

INF8225

Leçon 3
Christopher Pal
École Polytechnique de Montréal

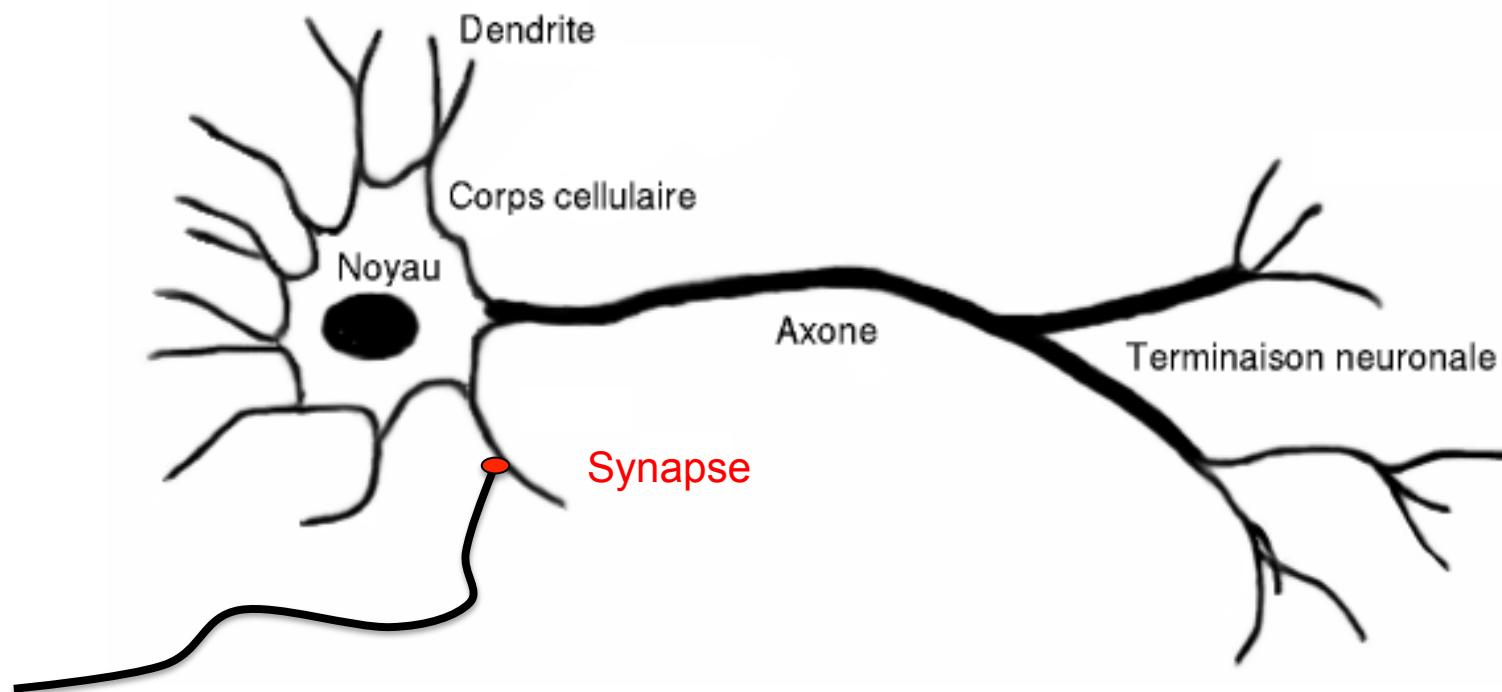
Plan du cours

- Réseaux de neurones
- Deep Learning
- Apprentissage avec rétropropagation
- Équations matricielles

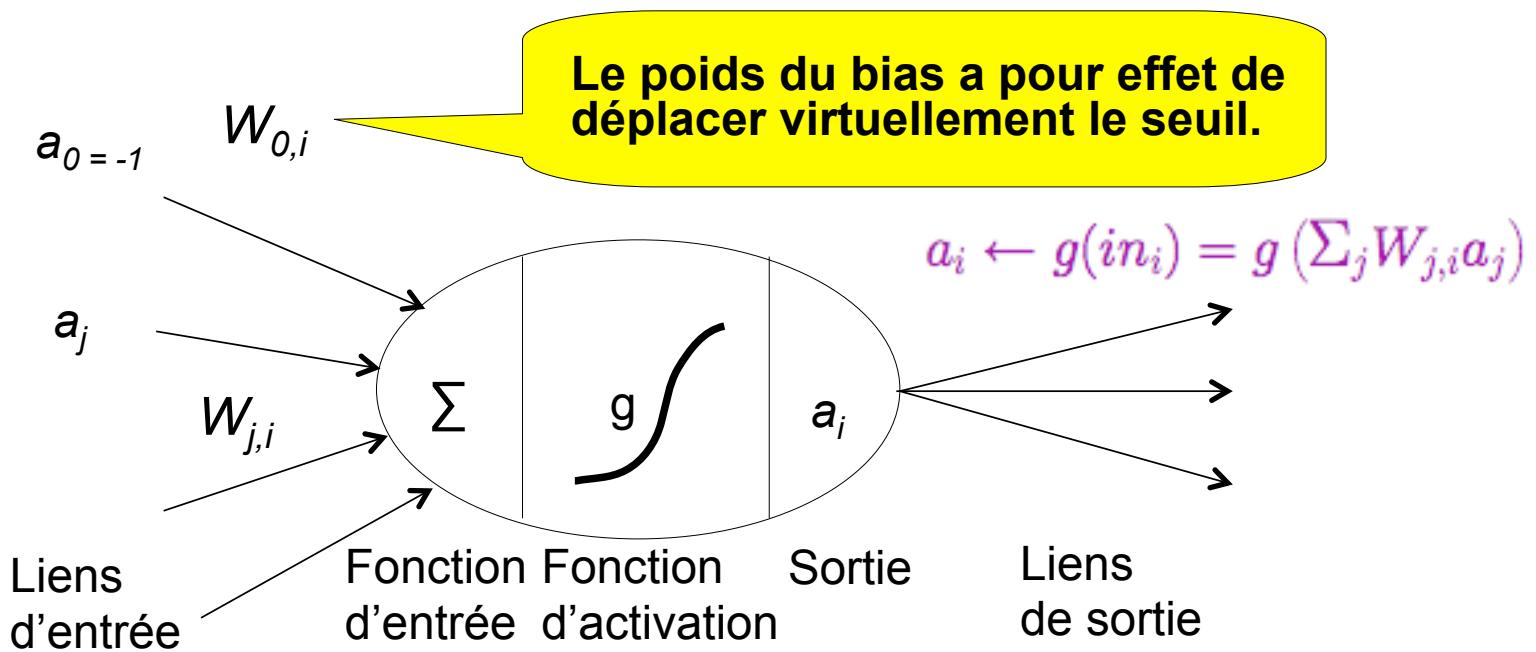
Réseaux de neurones

Réseaux de neurones

- 1ms-10ms temps pour des cycles
- 10^{11} neurones de plus de 20 types, 10^{14} synapses

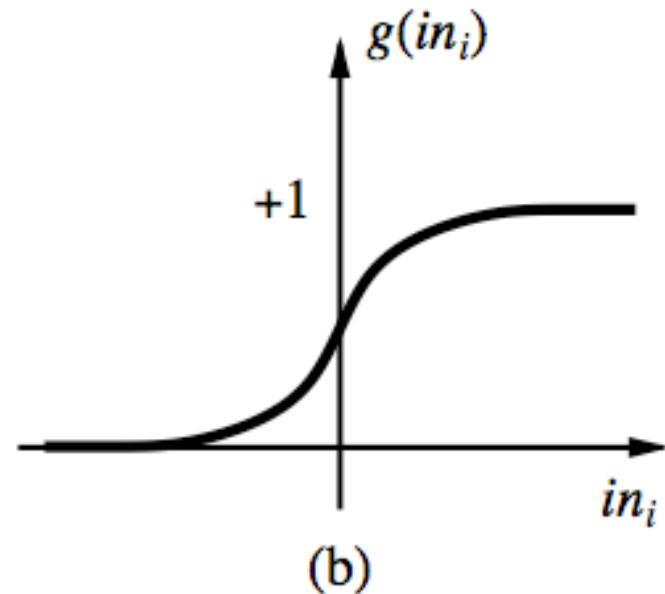
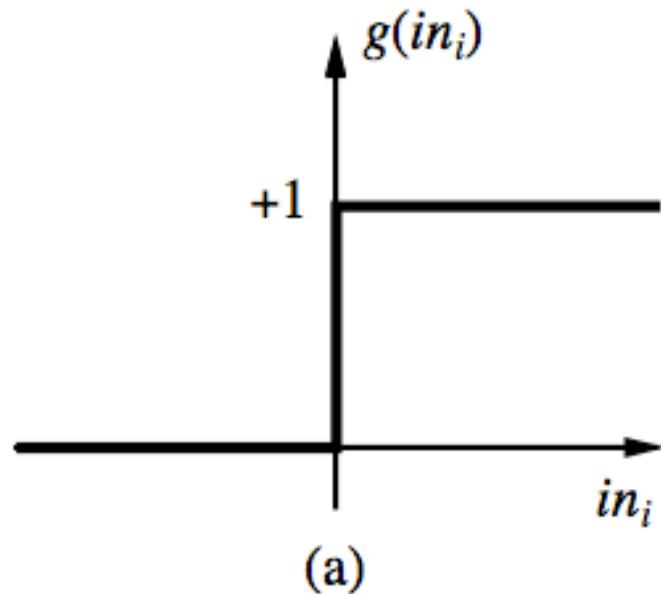


Modèle mathématique simple d'un neurone



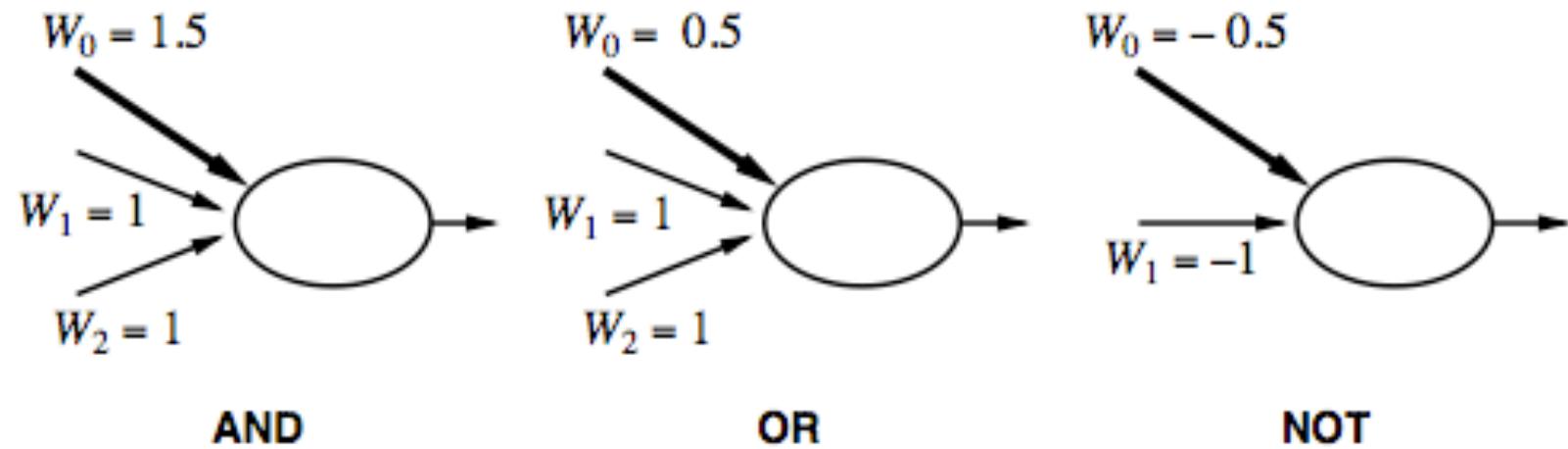
- La fonction d'activation g est conçue pour répondre à des désiderata, on veut que l'unité soit :
 1. « active » (proche de +1) quand les « bonnes » entrées sont données
 2. « inactive » (proche de 0) quand elles sont « mauvaises »
 3. L'activation doit être non linéaire, sinon nous avons une simple fonction linéaire

La fonction d'activation



- a) La fonction d'activation à seuil
- b) La fonction sigmoïde (fonction logistique)

Les unités avec fonction d'activation peuvent représenter des portes logiques

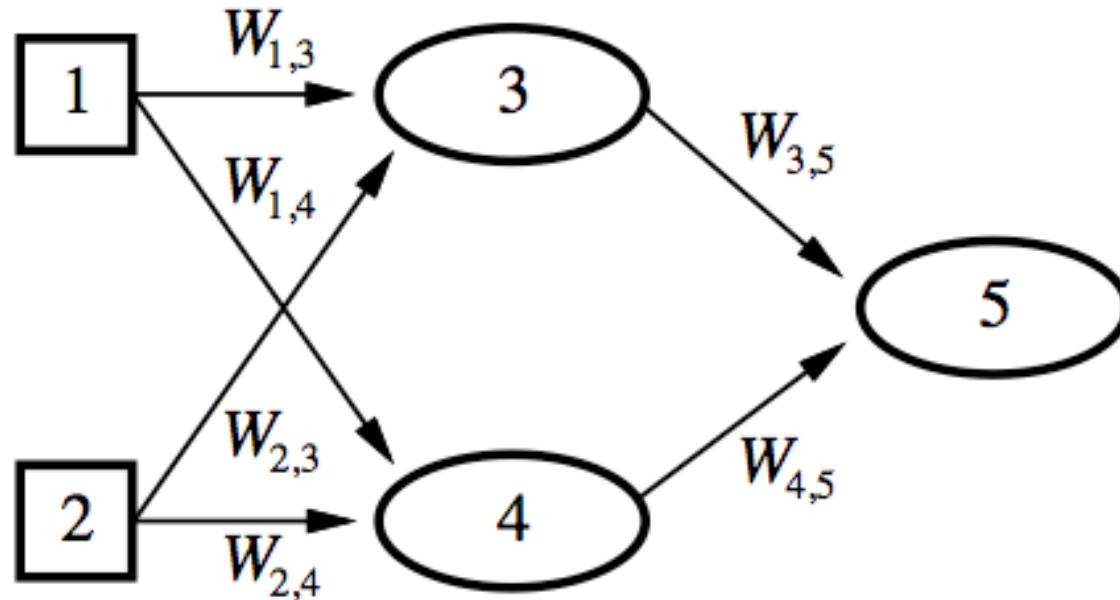


- Une motivation d'origine de la conception d'un modèle d'un neurone par McCulloch et Pitts, 1943.

Réseaux de neurones

- Plusieurs types de réseaux de neurones
 - réseaux « feed-forward » (perceptron multi-couches)
 - réseaux récurrents
- Principe d'apprentissage:
 - On fournit une entrée et la sortie désirée
 - On ajuste les poids en fonction de l'erreur
 - On fait la même chose avec les autres exemples
 - On répète jusqu'à ce qu'à un état où les poids varient très peu

Réseaux feed-forward

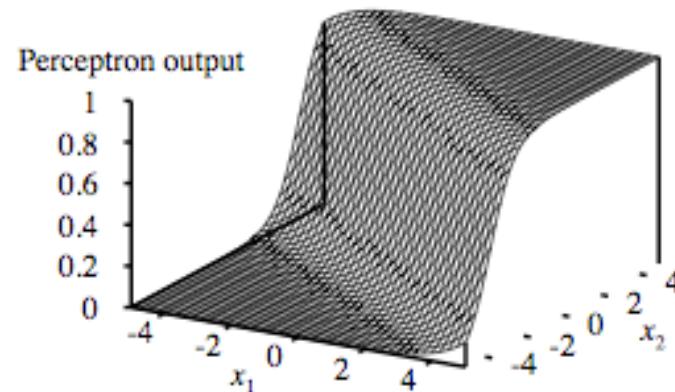
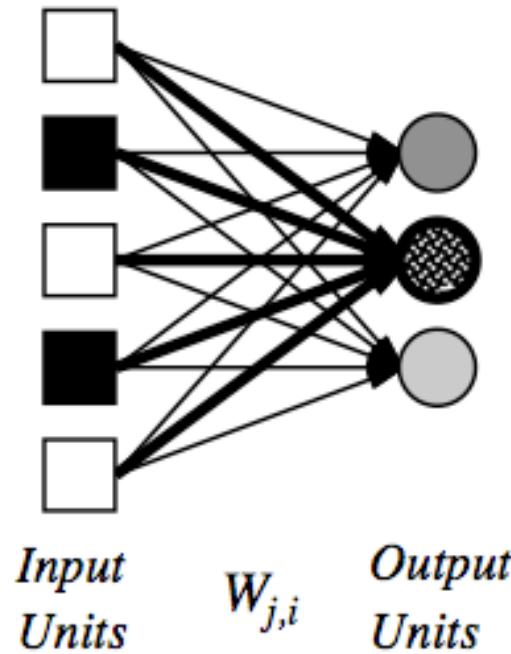


- Effectivement, c'est un paramétrage d'une famille de fonctions non linéaires

$$\begin{aligned}a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))\end{aligned}$$

- Apprentissage = l'ajustement des poids

Un perceptron monocouche

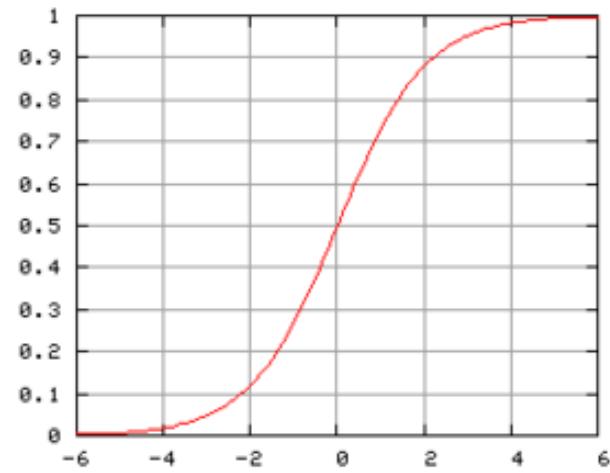


- Les ajustements des poids changent la position, l'orientation et la pente de la falaise

