Chapitre 6 : Les réseaux de neurones

1. Architecture d'un réseau de neurones :
   - Perceptron,
   - Couches et couches caches, poids, biais.
   - Fonction d'activation : ReLU, Sigmoïde, Softmax, ….
   - Exercices d'applications
2. Introduction au **Deep Learning** :
   - Notion de couches profondes.
   - Introduction au réseaux convolutifs (CNN)
3. **Exercices :**
   - Expliquer Tensorflow et PyTorch
   - Analyser un Dataset de texte et prédire des sentiments
   - ….

1) Introduction

Les réseaux de neurones artificiels sont des modèles mathématiques inspirés de la biologie. La brique de base de ces réseaux, le neurone artificiel, était issue au départ d'une volonté de modélisation du fonctionnement d'un neurone biologique.

Nous allons donc voir dans ce chapitre c'est quoi un neurone artificiel qui constitue les réseaux de neurones et son principe de fonctionnement. Nous allons par la suite savoir la différence entre un perceptron et réseau multicouches. Nous expliquons également l'algorithme de leurs apprentissages illustré par un exemple détaillé.

2) Neurone naturel vers neurone artificiel

2.1 Neurone naturel

Le cerveau se compose d'environ 1012 (mille milliards) de neurones interconnectés, avec 1000 à 10000 synapses (connexions) par neurone (Fig.1.1). Les neurones ne sont pas tous identiques, leur forme et certaines caractéristiques permettent de les répartir en quelques grandes classes :

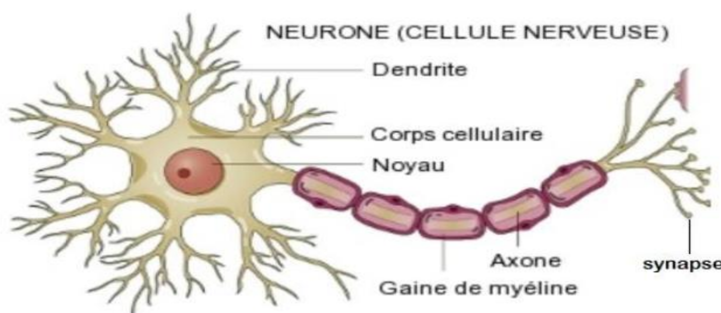


Fig.1 Cellule neuronale

- Le corps cellulaire contient le noyau du neurone et effectue les transformations biochimiques nécessaires à la synthèse des enzymes et des autres molécules qui assurent la vie de la cellule.

- Les dendrites sont de fines extensions tubulaires qui se ramifient autour du neurone et forment une sorte de vaste arborescence. Les signaux envoyés au neurone sont captés par les dendrites.

- L'axone est la fibre nerveuse, sert de moyen de transport pour les signaux émis par le neurone. Il est plus long que les dendrites, et se ramifie à son extrémité où il se connecte aux dendrites des autres neurones. Les connexions entre deux neurones se font en des endroits appelés synapses.

- Chaque neurone est une unité autonome au sein du cerveau. Le neurone reçoit en continu des entrées.

Le corps cellulaire du neurone est le centre de contrôle. C'est là que les informations reçues sont interprétés La réponse, unique, à ces signaux est envoyée au travers de l'axone. L'axone fait synapse sur d'autres neurones (un millier). Le signal transmis peut avoir un effet excitateur ou l'inverse.

2.2 Neurone artificiel

Le perceptron, encore appelé neurone artificiel ou neurone formel, cherche à reproduire le fonctionnement d'un neurone biologique. Il existe différents niveaux d'abstraction, suivant la précision de la modélisation voulue.

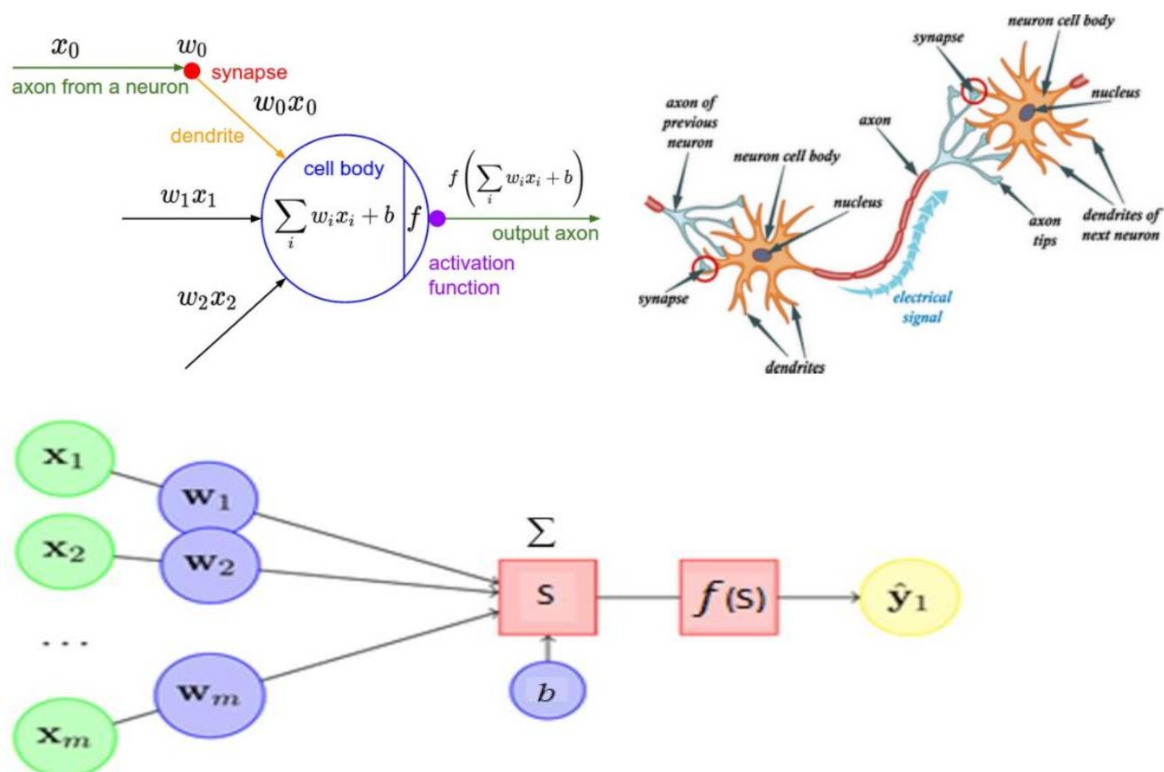


Fig.2 Neurone formel de Mac Culloch et Pitts.

Tels que :

- (**x1 , x 2 ,…, xm**) : sont les entrées du neurone (signaux qui lui parviennent).
- (**w1 , w2 , ..., wn**): les poids associés à chaque connexion.
- ***b*** : le seuil d'activation.
- **S** : la somme pondérée des entrées (potentiel d'activation)

$$S = \sum_{i=1}^{n} w_i x_i + b$$

• **ŷ** = ***f*****(S)** : la sortie du neurone (réponse du neurone « activé ŷ =1, ou non activé ŷ =0»).

$$\hat{y} = \begin{Bmatrix} 1 \ si \ S > 0 \\ 0 \ si \ S < 0 \end{Bmatrix}$$

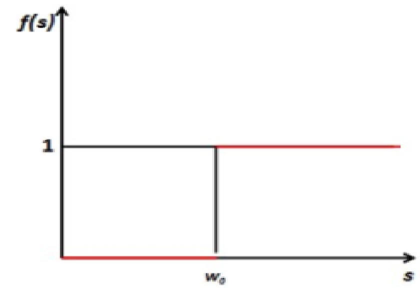


Fig.3 Fonction d'activation d'un neurone formel

A partir de ce modèle ont été définis divers modèles de neurones et avec d'autres fonctions d'activations.

