Introduction à l'apprentissage profond

Adrien CHAN-HON-TONG

17 mars 2020

1 Introduction

1.1 Un exemple concret

Vous avez forcément déjà entendu parler de *deep learning* (apprentissage profond en français) une technique d'apprentissage par ordinateur (*machine learning* en anglais) mettant en oeuvre des réseaux de neurones profonds. L'exemple le plus médiatique est probablement *alphago*[1]. On parle aussi beaucoup d'applications dans le médical. Néanmoins, ces exemples médiatiques ne sont pas représentatifs du domaine.

La figure 1 est un exemple bien plus représentatif : il s'agit de 3 images de la base VisDrone aiskyeye.com sur lesquelles on a appliqué l'algorithme Faster-RCNN (décrit ici arxiv.org/abs/1506.01497 et directement disponible en module tout bien emballé là pytorch.org/docs/stable/torchvision/models.html) après adaptation au jeu de données. Les détections calculées sont incrustées en bleu dans les images. Certes, on est loin de la performance d'un humain : il y a à la fois des objets non détectés et des fausses alarmes. Mais néanmoins, la qualité des détections est *assez* bonnes, en tout cas très largement au dessus des performances des algorithmes ne contenant pas d'apprentissage profond. Et surtout, ces détections ne proviennent pas d'un obscure algorithme caché par une grande puissance mais simplement d'un code présent partout sur internet que n'importe quel ingénieur peut (plus ou moins) rapidement mettre en oeuvre s'il a le matériel adapté !

Cet exemple est représentatif car les 4 forces qui financent la recherche mondiale en apprentissage profond sont :

- la vidéo surveillance principalement poussé par la Chine avec l'exemple de aiskyeye.com
- le renseignement à l'échelle planétaire principalement poussé par les États unis par exemple avec xviewdataset.org (c'est par ailleurs mon coeur de métier à l'ONERA).
- le numérique (l'exemple typique est le papier DeepFace de facebook disponible le site de CVPR)
- et la voiture autonome (www.cityscapes-dataset.com)

1. www.nature.com/articles/nature24270

FIGURE 1 – Illustration de l'algorithme Faster RCNN sur la base VisDrone.

1.2 La révolution de l'apprentissage profond

Ce cadre étant fixé, il est clair que le mot clé *apprentissage profond* est signifiant. On le retrouve dans la communication de presque toutes les start up, mais aussi dans d'innombrable appels à projets. D'un point de vue scientifique, les conférences clés de l'apprentissage par ordinateur CVPR, ICCV, ECCV, ICML, NIPS sont aujourd'hui le top des conférences pour tout un tas de métrique (dont par exemple www.guide2research.com/topconf) alors que l'apprentissage par ordinateur était un sujet parmi d'autres il y a 5 ans (CVPR était 60ème il y a 5 ans, elle est aujourd'hui 1ère).

Pour certains, le buzz autour de cette technique n'est rien qu'une mode. Pour d'autres, l'apprentissage profond est une révolution. Une révolution parce que l'apprentissage profond a démontré que l'apprentissage par ordinateur peut obtenir des performances humaines dans certains cas. Or, l'apprentissage par ordinateur est la seule façon d'essayer de résoudre certains problèmes d'intérêt.

En soit, l'apprentissage par ordinateur (notamment via l'apprentissage profond) ne va probablement pas autant changer nos vies qu'internet (au moins à court terme). Internet a permis une rupture dans l'organisation de nos sociétés à la fois d'un point de vue économique, industriel et récréatif. Mais internet reste un outil de communication qui ne dépassera jamais ce cadre de système de communication.

Inversement, l'apprentissage par ordinateur pourrait permettre d'implémenter de nombreux fantasmes de science fiction de l'arme autonome à l'esclave robot en passant par des systèmes de diagnostic automatique ou des systèmes de surveillance généralisé (*big brother* en beaucoup plus massif) ou encore le véhicule autonome. Bien qu'aucun de ces systèmes n'existent actuellement, l'apprentissage par ordinateur permet, avec raison, d'imaginer que ces systèmes puissent exister. C'est donc actuellement une révolution de l'imaginaire plus qu'une révolution réelle.