La fonction logistique

$$\text{Logistique}(z) = \frac{1}{1 + \exp(-z)}$$

- En utilisant un produit scalaire



$$h_{\mathbf{w}}(\mathbf{x}) = \text{Logistique}(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x})}$$

- Un avantage de cette fonction est que la dérivée satisfait $g'(z) = g(z)(1-g(z))$

Apprentissage perceptron classique

- Pour un seul exemple d'apprentissage avec une entrée x et une sortie vraie y , on écrit l'erreur quadratique

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2} (y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

- En utilisant la descente de gradient

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j} (y - g(\sum_{j=0}^n W_j x_j)) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

- Nous avons une règle simple pour une mise à jour

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

Apprentissage perceptron

répéter:

pour chaque exemple faire

pour chaque noeud j de la couche d'entrée du réseau faire

$a_j \leftarrow$ item j dans le vecteur d'entrée de l'exemple

$in \leftarrow \sum_j W_j \times a_j$

$a_{out} \leftarrow g(in)$

// soit y la sortie désirée pour cet exemple

$\Delta \leftarrow g'(in) \times (y - a_{out})$

$W_j \leftarrow W_j + \alpha \times a_j \times \Delta_i$

jusqu'à ce que le critère d'arrêt soit satisfait

Perceptron - exemple

répéter:

pour chaque exemple faire

pour chaque noeud j de la couche d'entrée du réseau faire

$a_j \leftarrow$ item j dans le vecteur d'entrée de l'exemple

$in \leftarrow \sum_j W_j \times a_j$

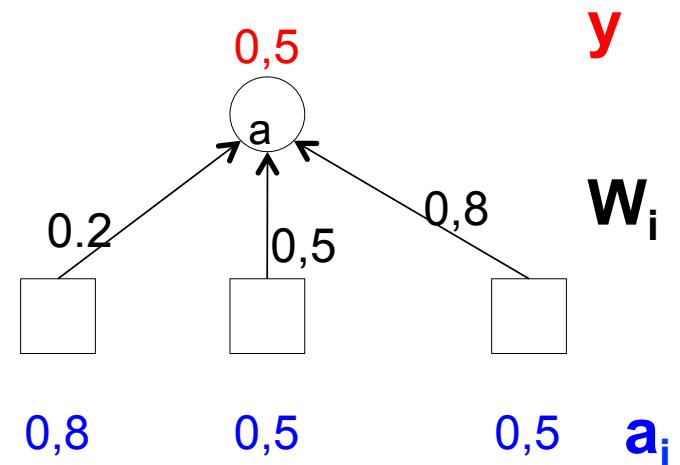
$a_{out} \leftarrow g(in)$

// soit y la sortie désirée pour cet exemple

$\Delta \leftarrow g'(in) \times (y - a_{out})$

$W_j \leftarrow W_j + \alpha \times a_j \times \Delta_i$

jusqu'à ce que le critère d'arrêt soit satisfait



$$\alpha = 0,9$$

- $in = .2 * .8 + .5 * .5 + .8 * .5 = .81$

Perceptron - exemple

répéter:

pour chaque exemple faire

pour chaque noeud j de la couche d'entrée du réseau faire

$a_j \leftarrow$ item j dans le vecteur d'entrée de l'exemple

$in \leftarrow \sum_j W_j \times a_j$

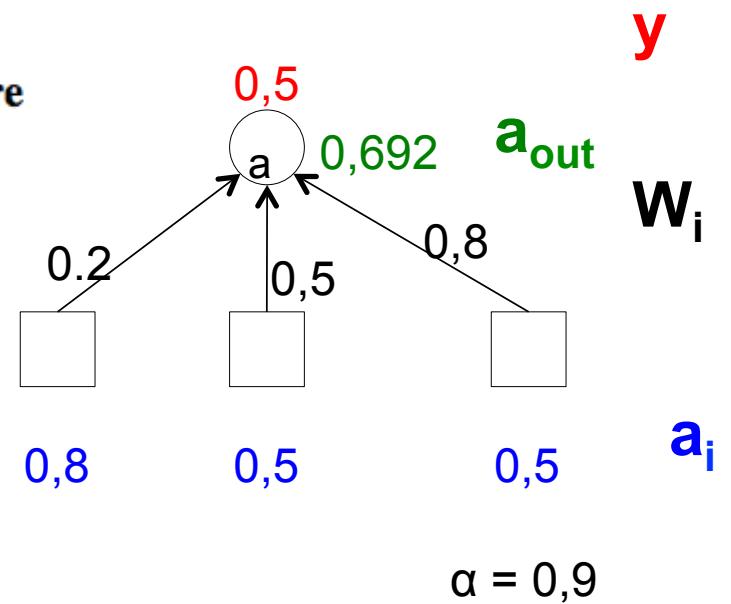
$a_{out} \leftarrow g(in)$

// soit y la sortie désirée pour cet exemple

$\Delta \leftarrow g'(in) \times (y - a_{out})$

$W_j \leftarrow W_j + \alpha \times a_j \times \Delta_i$

jusqu'à ce que le critère d'arrêt soit satisfait



$$a_{out} = g(.81) = \frac{1}{1 + \exp(-.81)} = .692$$

Perceptron - exemple

répéter:

pour chaque exemple faire

pour chaque noeud j de la couche d'entrée du réseau faire

$a_j \leftarrow$ item j dans le vecteur d'entrée de l'exemple

$$in \leftarrow \sum_j W_j \times a_j$$

$$a_{out} \leftarrow g(in)$$

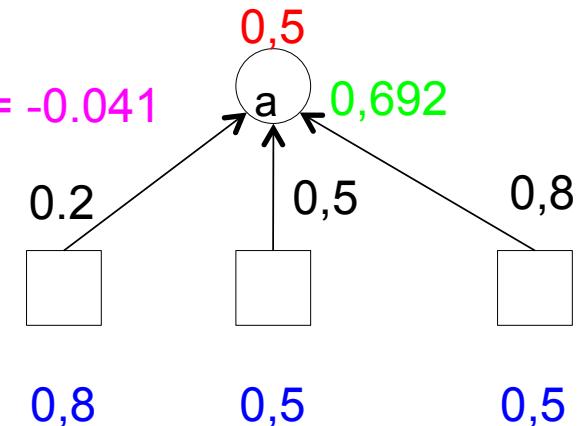
// soit y la sortie désirée pour cet exemple

$$\Delta \leftarrow g'(in) \times (y - a_{out})$$

$$W_j \leftarrow W_j + \alpha \times a_j \times \Delta_i$$

jusqu'à ce que le critère d'arrêt soit satisfait

$$\begin{aligned}
 \Delta &= g'(in) \times (y - a_{out}) & \alpha = 0,9 \\
 &= g(z)(1 - g(z)) \times (y - g(z)) \\
 &= .692(1 - .692) \times (.5 - .692) = -0.041
 \end{aligned}$$



Perceptron - exemple

répéter:

pour chaque exemple faire

pour chaque noeud j de la couche d'entrée du réseau faire

$a_j \leftarrow$ item j dans le vecteur d'entrée de l'exemple

$$in \leftarrow \sum_j W_j \times a_j$$

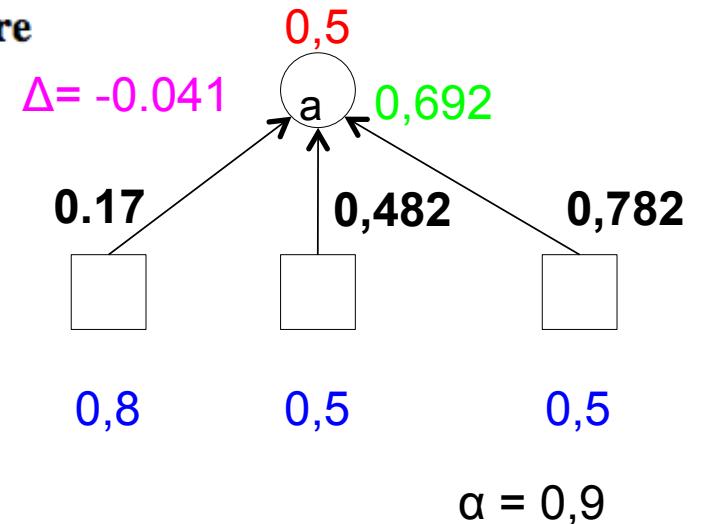
$$a_{out} \leftarrow g(in)$$

// soit y la sortie désirée pour cet exemple

$$\Delta \leftarrow g'(in) \times (y - a_{out})$$

$$W_j \leftarrow W_j + \alpha \times a_j \times \Delta_i$$

jusqu'à ce que le critère d'arrêt soit satisfait



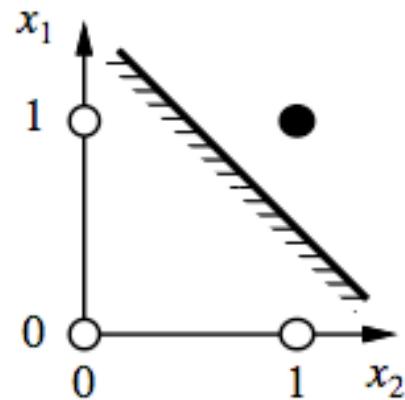
$$W_1 = .2 + .9 \times .8 \times -.041 = .17$$

$$W_2 = .5 + .9 \times .5 \times -.041 = .482$$

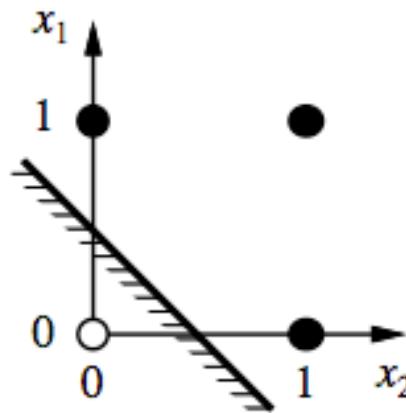
$$W_3 = .8 + .9 \times .5 \times -.041 = .782$$

Problèmes avec un séparateur linéaire

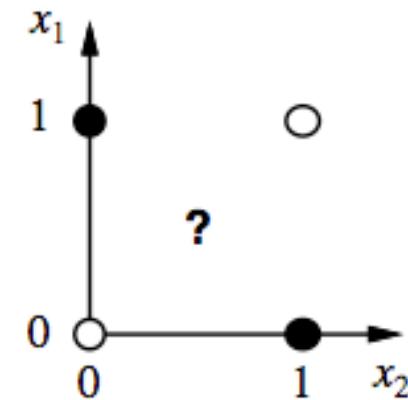
$$\sum_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$



(a) x_1 and x_2



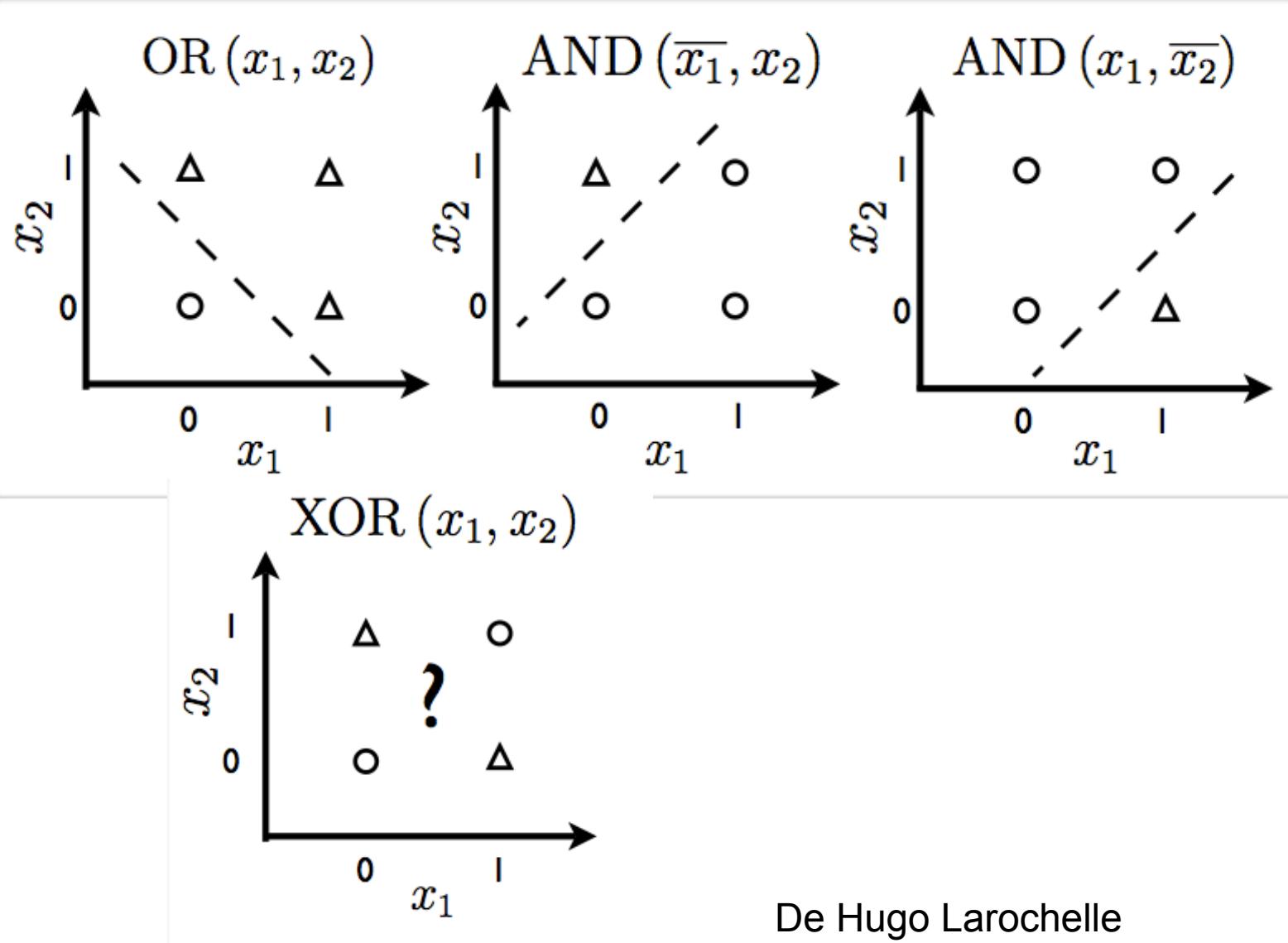
(b) x_1 or x_2



(c) x_1 xor x_2

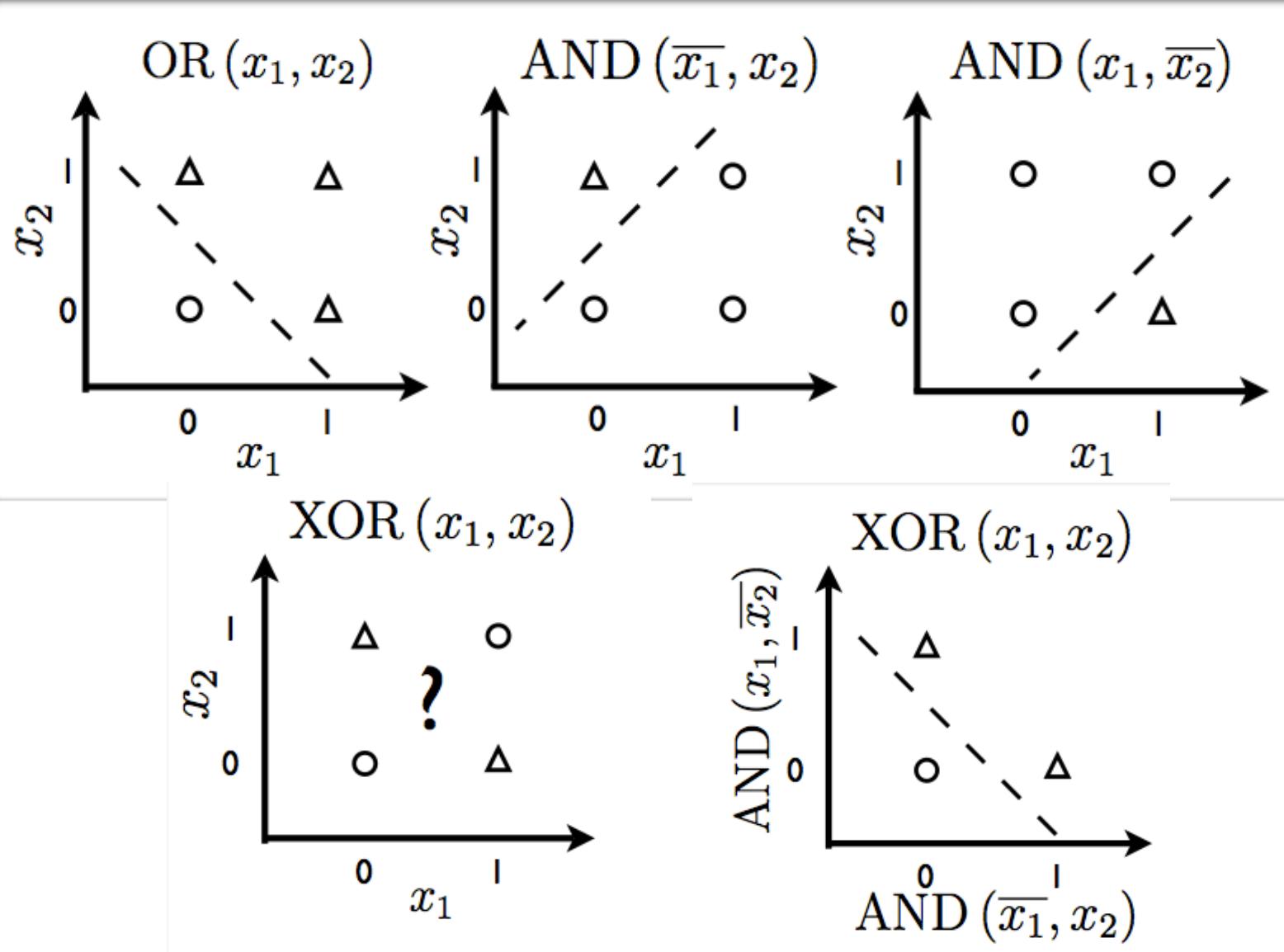
- Il peut représenter AND, OR et NOT mais pas XOR
- ... sauf si vous pouvez trouver une meilleure représentation. Q: La définition d'XOR est ?