Rappels sur la régression linéaire

Régression linéaire univariée : on explique la variable y à partir d'une variable x

$$y = \beta_0 + \beta_1 x + \epsilon$$

Régression linéaire multivariée : on explique la variable y à partir de d variables $x_1, x_2, \cdots, x_d$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_d x_d + \epsilon$$

ce qui s'écrit aussi

$$y = \mathbf{x} W + b + \epsilon$$

avec le prédicteur $\mathbf{x} = (x_1, \cdots, x_d)$, l'intercept $b = \beta_0$ et le vecteur de poids $W = (\beta_1, \cdots, \beta_d)^T$

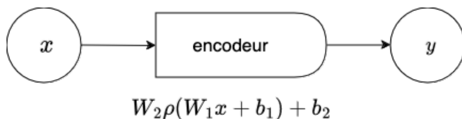
Sachant les observations $(y_i, \mathbf{x}_i)_{i=1,\cdots,n}$, on estime b et W en minimisant la fonction de perte des moindres carrés.

Exemple (multivarié) :

- y : temps de freinage de la voiture
- $\mathbf{x}$: (vitesse de la voiture, poids de la voiture, état des pneus, humidité de la route)

Réseau de neurones

Un **réseau de neurones "feedforward"** est un algorithme permettant de traiter une entrée $x \in \mathbb{R}^d$ et de renvoyer une sortie $y \in \mathbb{R}$ (ou $\mathbb{R}^k$).



L'algorithme implique l'application répétée de deux opérations simples :

1. une transformation linéaire affine,

$$A(x) = Wx + b$$

2. une fonction d'activation non linéaire ρ
appliquée coordonnée par coordonnée.

Dans le cas le plus simple, les deux opérations sont répétées plusieurs fois, par composition.

Un neurone = perceptron

Un neurone seul est défini comme suit :

$$\Phi(x) = \rho(Wx + b)$$

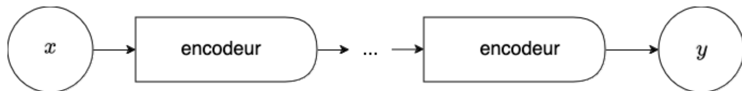
avec ρ la **fonction d'activation**, W les **poids** et b le **biais**.

On l'appelle **perceptron**.

Les paramètres inconnus sont les poids W et le biais b.

Réseau de neurones

Un **réseau de neurones "feedforward"** traite l'information par composition



1. Notons l'**entrée**, $x^0 = x$
2. Pour $1 \leq \ell \leq L$,

$$x^\ell = A^\ell(\rho(x^{\ell-1})) = W^\ell \rho(x^{\ell-1}) + b^\ell$$

3. La **sortie** est $y = \Phi(x) = x^L$ (dans le cas de la régression)

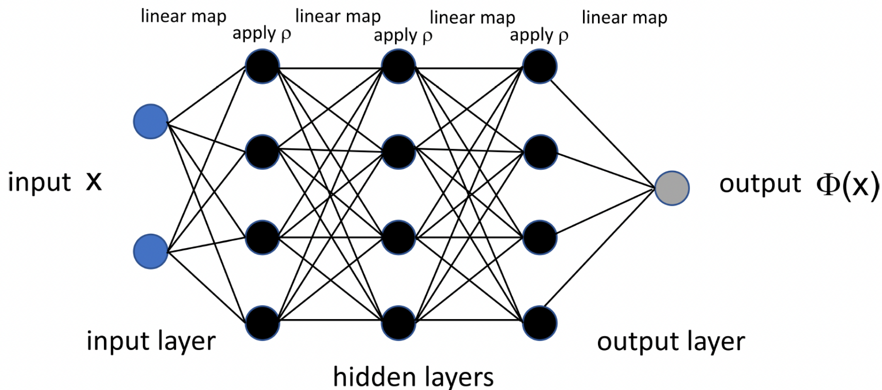
Les valeurs x^ℓ, $1 \leq \ell \leq L-1$, qui ne sont pas directement observées correspondent aux **couches latentes** [1] ou **variables latentes** du réseau de neurones.

Remarque : on verra plus loin que les valeurs des vecteurs de poids W^ℓ et de biais b^ℓ, $1 \leq \ell \leq L$, sont estimées par la minimisation d'une fonction de perte.

1. On les appelle aussi **couches cachées**

Réseau de neurones

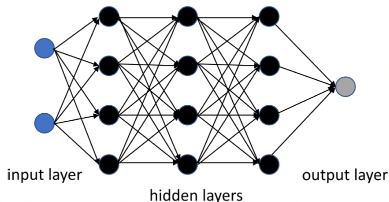
Représentation graphique d'un réseau de neurones avec une entrée $x \in \mathbb{R}^2$, une sortie $\Phi(x) \in \mathbb{R}$ et 4 couches (L=4) dont 3 couches cachées.



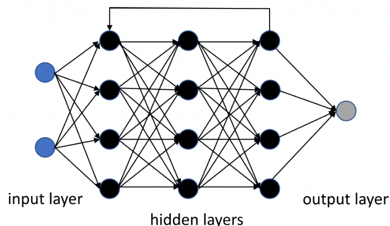
L'information est propagée de l'entrée vers la sortie
$\rightarrow$ **architecture feedforward**

Réseau de neurones

Quand l'information est propagée de l'entrée vers la sortie, le réseau de neurones est appelé **réseau de neurones feed-forward** ou **perceptron multicouches**.
Les réseaux neuronaux qui incluent des connexions vers l'arrière sont appelés **réseaux neuronaux récurrents** (non étudiées dans ce cours).



Feedforward neural network

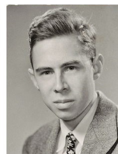


Recurrent neural network

Réseau de neurones feedforward : un peu d'histoire

Les réseaux neuronaux feedforward sont également appelés **perceptrons multicouches (MLP)** et familièrement appelés réseaux neuronaux "vanilla".

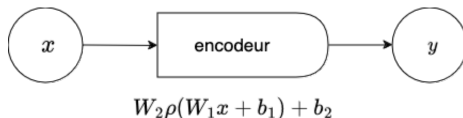
- En 1958, un perceptron multicouches (multilayer perceptron MLP), composé d'une couche d'entrée, d'une couche cachée avec des poids aléatoires d'une couche de sortie, a été introduit par Frank Rosenblatt dans son livre Perceptron. La fonction d'activation ρ était la fonction heavy-side.



- En 1967, S. Amari a pour la première fois estimé les poids d'un réseau de neurones à l'aide d'un algorithme de descente de gradient stochastique pour un problème de classification.

Réseau de neurones feedforward

Un réseau de neurones feedforward à une couche cachée est un réseau peu profond.



Dans ce cas, on applique une transformation linéaire à l'entrée $x \in \mathbb{R}^d$, puis la fonction d'activation ρ et de nouveau une transformation linéaire

$$\Phi(x) = W^2\rho(W^1x + b^1) + b^2$$

avec $W^1 \in \mathbb{R}^{N,d}$, $b^1 \in \mathbb{R}^N$, $W^2 \in \mathbb{R}^{1,N}$, $b^2 \in \mathbb{R}$ où N est le nombre de neurones cachés.

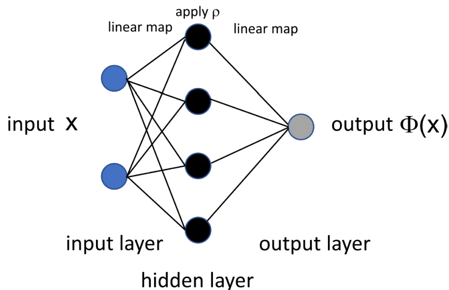
Si l'entrée est univariée ($d = 1$), on peut écrire

$$\Phi(x) = W^2\rho(\sum_{j=1}^{N} W_j^1 x + b_j^1) + b^2$$

Exercice :

1. Représenter un réseau de neurones peu profond à une couche cachée contenant 4 neurones (L=2), avec une entrée $x \in \mathbb{R}^2$ (ie deux variables) et une sortie $\Phi(x) \in \mathbb{R}$.
2. Écrire l'équation permettant de calculer $\Phi(x)$ et préciser la taille des matrices de poids et des vecteurs de biais.

