Réseaux de neurones

IFT 603-712

Réseaux de neurones multicouches

Par

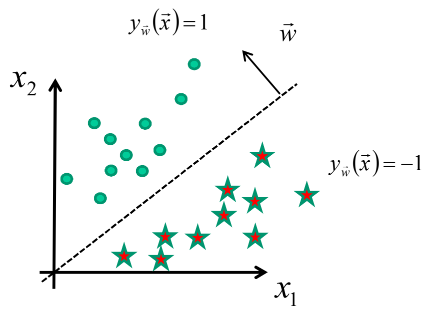
Pierre-Marc Jodoin

Rappel réseaux de neurones

(Perceptron, régression logistique, SVM)

Séparation linéaire

(2D et 2 classes)



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

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

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

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

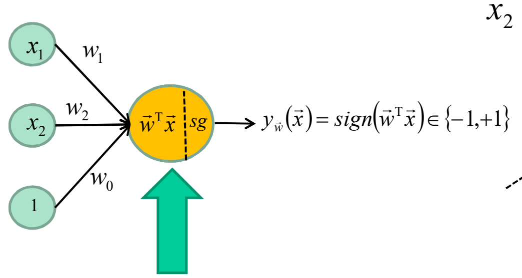
Par simplicité

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

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

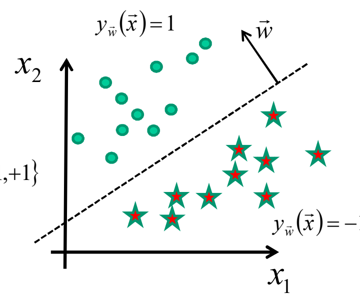
Perceptron

(2D et 2 classes)



Neurone

Produit scalaire + fonction d'activation



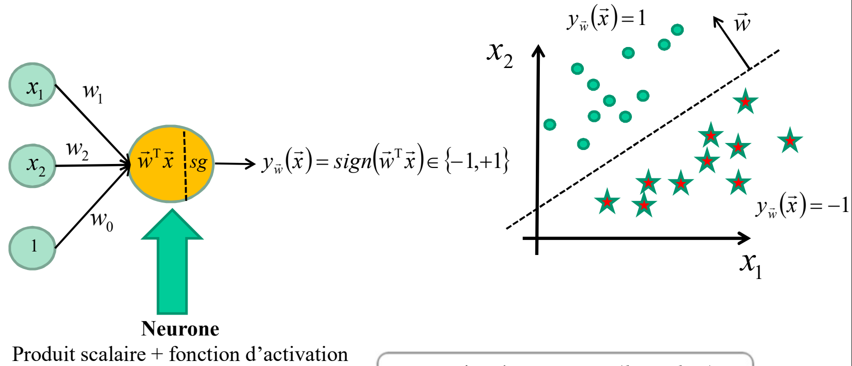
Fonction de coût Perceptron (loss) et gradient

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

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

Hinge Loss

(2D et 2 classes)



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

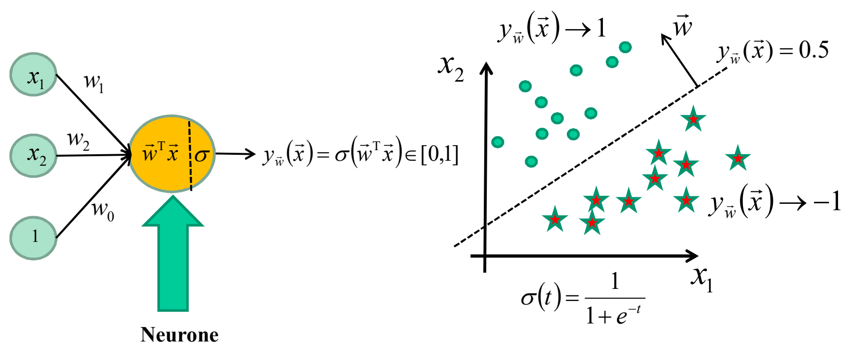
$$E_D(\vec{w}) = \sum_{n=1}^{N} \max\left(0, 1 - t_n \vec{w}^T \vec{x}\right)$$

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

Régression logistique

(2D, 2 classes)

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



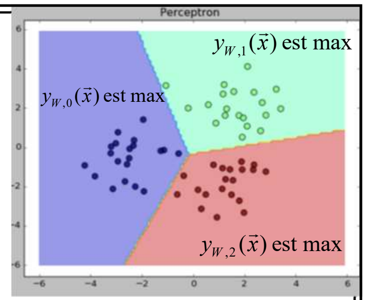
Fonction de coût (loss) et gradient

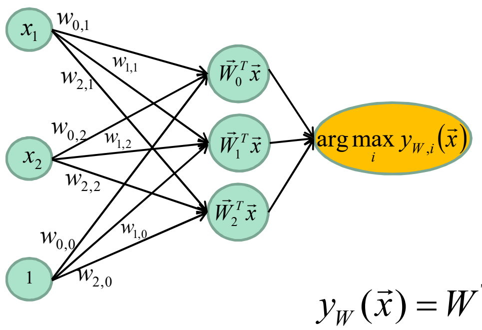
$$E_D(\vec{W}) = -\sum_{n=1}^{N} t_n \ln\left(y_W(\vec{x}_n)\right) + (1-t_n)\ln\left(1 - y_W(\vec{x}_n)\right)$$

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

Perceptron Multiclasse

(2D et 3 classes)





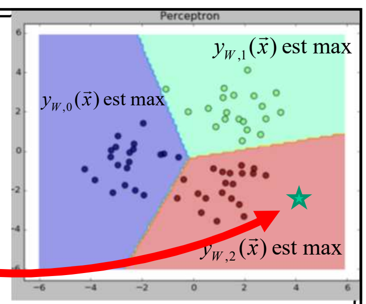
$$y_W(\vec{x}) = W^{\mathrm{T}}\vec{x}$$

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

Perceptron Multiclasse

Exemple

(1.1, -2.0)



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

Perceptron Multiclasse

Fonction de coût (***Perceptron loss***)

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

Somme sur l'ensemble des données mal classées

Score de la mauvaise classe

Score de la bonne classe

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

Hinge Multiclasse

Fonction de coût (***Hinge loss***)

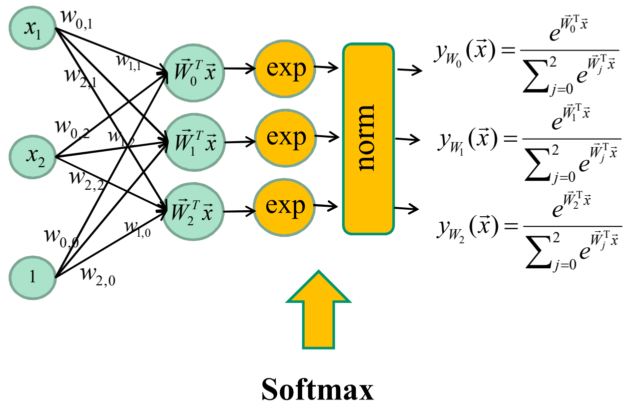
$$E_D(\mathrm{W}) = \sum_{n=1}^{N} \sum_{j} \max\left(0, 1 + \vec{W}_j^{\mathrm{T}} \vec{x}_n - \vec{W}_{t_n}^{\mathrm{T}} \vec{x}_n\right)$$

Score de la mauvaise classe

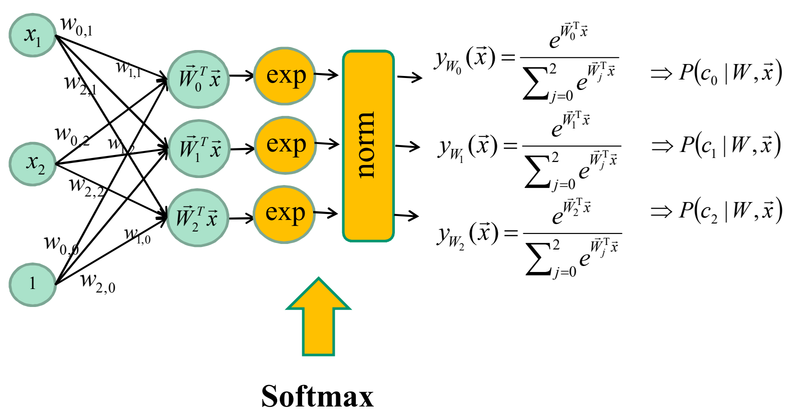
Score de la bonne classe

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

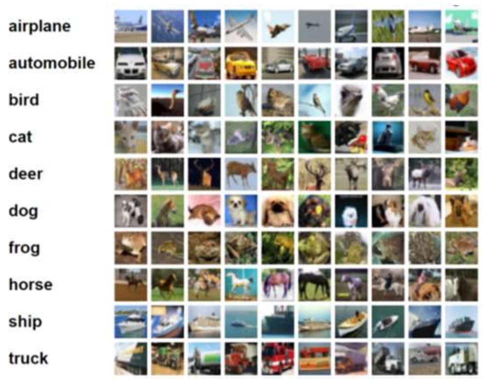
Régression logistique multiclasse



Régression logistique multiclasse



Régression logistique multiclasse



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

Étiquettes de classe **:** ***<u>one-hot vector</u>***

Régression logistique multiclasse

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

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

Devoir 4

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

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

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

Hinge loss

Score

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

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

Entropie croisée

Score

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

(Softmax)

Maximum *a posteriori*

Régularisation

Constante

$$\arg\min_{W} = E_D(W) \; + \lambda R(W)$$

Fonction de perte

Régularisation

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

Optimisation

Descente de gradient

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

Gradient de la function de coût

Taux d'apprentissage ou "learning rate".

