04/10/2021

Apprentissage et réseaux de neurones

Lilian BUZER

1. Introduction

Cette question est légitime et ce sera celle qui préoccupe tout élèves en début de ce cours : comment une machine peut-elle apprendre ? Pour vous, informaticiens, cette question est tout à fait pertinente car ni les boucles, ni les tests conditionnels ne semblent suffisants pour programmer une machine qui apprend. Pour arriver à cela, les réseaux de neurones requiert la maitrise de plusieurs sujets variés et complexes comme : la programmation, l'algèbre linéaire, l'optimisation, la modélisation ou encore l'algorithmique…

Le réseau : une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$

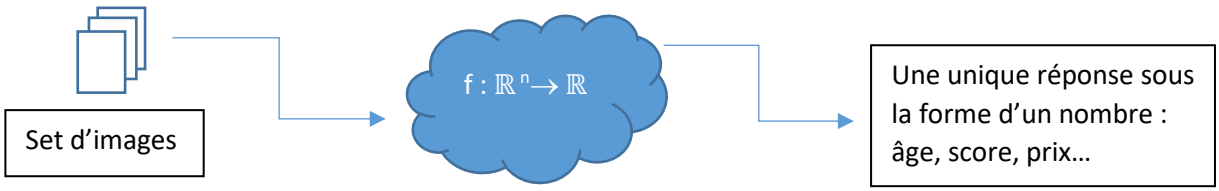
Avant d'apprendre, il faut déjà que le « programme » parle ! Ainsi, il faut que l'on construise un programme qui à partir de données source puisse fournir une réponse : un âge, un score, une catégorie… Les données en entrée, monde informatique oblige, seront numériques :

- Une image se représente sous la forme d'une matrice de valeurs entre 0 et 255.
- Un son se modélise comme une suite de valeurs issues de l'enregistrement depuis un micro.
- L'état de Pacman se traduit par sa position x,y, le score de partie, la position des fantômes…

Ainsi en entrée, nous avons plusieurs valeurs réelles et en sortie nous aurons une unique valeur. A ce niveau, vous devez reconnaître la définition d'une fonction mathématique f de $\mathbb{R}^n \rightarrow \mathbb{R}$ et notre système d'apprentissage sera modélisé par une fonction. Ceci a un impact direct sur le fonctionnement du système, car cela implique que toutes les entrées doivent avoir la même taille :

- Pour un set d'images, il faut donc retailler et redimensionner les images pour qu'elles aient toutes la même résolution et la même profondeur comme par exemple : 32x32x3
- Pour un son, il faut faire en sorte que tous les échantillons face la même durée avec la même vitesse d'échantillonnage.
- Pour des phrases, il faut que toutes ces phrases contiennent le même nombre de mots quitte à ajouter des mots « vides » pour compléter la fin de phrase.

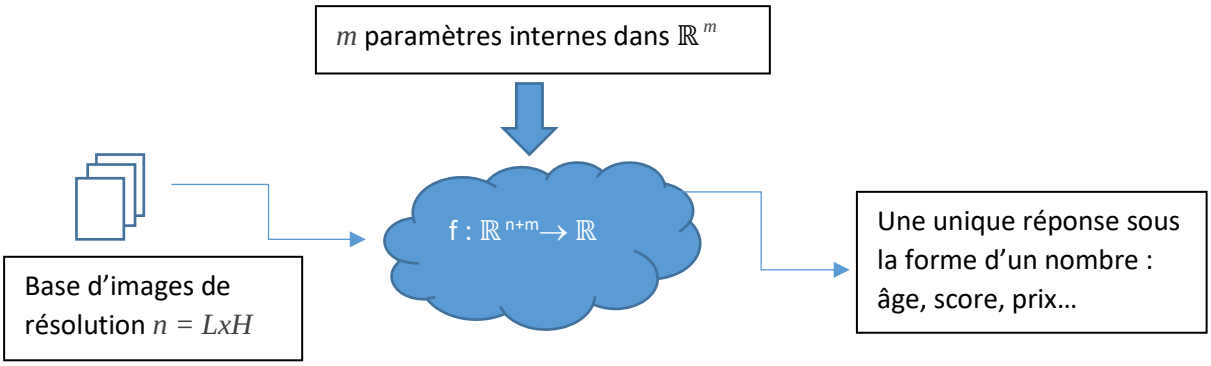
Voici donc notre premier système :



Nous avons construit un système qui prend des données en entrée et fournit une réponse en sortie. Cependant, vous l'avez remarqué, le système en question est incapable d'évoluer et d'apprendre.

Les paramètres d'apprentissage

Dans la modélisation précédente, la fonction f est fixe. Il faut faire en sorte que cette fonction puisse évoluer, sinon aucun apprentissage ne sera possible. Pour cela, nous allons ajouter des paramètres supplémentaires, dit paramètres internes d'apprentissage, qui vont être utilisés par la fonction pour calculer sa réponse :



Ces paramètres vont permettre l'apprentissage : en effet, si les réponses fournies par la fonction f sont incorrectes, on peut faire évoluer ces paramètres internes pour tenter d'obtenir une autre réponse, éventuellement meilleure. Durant la phase d'apprentissage, les images en entrée sont des données « constantes » au sens où elles ne sont pas modifiées durant l'apprentissage. La fonction f est généralement mise en place au tout début de l'expérimentation. Elle sera conservée, telle quelle, durant toute la phase d'apprentissage. **Les seuls paramètres pouvant évoluer durant la phase d'apprentissage sont les paramètres internes**.

Prenons un exemple, purement académique ! Nous allons considérer que chaque image possède un unique pixel de valeur x. Nous allons prendre pour fonction un polynôme du second degré pour construire $f(x,a,b,c) = ax^2+bx+c$. Ainsi les paramètres internes sont : a, b et c.

Pour $a = 1$, $b = 3$ et $c = 5$, la réponse pour l'image $x = 2$ est égale à : $f(x) = 2^2+3x2+5 = 15$.

Pour $a = 3$, $b = 1$ et $c = 3$, la réponse pour l'image $x = 2$ est égale à : $f(x) = 3.2^2+1x2+3 = 17$.

Différentes valeurs pour les paramètres internes fournissent des réponses différentes pour une même donnée en entrée.

Remarque : comment initialiser correctement les paramètres internes ? Impossible de répondre à cette question. La plupart du temps, ils seront initialisés aléatoirement en tirant des valeurs au hasard comprises entre 0 et 1.

Les paramètres d'apprentissage sont aussi appelés **poids du réseau** ou **weights** en anglais. Ils seront généralement notés w dans les énoncés.

L'erreur

Il nous manque encore une notion essentielle pour que l'apprentissage puisse apparaître : il faut que l'on soit capable d'évaluer la qualité de la réponse. Pour cela, nous mettons en place une approche utilisant des vérités terrain (des échantillons modèles). Ainsi pour chaque échantillon en entrée, on doit connaître la réponse associée :

- Pour l'évaluation de l'âge, nous traiterons une base de photos de personnes et pour chaque photo nous aurons l'âge exact de la personne.
- Pour la classification : on dispose d'un set d'images et pour chacune d'entre elles, on va donner la catégorie associée : chien, chat, grenouille…
- Pour la description automatique d'images : pour chaque image, on dispose d'une phrase décrivant le contenu de l'image.
- Pour la transcription automatique (Speech to Text), nous disposons de plusieurs phrases enregistrées ainsi que leurs transcriptions.

On parle ici d'**apprentissage supervisé**. L'apprentissage non supervisé est une thématique de recherche consistant à catégoriser, par exemple, des photos d'animaux sans connaissance à priori des espèces présentes sur les images.

Une fois que nous avons des échantillons et les réponses attendues, nous pouvons évaluer la qualité du réseau. Pour cela, nous mettons en place une **fonction Erreur** indiquant si la réponse de la fonction f est éloignée de la réponse attendue. Par convention, la fonction Erreur doit être positive et sa valeur doit augmenter plus la réponse s'éloigne de la valeur recherchée.

