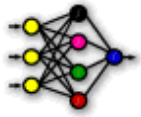
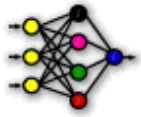
Réseaux de neurones



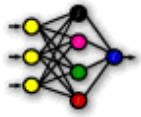
Cours 6

Réseaux multicouches – Réseaux profonds

Plan du chapitre



- Le neurone biologique
  - Fonctionnement
- Le neurone formel ou artificiel
  - Fonctionnement
  - Le neurone de Mc Culloch et Pitts
- Classement
  - Règles de décision
- Classification ou apprentissage
  - Non supervisé (loi de Hebbs)
  - Supervisé (Perceptron)
- Matlab : nftool, nnstart

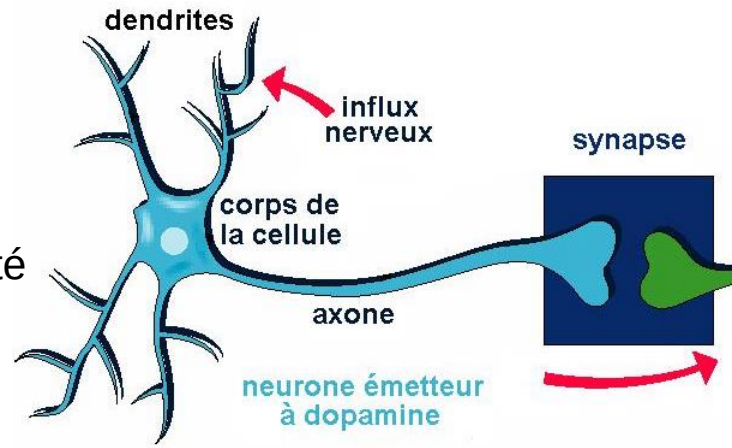


- https://github.com/rasmusbergpalm/DeepLearnToolbox/blob/master/README.md

Le neurone biologique

- Observation biologique (déf. usherbrooke)
  - Neurones :
    - cellules reliées entre elles par des axones et des dendrites
  - En première approche
    - on peut considérer que ces sortes de filaments sont conducteurs d'électricité et peuvent ainsi véhiculer des messages (dopamines) depuis un neurone vers un autre
  - Les dendrites représentent
    - les entrées
  - L'axone représente
    - la sortie

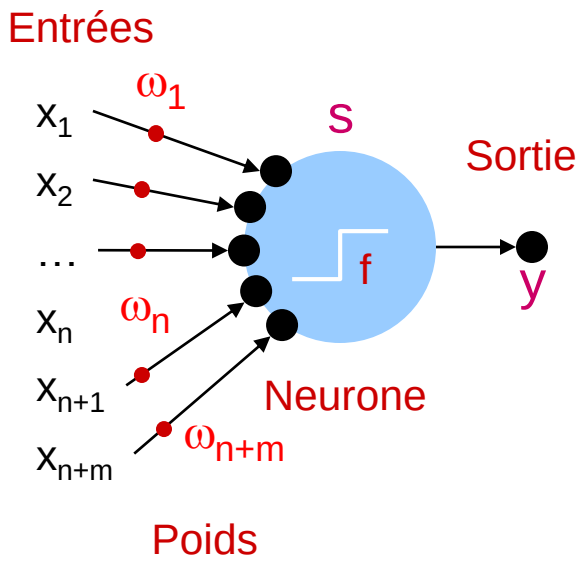
Un neurone biologique



*La dopamine est une molécule, un **neurotransmetteur** parmi de nombreux neurotransmetteurs...*
*C'est un **message chimique** du système nerveux*

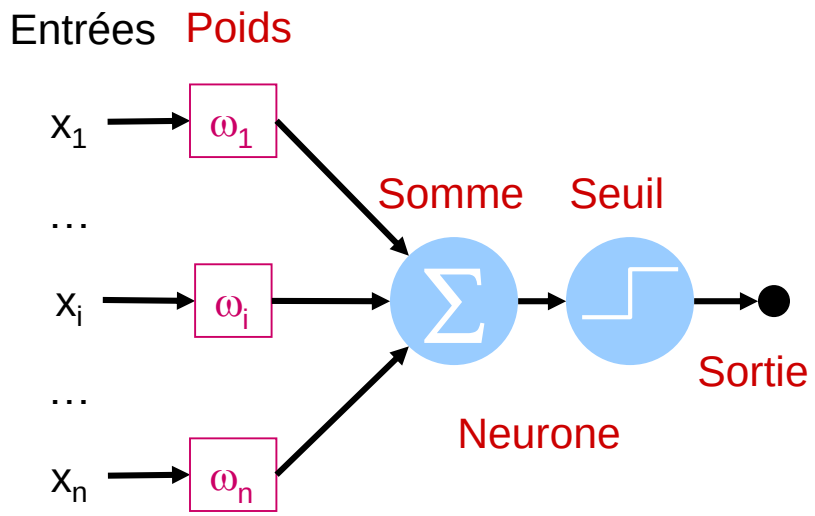
Le neurone artificiel

- Selon Mc Culloch et Pitts
  - Modèle mathématique très simple, dérivé de la réalité biologique
  - Neurone :
    - Un automate composé de
      1. Un ensemble d'entrées, x_i
      2. Un ensemble de poids, ω_i
      3. Un seuil, s
      4. Une fonction d'activation, f
      5. Une sortie, y



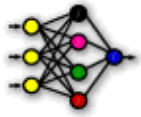
Le neurone artificiel

- 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



Un neurone formel, selon Mac Culloch et Pitts

Le neurone artificiel



- **Calcul de la sortie**

  Si a > s (seuil) $\Rightarrow$ x = 1

  sinon x = 0, -1 ou autre valeur $\neq$ 1 selon l'application

  – Ou bien

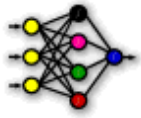
  $a = \Sigma (\omega_i . e_i) - s$

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

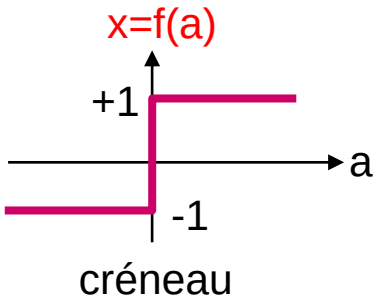
  x = signe(a) (si a > 0 alors x = +1 sinon x = -1)

Le neurone artificiel

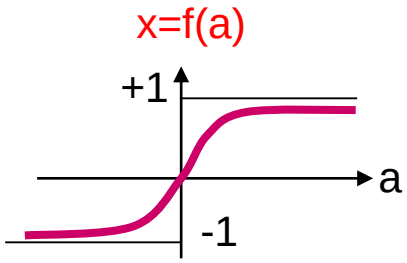
Fonctionnement



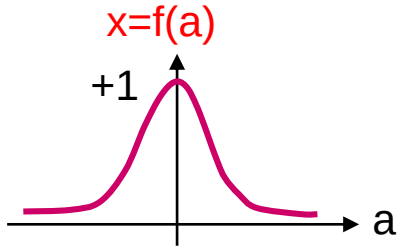
- La fonction de transfert :
  - La valeur est transmise aux neurones avals à l'aide d'une fonction de transfert
  - Plusieurs formes possibles, les plus courantes sont :



créneau



sigmoïde

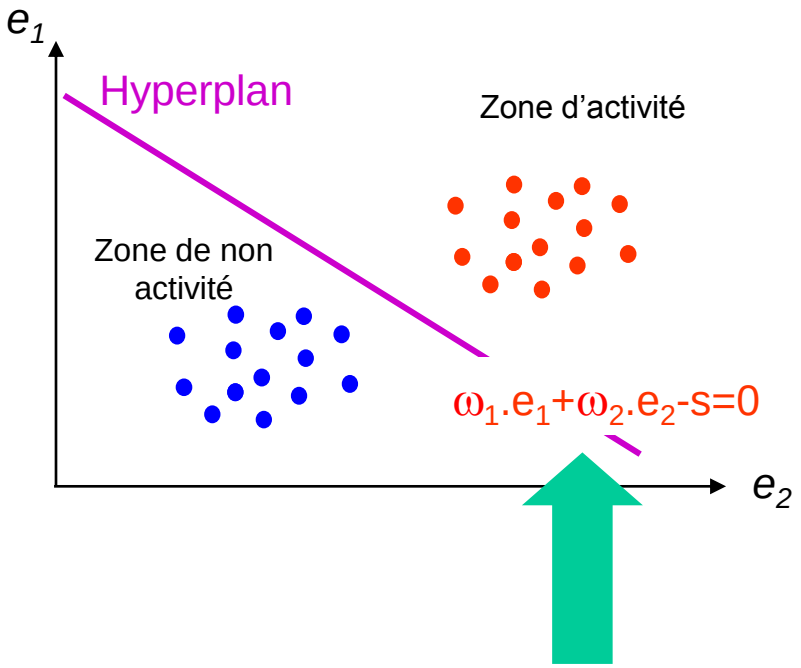


gaussienne

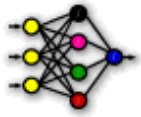
- Les valeurs obtenues sont dans les intervalles [-1, +1]

Utilité du neurone
Classement

- Idée fondamentale
  - Si on conçoit les entrées comme les composantes d'un vecteur d'observation (dans un espace)
  - La réponse parmi : +1 et 0 permet de dire :
    - que le neurone décompose son espace d'observation en deux :
      - ➢ zone d'activité et zone de non activité

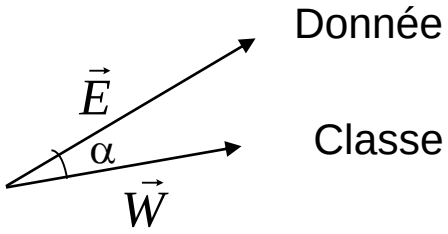


Classement



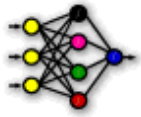
- Ceci s'explique aussi par la comparaison vectorielle entre
  - la donnée (E) et la classe (ω)
  - cette comparaison se fait par confrontation (multiplication scalaire) de leurs vecteurs

$$\vec{\omega} \otimes \vec{E} = \sum(\omega_i . e_i) = a = |\omega| . |E| . \cos(\alpha)$$

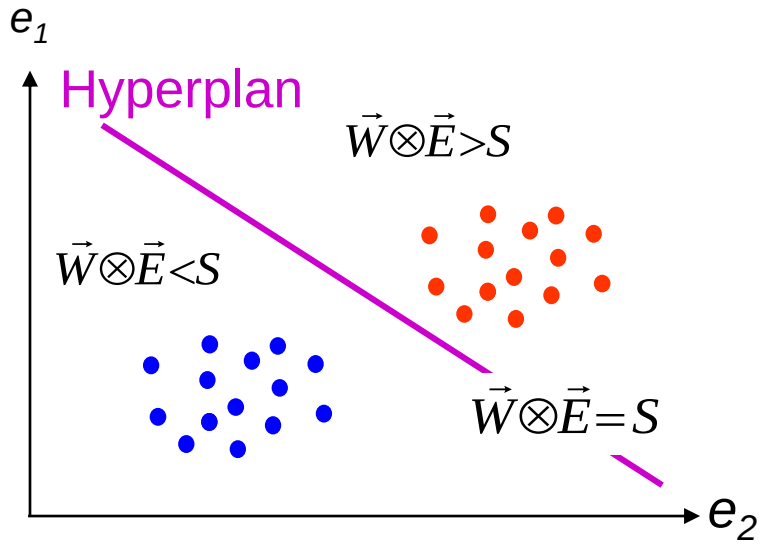


  - Le produit scalaire indique :
    - si les deux vecteurs sont dans la même direction, la valeur va a priori être grande, et l'attirance importante
    - s'ils sont très éloignés (i.e. perpendiculaires voir sens opposés), la valeur va être de plus en plus faible
    - plus la valeur est faible plus on est loin de la classe apprise, moins il y a de chance que le neurone modélise cette classe

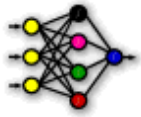
Classement



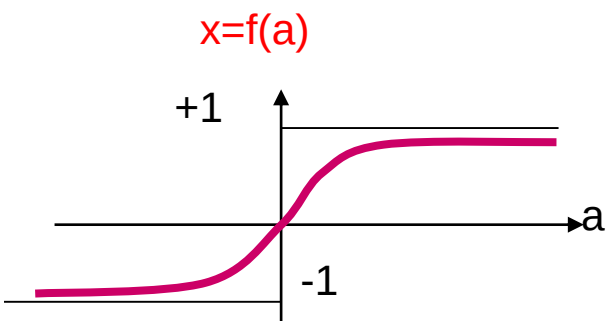
- En général, on fait une interprétation très simpliste de ce résultat comme ceci :
  - On se donne un seuil
    - Si le produit scalaire est supérieur à ce seuil, l'entrée fait partie de la classe apprise
    - Sinon, elle n'en fait pas partie
    - C'est pour cela, que souvent, on représente ce seuil comme un hyperplan qui délimite deux classes



Classement

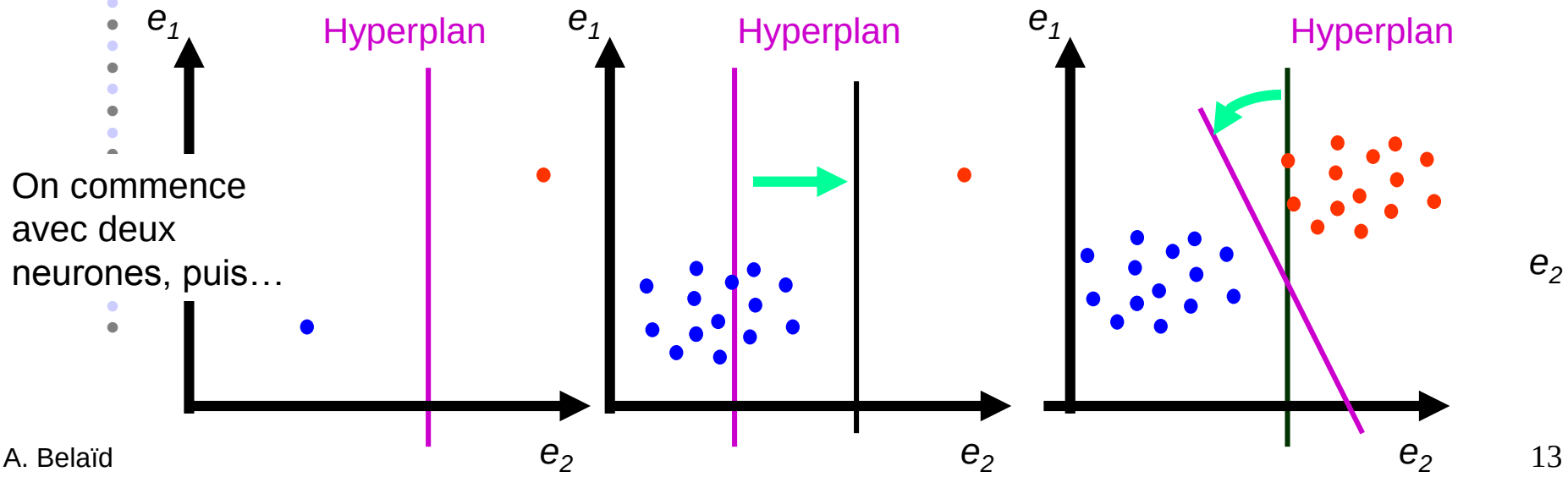


- Ceci s'explique aussi par la fonction de transfert
  - Plus la réponse est proche de 1, plus la valeur de l'acceptation de la classe par le neurone est forte
  - Plus la réponse s'éloigne de 1, et puis le neurone montre son ignorance de la classe
  - Capital : suivant que l'on monte ou que l'on descend : on s'approche ou on s'éloigne de la réponse (dérivée positive ou négative)



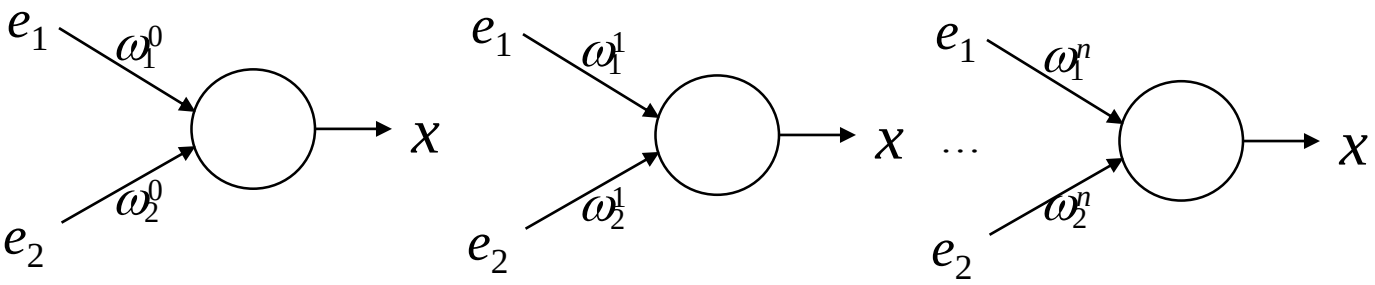
Apprentissage

- Un neurone formel peut donc séparer des vecteurs en deux classes
- Les classes sont déterminées par les coefficients synaptiques
- Le problème est alors de choisir les coefficients
  - Ceci s'effectue en se donnant des *exemples* : un ensemble de vecteurs dont on connaît les classes respectives
  - Ensuite, par itérations successives, on stabilise les coefficients