Devoir 4

Descente de gradient stochastique

Initialiser **w**
k=0
FAIRE k=k+1
FOR n = 1 to N

$$\mathbf{w} = \mathbf{w} - \eta^{[k]} \nabla E(\vec{x}_n)$$

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

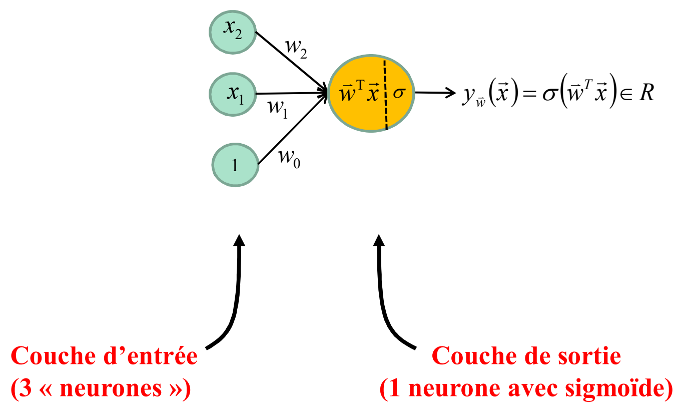
$$\mathbf{w} = \mathbf{w} - \eta^{[k]} \sum_i \nabla E(\vec{x}_i)$$

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

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

Maintenant, rendons le réseau **profond**

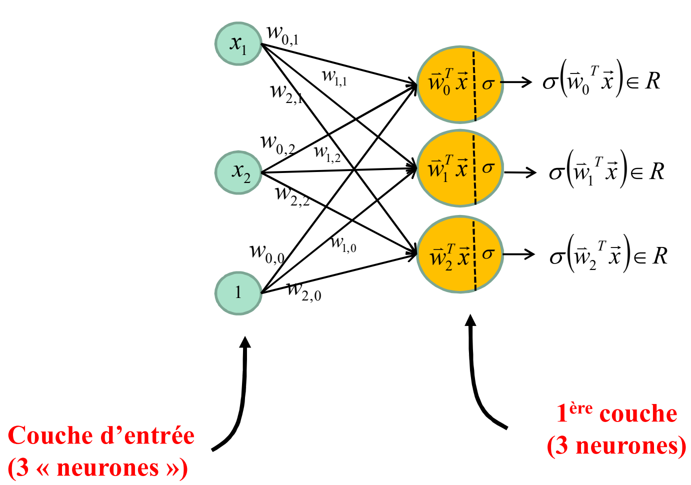
2D, 2Classes, Régression logistique linéaire



19

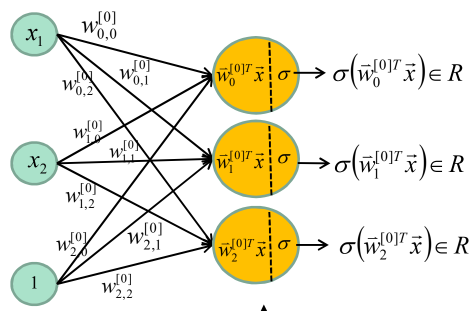
2D, 2Classes, Réseau à 1 couche cachée

Maintenant, ajoutons arbitrairement 3 neurons



2D, 2Classes, Réseau à 1 couche cachée

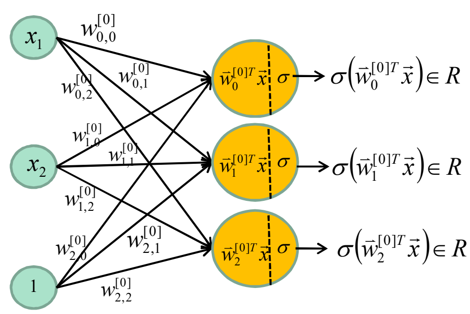
Puisque les poids sont **entre la couche d'entrée** et la **première couche** on va les identifier à l'aide de **l'indice [0]**



Couche d'entrée
(3 « neurones »)

1ère couche
(3 neurones)

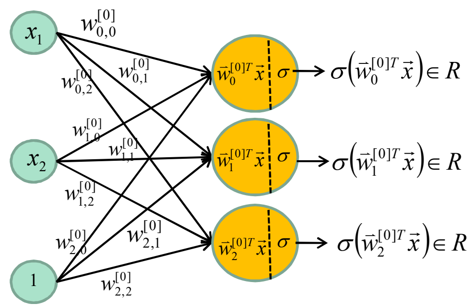
2D, 2Classes, Réseau à 1 couche cachée



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

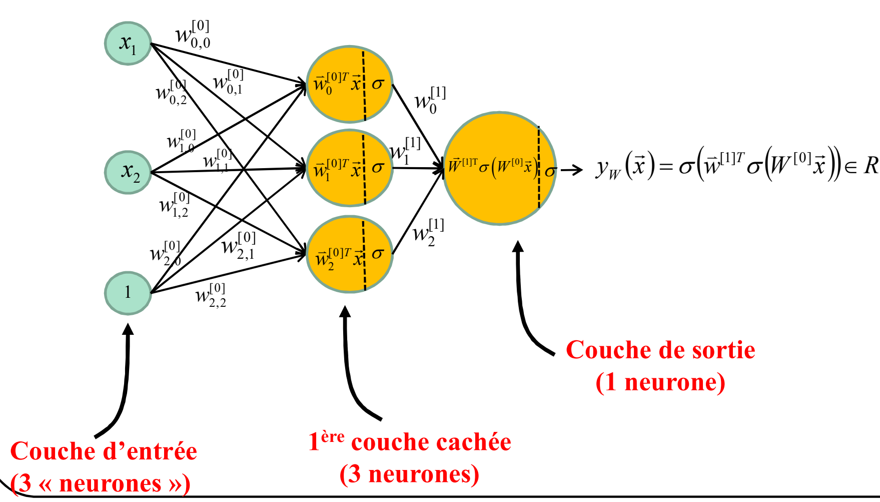


NOTE: représentation plus simple de la sortie de la 1ère couche (**3 réels**)

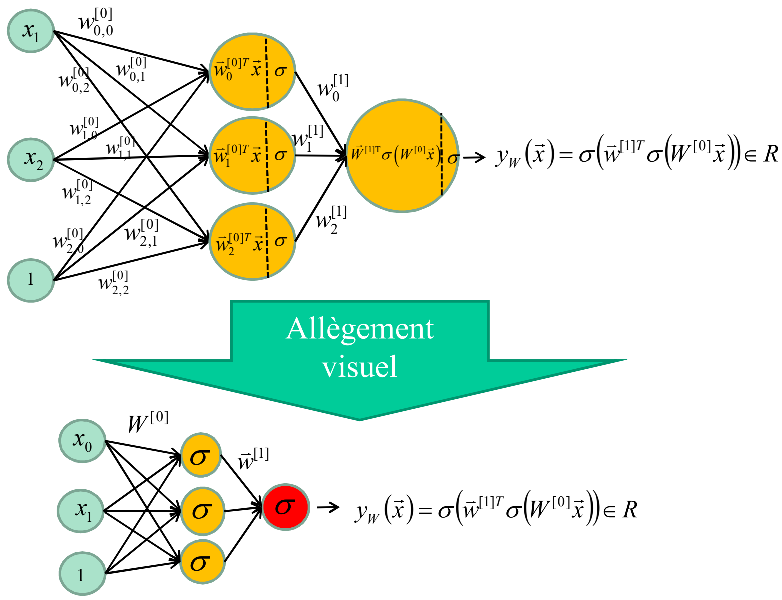
$$\sigma\left(W^{[0]}\vec{x}\right)$$

2D, 2Classes, Réseau à 1 couche cachée

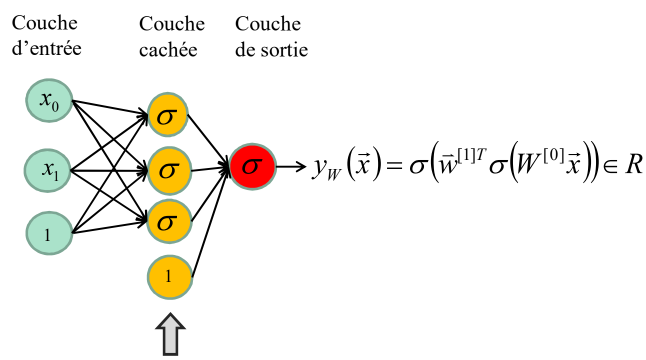
Si on veut effectuer une **classification 2 classes** via une **régression logistique** (donc **une fonction coût par « entropie croisée »**) on doit ajouter **un neurone de sortie.**



2D, 2Classes, Réseau à 1 couche cachée



2D, 2Classes, Réseau à 1 couche cachée

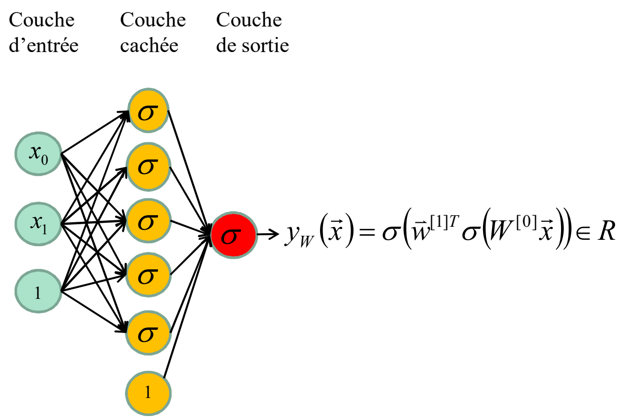


Très souvent, on ajoute un **<u>neurone de biais</u>** à la couche cachée.

Ce réseau possède au total **13 paramètres**

$\underbrace{3\text{x}3}_{\text{Couche entrée / Couche cachée}}$ $\underbrace{1\text{x}4}_{\text{Couche cachée / Couche de sortie}}$

2D, 2Classes, Réseau à 1 couche cachée



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

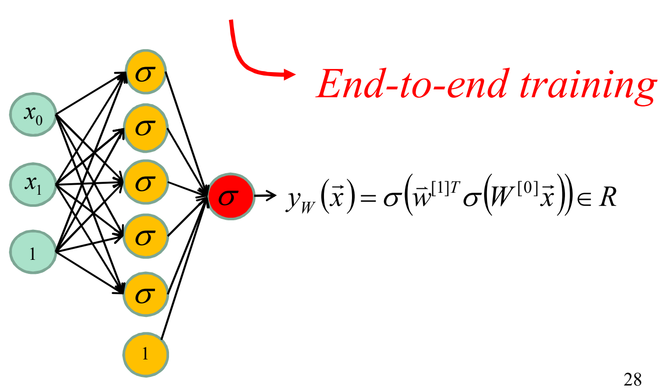
Ce réseau a 5x3+1x6=**21 paramètres**

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 $\sigma\left(W^{[0]}\vec{x}\right) \in R^5$) là où les **classes sont linéairement séparables**.

