Algorithmes et Structures de données
Introduction aux réseaux de neurones

INF TC1
2017-18 (S5)

Version Élève

☞ En séance 2 (et plus), aller directement à la section "Optimisation" section V page 11.
☞ La lecture des pages qui suivent permet la compréhension du reste du sujet. N'évitez pas cette lecture !

I Introduction

Un réseau de neurones (RN) est un dispositif d'apprentissage **entraîné** à utiliser ses entrées afin de prédire ses sorties.

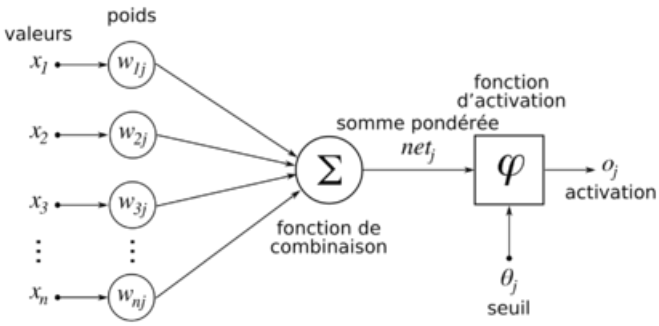
Supposons la table (de vérité) ci-contre.

Entrées			Sortie
x_1	x_2	x_3	y
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

Dans cette table, pour les 3 entrées x_i, on a $y = x_1 \wedge x_3$ (voire, y est identique à X_1 plus simplement !)
Un RN permet de mesurer / calculer ce type de relations entre ses entrées et ses sorties.
Une fois entraîné, le RN se comportera comme une boîte noire qui produira une sortie pour une combinaison de valeurs en entrée.

Un RN est en général composé d'une succession de couches dont chacune prend ses entrées sur les sorties de la précédente. Dans le cas simple présent, nous n'aurons que deux couches [1].

Dans notre RN, la couche d'entrée est composée de 3 neurones (x_i). Ces neurones seront connectés par une liaison (synapse) à la couche suivante (qui sera notre couche des sorties).
À chaque *synapse* est associé un poids synaptique.
Ainsi, les entrées sont multipliés par ce poids synaptique puis additionnés ; ce qui est équivalent à multiplier le vecteur d'entrée par une matrice de transformation (voir le *produit* '*' ci-dessous). Ceci constitue notre fonction de combinaison.
Le résultat de cette multiplication sera ensuite passé (en paramètre) à une fonction d'activation qui produira une valeur pour chaque neurone de la couche suivante (ici, pour notre unique neurone des sorties).
➛ Par exemple, si la somme pondérée ($\sum$) dépasse le seuil 0.75, on émettra $O_j = 1$ en sortie ("on active").

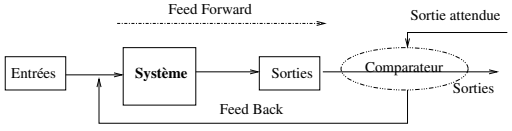


Structure d'un neurone artificiel (Fig. de Wikipédia).

Le neurone calcule la somme pondérée de ses entrées puis cette valeur passe à travers la fonction d'activation pour produire sa sortie.

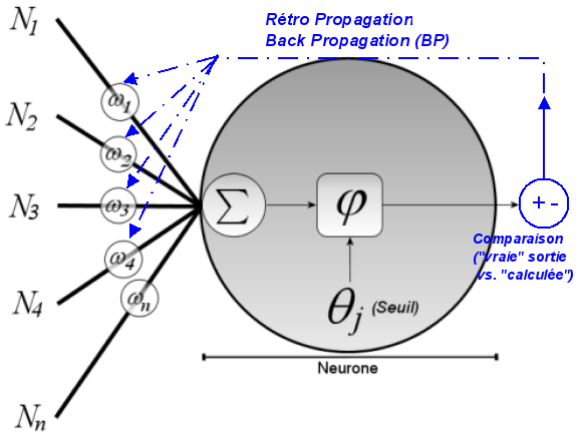
La rétro-propagation
Pour mieux apprendre un concept par notre RN [2], chaque neurone des sorties **propage** son nouvel état vers ses synapses via un **axone**. Cette action est appelée la **retro-propagation**. En biologie, un axone est le prolongement du neurone qui conduit le signal électrique du corps cellulaire vers les zones synaptiques.

• Le principe de la rétro-propagation (correction des entrées du système en fonction de l'erreur) est connu dans divers dispositifs : dans le cerveau humain (et animal) quand on apprend quelque choses (une langue, une activité nouvelle), en Automatique , en Contrôle de procédé, ...) :



Boucle en feed-back (information rétrogradée).

La comparaison est faite entre la sortie calculée et la sortie réelle.



Dans la figure de relative à notre RN ci-dessus (à droite), on note la "correction" des pondérations en fonction de l'erreur. [3]

Proposons le code Python de notre RN à 2 couches :

1. Nous verrons dans la section suivante un exemple de RN à 3 couches.
2. Un RN classique peut "apprendre des concepts" (comme *chien*, *homme*, *véhicule*, ...), mais apprendra bien plus difficilement des règles (comme *la règle de 3*) ou des procédures (séquence d'application de règles : comme pour jouer au *Morpion*).
3. Dans certains cas, les entrées participent également à la comparaison.

I-1 Un réseau de neurones (RN) simple

• Lire les explications qui suivent ce code.

```
# La sortie Y=1 si x1=1 (ou bien y=x1 and x3)
# Pas de hidden couche, donc les résultats ne seront pas top.

import numpy as np

# La fonction de combinaison (ici une sigmoide)
def sigmoide(x) :  return 1/(1+np.exp(-x))

# Et sa dérivée
def derivee_de_sigmoide(x) : return x*(1-x)

# Les entrées
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# Les sorties (ici un vecteur)
Y = np.array([[0,0,1,1]]).T     # Transposé

# On utilise seed pour rendre les calculs déterministes.
np.random.seed(1)

# Initialisation aléatoire des poids (avec une moyenne = 0)
synapse0 = 2*np.random.random((3,1)) -1

couche_entree = X

for iter in range(10000):     # On peut augmenter !
    # propagation vers l'avant (forward)
    couche_sortie = sigmoide(np.dot(couche_entree,synapse0))     # dot multiplication

    # Quelle est l'erreur (l'écart entre les sorties calculées et attendues)
    erreur_couche_sortie = Y - couche_sortie

    # Multiplier l'erreur (l'écart) par la pente du ïsigmode pour les valeurs dans couche_sortie
    delta_couche_sortie = erreur_couche_sortie * derivee_de_sigmoide(couche_sortie)

    # Mise à jour des poids : rétropropagation
    synapse0 += np.dot(couche_entree.T,delta_couche_sortie)

print ("Les sorties après l'apprentissage :")
print (couche_sortie)
```

A propos du code python précédent :

X : La matrice des entrées (données d'apprentissage) : chaque ligne est une observation.

Y : La matrice des sorties (attendues) correspondant aux entrées.

couche_entree : Première couche du RN, contenant les données en entrée

couche_sortie : Seconde couche ou la `couche des sorties` (ici, on n'a que 2 couches).

synapse0 : Pondérations (appelées *Synapse*) qui connectent les noeuds de la `couche_entree` à ceux de la `couche_sortie`.

* : produit élément par élément de deux vecteurs $\overrightarrow{u}$ et $\overrightarrow{v}$ (de tailles identiques).
Le résultat $\overrightarrow{w}$ est un vecteur de la même taille et $w_i = u_i * v_i$

- : Soustraction élément par élément (Voir aussi l'Addendum).

x.dot(y) : Produit scalaire. Voir Addendum en section X-1 page 30 plus loin.

Résultat de l'exécution :

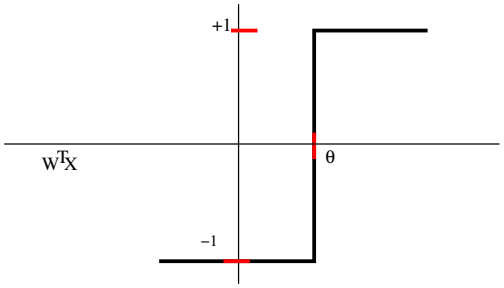
```
Les sorties après l'apprentissage :
[[ 0.00966449]
 [ 0.00786506]
 [ 0.99358898]  # Correspond à 1
 [ 0.99211957]] # Correspond à 1
```

➙ On remarque que la sortie correspond !

☞ Rappel : la fonction `derivee_de_sigmoide(x)` nous donne la pente de la tangente en $(x, sigmoide(x))$. Retenez-le pour plus tard.
Comme vous l'avez remarqué, pour la fonction $U = sigmoide(.)$, la fonction `derivee_de_sigmoide` est de la forme $U(1-U)$ (elle n'est pas vraiment égale à $U' = \frac{dU}{dx}$ et pour cause, $e^{(.)}$ n'y figure pas !). Car en fait, on applique U' ici à `couche_sortie` qui est le résultat de l'application de U à `couche_entree` : la dérivée est donc appliqué à U, pas à x !

A propos de l'activation de neurones :
La fonction **échelon** appelée également la marche d'escalier ou encore fonction de **Heaviside** (ou *Unit Step Function*), est la forme la plus simple d'une fonction d'activation. Cette fonction basique, simpliste et élémentaire d'activation de neurone se définit par :

$$h(z) = \begin{cases} 1, & \text{si } z \geq \theta. \\ -1, & \text{sinon.} \end{cases}$$



où $z = \sum_{i=1}^{m} w_i x_i = w_1 x_1 + w_2 x_2 + ... + w_m x_m = W^T X$
avec x_i les entrées des neurones et w_i les pondérations.
m = nombre de variables explicatives.
θ représente un seuil d'activation.

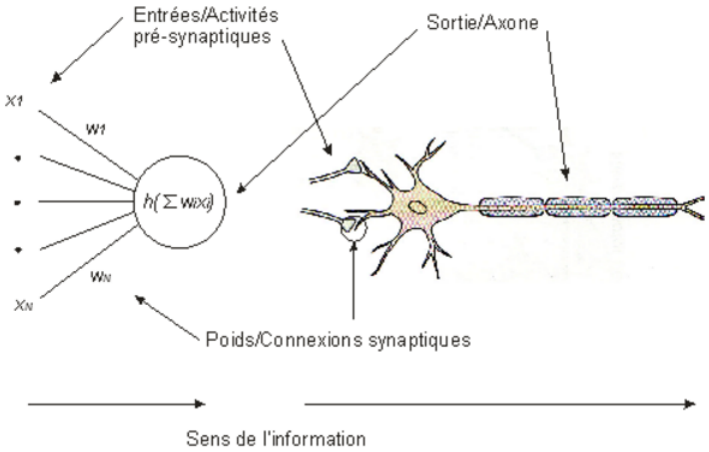
En fait, pour mieux coller à la définition de **Heaviside**, on peut supposer $x_0 = 1$ et $w_0 = -\theta$ et écrire :

$$h(z) = \begin{cases} 1, & \text{si } z \geq 0. \\ -1, & \text{sinon.} \end{cases}$$

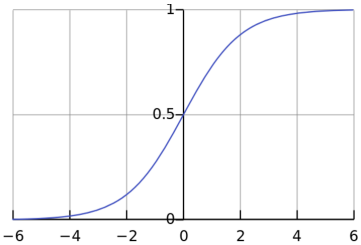
avec $z = \sum_{i=0}^{m} w_i x_i = w_1 x_1 + w_2 x_2 + ... + w_m x_m = W^T X$ où $x_0 = 1$ et $w_0 = -\theta$

La fonction **sigmoïde** (ou *logistique* [4] ou encore *escalier lissé*) que nous utilisons dans notre RN représente une forme non linéaire et continue de la fonction linéaire *échelon* ci-dessus [5]. [6]

Comparaison d'un neurone artificiel (à gauche) avec un neurone biologique (à droite).



I-2 Remarques sur le code Python



• La fonction `sigmoide` associe toute valeur à l'intervalle $[0, 1]$.
On l'utilise ici pour associer une probabilité à un nombre. Par sa simplicité, elle présente plusieurs propriétés utiles dans un RN.

• La fonction `derivee_de_sigmoide` calcule la dérivée de la sigmoïde.
Cette dérivée est comme la valeur de la pente de la sigmoïde à un point donné (différents points ont différentes pentes).

4. D'où l'appellation *neurones logistiques*.

5. La fonction sigmoïde : $f(z) = \dfrac{1}{1+e^{-z}}$ où $z = \sum_{i=0}^{m} w_i x_i = W^T X$

6. Pour être exact par rapport à la fonction logistique, on devrait écrire $z \equiv w_{1..m}.x_{1..m} + b$ où $b = w_0$ est une constante. En supposant $x_0 = 1$, on harmonise l'expression de z par $\sum_{i=0}^{m} w_i x_i$: ce qui équivaut à ajouter un biais dans le RN matérialisé par une entrée fictive supplémentaire $x_0 = 1$ ainsi que la pondération w_0 qui la relie à la première couche. Par conséquent, on peut écrire $z = w.x = \sum_{i=0}^{m} w_i x_i$. Nous ne représentons pas ce biais ici.

• La variable X représente les variables **explicatives**. Elle représente également la couche d'entrée et contient la matrice des données où chaque ligne correspond à un exemple d'apprentissage (une *observation*).
Le vecteur Y est la variable *expliquée* et le couple (X, y) (l'espace des données) représente les *observées* (nous avons 4 observations).
Notons que dans la matrice X, chaque colonne est liée à un même noeud du RN. De ce fait, nous avons 3 noeuds en entrée et un (pour y) en sortie.

• Le symbole ".T" est la fonction **transposée**. Après l'application de cette fonction, le vecteur y sera en colonne. Ce qui correspond aux 4 lignes de la table de vérité de notre exemple.

• On note également l'utilisation de la fonction `seed` (la souche d'initialisation des nombres aléatoires). L'utilisation de `seed` (toujours avec une même valeur) rend le code déterministe : toutes les exécutions donneront la même séquence de sorties et donc les mêmes résultats. Dans une version finale, on pourra au besoin supprimer `seed` pour éviter ce déterminisme.

• La variable `synapse0` représente la matrice des pondérations reliant chaque noeud de `couche_entree` au noeud de `couche_sortie`. Cette dernière est en fait un vecteur de 3 lignes et d'une colonne (on a 3 noeuds en entrée = `couche_entree` et un en sortie = `couche_sortie`).

Pour garder une trace de notre apprentissage, nous devons sauvegarder cette matrice (les couches d'entrées et les sorties n'en ont pas besoin) ainsi que la configuration du réseau (nb couches cachées, dimension des entrées / sorties). Par contre, pour une utilisation ultérieure de ce même réseau, seule la matrice des pondérations sera nécessaire.