Apprentissage

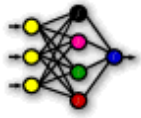
- Explication
  - On commence avec des poids initiaux, puis on corrige…



  - L'apprentissage est un processus par lequel les paramètres libres (poids) du réseau sont adaptés à travers un processus de stimulation continu par l'environnement
  - Le type d'apprentissage est déterminé par la manière dans laquelle les paramètres du réseau changent

Apprentissage

Non supervisé



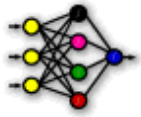
- Basé sur la loi de Hebb (1949)
  - S'applique aux connexions entre deux neurones
  - Règle de Hebb :

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \eta(t).x_i(t).y_j(t)$$

  - Elle indique que :
    - La modification de poids dépend de la co-activation des neurones pré-synaptiques et post-synaptiques
  - Ce qui se traduit par :
    - Si deux neurones de chaque côté d'une synapse (connexion) sont activés simultanément (synchronisés), alors la synapse est augmentée
    - Si les deux neurones sont activés de manière asynchrone, alors la synapse est diminuée

Apprentissage

Non supervisé



- Elle peut être modélisée comme suit :

  $\omega_{ij}(t+1) = \omega_{ij}(t) + \partial\omega_{ij}(t)$

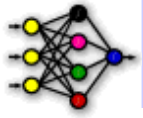
  $\delta\omega_{ij}(t) = e_i.e_j$

  – la co-activité est modélisée comme le produit des deux valeurs d'activation : comme le ET logique (voir table)
- L'algorithme d'apprentissage modifie de façon itérative (petit à petit) les poids
  – pour adapter la réponse obtenue à la réponse désirée (suivant les règles de la table)
  – Il s'agit en fait de modifier les poids lorsqu'il y a erreur seulement

x_i	x_j	$\delta\omega_{ij}$
0	0	0
0	1	0
1	0	0
1	1	+

Apprentissage

Non supervisé



- L'algorithme

1/ Initialisation des poids et du seuil s à des valeurs (petites) choisies au hasard

2/ Présentation d'une entrée E = (e_1, ... e_n) de la base d'apprentissage

3/ Calcul de la sortie obtenue x pour cette entrée :

$$a = \Sigma (\omega_i . e_i) - s$$

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

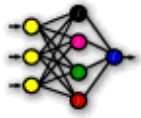
4/ Si la sortie x est différente de la sortie désirée d_l pour cet exemple d'entrée E_l alors modification des poids (μ est une constante positive qui spécifie le pas de modification des poids) :

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \mu.(x_i.x_j)$$ x_i et x_j = sorties des neurones i et j :

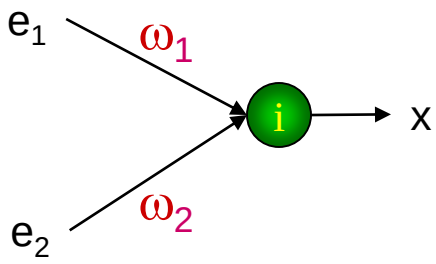
5/ Tant que tous les exemples de la base d'apprentissage ne sont pas traités correctement (i.e. modification des poids), retour à l'étape 2

Apprentissage

Non supervisé



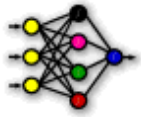
- Exemple
  - Choisissons pour les neurones un comportement binaire
  - Les entrées e_1 et e_2 sont considérées comme des neurones
  - Nous allons réaliser l'apprentissage sur un problème très simple
    - La base d'apprentissage est décrite par la table ci-contre



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

Apprentissage

Non supervisé



1/ Conditions initiales : $\mu = +1$, les poids et le seuil sont nuls

2/ Calculons la valeur de x pour l'exemple (1) :

3/ $a = \omega_1.e_1 + \omega_2.e_2 - s = 0.0 \,.\, 1 + 0.0 \,.\, 1 - 0.0 = 0$, $a \leq 0 \Rightarrow x = -1$

4/ La sortie est fausse, il faut donc modifier les poids en appliquant :

$\omega_1 = \omega_1 + e_1.x = 0.0 + 1.1 = 1$

$\omega_2 = \omega_2 + e_2.x = 0.0 + 1.1 = 1$

2/ On passe à l'exemple suivant (2) :

3/ $a = 1.1 + 1.-1 - 0.0 = 0$ $a \leq 0 \Rightarrow x = -1$

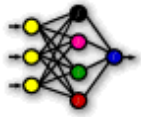
4/ La sortie est fausse, il faut donc modifier les poids en appliquant :

$\omega_1 = 1 + 1.1 = 2$

$\omega_2 = 1 + 1.-1 = 0$

Apprentissage

Non supervisé



…/ L'exemple suivant (3) est correctement traité :

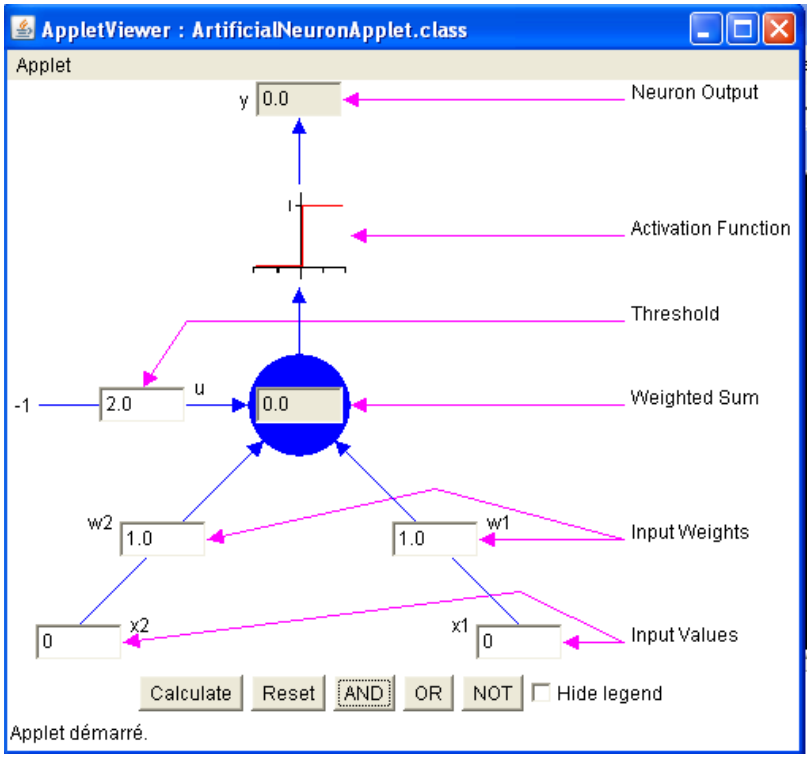
- a = -2 et x = -1 (la sortie est bonne)
- On passe directement, sans modification des poids à l'exemple (4)
- Celui-ci aussi est correctement traité
- On revient alors au début de la base d'apprentissage : l'exemple (1)
- Il est correctement traité, ainsi que le second (2)
- L'algorithme d'apprentissage est alors terminé :

  toute la base d'apprentissage a été passée en revue sans modification des poids

Apprentissage

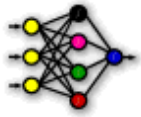
Non supervisé

- Exemple : applets >>…>> aneuron



Apprentissage

Non supervisé



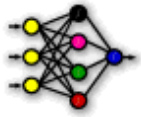
- Exercice
  - Soit le réseau composé de 4 entrées et d'un neurone de sortie ($\omega_1 = \omega_2 = \omega_3 = \omega_4 = s = 0$) et la base d'apprentissage :

	e_1	e_2	e_3	e_4	x
(1)	1	-1	1	-1	1
(2)	1	1	1	1	1
(3)	1	1	1	-1	-1
(4)	1	-1	-1	1	-1

  - Recherchez les valeurs de poids qui résolvent le problème

Apprentissage

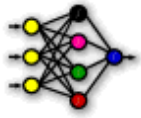
Non supervisé



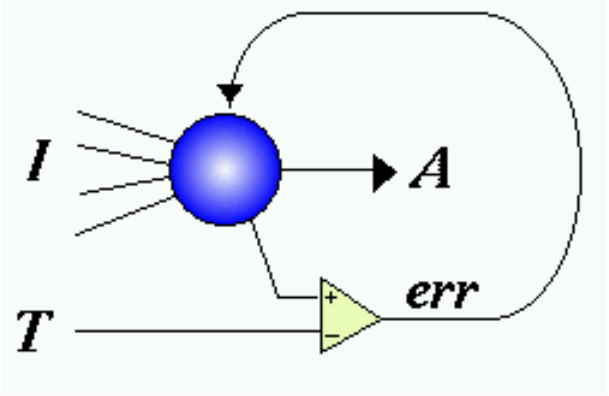
- Réponse
  - Cet algorithme d'apprentissage (par loi de Hebb) ne permet pas de trouver une solution à ce problème (on n'arrive pas à satisfaire toutes les sorties)
  - Nous ne sommes pas capables d'exprimer une combinaison des activations en corrélation avec la sortie
  - Pourtant, il existe des solutions comme par exemple (ω_1 = -0.2, ω_2 = -0.2, ω_3 = 0.6, ω_4 = 0.2)
  - Un algorithme de calcul efficace pour ce problème est l'apprentissage sur le modèle du Perceptron abordé au chapitre suivant

Apprentissage

Supervisé



- Le Perceptron
  - La règle de Hebb ne s'applique pas dans certains cas, bien qu'une solution existe
  - Un autre algorithme d'apprentissage a donc été proposé, qui tient compte de l'erreur observée en sortie
  - La différence se situe au niveau de la modification des poids



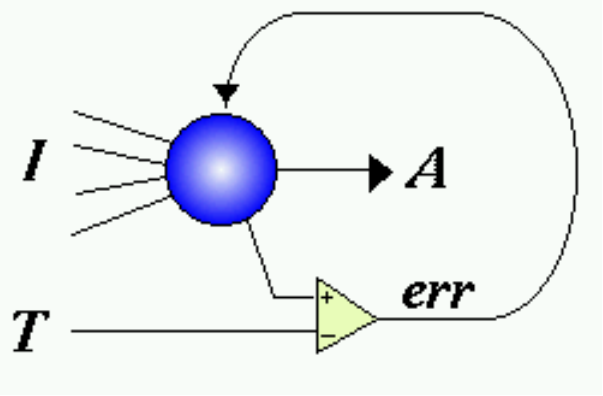
Apprentissage

Supervisé

- Règle d'apprentissage
  - La règle d'apprentissage a la forme suivante

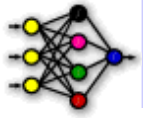
$$\omega_{ij}(t+1) = \omega_{ij}(t) + \mu.err(p)$$

  - où $0 \leq \mu < 1$ est un paramètre qui contrôle le taux d'apprentissage (learning rate) et
  - err(p) est l'erreur lorsque le motif d'entrée p est présenté



Apprentissage

Supervisé



- Algorithme

1/ Initialisation des poids et du seuil s à des valeurs (petites) choisies au hasard

2/ Présentation d'une entrée $E_i = (e_1, \ldots e_n)$ de la base d'apprentissage

3/ Calcul de la sortie obtenue x (du neurone considéré) pour cette entrée :

$a = \Sigma(\omega_i.e_i) - s$

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

4/ Si la sortie x du Perceptron est différente de la sortie désirée d_i pour cet exemple d'entrée E_i alors modification des poids :

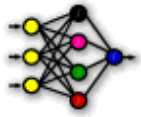
$$\omega_i(t+1) = \omega_i(t) + \mu.((d_i - x).e_i)$$

Remarque : $(d_i - x)$ est une estimation de l'erreur

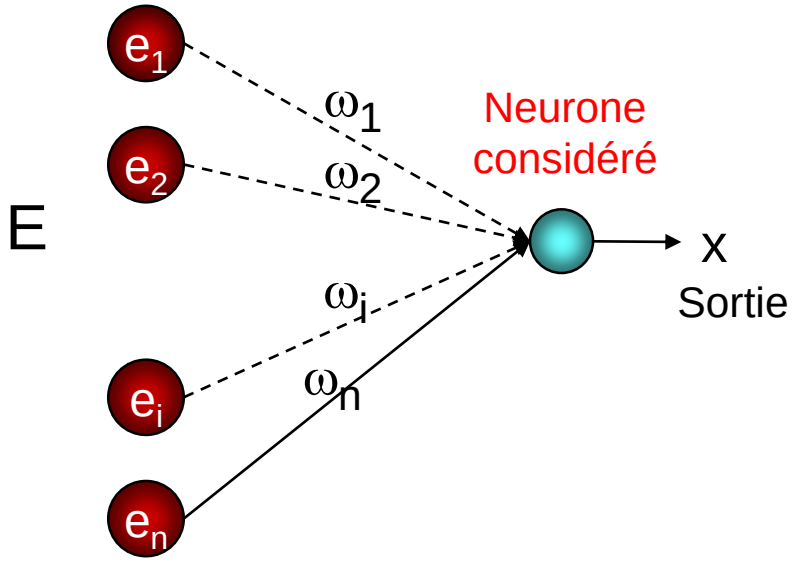
5/ Tant que tous les exemples de la base d'apprentissage ne sont pas traités correctement (i.e. modification des poids), retour à l'étape 2

Apprentissage

Supervisé

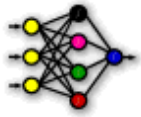


- Attention :
  - cela se passe entre un élément d'une entrée (neurone) et un autre neurone



Apprentissage

Supervisé



- Exemple

  Base d'exemples d'apprentissage :

  1/ Conditions initiales : $\omega_1 = -0.2$, $\omega_2 = +0.1$, $s = 0.2$, ($\mu = +0.1$)

  2/ a(1) = -0.2 + 0.1 -0.2 = -0.3

  3/ x(1) = -1 (la sortie désirée d(1) = +1, d'où modification des poids)

  4/ $\omega_1 = -0.2 + 0.1 \,.\, (1 + 1) \,.\, (+1) = 0$

  $\omega_2 = +0.1 + 0.1 \,.\, (1 + 1) \,.\, (+1) = +0.3$

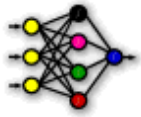
  2/ a(2) = +0.3 - 0.2 = +0.1

  3/ x(2) = +1 Faux

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

Apprentissage

Supervisé



4/ $\omega_1 = 0 + 0.1 \,.\, (-1 - 1) \,.\, (-1) = +0.2$

$\omega_2 = +0.3 + 0.1 \,.\, (-1 - 1) \,.\, (+1) = +0.1$

2-3/ a(3) = -0.2 -0.1 -0.2 = -0.5 Ok

2-3/ a(4) = +0.2 - 0.1 - 0.2 = -0.1 Ok

2-3/ a(1) = +0.2 + 0.1 - 0.2 = +0.1 Ok

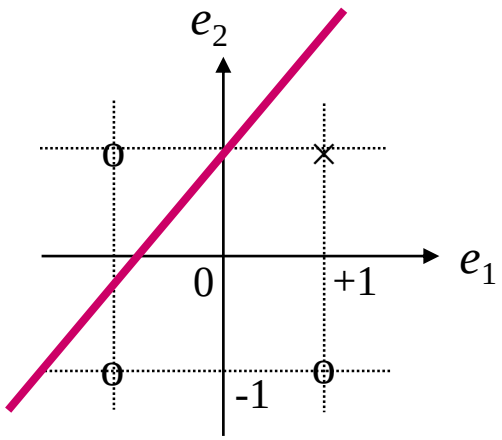
2-3/ a(2) = -0.2 + 0.1 - 0.2 = -0.1 Ok

5/ Tous les exemples de la base ont été correctement traités, l'apprentissage est terminé