Exercice: utilisez la definition d'XOR pour transformer les donnees

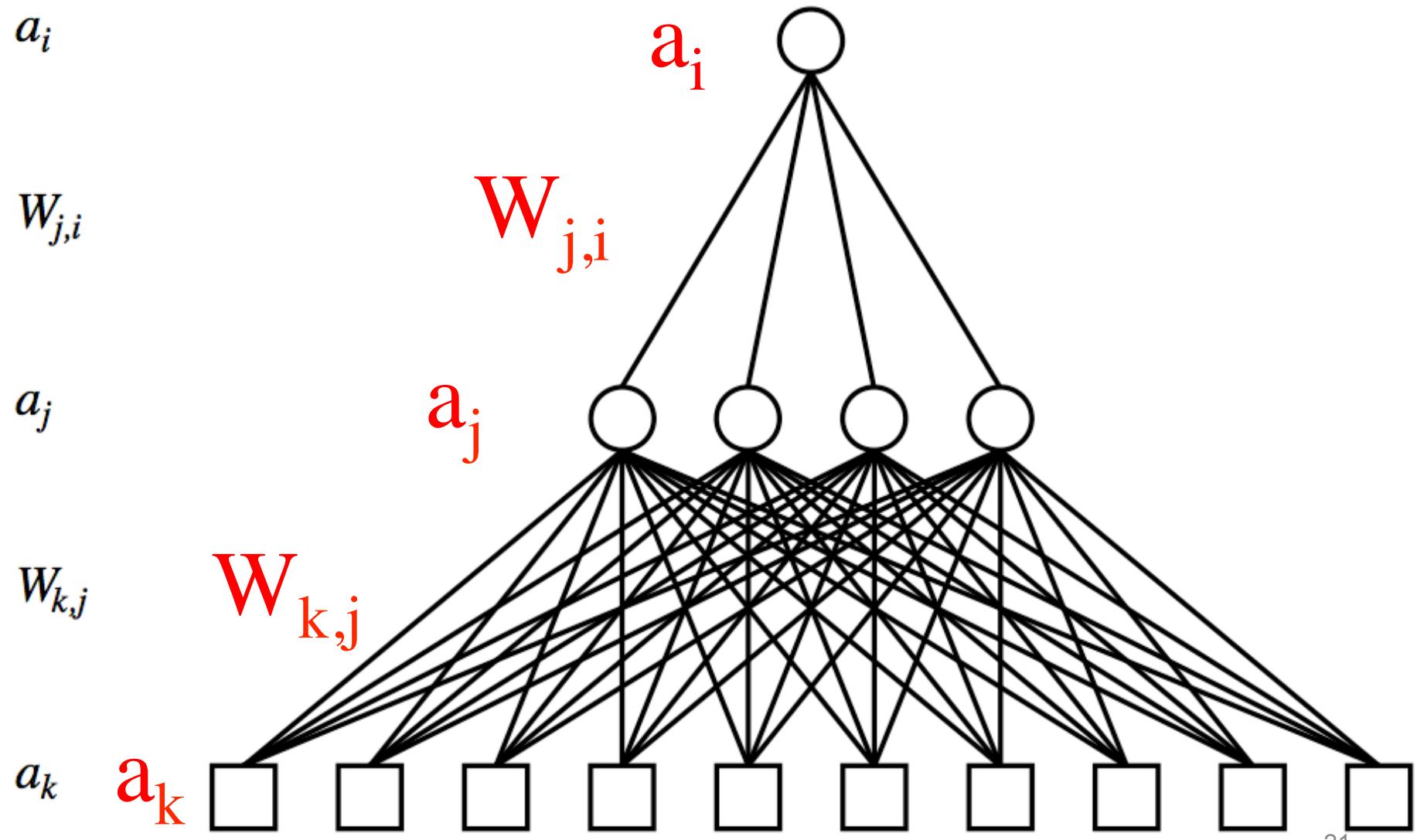


De Hugo Larochelle

Exercice: utilisez la definition d'XOR pour transformer les donnees

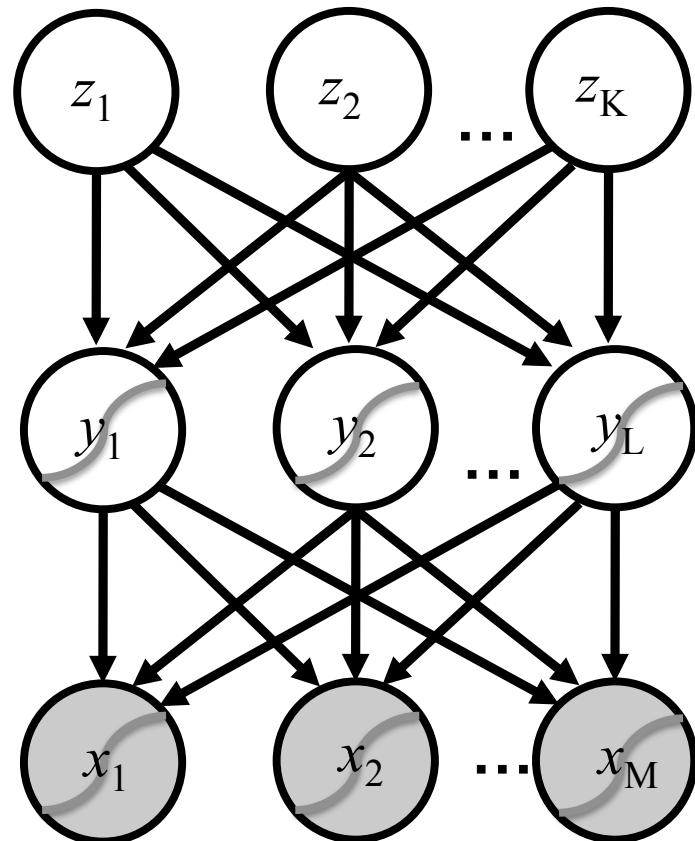


Réseau de neurones multicouche

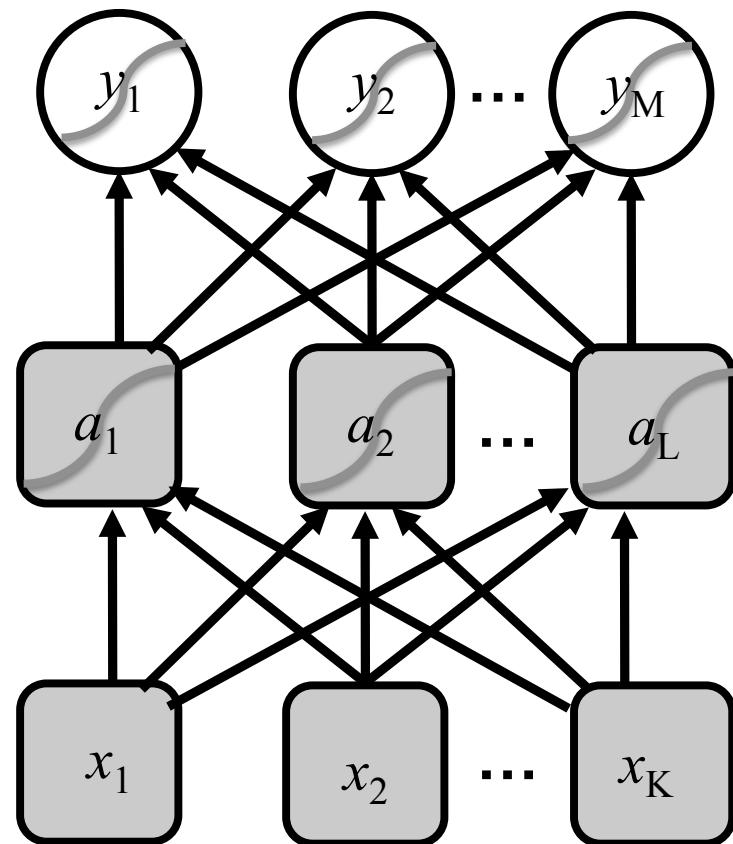


Important : BNs vs NNs

Réseau de Bayes



Réseau de neurones



Variables stochastique

● Observé

○ non-observé

Variables déterministe

■ Observé

Rétropropagation

Couche de sortie: mise à jour de même façon que d'une réseau avec un seule couche

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times \textcolor{purple}{a_j} \times \Delta_i \quad \text{où} \quad \Delta_i = Err_i \times g'(in_i)$$

Comparez avec notre mise à jour pour un seule couche $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times \textcolor{red}{x_j}$

Couche caché: rétropropager l'erreur de la couche de sortie

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i .$$

Règle de mise à jour pour les poids de la couche caché:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j .$$

Dérivation des équations de rétropropagation

En commençant avec l'erreur quadratique

$$E = \frac{1}{2} \sum_i (y_i - a_i)^2 ,$$

Où la somme est effectuée sur les nœuds de la couche de sortie

$$\begin{aligned}\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i) \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \frac{\partial g(in_i)}{\partial W_{j,i}} \\ &= -(y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i) g'(in_i) \frac{\partial}{\partial W_{j,i}} \left(\sum_j W_{j,i} a_j \right) \\ &= -(y_i - a_i) g'(in_i) a_j = -a_j \Delta_i\end{aligned}$$

avec

$$\Delta_i = Err_i \times g'(in_i)$$

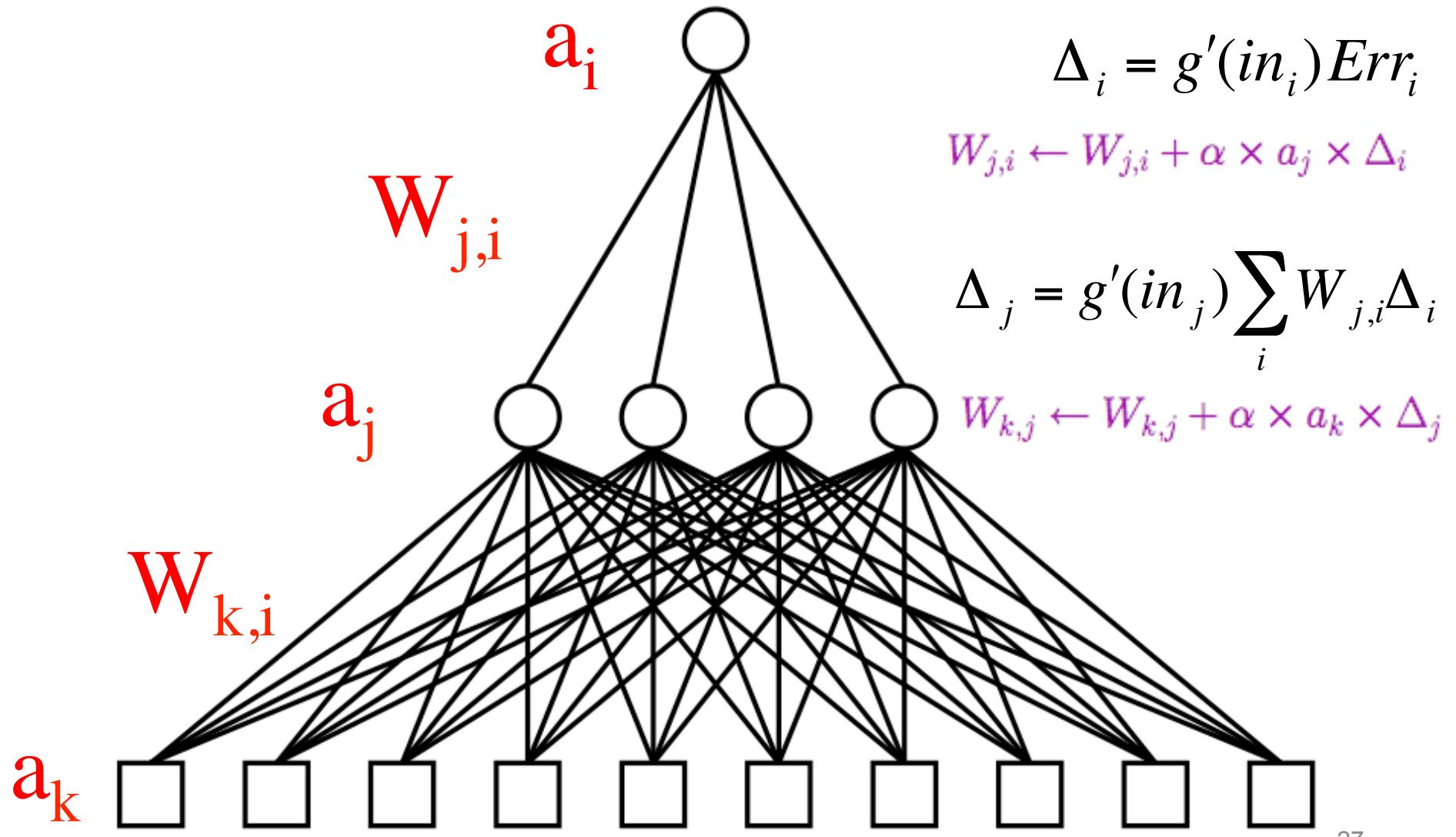
L'algorithme de rétropropagation

- Le processus de rétropropagation peut se résumer comme suit :
- Calculer les valeurs Δ pour les unités de sortie en utilisant l'erreur observée
- En commençant par la couche de sortie, répéter les étapes suivantes pour chaque couche du réseau, jusqu'à ce que la dernière couche cachée ait été atteinte
 - Rétropropager les valeurs Δ à la couche précédente
 - Mettre à jour les poids entre les deux couches

Dérivation de rétropropagation

$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= - \sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = - \sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} \\ &= - \sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} = - \sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left(\sum_j W_{j,i} a_j \right) \\ &= - \sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = - \sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} \left(\sum_k W_{k,j} a_k \right) \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) a_k = - a_k \Delta_j\end{aligned}$$

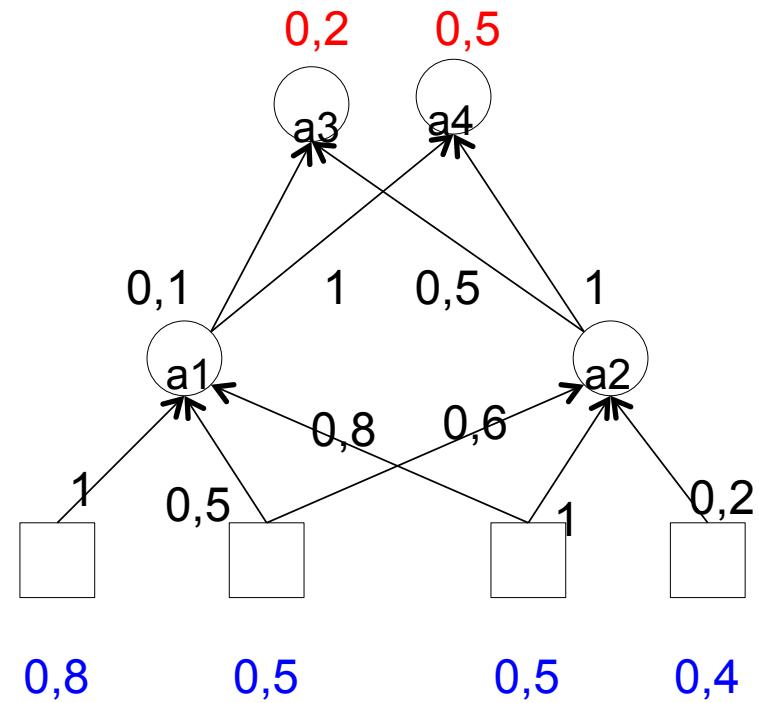
Réseau de neurones multicouche



Apprentissage avec rétropropagation

```
// Soit un réseau qui contient  $M$  couches
répéter:
    pour chaque exemple faire
        pour chaque noeud  $j$  de la couche d'entrée du réseau faire
             $a_j \leftarrow$  item  $j$  dans le vecteur d'entrée de l'exemple
        pour  $l = 2$  à  $M$  faire
             $in_i \leftarrow \sum_j W_{j,i} \times a_j$ 
             $a_i \leftarrow g(in_i)$ 
        pour chaque noeud  $i$  de la couche de sortie faire
            // soit  $y_i$  la sortie désirée pour le noeud  $i$  et  $a_i$  la sortie obtenue
             $\Delta_i \leftarrow g'(in_i) \times (y_i - a_i)$ 
        pour  $l = M-1$  à 1 faire
            pour chaque noeud  $j$  de la couche  $l$  faire
                 $\Delta_j \leftarrow g'(in_j) \times \sum_i W_{j,i} \Delta_i$ 
            pour chaque noeud  $i$  de la couche  $l+1$  faire
                 $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$ 
    jusqu'à ce que le critère d'arrêt soit satisfait
```