Représentation graphique d'un réseau de neurones peu profond à une couche cachée contenant 4 neurones (L=2), avec une entrée $x \in \mathbb{R}^2$ (ie deux variables) et une sortie $\Phi(x) \in \mathbb{R}$.



$$\Phi(x) = W^2 \rho(W^1 x + b^1) + b^2$$

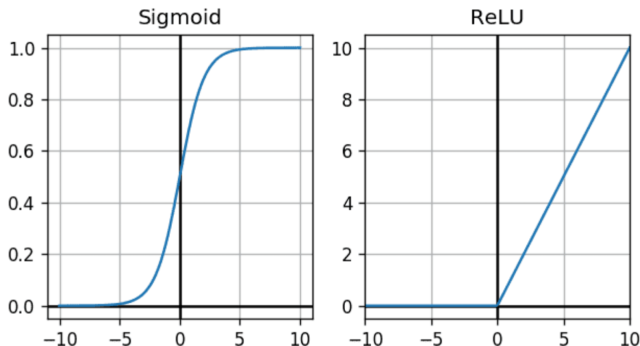
Les poids sont stockés dans des matrices $W^1 \in \mathbb{R}^{4,2}$ et $W^2 \in \mathbb{R}^{1,4}$.

Fonction d'activation

Le rôle de la fonction d'activation est d'introduire des non linéarités dans le réseau de neurones.

Les deux **fonctions d'activations** ρ les plus courantes sont

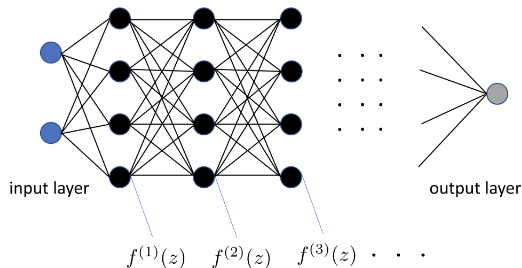
- la fonction **sigmoid** $\rho(x) = \frac{1}{1+e^{-x}}$
- la fonction **Rectified linear unit (ReLU)** : $\rho(x) = max(x, 0)$



La fonction RELU est la plus utilisée, mais la fonction sigmoid a l'avantage d'être différentiable. Une alternative à la fonction sigmoid est la fonction tanh.

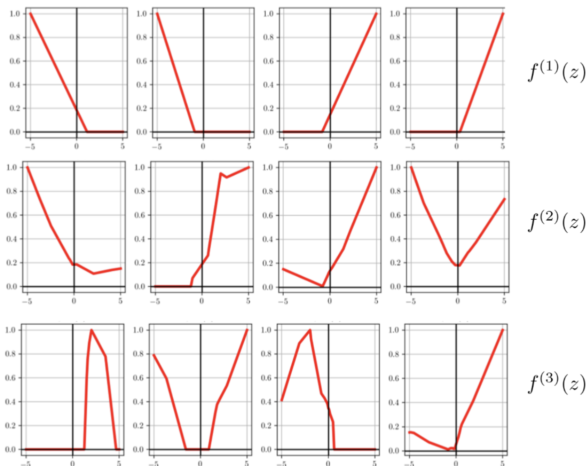
Le pouvoir des réseaux de neurones

En combinant des fonctions simples, un réseau de neurones peut représenter des formes complexes.
On considère un réseau de neurones à, au moins 3 couches cachées comme représenté ci-dessous.
On va montrer les sorties des neurones selon deux fonctions d'activation différentes.



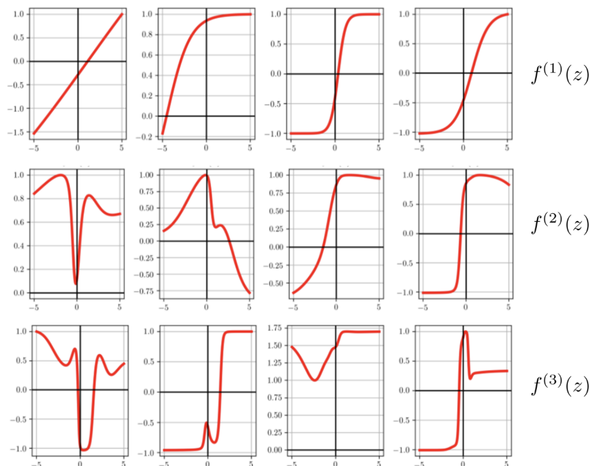
Fonction d’activation

Ici, on montre la sortie des neurones des 3 premières couches, pour des entrées aléatoires, en utilisant la fonction d’activation RELU.



Fonction d’activation

Ici, on montre la sortie des neurones des 2 premières couches, pour des entrées aléatoires, en utilisant la fonction d’activation tanh.

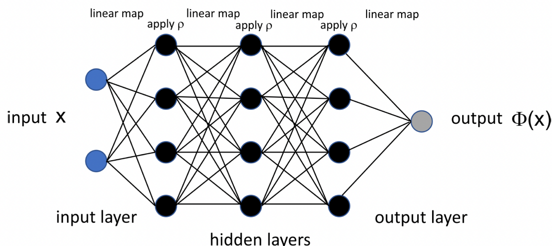


Réseau de neurones profond (deep neural network)

Un réseau de neurones profond a plusieurs couches cachées (parfois un très grand nombre)

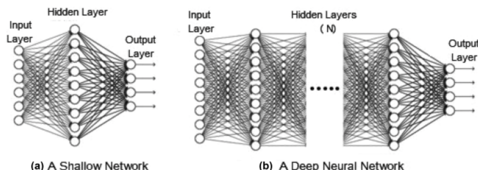
$$\Phi(x) = W^4\rho(W^3\rho(W^2\rho(W^1x + b^1) + b^2) + b^3) + b^4$$

Dans cet exemple, on a un réseau de neurones avec 4 couches dont 3 couches cachées.



Remarque : pour que le réseau Φ soit bien défini, il faut que les dimensions des matrices W^ℓ et des vecteurs b^ℓ soient cohérentes.

Réseau de neurones profond (deep neural network)



- Les architectures modernes sont le plus souvent très profondes.
- Une architecture profonde implique un grand nombre de paramètres (poids).
- Concevoir un réseau profond avec un grand nombre de paramètres peut être utile pour apprendre des modèles complexes pour des tâches non triviales, mais cela peut également conduire à un surajustement [2] important si l'ensemble d'apprentissage est petit ou si la tâche à accomplir est simple.
  Rqs
  1. Il existe des techniques pour réduire le risque de sur-ajustement des réseaux de neurones profonds.
  2. En pratique, c'est souvent une bonne stratégie de sur dimensionner les réseaux.

2. Donner un exemple !

Approximation universelle

Il existe des résultats théoriques qui montrent que les réseaux de neurones peuvent approcher toute fonction multivariée avec une précision fixée.

Definition

Une classe de fonctions $\mathcal{F}$ est appelée **approximateur universel** sur un compact $\mathcal{S}$ si pour toute fonction continue g et toute précision $\epsilon > 0$, il existe $f \in \mathcal{F}$ telle que

$$\sup_{x \in S} |f(x) - g(x)| \leq \epsilon$$

Approximation universelle

Les théorèmes d'approximation universelle disent que, quelle que soit la fonction que nous essayons d'apprendre, un réseau de neurones suffisamment profond sera capable de représenter cette fonction.

Theorem

(Hornik, 1991) Supposons que ρ est une fonction C^∞, non polynomiale. Alors la classe des réseaux de neurones peu profonds est un approximateur universel sur $[0,1]^d$. d est la dimension d'entrée du réseau de neurones.

Remarque : le réseau peut être arbitrairement large.

Theorem

Kidger and Lyons, 2020 Supposons que ρ est une fonction continue non affine qui est C^1 en au moins un point, avec une dérivée non nulle en ce point. La classe des réseaux de neurones profonds où le nombre de neurones N_ℓ pour chaque couche ℓ peut être borné par $D + d + 2$ est un approximateur universel sur $[0,1]^d$. d est la dimension d'entrée et D est la dimension de sortie du réseau de neurones.

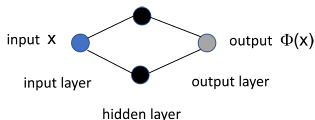
Remarque : le réseau peut être arbitrairement profond.

