

Réseaux de neurones  
IFT 780

Réseaux de neurones multicouches  
Par  
Pierre-Marc Jodoin

1

Séparation linéaire

(2D et 2 classes)

biais

poids

$$y_{\vec{w}}(\vec{x}) = w_0 + w_1x_1 + w_2x_2$$

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

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

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

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)

2

Perceptron

(2D et 2 classes)

Neurone

Produit scalaire + fonction d'activation

Fonction de coût perceptron (loss)  
et gradient

$$E_D(\vec{w}) = \frac{1}{N} \sum_{\vec{x}_n \in M} -t_n \vec{w}^T \vec{x}_n$$

où M est l'ensemble des données mal classées

$$\nabla E_D(\vec{w}) = \frac{1}{N} \sum_{\vec{x}_n \in M} -t_n \vec{x}_n$$

3

## Hinge Loss

(2D et 2 classes)

Neurone

Produit scalaire + fonction d'activation

Fonction de coût SVM (hinge loss) et gradient

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

$$\nabla E_D(\vec{w}) = \frac{1}{N} \sum_{\substack{n=1 \\ x_n \in M}}^N -t_n \vec{x}_n$$

4

---

---

---

---

---

---

---

---

## Régression logistique

(2D, 2 classes)

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

Neurone

Fonction de coût (loss) et gradient

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

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

5

---

---

---

---

---

---

---

---

## Perceptron Multiclasse

(2D et 3 classes)

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

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

6

---

---

---

---

---

---

---

---

# Perceptron Multiclasse

Exemple

$y_{\vec{w}}(\vec{x}) = \begin{bmatrix} -2 & -3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 1.1 \\ -2 \end{bmatrix} = \begin{bmatrix} -6.9 \\ -9.6 \\ 8.2 \end{bmatrix}$

Classe 0  
 Classe 1  
 Classe 2

7

---

---

---

---

---

---

---

---

# Perceptron Multiclasse

Fonction de coût (*Perceptron loss – One-VS-One*)

$$E_D(W) = \frac{1}{N} \sum_{\vec{x}_n \in M} (\bar{W}_j^T \vec{x}_n - \bar{W}_{t_n}^T \vec{x}_n)$$

Somme sur l'ensemble des données mal classées  
 Score de la classe faussement prédite par le modèle  
 Score de la bonne classe

8

---

---

---

---

---

---

---

---

# Perceptron Multiclasse

Fonction de coût (*Perceptron loss – One-VS-One*)

$$E_D(W) = \frac{1}{N} \sum_{\vec{x}_n \in M} \underbrace{(\bar{W}_j^T \vec{x}_n - \bar{W}_{t_n}^T \vec{x}_n)}_{E_{\vec{x}_n}}$$

$\nabla_{W_j} E_{\vec{x}_n} = \vec{x}_n$   
 $\nabla_{W_{t_n}} E_{\vec{x}_n} = -\vec{x}_n$   
 $\nabla_{W_i} E_{\vec{x}_n} = 0 \quad \forall i \neq j \text{ et } t_n$

9

---

---

---

---

---

---

---

---

## Perceptron Multiclasse one-vs-one

Descente de gradient stochastique (version naïve, batch\_size = 1)

```
Initialiser W
k=0, i=0
DO k=k+1
  FOR n= 1 to N
    j = argmax Wj  $\vec{x}_n$ 
    IF j ≠ ti THEN /* donnée mal classée*/
       $\vec{w}_j = \vec{w}_j - \eta \vec{x}_n$ 
       $\vec{w}_{t_i} = \vec{w}_{t_i} + \eta \vec{x}_n$ 
  UNTIL toutes les données sont bien classées.
```

10

---

---

---

---

---

---

---

---

## Perceptron Multiclasse one-vs-one

Exemple d'entraînement ( $\eta=1$ )

$$\vec{x}_n = (0.4, -1), t_n = 0$$

$$y_W(\vec{x}) = \begin{bmatrix} -2 & 3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 0.4 \\ -1 \end{bmatrix} = \begin{bmatrix} -1.6 \\ -7.1 \\ 0.5 \end{bmatrix} \begin{matrix} \text{Classe 0} \\ \text{Classe 1} \\ \text{Classe 2} \end{matrix}$$

FAUX!

11

---

---

---

---

---

---

---

---

## Perceptron Multiclasse

Exemple d'entraînement ( $\eta=1$ )

$$\vec{x}_n = (0.4, -1.0), t_n = 0$$

$$\vec{w}_0 \leftarrow \vec{w}_0 + \vec{x}_n \quad \begin{bmatrix} -2.0 \\ 3.6 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ 0.4 \\ -1 \end{bmatrix} = \begin{bmatrix} -1.0 \\ 4.0 \\ -0.5 \end{bmatrix}$$

$$\vec{w}_2 \leftarrow \vec{w}_2 - \vec{x}_n \quad \begin{bmatrix} -6.0 \\ 4.0 \\ -4.9 \end{bmatrix} - \begin{bmatrix} 1 \\ 0.4 \\ -1 \end{bmatrix} = \begin{bmatrix} -7.0 \\ 3.6 \\ -3.9 \end{bmatrix}$$

12

---

---

---

---

---

---

---

---

12

## Hinge Multiclasse

Fonction de coût (**Hinge loss** ou **SVM loss – One vs one**)

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

Somme sur l'ensemble des  
Données mal classées

Score de la mauvaise classe prédite

Score de la bonne classe

13

## Hinge Multiclasse

Fonction de coût (**Hinge loss** ou **SVM loss – One vs One**)

$$E_D(W) = \frac{1}{N} \sum_{\vec{x}_n \in M} \underbrace{\max(0, 1 + \bar{W}_j^T \vec{x}_n - \bar{W}_{t_n}^T \vec{x}_n)}_{E_{\vec{x}_n}}$$

$$\nabla_{\bar{W}_{t_n}} E_{\vec{x}_n} = \begin{cases} -\vec{x}_n & \text{si } \bar{W}_{t_n}^T \vec{x}_n < \bar{W}_j^T \vec{x}_n + 1 \\ 0 & \text{sinon} \end{cases}$$

$$\nabla_{\bar{W}_j} E_{\vec{x}_n} = \begin{cases} \vec{x}_n & \text{si } \bar{W}_{t_n}^T \vec{x}_n < \bar{W}_j^T \vec{x}_n + 1 \text{ et } j \neq t_n \\ 0 & \text{sinon} \end{cases}$$

14

## Hinge Multiclasse one-vs-one

Descente de gradient stochastique (version naïve, batch\_size = 1)

```
Initialiser W
k=0, i=0
DO k=k+1
  FOR n = 1 to N
    IF  $\bar{W}_{t_n}^T \vec{x}_n < \bar{W}_j^T \vec{x}_n + 1$  THEN
       $\bar{W}_{t_n} = \bar{W}_{t_n} - \eta \vec{x}_n$ 
       $\bar{W}_j = \bar{W}_j + \eta \vec{x}_n$ 
```

UNTIL toutes les données sont bien classées.

Au TP1, implanter cette version « naïve »

15

## Hinge Multiclasse

Fonction de coût (**Hinge loss** ou **SVM loss – One vs all**)

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

Somme sur l'ensemble des données

Score d'une mauvaise classe

Score de la bonne classe

16

## Hinge Multiclasse

Fonction de coût (**Hinge loss** ou **SVM loss – One vs all**)

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

$$\nabla_{\vec{W}_{t_n}} E_{\vec{x}_n} = \begin{cases} -\vec{x}_n & \text{si } \vec{W}_{t_n}^T \vec{x}_n < \vec{W}_j^T \vec{x}_n + 1 \\ 0 & \text{sinon} \end{cases}$$

$$\nabla_{\vec{W}_j} E_{\vec{x}_n} = \begin{cases} \vec{x}_n & \text{si } \vec{W}_{t_n}^T \vec{x}_n < \vec{W}_j^T \vec{x}_n + 1 \text{ et } j \neq t_n \\ 0 & \text{sinon} \end{cases}$$

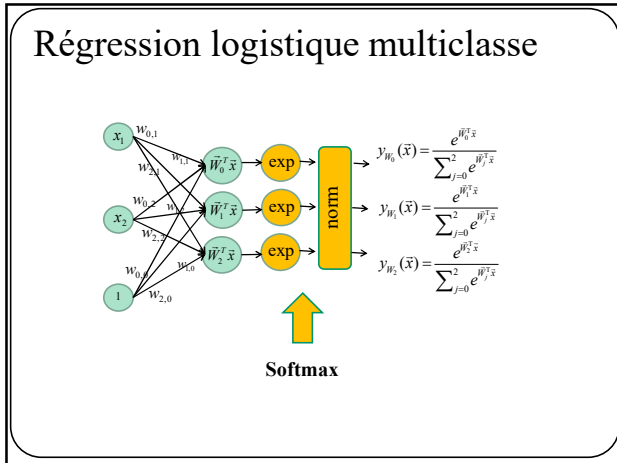
17

## Hinge Multiclasse one-vs-all

Descente de gradient stochastique (version naïve, batch\_size = 1)

```
Initialiser W
k=0, i=0
DO k=k+1
  FOR n = 1 to N
    IF  $\vec{W}_{t_n}^T \vec{x}_n < \vec{W}_j^T \vec{x}_n + 1$  THEN
       $\vec{w}_{t_n} = \vec{w}_{t_n} + \eta \vec{x}_n$ 
      FOR j=1 to NB_CLASSES THEN
        IF  $\vec{W}_{t_n}^T \vec{x}_n < \vec{W}_j^T \vec{x}_n + 1$  AND  $j \neq t_n$  THEN
           $\vec{w}_j = \vec{w}_j - \eta \vec{x}_n$ 
      UNTIL toutes les données sont bien classées.
```

18



19

---

---

---

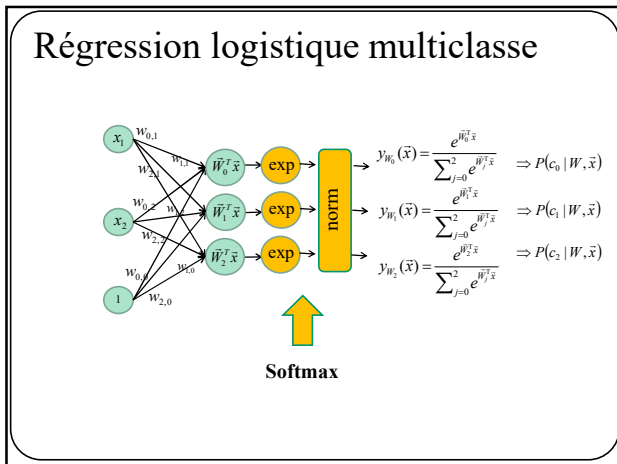
---

---

---

---

---



20

---

---

---

---

---

---

---

---

### Régression logistique multiclass

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

Étiquettes de classe : one-hot vector

21

---

---

---

---

---

---

---

---

## Régression logistique multiclasse

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

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

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

22

Tous les détails du gradient de l'entropie croisée :

[jodoin.github.io/cours/ift603/softmax\\_grad.html](http://jodoin.github.io/cours/ift603/softmax_grad.html)

Au tp1: implanter une **version naïve** avec des boucles for et une **version vectorisée SANS boucle for**.