Prenons comme exemple le problème de la détection de l'âge d'une personne. Nous avons en entrée une image IMG et l'âge age_{ref} de la personne présente sur l'image. Nous calculons la réponse age_{estim} fournie par la fonction f correspondant à l'âge estimé. Nous pouvons proposer comme fonction d'erreur la fonction suivante :

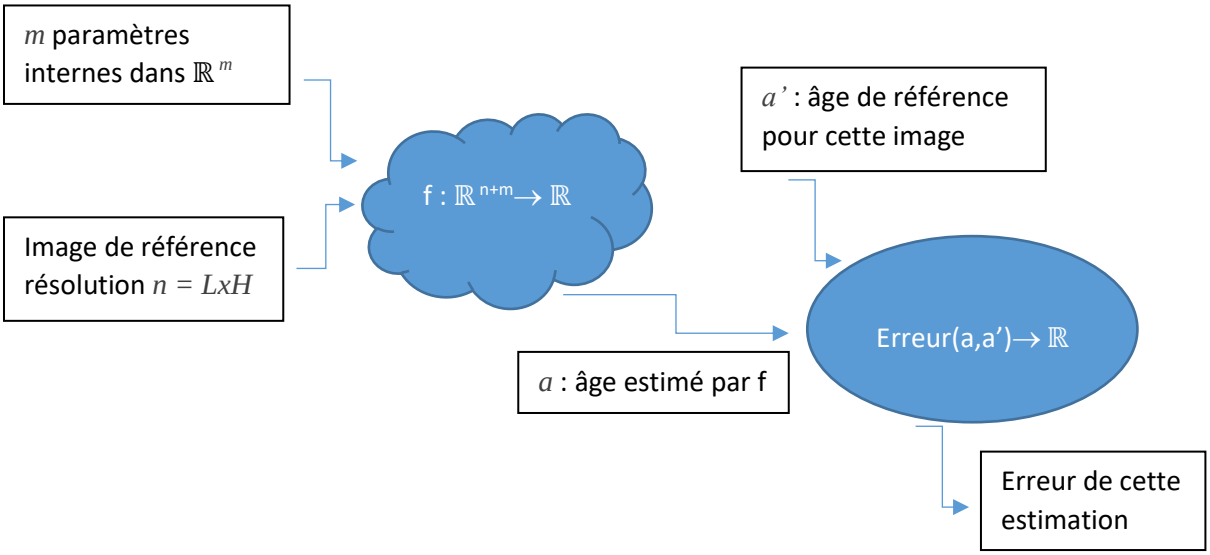
$$Erreur(age_{ref}, age_{estim}) = abs(age_{ref} - age_{estim})$$

Voici différentes réponses possibles pour une même image où la personne a 37 ans :

- Age estimé : 37, Erreur = |37-37| = 0 — correct, car on a trouvé la bonne réponse
- Age estimé : 39, Erreur = |37-39| = 2 — faible erreur, 2 ans trop vieux
- Age estimé : 30, Erreur = |37-30| = 7 — l'erreur augmente car l'écart augmente
- Age estimé : 87, Erreur = |37-87| = 40 — 40 ans trop vieux, l'erreur est forte

Remarque : dans cet exemple, la valeur de l'erreur est facile à comprendre car elle correspond à un écart en nombre d'années. Mais rarement la valeur de l'erreur aura une unité, nous ne serons donc pas en mesure de savoir si 0.1 ou 150 correspond à une erreur importante ou non. Cependant, il faut garder à l'esprit que si la valeur de la fonction d'erreur augmente cela suggère que la qualité de la réponse baisse.

Intégrons maintenant la fonction Erreur dans notre flux de calcul :



Durant la phase d'apprentissage, nous nous concentrons sur la valeur de l'erreur et moins sur la réponse estimée. Ainsi, la fonction f va momentanément être mises de côté pour étudier la nouvelle fonction err définie ainsi :

$$Err(I,w,ref_I) = Erreur(\ f(I,w)\ ,\ ref_I\ )$$

Où I correspond à une donnée en entrée (une image par exemple), w les paramètres d'apprentissage et ref_I la réponse exacte pour I la donnée en entrée.

L'apprentissage supervisé, pour être efficace, doit disposer de plusieurs milliers de données en entrée. Plus la taille de l'échantillon de référence est importante, plus vous augmentez la qualité de l'apprentissage. On peut citer quelques bases annotées utilisées régulièrement dans la formation et la recherche :

- MNIST : base d'images correspondant aux chiffres manuscrits de 0 à 9 ; utilisée à l'origine pour construire les machines automatisées de tri postal. 60 000 images, résolution de 28x28 en niveaux de gris : http://yann.lecun.com/exdb/mnist/
- CIFAR10 : base d'images pour la classification d'images. 60 000 images, résolution de 32x32 en couleur, 10 classes : avion, voiture, oiseau, chat, daim, chien, grenouille, cheval, navire et camion. https://www.cs.toronto.edu/~kriz/cifar.html
- CIFAR100 : 100 classes cette fois et 600 images par classe.
- ImageNet : base d'images qui sert de référence pour le problème de classification : 1000 classes, plus d'un million d'images en couleur de taille variable.
- COCO : base d'images pour la segmentation et la description d'images : 80 000 images accompagnées de leurs descriptions.

Ainsi, pour estimer l'erreur totale sur l'ensemble de la base d'apprentissage, il nous suffit de sommer les erreurs induites par chaque entrée :

$$ErrTot(f, w, DataSet) = \sum_{I, ref_I \in DataSet} Err(I, w, ref_I)$$

Comment apprendre ?

Si nous examinons la fonction *ErrTot*, elle prend en entée une fonction f, des paramètres internes w et une base d'apprentissage. Sur ces trois entrées, deux sont fixes : la base et la fonction f car la base ainsi que la fonction ne changent pas durant la phase d'apprentissage. Les seules entrées susceptibles d'évoluer sont les paramètres d'apprentissage w. Par conséquent, lorsque nous avons choisi une fonction f et la base à traiter, le problème de l'apprentissage se ramène à cette formulation :

$$\min_{w} ErrTot(w,f,Base)$$

Vous devez ici reconnaître un problème d'optimisation : recherche du minimum d'une fonction. Si vous n'êtes pas familier avec ce domaine, voici une méthode d'optimisation naïve pour présenter le principe de fonctionnement :

- Des valeurs de départ sont données pour les paramètres internes w
- On évalue l'erreur totale courante : `E = ErrTot(w,f,Base)`
- Les paramètres internes sont décrits comme un vecteur dont chaque élément s'écrit $w[i]$
- Nous générons un nouveau vecteur w' dont les valeurs sont proches des valeurs de w.
    - Pour cela, on choisit un $pas = 0.01$
    - On utilise la fonction rand(-1,1) retournant une valeur dans l'intervalle [-1,1]
    - On construit w' de cette façon : `w'[i] = w[i] + pas * rand(-1,1)`
    - L'écart maximum entre chaque valeur de $w[i]$ et $w'[i]$ est inférieure à la valeur du pas
- On évalue l'erreur totale associée aux paramètres w' : `E' = ErrTot(w',f,Base)`
- Si `E'` est plus faible que E, alors les paramètres sont de meilleure qualité et :
    - `w'` $\rightarrow$ `w`
    - `E'` $\rightarrow$ `E`
- On recommence ce processus à l'infini

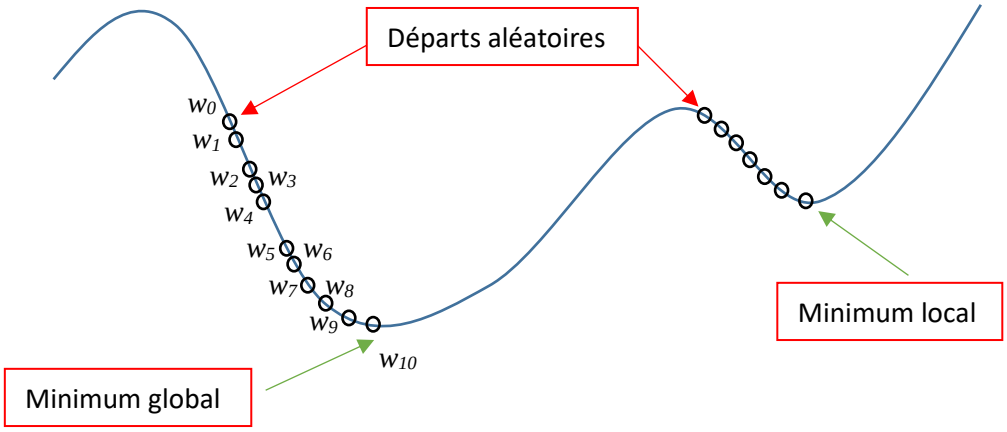
Voici le code associé :

```
Function OptimNaif(ErrTot,w,f,Base) :

    E = ErrTot(w,f,Base)
    Tant qu'il me reste du temps :
        # w' dans le voisinage de w
        pour i allant de 0 à len(w) :
            w'[i] = w[i] + pas * rand(-1,1)
        E'= ErrTot(w,f,Base)
        Si E' < E :
            w'→ w
            E'→ E
```

Ainsi, successivement, nous testons de nouveaux paramètres dans le voisinage des valeurs des paramètres actuels. Si les réponses sont de meilleure qualité, nous conservons ces nouveaux paramètres sinon nous les rejetons. Itérativement, nous diminuons donc l'erreur (lorsque nous faisons un bon tirage). L'apprentissage fonctionne !

