Machine Learning : Réseaux de neurones

D. Cornu

Automne 2019

Le cerveau comme inspiration

≪ There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears [...] wich is the squishy thing that your skull protects ≫

** Stephen Marsland*

Le cerveau comme inspiration

« There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears [...] wich is the squishy thing that your skull protects »

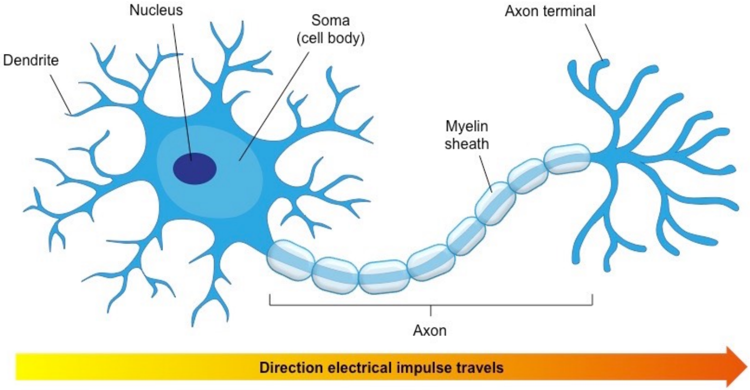
** Stephen Marsland*

Le cerveau fait exactement ce qui nous intéresse en science :

- Traiter des données bruitées ou incohérentes
- Fonctionner avec un nombre important de dimensions
- Donner un résultat le plus souvent correct
- Trouver une réponse en un temps très court
- Rester robuste malgré la perte de neurones avec l'âge

Neurone

Brique élémentaire (10^{11} dans le cerveau) $=$ une base pour reproduire l'apprentissage



Fait la somme de signaux d'entrée. Si celle-ci est suffisante il envoie un signal le long de son axone.

Réseau de neurones

Une Synapse est une connexion entre deux neurone (10^{14}).
Neurone = Unité de calcul simple "Tire" ou "Ne tire pas" (1 ou 0)

$\rightarrow$ **Calculateur massivement parrallèle de 10^{11} unités**

Réseau de neurones

Une Synapse est une connexion entre deux neurone (10^{14}).
Neurone = Unité de calcul simple "Tire" ou "Ne tire pas" (1 ou 0)

→ **Calculateur massivement parrallèle de 10^{11} unités**

Outils de l'apprentissage :

Les synapses représentent la **force** de la connexion entre deux neurones. Apprentissage = modification de ces liaisons
→ **plasticité**.

La **loi de Hebb** définit une règle d'apprentissage : la connexion entre deux neurones se renforce lorsqu'ils tirent au même moment
→ **conditionnement**.

Neurone Artificiel

Il est la brique élémentaire des réseaux qui seront construits.

Il est basé sur un **modèle mathématique** inspiré du neurone biologique.

C'est le modèle de McCulloch and Pitts.

Neurone Artificiel

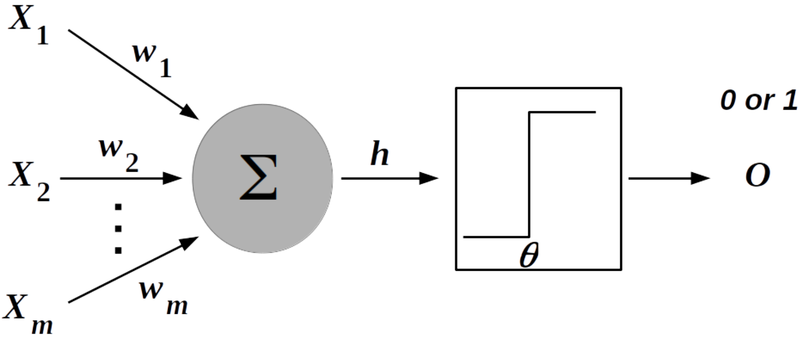
Il est la brique élémentaire des réseaux qui seront construits.
Il est basé sur un **modèle mathématique** inspiré du neurone biologique.
C'est le modèle de McCulloch and Pitts.