23

$$\vec{x} = \begin{bmatrix} -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}, t = 2$$

$$W = \begin{bmatrix} 0.0 & 0.01 & -0.05 & 0.1 & 0.05 \\ 0.2 & 0.7 & 0.2 & 0.05 & 0.16 \\ -0.3 & 0.0 & -0.45 & -0.2 & 0.03 \end{bmatrix}$$

**Hinge loss – One vs All**

Score

$$\begin{aligned} & \max(0, -2.85 - 0.28 + 1) + \\ & \max(0, 0.86 - 0.28 + 1) + \\ & \max(0, -2.13) + \max(0, 1.58) \\ & = 1.58 \end{aligned}$$

**Entropie croisée**

Score

$$\begin{aligned} & \begin{bmatrix} -2.85 \\ 0.86 \\ 0.28 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 0.06 \\ 2.36 \\ 1.32 \end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix} 0.02 \\ 0.63 \\ 0.35 \end{bmatrix} \\ & -\ln(0.35) = 0.452 \end{aligned}$$

(Softmax)

24



# Maximum *a posteriori*

Régularisation

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

Fonction de perte (pointing to  $E_D(W)$ )  
Constante (pointing to  $\lambda$ )  
Régularisation (pointing to  $R(W)$ )

En général L1 ou L2  $R(W) = \|W\|_1$  ou  $\|W\|_2$

25

---

---

---

---

---

---

---

---

# Optimisation

Descente de gradient

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

↗ 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
    w = w - ηk ∇E(xn)
  JUSQU'À ce que toutes les données
  soient bien classées ou k==MAX_ITER
        
```

Optimisation par Batch

```

Initialiser w
k=0
FAIRE k=k+1
  w = w - ηk ∑i ∇E(xi)
  JUSQU'À ce que toutes les données
  soient bien classées ou k==MAX_ITER
        