Ce qui est sur, c'est qu'en tant que jeunes ingénieurs, il est peut probable que vous n'ayez jamais à interagir avec ce type d'outils.

1.3 Quelle science pour l'apprentissage profond ?

La communauté scientifique en apprentissage profond se structure principalement autour de deux axes :

- des conférences plutôt théoriques comme NIPS ou ICML traitant
  - des mécanismes internes des réseaux de neurones
  - de l'expressivité de ces réseaux
  - des garanties probabilistes sous jacente
- des conférences plutôt appliquées comme CVPR ou ICCV traitant principalement des performances obtenues sur des problèmes classiques pour
  - différentes architectures
  - différentes méthodes d'optimisation
  - différentes fonctions d'erreurs
  - différentes normalisations

Présenter comme ça, le domaine semble plus technique que scientifique, avec éventuellement un peu plus de science du coté des conférences NIPS, ICML. De fait, le domaine reste très technique en pratique. Mais philosophiquement c'est bien un domaine scientifique : celui de la science des données.

Je me permets une métaphore : la science c'est oser énoncer l'hypothèse que $\vec{f} = m\vec{a}$ à partir d'observations de la nature. Les schémas numériques permettant des calcul plus ou moins fins de $\vec{a}$ sont tout au plus des mathématiques mais en tout cas n'ont rien à voir avec de la science. Or, comprendre comment fonctionne un réseau de neurones, c'est semblable aux questions sur les schémas numériques : ce n'est pas si scientifique que ça. De plus, il est probable que dans 10 ans on n'utilise plus des réseaux de neurones mais *la méthode du schmilblick*. La compréhension des mécanismes des réseaux de neurones ne servira plus à rien. Mais les données elles seront toujours là.

La science de l'apprentissage par ordinateur c'est de caractériser les données, par exemple, caractériser la distribution des données qu'on observe par exemple en prenant une photo depuis un satellite (c'est à dire émettre l'hypothèse $\vec{f} = m\vec{a}$ pas juste formaliser la notion d'accélération).

Philosophiquement, l'apprentissage par ordinateur est donc bien une science. Mais une science jeune qui étudie des objets encore trop complexes, et, qui en pratique tend à être utilitariste : on ne cherche pas tant à *caractériser la distribution des données* mais plus à être capable *d'avoir tels performances sur telle bases*. Dit autrement, la seule caractérisation qu'on cherche c'est la performance qu'on sait atteindre. De plus, les outils manipulés (donc des réseaux de neurones) sont assez instables. On entend souvent que l'apprentissage par ordinateur serait de la *cuisine*. En réalité quand je fais un gâteau au chocolat je suis sur que rajouter un peu d'amande ne dégradera pas trop mon gâteau. Mais quand j'optimise des réseaux de neurones, je sais que la moindre modification peut *tout casser*.

Tout cela rend difficile d'enseigner cette matière : vous apprendre comment appliquer Faster-RCNN à VisDrone pour obtenir la figure 1 serait trop spécifique, mais vouloir transmettre des connaissances générales du domaine se heurte au fait qu'il y en a peu.

1.4 Objectif pédagogique

Pour ces raisons, l'objectif pédagogique du cours est principalement de démystifier l'apprentissage par ordinateur. À la sortie de ce cours, mon objectif n'est pas tant que vous sachiez mettre en oeuvre un réseau de neurones (vu le nombre de tuto sur le net, je pense que vous trouverez si ça vous intéresse) mais que vous soyez en mesure de comprendre qu'il n'y a aucune magie dans ces techniques.

En soit, ce n'est pas forcément un cadeau : beaucoup d'entreprises préféreront sûrement quelqu'un qui sait faire sans forcément comprendre (qui pourra même dire des choses fausses sans mentir s'il y croit) que quelqu'un qui voit les limites avant de savoir faire. Je conçois donc que cet objectif soit partial et partiel et qu'il puisse ne pas correspondre à vos attentes. Si c'est le cas, vous serez

rassurés de savoir que le cours sera complété par un projet plus appliqué, sinon tant mieux.