Néanmoins, ni cette méthode, ni aucune autre méthode d'optimisation peut nous garantir de converger vers l'erreur optimale. En effet, il existe le problème des minimas locaux qui piègeront tout algorithme de minimisation. Voici un exemple :



Dans la partie gauche du graphique, les paramètres internes générés aléatoirement positionne $ErrTot(w_0)$ dans une pente qui descend vers le minimum global. En utilisant notre algorithme naïf, nous descendons le long de cette pente chaque fois que de meilleurs paramètres internes sont trouvés. Au final, nous convergeons vers le minimum.

En commençons à droite, nous minimisons, de la même manière, l'erreur totale à chaque changement de paramètres internes. Cependant, en suivant cette pente, nous atteignons cette fois un minimum local qui agit comme un aimant. Il n'y aura pas de moyen évident pour sortir de ce piège. Bien souvent, il faudra relancer le problème à partir d'une autre position.

Apprentissage vs Production

L'apprentissage comportent deux phases distinctes :

- La phase d'apprentissage consistant, à partir d'une base de vérités terrains fixées, à faire évoluer les paramètres internes afin de minimiser l'erreur totale.
- Une phase de production, où les paramètres internes sont maintenant figés : le système se comporte comme un expert auquel on présente de nouvelles informations pour connaître sa réponse.

Exercice 1 :

Que doit-on exporter comme données une fois que le système a fini la phase d'apprentissage ? Ces données doivent permettre au système de fonctionner pendant la phase de production. Entourez les bonnes réponses :

Les images contenues dans la base

La fonction/réseau f

La fonction d'erreur

Les réponses contenues dans la base

La méthode d'optimisation

Les paramètres internes

Remarque : vous pouvez à tout moment reprendre une phase d'apprentissage, même plusieurs jours après l'arrêt de la phase initiale. Pour cela, il vous suffit de relancer l'algorithme d'optimisation avec la fonction et les derniers paramètres internes choisis pour essayer d'obtenir de nouveaux meilleurs paramètres. On peut par ailleurs changer ou étendre la base de référence pour fournir plus d'échantillons.

Exercice 2 :

Attention, cet exercice est loin d'être facile et doit nécessiter plusieurs minutes de concentration pour fournir une réponse correcte.

Nous cherchons à apprendre le test booléen $T(x,y)$ suivant : 1 si $x \neq y$ et 0 si $x = y$

Pour cela, nous allons mettre en place nos vérités terrains, simple échantillonnage de cette fonction :

Vérités terrains :

x	y	Résultat
0	5	1
4	4	0
10	6	1

Construisez une fonction f servant de fonction d'apprentissage :

- Cette fonction doit avoir un seul paramètre interne a permettant l'apprentissage.
- Utilisant les fonctions mathématiques suivantes :
  - Tan, Exp, Log
  - Max, Min, Abs
  - +, /, -, x

Donnez une fonction d'erreur pouvant servir au problème d'apprentissage.

Intuitez une valeur a associé à une erreur totale nulle.

Cet apprentissage est-il optimal ? La fonction construite peut-elle être identique à la fonction T ?

Que peut-on faire pour améliorer la similarité entre la fonction construite et la fonction T.

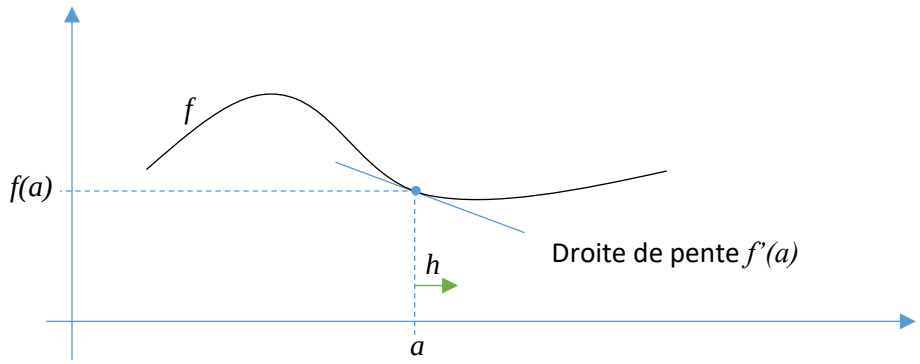
2. La méthode du gradient

Pour une fonction de $\mathbb{R} \rightarrow \mathbb{R}$

Nous cherchons à minimiser la fonction erreur totale de notre problème. Supposons qu'il s'agisse d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$. Dans ce cas particulier, le gradient correspond à la dérivée. Nous rappelons ici la formule d'approximation d'une fonction :

$$f(a+h) = f(a) + h.f'(a) + \varepsilon(h) \quad \text{avec } \varepsilon(.) \text{ tend vers 0 lorsque } h \text{ tend vers 0}$$

Ce qui signifie qu'autour de la valeur a, pour une valeur de h « petite », la fonction f se comporte comme une droite de pente $f'(a)$:



On en déduit ainsi facilement la direction dans laquelle la recherche du minimum doit avoir lieu :

- Si $f'(a)$ est positive, alors la fonction croit, il faut donc chercher le minimum à gauche de a
- Si $f'(a)$ est négative, la fonction décroit, il faut donc chercher le minimum à droite de a

En conclusion, il faut se déplacer dans la **direction opposée** au signe de $f'(a)$. Ainsi, nous pouvons mettre en place l'algorithme de descente du gradient suivant :

```
Function GradientDescent(f,a) :
   pas = 0.01
   tant que du temps est disponible :
      a = a – pas * f'(a)  # le signe – → direction opposée
```

Remarque : la méthode de descente du gradient ne garantit pas que la valeur de la fonction diminue à l'itération suivante. Par exemple, lorsque nous sommes proches d'un optimum, on peut « sauter » par-dessus pour atteindre finalement une valeur plus importante. En pratique, lorsque les valeurs atteintes à chaque itération se mettent à osciller, on suppose qu'un optimum est proche et que le pas du gradient est trop important pour continuer la convergence.

Exercice :

Supposons que l'erreur totale s'écrive sous la forme : $Err(x,a) = a^2 + a.x + 1$
Avec :

- x : donnée en entrée (constante)
- a : paramètre d'apprentissage.

Valeur à l'itération courante :	$x = 3$ et $a_0 = 5$
Combien vaut $Err(a_0)$?	$Err(a_0) = 5^2+3.5+1 = 41$
Exprimez $Err'(a)$?	$Err'(a) = 2a+x$
Combien vaut $Err'(a_0)$?	$Err'(5) = 2\times 5+3 = 13$
Devons-nous prendre pour prochaine valeur de a une valeur inférieure ou supérieure à la valeur courante ?	La dérivée est positive, on doit chercher à gauche de a
Prenons une valeur de $pas = 1/13$, donnez la nouvelle valeur de $a : a_1$?	$a_1 = a_0 - pas = 5 - 1 = 4$
Calculez $Err(a_1)$?	$Err(a_1) = 4^2+3.4+1 = 29$
Que pouvons-nous conclure ?	Cette itération a permis de diminuer l'erreur totale

Pour une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$

Nous devons ici utiliser les dérivées partielles qui étendent la notion de dérivée d'une fonction de $\mathbb{R} \rightarrow \mathbb{R}$ à une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$. Prenons l'exemple d'une fonction $f(x,y) \rightarrow \mathbb{R}$:

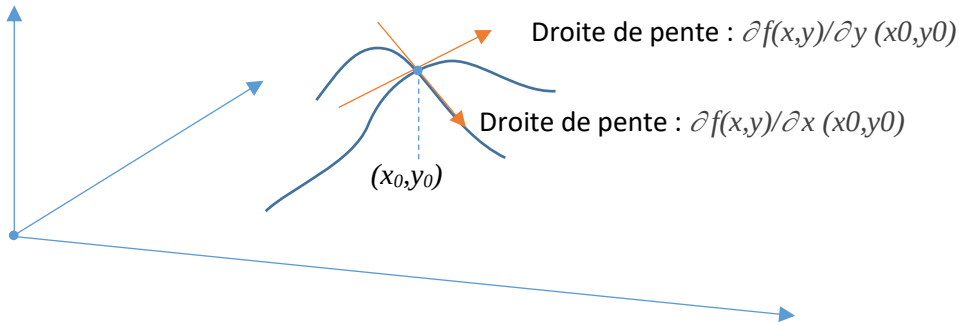
On étudie cette fonction autour d'une valeur (x,y) si nous faisons varier légèrement les valeurs de x et y, nous obtenons les mêmes formules que dans le cas 1D :

$$f(x+h,y) = f(x,y) + h.\partial f(x,y)/\partial x + \varepsilon(h)$$

et

$$f(x,y+h) = f(x,y) + h.\partial f(x,y)/\partial y + \varepsilon(h)$$

Comment s'interprètent les deux dérivées partielles ? Si nous faisons uniquement évoluer la valeur de x nous obtenons comme dans le cas 1D une droite tangente au point courant $f(x_0,y_0)$. De la même manière, si nous faisons évoluer y, nous obtenons une autre droite tangente. En combinant ces « deux directions tangentes », nous obtenons un plan tangent qui approche le comportement de la fonction :



Pour raccourcir les formules, nous utilisons la notations gradient : ∇f qui correspond à un vecteur contenant l'ensemble des dérivées partielles de la fonction au point courant. Ainsi, nous pouvons écrire :

$$f(x+h_0, y+h_1) \sim f(x,y) + (h_0, h_1).\nabla f(x,y)$$

⚠ L'opérateur . entre (h_0,h_1) et $\nabla f(x,y)$ désigne un produit scalaire entre 2 vecteurs.