Notre RN ayant seulement deux couches (pour l'instant), nous avons besoin d'une seule matrice de pondération qui connecte ces deux couches.

L'initialisation des pondérations est faite par les nombres aléatoires de moyenne nulle (fonction `random`). C'est une pratique courante dans les RNs.

☞ A propos de *random* :
Sous Python, `random.random()` génère un nombre aléatoire dans le demi ouvert $[0, 1)$ (ou $[0, 1[$).
Le module `numpy` propose les mêmes fonctions mais avec un plus grand choix de distributions (voir section X-2, page 31).
Sachant que `random.random()` (également notée `random_sample()`) de `numpy` génère un réel dans le demi-ouvert $[0, 1)$, pour obtenir un réel distribué uniformément dans $[a, b)$, il suffit d'écrire :

```
(b - a) * random.random() + a
```

➙ Par exemple, pour obtenir des réels uniformes dans $[-5, 0)$, on écrira `5 * random.random() -5`

Dans le code Python ci-dessus, pour obtenir des réels dans $[-1, 1)$ (donc de **moyenne nulle**), on avait écrit :

```
2 * random.random() -1
```

I-3 Déroulement du code Python

• Avec le peu de données disponibles en entrée, notre apprentissage se fait avec toutes ces données[7]. La variable `couche_entree` reçoit ces données avant le début des itérations.

A l'intérieur de l'itération, on procède à une prédiction (calcul) des sorties en fonction des entrées.
On commence par la multiplication de `couche_entree` par `synapse0` (via le produit scalaire – *dot-product* –) qui met à jour la valeur de chaque noeud dans `couche_sortie`.
Ensuite, on passe notre sortie par la sigmoïde (penser à une fonction – d'activation – de seuil).

A chaque itération, `couche_sortie` contient notre prédiction pour chaque (triplet d') entrée. On calcule notre erreur de prédiction par une soustraction (entre la sortie attendue y et `couche_sortie`)[8]. Notons également que ces erreurs peuvent être de signes différents (positifs ou négatifs)[9].

7. On dira qu'avec cette table seule, ce RN "résume" les données puisqu'il n'y en a pas d'autres (donc rien à prédire) !
8. Notons que cette distance (écart) est calculée (et minimisée) par la somme des carrées (dite *la moindre carrée* si minimisée, voir section IX) : $SSE = \sum_0^m (y_i - f(x_i))^2$ où $f(.)$ représente la sortie calculée par notre réseau ; elle peut être *l'erreur quadratique moyenne* qui est $MSE = \mathbb{E}(\text{couche\_sortie}^2 - Y^2)$. Noter que $MSE(\text{couche\_sortie}|Y) = \text{Biais}(couche\_sortie)^2 + \text{Var}(couche\_sortie)$; le *Biais* et la *Variance* sont deux des caractéristiques principales dont on cherche un *équilibre* dans tout système d'apprentissage.
9. Dans certains cas, on peut être intéressé à leur valeur absolue.

☞ Détaillons le code à cet endroit pour une (petite) introduction au gradient. Nous allons avoir recours au principe du calcul de l'erreur pondérée par la dérivée (gradient) [10].
L'erreur (de combien se trompe-t-on par rapport à la fonction d'activation sigmoïde) était :

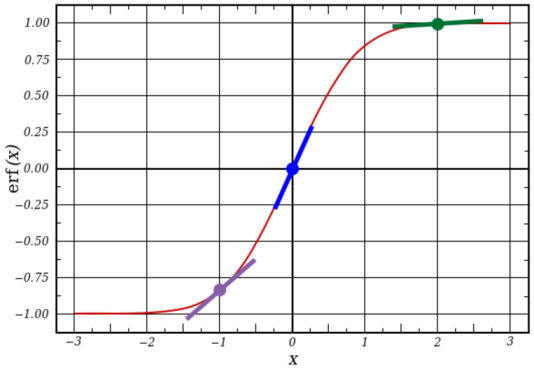
```
    # Quelle est l'erreur (l'écart entre les sorties calculées et attendues)
    erreur_couche_sortie = Y - couche_sortie
```

Pour calculer l'écart pondéré sur `couche_sortie`, on multiplie cette erreur par la pente de la (tangente à la) sigmoïde pour chaque noeud de `couche_sortie` [11].

```
delta_couche_sortie = erreur_couche_sortie * derivee_de_sigmoide(couche_sortie)
```

Pourquoi ?
En fait, plus l'erreur est élevée, plus il faut la corriger (par la pente). En multipliant l'erreur par la pente, on réduit l'erreur pour les prédictions de **bonne qualité**. On peut remarquer ceci sur la figure ci-dessous.

Dans la figure ci-dessous, si 3 valeurs possibles de `couche_sortie` représentent les 3 points repérés sur la sigmoïde, on calcule leur pente (par la fonction `derivee_de_sigmoide()`). Notons que ces dérivées sont entre 0 et 1.
Remarquons également que de grandes valeurs (cf. point vert) ou les petites valeurs (cf. point mauve) ont une pente douce. La plus grande pente, le point bleu, joue un rôle important.



On peut constater sur la courbe de notre sigmoïde que si la pente est douce (proche de 0), les prédictions ont une grande ou une petite valeur (proche de 1 ou proche de zéro) ; ce qui témoigne de la qualité de la prédiction dans notre cas (avec nos valeurs binaires) [12].
A contrario, si on a une prédiction de moins bonne qualité (proche du point $< 0, 0.5 >$), alors en multipliant les erreurs importantes par de plus grandes valeurs de pente, on tente de **sanctionner** ces erreurs. Ce principe de **pénalité**permet donc à la fois de conserver les *bonnes* prédictions (multipliées par une valeur moindre – de pente douce – proche de nulle) et de pénaliser les prédictions de moins bonne qualité [13].

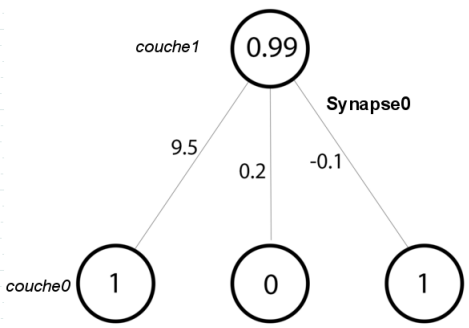
Enfin, les pondérations sont mises à jour par le dernier produit vectoriel (*dot-product*) de l'itération :

```
    # Mise à jour des poids
    synapse0 += np.dot(couche_entree.T, delta_couche_sortie)
```

Effet concret sur les pondérations : exemple chiffré
Pour illustrer ce point, considérons un seul des exemples d'apprentissage en entrée (la 3e entrée 101 donnant la sortie $y = 1$).

10. Il existe d'autres solutions mathématiques que `l'erreur pondérée par la dérivée`
11. La variable `erreur_couche_sortie` est une matrice (4,1) et `derivee_de_sigmoide(couche_sortie)` renvoie une matrice (4,1). On multiplie cette dernière par l'erreur `erreur_couche_sortie` produisant la matrice (4,1) `delta_couche_sortie`.
12. La représentation binaire des entrées et des sorties dans les RNs est une pratique courante et il existent des méthodes de conversion de tout type de valeur en binaire
13. Vue la sigmoïde, les meilleures prédictions sont celles proches de 0 ou de 1 et les moins bonnes se situent au milieu de ces deux extrêmes

Ici, on a une pondération = 0.95 qui sera mise à jour par les opérations suivantes.
Supposons que le neurone de sortie "affiche" la valeur 0.99. Elle est de bonne qualité et très proche de 1. La pente (la dérivée) sera d'autant plus douce.



Comme expliqué ci-dessus, on a :
`y = 1` (du vecteur des sorties)
`erreur_couche_sortie = y - couche_sortie = 0.01`
`delta_couche_sortie = 0.01 * petite_valeur_de_pente`
En multipliant la valeur 0.99 (bonne prédiction) par `delta_couche_sortie`, 9.5 sera très peu modifiée car la prédiction étant déjà juste, on la préserve. Ainsi, une petite valeur de pente signifie une toute petite mise à jour.

Comme nous l'avons indiqué ci-dessus, la dernière action de l'itération calcule les valeurs de mise à jour pour chaque entrée, les additionne puis met à jour les pondérations ($synapse0$).

On constate dans notre exemple, que lorsque les entrées et les sorties sont 1, on augmente la pondération. Et si une entrée = 1 et la sortie = 0, on diminue la pondération.

Ainsi, pour la table (rappelée ci-contre), la pondération entre la première valeur (colonne 1) et sa sortie **sera augmenté de manière importante ou restera inchangée** tandis que pour les deux autres colonnes de cette table (non corrélées avec leur sortie), les pondérations seront **TOUTES augmentées ou réduites** (à travers les 4 lignes des entrées).

Entrées (couche_entree)			Sorties (couche_sortie)
0	0	1	0
1	1	1	1
1	0	1	1
0	1	1	0

C'est ce qui conduit notre RN à apprendre en fonction des corrélations entre les entrées et les sorties.

II Travail-1 en séance

• Cette partie sera utile pour comprendre la suite.

• Recopiez le code fourni ; compilez et testez. En particulier :
◦ Comparer les valeurs de `couche_sortie` après la 1e itération avec celles issues de quelques autres valeurs d'itération en particulier celle de la dernière,
◦ Vérifier les variations de `erreur_couche_sortie` pendant les itérations ; afficher les erreurs successives (voire, créer une courbe) pour voir son évolution,
◦ Notez bien l'importance des lignes (calcul de `delta_couche_sortie`) ainsi que la dernière ligne (la mise à jour de `synapse0`).
◦ Notez également la fonction `sigmoide` ; c'est une fonction continue et dérivable qui donnera la sortie sous forme de probabilité,

◦ La première ligne du code Python fourni indique deux des relations possibles entre les entrées et la sortie
`La sortie :` $y = x_1$ `ou bien` $y = (x_1 \& x_3)$.
Compléter la table (2^3 lignes sont possibles), modifier les variable X et Y (du code Python) et tester.
➛ Comment le réseau apprend-il les 2 relations que vous avez envisagées (quelle erreur) ?
➛ Peut on affirmer l'hypothèse :
"Notre RN apprendre en fonction des corrélations entre les entrées et les sorties."

III Un exemple plus difficile

• Dans l'exemple précédent, les données (en particulier la première colonne) étaient corrélées à la sortie. Ce qui rendait l'apprentissage plus simple : les RNs apprennent bien ce type de corrélations.

• Considérons un exemple plus difficile où à première vue, on ne remarque pas de corrélation entre les entrées et les sorties. Ici, chaque colonne a $\frac{1}{2}$ chance de prédire la sortie (à la fois pour une sortie = 1 et = 0).

Entrées			Sorties
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Dans la table ci-contre, la colonne 3 est toujours à 1 mais on peut noter par exemple que la sortie est le résultat d'un *ou-exclusif* (XOR) sur les colonnes 1 et 2. De ce fait, il n'y a pas de relation linéaire entre les entrées et la sortie.
Dans l'exemple précédent, nous avions une correspondance linéaire un-à-un entre les entrées et les sorties mais ici (où la colonne 3 est toujours à 1) [14], la sortie est une combinaison des entrées [15]. En général, ce type de données nécessite un RN plus sophistiqué.

• Dans ces conditions, **une stratégie consiste à ajouter une couche (dite *cachée*) supplémentaire au RN** en vue de réaliser une sorte de *"transformation"* des entrées (pour extraire leurs *caractéristiques*) dans l'idée que cette couche (cachée) produira des résultats **intermédiaires** qui pourront à leur tour présenter une combinaison linéaire avec les sorties.

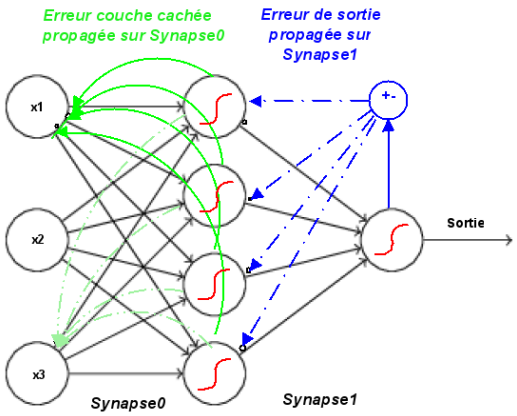
• Cette couche (la variable `couche_cachee`) [16] se place après la couche des entrées. Elle combinera les entrées ; produira des résultats intermédiaires qu'une seconde zone synaptique (matrice de pondérations) associera aux sorties. [17]

Pour ce faire, considérons la table suivante. Après avoir initialisé les pondérations pour la première couche cachée en amont (entre les entrées et un état intermédiaire calculé par les neurones de `couche_cachee`), on tentera d'établir une relation entre cet état (caché) et les sorties (en fait, un seul neurone de sortie).

Entrées (`couche_entree`)			Ex. de contenu de `couche_cachee` (= Entrées de `couche_sortie`)				Sorties (`couche_sortie`)
0	0	1	0.75	0.11	0.93	1	0
0	1	1	0.16	0.0003	0.019	0.9	1
1	0	1	1	0.9	0.02	0.84	1
1	1	1	0.95	0.024	0.00003	0.11	0

Dans cette table, on remarque **une certaine corrélation** entre la couche cachée avec les sortie.
Ainsi, par le code Python suivant, on tentera d'amplifier ces corrélations d'une part entre les entrées et l'état intermédiaire (par les pondérations `synapse0` comme dans l'exemple précédent) puis entre l'état intermédiaire et la sortie (par les pondérations associées `synapse1`).

➛ **La figure ci-contre** présente l'architecture du réseau à 3 couches. Notons que la retro-propagation se fait d'une couche vers la précédente.



14. En apprentissage automatique, cet invariant est recherché pour être supprimé permettant une réduction de dimension des données.
15. Aussi, on constate dans cette table que la sortie=1 si seulement deux des trois entrées=1.
16. La plupart des RNs fonctionnent suivant cette stratégie mais bien entendu, ils peuvent en utiliser d'autres !
17. Cette idée de *"transformation"* est également à la base du **Deep Learning** où, pour faire simple, les couches cachées supplémentaires (dites couches "profondes", placées en amont) dans le RN se chargent d'extraire des caractéristiques basiques des données qui constitueront les *attributs* (*features*) utilisés pour l'apprentissage.
Cette technique est également utilisée lorsque les entrées sont (les pixels) des images où à la place d'une relation directe entre les pixels en entrée et le contenu de l'image, il y a une relation entre une combinaison plus ou moins complexe des pixels en entrée et la valeur de la sortie. On cherchera à répondre aux questions telles que "a-t-on un chien dans l'image ?" ; "s'agit-il d'une image d'extérieur ou d'intérieur ?" ou encore toute autre recherche de motifs habituellement réalisée par une "segmentation" basée sur les "invariants" dans les images. On espère néanmoins que les combinaisons de pixels ne soient pas aléatoires !
Enfin, on peut projeter les données vers un autre espace (dit espace des **caractéristiques**) où une corrélation serait plus facile à trouver entre les entrées et les sorties.