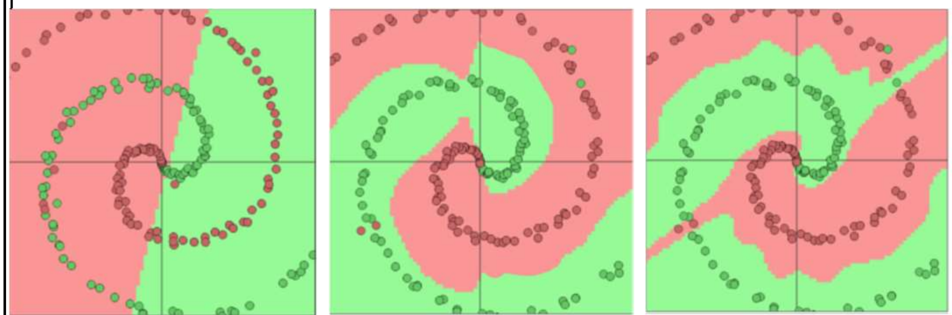
Car il ne faut pas oublier que la **couche de sortie** est une **régression logistique linéaire**.

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



Nombre de neurons VS Capacity

Aucun neurone caché | 12 neurones cachés | 60 neurones cachés

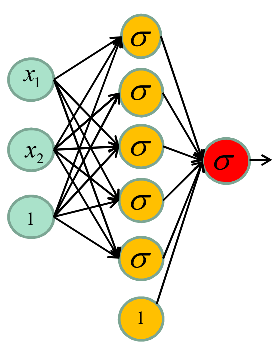


Cassification linéaire	Classification non linéaire	Classification non linéaire
Underfitting	**BON RÉSULTAT**	**Over fitting**
(pas assez de capacité)	(bonne capacité)	(trop grande capacité)

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

2D, 2Classes, Réseau à 1 couche cachée

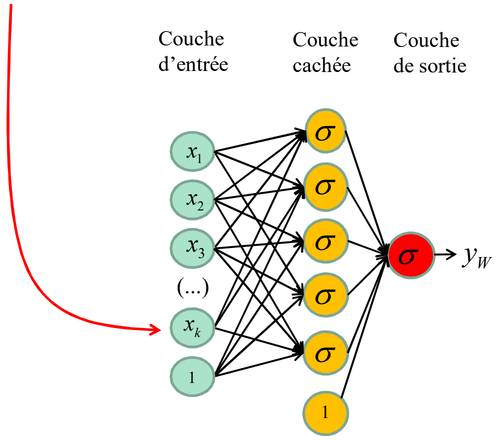
Couche d'entrée | Couche cachée | Couche de sortie



$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[1]T}\sigma\left(W^{[0]}\vec{x}\right)\right) \in R$$

30

kD, 2Classes, Réseau à 1 couche cachée

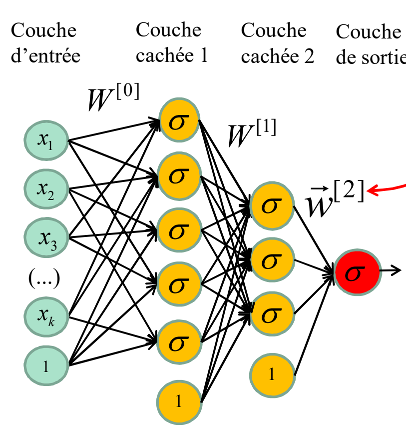


$y_W(\vec{x}) = \sigma\left(\vec{w}^{[1]T}\sigma\left(W^{[0]}\vec{x}\right)\right) \in R$

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 5x(k+1)+1x6 **paramètres**

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



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

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

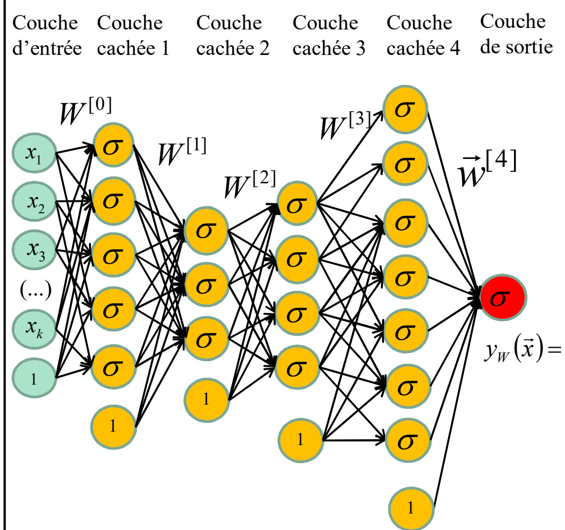
$\vec{w}^{[2]} \in R^{4}$

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

En ajoutant une couche cachée, on ajoute une multiplication matricielle

Ce réseau a 5x(k+1)+6x3 + 1x4 **paramètres**

kD, 2 Classes, Réseau à 4 couches cachées



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

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

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

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

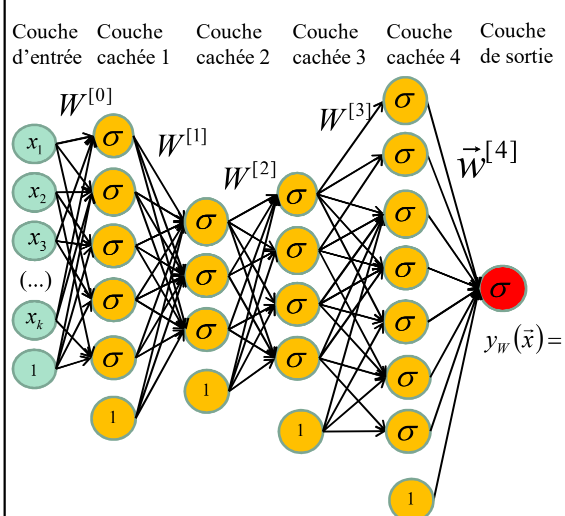
$\vec{w}^{[4]} \in R^{8}$

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

Ce réseau a 5x(k+1)+6x3 + 4x4+7x5+1x8 **paramètres**

33

kD, 2 Classes, Réseau à 4 couches cachées



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

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

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

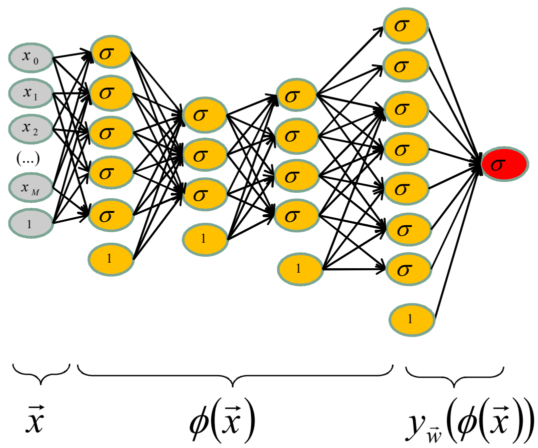
$W^{[3]} \in R^{5\times 7}$

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

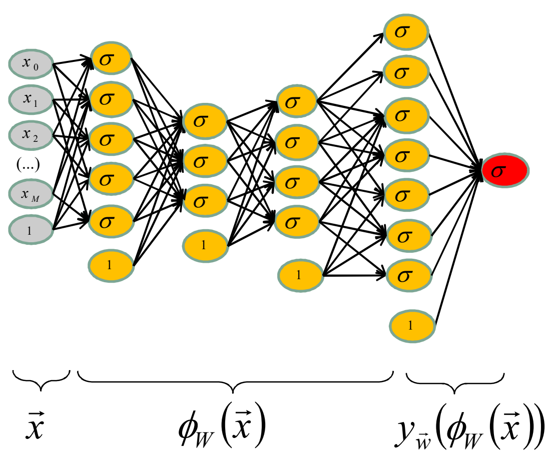
$$y_W(\vec{x}) = \sigma\left(\vec{w}^{[4]T}\sigma\left(W^{[3]}\sigma\left(W^{[2]}\sigma\left(W^{[1]}\sigma\left(W^{[0]}\vec{x}\right)\right)\right)\right)\right)$$

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

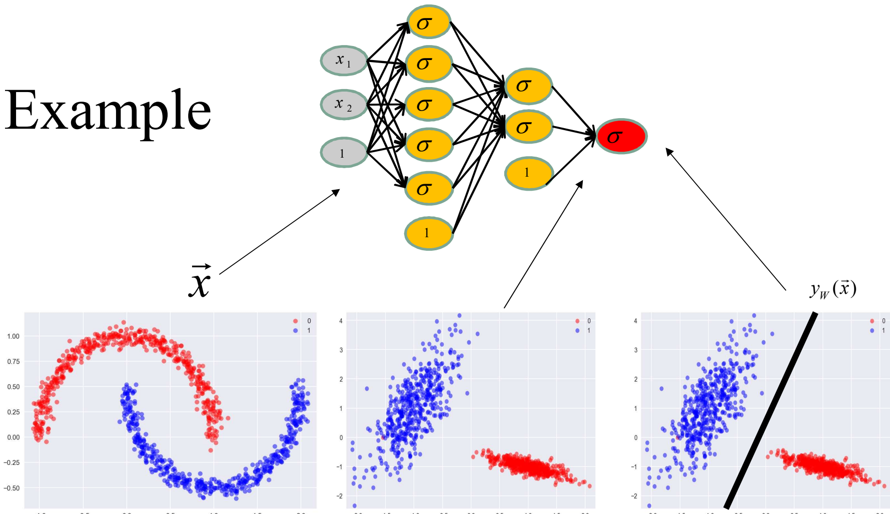
Réseau de neurones multicouches = apprendre la function de base



Réseau de neurones multicouches = apprendre la function 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

