

# Applied inductive learning - Lecture 5

## (Deep) Neural Networks

Louis Wehenkel & Pierre Geurts

Department of Electrical Engineering and Computer Science  
University of Liège

Montefiore - Liège - October 20, 2016

*Find slides:* <http://montefiore.ulg.ac.be/~lwh/AIA/>

## Introduction

### Single neuron models

Hard threshold unit (LTU) and the perceptron

Soft threshold unit and gradient descent

Theoretical properties

### Multilayer perceptron

Definition and expressiveness

Learning algorithms

Overfitting and regularization

### Other neural network models

Radial basis function networks

Convolutional neural networks

Recurrent neural networks

## Conclusion

# Batch-mode vs Online-mode Supervised Learning

- ▶ Objects (or observations):  $LS = \{o_1, \dots, o_N\}$
- ▶ Attribute vector:  $\mathbf{a}^i = (a_1(o_i), \dots, a_n(o_i))^T, \quad \forall i = 1, \dots, N.$
- ▶ Outputs:  $y^i = y(o_i)$  or  $c^i = c(o_i), \quad \forall i = 1, \dots, N.$
- ▶ LS Table

$o$	$a_1(o)$	$a_2(o)$	$\dots$	$a_n(o)$	$y(o)$
1	$a_1^1$	$a_2^1$	$\dots$	$a_n^1$	$y^1$
2	$a_1^2$	$a_2^2$	$\dots$	$a_n^2$	$y^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$N$	$a_1^N$	$a_2^N$	$\dots$	$a_n^N$	$y^N$

Focus for this lecture on numerical inputs, and numerical outputs (classes will be encoded numerically if needed).

# Batch-mode vs online mode learning

- ▶ In batch-mode
  - ▶ Samples provided and processed **together** to **construct** model
  - ▶ Need to store samples (not the model)
  - ▶ Classical approach for data mining
- ▶ In online-mode
  - ▶ Samples provided and processed **one by one** to **update** model
  - ▶ Need to store the model (not the samples)
  - ▶ Classical approach for adaptive systems
- ▶ But both approaches can be adapted to handle both contexts
  - ▶ Samples available together can be exploited one by one
  - ▶ Samples provided one by one can be stored and then exploited together

# Motivations for Artificial Neural Networks

Intuition: biological brain can learn, so let's try to be inspired by it to build learning algorithms.

- ▶ Starting point: single neuron models
  - ▶ perceptron, LTU and STU for linear supervised learning
  - ▶ online (biologically plausible) learning algorithms
- ▶ Complexify: multilayer perceptrons
  - ▶ flexible models for non-linear supervised learning
  - ▶ universal approximation property
  - ▶ iterative training algorithms based on non-linear optimization
- ▶ ...other neural network models of importance

## Introduction

### Single neuron models

Hard threshold unit (LTU) and the perceptron

Soft threshold unit and gradient descent

Theoretical properties

### Multilayer perceptron

Definition and expressiveness

Learning algorithms

Overfitting and regularization

### Other neural network models

Radial basis function networks

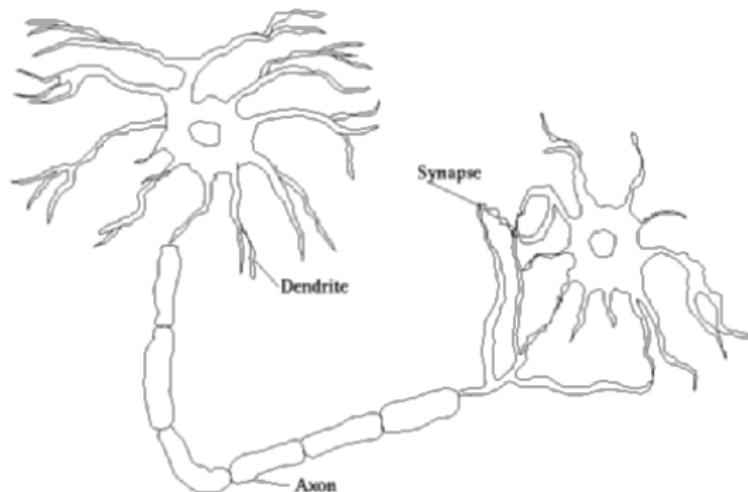
Convolutional neural networks

Recurrent neural networks

## Conclusion

# Single neuron models

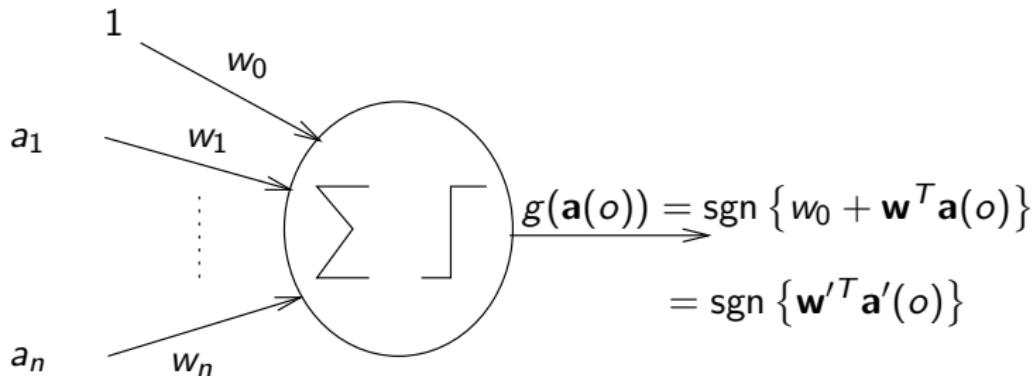
The biological neuron:



Human brain:  $10^{11}$  neurons, each with  $10^4$  synapses  
Memory (knowledge): stored in the synapses

# Hard threshold unit...

A simple (simplistic) mathematical model of the biological neuron



Parameters to adapt to problem:  $\mathbf{w}'$

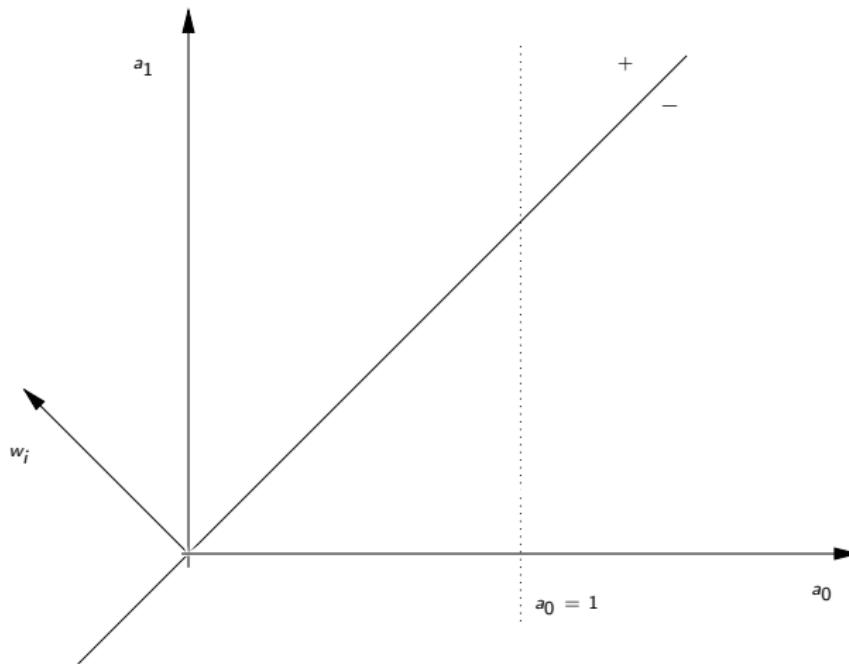
## ...and the perceptron learning algorithm

1. For binary classification:  $c(o) = \pm 1$ .
2. Start with an arbitrary initial weight vector, e.g.  $\mathbf{w}'_0 = \mathbf{0}$ .
3. Consider the objects of the *LS* in a cyclic or random sequence.
4. Let  $o_i$  be the object at step  $i$ ,  $c(o_i)$  its class and  $\mathbf{a}(o_i)$  its attribute vector.
5. Adjust the weight by using the following correction rule,