III-1 Le code Python

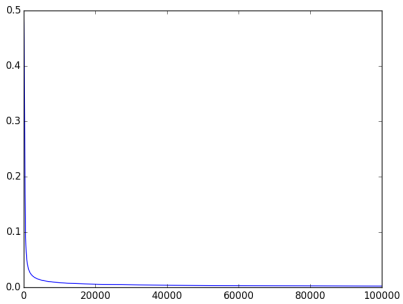
```
import numpy as np

def sigmoide_et_sa_derivee(x,deriv=False):
    if(deriv==True):
        return x*(1-x)

    return 1/(1+np.exp(-x))

X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

# On utilise seed pour rendre les calculs déterministes.
np.random.seed(1)

# Initialisation aléatoire des poids (avec une moyenne = 0 et écart-type=1)
# Ici, on met la moyenne à zéro, le std ne change pas.
# L'écriture X = b*np.random.random((3,4)) - a
# permet un tirage dans [a,b], ici entre b=1 et a=-1 (donc moyenne=0)
synapse0 = 2*np.random.random((3,4)) - 1
synapse1 = 2*np.random.random((4,1)) - 1

couche_entree = X
nb_iterations = 100000
for j in range(nb_iterations):

    # propagation vers l'avant (forward)
    # couche_entree = X
    couche_cachee = sigmoide_et_sa_derivee(np.dot(couche_entree,synapse0))
    couche_sortie = sigmoide_et_sa_derivee(np.dot(couche_cachee,synapse1))

    # erreur ?
    erreur_couche_sortie = y - couche_sortie

    if j % (nb_iterations//10) == 0:          # des traces de l'erreur
        print("Moyenne Erreur couche sortie :" + str(np.mean(np.abs(erreur_couche_sortie))))

    # pondération par l'erreur (si pente douce, ne pas trop changer sinon, changer pondérations,

    delta_couche_sortie = erreur_couche_sortie*sigmoide_et_sa_derivee(couche_sortie,deriv=True)

    # Quelle est la contribution de couche_cachee à l'erreur de couche_sortie
    # (suivant les pondérations)?
    error_couche_cachee = delta_couche_sortie.dot(synapse1.T)

    # Quelle est la "direction" de couche_cachee (dérivée) ?
    # Si OK, ne pas trop changer la valeur.
    delta_couche_cachee = error_couche_cachee * sigmoide_et_sa_derivee(couche_cachee,deriv=True)

    synapse1 += couche_cachee.T.dot(delta_couche_sortie)
    synapse0 += couche_entree.T.dot(delta_couche_cachee)

print ("Résultat de l'apprentissage :")
print (couche_sortie)
```

➛ Nous avons fusionné la fonction sigmoïde et sa dérivée à l'aide d'un paramètre formel à valeur par défaut (qui permettra de décider de la valeur à calculer).

• Traces d'exécution :

```
Moyenne Erreur couche sortie :0.496410031903
Moyenne Erreur couche sortie :0.00858452565325
Moyenne Erreur couche sortie :0.00578945986251
Moyenne Erreur couche sortie :0.00462917677677
Moyenne Erreur couche sortie :0.00395876528027
Moyenne Erreur couche sortie :0.00351012256786
Moyenne Erreur couche sortie :0.00318350238587
Moyenne Erreur couche sortie :0.00293230634228
Moyenne Erreur couche sortie :0.00273150641821
Moyenne Erreur couche sortie :0.00256631724004

Résultat de l'apprentissage :
[[ 0.00199094]
 [ 0.99751458]
 [ 0.99771098]
 [ 0.00294418]]
```

• L'évolution de l'erreur d'apprentissage est présenté par la figure suivante : elle diminue rapidement.
Habituellement, cette erreur est plus optimiste que l'erreur obtenue sur (un autre ensemble de) données de test mais dans le cas présent, nous n'aurons pas de données de test.



Les (nouvelles) variables du code python :

couche_cachee : La couche intermédiaire (cf. ci-dessus).
synapse0 : Première réseau de pondérations (Synapse 0) qui connecte `couche_entree` à `couche_cachee`.
synapse1 : Second réseau de pondérations (Synapse 1) qui connecte `couche_chachee` à `couche_sortie`.
erreur_couche_cachee : En pondérant `delta_couche_sortie` par les poids dans synapse1, nous pouvons calculer l'erreur de la couche cachée.
delta_couche_cachee : Il s'agit de l'erreur de la couche_cachee du RN ajustée par la confiance. Elle est quasi identique à erreur_couche_cachee sauf que chaque valeur de confiance est modifiée.

• On remarque bien que le code de ce second RN reprend le cas de l'exemple précédent en répétant le même traitement entre la couche des entrées et la couche cachée d'une part et entre la couche cachée et la couche des sorties de l'autre : la couche cachée fournit les entrées (davantage corrélées) de la couche des sorties.

• Notons que `error_couche_cachee = delta_couche_sortie.dot(synapse1.T)` utilise l'erreur pondérée par la confiance (pente) pour établir l'erreur de la couche cachée. Pour cela, il "transmet" les erreurs des pondérations `synapse1` (pondérations entre la couche cachée et la couche des sorties). On appelle cela "l'erreur pondérée par la contribution" car on apprend combien chaque noeud de la couche cachée "contribue" à l'erreur dans la couche des sorties et l'opération s'effectue comme pour notre premier RN. Cette étape est également appelée "rétro propagation". La mise à jour de `synapse0` a lieu en tenant compte de la contribution de la couche cachée à l'erreur du réseau et en utilisant la même logique que pour `synapse1`.

☞ **Pourquoi la couche cachée contient 4 noeuds ?**
Le nombre de couches cachées ainsi que le nombre de noeuds de chaque couche cachée relève de l'architecture des RNs et s'obtient généralement par tâtonnement. Dans le cas présent, on pourrait même réduire ce nombre à 3.
Mais si la relation entre les entrées / sorties devait être $sortie = X_1 \wedge X_3$, il faudra revenir à 4 noeuds pour obtenir de bons résultats.

III-2 Courbe de l'erreur

• Le code ci-dessous donne une solution pour obtenir les observations de l'erreur pour ensuite tracer la courbe de l'évolution de celle-ci.

```
# Le début ne change pas, jusqu'aux traits ————————————————
......
err=[]
for j in range(nb_iterations):
    ......
    couche_sortie_error = y - couche_sortie

    #————————————————————————————————————
    # Enregistrement de l'erreur
    moy=np.mean(np.abs(couche_sortie_error))
    err.append(abs(moy))

    # Afficher quelques valeurs de l'erreur
    if j% (nb_iterations//100) == 0:     # 1000 traces de l'erreur (tous les 100, par ex.)
        print("Moyenne Erreur couche sortie :" + str(moy))

    # Pas de changement dans la suite de l'itération (reprendre les 3 dernières lignes ci-dessous)
    # ...

print ("Résultat de l'apprentissage :")
```

```
print (couche_sortie)

# ------ La partie courbe de l'erreur ------
import matplotlib.pyplot as plt
myvec=np.array([range(len(err)),err])
plt.plot(myvec[0,],myvec[1,]);plt.show()
```

IV Travail pour cette séance (suite)

• Avancer autant que vous pouvez sur ces exercices. La travail à rendre pour ce BE est indiqué plus loin.

1. **Tester votre réseau :** Mettre en place le code Python pour tester votre réseau en utilisant la / les matrices de pondération. Notez qu'à partir de ces matrices, on peut trouver le nombre de noeuds en entrée et en sortie.
   Avec le réseau entraîné, **tester les vecteurs ci-contre** en entrée en comparer vos résultats avec les sorties attendues. Établir un taux d'erreur d'apprentissage.

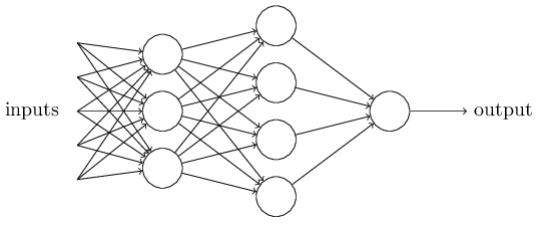
Entrées			Sorties
1	0	0	1
0	0	0	0
0	1	0	1

2. Modifier les entrées-sorties de cet exemple. Par exemple, considérer la table de vérité complète (8 lignes) de $X_1 \wedge X_2 \wedge \neg X_3$ avec les sorties correspondantes.
3. Lancer ce RN sur ces données ; donner les sorties et la courbe de l'erreur.
4. Modifier la fonction d'activation et refaire les tests. Essayer par exemple :
   ◦ la fonction Tangente Hyperbolique $f(x) = tanh(x) = \dfrac{2}{1+e^{-2x}} - 1$ dont la dérivée est $1 - f(x)^2$ et l'étendu (l'image) dans $[-1, 1]$.
   ☞ Remarquez bien que la dérivée f' contient sa primitive f (comme pour la sigmoïde).
   ◦ la fonction Gaussienne $f(x) = e^{-x^2}$ et dont la dérivée sera $f'(x) = -2xe^{-x^2} = -2xf(x)$ et l'étendu dans le demi $(0, 1]$. Remarquez bien que la dérivée f' contient f mais on a quand même besoin de x !
   ◦ la fonction Arc Tangente $f(x) = tan^{-1}(x)$ dont la dérivée est $\dfrac{1}{1+x^2}$ et l'étendu dans $[-\frac{\pi}{2}, \frac{\pi}{2}]$. [18]
   Donner la courbe de l'erreur pour chaque cas.
5. Ajouter une autre couche cachée et recommencer.
6. Comparer les résultats pour une et pour deux couches cachées. Donner les courbes d'erreur.
7. En conservant vos deux couches cachées, trouver l'exemple d'une expression moins triviale (par exemple, une expression utilisant l'implication logique, ou bien l'addition de deux nombres binaires chacun sur 4 bits qui donne un résultat et une retenue – donc 2 bits de sorties) ; modifier éventuellement le nombre des entrées.
   Entraîner le réseau ; donner les sorties et la courbe de l'erreur.
8. L'ajout d'une couche cachée supplémentaire modifie-t-il les résultats ? Expliquer.

Ajout de couches cachées

On peut créer, pour notre exemple, un RN à 2 couches cachées comme le montre la figure ci-contre.

➢ Nous aurons besoin des trois matrices de *synapse0*, *synapse1* et *synapse2*.



☞ Pour la prochaine séance, un complément sur l'optimisation des réseaux de neurones vous sera proposé suivi du travail final à rendre.
Il s'agira de créer un réseau de neurones pour l'entraînement et le test de reconnaissance de chiffres.

18. Sur le site https ://en.wikipedia.org/wiki/Activation_function, vous trouverez un tableau de fonctions d'activation.

V Optimisation

• Nous abordons ici la question de l'optimisation des RNs sous une forme simple. Voir la section VII page 24 pour en savoir plus.

• Le principe de **rétro propagation** nous a permis de mesurer combien les pondérations contribuent à l'erreur globale du RN. On a ensuite appliqué une **Descente de Gradient** pour modifier ces pondérations (en réinjectant les écarts, de couche en couche, dans le RN).

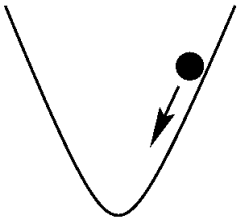
Cependant, le **rétro propagation** ne s'occupe pas d'optimiser les RNs ; il réinjecte une information (l'erreur pondérée) couche par couche depuis la fin du RN vers toutes les pondérations à l'intérieur du réseau. La tâche d'optimisation revient au réseau (caractérisé par ses couches, ses noeuds, ses pondérations, ...) lui-même. Différentes méthodes d'optimisation avancées sont abordées plus loin dans ce document.

• Dans la suite, on optimise notre RN en utilisant le support de la *Descente de Gradient* qui est la méthode la plus simple et la plus largement utilisée dans l'optimisation des RNs.

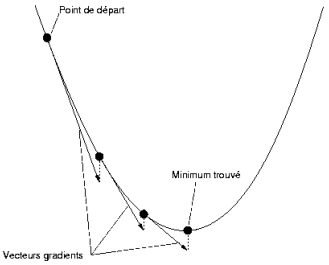
V-1 La Descente de Gradient

• Pour illustrer la stratégie d'optimisation par la **Descente de Gradient**[19], on peut utiliser l'exemple suivant.[20]

Une Descente de Gradient est comme le mouvement d'une bille que l'on met au bord intérieur d'un bol. La bille dont le but est d'atteindre le fond du bol ira sur les parois intérieures, de moins en moins haut pour enfin se stabiliser au fond.



La bille utilisera la pente de sa position (les positions successives) sur la paroi pour ajuster et tendre sa position vers le fond du bol. Notons que quand la pente est négative (en descendant de gauche à droite), la bille devrait aller à droite, et quand la pente est positive, la bille ira à gauche. Cette information est suffisante pour la bille pour atteindre le fond du bol après quelques itérations.



Dans une version simplifiée, on calcule la pente de la position courante ; si elle est négative alors on ira à droite sinon à gauche et ce jusqu'à atteindre le fond du bol dont la pente est nulle.

Une question se pose : de combien la bille devrait se déplacer à chaque étape (un "pas" de calcul) ? On remarque que plus la bille est loin du fond, plus la position est pentue (raide !).

• Pour exploiter cette nouvelle information dans notre RN, supposons une coupe 2D du bol dans le plan[21] des mouvements où la bille aura une position $< x, y >$. Dans ce cas, x augmente si on va à droite et diminue quand la bille va à gauche. Notons que plus la pente est raide, moins le x de la bille varie.[22]

Sous ces hypothèses, la démarche de Descente de Gradient naïve devient :

Répéter
  p = la pente à la position x actuelle, # La donnée y ne nous aide pas encore
  $x = x - p$
Jusqu'à $p \approx 0$ # ou Jusqu'à $p < \varepsilon$ avec $\varepsilon \approx 0$

➛ Par conséquent, pour une pente positive forte, on va à gauche (de beaucoup) tandis que pour une petite pente positive, on ira peu à gauche. Et plus la bille s'approchera du fond, plus elle fera de petits déplacements (de plus en plus petits) jusqu'à atteindre une pente nulle où elle s'arrêtera : ce point d'arrêt est appelé la **convergence**.

• Notons que la Descente de Gradient n'est pas parfaite et pose quelques problèmes (à notre RN aussi) passés en revue ci-après.

