

Réseaux de neurones  
IFT 603-712

Réseaux de neurones multicouches

Par  
Pierre-Marc Jodoin

1

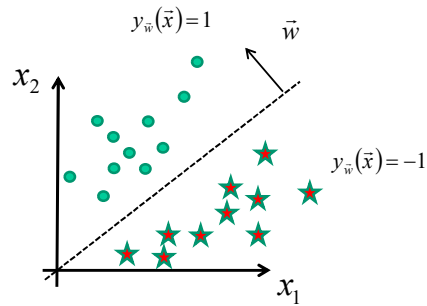
Rappel réseaux de neurones

(Perceptron, régression logistique, SVM)

2

# Séparation linéaire

(2D et 2 classes)



$$y_{\vec{w}}(\vec{x}) = \underset{\text{biais}}{w_0} + \underset{\text{poids}}{w_1 x_1 + w_2 x_2}$$

$$= w_0 + \vec{w}^T \vec{x}$$

$$= \vec{w}'^T \vec{x}'$$

$$y_{\vec{w}}(\vec{x}) = \vec{w}^T \vec{x}$$

Par simplicité

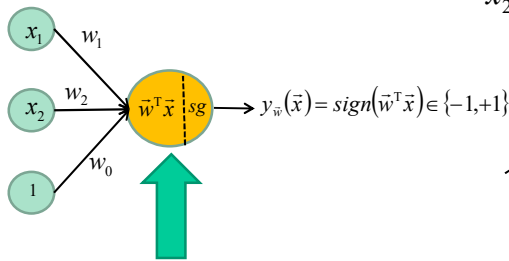
2 grands **avantages**. Une fois l'entraînement terminé,

1. Plus besoin de données d'entraînement
2. Classification est très rapide (**produit scalaire** entre 2 vecteurs)

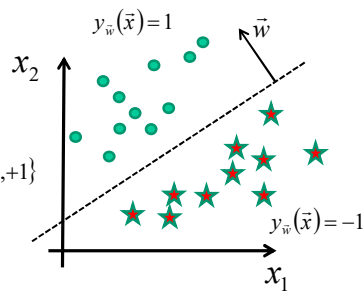
3

# Perceptron

(2D et 2 classes)



Produit scalaire + fonction d'activation



Fonction de coût Perceptron (loss) et gradient

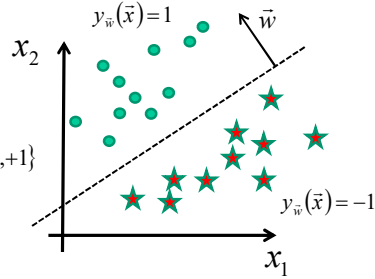
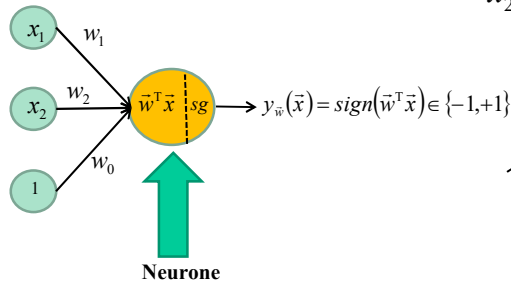
$$E_D(\vec{w}) = \sum_{n=1}^N -t_n \vec{w}^T \vec{x}_n \quad \text{où } M \text{ est l'ensemble des données mal classées}$$

$$\nabla E_D(\vec{w}) = \sum_{\vec{x}_n \in M} -t_n \vec{x}_n$$

4

# Hinge Loss

(2D et 2 classes)



Neurone  
Produit scalaire + fonction d'activation

Fonction de coût SVM (hinge loss) et gradient

$$E_D(\vec{w}) = \sum_{n=1}^N \max(0, 1 - t_n \vec{w}^T \vec{x}_n)$$

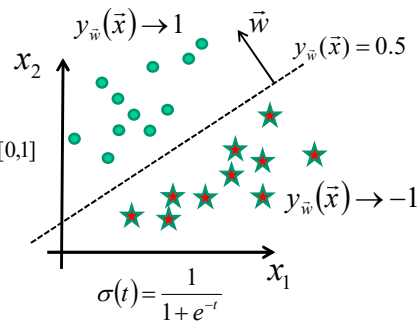
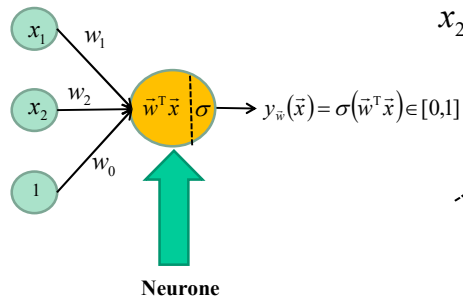
$$\nabla E_D(\vec{w}) = \sum_{\vec{x}_n \in M} -t_n \vec{x}_n$$

5

# Régression logistique

(2D, 2 classes)

Nouvelle fonction d'activation : **sigmoïde logistique**



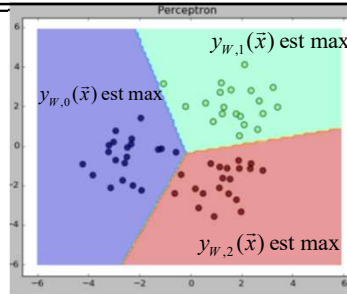
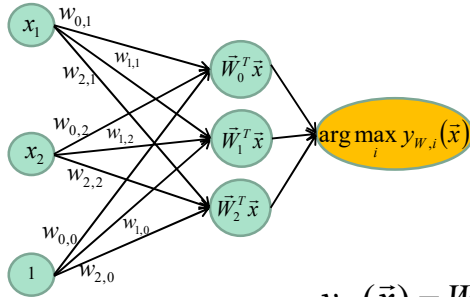
Fonction de coût (loss) et gradient

$$E_D(\vec{W}) = -\sum_{n=1}^N t_n \ln(y_w(\vec{x}_n)) + (1 - t_n) \ln(1 - y_w(\vec{x}_n))$$

$$\nabla E_D(\vec{W}) = \sum_{n=1}^N (y_w(\vec{x}_n) - t_n) \vec{x}_n$$

6

# Perceptron Multiclasse (2D et 3 classes)



$$y_W(\vec{x}) = W^T \vec{x}$$

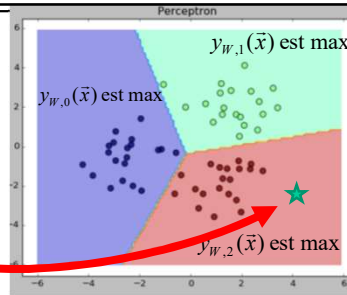
$$y_W(\vec{x}) = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

7

# Perceptron Multiclasse

Exemple

★ (1.1, -2.0)



$$y_W(\vec{x}) = \begin{bmatrix} -2 & -3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 1.1 \\ -2 \end{bmatrix} = \begin{bmatrix} -6.9 \\ -9.6 \\ 8.2 \end{bmatrix} \begin{matrix} \text{Classe 0} \\ \text{Classe 1} \\ \text{Classe 2} \end{matrix}$$

8

## Perceptron Multiclasse

Fonction de coût (**Perceptron loss**)

$$E_D(W) = \sum_{\vec{x}_n \in M} (\vec{W}_j^T \vec{x}_n - \vec{W}_{t_n}^T \vec{x}_n)$$

Somme sur l'ensemble des données mal classées

Score de la mauvaise classe

Score de la bonne classe

$$\nabla E_D(\vec{W}) = \sum_{\vec{x}_n \in M} \vec{x}_n$$

9

## Hinge Multiclasse

Fonction de coût (**Hinge loss**)

$$E_D(W) = \sum_{n=1}^N \sum_j \max(0, 1 + \vec{W}_j^T \vec{x}_n - \vec{W}_{t_n}^T \vec{x}_n)$$

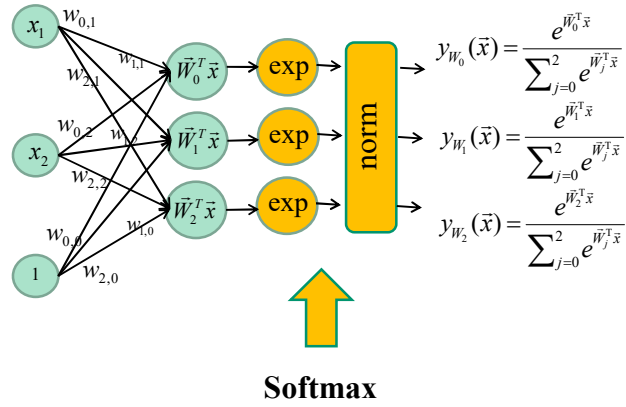
Score de la mauvaise classe

Score de la bonne classe

$$\nabla E_D(\vec{W}) = \sum_{\vec{x}_n \in M} \vec{x}_n$$

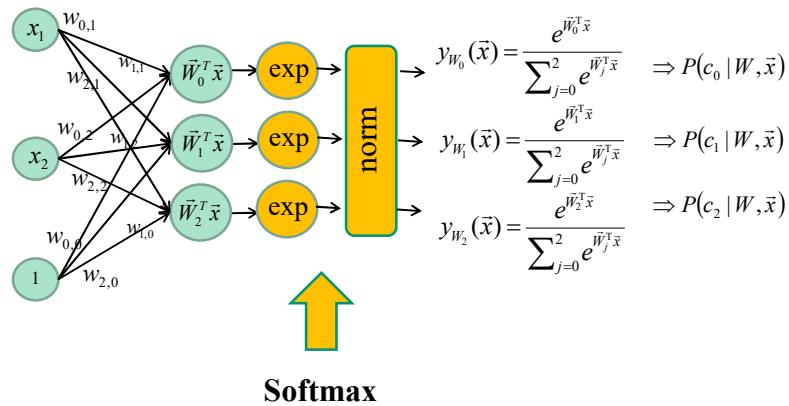
10

# Régression logistique multiclasse













11

# Régression logistique multiclasse



12

## Régression logistique multiclasse

airplane		'airplane' $\Rightarrow t = [1000000000]$
automobile		'automobile' $\Rightarrow t = [0100000000]$
bird		'bird' $\Rightarrow t = [0010000000]$
cat		'cat' $\Rightarrow t = [0001000000]$
deer		'deer' $\Rightarrow t = [0000100000]$
dog		'dog' $\Rightarrow t = [0000010000]$
frog		'frog' $\Rightarrow t = [0000001000]$
horse		'horse' $\Rightarrow t = [0000000100]$
ship		'ship' $\Rightarrow t = [0000000010]$
truck		'truck' $\Rightarrow t = [0000000001]$

Étiquettes de classe : **one-hot vector**

13

## Régression logistique multiclasse

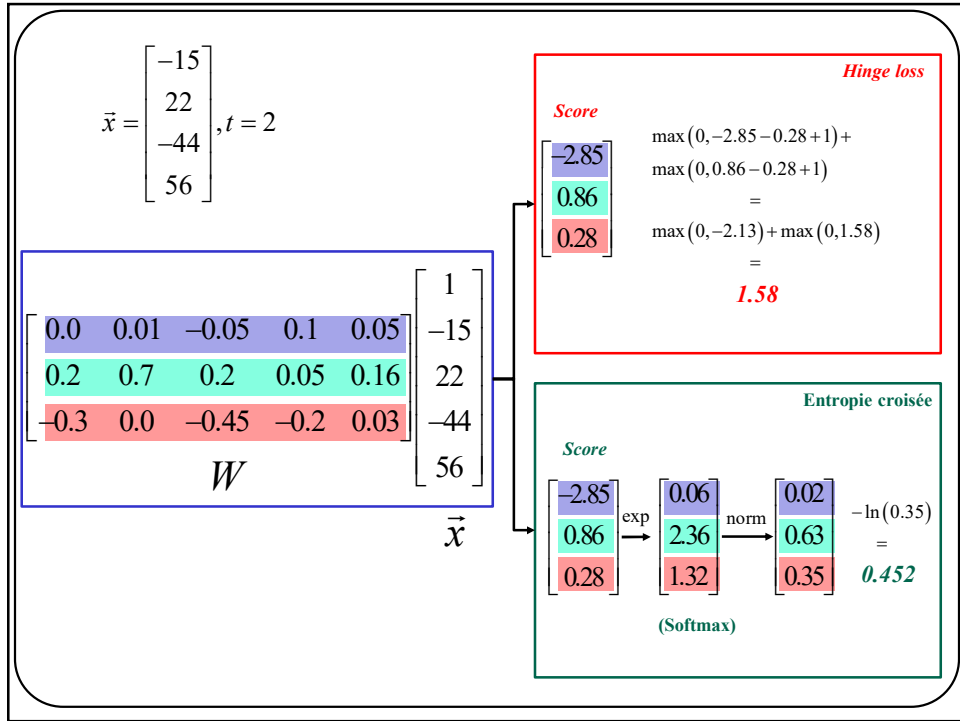
Fonction de coût est une **entropie croisée** (*cross entropy loss*)

$$E_D(\mathbf{W}) = -\sum_{n=1}^N \sum_{k=1}^K t_{kn} \ln y_{W_k}(\vec{x}_n)$$

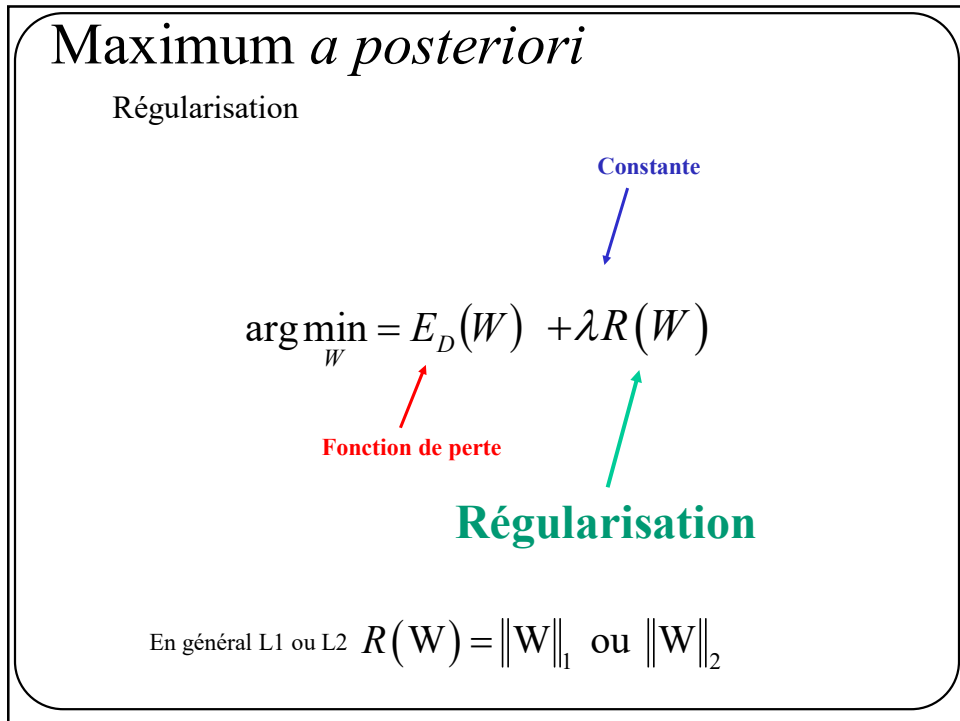
**Devoir 4**

$$\nabla E_D(\mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \vec{x}_n (y_W(\vec{x}_n) - t_{kn})$$

14



15



16



# Optimisation

## Descente de gradient

$$w^{[k+1]} = w^{[k]} - \eta^{[k]} \nabla_w E$$

→ Gradient de la fonction de coût

→ Taux d'apprentissage ou "learning rate".