```

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

26

---

---

---

---

---

---

---

---

# Maintenant, rendons le réseau profond

Maintenant, rendons le réseau

27

---

---

---

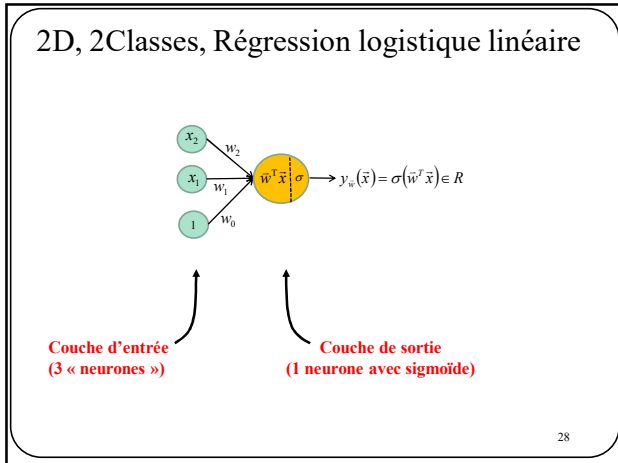
---

---

---

---

---




---

---

---

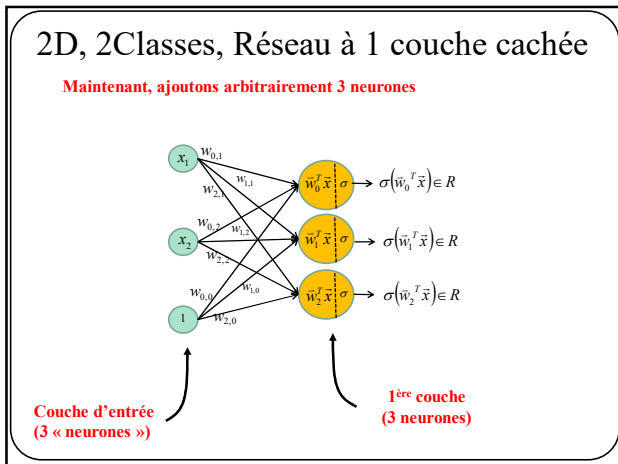
---

---

---

---

---




---

---

---

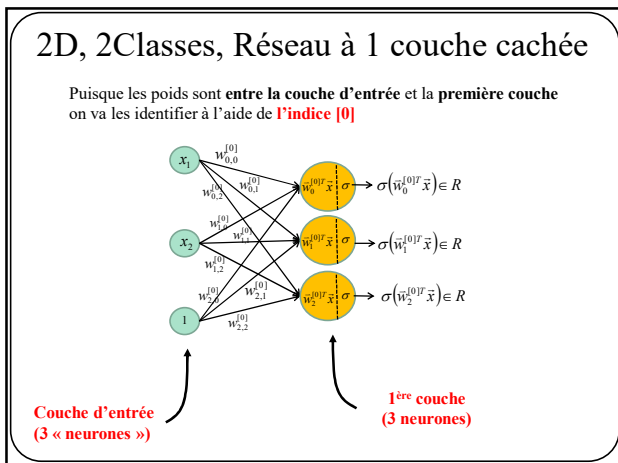
---

---

---

---

---




---

---

---

---

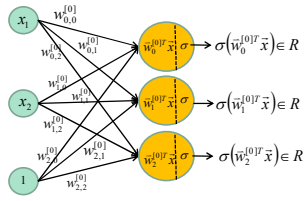
---

---

---

---

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

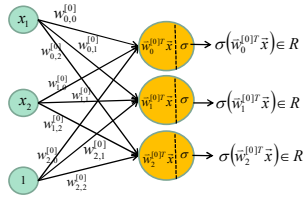


**NOTE:** à la sortie de la première couche, on a **3 réels** calculés ainsi

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

31

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



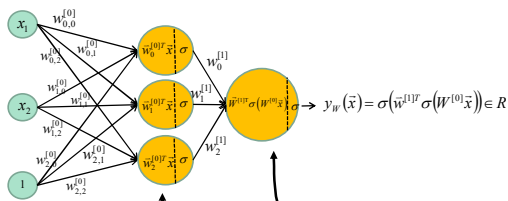
**NOTE:** représentation plus simple de la sortie de la 1<sup>ère</sup> couche (**3 réels**)

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

32

## 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**.

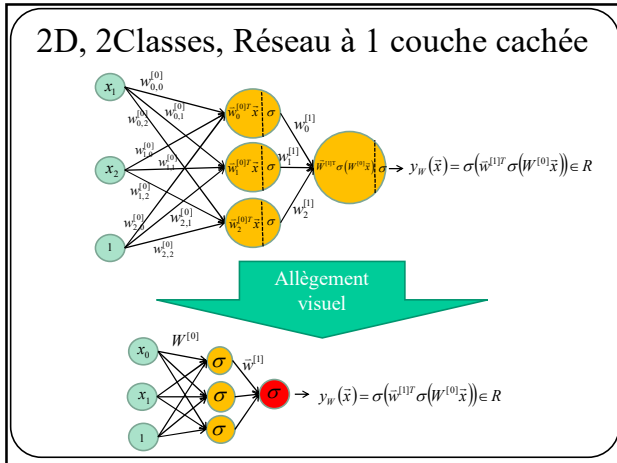


Couche d'entrée  
(3 « neurones »)

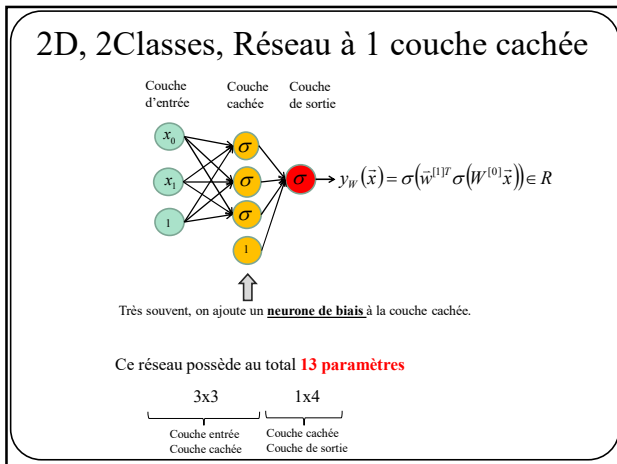
1<sup>ère</sup> couche cachée  
(3 neurones)

Couche de sortie  
(1 neurone)

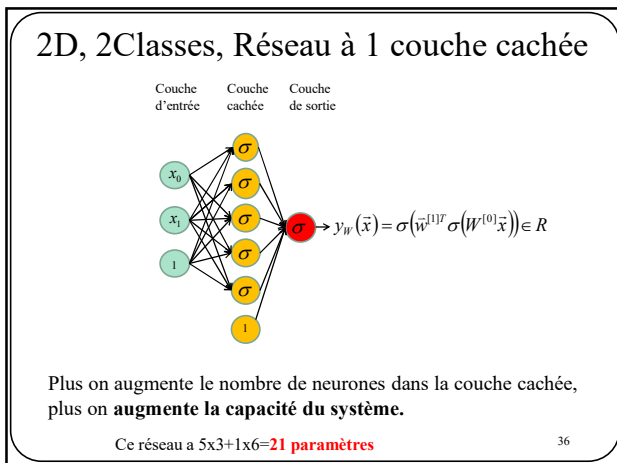
33



34



35



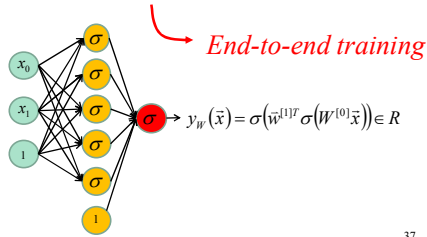
36

## NOTE Importante

Le but de la première couche est de **projeter les données d'entrée** (ici  $\vec{x} \in \mathbb{R}^2$ ) vers un espace dimensionnel plus grand (ici  $\sigma(W^{[0]}\vec{x}) \in \mathbb{R}^3$ ) 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**.

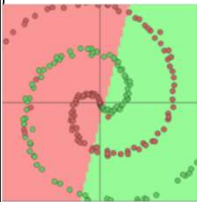


37

37

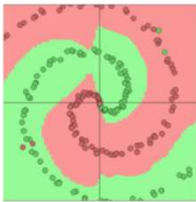
## Nombre de neurones VS Capacité

Aucun neurone caché



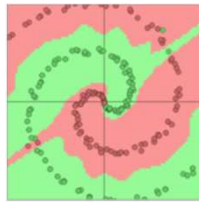
Classification linéaire  
**Underfitting**  
(pas assez de capacité)

12 neurones cachés



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

60 neurones cachés



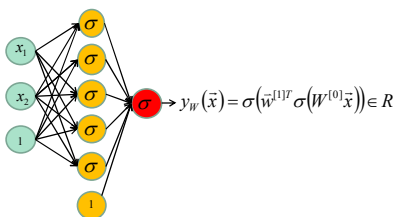
Classification non linéaire  
**Over fitting**  
(trop grande capacité)

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

38

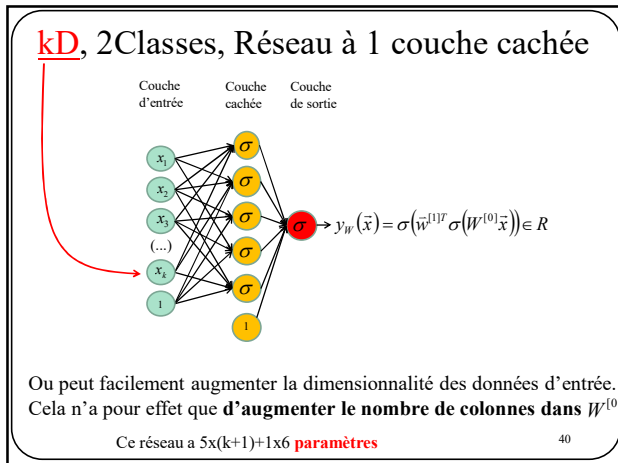
## 2D, 2Classes, Réseau à 1 couche cachée

Couche d'entrée    Couche cachée    Couche de sortie



39

39



40

---

---

---

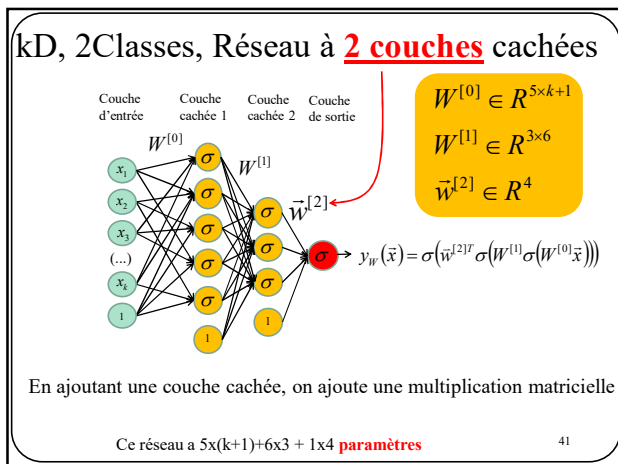
---

---

---

---

---



41

---

---

---

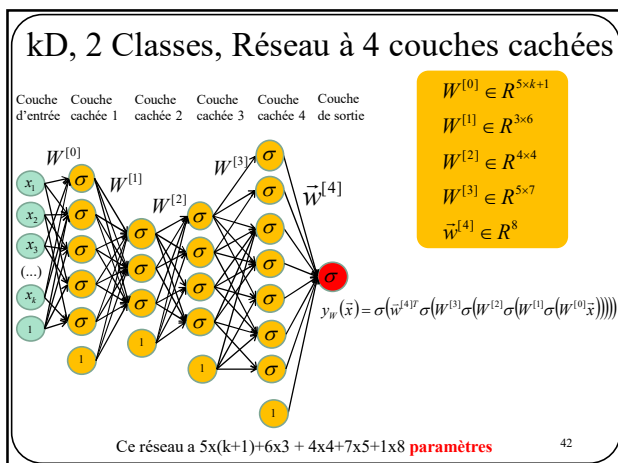
---

---

---

---

---



42

---

---

---

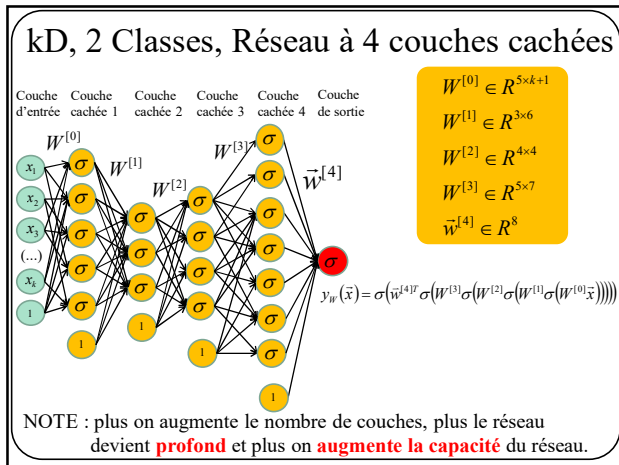
---

---

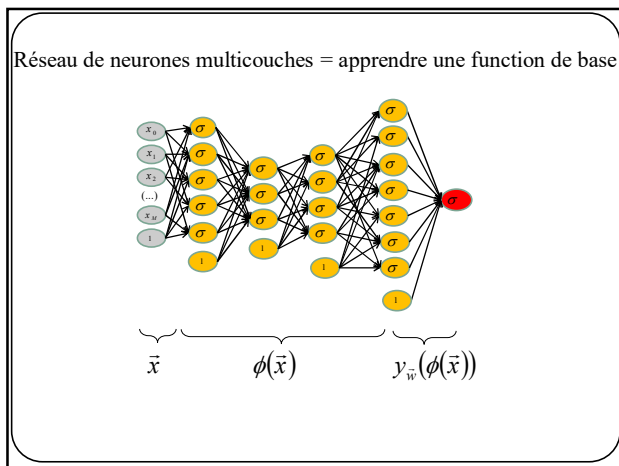
---

---

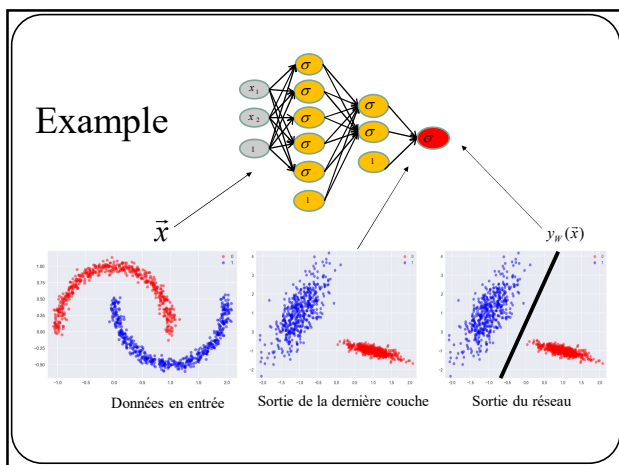
---



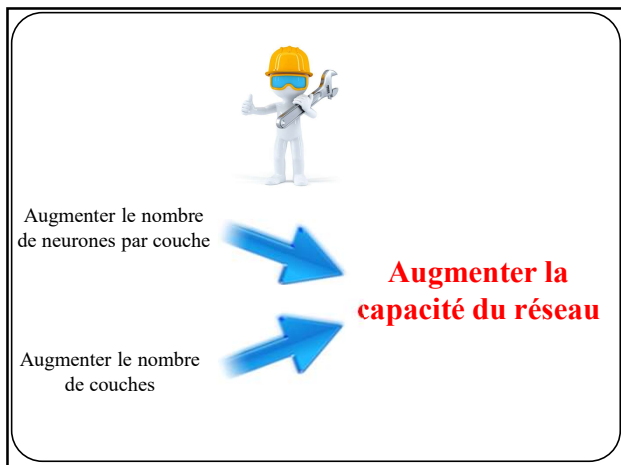
43



44



45



46

---

---

---

---

---

---

---



47

---

---

---

---

---

---

---



48

---

---

---

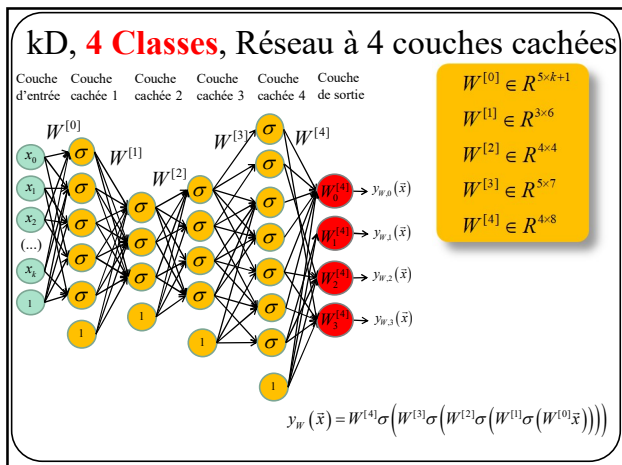
---

---

---

---





49

---

---

---

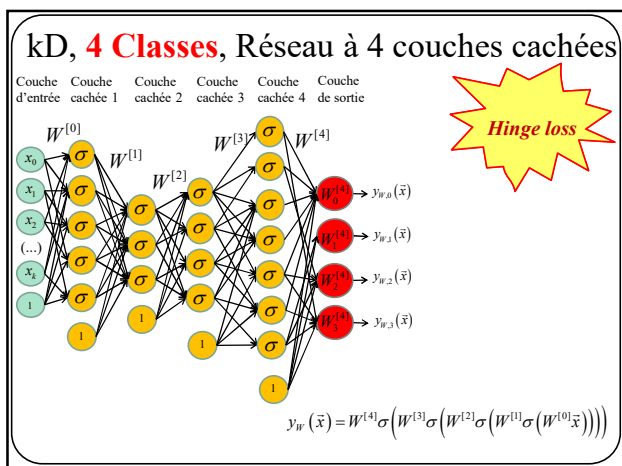
---

---

---

---

---



50

---

---

---

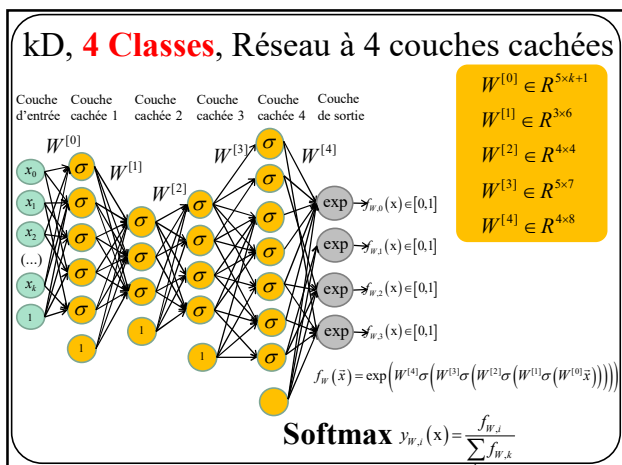
---

---

---

---

---



51

---

---

---

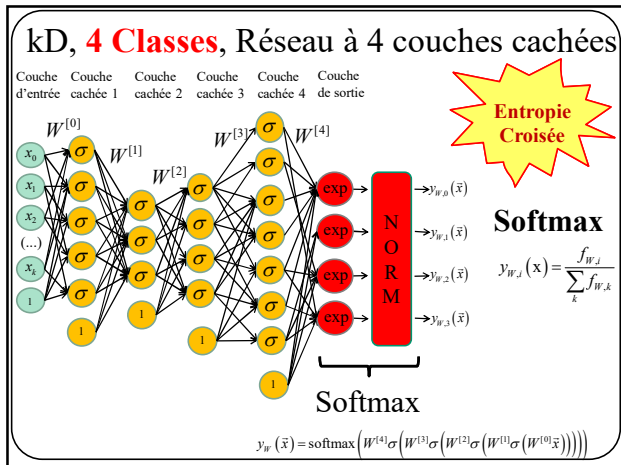
---

---

---

---

---



52

**Simulation**

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

53

**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
H3 = sigmoid(np.dot(W2,H2))
H3 = np.insert(H3,0,1) # Ajouter biais } Couche 3
y_pred = np.dot(W3,H3) } Couche sortie
```