19. Voir par exemple http ://www.iro.umontreal.ca/∼bengioy/ift6266/old/neuron/neuron.html
20. Le terme Gradient est équivalent à "Pente" (ou "Escarpement").
21. On suppose que c'est bien un plan 2D
22. Cette variation de la position aura lieu à une vitesse supérieure à celle d'une pente moins rude mais la vitesse de la bille n'est pas prise en compte ici.

1- Une pente trop forte :
Chaque déplacement de la bille dépend de la raideur de la pente de sa position. Parfois, cette raideur est telle qu'elle provoque un grand déplacement [23] (alors qu'il en faudrait des petits).
Cette exagération du déplacement pourrait nous conduire à une pente encore plus raide dans la direction opposée qui nous conduira encore plus à nous éloigner de l'objectif ... C'est ce qui est appelé la **divergence**.

Pour palier ce problème et faire des petits déplacements, on devra réduire les grands déplacements en les multipliant par un réel dans 0..1 (p. ex. 0.1). Ce paramètre (appelé **alpha** ci-dessous) nous permet de mieux **converger**. [24]

On aura alors l'algorithme suivant :

alpha=0.1 (ou un autre réel $\in 0..1$)
$\varepsilon = 10^{-2}$
Répéter
  p = la pente à la position x actuelle,
  $x = x - p * alpha$
Jusqu'à $p < \varepsilon$ OU atteindre nbr_iterations_max

2- Pente faible :
Un autre problème de la Descente de Gradient dans les RNs survient lorsque la pente est faible. Dans la figure ci-dessous, la bille rouge victime de cette situation est coincée ! Cette situation se produit si le paramètre $alpha$ (vu ci-dessus) est très petit.

On aura alors vite fait de croire que le minimum est atteint d'autant que la faible pente ne donnera pas à la bille la "puissance" de se sortir de ce trou !

Un cas limite de l'inconvénient de ces petits $delta$ (directement lié au paramètre $alpha$) est donné dans la figure suivante où le temps du calcul et de convergence est trop grand.

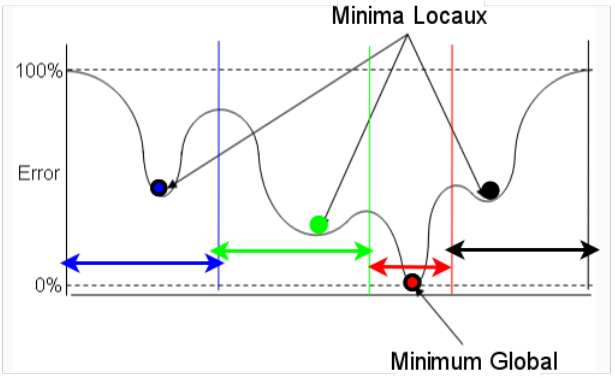
Pour palier ces deux problèmes, on peut augmenter la valeur de $alpha$. Une autre solution pratiquée dans certains RNs et de multiplier le $delta$ par une coefficient supérieur à 1.

3- Extrema locaux :
C'est de loin le problème majeur de toute optimisation, et quasi toutes les solutions à ce problème utilisent le principe d'une recherche aléatoire pour éviter de s'enfermer dans un extremum local.

Il se pose alors une question : si nous devons trouver un minimum (maximum) puis aléatoirement en chercher d'autres, pourquoi ne cherche-t-on pas aléatoirement ces points dès le départ ? Et l'extremum global ?

Pour répondre à cette question, considérons l'exemple suivant :

Si on place par exemple 100 billes sur cette courbe (ci-contre), puis on cherche les minima en optimisant leurs positions (pour trouver le minimum global), on finira par les 4 "trous" (cavités) dans la figure où chaque étendu ("$\Longleftrightarrow$") de couleur représente le domaine dans lequel le minimum a été trouvé. Par exemple, la cavité dans la zone bleu représente le minimum du domaine bleu. Ce qui veut dire que dans le cas d'une recherche aléatoire, nous devons trouver seulement 4 minima dans la totalité de cet espace.



23. Voire même on peut aller au delà du point de départ de la bille (et sortir du bol) !
24. Malgré sa simplicité, ce principe est utilisé dans un grand nombre de RNs.

Ce qui est bien plus optimal qu'une recherche totalement aléatoire sur tout cet espace où il y a potentiellement une infinité de positions (suivant la granularité de la discrétisation de tout ce domaine).

Transposition dans les RNS :
Dans les RNs, cette stratégie est mise en place par le choix de plusieurs couches cachées avec en particulier un grand nombre de noeuds chacune. Chaque noeud d'une couche cachée commencera par un état initial aléatoire différent. Ce qui permet aux noeuds cachés de converger vers un motif différent. En paramétrant notre RN par ces valeurs, on peut tester un très grand nombre de minima locaux dans un même RN.

De cette manière, les RNs seront plus efficaces et auront la capacité de chercher un espace plus grand. Dans la figure ci-dessus, la recherche dans tout l'espace nécessitera (en théorie) la prise en compte de seulement 5 billes dans les itérations. Une recherche purement aléatoire sur la totalité de cet espace prendrait un temps de calcul trop grand.

☞ Notons que la bonne mise en place de cette stratégie est un des points clés de la puissance des RNs.

On veut un grand nombre de noeuds cachés ? Soit. Mais on peut rétorquer que dans un RN, faire converger un nombre important de noeuds vers un même point (vers la même réponse) serait une perte de temps !

Pour palier cet inconvénient, les techniques telles que **"Dropout"** et **"DropConnect"** permettent d'éviter que les noeuds cachés calculent la même réponse.

DropConnect est une forme évoluée de *DropOut*.
Alors que DropOut cherche à éliminer des liaisons entre les noeuds cachées et la couche des sorties (évitant ainsi que des noeuds "similaires" / "redondants" influent trop sur les sorties), *DropConnect* généralise cette action et peut couper tous liens (synapses) jugés "peu prometteurs" / "redondants", en particulier entre les entrées et une couche cachée ou entre deux couches cachées, etc.
Ainsi, les noeuds cachés auront un nombre de connections différent et évitent de recalculer la même réponse.
Voir aussi plus loin sur ce point.

V-2 Illustration de la Descente de Gradient

Dans un RN, on cherche à minimiser l'erreur étant donné les pondérations. Si on s'intéresser à la courbe d'erreur du RN pour une seule pondération, on obtiendra une courbe semblable à la figure suivante. [25]

Et si nous calculons cette erreur pour chaque valeur de chaque pondération, on obtiendra une courbe dans un espace de dimension $\sum |synapse_i| + 1$ [26].

25. On prend "une seule pondération" ici pour voir la courbe en 2D avec x=valeurs de l'unique pondération, y=l'erreur du RN pour x.
26. $+1$ et pour l'axe de l'erreur, $|synapse|$ désigne le nombre de pondérations dans les synapses du RN

V-3 Un exemple

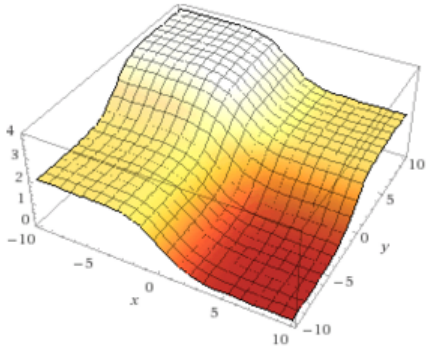
Rappel du code :

```
import numpy as np

# La fonction sigmoide
def sigmoide_et_sa_derivee(x,deriv=False):
    if(deriv==True): return x*(1-x)
    return 1/(1+np.exp(-x))

# Les entrées (seulement deux noeuds)
X = np.array([  [0,1],
                [0,1],
                [1,0],
                [1,0] ])

# Les sorties
y = np.array([[0,0,1,1]]).T

np.random.seed(1)

synapse0 = 2*np.random.random((2,1))   -1

couche_entree = X

for iter in range(10000):
    # propagation vers l'avant (forward)
    couche_sortie = sigmoide_et_sa_derivee(np.dot(couche_entree,synapse0))

    # Quelle erreur
    erreur_couche_sortie = couche_sortie - y        # pour "-", voir ci-dessous

     # Multiplier l'erreur (l'écart) par la pente du ïsigmode (paramètre "true") des valeurs de la
       couche_sortie
    delta_couche_sortie = erreur_couche_sortie * sigmoide_et_sa_derivee(couche_sortie,True)

    # La quantité pour la mise à jour des pondérations (on sépare pour mieux comprendre)
    derivee_synaps0 = np.dot(couche_entree.T, delta_couche_sortie)

    # Retropropagation : Mise à jour des pondération
    synapse0 -= derivee_synaps0      # "-" car "-" ci-dessus

print ("Les sorties après l'apprentissage :")
print (couche_sortie)
```

Pour ce RN (avec 2 pondérations), l'erreur calculée par la ligne

`erreur_couche_sortie = couche_sortie - y`

est un scalaire (puisqu'on a un seul noeud de sortie).
Noter dans la figure ci-contre (ainsi que dans le code) que nous avons deux pondérations (les axes x, y variant entre -10 et +10) et une (seule valeur d') erreur globale (axe z).
On remarque clairement dans cette figure que l'erreur est minimisée quand x (première pondération = 1e valeur de `synapse0`) est élevée.
De même, pour y (2nde valeur de `synapse0`), on a une décorrélation.



<u>L'optimisation de ce code :</u>
Les 3 lignes suivantes (du code) s'occupent de réduire l'erreur pondérée :

```
    # Multiplier l'erreur par la pente du ïsigmode (cf. "true") des valeurs de la couche_sortie
    delta_couche_sortie = erreur_couche_sortie * sigmoide_et_sa_derivee(couche_sortie,True)

    # La quantité pour la mise à jour des pondérations
    derivee_synaps0 = np.dot(couche_entree.T, delta_couche_sortie)

    # Retropropagation : Mise à jour des pondération
    synapse0 -= derivee_synaps0
```

C'est bien à cet endroit que la Descente de Gradient a lieu.
Nous allons reprendre le code du RN à 3 couches (section III-1, page 8) pour introduire un paramètre de contrôle et d'optimisation (`alpha`).

V-4 Optimisation de notre RN (à 3 couches)

Traitons les faiblesses constatées (cf. ci-dessus) de la Descente de Gradient.

Ajout du paramètre $alpha$:
Le paramètre $Alpha \in 0..1$ (appelé le*taux d'apprentissage*) permet de modérer / moduler les mises-à-jours de la position (pondérations $synapse0$ pour nous) dans chaque itération. Cette petite modification aura cependant un impact important sur le RN.

On reprend le réseau à 3 couches ci-dessus et on l'améliore suivant le pseudo algorithme rappelé ici :

alpha= un réel $\in 0..1$
Répéter
  p = la pente à la position x actuelle, # position = pondération
  $x = x - p * alpha$
Jusqu'à $p < \varepsilon$ ou *max_itérations* atteint.

Ce qui donne (vous ajouterez Jusqu'à $p < \varepsilon$ à l'itération en fixant ε p. ex. à 10^{-1} ou même 10^{-2}) :

```
# Le RN à 3 couches (une couche cachée) et on ajoute le param alpha
# On teste avec plusieurs alphas pour paufiner

import numpy as np

alphas = [0.001,0.01,0.1,1,10,100,1000]

# la fonction sigmoid (non linéaire)
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# On sépare pour simplifier le code :
# La fonction dérivée de la ïsigmode
def derivee_de_sigmoide(output):
    return output*(1-output)

X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

couche_entree = X
nb_iterations = 10000 #60000

# juste pour tracer de temps en temps : 10 trace par valeur de alpha testée
trace_tous_les_combien = nb_iterations //  10

for alpha in alphas:
    print("\nApprentissage Avec Alpha:" + str(alpha))
    np.random.seed(1)

    # Init des pondérations (avec mu=0)
    synapse_0 = 2*np.random.random((3,4)) - 1
    synapse_1 = 2*np.random.random((4,1)) - 1

    for j in range(nb_iterations) :

        # Propager à travers les couches
        couche_cachee = sigmoid(np.dot(couche_entree,synapse_0))
        couche_sortie = sigmoid(np.dot(couche_cachee,synapse_1))

        erreur_couche_sortie = couche_sortie - y

        if j% trace_tous_les_combien == 0: # Dix traces de l'erreur
            print("Moyenne abs(Erreur) après "+str(j)+" iterations : ", end='')
            print(str(np.mean(np.abs(erreur_couche_sortie))))

        # La dérivée pour connaitre la disrection de la valeur calculée (sortie)
        # Pondération par la dérivé de l'erreur
        delta_couche_sortie = erreur_couche_sortie*derivee_de_sigmoide(couche_sortie)

        # Quelle est la contribution de couche_cachee à l'erreur de couche_sortie
        # (suivant les pondérations)?
        erreur_couche_cachee = delta_couche_sortie.dot(synapse_1.T)

        # Quelle est la "direction" de couche_cachee (dérivée) ?
        # Si OK, ne pas trop changer la valeur.
        delta_couche_cachee = erreur_couche_cachee * derivee_de_sigmoide(couche_cachee)

        # Retropropagastion dans les pondérations
        synapse_1 -= alpha * (couche_cachee.T.dot(delta_couche_sortie))
        synapse_0 -= alpha * (couche_entree.T.dot(delta_couche_cachee))
```

On obtient la trace suivante pour différentes valeurs de $alpha$:

```
Apprentissage Avec Alpha:0.001
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Peu de changements ...
Moyenne abs(Erreur) après 5000 iterations : 0.495819152797
...
Moyenne abs(Erreur) après 9000 iterations : 0.495301195889
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.01
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Peu de changements ...
Moyenne abs(Erreur) après 5000 iterations : 0.48597903745
...
Moyenne abs(Erreur) après 9000 iterations : 0.465151532254
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.1
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Changements rapides ...
Moyenne abs(Erreur) après 5000 iterations : 0.0986588310456
...
Moyenne abs(Erreur) après 9000 iterations : 0.0475709483997
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:1
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Changements rapides ...
Moyenne abs(Erreur) après 5000 iterations : 0.0130184347753
....
Moyenne abs(Erreur) après 9000 iterations : 0.009129524184
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:10
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Changements rapides ...
Moyenne abs(Erreur) après 5000 iterations : 0.00459573231396
...
Moyenne abs(Erreur) après 9000 iterations : 0.00331623015848
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:100
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Changements rapides ...
Moyenne abs(Erreur) après 5000 iterations : 0.125699631251
...
Moyenne abs(Erreur) après 9000 iterations : 0.125504950847
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:1000
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Peu de changements ...
Moyenne abs(Erreur) après 5000 iterations : 0.5
...
Moyenne abs(Erreur) après 9000 iterations : 0.5
```

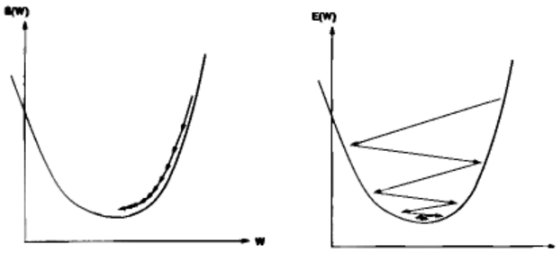
Commentaires sur les traces et sur $alpha$:

- $alpha = 0.001$: une petite valeur de $alpha$ empêche ce RN de converger. C'est bien le problème évoqué ci-dessus lorsque $alpha$ est trop petit.
- $alpha = 0.01$: meilleure convergence : il faudra itérer beaucoup plus pour y arriver ! Cela relève toujours du problème d'$alpha$ petit.
- $alpha = 0.1$: bonne progression puis moins bonne. Il faut augmenter la valeur d'$alpha$.
- $alpha = 1$: comme si on n'avait pas d'$alpha$! Convergence "normale".
- $alpha = 10$: on est surpris de voir que $alpha \notin 0..1$ donne la meilleure convergence dans ce RN et ce après 10000 itérations. Ceci indique que les valeurs précédentes de $alpha$ étaient trop conservatives. Notons que les valeurs plus petite d'$alpha$ souffraient d'une convergence trop lente mais dans la bonne direction.
- $alpha = 100$: cette fois, les quantités de mis-à-jours sont trop grandes et donc ce paramètre est moins intéressant , voire contre-productif. Ceci relève du problème d'$alpha$ trop grand et risque de nous faire manquer l'objectif en tournant autours sans vraiment atteindre le minimum.
- $alpha = 1000$ c'est un cas de divergence. On tourne autours de 0.5 mais cette erreur peut même augmenter au lieu de diminuer (si on augmente `nb_iterations`). Le RN risque fort de s'éloigner de tout minimum.