Il est constitué :

- D'un vecteur d'entrée X_i qui représente les différentes dimensions d'un même objet.
- D'un ensemble de poids w_i qui lient les entrées au neurone.
- D'une fonction de somme $h = \sum ...$ qui définie comment ces poids doivent êtres associés aux entrées correspondantes et envoyés vers le neurone.
- D'une fonction d'activation $g(h)$, qui définie si le neurone doit passer dans un état actif "1" ou non "0" en fonction du résultat de la somme précédente.

Neurone Artificiel

Il est la brique élémentaire des réseaux qui seront construits.
Il est basé sur un **modèle mathématique** inspiré du neurone biologique.
C'est le modèle de McCulloch and Pitts.



$$h = \sum_{i=1}^{m} X_i \omega_i \qquad O = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

Apprentissage supervisé $\rightarrow$ faire retrouver à un réseau la sortie attendue pour une entrée donnée.
Quel intérêt ? La sortie correcte est déja connue à l'avance ...

$\rightarrow$ **Généralisation :** Retrouver des "motifs" dans les données, et prédire les futures entrées dont le résultat est inconnu.

Comment changer les paramètres pour qu'un neurone apprenne ?

Réseau de neurones artificiels

Apprentissage supervisé $\rightarrow$ faire retrouver à un réseau la sortie attendue pour une entrée donnée.
Quel intérêt ? La sortie correcte est déja connue à l'avance ...

$\rightarrow$ **Généralisation :** Retrouver des "motifs" dans les données, et prédire les futures entrées dont le résultat est inconnu.

Comment changer les paramètres pour qu'un neurone apprenne ?

Entrée et Sortie fixées, l'apprentissage repose donc uniquement sur les **poids W_i et la limite d'activation θ**

Apprentissage

Il sagit d'un modèle **Supervisé**, chaque vecteur d'entrée est donc associé à une réponse attendue : la target t.

Supposons que l'on présente un vecteur d'entrée au neurone et que celui-ci ne donne pas la réponse attendue.
Il y a m poids w_i (avec i de 1 à m) qui sont connectés à ce neurones, correspondant à chaque noeud d'entrée.

Comment modifier les poids ? :

- Si le neurone tire alors qu'il ne devrait pas, les poids sont trop grands
- Si le neurone ne tire pas alors qu'il devrait, les poids sont trop petits

On calcule une erreur : $y_k - t_k$
La différence entre la sortie et la cible pour le neurone k

Cette diférence est multipliée par la valeur de chaque entrée et utilisée pour **changer le poid associé** $\Delta w_{ij} = -\eta\,(y_k - t_k) \times x_i$

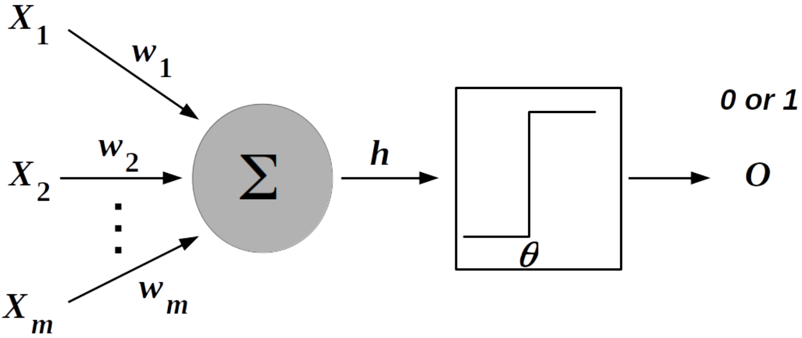
Taux d'apprentissage

On défini un **taux d'apprentissage** η qui permet de quantifier la vitesse à laquelle l'algorithme va modifier les poids.