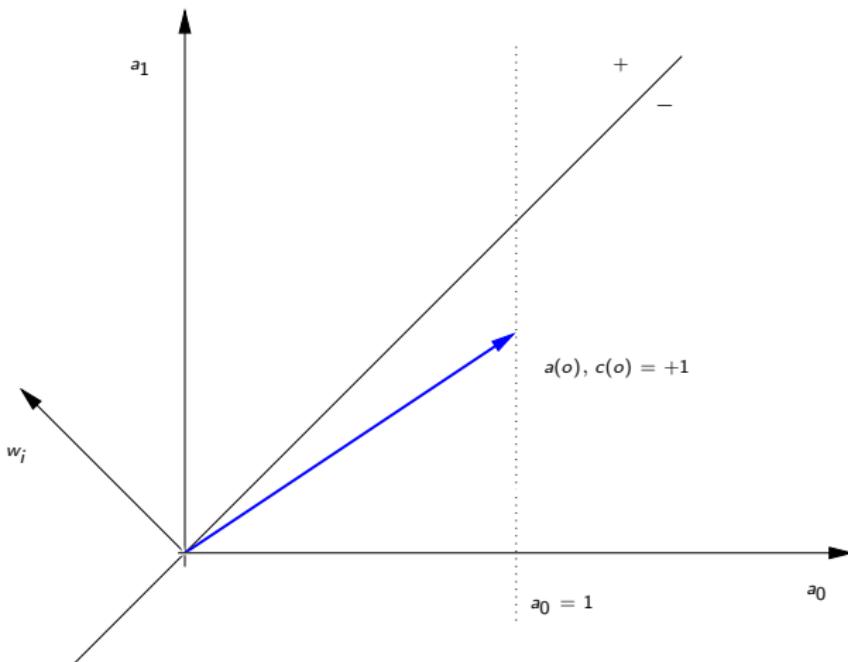
$$\mathbf{w}'_{i+1} = \mathbf{w}'_i + \eta_i (c(o_i) - g_i(\mathbf{a}(o_i))) \mathbf{a}'(o_i).$$

- ▶  $\mathbf{w}'_i$  changes only if  $o_i$  is not correctly classified
- ▶ it is changed in the **right** direction ( $\eta_i > 0$  is the learning rate)
- ▶ at any stage,  $\mathbf{w}'_i$  is a linear combination of the  $\mathbf{a}(o_i)$  vectors

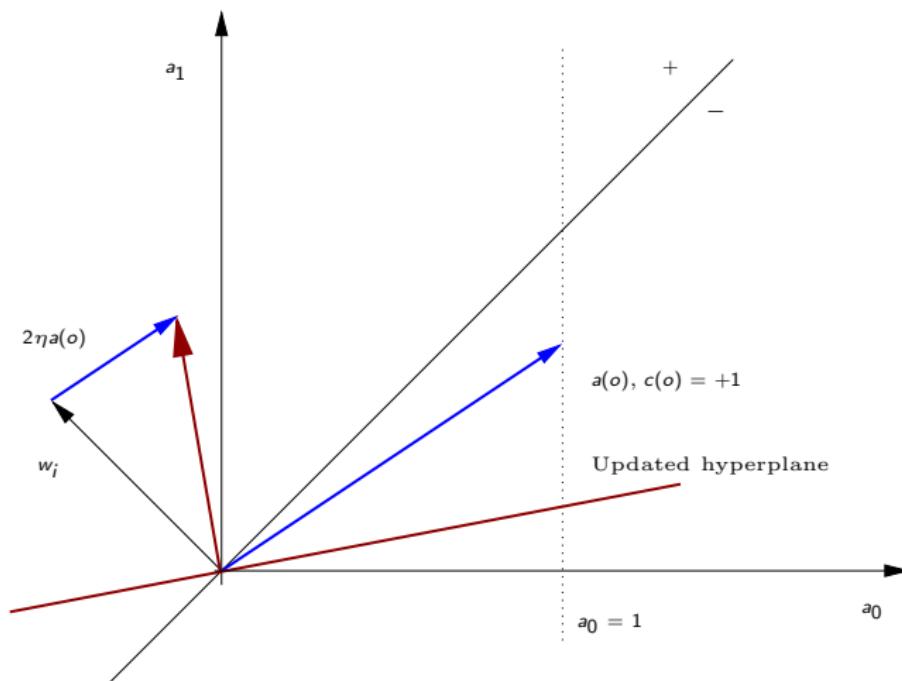
# Geometrical view of update equation



# Geometrical view of update equation



# Geometrical view of update equation



# Soft threshold units (STU)...

The input/output function  $g(\mathbf{a})$  of such a device is computed by

$$g(\mathbf{a}(o)) \stackrel{\triangle}{=} f(w_0 + \mathbf{w}^T \mathbf{a}(o)) = f(\mathbf{w}'^T \mathbf{a}'(o))$$

where the *activation* function  $f(\cdot)$  is assumed to be differentiable. Classical examples of activation functions are the sigmoid

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)},$$

and the hyperbolic tangent

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)},$$

## ... and gradient descent

Find vector  $\mathbf{w}'^T = (w_0, \mathbf{w}^T)$  minimizing the square error (TSE)

$$TSE(LS, \mathbf{w}') = \sum_{o \in LS} (g(\mathbf{a}(o)) - y(o))^2 = \sum_{o \in LS} \left( f(\mathbf{w}'^T \mathbf{a}'(o)) - y(o) \right)^2.$$

The gradient with respect to  $\mathbf{w}'$  is computed by

$$\nabla_{\mathbf{w}'} TSE(LS, \mathbf{w}') = 2 \sum_{o \in LS} (g(\mathbf{a}(o)) - y(o)) f'(\mathbf{w}'^T \mathbf{a}'(o)) \mathbf{a}'(o),$$

where  $f'(\cdot)$  denotes the derivative of the activation function  $f(\cdot)$ .

The gradient descent method works by iteratively changing the weight vector by a term proportional to  $-\nabla_{\mathbf{w}'} TSE(LS, \mathbf{w}')$ .

# ... and stochastic online gradient descent

**Fixed** step gradient descent in online-mode:

1. For binary classification:  $c(o) = \pm 1$ .
2. Start with an arbitrary initial weight vector, e.g.  $\mathbf{w}'_0 = \mathbf{0}$ .
3. Consider the objects of the  $LS$  in a cyclic or random sequence.
4. Let  $o_i$  be the object at step  $i$ ,  $c(o_i)$  its class and  $\mathbf{a}(o_i)$  its attribute vector.
5. Adjust the weight by using the following correction rule,

$$\begin{aligned}\mathbf{w}'_{i+1} &= \mathbf{w}'_i - \eta_i \nabla_{\mathbf{w}'} SE(o_i, \mathbf{w}'_i) \\ &= \mathbf{w}'_i + 2\eta_i [c(o_i) - g_i(\mathbf{a}(o_i))] f'(\mathbf{w}'^T \mathbf{a}'(o_i)) \mathbf{a}'(o_i),\end{aligned}$$

( $SE(o, \mathbf{w}')$  is the contribution of object  $o$  in  $TSE(LS, \mathbf{w}')$ .)

# Theoretical properties

- ▶ Convergence of the perceptron learning algorithm
  - ▶ If  $LS$  is linearly separable: converges in a finite number of steps.
  - ▶ Otherwise: converges with infinite number of steps, if  $\eta_i \rightarrow 0$ .
- ▶ Convergence of the online or batch gradient descent algorithm
  - ▶ if  $\eta_i \rightarrow 0$  (slowly), and infinite number of steps, same solution
  - ▶ if  $f(\cdot)$  linear, finds same solution as linear regression

NB: slow  $\eta_i \rightarrow 0$  means

- ▶  $\lim_{m \rightarrow \infty} \sum_{i=1}^m \eta_i = +\infty$
- ▶  $\lim_{m \rightarrow \infty} \sum_{i=1}^m \eta_i^2 < +\infty$

