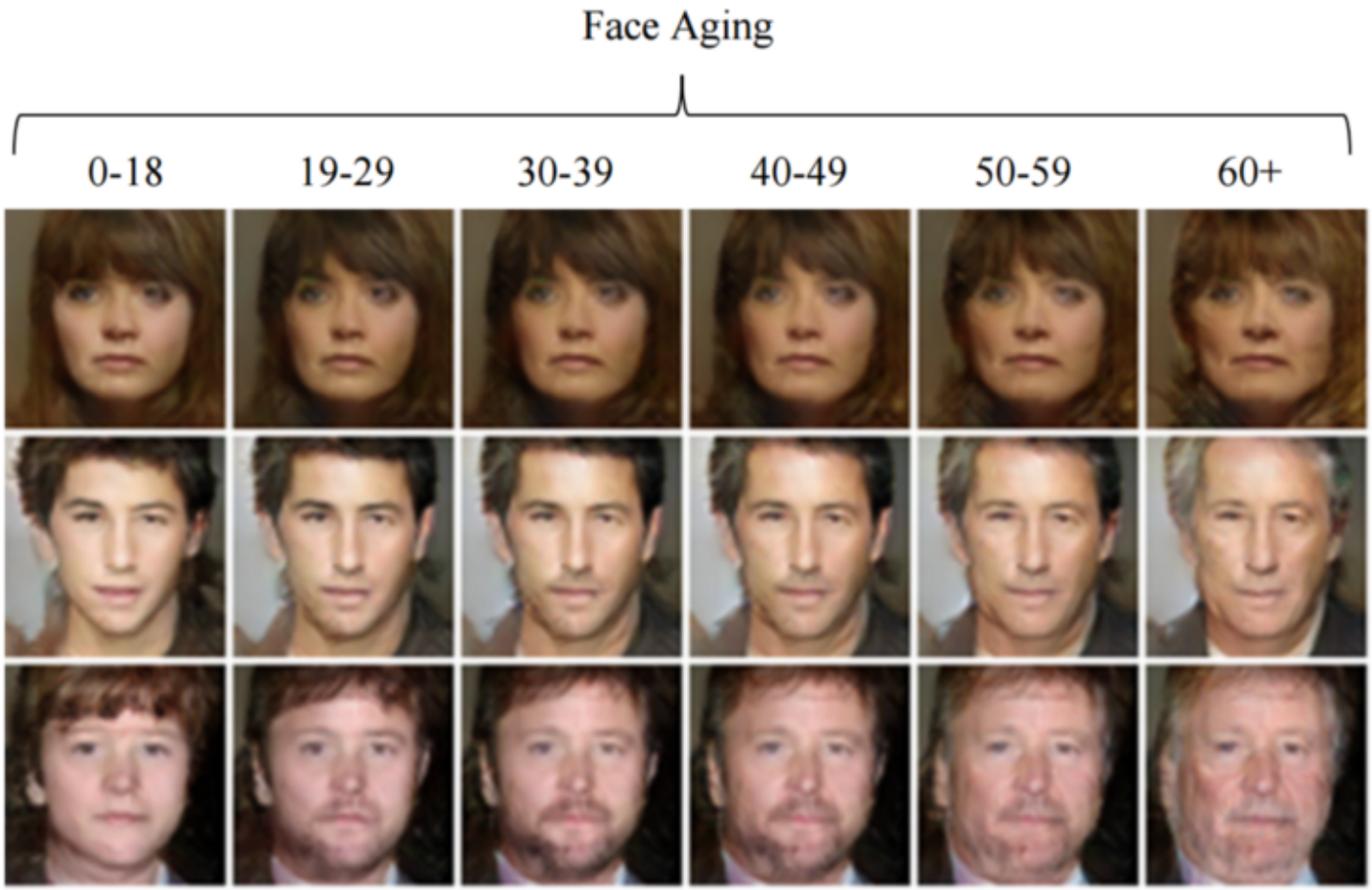
Generative adversarial Nets (GAN)