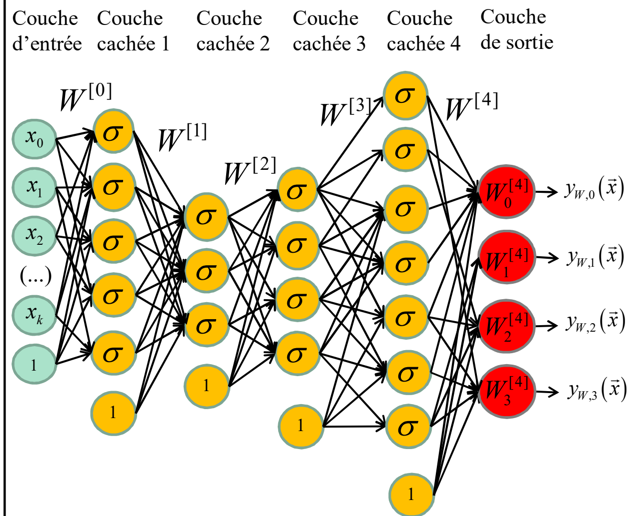
Augmenter la capacité du réseau

image by www.slon.pics - Freepik.com

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



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

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

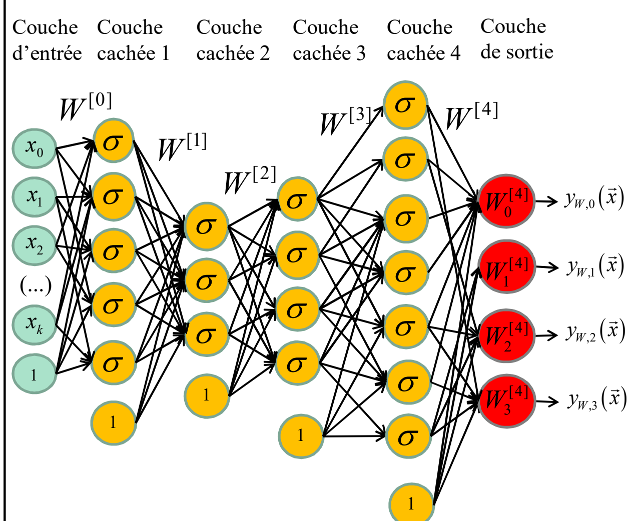
$W^{[2]} \in R^{4\times 4}$

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

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

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

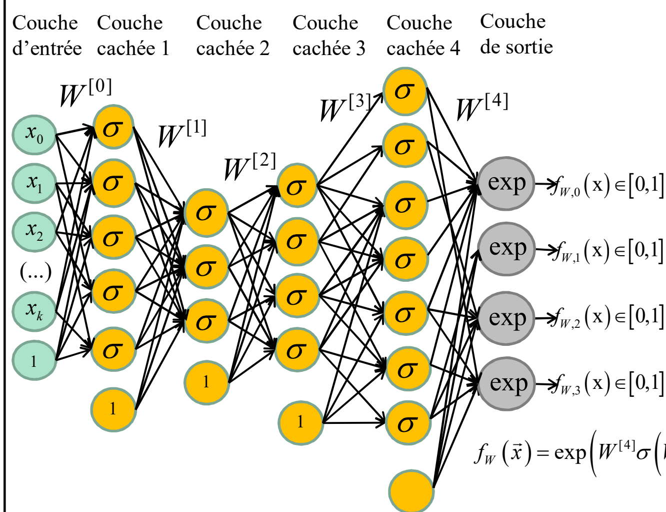
kD, **4 Classes**, Réseau à 4 couches cachées