## Introduction

### Single neuron models

Hard threshold unit (LTU) and the perceptron

Soft threshold unit and gradient descent

Theoretical properties

### Multilayer perceptron

Definition and expressiveness

Learning algorithms

Overfitting and regularization

### Other neural network models

Radial basis function networks

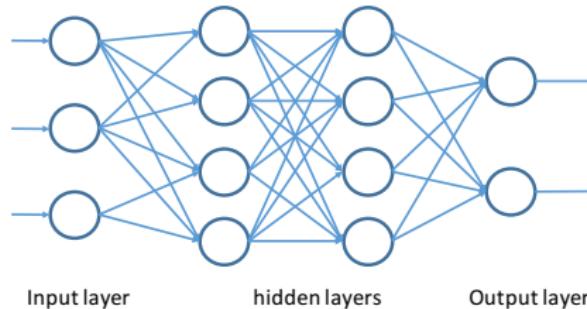
Convolutional neural networks

Recurrent neural networks

## Conclusion

# Multilayer perceptron

- ▶ Single neuron models are not more expressive than linear models
- ▶ Solution: connect several neurons to form a potentially complex non-linear parametric model
- ▶ Most common non-linear ANN structure is **multilayer perceptron**, i.e., multiple layers of neurons, with each layer fully connected to the next.
- ▶ E.g., MLP with 3 inputs, 2 hidden layers of 4 neurons each, and 2 outputs



# Multilayer perceptron: mathematical definition (1/3)

$L$  number of layers

- ▶ Layer 1 is the input layer
- ▶ Layer  $L$  is the output layer
- ▶ Layers 2 to  $L - 1$  are the hidden layers

$s_l$  ( $1 \leq l \leq L$ ): number of neurons in the  $l$ th layer ( $s_1 (= n)$  is the number of inputs,  $s_L$  is the number of outputs)

$a_i^{(l)}(o)$  ( $1 < l \leq L, 1 \leq i \leq s_l$ ): the activation (i.e., output) of the  $i$ th neuron of layer  $l$  for an object  $o$ .

$f^{(l)}$  ( $2 \leq l \leq L$ ): the activation function of layer  $l$

$w_{i,j}^{(l)}$  ( $1 \leq i \leq s_{l+1}, 1 \leq j \leq s_l$ ): the weight of the edge from neuron  $j$  in layer  $l$  to neuron  $i$  in layer  $l + 1$

$w_{i,0}^{(l)}$  ( $1 \leq i \leq s_{l+1}$ ): the bias/intercept of neuron  $i$  in layer  $l + 1$ .

# Multilayer perceptron: mathematical definition (2/3)

Predictions can be computed recursively:

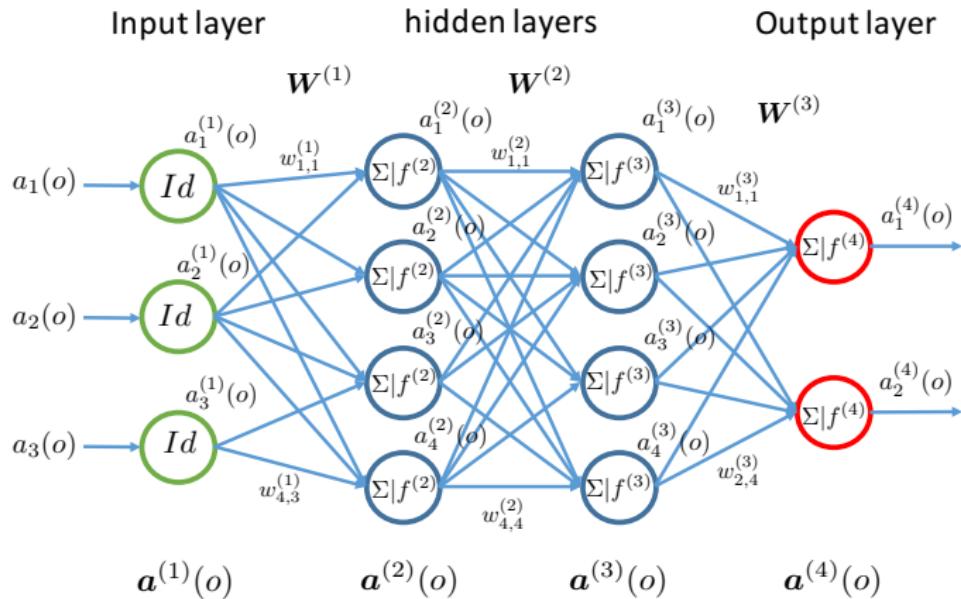
$$\begin{aligned} a_i^{(1)}(o) &= a_i(o), & \forall i : 1 \leq i \leq n \\ a_i^{(l+1)}(o) &= f^{(l+1)}(w_{i,0}^{(l)} + \sum_{j=1}^{s_l} w_{i,j}^{(l)} a_j^{(l)}(o)), & \forall 1 < l < L, 1 \leq i \leq s_l \end{aligned} \quad (1)$$

Or in matrix notation:

$$\begin{aligned} \mathbf{a}^{(1)}(o) &= \mathbf{a}(o), \\ \mathbf{a}^{(l+1)}(o) &= f^{(l+1)}(\mathbf{W}'^{(l)} \mathbf{a}'^{(l)}(o)) \quad \forall 1 < l < L, \end{aligned}$$

with  $\mathbf{W}'^{(l)} \in IR^{s_{l+1} \times s_l + 1}$  defined as  $(\mathbf{W}'^{(l)})_{i,j} = w_{i,j-1}^{(l)}$  and  $\mathbf{a}'$  defined as previously.

# Multilayer perceptron: mathematical definition (3/3)



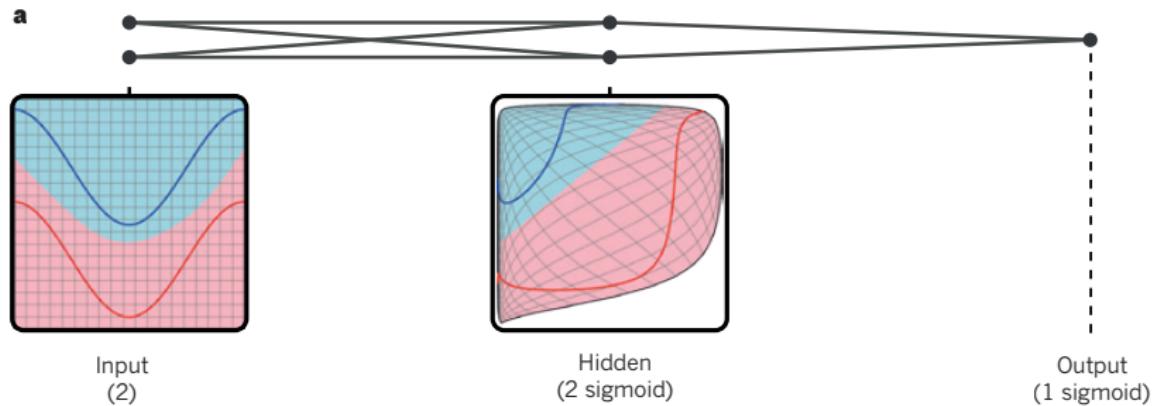
$$\mathbf{a}^{(l+1)}(o) = f^{(l)}(\mathbf{W}'^{(l)} \mathbf{a}'^{(l)}(o))$$

( $w_{i,0}^{(l)}$  weights are omitted from all figures)

# Representation capacity of MLP: classification (1/2)

- ▶ Geometrical insight in the representation capacity
- ▶ Two hidden layers of hard threshold units
  - ▶ First hidden layer: can define a collection of hyperplanes/semiplanes
  - ▶ Second hidden layer: can define arbitrary intersections of semiplanes
  - ▶ Output layer: can define arbitrary union of intersections of semi-planes
  - ▶ Conclusion: with a sufficient number of units, very complex regions can be described
- ▶ Soft threshold units:
  - ▶ hidden layers can distort the input space to make the classes linearly separable by the output layer

# Representation capacity of MLP: classification (2/2)



(lecun et al., Nature, 2015)

<http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

# Representation capacity of MLP: regression

- ▶ Function approximation insight
- ▶ One hidden layer of soft threshold units
  - ▶ One-dimensional input space illustration
  - ▶ Hidden layer defines  $K$  offset and scale parameters  $\alpha_i, \beta_i, i = 1 \dots K$ : responses  $f(\alpha_i x + \beta_i)$
  - ▶ Output layer (linear):  $\hat{y}(x) = b_0 + \sum_{i=1}^K b_i f(\alpha_i x + \beta_i)$
- ▶ Theoretical results:
  - ▶ Every bounded continuous function can be approximated with arbitrary small error
  - ▶ Any function can be approximated to arbitrary accuracy by a network with two hidden layers

# Learning algorithms for multilayer perceptrons

Main idea:

- ▶ Define a loss function that compares the output layer predictions (for an object  $o$ ) to the true outputs (with  $\mathcal{W}$  all network weights):

$$L(\mathbf{g}(\mathbf{a}(o); \mathcal{W}), \mathbf{y}(o))$$

- ▶ Training = finding the parameters  $\mathcal{W}$  that minimizes average loss over the training data

$$\mathcal{W}^* = \arg \min_{\mathcal{W}} \frac{1}{N} \sum_{o \in LS} L(\mathbf{g}(\mathbf{a}(o); \mathcal{W}), \mathbf{y}(o))$$

- ▶ Use gradient descent to iteratively improve an initial value of  $\mathcal{W}$ .