### Descente de gradient stochastique

```
Initialiser w
k=0
FAIRE k=k+1
  FOR n = 1 to N
     $w = w - \eta^{[k]} \nabla E(\tilde{x}_n)$ 
  JUSQU'À ce que toutes les données
  soient bien classées ou k==MAX_ITER
```

**Devoir 4**

### Optimisation par Batch

```
Initialiser w
k=0
FAIRE k=k+1
   $w = w - \eta^{[k]} \sum_i \nabla E(\tilde{x}_i)$ 
  JUSQU'À ce que toutes les données
  sont bien classées ou k==MAX_ITER
```

Parfois  $\eta^{[k]} = cst / k$

17

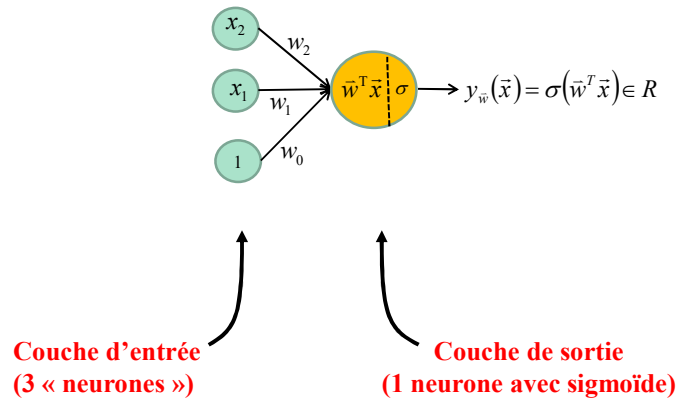
Maintenant, rendons le réseau

**profond**

Maintenant, rendons le réseau

18

## 2D, 2Classes, Régression logistique linéaire

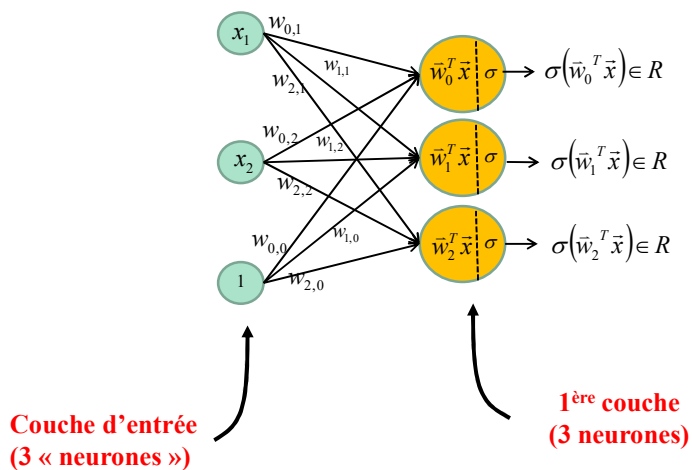


19

19

## 2D, 2Classes, Réseau à 1 couche cachée

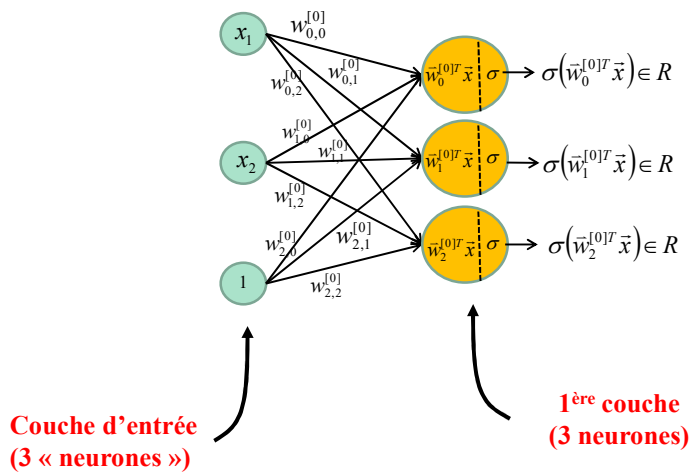
Maintenant, ajoutons arbitrairement 3 neurones



20

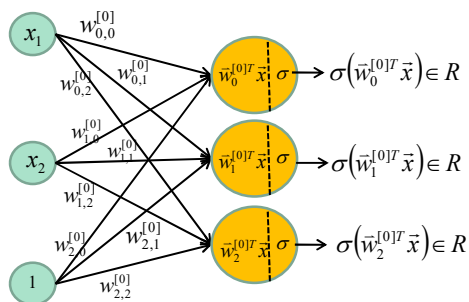
## 2D, 2Classes, Réseau à 1 couche cachée

Puisque les poids sont **entre la couche d'entrée et la première couche** on va les identifier à l'aide de **l'indice [0]**



21

## 2D, 2Classes, Réseau à 1 couche cachée

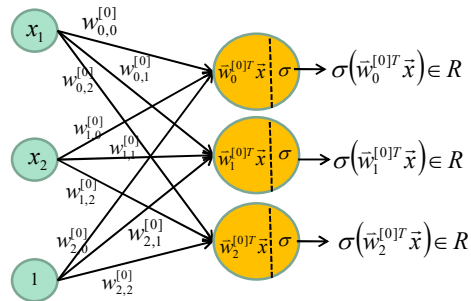


**NOTE:** à la sortie de la première couche, on a **3 réels** calculés ainsi

$$\sigma \left( \begin{bmatrix} w_{0,0}^{[0]} & w_{0,1}^{[0]} & w_{0,2}^{[0]} \\ w_{1,0}^{[0]} & w_{1,1}^{[0]} & w_{1,2}^{[0]} \\ w_{2,0}^{[0]} & w_{2,1}^{[0]} & w_{2,2}^{[0]} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ 1 \end{bmatrix} \right)$$

22

## 2D, 2Classes, Réseau à 1 couche cachée



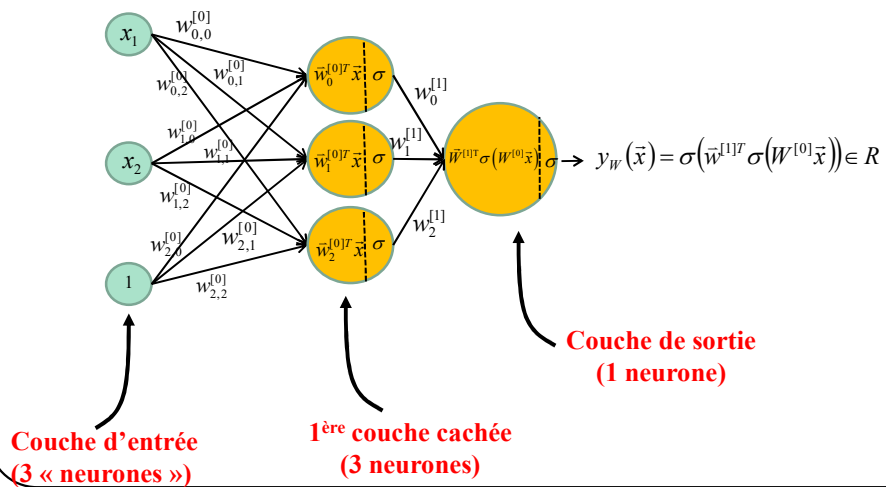
**NOTE:** représentation plus simple de la sortie de la 1<sup>ère</sup> couche (**3 réels**)

$$\sigma(W^{[0]}\vec{x})$$

23

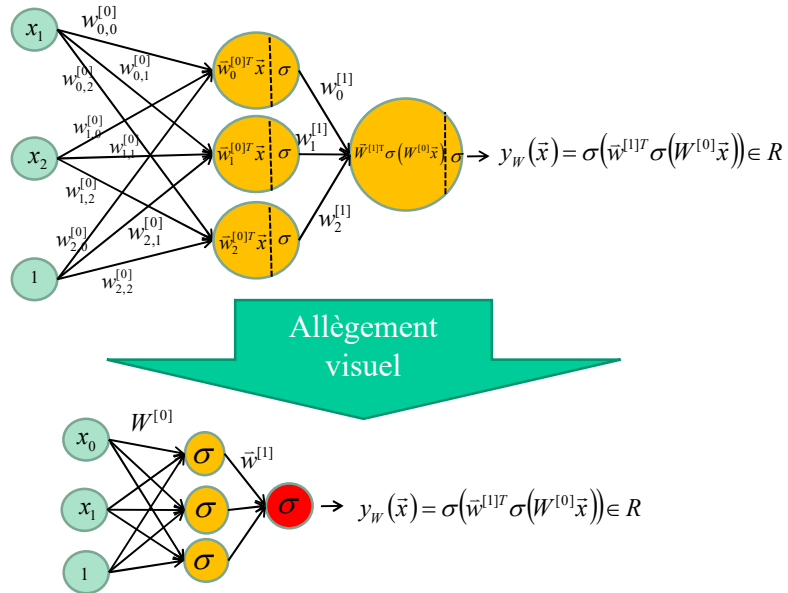
## 2D, 2Classes, Réseau à 1 couche cachée

Si on veut effectuer une **classification 2 classes** via une **régression logistique** (donc **une fonction coût par « entropie croisée »**) on doit ajouter **un neurone de sortie**.



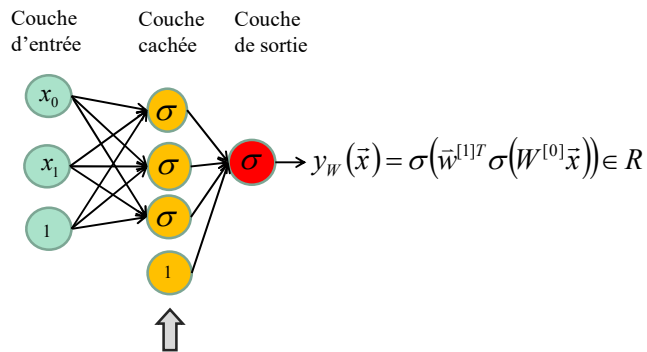
24

## 2D, 2Classes, Réseau à 1 couche cachée



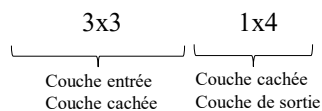
25

## 2D, 2Classes, Réseau à 1 couche cachée



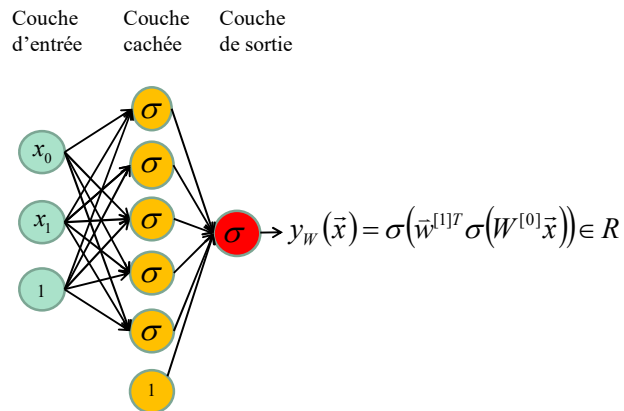
Très souvent, on ajoute un **neurone de biais** à la couche cachée.

Ce réseau possède au total **13 paramètres**



26

## 2D, 2Classes, Réseau à 1 couche cachée



Plus on augmente le nombre de neurones à la couche cachée, plus on **augmente la capacité du système**.

Ce réseau a  $5 \times 3 + 1 \times 6 = 21$  **paramètres**

27

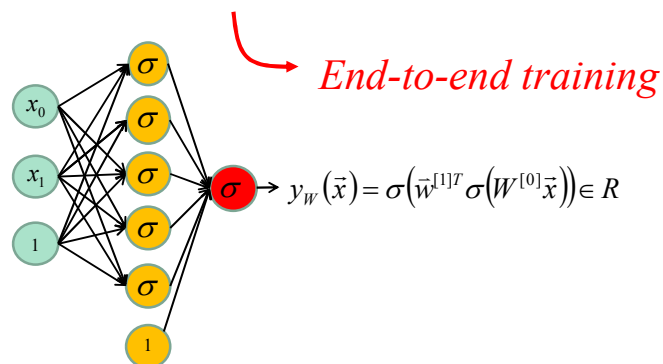
27

## NOTE Importante

Le but de la première couche est de **projeter les données d'entrée** (ici  $\vec{x} \in R^2$ ) vers un espace dimensionnel plus grand (ici  $\sigma(W^{[0]}\vec{x}) \in R^5$ ) là où les **classes sont linéairement séparables**.

Car il ne faut pas oublier que la **couche de sortie** est une **régression logistique linéaire**.

Par conséquent, au lieu de fixer nous même la fonction de base, on laisse le **réseau l'apprendre**.

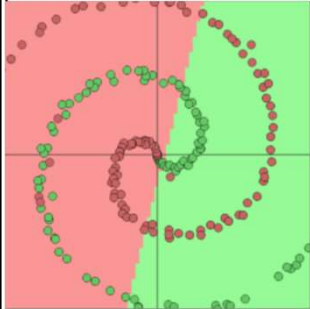


28

28

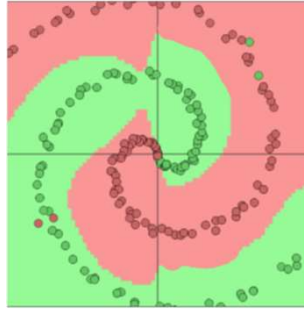
# Nombre de neurones VS Capacity

Aucun neurone caché



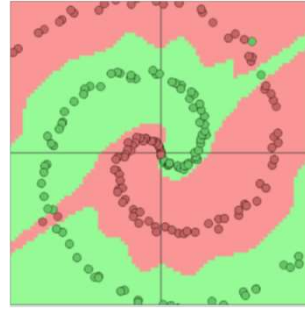
Classification linéaire  
**Underfitting**  
(pas assez de capacité)

12 neurones cachés



Classification non linéaire  
**BON RÉSULTAT**  
(bonne capacité)

60 neurones cachés



Classification non linéaire  
**Over fitting**  
(trop grande capacité)

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

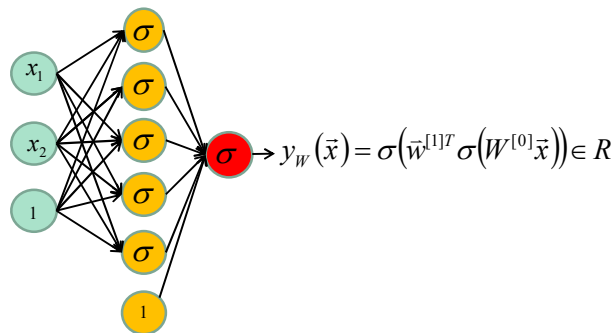
29

# 2D, 2Classes, Réseau à 1 couche cachée

Couche  
d'entrée

Couche  
cachée

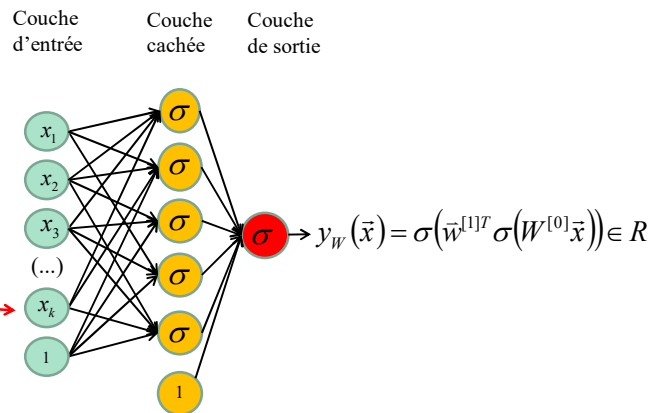
Couche  
de sortie



30

30

## kD, 2Classes, Réseau à 1 couche cachée



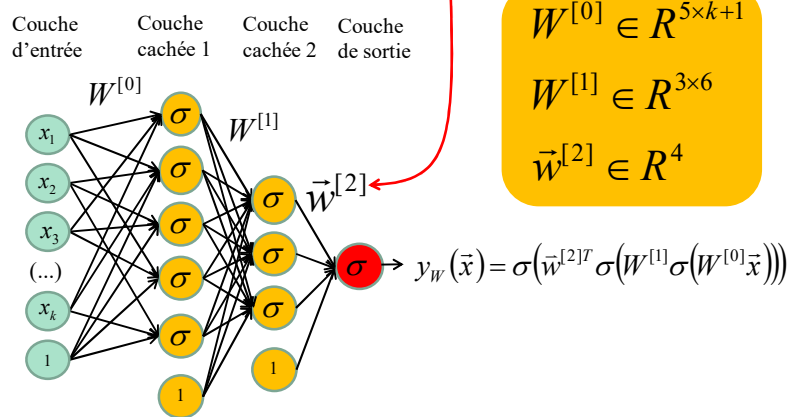
Au peut facilement augmenter la dimensionnalité des données d'entrée. Cela n'a pour effet que **d'augmenter le nombre de colonnes dans  $W^{[0]}$**