Le plan du cours est donc le suivant :

— Que peut et ne peut pas l'apprentissage par ordinateur ?

— Qu'est ce qu'un réseau de neurones ?

— Comment se passe l'optimisation d'un tel réseau ?

À noter que le premier point n'est pas spécifique à l'apprentissage profond puisqu'il s'agit de généralité sur l'apprentissage par ordinateur (l'apprentissage profond n'est qu'un outils). Il y aura donc probablement des redites avec d'autres cours et même d'autres séances de ce même cours. Néanmoins, comme c'est ce que je considère comme le plus important à bien comprendre, ça ne me gène pas d'insister sur ce point.

2 Que peut et ne peut pas l'apprentissage par ordinateur ?

2.1 La classification binaire supervisée

La classification binaire supervisée est le problème d'apprentissage par ordinateur le plus simple à formaliser. Le point de départ est que nous voulons disposer d'un ordinateur qui peut calculer la fonction y à valeur dans $\{-1, 1\}$. Par exemple y est la fonction qui prend une image et dit si cette image est une image de chat ou pas (avec 1 pour chat et -1 pour non chat).

Le problème c'est qu'on ne sait pas décrire cette fonction : même si on avait un ordinateur infiniment puissant, même si on pouvait générer en 1 seconde toutes les images RGB de 320×200 pixels (i.e. $255^{3\times320\times200}$ images), personne ne sait définir formellement ce que devrait être y. Il faut noter que ce point est une différence essentiel (d'un point de vue théorique) entre l'intelligence artificielle type planification de travaux, optimisation de trajectoires, recherche opérationnelle, jeux d'échec / de GO / Starcraft et l'apprentissage par ordinateur. Ces différents problèmes pourrait se résoudre avec un ordinateur arbitrairement puissant puisque tous les cas peuvent être testés. À l'inverse, en apprentissage par ordinateur, on ne sait pas ce que y devrait faire, on ne peut pas la calculer même avec un tel ordinateur.

Pourtant, si on montre une image à un humain, il est capable de dire si c'est l'image d'un chat. Donc, on peut avec un humain savoir ce que vaut $y(x)$ sur un x précis, mais pas avec un ordinateur même arbitrairement puissant : la ressource ultime de l'apprentissage par ordinateur ce n'est pas la puissance de calcul mais l'humain.

En pratique, on va donc prendre un humain et lui présenter plein d'images $x_1, ..., x_N$ et donc pour ces images, on connaîtra $y(x_1) = y_1, ..., y(x_N) = y_N$. Avec ces données annotées appelés exemples d'apprentissage on va chercher à construire une approximation f de y. Ce qu'on voudrait c'est que sur une autre donnée χ on ait $\text{signe}(f(\chi)) = y(\chi)$. χ est une donnée de test, c'est une donnée que l'on n'a pas utilisé pour concevoir f.

Pour résumer, la classification binaire supervisée c'est construire f à partir de $(x_1, y_1), ..., (x_N, y_N)$ des données d'apprentissage dans l'espoir que signe$(f) = y$.

2.2 Forces et faiblesses

La vrai force de l'apprentissage par ordinateur c'est que c'est la **seule** technique applicable sur des problèmes entrée-sortie mal posés notamment celui de savoir si une image est une image de chat (mais plus globalement, les problèmes de perception voire de traitement du signal). La seule alternative, c'est l'humain. Or, la perception est une brique essentiel de l'autonomie (arme/véhicule autonome) et le traitement du signal une brique essentiel du diagnostic.

Dit autrement, la force de l'apprentissage par ordinateur c'est que c'est la **seule** technique qui permettra de remplacer l'humain dans des taches mal posés comme la perception indispensable à l'autonomie des système.