Apprentissage

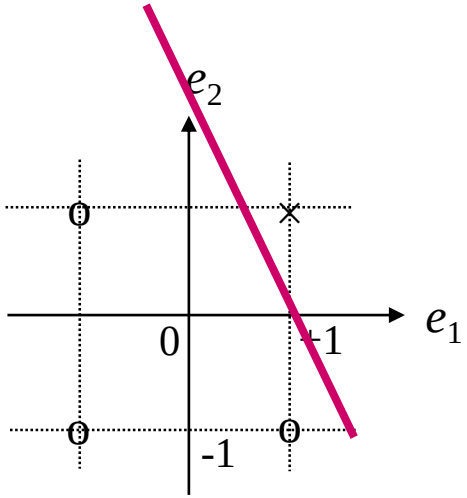
Supervisé

avant



$-0.2\ e_1 + 0.2\ e_2 = 0.2$

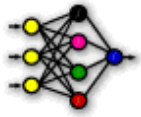
après



$0.2\ e_1 + 0.1\ e_2 = 0.2$

Apprentissage

Supervisé

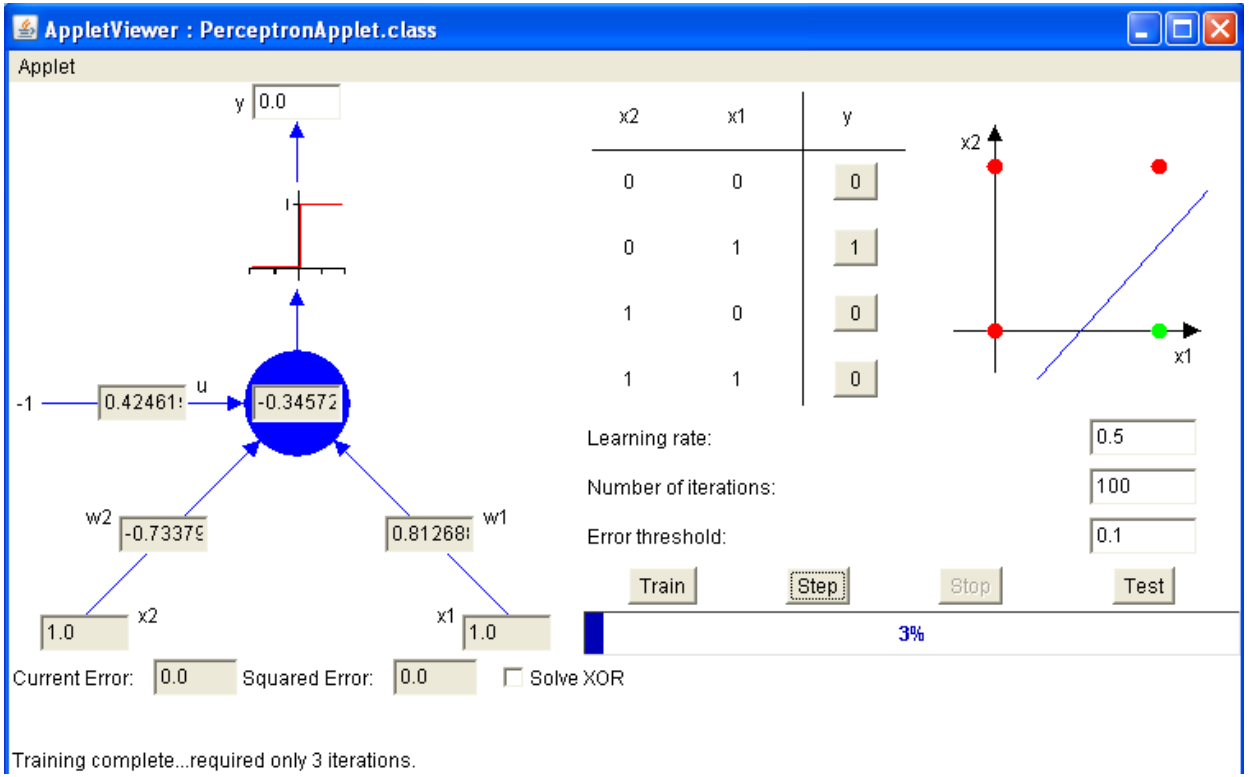


- Résultat
  - Le Perceptron réalise une partition de son espace d'entrée en 2 classes (1 et 2) selon la valeur de sa sortie (+1 ou -1)
  - La séparation de ces deux zones est effectuée par un hyperplan
  - L'équation de la droite séparatrice est : $\omega_1.e_1 + \omega_2.e_2 - s = 0$

Apprentissage

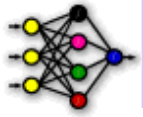
Supervisé

- Applet : applets >> tutorialPrg>> perceptron



Apprentissage

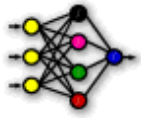
Supervisé



- Exercice 1
  - Soient les points suivants P_1=(-3,2), P_2=(-2,-2), P_3=(3,-3), P_4=(5,-1), P_5=(1,1), P_6=(-1,3)
  1. Ajustez les coefficients de l'équation de l'hyperplan : -x +y +1 = 0 de manière à séparer les points P_1, P_2, P_3 de P_4, P_5, P_6
  2. Combien d'itérations sont nécessaires pour faire converger l'apprentissage (le minimum)
  3. Classez les points P_7=(-1,1), P_8=(1,-2), P_9=(3,1)
  4. Validez les résultats avec l'Applet

Apprentissage

Supervisé



- Exercice

  1/ Simuler la fonction ET avec des poids fixes et sans apprentissage. Les paramètres du Perceptron sont les suivants :

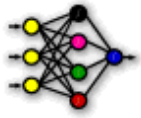
  $\omega_1 = 0.2$, $\omega_2 = 0.1$ et $s = -0.2$

  – Les exemples de comportement à vérifier (ET) sont rappelés sur la table suivante :

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

Apprentissage

Supervisé



2/ Pour la même base d'apprentissage, réaliser l'apprentissage (ne pas oublier la modification du seuil). Le choix des conditions initiales est confié au hasard. Dans une première étape, il est conseillé de refaire pas à pas l'exemple :

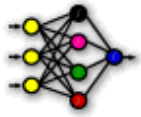
ω_1 = -0.2, ω_2 = +0.1, s = 0, μ = +0.1 (Conditions initiales). Puis faites varier μ.

3/ Essayer d'apprendre le XOR

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

Apprentissage

Supervisé



4/ Sachant que le XOR peut s'écrire comme : ((e_1 ET (NON(e_2))) OU (NON(e_1) ET e_2)) , proposez une solution pour réaliser le XOR avec 3 Perceptrons (NON(1) = -1 et inversement)

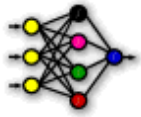
e_1	e_2	d	
1	1	1	(1)
-1	1	1	(2)
-1	-1	-1	(3)
1	-1	1	(4)

Table du OU

– L'apprentissage de chacun des Perceptrons est réalisé séparément des autres. Qu'en déduisez-vous quant aux possibilités d'un Perceptron ? d'une association de Perceptrons ?

Apprentissage

Supervisé

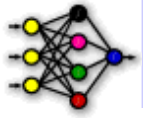


5/ Réaliser la fonction ET et OU avec 2 neurones. Dans ce cas, le réseau se compose de 2 entrées, 2 neurones et 4 poids. L'apprentissage de chacune des fonctions n'est pas séparé. Il faut donc construire une base d'apprentissage spécifique de ce problème (qui ne comprendra pas plus de 4 exemples)

Reconnaissance des formes

Perceptron Multicouches

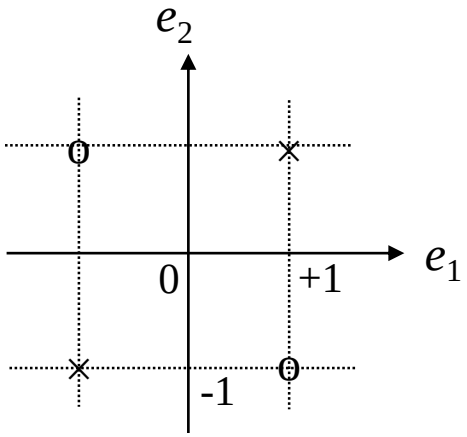
Limite du modèle simple



- Problème du OU exclusif
  - Soient les deux classes + et o représentées par les couples d'entrées suivantes
  - On cherche à savoir si elles peuvent être séparées par un neurone simple (une seule couche de neurones d'entrées)

e_1	e_2	s	$signe$
–1	–1	+1	×
–1	+1	–1	○
+1	–1	–1	○
+1	+1	+1	×

Graphiquement

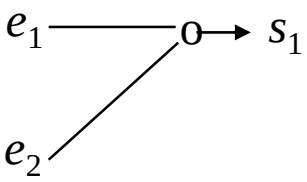


  - Constat :
    - non, car les deux classes ne sont pas séparables

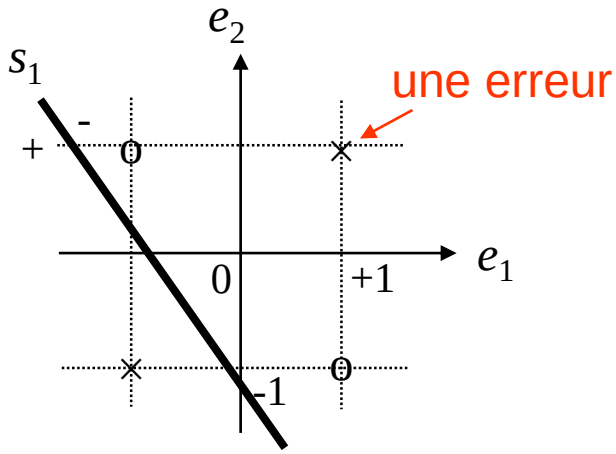
Limite du modèle simple

le OU exclusif

- Solution : ajouter des neurones
  - Pour raffiner la classification (une autre couche de neurones)
  - Solution : 3 étapes
    1. entraîner d'abord un neurone (ou une couche) pour faire une classification partielle (en réduisant les erreurs : 1 erreur résiduelle : sortie s_1(=-1))



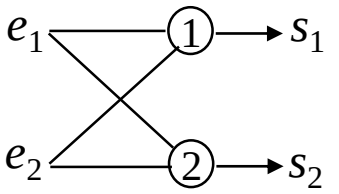
Graphiquement



Limite du modèle simple

le OU exclusif

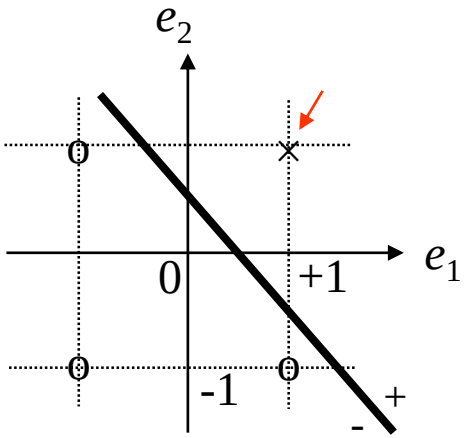
2. Comme il reste une erreur, on crée un autre neurone (ou une autre couche) pour la corriger. On l'entraîne sur cette erreur



Côté positif

On l'apprend avec l'erreur 1

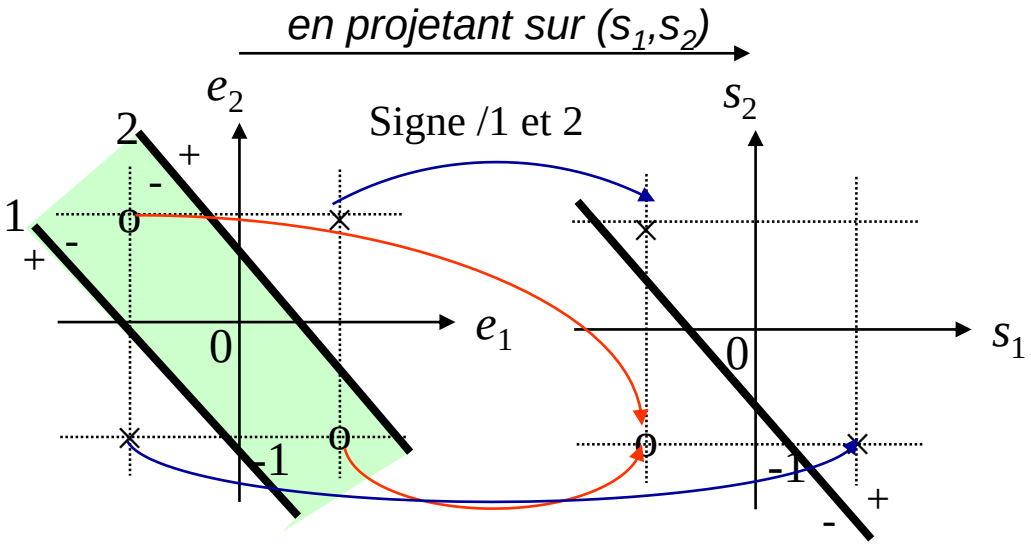
Graphiquement

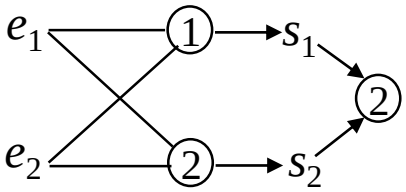


Limite du modèle simple

le OU exclusif

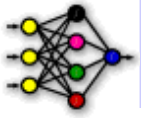
3. Maintenant, il faut trouver une seule sortie



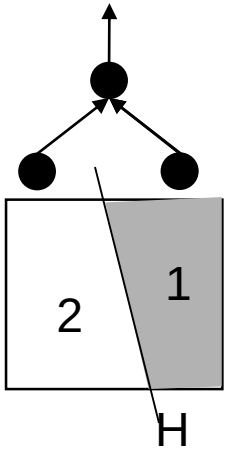


On y arrive mais recours à 3 couches de neurones

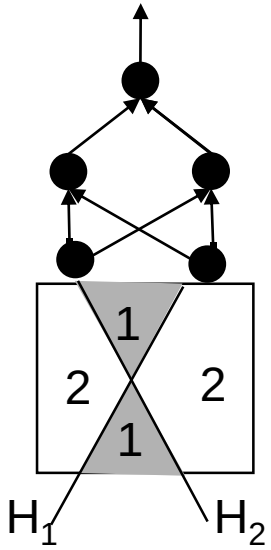
Vers des réseaux de neurones



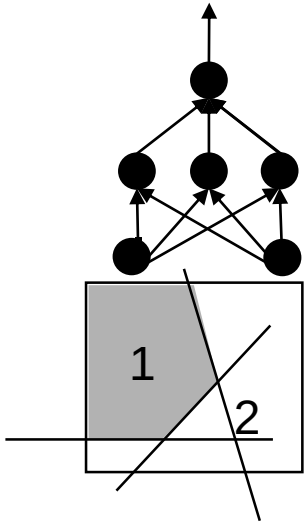
- Intérêt : résoudre des séparations non linéaires



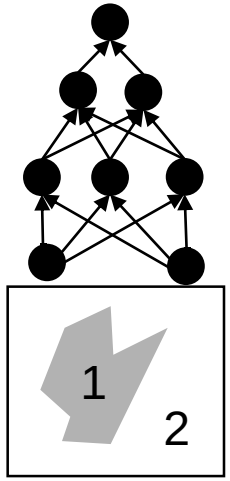
1 hyperplan



1 neurone,
1 hyperplan

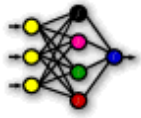


1 neurone,
1 hyperplan



etc.

