RESEAUX DE NEURONES

(ANN: Artificial Neural °Networks)

Dr Ghazi Aziz
Sup'Management

RESEAUX DE NEURONES

❖Compréhension et modélisation cerveau

❖Essai imitation pour reproduire fonctions évoluées

❖ Outils mathématiques pour l'ingénieur

RESEAUX DE NEURONES

Connexionisme

- Modélisation mathématique du cerveau humain.
- Réseaux de neurones formels = réseaux d 'unités de calcul élémentaire interconnectées.

- 2 axes de recherche :
  - étude et modélisation des phénomènes naturels d 'apprentissage (biologie, physiologie du cerveau)
  - algorithmes pour résoudre des problèmes complexes

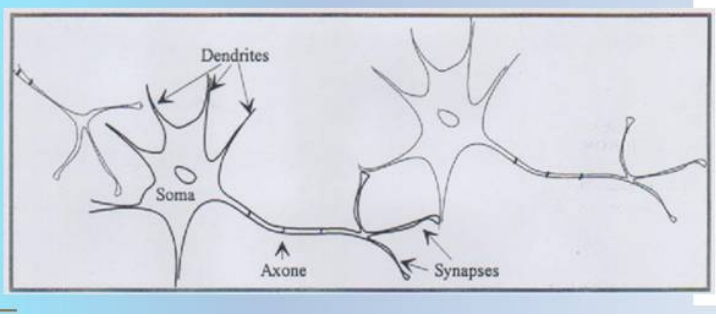
RESEAUX DE NEURONES

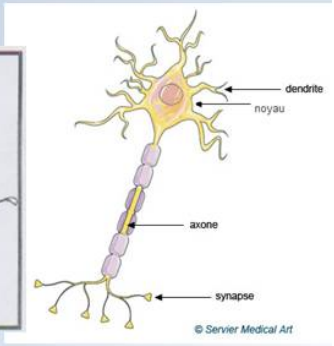
Applications :

- statistiques : analyse de données / prévision / classification
- robotique : contrôle et guidage de robots ou de véhicules autonomes
- imagerie / reconnaissance de formes
- traitement du signal
- simulation de l'apprentissage

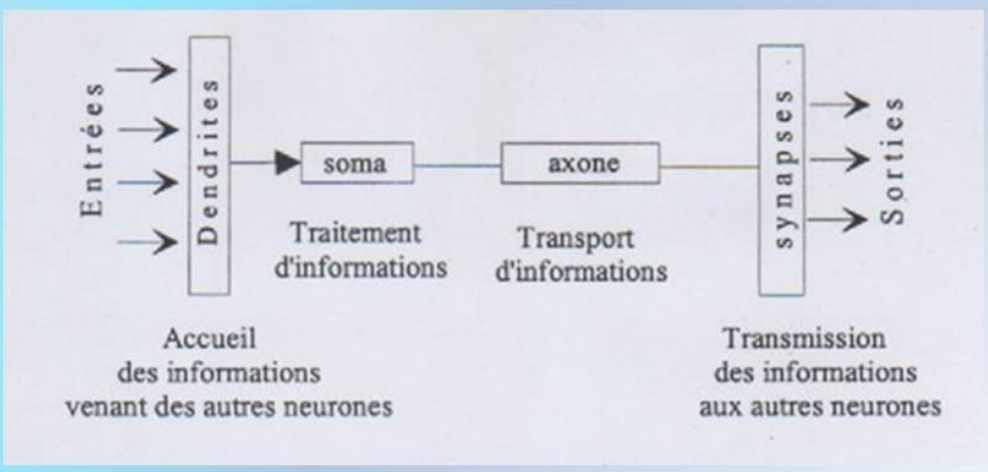
MODELE BIOLOGIQUE

• Dans le cerveau, les neurones sont reliés entre eux par l'intermédiaire d'axones et de dendrites. Les dendrites représentent les entrées du neurone et son axone sa sortie. Un neurone émet un signal en fonction des signaux qui lui proviennent des autres neurones. Une intégration des signaux reçus au cours du temps, c'est-à-dire une sommation des signaux, a lieu au niveau du neurone. En général, quand la somme dépasse un certain seuil, le neurone émet à son tour un signal électrique.



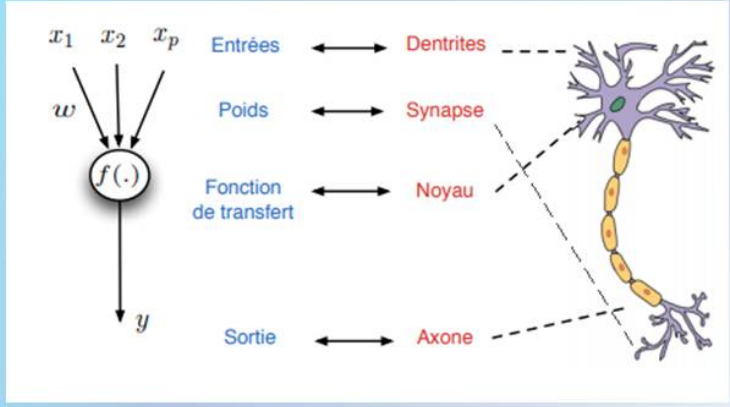


MODELE BIOLOGIQUE



Définition réseau de neurones artificiel

Un réseau de neurones aussi appelé réseau de neurones artificiels est un modèle mathématique très simple dérivé du neurone biologique dont les composantes élémentaires sont des unités de calcul qui reçoivent des entrées pour produire des sorties. Ce modèle reprend les principes du fonctionnement du neurone biologique, en particulier par la sommation des entrées et la pondération de la somme de ses entrées par des poids synaptiques (aussi appelés coefficients synaptiques)



HISTORIQUE

- Mac Culloch et Pitts (1943) : définition d 'un neurone formel
- Loi de Hebb (1949)
- Rosenblatt (1958), Widrow et Hoff : modèle avec processus d 'apprentissage, perceptron
- 1960 (Widrow, Hoff) ADALINE
- Minsky et Papert (1969) : limites des perceptrons Problème XOR
- Kohonen (1972) : mémoires associatives
- 1986 (Rumelhart et. al) MLP et backpropagation
- 1992 (Vapnik et. al) SVM
- 1998 (LeCun et. al) LeNet
- 2010 (Hinton et. al) Deep Neural Networks
- 2012 (Krizhevsky, Hinton et. al) AlexNet, ILSVRC'2012, GPU – 8 couches
- 2014 GoogleNet – 22 couches
- 2015 Inception (Google) – Deep Dream
- 2016 ResidualNet (Microsoft/Facebook) – 152 couches

Principes généraux

Idée générale des réseaux de neurones :

• combiner de nombreuses fonctions élémentaires pour former des fonctions complexes.

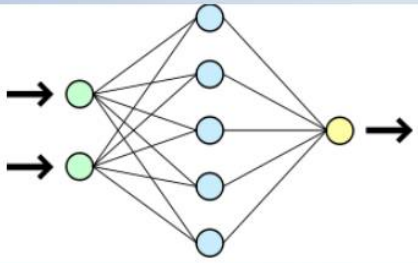
•Apprendre les liens entre ces fonctions simples a partir d'exemples étiquetés

Analogie avec le cerveau :

•Fonctions élémentaires (Perceptron) = neurones

•Connexion = synapse

•Apprentissage des connexions = la connaissance



Le neurone artificiel

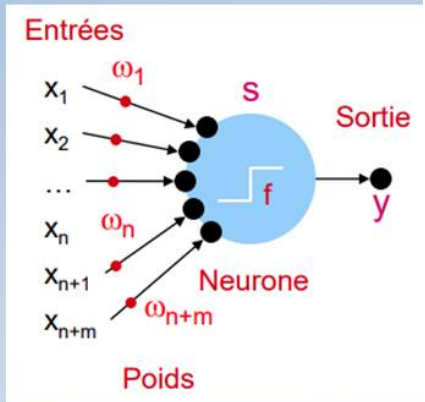
Selon Mc Culloch et Pitts :

– Modèle mathématique très simple, dérivé de la réalité biologique

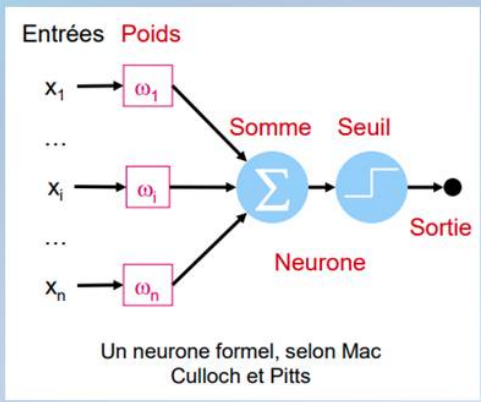
– Neurone :

- Un automate composé de :

1. Un ensemble d'entrées, xi
2. Un ensemble de poids, wi
3. Un seuil, s
4. Une fonction d'activation, f
5. Une sortie, y



Le neurone artificiel



Un neurone formel, selon Mac Culloch et Pitts

Fonctionnement

– L'intégration temporelle, difficile à représenter, est remplacée par
  - une sommation des signaux arrivant au neurone

– Les signaux sont appelés
  - les entrées

– La somme obtenue est comparée à un seuil et on déduit
  - la sortie

Le neurone artificiel

Calcul de la sortie

$a = \Sigma(\omega i \, .xi)$

Si a > s (seuil) sortie y= 1

sinon sortie y= 0, -1 ou autre valeur ≠ 1 selon l'application

– Ou bien

$a = \Sigma(\omega i \, .ei) - s$

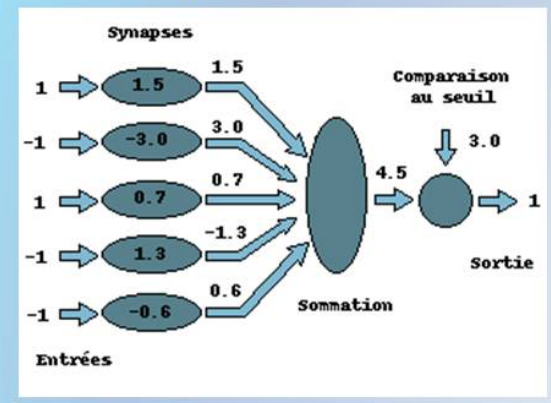
(la valeur de seuil est introduite ici dans le calcul de la somme pondérée)

sortie y= signe(a) (si a > 0 alors x = +1 sinon x = -1)

NEURONE FORMEL

Principes :

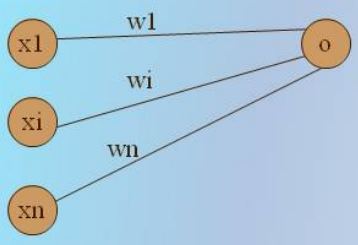
- pas de notion temporelle
- coefficient synaptique : coefficient réel
- sommation des signaux arrivant au neurone
- sortie obtenue après application d 'une fonction de transfert



NEURONE FORMEL

Modélisation :

Le neurone reçoit les entrées x1, …, xi, …, xn.
Le potentiel d'activation du neurone p est défini comme la somme pondérée (les poids sont les coefficients synaptiques wi) des entrées.
La sortie o est alors calculée en fonction du seuil θ.



Soit : $p = x.w = x1.w1 + \ldots + xi.wi + \ldots + xn.wn$

Alors : $o = 1$ si $p > \theta$
$o = 0$ si $p \leq \theta$

DEFINITIONS

Définitions :

• Déterminer un réseau de neurones = Trouver les coefficients synaptiques.
• On parle de phase d'***apprentissage*** : les caractéristiques du réseau sont modifiées jusqu'à ce que le comportement désiré soit obtenu.

• ***Base d'apprentissage*** : exemples représentatifs du comportement ou de fonction à modéliser. Ces exemples sont sous la forme de couples (entrée ; sortie) connus.

• ***Base d'essai*** : pour une entrée quelconque (bruitée ou incomplète), calculer la sortie. On peut alors évaluer la performance du réseau.

DEFINITIONS

Définitions :
• ***apprentissage supervisé*** : les coefficients synaptiques sont évalués en minimisant l 'erreur (entre sortie souhaitée et sortie obtenue) sur une base d 'apprentissage.

• ***apprentissage non-supervisé*** : on ne dispose pas de base d 'apprentissage. Les coefficients synaptiques sont déterminés par rapport à des critères de conformité : spécifications générales.

• ***sur-apprentissage*** : on minimise l 'erreur sur la base d 'apprentissage à chaque itération mais on augmente l 'erreur sur la base d 'essai. Le modèle perd sa capacité de généralisation : c 'est l ***'apprentissage par cœur***.
⇒ Explication : trop de variables explicatives dans le modèle ; on n 'explique plus le comportement global mais les résidus.

LOI DE HEBB

Réseau de neurones :

- n entrées e1, …, en
- m neurones N1, …, Nm.
- wij le coefficient synaptique de la liaison entre les neurones Ni et Nj
- une sortie o
- un seuil S
- Fonction de transfert : fonction Signe

  si $x > 0$: Signe(x) = +1

  si $x \leq 0$: Signe(x) = –1

LOI DE HEBB

Principe :

Si deux neurones sont activés en même temps, alors la force de connexion augmente.

Base d 'apprentissage :

On note S la base d 'apprentissage.

S est composée de couples (e, c) où :

e est le vecteur associé à l 'entrée (e1, …, en)

c la sortie correspondante souhaitée

Définitions :

a_i est la valeur d 'activation du neurone Ni.

LOI DE HEBB

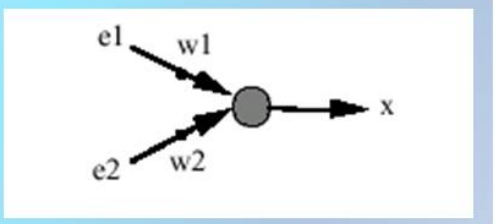
Algorithme :

- μ est une constante positive.
- Initialiser aléatoirement les coefficients wi
- Répéter :
  - Prendre un exemple (e, c) dans S
  - Calculer la sortie o du réseau pour l 'entrée e
  - Si c ≠ o
    - Modification des poids wij :
      - wij= wij+ μ *(ai *aj)
  - Fin Pour

  Fin Si
- Fin Répéter

LOI DE HEBB : exemple

Exemple :



e_1	e_2	x	
1	1	1	(1)
1	-1	1	(2)
-1	1	-1	(3)
-1	-1	-1	(4)

μ = 1

Conditions initiales : coefficients et seuils nuls

LOI DE HEBB

Remarque :

• Calcul des coefficients wij sans utiliser l 'algorithme itératif.

• Principe : les coefficients sont initialisés à 0, μ vaut 1, et on présente tous les exemples de la base d 'apprentissage.

• wij = Σ[(e,c) dans S] (ai * aj)

LOI DE HEBB : exemple

$a = \Sigma(\omega i \, .ei \,) - s$

Exemple :

Conditions initiales : μ = +1, les poids et le seuil sont nuls

• Exemple (1) : o = Signe(w1 * e1 + w2 * e2) = Signe(0) = – 1

o = – 1 ≠ 1 = x La sortie est fausse, il faut donc modifier les poids en appliquant :

wij= wij+ μ *(ai *aj)

⇒ w1 = w1 + e1 * x = 1

w2 = w2 + e2 * x = 1

• Exemple (2) : o = Signe((w1 * e1 + w2 * e2) = Signe(0) = – 1

o = – 1 ≠ 1 = x La sortie est fausse, il faut donc modifier les poids en appliquant :

⇒ w1 = w1 + e1 * x = 2

w2 = w2 + e2 * x = 0

• Exemples (3) et (4) OK o = Signe(w1 * e1 + w2*e2) = Signe(1+0) doit etre negatif ... ⇒ w1 = w1 + e1 * x = 2+1=3

w2 = w2 + e2 * x= -1 o = Signe(-3+ 1) est négatif OK

• Exemples (1) et (2) OK ⇒ STOP

LOI DE HEBB

Application : mémoires auto-associatives

• Kohonen (1977)

Principe : Avec les mémoires auto-associatives, il faut fournir une partie de l'information stockée pour obtenir l'information stockée

•Reconstruction de données :

en entrée : une information partielle ou bruitée

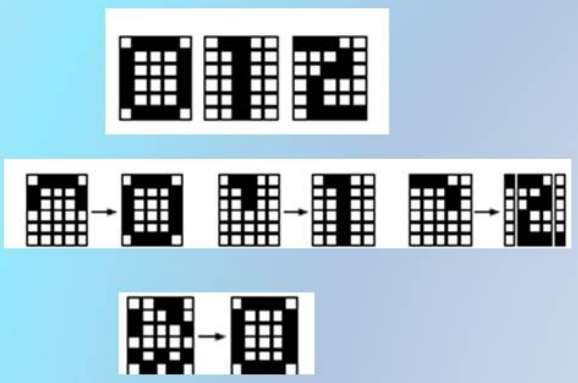
en sortie : le système complète ou corrige l'information

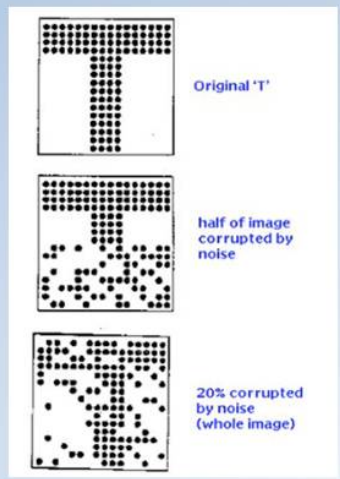
image apprise	image soumise au réseau	Image restituée par le réseau

EXERCICE : Loi de Hebb

Reconstitution d 'images :

les chiffres 0, 1 et 2



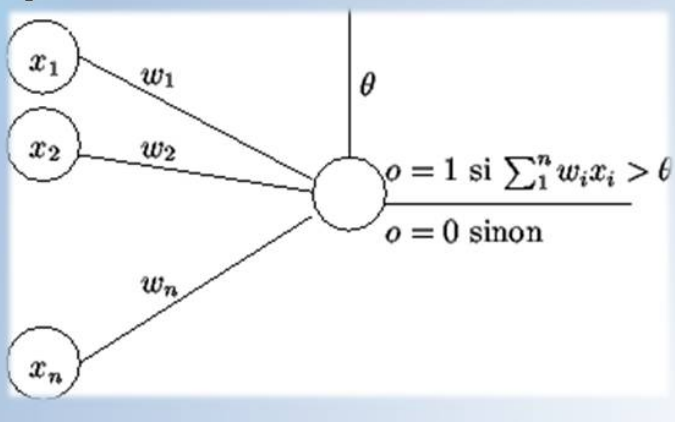


PERCEPTRON

Le perceptron est un modèle de réseau de neurones artificiels qui a été inventé en 1957 par Frank Rosenblatt. Il est utilisé pour la classification binaire, c'est-à-dire pour séparer deux classes. Le perceptron est un modèle simple mais puissant qui peut être utilisé pour résoudre des problèmes de classification linéaire.

Perceptron linéaire à seuil :

- n entrées x1, …, xn
- n coefficients synaptiques w1, …, wn
- une sortie o
- un seuil θ



Récapitulatif

1. Expliquez brièvement comment fonctionnent les neurones humains ?
2. Donnez l'analogie entre les neurones du cerveau et les neurones artificiels
3. Donnez une modélisation de fonctionnement des neurones artificiels ?
4. Donnez la définition de l'apprentissage supervisé
5. Fournir la définition de l' apprentissage non supervisé
6. Quel est le principe de la loi de Hebb ?
7. Donnez l'algorithme de la loi de Hebb
8. Quel est le principe de la mémoire Auto-associative de Kohonen
9. Donnez le schéma du perceptron linéaire à seuil ?