Ce réseau a  $5 \times (k+1) + 1 \times 6$  **paramètres**

31

31

## kD, 2Classes, Réseau à 2 couches cachées



En ajoutant une couche cachée, on ajoute une multiplication matricielle

Ce réseau a  $5 \times (k+1) + 6 \times 3 + 1 \times 4$  **paramètres**

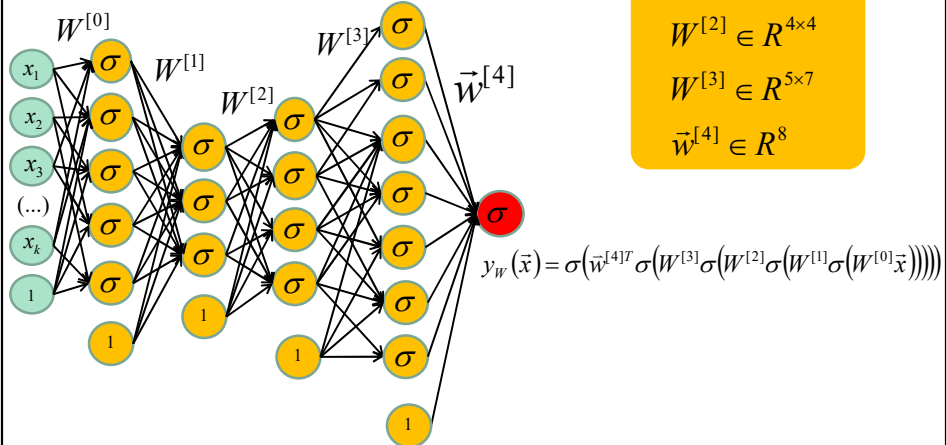
32

32



## kD, 2 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



$$W^{[0]} \in \mathbb{R}^{5 \times k+1}$$

$$W^{[1]} \in \mathbb{R}^{3 \times 6}$$

$$W^{[2]} \in \mathbb{R}^{4 \times 4}$$

$$W^{[3]} \in \mathbb{R}^{5 \times 7}$$

$$\vec{w}^{[4]} \in \mathbb{R}^8$$

$$y_w(\vec{x}) = \sigma(\vec{w}^{[4]T} \sigma(W^{[3]} \sigma(W^{[2]} \sigma(W^{[1]} \sigma(W^{[0]} \vec{x}))))))$$

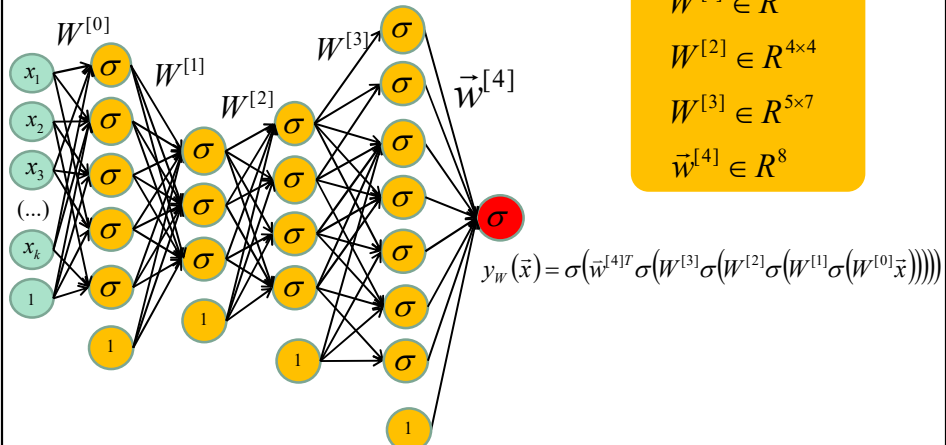
Ce réseau a  $5 \times (k+1) + 6 \times 3 + 4 \times 4 + 7 \times 5 + 1 \times 8$  **paramètres**

33

33

## kD, 2 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



$$W^{[0]} \in \mathbb{R}^{5 \times k+1}$$

$$W^{[1]} \in \mathbb{R}^{3 \times 6}$$

$$W^{[2]} \in \mathbb{R}^{4 \times 4}$$

$$W^{[3]} \in \mathbb{R}^{5 \times 7}$$

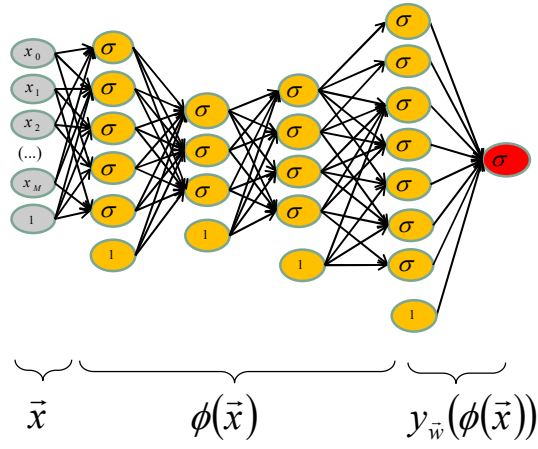
$$\vec{w}^{[4]} \in \mathbb{R}^8$$

$$y_w(\vec{x}) = \sigma(\vec{w}^{[4]T} \sigma(W^{[3]} \sigma(W^{[2]} \sigma(W^{[1]} \sigma(W^{[0]} \vec{x}))))))$$

NOTE : plus on augmente le nombre de couches, plus le réseau devient **profond** et plus on **augmente la capacité** du réseau.

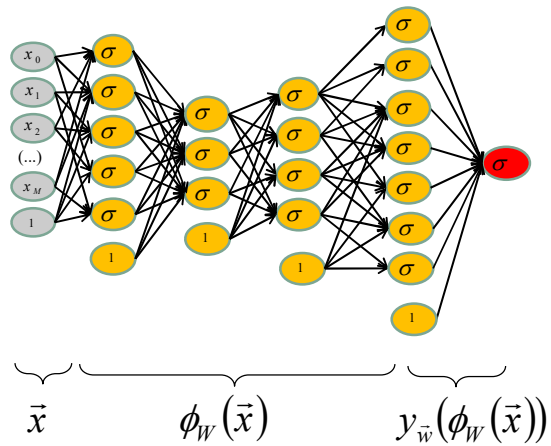
34

Réseau de neurones multicouches = apprendre la fonction de base

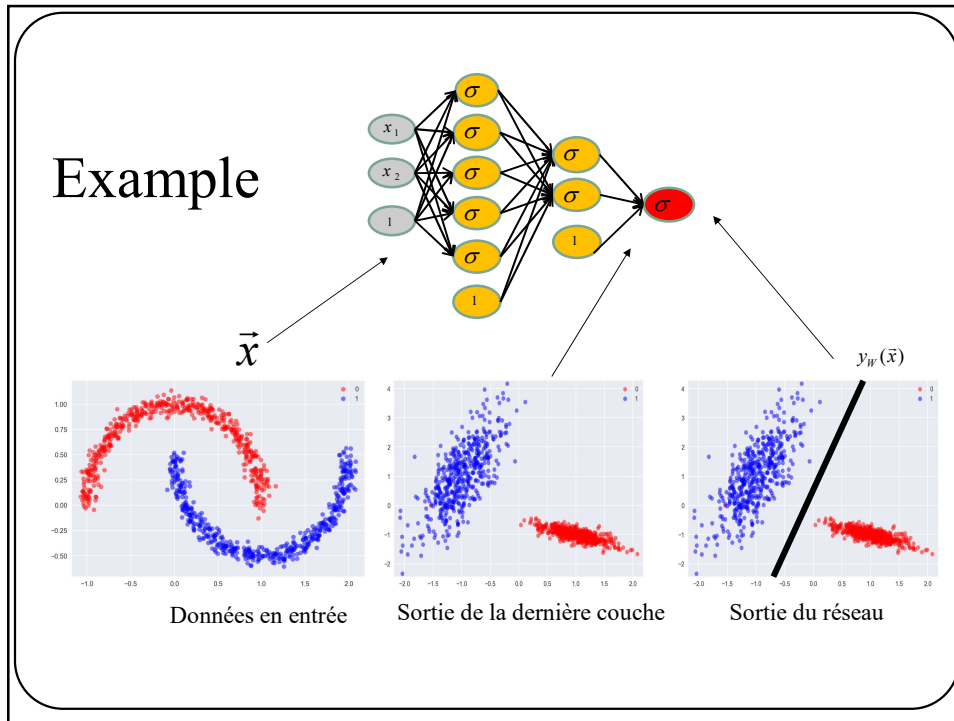


35

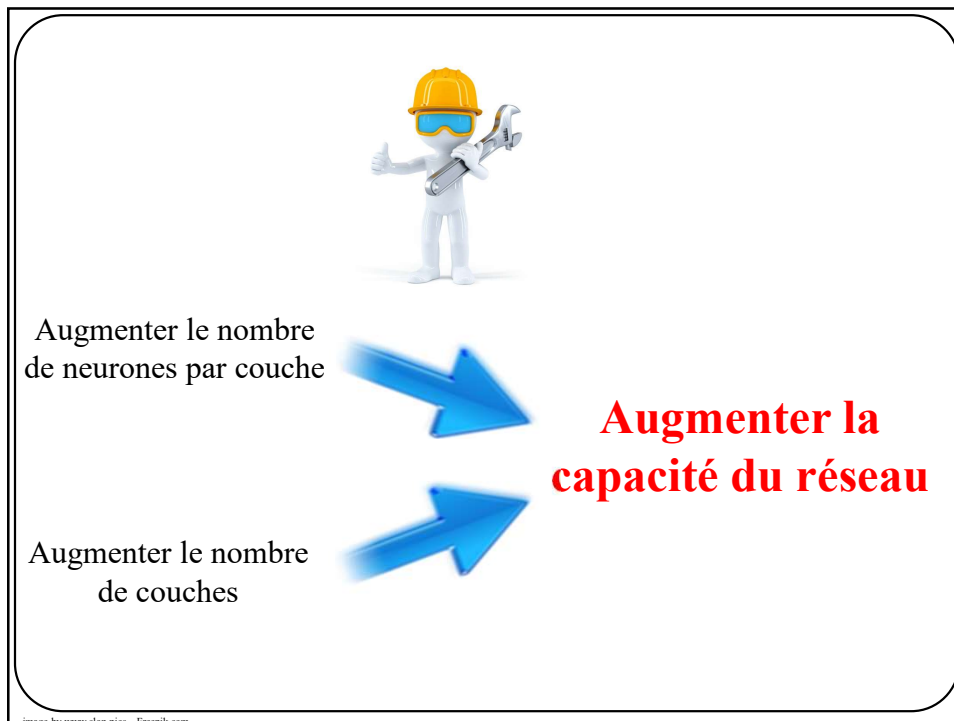
Réseau de neurones multicouches = apprendre la fonction de base



36



37



38



Augmenter la capacité d'un réseau peut entraîner du **sur-apprentissage**

39

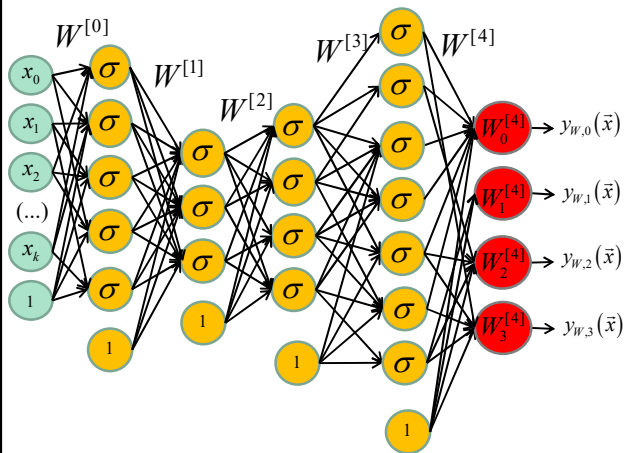


Lorsqu'un réseau doit prédire **plus de 2 classes**, on lui assigne **K neurones de sortie**, une par classe.

40

## kD, 4 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



$$W^{[0]} \in \mathbb{R}^{5 \times k+1}$$

$$W^{[1]} \in \mathbb{R}^{3 \times 6}$$

$$W^{[2]} \in \mathbb{R}^{4 \times 4}$$

$$W^{[3]} \in \mathbb{R}^{5 \times 7}$$

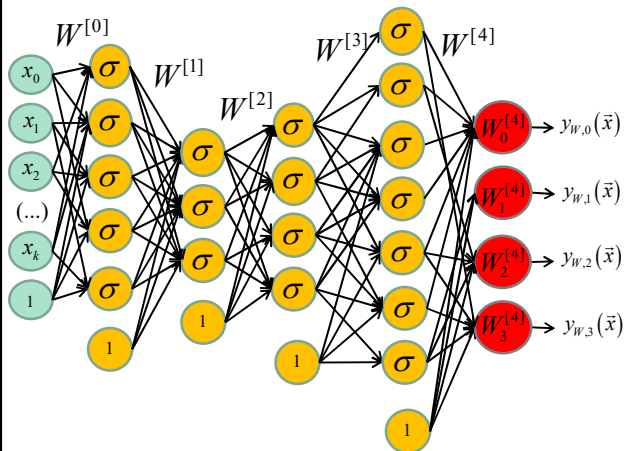
$$W^{[4]} \in \mathbb{R}^{4 \times 8}$$

$$y_w(\vec{x}) = W^{[4]} \sigma \left( W^{[3]} \sigma \left( W^{[2]} \sigma \left( W^{[1]} \sigma \left( W^{[0]} \vec{x} \right) \right) \right) \right)$$

41

## kD, 4 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



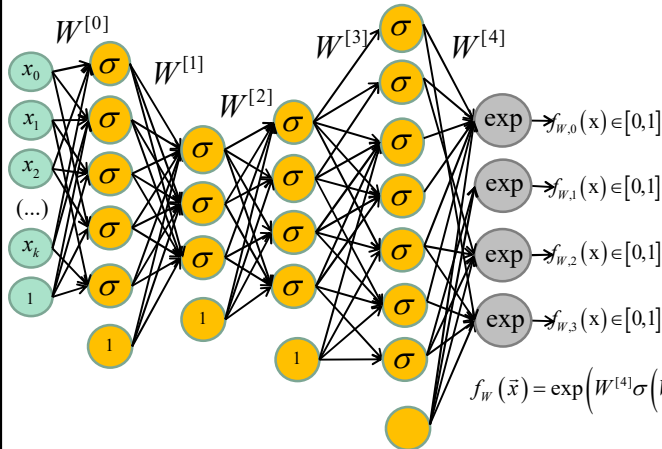
*Hinge loss*

$$y_w(\vec{x}) = W^{[4]} \sigma \left( W^{[3]} \sigma \left( W^{[2]} \sigma \left( W^{[1]} \sigma \left( W^{[0]} \vec{x} \right) \right) \right) \right)$$

42

## kD, 4 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



$$W^{[0]} \in \mathbb{R}^{5 \times k+1}$$

$$W^{[1]} \in \mathbb{R}^{3 \times 6}$$

$$W^{[2]} \in \mathbb{R}^{4 \times 4}$$

$$W^{[3]} \in \mathbb{R}^{5 \times 7}$$

$$W^{[4]} \in \mathbb{R}^{4 \times 8}$$

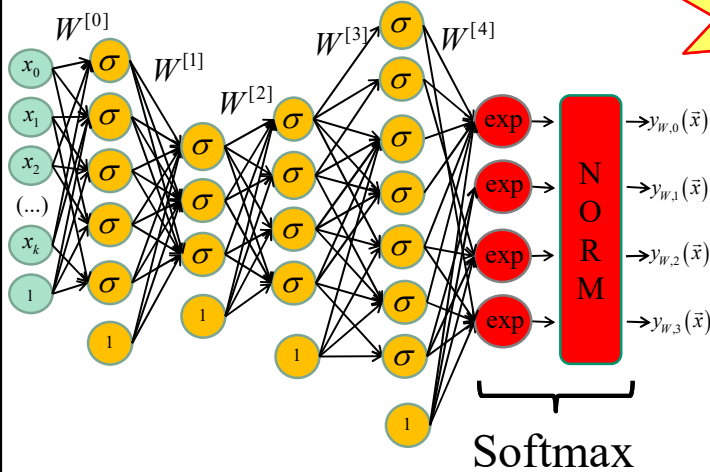
$$f_W(\vec{x}) = \exp\left(W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

$$\text{Softmax } y_{W,i}(\vec{x}) = \frac{f_{W,i}}{\sum_k f_{W,k}}$$

43

## kD, 4 Classes, Réseau à 4 couches cachées

Couche d'entrée    Couche cachée 1    Couche cachée 2    Couche cachée 3    Couche cachée 4    Couche de sortie



**Entropie Croisée**

**Softmax**

$$y_{W,i}(\vec{x}) = \frac{f_{W,i}}{\sum_k f_{W,k}}$$

$$y_W(\vec{x}) = \text{softmax}\left(W^{[4]}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

44

# Simulation

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

45

## Comment faire une prédiction?

Ex.: faire transiter un signal de **l'entrée à la sortie**  
d'un réseau à **3 couches cachées**

```
import numpy as np

def sigmoid(x):
    return 1.0 / (1.0+np.exp(-x))

x = np.insert(x,0,1) # Ajouter biais

H1 = sigmoid(np.dot(W0,x))
H1 = np.insert(H1,0,1) # Ajouter biais } Couche 1

H2 = sigmoid(np.dot(W1,H1))
H2 = np.insert(H2,0,1) # Ajouter biais } Couche 2

H2 = sigmoid(np.dot(W2,H1))
H2 = np.insert(H2,0,1) # Ajouter biais } Couche 3

y_pred = np.dot(W3,H2) } Couche sortie
```

*Forward pass*

46

## Comment optimiser les paramètres?

**0-** Partant de

$$W = \arg \min_W E_D(W) + \lambda R(W)$$

Trouver une fonction de régularisation. En général

$$R(W) = \|W\|_1 \text{ ou } \|W\|_2$$

47

47

## Comment optimiser les paramètres?

**1-** Trouver une loss  $E_D(W)$  comme par exemple

**Hinge loss**

**Entropie croisée (*cross entropy*)**



N'oubliez pas d'ajuster la sortie du réseau en fonction de la loss que vous aurez choisi.

