

Réseaux de neurones

IFT 725

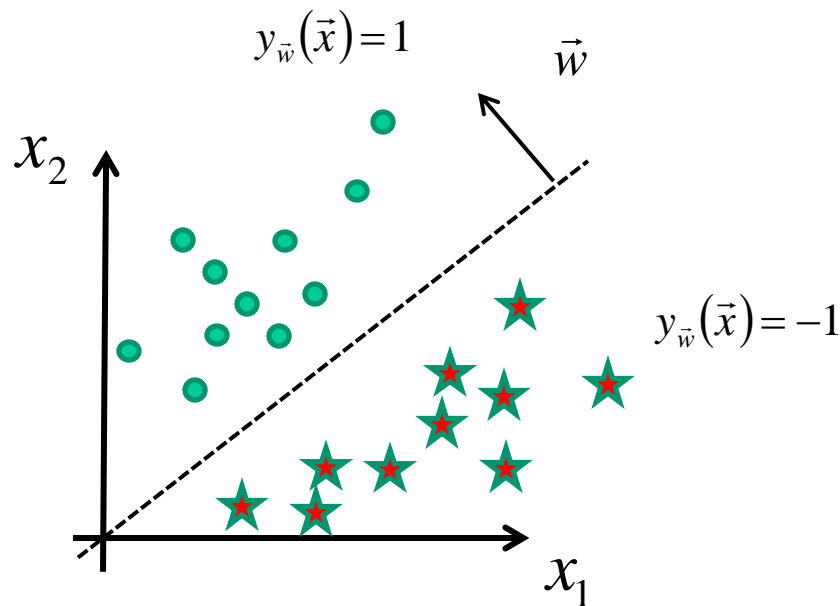
Réseaux de neurones multicouches

Par

Pierre-Marc Jodoin

Séparation linéaire

(2D et 2 classes)



biais **poids**

$$\begin{aligned} y_{\vec{w}}(\vec{x}) &= w_0 + w_1 x_1 + w_2 x_2 \\ &= w_0 + \vec{w}^T \vec{x} \\ &= \vec{w}'^T \vec{x}' \end{aligned}$$

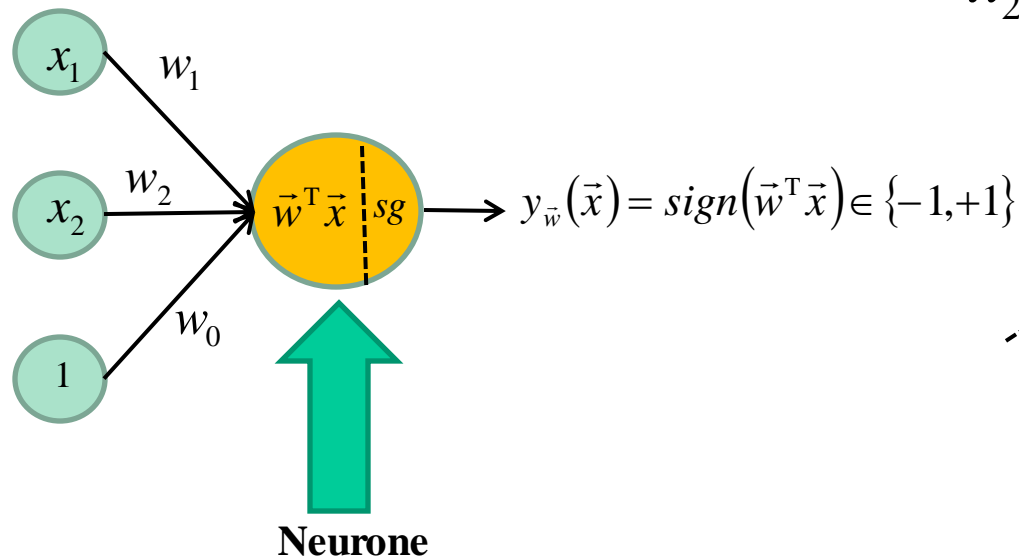
Par simplicité

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

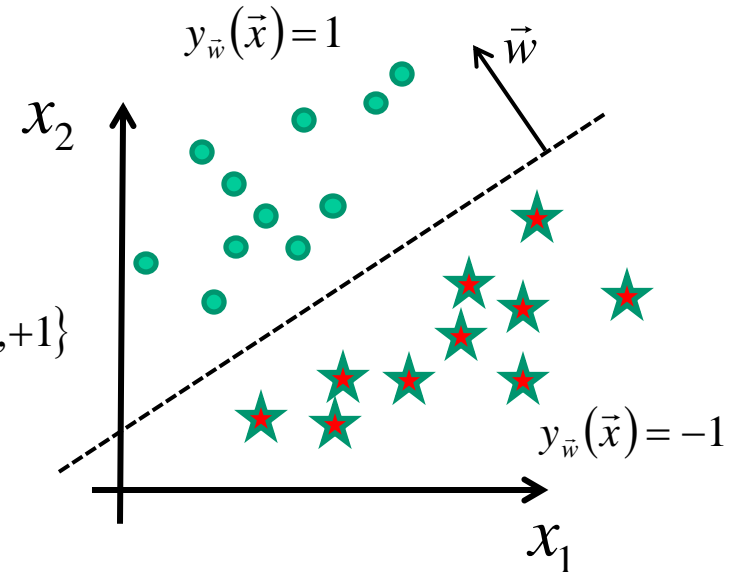
1. Plus besoin de données d'entraînement (vs. nearest neighbor p.e.)
2. Classification est très rapide (**produit scalaire** entre 2 vecteurs)

Perceptron (réseau de neurones)

(2D et 2 classes)



Produit scalaire + fonction d'activation



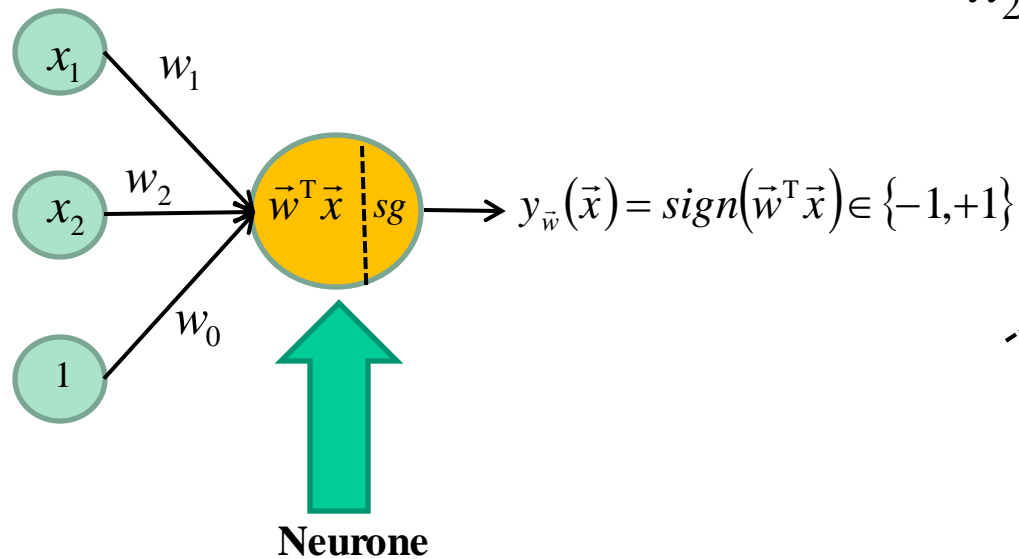
Fonction de coût perceptron (loss)
et gradient

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

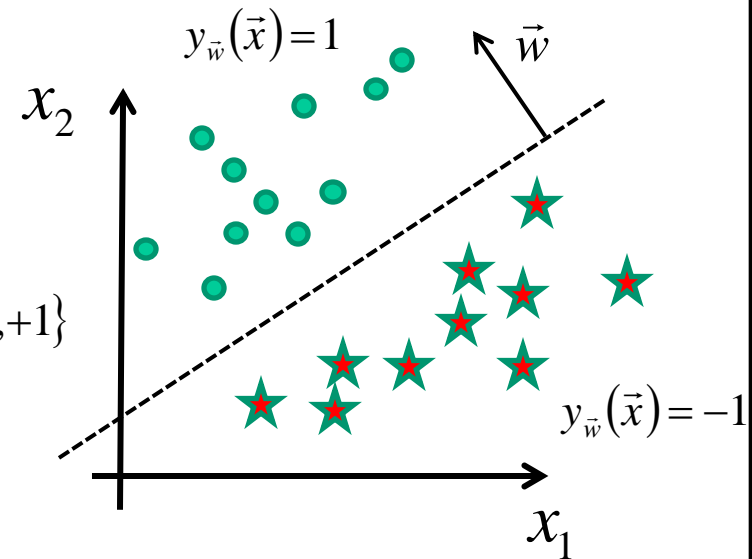
$$\nabla E_D(\vec{w}) = \sum_{\vec{x}_n \in M} -t_n \vec{x}_n$$

Hinge Loss

(2D et 2 classes)



Produit scalaire + fonction d'activation



Fonction de coût SVM (*hinge loss*)
et gradient

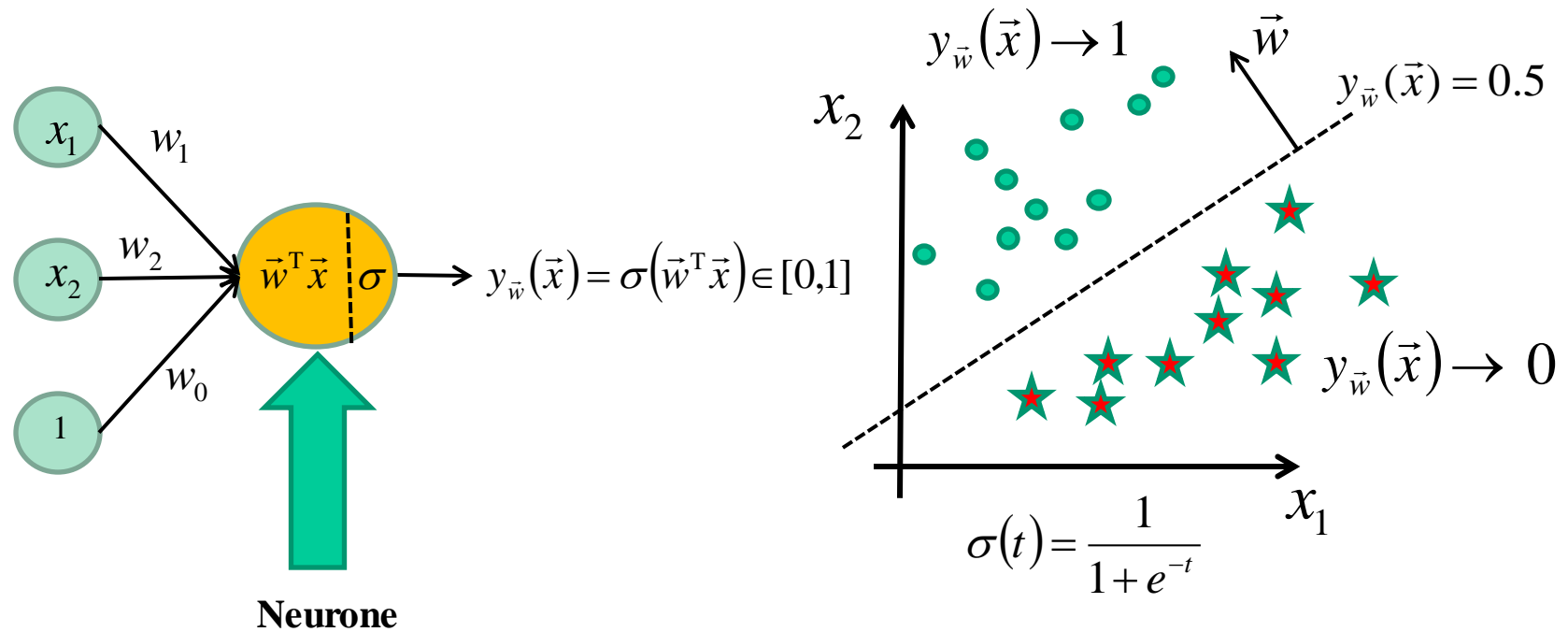
$$E_D(\vec{w}) = \sum_{n=1}^N \max(0, 1 - t_n \vec{w}^T \vec{x}_n)$$

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

Régression logistique

(2D, 2 classes)

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

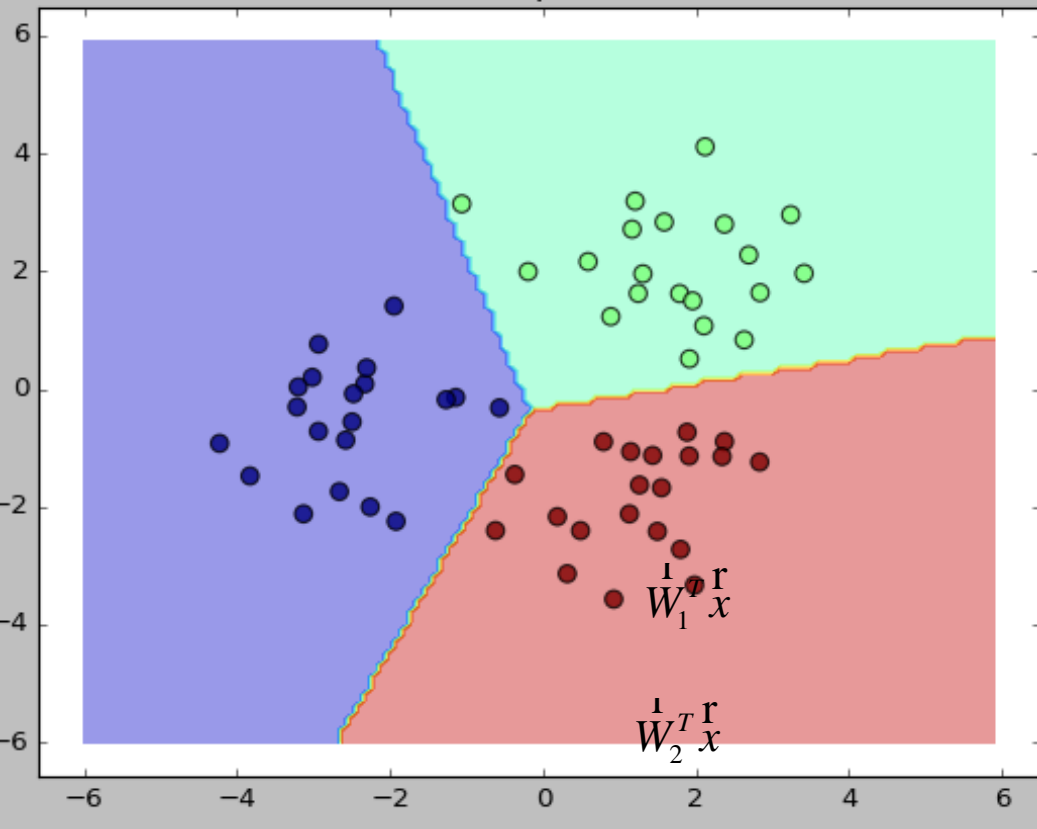


Fonction de coût (*loss*)
et gradient

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

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

Perceptron



sse

$y_{w,1}(\vec{x})$ est max

$y_{w,0}(\vec{x})$ est max

$y_{w,2}(\vec{x})$ est max

$\vec{r}_i(\vec{x})$

1 $w_{2,0}$

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

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

Perceptron Multiclasse

 $y_{w,1}(\vec{x})$ est max

 $y_{w,0}(\vec{x})$ est max

 $y_{w,2}(\vec{x})$ est max

Exemple

(1.1, -2.0)

 $y_w(\vec{x}) =$

$$\begin{bmatrix} -2 & -3.6 & 0.5 \\ -4 & 2.4 & 4.1 \\ -6 & 4 & -4.9 \end{bmatrix} \begin{bmatrix} 1 \\ 1.1 \\ -2 \end{bmatrix}$$

$$= \begin{bmatrix} -6.9 \\ -9.6 \\ 8.2 \end{bmatrix} \begin{matrix} \text{Classe 0} \\ \text{Classe 1} \\ \text{Classe 2} \end{matrix}$$

Perceptron Multiclasse

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

$$E_D(W) = \sum_{\substack{\mathbf{r} \\ x_n \in M}} \left(\bar{W}_j^T \mathbf{r} x_n - \bar{W}_{t_n}^T \mathbf{r} x_n \right)$$

Somme sur l'ensemble des
données mal classées

Score de la bonne classe

Score de la classe faussement
prédite par le modèle

$$\frac{\partial E}{\partial \bar{W}_j} = \vec{x}_n$$

$$\frac{\partial E}{\partial \bar{W}_{t_n}} = -\vec{x}_n$$

Perceptron Multiclasse

Fonction de coût (*Perceptron loss*)

$$E_D(W) = \sum_{\vec{x}_n \in M} \underbrace{\left(W_j^T \vec{x}_n - W_{t_n}^T \vec{x}_n \right)}_{E_{\vec{x}_n}}$$

$$\nabla_{W_j} E_{\vec{x}_n} = \vec{x}_n$$

$$\nabla_{W_{t_n}} E_{\vec{x}_n} = -\vec{x}_n$$

$$\nabla_{W_i} E_{\vec{x}_n} = 0 \quad \nabla i \neq j \text{ et } t_n$$

Perceptron Multiclasse

Descente de gradient stochastique (version naïve, batch_size = 1)

Initialiser \mathbf{W}

k=0, i=0

DO k=k+1

FOR n = 1 to N

$j = \operatorname{argmax} \mathbf{W}^T \vec{x}_n$

IF $j \neq t_i$ THEN /* donnée mal classée*/

$$\vec{w}_j = \vec{w}_j - \eta \vec{x}_n$$

$$\vec{w}_{t_n} = \vec{w}_{t_n} + \eta \vec{x}_n$$

UNTIL toutes les données sont bien classées.

Perceptron Multiclasse

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!

Perceptron Multiclasse

Exemple d'entraînement ($\eta=1$)

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

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

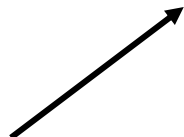
$$\vec{w}_2 \leftarrow \vec{w}_2 - \vec{x}_n \quad \begin{bmatrix} -6.0 \\ 4.0 \\ -4.9 \end{bmatrix} - \begin{bmatrix} 1 \\ 0.4 \\ -1 \end{bmatrix} = \begin{bmatrix} -7.0 \\ 3.6 \\ -3.9 \end{bmatrix}$$

Hinge Multiclasse

Fonction de coût (*Hinge loss* ou *SVM loss*)

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

Somme sur l'ensemble des
données



Score d'une mauvaise classe



Score de la bonne classe



Hinge Multiclasse

Fonction de coût (*Hinge loss* ou *SVM loss*)

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

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

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

Hinge Multiclasse

Descente de gradient stochastique (version naïve, batch_size = 1)

Initialiser \mathbf{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

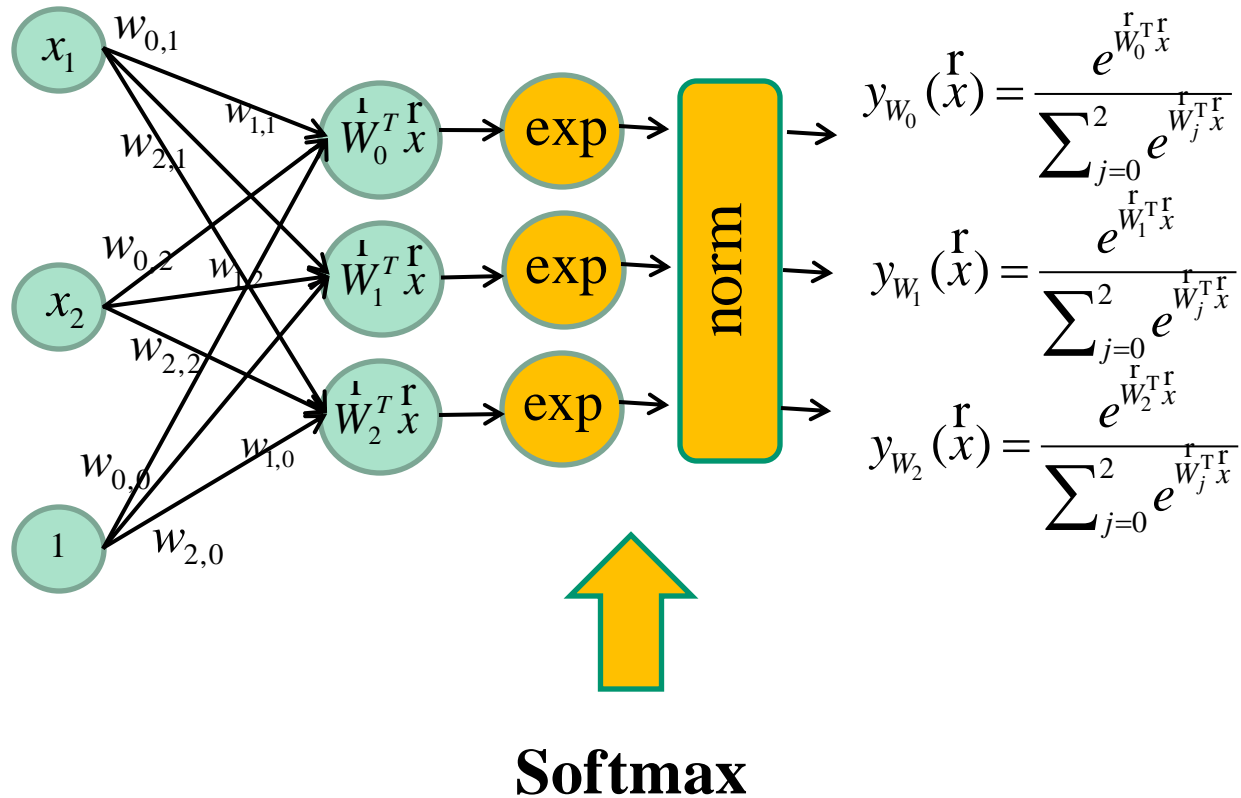
IF $\vec{w}_{t_n}^T \vec{x}_n < \vec{w}_j^T \vec{x}_n + 1$ AND $j \neq t_n$ THEN

$$\vec{w}_j = \vec{w}_j - \eta \vec{x}_n$$

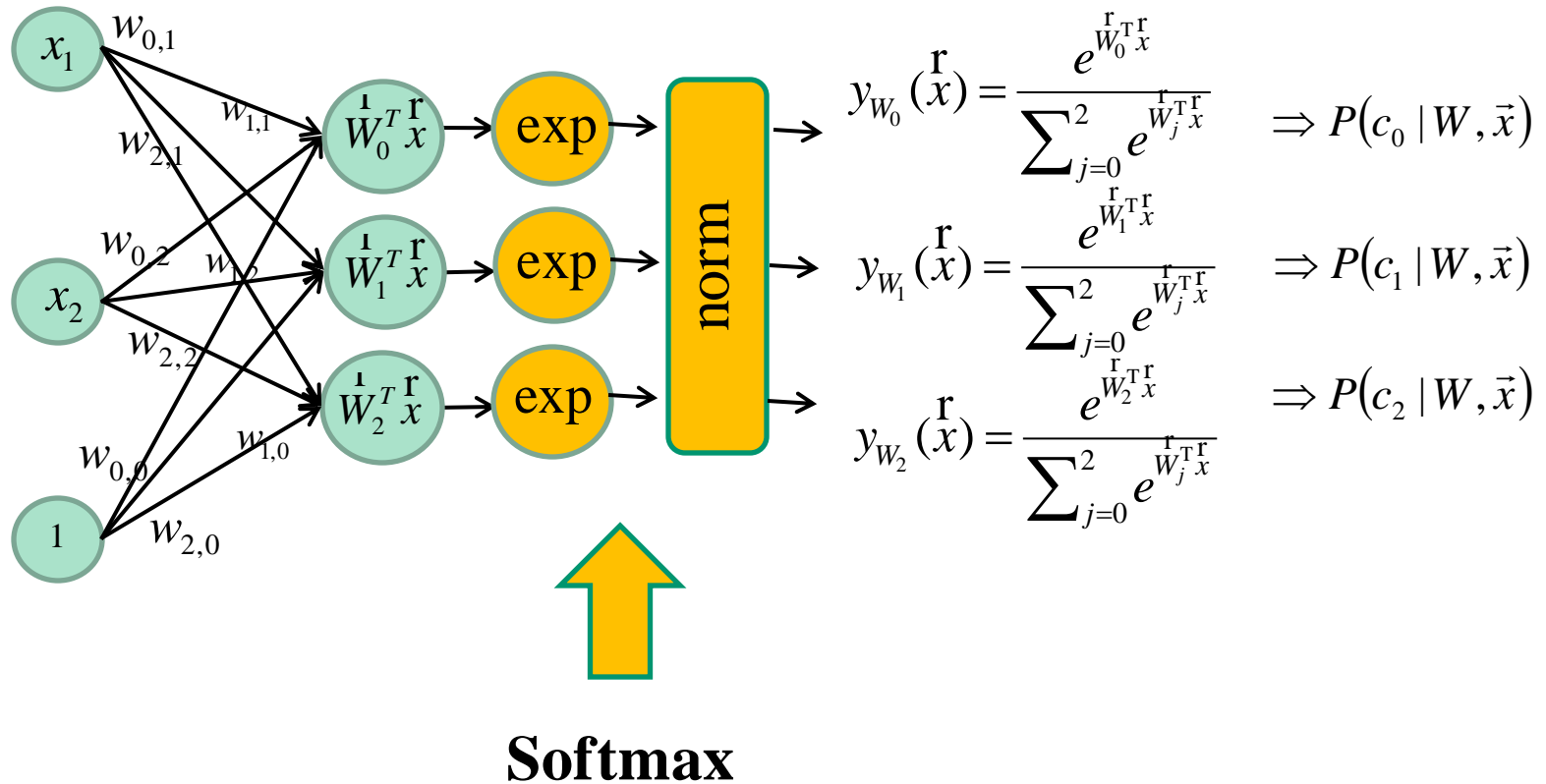
UNTIL toutes les données sont bien classées.

Au TP1, implanter la version naïve  + la version vectorisée
sans boucles for

Régression logistique multiclasse



Régression logistique multiclasse



airplane

automobile

bird

cat

deer

dog

frog

horse

ship

truck



Étiquettes de classe : one-hot vector

Régression logistique multiclasse

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

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

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

Tous les détails du gradient de l'entropie croisée :

info.usherbrooke.ca/pmjodoin/cours/ift603/softmax_grad.html

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

$$\mathbf{r}_x = \begin{bmatrix} -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}, t = 2$$

$$\begin{bmatrix} 0.0 & 0.01 & -0.05 & 0.1 & 0.05 \\ 0.2 & 0.7 & 0.2 & 0.05 & 0.16 \\ -0.3 & 0.0 & -0.45 & -0.2 & 0.03 \end{bmatrix} \begin{bmatrix} 1 \\ -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}$$

W

x

Hinge loss

Score

$$\begin{bmatrix} -2.85 \\ 0.86 \\ 0.28 \end{bmatrix}$$

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

Entropie croisée

Score

$$\begin{bmatrix} -2.85 \\ 0.86 \\ 0.28 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 0.06 \\ 2.36 \\ 1.32 \end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix} 0.02 \\ 0.63 \\ 0.35 \end{bmatrix}$$

(Softmax)

$$-\ln(0.35) = \mathbf{0.452}$$

Maximum *a posteriori*

Régularisation

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

Diagram illustrating the components of the Maximum *a posteriori* equation:

- Constante** (blue arrow) points to λ .
- Fonction de perte** (red arrow) points to $E_D(W)$.
- Régularisation** (green arrow) points to $R(W)$.