Maintenant, la faiblesse intrinsèque de l'apprentissage par ordinateur c'est que comme on l'applique sur des problèmes mal posé (c'est à dire qu'on ne connaît pas y), on ne peut et on ne pourra jamais prouvé que signe$(f) = y$. Sans avoir besoin de démonstration mathématique, il existe plusieurs y qui donne la même base d'apprentissage (il suffit que y soit différent en dehors de la base), or on ne connaît y qu'à travers cette base, donc plusieurs fonctions y donneront la même base et donc nécessairement le même f alors que signe(f) ne peut être égal qu'à une de ces fonctions.

À la différence d'autres problèmes d'intelligence artificielle qu'on peut spécifier et pour lequel on peut prouver mathématiquement qu'un algorithme les résout à 100%, si on utilise de l'apprentissage par ordinateur il persiste toujours la possibilité d'une erreur (par chance on peut avoir signe$(f) = y$ mais on ne peut pas le démontrer donc potentiellement signe$(f) \neq y$).

Après, rien n'impose à une société d'utiliser de l'apprentissage par ordinateur (voiture/arme autonome et etc) puisque ça ne sert qu'à remplacer l'humain. D'autant plus que, si on remplace l'humain ce sera forcément avec des algorithmes non prouvés à 100% (à minima pour la perception) et donc avec la possibilité d'erreurs. Personnellement, si je suis chercheur en apprentissage par ordinateur c'est parce que je pense qu'il y aurait moins d'accidents avec la voiture autonome qu'avec des conducteurs. Mais d'une part il faudrait le vérifier expérimentalement. Et d'autre part quand bien même ce serait vrai, chaque accident serait alors un accident collectif, et, non plus un accident individuel. Cette question politique ne sera évidemment pas débattue ici mais il faut noter qu'elle traverse notre société.

2.3 Comment évalue-t-on ces modèles en pratique

Maintenant puisqu'on ne peut pas prouver un algorithme d'apprentissage par ordinateur à 100% que fait on ? Puisque ces algorithmes ne peuvent pas se prouver, ils sont en pratique qualifiés. La démarche globale consiste à évaluer le modèle sur une base de test après apprentissage.

On va donc prendre un humain et lui présenter plein d'images $x_1, ..., x_N$ pour connaître $y(x_1), ..., y(x_N)$ et ainsi avoir une base d'apprentissage avec laquelle on va construire un modèle f.

Puis, on va ensuite remontrer à cet humain des images $\chi_1, ..., \chi_M$ pour connaître $y(\chi_1), ..., y(\chi_M)$ et ainsi former une base de test. On peut alors calculer l'erreur empirique e_e que fait f sur $\chi_1, ..., \chi_M$:

$$e_e = \frac{1}{M} \sum_{m \in \{1,...,M\}} \frac{|\text{signe}(f(x)) - y(x)|}{2}$$

qui revient à compter le nombre de fois que y est différent de signe(f) sur $\chi_1, ..., \chi_M$ car signe(f) et y sont à valeurs dans $\{-1, 1\}$. L'erreur réelle e_r de f quand on tire les données x avec une probabilité P est

$$e_r = \frac{1}{2} \int_x |\text{signe}(f(x)) - y(x)| P(x) \mathrm{d}x$$

C'est cet erreur réelle e_r qui nous intéresse (on la voudrait nulle ou à minima connue) mais on ne peut connaître que e_e (qui convergent vers e_r quand M tend vers l'infini mais rien de plus).

Noter aussi que l'erreur sur la base d'apprentissage peut être totalement décorrélée de l'erreur réelle : c'est le phénomène de l'apprentissage par coeur (on est bon sur les données d'apprentissage mais sur aucune donnée de test).

2.4 Deux mots sur le No-Free-Lunch théorème et le PAC learning

Au delà de la qualification, une question mathématique se pose : même si plusieurs y correspondent à la même base d'apprentissage, comme on peut supposer que la base d'apprentissage est tiré uniformément, n'avons nous pas une forte probabilité de ne pas faire trop d'erreur ? Cette question sera normalement traitée plus profondément dans un cours ultérieur. Je me permets ici juste d'en dire 2 mots.