***cross entropy => Softmax***

48



## Comment optimiser les paramètres?

**2-** Calculer le gradient de la loss par rapport à chaque paramètre

$$\frac{\partial (E_D(W) + \lambda R(W))}{\partial w_{a,b}^{[c]}}$$

et lancer un algorithme de descente de gradient pour mettre à jour les paramètres.

$$w_{a,b}^{[c]} = w_{a,b}^{[c]} - \eta \frac{\partial (E_D(W) + \lambda R(W))}{\partial w_{a,b}^{[c]}}$$

49

49

## Comment optimiser les paramètres?

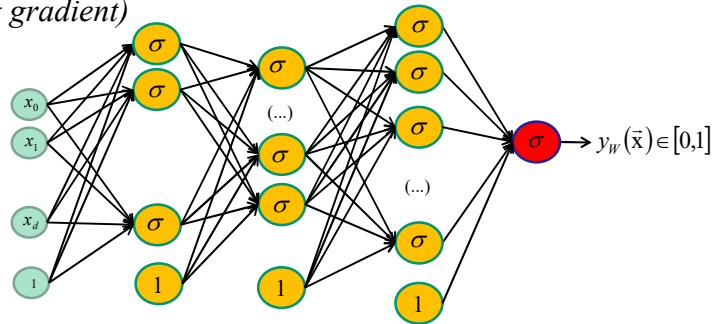
$$\frac{\partial (E_D(W) + \lambda R(W))}{\partial w_{a,b}^{[c]}} \Rightarrow \text{calculé à l'aide d'une rétropropagation}$$

50

50

## Disparition du gradient

(vanishing gradient)



Malheureusement, l'entraînement d'un **réseau profond** avec **rétro-propagation** et des fonctions d'activations **sigmoïdales** entraîne des problèmes de

**disparition du gradient**

Démonstration au  
tableau

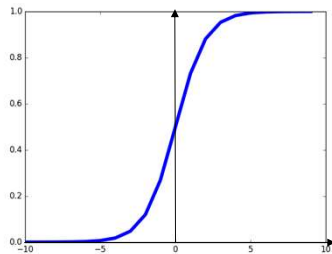
51

51

On résoud le problème de la  
disparition du gradient à l'aide  
**d'autres fonctions d'activations**

52

## Fonction d'activation



**Sigmoïde**

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

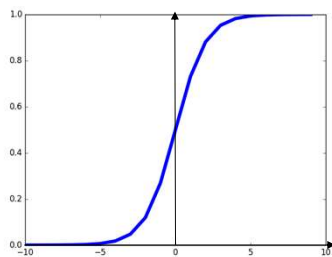
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 3 Problèmes :

- Un neurone saturé a pour effet de « **tuer** » les **gradients**

53

## Fonction d'activation



**Sigmoïde**

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

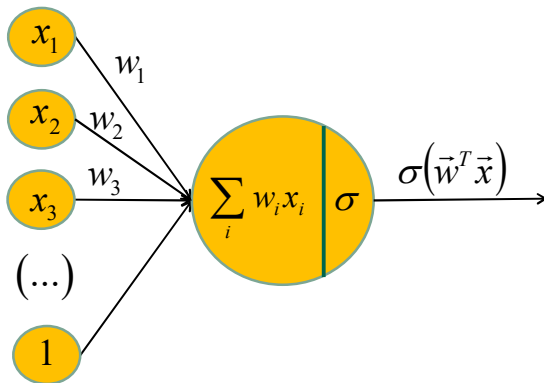
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 3 Problèmes :

- Un neurone saturé a pour effet de « **tuer** » les **gradients**
- Sortie d'une sigmoïde n'est **pas centrée à zéro**.

54

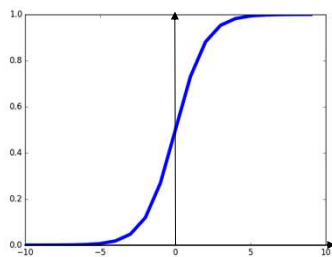
Qu'arrive-t-il lorsque le vecteur d'entrée  $\vec{x}$  d'un neurone est toujours positif?



Le gradient par rapport à  $\vec{w}$  est ... **Positif? Négatif?**

55

## Fonction d'activation



**Sigmoïde**

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

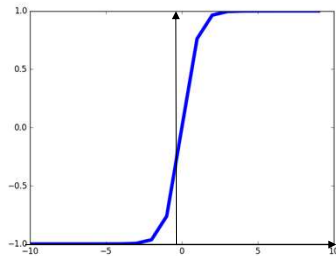
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 3 Problèmes :

- Un neurone saturé a pour effet de « **tuer** » les **gradients**
- Sortie d'une sigmoïde n'est **pas centrée à zéro**.
- $\exp()$  est **coûteux** lorsque le nombre de neurones est élevé.

56

## Fonction d'activation



**Tanh(x)**

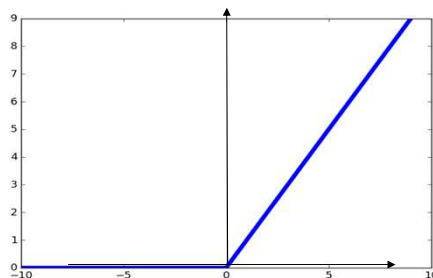
- Ramène les valeurs entre -1 et 1
- **Sortie centrée à zéro** 😊
- **Disparition du gradient** lorsque la fonction sature ☹️

[LeCun et al., 1991]

57

## Fonction d'activation

$$\text{ReLU}(x) = \max(0, x)$$



**ReLU(x)**

(Rectified Linear Unit)

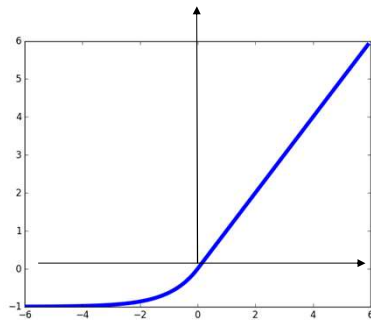
- Aucune **saturation** 😊
- Super **rapide** 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- Sortie **non centrée à zéro** ☹️
- **Un inconvénient** : qu'arrive-t-il au gradient lorsque  $x < 0$ ? ☹️

[Krizhevsky et al., 2012]

58

## Fonction d'activation

$$\text{ELU}(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha(e^x - 1) & \text{sinon} \end{cases}$$



**ELU(x)**  
(Exponential Linear Unit)

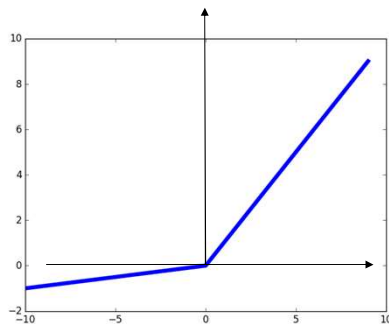
- Tous les avantages de **ReLU** 😊
- Sortie plus « centrée à zéro » 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- **Gradients meurent plus lentement** 😊
- $\exp()$  est **coûteux** 😞

[Clevert et al., 2015]

59

## Fonction d'activation

$$\text{LReLU}(x) = \max(0.01x, x)$$



**Leaky ReLU(x)**

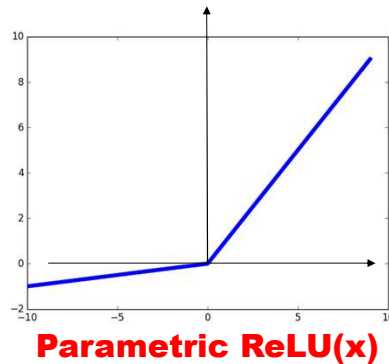
- Aucune **saturation** 😊
- Super **rapide** 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- **Gradients ne meurent pas** 😊
- 0.01 est un **hyperparamètre** 😊

[Mass et al., 2013]  
[He et al., 2015]

60

## Fonction d'activation

$$\text{PReLU}(x) = \max(\alpha x, x)$$



- Aucune **saturation** 😊
- Super **rapide** 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- **Gradients ne meurent pas** 😊
- $\alpha$  **appris** lors de la rétro-propagation 😊

[Mass et al., 2013]  
[He et al., 2015]

61

## En pratique

- Par défaut, le gens utilisent **ReLU**.
- Essayez **Leaky ReLU / PReLU / ELU**
- Essayez **tanh** mais n'attendez-vous pas à grand chose
- **Ne pas utiliser de sigmoïde** sauf à la sortie d'un réseau 2 classes.

62

# Les bonnes pratiques

63

## Optimisation

### Descente de gradient

$$\mathbf{w}^{[k+1]} = \mathbf{w}^{[k]} - \eta^{[k]} \nabla E$$

→ Gradient de la fonction de coût

→ Taux d'apprentissage ou "learning rate".

#### Descente de gradient stochastique

```
Initialiser w
k=0
FAIRE k=k+1
  FOR n = 1 to N
     $\mathbf{w} = \mathbf{w} - \eta^{[k]} \nabla E(\tilde{x}_n)$ 
  JUSQU'À ce que toutes les données
  sont bien classées ou k==MAX_ITER
```

#### Optimisation par Batch

```
Initialiser w
k=0
FAIRE k=k+1
   $\mathbf{w} = \mathbf{w} - \eta^{[k]} \sum_i \nabla E(\tilde{x}_i)$ 
  JUSQU'À ce que toutes les données
  sont bien classées ou k==MAX_ITER
```

Parfois  $\eta^{[k]} = cst / k$

64



# Optimisation

## Descente de gradient

$$\mathbf{w}^{[k+1]} = \mathbf{w}^{[k]} - \eta^{[k]} \nabla E$$

→ Gradient de la fonction de coût

→ Taux d'apprentissage ou "learning rate".

### Optimisation par **mini-batch**

Initialiser  $\mathbf{w}$   
 $k=0$   
FAIRE  $k=k+1$

FAIRE  $n=0$  à  $N$  par sauts de *MBS* /\*Mini-batch size\*/

$$\mathbf{w} = \mathbf{w} - \eta^{[k]} \sum_{i=n}^{n+MBS} \nabla E(\tilde{x}_i)$$

} **Itération**

JUSQU'À ce que toutes les données sont bien classées ou  
 $k=MAX\_ITER$

65

# Optimisation

## Descente de gradient

$$\mathbf{w}^{[k+1]} = \mathbf{w}^{[k]} - \eta^{[k]} \nabla E$$

→ Gradient de la fonction de coût

→ Taux d'apprentissage ou "learning rate".

### Optimisation par **mini-batch**

Initialiser  $\mathbf{w}$   
 $k=0$   
FAIRE  $k=k+1$

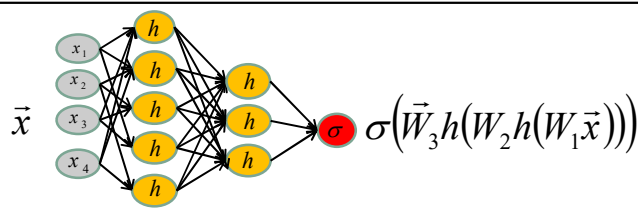
FAIRE  $n=0$  à  $N$  par sauts de *MBS* /\*Mini-batch size\*/

$$\mathbf{w} = \mathbf{w} - \eta^{[k]} \sum_{i=n}^{n+MBS} \nabla E(\tilde{x}_i)$$

} **Epoch**

JUSQU'À ce que toutes les données sont bien classées ou  
 $k=MAX\_ITER$

66

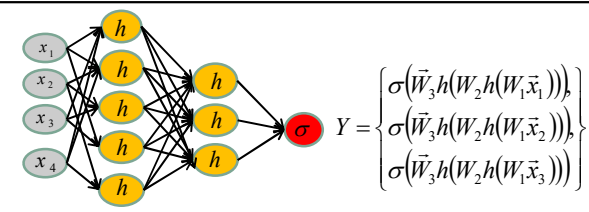


$\vec{x}$   $\sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x})))$

Propagation avant pour une donnée (7 étapes)

$\vec{x}$	$\in \mathbb{R}^4$
$W_1 \vec{x}$	$\in \mathbb{R}^5$
$h(W_1 \vec{x})$	$\in \mathbb{R}^{5+1}$
$W_2 h(W_1 \vec{x})$	$\in \mathbb{R}^3$
$h(W_2 h(W_1 \vec{x}))$	$\in \mathbb{R}^{3+1}$
$\vec{W}_3 (h(W_2 h(W_1 \vec{x})))$	$\in \mathbb{R}$
$\sigma(\vec{W}_3 (h(W_2 h(W_1 \vec{x}))))$	$\in \mathbb{R}$

67



$X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3\}$   $Y = \left\{ \begin{array}{l} \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_1))) \\ \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_2))) \\ \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_3))) \end{array} \right\}$

Propagation avant pour une batch de 3 données (7 étapes)

POUR  $i$  allant de 1 à 3

$\vec{x}_i = X[i]$	$\in \mathbb{R}^{M+1}$
$W_1 \vec{x}$	$\in \mathbb{R}^5$
$h(W_1 \vec{x})$	$\in \mathbb{R}^{5+1}$
$W_2 h(W_1 \vec{x})$	$\in \mathbb{R}^3$
$h(W_2 h(W_1 \vec{x}))$	$\in \mathbb{R}^{3+1}$
$\vec{W}_3 (h(W_2 h(W_1 \vec{x})))$	$\in \mathbb{R}$
$Y[i] = \sigma(\vec{W}_3 (h(W_2 h(W_1 \vec{x}))))$	$\in \mathbb{R}$

Solution  
naïve et peu  
efficace

TP4

68

## Solution

Il est plus efficace d'effectuer UNE multiplication matricielle que **PLUSIEURS produits scalaires** (exemple de la 6<sup>e</sup> étape)

$$\begin{array}{l}
 (w_1 \ w_2 \ w_3) \begin{pmatrix} a \\ b \\ c \end{pmatrix} = (w_1 a + w_2 b + w_3 c) \\
 (w_1 \ w_2 \ w_3) \begin{pmatrix} d \\ e \\ f \end{pmatrix} = (w_1 d + w_2 e + w_3 f) \\
 (w_1 \ w_2 \ w_3) \begin{pmatrix} g \\ h \\ i \end{pmatrix} = (w_1 g + w_2 h + w_3 i)
 \end{array}$$

$$\begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix} = Y$$

TROIS  
produits  
scalaires

69

## Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que PLUSIEURS produits scalaires (exemple de la 6<sup>e</sup> étape)