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

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

Optimisation par *Batch*

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

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

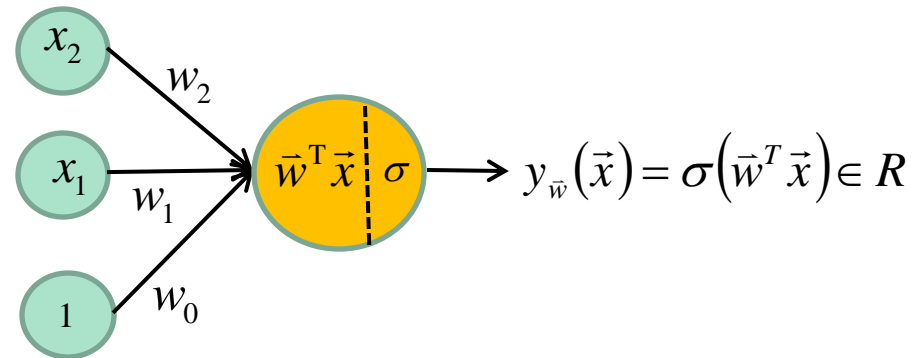
Maintenant, rendons le réseau

profond

profond

Maintenant, rendons le réseau

2D, 2Classes, Régression logistique linéaire

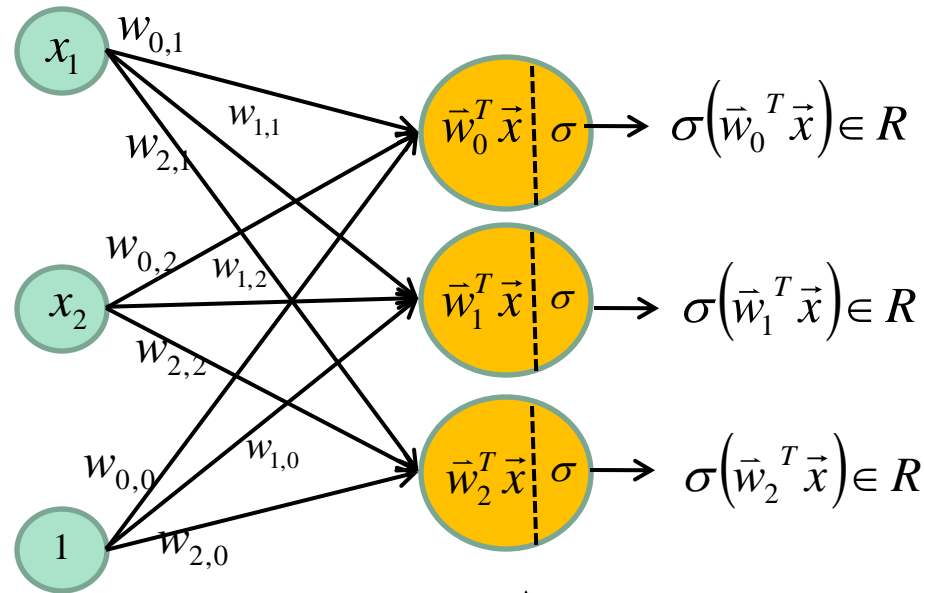


Couche d'entrée
(3 « neurones »)

Couche de sortie
(1 neurone avec sigmoïde)

2D, 2Classes, Réseau à 1 couche cachée

Maintenant, ajoutons arbitrairement 3 neurones

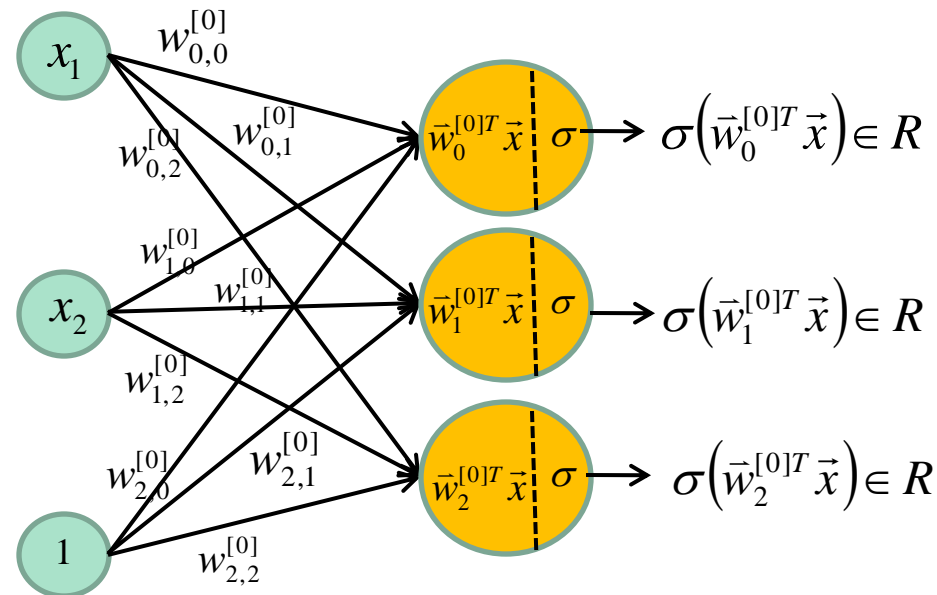


**Couche d'entrée
(3 « neurones »)**

**1^{ère} couche
(3 neurones)**

2D, 2Classes, Réseau à 1 couche cachée

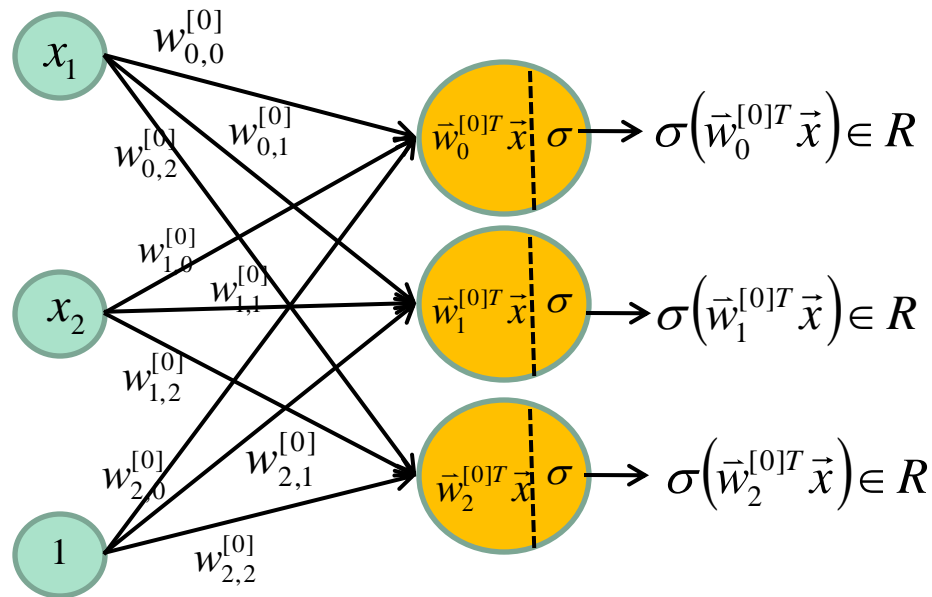
Puisque les poids sont **entre la couche d'entrée** et la **première couche** on va les identifier à l'aide de **l'indice [0]**



Couche d'entrée
(3 « neurones »)

1^{ère} couche
(3 neurones)

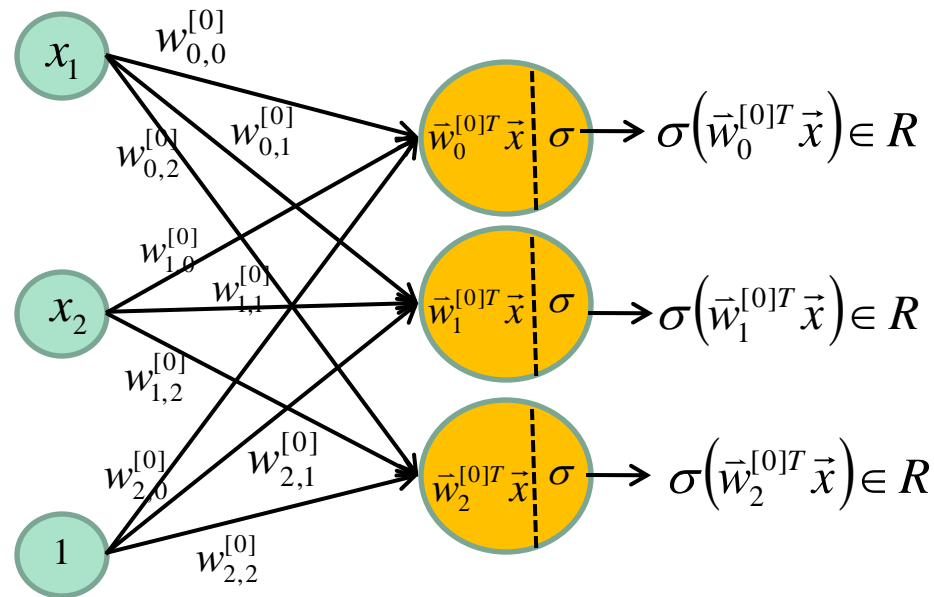
2D, 2Classes, Réseau à 1 couche cachée



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

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

2D, 2Classes, Réseau à 1 couche cachée

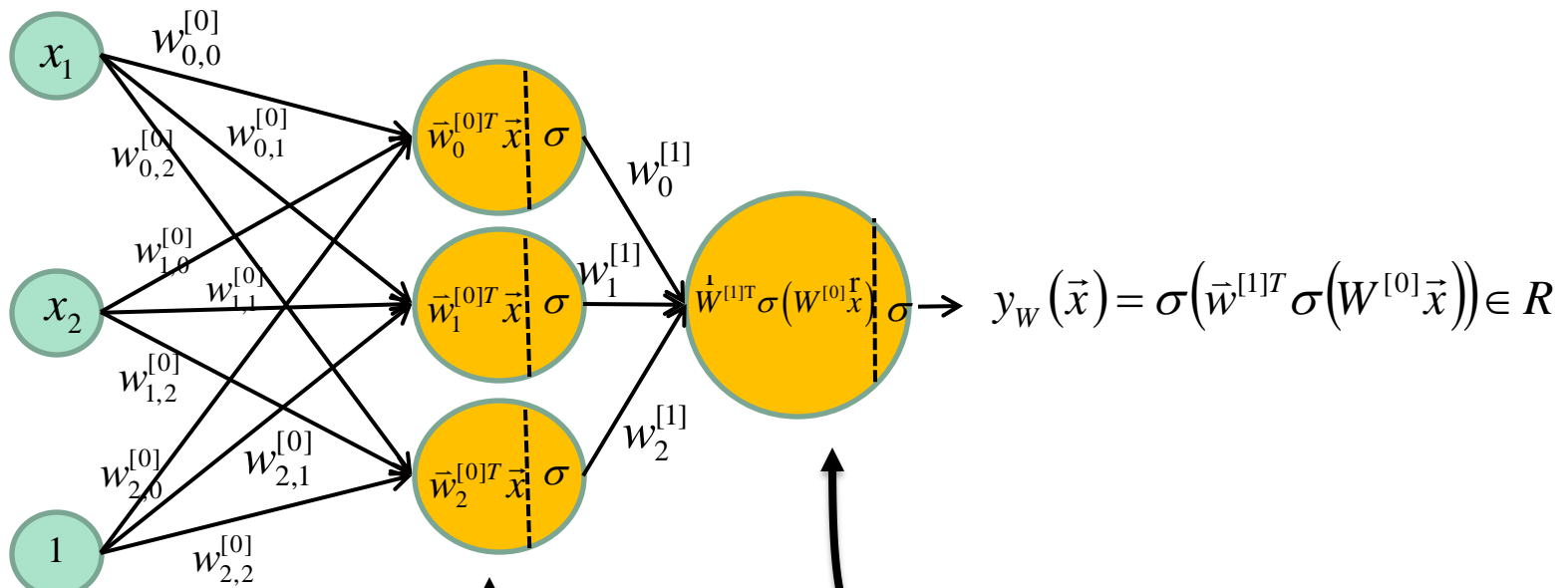


NOTE: représentation plus simple de la sortie de la 1^{ère} couche (3 réels)

$$\sigma\left(W^{[0]}\mathbf{r}_x\right)$$

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**.

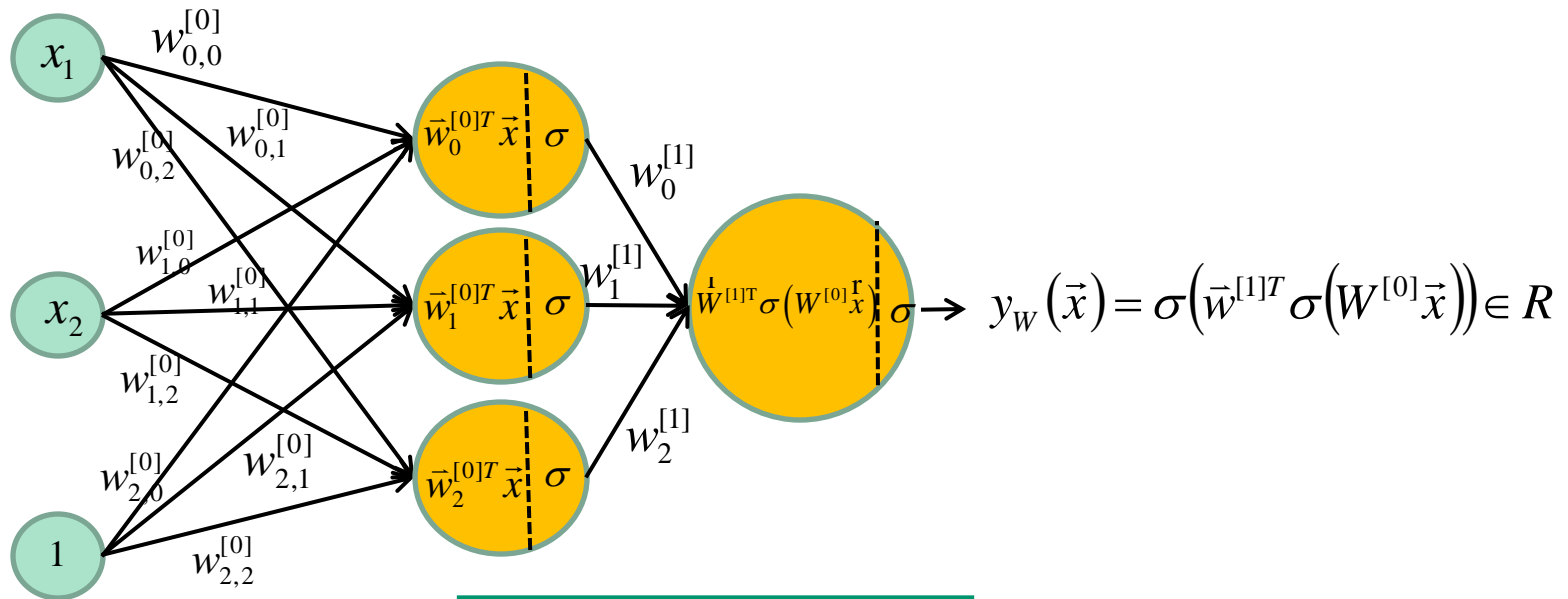


Couche d'entrée
(3 « neurones »)

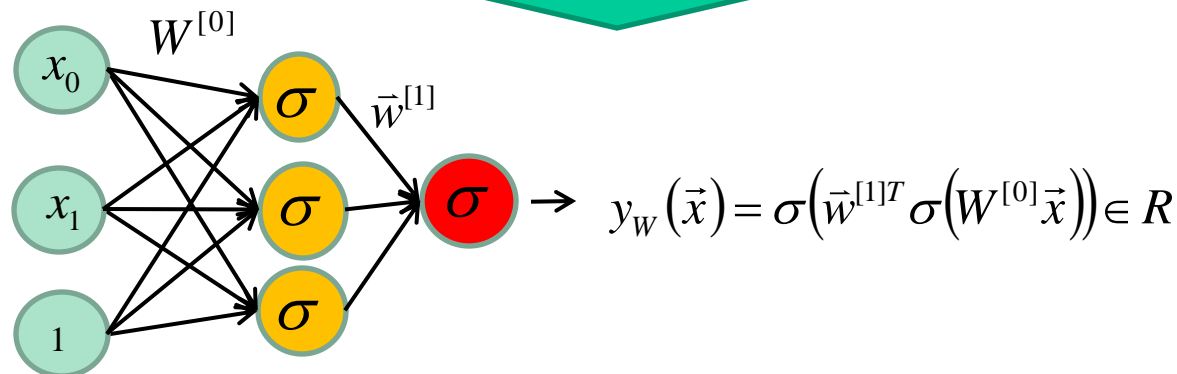
1^{ère} couche cachée
(3 neurones)

Couche de sortie
(1 neurone)

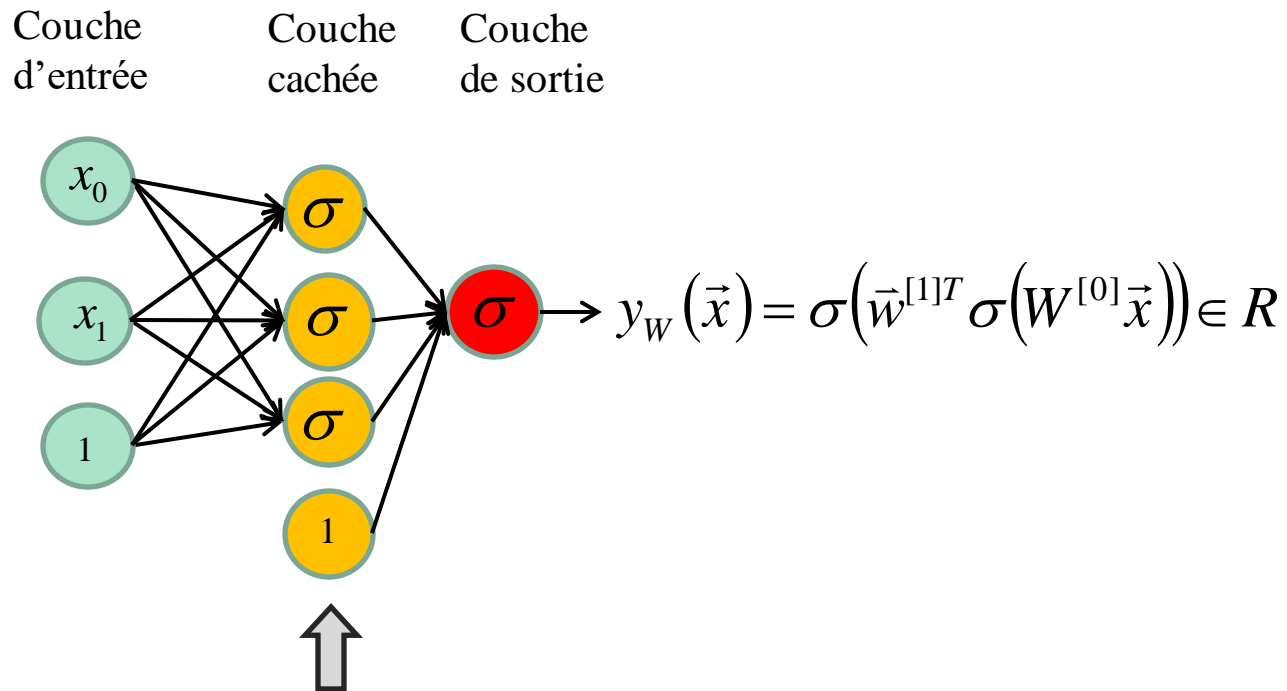
2D, 2Classes, Réseau à 1 couche cachée



Allègement
visuel

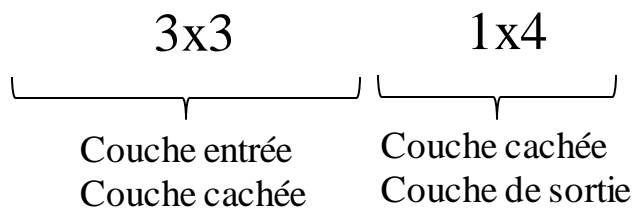


2D, 2Classes, Réseau à 1 couche cachée

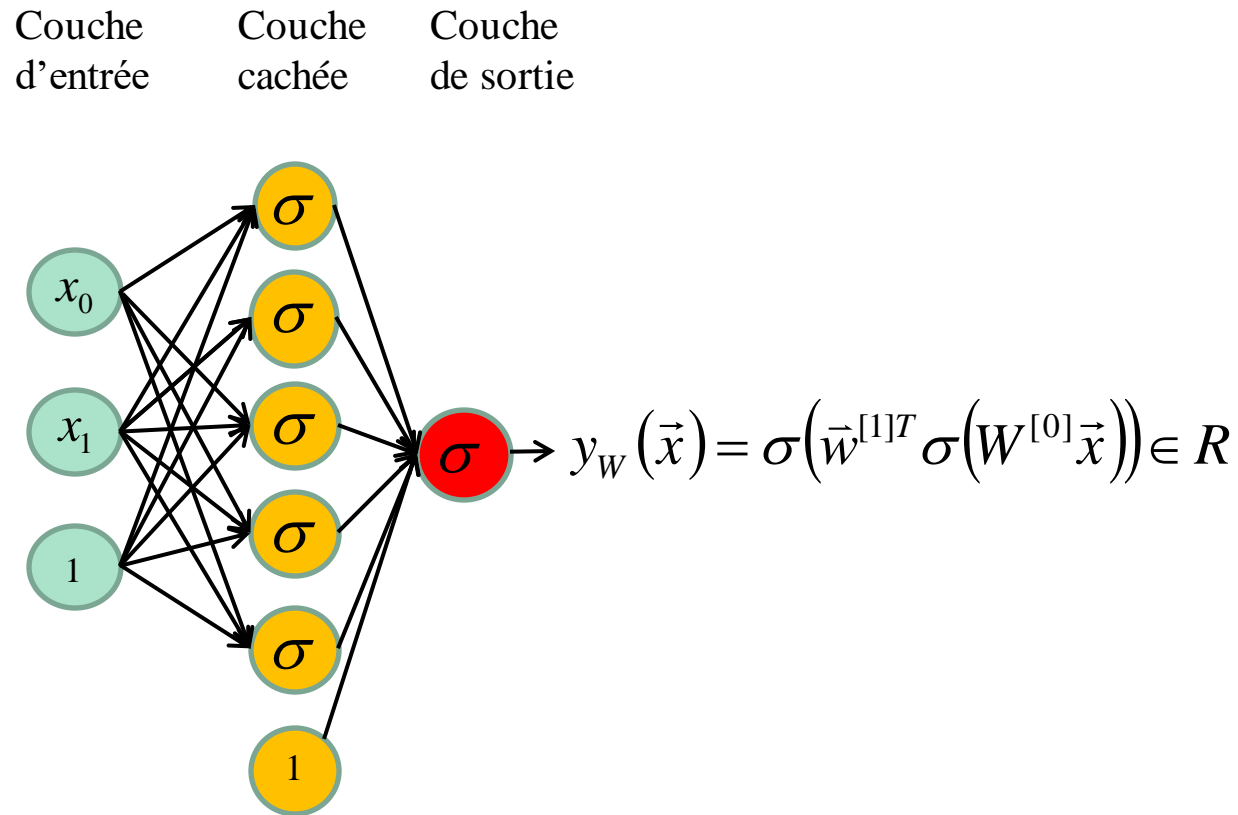


Très souvent, on ajoute un neurone de biais à la couche cachée.

Ce réseau possède au total **13 paramètres**



2D, 2Classes, Réseau à 1 couche cachée



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

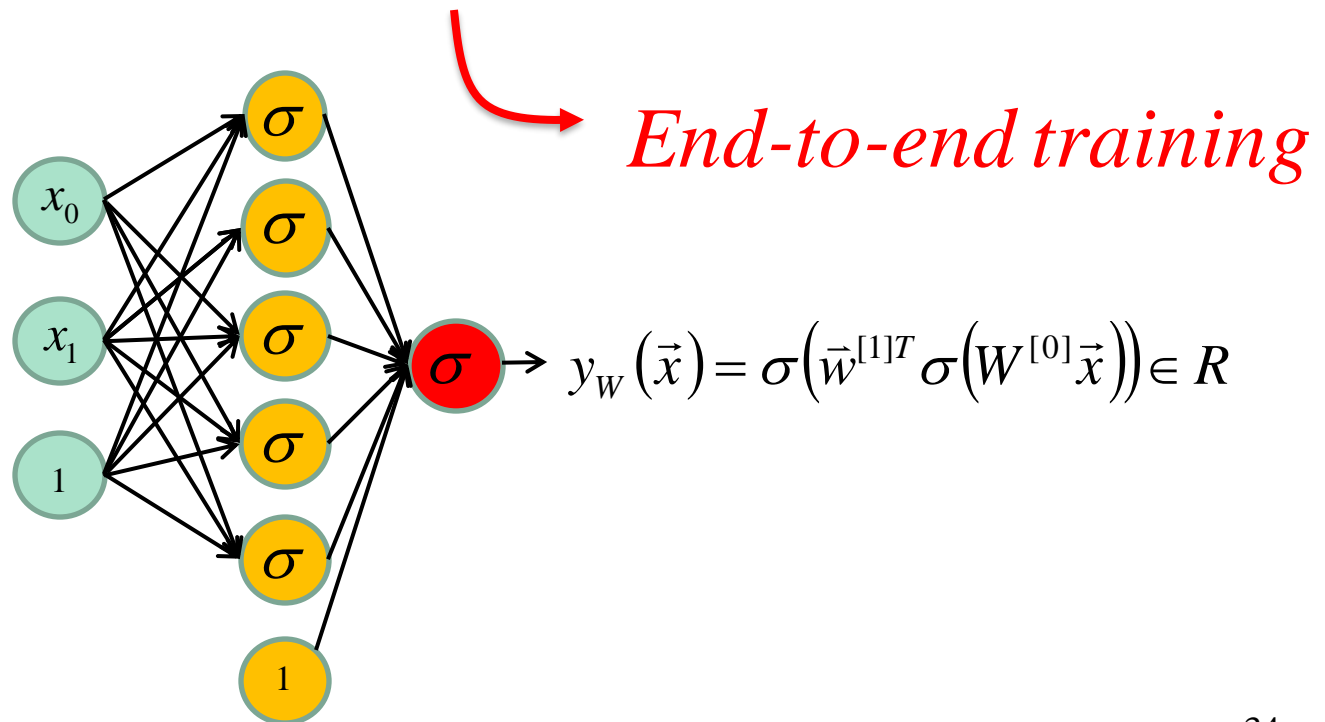
Ce réseau a $5 \times 3 + 1 \times 6 = \mathbf{21}$ **paramètres**

NOTE Importante

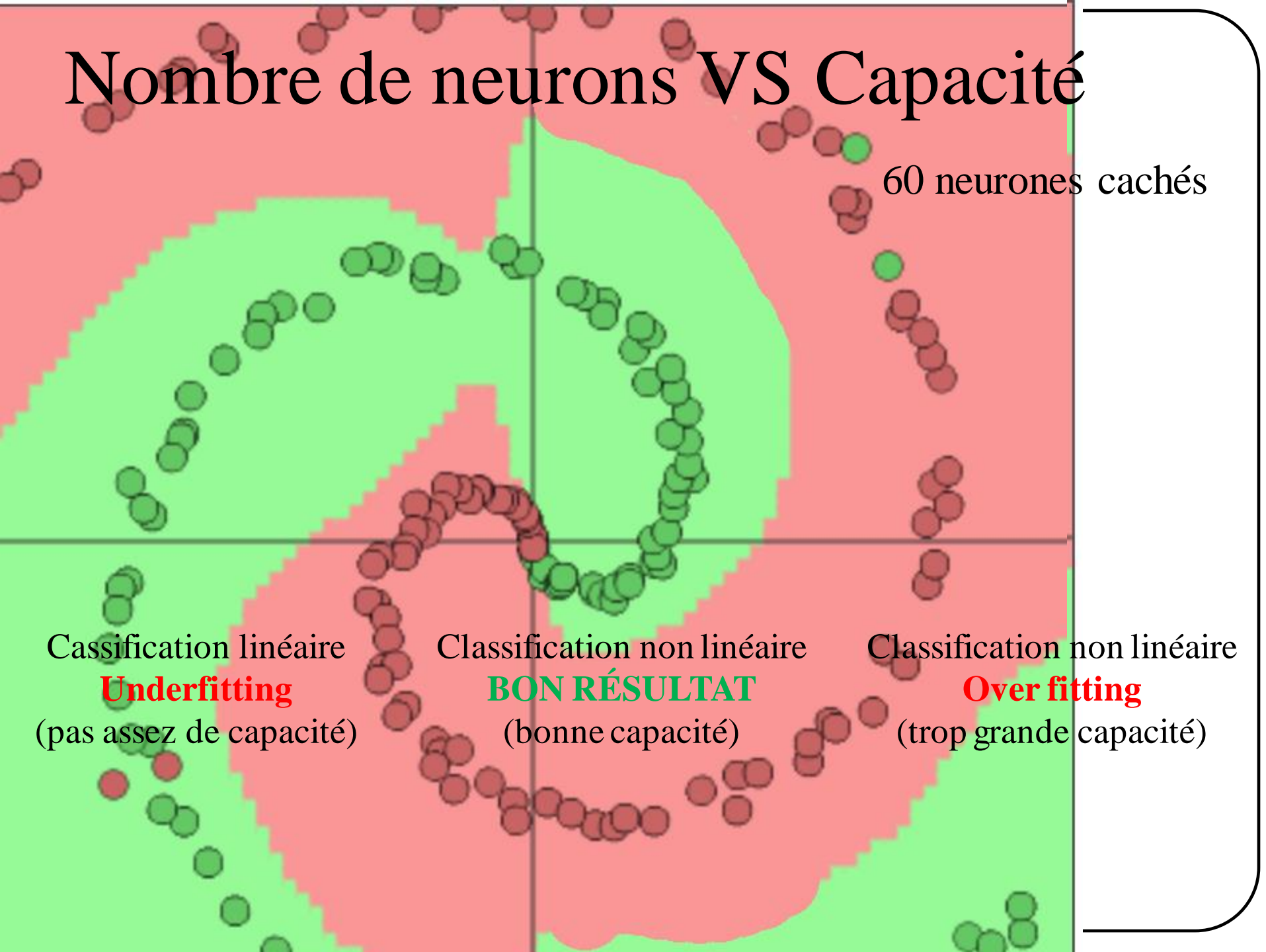
Le but de la première couche est de **projeter les données d'entrée** (ici $\vec{x} \in R^2$) vers un espace dimensionnel plus grand (ici $\sigma(W^{[0]}\vec{x}) \in R^5$) là où les **classes sont linéairement séparables**.