Si on considère que tous les problèmes y possibles avec la même probabilité, alors, tous les algorithmes se valent. C'est le *no free lunch theorem*, qui est donc une sorte de théorème de Godel au sens où cela supprime tout espoir d'une solution mathématique pure. Heureusement, il se trouve que les problèmes qu'on cherche à résoudre ne sont pas tous les problèmes. L'apprentissage par ordinateur est donc bien une science des données et ce théorème n'empêche pas qu'on puisse faire des systèmes qui marchent en pratique. Pour faire un parallèle, il serait impossible de faire voler un avion si g la constante de gravitation pouvait changer de façon aléatoire, il se trouve qu'elle est constante.

Ce résultat négatif n'est cependant pas en contradiction avec des résultats partiellement positif : il existe toute une théorie mathématique sur les garanties statistiques qu'on peut avoir sur des résultats d'algorithme d'apprentissage par ordinateur. Cette théorie s'appelle la *PAC : probably approximately correct*. Un

résultat fondamental basé sur la dimension de Vapnik Chervonenkis (VC), dit qu'avec une probabilité $1-\delta$ on a l'inégalité $|e_r - e_e| \leq G(VC(f), M, \delta)$ avec e_r et e_e les erreurs réelles et empirique et avec $G(M, \delta)$ une fonction qui décroît avec M et δ (bien évidemment, G tend vers l'infini quand δ tend vers 0).

On a donc ici un résultat qui permette de maîtriser l'erreur. Il est bien plus fort qu'une simple convergence de e_e vers e_r : ici, on peut borner l'erreur réelle. Mais ce résultat est probabiliste : il n'est vrai qu'avec une probabilité de $1-\delta$.

2.5 Bilan

L'objectif est de disposer d'un ordinateur qui peut calculer la fonction y que l'on ne sait pas définir mais qu'un humain peut évaluer.

Pour cela, on tire des données $x_1, ..., x_N$ selon une probabilité P (P et y sont le problème cible) et on utilise un humain pour connaître $y(x_1), ..., y(x_N)$ et ainsi avoir une base d'apprentissage.

Avec cette base, on construit un modèle f.

La performance de f peut être mesurée sur des données de test $\chi_1, ..., \chi_M$ par $\frac{1}{M} \sum_{m \in \{1,...,M\}} \frac{|\text{signe}(f(x)) - y(x)|}{2}$. Cette mesure approxime l'erreur réelle.

3 Qu'est ce qu'un réseau de neurones ?

Avant de parler de réseau de neurones puis d'apprentissage profond, il est important de bien comprendre que *la fin justifie les moyens*. La seule chose qui compte c'est de trouver le modèle f qui marche.

La science de l'apprentissage par ordinateur c'est de trouver des familles de fonction $\mathcal{F}$ qui sont à la fois expressives, régulières et qui s'apprennent bien. Il y a eu les arbres et les SVM (qui sont des réseaux à 1 couche), aujourd'hui c'est les réseaux profonds, demain ce sera autre chose. C'est à force de tests qui la communauté s'oriente vers telle ou telle familles, avec des effets de mode indéniables, mais aussi et surtout avec comme boussole la performance à l'instant t.

Ici, je vais présenter une famille : celle de l'apprentissage profond c'est à dire celle des réseaux de neurones. Certains points que je vais vous présenter sont explicables (ex : dues à une facilité pour l'apprentissage) mais d'autres se sont imposés sans raison apparentes autre que *ça marche*. Maintenant vous aurez peut être à la lecture de ce cours l'impression que ça pourrait être mieux en changeant ça ou ça. Sachez que la seule chose *vraie* c'est la performance finale sur la base de test donc il est sain de vouloir essayer autre chose (après avoir vérifier dans la littérature scientifique que cette autre chose n'existe pas déjà). C'est même un métier : celui de chercheur en apprentissage par ordinateur.

3.1 Le perceptron multi couche

Les réseaux de neurones existent en réalité depuis longtemps (1970). Ils ont émergés suite à un ensemble d'améliorations concomitantes avec une augmen-

tation de la puissance de calcul. Mais en réalité ce sont des objets extrêmement simples mathématiquement : toute ma longue introduction est juste là pour cacher le fait qu'on peut décrire un réseau de neurones en 10 lignes XD.