Require to compute the following gradient (for all  $i, j, l, o$ ):

$$\frac{\partial}{\partial w_{i,j}^{(l)}} L(\mathbf{g}(\mathbf{a}(o); \mathcal{W}), \mathbf{y}(o))$$

# Backpropagation of derivatives

- ▶ These derivatives can be computed efficiently using the **backpropagation** algorithm.
- ▶ Let us derive this algorithm in the case of a single regression output, square error, and assuming that all activation functions are similar:

$$L(g(\mathbf{a}(o); \mathcal{W}), y(o)) = \frac{1}{2}(g(\mathbf{a}(o); \mathcal{W}) - y(o))^2 = \frac{1}{2}(a_1^{(L)}(o) - y(o))^2$$

- ▶ In the following, we will denote by  $z_i^{(l)}(o)$  ( $1 \leq l < L$ ,  $1 \leq i \leq s_l$ ) the values sent through the activation functions:

$$z_i^{(l)}(o) = w_{i,0}^{(l-1)} + \sum_{j=1}^{s_{l-1}} w_{i,j}^{(l-1)} a_j^{(l-1)}(o) \quad \mathbf{z}^{(l)}(o) = \mathbf{W}'^{(l)} \mathbf{a}'^{(l)}(o) \quad (2)$$

(We thus have  $a_i^{(l)}(o) = f(z_i^{(l)}(o))$ )

# Backpropagation of derivatives

Using the chain rule of partial derivatives, we have<sup>1</sup>:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} L(g(\mathbf{a}; \mathcal{W}), y) = \frac{\partial L(\dots)}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial w_{i,j}^{(l)}}$$

Given the definitions of  $a_i^{(l+1)}$  and  $z_i^{(l+1)}$ , the last two factors are computed as:

$$\frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} = f'(z_i^{(l+1)}) \quad \frac{\partial z_i^{(l+1)}}{\partial w_{i,j}^{(l)}} = a_j^{(l)} \text{ (with } a_0^{(l)} = 1)$$

and thus:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} L(\dots) = \frac{\partial L(\dots)}{\partial a_i^{(l+1)}} f'(z_i^{(l+1)}) a_j^{(l)}. \quad (3)$$

---

<sup>1</sup>Object argument ( $(o)$ ) is omitted to simplify the notations

# Backpropagation of derivatives

For the last (output) layer, we have:

$$\frac{\partial L(\dots)}{\partial a_1^{(L)}} = \frac{\partial}{\partial a_1^{(L)}} \left\{ \frac{1}{2} (a_1^L - y)^2 \right\} = (a_1^L - y)$$

For the inner (hidden) layers, we have ( $1 \leq l < L$ ):

$$\begin{aligned} \frac{\partial L(\dots)}{\partial a_i^{(l)}} &= \sum_{j=1}^{s_{l+1}} \frac{\partial L(\dots)}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \frac{\partial L(\dots)}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_{j=1}^{s_{l+1}} \frac{\partial L(\dots)}{\partial a_j^{(l+1)}} f'(z_j^{(l+1)}) w_{j,i}^{(l)} \end{aligned}$$

Defining  $\delta_i^{(l)} = \frac{\partial L(\dots)}{\partial a_i^{(l)}} f'(z_i^{(l)})$ , we have<sup>2</sup> ( $2 \leq l < L$ ):

$$\delta_1^{(L)}(o) = (a_1^{(L)}(o) - y(o)) f'(z_1^{(L)}(o)) \quad \delta_i^{(l)}(o) = \left( \sum_{j=1}^{s_{l+1}} \delta_j^{(l+1)}(o) w_{j,i}^{(l)} \right) f'(z_i^{(l)}(o)) \quad (4)$$

---

<sup>2</sup>Reintroducing object argument

# Backpropagation of derivatives

Or in matrix notations:

$$\begin{aligned}\delta^{(L)}(o) &= (\mathbf{a}^{(L)}(o) - \mathbf{y}(o))f'(\mathbf{z}^{(L)}(o)) \\ \delta^{(l)}(o) &= ((\mathbf{W}^{(l)})^T \delta^{(l+1)}(o))f'(\mathbf{z}^{(l)}(o)) \quad 2 \leq l < L,\end{aligned}$$

with  $\mathbf{W}^{(l)} \in I\!\!R^{s_{l+1} \times s_l}$  defined as  $(\mathbf{W}^{(l)})_{i,j} = w_{i,j}^{(l)}$ .

# Backpropagation of derivatives: summary

To compute all partial derivatives  $\frac{\partial L(g(\mathbf{a}(o); \mathcal{W}), y(o))}{\partial w_{i,j}^{(l)}}$  for a given object  $o$ :

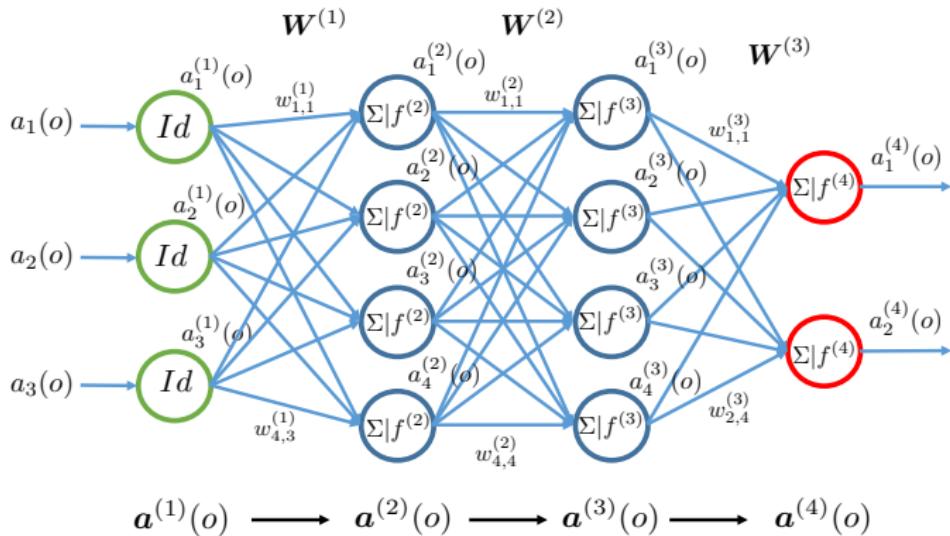
1. compute  $a_i^{(l)}(o)$  and  $z_i^{(l)}(o)$  for all neurons using (1) and (2)  
**(forward propagation)**
2. compute  $\delta_i^{(l)}(o)$  for all neurons using (4)  
**(backward propagation)**
3. Compute (using (3)):

$$\frac{\partial L(g(\mathbf{a}(o); \mathcal{W}), y(o))}{\partial w_{i,j}^{(l)}} = \delta_i^{(l+1)}(o) a_j^{(l)}(o)$$

NB: Backpropagation can be adapted easily to other (differentiable) loss functions and feedforward (i.e., without cycles) network structure