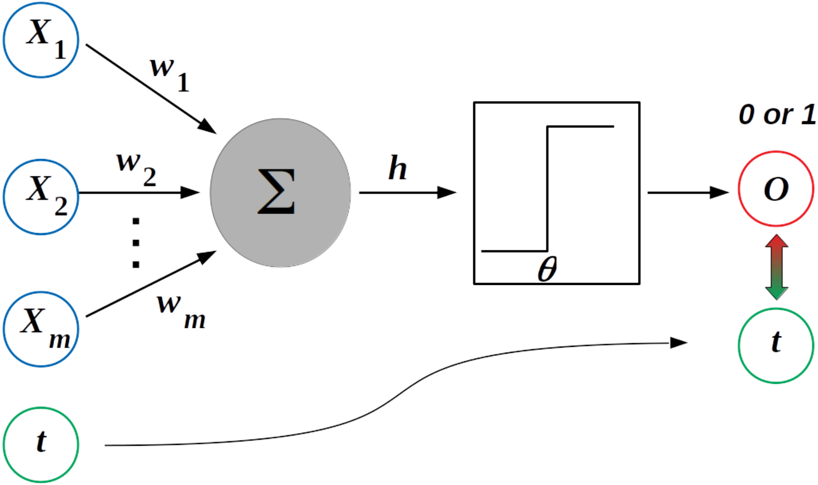
- **Une valeur trop importante de ce taux rends l'algorithme instable.**
- **Une valeur trop faible ralentie l'apprentissage.**

Une valeur typique se situe entre $0.1 < \eta < 0.4$ ce qui permet de rendre l'algorithme plus stable mais aussi plus résistant aux erreurs (bruit).

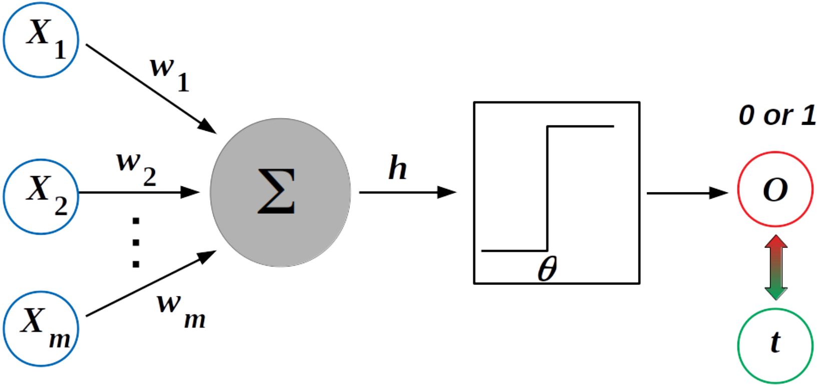
Le choix réside dans la qualité attendue des données d'apprentissage.