- Les théorèmes d’approximation universelle indiquent que les réseaux neuronaux ont la capacité d’approcher avec une certaine précision n’importe quelle fonction continue à lpusieurs .
- Le pouvoir de représentation d’un réseau de neurones augmente avec le nombre de neurones et de couches.
- Attention : même si un réseau de neurones est capable de représenter une fonction, l’algorithme d’entraînement peut échouer à apprendre cette fonction spécifique.
- L’apprentissage peut échouer
  (i) parce que l’algorithme d’optimisation utilisé pour l’entraînement peut ne pas être capable de trouver la valeur des paramètres correspondant à la fonction désirée ou
  (ii) parce que l’algorithme d’entraînement pourrait choisir la mauvaise fonction résultant d’un surajustement.

Exemple : la fonction triangle

$$T(x) = \begin{cases} 2x & \text{si} \quad 0 \leq x \leq 1/2 \\ 2(1-x) & \text{si} \quad 1/2 \leq x \leq 1 \end{cases}$$

On peut approcher la fonction T par un réseau de neurones à 1 couche cachée contenant 2 unités cachées et la fonction d'activation RELU.



On a un réseau de neurones à 7 paramètres

$$\Phi(x) = [w_{21} \quad w_{22}]\rho\left(\begin{bmatrix} w_{11} \\ w_{12} \end{bmatrix} x + \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix}\right) + b_2$$

avec avec la fonction d'activation RELU

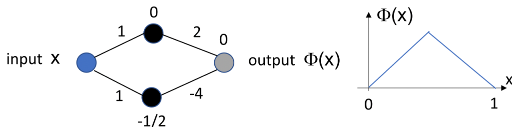
$$\rho(wx+b) = (wx+b)_+ = max(wx+b, 0)$$

On peut réécrire

$$\Phi(x) = (w_{21}(w_{11}x + b_{11})_+ + w_{22}(w_{12}x + b_{12})_+ + b_2)_+$$

Par ailleurs, on peut vérifier que

$$T(x) = (2(x-0)_+ - 4(x-1/2)_+)_+$$



On peut remarquer que chaque terme de Φ est associé à une portion d'une fonction linéaire par morceaux.

- On peut représenter n'importe quelle fonction triangle avec un réseau de neurones à 2 neurones cachés et des fonctions d'activation RELU.
- Plus généralement, toute fonction linéaire par morceaux peut être représentée par un réseau de neurones et des fonctions d'activation RELU.

Intelligence artificielle, machine learning

Les réseaux de neurones sont utilisés pour automatiser des tâches très variées

- Régression non linéaire
  ex : prédire le prix d'un logement, prédire une variable clinique, prédire une variable atmosphérique
- Classification
  ex : reconnaitre le contenu d'une image
- Génération d'images
  ex : création de logos
- Génération de texte
  ex : chatGPT, traduction automatique

D'un problème à l'autre, on change l'architecture du réseau de neurones, la fonction d'activation sur la couche de sortie et la fonction de perte.

Dans ce cours, on s'intéresse à la régression.

Régression

Soit $\{(x_1, y_1), \cdots, x_n, y_n)\}$ un ensemble d'apprentissage où $x_i \in \mathbb{R}^d$ et $y_i \in \mathbb{R}$.

La prédiction du réseau de neurones pour une entrée donnée x_i est le résultat de la passe avant

$$\Phi(x_i; \theta)$$

avec θ le vecteur de tous les poids w et biais b.

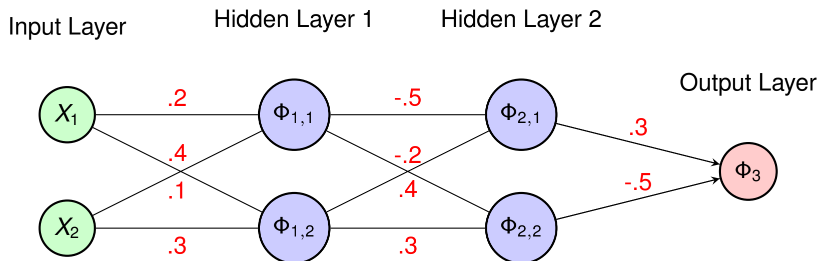
Comme dans le cas de la régression linéaire, on utilise l'erreur aux moindres carrés pour mesurer la qualité de la prédiction

$$L(x, y; \theta) = \frac{1}{2} \sum_{i=1}^{n} \left(y_i - \Phi(x_i; \theta)\right)^2$$

Pour estimer les paramètres θ, on a besoin de minimiser la fonction de perte et donc de calculer $\Phi(x_i; \theta)$ pour tout i. Or la fonction Φ n'admet pas de forme analytique simple : on l'évalue numériquement par la passe avant.

Exemple de passe avant

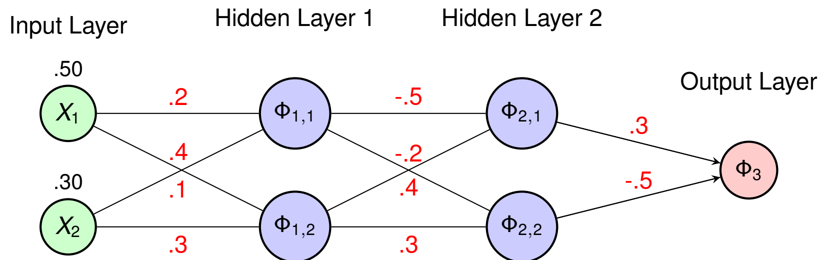
On considère un réseau de neurones à 2 entrées, 1 sortie et 2 couches cachées, contenant chacune 2 unités. On suppose que les biais sont nuls (réseau sans biais). La fonction d'activation de chacune des couches cachées est la fonction RELU.



Exercice : exécuter une passe avant pour calculer la sortie correspondant à l'entrée $X = (.5, .3)$.

Exemple de passe avant

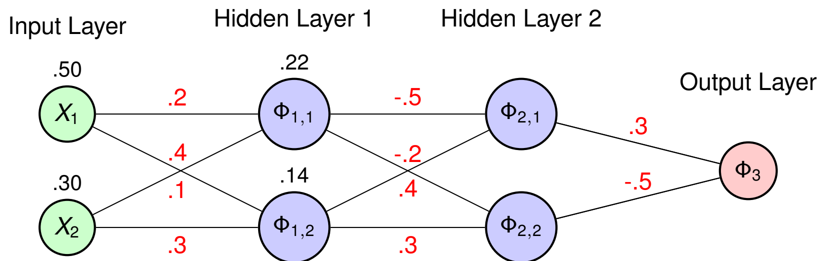
Entrée $X = (.5, .3)$



Exemple de passe avant

Première couche cachée (fonction d'activation RELU)

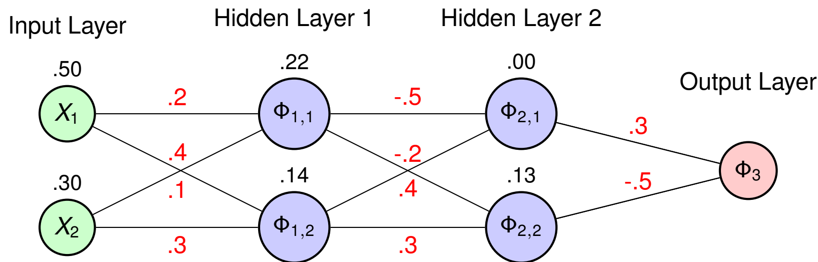
$$\Phi_{11}(X) = (.5 * .2 + .3 * .4)_+, \quad \Phi_{12}(X) = (.5 * .1 + .3 * .3)_+$$



Exemple de passe avant

Deuxième couche cachée (fonction d'activation RELU)

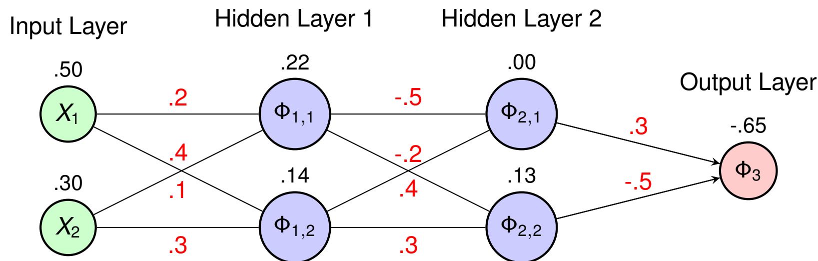
$$\Phi_{21}(X) = (-.22 * .5 - .14 * ..2)_{+} = 0, \quad \Phi_{22}(X) = (.22 * .4 + .14 * .3)_{+}$$



Exemple de passe avant

Couche de sortie (fonction d'activation identité)

$$\Phi_3(X) = (.00 * .3 - .13 * ..5)_+ = -0.65$$



Minimum de la fonction de perte

Pour estimer les paramètres d'un réseau de neurones MLP sachant un ensemble d'apprentissage $\{(x_i, y_i), i = 1 \cdots, n\}$, il faut trouver le minimum de la fonction de perte.