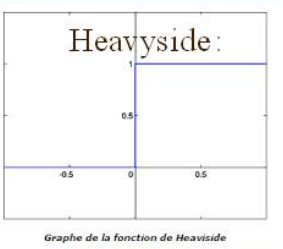
PERCEPTRON

On ajoute une entrée supplémentaire x0 (le biais), avec le coefficient synaptique suivant : w0 = – θ

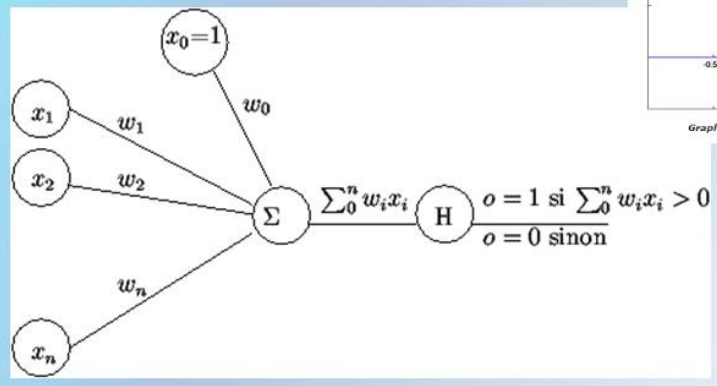
On associe comme fonction de transfert la fonction de

f(x) = 1 si x > 0

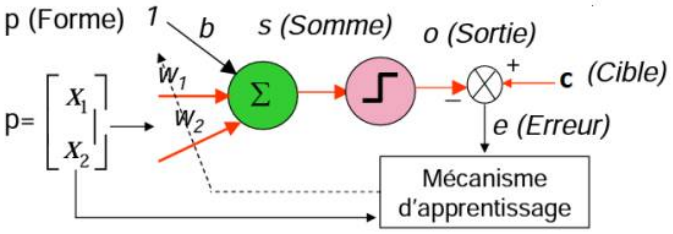
f(x) = 0 sinon



Graphe de la fonction de Heaviside



REGLE DU PERCEPTRON



Apprentissage : $W_i(k+1) = W_i(k) + \Delta W_i$

où: $\Delta W_i = \eta\ (c - o)\ X_i$

avec: η le coefficient d'apprentissage

REGLE DU Widrow-Hoff :

$$\Delta W_i = \eta\ (c - s)\ X_i$$

PERCEPTRON

Apprentissage par l'algorithme du perceptron

On note S la base d'apprentissage.
S est composée de couples (x, c) où :

x est le vecteur associé à l'entrée (x0, x1, …, xn)
c la sortie correspondante souhaitée

On cherche à déterminer les coefficients (w0, w1, …, wn).

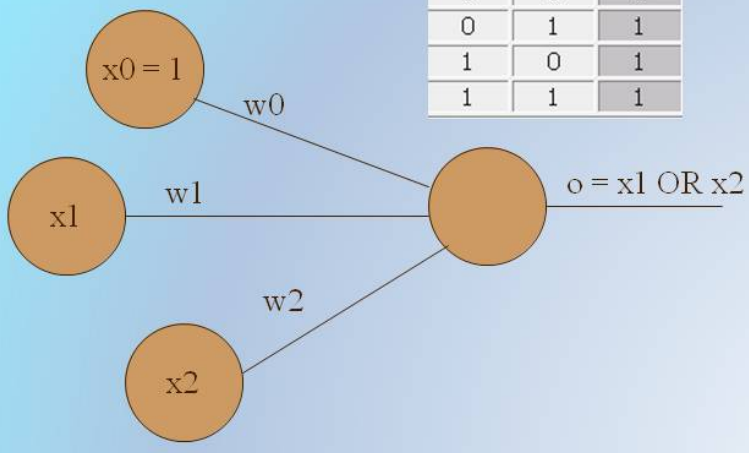
- Initialiser aléatoirement les coefficients wi.
- Répéter :
  - Prendre un exemple (x, c) dans S
  - Calculer la sortie o du réseau pour l'entrée x
  - Mettre à jour les poids :
    - Pour i de 0 à n :
      - $wi = wi + \varepsilon * (c - o) * xi$
    - Fin Pour
  - Fin Répéter

PERCEPTRON : exemple

Apprentissage par l'algorithme du perceptron : exemple

Apprentissage du OU logique

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1



PERCEPTRON : exemple

Apprentissage par l'algorithme du perceptron : exemple

ε = 1

x0 vaut toujours 1

Initialisation : w0 = 0 ; w1 = 1 ; w2 = – 1

$w_i = w_{i-1} + \varepsilon * (c - o) * x_{i-1}$

Étape	w_0	w_1	w_2	Entrée x0x1x2	$\Sigma_0^2 w_i x_i$	o	c	w_0	w_1	w_2
init								0	1	-1
1	0	1	-1	100	0	0	0	0+0x1	1+0x0	-1+0x0
2	0	1	-1	101	-1	0	1	0+1x1	1+1x 0	-1+1x1
3	1	1	0	110	2	1	1	1	1	0
4	1	1	0	111	2	1	1	1	1	0
5	1	1	0	100	1	1	0	1+(-1)x1	1+(-1)x0	0+(-1)x0
6	0	1	0	101	0	0	1	0+1x1	1+1x0	0+1x1
7	1	1	1	110	2	1	1	1	1	1
8	1	1	1	111	3	1	1	1	1	1
9	1	1	1	100	1	1	0	1+(-1)x1	1+(-1)x0	1 +(-1)x0
10	0	1	1	101	1	1	1	0	1	1

Donc : w0 = 0 ; w1 = 1 ; w2 = 1

Ce perceptron calcule le OU logique pour tout couple (x1 ; x2)

EXERCICE 1

Soit l'ensemble d'entraînement suivant :

x_i	y_i
[3, 2, 1]	0
[1, 1, 1]	1
[1, 2, 3]	1

1. Simulez manuellement l'algorithme du perceptron sur cet ensemble de données en les parcourant dans l'ordre du haut vers le bas.

Utilisez un taux d'apprentissage η = 0.1 et initialisez le vecteur de poids w à [0, 0, 0] et le biais b à 0.5.

2. En utilisant le logiciel R, simulez l'algorithme du perceptron sur cet ensemble de données. Z = x11 * w1 + x12 * w2 + x13 * w3 + b

SOLUTION

On possède l'ensemble d'entraînement suivant :

x_i	y_i
$[3, 2, 1]$	0
$[1, 1, 1]$	1
$[1, 2, 3]$	1

La simulation de l'algorithme du perceptron est la suivant :

— Pour x1 : Z = x11 ∗ w1 + x12 ∗ w2 + x13 ∗ w3 + b => Z = 0.5, ce qui est plus grand que 0, alors, f(x1) = 1. Puisque la prédiction est fausse, alors nous devons mettre à jour les poids.

W' = w + η ∗ (d − y) ∗ x

W1'= 0 + 0, 1 ∗ (0 − 1) ∗ 3 = −0, 3

w2'= 0 + 0, 1 ∗ (0 − 1) ∗ 2 = −0, 2

W3' = 0 + 0, 1 ∗ (0 − 1) ∗ 1 = −0, 1

b ' = 0, 5 + 0, 1 ∗ (0 − 1) ∗ 1 = 0,4 le nouveau calcul de Z donne -1 < 0 donc Z=0 ok

— Pour x2 : Z = x21 ∗ w1' + x22 ∗ w2' + x23 ∗ w3' + b'=>Z = −0.2, ce qui n'est pas plus grand que 0, alors, f(x2) = 0.

Puisque la prédiction est fausse, alors nous devons mettre à jour les poids.

w '' = w' + η ∗ (d − y) ∗ x

w1''= −0, 3 + 0, 1 ∗ (0 − 1) ∗ 1 = −0, 2

w2 '' = −0, 2 + 0, 1 ∗ (0 − 1) ∗ 1 = −0, 1

w3 '' = −0, 2 + 0, 1 ∗ (0 − 1) ∗ 1 = 0

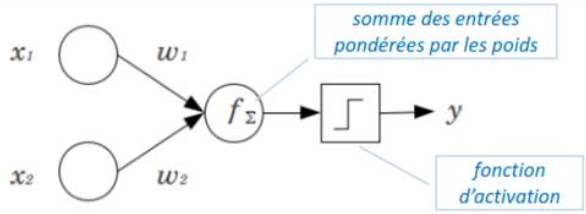
b''= 0, 4 + 0, 1 ∗ (0 − 1) ∗ 1 = 0, 5 le nouveau calcul de Z donne 0,2 > 0 donc Z=1 ok

— Pour x3 : Z = x31 ∗ w1''+ x32 ∗ w2''+ x33 ∗ w3'' + b'' Z = 0.1, ce qui est plus grand que 0, alors, f(x3) = 1. Ce qui est correct, alors aucune mise à jours des poids n'est requise.

2. La simulation de l'algorithme du perceptron sur cet ensemble de données en utilisant le logiciel R :

```
perceptron <- function(x, y, lr) {
  # initialisation du vesteur poids
  poids <- c(0.5,0,0,0)
  # Boucle pour les donn es d'entra nement
  for (j in 1:length(y)){
    z <- 0
    for (i in 1:3) {
      # Pr dire le "label" binaire
      z <- z +poids[i+1]*as.numeric(x[i,j])
    }
    z <- z + poids[1]
    if(z <= 0) {
      ypred <- 0
    } else {
      ypred <- 1
    }
    for (i in 1:3){
      poids[i+1] <- poids[i+1] + lr * (y[j] - ypred) *as.numeric(
          x[i,j])
    }
    poids [1] <- poids[1] + lr * (y[j] - ypred)
    # afficher le poids
    print(poids)
  }
}

x <- matrix(c(3,2,1,1,1,1,1,2,3),ncol = 3, nrow = 3)
y <- c(0,1,1)

err <- perceptron(x, y,0.1)
```

Construisons notre perceptron sur Excel



	A	B	C	D	E	F	G	H	I	J	K	L
1	Modèle perceptron				poids des connexions avant correction		somme des entrées pondérées	fonction d'activation	sortie effective	vitesse d'apprentissage 0,1	poids des connexions après correction	
2	entrées		sortie attendue									
3					w1	w2			y	signe de la correction	w1'	w2'
4	x1	x2	y_theor		0,1	0,1	initialisation aléatoire				=E4	=F4
5					=K4	=L4	=A5*E5+B5*F5	=SI(G5>0,5;1;0)	=H5	=SI(I5=C5;0;SI(I5<C5;1;-1))	=E5+J5*J$2	=F5+J5*J$2
6												

=c-o=C5-I5

- Ce modèle de réseau de neurones artificiels est un **perceptron (Rosenblatt, 1957).**
- Il peut apprendre tout seul à faire des opérations à partir de données d'entrée (x1 et x2) et du résultat attendu selon l'opération (y_theor).
C'est un **apprentissage supervisé (parce qu'on lui donne le résultat attendu).**
- Il va apprendre devant nous les opérations booléennes OU, ET, et XOR

Notre perceptron apprend le OU :

Table de vérité du OU

	A	B	C	D	E	F	G	H	I	J	K	L
1	Opération OU				poids des connexions avant correction		somme entrées pondérées	fonction d'activation	sortie effective	0,1 signe correction	poids des connexions après correction	
2	entrées		sortie attendue									
3					w1	w2			y		w1'	w2'
4	x1	x2	y_theor		0,1	0,1	initialisation aléatoire				0,1	0,1
5	0	0	0		0,1	0,1	0	0	0	0	0,1	0,1
6	1	0	1		0,1	0,1	0,1	0	0	1	0,2	0,2
7	0	1	1		0,2	0,2	0,2	0	0	1	0,3	0,3
8	1	1	1		0,3	0,3	0,6	1	1	0	0,3	0,3
9	0	0	0		0,3	0,3	0	0	0	0	0,3	0,3
10	1	0	1		0,3	0,3	0,3	0	0	1	0,4	0,4
11	0	1	1		0,4	0,4	0,4	0	0	1	0,5	0,5
12	1	1	1		0,5	0,5	1	1	1	0	0,5	0,5
13	1	1	1		0,5	0,5	1	1	1	0	0,5	0,5
14	1	0	1		0,5	0,5	0,5	0	0	1	0,6	0,6
15	1	1	1		0,6	0,6	1,2	1	1	0	0,6	0,6
16	0	1	1		0,6	0,6	0,6	1	1	0	0,6	0,6
17	0	0	0		0,6	0,6	0	0	0	0	0,6	0,6
18	0	0	0		0,6	0,6	0	0	0	0	0,6	0,6
19	1	0	1		0,6	0,6	0,6	1	1	0	0,6	0,6
20	1	0	1		0,6	0,6	0,6	1	1	0	0,6	0,6
21	1	1	1		0,6	0,6	1,2	1	1	0	0,6	0,6
22	0	0	0		0,6	0,6	0	0	0	0	0,6	0,6
23	1	1	1		0,6	0,6	1,2	1	1	0	0,6	0,6
24	0	1	1		0,6	0,6	0,6	1	1	0	0,6	0,6
25	0	1	1		0,6	0,6	0,6	1	1	0	0,6	0,6
26	0	1	1		0,6	0,6	0,6	1	1	0	0,6	0,6
27	0	1	1		0,6	0,6	0,6	1	1	0	0,6	0,6
28	1	0	1		0,6	0,6	0,6	1	1	0	0,6	0,6
29	1	0	1		0,6	0,6	0,6	1	1	0	0,6	0,6
30	0	0	0		0,6	0,6	0	0	0	0	0,6	0,6
31	0	0	0		0,6	0,6	0	0	0	0	0,6	0,6
32	1	1	1		0,6	0,6	1,2	1	1	0	0,6	0,6

Quand c'est 0, c'est OK (rien à corriger) !

Notre perceptron apprend le ET

Table de vérité du ET

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Opération ET**				poids des connexions avant correction		somme entrées pondérées	fonction d'activation	sortie effective	0,1 signe correction	poids des connexions après correction	
2	entrées		sortie attendue									
3					w1	w2			y		w1'	w2'
4	x1	x2	y_theor		0,1	0,1	*initialisation aléatoire*				0,1	0,1
5	0	0	0		0,1	0,1	0	0	0	0	0,1	0,1
6	1	0	0		0,1	0,1	0,1	0	0	0	0,1	0,1
7	0	1	0		0,1	0,1	0,1	0	0	0	0,1	0,1
8	1	1	1		0,1	0,1	0,2	0	0	1	0,2	0,2
9	1	1	1		0,2	0,2	0,4	0	0	1	0,3	0,3
10	1	0	0		0,3	0,3	0,3	0	0	0	0,3	0,3
11	1	1	1		0,3	0,3	0,6	1	1	0	0,3	0,3
12	0	1	0		0,3	0,3	0,3	0	0	0	0,3	0,3
13	0	0	0		0,3	0,3	0	0	0	0	0,3	0,3
14	1	1	1		0,3	0,3	0,6	1	1	0	0,3	0,3
15	0	0	0		0,3	0,3	0	0	0	0	0,3	0,3
16	1	0	0		0,3	0,3	0,3	0	0	0	0,3	0,3
17	1	0	0		0,3	0,3	0,3	0	0	0	0,3	0,3
18	0	0	0		0,3	0,3	0	0	0	0	0,3	0,3

Il apprend vite, notre petit perceptron

- y = y_theor chaque fois,
- poids stables

Notre perceptron a du mal avec le XOR

Table de vérité du XOR

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Opération XOR (OU exclusif)**				poids des connexions avant correction		somme entrées pondérées	fonction d'activation	sortie effective	0,1 signe correction	poids des connexions après correction	
2	entrées		sortie attendue									
3					w1	w2			y		w1'	w2'
4	x1	x2	y_theor		0,1	0,1	*initialisation aléatoire*				0,1	0,1
5	0	0	0		0,1	0,1	0	0		0	0,1	0,1
6	1	0	1		0,1	0,1	0,1	0		1	0,2	0,2
7	0	1	1		0,2	0,2	0,2	0		1	0,3	0,3
8	1	1	0		0,3	0,3	0,6	1		-1	0,2	0,2
9	1	0	1		0,2	0,2	0,2	0		1	0,3	0,3
10	0	0	0		0,3	0,3	0	0		0	0,3	0,3
11	1	0	1		0,3	0,3	0,3	0		1	0,4	0,4
12	1	0	1		0,4	0,4	0,4	0		1	0,5	0,5
13	0	1	1		0,5	0,5	0,5	0		1	0,6	0,6
14	0	1	1		0,6	0,6	0,6	1		0	0,6	0,6
15	0	1	1		0,6	0,6	0,6	1		0	0,6	0,6
16	1	1	0		0,6	0,6	1,2	1		-1	0,5	0,5
17	0	0	0		0,5	0,5	0	0		0	0,5	0,5
18	1	1	0		0,5	0,5	1	1		-1	0,4	0,4
19	0	1	1		0,4	0,4	0,4	0		1	0,5	0,5
20	0	1	1		0,5	0,5	0,5	0		1	0,6	0,6
21	0	0	0		0,6	0,6	0	0		0	0,6	0,6
22	1	1	0		0,6	0,6	0,6	1		0	0,6	0,6
23	0	0	0		0,6	0,6	0	0		0	0,6	0,6
24	1	0	1		0,6	0,6	0,6	1		0	0,6	0,6
25	0	1	1		0,6	0,6	0,6	1		0	0,6	0,6
26	1	1	0		0,6	0,6	1,2	1		-1	0,5	0,5
27	0	0	0		0,5	0,5	0	0		0	0,5	0,5
28	1	1	0		0,5	0,5	1	1		-1	0,4	0,4
29	1	0	1		0,4	0,4	0,8	1		-1	0,3	0,3
30	0	0	0		0,3	0,3	0	0		0	0,3	0,3
31	1	1	0		0,3	0,3	0,6	1		-1	0,2	0,2
32	1	0	1		0,2	0,2	0,2	0		1	0,3	0,3

Les poids ne se stabilisent pas,
et y calculé (après fonction d'activation)
est rarement égal à y attendu

Notre perceptron a du mal avec le XOR

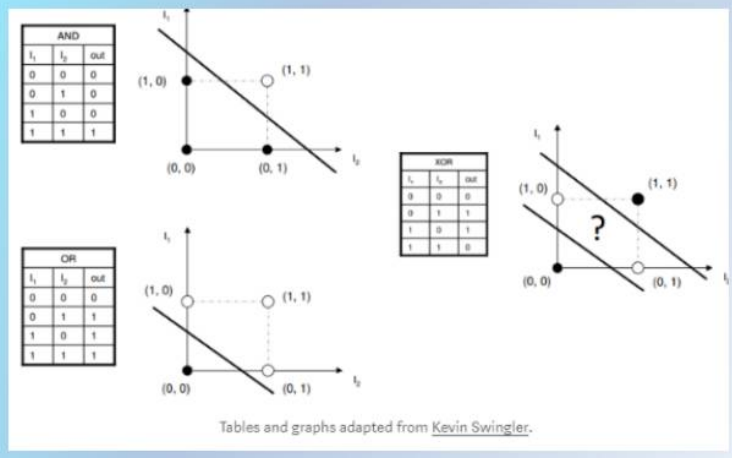
Table de vérité du XOR

Opération XOR (OU exclusif)				poids des connexions avant correction		somme entrées pondérées	fonction d'activation	sortie effective	0,1 signe correction	poids des connexions après correction	
entrées			sortie attendue	w1	w2			y		w1'	w2'
x1	x2		y_theor	0,1	0,1		initialisation aléatoire			0,1	0,1
0	0		0	0,1	0,1	0	0		0	0,1	0,1
1	0		1	0,1	0,1	0,1	0		1	0,2	0,2
0	1		1	0,2	0,2	0,2	0		1	0,3	0,3
1	1		0	0,3	0,3	0,6	1		-1	0,2	0,2
1	0		1	0,2	0,2	0,2	0		1	0,3	0,3
0	0		0	0,3	0,3	0	0		0	0,3	0,3
1	0		1	0,3	0,3	0,3	0		1	0,4	0,4
1	0		1	0,4	0,4	0,4	0		1	0,5	0,5
0	1		1	0,5	0,5	0,5	0		1	0,6	0,6
							1		0	0,6	0,6
							1		0	0,6	0,6
							1		-1	0,5	0,5
							0		0	0,5	0,5
							1		-1	0,4	0,4
							0		1	0,5	0,5
							0		1	0,6	0,6
							0		0	0,6	0,6
							1		0	0,6	0,6
							0		0	0,6	0,6
							1		0	0,6	0,6
							1		0	0,6	0,6
1	1		0	0,6	0,6	1,2	1		-1	0,5	0,5
0	0		0	0,5	0,5	0	0		0	0,5	0,5
1	1		0	0,5	0,5	1	1		-1	0,4	0,4
1	1		0	0,4	0,4	0,8	1		-1	0,3	0,3
0	0		0	0,3	0,3	0	0		0	0,3	0,3
1	1		0	0,3	0,3	0,6	1		-1	0,2	0,2
1	0		1	0,2	0,2	0,2	0		1	0,3	0,3

On aurait pu initialiser les deux poids avec des valeurs différentes.
On pourrait ne pas mettre à jour le poids correspondant à une entrée nulle.
Mais notre perceptron n'aurait quand même pas réussi à apprendre l'opération XOR.