Le Perception Multi-Couches (PMC)

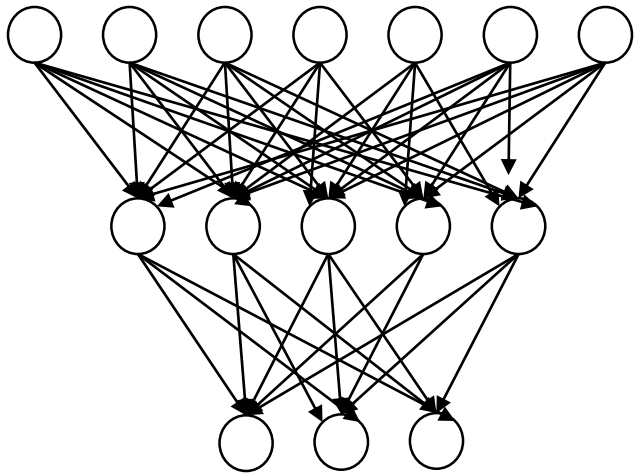


- Structure :

Couche d'entrée

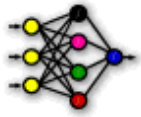
Couche cachée

Couche de sortie



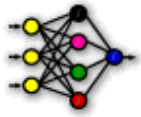
dit complètement connecté

Le PMC



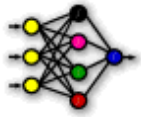
- Comment choisir une telle structure ?
  - Par expérience
  - Par les données :
    - Couche d'entrée :
      - Prendre un nombre de neurones égal à la taille de la donnée
    - Couche cachée :
      - Faire varier le nombre de neurones de manière à obtenir une meilleure précision
    - Couche de sortie
      - Le nombre est égal au nombre de classes que l'on cherche à distinguer

Le PMC



- Apprentissage
  - Supervisé
  - Basé sur la correction des erreurs de classement des données
    - Erreurs produites au niveau de chaque nœud par la fonction d'activation f
  - Correction
    - Ne pouvant constater l'erreur qu'à la sortie du réseau
      - On remonte le réseau par la fin en essayant de trouver les responsables des erreurs pour les corriger à la hauteur de leurs erreurs
      - Ensuite, cette correction est répercutée sur les précédents
    - Les erreurs dépendent des poids et des entrées
      - Erreur = f(ω,E) où ω sont les poids et E, les entrées
    - On corrige les poids ω en chaque nœud de manière à ce que à la prochaine re-rentrée des données, l'erreur générale soit moindre

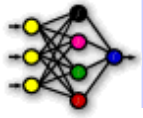
Le PMC



- Apprentissage : en d'autres termes
  - Pour la couche de sortie, on peut appliquer l'apprentissage du Perceptron car on est près de la sortie et l'on sait exactement ce que l'on veut obtenir en sortie : on peut corriger en conséquence
  - Pour les autres couches :
    - Problème : car pas de contact direct avec la solution (sortie)
    - Solution
      - On fait une estimation de l'erreur effectuée pour chaque neurone de la couche cachée
      - A partir de cette estimation, on corrige les poids
      - Cette solution s'appelle
        - » rétro-propagation du gradient

Le PMC

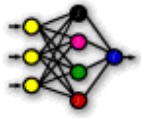
Rétro-propagation du gradient



- Algorithme d'apprentissage : 2 mouvements
  - A partir de l'erreur connue, calculée à la sortie :
    1. Mouvement de droite à gauche
       - On établit la "responsabilité" ou la contribution de chaque neurone des couches cachées dans la production de cette erreur
    2. Mouvement de gauche à droite
       - On corrige les poids à partir de l'estimation dans chaque neurone, sans utiliser de nouvelles données
       - L'entrée de nouvelles données se fera après, une fois les poids corrigés

Le PMC

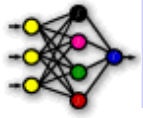
Rétro-propagation du gradient



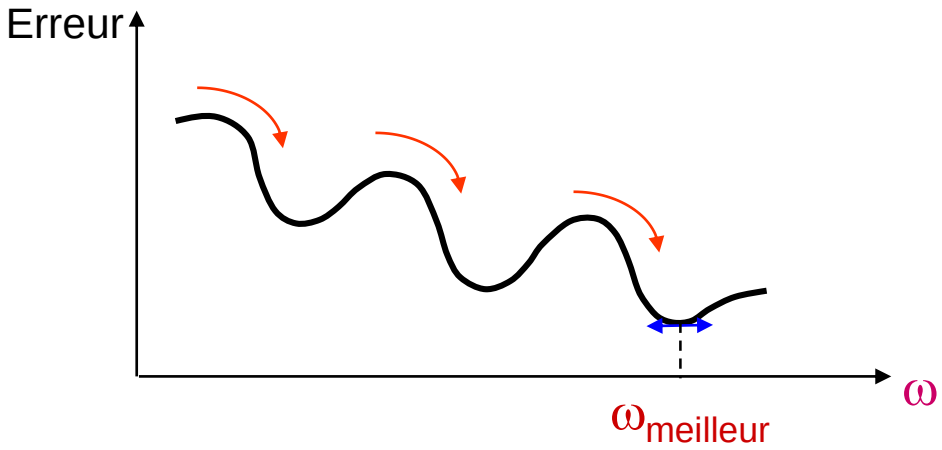
- Comment corriger ?
  - On se base sur la fonction générale de l'erreur f(ω,E)
    - On suit la pente de la fonction
      - pour orienter, à chaque entrée, la correction des poids dans le bon sens, conduisant à une erreur encore plus petite de f, jusqu'à atteindre le minimum
    - On utilise un "coefficient de correction" μ
      - qui permet de ne pas corriger abusivement :
        - si petit, risque de stagner,
        - si grand : risque de ne pas converger, car modifications seront trop fortes

Le PMC

Rétro-propagation du gradient

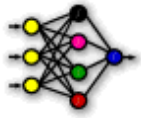


- But de la correction : recherche du minimum global du réseau
  - Les sauts et les bosses correspondent aux améliorations successives faites à chaque étape sur l'ensemble des données
  - À chaque fois, on essaie de se sortir d'un minimum local pour atteindre le meilleur vecteur poids $\omega_{meilleur}$

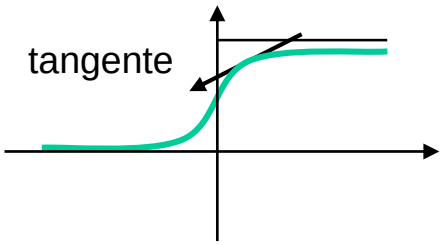


Le PMC

Rétro-propagation du gradient

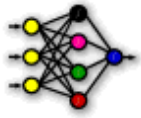


- Réduction de l'erreur
  - On ne sait pas réduire l'erreur globalement pour le réseau
    - Donc, on le fait neurone par neurone
  - Principe :
    1. Examiner la réponse de la fonction d'activation du neurone
    2. Estimer l'influence de cette erreur sur la correction des poids en entrée,
       - Loin de 0, i.e. dans une zone stable, alors correction faible
       - Près de 0, zone instable, faut en sortir au plus vite, en corrigeant drastiquement
         - Cela correspond à suivre la pente (tg) de la fonction f
    3. On corrige l'erreur en la pondérant par la pente

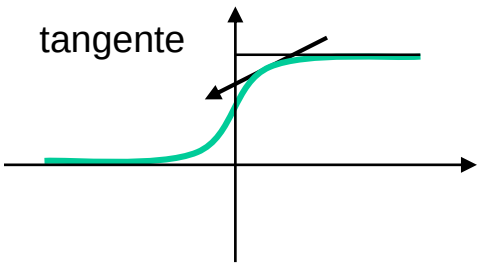


Le PMC

Rétro-propagation du gradient

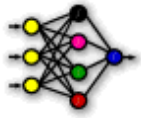


- Autre explication : réduction de l'erreur au niveau d'un neurone
  1. On pondère cette erreur d'après la réponse de sa fonction d'activation (sigmoïde) qui se traduit ainsi :
     - Si on est proche de 0 ou de 1, on a bien appris, la correction doit rester faible
     - Si la réponse n'est pas franche (proche de 0.5), le neurone hésite, on va donc le pousser vers une décision (0 ou 1)
       - » Pour ce faire, on pondère par la pente de la fonction f (tangente)
  2. On corrige les poids en fonction de cette erreur pondérée, et des contributions des neurones en entrée



Le PMC

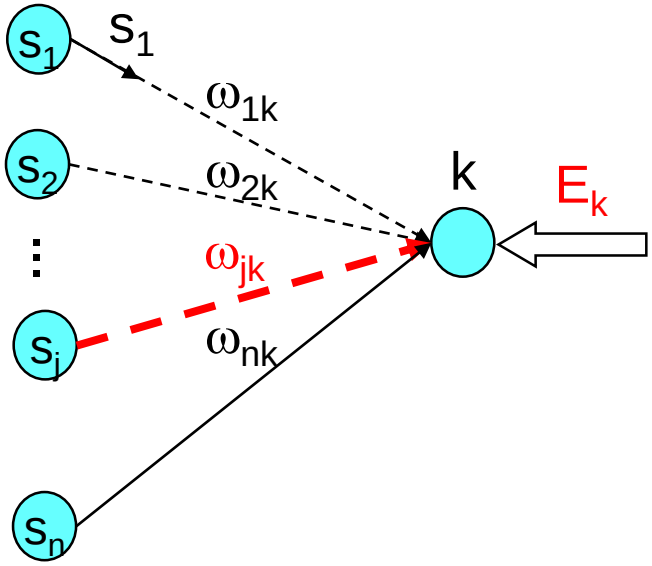
Rétro-propagation du gradient



- **Correction des poids**

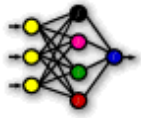
  $\Delta\omega_{jk} = \mu \times s_j \times \underline{err}$ où $\underline{err} = \delta_k = f'(s_k) \times E_k$

  - μ : coefficient de correction
  - s_j : sortie du neurone j, en entrée de k, la correction est proportion. à son importance
  - $\underline{err}$: erreur pondérée
  - $f'(s_k)$: dérivée de la fonction du neurone
  - E_k : erreur commise par le neurone, la correction sera proportionnelle à son importance



Le PMC

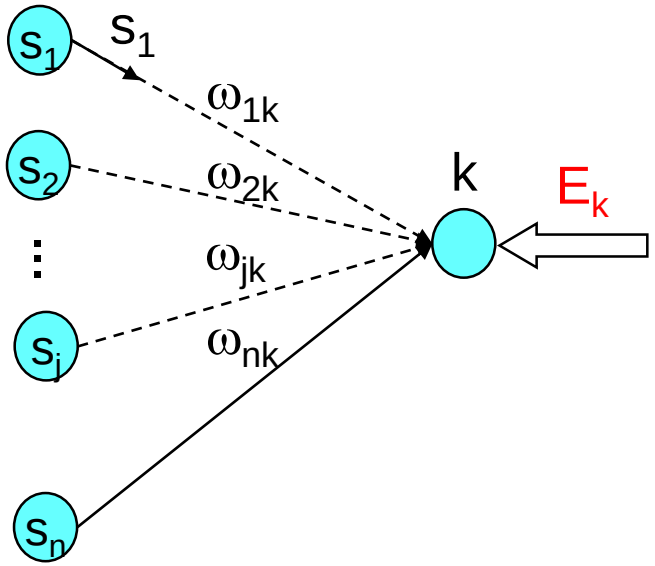
Rétro-propagation du gradient



- Calcul de l'erreur E_k :
  - Premier cas : E_k : est le neurone de sortie

    E_k = sortie désirée – s_k

    - L'erreur est visible : égale à la différence entre la valeur désirée et la valeur obtenue (réelle)

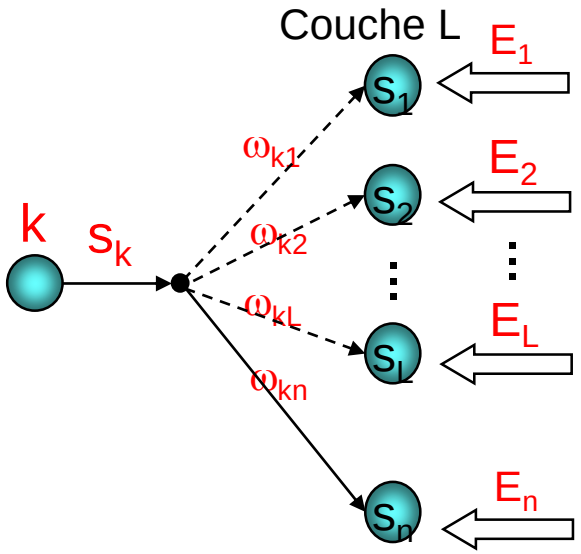


Le PMC

Rétro-propagation du gradient

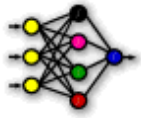
- Calcul de l'erreur E_k :
  - Cas général : pour un neurone interne k
  - L'erreur du neurone k s'est propagée à tous les neurones en aval (n)
  - E_k = somme des erreurs provenant des neurones en aval, pondérées par :
    - les poids reliant le neurone k à chacun des neurones en aval
      - accentuant la correction, en réponse à l'importance de l'erreur qu'il aura produite en aval
    - la pente de sa fonction d'activation
      - pour corriger cette erreur dans le bon sens

$$\delta_k = f'(s_k).\Sigma_L \omega_{kL}.\delta_L$$



Le PMC

Rétro-propagation du gradient



- L'algorithme
  - Étape 1 : calcul de l'erreur pour la couche de sortie k :

    $\delta_k = f'(s_k).E_k$, E_k =(sortie désirée - sortie réelle) $\forall k \in$ couche de sortie
  - Étape 2 : calcul de l'erreur pour tous les neurones des couches cachées j :

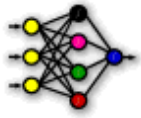
    $\delta_j = f'(s_j).\Sigma_L\, \omega_{jL}.\delta_L$, $\forall L \in$ couche suivante
  - Étape 3 : modification des poids

    $\Delta\, \omega_{jL} = \mu \times s_j \times \delta_L$

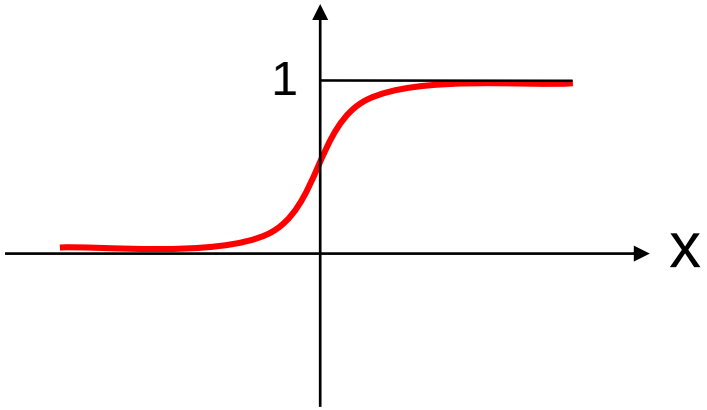
    $\omega_{jL}(t+1) = \omega_{jL}(t) + \Delta\omega_{jL}$

Le PMC

Rétro-propagation du gradient



- Pourquoi utilise-t-on la fonction sigmoïde ?



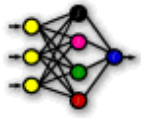
$$f(x)=\frac{1}{1+e^{-x}}$$

$$f'(x)=f(x).(1-f(x))$$

- Sa dérivée :
  - L'intérêt de la sigmoïde est que la dérivée se calcule directement à partir de la fonction elle même
    - Or, on a déjà calculé la valeur de la fonction lors de la propagation

Le PMC

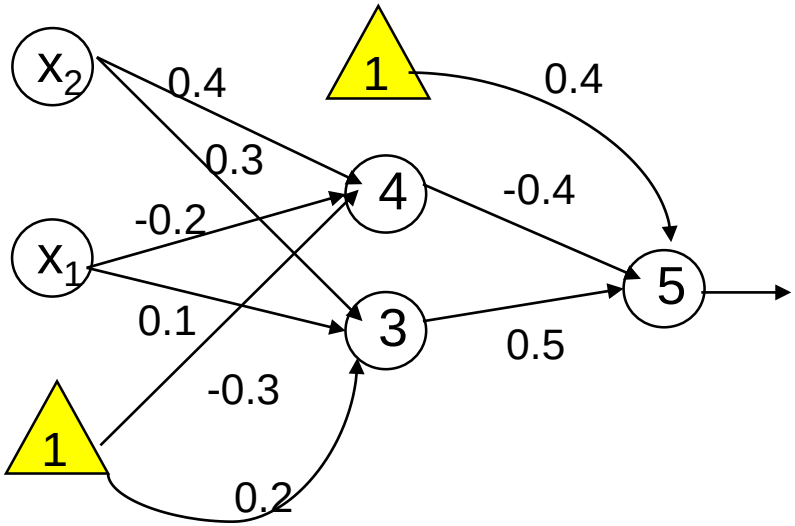
Rétro-propagation du gradient



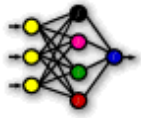
- Comment choisir le pas de modification des poids (μ) ?
  - Dans l'algorithme standard
    - La valeur du μ est entre 0 et 1, mais il n y a pas de règle pour le choix
    - Le choix est empirique (expérimental)
    - Pour résoudre les problèmes de "zigzag" et d' "engouffrement", on peut cependant proposer une heuristique :
      - Soit on commence avec un μ grand et pendant l'apprentissage, on diminue cette valeur petit à petit
      - Soit on prend 2 valeurs (une petite et une grande) qu'on alterne durant l'apprentissage

Le PMC

- Exemple :
  - Soit le réseau à 2 entrées x1 et x2 avec une couche cachée (composée de 2 neurones 3 et 4) et une seule sortie (5). Chaque couche contient un biais (neurone permettant d'intégrer le calcul du seuil S dans la somme pondérée, évitant de le faire manuellement). Ce neurone a toujours comme entrée 1



Le PMC



- Problème à résoudre : XOR
  - Cela veut dire qu'en prenant, par exemple, comme vecteur d'entrée x =(1,1), le résultat doit être égal à 0 (voir détail du calcul ci-dessous)

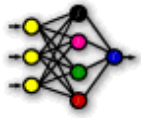
x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3	0.2+ 0.1×1+0.3 ×1=0.6	$1/(1+e^{-0.6}) \approx 0.65$
4	-0.3+ -0.2×1+0.4 ×1= -0.1	$1/(1+e^{0.1}) \approx 0.48$
5	0.4+ 0.5×0.65-0.4 ×0.48= 0.53	$1/(1+e^{-0.53}) \approx 0.63$