Hinge loss

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

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



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

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

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

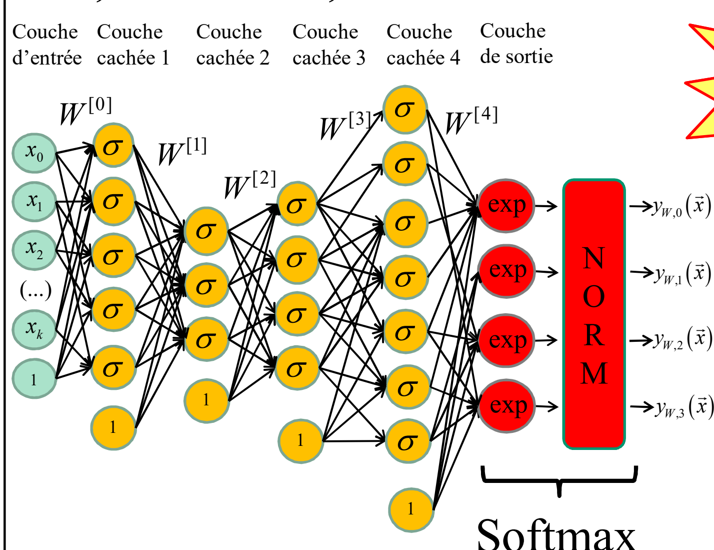
$W^{[3]} \in R^{5\times 7}$

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

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

Softmax $y_{W,i}(\mathrm{x}) = \dfrac{f_{W,i}}{\sum_k f_{W,k}}$

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



Entropie Croisée

Softmax

$y_{W,i}(\mathrm{x}) = \dfrac{f_{W,i}}{\sum_k f_{W,k}}$

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

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

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

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

y_pred = np.dot(W3,H2)
```

} Couche 1

} Couche 2

} Couche 3

} Couche sortie

Forward pass

Comment optimiser les paramètres?

0- Partant de

$$W = \arg\min_{W} E_D(W) + \lambda R(W)$$

Trouver une function de régularisation. En général

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

47

Comment optimiser les paramètres?

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

Hinge loss

Entropie croisée (*cross entropy*)

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

cross entropy => Softmax

Comment optimiser les paramètres?

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

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

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

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

49

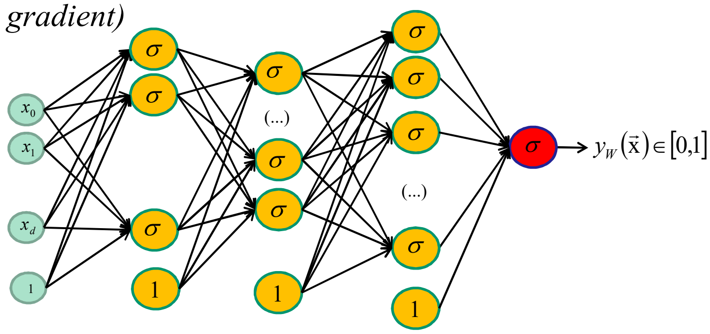
Comment optimiser les paramètres?

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

50

Disparition du gradient

(vanishing gradient)



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

disparition du gradient

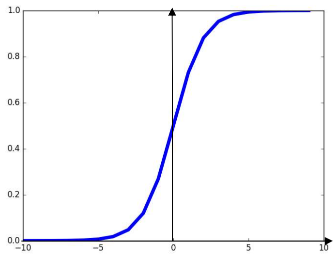
Démonstration au tableau

51

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

Fonction d'activation

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



Sigmoïde

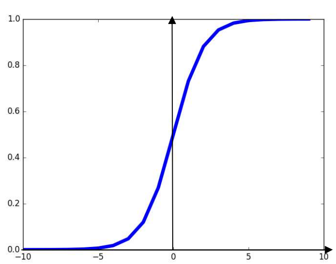
- Ramène les valeurs entre 0 et 1
- Historiquement populaire

3 Problèmes :

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

Fonction d'activation

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



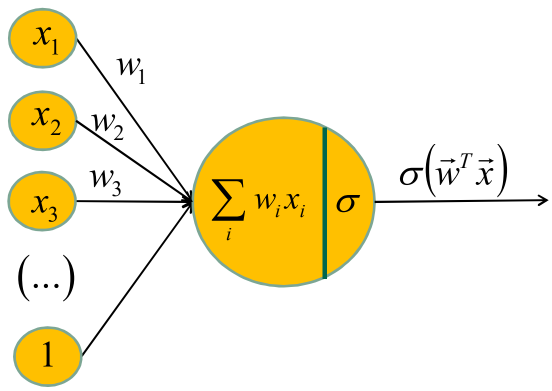
Sigmoïde

- 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.**

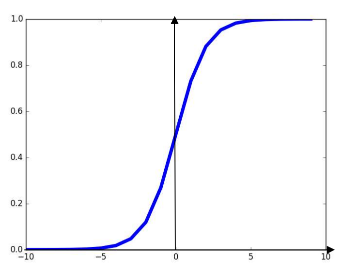
Qu'arrive-t-il lorsque le vecteur d'entrée $\vec{x}$ d'un neurone est toujours positif?



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

Fonction d'activation

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



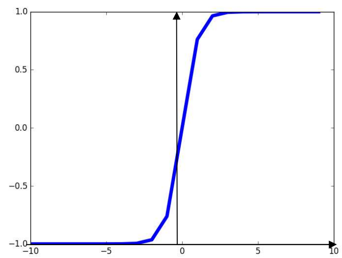
Sigmoïde

- 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



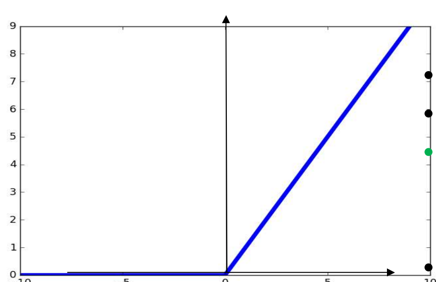
Tanh(x)

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

[LeCun et al., 1991]

Fonction d'activation

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



ReLU(x)
(Rectified Linear Unit)

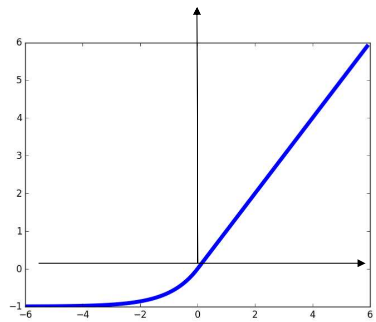
- Aucune **saturation** ☺
- Super **rapide** ☺
- **Converge plus rapide** que sigmoïde/tanh (5 à 10x) ☺

- Sortie **non centrée à zéro** ☹
- **Un inconvénient :** qu'arrive-t-il au gradient lorsque x<0? ☹

[Krizhevsky et al., 2012]

Fonction d'activation

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



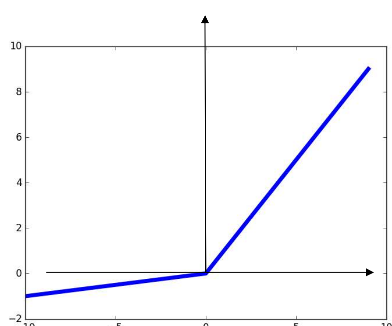
ELU(x)
(Exponential Linear Unit)

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

[Clevert et al., 2015]

Fonction d'activation

$$\mathrm{LReLU}(x) = \max\left(0.01x, x\right)$$



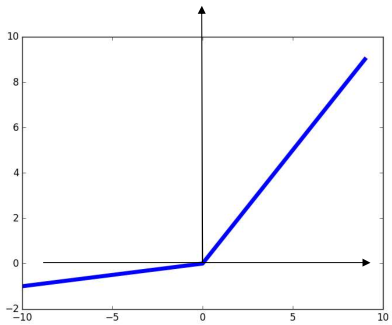
Leaky ReLU(x)

- 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 function de coût

Taux d'apprentissage ou "learning rate".

Descente de gradient stochastique

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

Optimisation par Batch

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

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

Optimisation

Descente de gradient

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

∇E → Gradient de la function de coût

$\eta^{[k]}$ → 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*/***

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

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

Itération

Optimisation

Descente de gradient

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

∇E → Gradient de la function de coût

$\eta^{[k]}$ → Taux d'apprentissage ou "learning rate".

Optimisation par **mini-batch**

Initialiser **w**
k=0
FAIRE k=k+1

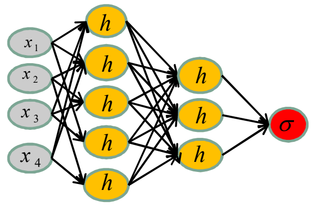
FAIRE n=0 à N par sauts de ***MBS /*Mini-batch size*/***

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

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

Epoch

$\vec{x}$

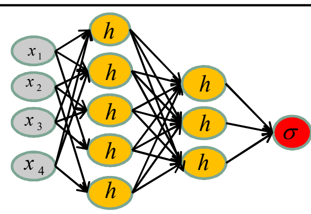


$\sigma\left(\vec{W}_3 h\left(W_2 h\left(W_1 \vec{x}\right)\right)\right)$

Propagation avant pour une donnée (7 étapes)

$\vec{x}$	$\in IR^4$
$W_1\vec{x}$	$\in IR^5$
$h(W_1\vec{x})$	$\in IR^{5+1}$
$W_2 h(W_1\vec{x})$	$\in IR^3$
$h(W_2 h(W_1\vec{x}))$	$\in IR^{3+1}$
$\vec{W}_3\left(h(W_2 h(W_1\vec{x}))\right)$	$\in IR$
$\sigma\left(\vec{W}_3\left(h(\vec{W}_2 h(W_1\vec{x}))\right)\right)$	$\in IR$

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



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

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

POUR i allant de 1 à 3

$\vec{x}_i = X[i]$	$\in IR^{M+1}$
$W_1\vec{x}$	$\in IR^5$
$h(W_1\vec{x})$	$\in IR^{5+1}$
$W_2 h(W_1\vec{x})$	$\in IR^3$
$h(W_2 h(W_1\vec{x}))$	$\in IR^{3+1}$
$\vec{W}_3\left(h(W_2 h(W_1\vec{x}))\right)$	$\in IR$
$Y[i] = \sigma\left(\vec{W}_3\left(h(\vec{W}_2 h(W_1\vec{x}))\right)\right)$	$\in IR$

Solution naïve et peu efficace

TP1

Solution

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

$$\begin{pmatrix} w_1 & w_2 & w_3 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \boxed{\left(w_1 a + w_2 b + w_3 c\right)}$$

$$\begin{pmatrix} w_1 & w_2 & w_3 \end{pmatrix}\begin{pmatrix} d \\ e \\ f \end{pmatrix} = \boxed{\left(w_1 d + w_2 e + w_3 f\right)}$$

$$\begin{pmatrix} w_1 & w_2 & w_3 \end{pmatrix}\begin{pmatrix} g \\ h \\ i \end{pmatrix} = \boxed{\left(w_1 g + w_2 h + w_3 i\right)}$$

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

TROIS produits scalaires

Solution

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

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

UNE multiplication matricielle

Solution

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

$$W_1\vec{x}_1 = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix}$$

$$W_1\vec{x}_2 = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} d \\ e \\ f \\ g \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$$

$$W_1\vec{x}_3 = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} h \\ i \\ j \\ k \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix}$$

$$\begin{bmatrix} u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \\ u_1 & v_1 & z_1 \end{bmatrix}$$

TROIS multi matrice-vecteur

Solution

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

$$W_1X = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{pmatrix} \begin{bmatrix} a & d & h \\ b & e & i \\ c & f & j \\ d & g & k \end{bmatrix} = \begin{bmatrix} u_1 & v_1 & z_1 \\ u_2 & v_2 & z_2 \\ u_3 & v_3 & z_3 \\ u_4 & v_4 & z_4 \\ u_5 & v_5 & z_5 \end{bmatrix}$$

UNE multiplication matricielle

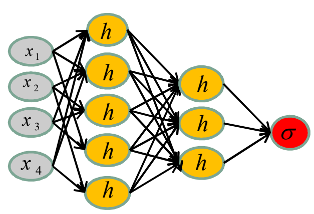
Vectorisation de la propagation avant

En résumé, lorsqu'on propage une « batch »

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

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

$$X = \begin{bmatrix} | & | & | \\ \vec{x}_1 & \vec{x}_2 & \vec{x}_3 \\ | & | & | \end{bmatrix}$$



$$Y = \begin{bmatrix} \sigma\left(h\left(\vec{W}_2 h\left(W_1 \vec{x}_1\right)\right)\right) \\ \sigma\left(h\left(\vec{W}_2 h\left(W_1 \vec{x}_2\right)\right)\right) \\ \sigma\left(h\left(\vec{W}_2 h\left(W_1 \vec{x}_3\right)\right)\right) \end{bmatrix}^T$$

Vectoriser la rétropropagation

Vectoriser la rétropropagation

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

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

$$\vec{W} \qquad\qquad X \qquad\qquad Y$$

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

Vectoriser la rétropropagation

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

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

$$W \qquad\qquad X \qquad\qquad Y$$

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

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

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

$$W \qquad X \qquad Y$$

Concentrons-nous sur w_1

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

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

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

(provient de la rétro-propagation)

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

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

$$W \qquad X \qquad Y$$

Et pour tous les poids

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

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

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

Matrice jacobienne

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

$$W \qquad X \qquad Y$$

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

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

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

Vectorisation de la rétro-propagation

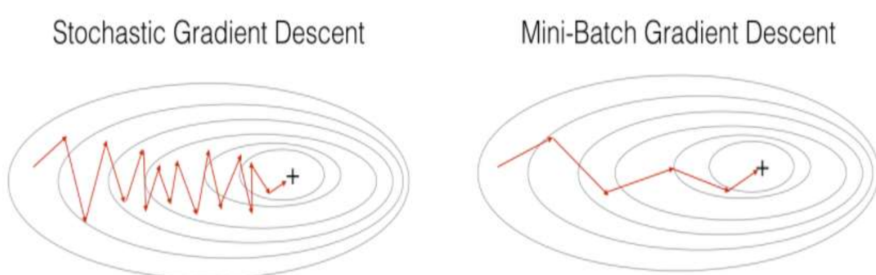
En résumé, lorsqu'on rétro-propage un gradient d'une batch

Au niveau **neuronal**	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/



Comment initialiser un réseau de neurones?

$$W = ?$$

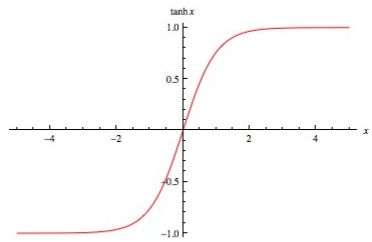
Initialisation

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.

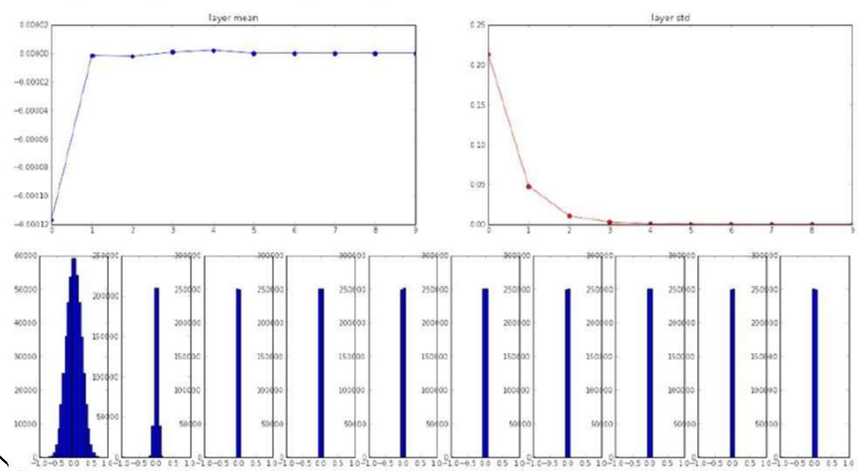
E.g. réseau à 10 couches avec 500 neurones par couche et des **tanh** comme fonctions d'activation.



Histogrammes des valeurs de sortie des 10 couches.

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

L'activation des couches devient zéro!

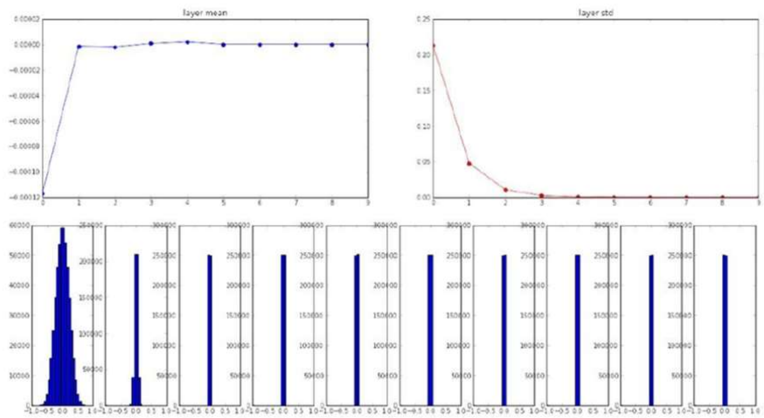


Crédit http://cs231n.stanford.edu/

Histogrammes des valeurs de sortie des 10 couches.

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

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

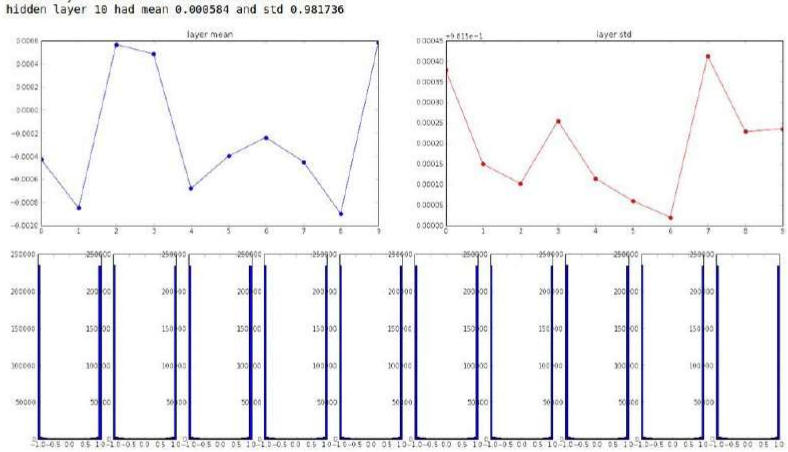


Crédit http://cs231n.stanford.edu/

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

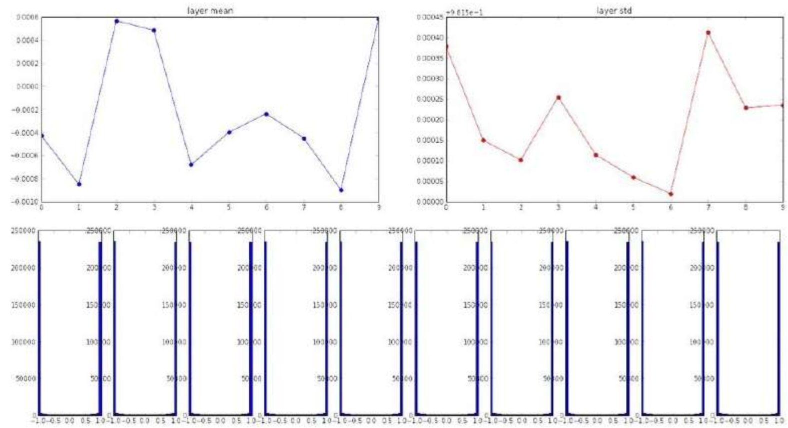


Crédit http://cs231n.stanford.edu/

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

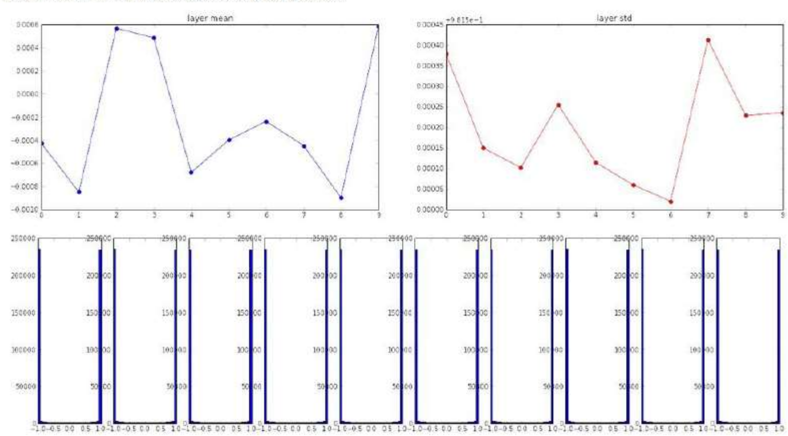


Crédit http://cs231n.stanford.edu/

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
```