☞ Normalement, on a $alpha \in 0..1$. Nous avons utilisé ici des valeurs supérieures pour simplement montrer l'effet de ce paramètre.

V-5 A propos du taux d'apprentissage

• Les détails ci-dessus ont montré l'effet de `alpha`. Ce paramètre (dit le *Learning Rate*, parfois également nommé η) est appelé également le **taux d'apprentissage**. Il s'agit d'un paramètre qui contrôle les changements (en taille et en biais) de l'algorithme d'apprentissage.
• La figure exemple ci-contre montre graphiquement son influence sur la Descente de Gradient :
  ◦ A gauche : un petit taux d'apprentissage → descente douce ;
  ◦ A droite : un grand qui provoque de grandes fluctuations.
• Voir aussi les ressources bibliographiques à la fin de ce doc.



L'effet du taux d'apprentissage sur la Descente de Gradient.

V-6 Comportement du taux alpha dans notre RN

Pour mieux comprendre le comportement de notre code, on va compter le nombre de fois où la dérivée **change** de direction.
Si une pente (la dérivée) change de direction, cela veut dire qu' on est passé "par dessus" un minimum et il faut revenir sur ses pas. Par contre, s'il n'y a pas de changement de direction, sauf à avoir de la chance et trouver le minimum, cela veut dire que le RN ne va pas assez loin.

```
# Le comportement du paramètre ALPHA.

import numpy as np

alphas = [0.001,0.01,0.1,1,10,100,1000]

# la fonction sigmoid (non linéaire)
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# On sépare pour simplifier le code :
# La fonction dérivée de la ïsigmode
def derivee_de_sigmoide(output):
    return output*(1-output)

X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

couche_entree = X
nb_iterations = 10000 #60000

# juste pour tracer de temps en temps : 10 trace par valeur de alpha testée
trace_tous_les_combien = nb_iterations //  10

for alpha in alphas:
    print("\nApprentissage Avec Alpha:" + str(alpha))
    np.random.seed(1)

    # Init des pondérations (avec mu=0)
    synapse_0 = 2*np.random.random((3,4)) - 1
    synapse_1 = 2*np.random.random((4,1)) - 1

    # Quelques variables de plus pour les détails du comportement
    MAJ_prec_de_synapse_0 = np.zeros_like(synapse_0)
    MAJ_prec_de_synapse_1 = np.zeros_like(synapse_1)
    nbr_orientations_synapse_0 = np.zeros_like(synapse_0)
    nbr_orientations_synapse_1 = np.zeros_like(synapse_1)

    for j in range(nb_iterations) :

        # Propager à travers les couches
        couche_cachee = sigmoid(np.dot(couche_entree,synapse_0))
        couche_sortie = sigmoid(np.dot(couche_cachee,synapse_1))

        # Erreur ?   : voir  "synapse_1 -= alpha ..." ci-dessous
        erreur_couche_sortie = couche_sortie - y

        if j% trace_tous_les_combien == 0:      # Dix traces de l'erreur
            print("Moyenne abs(Erreur) après "+str(j)+" iterations : ", end='')
            print(str(np.mean(np.abs(erreur_couche_sortie))))

        # La dérivée pour connaitre la disrection de la valeur calculée (sortie)
        # Pondération par la dérivé de l'erreur
        delta_couche_sortie = erreur_couche_sortie*derivee_de_sigmoide(couche_sortie)

        # Quelle est la contribution de couche_cachee à l'erreur de couche_sortie
        # (suivant les pondérations)?
        erreur_couche_cachee = delta_couche_sortie.dot(synapse_1.T)

        # Quelle est la "direction" de couche_cachee (dérivée) ?
        # Si OK, ne pas trop changer la valeur.
        delta_couche_cachee = erreur_couche_cachee * derivee_de_sigmoide(couche_cachee)

        MAJ_ponderation_synapse_1 = (couche_cachee.T.dot(delta_couche_sortie))
        MAJ_ponderation_synapse_0 = (couche_entree.T.dot(delta_couche_cachee))

        if(j > 0):
            nbr_orientations_synapse_0 += np.abs(((MAJ_ponderation_synapse_0 > 0)+0) - ((
              MAJ_prec_de_synapse_0 > 0) + 0))
            nbr_orientations_synapse_1 += np.abs(((MAJ_ponderation_synapse_1 > 0)+0) - ((
              MAJ_prec_de_synapse_1 > 0) + 0))

        # Retropropagastion dans les pondérations
        synapse_1 -= alpha * MAJ_ponderation_synapse_1
        synapse_0 -= alpha * MAJ_ponderation_synapse_0
```

```
        MAJ_prec_de_synapse_0 = MAJ_ponderation_synapse_0
        MAJ_prec_de_synapse_1 = MAJ_ponderation_synapse_1

        print("Synapse 0")
        print(synapse_0)
        print("nbr Changements direction MAJ de Synapse 0")
        print(nbr_orientations_synapse_0)
        print("Synapse 1")
        print(synapse_1)
        print("Nbr Changements direction MAJ de Synapse 1")
        print(nbr_orientations_synapse_1)
```

Et les traces :

```
Apprentissage Avec Alpha:0.001
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
... Peu de changements ...
Moyenne abs(Erreur) après 9000 iterations : 0.495301195889

Synapse 0
[[-0.18127786  0.40239989 -1.04967234 -0.40779068]
 [-0.71250086 -0.85019434 -0.7394128  -0.31032393]
 [-0.2076949   0.03076367 -0.19401027  0.37187215]]
nbr Changements direction MAJ de Synapse 0
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 1.  0.  0.  1.]]

Synapse 1
[[-0.61150352]
 [ 0.72612385]
 [-1.00009649]
 [ 0.32976372]]
Nbr Changements direction MAJ de Synapse 1
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]]
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.01
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.495164116472
... Peu de changements ...
Moyenne abs(Erreur) après 8000 iterations : 0.471818340948
Moyenne abs(Erreur) après 9000 iterations : 0.465151532254

Synapse 0
[[-0.34640939  0.32849702 -2.12933133 -0.58643852]
 [-0.78274191 -1.27198739 -2.16355854 -0.37095516]
 [-0.3236015  -0.24414033 -0.09958826  0.43812544]]
nbr Changements direction MAJ de Synapse 0
[[ 0.  1.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 1.  0.  1.  1.]]

Synapse 1
[[-0.56926393]
 [ 0.9076231 ]
 [-2.22504943]
 [ 0.55654243]]
Nbr Changements direction MAJ de Synapse 1
[[ 1.]
 [ 1.]
 [ 0.]
 [ 1.]]
#-------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.1
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.457476916271
Moyenne abs(Erreur) après 2000 iterations : 0.359157691594
... Changements rapides ...
Moyenne abs(Erreur) après 7000 iterations : 0.0626168399603
Moyenne abs(Erreur) après 8000 iterations : 0.0538178929937
Moyenne abs(Erreur) après 9000 iterations : 0.0475709483997

Synapse 0
[[ 2.91833353  2.95995842 -5.60888064 -3.45030033]
 [-0.76418074 -4.93837955 -5.90984917 -2.20900589]
 [-0.02905967 -1.41092737  2.14222189  4.02709001]]
nbr Changements direction MAJ de Synapse 0
[[ 1.  1.  0.  0.]
 [ 2.  0.  0.  2.]
 [ 4.  2.  1.  1.]]

Synapse 1
[[-4.38219611]
 [ 5.11087619]
 [-8.27448223]
 [ 5.33871131]]
Nbr Changements direction MAJ de Synapse 1
[[ 2.]
 [ 1.]
 [ 0.]
 [ 1.]]
#-------------------------------------------------------------------------------
```

```
Apprentissage Avec Alpha:1
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.0429846478107
Moyenne abs(Erreur) après 2000 iterations : 0.0241219498915
Moyenne abs(Erreur) après 3000 iterations : 0.0181214006457
... Changements rapides ...
Moyenne abs(Erreur) après 7000 iterations : 0.0105975968339
Moyenne abs(Erreur) après 8000 iterations : 0.00978619256891
Moyenne abs(Erreur) après 9000 iterations : 0.009129524184

Synapse 0
[[ 4.05751199  3.77592492 -6.07774883 -3.91512755]
 [-1.92447764 -5.52099478 -6.39028245 -3.19429746]
 [ 0.57447316 -1.83960582  2.41541223  5.23881505]]
nbr Changements direction MAJ de Synapse 0
[[ 1.  1.  0.  0.]
 [ 2.  0.  0.  2.]
 [ 4.  2.  1.  1.]]

Synapse 1
[[-6.01822566]
 [ 6.39184307]
 [-9.63829175]
 [ 6.90678217]]
Nbr Changements direction MAJ de Synapse 1
[[ 2.]
 [ 1.]
 [ 0.]
 [ 1.]]
#-------------------------------------------------------------------------
Apprentissage Avec Alpha:10
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.0115169747275
Moyenne abs(Erreur) après 2000 iterations : 0.00771622343588
... Changements rapides ...
Moyenne abs(Erreur) après 7000 iterations : 0.00381050449482
Moyenne abs(Erreur) après 8000 iterations : 0.00353896690845
Moyenne abs(Erreur) après 9000 iterations : 0.00331623015848

Synapse 0
[[ 3.98975093  5.46454119 -7.1647967  -5.16195814]
 [ 1.35009415 -6.9836178  -7.60494603 -1.42574054]
 [-3.55614756 -3.03230946  3.2184627   4.31699971]]
nbr Changements direction MAJ de Synapse 0
[[ 7.  19.  2.  6.]
 [ 7.  2.  0.  22.]
 [ 19.  26.  9.  17.]]

Synapse 1
[[ -7.4642371 ]
 [  9.65036463]
 [-13.26783489]
 [  6.84287437]]
Nbr Changements direction MAJ de Synapse 1
[[ 22.]
 [ 15.]
 [  4.]
 [ 15.]]
#-------------------------------------------------------------------------
Apprentissage Avec Alpha:100
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.127314465457
Moyenne abs(Erreur) après 2000 iterations : 0.126232089149
... Changements rapides ...
Moyenne abs(Erreur) après 8000 iterations : 0.125538402384
Moyenne abs(Erreur) après 9000 iterations : 0.125504950847

Synapse 0
[[-15.19564444   1.86312936 -16.95588048  -8.23474344]
 [  5.65442566 -15.19644824  -9.45006075  -7.92964239]
 [ -4.23563688  -0.37236143   2.84990678  -8.40051445]]
nbr Changements direction MAJ de Synapse 0
[[  8.  7.  3.  2.]
 [ 13.  8.  2.  4.]
 [ 16.  13.  12.  8.]]

Synapse 1
[[  8.70684903]
 [  8.4966112 ]
 [-16.04007097]
 [  6.27328315]]
Nbr Changements direction MAJ de Synapse 1
[[ 13.]
 [ 11.]
 [ 11.]
 [ 10.]]
#-------------------------------------------------------------------------
Apprentissage Avec Alpha:1000
Moyenne abs(Erreur) après 0 iterations : 0.496410031903
Moyenne abs(Erreur) après 1000 iterations : 0.499999993732
... Peu de changements ...
Moyenne abs(Erreur) après 8000 iterations : 0.5
Moyenne abs(Erreur) après 9000 iterations : 0.5

Synapse 0
[[-56.06177241  -4.67153615  -5.65196177 -23.05868769]
```

```
 [ -4.52271708  -4.78838262 -10.887702   -15.85879101]
 [-89.56678495  10.51125561  37.02351519 -48.33299795]]
nbr Changements direction MAJ de Synapse 0
[[ 3.  2.  4.  1.]
 [ 1.  2.  2.  1.]
 [ 6.  6.  4.  1.]]

Synapse 1
[[ 25.16188889]
 [ -8.63371706]
 [-98.76334082]
 [ 11.41582458]]
Nbr Changements direction MAJ de Synapse 1
[[ 7.]
 [ 7.]
 [ 7.]
 [ 3.]]
```

Remarques sur les traces :
Pour un petit alpha, les pentes (dérivées) changent rarement de direction. Aussi, les pondérations terminent avec des valeurs assez petites.

☞ Pour la valeur optimal d'$alpha$ (10) dont le taux d'erreur est le plus petit[27], on a beaucoup de changements de direction.

Pour une grande valeur d'$alpha$, le nombre de ces changements est moyen. Aussi, les pondérations terminent avec des valeurs assez grandes.