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

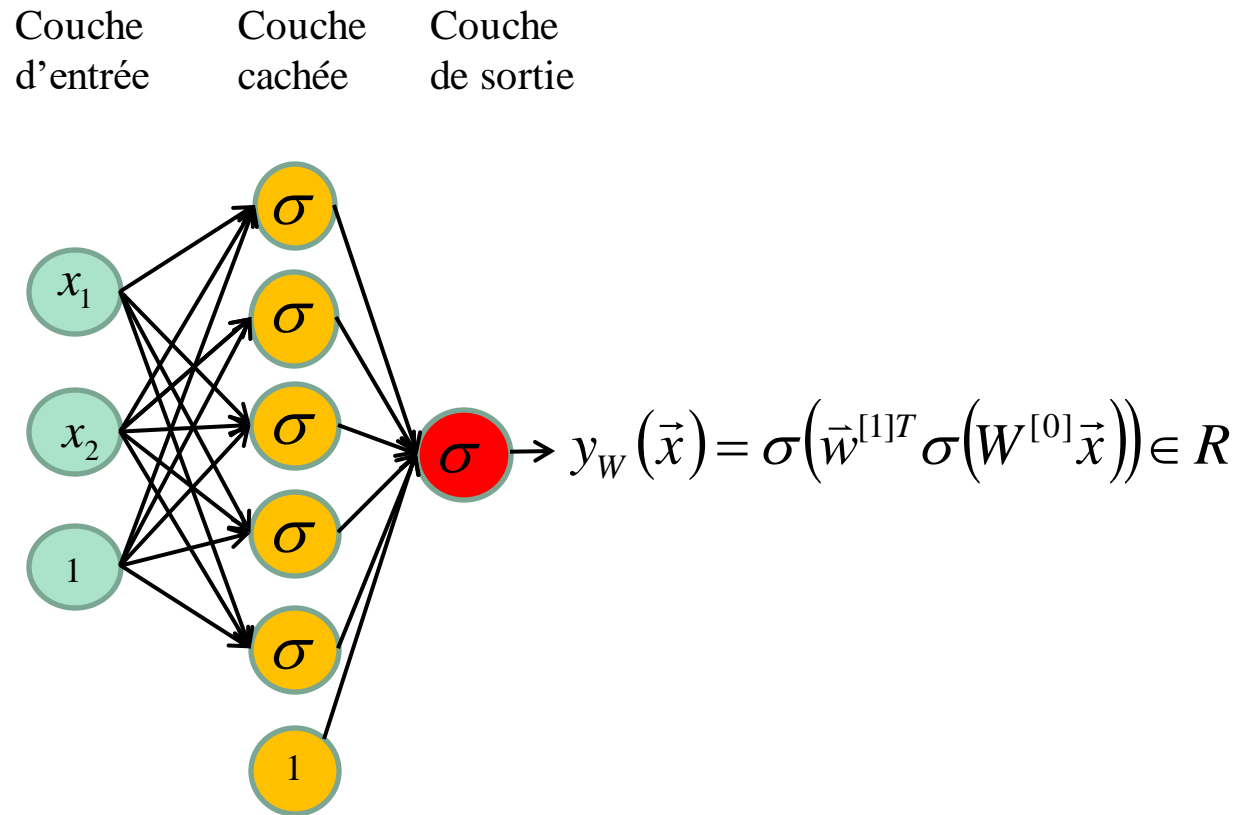
Par conséquent, au lieu de fixer nous même la fonction de base, on laisse le **réseau l'apprendre**.



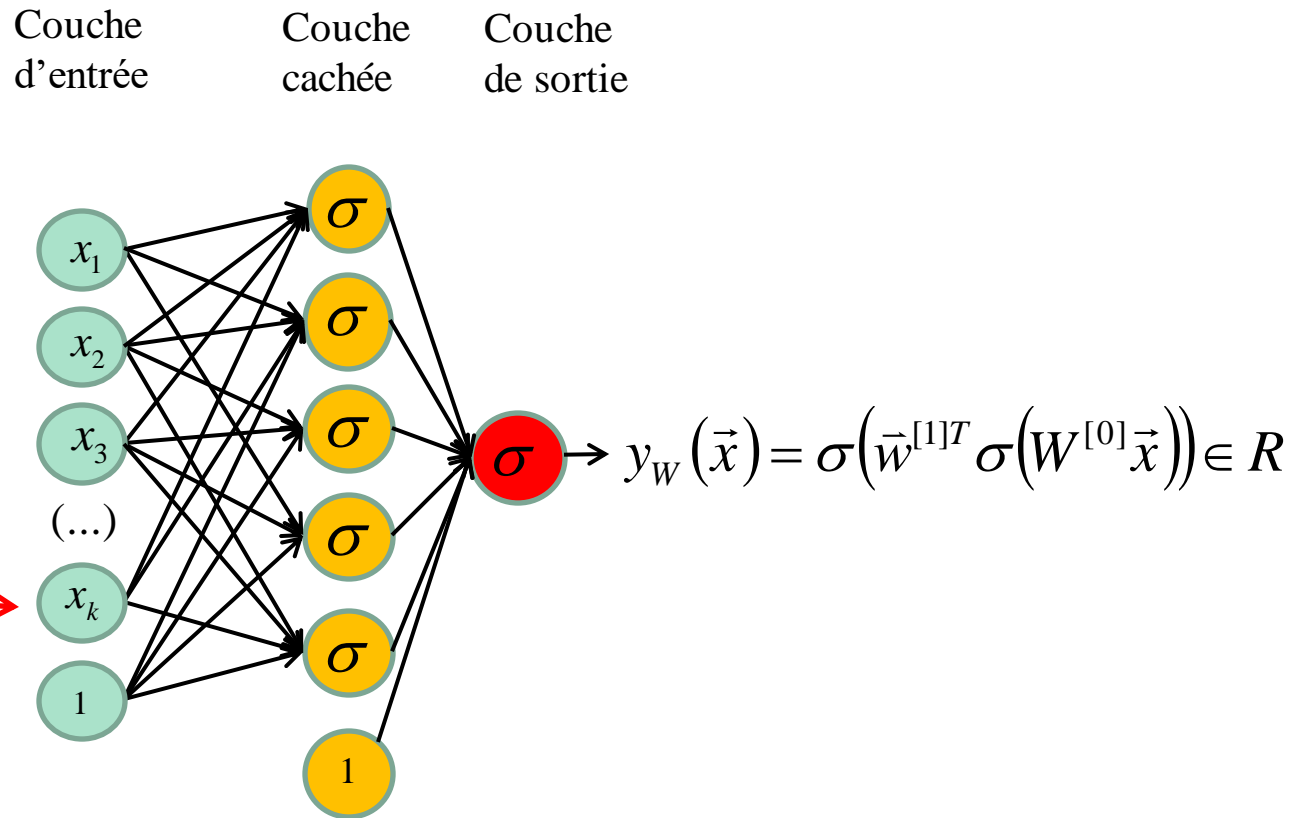
Nombre de neurones VS Capacité



2D, 2Classes, Réseau à 1 couche cachée



kD, 2Classes, Réseau à 1 couche cachée

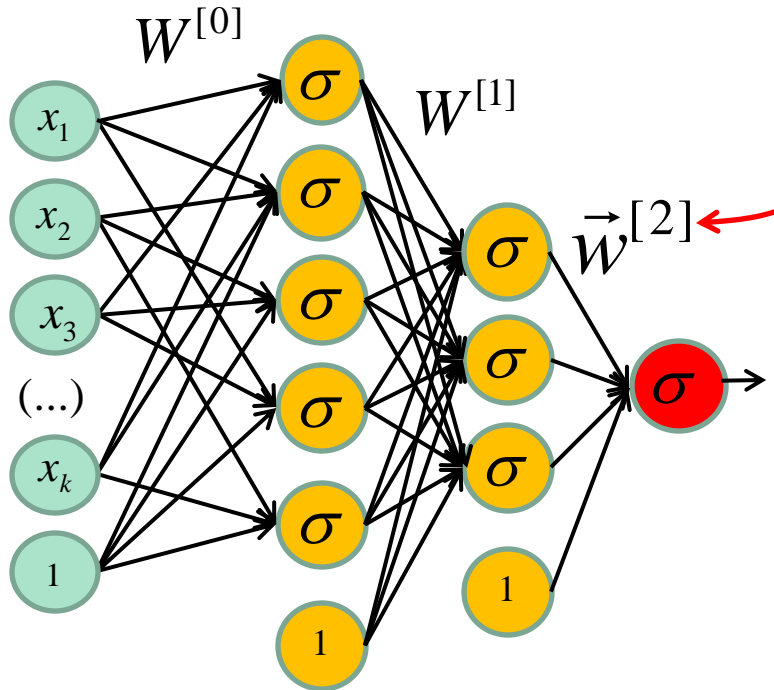


Au peut facilement augmenter la dimensionnalité des données d'entrée. Cela n'a pour effet que **d'augmenter le nombre de colonnes dans $W^{[0]}$**

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

kD, 2Classes, Réseau à 2 couches cachées

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



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

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

$$\vec{w}^{[2]} \in R^4$$

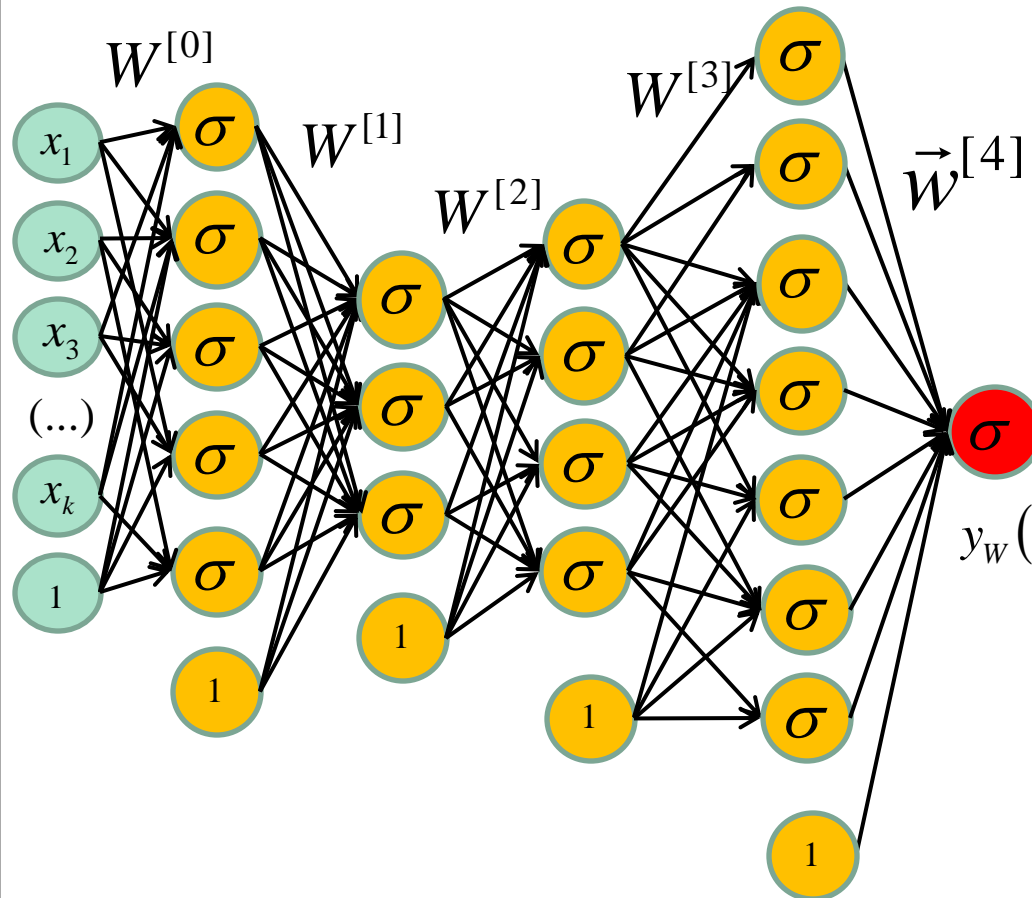
$$y_w(\vec{x}) = \sigma(\vec{w}^{[2]T} \sigma(W^{[1]} \sigma(W^{[0]} \vec{x})))$$

En ajoutant une couche cachée, on ajoute une multiplication matricielle

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

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



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

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

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

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

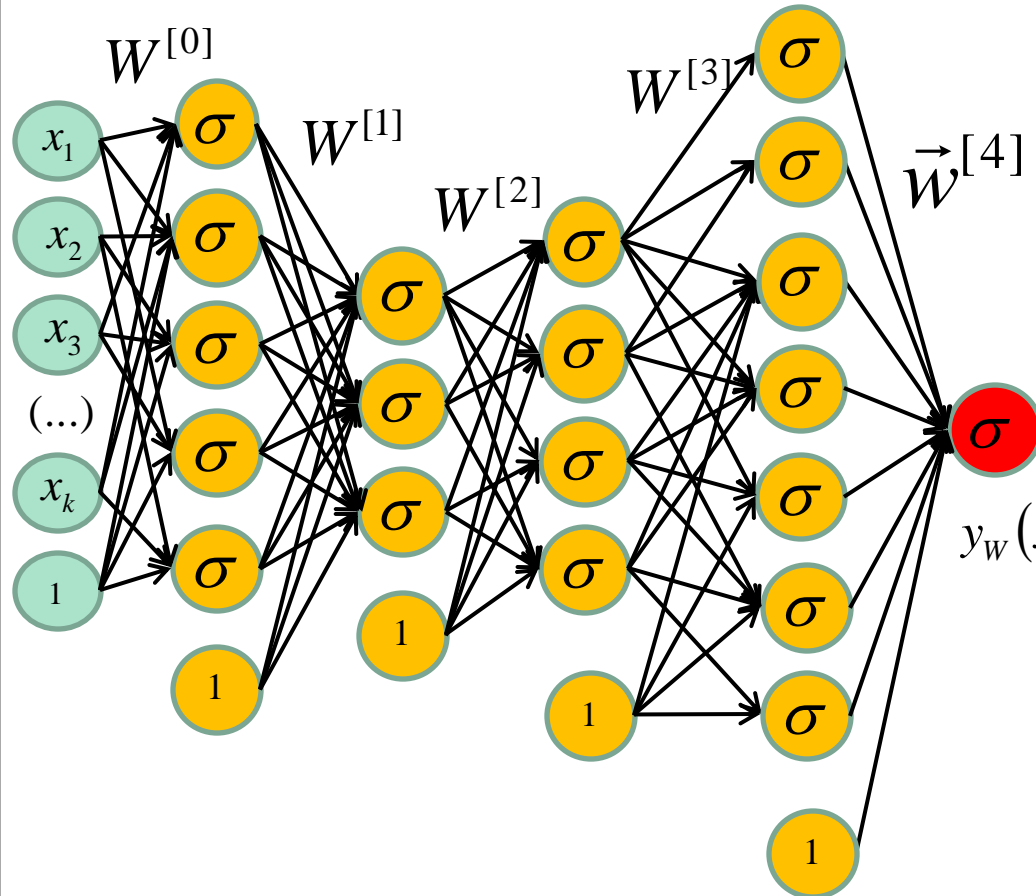
$$\vec{w}^{[4]} \in R^8$$

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

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

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



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

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

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

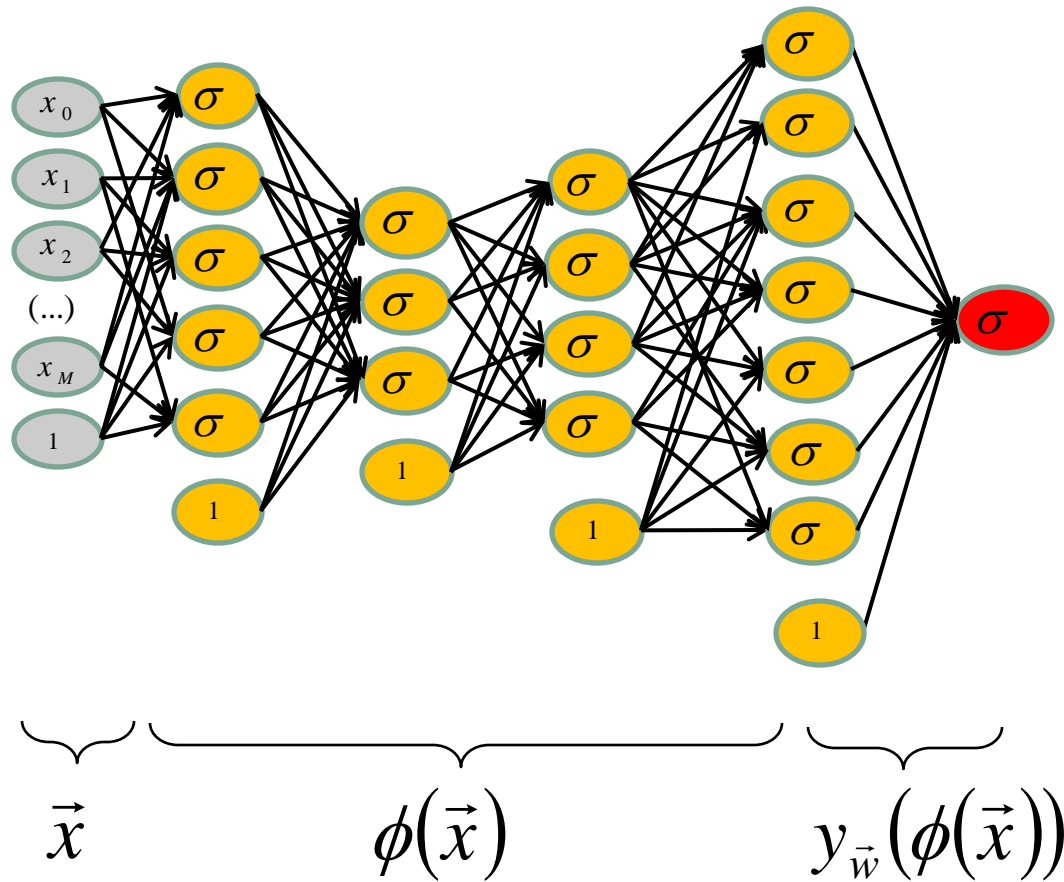
$$W^{[3]} \in R^{5 \times 7}$$

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

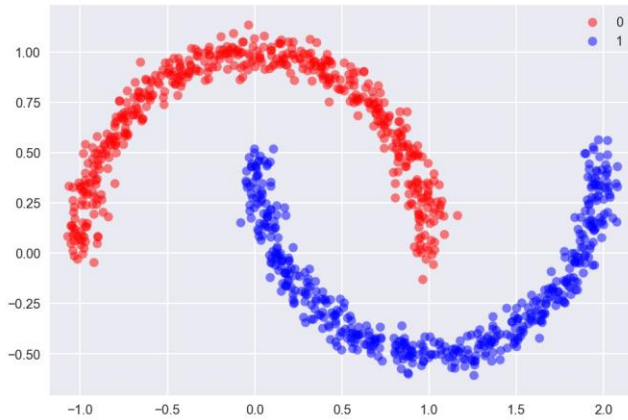
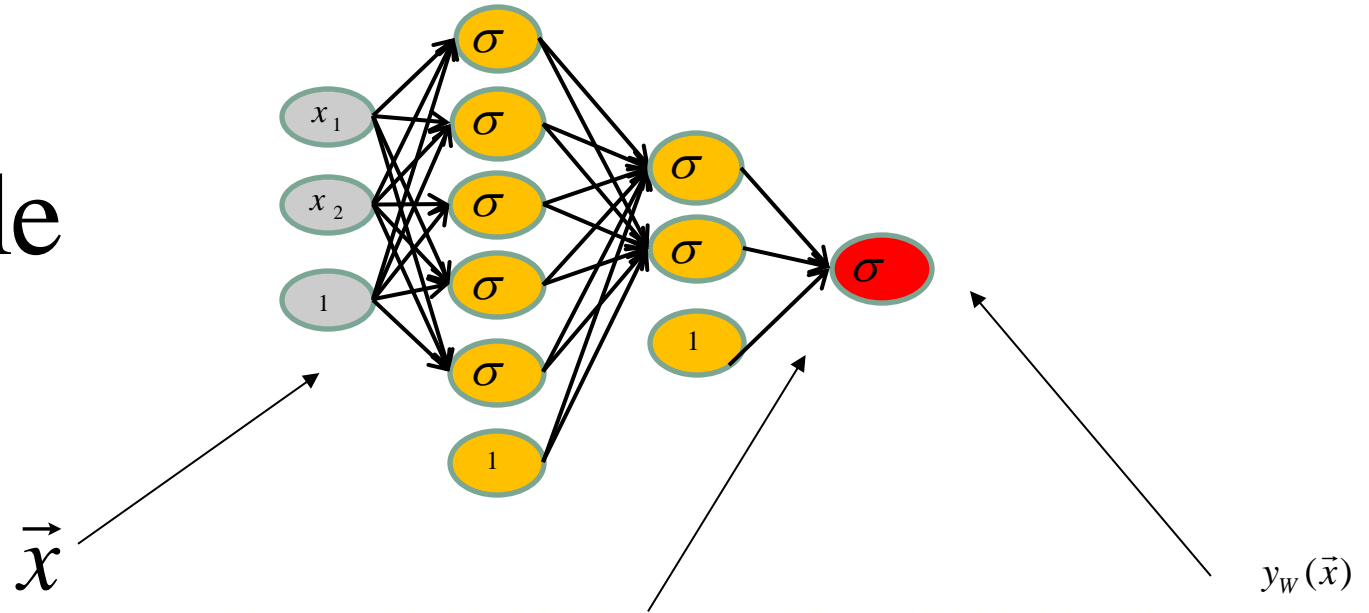
$$y_w(\vec{x}) = \sigma(\vec{w}^{[4]T} \sigma(W^{[3]} \sigma(W^{[2]} \sigma(W^{[1]} \sigma(W^{[0]} \vec{x}))))))$$

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

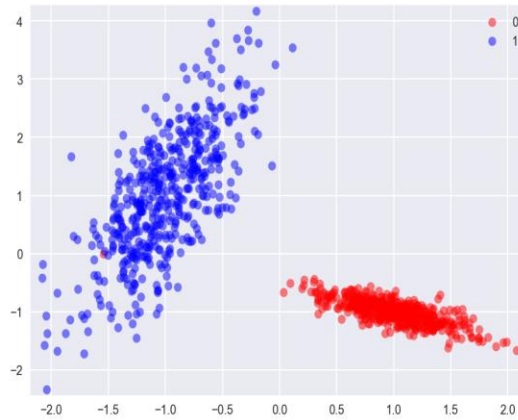
Réseau de neurones multicouches = apprendre une fonction de base



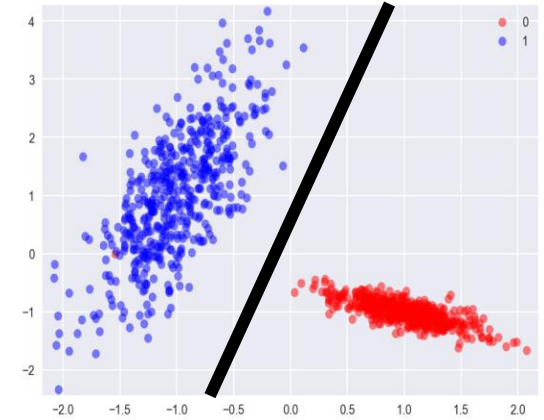
Example



Données en entrée



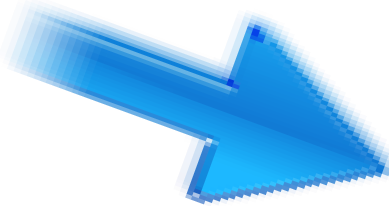
Sortie de la dernière couche



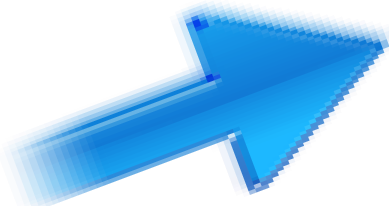
Sortie du réseau



Augmenter le nombre
de neurones par couche



Augmenter le nombre
de couches



**Augmenter la
capacité du réseau**



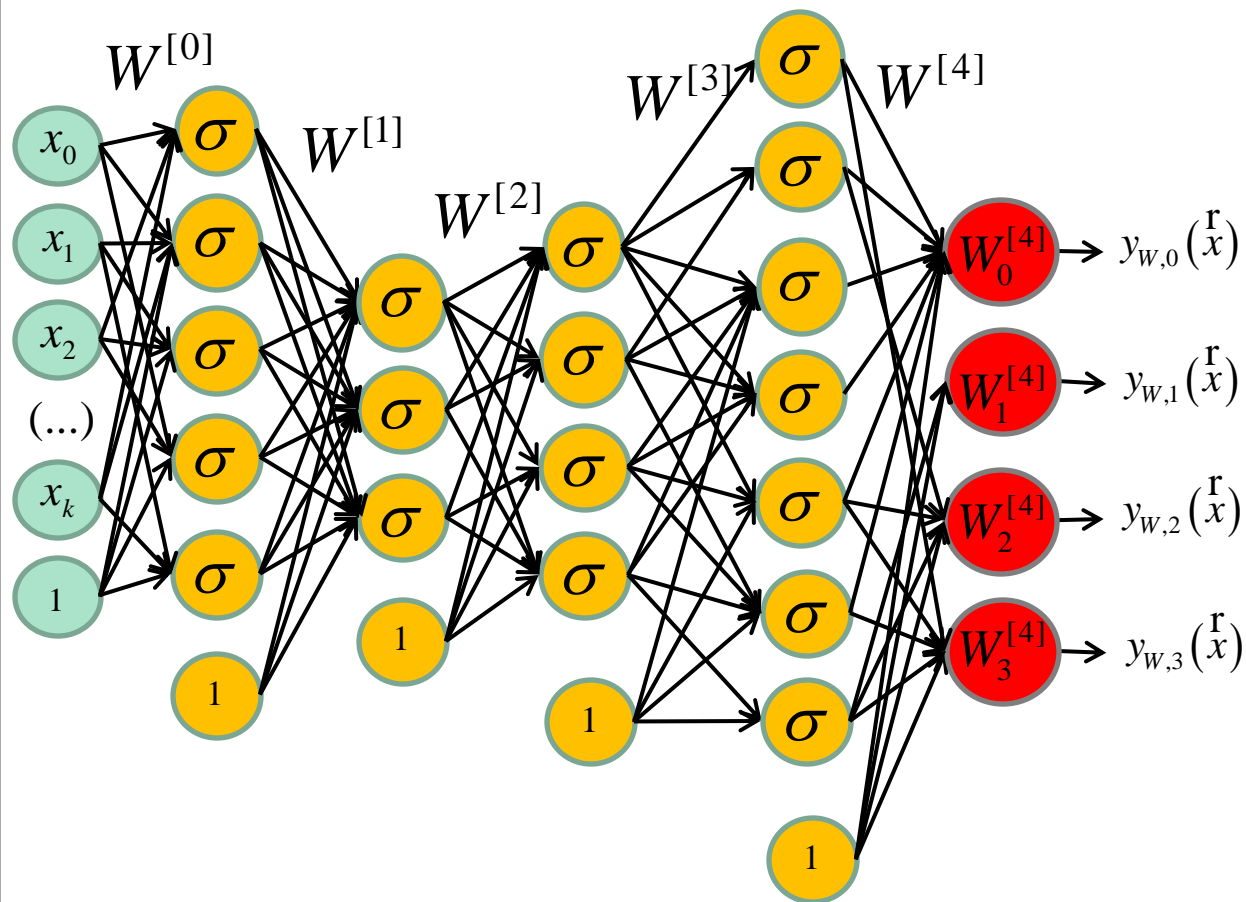
Augmenter la capacité d'un réseau
peut entraîner du **sur-apprentissage**