La fonction de perte des moindres carrés est dérivable mais elle n'est pas convexe : pour un réseau de neurones donné, elle admet plusieurs minima locaux.

Un algorithme de descente du gradient initialisé avec un vecteur de paramètres va converger vers un des minima locaux, qui correspond à une bonne solution.

Minimum de la fonction de perte

Dans un MLP, la sortie dépend de tous les poids, donc l'erreur obtenue au dernier noeud de sortie dépend également de tous les poids.

Pour calculer la dérivée de la fonction de perte par rapport aux poids, nous allons rétropropager l'erreur jusqu'au oeud d'entrée à partir du noeud de sortie.

La **rétropropagation** est un algorithme qui permet de calculer le gradient d'un MLP. C'est un des éléments clé de la force des réseaux de neurones.

Rétropropagation

Dans un réseau de neurones feedforward, considérons l'erreur associée à une donnée x_i. Il s'agit de l'erreur ente la sortie obtenue $\Phi_L(x_i)$ et le résultat attendu y_i, avec L l'indice de la dernière couche.

Notons $E_L(i)$, l'erreur associée à l'observation x_i au nud L et

$$e_L(i) = y_i - \Phi_L(x_i)$$

et $E_L(i)$ l'erreur aux moindres carrés à la sortie du noeud L

$$E_L(i) = \frac{1}{2} e_L(i)^2$$

Pour mettre à jour les poids, on utilise une descente de gradient : pour tout poids $w_{\ell m}$

$$w_{\ell m} \leftarrow w_{\ell m} - \eta \frac{\partial E_L(i)}{\partial w_{\ell m}}$$

Voyons comment calculer ce gradient.

Calcul du gradient : formule de composition

Un réseau de neurones est une composition de fonctions. On va donc utiliser la formule de composition ie le gradient de fonctions composées.
Soient F, g et h des fonctions différentiables [3]

$$\frac{\partial F(g(t),h(s))}{\partial t} = \frac{\partial F(g(t),h(s))}{\partial g(t)} \frac{\partial g(t)}{\partial t}$$

$$\frac{\partial F(g(t),h(s))}{\partial s} = \frac{\partial F(g(t),h(s))}{\partial h(s)} \frac{\partial h(s)}{\partial s}$$

Exercice
Soit F la fonction composée suivante, avec F et F_1 différentiables
$F(w_1, w_2) = F(F_1(w_1), w_2)$. Exprimer la dérivée de F par rapport à w_1 en fonction de la dérivée partielle de F_1 par rapport à w_1.

3. différentiable est l'équivalent de dérivable pour les fonctions à plusieurs variables

Calcul du gradient : formule de composition

Un réseau de neurones est une composition de fonctions. On va donc utiliser la formule de composition soit le gradient de fonctions composées.
Soient F, g et h des fonctions différentiables [4]

$$\frac{\partial F(g(t), h(s))}{\partial t} = \frac{\partial F(g(t), h(s))}{\partial g(t)} \frac{\partial g(t)}{\partial t}$$

$$\frac{\partial F(g(t), h(s))}{\partial s} = \frac{\partial F(g(t), h(s))}{\partial h(s)} \frac{\partial h(s)}{\partial s}$$

Exercice (corrigé)
Soit F la fonction composée suivante, avec F et F_1 différentiables $F(w_1, w_2) = F(F_1(w_1), w_2)$. Exprimer la dérivée de F par rapport à w_1 en fonction de la dérivée partielle de F_1 par rapport à w_1.

Alors on a

$$\frac{\partial F(F_1(w_1), w_2)}{\partial w_1} = \frac{\partial F(F_1(w_1), w_2)}{\partial F_1(w_1)} \frac{\partial F_1(w_1)}{\partial w_1}$$

4. différentiable est l'équivalent de dérivable pour les fonctions à plusieurs variables

Calcul du gradient

L'erreur à la sortie de la dernière couche s'écrit

$$E_L(i) = \frac{1}{2}\big(y_i - \Phi_L(x_i)\big)^2$$

avec

$$\Phi_L(x_i) = w_L \Phi_{L-1}(x_i)$$

et

$$\Phi_{L-1}(x_i) = \rho(w_{L-1}\Phi_{L-2}(x_i))$$

etc.

Remarque : on a allégé les notations ; en réalité $\Phi_K(x_i) = \Phi_K(w_K, x_i)$

Pour un poids de la **couche de sortie** (couche L), on a

$$\frac{\partial E_L(i)}{\partial w_L} = \frac{\partial E_L(i)}{\partial \Phi_L(x_i)} \frac{\partial \Phi_L(x_i)}{\partial w_L} = \big(y_i - \Phi_L(x_i)\big)\Phi_{L-1}(x_i)$$

On remarque que toutes quantités nécessaires pour calculer cette dérivée partielle (ie $\Phi_L(x_i)$ et $\Phi_{L-1}(x_i)$) ont déjà été calculées lors de la passe avant (ici on les réutilise sans calcul supplémantaire).

L'erreur à la sortie de la dernière couche s'écrit

$$E_L(i) = \frac{1}{2}\big(y_i - \Phi_L(x_i)\big)^2$$

avec

$$\Phi_L(x_i) = w_L \Phi_{L-1}(x_i)$$

et

$$\Phi_{L-1}(x_i) = \rho(w_{L-1}\Phi_{L-2}(x_i))$$

etc.

Pour un poids de la **dernière couche cachée** (couche L-1), on peut écrire

$$\begin{aligned} \frac{\partial E_L(i)}{\partial w_{L-1}} &= \frac{\partial E_L(i)}{\partial \phi_L(x_i)} \frac{\partial \phi_L(x_i)}{\partial \phi_{L-1}(x_i)} \frac{\partial \phi_{L-1}(x_i)}{\partial w_{L-1}} \\ &= \big(y_i - \Phi_L(x_i)\big) w_L \Phi_{L-2}(x_i) \rho'(w_{L-1}\Phi_{L-2}(x_i)) \end{aligned}$$

Remarques :
- si la fonction d'activation ρ est la fonction RELU alors

$$\rho'(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ 0 & \text{sinon} \end{cases}$$

- toutes les quantités nécessaires pour obtenir ces éléments du gradient ont été calculées lors de la passe avant.

Gradient de la fonction de perte

On a montré, par le calcul, que le gradient de $E_L(x_i)$ en fonction des poids w se calcule par une passe arrière le long du réseau de neurones.

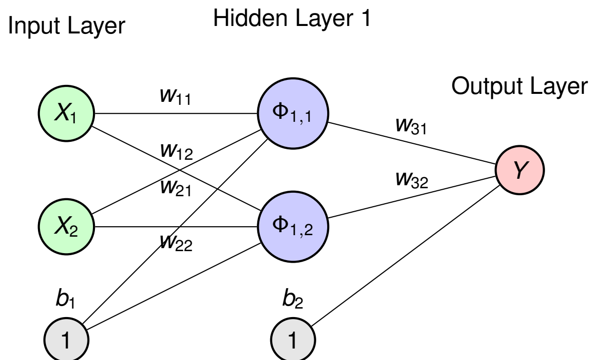
Le gradient de la fonction de perte :

$$\nabla L(w) = \nabla\left(\frac{1}{2}\sum_{i=1}^{n}(y_i - e_L(x_i))^2\right) = \sum_{i=1}^{n}\left(\nabla E_L(x_i)\right)\left(y_i - e_L(x_i)\right)$$

Toutes les quantités nécessaires ont déjà été calculées : le calcul du gradient ne coûte par cher.
C'est une des forces des réseaux de neurones : on utilise naturellement les algorithmes de descente de gradient, à un coût très raisonnable.