V-7 Paramétrage de la taille de la couche cachée

Augmentons la taille de la couche cachée en passant de 4 à 32 (on augmente aussi le nombre d'itérations).

```python
import numpy as np

alphas = [0.001,0.01,0.1,1,10,100,1000]
nb_noeuds_couche_cachee = 32

# la fonction sigmoid (non linéaire)
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# On sépare pour simplifier le code :  La fonction dérivée de la ïsigmode
def derivee_de_sigmoide(output):
    return output*(1-output)

X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

couche_entree = X
nb_iterations = 20000 #60000

# juste pour tracer de temps en temps : 10 trace par valeur de alpha testée
trace_tous_les_combien = nb_iterations //  10

for alpha in alphas:
    print("\nApprentissage Avec Alpha:" + str(alpha))
    np.random.seed(1)

    # Init des pondérations (avec mu=0)
    synapse_0 = 2*np.random.random((3,nb_noeuds_couche_cachee)) - 1
    synapse_1 = 2*np.random.random((nb_noeuds_couche_cachee,1)) - 1

    for j in range(nb_iterations) :

        # Propager à travers les couches
        couche_cachee = sigmoid(np.dot(couche_entree,synapse_0))
        couche_sortie = sigmoid(np.dot(couche_cachee,synapse_1))

        # Erreur ?
        erreur_couche_sortie = couche_sortie - y

        if j% trace_tous_les_combien == 0: # Dix traces de l'erreur
            print("Moyenne abs(Erreur) après "+str(j)+" iterations : ", end='')
            print(str(np.mean(np.abs(erreur_couche_sortie))))

        # La dérivée pour connaitre la disrection de la valeur calculée (sortie)
        # Pondération par la dérivé de l'erreur
        delta_couche_sortie = erreur_couche_sortie*derivee_de_sigmoide(couche_sortie)
```

27. Cela dépend bien entendu de l'expression de l'erreur : Quadratique comme dans notre cas, MSE, CEE, etc...(voir section IX, P. 27)

```
        # Quelle est la contribution de couche_cachee à l'erreur de couche_sortie
        # (suivant les pondérations)?
        erreur_couche_cachee = delta_couche_sortie.dot(synapse_1.T)

        # Quelle est la "direction" de couche_cachee (dérivée) ?
        # Si OK, ne pas trop changer la valeur.
        delta_couche_cachee = erreur_couche_cachee * derivee_de_sigmoide(couche_cachee)

        # Retropropagastion dans les pondérations
        synapse_1 -= alpha * (couche_cachee.T.dot(delta_couche_sortie))
        synapse_0 -= alpha * (couche_entree.T.dot(delta_couche_cachee))
```

La trace :

```
Apprentissage Avec Alpha:0.001
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
... Pas trop de Changements ...
Moyenne abs(Erreur) après 18000 iterations : 0.486258756323
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.01
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.484976966103
... Changements plus rapides ...
Moyenne abs(Erreur) après 16000 iterations : 0.203007160289
Moyenne abs(Erreur) après 18000 iterations : 0.170843943756
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:0.1
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.146959409321
Moyenne abs(Erreur) après 4000 iterations : 0.065140892714
... Changements plus rapides ...
Moyenne abs(Erreur) après 16000 iterations : 0.022061941318
Moyenne abs(Erreur) après 18000 iterations : 0.0204137097708
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:1
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.0194758908815
Moyenne abs(Erreur) après 4000 iterations : 0.0126139696475
... Changements encore plus rapides ...
Moyenne abs(Erreur) après 14000 iterations : 0.00607706174945
Moyenne abs(Erreur) après 16000 iterations : 0.00563584926605
Moyenne abs(Erreur) après 18000 iterations : 0.00527480944082
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:10
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.00593811418383
Moyenne abs(Erreur) après 4000 iterations : 0.00380648647718
... Changements encore plus rapides ...
Moyenne abs(Erreur) après 14000 iterations : 0.00186376477278
Moyenne abs(Erreur) après 16000 iterations : 0.00173308967623
Moyenne abs(Erreur) après 18000 iterations : 0.0016259993165
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:100
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.5
... Pas trop de Changements ...
Moyenne abs(Erreur) après 16000 iterations : 0.5
Moyenne abs(Erreur) après 18000 iterations : 0.5
#--------------------------------------------------------------------------------
Apprentissage Avec Alpha:1000
Moyenne abs(Erreur) après 0 iterations : 0.496439922501
Moyenne abs(Erreur) après 2000 iterations : 0.5
... Pas trop de Changements ...
Moyenne abs(Erreur) après 16000 iterations : 0.5
Moyenne abs(Erreur) après 18000 iterations : 0.5
```

On constate bien (avec $alpha = 10$ optimal dans notre cas) que l'erreur décroît mieux (même à la hauteur de 10000 itérations comme précédemment) et ceci grâce au nombre de noeuds de la couche cachée.

La différence n'est pas énorme ; cependant, cette stratégie est très importante dans les RNs, en particulier avec des données en entrées plus complexes.

Bien que nous ayons juste besoin de (au moins 3 noeuds pour notre couche cachée, avec nos 32 noeuds, l'espace de recherche dans chaque itération a augmenté permettant de converger plus rapidement :
pour $alpha = 10$ optimal et pour le même nombre d'itérations (observer les même valeurs d'itérations), la version à 4 noeuds (couche cachée) affiche une erreur plus grande qu'avec 32 noeuds et la convergence est plus rapide dans ce dernier cas.

V-8 Momentum

• Un autre paramètre très utilisé dans les RNs est un réel $\in 0..1$ appelé *momentum* (désigne "la quantité du mouvement"). Bien choisi (par tâtonnement, comme pour alpha), il permet d'accélérer la convergence de l'erreur du RN.
Lors de la mise à jour des pondérations, ce paramètre ajoute simplement une (petite) fraction m de la pondération précédente à la nouvelle [28] ; ce qui donne pour notre RN (p.ex. pour *synapse2*, faire de même pour les autres) :

$$synapse2 += alpha * couche\_cachee2.T.dot(delta\_couche\_sortie) + momentum * synapse2$$

En général, si la valeur du *momentum* est proche de 1, la taux d'apprentissage devrait être bas (petit). Si ces deux paramètres sont tous les deux proches de 1, on aura des sauts importants dans les variations des erreurs.

• Dans la figure ci-contre, l'erreur emprunte soit différentes courbures dans des directions différentes, soit elle se bloque dans un minimum local.
Dans ces RNs (manifestement mal conditionnés), le gradient ne prend pas rapidement la direction d'un minimum (fig de gauche).

L'ajout d'un *momentum* (figure de droite) permet d'accélérer la convergence de l'erreur, si convergence il y a [29].

VI Travail final à rendre

• On suppose que le travail de la première séance est (presque) terminé.
• Prévoir un RN à 3 couches (cf. ci-dessus). Il comportera une couche d'entrée (I), une couche de sortie (O) et une couche de sortie (H) nécessitant 2 matrices/vecteurs de synapse.
Ce réseau aura donc un seule couche cachée.
Une fois la phase d'apprentissage terminée, Il doit être capable de tester des données (cf. travail de la 1e séance).

Apprentissage de chiffres manuscrits :
• On souhaite apprendre à reconnaître des **chiffres manuscrits** (comme à la PTT, pour lire le code postal).
Pour cela, on dispose d'une base de données (BD) d'apprentissage (soit le fichier fourni *train.data*) et d'une BD de test (soit *test.data*).
Dans ces BDs, un chiffre est représenté par un vecteur (à l'origine, une matrice 6×8 pixels aplati en vecteur) de 48 pixels composé de 0 (pixel éteint) et de 1 (pixel allumé).
La décision (y) est un chiffre $\in 0..9$

(I) Avec votre RN, procédez à l'apprentissage en respectant les points suivants :

- Les chiffres vont de 0 à 10. Réfléchir à la représentation de la sortie (voir ci-dessous).
  ☞ La fonction d'activation dépend de la sortie et vice versa. Pour une sortie $\in \{0, 1\}$, la fonction sigmoide est intéressante. Par contre, pour une sortie (par exemple) $\in \{0, 9\}$, elle ne convient plus. Pour un domaine de l'image de la fonction d'activation $\in \{0, 9\}$, on peut utiliser (p. ex.) $f(x) = ln(1 + e^x)$ dont la dérivée est la sigmoide $\frac{1}{1+e^{-x}}$. Voir la section *Quelques Ressources* : VIII en P. 26.
- En phase d'apprentissage, faire varier le nombre de noeuds de la couche cachée et la valeur du taux `alpha`, observer l'erreur et consigner la meilleure combinaison.
- Tester à chaque fois la BD de test.
- Conservez et consignez les meilleurs résultats.

(II) Passez à un réseau à 2 couches cachées et répéter les étapes ci-dessus.
➛ Cette fois, soyez attentifs au rapport entre la taille des 2 couches cachées. Pour une sortie sur 4 bits (voir le codage Gray plus loin), les tailles 128 (pour la couche 1) et 8 (pour la 2e couche) devraient donner de bons résultats (pour alpha=0.1).

(III) Remplacer la fonction d'activation et refaire les tests.

28. Donc on ajoute pour chaque synapse $synapse_i += m * synapse_i$. Noter que le *momentum* est souvent plus **efficace** s'il s'appuie davantage sur le passé (p.ex. les deux derniers synapses) pour remplir sa mission. On aurait alors, pour l'étape $t + 1$:
$synapse_{i,t+1} += m * (synapse_{i,t} - synapse_{i,t-1})$. Nous ne stockons pas les $synapse_{i,t-1}$ et utilisons ici que $synapse_{i,t+1} += m * synapse_{i,t}$.
29. Dans un espace de dimension supérieur à 2 (p.ex. 3), l'erreur aura tendance à aller de paroi en paroi le long d'une vallée représentée par les différentes courbure de l'erreur.

VI-1 A propos des tests

• Pour la BD chiffres, nous apprenons sur un fichier et testons sur un autre. Il existe également des techniques d'apprentissage comme X-V (*cross-validation*), LOO (*leave one out*), etc....
Par exemple, dans X-V, on pourrait regrouper ces deux fichiers puis de procéder comme suit :
◦ Brasser les données (voir la fonction *np.random.shuffle(matrice)*).
◦ Mettre de côté 10% des données aléatoirement et sans remise pour les tests ; apprendre sur 90% et tester sur les 10% ; noter les résultats (erreurs, taux de succès, etc).
◦ Recommencer (10% autre, 90% autre)
◦ Répéter cela 10 fois et prendre les moyennes des mesures relevées.
• Et si on répète tout cela 10 fois, on aura appliqué la méthode *10-folds-XV* !

• Dans LOO, on apprend sur toutes les données sauf une et on test sur cet élément choisi aléatoirement. On répète cela (idéalement autant de fois que le nombre de données, sinon au moins 100 fois) ; moyennes, etc.

VI-2 Lecture des fichier de données / test

• Vous pouvez lire le fichier "train.data" contenant sur chaque ligne 48 bits + la valeur cible $\in 0..9$ par :

```
import os
def lecture_matrice_X_et_Y_from_file(nom_fic) :
    """ On lit depuis un fichier la matrice des données (X et Y)"""

    os.chdir('/repertoire/ou/aiJe/mis/ceTruc')
    if not nom_fic in os.listdir() :
        print("pb d'ouverture du fichier" + nom_fic)
        matrice_X_et_Y=[]
        return

    # La matrice des données sous forme de caractères contenant des chiffres (des int)
    mat_tous_les_caracteres = open(nom_fic).read()

    # matrice des données (les chiffres sont sous forme de caractères) : séparer les lignes
    matrix_cars = [item.split() for item in mat_tous_les_caracteres.split('\n')[:-1]]
    matrice_X_et_Y=[[int(row[i])   for i in range(len(row))] for row in matrix_cars]
    return matrice_X_et_Y   #continet (pour BD chiffres) 48 bits et un chiffre 0..9
```

• Il faudra ensuite, et pour chaque ligne de `matrice_X_et_Y`, séparer la décision du reste.

```
import numpy as np
def preparer_donnees(nom_fic) :
    """ pour l'efficacité, travailler avec les matrices en données globales """
    global matrice_X
    global matrice_cible_y
    global L_Gray_repr_binaire_de_0_a_9
    global vecteur_chiffres_0_9_cible_y

    matrice_X_et_Y=lecture_matrice_X_et_Y_from_file(nom_fic)     # Vu ci-dessus

    if (matrice_X_et_Y==[]) :
        print("pb constitution de la matrice_X_et_Y")
        quit()  # ou exit()

    # Ici, matrice_X_et_Y est prete : on enlève la décion (un chiffre 0..9 à la fin de la ligne)
    matrice_X = np.array([row[:-1] for row in matrice_X_et_Y])
    # Et on transforme la décision 0..9 en 4 bits (code Gray)
    matrice_cible_y =  np.array([code_Gray_repr_binaire_de_0_a_9(row[-1] for row in matrice_X_et_Y])
```

Si on décide que la décision (chiffres 0..9) doit être représentée par des 0/1 sur 4 bits, on utilise le codage **Gray** (à expliquer). On peut alors continuer à utiliser la fonction sigmoide.
La fonction `code_Gray_repr_binaire_de_0_a_9(chiffre_0_9)` donne une représentation binaire sur 4 bits de chaque chiffre. Cette représentation binaire de *Gray* pour un chiffre *ch* s'obtient par : `gray = ch ^ (ch >> 1)`

• Pour les données du test (*test.data*), faire de même : lire le fichier, séparer, convertir,...

VII Addendum : Autres méthodes d'optimisation

• Pour la tâche d'optimisation des RNs, on dispose d'une variété de **méthodes d'optimisation non linéaires** que l'on peut utiliser avec la *rétro propagation*.

• Parmi ces méthode, on note :
- Recuit simulé (Annealing)
- Descente de Gradient stochastique (Stochastic Gradient Descent = SGD)
- AW-SGD (adaptative weighted SGD)
- Momentum (SGD)
- Nesterov Momentum (SGD)
- Adaptative Gradient (AdaGrad)
- AdaDelta (une extension de AdaGrad)
- ADAM (Adaptative Momentum)
- BFGS (Broyden-Fletcher-Goldfarb-Shanno)
- LBFGS (Limited Memory BFGS)

• Plusieurs de ces méthodes ont d'autres utilisations et dans certains cas, on peut en combiner plusieurs ensemble.