Pourquoi le XOR passe mal ?

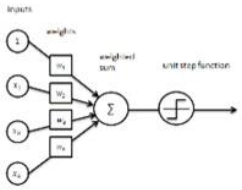
- *« Oh, ben si un perceptron n'est même pas capable d'apprendre une opération aussi simple que le XOR, ça ne sert à rien ! Il faut arrêter de subventionner les recherches connexionnistes… et donner les sous aux symbolistes. » (à peu près le discours de Minsky en 1969)*
- Le perceptron simple (à une seule couche) est un discriminant linéaire : il marche bien quand une seule ligne droite suffit à séparer les sorties possibles



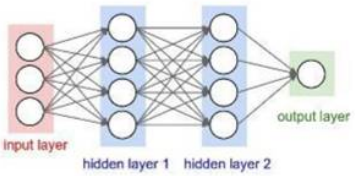
Tables and graphs adapted from Kevin Swingler.

Solution : ajouter des couches

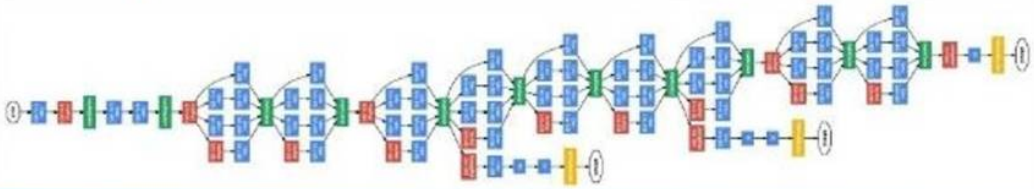
- Perceptron à une couche



- Perceptron multicouche



- Deep learning *(= beaucoup de couches)*



PERCEPTRON

Apprentissage par l'algorithme du perceptron : remarques

$w_i = w_{i-1} + \varepsilon * (c - o) * x_{i-1}$

- ε bien choisi, suffisamment petit
- Si ε trop grand : risque d'oscillation autour du minimum
- Si ε trop petit : nombre élevé d'itérations
- En pratique : on diminue graduellement ε au fur et à mesure des itérations

PERCEPTRON : Exemple de script Python

```
import numpy as np
# Définir les entrées et les sorties
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
Y = np.array([0, 0, 0, 1])
# Initialiser les poids synaptiques
W = np.array([0.1, 0.2])
# Définir le taux d'apprentissage
eta = 0.1
# Définir le nombre d'itérations
n = 10
# Boucle d'apprentissage
for i in range(n):
    for j in range(len(X)):
        # Calculer la sortie prédite
        y_pred = np.dot(X[j], W)
        # Calculer l'erreur
        error = Y[j] - y_pred
        # Mettre à jour les poids synaptiques
        W += eta * error * X[j]
# Afficher les poids synaptiques finaux
print("Les poids synaptiques finaux sont : ", W)
```

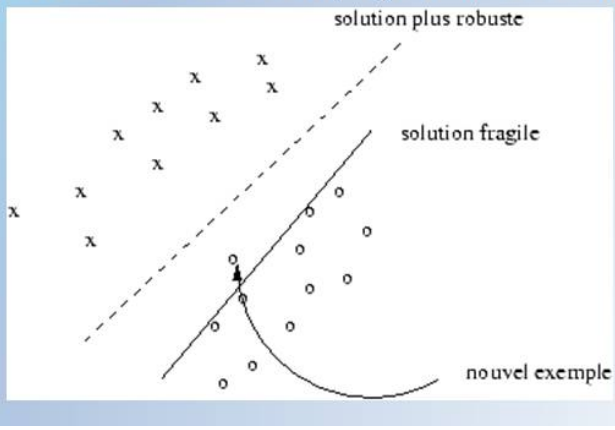
NB : NumPy est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux

```
Les poids synaptiques finaux sont :  [0.3319537  0.36682154]
```

PERCEPTRON

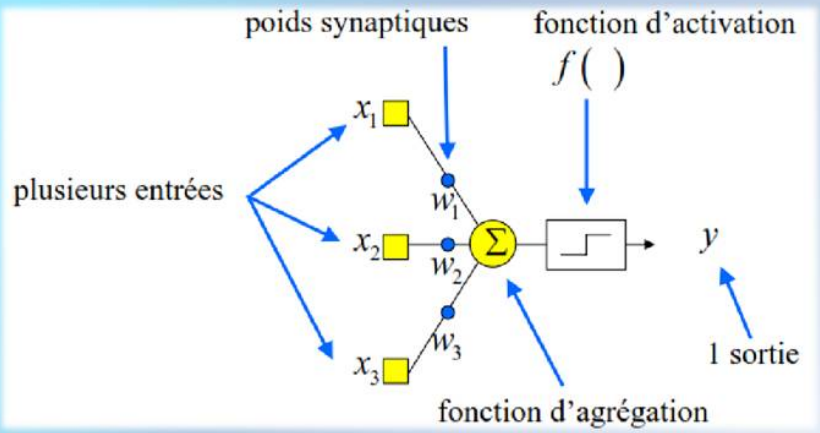
Apprentissage par l'algorithme du perceptron : remarques

- Si l'échantillon n'est pas linéairement séparable, l'algorithme ne converge pas.
- L'algorithme peut converger vers plusieurs solutions (selon les valeurs initiales des coefficients, la valeur de ε, l'ordre de présentation des exemples).
- La solution n'est pas robuste : un nouvel exemple peut remettre en cause le perceptron appris.

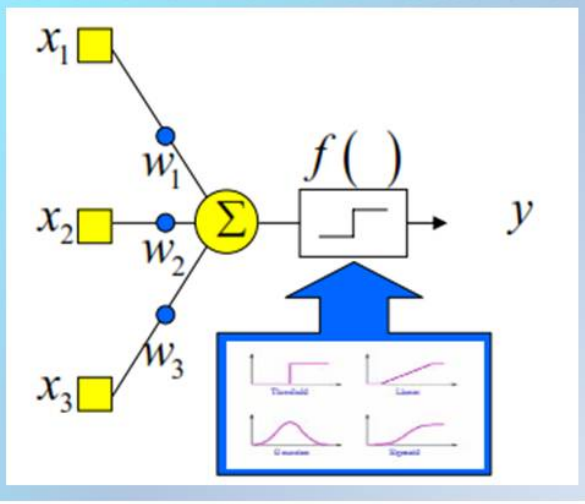


Architecture d'un neurone formel

Le neurone formel : exemple avec 3 entrées
Un neurone est une cellule qui "concentre" des signaux, qui les traite et qui les traduit en une réponse.



Le neurone formel : architecture et calculs



Calcul de la sortie du neurone :

$$y = f\left(w_0 + x_1 w_1 + x_2 w_2 + \ldots + x_n w_n\right) = f\left(w_0 + \mathbf{w}^T \mathbf{x}\right)$$

Correction des poids synaptiques :

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta \mathbf{w}_k$$

L' Adaline et le Perceptron

- **On distingue 2 types de neurone formel :**

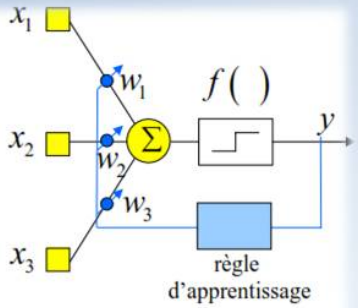
Le Perceptron de F. Rosenblatt (1958)
+ la règle de D. Hebb (1949)
L'Adaline (ADApitve LINear Element) de B. Widrow et T. Hoff (1960)

- **Le Perceptron de F. Rosenblatt (1958) et la règle de D. Hebb (1949)**

A chaque itération :

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}_k$$

$$\Delta\mathbf{w}_k = \eta y_k \mathbf{x}_k$$



η représente le coefficient d'apprentissage
(sa valeur est généralement comprise entre 0 et 1)

L' Adaline

L'Adaline de B. Widrow et T. Hoff (1960)
(ADAptive LINear Element, la fonction d'activation est unitaire)

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}_k$$

$$\varepsilon_k = d_k - y_k = d_k - \mathbf{w}_k^T \mathbf{x}_k$$

d la sortie désirée *y* la sortie réelle

On peut utiliser 2 règles :

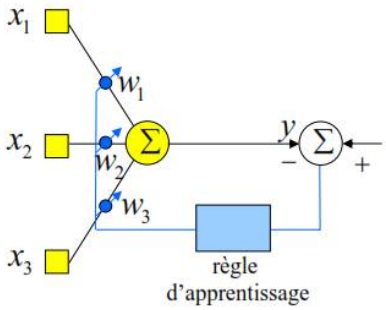
μ LMS

LMS (Least Mean Square)

$$\Delta\mathbf{w}_k = \mu \varepsilon_k \mathbf{x}_k \alpha$$

α LMS

$$\Delta\mathbf{w}_k = \alpha \frac{\varepsilon_k \mathbf{x}_k}{\lambda + \|\mathbf{x}_k\|^2}$$

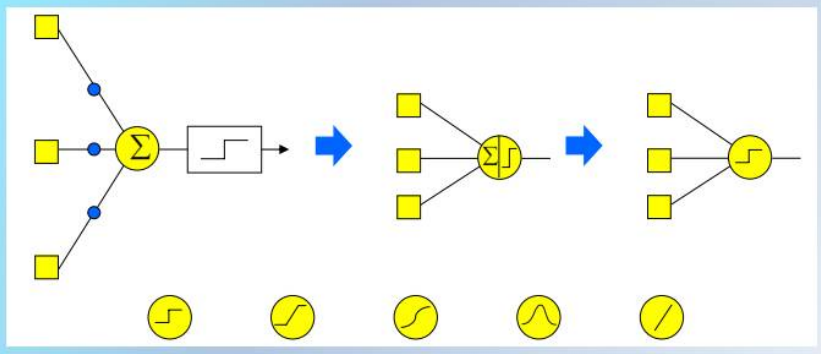


L'apprentissage d'un Adaline s'effectue en ligne et est basé sur la minimisation d'une erreur, l'erreur quadratique moyenne.

Adaline converge vers l'erreur des moindres carrés qui est : $E = (d - o)^2$

Neurones formels utilisés

Que ce soit le Perceptron ou l'Adaline, ces 2 modèles sont toujours actuellement utilisés. Ils sont la base de tous réseaux de neurones artificiels dans le domaine de l'intelligence artificielle en particulier dans le traitement du signal.
La représentation d'un neurone formel peut se simplifier :



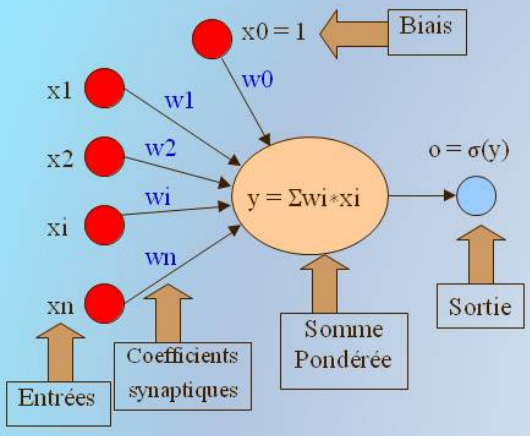
PERCEPTRON : REGLE DELTA GENERALISEE

La loi de delta généralisée du perceptron est une extension de la loi de delta du perceptron, qui est un algorithme d'apprentissage supervisé pour les réseaux de neurones artificiels. Cette extension permet de résoudre des problèmes de classification non linéaires en utilisant des fonctions d'activation non linéaires. Elle a été proposée par Rumelhart, Hinton et Williams en 1986

PERCEPTRON : REGLE DELTA GENERALISEE

Fonctionnement d 'un neurone :

- somme pondérée des signaux reçus (en tenant compte du biais)
- puis application d 'une fonction de transfert (ou d 'activation) : sigmoïde log, sigmoïde tangente hyperbolique, linéaire



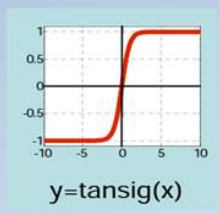
On note :
- y = x.w
- o = σ(x.w)

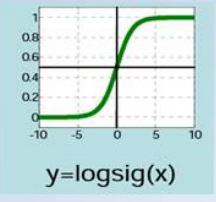
X0 = 1 : ***biais***
Représente le seuil d 'activation du neurone

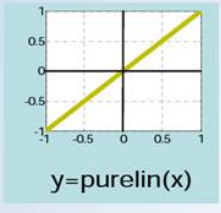
PERCEPTRON : REGLE DELTA GENERALISEE

Fonctions de transfert :

- Sigmoïde tangente hyperbolique : ***tansig***
  - sorties entre -1 et +1
  - $\sigma(x) = \tanh(x)$ et $\sigma'(x) = 1 - \tanh^2(x) = 1 - \sigma^2(x)$
- Sigmoïde log : ***logsig***
  - sorties entre 0 et 1
  - $\sigma(x) = e^x/(e^x + 1) = 1/(1 + e^{-x})$ et $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$
- Linéaire : ***purelin***
  - $\sigma(y) = y$ et $\sigma'(y) = 1$







PERCEPTRON : REGLE DELTA GENERALISEE

Apprentissage par descente de gradient :

Soient le vecteur des entrées x et le vecteur des coefficients synaptiques w.
La sortie vaut alors : $o = \sigma(x.w) = \sigma(x0.w0 + \ldots + xn.wn)$,
σ étant une fonction de transfert continue et dérivable.

Posons : $y = x.w$

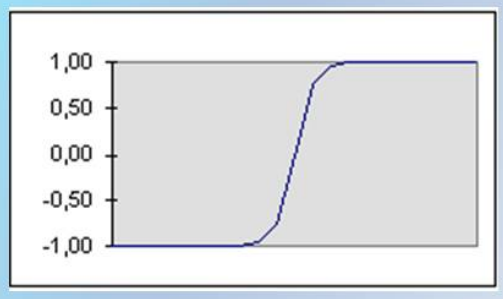
Soit S la base d 'apprentissage composée de couples (x, c), où c est la sortie attendue pour x.

On définit ainsi l 'erreur sur le réseau pour la base d 'apprentissage S :
$E(w) = 1/2 \Sigma[(x,c) \text{ dans } S] (c - o)^2$

Problème : trouver w qui minimise E(w).
⇒ Méthode du gradient.

PERCEPTRON : REGLE DELTA GENERALISEE

- Fonction de transfert $\sigma(y)$
- Par exemple : σ est la tangente hyperbolique (sigmoïde tansig)
  $\sigma(y) = \tanh(y)$ et $\sigma'(y) = 1 - \tanh^2(y) = 1 - \sigma^2(y)$



- La sortie o calculée par le perceptron pour une entrée x est :
  $o = \sigma(x.w)$

PERCEPTRON : REGLE DELTA GENERALISEE

Méthode du gradient (rappel) :

$$x_{n+1} = x_n - \varepsilon * f'(x_n) = x_n + \Delta x_n$$

$$E(w) = 1/2\,\Sigma[(x,c) \text{ dans } S]\,(c - o)^2$$

$$\frac{\partial E(W)}{\partial wi} = \frac{1}{2}\frac{\partial}{\partial wi}\sum_S (c-o)^2 = \frac{1}{2}\sum_S \frac{\partial}{\partial wi}(c-o)^2$$

$$\frac{\partial E(W)}{\partial wi} = \frac{1}{2}\sum_S 2(c-o)\frac{\partial}{\partial wi}(c-o) = \sum_S (c-o)\frac{\partial}{\partial wi}(c-\sigma(x\cdot w))$$

$$Or : \frac{\partial}{\partial wi}(c-\sigma(x.w)) = \frac{\partial}{\partial wi}(c-\sigma(y)) = \frac{\partial}{\partial y}(c-\sigma(y)) * \frac{\partial y}{\partial wi} = -\sigma'(y) * xi$$

$$\frac{\partial E(W)}{\partial wi} = \sum_S (c-o)(-xi)\sigma'(x\cdot w)$$

$$D'où : \Delta wi = -\varepsilon * \frac{\partial E(W)}{\partial wi} = \varepsilon \sum_S xi * (c-o) * \sigma'(x\cdot w)$$

PERCEPTRON : REGLE DELTA GENERALISEE

Apprentissage par descente de gradient : algorithme

- Initialiser aléatoirement les coefficients wi.
- Répéter :
  - Pour tout i :
    - Δwi = 0
  - Fin Pour
  - Pour tout exemple (x, c) dans S
    - Calculer la sortie o du réseau pour l 'entrée x
    - Pout tout i :
      - Δwi = Δwi + ε * (c − o) * xi * σ'(x.w)
    - Fin Pour
  - Fin Pour
  - Pour tout i :
    - wi = wi + Δwi
  - Fin Pour
- Fin Répéter

PERCEPTRON : REGLE DELTA GENERALISEE

Variante de la Règle Delta généralisée :

• On ne calcule pas les variations de coefficients en sommant sur tous les exemples de S mais on modifie les poids à chaque présentation d'exemple.

Initialiser aléatoirement les coefficients wi.
• Répéter :
  •Prendre un exemple (x, c) dans S
  •Calculer la sortie o du réseau pour l'entrée x
  •Pour i de 1 à n :
    • $wi = wi + \varepsilon * (c - o) * xi * \sigma'(x.w)$
  •Fin Pour
•Fin Répéter

TP DESCENTE DU GRADIENT

PERCEPTRON : REGLE DELTA

La loi de delta du perceptron est un algorithme d'apprentissage supervisé qui permet de mettre à jour les poids des connexions entre les neurones du réseau de neurones artificiels. Elle est basée sur la règle de Widrow-Hoff et utilise la descente de gradient stochastique pour minimiser l'erreur de classification

Apprentissage : algorithme de Widrow-Hoff (adaline / règle delta)

La fonction de transfert est linéaire (purelin) : $\sigma(y) = y$
Donc : $\sigma'(y) = 1$
• Initialiser aléatoirement les coefficients wi.
• Répéter :
  •Prendre un exemple (x, c) dans S
  •Calculer la sortie o du réseau pour l'entrée x
  •Pour i de 1 à n :
    • $wi = wi + \varepsilon * (c - o) * xi$
  •Fin Pour
•Fin Répéter

PERCEPTRONS / REGLES DELTA

Remarques

• ε : pas d'apprentissage

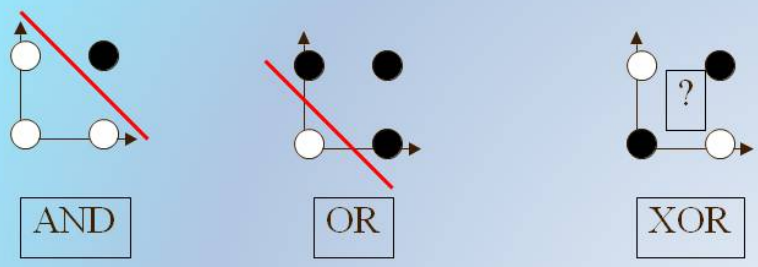
• Règles « delta » et « delta généralisée » : les algorithmes convergent vers la solution des moindres carrés.