La conclusion est identique au cas 1D : pour minimiser la valeur de f, il faut se déplacer dans le sens opposé au gradient. Ainsi, nous obtenons l'algorithme de descente du gradient dans un cas quelconque. A noter que ci-dessous x désigne un vecteur de $\mathbb{R}^n$:

```
Function GradientDescent(f,x) :
        pas = 0.01
        tant que du temps est disponible :
           x = x - pas * ∇f(x)   # le signe - → direction opposée
```

Remarque : le gradient est un vecteur de taille égale aux nombres de paramètres internes. Ceci permet d'effectuer le calcul x - $pas*\nabla$ car pas est un réel, et ainsi x et ∇ doivent être deux vecteurs de même dimension.

Exercice :

Supposons que l'erreur totale s'écrive sous la forme : $Err(x,a,b) = a^2 + b.x + 1$
Avec :

- x : donnée en entrée (constante)
- a, b : paramètres d'apprentissage.

Valeur à l'itération courante :	$x = 6,\ a_0 = 3$ et $b_0 = 2$
Combien vaut $Err(a_0, b_0)$?	$Err(a_0, b_0) = 3^2 + 2 \times 6 + 1 = 21$
Exprimez $\nabla Err(a,b)$?	$\partial Err/\partial a = 2a$ $\partial Err/\partial b = x$ $\nabla Err(a,b) = (2a, x)$
Combien vaut $\nabla Err(a_0, b_0)$?	$\nabla Err(a_0, b_0) = (2 \times 3,6) = (6,6)$
Prenons une valeur de $pas = 1/6$, donnez les nouvelles valeurs a_1, b_1 ?	$(a_1, b_1) = (a_0, b_0) - pas. \nabla Err(a_0, b_0)$ $= (3,2) - (6,6)/6$ $= (2,1)$
Calculez $Err(a_1, b_1)$?	$Err(a_1, b_1) = 2^2 + 1 \times 6 + 1 = 11$
Que pouvons-nous conclure ?	Cette itération a diminué l'erreur totale
Si nous avions un pas de 4, quels auraient été les valeurs de a_1, b_1 ?	$(a_1, b_1) = (a_0, b_0) - pas. \nabla Err(a_0, b_0)$ $= (5,4) - 4 \times (6,6)$ $= (-19,-20)$
Calculez $Err(a_1, b_1)$?	$Err(a_1, b_1) = (-19)^2 - 20 \times 6 + 1 = 242$
Que pouvons-nous conclure ?	La valeur de pas est trop importante En conséquence, cette itération a fait augmenter l'erreur totale

Calcul du gradient automatisé

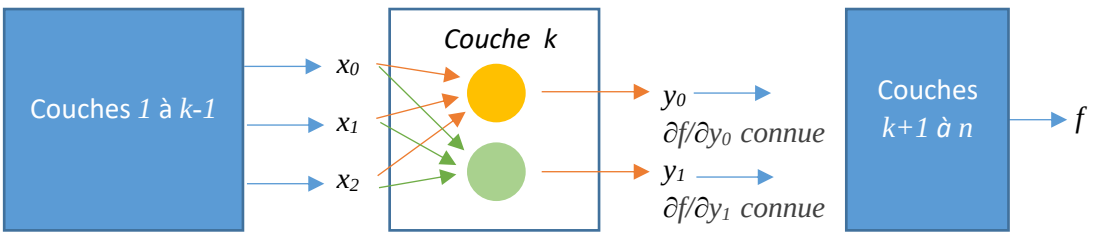
Les librairies de réseaux de neurones (Tensorflow, Pytorch) comme la librairie Numpy (avec Autograd) permettent de calculer le gradient d'une fonction de manière automatique. Pour cela, on utilise on système en trois phases :

- Dans la première phase appelée **Forward**, on calcule la valeur de la fonction. Ici les calculs sont effectués de manière progressive comme on les ferait à la main. Parallèlement, la librairie construit un graph de calcul qui représente le flot d'opérations effectuées pour arriver au résultat final. Dans ce graph sont stockés les résultats intermédiaires de chaque calcul.
- Dans la deuxième phase appelée **Backward** ou aussi **rétropropagation**, la librairie va parcourir le graph de calcul en sens inverse. Cette phase permet de construire toutes les dérivées partielles par rapport aux paramètres internes. Cette phase utilise les formules classiques de dérivation.
- Une troisième phase consiste à remettre à zéro les informations stockées pour le calcul du gradient lors de la rétropropagation. A ce niveau, les résultats intermédiaires stockés dans le graph doivent être mis à zéro pour éviter qu'il soit pris en compte lors de la prochaine itération.

Le calcul du gradient par rétropropagation

Les réseaux de neurones avec lesquels nous allons travailler sont décrits sous forme de couches (layer). Chaque couche prend les informations en entrée, effectue des calculs sur ces données et fournit alors des résultats pour la couche suivante. Le réseau complet constitue la fonction f et nous cherchons à déterminer les dérivées partielles de f par rapport à chaque paramètre interne. Une fois les dérivées partielles déterminés, nous avons donc le gradient nécessaire à l'algorithme de descente.

Durant la passe Backward, nous allons partir de la couche terminale (à droite) et reculer à chaque itération pour déterminer les dérivées partielles de la couche précédente. Ainsi, à l'itération n-k, nous pouvons modéliser notre système ainsi :



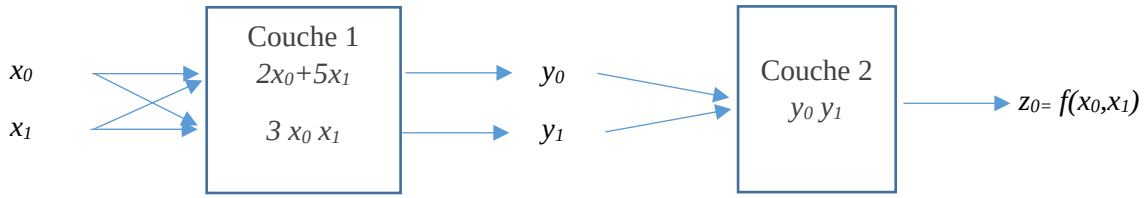
Les itérations précédentes ont permis de déterminer les valeurs de $\partial f/\partial y_0$ et $\partial f/\partial y_1$ et maintenant nous cherchons à déterminer les valeurs de $\partial f/\partial x_0$, $\partial f/\partial x_1$ et $\partial f/\partial x_2$. Pour cela, nous utilisons le théorème général des dérivations qui nous indique que :

$$\frac{\partial f}{\partial x_i} = \sum_{j=0}^{m} \frac{\partial f}{\partial y_j} \times \frac{\partial y_j}{\partial x_i}$$

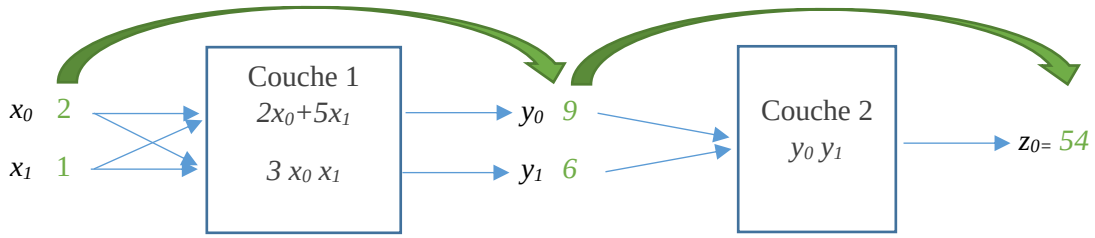
Les valeurs de $\partial f/\partial y_j$ ont été déterminées à l'itération précédente. Les valeurs de la forme $\partial y_j/\partial x_i$ correspondent aux dérivées partielles des fonctions utilisées dans la couche courante k. Ces fonctions sont connues : max, mul, exp, sin … et on peut trouver leurs dérivées en utilisant les formules de dérivations usuelles comme : $sin'(x_2)=cos(x_2)$. Comme les valeurs x_i sont connues depuis la passe Forward, il reste à évaluer l'expression : $cos(x_2)$ pour connaître les valeurs recherchées.