**Forward pass**

54

## Comment optimiser les paramètres?

0- Partant de

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

Trouver une fonction de régularisation. En général

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

55

55

---

---

---

---

---

---

---

## 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*

56

---

---

---

---

---

---

---

## 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.

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

57

57

---

---

---

---

---

---

---

## Comment optimiser les paramètres?

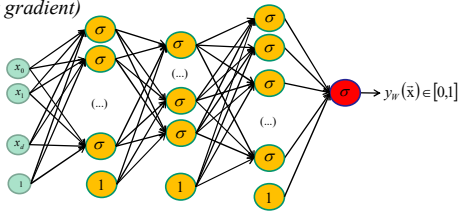
$$\frac{\partial (E_D(W) + \lambda R(W))}{\partial W_{a,b}^{[c]}} \Rightarrow \text{calculé à l'aide d'une rétropropagation}$$

58

58

## 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**

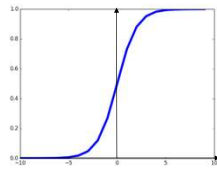
59

59

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

60

## Fonction d'activation



**Sigmoïde**

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

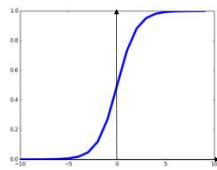
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 3 Problèmes :

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

61

## Fonction d'activation



**Sigmoïde**

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

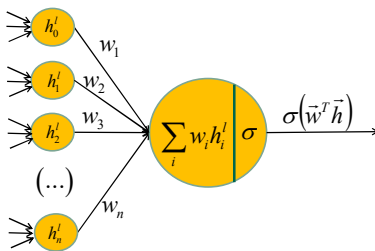
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 3 Problèmes :

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

62

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

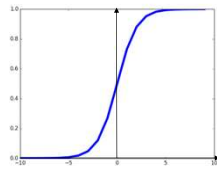


Le gradient par rapport à  $\vec{w}$  est ... **Positif? Négatif?**

Réponse : <https://rohanvarma.me/inputnormalization/>

63

## Fonction d'activation



**Sigmoïde**

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

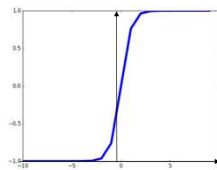
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

### 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.

64

## Fonction d'activation



**Tanh(x)**

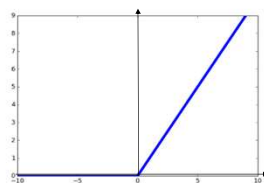
- Ramène les valeurs entre -1 et 1
- **Sortie centrée à zéro** 😊
- **Disparition du gradient** lorsque la fonction sature ☹️

[LeCun et al., 1991]

65

## Fonction d'activation

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



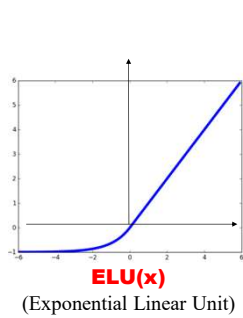
**ReLU(x)**  
(Rectified Linear Unit)

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

[Krizhevsky et al., 2012]

66

## Fonction d'activation



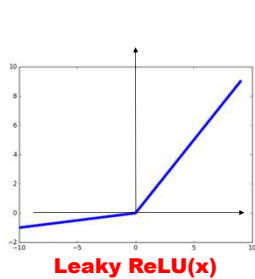
$$\text{ELU}(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha(e^x - 1) & \text{sinon} \end{cases}$$

- 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]

67

## Fonction d'activation



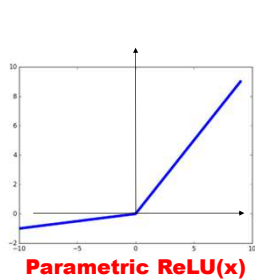
$$\text{LReLU}(x) = \max(0.01x, x)$$

- 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]

68

## 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]

69

## Fonction d'activation

Plusieurs autres fonctions d'activation ont été proposée :

- GELU (Gaussian Error Linear Unit)
- SiLU (Sigmoid Linear Unit)
- Swish
- GLU (Gated Linear Unit)
- ReGLU (Rectified Gated Linear Unit)
- GEGLU (Gaussian Error Gated Linear Unit)
- SwiGLU (Swish-Gated Linear Unit)
- Etc.

À vous de les découvrir!