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



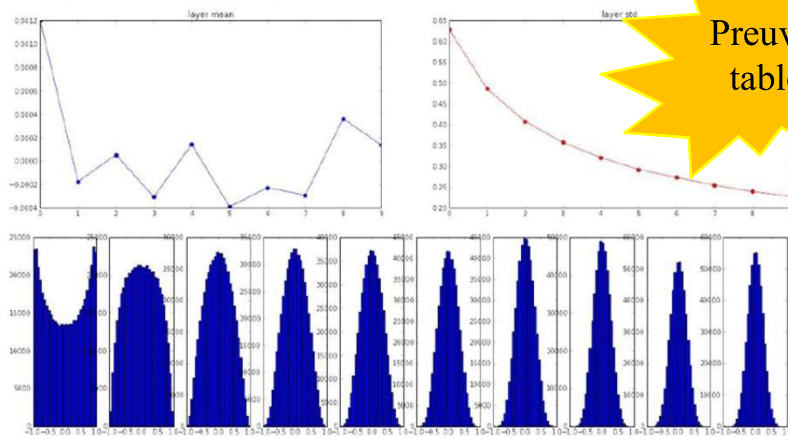
Crédit http://cs231n.stanford.edu/

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

```
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 -0.000175 and std 0.486051
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
```

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

Preuve au tableau

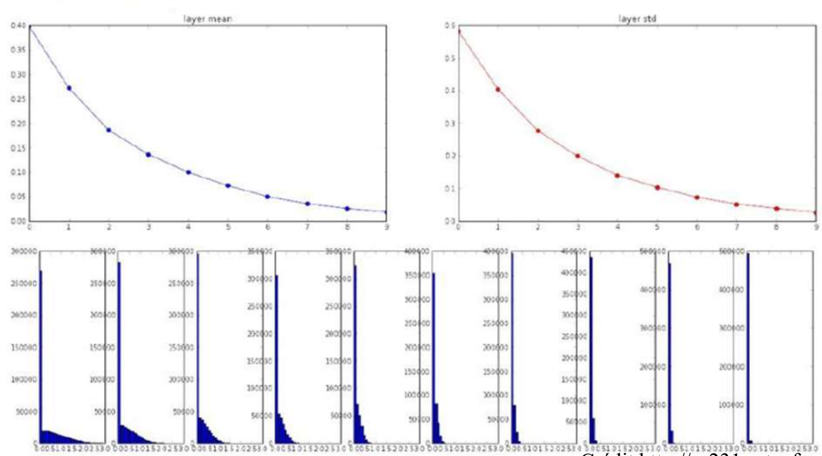


Crédit http://cs231n.stanford.edu/

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

```
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 0.272352 and std 0.403795
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
```

Par contre en utilisant ReLU
on a encore un problème

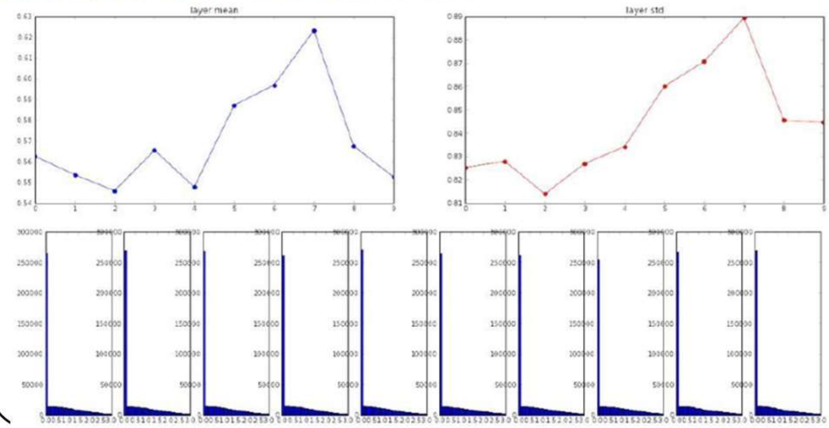


Crédit http://cs231n.stanford.edu/

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

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.84452
```

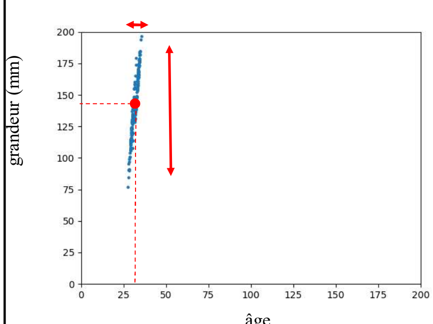
He et al., 2015
(note additional /2)

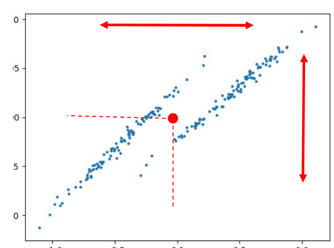


Crédit http://cs231n.stanford.edu/

Prétraitement des données

Centrer et **normaliser** les données d'entrée





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

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***.

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

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

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

Sanity checks

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

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

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

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

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

Crédit http://cs231n.stanford.edu

Sanity checks

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