Exemple : $f(x_0, x_1) = (2x_0+5x_1)\times(3\ x_0\ x_1)$

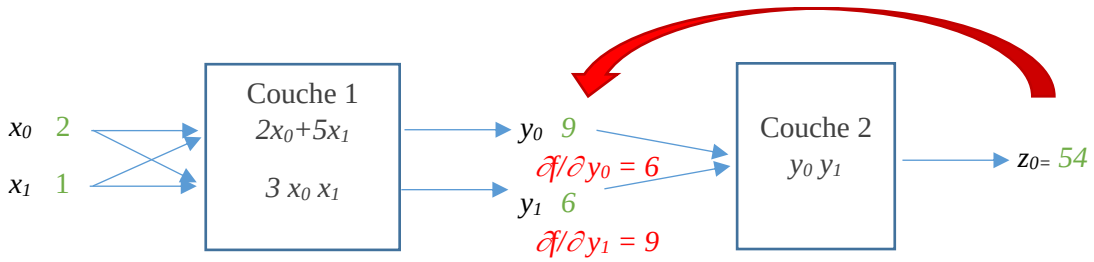
Voici le graph de calcul que nous pouvons associer à cette fonction :



Prenons $x_0=2$ et $x_1=1$. Nous allons tout d'abord effectuer la passe Forward :



La passe Forward est maintenant terminée. Nous commençons la rétropropagation en démarrant par la droite. La couche 2 effectue le calcul suivant $z_0(y_0y_1)= y_0\times y_1$. Les formules classiques de dérivation s'appliquent dans ce cas, ainsi, nous savons que $\partial z_0/\partial y_0=\partial(y_0\ y_1)/\partial y_0 = y_1$. Or la valeur y_1 est connue car elle a été déterminée durant la passe Forward : $y_1=6$. Nous connaissons donc la valeur de $\partial f/\partial y_0$. Le raisonnement est identique pour y_1 : $\partial f/\partial y_1 = \partial z_0/\partial y_1=\partial(y_0\ y_1)/\partial y_1 = y_0=9$.



Nous allons maintenant passer à l'étape suivante où nous allons rétropropager les valeurs des dérivées $\partial f/\partial y_0$ et $\partial f/\partial y_1$ pour trouver $\partial f/\partial x_0$ et $\partial f/\partial x_1$. Dans cette configuration plus complexe où $\partial f/\partial x_0$ dépend de $\partial f/\partial y_0$ et $\partial f/\partial y_1$, nous devons utiliser le théorème général :

$$\frac{\partial f}{\partial x_0} = \frac{\partial f}{\partial y_0} \times \frac{\partial y_0}{\partial x_0} + \frac{\partial f}{\partial y_1} \times \frac{\partial y_1}{\partial x_0}$$

Il reste à calculer :

- $\partial y_0/\partial x_0 = \partial(2x_0+5x_1)/\partial x_0 = 2$
- $\partial y_1/\partial x_0 = \partial(3\, x_0\, x_1)/\partial x_0 = 3\, x_1 = 3 \times 1 = 3$

Ainsi : $\partial f/\partial x_0 = 6 \times 2 + 9 \times 3 = 39$

Nous calculons $\partial f/\partial x_1$ de la même manière :

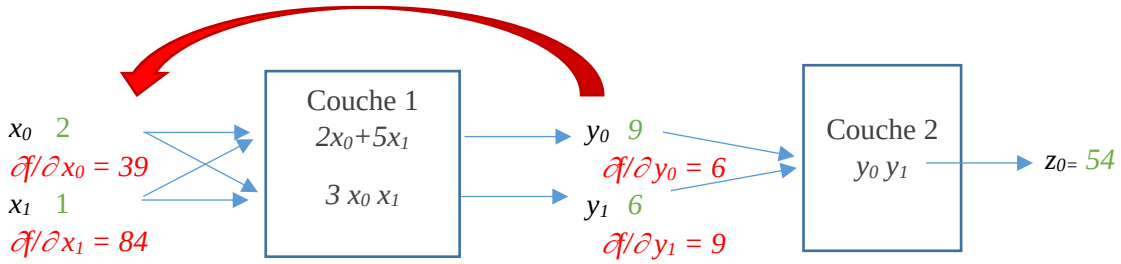
$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial y_0} \times \frac{\partial y_0}{\partial x_1} + \frac{\partial f}{\partial y_1} \times \frac{\partial y_1}{\partial x_1}$$

Il reste à calculer :

- $\partial y_0/\partial x_1 = \partial(2x_0+5x_1)/\partial x_1 = 5$
- $\partial y_1/\partial x_1 = \partial(3\, x_0\, x_1)/\partial x_1 = 3\, x_0 = 6$

Ainsi : $\partial f/\partial x_1 = 6 \times 5 + 9 \times 6 = 84$

Nous obtenons finalement :



Quel est l'intérêt de cette approche ? Elle est relativement simple à mettre en place, informatiquement parlant. Elle ne nécessite pas le calcul complexe des dérivées partielles très longues résultant de l'imbrication de plusieurs de couches. En effet, habituellement pour résoudre cette approche, nous aurions appliqué d'abord calculer littéralement les dérivées partielles de $f(x_0, x_1) = [(2x_0+5x_1) \times (3\, x_0\, x_1)]$ et uniquement à la fin, nous aurions remplacé les variables x_0, x_1 par leur valeurs respectives : 2 et 1. La méthode de la rétropropagation permet de faire les calculs de dérivation localement dans chaque bloc, ce qui est beaucoup plus simple.

D'ailleurs, nos résultats sont-ils justes ? Pour vérifier, plutôt que de dériver $f(x_0, x_1)$ nous allons plutôt utiliser la formule d'approximation pour vérifier notre calcul :

$$\partial f/\partial x_0 = [\, f(x_{0+}\varepsilon, x_1) - f(x_0, x_1)] / \varepsilon$$

$$\partial f/\partial x_1 = [\, f(x_0, x_{1+}\varepsilon) - f(x_0, x_1)] / \varepsilon \quad \text{avec } \varepsilon = 0.001$$

3. Les écueils

Bien que les réseaux de neurones aient permis de grandes avancés en quelques années, il faut rester humble, car les réseaux d'aujourd'hui sont les résultats des travaux de plusieurs milliers de chercheurs ayant effectués des milliers de tentatives avant d'arriver à une solution fonctionnelle. On l'oublie trop souvent. Voici les différents écueils que vous pouvez rencontrer lors de la mise en place d'une méthode d'apprentissage :

La non convergence : toutes phases d'apprentissage ne convergent pas et même celles sensées converger ! En effet, si vous prenez un réseau connu, avec sa base d'entraînement et la méthode de gradient utilisée pour son apprentissage et que vous relancez l'apprentissage de zéro, vous n'allez pas forcément obtenir la convergence. Cela semble étrange, mais il faut être conscient que les phases d'apprentissage dépendent énormément des valeurs d'initialisation des paramètres internes et que ces dernières sont générées aléatoirement au lancement. Cela pose problème car certains réseaux sont très difficiles à faire converger et il semble impossible de pouvoir retrouver des paramètres internes qui fonctionnent. Des entités de recherche comme Google ou Microsoft ont pu lancer des expériences en série sur plusieurs jours et sur plusieurs GPU en parallèle. Ainsi, sur 1000 ou 10 000 essais, seuls 1 ou 2 ont pu s'avérer satisfaisants. Avec les moyens que l'on dispose de notre côté, il est difficile de réentraîner de tel réseau from skratch.

Un mauvais score : si votre réseau converge (l'erreur a diminué au fur et à mesure de l'apprentissage) cela ne signifie pas qu'il a atteint un taux de prédiction intéressant. Ne soyez pas impressionnés par un pourcentage, un réseau qui prédit le sexe d'une personne avec un taux de réussite de 60% ne fait en fait que 10% de mieux qu'un tirage à pile ou face qui fournit, de base, une performance de 50 %.