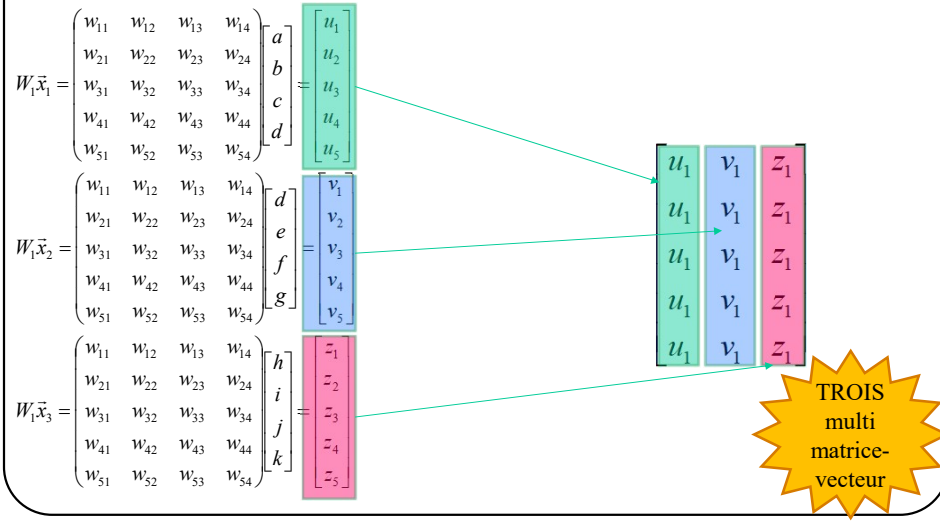
$$[w_1 \ w_2 \ w_3] \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} = \begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix} = Y$$

UNE  
multiplication  
matricielle

70

# Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que **PLUSIEURS matrice-vecteur** (exemple de la 1<sup>e</sup> étape, batch de 3)



71

# Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que **PLUSIEURS matrice-vecteur** (exemple de la 1<sup>e</sup> étape)

$$W_1 X = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} a & d & h \\ b & e & i \\ c & f & j \\ d & g & k \end{bmatrix} = \begin{bmatrix} u_1 & v_1 & z_1 \\ u_2 & v_2 & z_2 \\ u_3 & v_3 & z_3 \\ u_4 & v_4 & z_4 \\ u_5 & v_5 & z_5 \end{bmatrix}$$



72

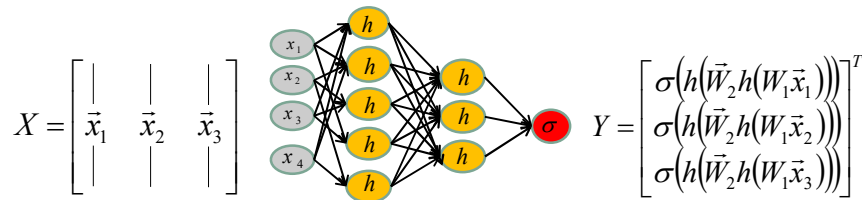
# Vectorisation de la propagation avant

En résumé, lorsqu'on propage une « batch »

Au niveau neuronal	Multi. Vecteur-Matrice	$\vec{W}X = [w_1 \ w_2 \ w_3] \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$
--------------------	---------------------------	--

Au niveau de la couche	Multi. Matrice-Matrice	$WX = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} a & d & h \\ b & e & i \\ c & f & j \\ d & g & k \end{bmatrix}$
------------------------	---------------------------	---

73



Vectoriser la rétropropagation

74

## Vectoriser la rétropropagation

Exemple simple pour **un neurone et une batch de 3**

$$\begin{matrix} [w_1 & w_2 & w_3] \\ \vec{W} \end{matrix} \begin{matrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \\ X \end{matrix} = \begin{matrix} \begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix}^T \\ Y \end{matrix}$$

En supposant qu'on connaît le gradient pour les 3 éléments de Y provenant de sortie du réseau, comment faire pour propager le gradient vers W?

75

## Vectoriser la rétropropagation

Exemple simple pour **1 neurone et une batch de 3**

$$\begin{matrix} [w_1 & w_2 & w_3] \\ W \end{matrix} \begin{matrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \\ X \end{matrix} = \begin{matrix} \begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix}^T \\ Y \end{matrix}$$

Rappelons que l'objectif est de faire une **descente de gradient**, i.e.

$$w_1 \leftarrow w_1 - \eta \frac{\partial E}{\partial w_1} \quad w_2 \leftarrow w_2 - \eta \frac{\partial E}{\partial w_2} \quad w_3 \leftarrow w_3 - \eta \frac{\partial E}{\partial w_3}$$

76

$$\underset{W}{\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}} \underset{X}{\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}} = \underset{Y}{\begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix}}^T$$

Concentrons-nous sur  $w_1$

$$w_1 \leftarrow w_1 - \eta \frac{\partial E}{\partial w_1}$$

$$w_1 \leftarrow w_1 - \eta \frac{\partial E^T}{\partial Y} \frac{\partial Y}{\partial w_1} \quad (\text{par propriété de la dérivée en chaîne})$$

$$w_1 \leftarrow w_1 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} a \\ d \\ g \end{bmatrix} \quad (\text{provient de la rétro-propagation})$$

$$w_1 \leftarrow w_1 - \eta \left( \frac{\partial E_1}{\partial Y} a + \frac{\partial E_2}{\partial Y} b + \frac{\partial E_3}{\partial Y} c \right)$$

77

$$\underset{W}{\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}} \underset{X}{\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}} = \underset{Y}{\begin{pmatrix} w_1 a + w_2 b + w_3 c \\ w_1 d + w_2 e + w_3 f \\ w_1 g + w_2 h + w_3 i \end{pmatrix}}^T$$

Et pour tous les poids

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}^T \leftarrow \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}^T - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$\bar{W}^T \leftarrow \bar{W}^T - \eta \frac{\partial \bar{E}}{\partial Y} \begin{bmatrix} \partial Y_1 / \partial w_1 & \partial Y_1 / \partial w_2 & \partial Y_1 / \partial w_3 \\ \partial Y_2 / \partial w_1 & \partial Y_2 / \partial w_2 & \partial Y_2 / \partial w_3 \\ \partial Y_3 / \partial w_1 & \partial Y_3 / \partial w_2 & \partial Y_3 / \partial w_3 \end{bmatrix}$$

$$\bar{W}^T \leftarrow \bar{W}^T - \eta \frac{\partial \bar{E}}{\partial Y} \frac{\partial Y}{\partial \bar{W}}$$

Matrice jacobienne

78

$$\begin{matrix}
 w_{11} & w_{12} & w_{13} & w_{14} \\
 w_{21} & w_{22} & w_{23} & w_{24} \\
 w_{31} & w_{32} & w_{33} & w_{34} \\
 w_{41} & w_{42} & w_{43} & w_{44} \\
 w_{51} & w_{52} & w_{53} & w_{54}
 \end{matrix}
 \begin{bmatrix}
 a & d & h \\
 b & e & i \\
 c & f & j \\
 d & g & k
 \end{bmatrix}
 =
 \begin{bmatrix}
 u_1 & v_1 & z_1 \\
 u_2 & v_2 & z_2 \\
 u_3 & v_3 & z_3 \\
 u_4 & v_4 & z_4 \\
 u_5 & v_5 & z_5
 \end{bmatrix}$$

$$\mathbf{W} \quad \mathbf{X} \quad \mathbf{Y}$$

Même chose pour **1 couche et un batch de 3**

$$\mathbf{W} \leftarrow \mathbf{W}^T - \eta \frac{\partial E}{\partial \mathbf{Y}}
 \begin{bmatrix}
 \frac{\partial E}{\partial Y_{11}} & \frac{\partial E}{\partial Y_{12}} & \frac{\partial E}{\partial Y_{13}} \\
 \frac{\partial E}{\partial Y_{21}} & \frac{\partial E}{\partial Y_{22}} & \frac{\partial E}{\partial Y_{23}} \\
 \frac{\partial E}{\partial Y_{31}} & \frac{\partial E}{\partial Y_{32}} & \frac{\partial E}{\partial Y_{33}} \\
 \frac{\partial E}{\partial Y_{41}} & \frac{\partial E}{\partial Y_{42}} & \frac{\partial E}{\partial Y_{43}} \\
 \frac{\partial E}{\partial Y_{51}} & \frac{\partial E}{\partial Y_{52}} & \frac{\partial E}{\partial Y_{53}}
 \end{bmatrix}
 \begin{bmatrix}
 a & b & c & d \\
 d & e & f & g \\
 h & i & j & k
 \end{bmatrix}$$

$$\mathbf{W}^T \leftarrow \mathbf{W}^T - \eta \frac{\partial E}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{W}}$$

79

## Vectorisation de la rétro-propagation

En résumé, lorsqu'on rétro-propage un gradient d'une batch

<p>Au niveau <b>neuronal</b></p>	<p>Multi. <b>Vecteur-Matrice</b></p>	$  \bar{\mathbf{W}}^T \leftarrow \bar{\mathbf{W}}^T - \eta \frac{\partial \bar{E}}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \bar{\mathbf{W}}}  $ $  \bar{\mathbf{W}}^T \leftarrow \bar{\mathbf{W}}^T - \eta \frac{\partial \bar{E}}{\partial \mathbf{Y}} \mathbf{X}^T  $
<p>Au niveau <b>de la couche</b></p>	<p>Multi. <b>Matrice-Matrice</b></p>	$  \mathbf{W}^T \leftarrow \mathbf{W}^T - \eta \frac{\partial E}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{W}}  $ $  \mathbf{W}^T \leftarrow \mathbf{W}^T - \eta \frac{\partial E}{\partial \mathbf{Y}} \mathbf{X}^T  $

80

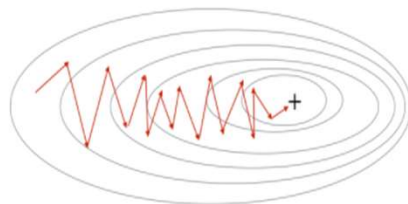


## Pour plus de détails:

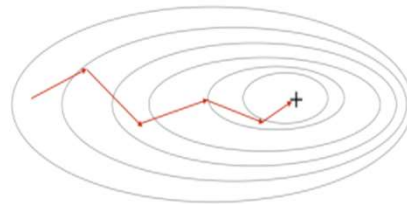
<https://medium.com/datathings/vectorized-implementation-of-back-propagation-1011884df84>  
<https://peterroelants.github.io/posts/neural-network-implementation-part04/>

81

Stochastic Gradient Descent



Mini-Batch Gradient Descent



82

Comment initialiser un réseau de neurones?

$$W = ?$$

83

## Initialisation

**Première idée:** faibles valeurs aléatoires  
(Gaussienne  $\mu = 0, \sigma = 0.01$ )

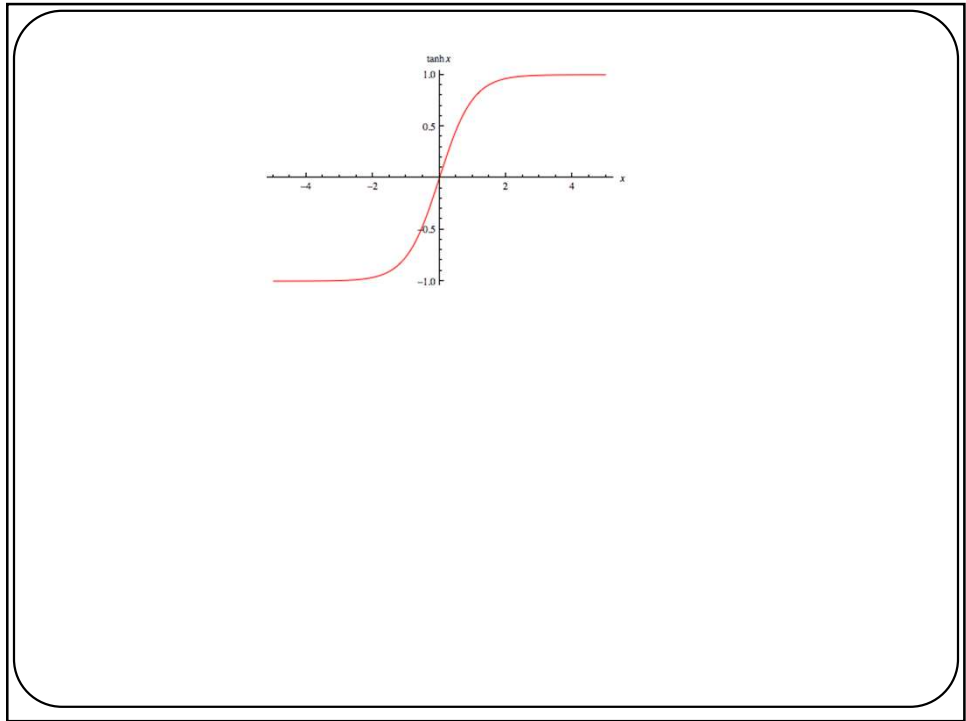
```
W_i=0.01*np.random.randn(H_i,H_im1)
```

Fonctionne bien pour de petits réseaux mais  
**pas pour des réseaux profonds.**

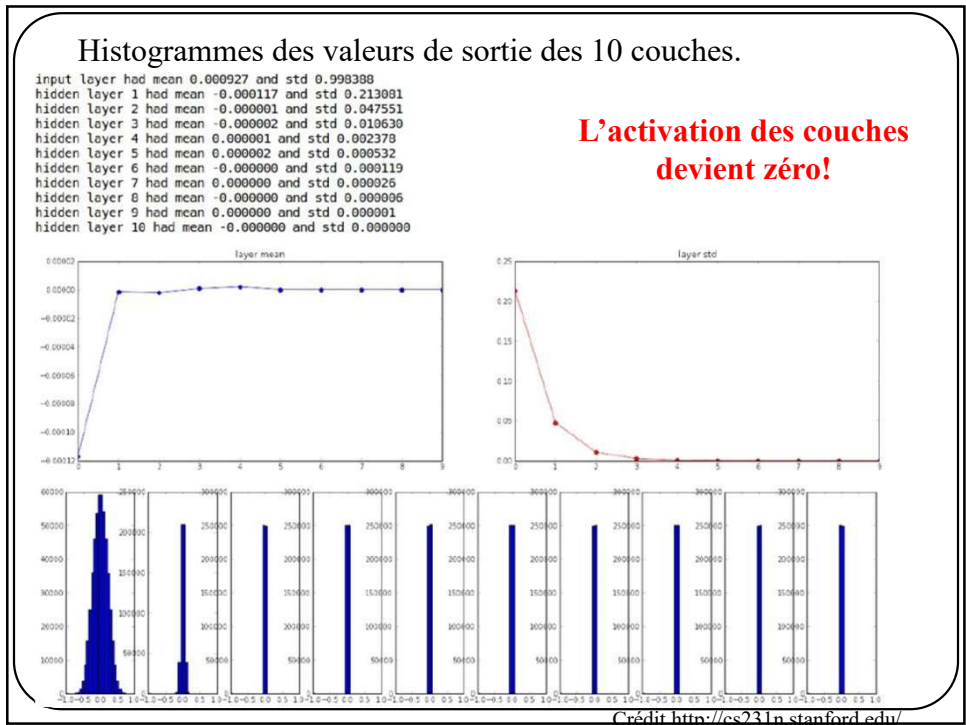


E.g. réseau à 10 couches avec 500 neurones par couche et des **tanh** comme fonctions d'activation.