[https://vitalab.github.io/blog/2024/08/20/new\\_activation\\_functions.html](https://vitalab.github.io/blog/2024/08/20/new_activation_functions.html)

70

---

---

---

---

---

---

---

## En pratique

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

71

---

---

---

---

---

---

---

## Les bonnes pratiques

72

---

---

---

---

---

---

---



## 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
    w = w - η[k] ∇E(xn)
  JUSQU'À ce que toutes les données
  soient bien classées ou k== MAX_ITER
        
```

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

**Optimisation par Batch**

```

Initialiser w
k=0
FAIRE k=k+1
  w = w - η[k] ∑i ∇E(xi)
  JUSQU'À ce que toutes les données
  soient bien classées ou k==MAX_ITER
        
```

73

---

---

---

---

---

---

---

---

## 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 w
k=0
FAIRE k=k+1
  FAIRE n=0 à N par sauts de MBS /*Mini-batch size*/
    w = w - η[k] ∑i=nn+MBS ∇E(xi)
  JUSQU'À ce que toutes les données soient bien classées ou
  k==MAX_ITER
        
```

TP1 TP2  
TP3 TP4

} Itération

74

---

---

---

---

---

---

---

---

## 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 w
k=0
FAIRE k=k+1
  FAIRE n=0 à N par sauts de MBS /*Mini-batch size*/
    w = w - η[k] ∑i=nn+MBS ∇E(xi)
  JUSQU'À ce que toutes les données sont bien classées ou
  k==MAX_ITER
        