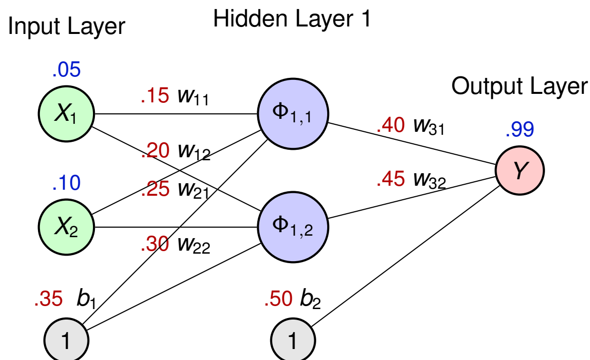
Exemple de rétropropagation

Nous allons illustrer la rétropropagation avec un réseau simple comportant une seule couche cachée à 2 unités cachées.



Exemple de rétropropagation

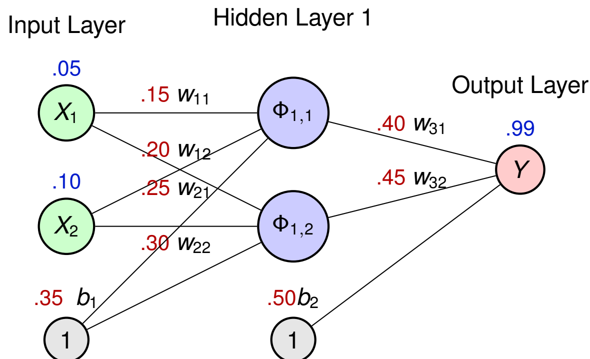
Le réseau de neurones est initialisé comme suit



Il y a un seul exemple dans l'ensemble d'apprentissage : pour l'entrée $X_1 = 0.05$ et $X_2 = 0.10$, on veut que la sortie du réseau de neurone soit $Y = 0.99$.
On choisit la fonction d'activation RELU et le taux d'apprentissage $\alpha = 0.1$.

Exemple de rétropropagation

On commence par faire une passe avant pour prédire la sortie du réseau de neurones avec les poids et les biais courants pour l'entrée 0.05 et 0.10.



L'entrée du noeud $\Phi_{1,1}$ est

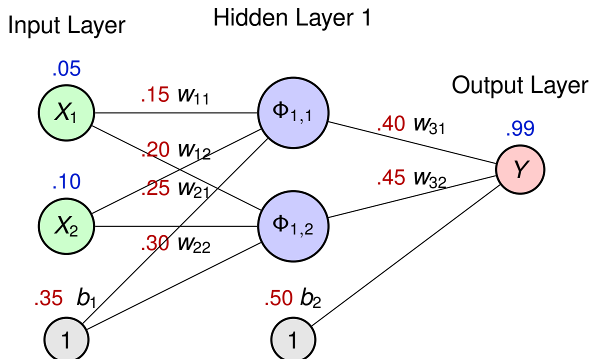
$vh1 = w_{11}X_1 + w_{12}X2 + b1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$

et sa sortie est

$yh1 = \rho(vh1) = 0.3775$

Exemple de rétropropagation

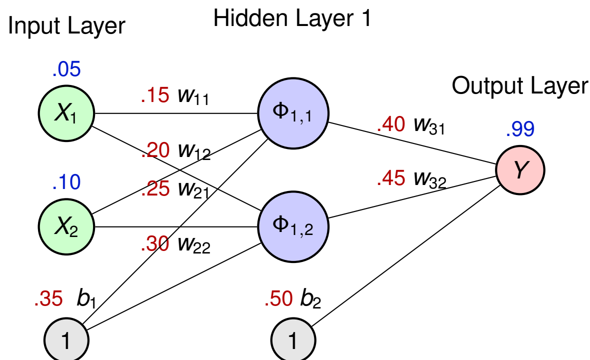
On commence par faire une passe avant pour prédire la sortie du réseau de neurones avec les poids et les biais courants pour l'entrée 0.05 et 0.10.



L'entrée du noeud $\Phi_{1,1}$ est $vh2 = w_{12}X_1 + w_{22}X_2 + b1 = 0.3925$
et sa sortie est
$yh2 = \rho(vh2) = 0.3925$

Exemple de rétropropagation

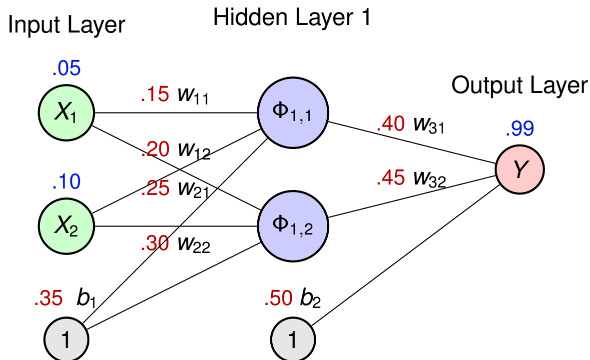
On commence par faire une passe avant pour prédire la sortie du réseau de neurones avec les poids et les biais courants pour l'entrée 0.05 et 0.10.



La sortie est
$\hat{Y} = w_{31}\Phi_{1,1} + w_{32}\Phi_{1,2} + b_2 = 0.40 * 0.3775 + 0.45 * 0.3925 = 0.8276$

Exemple de rétropropagation

On peut maintenant calculer l'erreur



$$e = Y - \hat{Y} = 0.1624$$

$$E = \frac{1}{2}(e)^2 = \frac{1}{2}(0.99 - .8276)^2 = 0.0132$$

Exemple de rétropropagation

Le but de la rétropropagation est de mettre à jour tous les poids et les biais de sorte que la prédiction soit plus proche de la sortie attendue.

On applique les formules de la descente de gradient avec le taux d'apprentissage $\alpha = 0.1$.

On commence par les poids de la couche de sortie. On rappelle que la fonction d'activation de la couche de sortie est l'identité

$$\frac{\partial E}{\partial w_{3j}} = \frac{\partial E}{\partial e} \frac{\partial e}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial w_{3j}}$$

$$\frac{\partial E}{\partial w_{31}} = e(-1)yh1 = -0.1624 * 0.3775 = -0.0613$$

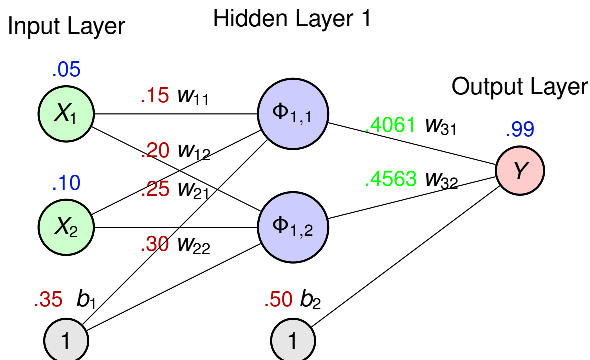
$$\frac{\partial E}{\partial w_{32}} = e(-1)yh2 = -0.1624 * 0.3925 = -0.0637$$

et la mise à jour

$$w_{31}^{\text{new}} = w_{31} - \alpha \frac{\partial e}{\partial w_{31}} = 0.40 + 0.1 * 0.0613 = 0.4061$$

$$w_{32}^{\text{new}} = w_{32} - \alpha \frac{\partial e}{\partial w_{32}} = 0.45 + 0.1 * 0.0637 = 0.4563$$