3.1.1 Le neurones

Le neurone, la brique de base de tout le réseau, est un simple opérateur linéaire : il prend en entrée un vecteur et retourne le produit scalaire noté par le point . de ce vecteur avec un vecteur de même taille qui correspond au poids du neurones (auquel on ajoute généralement un biais) :

$$neurone_{w,b} : \begin{array}{ccc} \mathbb{R}^D & \rightarrow & \mathbb{R} \\ x & \rightarrow & w.x + b \end{array}$$

3.1.2 La couche de neurones

Une couche de K de neurones est simplement un K-uplet de neurones prenant la même entrée, et, dont les K sorties individuelles sont regroupées en 1 vecteur :

$$couche_{W,B} : \begin{array}{ccc} \mathbb{R}^D & \rightarrow & \mathbb{R}^K \\ x & \rightarrow & \begin{pmatrix} neurone_{W_1,B_1}(x) \\ ... \\ neurone_{W_K,B_K}(x) \end{pmatrix} \end{array}$$

3.1.3 Le réseau de neurones

L'entrée et la sortie d'une couche de neurones sont des vecteurs de taille quelconque (car il n'y a pas de contrainte sur D et car K est choisi - c'est le nombre de neurones de votre couches). On peut donc appliquer une couche de neurones à la sortie d'une couche de neurones.

Cependant, la composition de 2 opérateurs linéaires est simplement un opérateur linéaire. Pour pouvoir obtenir des réseaux ayant un comportement non linéaire, on ajoute entre 2 couches une fonction non linéaires dite fonction d'activation. La plus utilisée aujourd'hui est la fonction **relu** de $\mathbb{R}$ dans $\mathbb{R}$ définie par $relu(x) = \max(0, x)$ parfois notée $[x]_+$. Cette notation est ensuite étendue aux vecteurs en dimension D quelconque : $relu(x) = \begin{pmatrix} relu(x_1) \\ ... \\ relu(x_D) \end{pmatrix}$.

Ainsi, un réseau de neurones entièrement connectées (multi layer perceptron en anglais) à activation relu de profondeur Q est juste un empilement de Q couche de neurones - la dernière est classiquement un seul neurone :

$$reseau_\theta : \begin{array}{ccc} \mathbb{R}^D & \rightarrow & \mathbb{R} \\ x & \rightarrow & C_{W_Q,B_Q}(relu(C_{W_{Q-1},B_{Q-1}}(...relu((C_{W_1,b_1}(x)))...))) \end{array}$$

3.1.4 Choix des poids

Chaque neurone de chaque couche a un vecteur de poids. En changeant les poids d'un réseau (sans même changer son architecture), on peut obtenir une fonction très différente.

Mais comment choisit-on l'architecture puis chaque poids ? Pour ce qui est de l'architecture, on dérive généralement ce qui se fait dans la communauté scientifique (plus pour les réseaux convolutionnels qu'on verra après, en soit il n'y a pas trop de choix dans un perceptron multicouches). Pour ce qui est des poids, on les optimise à architecture fixé sur la base d'apprentissage c'est l'objet de la section 4. L'objet de cette section est de présenter fonctionnellement ce que sont ces objets.

On manipulera donc beaucoup de poids triviaux (comme quand on cherche une racine évidente). Par exemple, si je vous dis, je cherche un réseau travaillant sur des points 2D (i.e. $\mathbb{R}^2$) à 2 couches tel que signe$(f((-1,-1))) = -1$, signe$(f((2,-1))) = -1$, signe$(f((-1,2))) = -1$ mais signe$(f((1,1))) = 1$, une solution c'est $f(x) = 1 - 2relu((-1,0).x) - 2relu((0,-1).x)$.

On est bien d'accord qu'à ce stade, il n'y a pas de méthode (autre que tout explorer pour trouver cette solution). Cela permet néanmoins de se familiariser avec ces objets.