X_1
w_1
X_2
w_2
X_m
w_m
Σ
h
θ
0 or 1
O

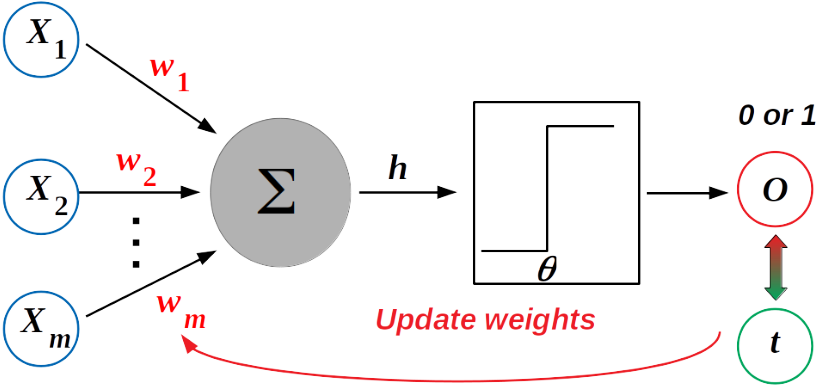
Apprentissage illustré



Apprentissage illustré



Apprentissage illustré

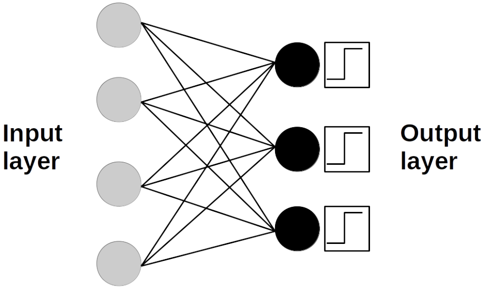


$$\omega_i \leftarrow \omega_i - \eta\,(o - t) \times X_i$$

Le Perceptron

Un seul neurone est dans la majorité des cas insuffisant pour représenter correctement un problème. La façon la plus simple d'en ajouter est de le faire sur une "couche/layer". Ces neurones restent indépendants, chacun représentant une séparation linéaire dans l'espace des paramètres d'entrée, mais l'ensemble peut servir à encoder une information.

- Les neurones sont indépendants entre eux
- Les poids sont séparés pour chaque neurone
- Le nombre de dimensions en entrée et le nombre de neurones (sorties) sont indépendants
- Les dimensions d'entrée et de sortie sont imposées par les données
- La sortie est un shéma de 0 et de 1 qui encode l'information voulue



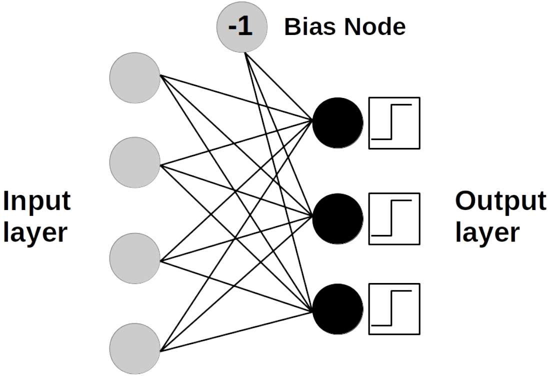
Vecteur d'entrée connecté de manière pondérée à des neurones de McCulloch

The Perceptron

L'entrée biaisée

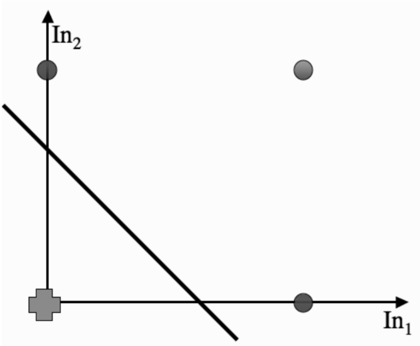
Problème :
Si toutes les entrées sont à 0 alors tous les neurones en sortie auront le même comportement... Pour contrer cet effet la limite d'activation peut etre ajusté, mais difficle à mettre en oeuvre

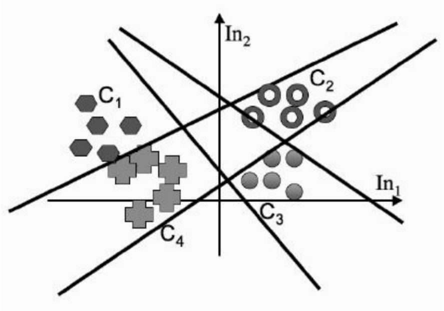
Solution :
Ajouter un noeud d'entrée qui a une valeur fixée à -1 et des poids associés. Définira le comportement pour des entrées proches de zero.



Note sur la séparabilité linéaire

Les neurones du perceptron tel que décris ici ne sont capables que de tracer des séparations linéaires entre les classes étudiées.





Il est donc nécessaire de représenter les données en entrée de manière astucieuse en ajoutant un nombre de dimensions suffisant afin de garantir leur séparabilité.

Réalisme du model ?

Un neurone réel reste assez différent :

- La somme des inputs peut etre non linéaire
- La sortie n'est pas binaire mais est une séquence d'impulsion ce qui contient de l'information supplémentaire
- La limite de tir varie au cours du temps
- L'interrogation des neurones n'est pas séquentielle (mise à jour Asynchrone)
- Les poids peuvent être négatifs ou positifs mais pas de passage de l'un à l'autre
- Les synapses peuvent revenir sur le neurone d'origine (feedback)
- Après avoir tiré, un neurone à un temps de recharge

Le model théorique est cependant suffisant pour apprendre des images, représenter des fonctions, faire du classement, ...