Réseau de neurones - exemple



Réseau de neurones - exemple

pour chaque noeud j de la couche d'entrée du réseau **faire**

$a_j = x_j$ ou item j dans le vecteur d'entrée de l'exemple

pour $i = 2$ à M **faire**

$$in_i = \sum_j W_{j,i} \times a_j$$

$$a_i = g(in_i)$$

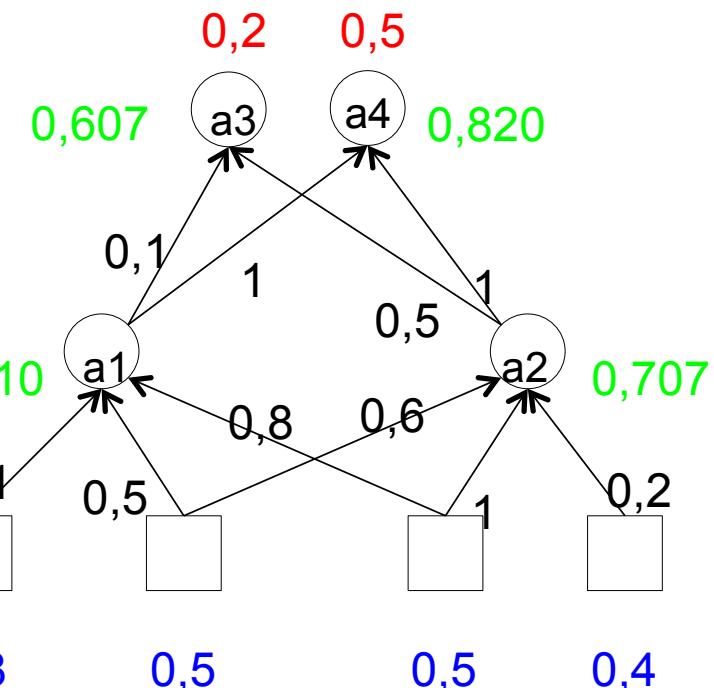
Exemple:

$$a_1 = g(w \cdot a)$$

$$= g([1 \quad .5 \quad .8] \cdot [.8 \quad .5 \quad .5])$$

$$= g(1.45)$$

$$= \frac{1}{1 + \exp(-1.45)} = .810$$



Réseau de neurones - exemple

pour chaque noeud j de la couche d'entrée du réseau **faire**

$a_j = x_j$ ou item j dans le vecteur d'entrée de l'exemple

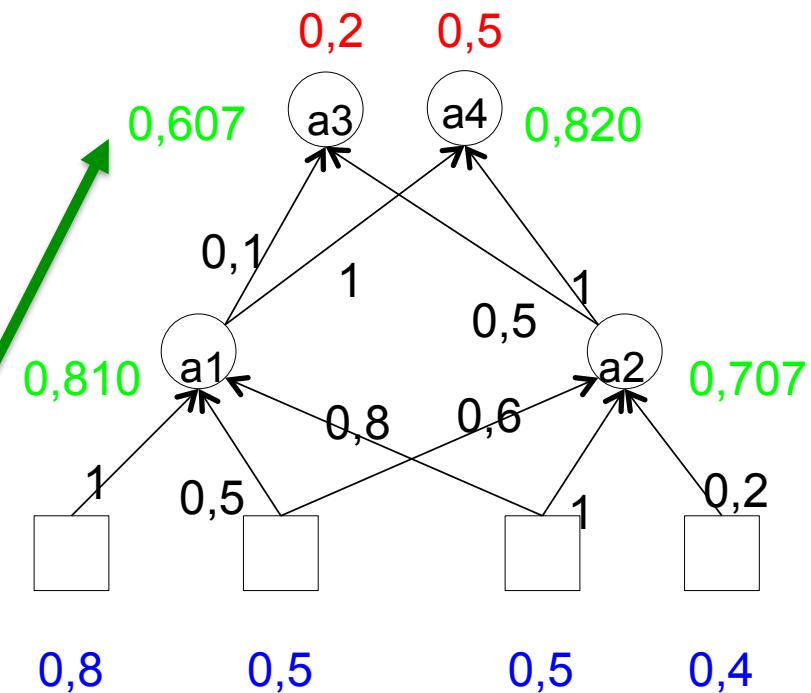
pour $i = 2$ à M **faire**

$$in_i = \sum_j W_{j,i} \times a_j$$

$$a_i = g(in_i)$$

Exemple:

$$\begin{aligned} a_1 &= g(w \cdot a) \\ &= g([.1 \quad .5] \cdot [.810 \quad .707]) \\ &= g(.4345) \\ &= \frac{1}{1 + \exp(-.4345)} = .607 \end{aligned}$$



Réseau de neurones - exemple

// soit y_i la sortie désirée pour le noeud i et a_i la sortie obtenue

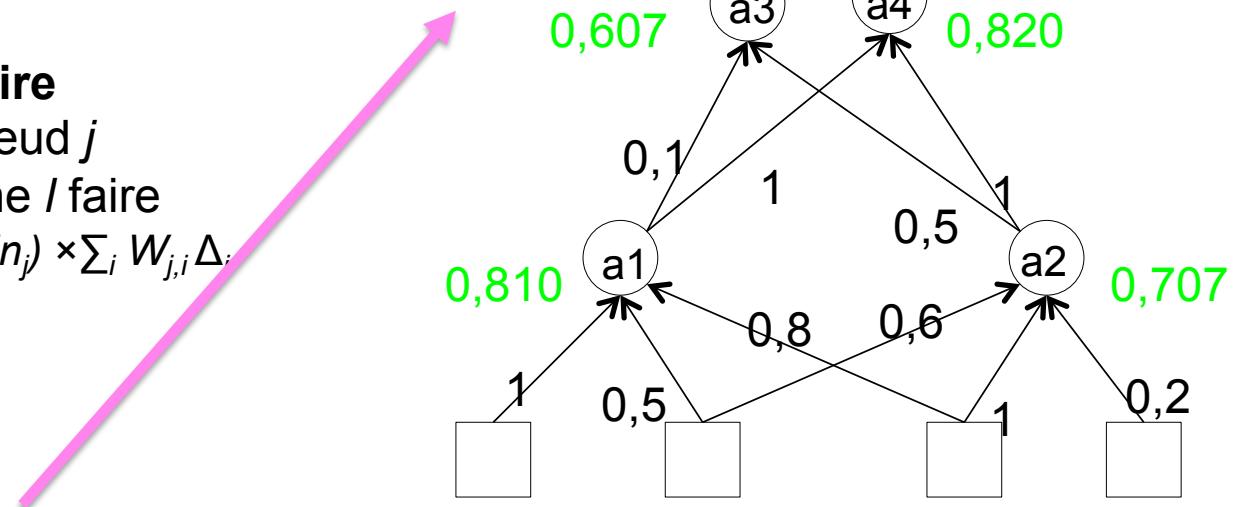
$$\Delta_i = g'(in_i) \times (y_i - a_i)$$

pour $l = M-1$ à 1 faire

pour chaque noeud j

de la couche l faire

$$\Delta_j = g'(in_j) \times \sum_i W_{j,i} \Delta_i$$

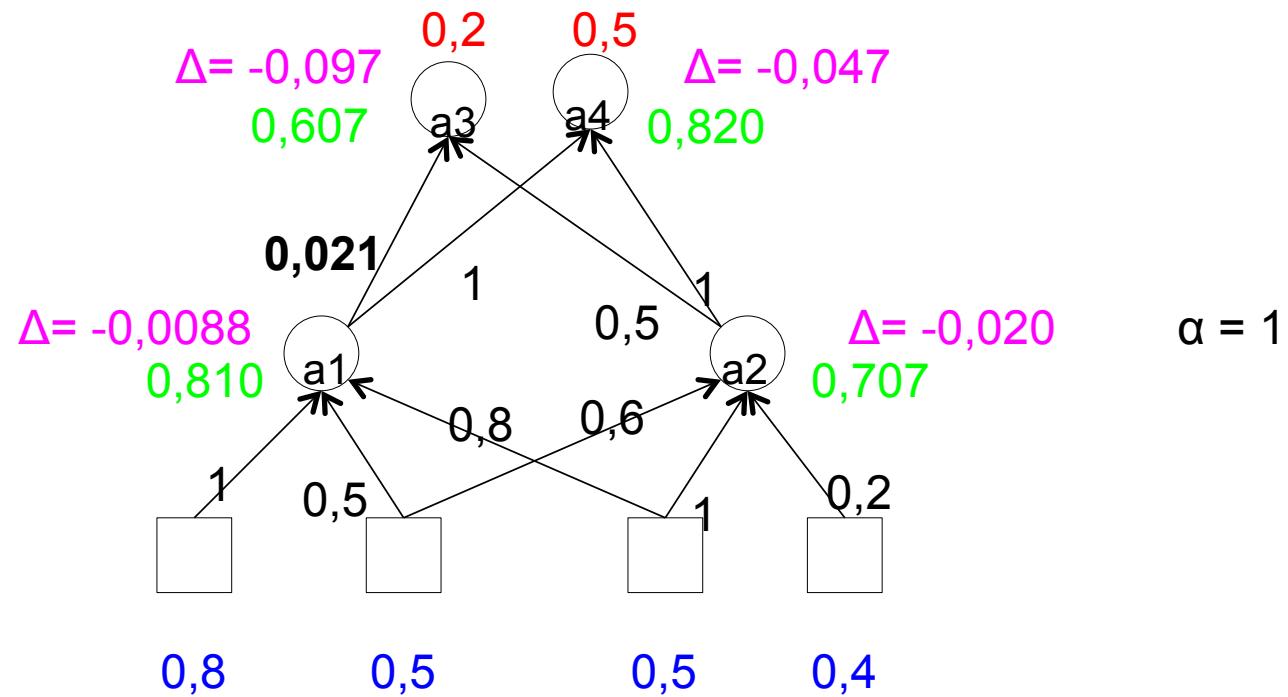


$$\begin{aligned}
 \Delta_i &= g'(in_i) \times (y_i - a_i) \\
 &= g(z)(1 - g(z)) \times (y - g(z)) \\
 &= .607(1 - .607) \times (.2 - .607) = -0.097
 \end{aligned}$$

Réseau de neurones - exemple

pour chaque noeud i de la couche $l+1$ faire

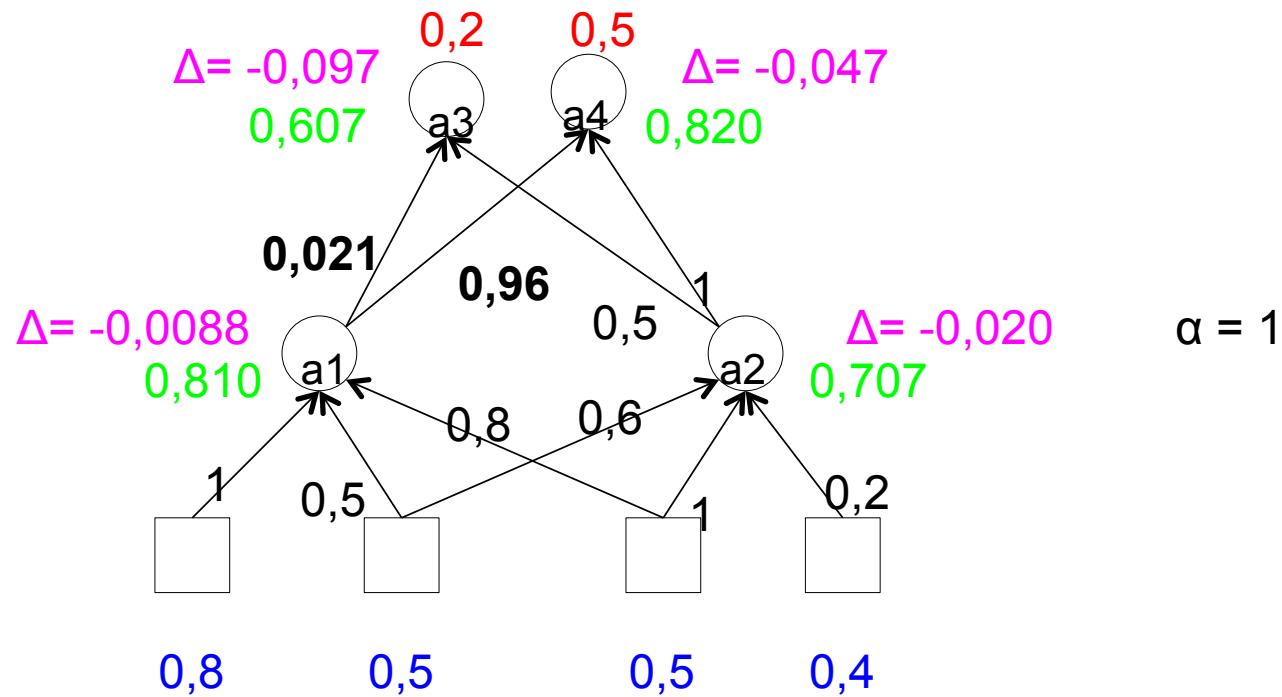
$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$



Réseau de neurones - exemple

pour chaque noeud i de la couche $l+1$ faire

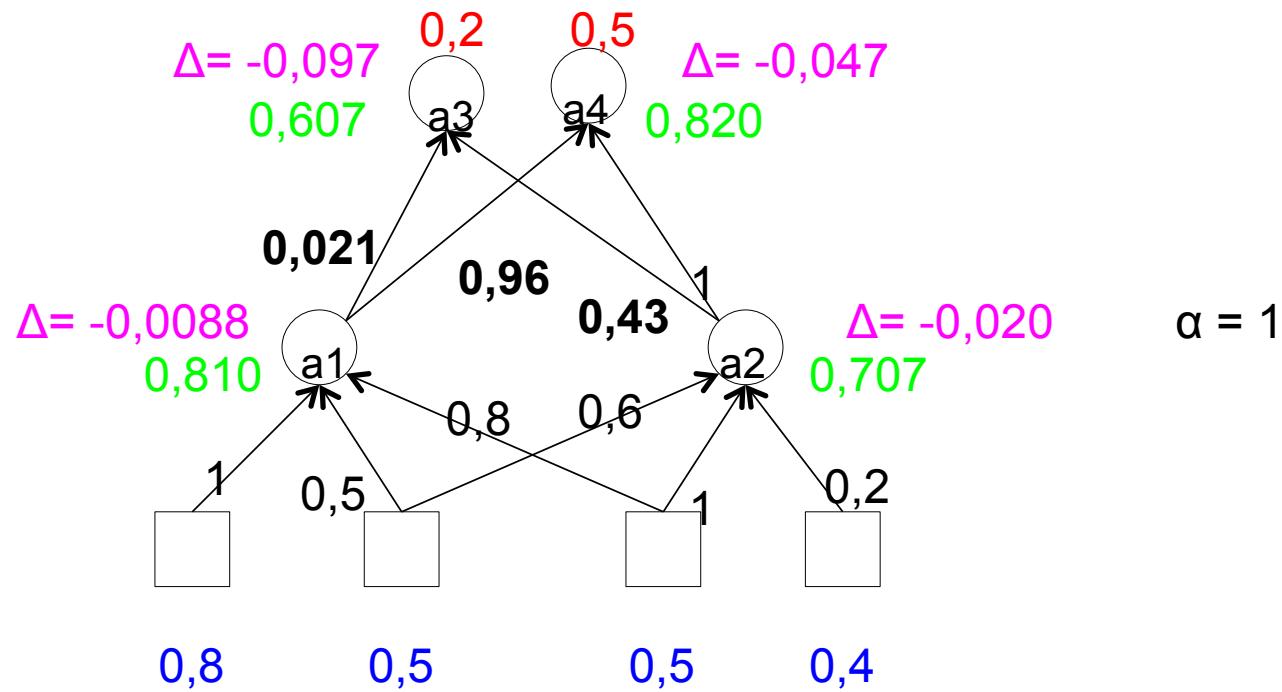
$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$



Réseau de neurones - exemple

pour chaque noeud i de la couche $l+1$ faire

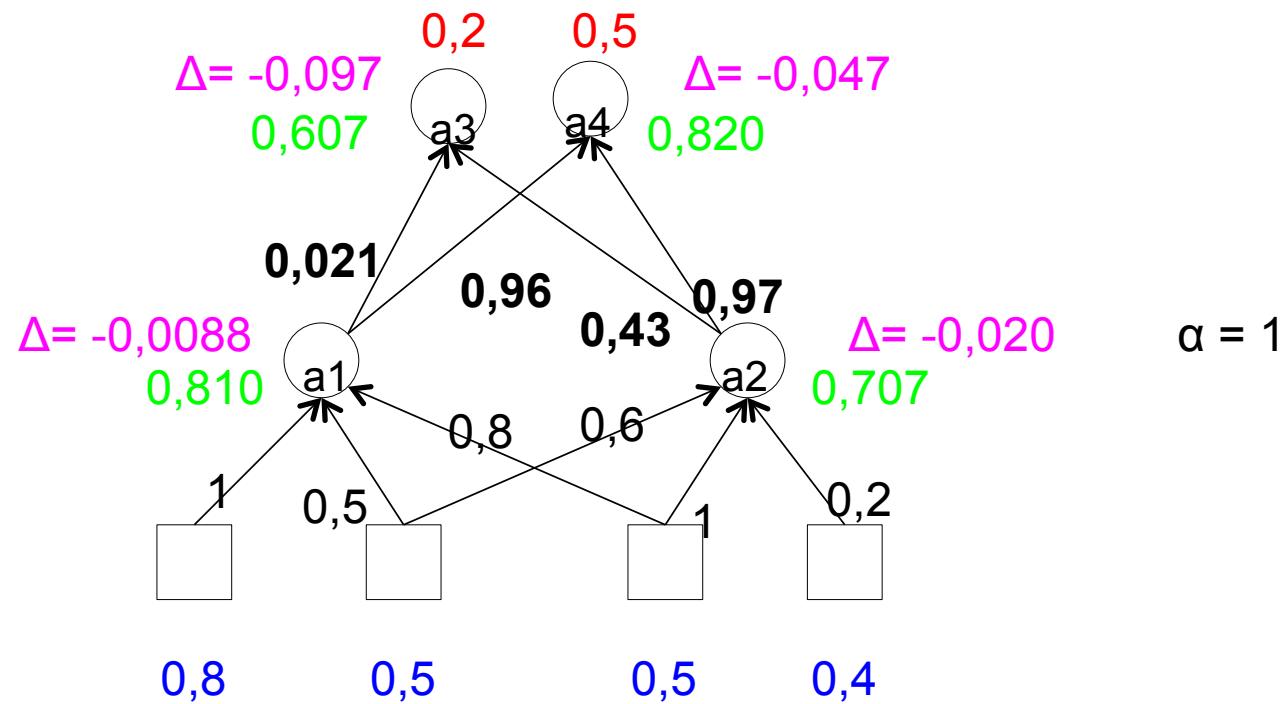
$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$