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

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



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

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

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

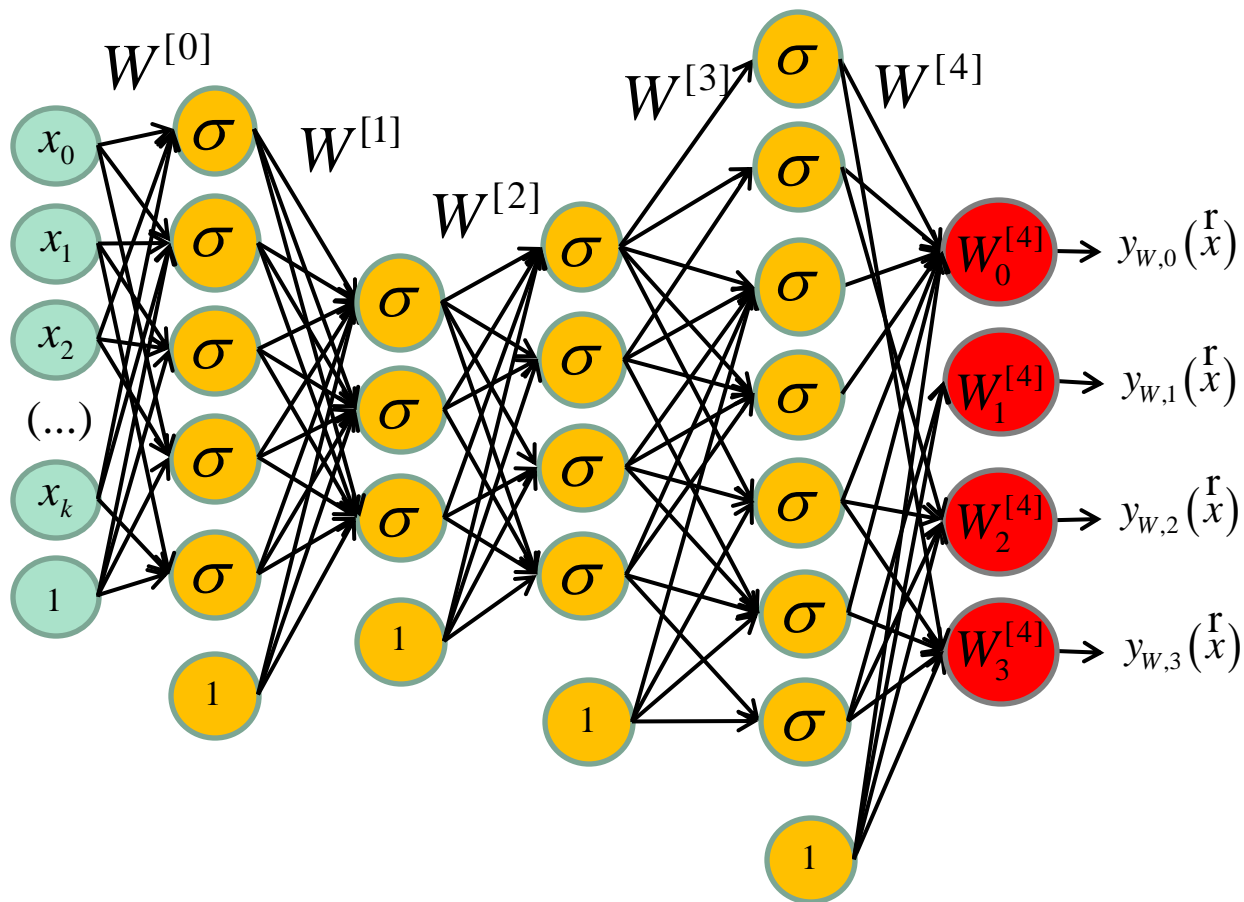
$$W^{[3]} \in R^{5 \times 7}$$

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

$$y_w \left(\begin{smallmatrix} \mathbf{r} \\ \mathbf{x} \end{smallmatrix} \right) = W^{[4]} \sigma \left(W^{[3]} \sigma \left(W^{[2]} \sigma \left(W^{[1]} \sigma \left(W^{[0]} \begin{smallmatrix} \mathbf{r} \\ \mathbf{x} \end{smallmatrix} \right) \right) \right) \right)$$

kD, 4 Classes, Réseau à 4 couches cachées

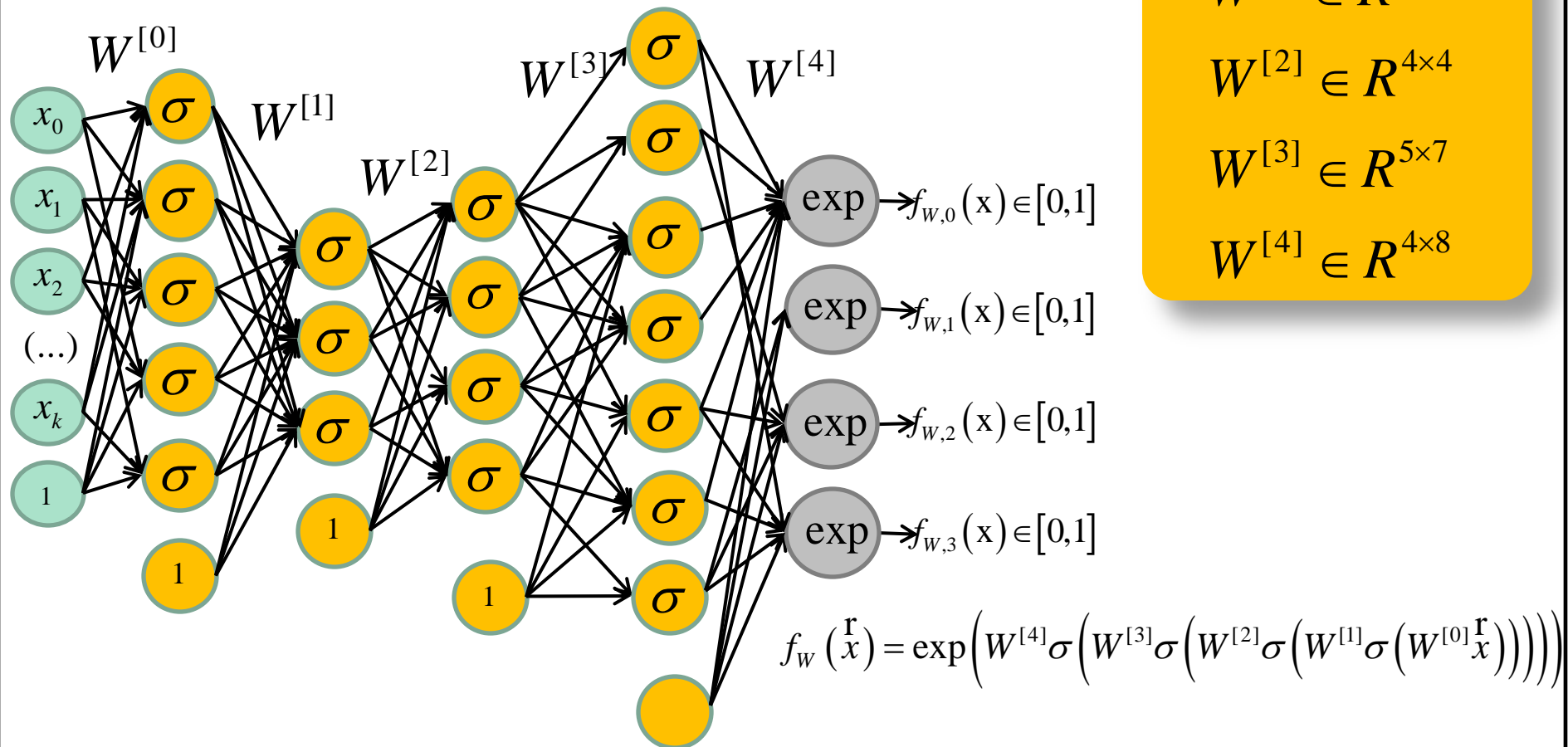
Couche d'entrée Couche cachée 1 Couche cachée 2 Couche cachée 3 Couche cachée 4 Couche de sortie



$$y_w \left(\begin{smallmatrix} \mathbf{r} \\ \mathbf{x} \end{smallmatrix} \right) = W^{[4]} \sigma \left(W^{[3]} \sigma \left(W^{[2]} \sigma \left(W^{[1]} \sigma \left(W^{[0]} \begin{smallmatrix} \mathbf{r} \\ \mathbf{x} \end{smallmatrix} \right) \right) \right) \right)$$

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



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

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

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

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

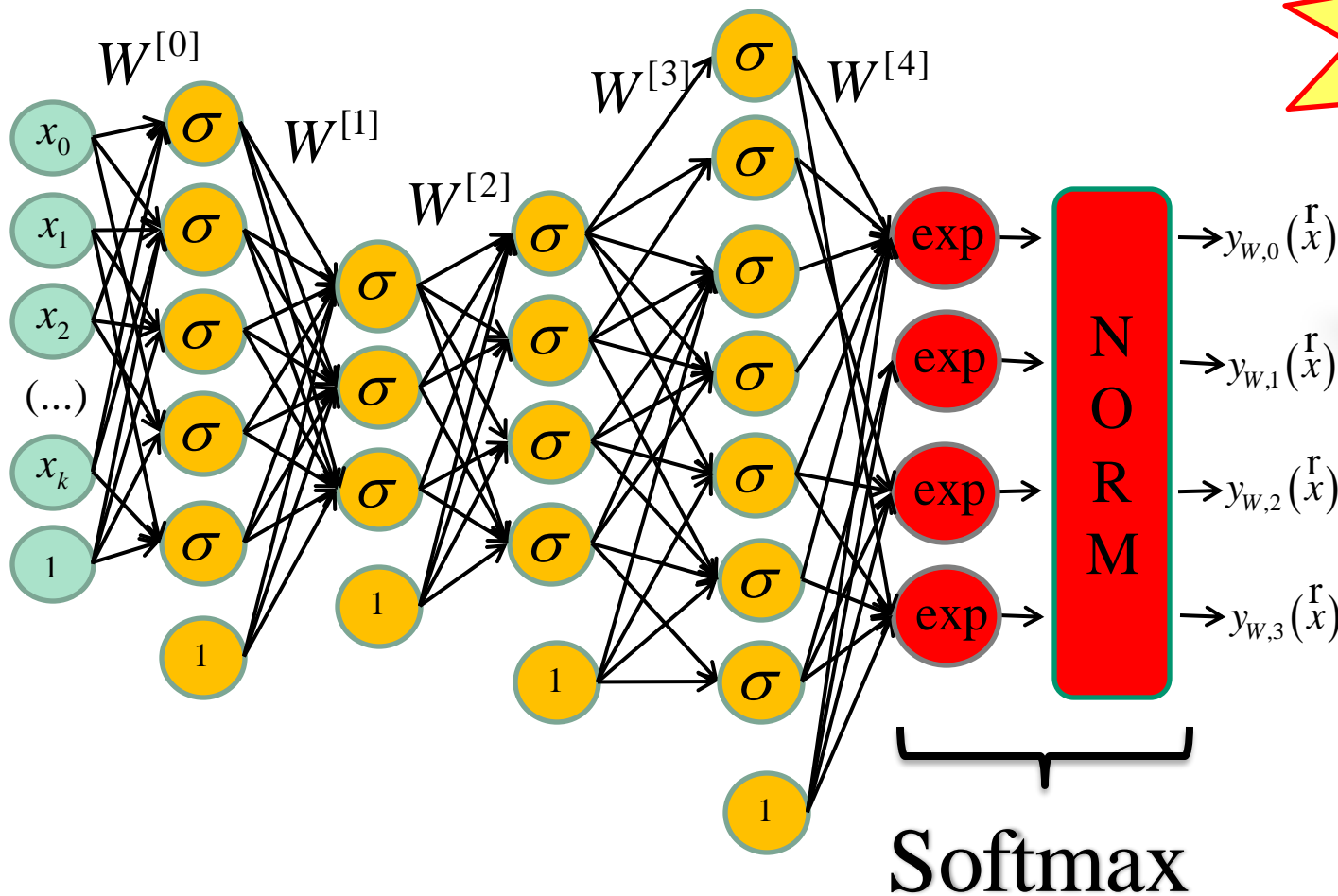
$$W^{[4]} \in R^{4 \times 8}$$

$$f_w \left(\begin{matrix} \mathbf{r} \\ \mathbf{x} \end{matrix} \right) = \exp \left(W^{[4]} \sigma \left(W^{[3]} \sigma \left(W^{[2]} \sigma \left(W^{[1]} \sigma \left(W^{[0]} \begin{matrix} \mathbf{r} \\ \mathbf{x} \end{matrix} \right) \right) \right) \right) \right)$$

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

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

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

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

Simulation

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

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



Comment optimiser les paramètres?

0- Partant de

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

Trouver une fonction de régularisation. En général

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

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

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.

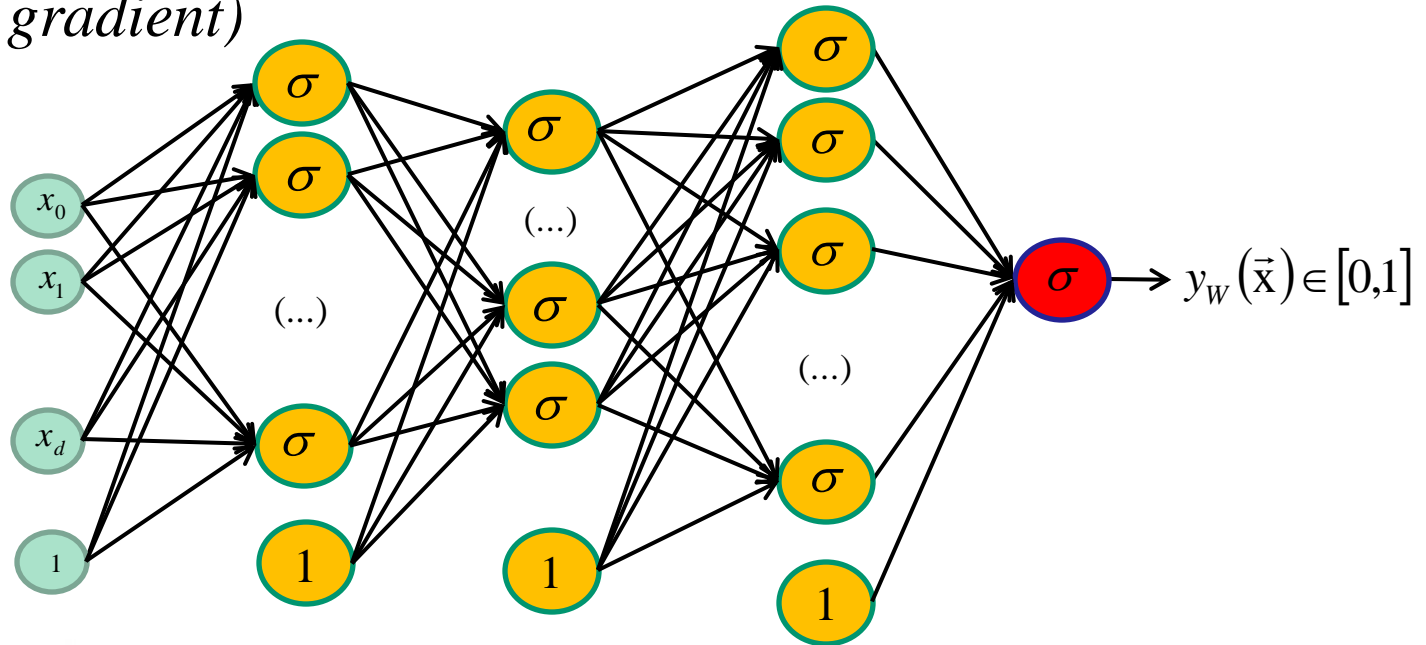
$$w_{a,b}^{[c]} = w_{a,b}^{[c]} - \eta \frac{\partial (E_D(W) + \lambda R(W))}{\partial w_{a,b}^{[c]}}$$

Comment optimiser les paramètres?

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

Disparition du gradient

(*vanishing gradient*)

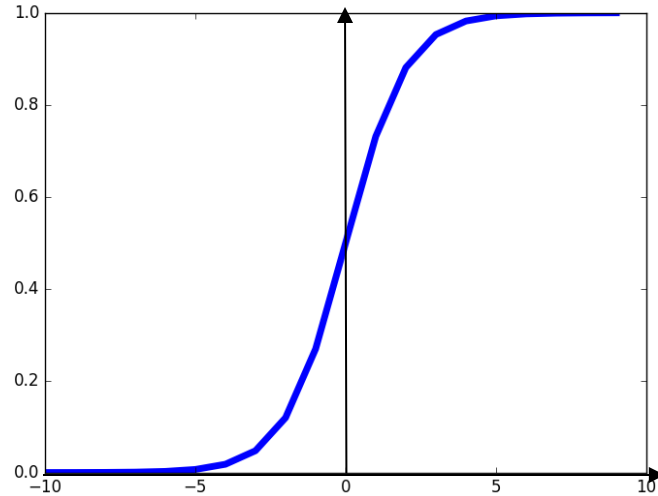


Malheureusement, l'entraînement d'un **réseau profond** avec **rétro-propagation** et des fonctions d'activations **sigmoïdales** entraîne des problèmes de

disparition du gradient

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

Fonction d'activation



Sigmoïde

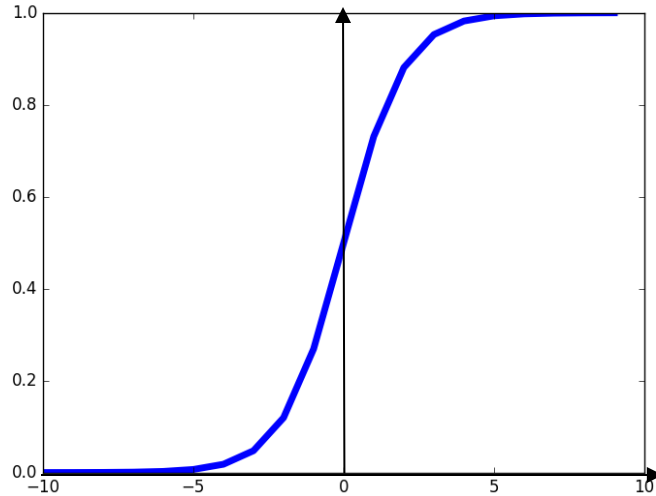
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Ramène les valeurs entre 0 et 1
- Historiquement populaire

3 Problèmes :

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

Fonction d'activation



Sigmoïde

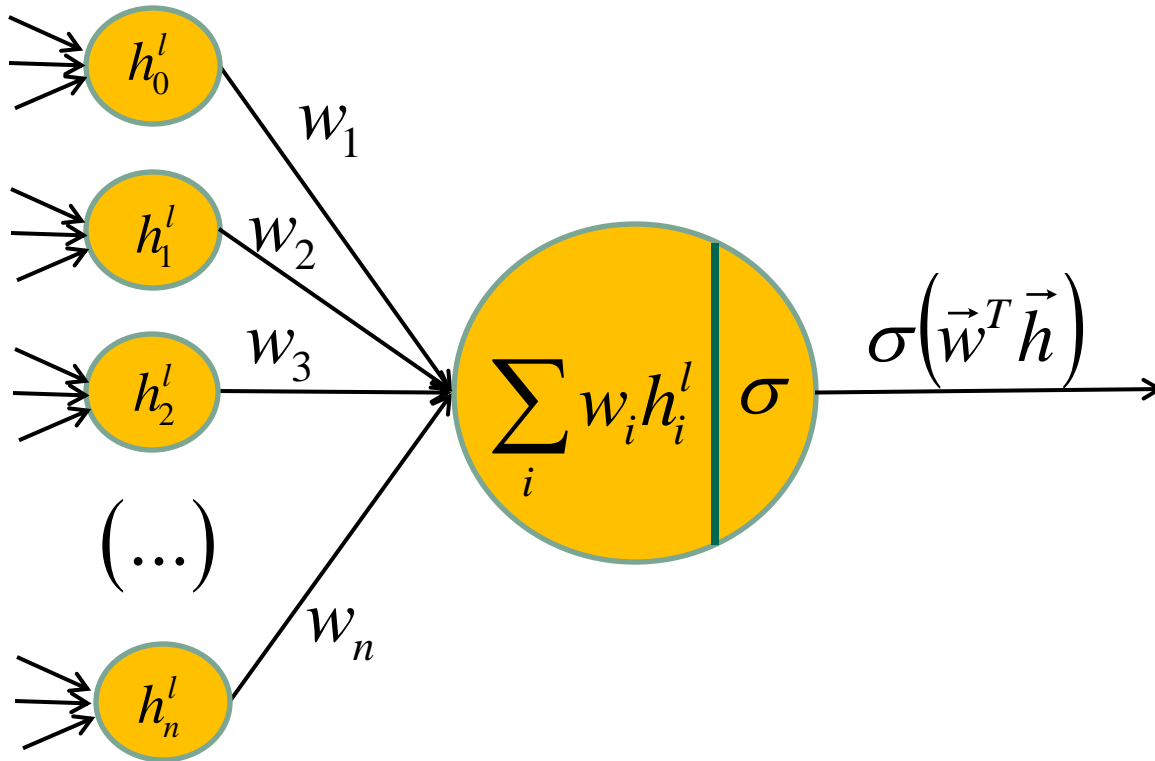
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Ramène les valeurs entre 0 et 1
- Historiquement populaire

3 Problèmes :

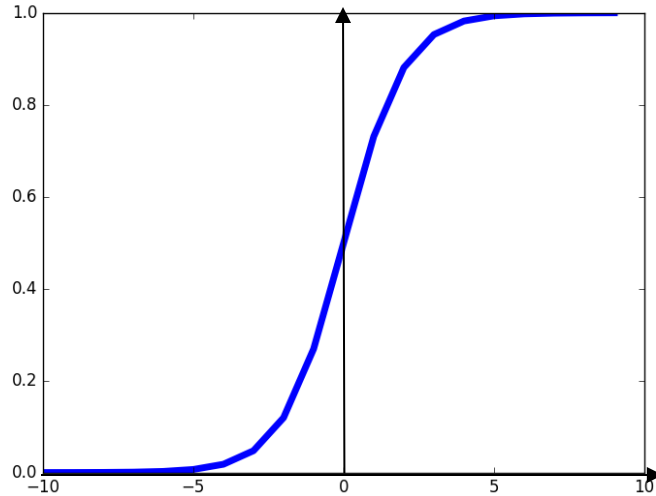
- Un neurone saturé a pour effet de « **tuer** » les **gradients**
- Sortie d'une sigmoïde n'est **pas centrée à zéro**.

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?**

Fonction d'activation



Sigmoïde

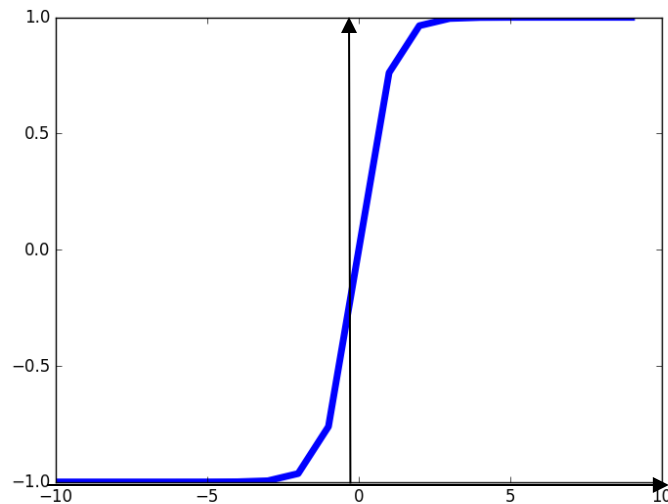
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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.