Perceptron : Algorithme

- Initialisation :
  - Définir tous les poids w_i avec de petites valeurs aléatoire (positives et négatives)
- Entrainement
  - pour T itérations ou jusqu'à ce que la sortie soit correcte
    - ⋆ Pour chaque vecteur d'entrée
      - · Calculer l'activation de chaque neurone j avec la fonction d'activation g :

$$y_j = g\left(\sum_{i=0}^{m} w_{ij} x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij} x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij} x_i \leq 0 \end{cases} \quad (1)$$

      - · Mettre à jour les poids individuellement avec :

$$w_{ij} \leftarrow w_{ij} - \eta\,(y_j - t_j) \cdot x_i \quad (2)$$

- Rappel
  - Calculer l'activation de chaque neurone j pour chaque vecteur

Première application : la porte OU

Un exemple très simple de l'application de cet algorithme est de tenter de lui apprendre **la porte logique "OU"**.

Le perceptron prend donc une entrée à deux dimensions et ne possède qu'un seul neurone qui doit donner la sortie attendue sous la forme d'un 0 ou d'un 1.

Codez donc un premier programme qui declare un tableau contenant les entrées possibles pour cette porte logique, et un tableau pour les cibles correspondantes.

Declarez les variables necessaire à l'algorithme, initialisez les deux poids à des petites valeurs aléatoires, et tentez de faire un entrainement sur quelques itérations. **N'oubliez pas le noeud de biais !**

Affichez la sortie, à chaque iteration pour observer la convergence.

```fortran
program logic_or
  implicit none

  integer, dimension(4,3) :: input
  integer, dimension(4) :: targ
  integer :: output
  real :: h, learn_rate = 0.1
  real, dimension(3) :: weights
  integer :: i, j, t

  input = reshape((/0, 0, 1, 1, 0 ,1 ,0 ,1, -1, -1, -1, -1 /), shape(input))
  targ = (/0, 1, 1, 1/)

  call random_number(weights)
  weights(:) = weights(:)*(0.02)-0.01

  !##################################################
  !                    Main training loop
  !##################################################
  do t = 1, 5
    write(*,*) "Iteration :", t
    !##################################################
    ! Testing the result of the network with a forward
    !##################################################

    do i=1, 4
      !Forward phase
      h = 0.0
      do j=1, 3
        h = h + weights(j)*input(i,j)
      end do

      if(h > 0) then
        output = 1
      else
        output = 0
      end if

      write(*,*) "Input  :", input(i,:)
      write(*,*) "Target :", targ(i)
      write(*,*) "Output :", output
      write(*,*) " "
    end do


    !##################################################
    !              Training on all data once
    !##################################################
    do i=1, 4
      !Forward Step
      h = 0.0
      do j=1, 3
        h = h + weights(j)*input(i,j)
      end do

      if(h > 0) then
        output = 1
      else
        output = 0
      end if

      !Back-propagation phase
      do j=1, 3
        weights(j) = weights(j) - learn_rate*(output-targ(i))*input(i,j)
      end do
    end do
  end do
end program logic_or
```

The Pima Indian Dataset

Fait parti du UCI Machine Learning repository, très utile pour récupérer des données de test.

Donne 8 mesures éffectuées sur une population de natifs américains nommés les Pimas, et les classe selon que la personne soit atteinte de diabète ou non.

Malgré ses limitations le perceptron une fois correctement entrainé devrait être capable de trouver le résultat attendu dans 60 à 70% des cas.

Modifiez votre programme précédent pour lire les données pima. Adapatez le nombre d'entrées et les calculs effectués. Mesurez la "précision" de votre algorithme et son évolution au fil des itérations.