• La règle « delta généralisée », par la non-linéarité de la fonction de transfert σ, permet de minimiser l'importance d'un élément étranger (erreur de mesure, bruit trop important, …).

PERCEPTRONS / REGLES DELTA

Remarques

• Perceptrons = classificateurs linéaires (ensembles linéairement séparables).



• Pour apprendre le OU Exclusif (XOR), on utilise un Perceptron Multi-Couches.

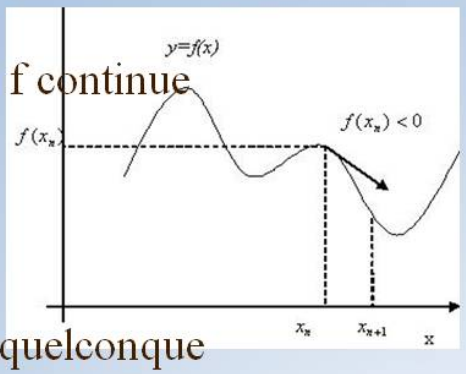
Optimisation : gradient (rappels)

Problème :

Trouver le minimum de la fonction f continue et dérivable : x → f(x)

On construit la suite :

Xn telle que :



- On part d'une valeur initiale X0 quelconque
- $X_{n+1} = X_n - \varepsilon * f'(Xn)$, avec ε valeur réelle non nulle « bien choisie » entre 0 et 1

Optimisation : gradient

Remarques :

- Le choix de ε est empirique
- Si ε trop petit : le nombre d'itérations est trop grand
- Si ε est trop grand : les valeurs de la suite risquent d'osciller ⇒ pas de convergence
- Rien ne garantit que le minimum trouvé est un minimum global

Optimisation : gradient

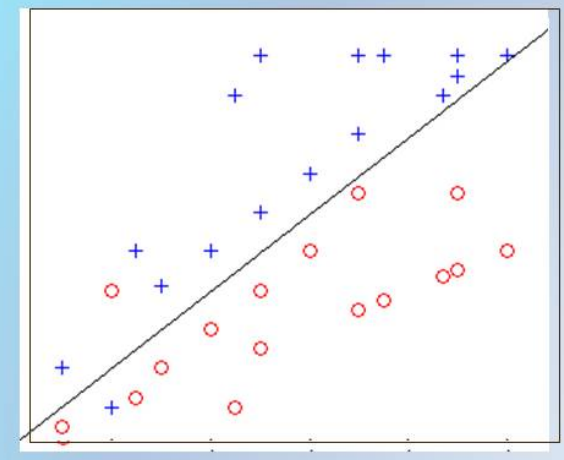
Dans la pratique :

- Critères d'arrêt :
  - nombre d'itérations max.
  - norme(f(Xi+1)−f(Xi)) / norme (Xi+1 − Xi) inférieure à η (valeur très petite)
- ε est réajusté à chaque itération
  - valeur initiale ε_0 (0.01 par exemple)
  - 1ère stratégie :
    - si f(Xi+1) > f(Xi) i.e. f augmente, on augmente ε de 10%
    - Sinon i.e. f diminue, on diminue ε en le divisant par 2
  - 2ème stratégie : on diminue la valeur de ε toutes les K itérations

EXERCICE : perceptron

Classificateur linéaire :

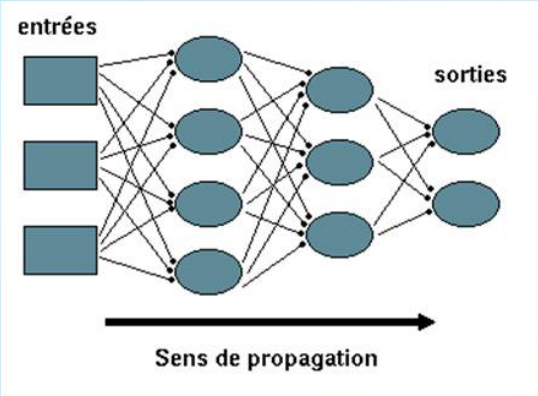
Trouver la droite qui sépare deux groupes de points.



PERCEPTRON MULTI-COUCHES

Les différentes couches :

- Cellules réparties dans q couches : C0, C1, …, Cq
- C0 : couche d 'entrée (la rétine) ⇒ les variables d 'entrée
- Cq : couche de sortie
- C1, …, Cq-1 : les couches cachées



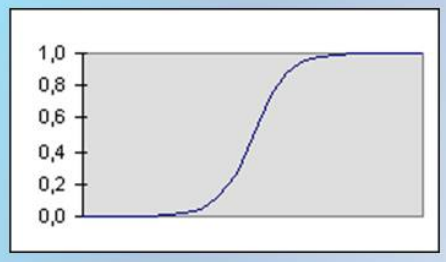
PERCEPTRON MULTI-COUCHES

Fonction de transfert :

- fonction sigmoïde logsig :

$$\sigma_k(x) = e^{kx}/(e^{kx}+1) = 1/(1+e^{-kx})$$

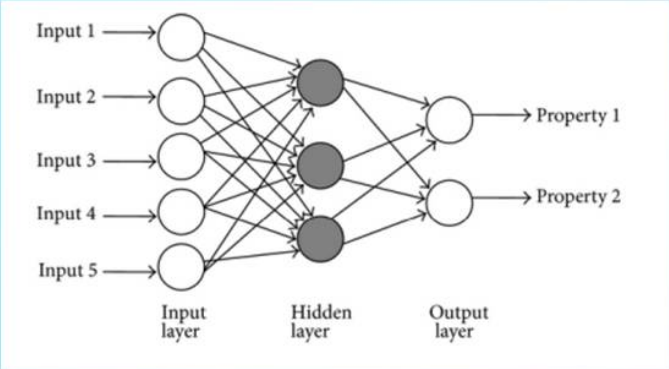
- k = 1 : $\sigma(x) = e^{x}/(e^{x}+1) = 1/(1+e^{-x})$



Dérivée :

$$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$$

Le perceptron multi-couches (PMC ou MLP)



Definition

- Au moins une couche avec fonction d'activation non-lineaire
- Les neurones de la couche i servent d'entrees aux neurones de la couche i + 1
- Neurones d'entrees, neurones de sortie, neurones cachés
- Genéralement : tous les neurones d'une couche sont connectés a tous les neurones des couches précédentes et suivantes
- Toutes les connections i → j sont ponderées par le poids wij

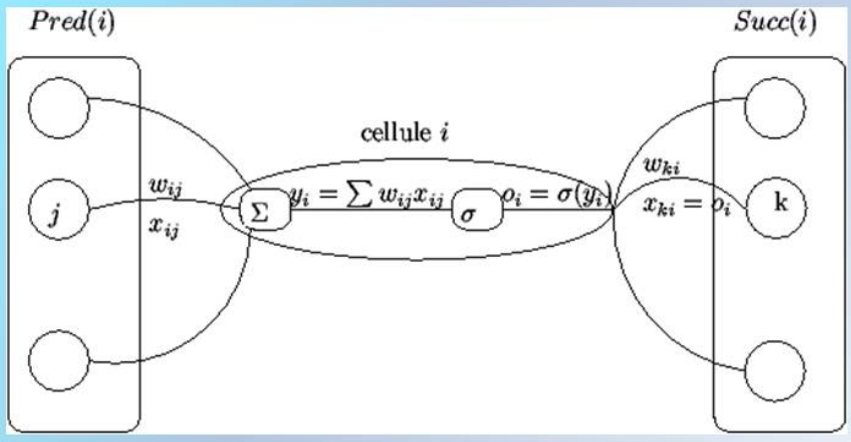
PERCEPTRON MULTI-COUCHES

Notations :

- n cellules
- Cellules désignées par un indice i, $0 \leq i < n$
- p cellules de sortie
- k indice d 'une cellule de sortie
- ck : sortie attendue pour la cellule de sortie k avec l 'entrée x
- ok : sortie calculée pour la cellule de sortie k avec l 'entrée x
- xij : entrée associée au lien entre cellule i vers cellule j
- wij : coefficient synaptique associé au lien entre cellule i vers cellule j
- Succ(i) : ensemble des cellules qui prennent comme entrée la sortie de la cellule i.
- Pred(i) : ensemble des cellules dont la sortie est une entrée de la cellule i.
- yi : entrée totale de la cellule i : $yi = \Sigma(j \in Pred(i))(wij * xij)$
- oi : sortie de la cellule i : $oi = \sigma(yi)$

PERCEPTRON MULTI-COUCHES

Notations :



PERCEPTRON MULTI-COUCHES

Algorithme de rétropropagation du gradient :

- Initialiser aléatoirement les coefficients wij dans [-0.5 ; 0.5]
- Répéter
  - Prendre un exemple (x, c) de S
  - Calculer la sortie o
  - Pour toute cellule de sortie i
    - $\delta i = \sigma'(yi) * (ci - oi) = oi * (1 - oi) * (ci - oi)$
  - Fin Pour
  - Pour chaque couche de q – 1 à 1
    - Pour chaque cellule i de la couche courante
      - $\delta i = \sigma'(yi) * [\Sigma(k \in Succ(i)) (\delta k * wki)]$
        $= oi * (1 - oi) * [\Sigma(k \in Succ(i)) (\delta k * wki)]$
    - Fin Pour
  - Fin Pour
  - Pour tout poids wij
    - $wij = wij + \varepsilon * \delta i * xij$
  - Fin Pour
- Fin Répéter

Récapitulatif

1. Donnez l'algorithme d'apprentissage par la méthode de Perceptron ?
2. Donnez les deux types de neurone formel
3. Quelle est le formalisme de calcul de l'ADALINE αLMS et /αLMS
4. Quelle est la méthode du gradient ?
5. Donnez la règle de Delta généralisée
6. Donnez l'algorithme d'apprentissage par descente de gradient
7. Donnez l'algorithme d'apprentissage par la méthode de Widrow-Hoff (Régle Delta)
8. Donnez le schéma du Perceptron Multi-couche
9. Donnez l'Algorithme de rétropropagation du gradient

PERCEPTRON MULTI-COUCHES

Ajout d 'un paramètre inertiel (momentum) β: éviter les oscilations

- Initialiser aléatoirement les coefficients wi dans [-0.5 ; 0.5]
- Répéter
  - Prendre un exemple (x, c) de S
  - Calculer la sortie o
  - Pour toute cellule de sortie i
    - $\delta i = \sigma'(yi) * (ci - oi) = oi * (1 - oi) * (ci - oi)$
  - Fin Pour
  - Pour chaque couche de q − 1 à 1
    - Pour chaque cellule i de la couche courante
      - $\delta i = \sigma'(yi) * [\Sigma(k \in Succ(i)) (\delta k * wki)]$
        $= oi * (1 - oi) * [\Sigma(k \in Succ(i)) (\delta k * wki)]$
    - Fin Pour
  - Fin Pour
  - Pour tout poids wij à litération k
    - $wij(k) = wij(k-1) + \varepsilon * \delta i * xij + \beta * (wij(k-1) - wij(k-2))$
  - Fin Pour
- Fin Répéter

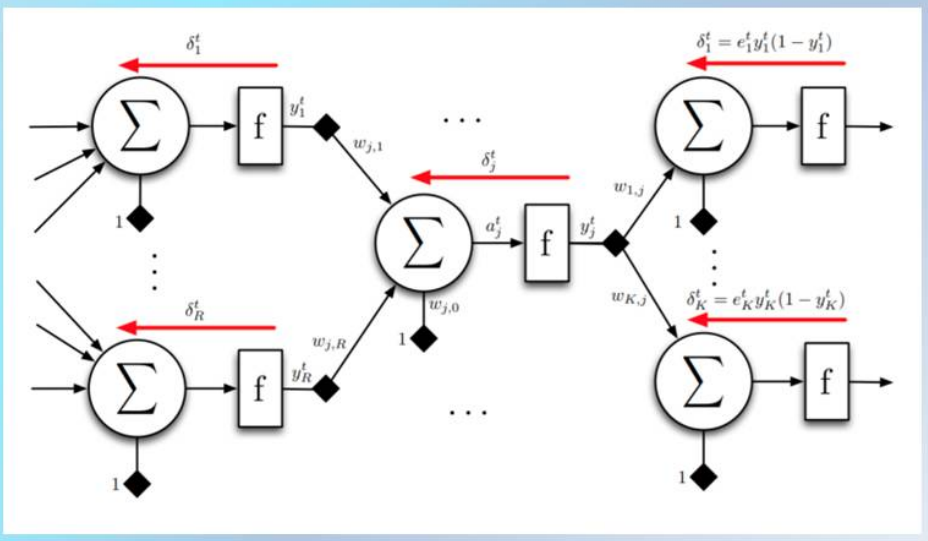
PERCEPTRON MULTI-COUCHES

Remarques :

Perceptron Multi-Couches
= généralisation du perceptron et de la règle delta.

- Principe similaire à l'apprentissage du perceptron simple
- ajout d'un mécanisme de rétropropagation de l'erreur permet d'ajuster les poids en faisant remonter l'erreur des sorties vers les différentes couches
- 2 phases s'alternent jusqu'a la convergence
  1 **Feed-forward :** calcul des sorties, les signaux transitent par les différentes couches
  2 **Backpropagation** : les sorties obtenues sont comparées aux sorties attendues une erreur est calculée pour chaque sortie les poids des liens ayant contribué a chaque erreur sont ajustés de la fin vers le début
- on cherche à minimiser les erreurs au fur et à mesure (descente de gradient)

Rétro-propagation du gradient



En se basant sur l'idée précédent on va se servir d'une fonction de coût, qui va représenter la perte entre la sortie du réseau et les prédictions. On considérera alors que le réseau apprend si le coût diminue (en respectant les poids).
Il n'est pas possible de minimiser le coût analytiquement, on va donc se servir du gradient

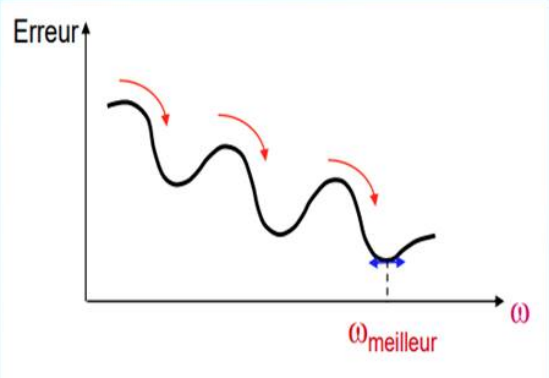
Rétro-propagation du gradient

But de la correction :

recherche du minimum global du réseau

– Les sauts et les bosses correspondent aux améliorations successives faites à chaque étape sur l'ensemble des données

– À chaque fois, on essaie de se sortir d'un minimum local pour atteindre le meilleur vecteur poids



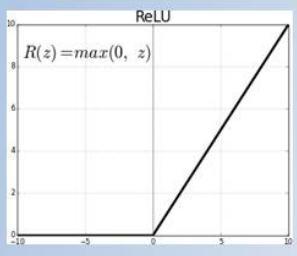
Neurone et fonction de transfert ReLu

Chaque neurone calcule son état :

$$y = f(W.X) = f(\sum_i w_{ix_i})$$

où f est une fonction linéaire rectifiée (ReLu) :

$$f(s) = \begin{cases} s \text{ si } s > 0 \\ 0 \text{ sinon} \end{cases}$$



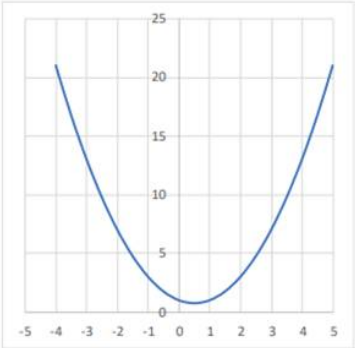
Apprentissage par descente du gradient

- Position du problème

Ex. f(x) = x² - x + 1 ; à minimiser par rapport à x

$$\min_x f(x) = x^2 - x + 1$$

f() est la fonction à minimiser
x joue le rôle de paramètre ici c.-à-d. on recherche la valeur de x qui minimise f()



La solution analytique passe par :

$$f'(x) = 0$$

En s'assurant que $f''(x) > 0$

$$f'(x) = 2x - 1 = 0 \Rightarrow x^* = \frac{1}{2}$$

Apprentissage par descente du gradient

Apprentissage et optimisation

L'apprentissage consiste à trouver les bons paramètres w_i de toutes les connexions.
On cherche à minimer le *coût* (loss) *L* sur les exemples.

$$w = w - \alpha \frac{\partial loss}{\partial w}$$

Exemples de coûts :

Quadratique (MSE (mean squared error)) :

- Loss function :

$$L(\theta) = f\left(\hat{Y}(\theta), Y\right)$$

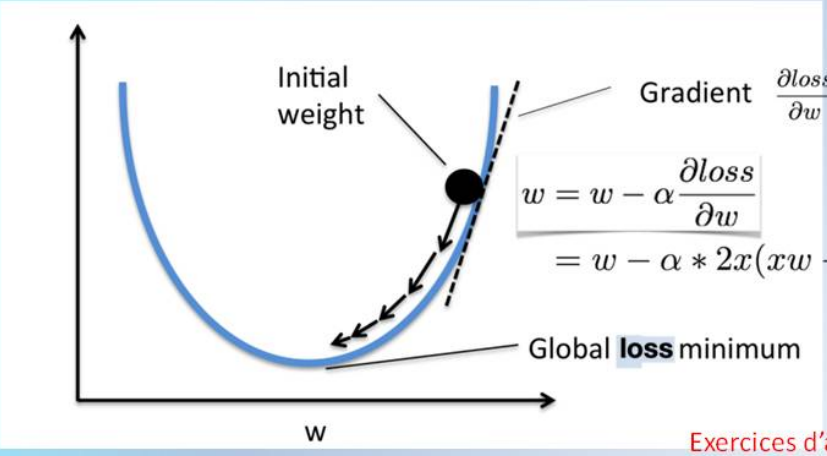
- Exemples de fonctions de coût :
  - Distance euclidienne (au carré) : $f\left(\hat{Y}(\theta), Y\right) = \left\|\hat{Y}(\theta) - Y\right\|_2^2 = \sum_{ij}\left(\hat{y}_{ij}(\theta) - y_{ij}(\theta)\right)^2$
  - Binary cross-entropy, pour la classification 0 ou 1 : $f\left(\hat{Y}(\theta), Y\right) = \sum_{ij}\left(y_{ij}\ln\left(\hat{y}_{ij}(\theta)\right) + \left(1 - y_{ij}\right)\ln\left(1 - \hat{y}_{ij}(\theta)\right)\right)$
  - Distance en norme 1 : $f\left(\hat{Y}(\theta), Y\right) = \left\|\hat{Y}(\theta) - Y\right\|_1 = \sum_{ij}\left|\hat{y}_{ij}(\theta) - y_{ij}(\theta)\right|$
  - Et d'autres...