Le surapprentissage : vous avez entraîné un réseau qui obtient un taux de bonnes prédictions de 99%, c'est un miracle ! Non pas forcément. Cette situation peut être synonyme d'un problème qui concerne tout processus d'apprentissage. En effet, si trop de neurones sont présents dans le réseau, il peut apparaître une sorte de phénomène d'apprentissage par cœur. Ainsi, le réseau arrive à identifier chaque image et à retenir la réponse associée. Il n'y a donc plus apprentissage au sens où le système n'a pas recherché des critères utiles pour fournir une réponse de qualité, mais il semble avoir appris par cœur la réponse associée à chaque input. Pour prévenir ce phénomène, on sépare la base d'apprentissage en deux : on garde 80% des données pour la phase d'apprentissage et 20% des données pour la phase de validation. Ainsi, on ne présente jamais le groupe de validation durant l'apprentissage et les performances du réseau sont évaluées uniquement sur ses performances par rapport à des données « nouvelles ». Cette approche est très rigoureuse et permet de détecter le surapprentissage.

L'évidence : vous avez entre les mains un système connu pour pouvoir apprendre et si vous le mettait dans des situations évidentes, il ne va pas s'embêter il va aller à la conclusion la plus rapide. Ce phénomène pourra parfois vous dérouter. Ainsi, si vous lancez un problème de classification entre des

grenouilles, des dauphins et des lions, en quelques secondes, quelle que soit la taille de la base d'images, vous allez arriver à un score de 100% de prédictions justes. Une nouvelle configuration de réseau très intelligente ? Un algorithme de descente ultra-performant ? Non rien de cela… En faisant d'autres tests, vous finirez par vous apercevoir qu'avec un seul neurone, le score restera à 100%. Pourquoi ? Parce que pour le réseau, la réponse est évidente. En moyennant les couleurs de l'image, il obtient un ton moyen qui représente la photo. Un ton plutôt vert est associé à la grenouille, plutôt jaune aux lions et plutôt bleu aux dauphins.

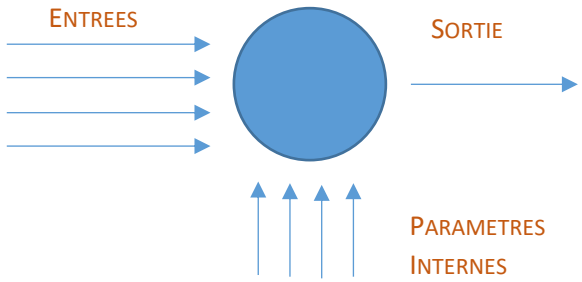
Des ensembles déséquilibrés : si vous cherchez à construire un réseau qui distingue les chats et les chiens et que vous fournissez 1000 photos de chats et 10 photos de chiens, le réseau va afficher un très bon score de 99%. En effet, en optimisant la fonction d'erreur, le réseau va vite détecter qu'en répondant toujours « chats » à la question sans regarder l'image en entrée, il a 99% de chance de voir juste ceci car vous avez plus de photos de chats dans votre base. C'est idiot comme réaction, mais un réseau en phase d'apprentissage fait ce qu'il a à faire, et s'il trouve une brèche : une réponse facile qui marche à tous les coups, il choisira cette option avec de grande chance. Il en est de même des photos de chauve-souris prises de nuit (fond noir) face à des photos d'oiseaux prises de jour (fond bleu clair).

Les phénomènes rares : si vous cherchez à détecter des signes annonciateurs de tremblements de terre dans des relevés sismiques, la tâche peut s'avérer impossible. En effet, ce sont des phénomènes rares et sur l'ensemble de vos données, par exemple 1000 jours d'enregistrement, il n'y aura qu'un seul tremblement de terre présent. Entraîner un réseau sur des cas aussi rares pose problème car les données sont déséquilibrées en faveur des moments sans évènements.

Remarque : lorsque vous manquez d'échantillons pour votre apprentissage, une technique consiste à utiliser du boosting pour simuler des données supplémentaires. On peut par exemple citer pour l'image : une symétrie verticale, une légère translation / rotation, l'ajout de bruit ou de tâches dans l'image d'origine. Ces techniques ont aussi pour objectif de rendre le réseau plus robuste aux perturbations.

4. Les réseaux de neurones

Le neurone informatique



Voici le schéma d'un neurone informatique. Il compte plusieurs entrées pour une UNIQUE sortie. Les entrées sont des valeurs numériques notées x_i. Ses paramètres internes sont appelés poids, en anglais weights, ce qui donne la notation w_i. Tous les neurones dans un réseau effectuent la même opération :

$$Sortie = \sum_{0}^{n-1} x_i * w_i$$

La sortie est donc un nombre flottant et va servir d'entrée aux neurones de la couche suivante. Cette opération peut être vue de différente manière, certains y voient une multiplication membre à membre, d'autres un produit scalaire entre deux vecteurs, d'autres une convolution. La formule, dans tous les cas, reste inchangée. On peut bien sûr mettre cette formule sous la forme de tenseurs en écrivant :

$$s = W.x$$

Le vecteur x est ici un vecteur colonne correspondant à l'ensemble des valeurs d'entrée. La matrice W dans le cas d'un neurone unique correspond à un vecteur ligne et, dans ce cas, le vecteur s contient une seule valeur. L'avantage de cette notation est qu'elle peut facilement se généraliser à k neurones traités en parallèle. Ainsi, si une couche (empilement de plusieurs neurones) contient 3 neurones traitant 2 valeurs d'entrées, nous aurons en sortie 3 valeurs, car chaque neurone fournit 1 valeur de sortie :

$$\begin{vmatrix} s_0 \\ s_1 \\ s_2 \end{vmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \end{bmatrix} \times \begin{vmatrix} x_0 \\ x_1 \end{vmatrix}$$

Vecteur colonne stockant les deux valeurs d'entrée

Vecteur colonne stockant les valeurs de sortie.

Matrice stockant les poids des neurones
Chaque ligne représente 1 neurone avec ses 2 poids

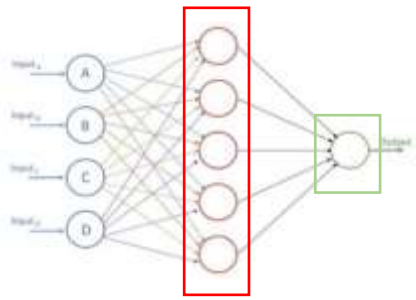
Dans cette couche, tous les neurones partagent et utilisent les mêmes valeurs d'entrée.

Couches et réseaux

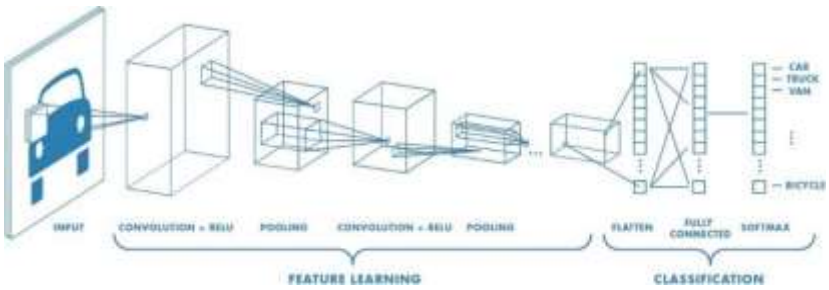
Une couche (layer) de neurones peut être représenté à partir d'un rectangle vertical, contenant des ronds correspondant aux neurones, lorsqu'il y en a peu en tout cas ! Les entrées, peuvent être aussi désignées comme **couche d'entrée – input layer**- même si aucun neurone n'est présent. Cependant, l'entrée ayant pour fonction de « charger » les données (images), elle a donc un rôle « actif » et peut recevoir ainsi le nom de layer.

La couche décrite par la formule $S = W.x$ s'appelle une **couche FC**, abréviation de FULLY CONNECTED. En effet, chaque neurone est connecté à chaque valeur d'entrée. Ainsi, si nous avons 50 neurones et 1000 valeurs d'entrée, nous aurons 50 000 poids stockées dans la matrice. Les couches FC sont des « classiques » des réseaux de neurones, elles sont pratiques car elles peuvent « tout faire » mais elles requièrent beaucoup de poids, ce qui n'est pas une bonne chose.