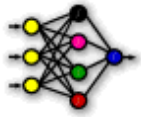
Résultat obtenu

Le PMC



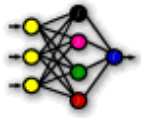
- Correction (suite de l'apprentissage)
  - Retour sur l'exemple
  - On va essayer par correction de rabaisser la valeur de sortie (0.63) pour l'entrée (1,1)
  - On commence à appliquer l'algorithme :
    - Pour chaque couche, on calcule le δ_j
    - (Étape 1) 1er cas : on est sur le neurone 5 de la couche de sortie
      - On calcule $\delta_5 = (0-0.63)\times 0.63\times(1-0.63) = -0.147$
      - On calcule $\Delta\omega_{05}$, $\Delta\omega_{35}$, $\Delta\omega_{45}$ à partir de δ_5 en fixant la valeur de $\mu=1$
      - $\Delta\omega_{05} = -0.147 \times 1 \approx -0.147$
      - $\Delta\omega_{35} = -0.147 \times 0.65 \approx -0.1$
      - $\Delta w_{45} = 0.48 \times -0.147 \approx -0.07$

Le PMC



- Correction (suite de l'apprentissage)
  - (Étape 2) 2ème cas : on est sur la couche cachée, on commence par 4
    - $\delta_4 = y_4 \times (1- y_4) \times \delta_5 \times \omega_{45} = 0.48 \times (1-0.48) \times -0.147 \times - 0.4 \approx 0.015$
    - $\Delta\omega_{14}$, $\Delta\omega_{24}$, $\Delta\omega_{04}$ à partir de δ_4 en fixant $\mu=1$
      - $\Delta\omega_{14} = 0.015 \times 1 \approx 0.015$
      - $\Delta\omega_{24} = 0.015 \times 1 \approx 0.015$
      - $\Delta\omega_{04} = 0.015 \times 1 \approx 0.015$
    - $\delta_3 = y_3 \times (1- y_3) \times \delta_5 \times \omega_{35} = 0.65 \times (1-0.65) \times -0.147 \times 0.5 \approx -0.017$
    - $\Delta\omega_{13}$, $\Delta\omega_{23}$, $\Delta\omega_{03}$ à partir de δ_3 en fixant la valeur de $\mu=1$
      - $\Delta\omega_{13} = -0.017 \times 1 \approx -0.017$
      - $\Delta\omega_{23} = -0.017 \times 1 \approx -0.017$
      - $\Delta\omega_{03} = -0.017 \times 1 \approx -0.017$

Le PMC



- Correction (suite de l'apprentissage)
  - Pour chaque connexion ω_{ij} on ajuste (Étape 3) les poids selon la formule $\omega_{ij} = \omega_{ij} + \Delta\omega_{ij}$

$\omega_{05} = \omega_{05} + \Delta\omega_{05} = 0.4 - 0.147 \approx 0.25$

$\omega_{35} = \omega_{35} + \Delta\omega_{35} = 0.5 - 0.1 \approx 0.4$

$\omega_{45} = \omega_{45} + \Delta\omega_{45} = -0.4 - 0.07 \approx -0.47$

$\omega_{03} = \omega_{03} + \Delta\omega_{03} = 0.2 - 0.017 \approx 0.183$

$\omega_{13} = \omega_{13} + \Delta\omega_{13} = 0.1 - 0.017 \approx 0.083$

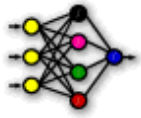
$\omega_{23} = \omega_{23} + \Delta\omega_{23} = 0.3 - 0.17 \approx 0.283$

$\omega_{04} = \omega_{04} + \Delta\omega_{04} = -0.3 + 0.015 \approx -0.285$

$\omega_{14} = \omega_{14} + \Delta\omega_{14} = -0.2 + 0.15 \approx -0.185$

$\omega_{24} = \omega_{24} + \Delta\omega_{24} = 0.4 + 0.15 \approx 0.415$

Le PMC



- Correction (suite de l'apprentissage)
  - Ensuite, avec les poids corrigés, on repasse le même vecteur (1,1) à travers le réseau, on peut remarquer une baisse de la valeur de sortie
  - On recommence en utilisant le même vecteur ou un autre : (0,0) ou (1,0) ou (0,1)

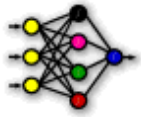
x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone formel j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3	$0.183+ 0.083\times1+0.283 \times1=0.55$	$1/(1+e^{-0.55}) \approx 0.63$
4	$-0.285+ -0.185\times1+0.415 \times1= -0.055$	$1/(1+e^{0.055}) \approx 0.51$
5	$0.25+ 0.4\times0.63-0.47 \times0.51= 0.26$	$1/(1+e^{-0.26}) \approx 0.56$

Résultat obtenu

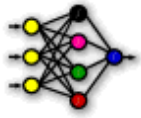
Conclusion sur Les réseaux de neurones



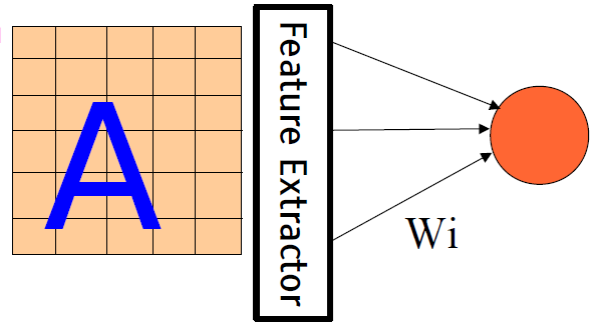
- Avantages
  - Classifieurs universels offrant :
    - Rapidité d'exécution
    - Robustesse des solutions, résistance au bruit des données
    - Facilité de développement
- Inconvénients
  - Le choix de l'architecture n’est pas évident
  - Le temps d'apprentissage peut être long
  - Présence de minima locaux de la fonction de coût

RN : autre inconvénient :

Le modèle de base est linéaire



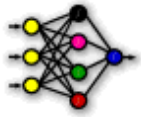
- Le perceptron
  - Est un classifieur linéaire au-dessus d'un vecteur de caractéristiques (calcule la décision par combinaison linéaire des échantillons)
  - L'extraction des primitives nécessite la connaissance de l'expert



$$y = sign\left(\sum_{i=1}^{N} W_i F_i(X) + b\right)$$

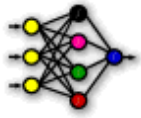
- Il a été observé que
  - l'utilisation de transformations non linéaires simples permettait d'améliorer de façon significative la précision des estimations obtenues à l'aide d'un système

Autre idée de l'apprentissage profond

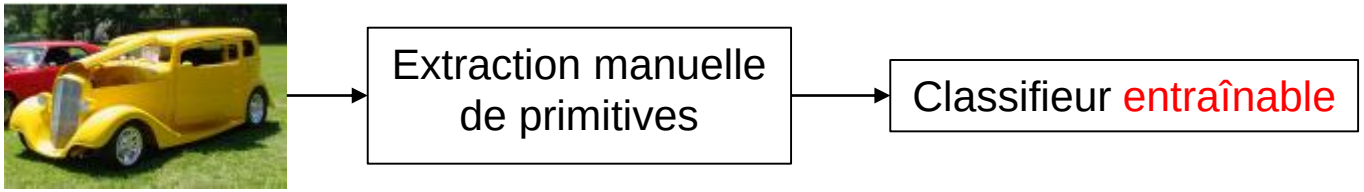


- Apprentissage des représentations :
  - Approche classique :
    - Pré-traitement des données brutes
    - Extraction « manuelle » de caractéristiques !
  - Idée : apprendre les caractéristiques à extraire
    - Problème considéré jusque-là comme trop difficile
    - Quelles caractéristiques ? liées aux tâches, aux données ?
    - Sensibilité aux variations (rotation, angle de vue, fréquence, etc.)

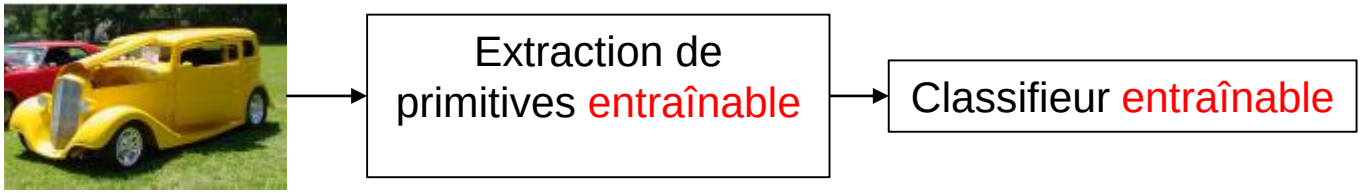
RN vs RP



- RN apprend uniquement le classifieur :
  - Caractéristiques fixes (ou noyau fixe) + classifieur entraînable

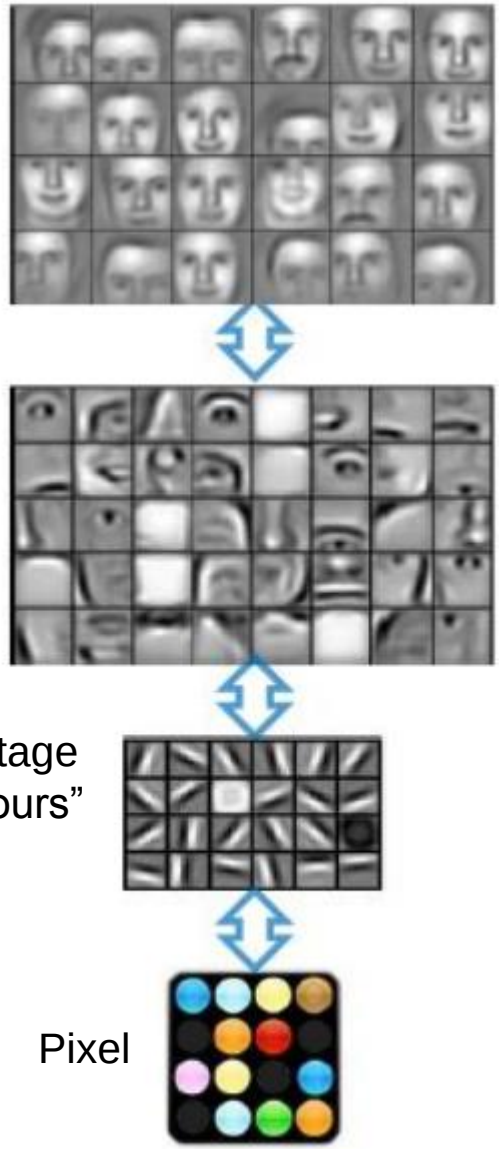


- RP pratique un apprentissage de bout en bout, un apprentissage de caractéristiques :
  - Caractéristiques apprises (ou noyau appris) + classifieur entraînable
  - Les caractéristiques et le classifieur sont appris ensemble

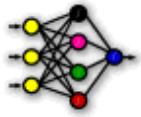


Autre idée de l'apprentissage profond

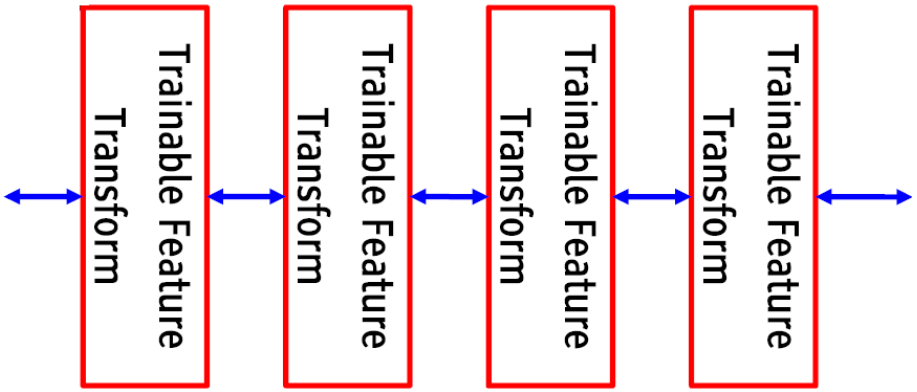
- Multiplication des couches
  - Les couches successives créent des abstractions de plus en plus haut niveau,
  - les couches basses apprennent les représentations



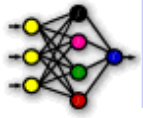
RP : étages d'abstraction



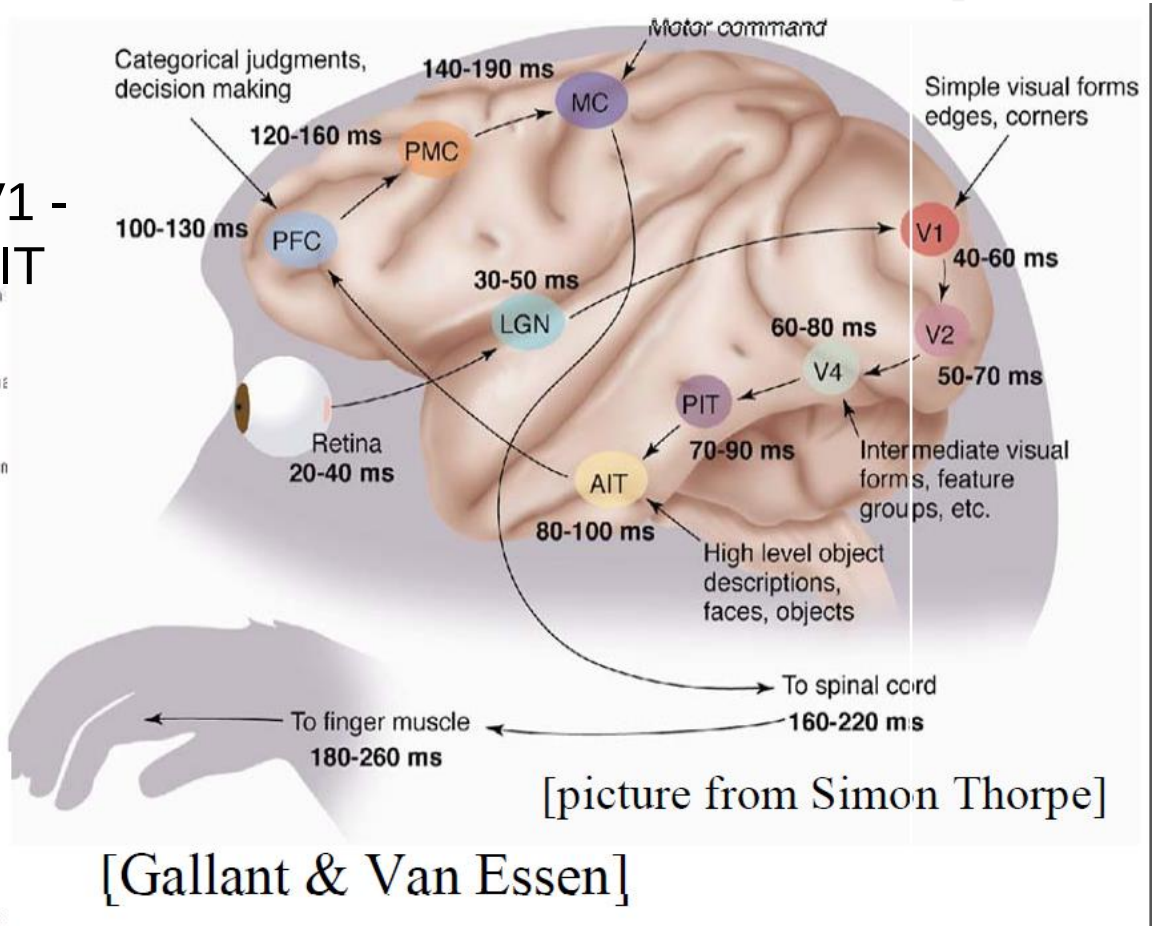
- Un modèle profond comporte une hiérarchie de représentations avec un niveau d'abstraction croissant
  - Chaque étape : sorte de transformation de fonctionnalité transformable
  - Plus facile de surveiller ce qui est appris et de guider la machine vers de meilleurs sous-espaces



Plusieurs modèles dans la nature
Ex: le modèle visuel est profond

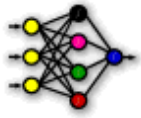


- Plusieurs étages :
  - Retina - LGN - V1 - V2 - V4 - PIT - AIT ....
- Utilise des représentations hiérarchiques
- de type modulaire, sans boucle : feed-forward

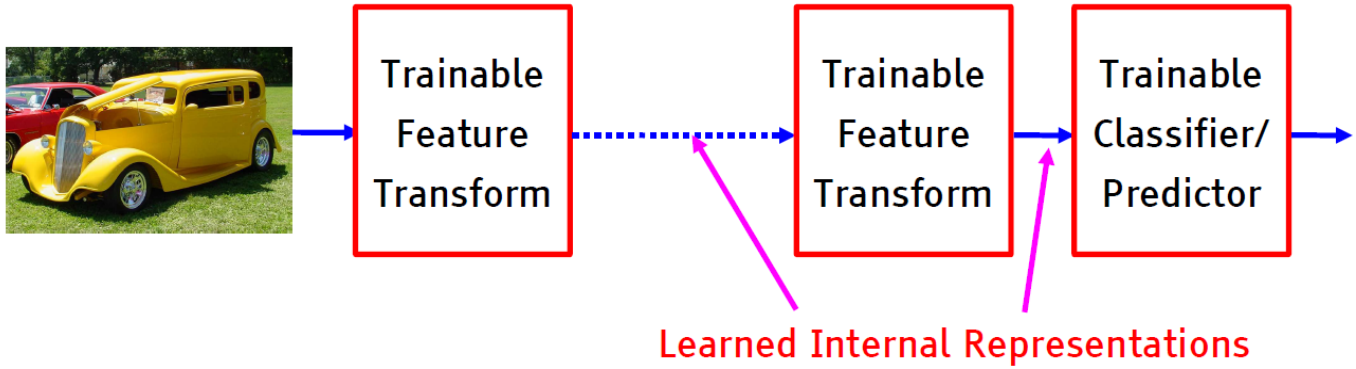


[Gallant & Van Essen]