```

} Epoch

75

---

---

---

---

---

---

---

---

Mini-batch = **vectorisation** de la  
propagation avant et de la rétro-  
propagation

76

---

---

---

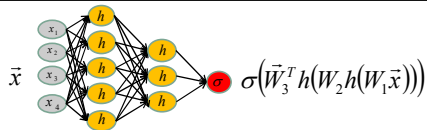
---

---

---

---

---



Propagation avant pour un réseau à 2 couches cachées (7 étapes)

Étape 2	$\vec{x}$	$\in \mathbb{R}^4$
	$W_1 \vec{x}$	$\in \mathbb{R}^5$
	$h(W_1 \vec{x})$	$\in \mathbb{R}^5$
	$W_2 h(W_1 \vec{x})$	$\in \mathbb{R}^3$
	$h(W_2 h(W_1 \vec{x}))$	$\in \mathbb{R}^3$
Étape 6	$\vec{W}_3^T (h(W_2 h(W_1 \vec{x})))$	$\in \mathbb{R}$
	$\sigma(\vec{W}_3^T (h(W_2 h(W_1 \vec{x}))))$	$\in \mathbb{R}$

77

---

---

---

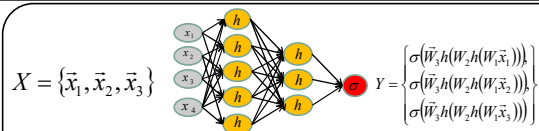
---

---

---

---

---



Propagation avant pour un réseau à 2 couches cachées (7 étapes)

POUR i allant de 0 à 2

$\vec{x} = X[i]$	$\in \mathbb{R}^4$
$W_1 \vec{x}$	$\in \mathbb{R}^5$
$h(W_1 \vec{x})$	$\in \mathbb{R}^5$
$W_2 h(W_1 \vec{x})$	$\in \mathbb{R}^3$
$h(W_2 h(W_1 \vec{x}))$	$\in \mathbb{R}^3$
$\vec{W}_3^T (h(W_2 h(W_1 \vec{x})))$	$\in \mathbb{R}$
$Y[i] = \sigma(\vec{W}_3^T (h(W_2 h(W_1 \vec{x}))))$	$\in \mathbb{R}$

Solution  
naïve et peu  
efficace

TP1

78

---

---

---

---

---

---

---

---

### Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que **PLUSIEURS multiplications matrice-vecteur** (exemple de la 2<sup>e</sup> étape, batch de 3)

$$\begin{aligned}
 W_1 \vec{x}_1 &= \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} \\
 W_1 \vec{x}_2 &= \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} d \\ e \\ f \\ g \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \\
 W_1 \vec{x}_3 &= \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} h \\ i \\ j \\ k \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix}
 \end{aligned}$$

TROIS multi. matrice-vecteur

79

---

---

---

---

---

---

---

---

### Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que **PLUSIEURS matrice-vecteur** (exemple de la 2<sup>e</sup> étape, batch de 3)

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

UNE multiplication matricielle

80

---

---

---

---

---

---

---

---

### Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que **PLUSIEURS produits scalaires** (exemple de la 6<sup>e</sup> étape, batch de 3)

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

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

TROIS produits scalaires

81

---

---

---

---

---

---

---

---

# Solution

Il est plus efficace d'effectuer **UNE multiplication matricielle** que PLUSIEURS produits scalaires (**exemple de la 6<sup>e</sup> étape, batch de 3**)

$$\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} = \begin{pmatrix} w_1a + w_2b + w_3c \\ w_1d + w_2e + w_3f \\ w_1g + w_2h + w_3i \end{pmatrix} = Y$$



82

---

---

---

---

---

---

---

---

# Vectorisation de la propagation avant

En résumé, lorsqu'on propage une « batch » de données

Au niveau <b>Neuronal</b> (batch = 1)	Multi. <b>Vecteur-Vecteur</b>	$\vec{w}^T \vec{x} = [w_1 \ w_2 \ w_3] \begin{pmatrix} a \\ b \\ c \end{pmatrix}$
Au niveau <b>Neuronal</b> (batch = 3)	Multi. <b>Vecteur-Matrice</b>	$\vec{w}^T X = [w_1 \ w_2 \ w_3] \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$
Au niveau <b>de la couche</b> (batch = 3)	Multi. <b>Matrice-Matrice</b>	$W X = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{pmatrix} a & d & h \\ b & e & i \\ c & f & j \end{pmatrix}$

83

---

---

---

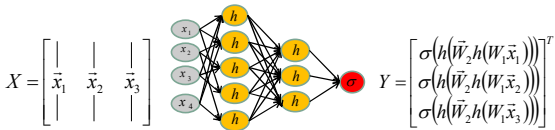
---

---

---

---

---



Vectoriser la rétropropagation

84

---

---

---

---

---

---

---

---

## Vectoriser la rétropropagation

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

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

En supposant qu'on connaît le gradient pour les 3 éléments de Y provenant de la sortie du réseau, comment faire pour propager le gradient vers  $\vec{w}$ ?

85

## Vectoriser la rétropropagation

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

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

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

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

86

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

Concentrons-nous sur  $w_1$

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

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

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

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

87

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

$$\vec{w}^T \quad X \quad Y$$

Concentrons-nous sur  $w_1$

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

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

$$w_1 \leftarrow w_1 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} a \\ d \\ g \end{bmatrix} \quad (\text{Puisqu'on a une batch de 3 éléments, on a 3 prédictions, et donc 3 gradients})$$

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

88

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

$$\vec{w}^T \quad X \quad Y$$

Donc en résumé ...

$$w_1 \leftarrow w_1 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

89

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

$$\vec{w}^T \quad X \quad Y$$

Et pour tous les poids

$$w_1 \leftarrow w_1 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

$$w_2 \leftarrow w_2 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} b \\ e \\ h \end{bmatrix}$$

$$w_3 \leftarrow w_3 - \eta \left[ \frac{\partial E_1}{\partial Y} \quad \frac{\partial E_2}{\partial Y} \quad \frac{\partial E_3}{\partial Y} \right] \begin{bmatrix} c \\ f \\ i \end{bmatrix}$$

90

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


$\vec{w}^T$ 
 $X$ 
 $Y$

Et pour tous les poids

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

$$\vec{w}^T \leftarrow \vec{w}^T - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} \frac{\partial Y_1}{\partial w_1} & \frac{\partial Y_1}{\partial w_2} & \frac{\partial Y_1}{\partial w_3} \\ \frac{\partial Y_2}{\partial w_1} & \frac{\partial Y_2}{\partial w_2} & \frac{\partial Y_2}{\partial w_3} \\ \frac{\partial Y_3}{\partial w_1} & \frac{\partial Y_3}{\partial w_2} & \frac{\partial Y_3}{\partial w_3} \end{bmatrix}$$

$$\vec{w}^T \leftarrow \vec{w}^T - \eta \frac{\partial E^T}{\partial Y} \frac{\partial Y}{\partial \vec{w}}$$



Matrice jacobienne

91

---

---

---

---

---

---

---

---

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

$W$ 
 $X$ 
 $Y$

Même chose pour 1 couche 5x4 et une batch de 3 données

$$W \leftarrow W^T - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y_1} & \frac{\partial E_2}{\partial Y_1} & \frac{\partial E_3}{\partial Y_1} \\ \frac{\partial E_1}{\partial Y_2} & \frac{\partial E_2}{\partial Y_2} & \frac{\partial E_3}{\partial Y_2} \\ \frac{\partial E_1}{\partial Y_3} & \frac{\partial E_2}{\partial Y_3} & \frac{\partial E_3}{\partial Y_3} \\ \frac{\partial E_1}{\partial Y_4} & \frac{\partial E_2}{\partial Y_4} & \frac{\partial E_3}{\partial Y_4} \\ \frac{\partial E_1}{\partial Y_5} & \frac{\partial E_2}{\partial Y_5} & \frac{\partial E_3}{\partial Y_5} \end{bmatrix} \begin{bmatrix} a & b & c & d \\ d & e & f & g \\ h & i & j & k \end{bmatrix}$$

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

92

---

---

---

---

---

---

---

---

## Vectorisation de la rétro-propagation

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

Au niveau neuronal	Multi. Vecteur-Matrice	$w_i \leftarrow w_i - \eta \frac{\partial E^T}{\partial Y} \frac{\partial Y}{\partial w_i}$ $w_i \leftarrow w_i - \eta \frac{\partial E^T}{\partial Y} x_i$
Au niveau de la couche	Multi. Matrice-Matrice	$W^T \leftarrow W^T - \eta \frac{\partial E^T}{\partial Y} \frac{\partial Y}{\partial W}$ $W^T \leftarrow W^T - \eta \frac{\partial E^T}{\partial Y} X$

93

---

---

---

---

---

---

---

---

# Vectorisation de l'entropie croisée

94

---

---

---

---

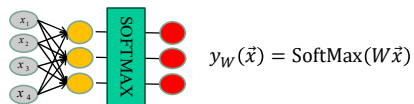
---

---

---

---

Rappel : entropie croisée, 1 donnée



$$L_{\vec{x}}(W) = - \sum_{k=1}^K t_{kn} \ln y_{W_k}(\vec{x})$$

$$= -\vec{t}^T \ln y_W(\vec{x})$$

$$\nabla_W L_{\vec{x}}(W) = (y_W(\vec{x}_n) - \vec{t}) \vec{x}^T$$

95

---

---

---

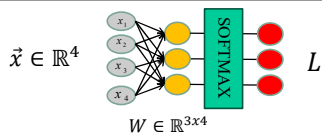
---

---

---

---

---



Propagation avant d'une donnée pour un réseau à 1 couche (3 étapes)

$\vec{x}$	$\in \mathbb{R}^4$
$W \vec{x}$	$\in \mathbb{R}^3$
$Y = SM(W \vec{x})$	$\in \mathbb{R}^3$
$L = -\vec{t}^T \ln Y$	$\in \mathbb{R}$

96

---

---

---

---

---

---

---

---



$\vec{x} = [1, 2, 3, 4]^T$ 

$W \in \mathbb{R}^{3 \times 4}$

$L = -\vec{t}^T \ln y_W(\vec{x})$ 

$Y$

**Exemple de perte:**

$$Y = \begin{bmatrix} 0.1 \\ 0.6 \\ 0.3 \end{bmatrix} \text{ et } \vec{t} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$L = -[0 \ 0 \ 1] \ln \begin{bmatrix} 0.1 \\ 0.6 \\ 0.3 \end{bmatrix}$$

$$= -[0 \ 0 \ 1] \begin{bmatrix} -2.3 \\ -0.5 \\ -1.2 \end{bmatrix}$$

$$= 1.2$$


---

---

---

---

---

---

---

---

97

$\vec{x} = [1, 2, 3, 4]^T$ 

$W \in \mathbb{R}^{3 \times 4}$

$Y = \begin{bmatrix} 0.1 \\ 0.6 \\ 0.3 \end{bmatrix}, \vec{t} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

**Exemple de gradient**

$$\nabla_W L = (Y - \vec{t}) \vec{x}^T$$

$$= \left( \begin{bmatrix} 0.1 \\ 0.6 \\ 0.3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) [1, 2, 3, 4]$$

$$= \begin{bmatrix} 0.1 \\ 0.6 \\ -0.7 \end{bmatrix} [1, 2, 3, 4]$$

$$= \begin{bmatrix} .1 & .2 & .3 & .4 \\ .6 & 1.2 & 1.8 & 2.4 \\ -.3 & -.6 & -.9 & -1.2 \end{bmatrix} \in \mathbb{R}^{3 \times 4}$$


---

---

---

---

---

---

---

---

98

$X = \{\vec{x}_1, \vec{x}_2\}$ 

$W \in \mathbb{R}^{3 \times 4}$

$Y$

**Approche par batch**

Solution par batch

$$\begin{aligned} X &\in \mathbb{R}^{4 \times 2} \\ W \vec{x} &\in \mathbb{R}^{3 \times 2} \\ Y = SM(W \vec{x}) &\in \mathbb{R}^{3 \times 2} \\ L = -T^T \ln Y &\in \mathbb{R}^2 \end{aligned}$$

TP1

---

---

---

---

---

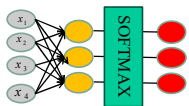
---

---

---

99

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$



$$L = -T^T \ln Y$$

$$Y = \begin{bmatrix} 0.1 & 0.7 \\ 0.6 & 0.25 \\ 0.3 & 0.05 \end{bmatrix}$$

$$\text{et } T = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$Y = \begin{bmatrix} 0.1 & 0.7 \\ 0.6 & 0.25 \\ 0.3 & 0.05 \end{bmatrix}$$

**Exemple de perte:**

$$L = - \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \ln \begin{bmatrix} 0.1 & 0.7 \\ 0.6 & 0.25 \\ 0.3 & 0.05 \end{bmatrix}$$

$$= - \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -2.3 & -0.4 \\ -0.5 & -1.4 \\ -1.2 & -3.0 \end{bmatrix}$$

$$= \begin{bmatrix} 1.2 & 0.4 \end{bmatrix}$$


---

---

---

---

---

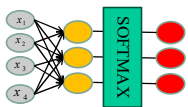
---

---

---

100

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$



$$Y = \begin{bmatrix} 0.1 & 0.7 \\ 0.6 & 0.25 \\ 0.3 & 0.05 \end{bmatrix}$$

$$\text{et } T = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$$

**Exemple de gradient**

$$\nabla_W L = (Y - T) X$$

$$= \left( \begin{bmatrix} 0.1 & 0.7 \\ 0.6 & 0.25 \\ 0.3 & 0.05 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} \right) \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 0.1 & -0.3 \\ 0.6 & 0.25 \\ -0.7 & 0.05 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} -1.4 & -1.6 & -1.8 & -2 \\ 1.8 & 2.7 & 3.6 & 4.4 \\ -5 & -1.1 & -1.7 & -2.4 \end{bmatrix}$$


---

---

---

---

---

---

---

---

101

## Pour plus de détails:

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

---

---

---

---

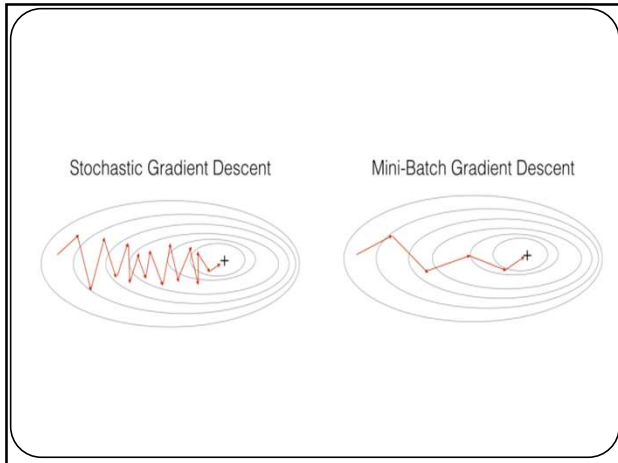
---

---

---

---

102



103

---

---

---

---

---

---

---

---

Comment initialiser un réseau de neurones?

$$W = ?$$

104

---

---

---

---

---

---

---


---

**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.

105

---

---

---

---

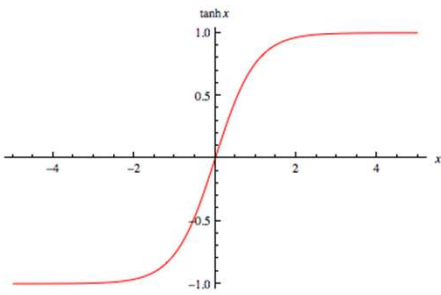
---

---

---

---

Rappel tanh...



---

---

---

---

---

---

---

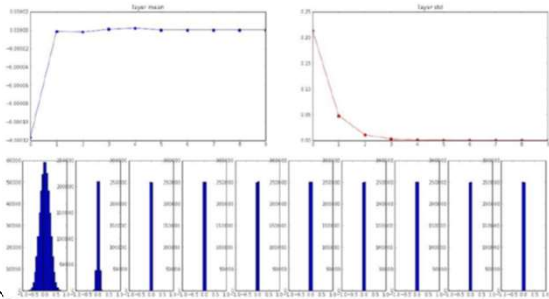
---

106

Histogrammes des valeurs de sortie des 10 couches.

input layer has mean 0.000527 and std 0.990300  
hidden layer 1 has mean -0.000117 and std 0.213001  
hidden layer 2 has mean -0.000001 and std 0.047551  
hidden layer 3 has mean -0.000002 and std 0.019630  
hidden layer 4 has mean 0.000001 and std 0.002379  
hidden layer 5 has mean 0.000002 and std 0.000532  
hidden layer 6 has mean -0.000000 and std 0.000119  
hidden layer 7 has mean 0.000000 and std 0.000020  
hidden layer 8 has mean -0.000000 and std 0.000006  
hidden layer 9 has mean 0.000000 and std 0.000001  
hidden layer 10 has mean -0.000000 and std 0.000000

L'activation des couches  
devient zéro!



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

---

---

---

---

---

---

---

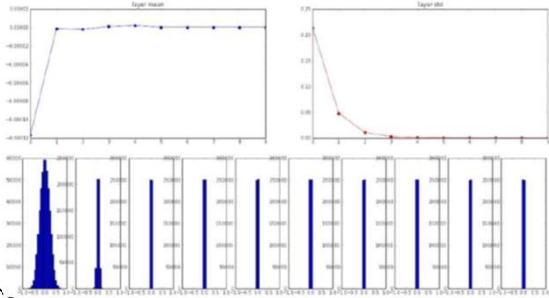
---

107

Histogrammes des valeurs de sortie des 10 couches.

input layer has mean 0.000527 and std 0.990300  
hidden layer 1 has mean -0.000117 and std 0.213001  
hidden layer 2 has mean -0.000001 and std 0.047551  
hidden layer 3 has mean -0.000002 and std 0.019630  
hidden layer 4 has mean 0.000001 and std 0.002379  
hidden layer 5 has mean 0.000002 and std 0.000532  
hidden layer 6 has mean -0.000000 and std 0.000119  
hidden layer 7 has mean 0.000000 and std 0.000020  
hidden layer 8 has mean -0.000000 and std 0.000006  
hidden layer 9 has mean 0.000000 and std 0.000001  
hidden layer 10 has mean -0.000000 and std 0.000000

Vous voyez ce qui arrive  
à la **forward pass** et la  
**rétropropagation**?



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

---

---

---

---

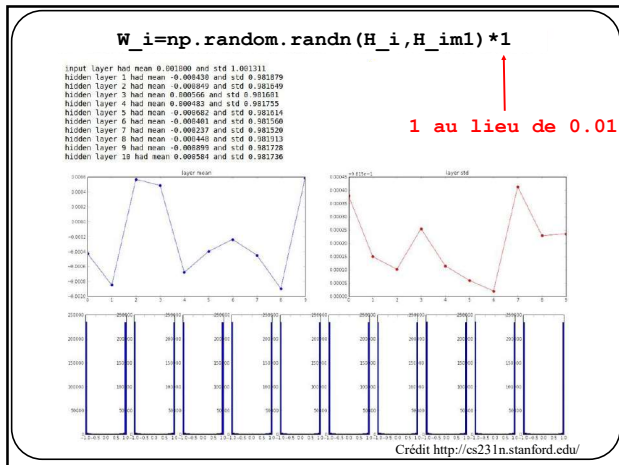
---

---

---

---

108



109

---

---

---

---

---

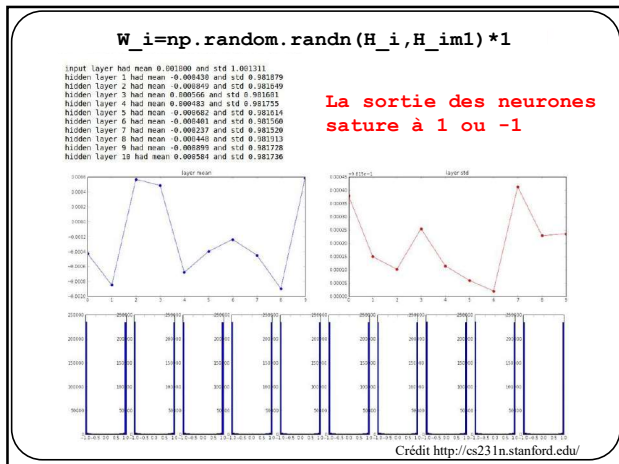
---

---

---

---

---



110

---

---

---

---

---

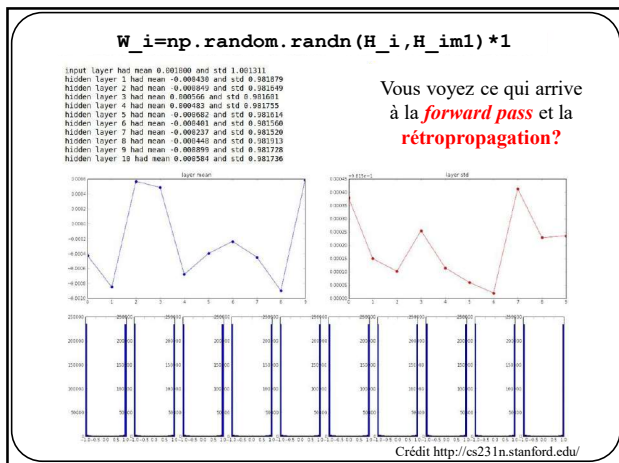
---

---

---

---

---



111

---

---

---

---

---

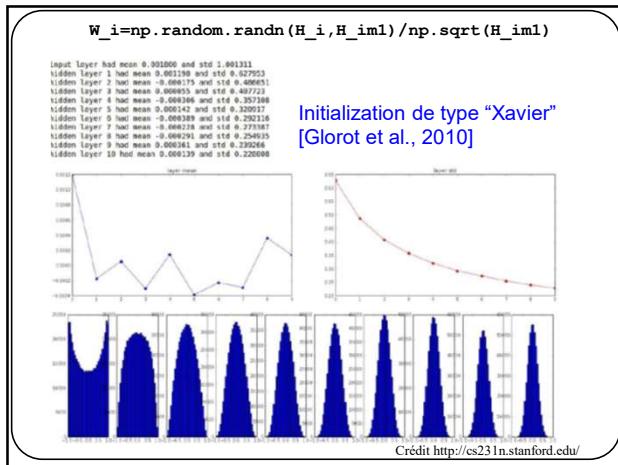
---

---

---

---

---



112

---

---

---

---

---

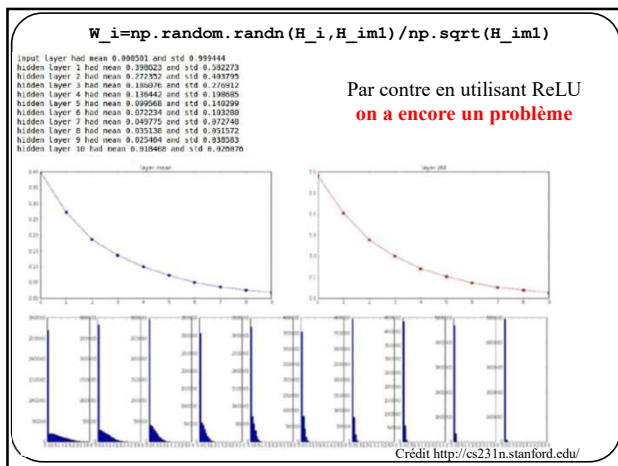
---

---

---

---

---



113

---

---

---

---

---

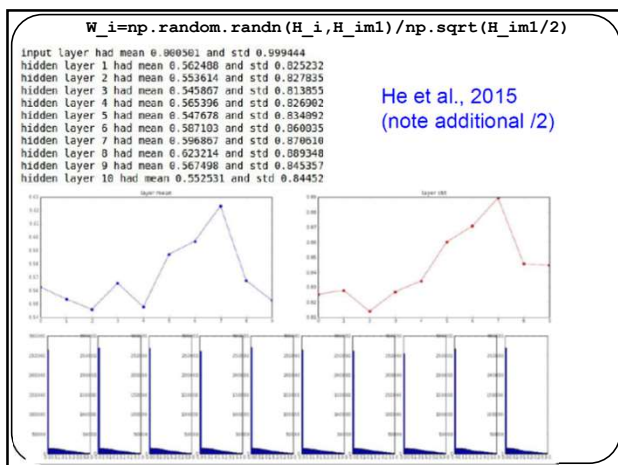
---

---

---

---

---



114

---

---

---

---

---

---

---

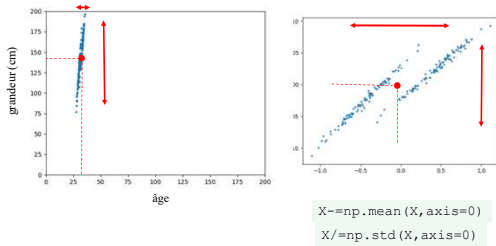
---

---

---

## Prétraitement des données

### Centrer et normaliser les données d'entrée



115

## Les « sanity checks » ou vérifications diligentes

116

### *Sanity checks*

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

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

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

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

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


117

## Sanity checks

1. Toujours s'assurer qu'une initialisation 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
```