Réseau de neurones - exemple

pour chaque noeud i de la couche $l+1$ faire

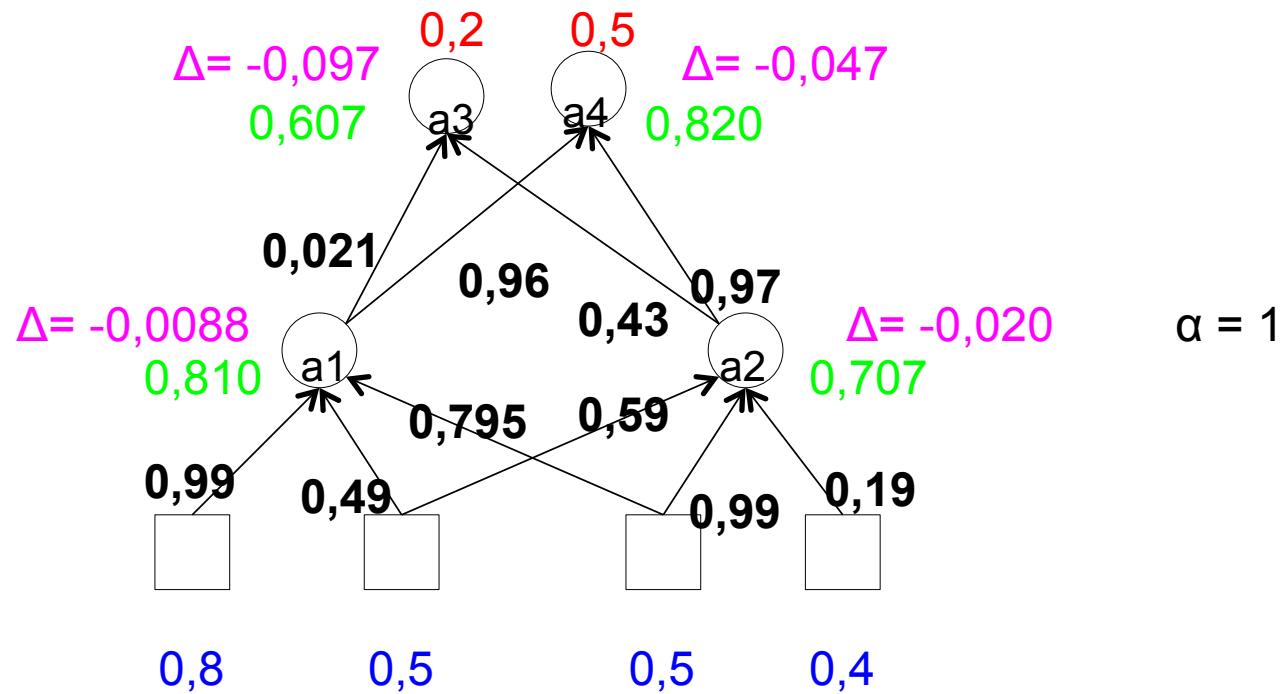
$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$



Réseau de neurones - exemple

pour chaque noeud i de la couche $l+1$ faire

$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$

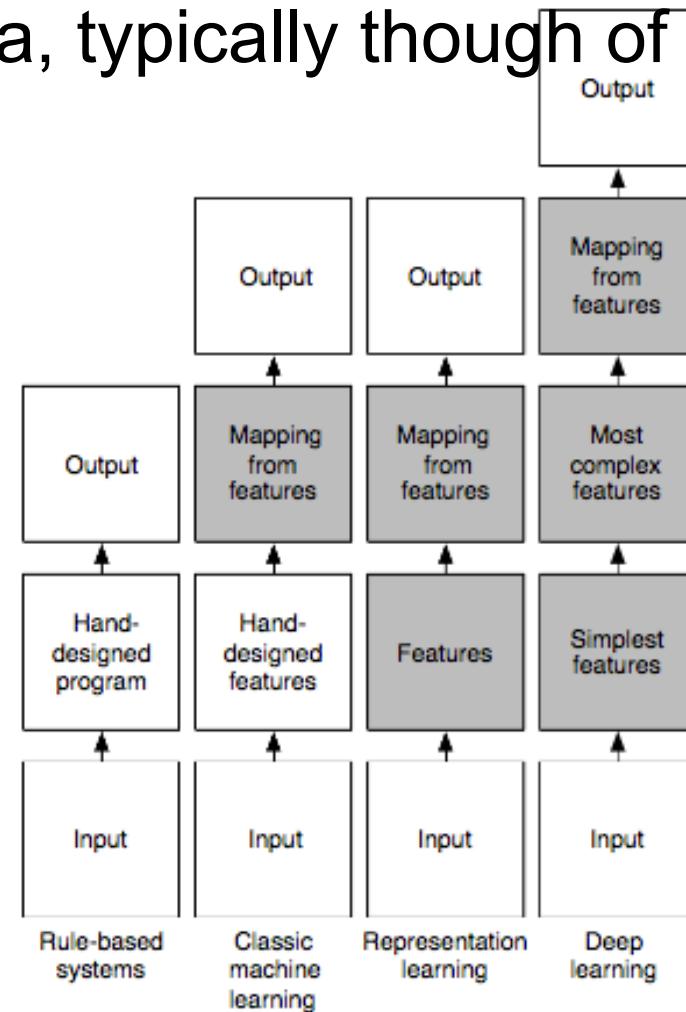


Deep Learning

Voir la leçon 6

Deep Learning

- Deep Learning methods seek to learn representations through a sequence of transformations of the data, typically thought of as layers in a network
- Flow charts from Bengio et al. (2014) “showing how the different parts of an AI system relate to each other within different AI disciplines.
- Shaded boxes indicate components that are able to learn from data”





Why are Deep Neural Networks so Hot?

- **Large-Vocabulary Speech Recognition**
Significant increase in performance (Dahl, Yu, Deng & Acero, 2012)
- Deep Neural Network : 16-23% relative error rate reduction over the previous state-of-the-art
- **Visual Recognition of 1000 Classes**
Top results on the ImageNet contest (Krizhevsky, Sutskever & Hinton, 2012), in 1.2 million images
- Deep Neural Network : 15% top-5 error rate
- Second best entry: 26% top-5 error rate
- **Face Verification**
Near human performance, 97.35% accuracy on the Labeled Faces in the Wild (LFW) evaluation, reducing the error of the state of the art at the time by more than 27% (Facebook Research, 2014)
- **Neural Machine Translation**
Much more accurate translations compared to a strong, highly engineered production grade system at Google



Recent article in Wired & New developments at Facebook

Google Hires Brains that Helped Supercharge Machine Learning

BY ROBERT MCMILLAN 03.13.13 6:30 AM



Geoffrey Hinton (right) Alex Krizhevsky, and Ilya Sutskever (left) will do machine learning work at Google. Photo: U of T

Google has hired the man who showed how to make computers learn much like the human brain.

- After winning ImageNet challenge
- Google buys Geoff Hinton's company



Yann LeCun

December 9, 2013 ·

- Facebook has created a new research lab with the ambitious, long-term goal of bringing about major advances in Artificial Intelligence.
- Simultaneously, Facebook and New York University's Center for Data Science are entering a partnership to carry out research in data science, machine learning, and AI.

An Explosion in Interest

- Lasagne (Library to easily make layers in Theano)
- Tensor Flow (Google – Nov. 2015)
- Computational Network Toolkit (CNTK) Microsoft
- Torch (Facebook)
- Chainer (Japanese effort)

Other related software

- Caffe (Berkeley)

Data Mining

Practical Machine Learning Tools and Techniques

Slides 1-55 for Chapter 10 - Deep learning

of *Data Mining* by I. H. Witten, E. Frank,
M. A. Hall, and C. J. Pal

Introducing Deep Learning

- In recent years, so-called “deep learning” approaches to machine learning have had a major impact on speech recognition and computer vision
- Other disciplines, such as natural language processing, are also starting to see benefits
- A critical ingredient is the use of much larger quantities of data than has heretofore been possible
- Recent successes have arisen in settings involving *high capacity* models—ones with many parameters
- Here, deep learning methods create flexible models that exploit information buried in massive datasets far more effectively than do traditional machine learning techniques using hand-engineered features

Views on machine learning

- One way to view machine learning is in terms of three general approaches:
 1. *Classical machine learning* techniques, which make predictions directly from a set of features that have been pre-specified by the user;
 2. *Representation learning* techniques, which transform features into some intermediate representation prior to mapping them to final predictions; and
 3. *Deep learning* techniques, a form of representation learning that uses multiple transformation steps to create very complex features

The neural network *renaissance* and deep learning *revolution*

- The term “renaissance” captures a massive resurgence of interest in neural networks and deep learning techniques
- Many high-profile media (e.g. *The New York Times*) have documented the striking successes of deep learning techniques on key benchmark problems
- Starting around 2012, impressive results were achieved on long-standing problems in speech recognition and computer vision, and in competitive challenges such as the ImageNet *Large Scale Visual Recognition Challenge* and the *Labeled Faces in the Wild* evaluation

GPUs, graphs and tensors

- The easy availability of high-speed computation in the form of graphics processing units has been critical to the success of deep learning techniques
- When formulated in matrix-vector form, computation can be accelerated using optimized graphics libraries and hardware
- This is why we will study backpropagation in matrix-vector form
 - Readers unfamiliar with manipulating functions that have matrix arguments, and their derivatives are advised to consult Appendix A.1 for a summary of some useful background
- As network models become more complex, some quantities can only be represented using multidimensional arrays of numbers
 - Such arrays are sometimes referred to as tensors, a generalization of matrices that permit an arbitrary number of indices
- Software for deep learning supporting *computation graphs* and *tensors* is therefore invaluable for accelerating the creation of complex network structures and making it easier to learn them

Key developments

The following developments have played a crucial role in the resurgence of neural network methods:

- the proper evaluation of machine learning methods;
- vastly increased amounts of data;
- deeper and larger network architectures;
- accelerated training using GPU techniques

Mixed National Institute of Standards and Technology (MNIST)

- Is a database and evaluation setup for handwritten digit recognition
- Contains 60,000 training and 10,000 test instances of hand-written digits, encoded as 28×28 pixel grayscale images
- The data is a re-mix of an earlier NIST dataset in which adults generated the training data and high school students generated the test set
- Lets compare the performance of different methods

MNIST

Classifier	Test Error Rate (%)	References
Linear classifier (1-layer neural net)	12.0	LeCun et al. (1998)
K-nearest-neighbors, Euclidean (L2)	5.0	LeCun et al. (1998)
2-Layer neural net, 300 hidden units, mean square error	4.7	LeCun et al. (1998)
Support vector machine, Gaussian kernel	1.4	MNIST Website
Convolutional net, LeNet-5 (no distortions)	0.95	LeCun et al. (1998)
Methods using distortions		
Virtual support vector machine, deg-9 polynomial, (2-pixel jittered and deskewing)	0.56	DeCoste and Scholkopf (2002)
Convolutional neural net (elastic distortions)	0.4	Simard, Steinkraus, and Platt (2003)
6-Layer feedforward neural net (on GPU) (elastic distortions)	0.35	Ciresan, Meier, Gambardella, and Schmidhuber (2010)
Large/deep convolutional neural net (elastic distortions)	0.35	Ciresan, Meier, Masci, Maria Gambardella, and Schmidhuber (2011)
Committee of 35 convolutional networks (elastic distortions)	0.23	Ciresan, Meier, and Schmidhuber (2012)

Losses and regularization

- Logistic regression can be viewed as a simple neural network with no hidden units
- The underlying optimization criterion for predicting $i=1,\dots,N$ labels y_i from features \mathbf{x}_i with parameters θ consisting of a matrix of weights \mathbf{W} and a vector of biases \mathbf{b} can be viewed as

$$\sum_{i=1}^N -\log p(y_i | \mathbf{x}_i; \mathbf{W}, \mathbf{b}) + \lambda \sum_{j=1}^M w_j^2 = \sum_{i=1}^N L(f_i(\mathbf{x}_i; \theta), y_i) + \lambda R(\theta)$$

- where the first term, $L(f_i(\mathbf{x}_i; \theta), y_i)$, is the negative conditional log-likelihood or *loss*, and
- the second term, $\lambda R(\theta)$, is a weighted *regularizer* used to prevent overfitting

Empirical risk minimization

- This formulation as a loss- and regularizer-based objective function gives us the freedom to choose either probabilistic losses or other loss functions
- Using the *average loss* over the training data, called the *empirical risk*, leads to the following formulation of the optimization problem: minimize the empirical risk plus a regularization term, i.e.

$$\arg \min_{\theta} \left[\frac{1}{N} \sum_{i=1}^N L(f_i(\mathbf{x}_i; \theta), \mathbf{y}_i) + \lambda R(\theta) \right].$$

- Note that the factor N must be accounted for if one relates the regularization weight here to the corresponding parameter derived from a formal probabilistic model for a distribution on parameters

In practice

- In deep learning we are often interested in examining learning curves that show the loss or some other performance metric on a graph as a function of the number of passes that an algorithm has taken over the data.
- It is much easier to compare the *average* loss over a training set with the *average* loss over a validation set on the same graph, because dividing by N gives them the same scale.

Common losses for neural networks

- The final output function of a neural network typically has the form $f_k(\mathbf{x})=f_k(a_k(\mathbf{x}))$, where $a_k(\mathbf{x})$ is just one of the elements of vector function $\mathbf{a}(\mathbf{x})=\mathbf{W}\mathbf{h}(\mathbf{x})+\mathbf{b}$
- Commonly used output loss functions, output activation functions, and the underlying distributions from which they derive are shown below

Loss Name, $L(f_i(\mathbf{x}_i; \theta), \mathbf{y}_i) =$	Distribution Name, $P(f_i(\mathbf{x}_i; \theta), \mathbf{y}_i) =$	Output Activation Function, $f_k(a_k(\mathbf{x})) =$
Squared error, $\sum_{k=1}^K (f_k(\mathbf{x}) - y_k)^2$	Gaussian, $N(\mathbf{y}; \mathbf{f}(\mathbf{x}; \theta), \mathbf{I})$	$\frac{1}{(1 + \exp(-a_k(\mathbf{x})))}$
Cross entropy, $-\sum_{k=1}^K [y_k \log f_k(\mathbf{x}) + (1 - y_k) \log(1 - f_k(\mathbf{x}))]$	Bernoulli, $\text{Bern}(\mathbf{y}; \mathbf{f}(\mathbf{x}; \theta))$	$\frac{1}{(1 + \exp(-a_k(\mathbf{x})))}$
Softmax, $-\sum_{k=1}^K y_k \log f_k(\mathbf{x})$	Discrete or Categorical, $\text{Cat}(\mathbf{y}; \mathbf{f}(\mathbf{x}; \theta))$	$\frac{\exp(a_k(\mathbf{x}))}{\sum_{j=1}^K \exp(a_j(\mathbf{x}))}$