84



85

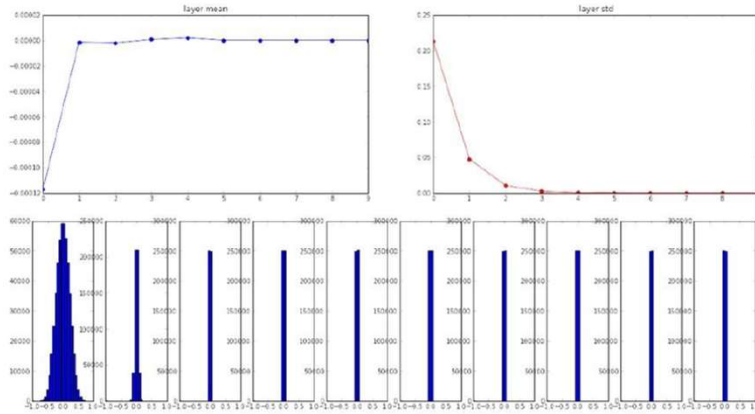


86

## Histogrammes des valeurs de sortie des 10 couches.

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.947551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

Voyez ce qui arrive à la  
*forward pass* et la  
*rétropropagation*



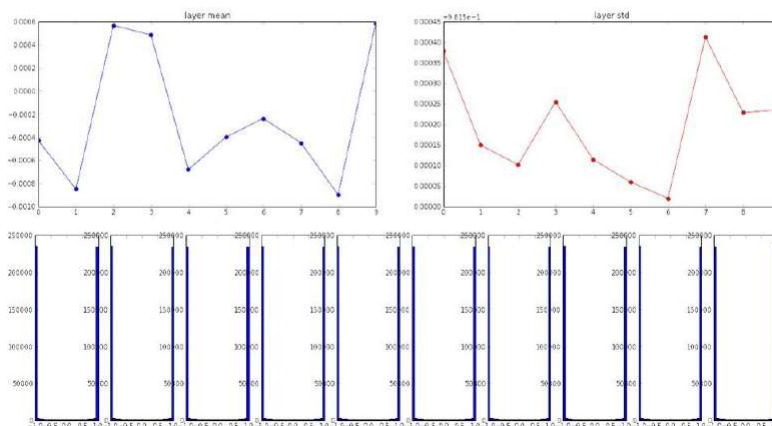
Crédit <http://cs231n.stanford.edu/>

87

$W_i = np.random.randn(H_i, H_{i-1}) * 1$

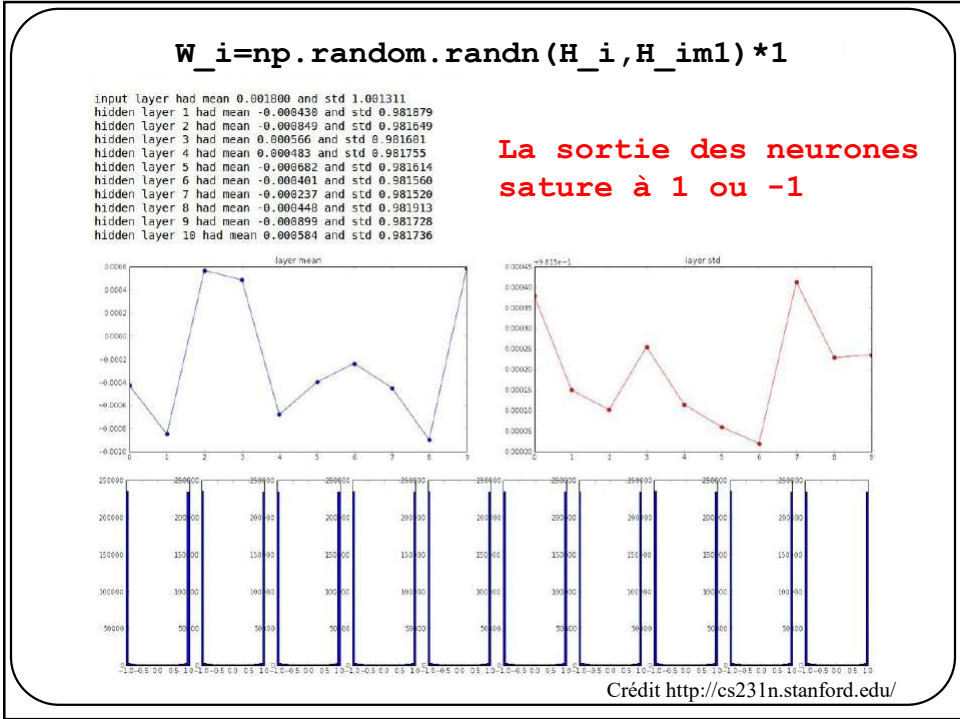
```
input layer had mean 0.001800 and std 1.061311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000949 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981681
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

1 au lieu de 0.01

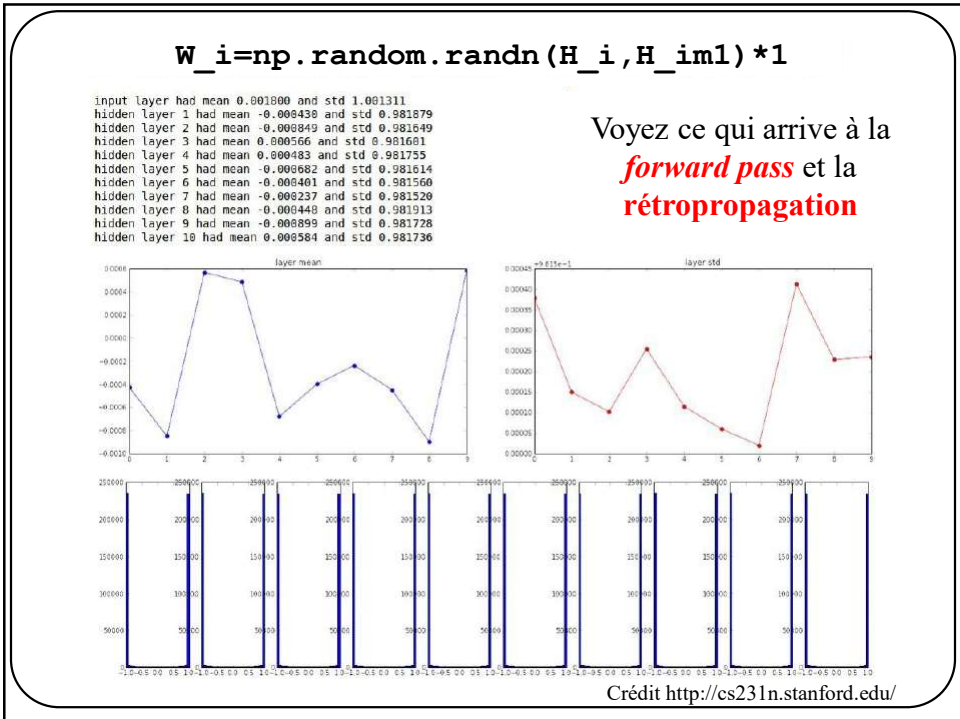


Crédit <http://cs231n.stanford.edu/>

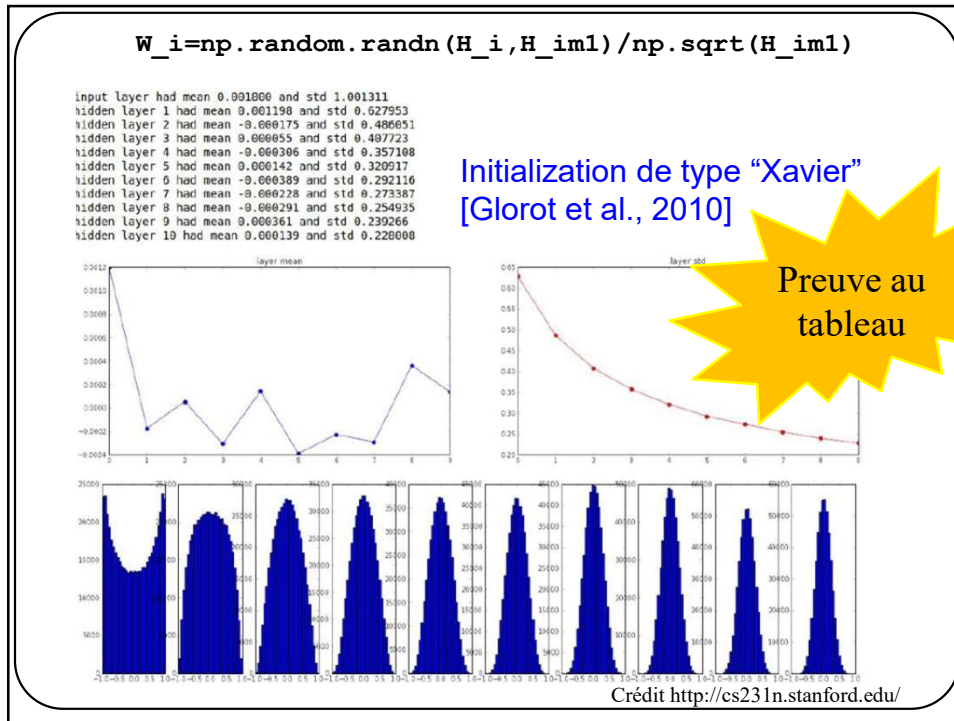
88



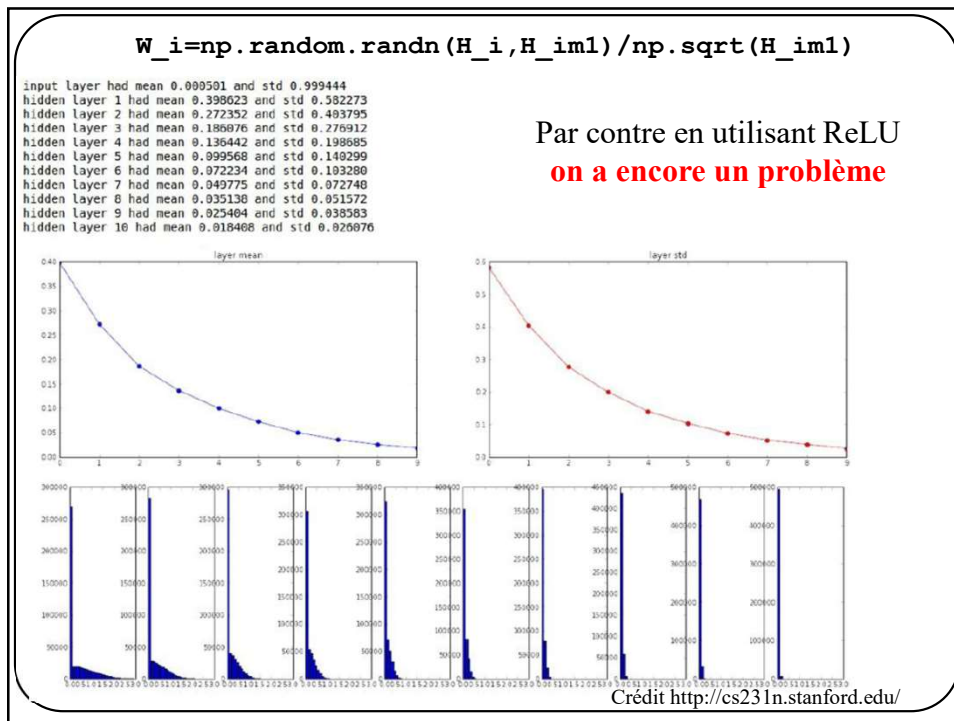
89



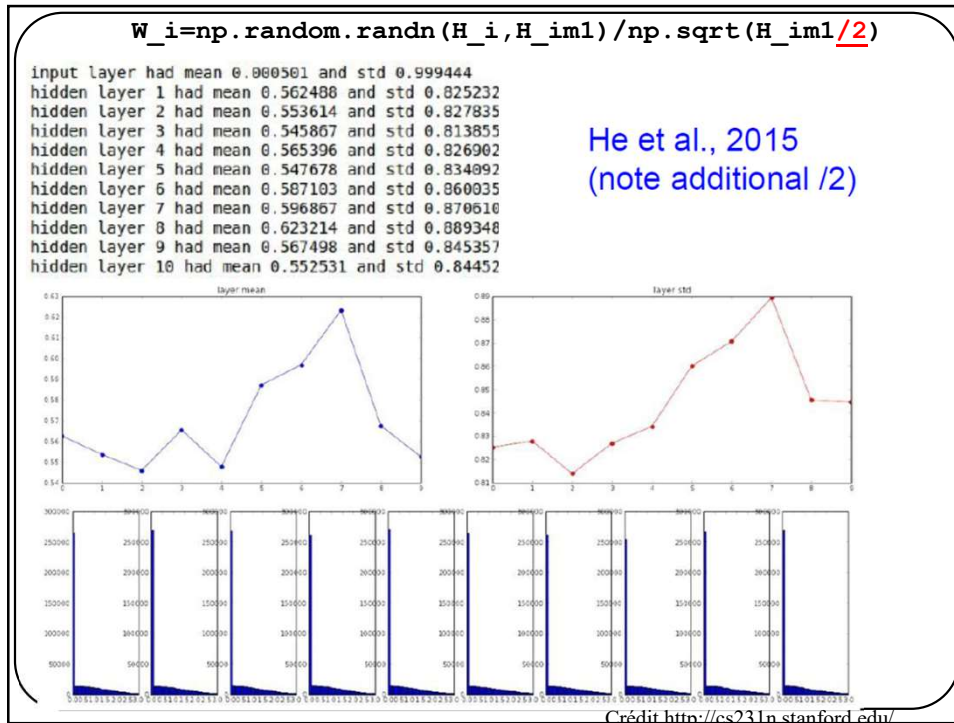
90



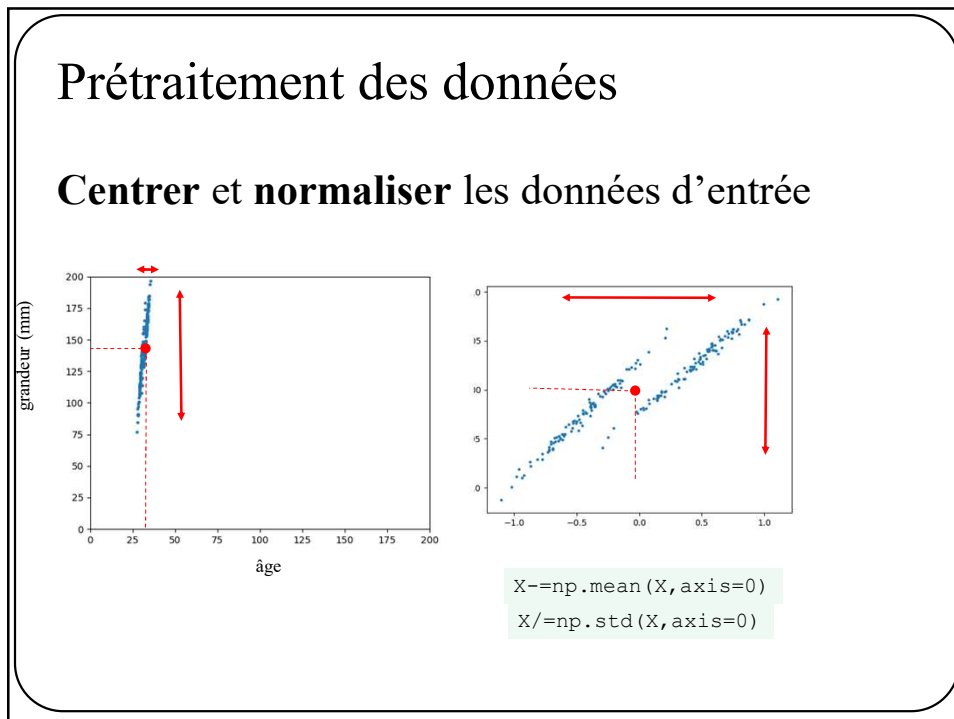
91



92



93



94

## Sanity checks

1. Toujours s'assurer qu'une initialization aléatoire donne une **perte (loss) maximale**

Exemple : pour le cas **10 classes**, une **régularisation à 0** et une **entropie croisée**.

$$E_D(W) = -\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K t_{kn} \ln y_{W,k}(\bar{x}_n)$$

Si l'initialisation est aléatoire, alors la probabilité sera égale pour chaque classe

$$\begin{aligned} E_D(W) &= -\frac{1}{N} \sum_{n=1}^N \ln \frac{1}{10} \\ &= \ln(10) \\ &= 2.30 \end{aligned}$$

95

## Sanity checks

1. Toujours s'assurer qu'une initialization aléatoire donne une **perte (loss) maximale**

Exemple : pour le cas **10 classes**, une **régularisation à 0** et une **entropie croisée**.

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden_size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization
print loss
```