Fonction d'activation

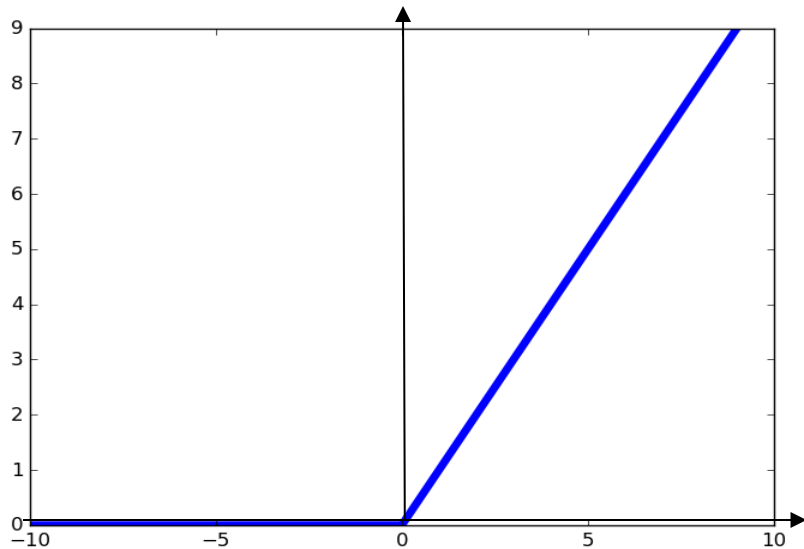


Tanh(x)

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

Fonction d'activation

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

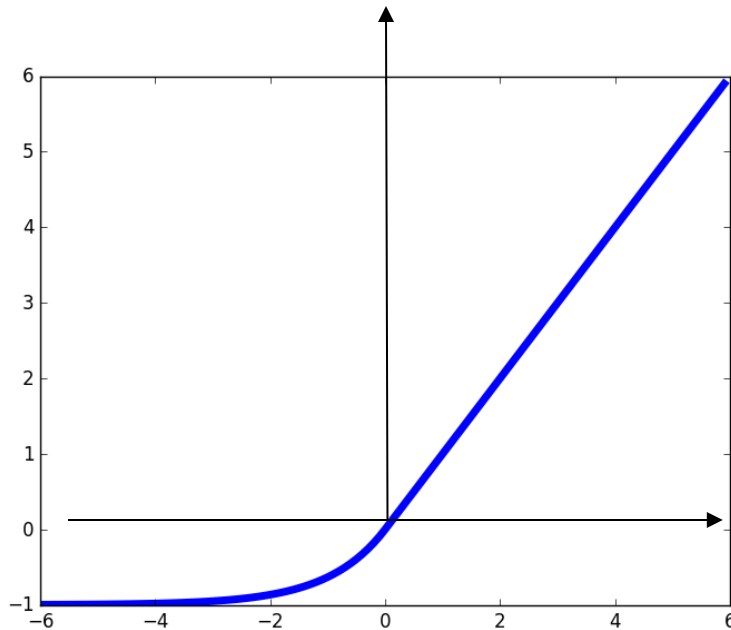


ReLU(x)
(Rectified Linear Unit)

- 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 $x < 0$? 😞

Fonction d'activation

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



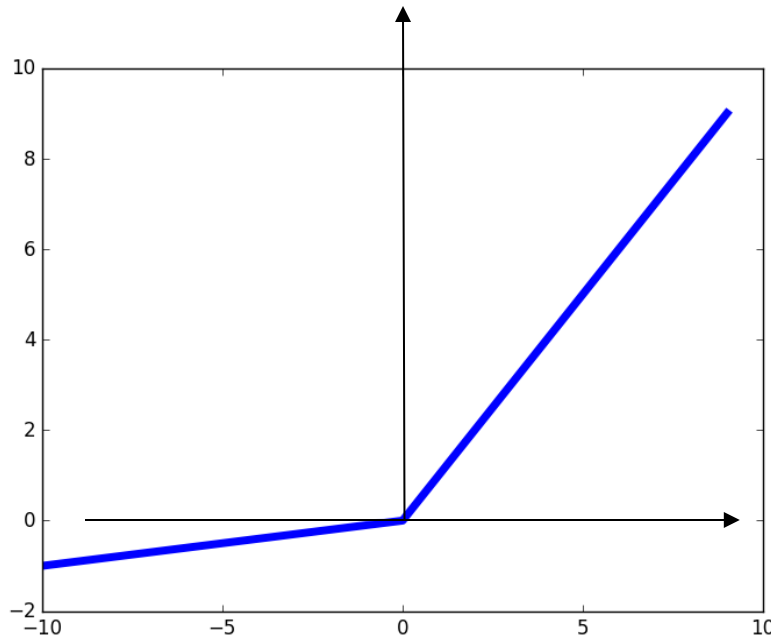
ELU(x)

(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** 😞

Fonction d'activation

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



Leaky ReLU(x)

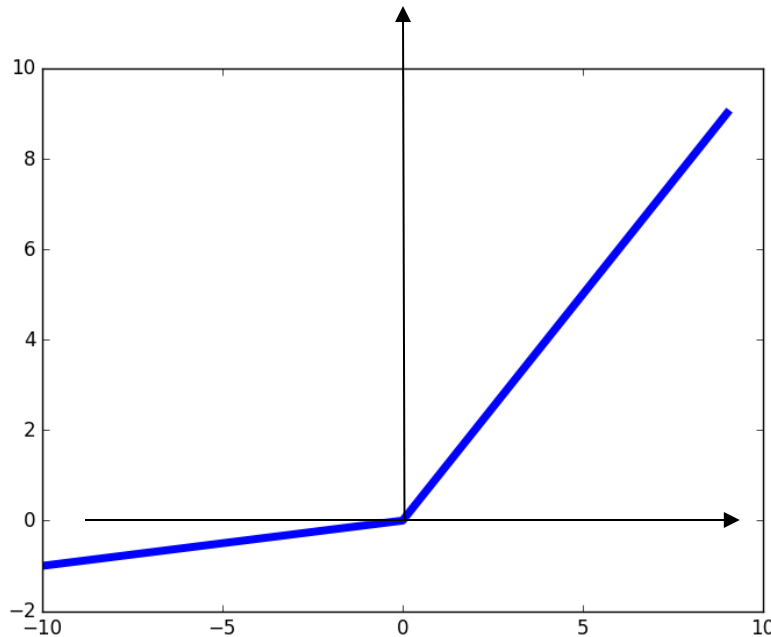
- Aucune **saturation** 😊
- Super **rapide** 😊
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) 😊
- **Gradients ne meurent pas** 😊
- 0.01 est un **hyperparamètre** 😊

[Mass et al., 2013]

[He et al., 2015]

Fonction d'activation

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



Parametric ReLU(x)

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

[Mass et al., 2013]

[He et al., 2015]

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.

Les bonnes pratiques

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 \mathbf{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 $k == \text{MAX_ITER}$

Optimisation par *Batch*

Initialiser \mathbf{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 $k == \text{MAX_ITER}$

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

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*”.

TP1
TP2
TP3

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**/

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

JUSQU'À ce que toutes les données sont bien classées ou

$k==\text{MAX_ITER}$

} **Itération**

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 $k=k+1$

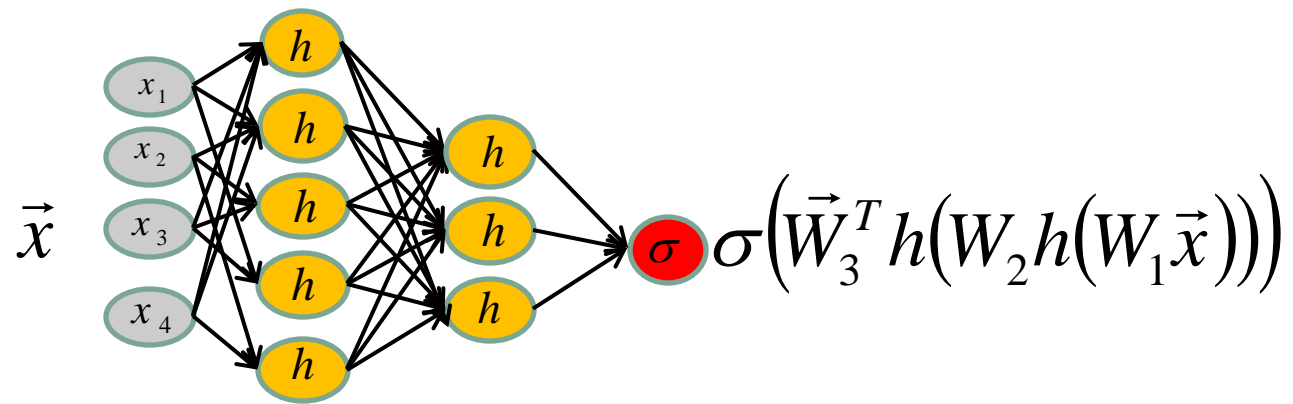
FAIRE $n=0$ à N par sauts de *MBS* /**Mini-batch size**/

$$\mathbf{w} = \mathbf{w} - \eta^{[k]} \sum_{i=n}^{n+MBS} \nabla E(\mathbf{r}_{x_i})$$

JUSQU'À ce que toutes les données sont bien classées ou

$k==\text{MAX_ITER}$

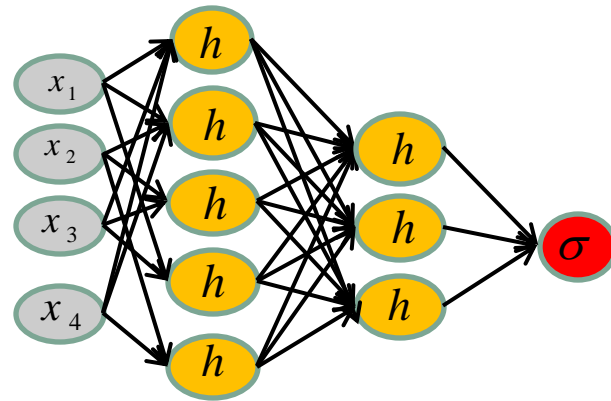
} **Epoch**



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

\vec{x}	$\in IR^4$
$W_1 \vec{x}$	$\in IR^5$
$h(W_1 \vec{x})$	$\in IR^5$
$W_2 h(W_1 \vec{x})$	$\in IR^3$
$h(W_2 h(W_1 \vec{x}))$	$\in IR^3$
$\vec{W}_3^T (h(W_2 h(W_1 \vec{x})))$	$\in IR$
$\sigma(\vec{W}_3^T (h(W_2 h(W_1 \vec{x}))))$	$\in IR$

$$X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3\}$$



$$Y = \begin{Bmatrix} \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_1))) \\ \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_2))) \\ \sigma(\vec{W}_3 h(W_2 h(W_1 \vec{x}_3))) \end{Bmatrix}$$

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

POUR i allant de 1 à 3

$$\vec{x}_i = X[i]$$

$$W_1 \vec{x}$$

$$h(W_1 \vec{x})$$

$$W_2 h(W_1 \vec{x})$$

$$h(W_2 h(W_1 \vec{x}))$$

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

$$Y[i] = \sigma(\vec{W}_3^T (h(W_2 h(W_1 \vec{x}))))$$

$$\in \mathbb{R}^4$$

$$\in \mathbb{R}^5$$

$$\in \mathbb{R}^5$$

$$\in \mathbb{R}^3$$

$$\in \mathbb{R}^3$$

$$\in \mathbb{R}$$

$$\in \mathbb{R}$$

Solution
naïve et peu
efficace

TP1

Solution

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

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

TROIS
produits
scalaires

Solution

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

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

UNE
multiplication
matricielle

Solution

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

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

The diagram illustrates the batching of three matrix-vector multiplications. On the left, three equations show the same 5x4 weight matrix W_1 being multiplied by different input vectors. The first equation uses input $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$ to produce output $\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix}$. The second equation uses input $\begin{bmatrix} d \\ e \\ f \\ g \end{bmatrix}$ to produce output $\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$. The third equation uses input $\begin{bmatrix} h \\ i \\ j \\ k \end{bmatrix}$ to produce output $\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix}$. On the right, these three output vectors are shown as columns of a single matrix: $\begin{bmatrix} u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \end{bmatrix}$. Arrows connect the output vectors from the equations to their corresponding columns in the batched matrix.

TROIS
multi
matrice-
vecteur

Solution

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

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

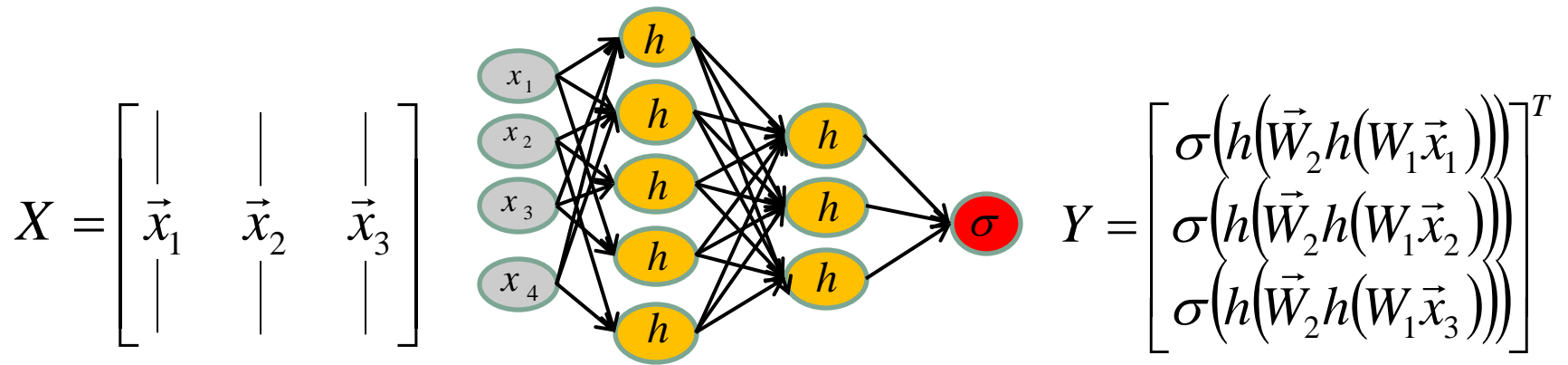
UNE
multiplication
matricielle

Vectorisation de la propagation avant

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

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

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



Vectoriser la rétropropagation

Vectoriser la rétropropagation

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

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

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}^T ?

Vectoriser la rétropropagation

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

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

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

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

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

Concentrons-nous sur w_1

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

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

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

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

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


Concentrons-nous sur w_1

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

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

$$w_1 \leftarrow w_1 - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

(Puisqu'on a une batch de 3 éléments, on a 3 prédictions et donc 3 gradients)



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

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

Donc en résumé ...

$$w_1 \leftarrow w_1 - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

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

Et pour tous les poids

$$w_1 \leftarrow w_1 - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

$$w_2 \leftarrow w_2 - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} b \\ e \\ h \end{bmatrix}$$

$$w_3 \leftarrow w_3 - \eta \begin{bmatrix} \frac{\partial E_1}{\partial Y} & \frac{\partial E_2}{\partial Y} & \frac{\partial E_3}{\partial Y} \end{bmatrix} \begin{bmatrix} c \\ f \\ i \end{bmatrix}$$

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

Et pour tous les poids

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

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

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

Matrice jacobienne

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

$$W \quad X \quad Y$$

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

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

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

Vectorisation de la rétro-propagation

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

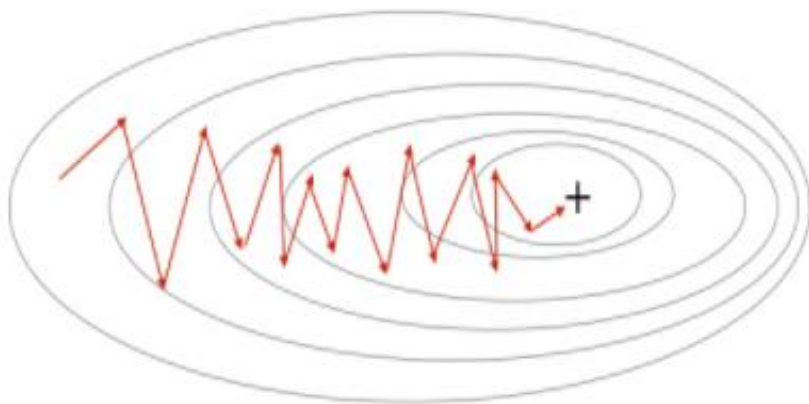
Au niveau neuronale	Multi. Vecteur-Matrice	$\vec{W}^T \leftarrow \vec{W}^T - \eta \frac{\partial \vec{E}}{\partial Y} \frac{\partial Y}{\partial \vec{W}}$ $\vec{W}^T \leftarrow \vec{W}^T - \eta \frac{\partial \vec{E}}{\partial Y} X^T$
-------------------------------	----------------------------------	---

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

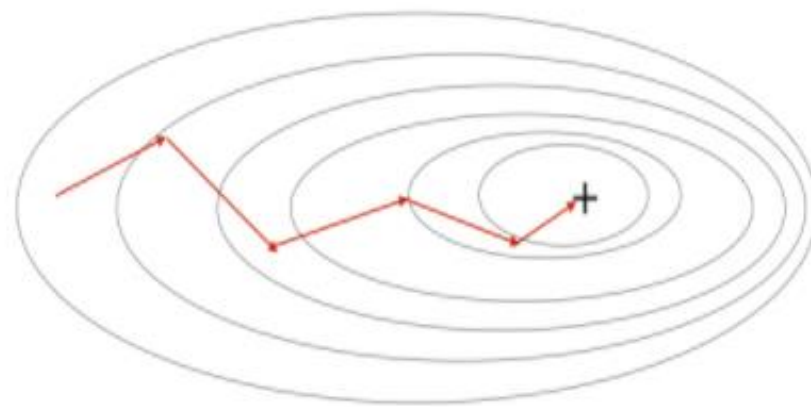
Pour plus de détails:

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

Stochastic Gradient Descent



Mini-Batch Gradient Descent



Comment initialiser un réseau de neurones?

$$W = ?$$

Initialisation

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

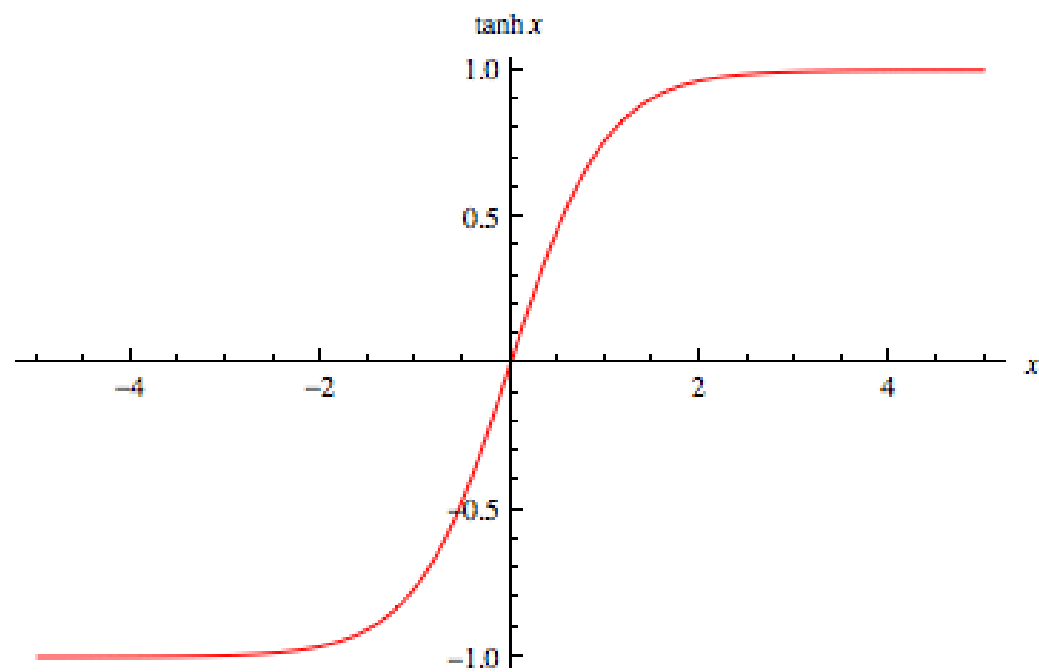
```
W_i=0.01*np.random.randn(H_i,H_im1)
```

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.

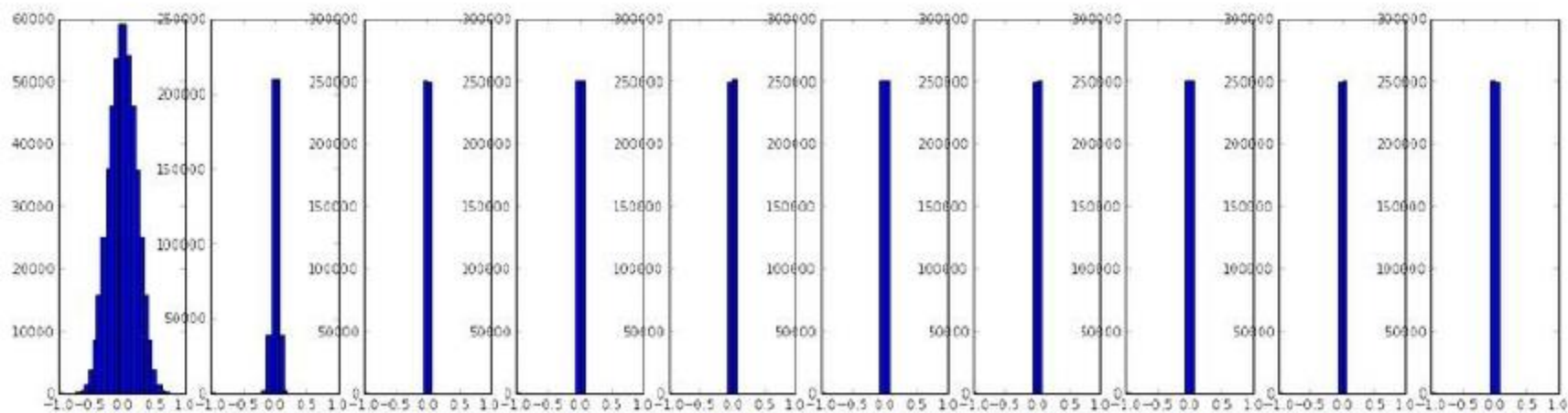
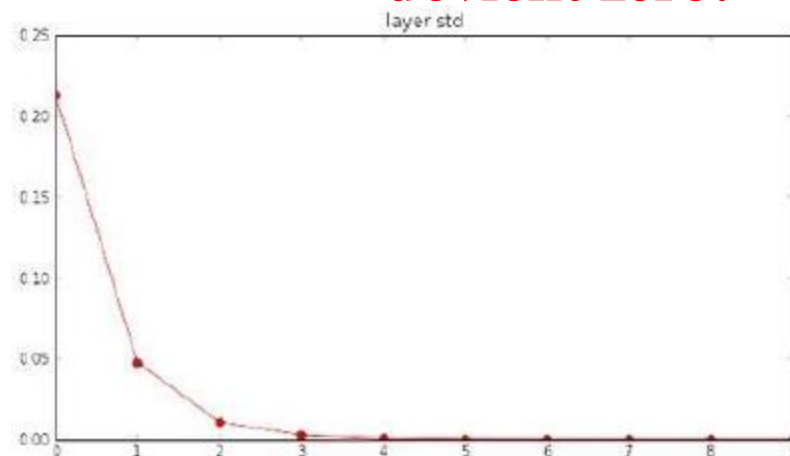
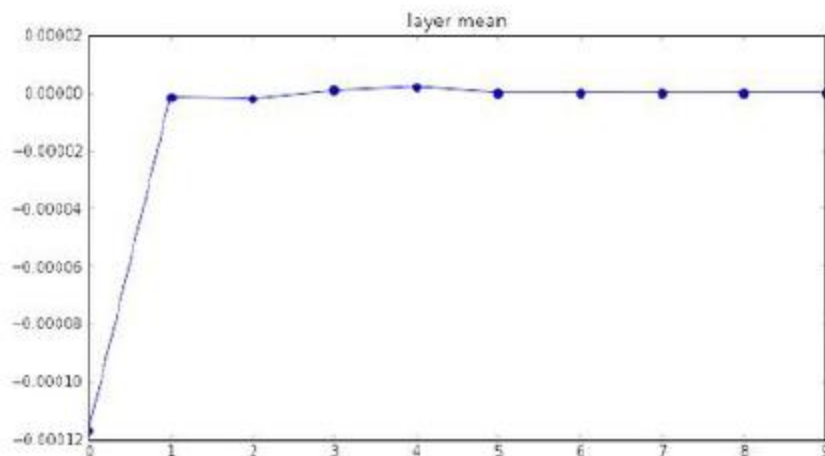
Rappel tanh...



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 0.047551
 hidden layer 3 had mean 0.000002 and std 0.006636
 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

Histogrammes des valeurs de sortie des 10 couches.

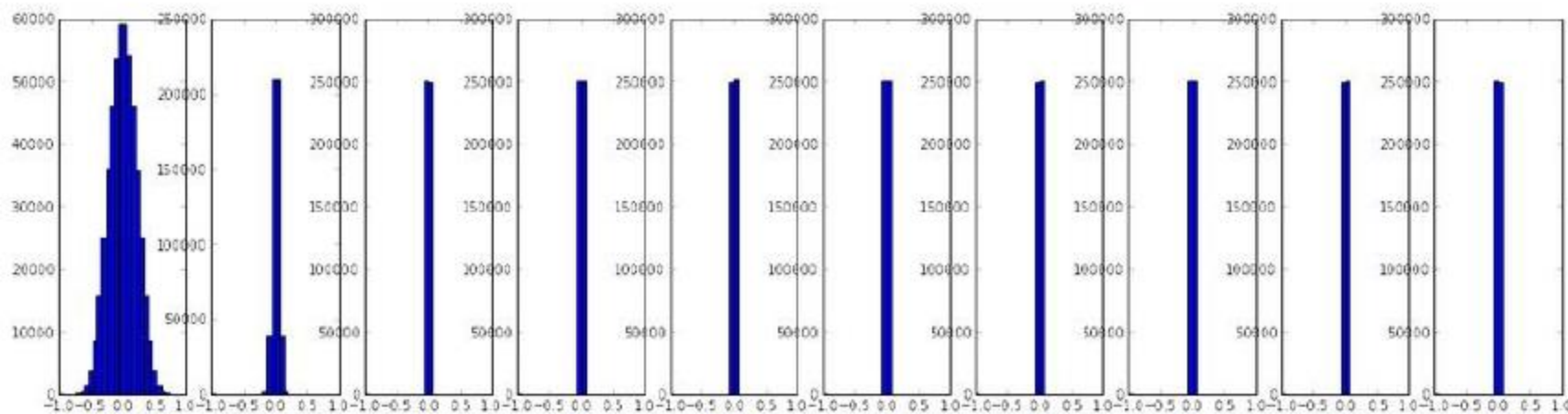
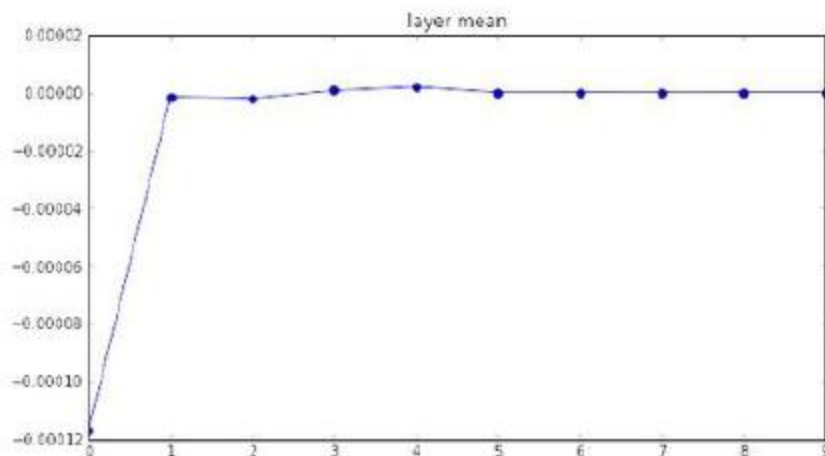
L'activation des couches devient zéro!



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 0.047551
 hidden layer 3 had mean 0.000002 and std 0.006636
 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

Histogrammes des valeurs de sortie des 10 couches.

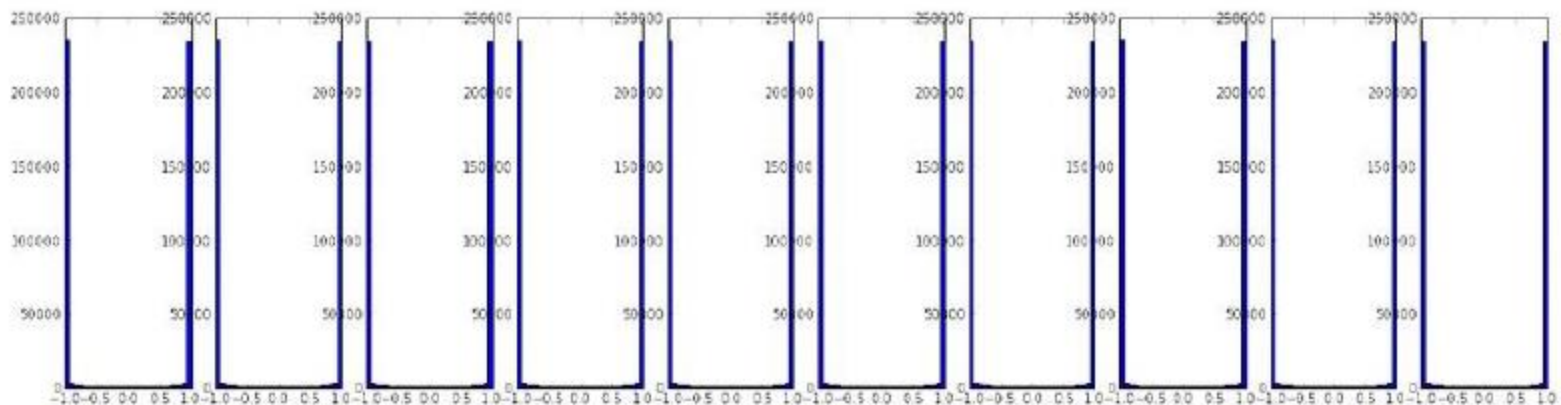
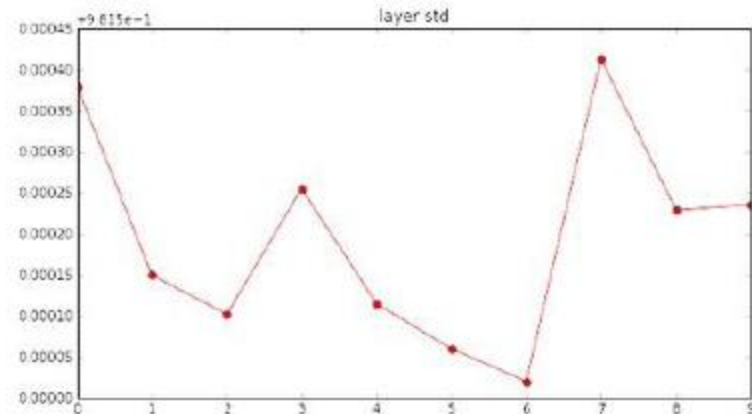
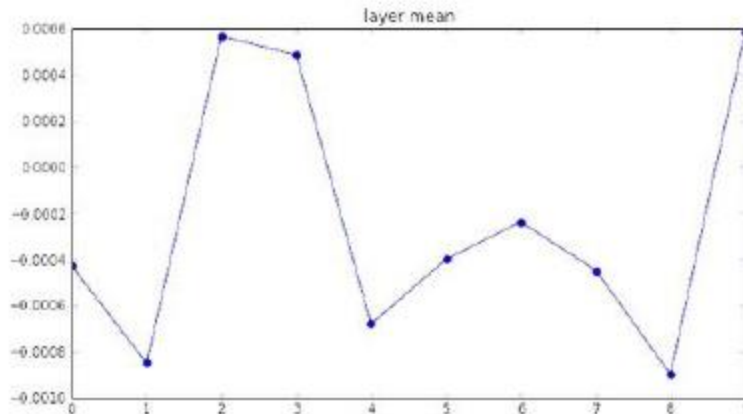
Vous voyez ce qui arrive
 à la *forward pass* et la
rétropropagation?