Correction pima

```fortran
program logic_or
  implicit none

  integer, parameter :: nb_dat = 768
  real, dimension(nb_dat,9) :: input
  integer, dimension(nb_dat) :: targ
  integer :: output, precis
  real :: h, learn_rate = 0.1
  real, dimension(9) :: weights
  integer :: i, j, t


  open(10, file="pima-indians-diabetes.data")

  do i=1, nb_dat
    read(10,*) input(i,1:8), targ(i)
  end do
  input(:,9) = -1.0


  call random_number(weights)
  weights(:) = weights(:)*(0.02)-0.01

  !#################################################
  !                  Main training loop
  !#################################################
  do t = 1, 50
    write(*,*) "Iteration :", t
    !#################################################
    ! Testing the result of the network with a forward
    !#################################################
    precis = 0.0
    do i=1, nb_dat
      !Forward phase
      h = 0.0
      do j=1, 9
        h = h + weights(j)*input(i,j)
      end do

      if(h > 0) then
        output = 1
      else
        output = 0
      end if
      if(output == targ(i)) then
        precis = precis + 1
      end if

    end do
    write(*,*) "Precision rate :", real(precis)/real(nb_dat)


    !#################################################
    !           Training on all data once
    !#################################################
    do i=1, nb_dat
      !Forward phase
      h = 0.0
      do j=1, 9
        h = h + weights(j)*input(i,j)
      end do

      if(h > 0) then
        output = 1
      else
        output = 0
      end if

      !Back-propagation phase
      do j=1, 9
        weights(j) = weights(j) - learn_rate*(output-targ(i))*input(i,j)
      end do
    end do
  end do

end program logic_or
```

Amélioration : Préparer les données

Un neurone donne une sortie 0 ou 1 il est donc assez évident de transformer la sortie attendue pour qu'elle puisse être encodée avec ce type de sortie.

Il est également utile de faire quelques changements sur les entrées tel que celles ci se trouvent entre -1 et 1. On parle de normalisation des données.

Il est aussi possible de faire des groupes dans certains paramètres, par exemple l'âge, en faisant des catégories 21-30,31-40,... Plus facilement reconnaissable et plus logique.

Autre modification, la réduction dimensionnelle, qui consiste à retirer successivement certains paramètres en entrée et de voir si le résultat change ou non.

Correction pima optimisation

```fortran


do i = 0, nb_dat
  if(input(i,1) > 8) input(i,1) = 8
  input(i,8) = int(mod(input(i,8)-30,10.0))
  if(input(i,8) > 5) input(i,8) = 5
end do

do i=1, 8
  input(:,i) = input(:,i)-(sum(input(:,i))/nb_dat)
  input(:,i) = input(:,i)/maxval(abs(input(:,i)))
end do
```

Iris dataset

Ce jeu de données vient également de l'UCI repository. Il donne 4 mesures effectuées sur des iris et les classe en 3 catégories.

La meilleure façon pour faire une classifiaction qui n'est pas binaire est d'avoir autant de neurone que de classes de sortie. La target voulue sera alors encodée de telle manière que le neurone associé à la classe vaudra 1 et que les autres vaudront 0.

Exemple : classe 1 : 1 0 0 - classe 2 : 0 1 0 - class 3 : 0 0 1

Modifiez votre algorithme pour qu'il lise ce nouveau jeu de donnée et associe correctement les cibles. Adaptez le coeur de l'algorithme comme expliqué plus haut pour prendre en charge plusieurs neurones de sortie indépendants.

Correction Iris dataset

Voir code (trop long pour être affiché)

Note sur le sur-apprentissage

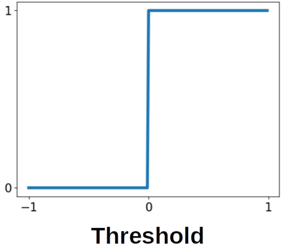
Entrainer trop longtemps l'algorithme peux avoir pour conséquence un apprentissage du bruit contenu dans les données. L'algorithme perd alors trace de la tendance...