L'empilement de plusieurs couches permet de construire un réseau. Les entrées sont généralement situées sur la gauche, les couches successives sont empilées de gauche à droite et la sortie correspond à la couche la plus à droite :



Un réseau des années 80 😊 : 4 inputs, 5 neurones sur une couche FC centrale
et une deuxième couche FC avec un seul neurone donnant une valeur de sortie unique



Un réseau des années 2020 : les inputs correspondent à des images et peuvent contenir 600 valeurs
Chaque couche est symbolisée par un volume ou une colonne
Chaque couche porte un nom : CONV / POOL / FLATTEN / FULLY CONNECTED…

La fonction d'activation

Suffit-il d'empiler des couches pour construire un réseau ? Pas si sûr. En effet, prenons le cas de trois couches Fully Connected empilées. En utilisant la formule associée, nous obtenons :

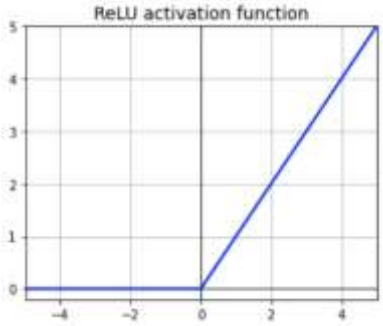
$$Out_{Réseau} = W_3.W_2.W_1.Input$$

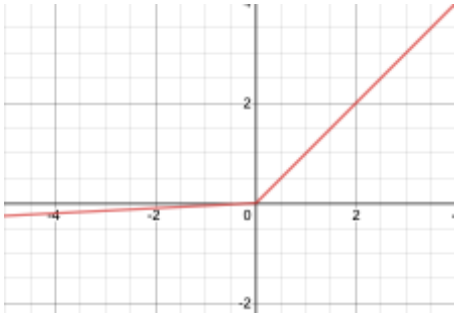
Où chaque matrice W_i correspond à une couche du réseau. Si nous nous contentons « juste » de cela, ce réseau à trois couches se ramènent à un réseau à seule couche. En effet, il suffit de prendre $M = W_3.W_2.W_1$ ce qui donne au final :

$$Out_{Réseau} = M.Input$$

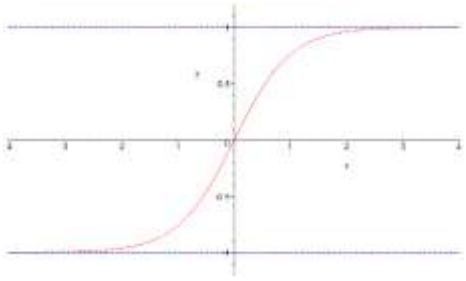
En effet, les Mathématiques nous disent que la composition de deux fonctions linéaires est linéaire, ce qui explique que tout réseau de plusieurs couches FC peut s'écrire de manière équivalent comme un réseau à 1 unique couche. Ainsi, pour avoir un réseau à plusieurs couches, il faut rajouter entre chaque couche une « non-linéarité » : c'est le rôle des fonctions d'activation. Ces fonctions sont des fonctions de $\mathbb{R} \rightarrow \mathbb{R}$ non-linéaire qui se branche à la sortie de chaque neurone. A la sortie d'une même couche, tous les neurones utilisent la même fonction d'activation. Une fonction courante aujourd'hui est la fonction *Relu*() :

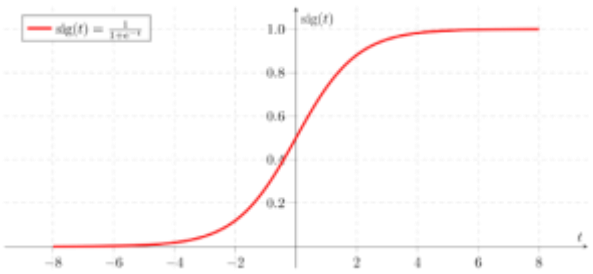
***Relu**(x) = max(0,x)* ou sa variante ***LeakyRelu**(x) = max(α.x,x)* avec *0<α<0.1*





Une fonction continue, dérivable quasiment partout et peu couteuse en calcul ! Dans les schémas des réseaux de neurones, vous trouverez ainsi des commentaires sous chaque couche du type FC+RELU pour indiquer le type de couche ainsi que sa fonction d'activation. Comme autre fonction d'activation, on peut trouver tanh et sigmoïde : $sig(x) = 1/(1+e^{-x})$





Ces deux fonctions sont non-linéaires. Lorsque les valeurs de sortie des neurones sont importantes, elles ramènent ces valeurs dans l'intervalle [-1,1] ou [0,1] respectivement, ceci permet de contrecarrer les artefacts. La fonction sigmoïde a une propriété notable : $sig'(x) = sig(x) \times [\, 1 - sig(x) \,]$.

5. Les différentes couches

La couche POOL

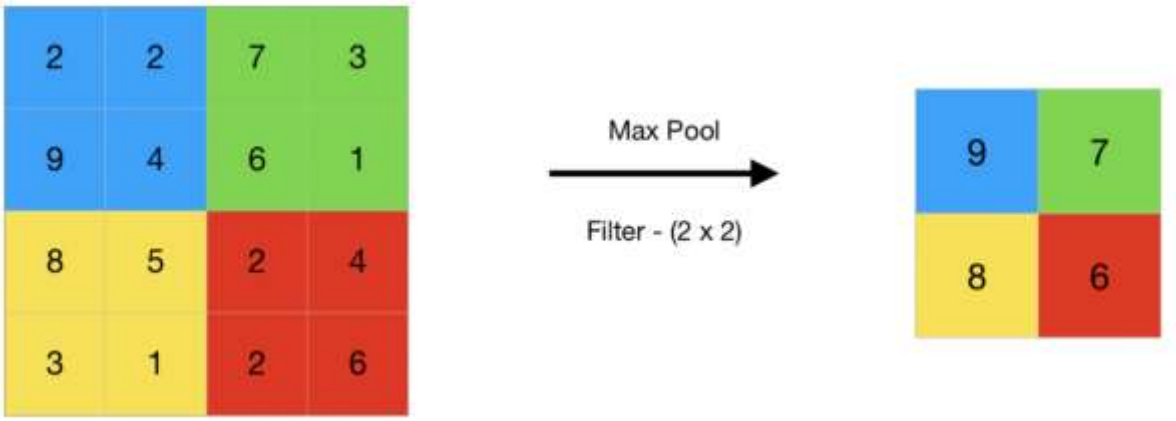
La couche POOL est une couche spécialisée qui se distingue des couches FC. Son objectif est de réduire la taille de la sortie d'une couche, ce n'est pas vraiment une couche effectuant un « traitement » des données. Ainsi, pour un tenseur 2D de résolution 1024x768, on peut vouloir réduire sa taille. Pour cela, nous allons découper ce tenseur en bloc de taille 2x2 ou 4x4 et effectuer pour chaque bloc une opération du type : moyennage, min ou max…

Le nombre de poids d'une couche dépend souvent de la taille des informations en entrée. Ainsi, si l'on a trop d'entrées, on va augmenter la quantité de poids et cela est toujours mauvais pour le surapprentissage et les temps de calcul. Les couches POOL sont ainsi des couches permettant de limiter le flux de données d'une couche à l'autre.

Vous pouvez lire plus d'informations dans la documentation Pytorch ici :

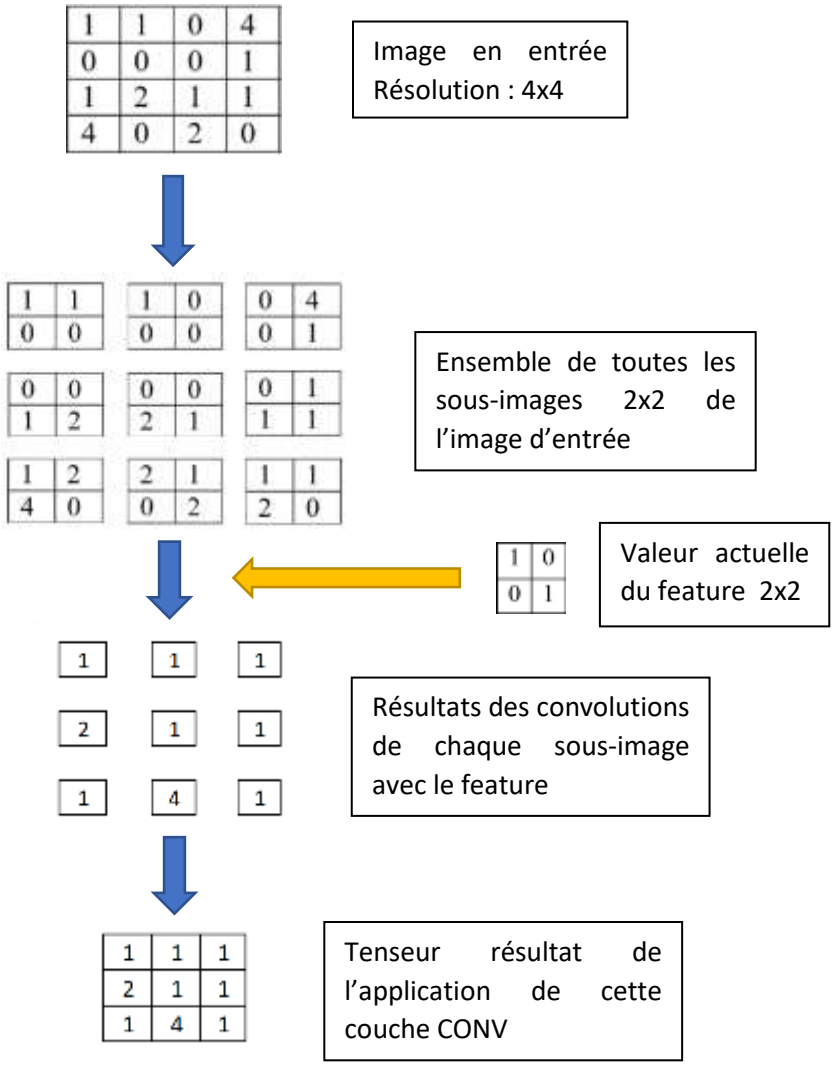
https://pytorch.org/docs/stable/nn.html#pooling-layers