Deep neural network architectures

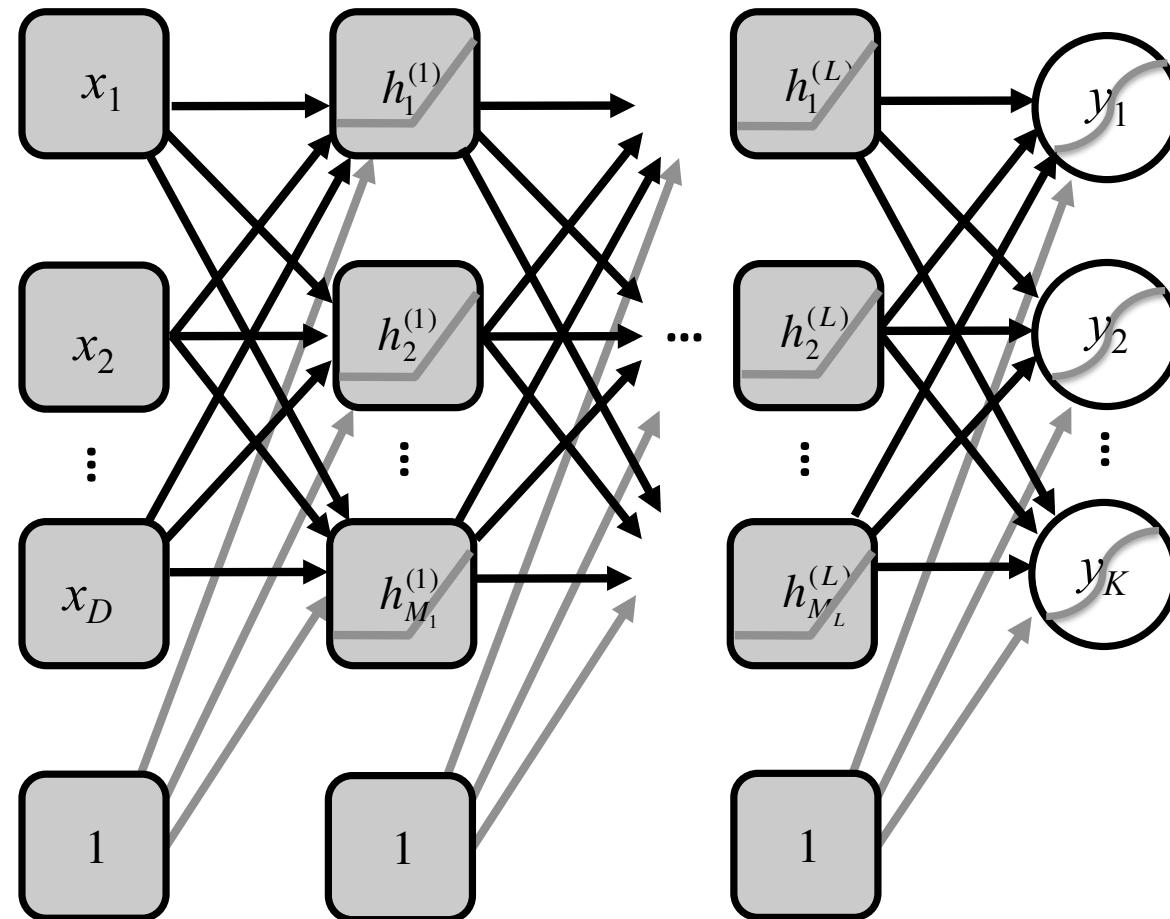
- Compose computations performed by many layers
- Denoting the output of hidden layers by $\mathbf{h}^{(l)}(\mathbf{x})$, the computation for a network with L hidden layers is:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f} \left[\mathbf{a}^{(L+1)} \left(\mathbf{h}^{(L)} \left(\mathbf{a}^{(L)} \left(\dots \left(\mathbf{h}^{(2)} \left(\mathbf{a}^{(2)} \left(\mathbf{h}^{(1)} \left(\mathbf{a}^{(1)}(\mathbf{x}) \right) \right) \right) \right) \right) \right) \right) \right]$$

- Where *pre-activation functions* $\mathbf{a}^{(l)}(\mathbf{x})$ are typically linear, of the form $\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}$ with matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$
- This formulation can be expressed using a single parameter matrix Θ with the trick of defining $\hat{\mathbf{x}}$ as \mathbf{x} with a 1 appended to the end of the vector; we then have $\mathbf{a}^{(l)}(\hat{\mathbf{x}}) = \Theta^{(l)}\hat{\mathbf{x}}$, $l=1$

$$\mathbf{a}^{(l)}(\hat{\mathbf{h}}^{(l-1)}) = \Theta^{(l)}\hat{\mathbf{h}}^{(l-1)}, l>1$$

Deep feedforward networks

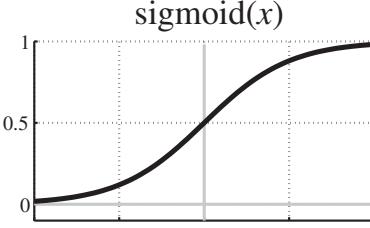
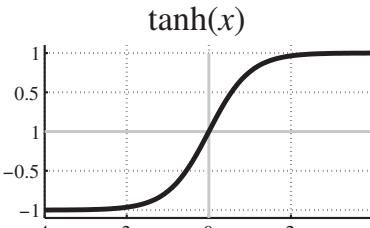
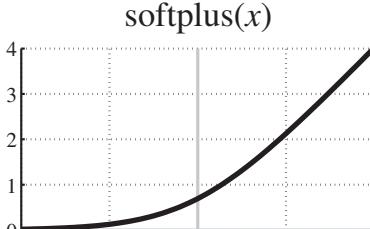
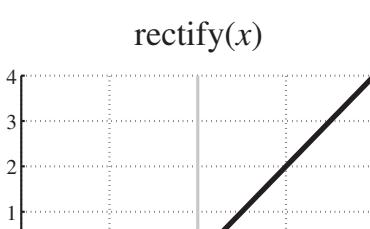


- Unlike Bayesian networks the hidden units here are *intermediate deterministic computations* not random variables, which is why they are not represented as circles
- However, the output variables y_k are drawn as circles because they can be formulated probabilistically

Activation functions

- Activation functions, $\mathbf{h}^{(l)}(\mathbf{x})$ generally operate on the pre-activation vectors in an *element-wise* fashion
- While sigmoid functions have been popular, the hyperbolic tangent function is sometimes preferred, partly because it has a steady state at 0
- More recently the *rectify()* function or rectified linear units (ReLUs) have been found to yield superior results in many different settings
 - Since ReLUs are 0 for negative argument values, some units in the model will yield activations that are 0, giving a sparseness property that is useful in many contexts
 - The gradient is particularly simple—either 0 or 1
 - This helps address the *exploding gradient problem*
- A number of software packages make it easy to use a variety of activation functions, determining gradients automatically using symbolic computations

Activation functions

Name and Graph	Function	Derivative
 <p>$\text{sigmoid}(x)$</p>	$h(x) = \frac{1}{1 + \exp(-x)}$	$h'(x) = h(x)[1 - h(x)]$
 <p>$\tanh(x)$</p>	$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$h'(x) = 1 - h(x)^2$
 <p>$\text{softplus}(x)$</p>	$h(x) = \log(1 + \exp(x))$	$h'(x) = \frac{1}{1 + \exp(-x)}$
 <p>$\text{rectify}(x)$</p>	$h(x) = \max(0, x)$	$h'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$

Bibliographic Notes & Further Reading

- The backpropagation algorithm has been known in close to its current form since Werbos (1974)'s PhD thesis
- In his extensive literature review of deep learning, Schmidhuber (2015) traces key elements of the algorithm back even further.
 - He also traces the idea of “deep networks” back to the work of Ivakhnenko and Lapa (1965).
- The popularity of neural network techniques has gone through several cycles and while some factors are social, there are important technical reasons behind the trends.
- A single-layer neural network cannot solve the XOR problem, a failing that was derided by Minsky and Papert (1969) and which stymied neural network development in the following decades.

Bibliographic Notes & Further Reading

- It is well known that networks with one additional layer can approximate any function (Cybenko, 1989; Hornik, 1991), and Rumelhart et al. (1986)'s influential work re-popularized neural network methods for a while.
- By the early 2000s neural network methods had fallen out of favor again – kernel methods like SVMs yielded state of the art results on many problems and were convex
- Indeed, the organizers of NIPS, the *Neural Information Processing Systems* conference, which was (and still is) widely considered to be the premier forum for neural network research, found that the presence of the term “neural networks” in the title was highly correlated with the paper’s rejection!
 - A fact that is underscored by citation analysis of key neural network papers during this period.
- In this context, the recent resurgence of interest in deep learning really does feel like a “revolution.”

Bibliographic Notes & Further Reading

- It is known that most complex Boolean functions require an exponential number of two-step logic gates for their representation (Wegener, 1987).
- The solution appears to be greater depth: according to Bengio (2014), the evidence strongly suggests that “functions that can be compactly represented with a depth- k architecture could require a very large number of elements in order to be represented by a shallower architecture”.

Backpropagation revisited in vector matrix form

Backpropagation in matrix vector form

- Backpropagation is based on the chain rule of calculus
- Consider the loss for a single-layer network with a softmax output (which corresponds exactly to the model for multinomial logistic regression)
- We use multinomial vectors \mathbf{y} , with a single dimension $y_k = 1$ for the corresponding class label and whose other dimensions are 0
- Define $\mathbf{f} = [f_1(\mathbf{a}), \dots, f_K(\mathbf{a})]^T$, and $a_k(\mathbf{x}; \boldsymbol{\theta}_k) = \boldsymbol{\theta}_k^T \mathbf{x}$, $\mathbf{a}(\mathbf{x}; \boldsymbol{\theta}) = [a_1(\mathbf{x}; \boldsymbol{\theta}_1), \dots, a_K(\mathbf{x}; \boldsymbol{\theta}_K)]^T$ where $\boldsymbol{\theta}_k$ is a column vector containing the k^{th} row of the parameter matrix
- Consider the softmax loss for $\mathbf{f}(\mathbf{a}(\mathbf{x}))$

Logistic regression and the chain rule

- Given loss $L = -\sum_{k=1}^K y_k \log f_k(\mathbf{x}), \quad f_k(\mathbf{x}) = \frac{\exp(a_k(\mathbf{x}))}{\sum_{c=1}^K \exp(a_c(\mathbf{x}))}$.
- Use the chain rule to obtain

$$\frac{\partial L}{\partial \theta_k} = \frac{\partial \mathbf{a}}{\partial \theta_k} \frac{\partial \mathbf{f}}{\partial \mathbf{a}} \frac{\partial L}{\partial \mathbf{f}} = \frac{\partial \mathbf{a}}{\partial \theta_k} \frac{\partial L}{\partial \mathbf{a}}.$$

- Note the order of terms - in vector matrix form terms build from right to left

$$\begin{aligned}\frac{\partial L}{\partial a_j} &= \frac{\partial}{\partial a_j} \left[-\sum_{k=1}^K y_k \left[a_k - \log \left[\sum_{c=1}^K \exp(a_c) \right] \right] \right] \\ &= - \left[y_{k=j} - \frac{\exp(a_{k=j})}{\sum_{c=1}^K \exp(a_c)} \right] = -[y_j - p(y_j | \mathbf{x})] = -[y_j - f_j(\mathbf{x})],\end{aligned}$$

Matrix vector form of gradient

- We can write

$$\frac{\partial L}{\partial \mathbf{a}} = -[\mathbf{y} - \mathbf{f}(\mathbf{x})] \equiv -\Delta$$

and since

$$\frac{\partial a_j}{\partial \theta_k} = \begin{cases} \frac{\partial}{\partial \theta_k} \theta_k^T \mathbf{x} = \mathbf{x} & , j = k \\ 0 & , j \neq k \end{cases}$$

we have

$$\frac{\partial \mathbf{a}}{\partial \theta_k} = \mathbf{H}_k = \begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}$$

- Notice that we avoid working with the partial derivative of the vector \mathbf{a} with respect to the matrix θ , because it cannot be represented as a matrix — it is a multidimensional array of numbers (a tensor).

A compact expression for the gradient

- The gradient (as a column vector) for the vector in the k th row of the parameter matrix

$$\frac{\partial L}{\partial \theta_k} = \frac{\partial \mathbf{a}}{\partial \theta_k} \frac{\partial L}{\partial \mathbf{a}} = - \begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix} [\mathbf{y} - \mathbf{f}(\mathbf{x})]$$
$$= -\mathbf{x}(y_k - f_k(x)).$$

- With a little rearrangement the gradient for the entire matrix of parameters can be written compactly:

$$\frac{\partial L}{\partial \theta} = -[\mathbf{y} - \mathbf{f}(\mathbf{x})] \mathbf{x}^T = -\Delta \mathbf{x}^T.$$

Consider now a multilayer network

- Using the same activation function for all L hidden layers, and a softmax output layer
- The gradient of the k^{th} parameter vector of the $L+1^{\text{th}}$ matrix of parameters is

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(L+1)}} &= \frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}, \quad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -\Delta^{(L+1)} \\ &= -\frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \Delta^{(L+1)} \\ &= -\mathbf{H}_k^L \Delta^{(L+1)} \quad \Rightarrow \quad \frac{\partial L}{\partial \boldsymbol{\theta}^{(L+1)}} = -\Delta^{(L+1)} \tilde{\mathbf{h}}_{(L)}^T.\end{aligned}$$

where \mathbf{H}_k^L is a matrix containing the activations of the corresponding hidden layer, in column k

Backpropagating errors

- Consider the computation for the gradient of the k^{th} row of the L^{th} matrix of parameters
- Since the bias terms are constant, it is unnecessary to backprop through them, so

$$\begin{aligned}\frac{\partial L}{\partial \Theta_k^{(L)}} &= \frac{\partial \mathbf{a}^{(L)}}{\partial \Theta_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} \\ &= -\frac{\partial \mathbf{a}^{(L)}}{\partial \Theta_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)}, \quad \Delta^{(L)} \equiv \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)} \\ &= -\frac{\partial \mathbf{a}^{(L)}}{\partial \Theta_k^{(L)}} \Delta^{(L)}\end{aligned}$$

- Similarly, we can define $\Delta^{(l)}$ recursively in terms of $\Delta^{(l+1)}$

Backpropagating errors

- The backpropagated error can be written as simply

$$\Delta^{(l)} = \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \Delta^{(l+1)}, \quad \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{a}^{(l)}} = \mathbf{D}^{(l)}, \quad \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{h}^{(l)}} = \mathbf{W}^{T(l+1)},$$
$$\Delta^{(l)} = \mathbf{D}^{(l)} \mathbf{W}^{T(l+1)} \Delta^{(l+1)}$$

where $\mathbf{D}^{(l)}$ contains the partial derivatives of the hidden-layer activation function with respect to the pre-activation input.

- $\mathbf{D}^{(l)}$ is generally diagonal, because activation functions usually operate on an elementwise basis
- $\mathbf{W}^{T(l+1)}$ arises from the fact that $\mathbf{a}^{(l+1)}(\mathbf{h}^{(l)}) = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}$

A general form for gradients

- The gradients for the k^{th} vector of parameters of the l^{th} network layer can therefore be computed using products of matrices of the following form

$$\frac{\partial L}{\partial \theta_k^{(l)}} = -\mathbf{H}_k^{(l-1)} \mathbf{D}^{(l)} \mathbf{W}^{T(l+1)} \dots \mathbf{D}^{(L)} \mathbf{W}^{T(L+1)} \Delta^{(L+1)}, \quad \frac{\partial L}{\partial \theta^{(l)}} = -\Delta^{(l)} \hat{\mathbf{h}}_{(l-1)}^T$$

- When $l=1$, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since \mathbf{D} is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product \circ by extracting the diagonal for \mathbf{d}

$$\Delta^{(l)} = \mathbf{D}^{(l)} (\mathbf{W}^{T(l+1)} \Delta^{(l+1)}) = \mathbf{d}^{(l)} \circ (\mathbf{W}^{T(l+1)} \Delta^{(l+1)})$$

Visualizing backpropagation

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)}$$

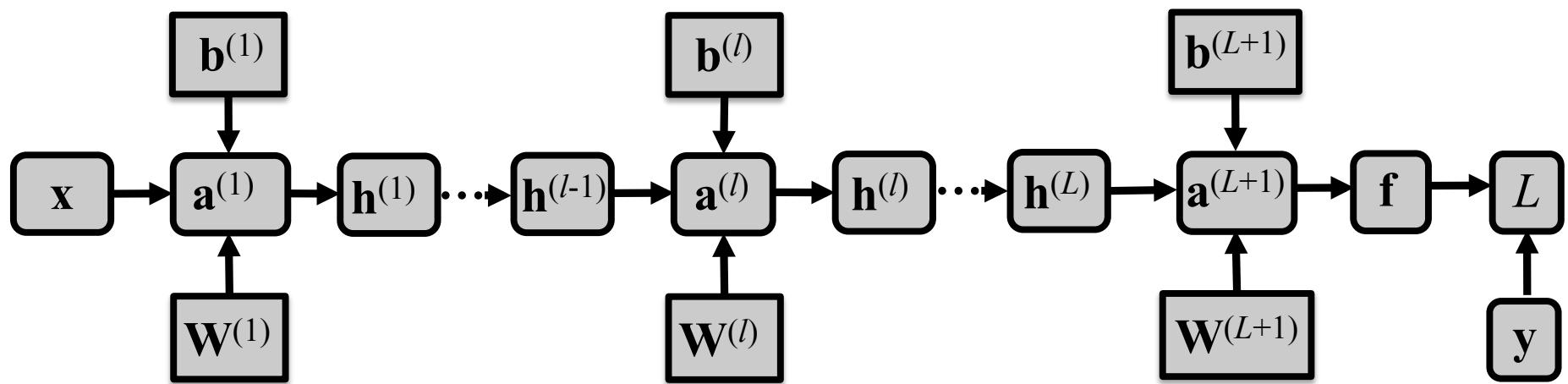
$$\mathbf{h}^{(1)} = \text{act}(\mathbf{a}^{(1)})$$

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{h}^{(l)} = \text{act}(\mathbf{a}^{(l)})$$

$$\mathbf{a}^{(L+1)} = \mathbf{W}^{(L+1)} \mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