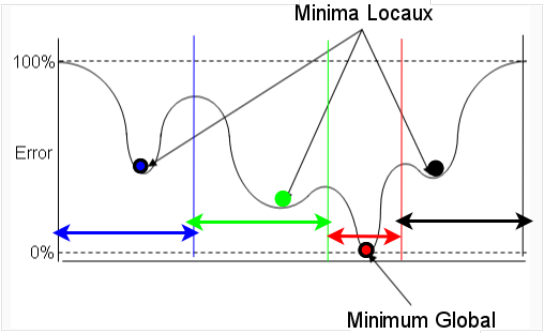
VII-1 A propos de DropOut

Technique très importante dans les RNs.
On reprend notre RN à 3 couches de ci-dessus en y ajoutant le *DropOut* (et le paramètre $alpha$), voir la couleur bleu .

```
# Mai 2016
# Avec DropOut
# On reprend le résueau "bis1 avec alpha" (à 3 couches) et on applique le Dropout
#-------------------------------------
# cmnt de la version bis1 (avril 2016) :
# Mai 2016
# Le RN à 3 couches (une couche cachée) et on ajoute le param alpha
# On teste avec plusieurs alphas pour paufiner

import numpy as np

def sigmoide_et_sa_derivee(x,deriv=False):
    if(deriv==True):
        return x*(1-x)

    return 1/(1+np.exp(-x))

X = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

# Les params
alpha,taille_hidden_layer,pourcentage_DropOut,dropout_to_do = (0.5,4,0.2,True)

# On utilise seed pour rendre les calculs déterministes.
np.random.seed(1)

# Initialisation aléatoire des poids (avec une moyenne = 0 et écart-type=1 ?)
# Ici, on met la moyenne à zéro, le std ne change pas.
# L'écriture X = b*np.random.random((3,4)) - 1
# permet un tirage dans [a,b), ici entre b=1 et a=-1 (donc moyenne=0)
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1

couche_entree = X
nb_iterations = 100000
for j in range(nb_iterations):

    # propagation vers l'avant (forward)
    # couche_entree = X déjà fait ci-dessus
    couche_cachee = sigmoide_et_sa_derivee(np.dot(couche_entree,syn0))

    # La partie DropOut
    if (dropout_to_do) :
        couche_cachee *= np.random.binomial([np.ones((len(couche_entree),taille_hidden_layer))],
                                            1-pourcentage_DropOut)[0] *(1.0/(1-pourcentage_DropOut))

    couche_sortie = sigmoide_et_sa_derivee(np.dot(couche_cachee,syn1))

    # erreur ?
```

```
    erreur_couche_sortie = y - couche_sortie

    if (j% (nb_iterations//10) == 0: # Dix traces de l'erreur
        print("Moyenne Erreur couche sortie :" + str(np.mean(np.abs(erreur_couche_sortie))))

    # pondération par l'erreur (si pente douce, ne pas trop changer sinon, changer pondérations) (voir
     BE4_TC1_RN-16_17)

    delta_couche_sortie = erreur_couche_sortie*sigmoide_et_sa_derivee(couche_sortie,deriv=True)

    # Quelle est la contribution de couche_cachee à l'erreur de couche_sortie
    # (suivant les pondérations)?
    erreur_couche_cachee = delta_couche_sortie.dot(syn1.T)

    # Quelle est la "direction" de couche_cachee (dérivée) ?
    # Si OK, ne pas trop changer la valeur.
    delta_couche_cachee = erreur_couche_cachee * sigmoide_et_sa_derivee(couche_cachee,deriv=True)

    syn1 += couche_cachee.T.dot(delta_couche_sortie)
    syn0 += couche_entree.T.dot(delta_couche_cachee)

print ("Résultat de l'apprentissage :")
print (couche_sortie)
print("Les noeuds de la couche cachée :")
print(couche_cachee)
```

Dans un RN, chaque noeud cherche à établir une corrélation entre les entrées et les sorties.



Dans cette figure, les billes représentent les pondérations et la ligne noire l'erreur du RN pour chaque valeur de pondération (`synapse`). Les points les plus "bas" représentent les erreurs moindres, signifiant que ces pondérations ont trouvé les meilleures corrélations entre les entrées et les sorties. Le but est de trouver les billes situées les plus bas (en particulier la rouge). Les différentes couleurs désignent les domaines (discuté plus haut en section "Optimisation").

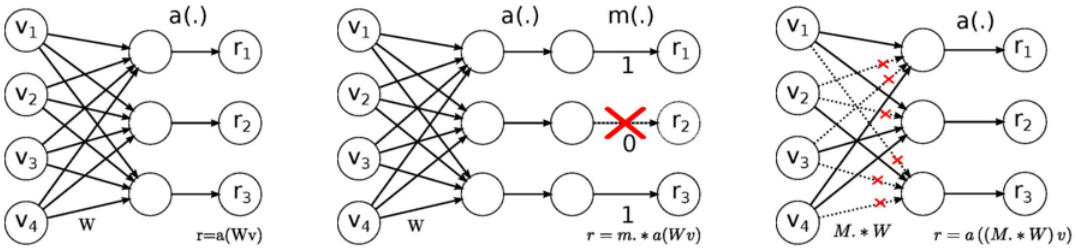
Initialement, les billes sont placées aléatoirement (comme les pondérations du RN). Si 2 billes commencent au même endroit, elles convergeront vers le même point. Ce qui les rend redondantes et gaspillent du temps de calculs (et des ressources). C'est le cas dans les RNs.

Le DropOut permet d'éviter que des pondérations convergent vers le même point. Pour cela, on "désactive" aléatoirement des noeuds (des couches) lors de la propagation en avant. **Mais ensuite, la retro-propagation les réutilise** tous.
Dans le code ci-dessus, pour une couche donnée, on fixe aléatoirement certaines valeurs à zéro pendant la propagation vers l'avant (la couleur bleue dans le code). Un booléen dans ce code permet d'activer ou non le DropOut. On peut par exemple utiliser le *DropOut* pendant l'apprentissage mais pas pendant les tests.

Dans les lignes de code signalées, on augmente la valeur propagée en avant. Ceci est fait en fonction du nombre de valeurs déactivées. L'idée est que si on désactive la moitié de la couche cachée, on pourrait "doubler" les valeurs restées actives pour compenser.

Dans la définition des paramètres, on utilise un pourcentage affectant la proba que chaque noeud sera désactivé. La bonne pratique met cette valeur à 50%. Si le DropOut est appliqué à la couche d'entrées, on n'excède en général pas 25%.

Une **bonne pratique** est de peaufiner le DropOut avec la taille de la couche cachée. On augmente le nombre des noeuds de la couche cachée tout en désactivant le DropOut (booléen à false) jusqu'à obtenir un RN performant puis , on utilise DropOu avec la couche cachée ainsi retenue. Cette combinaison donne en général une configuration optimale du RN. Penser à désactiver DropOut à la fin de l'apprentissage.



VIII Quelques ressources

- *Neural Networks : A Comprehensive Foundation* de Simon Haykin (1998, 2 ed.). Prentice Hall. ISBN 0-13-273350-1.
- http ://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
- http ://www.bogotobogo.com/python/python_Neural_Networks_Backpropagation_for_XOR_using_one_hidden_layer.ph
- Voir les librairies de RNs pour Python sur
https ://wiki.python.org/moin/PythonForArtificialIntelligence
- http ://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
- sebastianruder.com/optimizing-gradient-descent/
- Pour les fonctions d'activation, voir https ://en.wikipedia.org/wiki/Activation_function

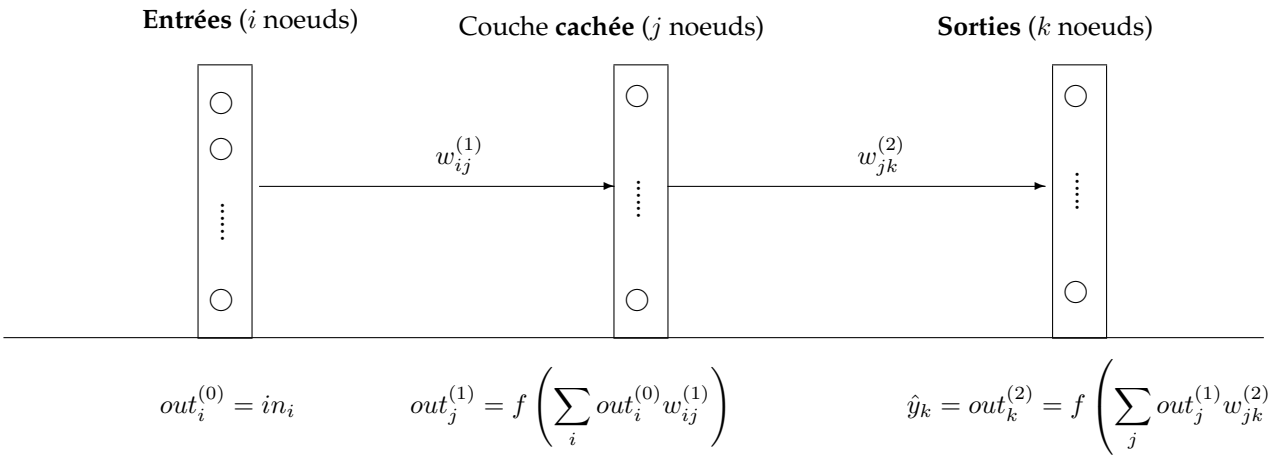
- Les librairies Python proposant des RNs : **Keras**, **skflow** pour ne citer que ces deux !

IX Justifications Mathématiques

1- Pourquoi calcule-t-on l'erreur par une soustraction dans le calcul de l'erreur ?
2- Autres expressions d'erreur (CEE).

IX-1 Détails de calculs de l'erreur SSE

Soit notre RN avec une couche cachée. Pour simplifier les notations, écrivons $synapse_{ij}$ par w_{ij}. Nous mettrons en exposant (et entre parenthèses) les numéros de couches ; quand ils apparaissent dans les notations [30] :



Pour notre RN, $out_j^{(2)}$ est la sortie calculée (pour la couche 2 = la couche de sortie et pour ses k noeuds). Dans les codes proposés, la fonction f est notre fonction d'activation *sigmoide*.

L'expression de l'erreur envisagée dans notre RN (comme dans la majorité des RNs) est l'erreur SSE (*sum square error* ou la moindre carrée) donnée ci-dessous.

Soit $\hat{y}$ la sortie (la dernière couche de k noeuds) calculée par le réseau et y la sortie attendue, sur l'ensemble des instances de la BD représenté par l'indice p ci-dessous.

L'erreur SSE due à la matrice de pondération qui précède la sortie :

$$E(w_{ij}^{(n)}) = \frac{1}{2}\sum_p \sum_{j=1}^{k} \left(y_j^p - \hat{y}_j^p\right)^2 = \frac{1}{2}\sum_p \left(y_p - \hat{y}_p\right)^2$$

➛ L'utilisation de la fraction $\frac{1}{2}$ est une pratique heuristique de calcul ; elle s'annulera avec la dérivée.

☞ L'erreur est d'abord calculée sur la sortie du réseau (dernière couche) dont la valeur dépend des couches précédentes. C'est pourquoi cette erreur est ensuite propagée en arrière (pondérée par les dérivées successives).

La *Descente de Gradient* mettra à jour la matrice des pondérations (précédant chaque couche du réseau) par (ici η est notre *learning rate*) :

$$\Delta_{w_{kl}}^{(n)} = -\eta \frac{\partial E(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

Les sorties calculées par notre réseau avec une couche cachée est :

$$\hat{y}_k^{(2)} = out_k^{(2)} = f\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f\left(\sum_j f\left(\sum_i in_i \,.\, w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

Si on applique la dérivée de l'erreur SSE en fonction des pondérations $w_{hl}^{(1)}$ et $w_{hl}^{(2)}$ (notre $synapse_0$ et $synaapse_1$) :

$$\frac{\partial E(w_{ij}^{(n)})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_k \left(y_k - \hat{y}_k^{(2)}\right).\frac{\partial \hat{y}_k^{(2)}}{\partial w_{hl}^{(m)}}$$

Où $$\frac{\partial \hat{y}_k^{(2)}}{\partial w_{hl}^{(m)}} = f'\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right).out_h^{(1)}.\delta_{kl} = f'\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right).f'\left(\sum_i in_i \,.\, w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

Ce qui nous donne une dérivée de l'erreur :

30. Dans les notations qui suivent, la toute première couche (des entrées) est la couche 0

$$\frac{\partial E(w_{ij}^{(n)})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_k \left(y_k - \hat{y}_k^{(2)}\right).f'\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right).f'\left(\sum_i in_i\,.\,w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

Nous avions ci-dessus l'expression de *delta* de l'erreur pour notre réseau :

$$\Delta_{w_{kl}}^{(n)} = -\eta\,\frac{\partial E(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

Ce qui est :

$$\Delta_{w_{hl}}^{(2)} = \eta\sum_p \left(y_l - \hat{y}_l^{(2)}\right).f'\left(\sum_j out_j^{(1)}.w_{jl}^{(2)}\right).out_h^{(1)}$$

➛ Dans notre RN, l correspond au nombre de noeuds de la couche de sortie et h celui de la couche cachée.

Et pour la couche précédente (la couche cachée = couche numéro 1) :

$$\Delta_{w_{hl}}^{(1)} = \eta\sum_p \sum_k \left(y_k - \hat{y}_k^{(2)}\right).f'\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right).f'\left(\sum_i in_i\,.\,w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

☞ On peut remarquer la différence (soustraction) que nous avons appelée *erreur_couche_sortie* dans le code Python trouve son origine dans ce calcul, due à l'erreur SSE et qui ne dépend pas de la fonction d'activation

Maintenant, si nous considérons notre fonction d'activation sigmoide [31], la quantité de mise-à-jour concernant notre $synapse_1$ sera (noter l'expression $f(x).(1-f(x))$) :

$$\Delta_{w_{hl}}^{(2)} = \eta\sum_p \left(y_l - \hat{y}_l^{(2)}\right).\hat{y}_l^{(2)}.\left(1-\hat{y}_l^{(2)}\right).out_h^{(1)}$$

Et pour la couche précédente, $synapse_0$ sera mis à jour par :

$$\Delta_{w_{hl}}^{(1)} = \eta\sum_p \sum_k \left(y_k - \hat{y}_k^{(2)}\right).\hat{y}_k^{(2)}.\left(1-\hat{y}_k^{(2)}\right).\ w_{lk}^{(2)}\ .\ out_l^{(1)}.\left(1-out_l^{(1)}\right).in_h$$

IX-2 Codage du RN

Lors du codage d'un RN avec la rétro-propagation, avec la fonction sigmoide, on obtient l'écart (appelé *delta_couche_sortie* dans le code Python) :

$$delta_k^{(2)} = \left(y_k - \hat{y}_k^{(2)}\right).f'\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = \left(y_k - \hat{y}_k^{(2)}\right).\hat{y}_k^{(2)}.\left(1-\hat{y}_k^{(2)}\right)$$

qui est la déviation généralisée de la sortie, on peut écrire la quantité de mis–jour des pondérations :

$$\Delta_{w_{hl}}^{(2)} = \eta\ \sum_p delta_l^{(2)}.out_h^{(1)}$$

Et pour la matrice des pondérations précédente :

$$\Delta_{w_{hl}}^{(1)} = \eta\ \sum_p \left(\sum_k delta_k^{(2)}.w_{lk}^{(2)}\right).out_l^{(1)}.(1-out_l^{(1)}).in_h$$

➛ où k est le nombre de noeuds de la couche sortie et p le nombre d'instances dans la BD.