The screenshot shows a Jupyter Notebook cell with the following code and output:

```
model = init_two_layer_model(x_train, 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
```

The output of the cell is:

```
2.30261216167
```

Annotations on the image explain the code and output:

- A red arrow points from the text "disable regularization" to the value `0.0` in the code.
- A red arrow points from the text "loss ~2.3. 'correct' for 10 classes" to the output value `2.30261216167`.
- A red arrow points from the text "returns the loss and the gradient for all parameters" to the variable `grad` in the code.

A watermark "cs231n.stanford.edu" is visible in the bottom right corner.

118

### 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, le3l) # crank up regularization
print loss
```

3.06859716482 ← loss went up, good. (sanity check)

Crédit <http://cs231n.stanford.edu>

119

## 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!

[illegible]

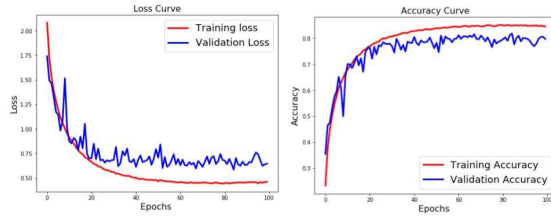
Credit <http://cs231n.stanford.edu>

120



## Sanity checks

4. 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>

121

---

---

---

---

---

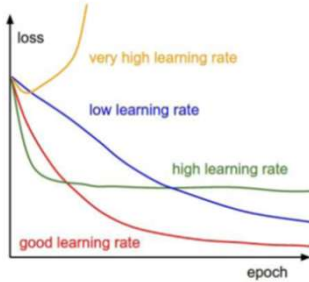
---

---

---

## Sanity checks

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



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

122

---

---

---

---

---

---

---

---

## Sanity checks

5. Toujours vérifier la validité d'un gradient

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

Rappel

Approximation numérique de la dérivée

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

123

---

---

---

---

---

---

---

---

## Sanity checks

5. 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

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

124

---

---

---

---

---

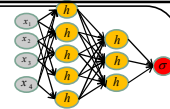
---

---

---

## Vérification du gradient

(exemple)



**W**                      **W+h**                      **gradient W**

$W_{00} = 0.34$	$W_{00} = 0.34$	$+0.0001$	$-2.5 = (1.25322 - 1.25347) / 0.0001$
$W_{01} = -1.11$	$W_{01} = -1.11$		
$W_{02} = 0.78$	$W_{02} = 0.78$		
...	...		
$W_{20} = -3.1$	$W_{20} = -3.1$		
$W_{21} = -1.5$	$W_{21} = -1.5$		
$W_{22} = 0.33$	$W_{22} = 0.33$		

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

125

---

---

---

---

---

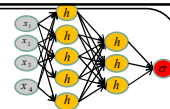
---

---

---

## Vérification du gradient

(exemple)



**W**                      **W+h**                      **gradient W**

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

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

126

---

---

---

---

---

---

---

---

### Vérification du gradient

(exemple)

W	W+h	gradient W
$W_{00} = 0.34$	$W_{00} = 0.34$	-2.5
$W_{01} = -1.11$	$W_{01} = -1.11$	0.6
$W_{02} = 0.78$	$W_{02} = 0.78 + 0.0001$	$0.0 = (1.25347 - 1.25347) / 0.0001$
...	...	...
$W_{20} = -3.1$	$W_{20} = -3.1$	1.1
$W_{21} = -1.5$	$W_{21} = -1.5$	1.3
$W_{22} = 0.33$	$W_{22} = 0.33$	-2.1

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

127

---

---

---

---

---

---

---

---

### Vérification du gradient

(exemple)

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

$E(W) = 1.25347$

128

---

---

---

---

---

---

---

---

### 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

Vérification réussie

129

---

---

---

---

---

---

---

---

Autre bonne pratique

# Dropout

130

---

---

---

---

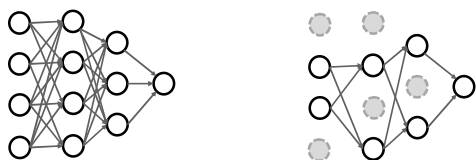
---

---

---

## Dropout

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



Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

131

---

---

---

---

---

---

---

## Dropout

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

132

---

---

---

---

---

---

---

## 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/>

133

## Dropout

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

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

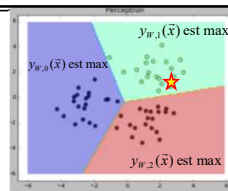
$$pred = y_w(\vec{x}, Z)$$

↑  
masque aléatoire

134

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

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



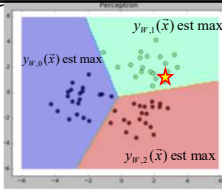
**Si on lance le modèle 10 fois, on aura 10 réponses différentes**

[ 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]

135

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]

136

---

---

---

---

---

---

---

---

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 l'espérance mathématique!**

137

---

---

---

---

---

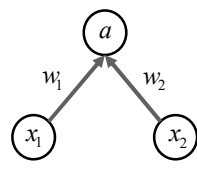
---

---

---

### Regardons pour un neurone

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



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

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

138

---

---

---

---

---

---

---

---

```
*** 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/>

139

---

---

---

---

---

---

---

---

NOTE

Au tp2, vous implanterez un **dropout inverse**. À vous de le découvrir!