Exemple : optimisation MSE modèle linéaire

$$\hat{y} = w.x$$

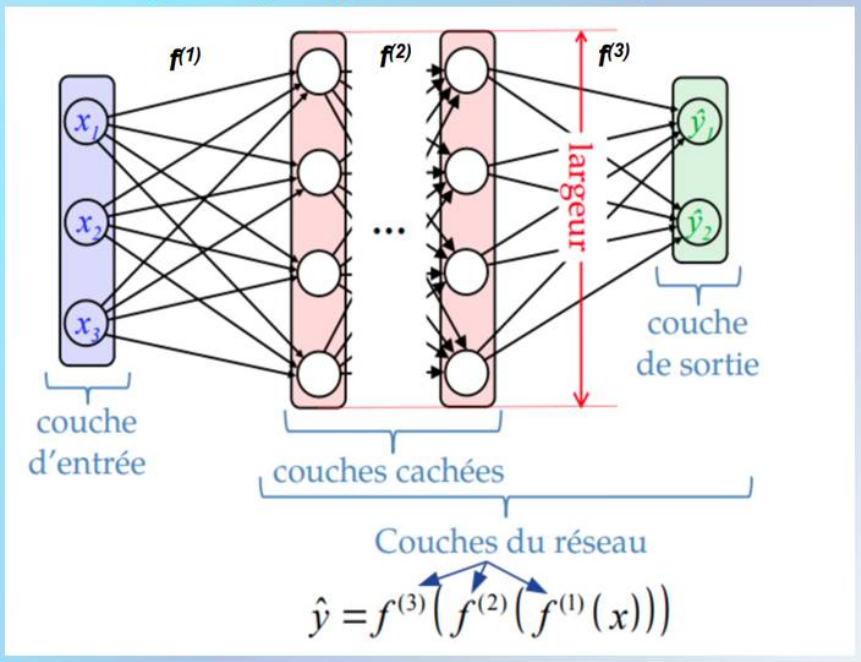
coût = $(\hat{y} - y)^2$

$$w = w - \alpha \frac{\partial loss}{\partial w}$$



Exercices d'application

Rétro-propagation du gradient



Présentation PyTorch

PyTorch

- PyTorch est un framework de deep learning relativement récent basé sur Torch. Développé par le groupe de recherche AI de Facebook et mis en open-source sur GitHub en 2017, il est utilisé pour des applications de traitement du langage naturel (NLP). PyTorch est réputé pour sa simplicité, sa facilité d'utilisation, sa flexibilité, son utilisation efficace de la mémoire et ses graphes de calcul dynamiques. Il se sent également natif, ce qui rend le codage plus gérable et augmente la vitesse de traitement.manipuler des tenseurs (tableaux multidimensionnels), de les échanger facilement avec Numpy et d'effectuer des calculs efficaces sur CPU ou GPU (par exemple, des produits de matrices ou des convolutions);
- calculer des gradients pour appliquer facilement des algorithmes d'optimisation par descente de gradient. PyTorch utilise la bibliothèque Autograd(La bibliothèque Autograd permet le calcul automatique de gradients d'une fonction par rapport à ses paramètres).

Présentation PyTorch

PyTorch

- **Les tenseurs**

Un tenseur est une structure de données importante dans PyTorch et également dans d'autres frameworks d'apprentissage profond tels que Tensorflow. C'est une généralisation d'un vecteur ou d'une matrice à un nombre arbitraire de dimensions. L'avantage d'un tenseur par rapport à un tableau NumPy (NumPy array) est qu'il sert à effectuer des opérations très rapides sur des GPU, à distribuer le calcul sur plusieurs machines et à garder trace des opérations qui l'ont créé.

- Vous pouvez commencer par vous-même à manipuler des tenseurs. Voici des références qui vous seront utiles :
- Tutoriel sur les tenseurs :

http://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html.

Exercice

Soit le modèle

$$\hat{y} = x^2.w_2 + x.w_1 + b$$
$$\text{loss} = (\hat{y} - y)^2$$

1. Calculer (sachant que b = 0) :

$$\frac{\partial loss}{w_1} = ?$$

$$\frac{\partial loss}{w_2} = ?$$

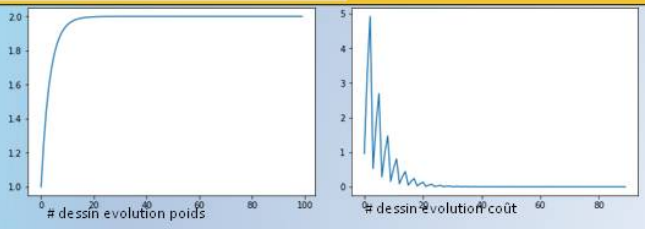
2. Coder ce calcul en Python.
3. Faire le même calcul avec PyTorch.

TP Pytorch : https://ledatascientist.com/debuter-avec-pytorch/
(conda install pytorch torchvision -c pytorch)

Exemple : optimisation MSE modèle linéaire

```python
# Données: entrées x (valeurs scalaires), sorties désirées y
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0  # poids (paramètre) initial, au hasard

# modèle: calcul de la sortie ("forward pass")
def forward(x):
    return x * w


# Fonction de coût
def cout(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)


# calcul du gradient du cout par rapport au poids
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```

```python
# Before training
print("prévision avant apprentissage:",  4, forward(4))

# Boucle d'apprentissage
for epoch in range(100):
    for x_val, y_val in zip(x_data, y_data):
        grad = gradient(x_val, y_val)
        w = w - 0.01 * grad
        print("\tgrad: ", x_val, y_val, grad)
        l = cout(x_val, y_val)

    print("progress:", epoch, "w=", w, "loss=", l)