2.30261216167

loss ~2.3.  
"correct" for  
10 classes

returns the loss and the  
gradient for all parameters

Credit <http://cs231n.stanford.edu>

96



## Sanity checks

2. Et lorsqu'on **augmente la régularisation**, la perte augmente aussi

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 1e3) # crank up regularization  
print loss  
3.06859716482
```

loss went up, good. (sanity check)

Crédit <http://cs231n.stanford.edu/>

97

## Sanity checks

3. Toujours s'assurer qu'on peut « **over-fitter** » sur un petit nombre de données.

Lets try to train now...

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

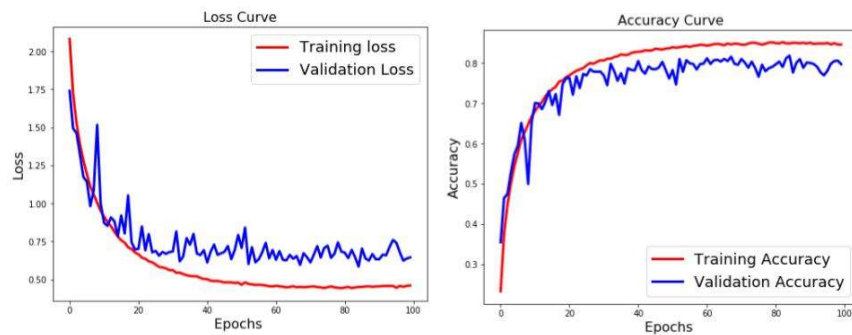
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
trainer = ClassifierTrainer()  
X_tiny = X_train[:20] # take 20 examples  
y_tiny = y_train[:20]  
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,  
    model, two_layer_net,  
    num_epochs=200, reg=0,  
    update='sgd', learning_rate_decay=1,  
    sample_batches = False,  
    learning_rate=1e-3, verbose=True)  
  
Finished epoch 1 / 200: cost: 2.302603, train: 0.400000, val: 0.400000, lr: 1.000000e-03  
Finished epoch 2 / 200: cost: 2.302258, train: 0.600000, val: 0.450000, lr: 1.000000e-03  
Finished epoch 3 / 200: cost: 2.301049, train: 0.600000, val: 0.600000, lr: 1.000000e-03  
Finished epoch 4 / 200: cost: 2.301196, train: 0.650000, val: 0.650000, lr: 1.000000e-03  
Finished epoch 5 / 200: cost: 2.300644, train: 0.600000, val: 0.650000, lr: 1.000000e-03  
Finished epoch 6 / 200: cost: 2.297864, train: 0.550000, val: 0.550000, lr: 1.000000e-03  
Finished epoch 7 / 200: cost: 2.293395, train: 0.600000, val: 0.600000, lr: 1.000000e-03  
Finished epoch 8 / 200: cost: 2.285696, train: 0.500000, val: 0.550000, lr: 1.000000e-03  
Finished epoch 9 / 200: cost: 2.268094, train: 0.550000, val: 0.550000, lr: 1.000000e-03  
Finished epoch 10 / 200: cost: 2.234787, train: 0.500000, val: 0.500000, lr: 1.000000e-03  
Finished epoch 11 / 200: cost: 2.173187, train: 0.500000, val: 0.500000, lr: 1.000000e-03  
Finished epoch 12 / 200: cost: 2.076862, train: 0.500000, val: 0.500000, lr: 1.000000e-03  
Finished epoch 13 / 200: cost: 1.974899, train: 0.400000, val: 0.400000, lr: 1.000000e-03  
Finished epoch 14 / 200: cost: 1.895085, train: 0.400000, val: 0.400000, lr: 1.000000e-03  
Finished epoch 15 / 200: cost: 1.828875, train: 0.450000, val: 0.450000, lr: 1.000000e-03  
Finished epoch 16 / 200: cost: 1.737439, train: 0.450000, val: 0.450000, lr: 1.000000e-03  
Finished epoch 17 / 200: cost: 1.642356, train: 0.500000, val: 0.500000, lr: 1.000000e-03  
Finished epoch 18 / 200: cost: 1.535239, train: 0.600000, val: 0.600000, lr: 1.000000e-03  
Finished epoch 19 / 200: cost: 1.421527, train: 0.600000, val: 0.600000, lr: 1.000000e-03  
Finished epoch 200 / 200: cost: 0.002654, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
Finished epoch 196 / 200: cost: 0.002674, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
Finished epoch 197 / 200: cost: 0.002055, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
Finished epoch 198 / 200: cost: 0.002035, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
Finished epoch 199 / 200: cost: 0.002017, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
Finished epoch 200 / 200: cost: 0.002597, train: 1.000000, val: 1.000000, lr: 1.000000e-03  
finished optimization, best validation accuracy: 1.000000
```

Crédit <http://cs231n.stanford.edu/>

98

## Sanity checks

3. Toujours visualiser les courbes d'apprentissage et de validation

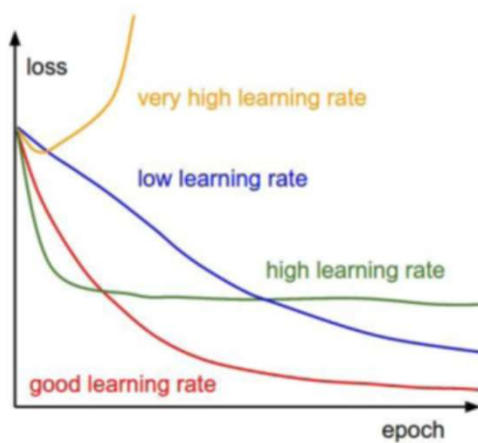


Source: <https://www.learnopencv.com/wp-content/uploads/2017/11/cnn-keras-curves-without-aug.jpg>

99

## Sanity checks

3. Toujours visualiser les courbes d'apprentissage et de validation



Crédit <http://cs231n.stanford.edu>

100

## Sanity checks

### 3. Toujours vérifier la validité d'un gradient

Comme on l'a vu, calculer un gradient est sujet à erreur. Il faut donc s'assurer que nos gradients sont bons au fur et à mesure qu'on rédige notre code. En voici la meilleure façon

#### Rappel

Approximation numérique de la dérivée

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

101

## Sanity checks

### 3. Toujours vérifier la validité d'un gradient

On peut facilement calculer un gradient à l'aide d'une approximation numérique.

#### Rappel

Approximation numérique du gradient

$$\nabla E(W) \approx \frac{E(W+H) - E(W)}{H}$$

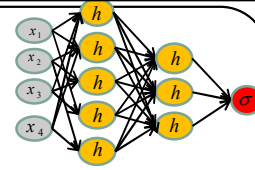
En calculant

$$\frac{\partial E(W)}{\partial w_i} \approx \frac{E(w_i+h) - E(w_i)}{h} \quad \forall i$$

102

## Vérification du gradient

(exemple)



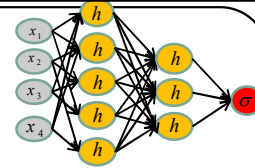
<b>W</b>	<b>W+h</b>	<b>gradient W</b>
$W_{00} = 0.34$	$W_{00} = 0.34 + 0.0001$	$-2.5 = (1.25322 - 1.25347) / 0.0001$
$W_{01} = -1.11$	$W_{01} = -1.11$	
$W_{02} = 0.78$	$W_{02} = 0.78$	
...	...	
$W_{20} = -3.1$	$W_{20} = -3.1$	
$W_{21} = -1.5,$	$W_{21} = -1.5,$	
$W_{22} = 0.33$	$W_{22} = 0.33$	

$E(W) = 1.25347$     $E(W+h) = 1.25322$

103

## Vérification du gradient

(exemple)



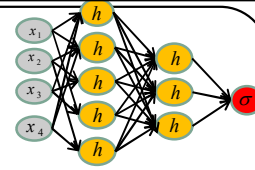
<b>W</b>	<b>W+h</b>	<b>gradient W</b>
$W_{00} = 0.34$	$W_{00} = 0.34$	$-2.5$
$W_{01} = -1.11$	$W_{01} = -1.11 + 0.0001$	$0.6 = (1.25353 - 1.25347) / 0.0001$
$W_{02} = 0.78$	$W_{02} = 0.78$	
...	...	
$W_{20} = -3.1$	$W_{20} = -3.1$	
$W_{21} = -1.5,$	$W_{21} = -1.5,$	
$W_{22} = 0.33$	$W_{22} = 0.33$	

$E(W) = 1.25347$     $E(W+h) = 1.25353$

104

## Vérification du gradient

(exemple)



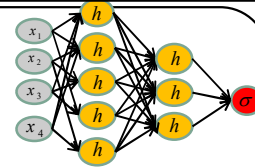
<b>W</b>	<b>W+h</b>	<b>gradient W</b>
$W_{00} = 0.34$	$W_{00} = 0.34$	<b>-2.5</b>
$W_{01} = -1.11$	$W_{01} = -1.11$	<b>0.6</b>
$W_{02} = 0.78$	$W_{02} = 0.78 + 0.0001$	<b><math>0.0 = (1.25347 - 125347) / 0.0001</math></b>
...	...	
$W_{20} = -3.1$	$W_{20} = -3.1$	
$W_{21} = -1.5,$	$W_{21} = -1.5,$	
$W_{22} = 0.33$	$W_{22} = 0.33$	

**$E(W) = 1.25347$     $E(W+h) = 1.25347$**

105

## Vérification du gradient

(exemple)



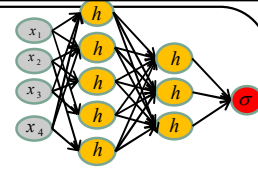
<b>W</b>	<b>W+h</b>	<b>gradient W</b>
$W_{00} = 0.34$	$W_{00} = 0.34$	<b>-2.5</b>
$W_{01} = -1.11$	$W_{01} = -1.11$	<b>0.6</b>
$W_{02} = 0.78$	$W_{02} = 0.78$	<b>0.0</b>
...	...	...
$W_{20} = -3.1$	$W_{20} = -3.1$	<b>1.1</b>
$W_{21} = -1.5,$	$W_{21} = -1.5,$	<b>1.3</b>
$W_{22} = 0.33$	$W_{22} = 0.33$	<b>-2.1</b>

**$E(W) = 1.25347$**

106



## Vérification du gradient

(exemple)



**gradient W**  
**(numérique)**

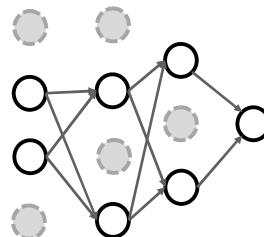
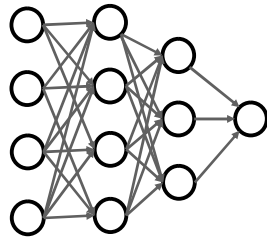
**gradient W**  
**(retro-propagation)**

-2.5		-2.5
0.6		0.6
0.0		0.0
...		...
1.1		1.1
1.3		1.3
-2.1		-2.1

107

## Autre bonne pratique : *Dropout*

**Forcer à zéro certains neurones** de façon aléatoire à chaque itération



108

## Autre bonne pratique : *Dropout*

Idée : s'assurer que chaque neurone apprend pas lui-même en brisant au hasard des chemins.

Crédit <http://cs231n.stanford.edu/>

109

## Autre bonne pratique : *Dropout*

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Crédit <http://cs231n.stanford.edu/>

110

## Autre bonne pratique : *Dropout*

Le problème avec *Dropout* est en **prédiction** (« test time »)

car *dropout* **ajoute du bruit** à la prédiction

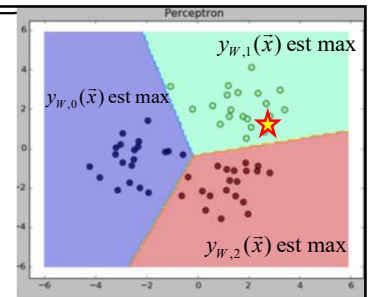
$$pred = y_W(\vec{x}, Z)$$

↑  
masque aléatoire

111

*dropout* **ajoute du bruit** à la prédiction.

Exemple simple :  $\vec{x} = \begin{pmatrix} 2.2 \\ 1.3 \end{pmatrix}, t = 1$



**Si on lance le modèle 10 fois, on aura 10 réponses différentes**

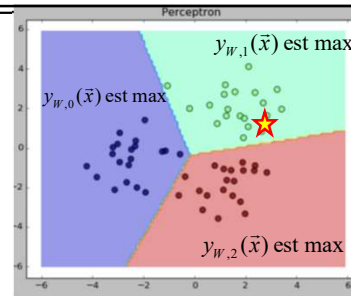
[ 0.09378555	0.76511644	0.141098 ]
[ 0.13982909	0.62885327	0.23131764]
[ 0.23658253	0.61960162	0.14381585]
[ 0.23779425	0.51357115	0.24863461]
[ 0.16005442	0.68060227	0.1593433 ]
[ 0.16303195	0.50583392	0.33113413]
[ 0.24183069	0.51319834	0.24497097]
[ 0.14521815	0.52006858	0.33471327]
[ 0.09952161	0.66276146	0.23771692]
[ 0.16172851	0.6044877	0.23378379]

112



dropout **ajoute du bruit** à la prédiction.

Exemple simple :  $\vec{x} = \begin{pmatrix} 2.2 \\ 1.3 \end{pmatrix}, t = 1$



**Solution**, exécuter le modèle un grand nombre de fois et **prendre la moyenne**.

```
[ 0.09378555  0.76511644  0.141098 ]
[ 0.13982909  0.62885327  0.23131764]
[ 0.23658253  0.61960162  0.14381585]
[ 0.23779425  0.51357115  0.24863461]
[ 0.16005442  0.68060227  0.1593433 ]
[ 0.16303195  0.50583392  0.33113413]
[ 0.24183069  0.51319834  0.24497097]
[ 0.14521815  0.52006858  0.33471327]
[ 0.09952161  0.66276146  0.23771692]
[ 0.16172851  0.6044877   0.23378379]
(...)
```

[ 0.15933813, 0.65957005, 0.18109183]

113

Exécuter le modèle un grand nombre de fois et **prendre la moyenne** revient à calculer **l'espérance mathématique**

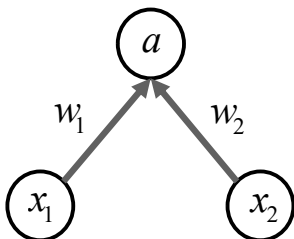
$$pred = E_z [y_w(\vec{x}, \vec{z})] = \sum_i P(\vec{z}) y_w(\vec{x}, \vec{z})$$

**Bonne nouvelle, on peut faire plus simple en approximant cette l'espérance mathématique!**

114

## Regardons pour un neurone

Avec une probabilité de *dropout* de 50%, en prédiction  $w_1$  et  $w_2$  seront **nuls 1 fois sur 2**



$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(w_1x_1 + 0x_2) \\ &\quad + \frac{1}{4}(0x_1 + w_2x_2) + \frac{1}{4}(0x_1 + 0x_2) \\ &= \frac{1}{2}(w_1x_1 + w_2x_2) \end{aligned}$$

En prédiction, on a qu'à multiplier par la prob. de *dropout*.

115

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensemble forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

En prédiction, tous les neurones sont actifs

→ tout ce qu'il faut faire est de **multiplier la sortie de chaque couche par la probabilité de dropout**

Crédit <http://cs231n.stanford.edu/>

116

Descente de gradient  
**version améliorée**

117

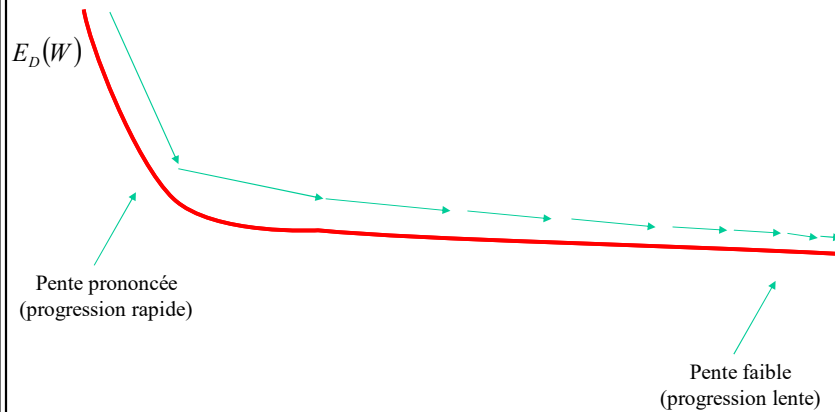
Descente de gradient

$$W^{[t+1]} = W^{[t]} - \eta \nabla E_D(W^{[t]})$$

118

## Descente de gradient : **problème**

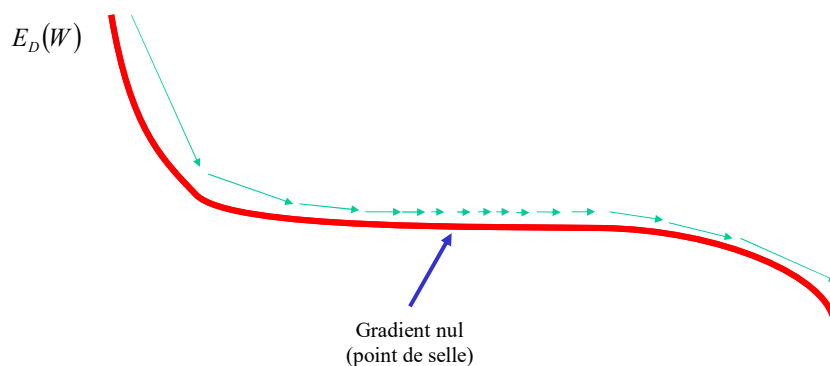
**Progrès quasi nul lorsque la pente est très faible**



119

## Descente de gradient : **problème**

**Les points de selles sont fréquents en haute dimension**



120

## Descente de gradient : **problème**

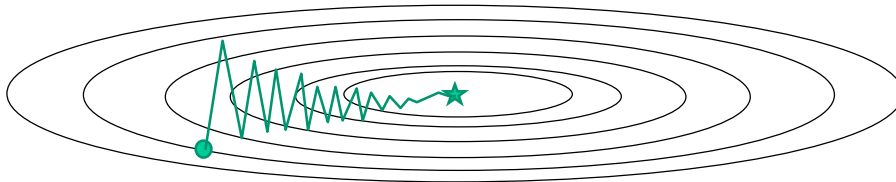
Qu'arrive-t-il si la fonction de coût (loss) a une pente prononcée dans une direction et moins prononcée dans une autre direction?