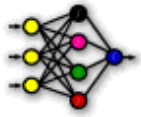
RP : apprentissage de bout en bout



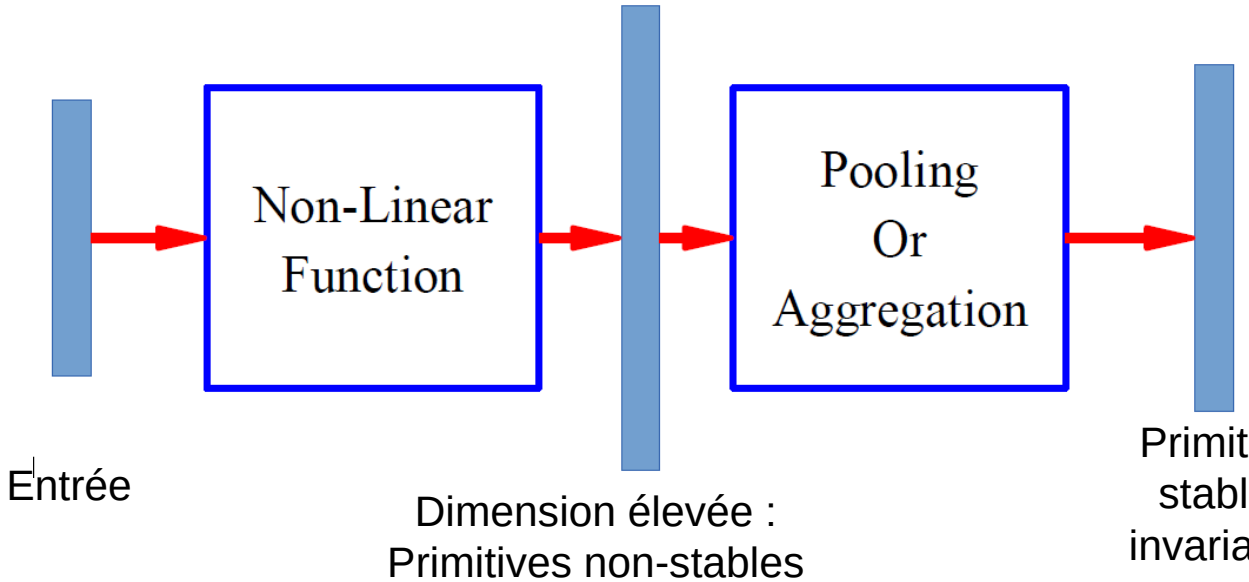
- Le modèle fait en sorte que tous ses modules puissent être appris et les conduit à apprendre des représentations appropriées
- Le modèle contient une hiérarchie des transformations de caractéristiques
  - Chaque module transforme sa représentation en entrée en un niveau supérieur
  - Les fonctionnalités de haut niveau sont plus globales et plus invariantes
  - Les fonctionnalités de bas niveau sont partagées entre les classes



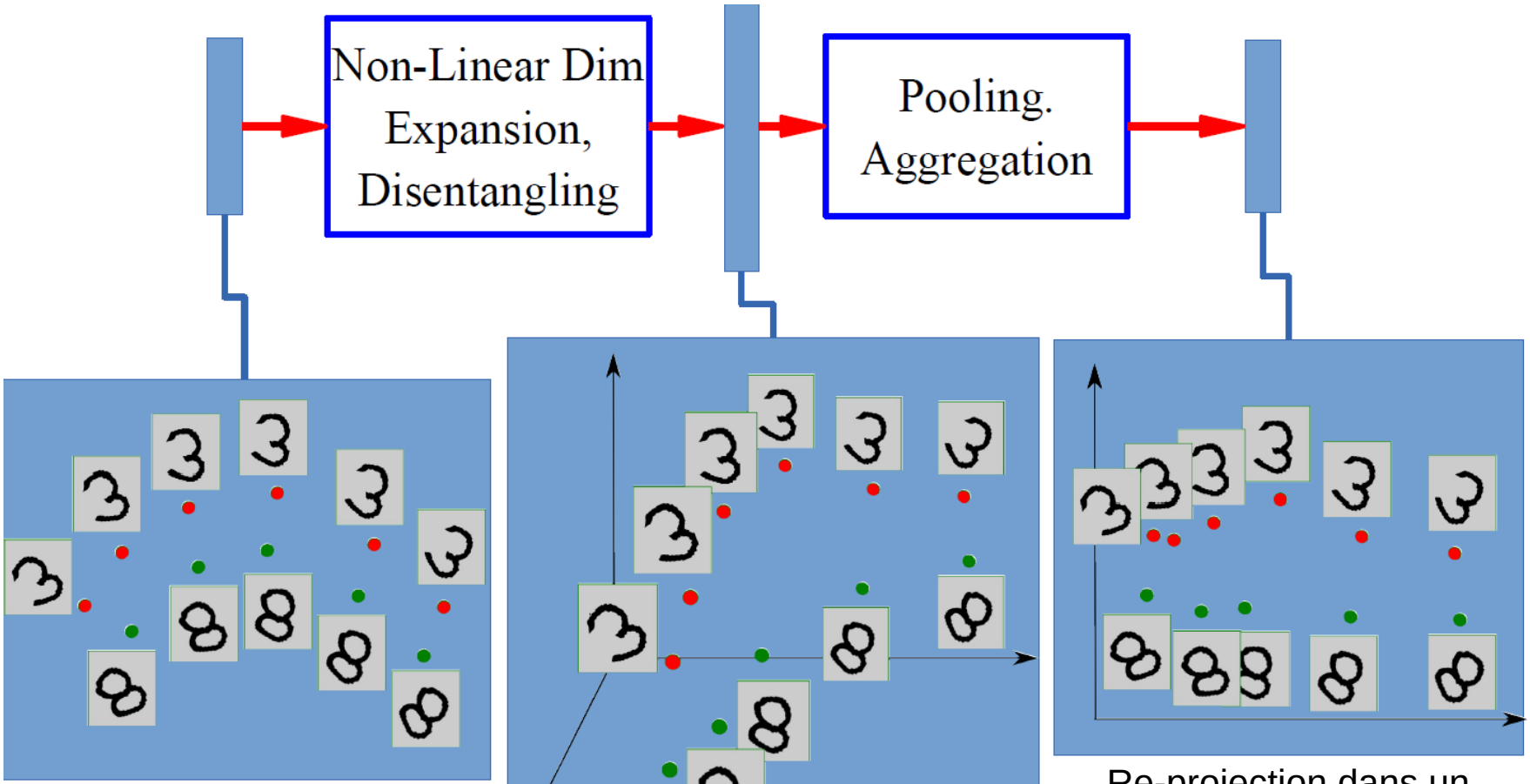
Idée de base pour l'extraction de primitives invariantes



- Intégrer l'entrée non-linéaire dans un espace dimensionnel élevé
  - Dans le nouvel espace, les choses qui étaient non séparables peuvent devenir séparables
- Rassembler les régions du nouvel espace ensemble
  - Rassembler les choses qui sont sémantiquement similaires (pooling)



Exemple de transformation non-linéaire

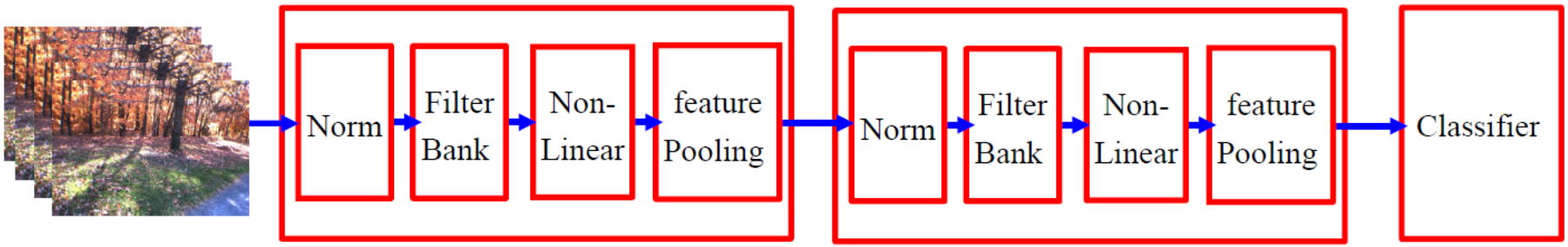


On ne peut pas séparer ces deux ensembles de chiffres par un séparateur linéaire : juste quelques pixels les séparent

Extension de l'espace à 3 D, en rajoutant une dimension liée à l'orientation et l'ouverture

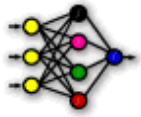
Re-projection dans un espace à 2D : les chiffres deviennent séparables linéairement

RP : Architecture générale



- Empiler plusieurs étapes de
  - [Normalisation → Banque de filtres → Non linéarité → Regroupement]
- Normalisation : de la variation du contraste
  - Soustractive: suppression moyenne, filtrage passe-haut
  - Divisive: normalisation du contraste local, normalisation de la variance
- Banc de filtres : extension de la dimension, projection sur une base
- Non-linéarité : sparsification, saturation, inhibition latérale ....
  - Rectification (ReLU), retrait des composantes, sigmoïde,…
- Pooling: agrégation sur l'espace ou le type de primitive

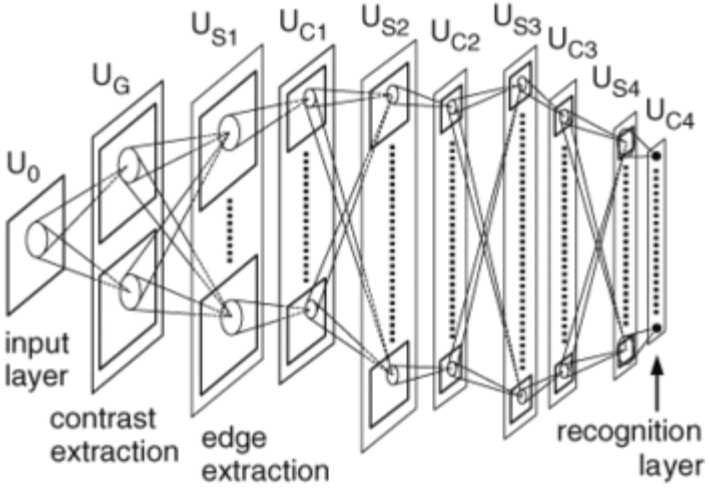
Un RP particulier : Convolutional Neural Network (CNN)



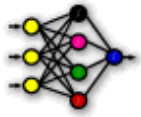
- Quatre approches expliquent son succès :
  - Convolution
  - ReLu, Pooling
  - GPU
  - Dropout
- Trois modèles marquants
  - Le Neocognitron (Hubel & Wiesel)
  - ImageNet
  - LeNet (LeCun)

Le modèle Neocognitron

- Présentation
  - Des travaux par Hubel & Wiesel en neurosciences ont inspiré Fukushima pour la création de ce modèle
  - Un modèle hiérarchique avec deux grands types principaux de neurones : les cellules simples et les cellules complexes
  - Des couches alternées composées de cellules simples, puis complexes, constituent le modèle

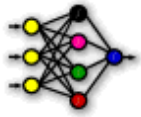


Le modèle Neocognitron



- Vision différente du modèle complètement connecté
  - L'entrée est une image
  - On observe des régions locales
  - Le modèle utilise des filtres de convolution pour extraire des contours dans ces régions
  - Les poids sont partagés entre les régions représentant les mêmes informations
  - On utilise le sous-échantillonnage (pooling) pour passer des informations de bas niveau à des informations de haut niveau

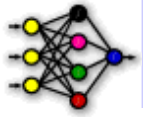
Le modèle Neocognitron



- Les cellules simples détectent les primitives locales
  - Elles réagissent à la présence d'un motif dans leur champ réceptif
    - Exemple : détecter la présence de petits bouts de contours (edges) dans une orientation particulière dans leur champ réceptif
  - Si le motif corrèle assez bien avec l'activité dans le champ réceptif, le neurone s'active
  - La réponse des cellules simples est apprise durant l'entraînement du modèle

Le modèle Neocognitron

(les cellules simples)

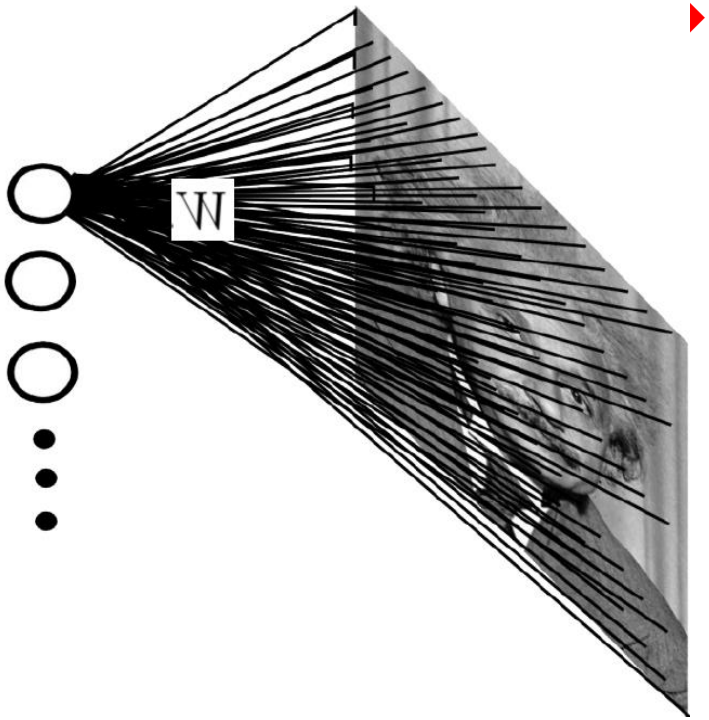


- Partage de poids
  - La détection de contour orientée à 45 degrés n'est pas quelque chose de spécifique à un neurone particulier, mais plutôt partagé par des neurones couvrant l'ensemble du champ visuel
  - On tente de détecter chaque motif appris à chaque position de l'image d'entrée, sans apprendre un motif séparé pour chaque position
  - Pour détecter les motifs, chaque couche de cellules simples est composée d'un certain nombre de filtres de convolution

Cette vision réduit les connections aux régions locales et donc la complexité

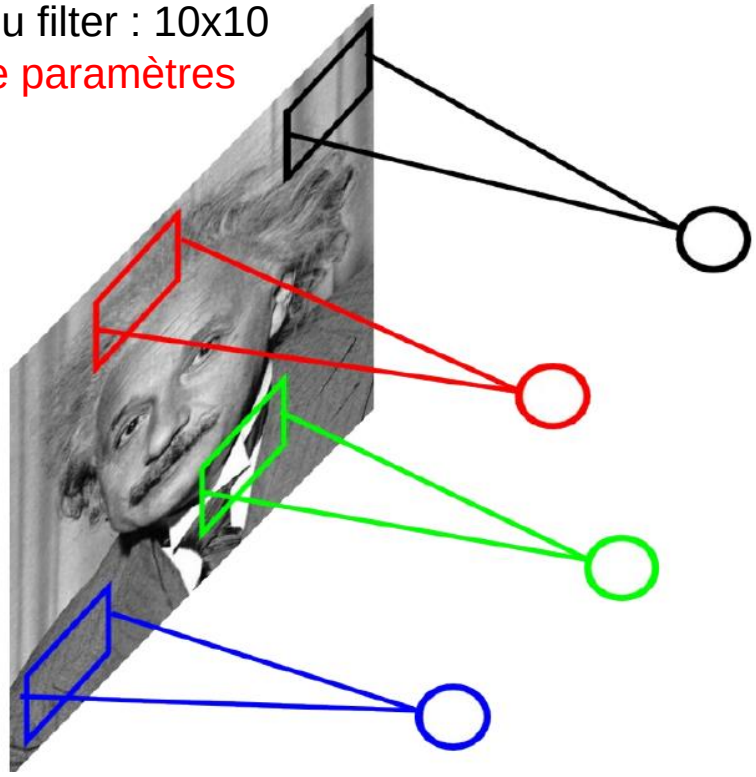
- L'entrée : 1000 x 1000 pixels
- 1M d'unités cachées