Exemple de rétropropagation



Exemple de rétropropagation

On s'intéresse maintenant aux poids de la couche cachée.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e} \frac{\partial e}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial \Phi_{1,j}} \frac{\partial \Phi_{1,j}}{\partial w_{ij}}$$

$$\frac{\partial \hat{Y}}{\partial \Phi_{1,j}} = w_{3,j}$$

En notant $\mathbb{I}_A(x)$ la fonction indicatrice qui vaut 1 si $x \in A$ et 0 ailleurs,

$$\frac{\partial \Phi_{i,j}}{\partial w_{i,j}} = X_i \rho'(vhi) = X_i \mathbb{I}_{[0,+\infty[}(vhi)$$

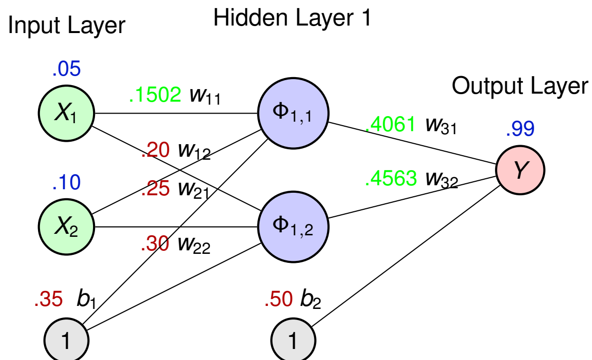
Par exemple,

$$\frac{\partial E}{\partial w_{11}} = e(-1) w_{31} X_1 \mathbb{I}_{[0,+\infty[}(vh1) = -0.1624 * 0.40 * 0.05 = -0.0032$$

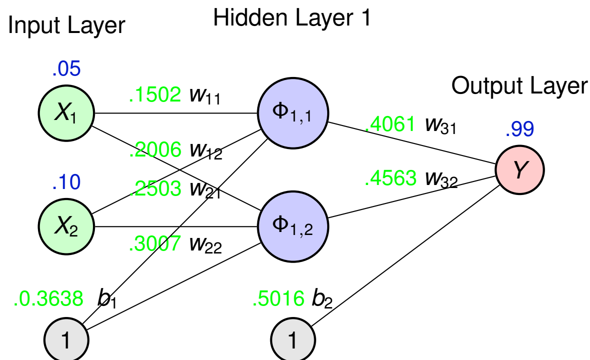
et

$$w_{11}^{\text{new}} = 0.15 + 0.1 * 0.0023 = 0.1502$$

Exemple de rétropropagation



Exemple de rétropropagation



Une passe avant avec les poids et les biais mis à jour conduit à $\hat{Y} = 0.9248$.
On s'est donc bien approché de la cible qui est 0.99.

En itérant les passes avant et les rétropropagations, on va petit à petit atteindre la cible.

Lors de la rétropropagation, le gradient de la fonction de perte est transporté de chaque couche vers la précédente.

Stochastic Gradient Descent (SGD)

L'algorithme de descente du gradient stochastique est inspiré de l'algorithme de descente de gradient, mais à chaque itération, le pas est calculé à partir d'un seul exemple.

$$L(\mathbf{w}) = \sum_{i=1}^{n} e_i(\mathbf{w}) \text{ et } e_i(\mathbf{w}) = (y_i - \phi(x_i; \mathbf{w}))^2$$

Algorithme de descente du gradient

Initialisation : choisir un point $\mathbf{w}_0 \in \Omega$
Tant que $\nabla L(\mathbf{w}) > \epsilon$ et $k <$ maxiter faire
$\quad \text{pas}_k = -\alpha_k \nabla_{\mathbf{w}} L(\mathbf{w}_k)$
$\quad \mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{pas}_k$

Algorithme de descente du gradient stochastique (strict)

Initialisation : choisir un point $\mathbf{w}_0 \in \Omega$
Tant que $\nabla L(\mathbf{w}) > \epsilon$ et $k <$ maxiter faire
$\quad$ Tirer aléatoirement i dans $\{1, \cdots, n\}$
$\quad \text{pas}_k = -\alpha_k \nabla_{\mathbf{w}} e_i(\mathbf{w})$
$\quad \mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{pas}_k$

Trouver les différences !

Stochastic Gradient Descent (SGD)

Les avantages

- pour une itération on fait (beaucoup) moins de calculs (pour calculer le gradient)
- on explore mieux l'espace des possibles et on a donc moins de chances de rester bloqué dans un minimum local

Les inconvénients

- la "descente" est stochastique : les valeurs des poids varient beaucoup d'une itération à l'autre
- la convergence peut être lente (on fait plus d'itérations...)

Le compromis du **mini-batch** : on utilise un petit sous ensemble d'exemples. Cet ensemble est appelé **mini batch**.

Stochastic Gradient Descent (SGD)

Trouver les différences !

Algorithme de descente du gradient stochastique (strict)

Initialisation : choisir un point $\mathbf{w}_0 \in \Omega$
Tant que $\nabla L(\mathbf{w}) > \epsilon$ et $k <$ maxiter faire
  Tirer aléatoirement i dans $\{1, \cdots, n\}$
  $\text{pas}_k = -\alpha_k \nabla_{\mathbf{w}} e_i(\mathbf{w})$
  $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{pas}_k$

Algorithme de descente du gradient stochastique (avec mini batch)

Initialisation : choisir un point $\mathbf{w}_0 \in \Omega$
Tant que $\nabla L(\mathbf{w}) > \epsilon$ et $k <$ maxiter faire
  Tirer aléatoirement un ensemble MB de b indices dans $\{1, \cdots, n\}$
  $\text{pas}_k = -\alpha_k \nabla_{\mathbf{w}} \sum_{i \in MB} e_i(\mathbf{w})$
  $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{pas}_k$

Avantages des minibatchs : on utilise moins de mémoire (seules les données du minibatch sont chargées), on fait un nombre de calculs raisonnable, on obtient une meilleure généralisation
(meilleures prédictions sur l'ensemble de test)

Multi Layers Perceptron (MLP)

Pour ajuster un modèle de type perceptron multicouches, on peut utiliser le module python scikit learn. La classe MLPRegressor, s'utilise comme suit.

```
from sklearn.neural_network import MLPRegressor

mlp = MLPRegressor(
       hidden_layer_sizes=(4,4), # architecture du MLP
       activation="relu", # fonction d'activation
       solver="sgd",  # algorithme d'optimisation
       batch_size=512, # taille du batch
       learning_rate="constant", # variation du taux d'apprenti
       learning_rate_init=0.0001, #  taux d'apprentissage
       max_iter=2000, # nombre maximum d'itérations
       shuffle=False, # permutations aléatoires des données
                      # change ou non (mélange) les batchs
       tol=0.0001) # précision}
mlp.fit(Xtrain ,Ytrain )
yval = mlp.predict(Xval)
```

...plus d'hyper-paramètres sur la page de documentation.

Multi Layers Perceptron (MLP)

```
mlp = MLPRegressor(
       hidden_layer_sizes=(4,4), # architecture du MLP
       activation="relu", # fonction d'activation
       solver="sgd",  # algorithme d'optimisation
       batch_size=512, # taille du batch
       learning_rate="constant", # variation du taux d'apprenti
       learning_rate_init=0.0001, #  taux d'apprentissage
       max_iter=2000, # nombre maximum d'itérations
       shuffle=False, # permutations aléatoires des données
                      # change ou non (mélange) les batchs
       tol=0.0001) # précision}
```

Dans un algorithme de descente de gradient (stochastique), on a souvent intérêt à faire décroitre le **taux d'apprentissage** : plus on s'approche de la solution, plus on veut être précis.