Exemple de MaxPool avec une réduction 2x2 :



La couche CONV

La couche CONV est la couche qui a, sans aucun doute, donné toute la puissance aux réseaux de neurones actuels. Nous allons présenter une version « basique » de cette couche en introduction. Pour cela, nous allons lui associer un unique « feature » (caractéristique) aussi appelé « kernel » (noyau). Sur une image 2D, nous choisissons un feature 2D de taille 2x2. La couche CONV parcourt l'image en x et en y et effectue pour chaque sous-image 2x2 une convolution (multiplication membre à membre) avec ce feature. Le résultat, un scalaire, correspond à la sortie de la couche CONV pour la sous-image courante. Voici l'exemple correspondant :



Pour une image de taille $n \times m$ et un feature de taille $u \times v$, le tenseur résultat de la couche CONV a pour dimension :

$$(n-u+1, m-v+1)$$

Ceci est dû au parcours de l'image de x allant de la valeur 0 à la valeur $n-u$ et de y allant de 0 à la valeur $m-v$. Globalement, les features sont de petites tailles. Ainsi, le tenseur résultat a une dimension proche de l'image en entrée (on suppose ici que la taille d'une image est de l'ordre de 32x32).

Cependant, comme dans une image les zones de transition sont souvent « douces », il n'est pas nécessaire de calculer la convolution en chaque point de l'image, mais par exemple tous les 2 pixels. Cela permet de réduire la taille du tenseur de sortie et ceci fait gagner un temps de calcul conséquent. Le **pas** désigne ainsi cette quantité et permet de régler le « saut » entre chaque convolution. Dans le cas où la taille du pas correspond à la taille du feature, cela veut dire qu'aucunes sous-images utilisées pour la convolution ne se chevauchent et que tous les pixels de l'image ont été utilisés dans les calculs.

Les bords peuvent poser problème, car en effet, la taille du tenseur de sortie est légèrement diminuée en rapport avec la taille du feature. Pour éviter cela, il suffit au départ d'élargir virtuellement l'image d'entrée avec des 0. Ce paramètre de **marge ou zero padding** permet ainsi de régler finement la taille du tenseur de sortie.

Une couche CONV dans le cas général peut contenir plusieurs features. Ainsi, on effectue les mêmes traitements que dans le cas d'un unique feature, sauf que l'on réitère autant de fois que ce qu'il y a de feature gérés par la couche CONV, cette grandeur est appelée **la profondeur de la couche CONV**. Par exemple, une couche CONV avec un pas de 1 et gérant 7 features de taille 3x3 donne en sortie de traitement d'une image 10x20, un tenseur de taille : [8x18x7].

Le pas, la marge et la profondeur de la couche CONV sont des hyperparamètres d'apprentissage du réseau : ils sont choisis par l'utilisateur en début d'apprentissage et n'évoluent pas durant l'apprentissage.

La couche CONV participe aux calculs Forward du réseau « comme les autres » : son résultat contribue à l'erreur finale. Ainsi les valeurs de ses features se comportent comme des poids du réseau qui vont être optimisés par la méthode du gradient.

Mais à quoi peuvent correspondre les features une fois appris ? Il s'agit d'une question difficile. S'il est difficile de savoir pour les couches supérieures du réseau, on sait que pour les couches proches de l'image d'entrée, ces features correspondent à des détecteurs de contours. Voici un exemple de résultat pour la classification des chiffres manuscrits :

Ainsi les features cherchent à détecter dans le set d'entrée des « caractéristiques » permettant de décrire l'information présente dans l'image. Si une couche CONV a seulement 5 features, elle va devoir en choisir 5 qui représentent au mieux l'information recherchée dans l'image.

Une couche CONV génère peu de poids comparée à une couche FC.

Vous pouvez lire plus d'informations dans la documentation Pytorch ici :

https://pytorch.org/docs/stable/nn.html#convolution-layers

A noter que l'on parle de réseau CNN (convolutional neural network) lorsqu'au moins une couche CONV est présente dans le réseau. L'utilisation seule de couches FC ne produit pas un CNN !

6. La classification

Le problème de classification d'images consiste à associer une image à une catégorie parmi une liste de catégories connues. Les bases les plus classiques (CIFAR10) proposent une liste de 10 catégories : avion, voiture, oiseau, chat, daim, chien, grenouille, cheval, navire et camion. Le réseau est représenté par une fonction à valeur dans $\mathbb{R}^{10}$.Ces 10 valeurs notées $S_0 \ldots S_9$ vont correspondre au score associé à chaque catégorie. Le score le plus haut indique la catégorie proposée. La base d'images nous donne le numéro, noté k, de la catégorie exacte de chaque image. Comment construire la fonction d'erreur ? Ce n'est pas évident ! Partons du principe que la prédiction est incorrecte. Ainsi certains scores sont supérieurs au score de la bonne catégorie. On pourrait donc écrire ceci :

$$Erreur = \sum_{i=0}^{9} S_i - S_k$$

On remarque que pour i=k, la grandeur S_k-S_k vaut 0 et elle n'intervient dans la valeur finale de l'erreur. Pour i≠k, lorsque $S_i > S_k$ le score de la catégorie i est supérieur au score de la bonne catégorie, la grandeur S_i-S_k est donc positive et contribue à augmenter l'erreur. Plus l'écart est important, plus l'erreur augmente, ce que nous recherchons. Cependant, lorsque pour i≠k, $S_i < S_k$ le score de la mauvaise catégorie est inférieur au score de la bonne catégorie. Cependant, la grandeur S_i-S_k intervient dans l'erreur en tant que grandeur négative et contribue à diminuer l'erreur. Cela pourrait être intéressant mais cela s'avère contreproductif. En effet, un bon score ne doit pas contrebalancer un mauvais score quitte à créer une erreur nulle (donc un résultat juste). Il ne faut donc pas tenir compte de ces situations. Pour cela, il faut gommer de la fonction erreur toutes les configurations où le score d'une mauvaise catégorie est inférieur au score de la bonne catégorie. Cette demande ressemble à l'écriture d'une condition, c'est donc une fonction max/min qui va pouvoir nous aider ici (sinus, sqrt… ne permettent pas cela). On va donc réécrire la fonction erreur de la manière suivante :

$$Erreur = \sum_{i=0}^{9} max(0, S_i - S_k)$$

Ainsi ne contribue à la valeur de l'erreur que les scores, des mauvaises catégories, supérieurs au score S_k de la bonne catégorie. Cette approche fonctionne. Cependant, l'expérience va montrer une certaine instabilité dans les résultats. Pourquoi ? Parce qu'il se peut que le score de la catégorie avion soit de 3.999 et celui de la catégorie chien de 4.000. Si l'image correspond à celle d'un chien, nous n'avons théoriquement aucun problème. Pourtant, les deux scores sont si proches, qu'il suffirait d'un peu de bruit pour que le score de l'avion puisse passer devant celui du chien. Quel est notre problème ? Le critère supérieur n'est pas suffisant, car des scores très proches vont entraîner de l'instabilité dans les résultats. Ainsi, nous cherchons maintenant à ce que le score de la bonne catégorie soit bien au-dessus des autres d'un valeur Δ. Comment faire cela ? Il suffit de changer légèrement l'expression de l'erreur :

$$Erreur = \sum_{i=0}^{9} max(0, (\Delta + S_i) - S_k)$$

Ainsi un score de S_k = 4 face à un score S_i = 3 et un Δ de 2 produit une erreur : (2+3) - 4 = 1. Il faut que le score S_k atteigne la valeur 5 = $S_i + \Delta$ pour que l'erreur associée soit nulle.