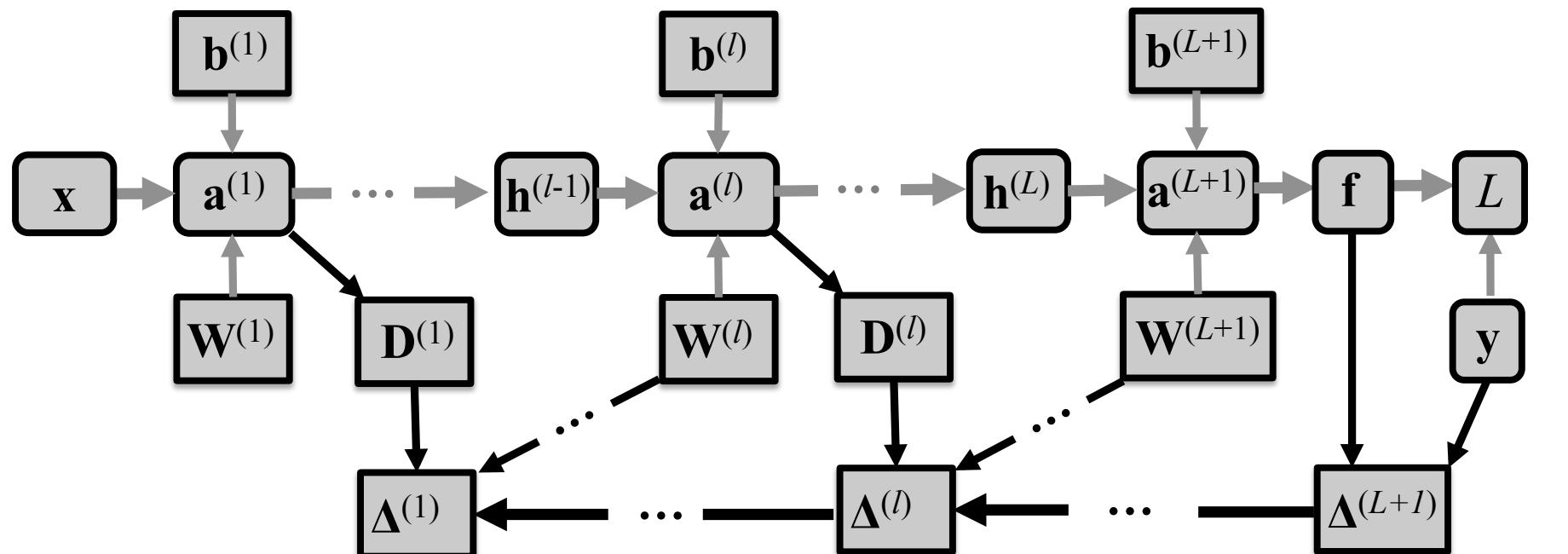
$$\mathbf{f} = \text{out}(\mathbf{a}^{(L+1)})$$



- In the forward propagation phase we compute terms of the form above
- The figure above is a type of computation graph, (which is different from the probability graphs we saw earlier)

Visualizing backpropagation

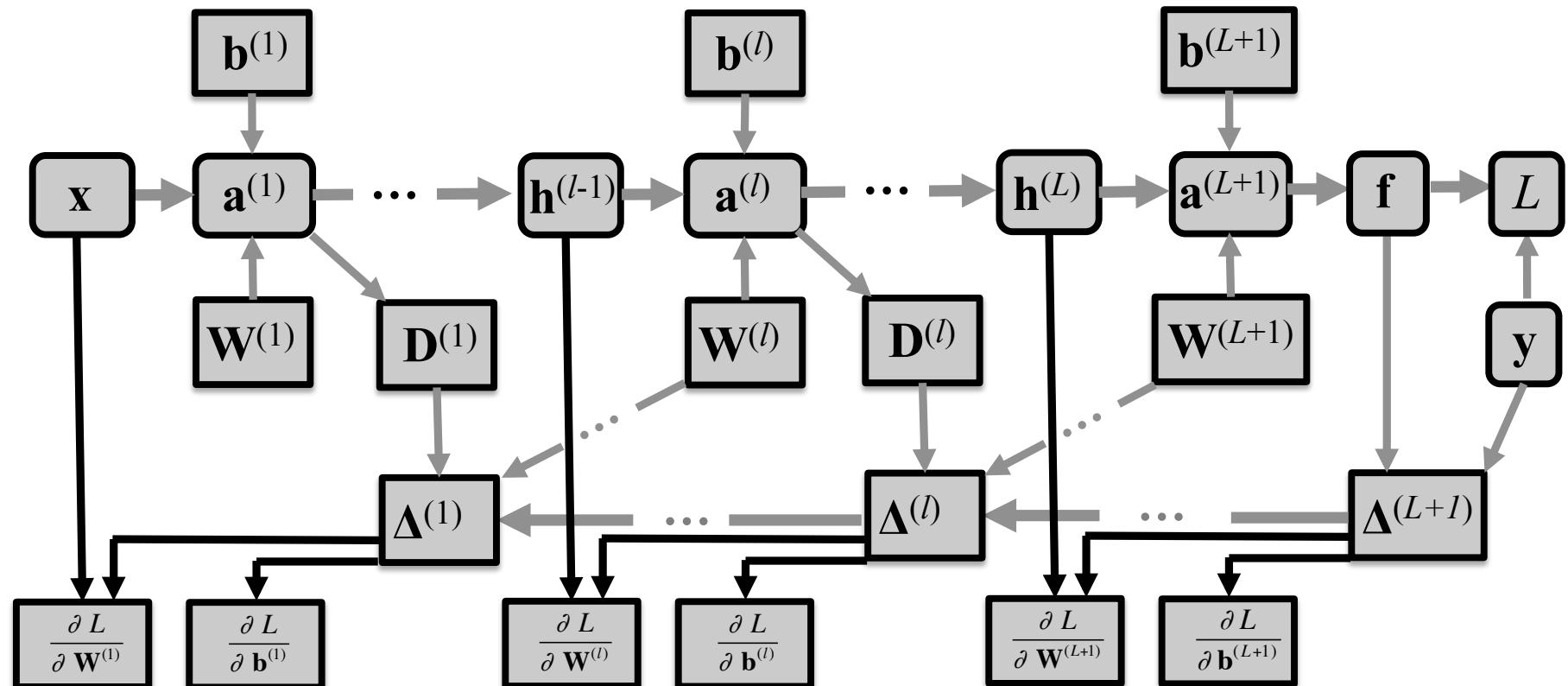
- In the backward propagation phase we compute terms of the form below



$$\Delta^{(l)} = \mathbf{D}^{(l)} \mathbf{W}^{\text{T}(l+1)} \Delta^{(l+1)} \quad \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{a}^{(l)}} = \mathbf{D}^{(l)} \quad \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{h}^{(l)}} = \mathbf{W}^{\text{T}(l+1)} \quad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -[\mathbf{y} - \mathbf{f}(\mathbf{x})] = -\Delta^{(L+1)}$$

Visualizing backpropagation

- We update the parameters in our model using the simple computations below



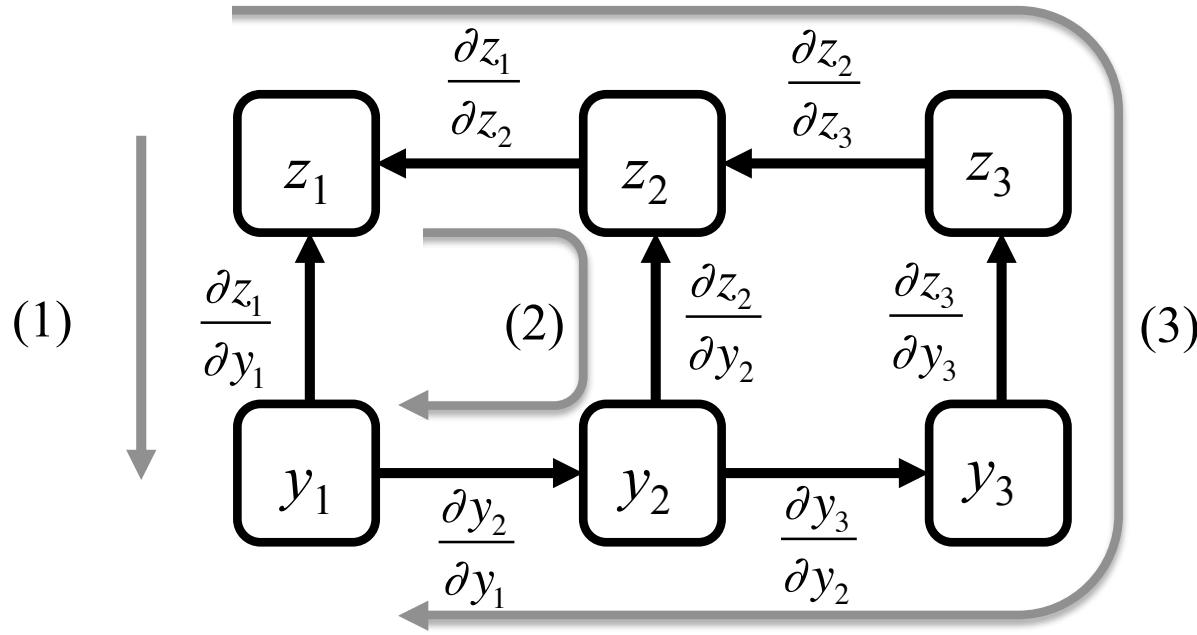
$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = -\Delta^{(1)} \mathbf{x}^T$$

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = -\Delta^{(1)}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = -\Delta^{(l)} \mathbf{h}_{(l-1)}^T$$

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = -\Delta^{(l)}$$

Computation graphs



- For more complicated computations, computation graphs can help us keep track of how computations decompose, ex. $z_1 = z_1(y_1, z_2(y_2(y_1)), z_3(y_3(y_2(y_1)))))$

$$\frac{\partial z_1}{\partial y_1} = \underbrace{\frac{\partial z_1}{\partial y_1}}_{(1)} + \underbrace{\frac{\partial z_1}{\partial z_2} \frac{\partial z_2}{\partial y_2} \frac{\partial y_2}{\partial y_1}}_{(2)} + \underbrace{\frac{\partial z_1}{\partial z_2} \frac{\partial z_2}{\partial z_3} \frac{\partial z_3}{\partial y_3} \frac{\partial y_3}{\partial y_2} \frac{\partial y_2}{\partial y_1}}_{(3)}$$

Checking an implementation of backpropagation and software tools

- An implementation of the backpropagation algorithm can be checked for correctness by comparing the analytic values of gradients with those computed numerically
- For example, one can add and subtract a small perturbation to each parameter and then compute the symmetric finite difference approximation to the derivative of the loss:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \varepsilon) - L(\theta - \varepsilon)}{2\varepsilon}$$

- Many software packages use computation graphs to allow complex networks to be more easily defined and optimized
- Examples include: Theano, TensorFlow, Keras and Torch

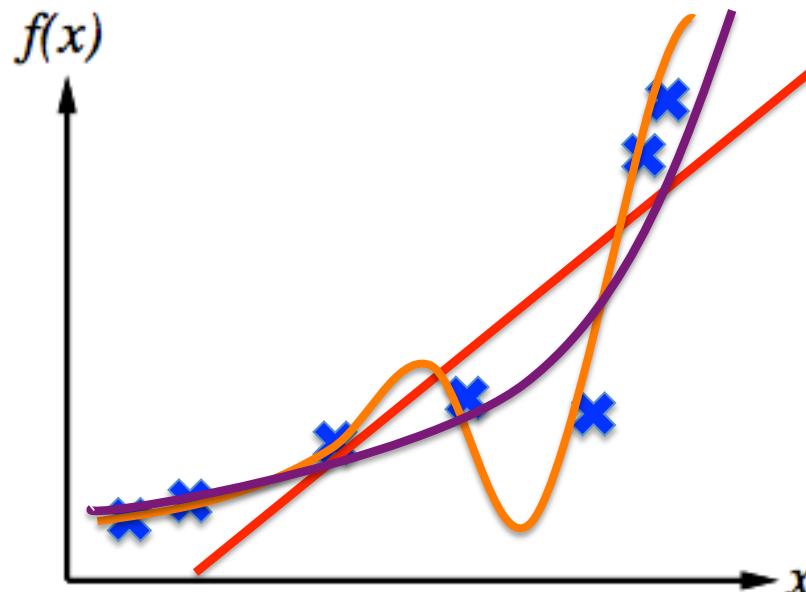
Bibliographic Notes & Further Reading

- Many neural network books (Haykin, 1994; Bishop, 1995; Ripley, 1996) do not formulate backpropagation in vector-matrix terms.
- However, recent online courses (e.g. by Hugo Larochelle), and Rojas (1996)'s text, do adopt this formulation, as we have done here

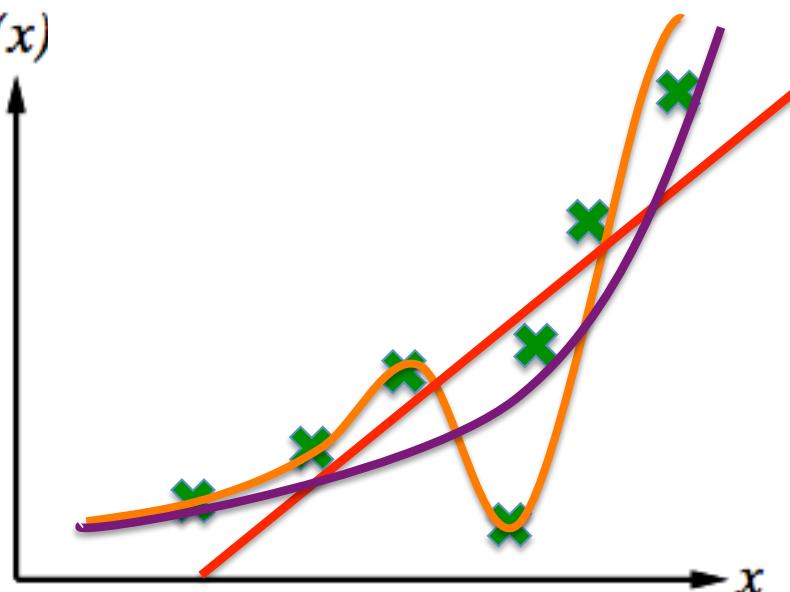
Le choix du modèle et l'ajustement des hyperparamètres

Exemple

- Sélection de la complexité du modèle
- Utilisez l'ensemble de validation pour choisir l'ordre du polynôme. Q: Comment exactement?



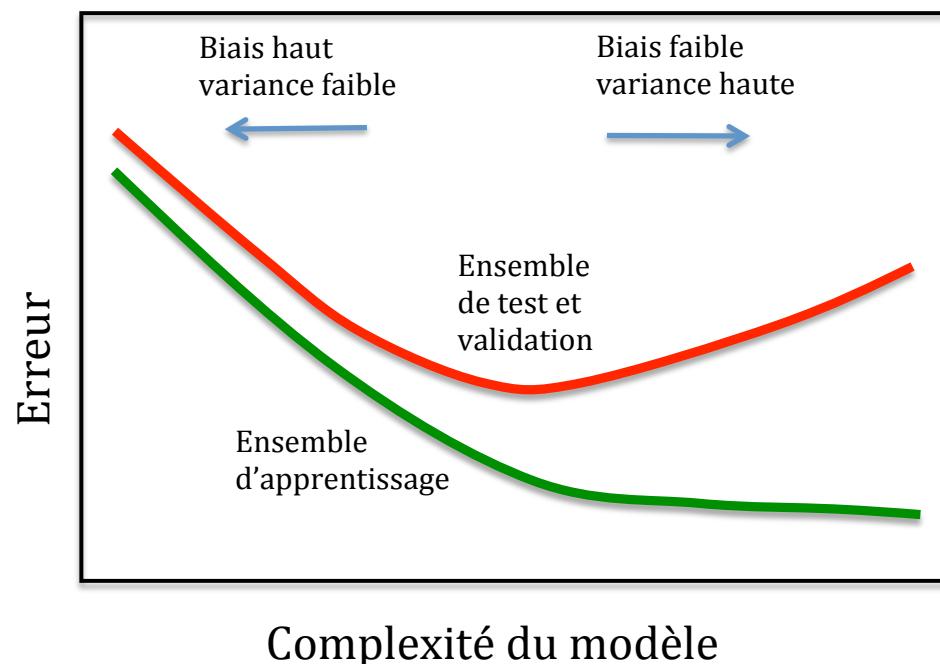
Ensemble d'apprentissage



Ensemble de validation

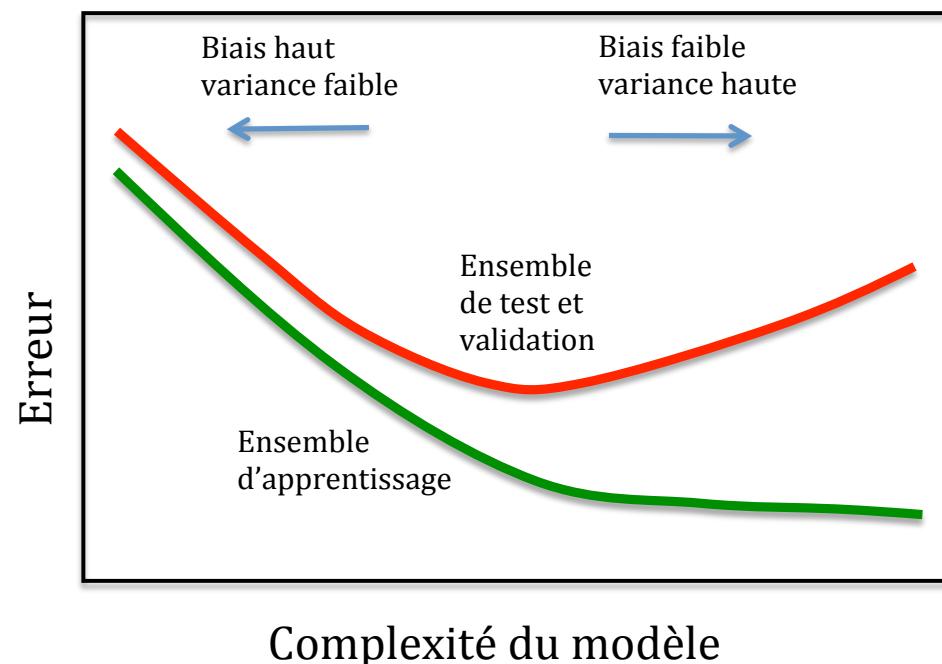
La compromis entre le biais et la variance

- En général, il existe un compromis entre des hypothèses complexes qui approchent bien les données d'apprentissage et des hypothèses plus simples qui peuvent mieux généraliser



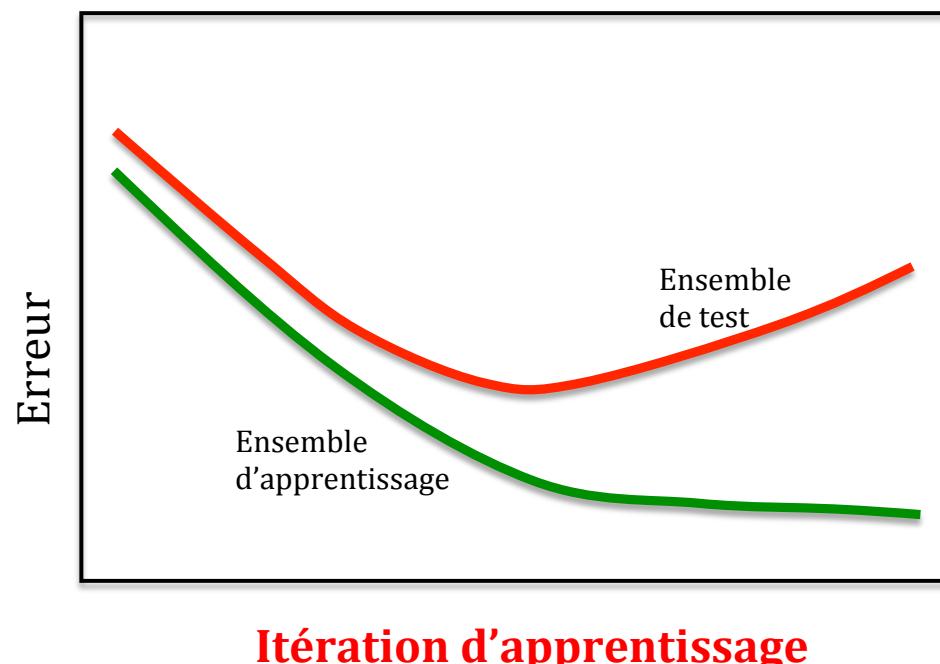
Le compromis entre le biais et la variance

- Exercice: faire une recherche sur les hyperparameters pour sélectionner le modèle le plus performant sur l'ensemble de validation
- Pourriez-vous expliquer cette recherche en termes de ce compromis?