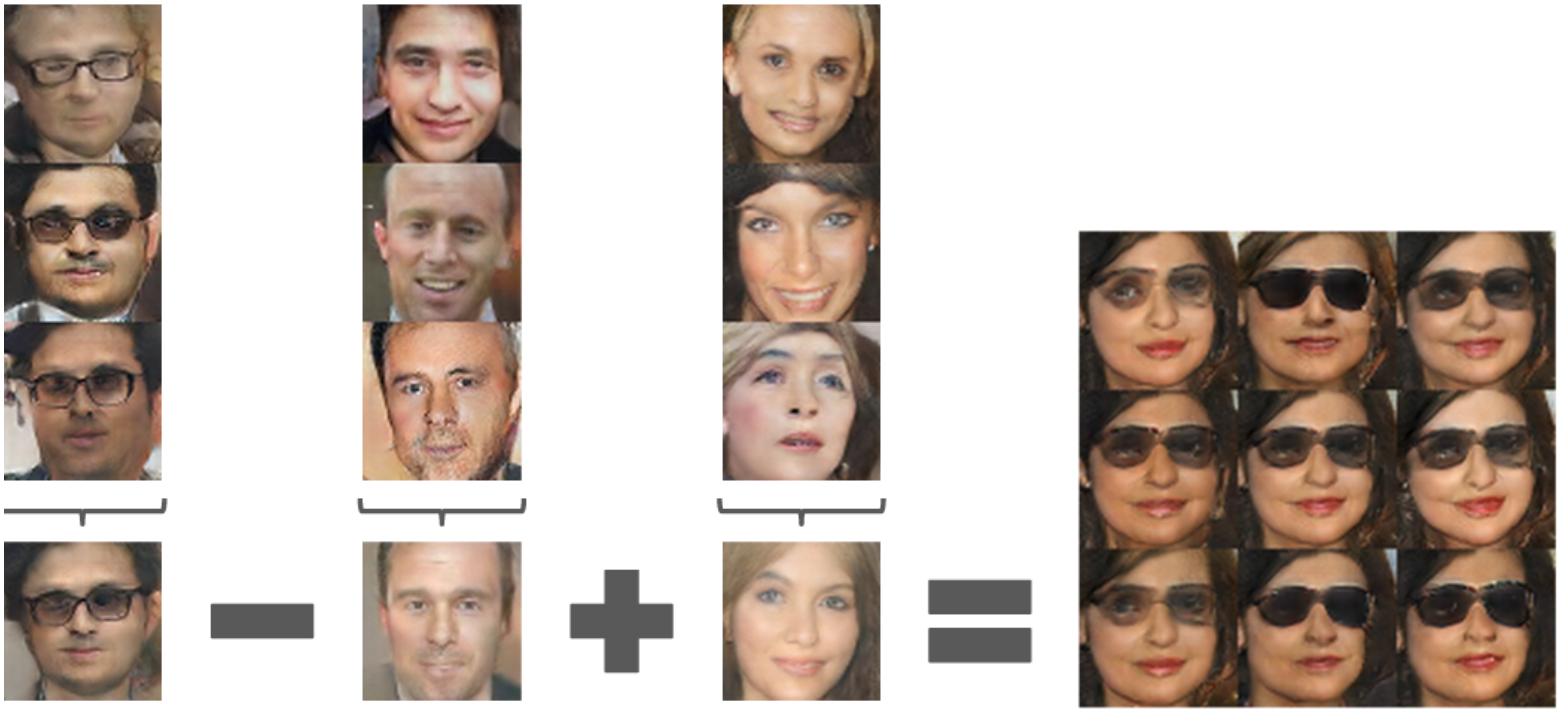
- The **Discriminator** is trained to **discriminate** real from generated images
- The **Generator** is trained to **"fool"** the discriminator (adversarial way)
- During the learning phase, neither the **Generator** nor the **Discriminator** should become stronger than the other