3.1.5 Description graphique, mathématique ou informatique

Pour être plus concret, je vous propose 3 représentations équivalentes d'un même réseau de neurones.

La représentation mathématique consiste à écrire le réseau comme une fonction

$$f(x) = \begin{array}{l}\text{relu(relu((1,0).x)-relu((2,-1).x)+relu((5,2).x))} \\ \text{- relu(relu((1,0).x)+relu((2,-1).x)-3relu((5,2).x))}\end{array}$$

Il s'agit d'un réseau à 3 couches qu'on peut décrire informatiquement par :

- la première couche a 3 neurone, prend $z \in \mathbb{R}^2$ et retourne $\begin{pmatrix}\text{(1,0).z}\\ \text{(2,-1).z}\\ \text{(5,2).z}\end{pmatrix}$
- la deuxième couche $z \in \mathbb{R}^3$ et retourne $\begin{pmatrix}\text{(1,-1,1).z}\\ \text{(1,1,-3).z}\end{pmatrix}$
- la dernière couche prend $z \in \mathbb{R}^2$ et retourne $(1,-1).z$
- bien entendu, le z en entrée de la couche 2 est la sortie de la couche 1 avec un relu entre les deux !

On peut aussi le représenter d'une façon graphique par la figure.

Finalement, si vous utiliser pytorch (pytorch.org), le code nécessaire pour utiliser ce réseau est présenté dans la table 1.

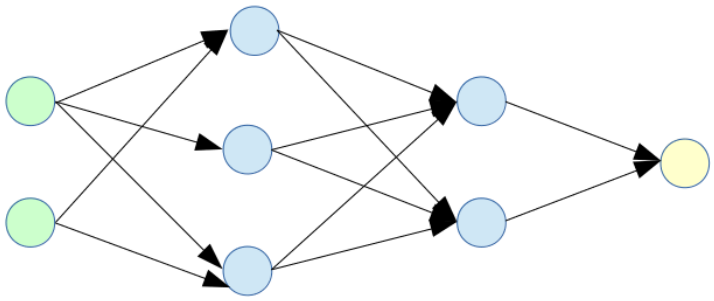

FIGURE 2 – illustration graphique du réseau de neurones de la section 3.1.5

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(2, 3, bias=False)
        self.l1.weight.data = torch.Tensor([[0,1],[2,-1],[5,2]])

        self.l2 = nn.Linear(3, 2, bias=False)
        self.l2.weight.data = torch.Tensor([[1,-1,1],[1,1,-3]])

        self.l3 = nn.Linear(2, 1, bias=False)
        self.l3.weight.data = torch.Tensor([[1,-1]])

    def forward(self,x):
        return self.l3(F.relu(self.l2(F.relu(self.l1(x)))))