# Backpropagation of derivatives: illustration

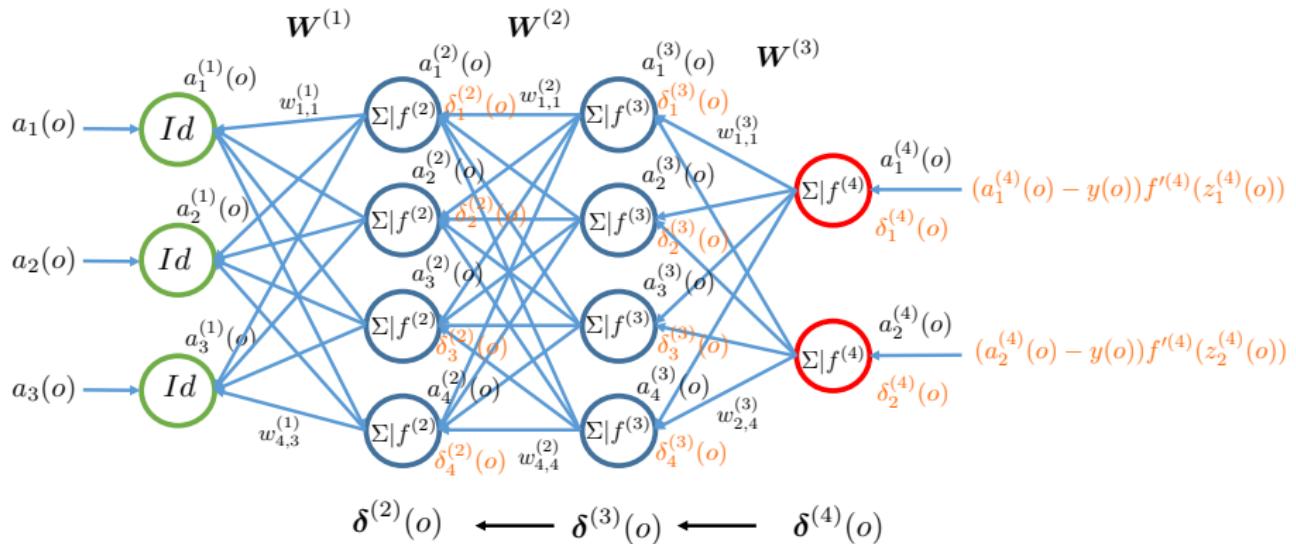
## Forward propagation



$$\mathbf{a}^{(l+1)}(o) = f^{(l)}(\mathbf{W}'^{(l)} \mathbf{a}'^{(l)}(o))$$

# Backpropagation of derivatives: illustration

## Backward propagation



$$\delta^{(l)}(o) = ((\mathbf{W}^{(l)})^T \delta^{(l+1)}(o)) f'(z^{(l)}(o))$$

# Online or batch gradient descent with backpropagation

1. Choose a network structure and a loss function  $L$ .
2. Initialize all network weights  $w_{i,j}^{(l)}$  appropriately.
3. Repeat until some stopping criterion is met:

**3.1** Using backpropagation, compute either (batch mode):

$$\Delta w_{i,j}^{(l)} = \frac{1}{N} \sum_{o \in LS} \frac{\partial L(g(\mathbf{a}(o); \mathcal{W}), y(o))}{\partial w_{i,j}^{(l)}},$$

or (online mode):

$$\Delta w_{i,j}^{(l)} = \frac{\partial L(g(\mathbf{a}(o); \mathcal{W}), y(o))}{\partial w_{i,j}^{(l)}}$$

for a single object  $o \in LS$  chosen at random or in a cyclic way.

**3.2** Update the weights according to:

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \Delta w_{i,j}^{(l)},$$

with  $\eta \in ]0, 1]$ , the learning rate.

# Between online and batch gradient descent

Mini-batch is commonly used

- ▶ Compute each gradient over a small subset of  $q$  objects
- ▶ Between stochastic ( $q = 1$ ) and batch ( $q = N$ ) gradient descent
- ▶ Sometimes can provide a better tradeoff in terms of optimality and speed.
- ▶ One gradient computation is called an iteration, one sweep over all training examples is called an epoch.
- ▶ It's often beneficial to keep original class proportion in mini-batches

Initial values of the weights:

- ▶ They have an influence on the final solution
- ▶ Not all to zero to break symmetry
- ▶ Typically: small random weights, so that the network first operates close to linearity and then its non-linearity increases when training proceeds.

# More on backpropagation and gradient descent

- ▶ Will find a local, not necessarily global, error minimum.
- ▶ Computational complexity of gradient computations is low (linear w.r.t. everything) but training can require thousands of iterations.
- ▶ Any general technique to make gradient descent converge faster or better can be applied to MLP training (second-order techniques, conjugate gradient, learning rate adaptation, etc.).
- ▶ Common improvement of SGD: Momentum update (with  $\mu \in [0, 1]$ )

$$\Delta_{i,j}^{(l)} \leftarrow \mu\Delta_{i,j}^{(l)} - \eta\Delta w_{i,j}^{(l)}; \quad w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} + \Delta_{i,j}^{(l)}$$

# Multi-class classification

- ▶ **One-hot encoding:**  $k$  classes are encoded through  $k$  numerical outputs, with  $y_i(o) = 1$  if  $o$  belongs to the  $i$ th class, 0 otherwise
- ▶ Loss function could be average square error over all outputs
- ▶ A better solution:
  - ▶ Transform neural nets outputs using **softmax**:

$$p_i(o) = \frac{\exp(a_i^{(L)}(o))}{\sum_k \exp(a_k^{(L)}(o))}$$

(such that  $p_i(o) \in [0; 1]$  and  $\sum_i p_i(o) = 1$ ).

- ▶ Use **cross-entropy** as a loss function:

$$L(\mathbf{g}(\mathbf{a}(o); \mathcal{W}), \mathbf{y}(o)) = - \sum_{i=1}^k y_i(o) \log p_i(o)$$

# Activation functions

As for STU, common activation functions are sigmoid and hyperbolic tangent.

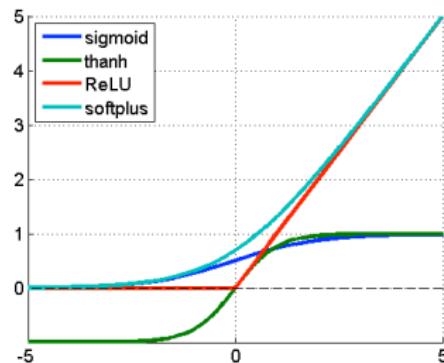
A recent alternative is **ReLU** (rectifier linear unit):

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

(or its smooth approximation, softplus:  $f(x) = \ln(1 + e^x)$ )

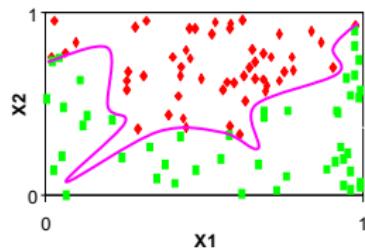
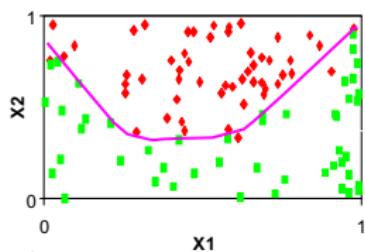
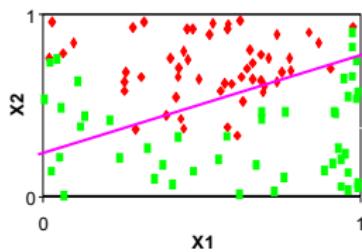
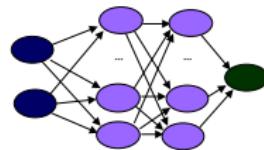
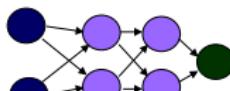
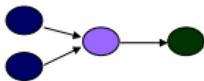
Several advantages:

- ▶ Sparse activation (some neurons are inactive)
- ▶ Efficient gradient propagation (avoid vanishing or exploding gradient)
- ▶ Efficient computation (comparison, addition, and multiplication only)



# Overfitting

Too complex networks will clearly overfit

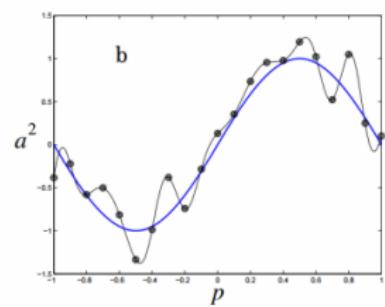
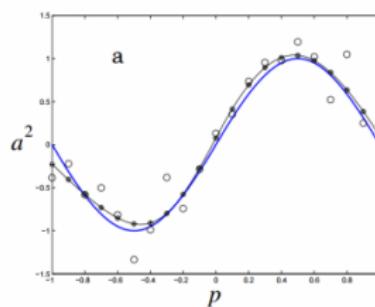
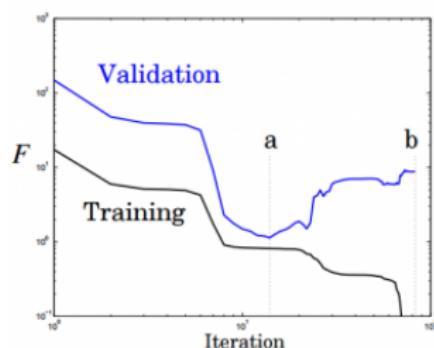


One could select optimal network size using cross-validation but better results are often obtained by carefully training complex networks instead.