# After training
print("predict (after training)",  "4 hours", forward(4))
```



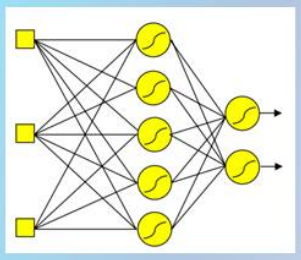
voir 01-exemple-1d.ipynb

PERCEPTRON MULTI-COUCHES

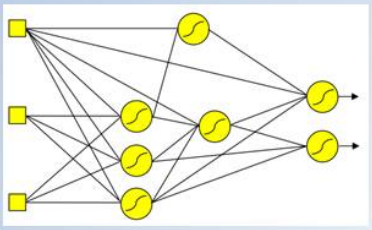
Remarque :

Le nombre de couches cachées et le nombre de neurones par couche ont une influence sur la qualité de l'apprentissage.

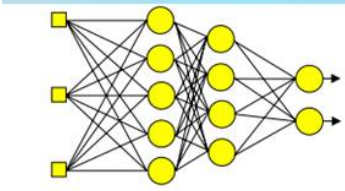
Exemple d'un réseau multicouche complètement connecté



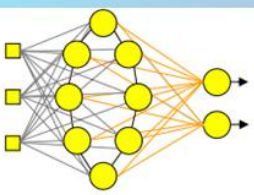
Exemple d'un réseau non ordonné et partiellement connecté



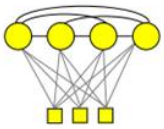
Différentes architectures neuronales



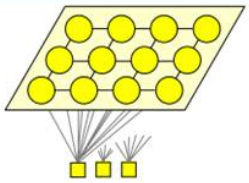
multilayer feedforward network



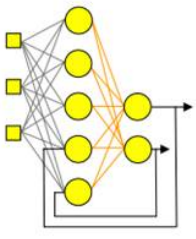
fully recurrent network



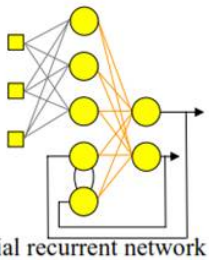
competitive network



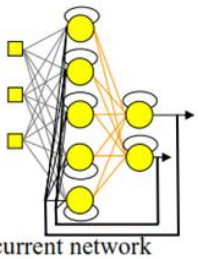
self-organizing map



Jordan network



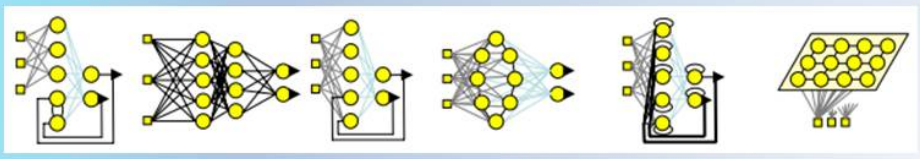
partial recurrent network with dual connections



recurrent network with self connections

Stratégies d'apprentissage

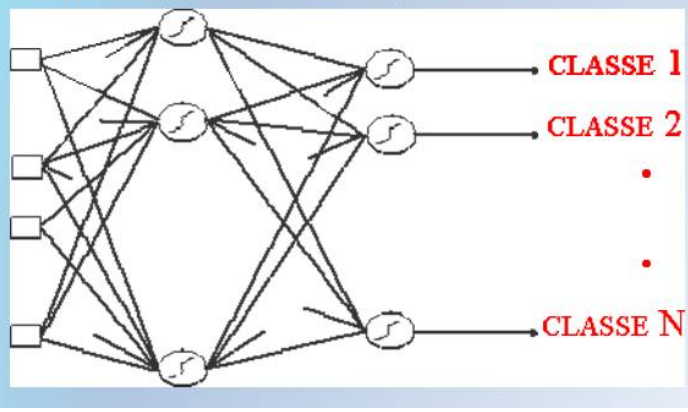
- L'apprentissage au sein des différentes architectures dépend de l'architecture du réseau et de l'environnement du problème.
- Les 2 règles d'apprentissage pour mettre corriger les poids d'un neurone (règle de Hebb et de Widrow) ne concernent qu'un neurone seul.
- Ces règles peuvent servir pour mettre à jour les poids d'un neurone, de certains réseaux de neurones, mais ne peuvent être généralisées et s'appliquer à n'importe quelle architecture.
- Chaque architecture possède ses spécificités et nécessite une règle d'adaptation des poids qui lui est propre.



PERCEPTRON MULTI-COUCHES

Classification / Discrimination :

- 1-ère modélisation :
  - chaque neurone de sortie indique la probabilité d'appartenance à la classe correspondante
  - l'exemple fourni en entrée appartient à la classe pour laquelle la probabilité d'appartenance est la plus forte



PERCEPTRON MULTI-COUCHES

Classification / Discrimination :

- Inconvénients :
  - apprentissage très long (très nombreuses connexions)
  - une nouvelle classe : il faut tout réapprendre !

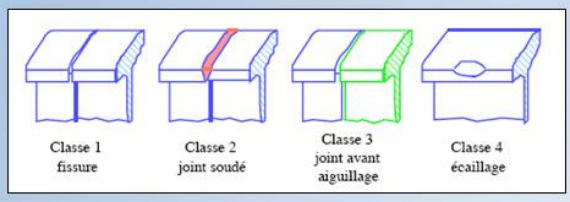
- 2-ème modélisation : un réseau par classe
  - pour chaque réseau, une sortie indique si l'exemple soumis appartient à la classe correspondante

PERCEPTRON MULTI-COUCHES

Classification / Discrimination :

Applications :

- Reconnaissance de formes
  - codespostaux, signatures, visages,
  - défauts de rails



- Traitement du signal : reconnaissance de la parole
- Informatique décisionnelle (à quelle classe de consommateurs appartient un individu ?)
- …

PERCEPTRON MULTI-COUCHES

Régression (Approximation de fonction) :

On donne x en entrée.
Le réseau calcule f(x) en sortie

- Réseaux de neurones à 2 couches (1 couche cachée)
- Fonctions de transfert « logsig » pour la couche cachée
- Fonctions de transfert « purelin » pour la couche de sortie
- Perceptron Multi-Couches = ***approximateur universel***.

  ⇒ Permet d'approximer n'importe quelle fonction, à condition que le nombre de neurones dans les couches cachées soit suffisant !

PERCEPTRON MULTI-COUCHES

Prévision :

On donne K valeurs successives de f en entrée.
Le réseau calcule la valeur suivante en sortie

Même type de réseau que pour la régression.

Récapitulatif

1. Quelle est la généralisation que le perceptron multicouche introduit ?
2. Sur quoi agit le nombre des couches cachés
3. Quels sont les inconvénients de la méthode du perceptron multicouche
4. Quels sont les applications de la méthode du perceptron multicouche
5. Quel est le but de la rétro-propagation du gradient
6. Donnez la formule de calcul de la fonction d'activation RELU
7. Donnez la formule de calcul du MSE
8. Donnez la formule de calcul du poids w par Descente de gradient dans le cas ou la fonction de sortie est f(x) = wx
9. Définir le framework Pytorch
10. Définir la notion tenseur en Pytorch

PERCEPTRON MULTI-COUCHES

Régression / Prévision :

Applications :

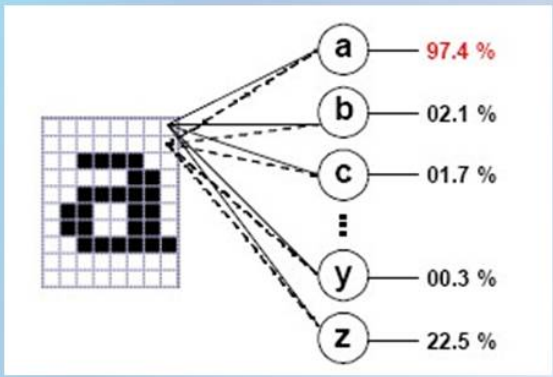
- approximation de fonctions « compliquées »
- régression à partir de données expérimentales (relevés ou mesures expérimentales)
- prévisions météorologiques
- prévisions financières
- …

EXERCICES : PMC

Classification :

Reconnaissance de formes :

- les lettres de l 'alphabet
- les chiffres de 0 à 9

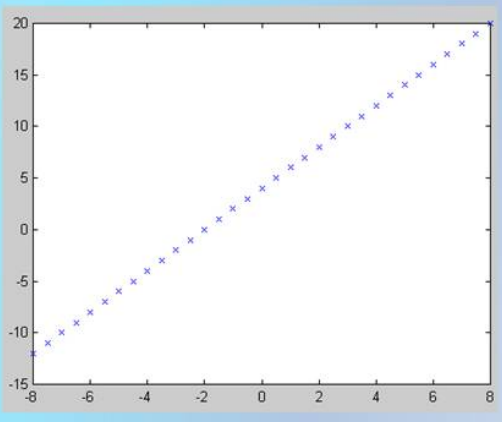


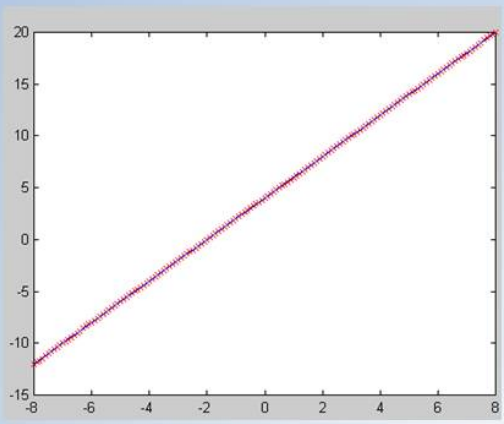
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*x + 4 sur [-8 ; 8]



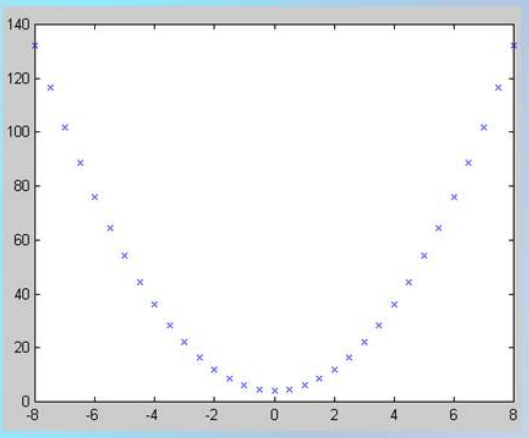


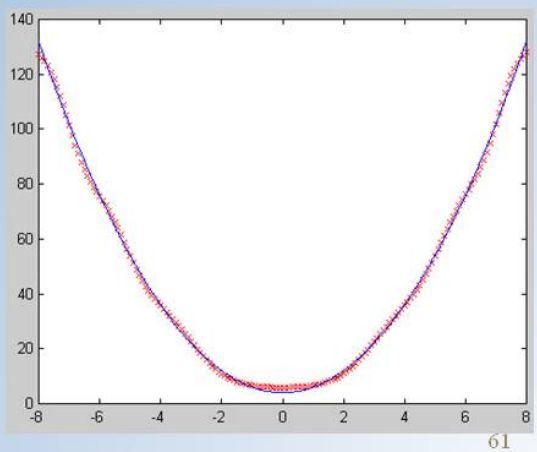
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*x^2 + 4 sur [-8 ; 8]



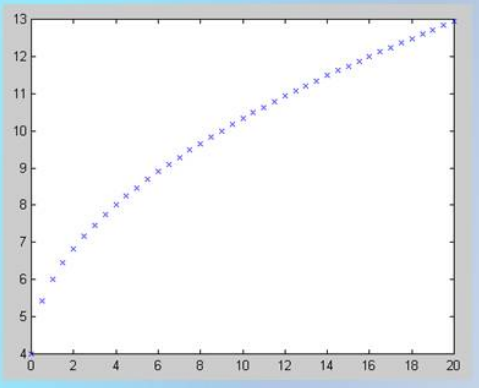


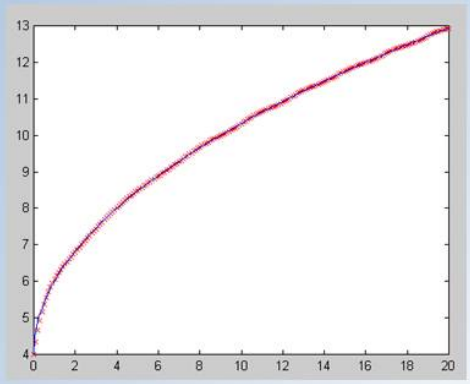
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*√x + 4 sur [0 ; 20]



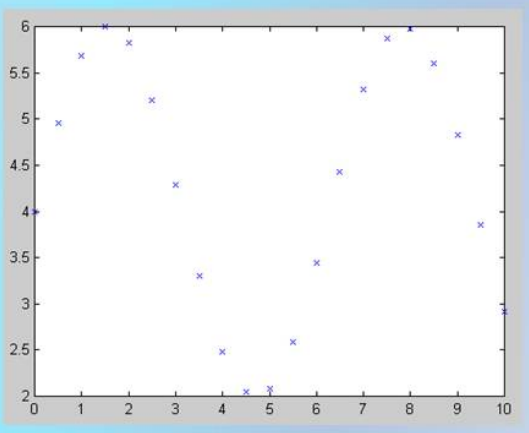


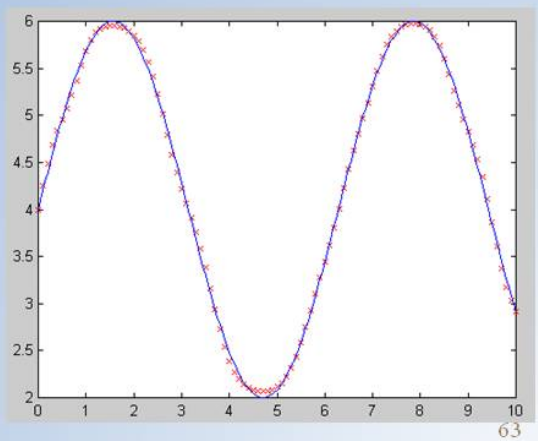
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*sin(x)+ 4 sur [0 ; 10]



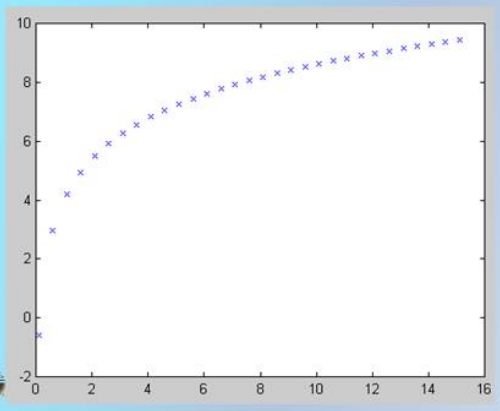


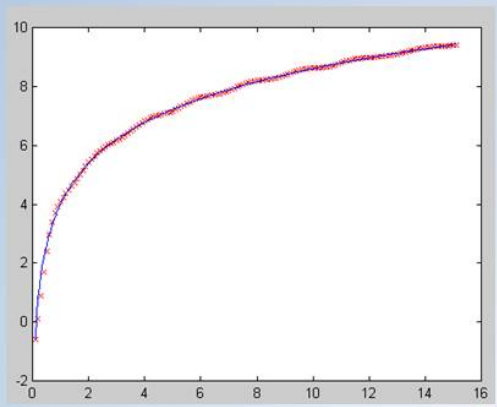
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*ln(x)+ 4 sur [0.1 ; 15.1]



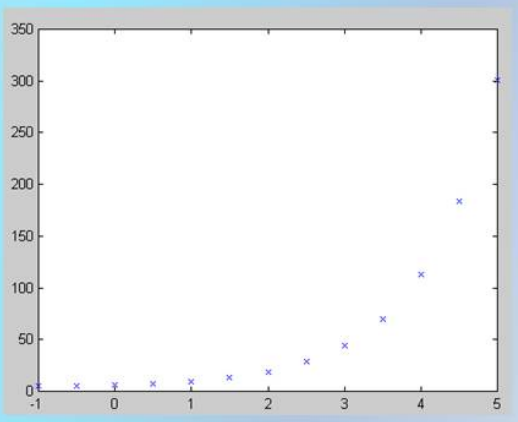


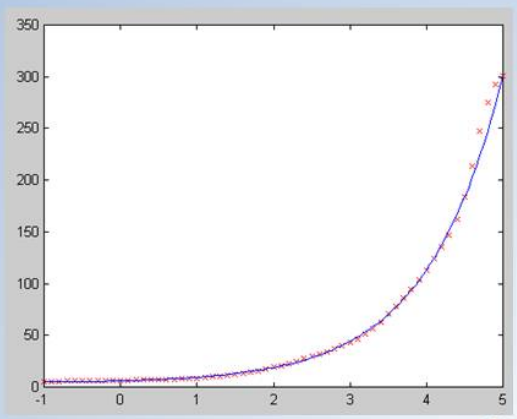
EXERCICES : PMC

Régression :

Approximation de fonctions :

- y = 2*exp(x)+ 4 sur [-1 ; 5]





EXERCICE : MLP

L'objectif de cet exercice est de simuler la règle de propagation dans un perceptron multicouche. Considérons le PMC a deux couches de la figure. La fonction d'activation considérée est la fonction sigmoïde. Les neurones de la couche d'entrée sont numérotes par x1 (pour 1) et x2 (pour 2). Les neurones de la couche cachée sont numérotés par 3 et 4 et le neurone de la couche de sortie par 5. La fonction d'activation considérée est la fonction sigmoïde et les poids synaptiques étant fixés selon le tableau suivant :

w0,3 = 0.2	w1,3 = 0.1	w2,4 = 0.4
w0,4 = –0.3	w1,4 = –0.2	w3,5 = 0.5
w0,5 = 0.4	w2,3 = 0.3	w4,5 = –0.4

Soit le vecteur d'entrée x = [1; 1], calculer la valeur de la sortie du cinquième neurones selon la méthode du perceptron multicouche ?

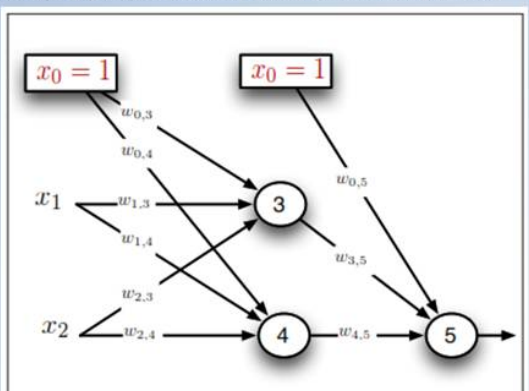


Figure : Architecture d'un perceptron multicouche.

Réponse : la transmission de l'information dans chacun de neurone se fait comme suit :

Le neurone 3 :

$$\sigma_3 = w_{0,3} + \sum_{j=1}^{2} w_{j,3} x_j = w_{0,3} + w_{1,3} \times x_1 + w_{2,3} \times x_2 = 0.1 + 0.2 + 0.3 = 0.6$$

La sortie du neurone 3 est alors :

$$a_3 = f(\sigma_3) = \frac{1}{1 + e^{-0.6}} = 0.65$$

Le neurone 4 :

$$\sigma_4 = w_{0,4} + \sum_{j=1}^{2} w_{j,4} y_j = w_{0,4} + w_{1,4} \times x_1 + w_{2,4} \times x_2 = -0.3 - 0.2 \times 1 + 0.4 \times 1 = -0.1$$

La sortie du neurone 3 est alors :

$$a_4 = f(\sigma_4) = \frac{1}{1 + e^{0.1}} = 0.48$$

Le neurone 5 :

Les entrées du neurone 5 sont les sorties des neurones 2 et 3. Il faut donc faire attention aux appellations $\mathbf{x}_i$ et y_i puisque les entrées d'une couche cachée sont ou bien les données d'entrées (souvent notées $\mathbf{x}_i$) ou bien les données de sorties de la couche précédente (souvent notées $\mathbf{y}_i$).

$$\sigma_5 = w_{0,5} + \sum_{j=3}^{4} w_{j,5} x_j = w_{0,5} + w_{3,5} \times y_3 + w_{4,5} \times y_4 = 0.4 + 0.5 \times 0.65 - 0.4 \times 0.48 = 0.53$$

La sortie du neurone 5 est alors :

$$a_5 = f(\sigma_3) = \frac{1}{1 + e^{-0.53}} = 0.63$$

En conclusion, pour une entrée $\mathbf{x} = [1, 1]$, la propagation se fait selon :

$$\begin{cases} a_3 = 0.65 \\ a_4 = 0.48 \\ a_5 = 0.63 \end{cases}$$

Autres TD !

CARTE DE KOHONEN

L'origine :

Les neurones sont modélisés de la façon la plus simple possible, on recherche ici un modèle de neurone proche de la réalité. Ces réseaux sont inspirés des observations biologiques du fonctionnement des systèmes nerveux de perception des mammifères. En effet, il existe des zones du cerveau (dans le cortex visuel par exemple) qui présentent la même topologie que les capteurs sensoriels. C'est à dire deux zones proches dans le cortex visuel correspondent à deux zones proches dans la rétine. Ces propriétés se retrouvent avec le bulbe olfactif, ou l'appareil auditif. Il est intéressant de souligner que ces dispositions au sein du cerveau ne sont pas génétiques, mais sont dues à un apprentissage. A la suite de ces observations, Kohonen proposa un modèle de carte topologique auto-adaptative. Seules les entrées modifient le processus, l'apprentissage est donc non-supervisé.

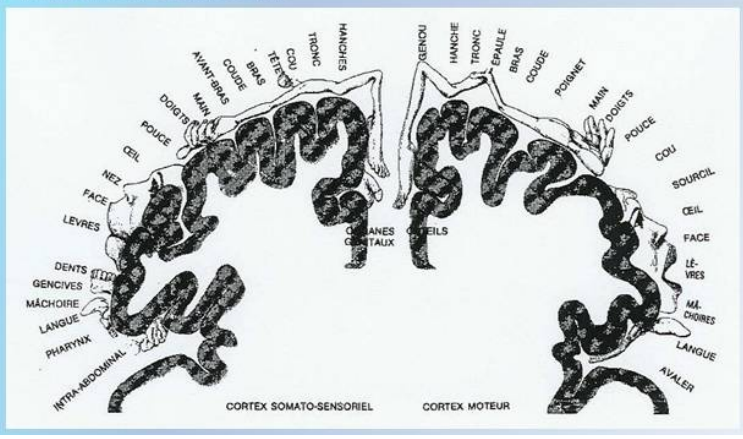
CARTE DE KOHONEN

• Réseaux de Neurones à ***Compétition*** : un seul neurone de sortie est activé pour une entrée donnée.

• Mise en correspondance de l'espace d'entrée avec l'espace du réseau.

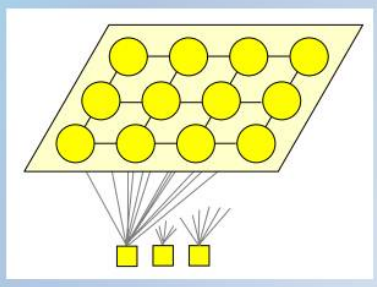
⇒ ***Auto-organisation***

• ***Origine biologique*** : une zone du cortex correspond à une zone des organes sensoriels et moteurs



Apprentissage d'une carte auto-organisatrice

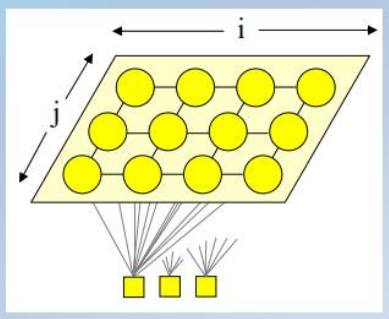
L'auto-organisation d'une carte de Kohonen est un apprentissage non supervisé (SOM : Self-Organizing Map).
Les neurones d'une carte de Kohonen sont organisés dans une seule couche qui peut être considérée comme une grille multidimensionnelle.



L'exemple ci-contre comprend :
1 couche = 1 grille bidimensionnelle de 3x4 neurones
3 entrées

Apprentissage d'une carte auto-organisatrice

Principe de l'auto-organisation :



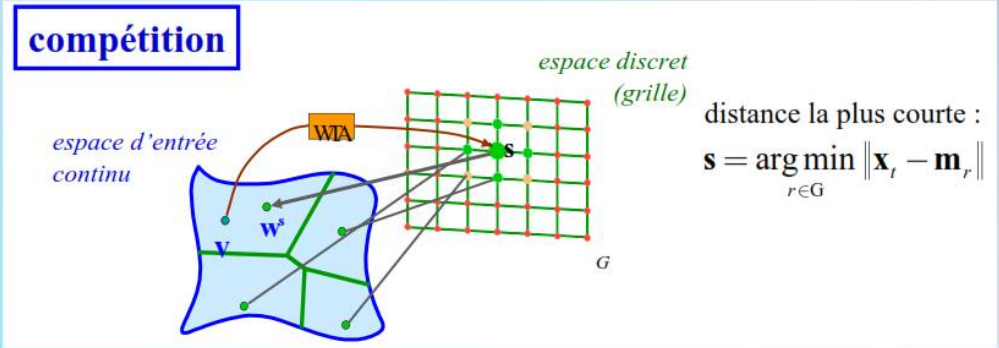
Il n'y a pas de couche de sortie, pas plus que des neurones de sortie.
La position (indices i et j) d'un neurone au sein de la grille est fondamentale et permet de définir la notion de voisinage.
Chaque neurone possède des poids qui définissent sa position dans l'espace des entrées.
L'apprentissage s'effectue en 2 étapes :

- la compétition entre les neurones,
- l'adaptation des neurones.

Apprentissage d'une carte auto-organisatrice

Principe de l'auto-organisation : la compétition

compétition



distance la plus courte :

$$\mathbf{s} = \arg\min_{r \in G} \|\mathbf{x}_t - \mathbf{m}_r\|$$

Lorsqu'on présente un vecteur d'entrée, on cherche le neurone le plus proche.
Pour cela on calcule la distance entre l'entrée X_t et les poids des neurones $\mathbf{m}_r$

- $r=\{i,j\} \in G$ est le vecteur d'indice des neurones
- $s \in G$ est l'indice du neurone vainqueur

Apprentissage d'une carte auto-organisatrice

Principe de l'auto-organisation : l'adapation

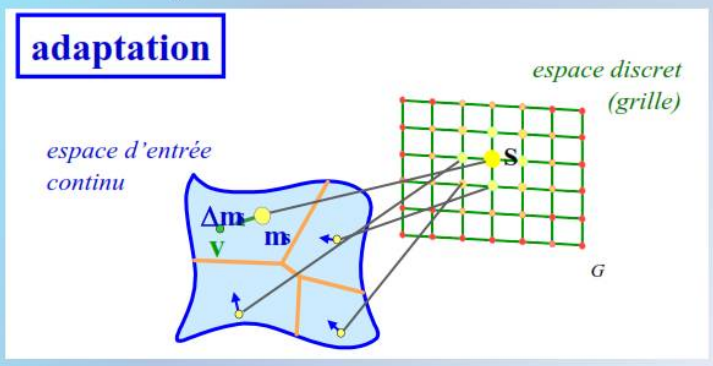
Les poids du neurone vainqueur et de ses voisins sont adaptés :

$\Delta \boldsymbol{m}_r = \mu h_{rs}(x - \boldsymbol{m}_r)$

μ est un coefficient d'apprentissage

$\boldsymbol{m}_r$ est le vecteur poids des neurones

h_{rs} est une fonction de voisinage



CARTE DE KOHONEN

- Applications :

  ⇒ robotique : pilotage de robots

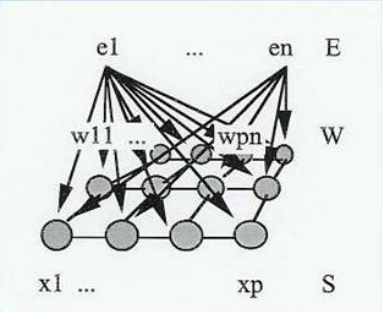
  ⇒ compression de données

  ⇒ statistiques : méthode semblable à l'Analyse en Composantes Principales (PCA)

CARTE DE KOHONEN

Principes (en 2D) :

- n cellules d'entrée e = (e1, …, en)
- une ***carte*** : réseau de m neurones de sortie x1, …, xm
- connexions latérales (coefficients fixes) entre les neurones de sortie : un neurone est connecté à ses 4 plus proches voisins
- connexions de coefficient wij entre une cellule d'entrée ei et un neurone de sortie xj



CARTE DE KOHONEN

Principes :

Pour une entrée, un seul neurone sur la carte est sélectionné (valeur 1).

On encourage le vainqueur : « the winner takes all ».

Ce neurone correspond le plus possible à l'entrée : ***minimisation d'une distance***.

Remarques :

En pratique, on normalise les entrées.

CARTE DE KOHONEN

Algorithme d'apprentissage :

- Initialiser aléatoirement les coefficients wij
- Répéter
  - Prendre une entrée e = (e1, …, ei, …, en)
  - Calculer la distance dj de chaque neurone xj par rapport à e

$$d_j = \sqrt{\sum_{i=1}^{n} (e_i - w_{ij})^2}$$

  - Sélectionner le neurone xk le plus proche de e : dk = Min(dj)
  - Modifier les coefficients pour le neurone sélectionné et ses plus proches voisins (4 pour une carte 2D) :
    - Pour tout i :
      - wik = wik + μ * (ej – wik)
      - wil = wil + β * (ej – wil) où xl est un voisin de xk
    - Fin Pour
- Fin Répéter

CARTE DE KOHONEN

Algorithme d'utilisation :

La sortie sk du neurone xk, pour une entrée e, est donnée par :

sk = 1 si dk = Min(di)

sk = 0 si i ≠ k

Différentes types de cartes auto-organisatrices

Il existe différentes cartes auto-organisatrices ou carte topographique (à apprentissage compétitif) :

- la carte auto-organisatrice de Kohonen (1982) à apperntissage non supervisé,
- le neural gas,
- les SVM.

Les machines à vecteurs de support (en anglais Support Vector Machine, SVM) sont un ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de discrimination et de régression. Les SVM sont une généralisation des classifieurs linéaires et ont été développés dans les années 1990.

CARTE DE KOHONEN

Application :