121

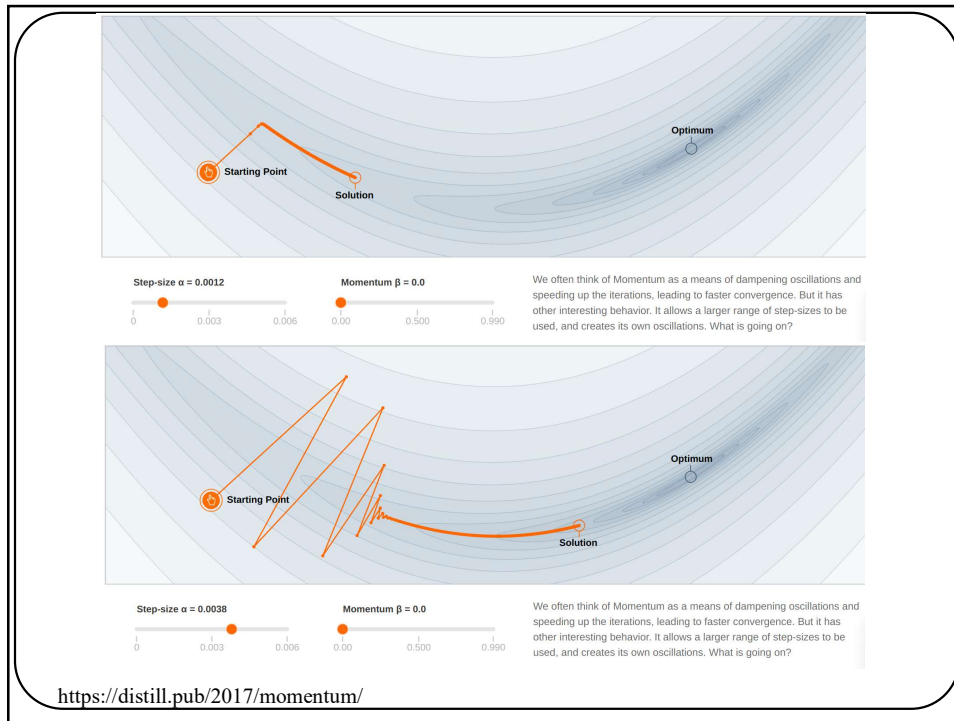
## Descente de gradient : **problème**

Qu'arrive-t-il si la fonction de coût (loss) a une pente prononcée dans une direction et moins prononcée dans une autre direction?

**Progrès très lent le long de la pente la plus faible et oscillation le long de l'autre direction.**



122



123

## Descente de gradient + Momentum

Descente de gradient  
stochastique

$$w_{t+1} = w_t - \eta \nabla E_{\bar{x}_n}(w_t)$$

Descente de gradient  
stochastique + **Momentum**

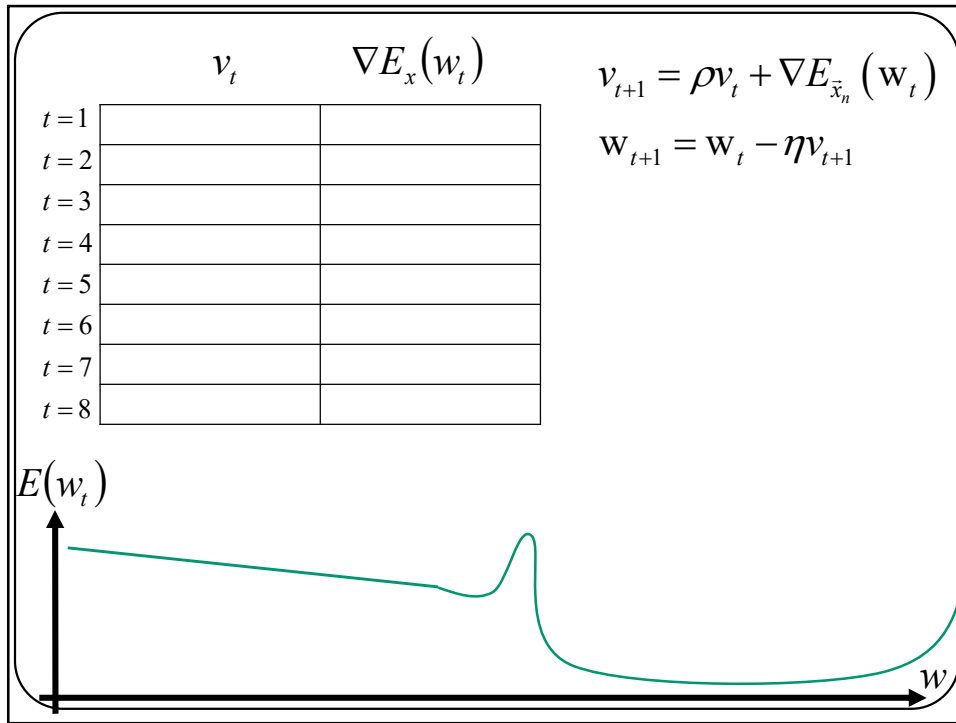
$$v_{t+1} = \rho v_t + \nabla E_{\bar{x}_n}(w_t)$$

$$w_{t+1} = w_t - \eta v_{t+1}$$

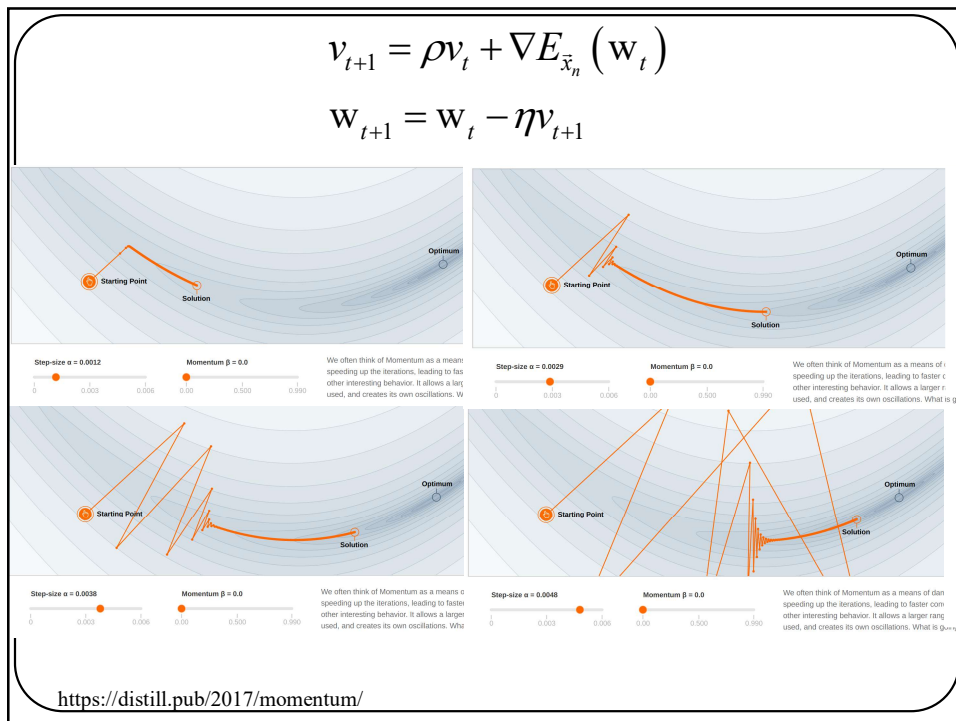
Provient de l'équation de la vitesse  
(à démontrer en devoir ou en exercice)

$\rho$  exprime la « friction », en général  $\in [0.5, 1[$

124



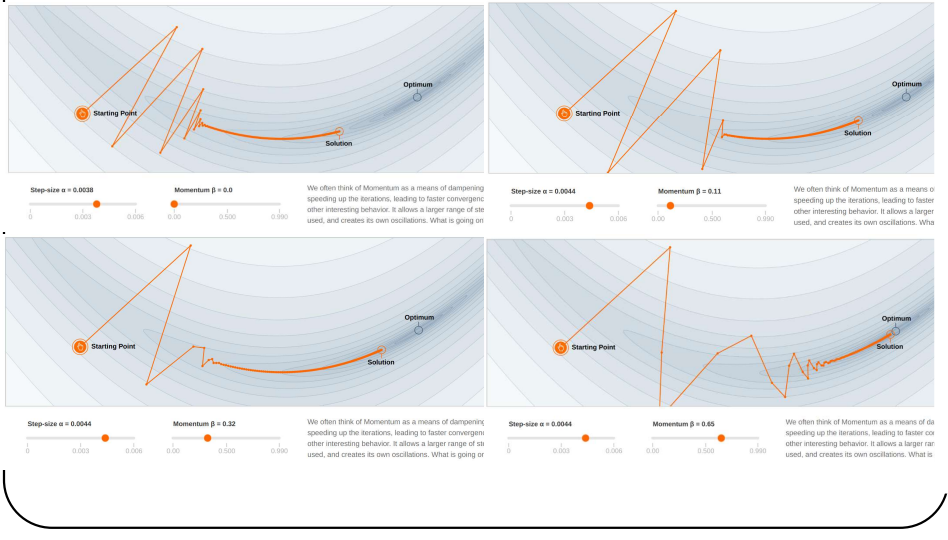
125



126

$$v_{t+1} = \rho v_t + \nabla E_{\bar{x}_n} (w_t)$$

$$w_{t+1} = w_t - \eta v_{t+1}$$



127

## AdaGrad (décroissance automatique de $\eta$ )

Descente de gradient  
stochastique

**AdaGrad**

$$w_{t+1} = w_t - \eta \nabla E_{\bar{x}_n} (w_t)$$

$$dE_t = \nabla E_{\bar{x}_n} (w_t)$$

$$m_{t+1} = m_t + |dE_t|$$

$$w_{t+1} = w_t - \frac{\eta}{m_{t+1} + \epsilon} dE_t$$

128



## AdaGrad (décroissance automatique de $\eta$ )

Descente de gradient  
stochastique

**AdaGrad**

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$

$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$

$$m_{t+1} = m_t + |dE_t|$$

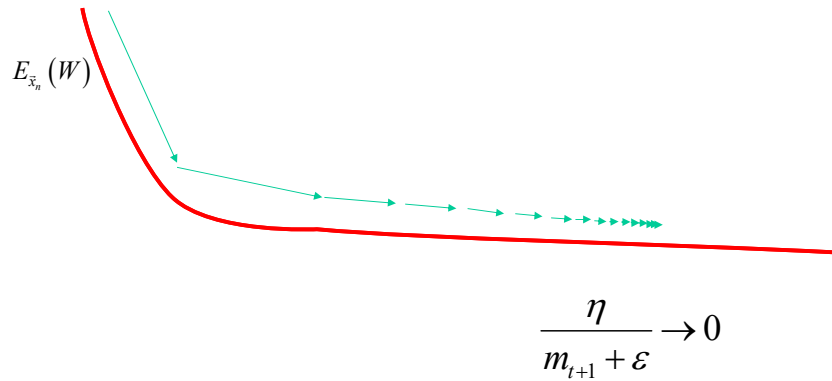
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} dE_t$$

$\eta$  décroît sans cesse au fur  
et à mesure de l'optimisation

129

## AdaGrad (décroissance automatique de $\eta$ )

Qu'arrive-t-il à long terme?



130

## RMSProp (AdaGrad amélioré)

### AdaGrad

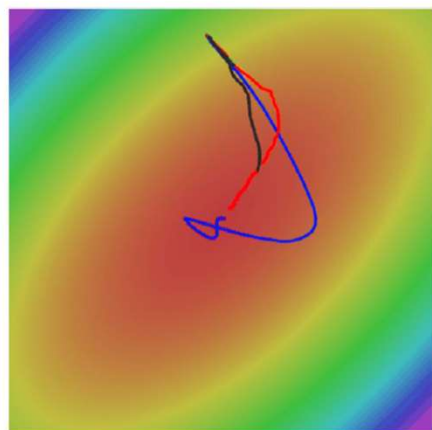
$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$
$$m_{t+1} = m_t + |dE_t|$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} dE_t$$

### RMSProp

$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$
$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} dE_t$$

$\eta$  décroît lorsque le gradient est élevé  
 $\eta$  augmente lorsque le gradient est faible

131



— SGD  
— SGD+Momentum  
— RMSProp

132

## Adam (Combo entre Momentum et RMSProp)

### Momentum

$$v_{t+1} = \rho v_t + \nabla E_{\bar{x}_n} (w_t)$$

$$w_{t+1} = w_t - \eta v_{t+1}$$

### Adam

$$dE_t = \nabla E_{\bar{x}_n} (w_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) dE_t$$

$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

$$w_{t+1} = w_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

133

## Adam (Combo entre Momentum et RMSProp)

### Momentum

$$v_{t+1} = \rho v_t + \nabla E_{\bar{x}_n} (w_t)$$

$$w_{t+1} = w_t - \eta v_{t+1}$$

### Adam

$$dE_t = \nabla E_{\bar{x}_n} (w_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) dE_t$$

$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

$$w_{t+1} = w_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

*Momentum*

134

## Adam (Combo entre Momentum et RMSProp)

### RMSProp

$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$

$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} dE_t$$

### Adam

$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) dE_t$$

$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

135

## Adam (Version complète)

$$v_{t=0} = 0$$

$$m_{t=0} = 0$$

for t=1 à num\_iterations

for n=0 à N

$$dE_t = \nabla E_{\bar{x}_n}(\mathbf{w}_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) dE_t$$

$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

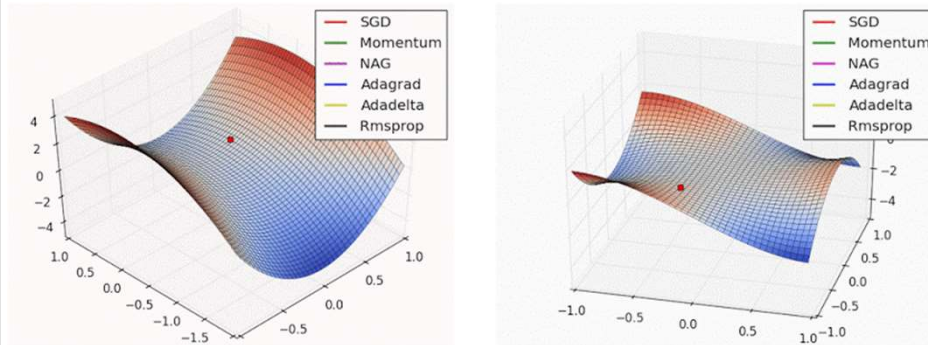
$$v_{t+1} = \frac{v_{t+1}}{1 - \beta_1^t}, m_{t+1} = \frac{m_{t+1}}{1 - \beta_2^t}$$

$$\beta_1 = 0.9, \beta_2 = 0.99$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

136

## Illustrations



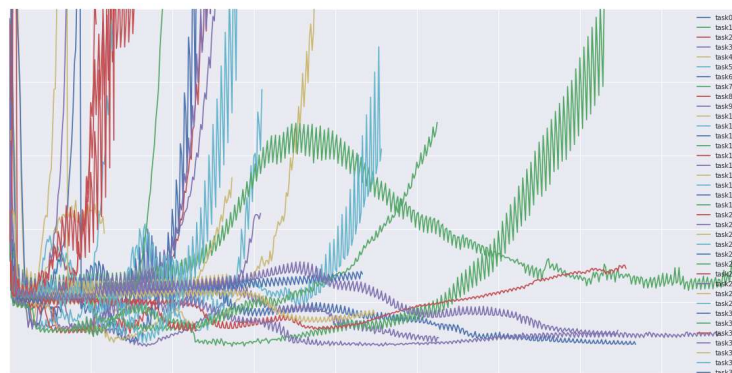
À voir sur :

[www.denizyuret.com/2015/03/alec-radfords-animations-for.html](http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html)

137

## Autre excellent survol

<http://ruder.io/optimizing-gradient-descent/>



138