# Avoiding overfitting with neural networks

Early stopping:

- ▶ Stop gradient descent iterations before convergence, by controlling the error on an independent validation set
- ▶ If initial weights are small, the more iterations, the more non-linear becomes the model.



Source: <http://www.turingfinance.com/misconceptions-about-neural-networks/>

# Avoiding overfitting with neural networks

Weight decay:

- Add an extra-term to the loss function that penalizes too large weights:

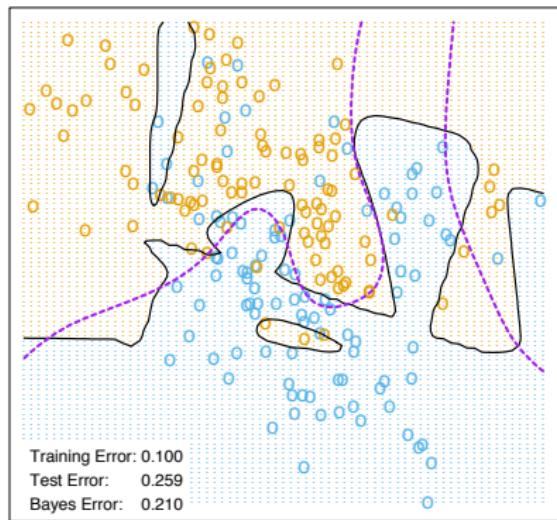
$$\mathcal{W}^* = \arg \min_{\mathcal{W}} \frac{1}{N} \sum_{o \in LS} L(\mathbf{g}(\mathbf{a}(o); \mathcal{W}), \mathbf{y}(o)) + \lambda \frac{1}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (w_{i,j}^{(l)})^2,$$

- $\lambda$  controls complexity (since larger weights mean more non-linearity) and can be tuned on a validation set
- Modified weight update:  $w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta (\Delta w_{i,j}^{(l)} + \lambda w_{i,j}^{(l)})$
- Alternative: L1 penalization:  $(w_{i,j}^{(l)})^2 \Rightarrow |w_{i,j}^{(l)}|$ . Makes some weights exactly equal to zero (a form of edge pruning).

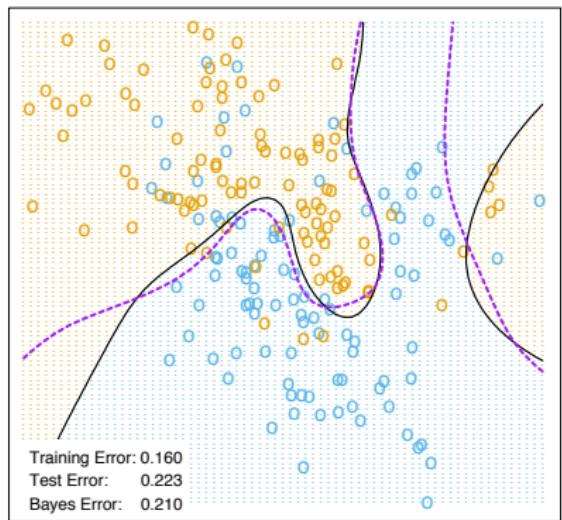
# Avoiding overfitting with neural networks

Weight decay:

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



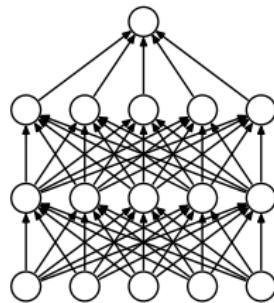
Source: Figure 10.4, Hastie et al., 2009

# Avoiding overfitting with neural networks

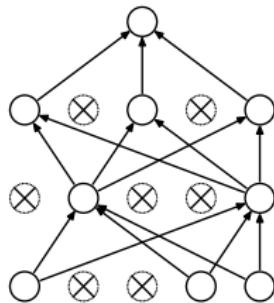
Dropout:

(Srivastava et al., JMLR, 2011)

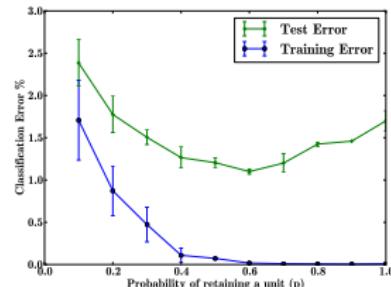
- ▶ Randomly drop neurons from each layer with probability  $\Phi$  and train only the remaining ones
- ▶ Make the learned weights of a node more insensitive to the weights of the other nodes.
- ▶ This forces the network to learn several independent representations of the patterns and thus decreases overfitting.



(a) Standard Neural Net



(b) After applying dropout.



# Avoiding overfitting with neural networks

Unsupervised pretraining:

- ▶ Main idea:
  - ▶ Train each hidden layer in turn in an unsupervised way, so that it allows to reproduce the input of the previous layer.
  - ▶ Introduce the output layer and then fine-tune the whole system using backpropagation
- ▶ Allowed in 2006 to train deeper neural networks than before and to obtain excellent performance on several tasks (computer vision, speech recognition).
- ▶ Unsupervised pretraining is especially useful when the number of labeled examples is small.

## Introduction

### Single neuron models

Hard threshold unit (LTU) and the perceptron

Soft threshold unit and gradient descent

Theoretical properties

### Multilayer perceptron

Definition and expressiveness

Learning algorithms

Overfitting and regularization

### Other neural network models

Radial basis function networks

Convolutional neural networks

Recurrent neural networks

## Conclusion

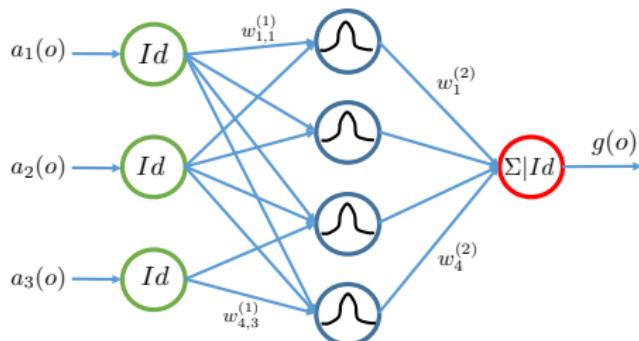
# Other neural network models

Beyond MLP, many other neural network structures have been proposed in the literature, among which:

- ▶ Radial basis function networks
- ▶ Convolutional networks
- ▶ Recurrent neural networks
- ▶ Restricted Boltzman Machines (RBM)
- ▶ Kohonen maps (see lecture on unsupervised learning)
- ▶ Auto-encoders (see lecture on unsupervised learning)

# Radial basis functions networks

A neural network with a single hidden layer with **radial basis functions (RBF)** as activation functions



Output is of the following form:

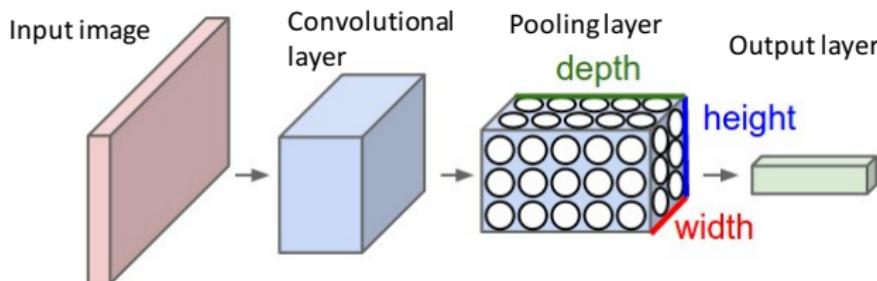
$$g(o) = w_0^{(2)} + \sum_{k=1}^{s_2} w_k^{(2)} \exp \left( -\frac{\|\mathbf{a}^{(1)}(o) - \mathbf{w}_i^{(1)}\|^2}{2\sigma^2} \right)$$