Multi Layers Perceptron (MLP)

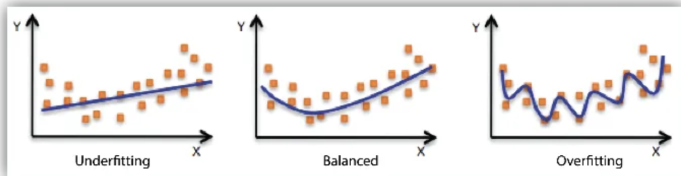
```
mlp = MLPRegressor(
       hidden_layer_sizes=(4,4), # architecture du MLP
       activation="relu", # fonction d'activation
       solver="sgd",  # algorithme d'optimisation
       batch_size=512, # taille du batch
       learning_rate="constant", # variation du taux d'apprenti
       learning_rate_init=0.0001, #  taux d'apprentissage
       max_iter=2000, # nombre maximum d'itérations
       shuffle=False, # permutations aléatoires des données
                      # change ou non (mélange) les batchs
       tol=0.0001) # précision}
```

La **taille des mini-batchs**, le **nombre d'itérations** et le **taux d'apprentissage** sont des hyper-paramètres qui sont liés entre eux. En général,

- quand on diminue la taille des batchs, on augmente le nombre d'itérations
- quand on diminue le taux d'apprentissage, on augmente le nombre d'itérations

Le problème du sur-ajustement (overfitting)

Le **sur-ajustement (overfitting)** se produit lorsqu'un modèle d'apprentissage automatique se concentre trop sur les détails spécifiques et le bruit dans les données d'entraînement, plutôt que d'apprendre les schémas généraux. C'est comme un étudiant qui mémorise les exemples d'un manuel mais ne comprend pas les concepts sous-jacents.



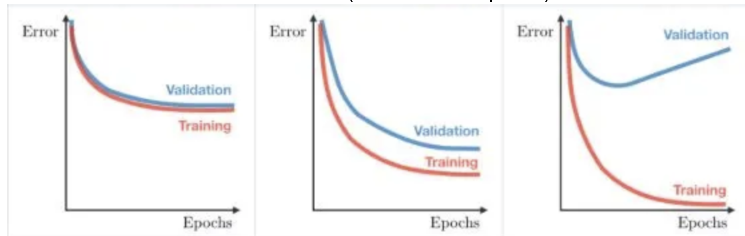
Les réseaux de neurones impliquent un (très) grand nombre de paramètres (les poids et les biais). Ils sont donc souvent assez flexibles pour "apprendre" à reproduire des détails.

Par exemple, avec 18 variables explicatives, si on choisit une architecture avec 2 couches cachées et 10 neurones chacune, on a 18*10+10*10+3=283 inconnues à estimer.

Diagnostiquer le sur-ajustement (overfitting)

En cas de sur apprentissage, le modèle entrainé a de mauvaises performances de généralisation : la valeur de la fonction de perte de l'ensemble de validation est bien supérieur à celle de l'ensemble d'entrainement.

On trace l'évolution des erreurs (fonctions de perte) en fonction des itérations



- **"under fitting"** les 2 erreurs sont grandes (gauche)
- **"good fitting"** les deux erreurs décroissent et l'erreur d'entrainement est un peu meilleure (milieu)
- **"over fitting"** l'erreur de validation commence à croitre alors que l'erreur d'entrainement continue à décroitre (droite)

Prévenir le sur-ajustement (overfitting)

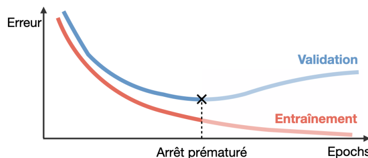
Pour prévenir le sur apprentissage (overfitting), il existe plusieurs solutions qu'on peut combiner

- On ajoute un terme de **régularisation** à la fonction qu'on optimise :

$$\min_{\mathbf{w}} \underbrace{L(X, y; \mathbf{w})}_{\text{diminue avec } \mathbf{w}} + \lambda \underbrace{||\mathbf{w}||_2}_{\text{augmente avec } \mathbf{w}}$$

  pour contrôler l'augmentation de $||\mathbf{w}||_2$, l'algorithme d'optimisation cherche à mettre des poids nuls aux variables/neurones qui ne permettent pas de faire décroitre la fonction de perte $L(X, y; \mathbf{w})$.
  la constante λ sert à gérer l'équilibre entre les deux termes [5].
  Exemple : https://playground.tensorflow.org

- On peut par ailleurs utiliser l'astuce de **l'arrêt prématuré (early stopping)** qui consiste à stopper l'algorithme d'optimisation quand la fonction de perte de validation commence à croitre.



5. C'est l'argument α dans sklearn !

Prévenir le sur-ajustement (overfitting)

```
mlp = MLPRegressor(hidden_layer_sizes=(4,4),
        activation="relu", # fonction d'activation
        solver="sgd",  # algorithme d'optimisation
        batch_size=512, # taille du batch
        learning_rate="constant", # variation du taux d'apprenti
        learning_rate_init=0.0001,
        early_stopping=True, # arrêt prématuré
        validation_fraction=0.1, # pourcentage de l'ensemble
                                 # d'entrainement
                                 # utilisé pour la validation
        n_iter_no_change=10,  # nombre d'itérations
                              # sans amélioration
                              # de la fonction de perte
                              # de validation
                              # pour l'arrêt prématuré
        alpha=1) # constante de régularisation
                 # (noté lambda ci-dessus)
mlp.fit(Xtrain ,Ytrain )
```

Prévenir le sur-ajustement (overfitting)

L'idée de la régularisation existe aussi pour la régression linéaire multiple. Cette méthode permet de donner plus de poids aux variables explicatives qui sont vraiment importantes.

Quand on utilise la régularisation en norme L2 (comme pour les réseaux de neurones), la méthode de régression linéaire s'appelle aussi **régression linéaire Ridge** .

```
from  sklearn import linear_model

ridge = linear_model.Ridge(alpha=.1)
ridge.fit(Xtrain ,Ytrain )
```

Normalisation

Quand on utilise des réseaux de neurones, on normalise toujours les entrées de façon à ce qu'elles prennent leurs valeurs entre 0 et 1. Ceci permet d'éviter que les sortie des neurones prennent des valeurs trop grandes.

On calcule les bornes à aprtie de l'ensemble d'entrainement `Xtrain`. Et on applique strictement la même transformation aux ensemble d'entrainement `Xtrain`, de validation `Xval` et de test `Xtest`.

```
from sklearn.preprocessing import MinMaxScaler

caler = MinMaxScaler()
Xtrain = scaler.fit_transform(Xtrain)
Xtest = scaler.transform(Xtest)
Xval = scaler.transform(Xval)
```

Récapitulatif

Dans cette partie du cours, nous avons étudié deux modèles importants en intelligence artificielle

- le modèle de régression linéaire (simple et multiple) et abordé (très rapidement à la régression Ridge,
- les réseaux de neurones de type MLP.

Nous avons étudié des algorithmes d'optimisation parmi les plus utilisés en intelligence artificielle

- algorithme de descente de gradient,
- algorithme de descente de gradient stochastique (sgd).

Nous avons aussi insisté sur l'importance de la validation croisée pour

- mesurer la performance d'un modèle et comparer des modèles entre eux ;
- pour diagnostiquer le sur ajustement.

Les étapes d'un projet de Machine Learning

Les principales étapes d'un projet de machine learning sont

1. préparation des données
   - nettoyage éventuel (valeurs manquantes, valeurs aberrantes, etc)
   - préparation des ensemble d'entrainement, de validation (et de test)
   - normalisation
2. choix d'un ou plusieurs algorithmes/modèles
3. choix des hyper-paramètres
   - architecture
   - algorithme d'optimisation
   - taille des batchs
   - constante de régularisation
   - etc
4. ajustement des paramètres (poids, biais)
5. calcul des scores de validation
6. mise en forme des résultats (tableaux, graphiques, etc) et rédaction (présentation de la méthodologie et analyse critique des résultats)

Remarques
- Il est courant de revenir à l'étape 2 après l'étape 5 et de répéter plusieurs fois le cycle.
- On peut utiliser la validation croisée pour choisir les hyper-paramètres (voir par exemple `sklearn.model_selection.cross_val_score`)

Le mot de la fin

Si les réseaux de neurones sont un des éléments clé de l'intelligence artificielle, il existe d'autres algorithmes en machine learning (voir cours Introduction à la Science de Données (ISD) en L3).
Il vous reste plein de choses à découvrir !

Les informaticiens sont bien identifiés dans le monde de l'intelligence artificielle, mais les mathématiciens sont à l'origine de la plupart des avancées les plus importantes dans ce domaine !

Participez au challenge Kaggle pour **prédire la popularité d'un morceau de musique sur spotify** : votre première expérience en tant qu'acteur de l'IA !