➔ 1B de paramètres



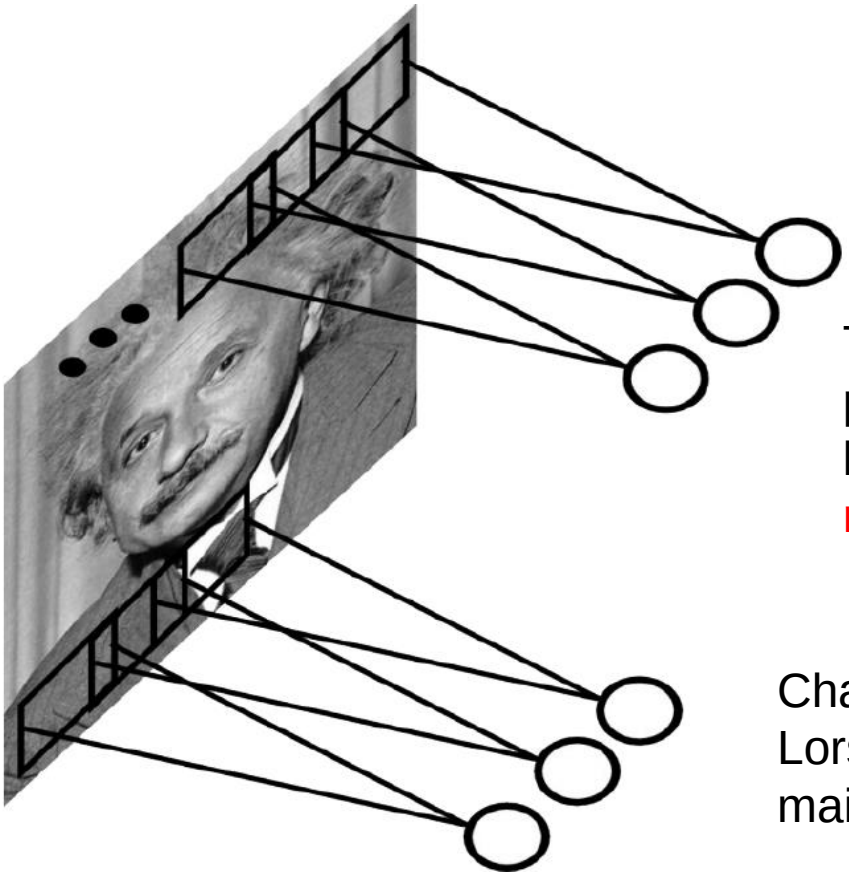
- L'entrée : 1000 x 1000 pixels
- 1M d'unités caches
- Taille du filter : 10x10

➔ 10M de paramètres



la corrélation spatiale est locale

Le modèle réutilise partout le même noyau



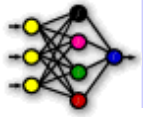
Car les bonnes caractéristiques (contours) peuvent se trouver partout dans l'image

Toutes les unités partagent les mêmes paramètres entre les différentes localisations : convolutions avec des noyaux appris

Changement de traitement équivariant : Lorsque l'entrée bouge, la sortie aussi, mais reste inchangée

Le modèle Neocognitron

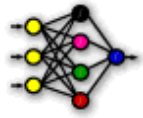
(les cellules simples)



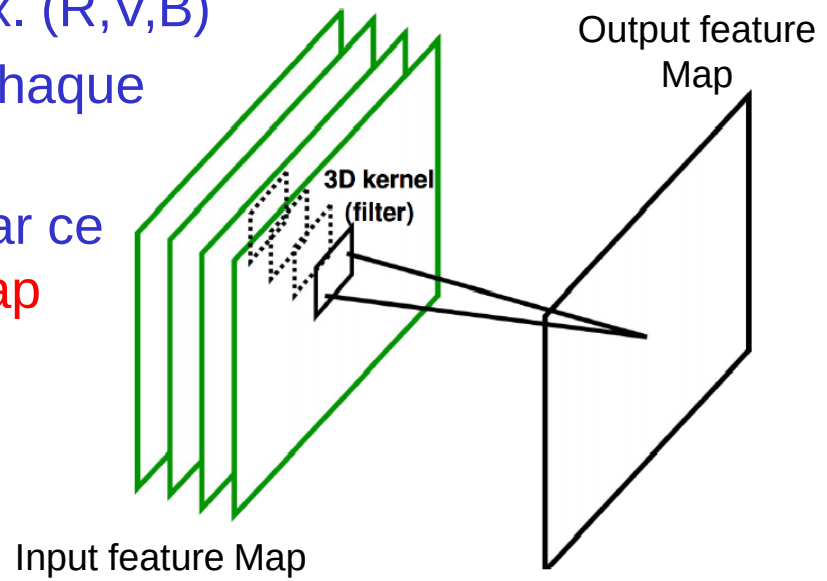
- Filtres de convolution, Feature Map
  - Chaque couche de cellules simples est composée d'un certain nombre de filtres de convolution
  - Pour appliquer le même motif à toute l'entrée, une opération de convolution du filtre fait le travail
  - On obtient, pour chaque filtre, une "feature map" de sortie donnant la réponse, à chaque position de l'entrée, de la somme des poids du filtre multipliés un à un aux pixels correspondants dans le voisinage de cette position

Le modèle Neocognitron

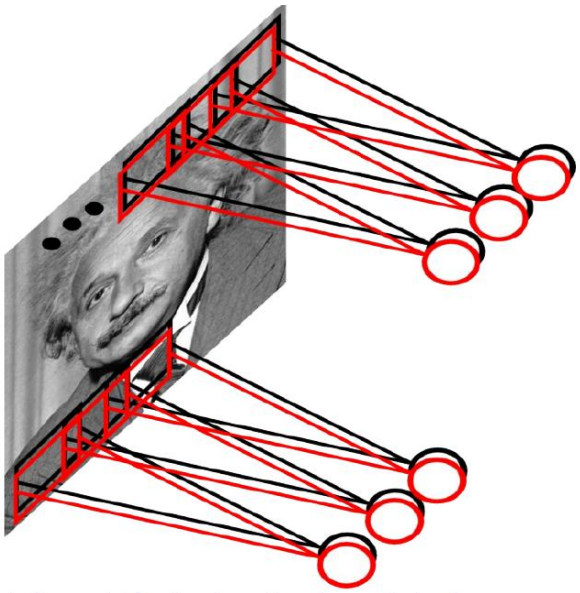
(les cellules simples)

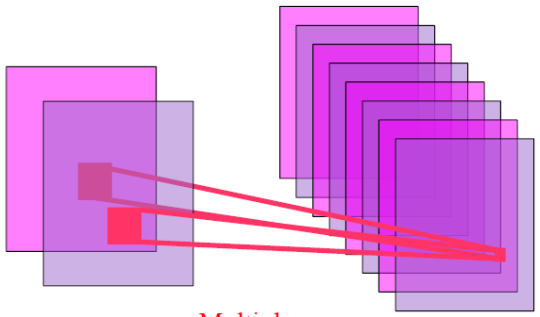


- Feature Map : définition
  - Si l'entrée a 3 canaux, par ex. (R,V,B)
  - 1 filtre k×k est appliqué sur chaque canal
  - La sortie de la convolution par ce filtre est appelée : feature Map



Convolutions multiples avec différents noyaux





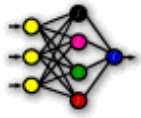
- Détecte plusieurs motifs à chaque emplacement
- La collection d'unités regardant le même patch est semblable à un vecteur de caractéristiques pour ce patch
- Le résultat est un tableau 3D, où chaque tranche est une carte de caractéristiques (feature map)

- L'entrée : 1000 x 1000 pixels
- 100 filtres
- Taille du filtre : 10x10

➔ 10K de paramètres

Le modèle Neocognitron

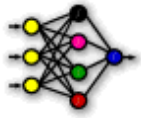
(les cellules simples)



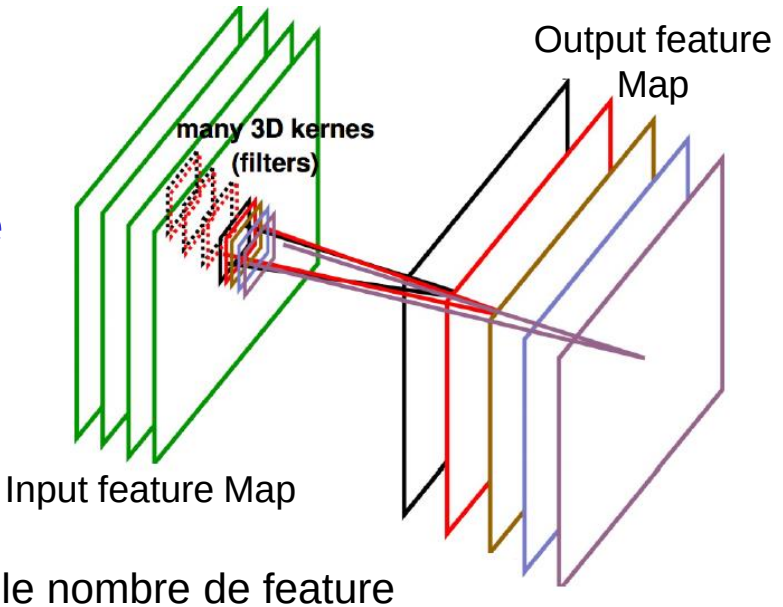
- Plusieurs feature maps
  - Le fait d'avoir plusieurs filtres permet d'avoir différents motifs pour une même couche, ce qui est bien nécessaire si on veut pouvoir détecter, par exemple, plusieurs orientations de bords (dans le cas où des détecteurs de bords sont appris)
  - On obtient donc plusieurs feature maps

Le modèle Neocognitron

(Plusieurs feature Maps)

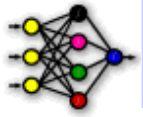


- Comment ?
  - Chaque filtre détecte les caractéristiques dans la sortie de la couche précédente
  - Donc, pour extraire une variété de caractéristiques, le CNN apprend plusieurs filtres



Remarque : le nombre de feature Maps de sortie est habituellement plus grand que le nombre de feature Maps d'entrée

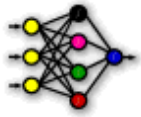
Le modèle Neocognitron



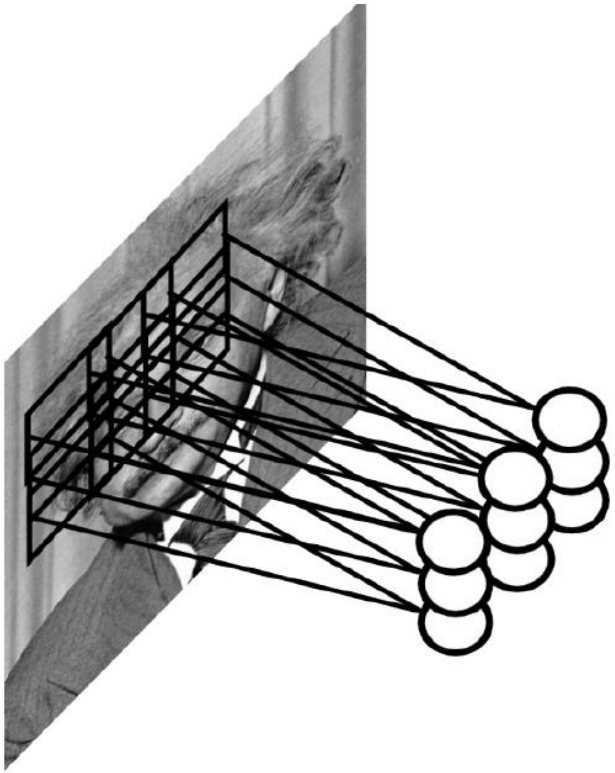
- Les cellules complexes "rassemblent" les sorties de cellules simples dans un voisinage rétinotopique
  - La réponse des cellules simples est floue sur la position du motif
    - S'il se déplace un peu, la réponse peut changer beaucoup
  - C'est là qu'interviennent les cellules complexes, en introduisant de l'invariance sur la position
  - Ainsi, une cellule complexe prend en entrée l'état de quelques cellules simples à champs réceptifs proches, le pool de la cellule complexe
  - Cette dernière s'active si l'une ou l'autre des cellules simples est active. On gagne donc une certaine indifférence ou invariance à la position du motif

Ce phénomène s'appelle :

Le pooling

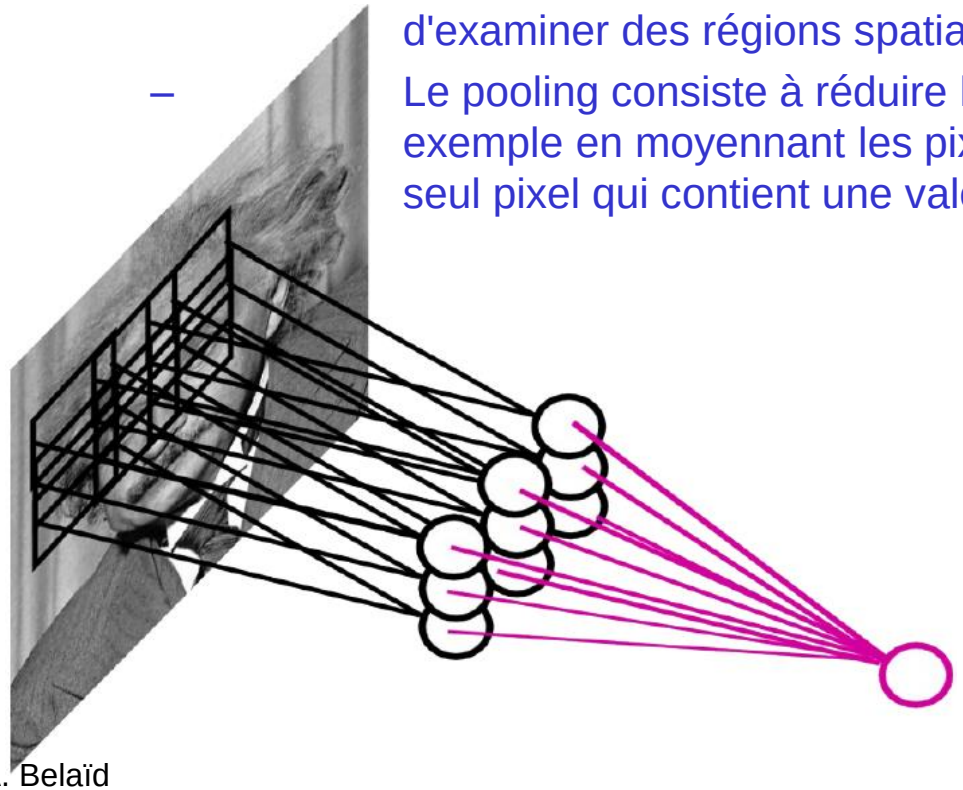


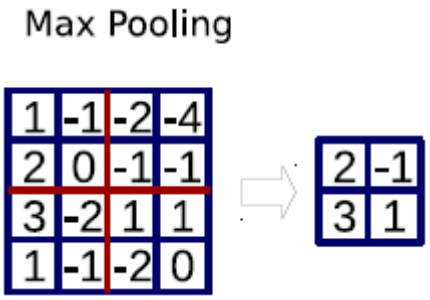
- Intérêts :
  1. Aggrégation : mise en commun de choses sémantiquement similaires
  2. Construction d'une invariance à la translation
- Agissement par les filtres
  - Supposons que le filtre est un détecteur d' »œil »
  - Q : comment pouvons-nous rendre la détection robuste à l'emplacement exact de l'œil?