140

---

---

---

---

---

---

---

---

Descente de gradient  
**version améliorée**

141

---

---

---

---

---

---

---

---

## Descente de gradient

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

142

---

---

---

---

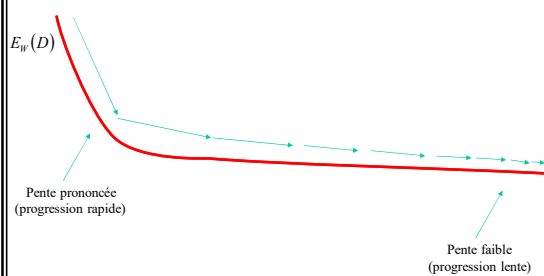
---

---

---

## Descente de gradient : **problème**

Progrès quasi nul lorsque la pente est très faible



143

---

---

---

---

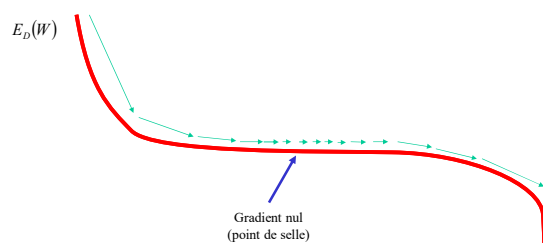
---

---

---

## Descente de gradient : **problème**

Les points de selles sont fréquents en haute dimension



144

---

---

---

---

---

---

---



## 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?

145

---

---

---

---

---

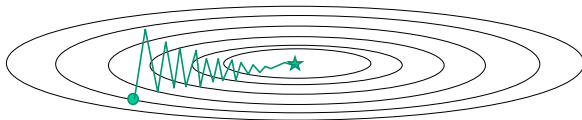
---

---

## 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.**



146

---

---

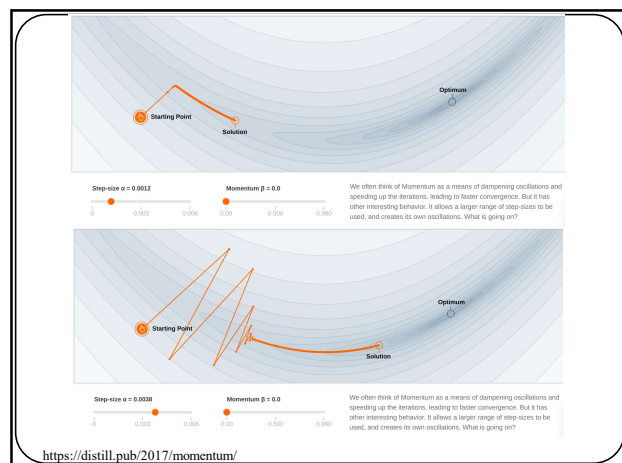
---

---

---

---

---



147

---

---

---

---

---

---

---

## Descente de gradient + **Momentum**

Descente de gradient  
 $E_D(W)$  stochastique

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

Descente de gradient  
stochastique + **Momentum**

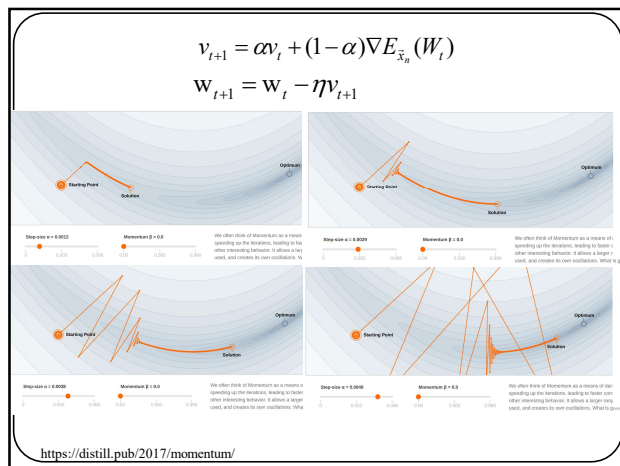
$$v_{t+1} = \alpha v_t + (1 - \alpha) \nabla E_{\tilde{x}_n}(w_t)$$

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

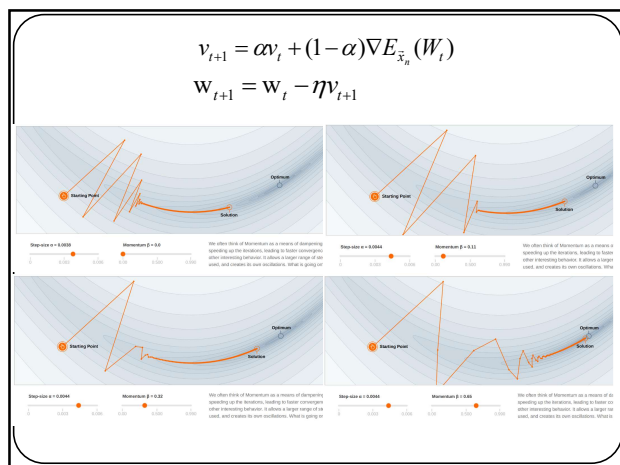
Provient de l'équation de la vitesse

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

148



149



150

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

Descente de gradient  
stochastique

**AdaGrad**

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

151

---

---

---

---

---

---

---

---

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

Descente de gradient  
stochastique

**AdaGrad**

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \eta \nabla E_{\tilde{x}_n}(\mathbf{w}_t) \\ dE_t &= \nabla E_{\tilde{x}_n}(\mathbf{w}_t) \\ m_{t+1} &= m_t + |dE_t| \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{\underbrace{m_{t+1} + \varepsilon}} dE_t \end{aligned}$$

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

152

---

---

---

---

---

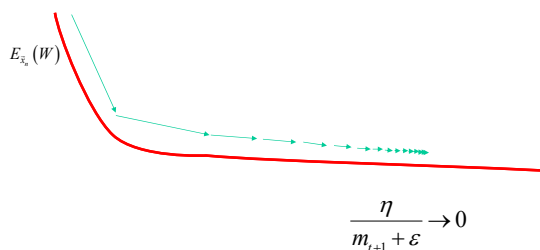
---

---

---

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

Qu'arrive-t-il à long terme?



153

---

---

---

---

---

---

---

---

## RMSProp (AdaGrad amélioré)

### AdaGrad

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

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

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

### RMSProp

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

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

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

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

154

---

---

---

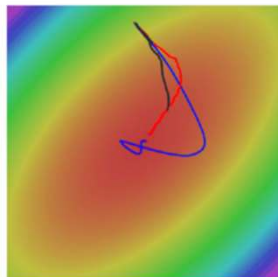
---

---

---

---

---



— SGD  
— SGD+Momentum  
— RMSProp

155

---

---

---

---

---

---

---

---

## Adam (Combo entre Momentum et RMSProp)

### Momentum

$$v_{t+1} = \rho v_t + \nabla E_{\tilde{x}_n}(\mathbf{w}_t)$$

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

### Adam

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

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

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

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

156

---

---

---

---

---

---

---

---

## Adam (Combo entre Momentum et RMSProp)

**Momentum**

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

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

**Adam**

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

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

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

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

*Momentum*

157

## Adam (Combo entre Momentum et RMSProp)

**RMSProp**

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

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

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

**Adam**

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

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

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

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

*RMSProp*

158

## Adam (Version complète)

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

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

for t=1 à num\_iterations  
for n=0 à N

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

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

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

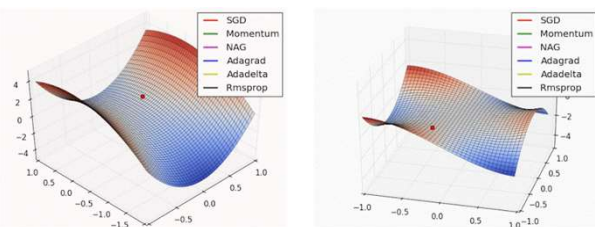
$$v_{t+1} = \frac{v_{t+1}}{1 - \beta_1^t}, m_{t+1} = \frac{m_{t+1}}{1 - \beta_2^t}$$

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

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

159

## Illustrations



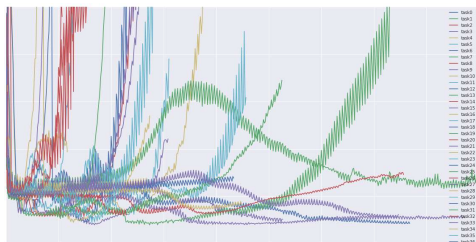
À voir sur :

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

160

## Autre excellent survol

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



161