`W_i=np.random.randn(H_i,H_im1)*1`

input layer had mean 0.001800 and std 1.001311
 hidden layer 1 had mean -0.000430 and std 0.981879
 hidden layer 2 had mean -0.000849 and std 0.981649
 hidden layer 3 had mean 0.000566 and std 0.981601
 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

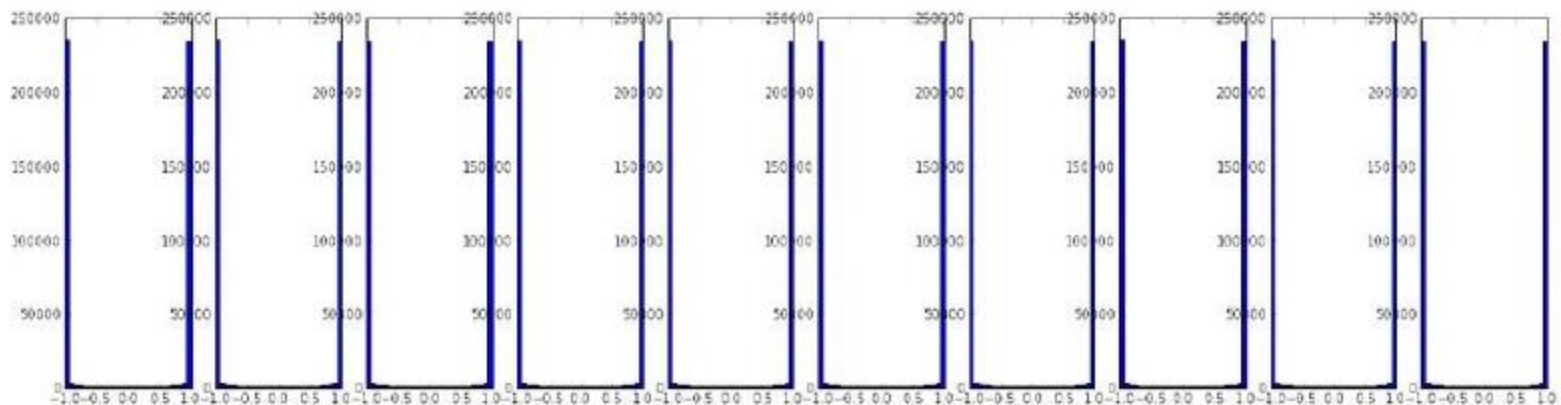
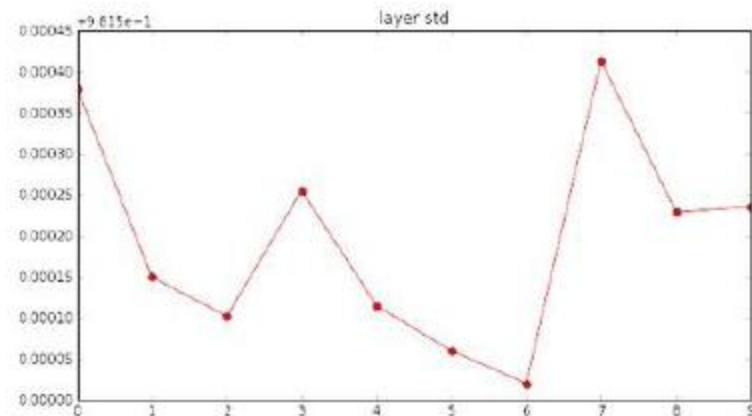
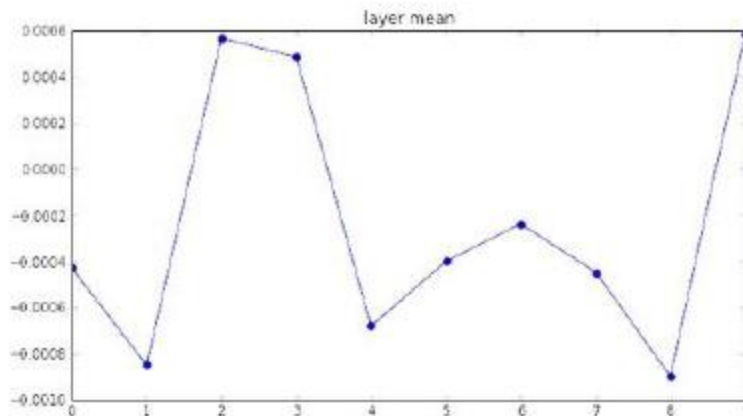
1 au lieu de 0.01



`W_i=np.random.randn(H_i,H_im1)*1`

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
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
```

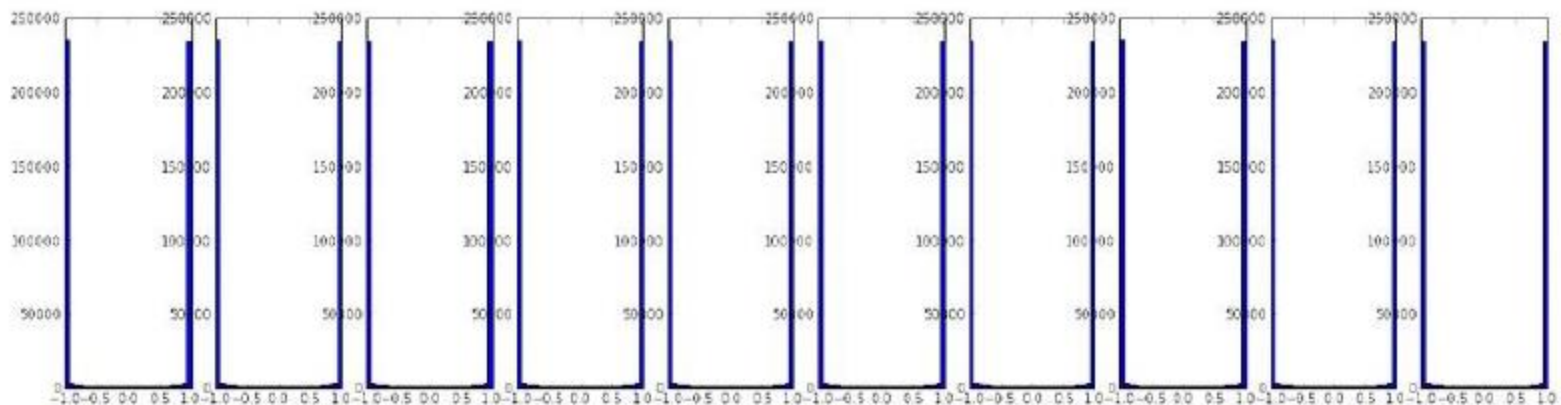
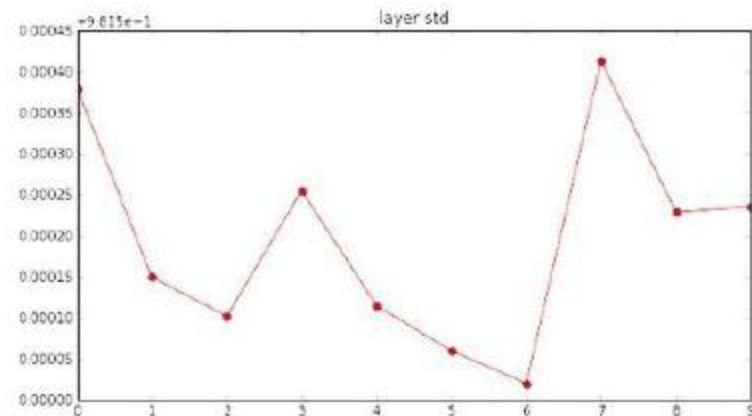
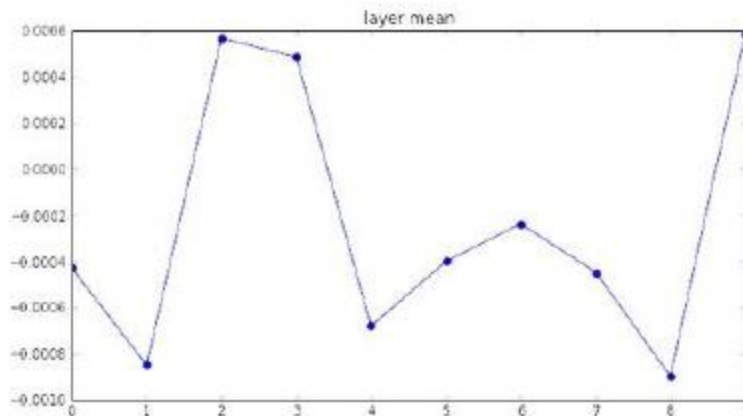
**La sortie des neurones
sature à 1 ou -1**



`W_i=np.random.randn(H_i,H_im1)*1`

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
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
```

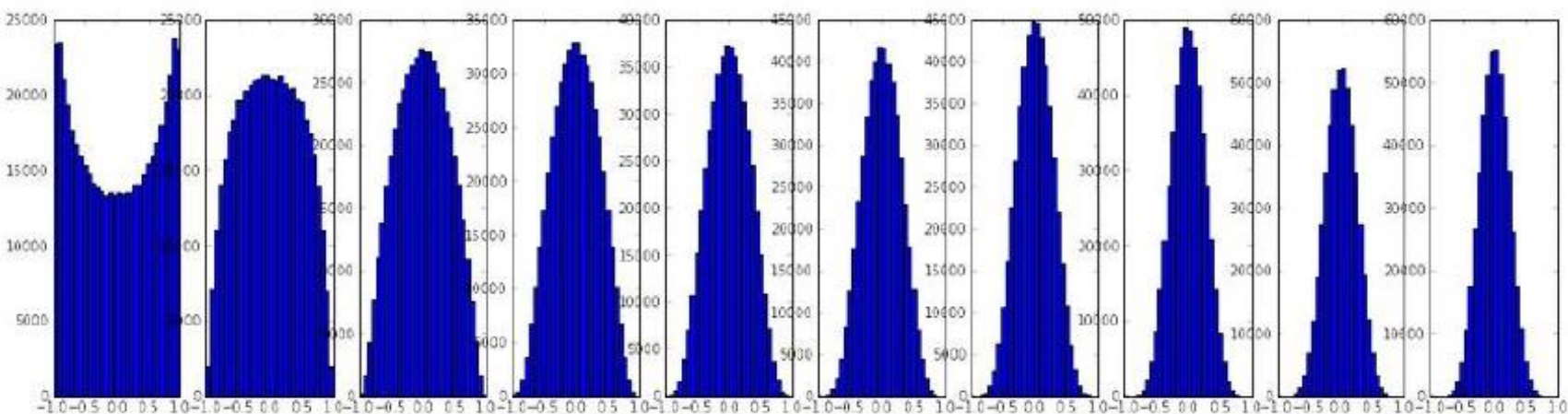
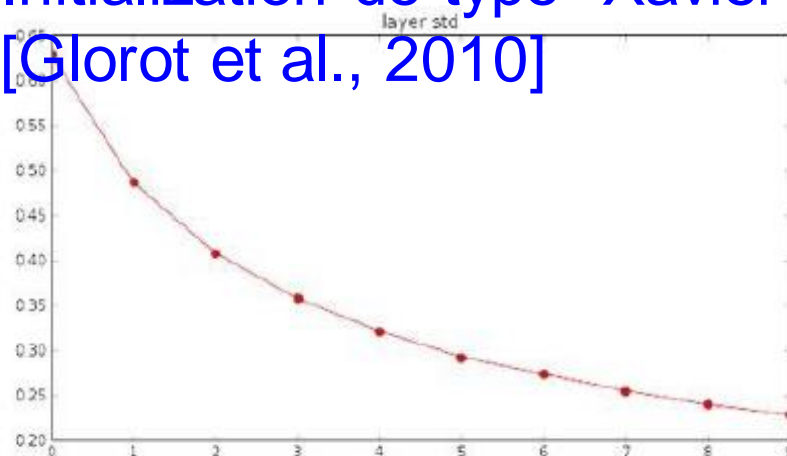
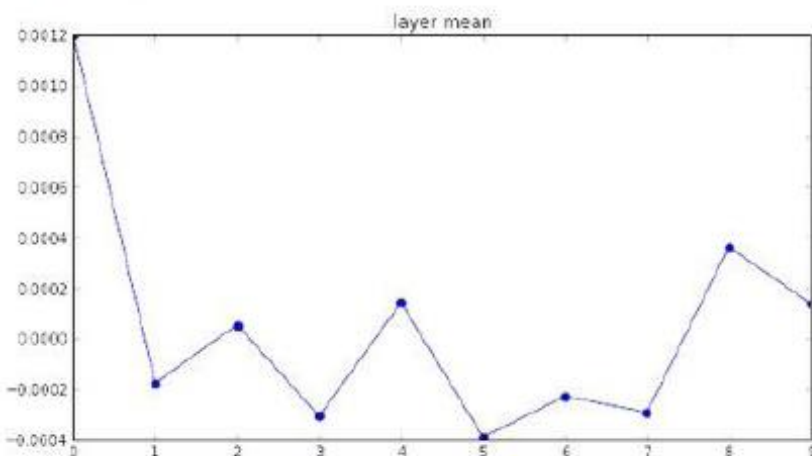
Vous voyez ce qui arrive
à la *forward pass* et la
rétropropagation?



input layer had mean 0.001800 and std 1.001311
 hidden layer 1 had mean 0.001198 and std 0.627953
 hidden layer 2 had mean -0.000751 and std 0.660117
 hidden layer 3 had mean 0.000055 and std 0.407723
 hidden layer 4 had mean -0.000306 and std 0.357108
 hidden layer 5 had mean 0.000142 and std 0.320917
 hidden layer 6 had mean -0.000389 and std 0.292116
 hidden layer 7 had mean -0.000228 and std 0.273387
 hidden layer 8 had mean -0.000291 and std 0.254935
 hidden layer 9 had mean 0.000361 and std 0.239266
 hidden layer 10 had mean 0.000139 and std 0.228008

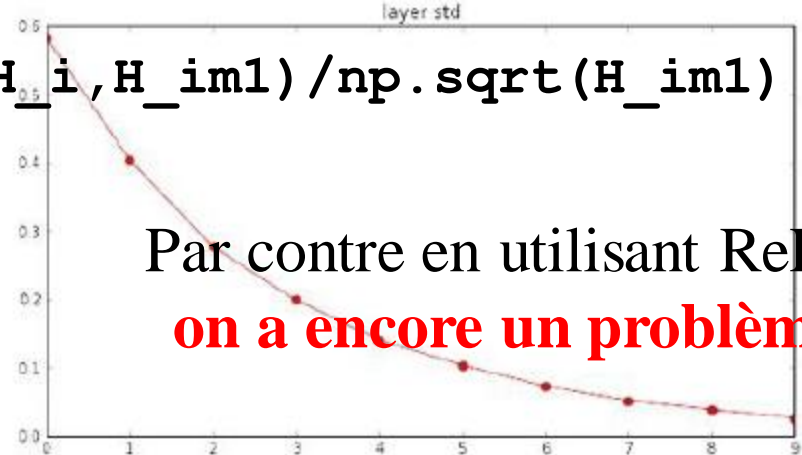
$W_i = \text{np.random.randn}(H_i, H_{i+1}) / \text{np.sqrt}(H_{i+1})$

Initialization de type “Xavier”
[Glorot et al., 2010]

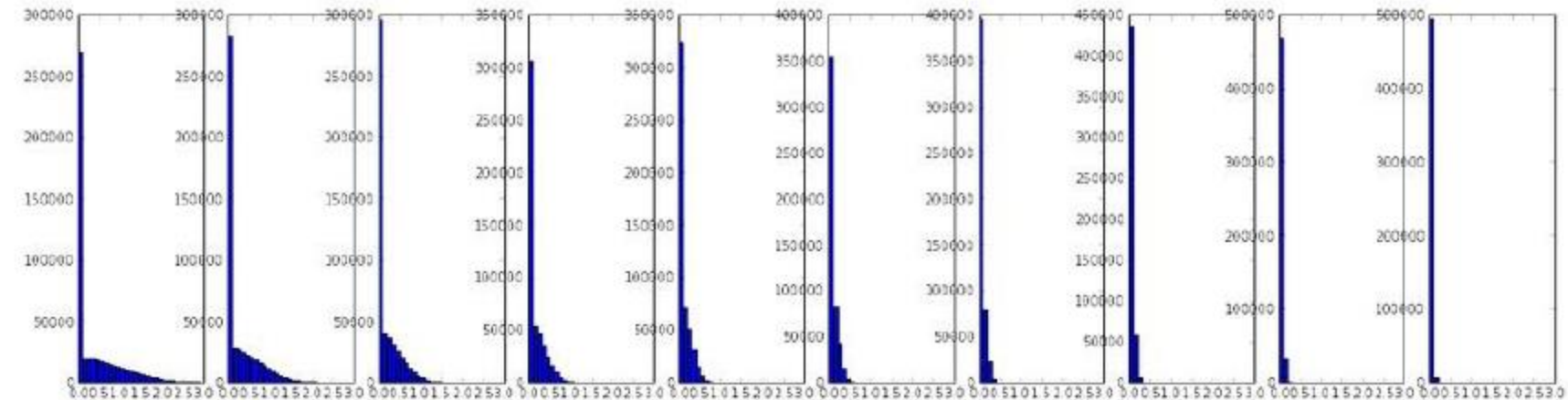


input layer had mean 0.000501 and std 0.999444
 hidden layer 1 had mean 0.398623 and std 0.582273
 hidden layer 2 had mean 0.008525 and std 0.037913
 hidden layer 3 had mean 0.186076 and std 0.276912
 hidden layer 4 had mean 0.136442 and std 0.198685
 hidden layer 5 had mean 0.099568 and std 0.140299
 hidden layer 6 had mean 0.072234 and std 0.103280
 hidden layer 7 had mean 0.049775 and std 0.072748
 hidden layer 8 had mean 0.035138 and std 0.051572
 hidden layer 9 had mean 0.025404 and std 0.038583
 hidden layer 10 had mean 0.018408 and std 0.026076

$W_i = \text{np.random.randn}(H_i, H_{i+1}) / \text{np.sqrt}(H_{i+1})$

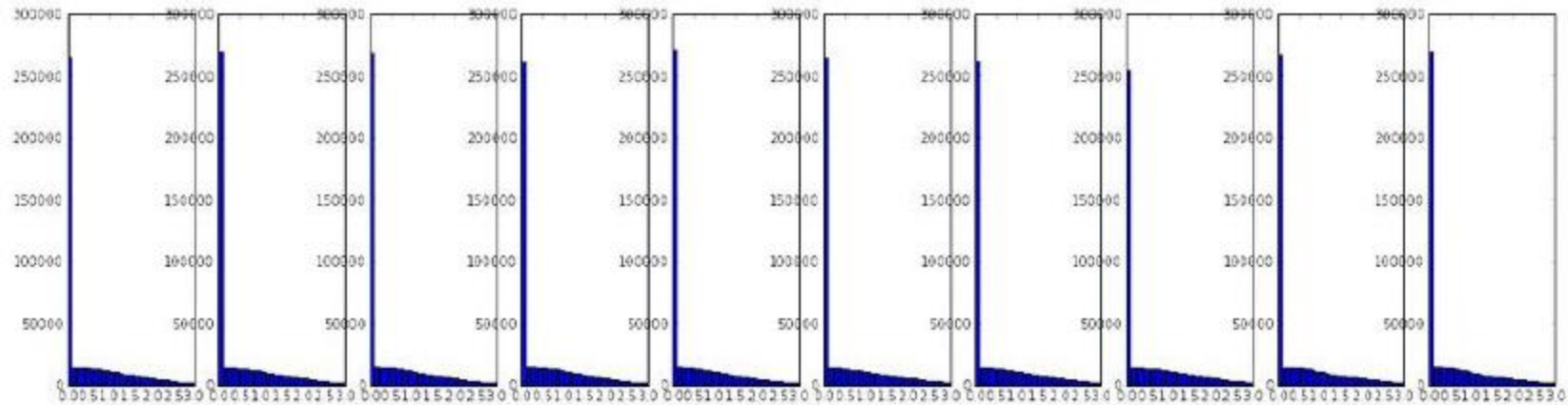
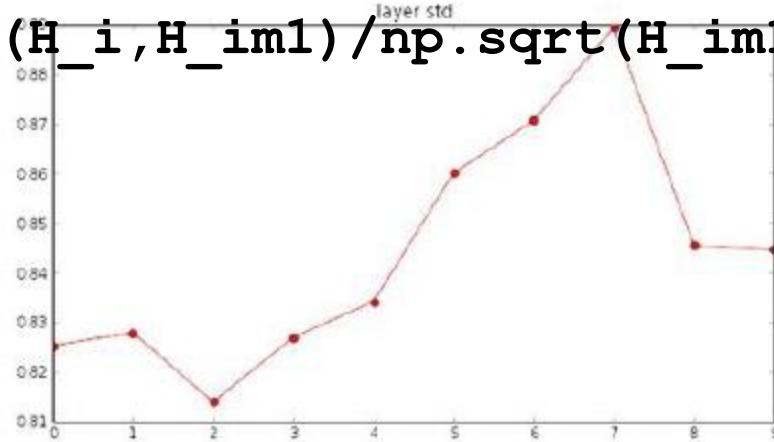
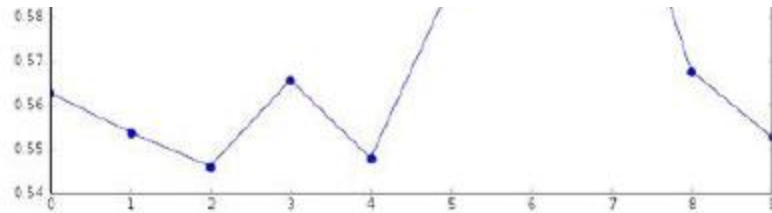


Par contre en utilisant ReLU
 on a encore un problème



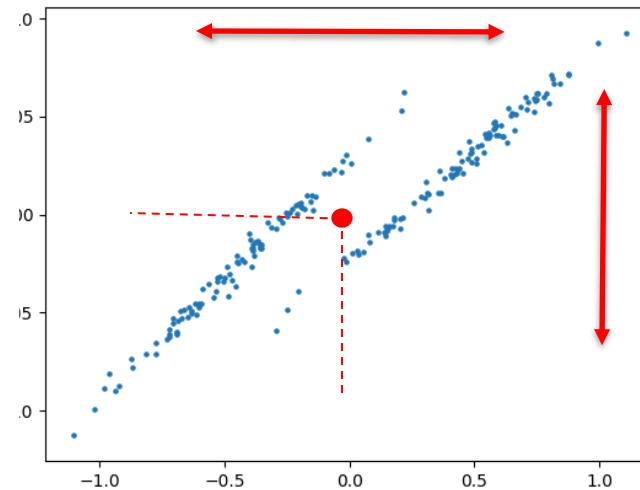
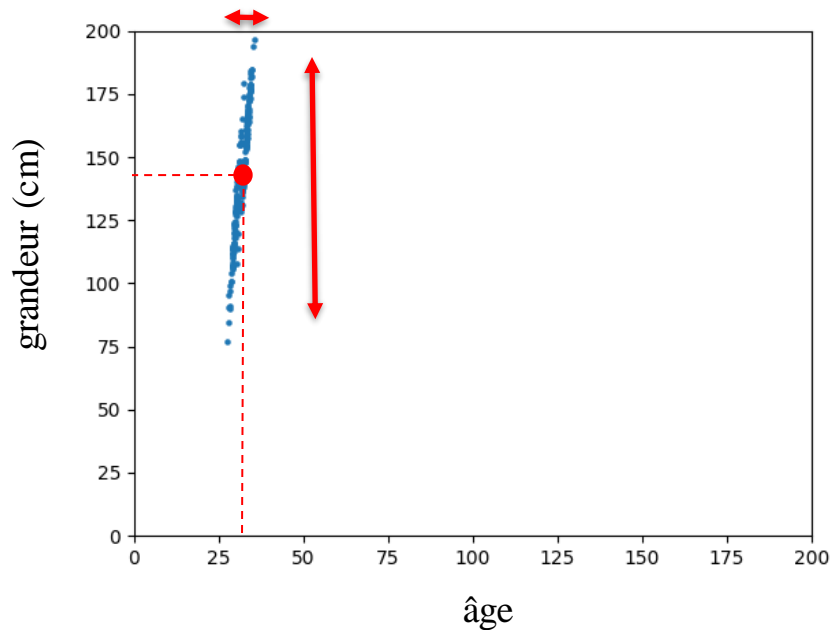
`W_i = np.random.randn(H_i, H_im1) / np.sqrt(H_im1/2)`

He et al., 2015
(note additional /2)



Prétraitement des données

Centrer et normaliser les données d'entrée



```
X=np.mean(X,axis=0)
```

```
X/=np.std(X,axis=0)
```

Les « sanity checks » ou
vérifications diligentes

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**.