```python
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
```

```python
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

Crédit http://cs231n.stanford.edu/

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

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

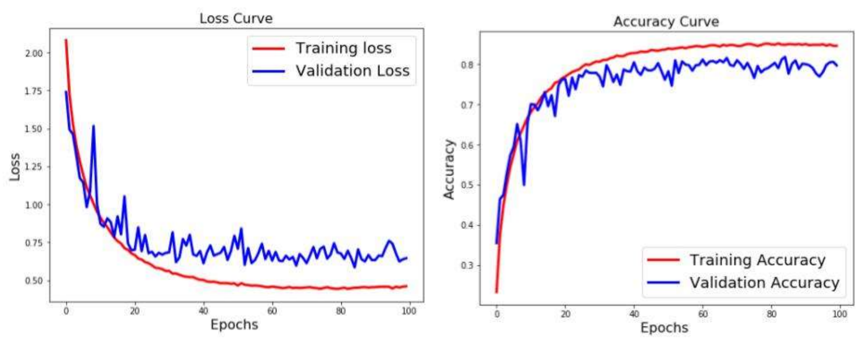
```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Very small loss, train accuracy 1.00, nice!

Crédit http://cs231n.stanford.edu

Sanity checks

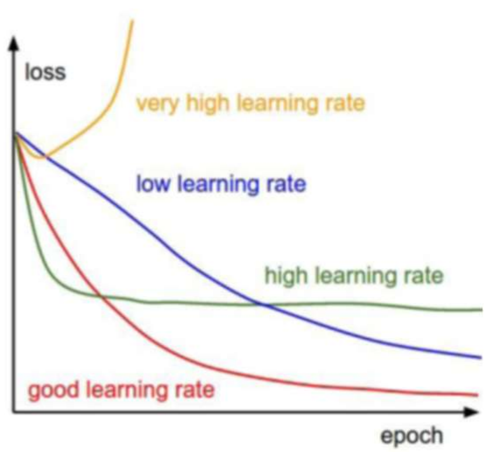
3. Toujours visualiser les courbes d'apprentissage et de validation



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

Sanity checks

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



Crédit http://cs231n.stanford.edu

Sanity checks

3. Toujours vérifier la validité d'un gradient

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

Rappel

Approximation numérique de la dérivée

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

Sanity checks

3. Toujours vérifier la validité d'un gradient

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

Rappel

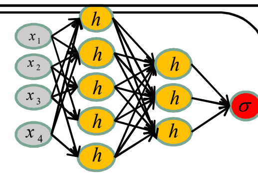
Approximation numérique du gradient

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

En calculant

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

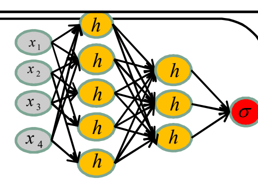
Vérification du gradient
(exemple)



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 **E(W+h)=1.25322**

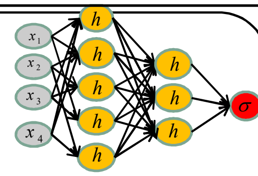
Vérification du gradient
(exemple)



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=(1.25353-125347)/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**

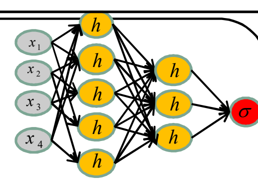
Vérification du gradient
(exemple)



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=(1.25347-125347)/0.0001**
...	...	
$W_{20} = -3.1$	$W_{20} = -3.1$	
$W_{21} = -1.5,$	$W_{21} = -1.5,$	
$W_{22} = 0.33$	$W_{22} = 0.33$	

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

Vérification du gradient
(exemple)

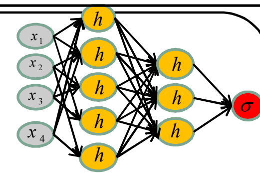


W	W+h	gradient W
$W_{00} = 0.34$	$W_{00} = 0.34$	**-2.5**
$W_{01} = -1.11$	$W_{01} = -1.11$	**0.6**
$W_{02} = 0.78$	$W_{02} = 0.78$	**0.0**
...	...	**…**
$W_{20} = -3.1$	$W_{20} = -3.1$	**1.1**
$W_{21} = -1.5,$	$W_{21} = -1.5,$	**1.3**
$W_{22} = 0.33$	$W_{22} = 0.33$	**-2.1**

E(W)=1.25347

Vérification du gradient

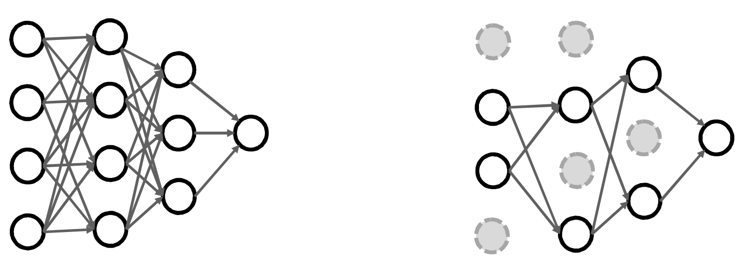
(exemple)



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

Autre bonne pratique : *Dropout*

Forcer à zéro certains neurones de façon aléatoire à chaque itération



Autre bonne pratique : *Dropout*

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

Crédit http://cs231n.stanford.edu/

Autre bonne pratique : *Dropout*

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

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

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

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

Crédit http://cs231n.stanford.edu/

Autre bonne pratique : *Dropout*

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

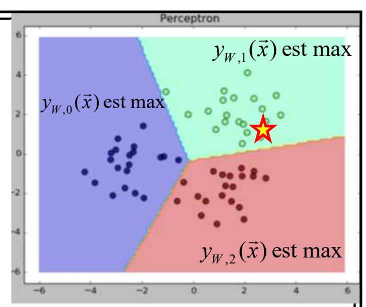
car *dropout* **ajoute du bruit** à la prédiction

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

masque aléatoire

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

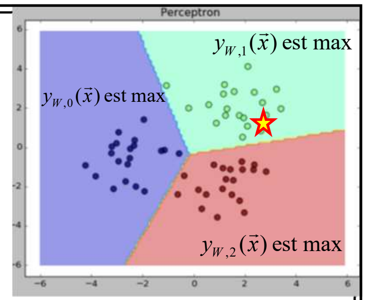
Exemple simple : $\vec{x} = \begin{pmatrix} 2.2 \\ 1.3 \end{pmatrix}, t = 1$



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]
```

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]
```

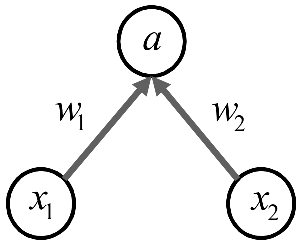
Exécuter le modèle un grand nombre de fois et **prendre la moyenne** revient à calculer **l'espérance mathématique**

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

Bonne nouvelle, on peut faire plus simple en approximant cette l'expérance 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**



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

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

```python
""" 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**

Crédit http://cs231n.stanford.edu/

Descente de gradient

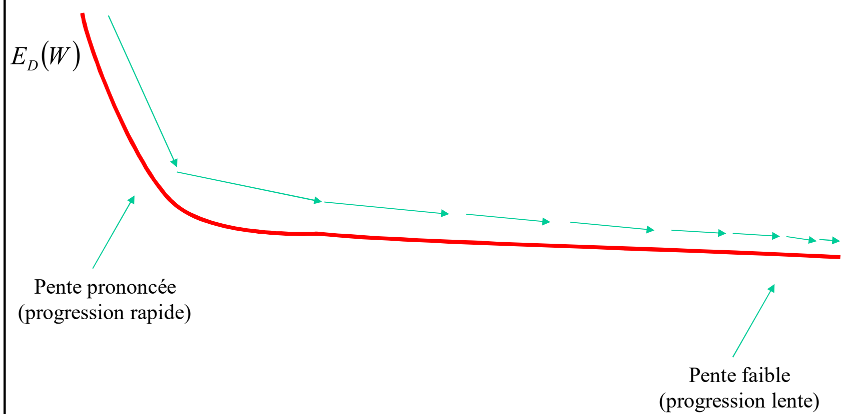
version améliorée

Descente de gradient

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

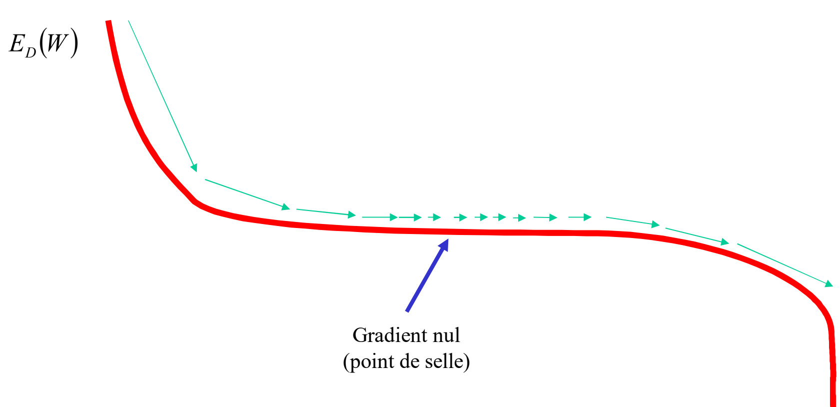
Descente de gradient : **problème**

Progrès quasi nul lorsque la pente est très faible



Descente de gradient : **problème**

Les points de selles sont fréquents en haute dimension



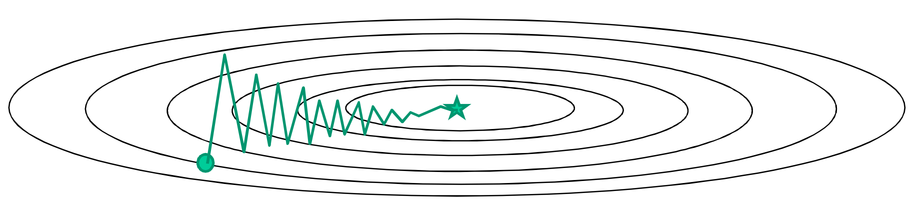
Descente de gradient : **problème**

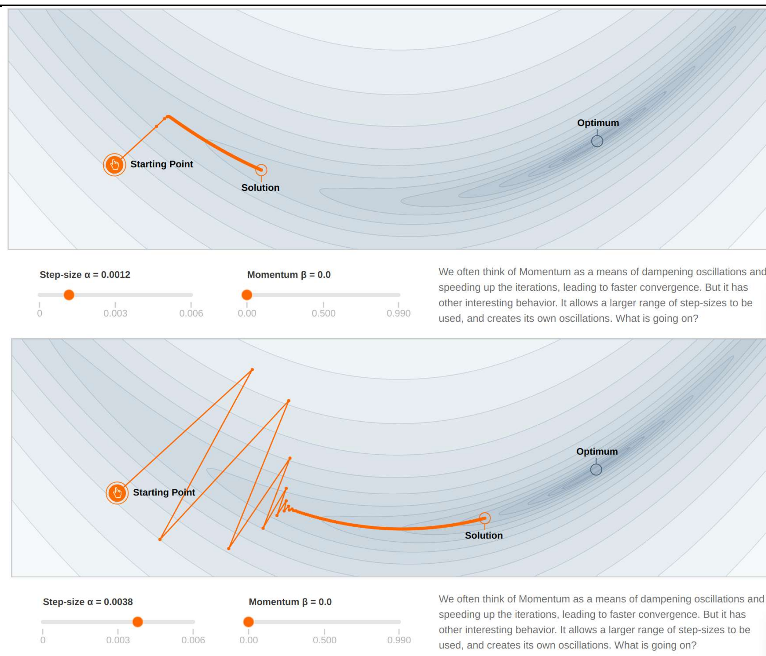
Qu'arrive-t-il si la fonction de coût (loss) a une pente prononcée dans une direction et moins prononcée dans une autre direction?