Analyse de données (cf. analyse en composantes principales)

Exemple :

Analyse de données socio-économiques (PIB, croissance du PIB, mortalité infantile, taux d'illettrisme, …) de 52 pays

1 neurone de la carte = 1 groupe de pays (même situation socio-économique)

Neurones voisins = groupes de pays proches (du point de vue de la situation socio-économique)

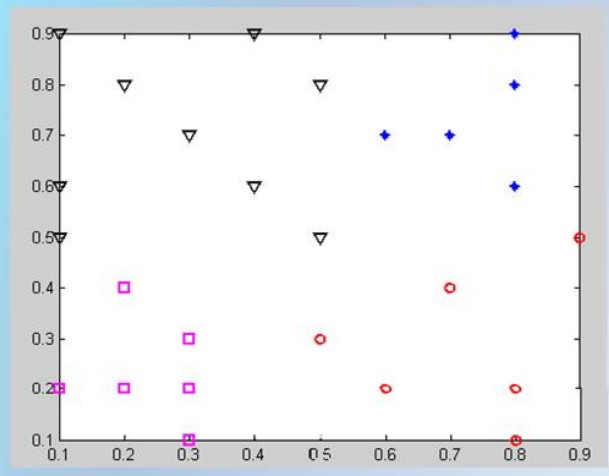
CARTE DE KOHONEN

Suède Suisse	France	Australie	Italie	Yougos-lavie Grèce		Koweit	
RFA	Etats-Unis Japon Canada		Irlande			Bahrein	
Finlande		Espagne		Chili	Mexique	Pérou	Arabie Saoudite
URSS RDA	Royaume Uni		Israël	Véné-zuela	Brésil		
Cuba		Argentine			Mada-gascar	Afrique du Sud	Mozam-bique Nigéria
Corée du Sud	Pologne	Hongrie		Turquie	Maroc Algérie		
Chine			Indonésie			Sénégal	
Vietnam	Nicaragua	Syrie Kénya	Egypte Iran	Cameroun Inde	Ethiopie	Niger	Burkina Faso

EXERCICE : carte de Kohonen

Analyse de données / clustering:

4 classes dans un nuage de points

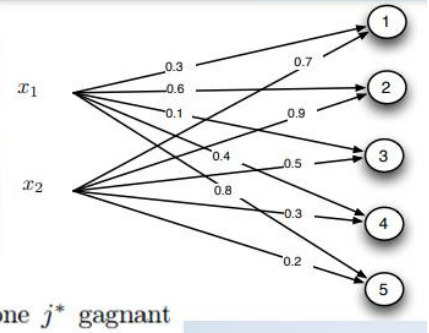


Récapitulatif

1. Que signifie un réseaux de Neurones à ***Compétition***
2. Quelle est l'origine biologique de la carte de Kohonen
3. Quelles sont les étapes de la carte d'auto-organisation
4. Quelles sont les applications de la carte d'auto-organisation
5. Donnez l'algorithme d'apprentissage par la carte d'auto-organisation
6. Donner les différentes types de cartes auto-organisatrices
7. Définir l'algorithme des machines à vecteurs de support SVM

Exercice

L'objectif de cet exercice est de faire l'apprentiissage d'un réseau de Kohonen. Soit la carte de Kohonen à 2 entreès et à 5 sorties j illustrées dans la figure suivante. Soit un vecteur de caractéritique $\mathbf{x} = (x_1, x_2)$ décrivent une forme de dimension 2. La similarité entre $\mathbf{x}$ et le vecteur de poids du neurone j est mesurée par la distance euclidienne au carré d_{ij}^2



1. Qu'est ce qu'un neurone gagnant ?
2. Soit une entrée $\mathbf{x} = (0.5, 0.2)$. Déterminer le neurone j^* gagnant correspondant à cette entrée.
3. En se référant sur l'algorithme d'apprentissage de la mémoire de Kohonen du module 5, calculez la valeur des poids mis à jour de ce neurone. Nous supposons que $\epsilon_n = 0.2$ et σ_n =0.1

Solution

1. Calculer la distance euclidienne entre les entrées et les poids :

$$D1 = \Sigma(X - w1)^2 = (0,5 - 0,3)^2 + (0,2 - 0,7)^2 = 0,29$$
$$D2 = \Sigma(X - w2)^2 = (0,5 - 0,6)^2 + (0,2 - 0,9)^2 = 0,5$$
$$D3 = \Sigma(X - w3)^2 = (0,5 - 0,1)^2 + (0,2 - 0,5)^2 = 0,25$$
$$D4 = \Sigma(X - w4)^2 = (0,5 - 0,4)^2 + (0,2 - 0,3)^2 = 0,02$$
$$D5 = \Sigma(X - w5)^2 = (0,5 - 0,8)^2 + (0,2 - 0,2)^2 = 0,09$$

Alors le neurone gagnant est le neurone numéro 4.

2. La valeur du poids mise à jours pour le neurone numéro 4 est calculé à l'aide de la fonction suivante :

$$w(nouveau) = w(ancien) + a(x + w(ancien))$$

Alors :

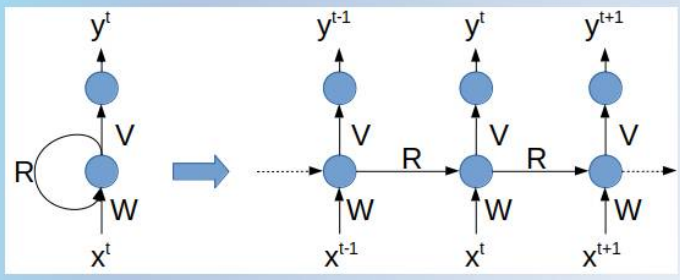
$$w_{41}(nouveau) = 0,4 + 0,2 * (0,5 + 0,4) = 0,58$$
$$w_{42}(nouveau) = 0,3 + 0,2 * (0,2 + 0,3) = 0,4$$

3. Les valeurs mises à jours pour les autres neuronnes :

$$w_{11}(nouveau) = 0,3 + 0,2 * (0,5 + 0,3) = 0,46$$
$$w_{12}(nouveau) = 0,7 + 0,2 * (0,2 + 0,7) = 0,88$$
$$w_{21}(nouveau) = 0,6 + 0,2 * (0,5 + 0,6) = 0,82$$
$$w_{31}(nouveau) = 0,1 + 0,2 * (0,5 + 0,1) = 0,22$$
$$w_{32}(nouveau) = 0,5 + 0,2 * (0,2 + 0,5) = 0,64$$
$$w_{51}(nouveau) = 0,8 + 0,2 * (0,5 + 0,8) = 1,06$$
$$w_{52}(nouveau) = 0,2 + 0,2 * (0,2 + 0,2) = 0,28$$

RESEAUX RECURRENTS / BOUCLES

Un réseau de neurones récurrent (RNN, recurrent neural network) est un type de réseau de neurones artificiels principalement utilisé dans la reconnaissance vocale et le traitement automatique du langage naturel (TAL, NLP, TNL). Les RNN sont conçus de manière à reconnaître les caractéristiques séquentielles et les modèles d'utilisation des données requis pour prédire le scénario suivant le plus probable.
Les RNN sont utilisés dans le cadre de l'apprentissage profond et dans le développement de modèles qui simulent l'activité du système cérébral humain. Ils sont particulièrement puissants dans les scénarios faisant intervenir le contexte dans la prédiction d'un résultat.



RESEAUX RECURRENTS / BOUCLES

1) SYSTEMES DYNAMQUES A TEMPS DISCRET

Équations d'état :

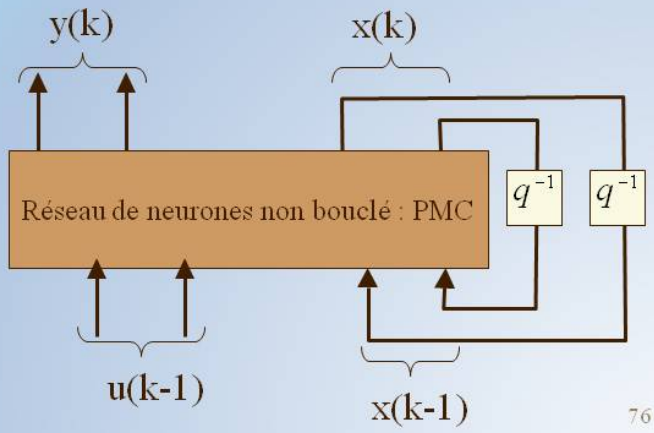
- $\begin{cases} x(k) = \phi[(x(k-1), u(k-1)] \\ y(k) = \varphi[x(k)] \end{cases}$
- Avec u(k) : vecteur des entrées à l'instant k*t
  y(k) : vecteur des sorties à l'instant k*t
  x(k) : vecteur des variables d'état k*t

RESEAUX RECURRENTS / BOUCLES

1) SYSTEMES DYNAMQUES A TEMPS DISCRET

Représentation sous forme de réseau de neurones récurrent ou bouclé :

q^{-1} Symbole représentant un retard d'une unité de temps



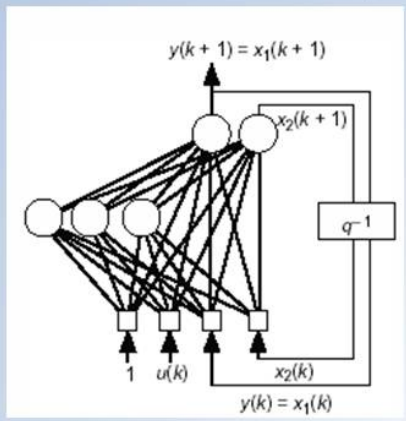
RESEAUX RECURRENTS / BOUCLES

1) SYSTEMES DYNAMQUES A TEMPS DISCRET

Applications :

- Automatique / Robotique : contrôle commande, asservissement

Commande d'un actionneur hydraulique d'un bras de robot



- Séries temporelles

RESEAUX RECURRENTS

RESEAUX DE HOPFIELD

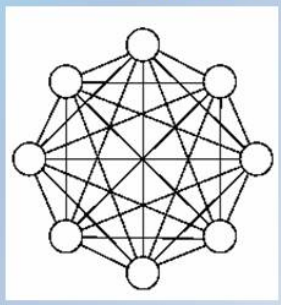
Le réseau de neurones d'Hopfield est un modèle de réseau de neurones récurrents à temps discret dont la matrice des connexions est symétrique et nulle sur la diagonale et où la dynamique est asynchrone (un seul neurone est mis à jour à chaque unité de temps). Il a été découvert par le physicien John Hopfield en 1982. Sa découverte a permis de relancer l'intérêt dans les réseaux de neurones qui s'était essoufflé durant les années 1970 suite à un article de Marvin Minsky et Seymour Papert.

Un réseau de Hopfield est une mémoire adressable par son contenu : une forme mémorisée est retrouvée par une stabilisation du réseau, s'il a été stimulé par une partie adéquate de cette forme.

RESEAUX RECURRENTS

2) RESEAUX DE HOPFIELD

- Réseaux totalement connectés



- Mémoires auto-associatives
  - Associer quelque chose à une suite de données
  - Exemple : on fournit une partie du visage et le on retrouve le visage entier

RESEAUX RECURRENTS / BOUCLES

2) RESEAUX DE HOPFIELD

- Chaque neurone a 2 états possibles : -1 ou +1
- L'activation du neurone i est :
  - $\begin{cases} ai = 1 \text{ si } wi1*y1 + \ldots + wij*yj + \ldots + win*yn > 0 \\ ai = -1 \text{ sinon} \end{cases}$

    La fonction de transfert est la fonction Signe
  - Avec :

    yi état du neurone i

    wij coefficient synaptique entre les neurones i et j

    wij = wji (matrice W symétrique avec diagonale nulle)

RESEAUX RECURRENTS / BOUCLES

2) RESEAUX DE HOPFIELD

- Modèle dynamique :

L'état du neurone i à l'instant t+1 est donné par :

yi(t) = ai(t)

= Signe(wi1*y1(t) + … + wij*yj (t) + … + win*yn (t))

RESEAUX RECURRENTS / BOUCLES

2) RESEAUX DE HOPFIELD

- Apprentissage :
  - On apprend avec une base de :
    - couples {xk ; f(xk)}
    - vecteurs yk
  - Algorithme d'apprentissage : analogue à la loi de Hebb :

$$wij = \frac{1}{N}\sum_{k=1}^{P} y_i^k \times y_j^k$$

N : nombre de neurones (nombre d'entrées)

P : nombre de vecteurs yk

  - Convergence vers des états stables (points fixes) et les bassins d'attraction associés
  - Bassin d'attraction du vecteur y = ensemble des vecteurs initiaux qui conduisent au vecteur y
  - Attention : états stables parasites !

RESEAUX RECURRENTS / BOUCLES

2) RESEAUX DE HOPFIELD

- Utilisation :

  Après présentation des entrées, réinjecter les sorties jusqu'à ce que l'état interne du réseau devienne stable
  - ⑨ On fournit x en entrée et le réseau donne f(x)
  - ⑨ On fournit y incomplet ou bruité et le réseau reconstitue y

Récapitulatif

1. Définir un réseau de neurones récurent
2. Donnez une représentation schématique d'un réseau de neurones récurent
3. Donnez qq applications des RNN
4. Définir un réseau de Hopfields
5. Donnez le modèle dynamique d'un réseau de Hopfield
6. Donnez les caractéristique de l'apprentissage par réseau de Hopfield

RESEAUX EVOLUTIONNAIRES

les *algorithmes évolutionnaires* (AEs) (dont les plus connus sont les *algorithmes génétiques*) s'inspirant de l'évolution darwinienne des populations biologiques.

Le squelette

•La pression de l'environnement" est simulée à l'aide de la fonction d'adaptation *F*, et le principe darwinien *Les plus adaptés survivent et se reproduisent*, est implanté de la manière suivante :**Initialisation** de la *population* Π_0 en choisissant *P* individus dans Ω, généralement par tirage aléatoire avec une probabilité uniforme sur Ω ;

•**Evaluation** des individus de Π_0 (i.e. calcul des valeurs de *F* pour tous les individus) ;

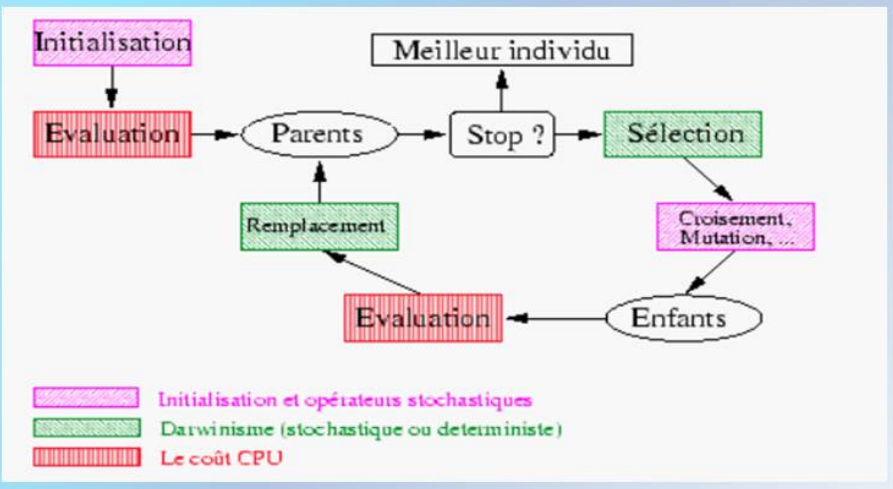
•La génération *i* construit la population Π_i à partir de la population Π_{i-1} :

- **Sélection** des individus les plus performants (au sens de *F*) de Π_{i-1} ;
- Application (avec une probabilité donnée) des **opérateurs génétiques** aux *parents* sélectionnés, ce qui génère de nouveaux individus, les *enfants* ; on parlera de *mutation* pour les opérateurs unaires, et de *croisement* pour les opérateurs binaires (ou n-aires) ;
- **Evaluation** des enfants ;
- **Remplacement** des parents au moyen d'une sélection darwinienne parmi les enfants, avec participation éventuelle des parents.

•

•L'évolution stoppe quand le niveau de performance souhaité est atteint, ou qu'un nombre fixé de générations s'est écoulé sans améliorer l'individu le plus performant.

RESEAUX EVOLUTIONNAIRES



Squelette d'un algorithme évolutionnaire

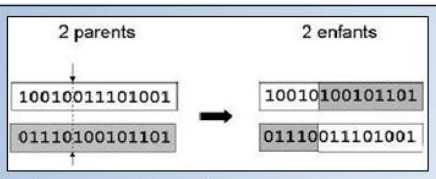
Optimisation : AG (rappels)

Principe des algorithmes génétiques :

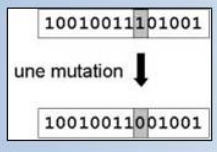
```
Génération aléatoire d'une population initiale
nb_iter = 0
Evaluation de l'adaptation de chaque individu de la population
Tant que (solution non satisfaisante) et (nb_generations < nb_ generations _max)
Faire
        Incrémenter le nombre d'itérations nb_ generations
        Sélectionner les parents P1 et P2 de façon aléatoire
        Appliquer l'opérateur de croisement aux parents P1 et P2
        Faire subir des mutations aléatoires à la population
        Evaluer l'adaptation de chaque individu
        Sélectionner les plus adaptés
Fin Tant que
```

Opérateurs

- croisement : ***exploiter***



- mutation : ***explorer***



RESEAUX EVOLUTIONNAIRES

Exemple 1 :

Problématique :

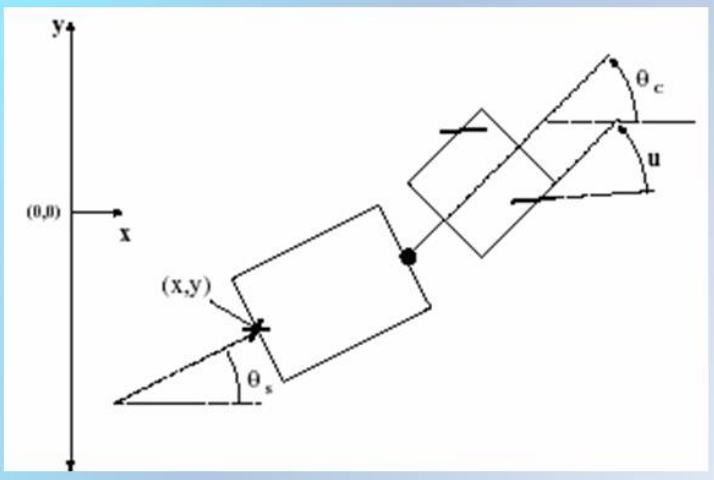
- Garer un semi-remorque en marche arrière

Solution :

- Pilotage du véhicule par un réseau de neurones (PMC)
- Apprentissage par un algorithme évolutionnaire
  - Pas de base d'apprentissage
  - L'expérience permet d'apprendre : comportement émergent

RESEAUX EVOLUTIONNAIRES

Principe :



RESEAUX EVOLUTIONNAIRES

Variables :

x, y	coordinates of the center rear of the trailer
θ_S	angle of the trailer with x-axis
θ_C	angle of the cab with x-axis
u	steering angle of the front wheels relative to cab orientation

Paramètres :

r	$3m$	distance front wheel moves per time step.
L_S	$14m$	length of the trailer, from rear to pivot.
L_C	$6m$	length of the cab, from pivot to front axle.

Contraintes :

$x > 0$	the loading dock is at $x = 0$
$\lvert\theta_S - \theta_C\rvert \leq 90°$	the angle between the cab and trailer can't exceed 90°
$-70° \leq u \leq 70°$	limit of the steering of the front wheel

Équations de mouvement :

$$A = r * \cos(u)$$

$$B = A * \cos(\theta_C[t] - \theta_S[t])$$
$$x[t+1] = x[t] - B * \cos(\theta_S[t])$$
$$y[t+1] = y[t] - B * \sin(\theta_S[t])$$
$$\theta_S[t+1] = \theta_S[t] - \arcsin\left(\frac{A * \sin(\theta_C[t] - \theta_S[t])}{L_S}\right)$$
$$\theta_C[t+1] = \theta_C[t] + \arcsin\left(\frac{r * \sin(u)}{L_S + L_C}\right)$$

$\theta_C[t+1]$ is then adjusted to respect the constraint on $\theta_S - \theta_C$

RESEAUX EVOLUTIONNAIRES

PMC :

3 couches cachées

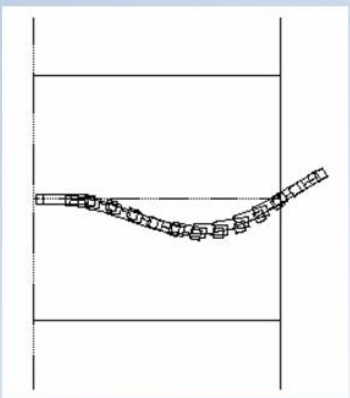
Entrées : x, y, θs, θc

Sortie : u

RESEAUX EVOLUTIONNAIRES

Principe :

- Position de départ : x0, y0, θs0, θc0
- 1 mouvement (itération) : la position suivante est donnée par la sortie du réseau u et par les équations de mouvement
- On autorise 300 mouvements (itérations) max.
- Le point de garage est donné par :
  Xg = 0 ; Yg = 0 ; θsg = 0



RESEAUX EVOLUTIONNAIRES

Apprentissage par algorithme évolutionnaire :

- un individu = 1 PMC = liste des coefficients synaptiques
- population de PMC
- opérateurs de croisement, mutation, sélection
- au bout de N générations : le PMC optimal pour une fonction d 'évaluation (***fitness function***) ⇔ le PMC le plus adapté

RESEAUX EVOLUTIONNAIRES

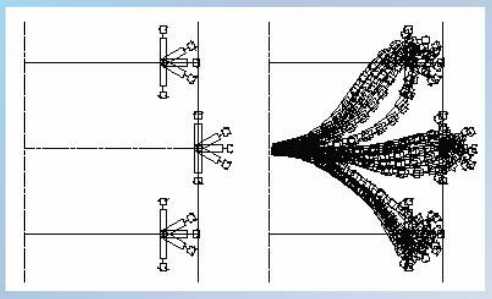
Fitness function :

- On veut minimiser
  - la distance au point de garage
  - le nombre de mouvements (itérations)
- fitness function = $1 / (\varepsilon + d^2 + \gamma * nb\_iter)$, où
  - d est la distance au point de garage
  - nb_iter est le nombre de mouvements (itérations) effectués
  - $\varepsilon = 0.1$ et $\gamma = 0.001$

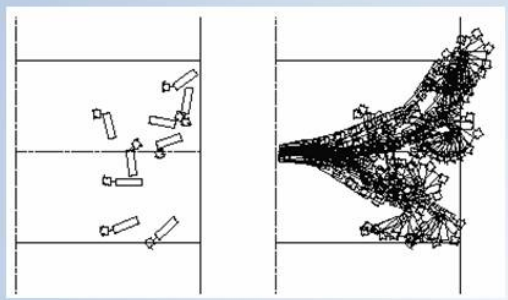
RESEAUX EVOLUTIONNAIRES

Résultats :

- Apprentissage :



- Tests (généralisation) :



RESEAUX EVOLUTIONNAIRES

Exemple 2 :

Problématique :