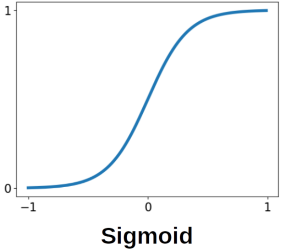
Soit limiter l'entrainement manuellement, soit surveiller l'évolution de l'apprentissage.
De plus il est important de **mélanger** les données (en conservant la bonne association de cibles) à chaque iteration sur l'ensemble du dataset, afin d'éviter un biais dans l'organisation des données.

Dans le cas du perceptron ces précautions ne sont pas vraiment necessaires puisque la convergence de l'algorithme est dominée par les effets non linéaires manquants.

Nécessite deux amélioration principales :

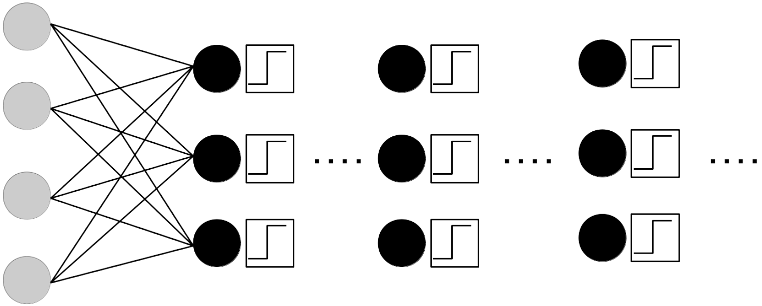
- Changement de la fonction d'activation : Passer à une fonction continue dérivable non binaire. Bon exemple : **la Sigmoïd**.





Nécessite deux amélioration principales :

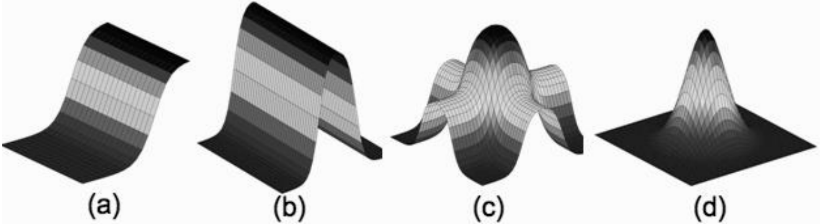
- Changement de la fonction d'activation : Passer à une fonction continue dérivable non binaire. Bon exemple : **la Sigmoïd**.
- Ajouter des neurones sur une autre couche pour permettre les combinaisons **non linéaires**.



Nécessite deux amélioration principales :

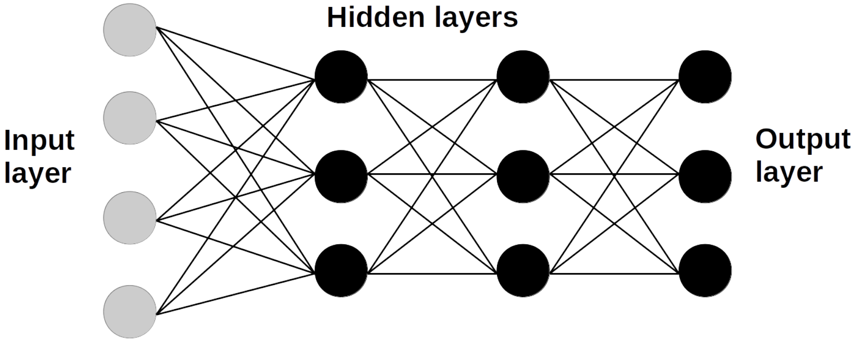
- Changement de la fonction d'activation : Passer à une fonction continue dérivable non binaire. Bon exemple : **la Sigmoïd**.
- Ajouter des neurones sur une autre couche pour permettre les combinaisons **non linéaires**.

Ces deux améliorations font de cet algorithme un **"Universal Function Approximator"**.