An example



From Antipov et al., Arxiv1702.01983, 2017

Another example

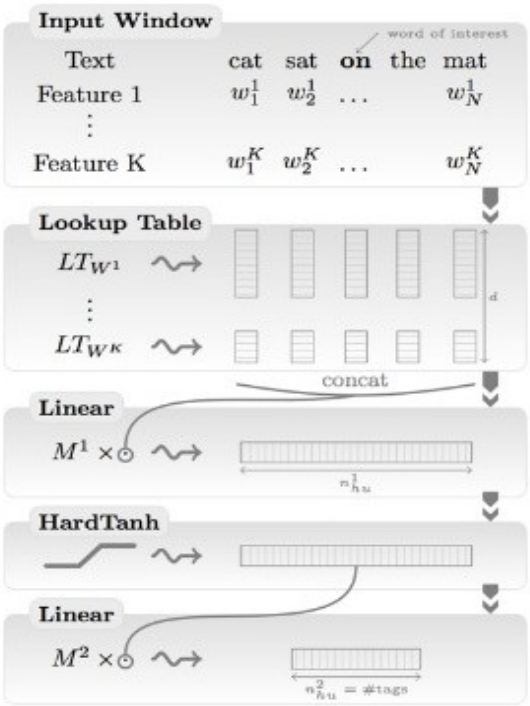


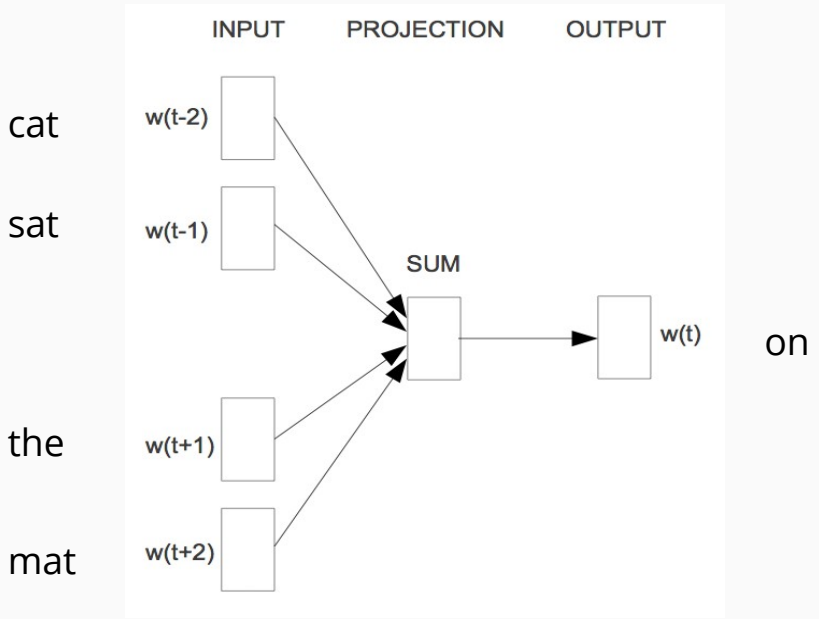
From Radford et al., ICLR'16

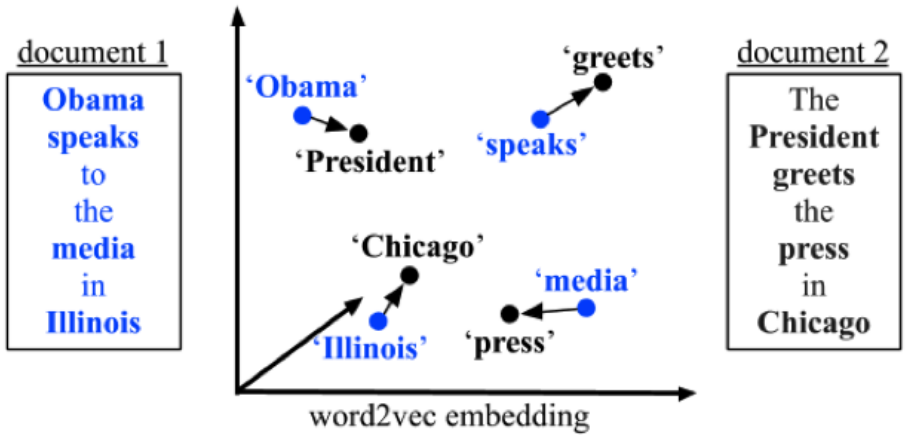
Another example

From Radford et al., ICLR'16

Learning Word Embeddings







Conclusion

- Important area today
- Allows to find relevant representation in an automatic way
- Has improved state of the art results in a lot of area
  Among the 10 breakthough technologies in 2013 (MIT Technology Review)
- Structure is hard to define (test and trial), training is very time consuming
- No real theoretical guarantees
- Nice demo tool `http://playground.tensorflow.org/`
- Existing libraries
  TensorFlow (Google), Torch (Facebook/Twitter, Deepmind), Theano (LISA lab)