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

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

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


```
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
```

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**.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of
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 parameter

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  
loss, grad = two_layer_net(X_train, model, y_train, 1e3) # crank up regulari  
print loss
```

3.06859716482

loss went up, good. (sanity check)

Lets try to train now...
Sanity checks

Tip: Make sure that
3. Toujours s'assurer qu'on peut « over-fitter » sur un petit nombre
de données.
you can overfit very
small portion of the
training data

Very small loss,
train accuracy 1.00,
nice!

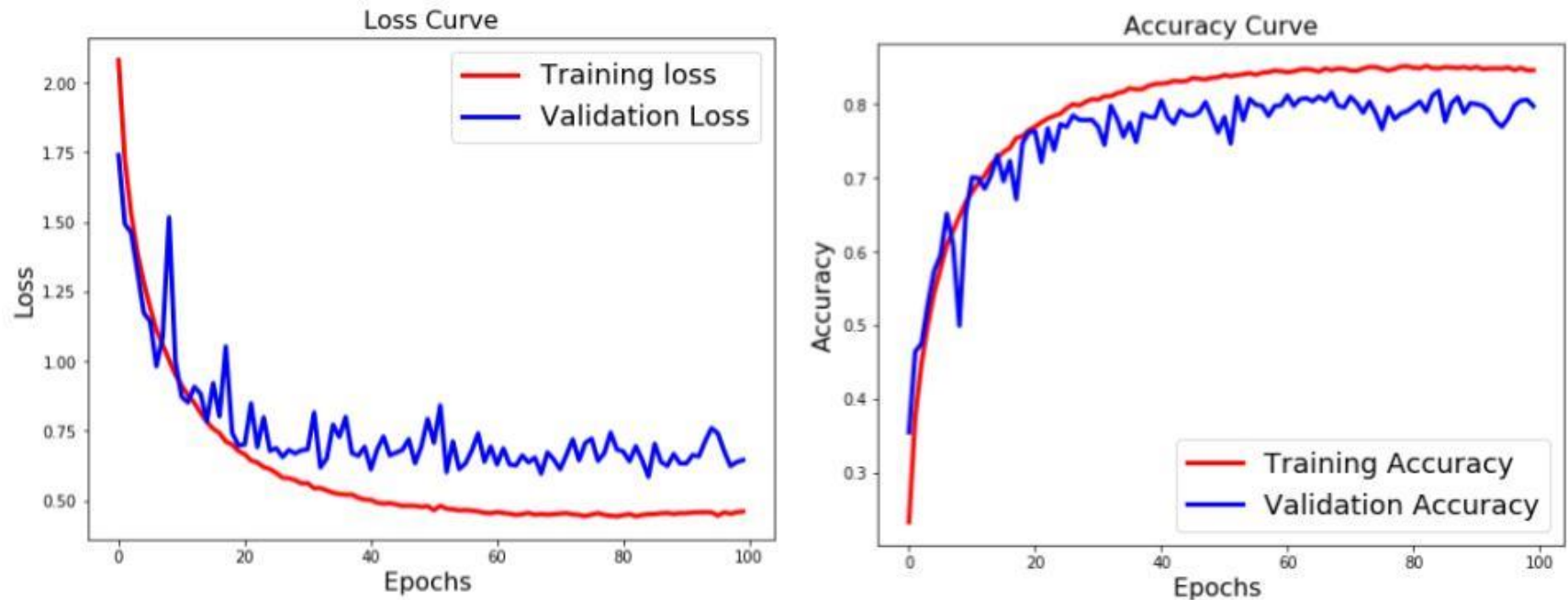
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, output size
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_train, y_train,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate=0.01,
                                  num_batches=100,
                                  learning_rate=1e-3, verbose=1)
```

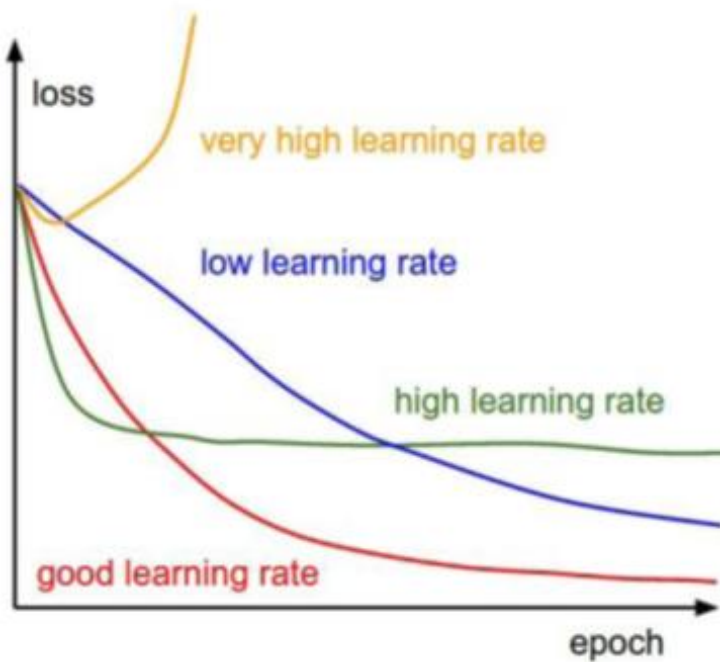
```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.000000
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.000000
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.000000
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.000000
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.000000
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.000000
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.000000
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.000000
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.000000
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.000000
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.000000
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.000000
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.000000
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.000000
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.000000
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.000000
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.000000
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.000000
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.000000
Finished epoch 20 / 200: cost 1.305760, train: 0.650000, val 0.000000
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 0.000000
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 0.000000
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 0.000000
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 0.000000
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 0.000000
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 0.000000
finished optimization. best validation accuracy: 1.000000
```

Sanity checks

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





s courbes d'apprentissage et de validation

Sanity checks

4. 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

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

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

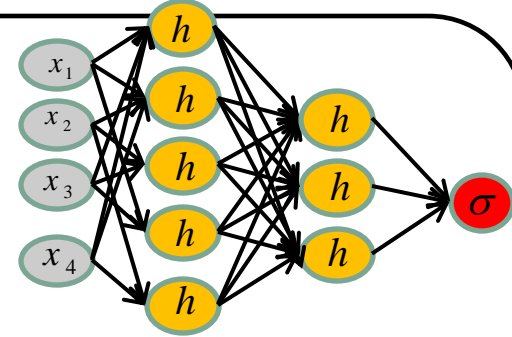
$$\nabla E(W) \approx \frac{E(W + H) - E(W)}{H}$$

En calculant

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

Vérification du gradient

(exemple)



W

W+h

gradient W

$$W_{00} = 0.34$$

$$W_{00} = 0.34 + 0.0001$$

$$-2.5 = (1.25322 - 1.25347) / 0.0001$$

$$W_{01} = -1.11$$

$$W_{01} = -1.11$$

$$W_{02} = 0.78$$

$$W_{02} = 0.78$$

...

...

$$W_{20} = -3.1$$

$$W_{20} = -3.1$$

$$W_{21} = -1.5,$$

$$W_{21} = -1.5,$$

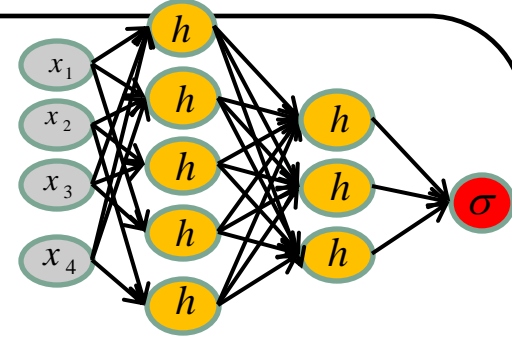
$$W_{22} = 0.33$$

$$W_{22} = 0.33$$

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

Vérification du gradient

(exemple)



W

$$W_{00} = 0.34$$

$$W_{01} = -1.11$$

$$W_{02} = 0.78$$

...

$$W_{20} = -3.1$$

$$W_{21} = -1.5,$$

$$W_{22} = 0.33$$

W+h

$$W_{00} = 0.34$$

$$W_{01} = -1.11 + 0.0001$$

$$W_{02} = 0.78$$

...

$$W_{20} = -3.1$$

$$W_{21} = -1.5,$$

$$W_{22} = 0.33$$

gradient W

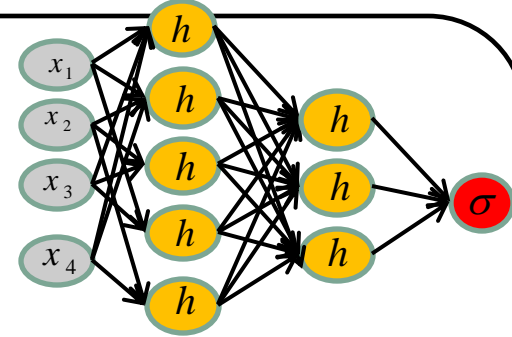
$$-2.5$$

$$0.6 = (1.25353 - 1.25347) / 0.0001$$

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

Vérification du gradient

(exemple)



W

$$W_{00} = 0.34$$

$$W_{01} = -1.11$$

$$W_{02} = 0.78$$

...

$$W_{20} = -3.1$$

$$W_{21} = -1.5,$$

$$W_{22} = 0.33$$

W+h

$$W_{00} = 0.34$$

$$W_{01} = -1.11$$

$$W_{02} = 0.78 + 0.0001$$

...

$$W_{20} = -3.1$$

$$W_{21} = -1.5,$$

$$W_{22} = 0.33$$

gradient W

$$-2.5$$

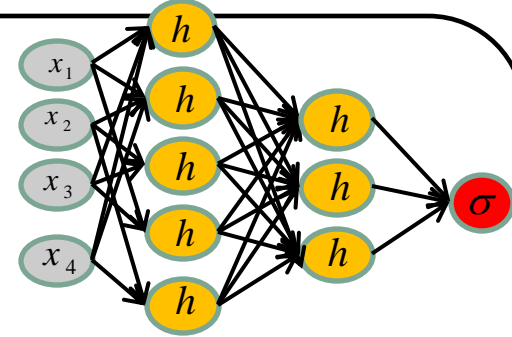
$$0.6$$

$$0.0 = (1.25347 - 125347) / 0.0001$$

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

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

$$W_{22} = 0.33$$

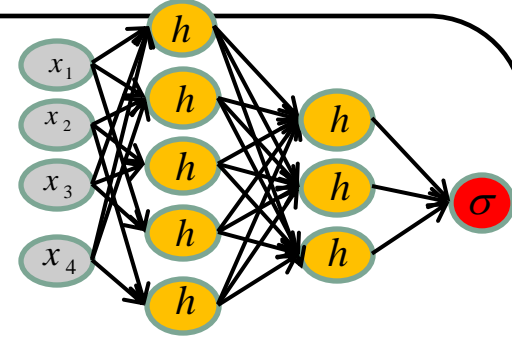
$$W_{22} = 0.33$$

-2.1

$$\mathbf{E(W)=1.25347}$$

Vérification du gradient

(exemple)



gradient W
(numérique)

-2.5

0.6

0.0

...

1.1

1.3

-2.1



gradient W
(retro-propagation)

-2.5

0.6

0.0

...

1.1

1.3

-2.1

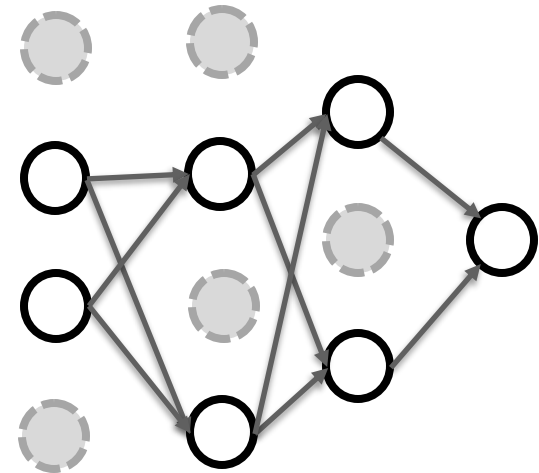
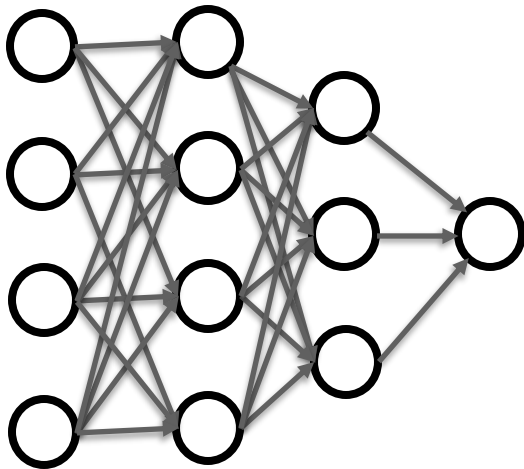
Vérification
réussie

Autres bonnes pratique

Dropout

Dropout

Forcer à zéro certains neurones de façon aléatoire à chaque itération



Dropout

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

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

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)
```


Dropout

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

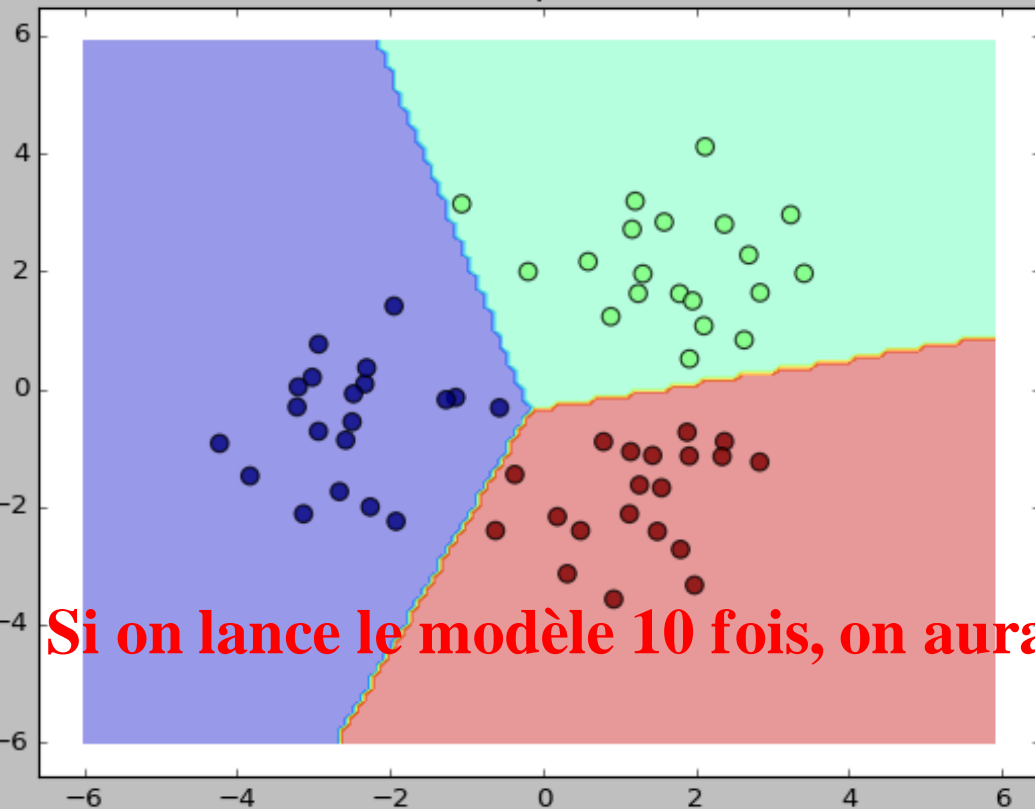
car *dropout* **ajoute du bruit** à la prédiction

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



masque aléatoire

Perceptron



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

$y_{w,1}(\vec{x})$ est max

$y_{w,0}(\vec{x})$ est max



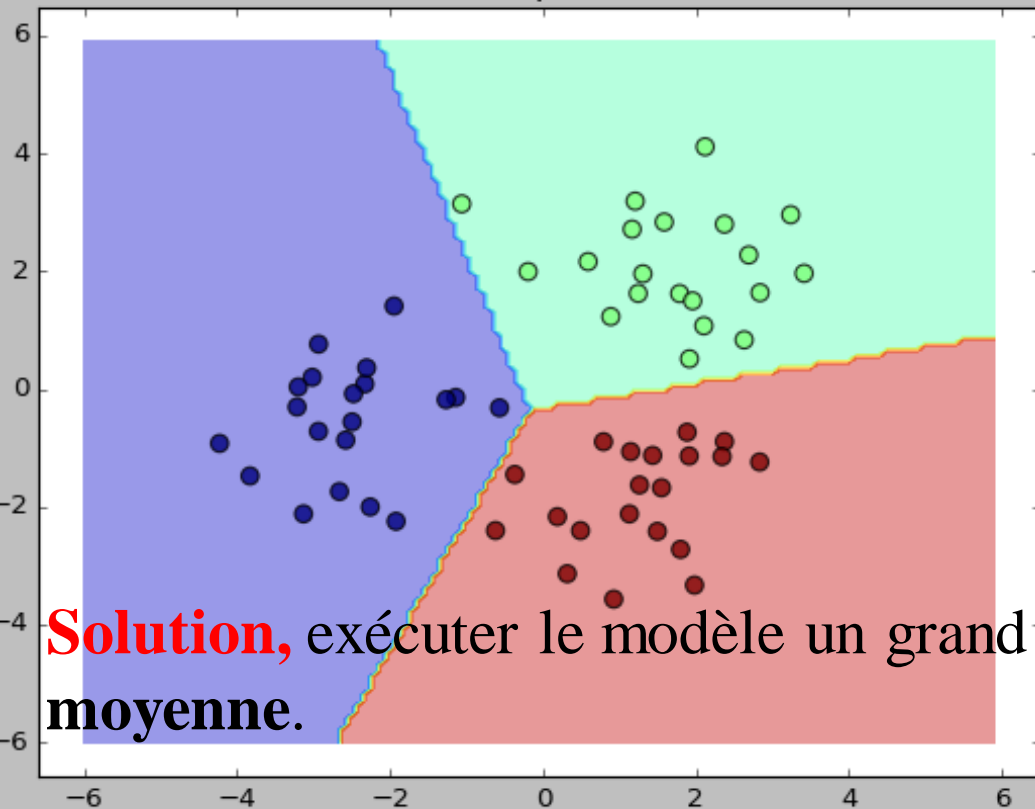
$y_{w,2}(\vec{x})$ est max

n.

1

[0.23658253	0.61960162	0.141098]
[0.23779425	0.51357115	0.23131764]
[0.16005442	0.68060227	0.14381585]
[0.16303195	0.50583392	0.24863461]
[0.24183069	0.51319834	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]

Perceptron



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

$y_{w,1}(\vec{x})$ est max

$y_{w,0}(\vec{x})$ est max



$y_{w,2}(\vec{x})$ est max

n.

1

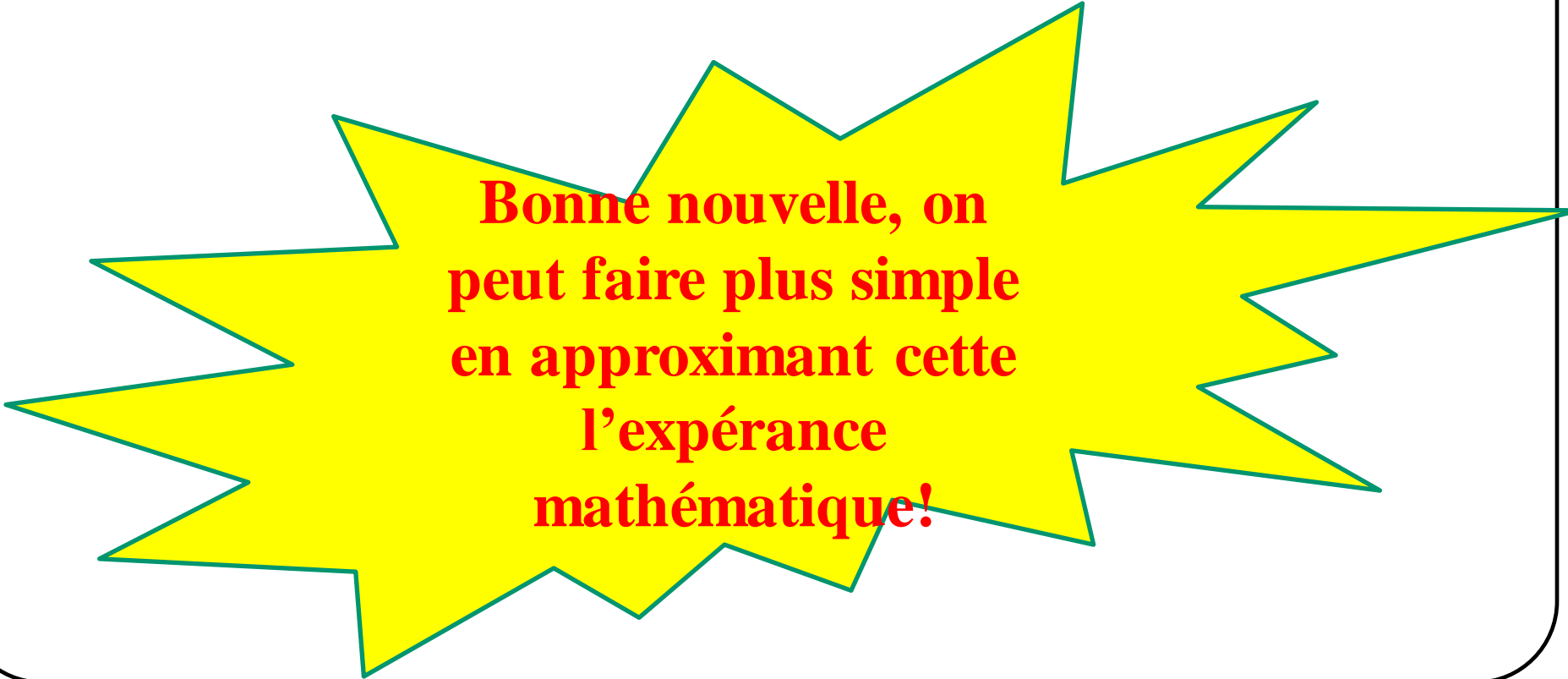
```
1644 0.141098 ]
5327 0.23131764]
[ 0.23658253 0.61960162 0.14381585]
[ 0.23779425 0.51357115 0.24863461]
[ 0.16005442 0.68060227 0.1593433 ]
[ 0.16303195 0.50583392 0.33113413]
[ 0.24183069 0.51319834 0.24497097]
[ 0.14521815 0.52006858 0.33471327]
[ 0.09952161 0.66276146 0.23771692]
[ 0.16172851 0.6044877 0.23378379]
```

(...)

[0.15933813, 0.65957005, 0.18109183]

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

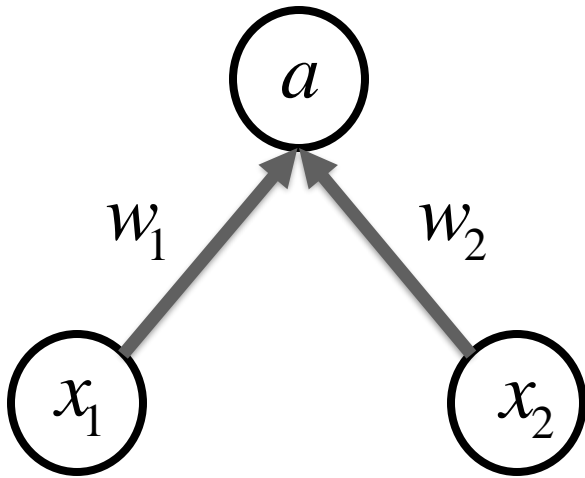
$$pred = E_z \left[y_W \left(\mathbf{x}, \mathbf{z} \right) \right] = \sum_i P(\mathbf{z}) y_W \left(\mathbf{x}, \mathbf{z} \right)$$



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

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*.



```
""" 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):
```

```
    # ensembled 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**

NOTE

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

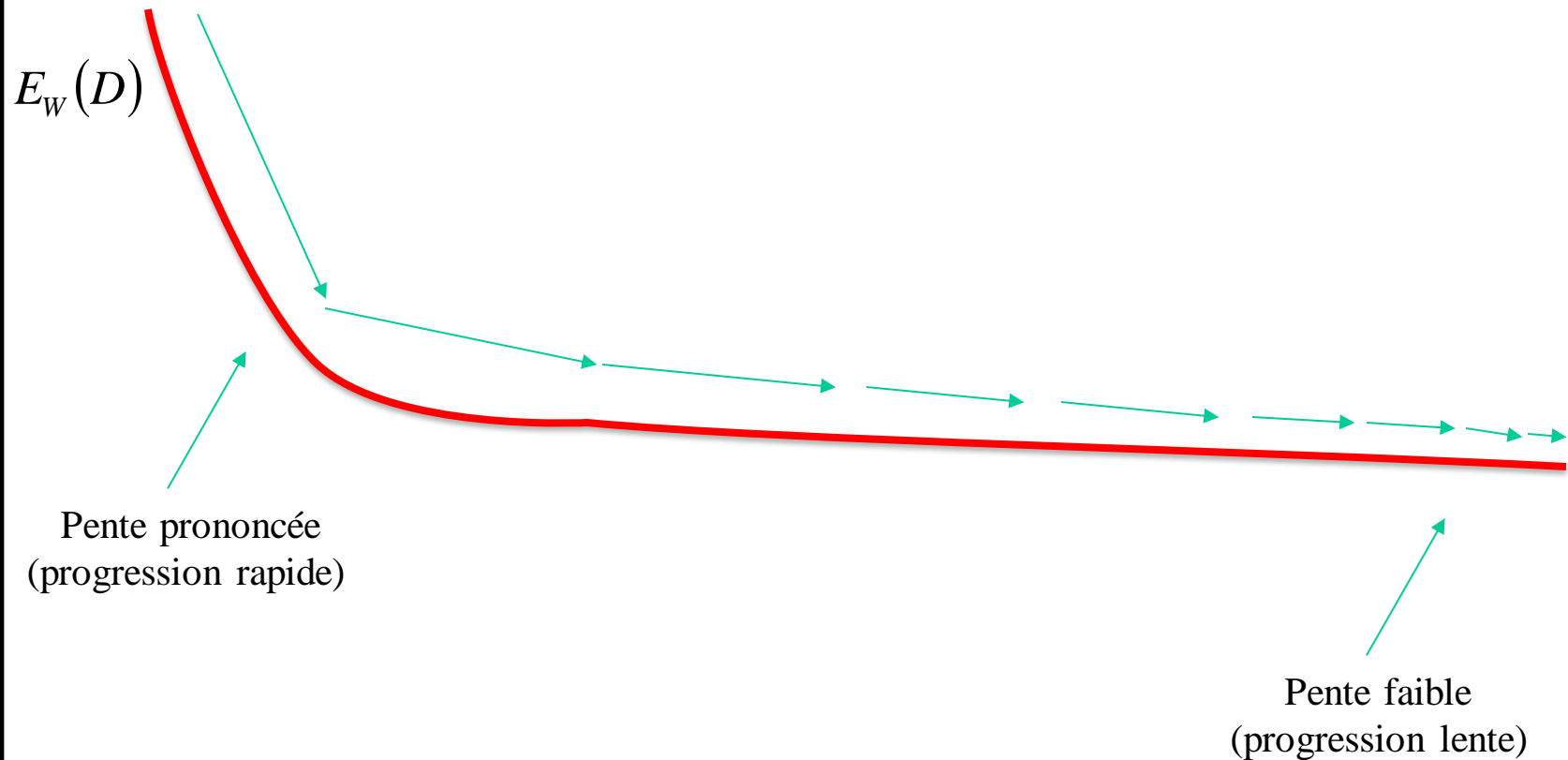
Descente de gradient **version améliorée**