☞ Ainsi, les pondérations $w_{hl}^{(2)}$ entre les couches h et l sont mises à jour proportionnellement aux noeuds de la couche h et l'écart ($delta$) de la couche l.
Les modifications des pondérations sur la première couche prend la même forme que sur la couche finale. Mais l'erreur de chaque couche l est rétro-propagée depuis toute couche de sortie k via les pondérations $w_{lk}^{(2)}$.

31. Rappel : Si $f(x)$ est la fonction sigmoide, $f'(x) = f(x).(1-f(x))$

IX-3 Généralisation de la rétro-propagation à N couches

On peut maintenant étendre la Descente Gradient pour un nombre quelconque de couches : si la dernière couche est la couche D (2 dans notre RN), alors (en bleu , si f est la sigmoide)

$$delta_k^{(D)} = (y_k - \hat{y}_k^D).f'\left(\sum_j out_j^{(D-1)}.w_{jk}^{(D)}\right) = \left(y_k - \hat{y}_k^{(D)}\right).\hat{y}_k^{(D)}.\left(1 - \hat{y}_k^{(D)}\right)$$

sera le delta (l'écart) de la couche de sortie (D (ernière) couche). On rétro-propage cet écart vers les couches antérieures par (en bleu , si f est la sigmoide) :

$$delta_k^{(n)} = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).f'\left(\sum_j out_j^{(n-1)}.w_{jk}^{(n)}\right) = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).out_k^{(n)}.(1 - out_k^{(n)})$$

Et la mise à jour généralisée des pondérations :

$$\Delta_{w_{hl}}^{(n)} = \eta \sum_p delta_l^{(n)}.out_h^{(n-1)}$$

IX-4 Autre mesure de l'erreur : CEE

Ci-dessus et dans le code Python, nous avons utilisé l'erreur SSE pour mesurer et corriger notre RN sur la base de la Descente de Gradient.
Il est cependant possible d'utiliser une autre mesure d'erreur très utile dans les RNs : l'entropie croisée ou *Cross Entropy Error* **CEE**.

$$E_{ce}(w_{ij}) = -\sum_p \sum_j \left[y_j^p\ .\ log(out_j) + (1 - y_j^p)\ .\ log(1 - out_j)\right]$$

☞ **Cette mesure permet d'interpréter les sorties du réseau comme des probabilités**. Elle a plusieurs avantages par rapport à SSE.
Quand on calculera les dérivées partielles pour les mises à jour via la Descente de Gradient, la dérivée de la **Sigmoide** s'annule et on obtient simplement (on a un RN avec **D** couches) :

$$\Delta_{w_{hl}}^{(D)} = \eta \sum_p \left(y_l - \hat{y}_l^{(D)}\right).out_h^{(D-1)}$$

qui est plus simple à calculer que la SSE. De plus, cette expression ne devient pas nulle lorsque les sorties sont totalement erronées [32].

---

32. Cet inconvénient de la SSE est traité par des ajustements ad-hoc que nous n'avons pas traitées dans notre RN.

X Annexes (séance 1)

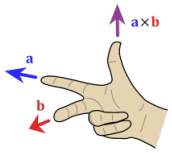
X-1 Produit Vectoriel et Scalaire

Addendum :

(I) ☞ Rappels sur le **produit vectoriel** (*cross product* ou *outer product*) par la fonction $\wedge$ (mais aussi par $\times$) :

• Un exemple (calcul en composantes) :

➛ Le produit vectoriel est défini par $u \times v = ||u||\,||v||\,sin\theta\,\mathbf{n}$ où $\mathbf{n}$ est un vecteur unitaire perpendiculaire au plan des deux vecteurs et dont la direction est donnée par la règle du pouce droit.



$$u \wedge v = \begin{pmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{pmatrix}.$$

• Sous Python (numpy), le nom de cet opérateur est *cross* (product) :

```
import numpy as np
x = [1, 2, 3]
y = [4, 5, 6]
np.cross(x, y)
# Donne : array([-3,  6, -3])
```

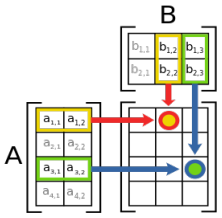
• Plus simplement, pour le vecteur $\overrightarrow{w}$ comme le produit vectoriel de $\overrightarrow{u}$ et $\overrightarrow{v}$, on a : $||\overrightarrow{w}|| = ||\overrightarrow{u}||||\overrightarrow{v}||\,|sin(\widehat{\overrightarrow{u},\overrightarrow{v}})|$

• Dans un certain sens, le *cross product* donne le pourcentage de la différence (*sin*) de deux vecteurs (les opérateurs trigonométriques mesurent un pourcentage).

(II) ☞ Dans notre code, nous avons utilisé '*' : simple produit élément par élément de 2 vecteurs :

```
import numpy as np
x = [1, 2, 3]
y = [4, 5, 6]
np.array(x)*np.array(y)
# Donne : array([ 4, 10, 18])
```

(III) ☞ Rappels sur le **produit scalaire** (*dot product* ou *inner product*) par la fonction **dot** :

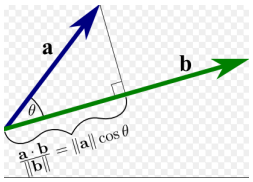


◦ Le scalaire défini par $u.v = \sum u_i v_i = ||u||\,||v||\,cos\theta$ pour deux vecteurs $\overrightarrow{u}$ et $\overrightarrow{v}$.

◦ Si u et v sont toutes deux des matrices, on aura une multiplication de matrices (ci-contre).

◦ Si seul l'un des deux est une matrice, il s'agira d'une multiplication de vecteur par matrice.

◦ Et plus généralement, pour a et b de dimension N, il s'agira de la somme de produits des derniers axes de a et du second au dernier axes de b.



◦ L'opération est caractérisée par : $dot(a,b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])$ où le résultat sera un scalaire si a et b sont des scalaires ou deux vecteurs ; un `array` sinon.

◦ Par exemple :

```
> np.dot(3, 4)
12

> a = [[1, 0], [0, 1]]
> b = [[4, 1], [2, 2]]
> np.dot(a, b)
array([[4, 1],
       [2, 2]])


# On génère les entiers [0,260) (car 3*4*5*6=260) transformés en une matrice de dimension 4 avec
  successivement 3,4,5 et 6 éléments.
> a = np.arange(3*4*5*6).reshape((3,4,5,6))

# La même chose mais considérer les entiers [0,260) à l'envers (notation [::-1])
> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
> np.dot(a, b)[2,3,2,1,2,2]
499128
> sum(a[2,3,2,:] * b[1,2,:,2])
499128

# Et enfin, un exemple avec des nombres complexes :
> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

• On dit que le *dot product* donne un pourcentage de la similarité (*cos* comme l'inverse de *sin*) de deux vecteurs.

X-2 Tirage aléatoires de nombres (numpy)

• Les fonction de tirage aléatoires de `numpy` :

$beta(a, b[, size])$	Samples from a Beta distribution.
$binomial(n, p[, size])$	Samples from a binomial distribution.
$bytes(length)$	Return random bytes.
$chisquare(df[, size])$	Samples from a chi-square distribution.
$choice(a[, size, replace, p])$	Generates a random sample from a given 1-D array ...
$dirichlet(alpha[, size])$	Samples from the Dirichlet distribution.
$exponential([scale, size])$	Samples from an exponential distribution.
$f(dfnum, dfden[, size])$	Samples from an F distribution.
$gamma(shape[, scale, size])$	Samples from a Gamma distribution.
$geometric(p[, size])$	Samples from the geometric distribution.
$get\_state()$	Return a tuple representing the internal state of the generator.
$gumbel([loc, scale, size])$	Samples from a Gumbel distribution.
$hypergeometric(ngood, nbad, nsample[, size])$	Samples from a Hypergeometric distribution.
$laplace([loc, scale, size])$	Samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
$logistic([loc, scale, size])$	Samples from a logistic distribution.
$lognormal([mean, sigma, size])$	Samples from a log-normal distribution.
$logseries(p[, size])$	Samples from a logarithmic series distribution.
$multinomial(n, pvals[, size])$	Samples from a multinomial distribution.
$multivariate\_normal(mean, cov[, size])$	Draw random samples from a multivariate normal distribution.
$negative\_binomial(n, p[, size])$	Samples from a negative binomial distribution.
$noncentral\_chisquare(df, nonc[, size])$	Samples from a noncentral chi-square distribution.
$noncentral\_f(dfnum, dfden, nonc[, size])$	Samples from the noncentral F distribution.
$normal([loc, scale, size])$	Draw random samples from a normal (Gaussian) distribution.
$pareto(a[, size])$	Samples from a Pareto II or Lomax distribution with specified shape.
$permutation(x)$	Randomly permute a sequence, or return a permuted range.
$poisson([lam, size])$	Samples from a Poisson distribution.
$power(a[, size])$	Samples in [0, 1] from a power distribution with positive exponent a - 1.
$rand(d0, d1, ..., dn)$	Random values in a given shape.
$randint(low[, high, size])$	Return random integers from low (inclusive) to high (exclusive).
$randn(d0, d1, ..., dn)$	Return a sample (or samples) from the "standard normal" distribution.
$random\_integers(low[, high, size])$	Return random integers between low and high, inclusive.
$random\_sample([size])$	Return random floats in the half-open interval [0.0, 1.0).
$rayleigh([scale, size])$	Samples from a Rayleigh distribution.
$seed([seed])$	Seed the generator.
$set\_state(state)$	Set the internal state of the generator from a tuple.
$shuffle(x)$	Modify a sequence in-place by shuffling its contents.
$standard\_cauchy([size])$	Samples from a standard Cauchy distribution with mode = 0.
$standard\_exponential([size])$	Samples from the standard exponential distribution.
$standard\_gamma(shape[, size])$	Samples from a standard Gamma distribution.
$standard\_normal([size])$	Samples from a standard Normal distribution (mean=0, stdev=1).
$standard\_t(df[, size])$	Samples from a standard Student's t distribution with df degrees of freedom.
$tomaxint([size])$	Random integers between 0 and sys.maxint, inclusive.
$triangular(left, mode, right[, size])$	Samples from the triangular distribution.
$uniform([low, high, size])$	Samples from a uniform distribution.
$vonmises(mu, kappa[, size])$	Samples from a von Mises distribution.
$wald(mean, scale[, size])$	Samples from a Wald, or inverse Gaussian, distribution.
$weibull(a[, size])$	Samples from a Weibull distribution.
$zipf(a[, size])$	Samples from a Zipf distribution.

X-3 Génération de nombres aléatoires numpy (bis)

Utility functions

$random\_sample$	Uniformly distributed floats over [0, 1).
$random$	Alias for random_sample.
$bytes$	Uniformly distributed random bytes.
$random\_integers$	Uniformly distributed integers in a given range.
$permutation$	Randomly permute a sequence / generate a random sequence.
$shuffle$	Randomly permute a sequence in place.
$seed$	Seed the random number generator.

Compatibility functions

$rand$	Uniformly distributed values.
$randn$	Normally distributed values.
$ranf$	Uniformly distributed floating point numbers.
$randint$	Uniformly distributed integers in a given range.

Univariate distributions

$beta$	Beta distribution over [0, 1].
$binomial$	Binomial distribution.
$chisquare$	χ^2 distribution.
$exponential$	Exponential distribution.
f	F (Fisher-Snedecor) distribution.
$gamma$	Gamma distribution.
$geometric$	Geometric distribution.
$gumbel$	Gumbel distribution.
$hypergeometric$	Hypergeometric distribution.
$laplace$	Laplace distribution.
$logistic$	Logistic distribution.
$lognormal$	Log-normal distribution.
$logseries$	Logarithmic series distribution.
$negative\_binomial$	Negative binomial distribution.
$noncentral\_chisquare$	Non-central chi-square distribution.
$noncentral\_f$	Non-central F distribution.
$normal$	Normal / Gaussian distribution.
$pareto$	Pareto distribution.
$poisson$	Poisson distribution.
$power$	Power distribution.
$rayleigh$	Rayleigh distribution.
$triangular$	Triangular distribution.
$uniform$	Uniform distribution.
$vonmises$	Von Mises circular distribution.
$wald$	Wald (inverse Gaussian) distribution.
$weibull$	Weibull distribution.
$zipf$	Zipf's distribution over ranked data.

Multivariate distributions

$dirichlet$	Multivariate generalization of Beta distribution.
$multinomial$	Multivariate generalization of the binomial distribution.
$multivariate\_normal$	Multivariate generalization of the normal distribution.

Standard distributions

$standard\_cauchy$	Standard Cauchy-Lorentz distribution.
$standard\_exponential$	Standard exponential distribution.
$standard\_gamma$	Standard Gamma distribution.
$standard\_normal$	Standard normal distribution.
$standard\_t$	Standard Student's t-distribution.

Internal functions

get_state	Get tuple representing internal state of generator.
set_state	Set state of generator.

Table des Matières

I. Introduction . . . 1
I-1. Un réseau de neurones (RN) simple . . . 2
I-2. Remarques sur le code Python . . . 3
I-3. Déroulement du code Python . . . 4
II. Travail-1 en séance . . . 6
III. Un exemple plus difficile . . . 7
III-1. Le code Python . . . 8
III-2. Courbe de l'erreur . . . 9
IV. Travail pour cette séance (suite) . . . 10
V. Optimisation . . . 11
V-1. La Descente de Gradient . . . 11
V-2. Illustration de la Descente de Gradient . . . 13
V-3. Un exemple . . . 14
V-4. Optimisation de notre RN (à 3 couches) . . . 15
V-5. A propos du taux d'apprentissage . . . 16
V-6. Comportement du taux alpha dans notre RN . . . 17
V-7. Paramétrage de la taille de la couche cachée . . . 20
V-8. Momentum . . . 22
VI. Travail final à rendre . . . 22
VI-1. A propos des tests . . . 23
VI-2. Lecture des fichier de données / test . . . 23
VII. Addendum : Autres méthodes d'optimisation . . . 24
VII-1. A propos de DropOut . . . 24
VIII. Quelques ressources . . . 26
IX. Justifications Mathématiques . . . 27
IX-1. Détails de calculs de l'erreur SSE . . . 27
IX-2. Codage du RN . . . 28
IX-3. Généralisation de la rétro-propagation à N couches . . . 29
IX-4. Autre mesure de l'erreur : CEE . . . 29
X. Annexes (séance 1) . . . 30
X-1. Produit Vectoriel et Scalaire . . . 30
X-2. Tirage aléatoires de nombres (numpy) . . . 31
X-3. Génération de nombres aléatoires numpy (bis) . . . 32