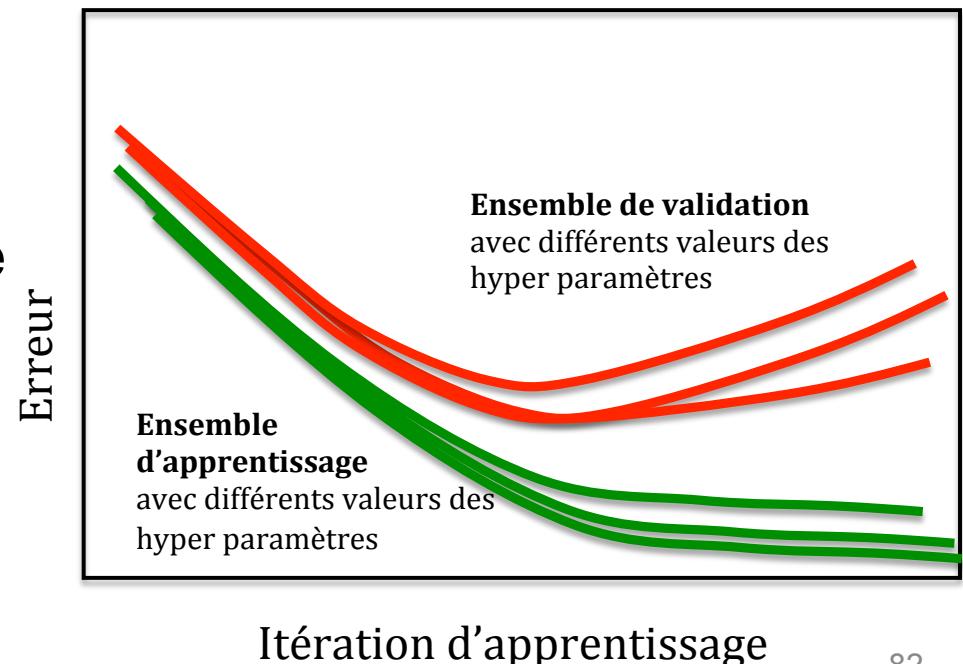
Surapprentissage

- Avec plusieurs itérations d'apprentissage, (ex. utilisant un réseau de neurones), on peut obtenir un modèle trop spécialisée pour notre ensemble d'apprentissage



Stratégies pour minimiser les effets de « surapprentissage »

- Il est typique à utiliser une distribution gaussienne (moyenne zéro) sur les paramètres pour les régressions logistique
- Avec des réseaux neuronales on utilise « early stopping »
- De plus, souvent il existe des hyper paramètres (par exemple la variance d'une gaussienne sur les paramètres),
- donc, en pratique on utilise un troisième ensemble : l'ensemble de validation avec une validation croisée « hold-out »
- Cependant, un but des méthodes plus bayésiens comme BIC est à éviter le besoin pour cette phase



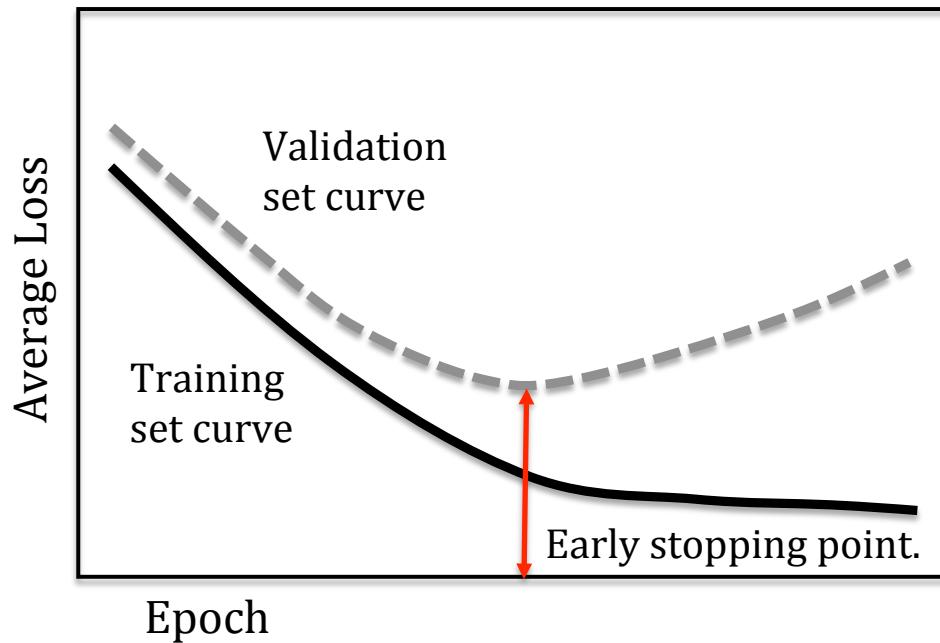
Training and evaluating deep networks

Part I

Early stopping

- Deep learning involves high capacity architectures, which are susceptible to overfitting even when data is plentiful,
- Early stopping is standard practice even when other methods to reduce overfitting are employed, ex. regularization and dropout
- The idea is to monitor learning curves that plot the average loss for the training and validation sets as a function of epoch
- The key is to find the point at which the validation set average loss begins to deteriorate

Early stopping



- In practice the curves above can be more noisy due to the use of stochastic gradient descent
- As such, it is common to keep the history of the validation set curve when looking for the minimum – even if it goes back up it might come back down

Validation sets and hyperparameters

- In deep learning hyperparameters are tuned by identifying what settings lead to best performance on the validation set, using early stopping
- Common hyperparameters include the strength of parameter regularization, but also model complexity in terms of the number of hidden units and layers and their connectivity, the form of activation functions, and parameters of the learning algorithm itself.
- Because of the many choices involved, performance monitoring on validation sets assumes an even more central role than it does with traditional machine learning methods.

Test sets

- *Should be set aside for a truly final evaluation*
- Repeated rounds of experiments using test set data give misleading (ex. optimistic) estimates of performance on fresh data
- For this reason, the research community has come to favor public challenges with hidden test-set labels, a development that has undoubtedly helped gauge progress in the field
- Controversy arises when participants submit multiple entries, and some favor a model where participants submit code to a competition server, so that the test data itself is hidden

Validation sets vs. cross-validation

- The use of a validation set is different from using k -fold cross-validation to evaluate a learning technique or to select hyperparameters.
- Cross-validation involves creating multiple training and testing partitions.
- Datasets for deep learning tend to be so massive that a single large test set adequately represents a model's performance, reducing the need for cross-validation
 - Since training often takes days or weeks, even using GPUs, cross-validation is often impractical anyway.
- If you do use cross validation you need to have an intern validation set *for each fold* to adjust hyperparameters or perform cross validation only using the training set

Validation set data and the ‘end game’

- To obtain the best possible results, one needs to tune hyperparameters, usually with a single validation set extracted from the training set.
- However, there is a dilemma: omitting the validation set from final training can reduce performance in the test.
- It is advantageous to train on the combined training and validation data, but this risks overfitting.
- One solution is to stop training after the same number of epochs that led to the best validation set performance; another is to monitor the average loss over the combined training set and stop when it reaches the level it was at when early stopping was performed using the validation set.
- One can use cross validation within the training set, treating each fold as a different validation set, then train the final model on the entire training data with the identified hyperparameters to perform the final test

Hyperparameter tuning

- A weighted combination of L_2 and L_1 regularization is often used to regularize weights
- Hyperparameters in deep learning are often tuned heuristically by hand, or using grid search
- An alternative is random search, where instead of placing a regular grid over hyperparameter space, probability distributions are specified from which samples are taken
- Another approach is to use machine learning and Bayesian techniques to infer the next hyperparameter configuration to try in a sequence of experimental runs
- Keep in mind even things like the learning rate schedule (discussed below) are forms of hyperparameters and you need to be careful not to tune them on the test set

Mini-batch based stochastic gradient descent (SGD)

- Stochastic gradient descent updates model parameters according to the gradient computed from one example
- The mini-batch variant uses a small subset of the data and bases updates to parameters on the average gradient over the examples in the batch
- This operates just like the regular procedure: initialize the parameters, enter a parameter update loop, and terminate by monitoring a validation set
- Normally these batches are randomly selected disjoint subsets of the training set, perhaps shuffled after each epoch, depending on the time required to do so

Mini-batch based SGD

- Each pass through a set of mini-batches that represent the complete training set is an *epoch*
- Using the empirical risk plus a regularization term as the objective function, updates are

$$\boldsymbol{\theta}^{\text{new}} \leftarrow \boldsymbol{\theta} - \eta_t \left[\frac{1}{B_k} \sum_{i \in I} \left[\frac{\partial}{\partial \boldsymbol{\theta}} L(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \boldsymbol{\theta}} R(\boldsymbol{\theta}) \right]$$

- η_t is the learning rate and may depend on the epoch t
- The batch is represented by a set of indices $I=I(t,k)$ into the original data; the k th batch has B_k examples
- N is the size of the training set
- $L(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i)$ is the loss for example \mathbf{x}_i , label \mathbf{y}_i , params $\boldsymbol{\theta}$
- $R(\boldsymbol{\theta})$ is the regularizer, with weight λ

Mini-batches

- Typically contain two to several hundred examples
 - For large models the choice may be constrained by resources
- Batch size often influences the stability and speed of learning; some sizes work particularly well for a given model and data set.
- Sometimes a search is performed over a set of potential batch sizes to find one that works well, before doing a lengthy optimization.
- The mix of class labels in the batches can influence the result
 - For unbalanced data there may be an advantage in pre-training the model using mini-batches in which the labels are balanced, then fine-tuning the upper layer or layers using the unbalanced label statistics.

Momentum

- As with regular gradient descent, ‘momentum’ can help the optimization escape plateaus in the loss
- Momentum is implemented by computing a moving average: $\Delta\theta = -\eta \nabla_{\theta} L(\theta) + \alpha \Delta\theta^{\text{old}}$
 - where the first term is the current gradient of the loss times a learning rate
 - the second term is the previous update weighted by $\alpha \in [0,1]$
- Since the mini- batch approach operates on a small subset of the data, this averaging can allow information from other recently seen mini-batches to contribute to the current parameter update
- A momentum value of 0.9 is often used as a starting point, but it is common to hand-tune it, the learning rate, and the schedule used to modify the learning rate during the training process

Learning rate schedules

- The learning rate is a critical choice when using mini-batch based stochastic gradient descent.
- Small values such as 0.001 often work well, but it is common to perform a logarithmically spaced search, say in the interval $[10^{-8}, 1]$, followed by a finer grid or binary search.
- The learning rate may be adapted over epochs t to give a learning rate schedule, ex. $\eta_t = \eta_0(1 + \varepsilon t)^{-1}$
- A fixed learning rate is often used in the first few epochs, followed by a decreasing schedule
- Many other options, ex. divide the rate by 10 when the validation error rate ceases to improve

Mini-batch SGD pseudocode

```
 $\theta = \theta_0$  // initialize parameters  
 $\Delta\theta = 0$   
 $t = 0$   
while converged == FALSE  
     $\{I_1, \dots, I_K\} = \text{shuffle}(X)$  // create  $K$  mini - batches  
    for  $k = 1 \dots K$   
         $\mathbf{g} = \frac{1}{B_k} \sum_{i \in I_k} \left[ \frac{\partial}{\partial \theta} L(f(\mathbf{x}_i; \theta), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \theta} R(\theta)$   
         $\Delta\theta \leftarrow -\eta_t \mathbf{g} + \alpha \Delta\theta$   
         $\theta \leftarrow \theta + \Delta\theta$   
    end  
     $t = t + 1$   
end
```

Training and evaluating deep networks

Part II

Dropout

- A form of regularization that randomly deletes units and their connections during training
- Intention: reducing hidden unit co-adaptation & combat over-fitting
- Has been argued it corresponds to sampling from an exponential number of networks with shared parameters & missing connections
- One averages over models at test time by using original network without dropped-out connections, but with scaled-down weights
- If a unit is retained with probability p during training, its outgoing weights are rescaled or multiplied by a factor of p at test time
- By performing dropout a neural network with n units can be made to behave like an ensemble of 2^n smaller networks
- One way to implement it is with a binary mask vector $\mathbf{m}^{(l)}$ for each hidden layer l in the network: the dropped out version of $\mathbf{h}^{(l)}$ masks out units from the original version using element-wise multiplication, $\mathbf{h}_d^{(l)} = \mathbf{h}^{(l)} \odot \mathbf{m}^{(l)}$
- If the activation functions lead to diagonal gradient matrices, the backpropagation update is $\Delta^{(l)} = \mathbf{d}^{(l)} \odot \mathbf{m}^{(l)} \odot (\mathbf{W}^{(l+1)} \Delta^{(l+1)})$.

Batch normalization

- A way of accelerating training for which many studies have found to be important to obtain state-of-the-art results
- Each element of a layer is normalized to zero mean and unit variance based on its statistics within a mini-batch
 - This can change the network's representational power
- Each activation has learned scaling and shifting parameter
- Mini-batch based SGD is modified by calculating the mean μ_j and variance σ_j^2 over the batch for each hidden unit h_j in each layer, then normalize the units, scale them using the learned scaling parameter γ_j and shift them by the learned shifting parameter β_j such that

$$\hat{h}_j \leftarrow \gamma_j \frac{h_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} + \beta_j.$$

- To update the γ_j and β_j one needs to backpropagate the gradient of the loss through these additional parameters

Parameter initialization

- Can be deceptively important!
- Bias terms are often initialized to 0 with no issues
- Weight matrices more problematic, ex.
 - If initialized to all 0s, can be shown that the tanh activation function will yield zero gradients
 - If the weights are all the same, hidden units will produce same gradients and behave the same as each other (wasting params)
- One solution: initialize all elements of weight matrix from uniform distribution over interval $[-b, b]$
- Different methods have been proposed for selecting the value of b , often motivated by the idea that units with more inputs should have smaller weights
- Weight matrices of rectified linear units have been successfully initialized using a zero-mean isotropic Gaussian distribution with standard deviation of 0.01

Unsupervised pre-training

- Idea: model the distribution of unlabeled data using a method that allows the parameters of the learned model to inform or be somehow transferred to the network
- Can be an effective way to both initialize and regularize a feedforward network
- Particularly useful when the volume of labeled data is small relative to the model's capacity.
- The use of activation functions such as rectified linear units (which improve gradient flow in deep networks), along with good parameter initialization techniques, can mitigate the need for sophisticated pre-training methods

Data augmentation

- Can be critical for best results
- As seen in MNIST table, augmenting even a large dataset with transformed data can increase performance
- A simple transformation is simply to jiggle the image
- If the object to be classified can be cropped out of a larger image, random bounding boxes can be placed around it, adding small translations in the vertical and horizontal directions
- Can also use rotations, scale changes, and shearing
- There is a hierarchy of rigid transformations that increase in complexity as parameters are added which can be used

Bibliographic Notes & Further Reading

- Bergstra and Bengio (2012) give empirical and theoretical justification for the use of random search for hyperparameter settings.
- Snoek et al. (2012) propose the use of Bayesian learning methods to infer the next hyperparameter setting to explore, and their Spearmint software package performs Bayesian optimizations of both deep network hyperparameters and general machine learning algorithm hyperparameters.
- Stochastic gradient descent methods go back at least as far as Robbins and Monro (1951).

Bibliographic Notes & Further Reading

- Bottou (2012) is an excellent source of tips and tricks for learning with stochastic gradient descent, while Bengio (2012) gives further practical recommendations for training deep networks.
- Glorot and Bengio (2010) cover various weight matrix initialization heuristics, and how the concepts of fan-in and fan-out can be used to justify them for networks with different kinds of activation functions.
- The origins of dropout and more details about it can be found in Srivastava et al. (2014).
- Ioffe and Szegedy (2015) proposed batch normalization and give more details on its implementation.