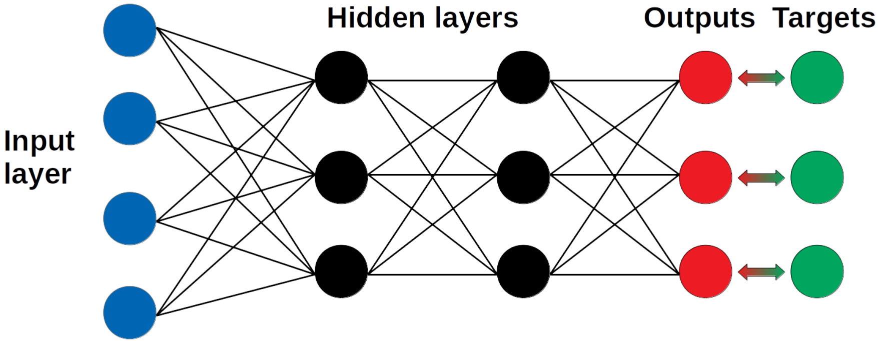
Multi Layer Perceptron

Ce nouvel algorithme, le MLP, ajoute les neurones sous la forme de couches dites "cachées/hidden"



Problème : Comment estimer l'erreur sur la couche cachée ?

Multi Layer Perceptron algorithm



$$\frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial \omega_{ij}} \qquad \delta_l(j) \equiv \frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial h_j} = g'(a_j)\sum_k \omega_{kj}\delta_{l-1}(k)$$

Réseau concret

Les changements précédents se traduise par les modification suivantes dans l'algorithme du perceptron :

- Changement de la fonction d'activation de tous les neurones tel que :

$$a_k = g(h_k) = 1/\big(1 + exp(-\beta h_k)\big)$$

  avec β un réel positif supérieur ou égale à 1.0 définissant la "pente".
- Ajout d'une couche cachée, avec son noeud de biais associé.
- Changement de la "back propagation" avec $E(a,t) = \frac{1}{2}\sum_{k=1}^{N}(a_k - t_k)^2$:

$$\delta_o(k) = \beta(a_k - t_k)a_k(1 - a_k)$$

$$\delta_h(j) = \beta a_j(1 - a_j)\sum_{k=1}^{N}\delta_o(k)\omega_{jk}$$

$$\omega_{jk} \leftarrow \omega_{jk} - \eta\delta_o(k)a_j$$

$$v_{ij} \leftarrow v_{ij} - \eta\delta_h(j)x_i$$

Remarques :

Les equations précédentes sont valable pour le cas ou tout les neurones ont une fonction d'activation sigmoïd et pour une erreur quadratique classique.

Dans le cas général les neurones cachés seront toujours sous la forme de sigmoid ou d'une fonction proche. Néanmoins les neurones de sortie peuvent être adaptés pour etre plus performants sur certaines taches.

Cet algorithme est capable de faire de la régression, pour cela il faut changer la fonction d'activation du/des neurones de sorties vers une fonction linéaire $o = g(h) = h$. Il faut donc adapter le calcul de $\delta^o = (o - t)$.

Pour de la classification, une sortie ou chaque neurone correspond à une classe est plus adapté mais on voudrait normaliser cette sortie. Pour cela on peux utiliser la fonction d'activation SOFT-MAX telle que : $a_k = g(h_k) = \frac{exp(h_k)}{\sum_{k=1}^{N} exp(h_k)}$. Il faut alors adapter δ^o, qui se simplifie par la même fonction que pour la sigmoid mais sans le parametre β.

Sur apprentissage : Séparation des données

Apprentissage = voir plusieurs fois l'echantillon

Entraîner trop longtemps cause un **surapprentissage** !
$\rightarrow$ apprend du bruit, ou des spécificités de l'échantillon.
L'algorithme perd alors trace de la tendance... **Pour contrer ce problème il convient de séparer les données :**

- **Entrainement** : Majorité des données, sert à la phase d'entrainement de l'algorithme.
- **Validation** : partie plus modeste des données, sert à mesurer l'erreur au cours de l'apprentissage. Si elle augmente de manière significative il faut stopper l'apprentissage.
- **Test** : permet de controler sur des données indépendantes la qualité de l'apprentissage une fois celui-ci terminé.

Des proportions usuelles sont $50 : 25 : 25$ ou encore $60 : 20 : 20$