Ce phénomène s'appelle
Le pooling

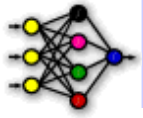
- Réponse
  - Le pooling regarde les réponses des filtres à différents emplacements, permet de pointer les emplacements des filtres qui donnent les mêmes réponses
  - Ainsi, il sous-échantillonne l'image, permettant à la couche suivante d'examiner des régions spatiales plus grandes
  - Le pooling consiste à réduire la résolution des images filtrées, par exemple en moyennant les pixels contigus (ex : 4 pixels deviennent 1 seul pixel qui contient une valeur moyenne)





Le modèle Neocognitron

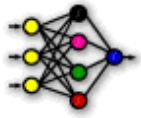
(Les cellules complexes)



- Enchaînement – connexion des couches
  - On connecte une première couche de cellules simples à l'entrée, puis des cellules complexes à ces cellules simples
  - Ceci peut être recommencé en prenant la sortie de ces premières cellules complexes comme entrées d'une prochaine couche de cellules simples…
  - Ainsi par les couches de cellules simples on pourra détecter des motifs de plus en plus haut niveau et complexité

Autre notion

ReLu (Rectified Linear Unit)



- C’est quoi ? C’est une couche de correction
  - Souvent, il est possible d'améliorer l'efficacité du traitement en intercalant entre les couches de traitement une couche qui va opérer une fonction mathématique (fonction d'activation) sur les signaux de sortie
  - On a notamment :
    - La correction ReLU : f(x) = max(0,x)
  - Cette fonction, appelée aussi "fonction d'activation non saturante", augmente les propriétés non linéaires de la fonction de décision et de l'ensemble du réseau sans affecter les champs récepteurs de la couche de convolution

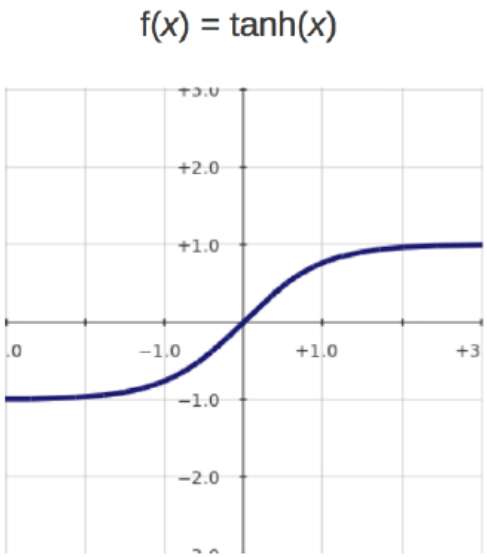
Architecture

RELU Nonlinearity

- Standard way to model a neuron

$f(x) = tanh(x)$ or $f(x) = (1 + e^{-x})^{-1}$

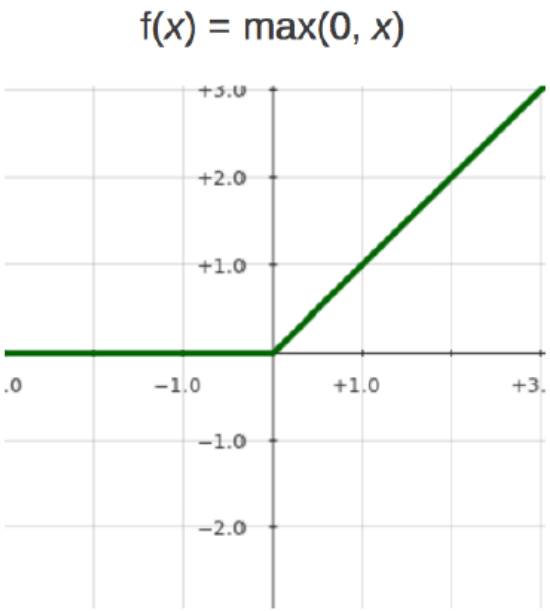
Very slow to train



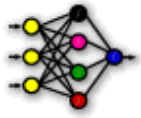
- Non-saturating nonlinearity (RELU)

$f(x) = max(0, x)$

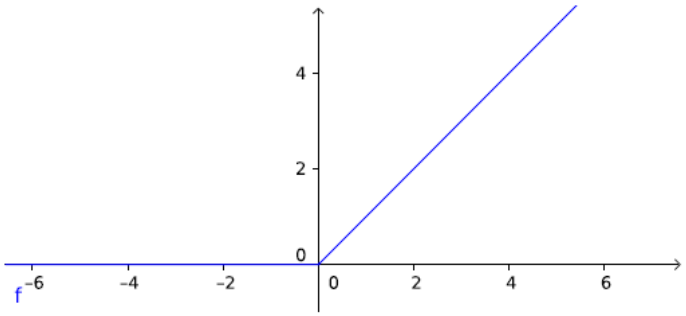
Quick to train



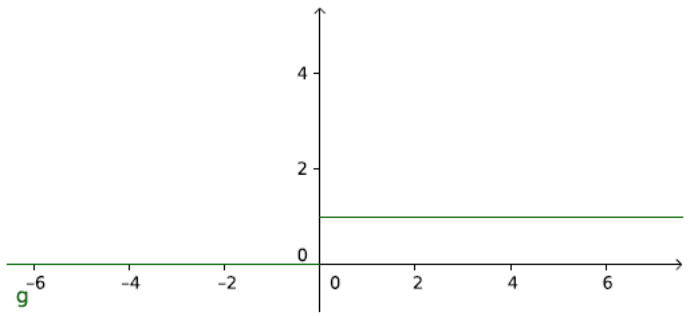
Propriétés intéressantes de la ReLU



- La ReLU a plusieurs propriétés intéressantes :
  - Biologiquement plausible : one-sided comparé à l'anti-symétrique tanh
  - Activations creuses: 50% des neurones ont une activation nulle après l'initialisation aléatoire des poids
  - Pas de vanishing gradient : La dérivé vaut 1 partout dans la portion positive
  - Complexité faible : comparaison

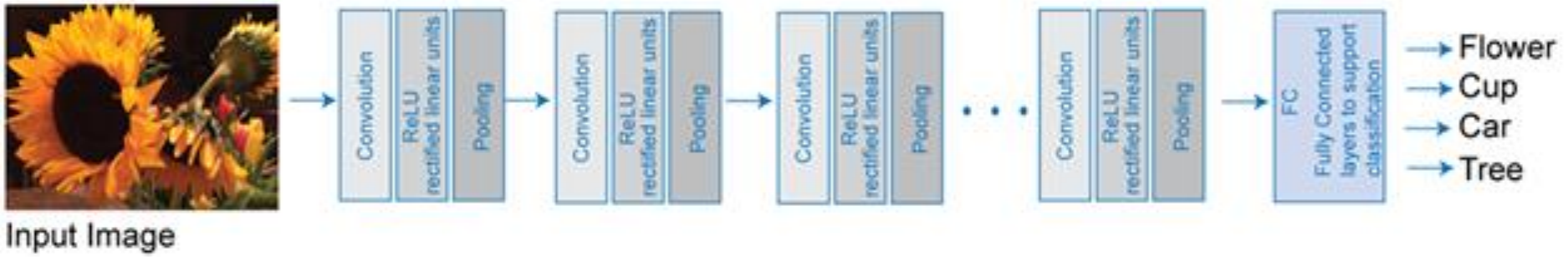


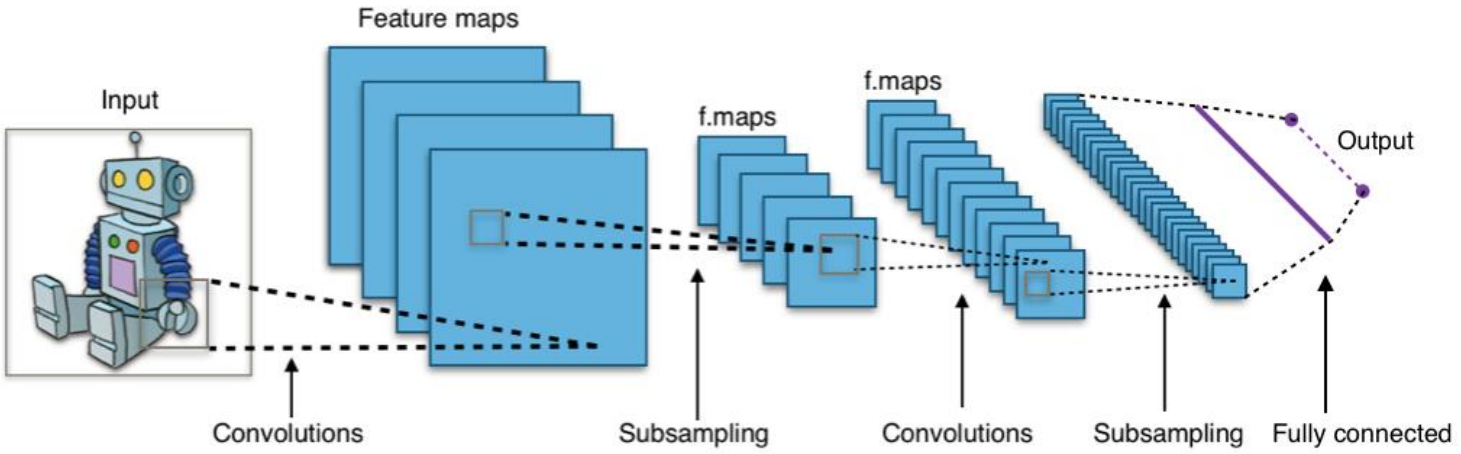
ReLU



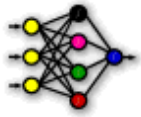
Sa dérivée

- La forme la plus commune d'un CNN
  - empile quelques couches Conv-ReLU, les suit avec des couches Pool, et répète ce schéma jusqu'à ce que l'entrée soit réduite dans un espace d'une taille suffisamment petite
  - La dernière couche entièrement connectée est reliée vers la sortie



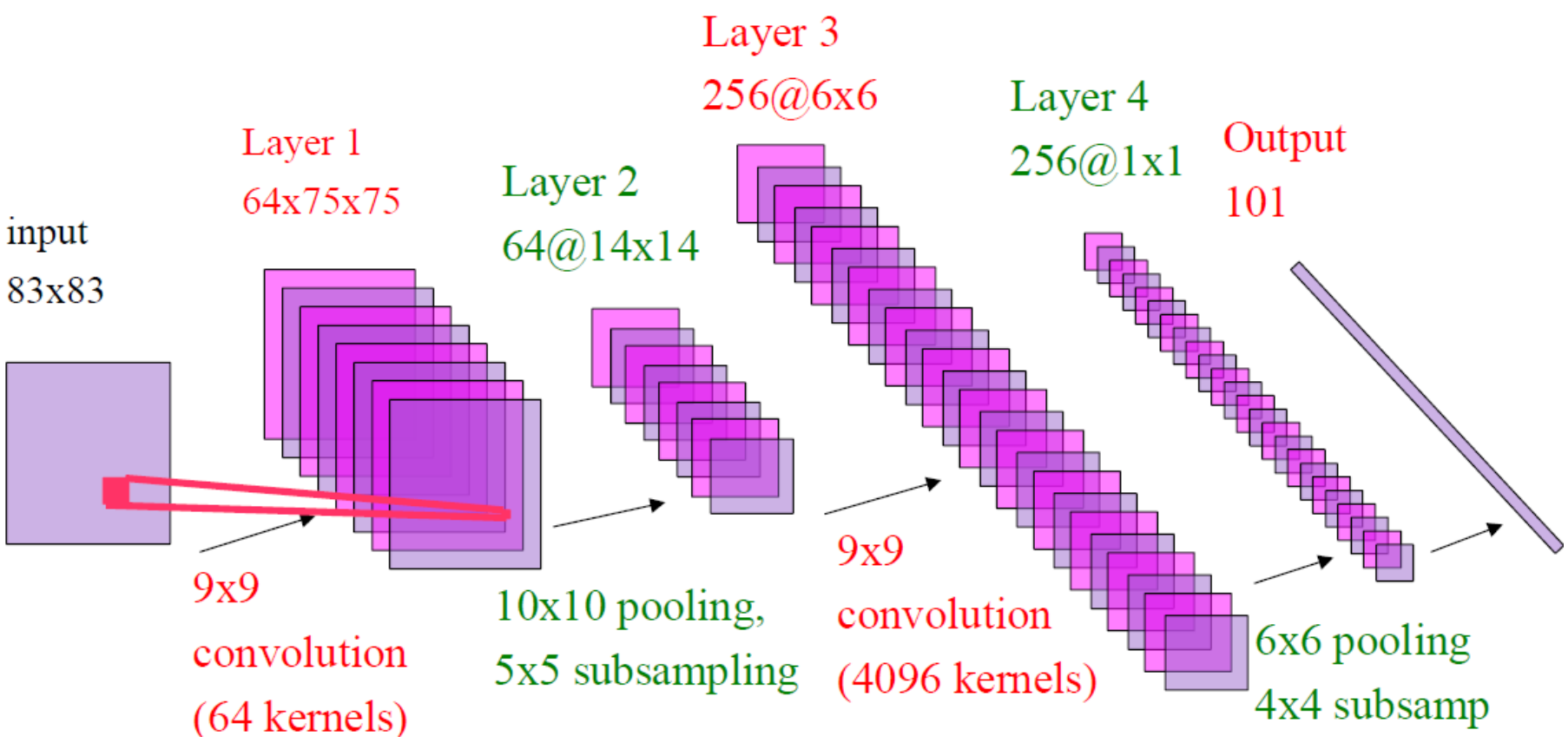


Exemples d'architectures CNN



- qui suivent le modèle précédent
  1. INPUT -> FC implémente un classifier linéaire
  2. INPUT -> CONV -> RELU -> FC
  3. INPUT -> [CONV -> RELU -> POOL] * 2 -> FC -> RELU -> FC Ici, il y a une couche de CONV unique entre chaque couche POOL
  4. INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL] * 3 -> [FC -> RELU] * 2 -> FC Ici, il y a deux couches CONV empilées avant chaque couche POOL
- Remarque
  - L'empilage des couches CONV avec de petits filtres de pooling (plutôt un grand filtre) permet un traitement plus puissant, avec moins de paramètres, mais avec l'inconvénient de demander plus de puissance de calcul (pour contenir tous les résultats intermédiaires de la couche CONV)

ConvNet

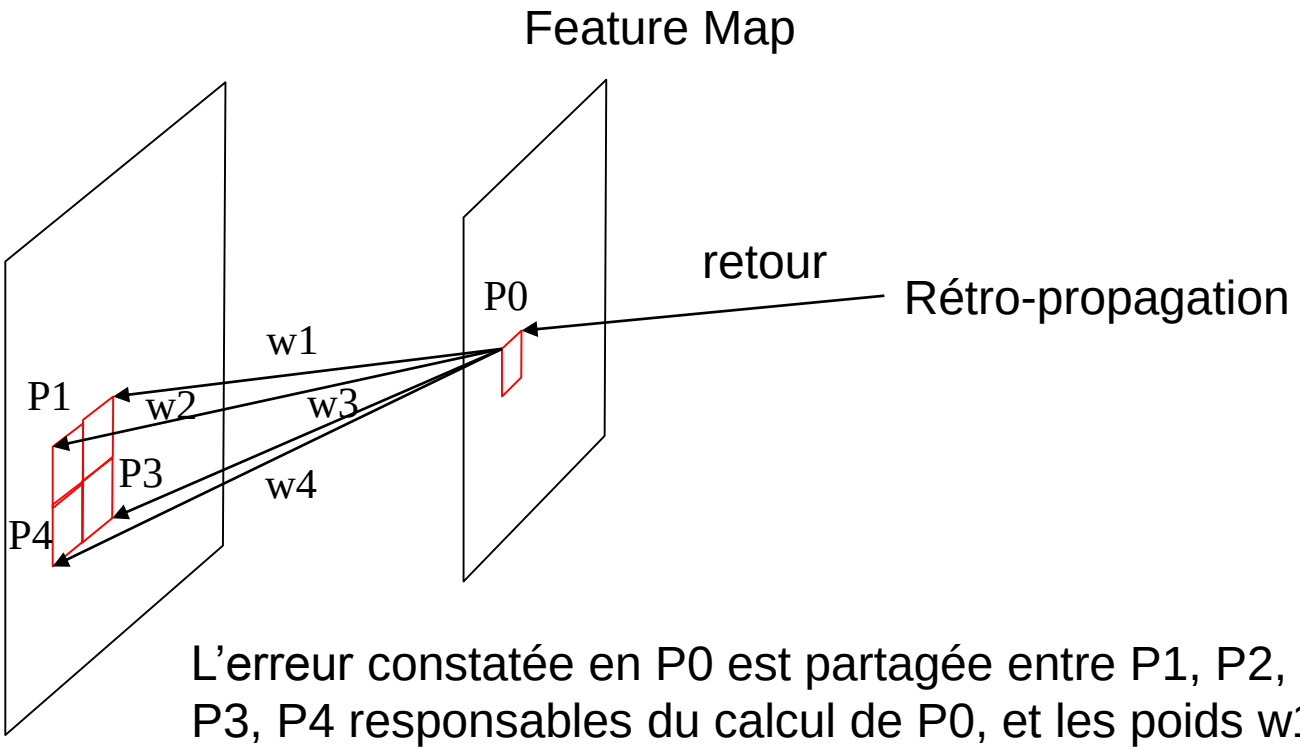


- Non-linéarité : redressement demi-onde, fonction de rétrécissement, sigmoïde
- Pooling : moyenne, L1, L2, max
- Apprentissage : supervisé (1988-2006), non supervisé + supervisé (2006-maintenant)

Question : comment entraîner ces réseaux ?

- Apprentissage supervisé
- On utilise l'algorithme de rétro-propagation
  - Dans une couche de convolution, chaque feature map a son propre noyau de convolution
  - Chaque noyau est initialisé aléatoirement au début, puis entraîné normalement, car chaque poids du noyau s'entraine comme un neurone
  - Gros détail lors de l'entrainement d'une couche de convolution :
    - Le noyau étant appliqué à toute la map, l'erreur est donc calculée en fonction de TOUS les neurones de la map

Question : comment entraîner ces réseaux ?



L'erreur constatée en P0 est partagée entre P1, P2, P3, P4 responsables du calcul de P0, et les poids w1, w2, w3 et w4 sont corrigés à la hauteur de l'erreur

Les coefficients du noyau sont ainsi corrigés