# Radial basis function networks

- ▶ Training:
  - ▶ Input layer: vectors  $w_i^{(1)}$  are trained by unsupervised clustering techniques and  $\sigma$  commonly set to  $d/\sqrt{2s_2}$ , with  $d$  the maximal euclidean distance between two weight vectors.
  - ▶ Output layer: can be trained by any linear method (least-square, perceptron...).
  - ▶ Size  $s_2$  of hidden layer is determined by cross-validation.
- ▶ Much faster to train than MLP
- ▶ Similar to the k-NN method

# Convolutional neural networks (ConvNets)

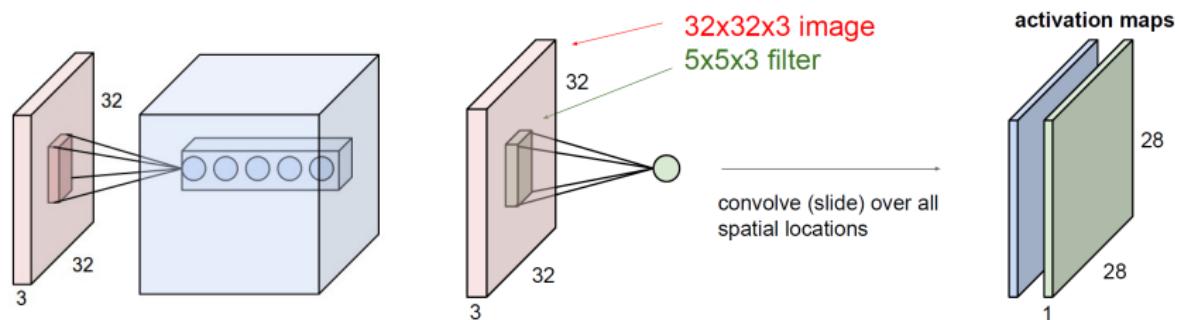
- ▶ A (feedforward) neural network structure initially designed for images
  - ▶ But can be extended to any input data composed of values that can be arranged in a 1D, 2D, 3D or more structure. E.g., sequences, texts, videos, etc.
- ▶ Built using three kinds of hidden layers: convolutional, pooling, and fully-connected
- ▶ Neurons in each layer can be arranged into a 3D structure.



Source: <http://cs231n.github.io/convolutional-networks/>

# Convolutional layer

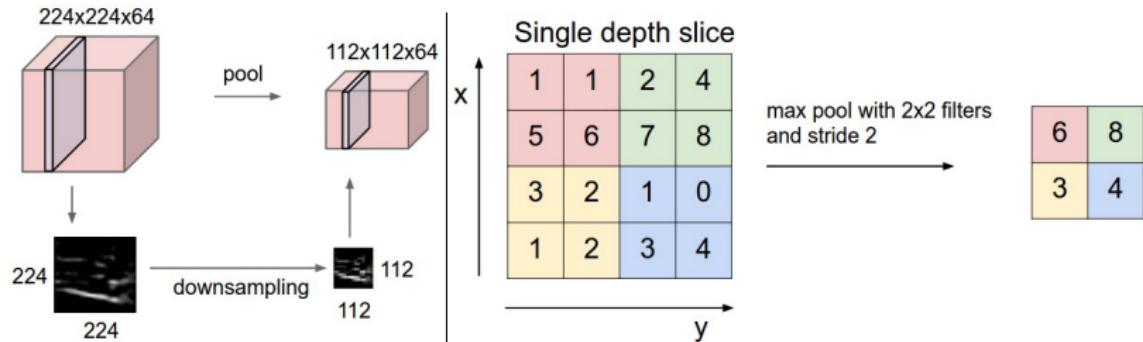
- ▶ Each neuron is connected only to a **local** region (along width and height, not depth) in the previous layer (the receptive field of the neuron)
- ▶ The receptive field of neurons at the same depth are滑 by some fixed **stride** along width and height.
- ▶ All neurons at the same depth **share** the same set of weights (and thus detect the same feature at different locations)



Source: <http://cs231n.github.io/convolutional-networks/>

# Pooling (or subsampling) layer

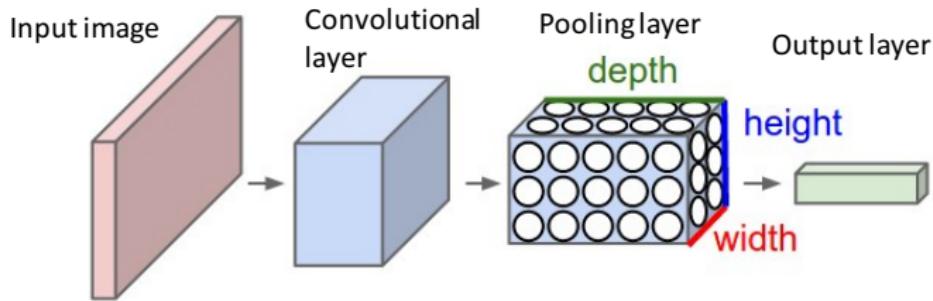
- ▶ Each neuron is connected only to a local region (along width and height) at the same depth as its own depth in the previous layer.
- ▶ The receptive field of neurons at the same depth are滑动 by some fixed **stride** along width and height (stride > 1 means subsampling).
- ▶ Output of the neuron is an aggregation of the values in the local region. E.g., the maximum or the average in that region.



Source: <http://cs231n.github.io/convolutional-networks/>

# Fully connected layer

- ▶ Width and height are equal to 1
- ▶ Each neuron is connected to all neurons of the previous layer
- ▶ At least, the output layer is a fully connected layer, with one neuron per output



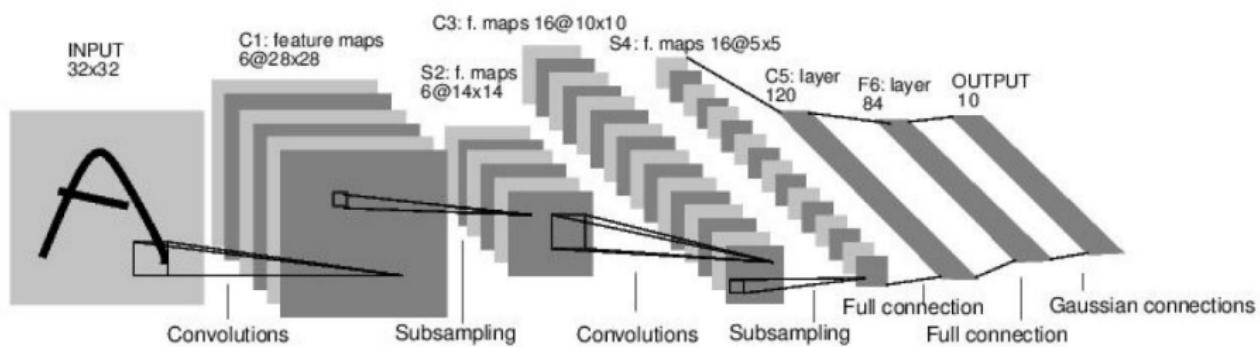
Source: <http://cs231n.github.io/convolutional-networks/>

# Why convolutional networks?

- ▶ It's possible to compute the same outputs in a fully connected MLP, but:
  - ▶ The network would be much harder to train.
  - ▶ Convolutional networks have much less parameters due to weight sharing.
  - ▶ They are less prone to overfitting.
- ▶ It makes sense to detect features (by convolution) and to combine them (by pooling). Max pooling allows to detect shift-invariant features.
- ▶ It's possible to draw analogy with the way our brain works.

## LeNet-5

(Lecun et al., 1998)



$5 \times 5$  convolutional layers at stride 1,  $2 \times 2$  max pooling layers at stride 2.

First successful application of convolutional networks. Used by banks in the US to read cheques.