Descente de gradient

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

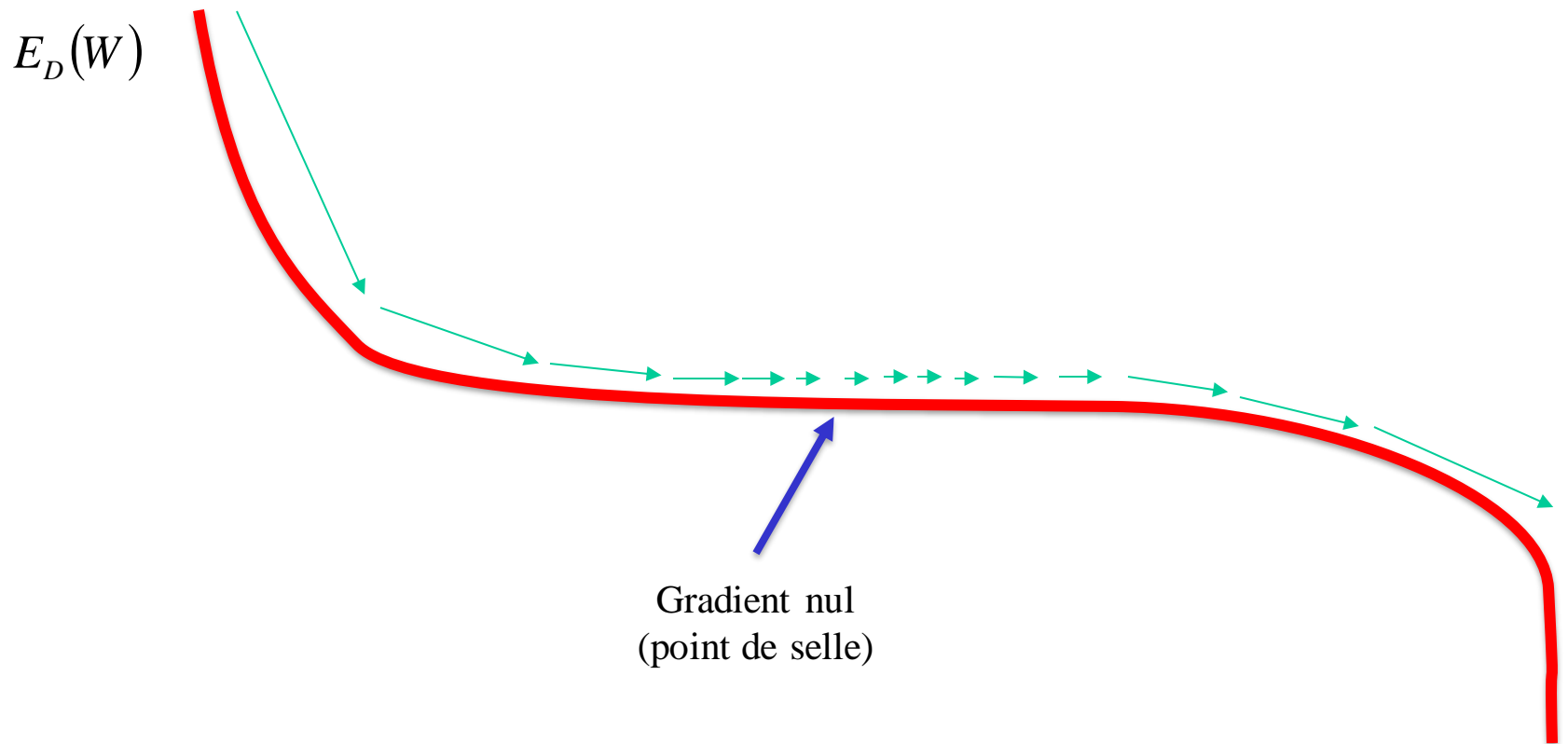
Descente de gradient : **problème**

Progrès quasi nul lorsque la pente est très faible



Descente de gradient : **problème**

Les points de selles sont fréquents en haute dimension



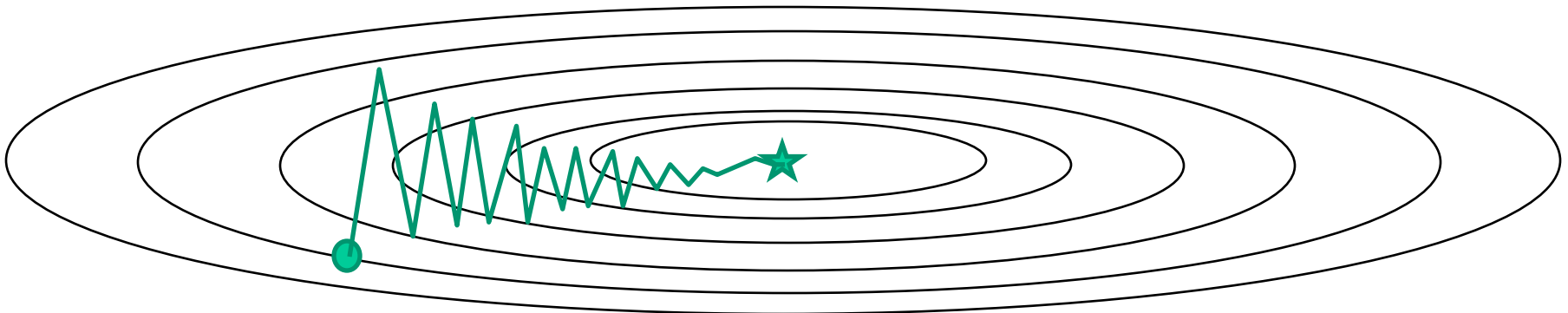
Descente de gradient : **problème**

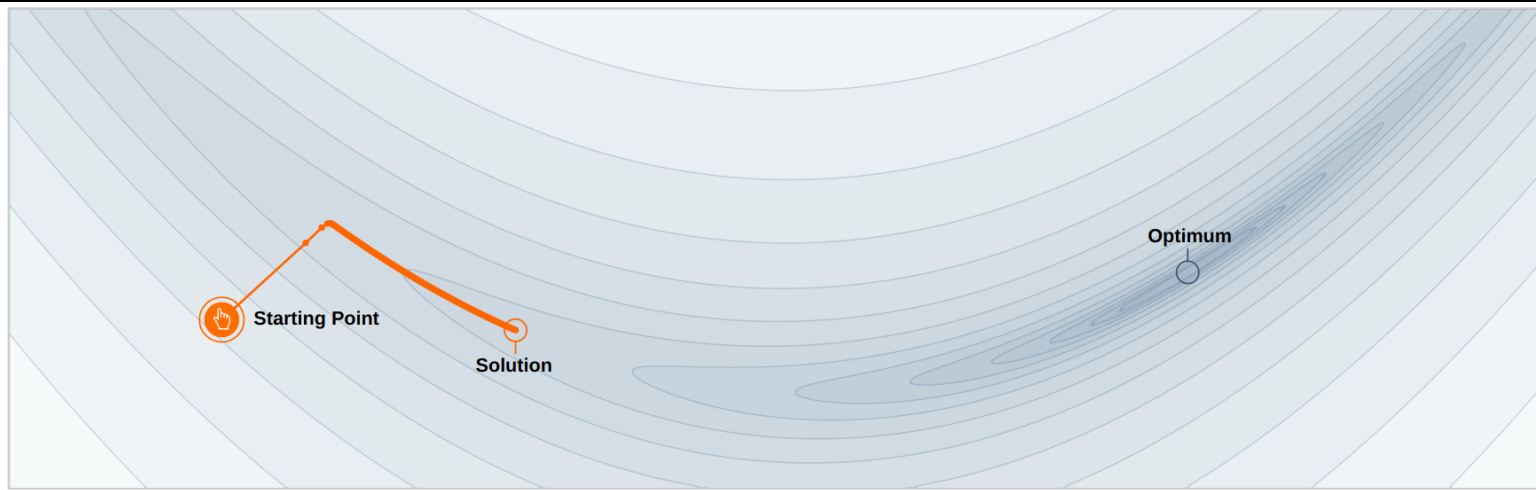
Qu'arrive-t-il si la fonction de coût (loss) a une pente prononcée dans une direction et moins prononcée dans une autre direction?

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.





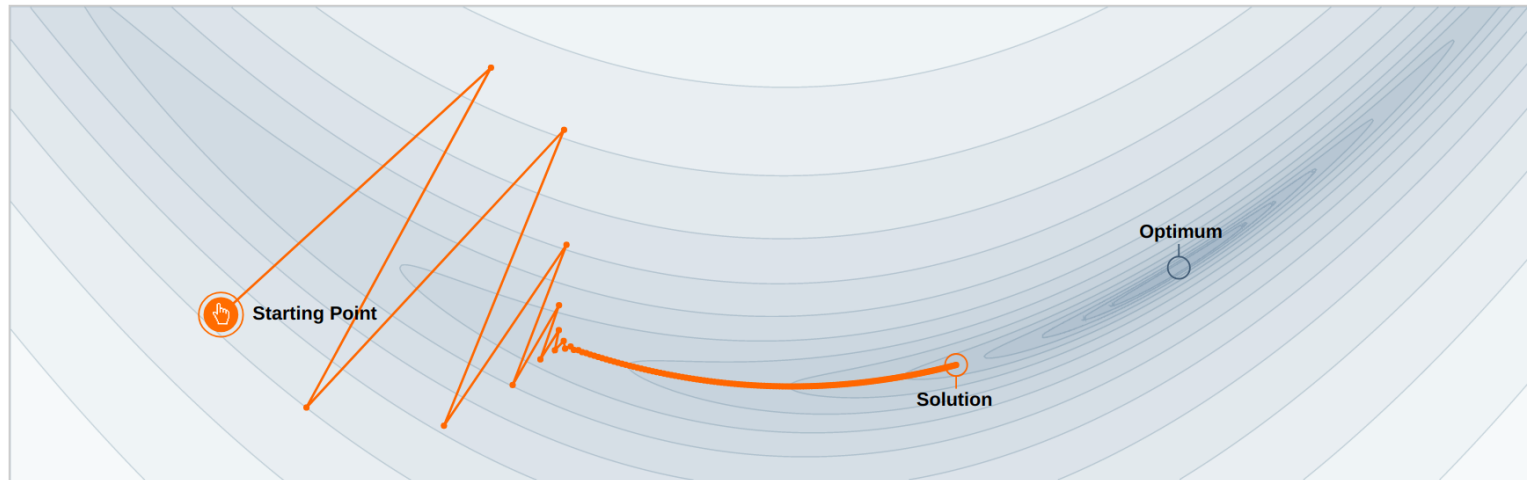
Step-size $\alpha = 0.0012$



Momentum $\beta = 0.0$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.0038$



Momentum $\beta = 0.0$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Descente de gradient + **Momentum**

Descente de gradient
stochastique

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla E_{x_n}^r(\mathbf{w}_t)$$

Descente de gradient
stochastique + **Momentum**

$$\mathbf{v}_{t+1} = \rho \mathbf{v}_t + \nabla E_{x_n}^r(\mathbf{w}_t)$$

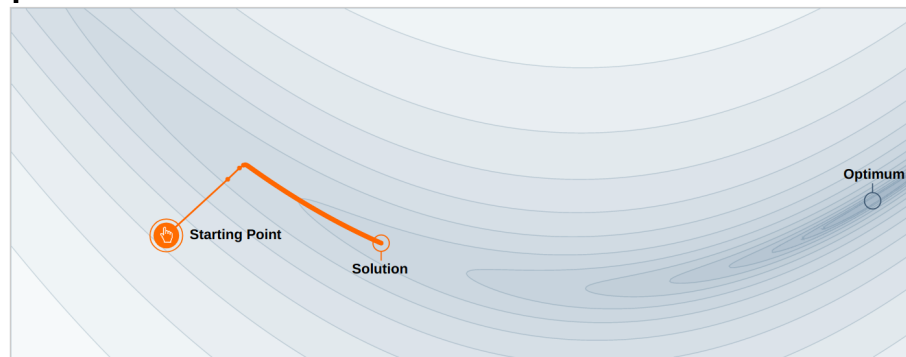
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{v}_{t+1}$$

Provient de l'équation de la vitesse

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

$$v_{t+1} = \rho v_t + \nabla E_{x_n}^r(w_t)$$

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

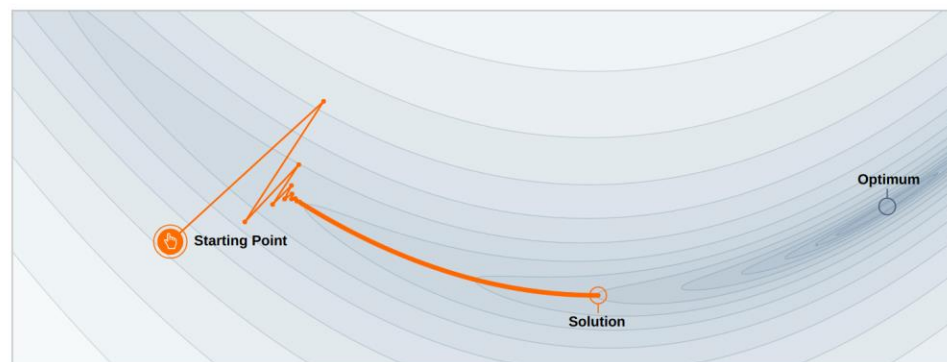


Step-size $\alpha = 0.0012$

Momentum $\beta = 0.0$



We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

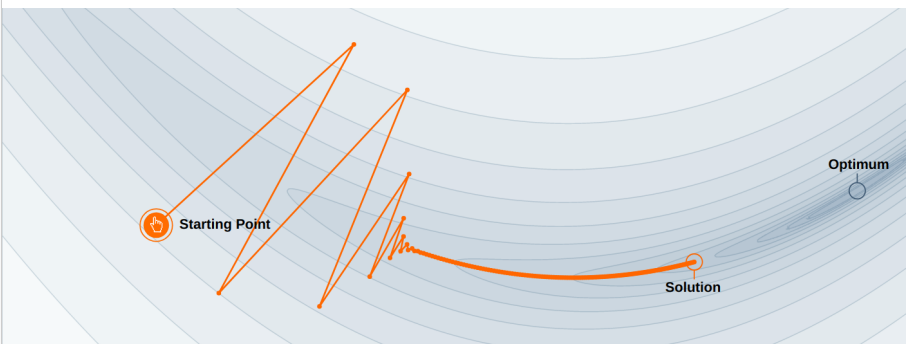


Step-size $\alpha = 0.0029$

Momentum $\beta = 0.0$



We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

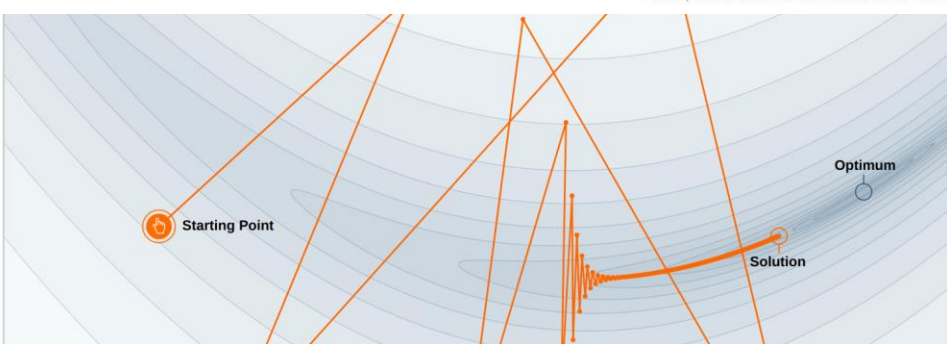


Step-size $\alpha = 0.0038$

Momentum $\beta = 0.0$



We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.0048$

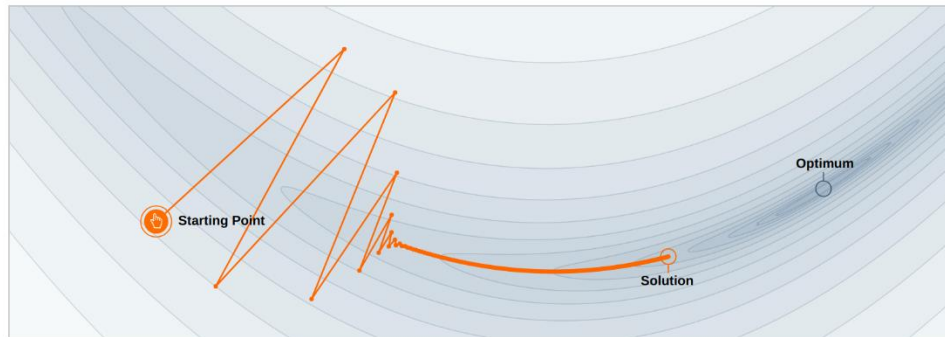
Momentum $\beta = 0.0$



We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

$$v_{t+1} = \rho v_t + \nabla E_r(w_t)$$

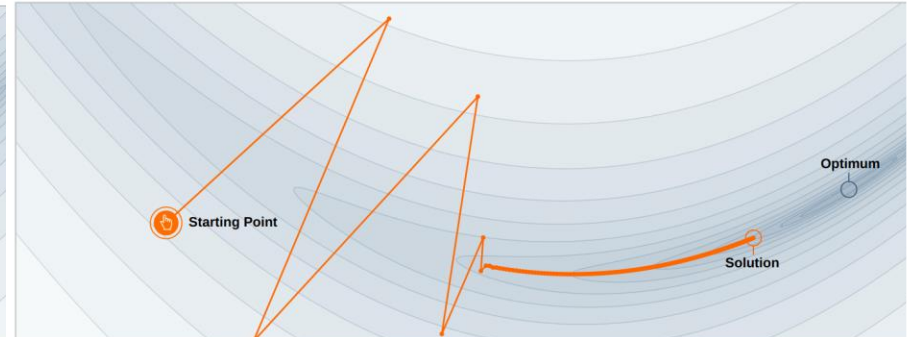
$$w_{t+1} = w_t - \eta v_{t+1}$$



Step-size $\alpha = 0.0038$

Momentum $\beta = 0.0$

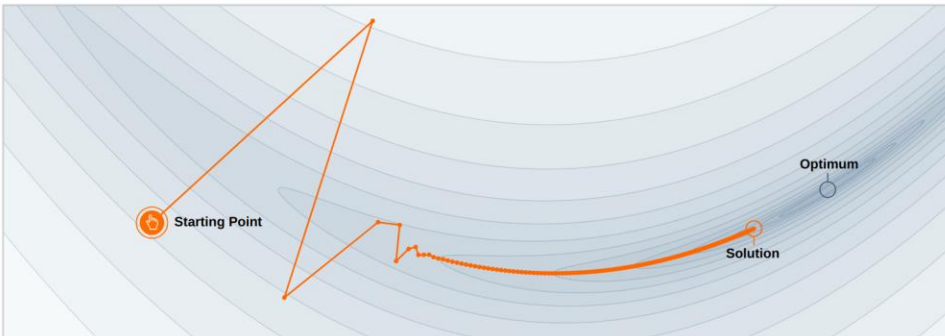
We often think of Momentum as a means of dampening speeding up the iterations, leading to faster convergence. It allows a larger range of step sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.0044$

Momentum $\beta = 0.11$

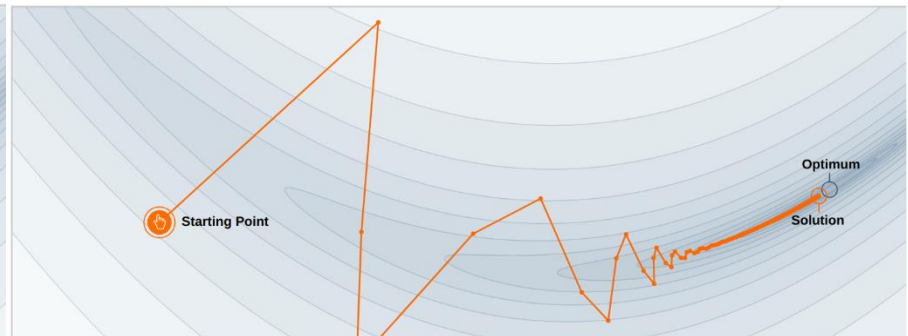
We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.0044$

Momentum $\beta = 0.32$

We often think of Momentum as a means of dampening speeding up the iterations, leading to faster convergence. It allows a larger range of step sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.0044$

Momentum $\beta = 0.65$

We often think of Momentum as a means of speeding up the iterations, leading to faster convergence. It allows a larger range of step sizes to be used, and creates its own oscillations. What is going on?

AdaGrad (décroissance automatique de η)

Descente de gradient
stochastique

AdaGrad

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

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

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

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

AdaGrad (décroissance automatique de η)

Descente de gradient
stochastique

AdaGrad

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

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

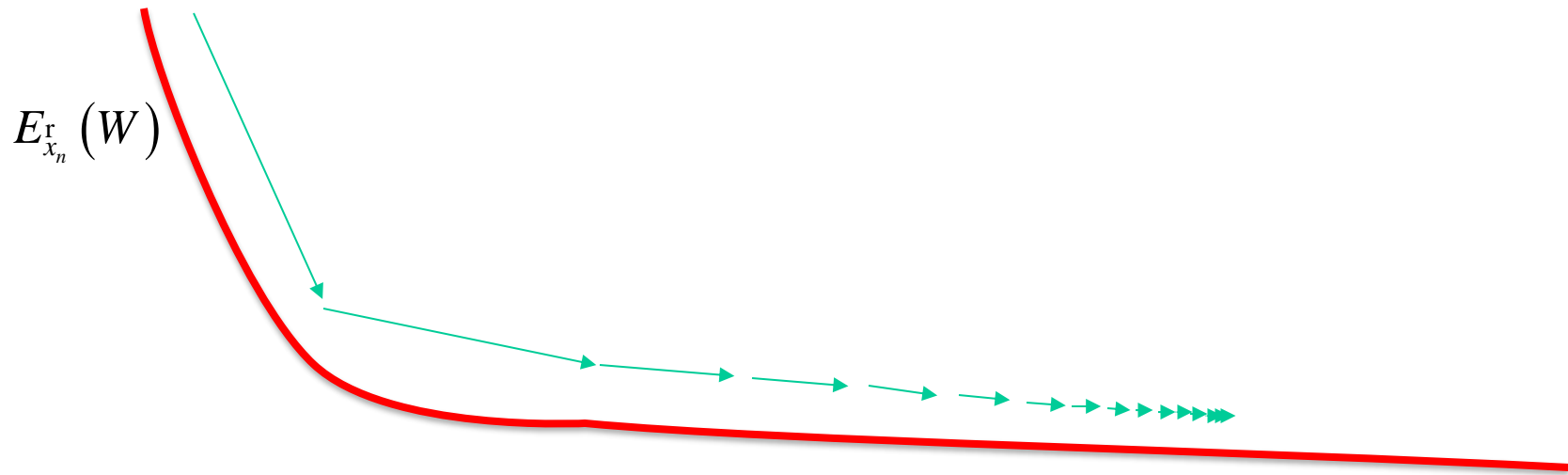
$$m_{t+1} = m_t + |dE_t|$$

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

η décroît sans cesse au fur
et à mesure de l'optimisation

AdaGrad (décroissance automatique de η)

Qu'arrive-t-il à long terme?



$$\frac{\eta}{m_{t+1} + \varepsilon} \rightarrow 0$$

RMSProp (AdaGrad amélioré)

AdaGrad

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

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

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

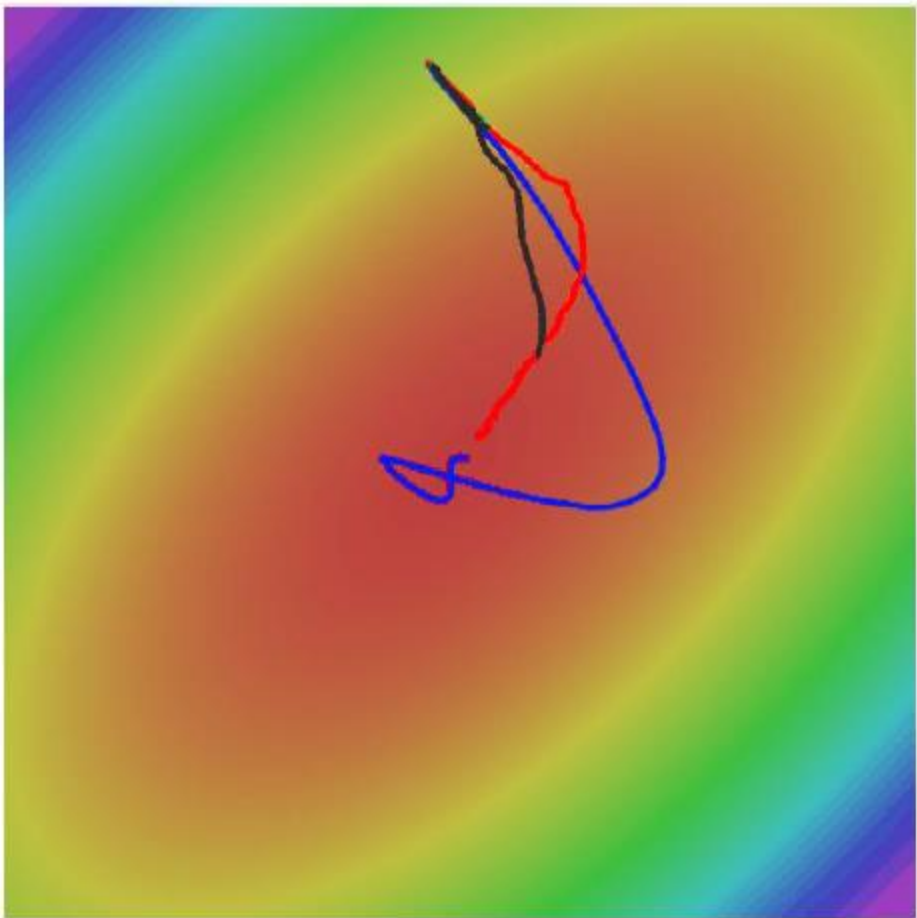
RMSProp

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

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

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

η décroît lorsque le gradient est élevé
 η augmente lorsque le gradient est faible



— SGD

— SGD+Momentum

— RMSProp

Adam (Combo entre Momentum et RMSProp)

Momentum

$$v_{t+1} = \rho v_t + \nabla E_{x_n}^r (w_t)$$

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

Adam

$$dE_t = \nabla E_{x_n}^r (w_t)$$

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

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

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

Adam (Combo entre Momentum et RMSProp)

Momentum

$$v_{t+1} = \rho v_t + \nabla E_{x_n}^r (w_t)$$

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

Adam

$$dE_t = \nabla E_{x_n}^r (w_t)$$

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

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

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

Momentum

Adam (Combo entre Momentum et RMSProp)

RMSProp

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

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

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

Adam

$$dE_t = \nabla E_{x_n}(\mathbf{w}_t)$$

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

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

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

RMSProp

Adam (Version complète)

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

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

for t=1 à num_iterations
for n=0 à N

$$dE_t = \nabla E_{x_n} (w_t)$$

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

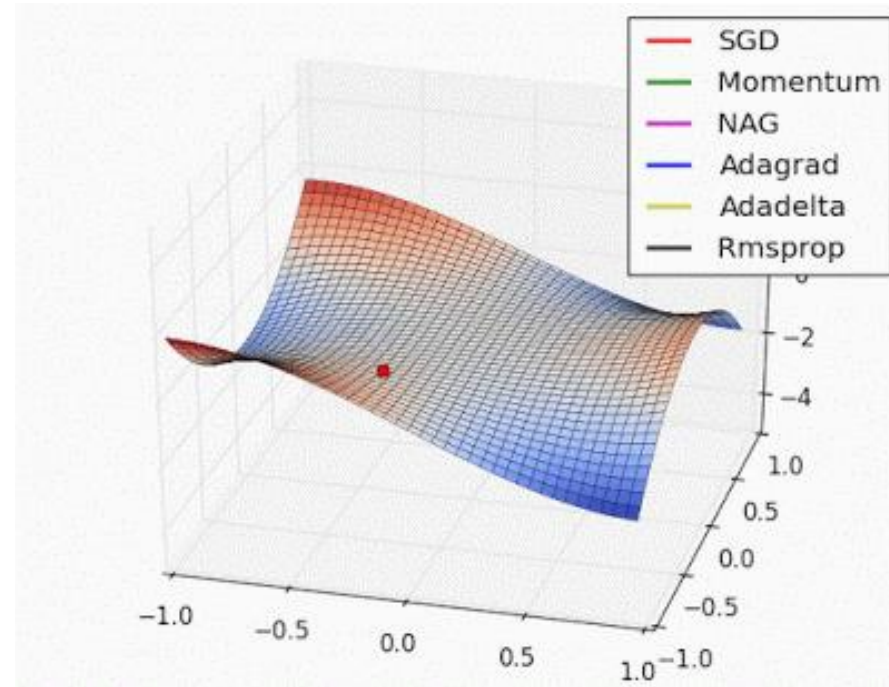
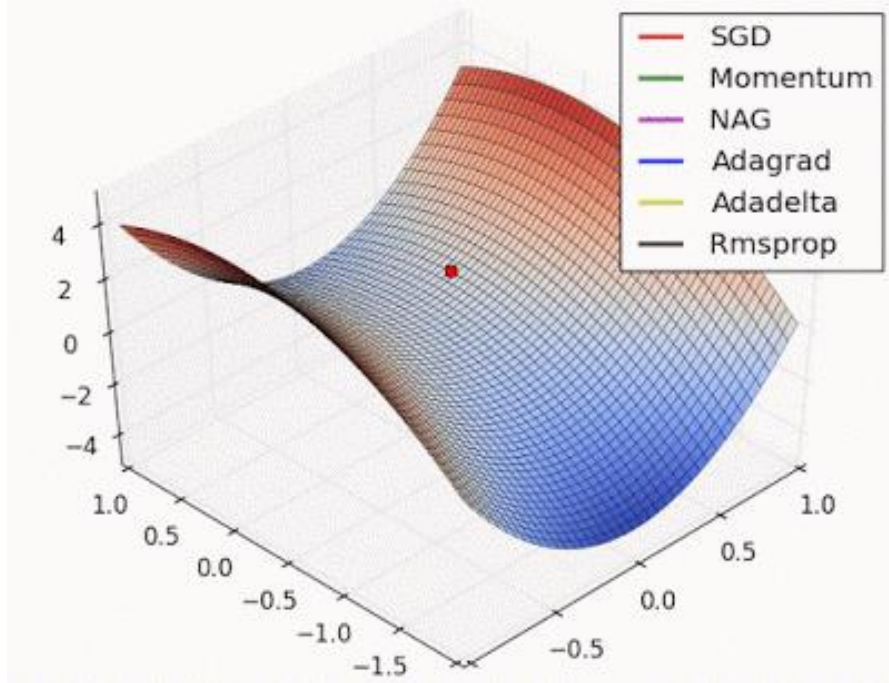
$$m_{t+1} = \gamma m_t + (1 - \gamma) |dE_t|$$

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

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

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

Illustrations



À voir sur :

www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Autre excellent survol

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