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.





https://distill.pub/2017/momentum/

Descente de gradient + **Momentum**

Descente de gradient stochastique	Descente de gradient stochastique + **Momentum**
$\mathrm{w}_{t+1} = \mathrm{w}_t - \eta \nabla E_{\vec{x}_n}(\mathrm{w}_t)$	$v_{t+1} = \rho v_t + \nabla E_{\vec{x}_n}(\mathrm{w}_t)$ $\mathrm{w}_{t+1} = \mathrm{w}_t - \eta v_{t+1}$

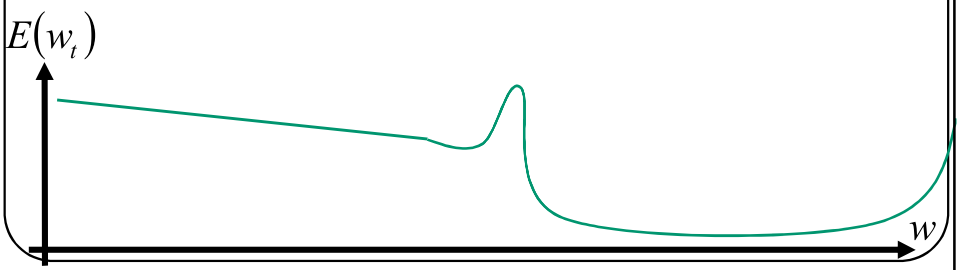
Provient de l'équation de la vitesse
(à démontrer en devoir ou en exercice)

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

	v_t	$\nabla E_x(w_t)$
$t=1$		
$t=2$		
$t=3$		
$t=4$		
$t=5$		
$t=6$		
$t=7$		
$t=8$		

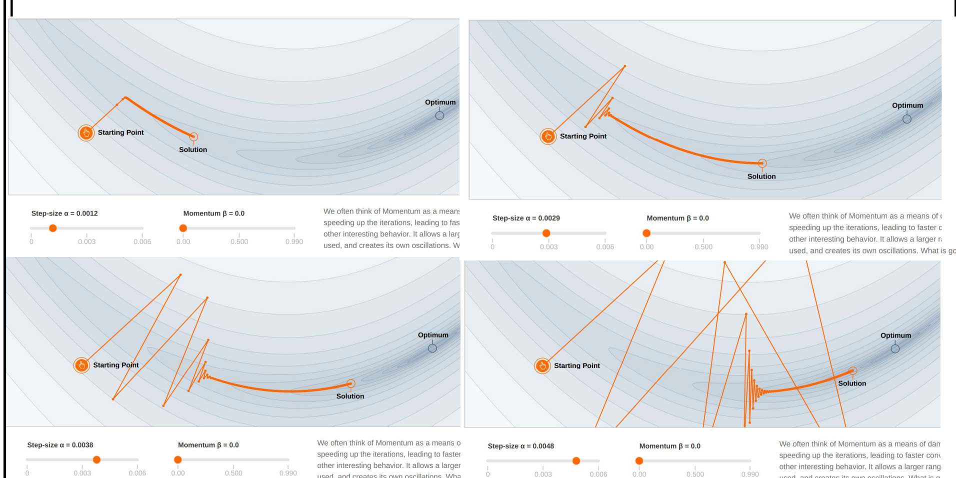
$$v_{t+1} = \rho v_t + \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$

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



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

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



https://distill.pub/2017/momentum/

$$v_{t+1} = \rho v_t + \nabla E_{\vec{x}_n}\left(\mathrm{w}_t\right)$$

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

AdaGrad (décroissance automatique de η)

Descente de gradient stochastique

$$\mathrm{w}_{t+1} = \mathrm{w}_t - \eta \nabla E_{\vec{x}_n}\left(\mathrm{w}_t\right)$$

AdaGrad

$$dE_t = \nabla E_{\vec{x}_n}\left(\mathrm{w}_t\right)$$

$$m_{t+1} = m_t + \left|dE_t\right|$$

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

AdaGrad (décroissance automatique de η)

Descente de gradient stochastique

$$\mathrm{w}_{t+1} = \mathrm{w}_t - \eta \nabla E_{\vec{x}_n}\left(\mathrm{w}_t\right)$$

AdaGrad

$$dE_t = \nabla E_{\vec{x}_n}\left(\mathrm{w}_t\right)$$

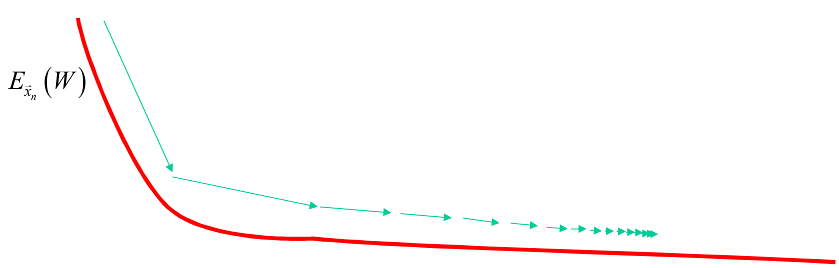
$$m_{t+1} = m_t + \left|dE_t\right|$$

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

η décroit 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)

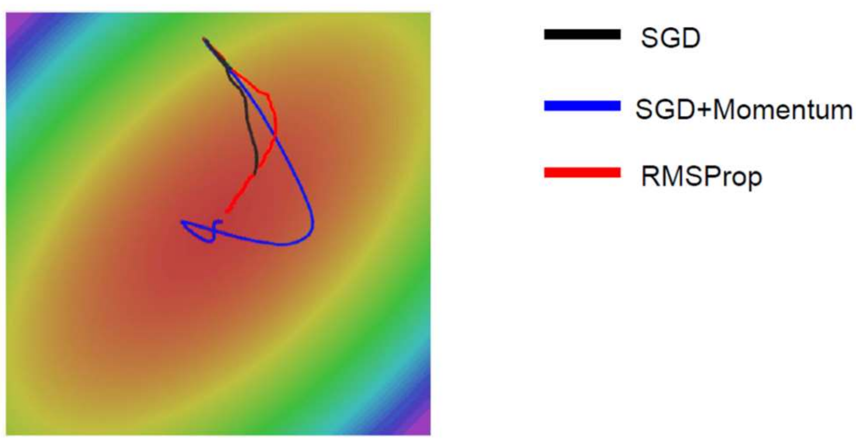
AdaGrad

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

RMSProp

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

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



Adam (Combo entre Momentum et RMSProp)

Momentum

$$v_{t+1} = \rho v_t + \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \eta v_{t+1}$$

Adam

$$dE_t = \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$
$$v_{t+1} = \alpha v_t + (1-\alpha) dE_t$$
$$m_{t+1} = \gamma m_t + (1-\gamma)|dE_t|$$
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

Adam (Combo entre Momentum et RMSProp)

Momentum

$$v_{t+1} = \rho v_t + \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \eta v_{t+1}$$

Adam

$$dE_t = \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$

Momentum

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

$$m_{t+1} = \gamma m_t + (1-\gamma)|dE_t|$$
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \frac{\eta}{m_{t+1} + \varepsilon} v_{t+1}$$

Adam (Combo entre Momentum et RMSProp)

RMSProp

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

Adam

$$dE_t = \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$
$$v_{t+1} = \alpha v_t + (1-\alpha) dE_t$$
$$\boxed{m_{t+1} = \gamma m_t + (1-\gamma)|dE_t|}$$
RMSProp
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \frac{\eta}{m_{t+1}+\varepsilon} v_{t+1}$$

Adam (Version complète)

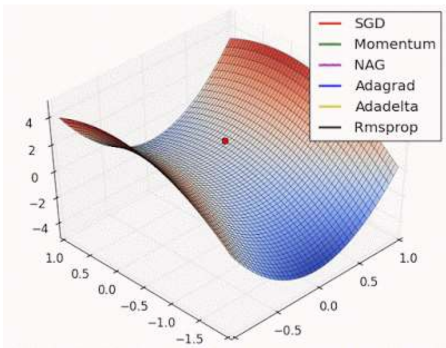
$$v_{t=0} = 0$$
$$m_{t=0} = 0$$

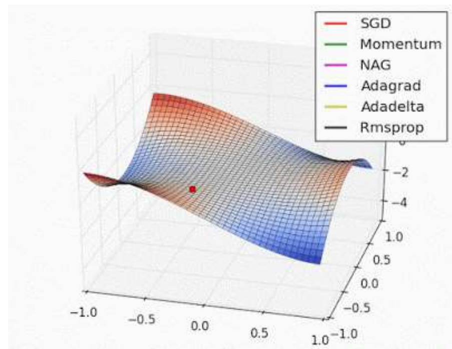
```
for t=1 à num_iterations
     for n=0 à N
```

$$dE_t = \nabla E_{\vec{x}_n}(\mathrm{w}_t)$$
$$v_{t+1} = \alpha v_t + (1-\alpha) dE_t$$
$$m_{t+1} = \gamma m_t + (1-\gamma)|dE_t|$$
$$v_{t+1} = \frac{v_{t+1}}{1-\beta_1^{\,t}}, m_{t+1} = \frac{m_{t+1}}{1-\beta_2^{\,t}}$$
$$\mathrm{w}_{t+1} = \mathrm{w}_t - \frac{\eta}{m_{t+1}+\varepsilon} v_{t+1}$$

$\beta_1 = 0.9, \beta_2 = 0.99$

Illustrations





À voir sur :
www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Autre excellent survol

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