# ImageNet - A large scale visual recognition challenge

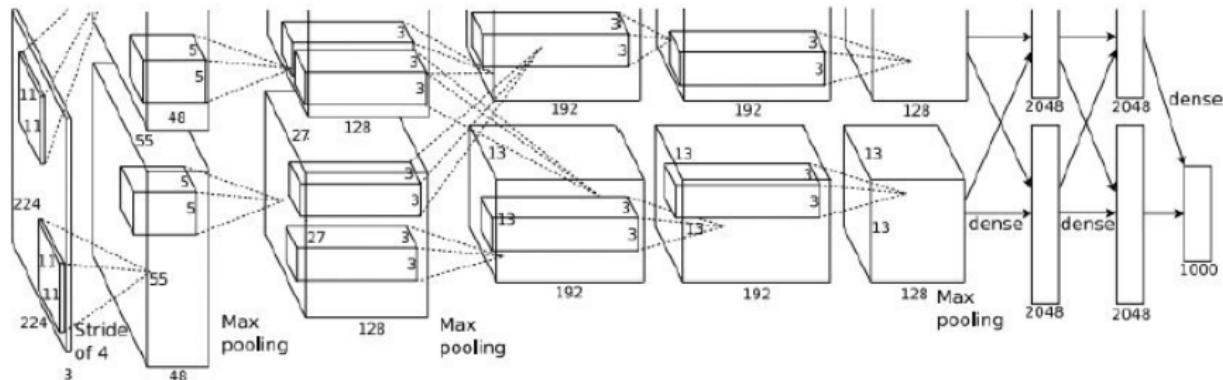


<http://image-net.org>

- ▶ 1.2 million images, 1000 image categories in the training set
- ▶ Task: identify the object in the image (ie., a multi-class classification problem with 1000 classes)
- ▶ Evaluation: top-5 error ("is one of the best 5 class predictions correct?")
- ▶ Human error: 5.1% (<http://cs.stanford.edu/people/karpathy/ilsvrc/>)

# Best performer in 2012: AlexNet

(Krizhevsky et al., 2012)



Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

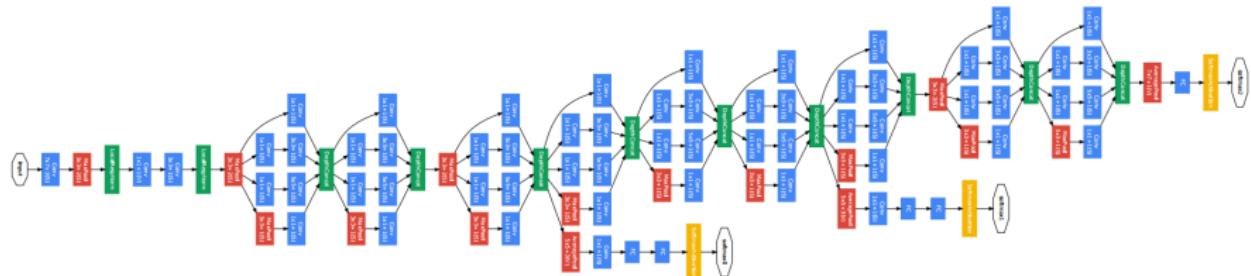
## Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

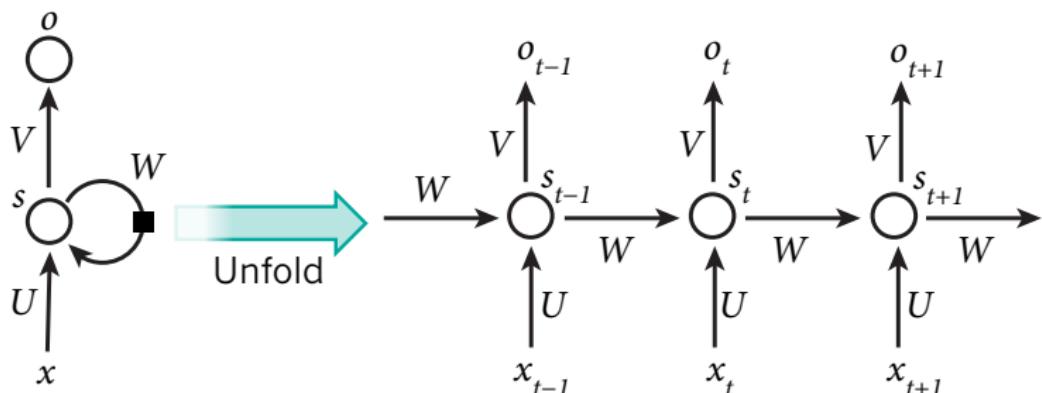
# Best performer in 2014: GoogLeNet

(Szegedy et al., 2014)

- ▶ 6.7% top-5 error. Very close to human performance.
- ▶ Very deep: 100 layers (22 with tuned parameters), more than 4M parameters
- ▶ Several neat tricks (heterogeneous set of convolutions, inception modules, softmax outputs in the middle of the network, etc.)



# Recurrent neural networks



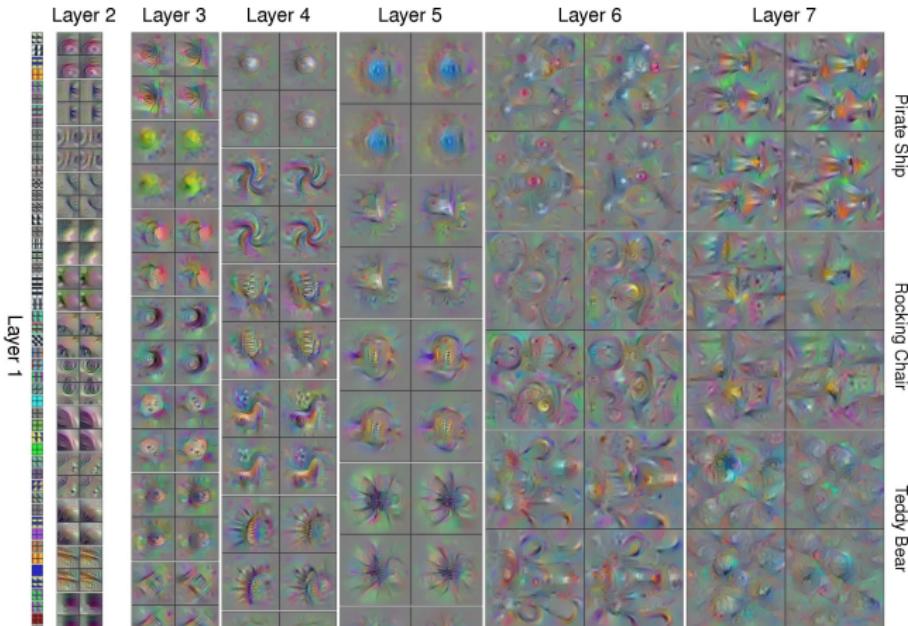
- ▶ Neural networks with feedback connections
- ▶ When unfolded, can be trained using back-propagation
- ▶ Allows to model non-linear dynamical phenomenon
- ▶ Best approach for language modeling (e.g., word prediction)
  - E.g., <http://twitter.com/DeepDrumpf>

# Conclusions

- ▶ Neural networks are universal (parametric) approximators
- ▶ Deep (convolutional) neural networks provide state-of-the-art performance in several application domains (speech recognition, computer vision, texts...)
- ▶ Training deep networks is very expensive and requires a lot of data...
- ▶ ...but dedicated GPUs start appearing that reduce strongly computing times
- ▶ Often presented as automatic feature extraction techniques...
- ▶ ...but a lot of engineering (or art) is required to tune their hyper-parameters (structure, regularization, activation and loss functions, weight initialization, etc.).

# Conclusions

- ▶ Essentially black-box models although some model inspection is possible



<http://yosinski.com/deepvis>

# References and softwares

## References:

- ▶ Hastie et al.: Chapter 11 (11.3-11.7)
- ▶ Goodfellow, Bengio, and Courville, Deep Learning, MIT Press, 2016  
<http://www.deeplearningbook.org>
- ▶ Many tutorial on the web and also videos on Youtube: Andrew Ng, Hugo Larochelle...

## Main toolboxes:

- ▶ Tensorflow (Google), Python, <https://www.tensorflow.org>
- ▶ Theano (U. Montreal), Python,  
<http://deeplearning.net/software/theano/index.html>
- ▶ Caffe (U. Berkeley), Python), <http://caffe.berkeleyvision.org/>
- ▶ Torch (Facebook, Twitter), Lua, <http://torch.ch>