- Apprentissage du comportement suivant
  - Trouver le plus court-chemin jusqu'à un point-cible
  - Et éviter des obstacles

Solution :

- Pilotage du robot par un réseau de neurones (PMC)
- Apprentissage par un algorithme évolutionnaire
  - Pas de base d'apprentissage
  - L'expérience permet d'apprendre : comportement émergent

RESEAUX EVOLUTIONNAIRES

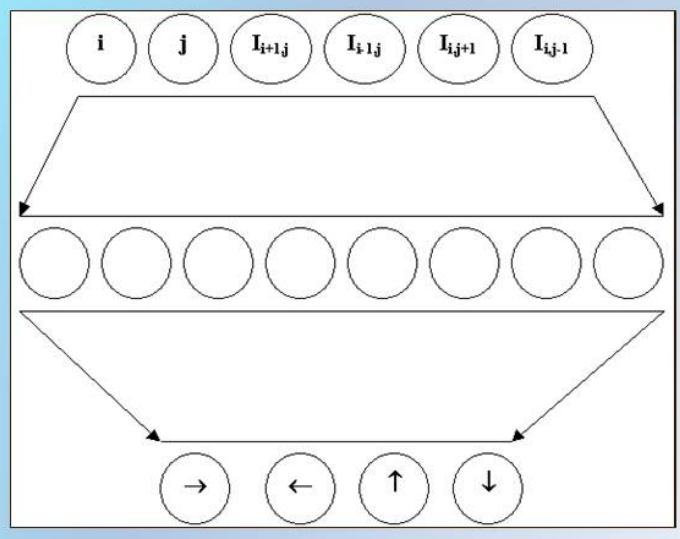
- Modélisation de l 'univers : quadrillage NxM

- Modélisation d 'un obstacle : indice de présence Iij d 'une menace sur une case (i ; j)
  - Iij = 0 si aucune menace et aucun obstacle
  - Iij = -1 si un obstacle
  - Iij = -0.5 si un obstacle sur les cases voisines

- PMC :
  - 6 entrées : coordonnées du point courant et indices de présence d 'un obstacle sur les cases voisines
  - 4 sorties : probabilité de déplacement vers le nord, le sud, l 'ouest ou l 'est. La sortie la plus élevée indique la direction à suivre

RESEAUX EVOLUTIONNAIRES

PMC :



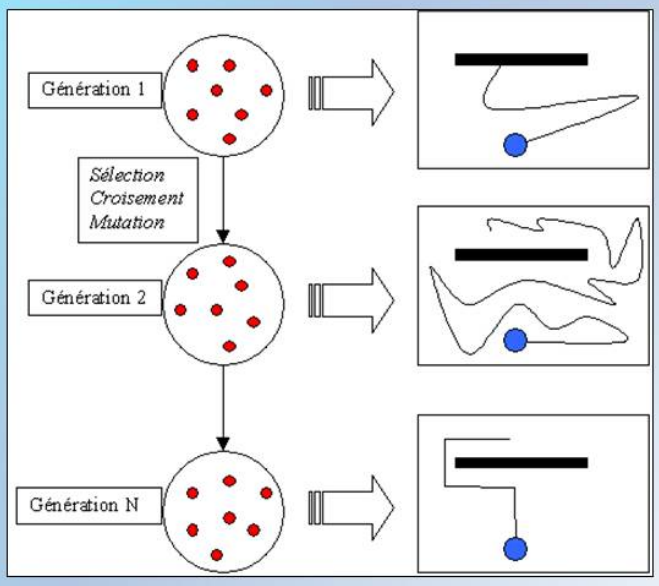
RESEAUX EVOLUTIONNAIRES

- Pas de base d 'apprentissage ! Pas de couples {*position de départ ; liste des déplacements successifs*}

- Apprentissage par des algorithmes génétiques
  - un individu = 1 PMC = liste des coefficients synaptiques
  - population de PMC
  - opérateurs de croisement, mutation, sélection
  - au bout de N générations : le PMC optimal pour une fonction d 'évaluation (***fitness function***) ⇔ le PMC le plus adapté

RESEAUX EVOLUTIONNAIRES

Principe :



RESEAUX EVOLUTIONNAIRES

Principe de l'apprentissage :

Pour tout point de départ d'apprentissage X_i
  Point_Courant = X_i
  $nb\_iter_i = 0$
  Tant que (point_cible A non atteint) et
    (Point_Courant n'est pas dans un obstacle) et
    ($nb\_iter_i < nb\_iter\_max$) Faire
    Incrémenter nb_iter_i
    Calculer les 4 sorties du PMC pour Point_Courant (coordonnées et voisinage)
    Déplacement_Elémentaire = Max (Sortie →, Sortie ←, Sortie ↑, Sortie ↓)
    Déplacer Point_Courant selon Déplacement_Elémentaire
  Fin Tant que
  Si (Point_Courant est dans un obstacle) Alors
    $nb\_iter_i = nb\_iter\_max$
  Fin Si
Fin Pour tout

RESEAUX EVOLUTIONNAIRES

- Comment sélectionner les meilleurs réseaux ?
- Tenir compte de plusieurs critères :
  - la distance au point-cible
  - le nombre de déplacements effectués
  - le cumul des indices de présence de menace des cases parcourues
- D 'où la *fitness function* suivante à maximiser :

$$f(x) = \sum_{i=1}^{K} \frac{\alpha}{1 + 10 \times d(X_i, A)} + \frac{\beta}{nb\_iter_i}$$

Avec Xi un point de départ parmi les K points de départ d 'apprentissage,

A le point-cible à atteindre et nb_iter_i le nombre de déplacements effectués depuis Xi.

RESEAUX EVOLUTIONNAIRES

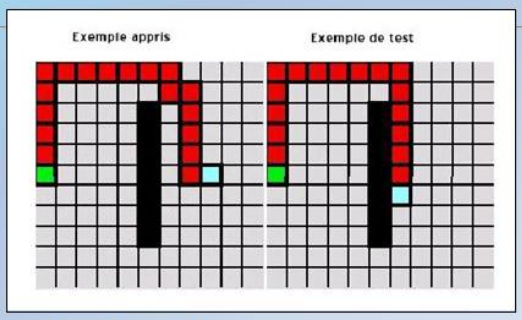
Paramètres AG :

- taille de la population : 100
- nombre de générations : 500
- taux de mutation : 0.1
- taux de croisement : 0.7
- fitness function :
  - $\alpha = 500$,
  - $\beta = 100$

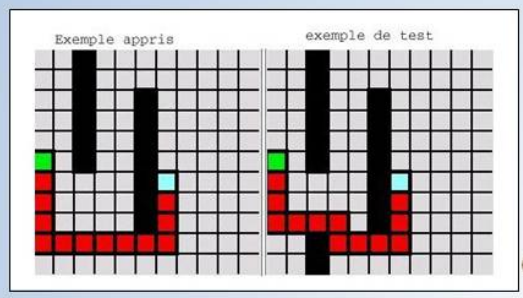
RESEAUX EVOLUTIONNAIRES

Tests : généralisation sur

- les points de départ



- la position et le nombre des obstacles



RESEAUX EVOLUTIONNAIRES

Références :

• ***Darwin revisité par la sélection artificielle***, La Recherche n° de février 2002

• M. Schoenauer et E. Ronald, ***Neuro-Genetic Truck Backer-Upper Controller***, 1994

• D. Floreano et F. Mondada, ***Evolution of Homing Navigation in a Real Mobile Robot***, 1996

EXERCICE: Réseaux évolutionnaires

Apprentissage du comportement suivant :

- Trouver le plus court-chemin jusqu'à un point-cible
- Eviter des obstacles
- Traverser le moins de menaces (zones dangereuses) possible

- EXPOSE COULIBALY

Application des reseaux des neurones à la numerisation des textes

Figure 1. Système de reconnaissance des caractères

MNIST

MNIST (Modified National Institute of Standards and Technology database) est une base de données de caractères manuscrits numérisés qui est largement utilisée dans la recherche en intelligence artificielle et en apprentissage automatique. Elle a été compilée par le National Institute of Standards and Technology (NIST) aux États-Unis et modifiée pour être utilisée comme jeu de données de test dans les systèmes de reconnaissance de caractères.
La base de données MNIST contient **60 000** images de caractères manuscrits pour l'entraînement et **10 000** images pour le test. Chaque image est un caractère manuscrit numérique (de 0 à 9) qui a été normalisé et redimensionné à une taille de 28x28 pixels. Les images sont en niveaux de gris et ont un fond blanc. La base de données MNIST est souvent utilisée comme jeu de données de base pour tester les performances des réseaux de neurones et d'autres modèles de reconnaissance de caractères.

TensorFlow

C'est une plate-forme d'apprentissage automatique open source de bout en bout. Il offre une collection complète et flexible d'outils, de ressources communautaires et de bibliothèques pour aider les chercheurs et les développeurs à créer et à déployer facilement des applications optimisées par ML.
Vous pouvez utiliser ses API intuitives et de haut niveau, telles que Keras, avec une implémentation rapide pour développer et former des modèles ML et les itérer et les déboguer facilement. Vous pouvez déployer des modèles ML sur site, dans votre navigateur, sur l'appareil ou dans le cloud sans vous soucier du langage de programmation utilisé.

- EXPOSE DE KWAME

LE TRAITEMENT NATUREL DU LANGAGE(NLP)

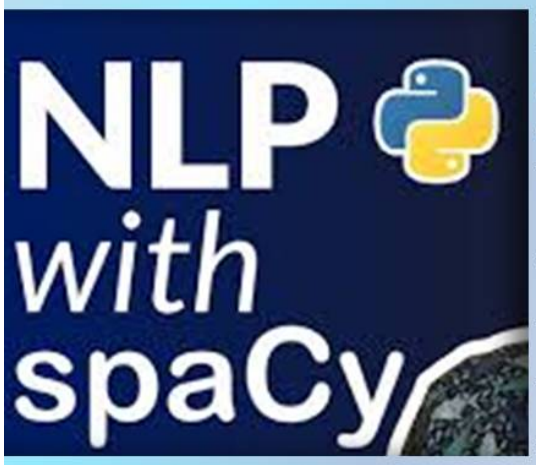
COMMENT FONCTIONne LE NLP

- Le NLP combine l'intelligence artificielle et traitement linguistique. La dernière génération des technologies de NLP s'adosse à des réseaux de neurones artificiels ou de simples modèles de machine learning statistiques. Des modèles d'apprentissage auront été entrainés sur des volumes importants de texte.

Natural language toolkit (nltk)

NLTK comprend des bibliothèques pour la plupart des tâches NLP énumérées ci-dessus, ainsi que des bibliothèques pour des sous-tâches, telles que l'analyse syntaxique des phrases, la segmentation des mots, le déracinement et la lemmatisation (méthodes permettant de réduire les mots à leur racine), et la tokenisation (pour décomposer les phrases, les paragraphes et les passages en jetons qui aident l'ordinateur à mieux comprendre le texte). Il comprend également des bibliothèques permettant de mettre en œuvre des fonctionnalités telles que le raisonnement sémantique, c'est-à-dire la capacité de tirer des conclusions logiques à partir de faits extraits du texte.

spaCy



SpaCy est une librairie Python permettant de faire du natural language processing . Cette librairie prend une phrase en entrée et nous permet d'effectuer plusieurs opérations de traitement. Par exemple, nous pouvons **effectuer :**

- Tokenisation : Il s'agit du processus consistant à briser un flux de texte en plusieurs mots, phrases, symboles ou tout autre élements significatifs dénommés Signes (tokens).
- Part of speech Tagging : Associer le type de mot à chacun des tokens (ex : verbe, nom ..)
- Dependency Parsing : Décrire les relations entre les tokens (ex : sujets, objets..)
- Lemmatization : Donner la forme basique des mots (ex : lu → lire, réglées → règle)
- Named Entity Recognition : Libeller des objets du monde réel (ex : Google → Entreprise)

EXPOSE zoubir

Algorithme Génétique

16
3

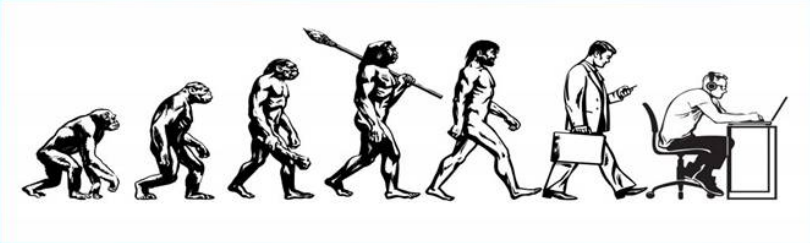
Définition

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable. Les algorithmes génétiques utilisent la notion de sélection naturelle et l'appliquent à une population de solutions potentielles au problème donné.

Principes

Les algorithmes génétiques utilisent la théorie de Darwin sur l'évolution des espèces. Elle repose sur trois principes :

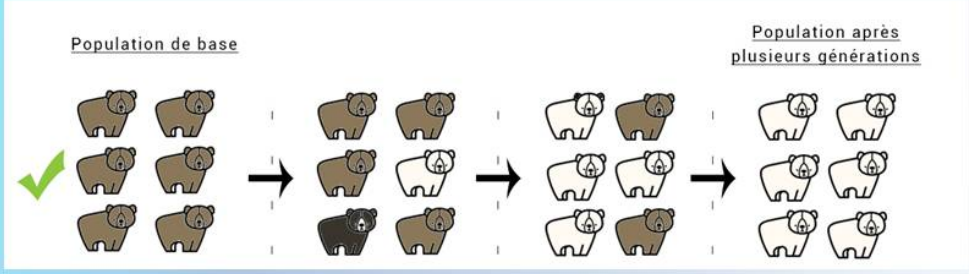
- **Le principe de variation**
- **Le principe d'adaptation**
- **Le principe d'hérédité**



Opérateurs d'évolution

Il y a trois opérateurs d'évolution dans les algorithmes génétiques :

- **La sélection**
- **Le croisement**
- **La mutation**

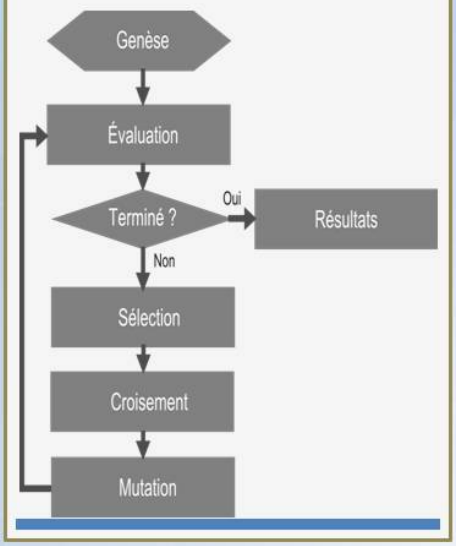


La genèse est l'étape de la création d'une population aléatoire. C'est le point de départ de notre algorithme

L'évaluation est l'analyse des individus pour analyser si une solution est disponible. Pour ceci, nous utilisons un fonction de coût, ou d'erreur, afin de définir le score d'adaptation des individus lors du processus de sélection.

Nous effectuons une boucle tant que l'évaluation estime que la solution n'est pas optimale

Schéma d'un algorithme génétique



EXPOSE ELOM

RECONNAISSANCE DE FORME

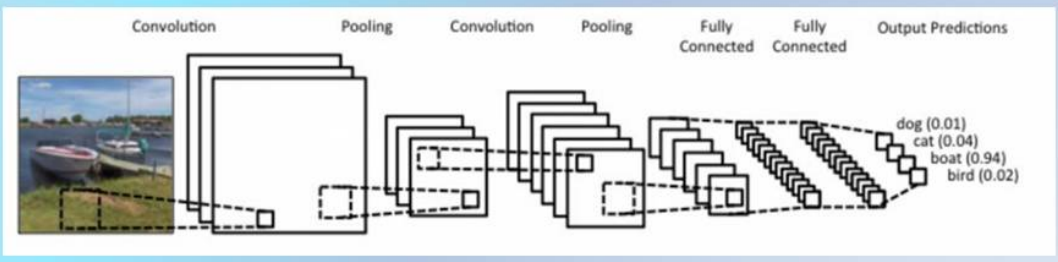
Architecture du modèle Convolutif

On utilise habituellement trois types de couches pour former un réseau convolutif. Celui-ci est composé de couche(s) de convolution (CONV) , de couche(s) de pooling (POOL) et de couche(s) entièrement conn ctée (FC). La ou les couches entièrement connectées sont souvent utilisées pour la sortie du réseau. Habituellement, une couche de convolution est suivie d'une fonction d'activation puis d'une couche de pooling, cette séquence peut être répétée plusieurs fois jusqu'à la couche entièrement connectée pour former un réseau convolutif que l'on dénote souvent sous la notation CONVNET .

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau = (g * f)(t).$$

L'architecture CNN est formée par un empilement de couches de traitement indépendantes :

- La couche de convolution (CONV) qui traite les données d'un champ récepteur.
- La couche de pooling (POOL), qui permet de compresser l'information en réduisant la taille de l'image intermédiaire (souvent par sous-échantillonnage).
- La couche de correction (ReLU), souvent appelée par abus 'ReLU' en référence à la fonction d'activation (Unité de rectification linéaire).
- La couche "entièrement connectée" (FC), qui est une couche de type perceptron.
- La couche de perte (LOSS).



Exemple illustratif d'un modèle CNN

Implémentation: Keras

Keras est une API de réseaux de neurones de haut niveau, écrite en Python et capable de fonctionner sur TensorFlow ou Theano. Il a été développé en mettant l'accent sur l'expérimentation rapide. Être capable d'aller de l'idée à un résultat avec le moins de délai possible est la clé pour faire de bonnes recherches. Il a été développé dans le cadre de l'effort de recherche du projet ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), et son principal auteur et mainteneur est François Chollet, un ingénieur Google. En 2017, l'équipe TensorFlow de Google a décidé de soutenir Keras dans la bibliothèque principale de TensorFlow. Chollet a expliqué que Keras a été conçue comme une interface plutôt que comme un cadre d'apprentissage end to end. Il présente un ensemble d'abstractions de niveau supérieur et plus intuitif qui facilitent la configuration des réseaux neuronaux indépendamment de la bibliothèque informatique de backend. Microsoft travaille également à ajouter un backend CNTK à Keras aussi.

ImageNet

ImageNet est une base de données d'images annotées produit par l'organisation du même nom, à destination des travaux de recherche en vision par ordinateur. En 2016, plus de dix millions d'URLs ont été annotées à la main pour indiquer quels objets sont représentés dans l'image ; plus d'un million d'images bénéficient en plus de boîtes englobantes autour des objets. La base de données d'annotations sur des URL d'images tierces est disponible librement, **ImageNet** ne possédant cependant pas les images elles-mêmes. De 2010 à 2017, le projet **ImageNet** a organisé un concours annuel : ImageNet Large Scale Visual Recognition Challenge (ILSVRC), ou "Compétition ImageNet de Reconnaissance Visuelle à Grande Échelle". Elle consistait en une compétition logicielle dont le but était de détecter et classifier précisément des objets et des scènes dans les images naturelles.

EXPOSES : 20 Diapos + TP

1. Application des RN à la numérisation de texte (reconaissance des manuscrits) ALBERT ESSIEN
2. Application des RN à la reconnaissance de formes Lydia Afrakoma A
3. Application des RN au traitement du langage NATUREL ouiame essaid
4. Application de l'algorithme de la carte de Kohonen (oumayma ouzennou)
5. Algorithme génétique(Gobe Omar Elmi) A
6. Réseux de neurones multicouches MLP WILLIAMS ASANTE
7. Réseaux de neurones récurrents Rahma Toosimle
8. Réseux de neurones convolutifs Emmanuel Osei
9. application des réseaux de neurones en medecin (kaoutar karouach)
10. Algorithme de la rétropropagation du gradient KHAOULA
11. Application des réseaux de neurones avec Matlab YASSINE
12. Historique des réseaux de neurones du débit à nos jours (Zaitun Ishaq)
13. Evolution des IA et les différentes solutions actuelles Mohamed El mamoune Abdelhamid
14. Les Chatbots et les méthodes de création des chatbots Sadam soufiani
15. TENSORFLOW Framework Lakhouil houda
16. KERAS Framework Jad boukham

REFERENCES

- https://gisnt.org/pdf/R%C3%A9seau%20neuronaux%20artifitiels.pdf