net = Net()
for i in range(-5,6):
    for j in range(-5,6):
        print(i,j,net(torch.Tensor([i,j])))
```

TABLE 1 – Code pytorch de l'exemple de la section 3.1.5

3.2 Les réseaux convolutifs

3.2.1 Image et dimension

Prenons une image standard $320x200$ pixels, si on veut utiliser un réseau de neurone classique, il convient de transformer l'image en un vecteur de dimension $320x200$. La première couche de notre réseau va donc voir des poids en dimension $320x200$! Bref, il est complètement impossible de traiter de vraies images avec un perceptron multi couches.

En plus, dans une image l'information est locale : il n'y a pas trop d'intérêt de considérer $(1, 0, ...0, -1).x$ qui correspondrait à regarder la différence de valeur entre le pixel en haut à gauche et celui en bas à droite.
La solution à ces deux problèmes est le neurone convolutif (associé au pooling).

3.2.2 La convolution

Le neurone convolutif est un opérateur linéaire local qui prend une image en entrée et produit une image en sortie. On pourra coupler ce neurone convolutif à une activation comme dans le cas d'un neurone classique. Précisément, il prend en entrée une image à K channels (par exemple 3 pour RGB) et il produit en sortie une image 1 channel. Une convolution est paramétré par une petite matrice appelé noyau généralement bien plus petite que l'image. Ce noyau se comporte comme un neurone sur chaque bout de l'image puis tout est regroupé en une nouvelle image. Mathématique l'équation d'une convolution paramétré de noyaux w (et un biais b) de taille $(2\delta_H + 1) \times (2\delta_W + 1)$ prenant une image $H \times W$ à Ch channels est :

$$conv_{w,b}(x) = \left(b + \sum_{\substack{dh \in [-\delta_H, \delta_H] \\ dw \in [-\delta_W, \delta_W] \\ ch \in [1, Ch]}} \begin{matrix} x_{h+dh, w+dw, ch} \\ \times w_{dh, dw, ch} \end{matrix} \right)_{\substack{h \in [1, H] \\ w \in [1, W]}}$$

Précisément, cette équation contient des termes mal défini car $x_{-1,-1,0}$ n'existe pas : il y a des effets de bords qu'on peut gérer par du padding ou en acceptant que l'image de sortie soit légèrement plus petite que l'image d'entrée (voir le TP).

3.2.3 Le pooling

Un autre élément essentiel est le pooling qui permet d'avoir une invariance aux petites déformations. En effet, une image de chat reste une image de chat même si on décale tous les pixels de 1. Il est donc primordial d'introduire des

opérations permettant cette invariance. Typiquement, si on considère un maximum, on peut voir que le maximum ne change pas si on translate les valeurs. C'est l'idée du pooling :

$$pool(x) = \left(\begin{array}{cc} \max & x_{2h+dh,2w+dw,ch} \\ dh \in \{0,1\} & \times w_{dh,dw,ch} \\ dw \in \{0,1\} & \end{array} \right)_{\substack{h \in [1, \frac{H}{2}] \\ w \in [1, \frac{W}{2}]}}$$

Le pooling est une sorte d'activation spécifique à l'aspect spatial qui permet en plus de rendre possible le traitement de l'image : compte tenu des contraintes matériel il quasiment impossible d'avoir des réseaux profond sans pooling. On peut noter que quelques articles ont introduit des notions de pooling dans des perceptron multi couches en groupant des dimension. Mais il s'agit dans ce cas d'une amélioration pour gagner en performance, alors que le pooling est indispensable au traitement de l'image.

Ces deux notions sont très simples en image voir la figure 3. Exactement, comme des neurones classiques, on peut former des couches de neurones convolutifs qui produisent des images avec un nombre arbitraire de channels puis un réseau composé d'une succession de couches + activation + pooling. Un réseau de neurone typique est VGG (www.robots.ox.ac.uk/ vgg/research/very_deep/) vous pourrez trouver des illustrations sur internet.

3.3 Un peu d'histoire

Mais d'où vient cette idée de réseau de neurones. D'un coté, il est clair qu'il y a une vrai inspiration du neurones présents dans le cerveau. Mais pas seulement. Si on considère le traitement d'images, la convolution est le point de départ du traitement d'images : voir filtre de gabor ou filtre de sobel sur wikipédia. Seulement, ces filtres là était sélectionné à la main. Ce qui rendait difficile de *faire des filtres qui travaillent sur une image qui a déjà été filtrée*. Des articles travaillant avec des banques de filtres exitent depuis longtemps. L'état de l'art juste avant les réseaux de neurones était probablement les arbres de décision qui justement fonctionne avec une hiérarchie de filtre, chaque filtre étant tirée aléatoirement parmi un ensemble de filtres possibles suivant un certain nombre de critère. Les ondelettes sont exactement un ensemble de filtre multi échelle adapté à linéariser le signal.

Donc l'idée de faire des filtres en cascade avec les premiers filtres étant similaire à des filtres classiques et les filtres suivants étant de plus en plus haut niveau est une idée très ancienne. Ce qui a changé en 2012 c'est que ça a commencé à marcher (voir notamment les filtres de la première couche du réseau AlexNet : on y retrouve notamment des filtres classiques - mais ils ont été appris automatiquement). Principalement, grâce à un essor de la puissance de calcul qui permet d'apprendre de façon brutale toute la hiérarchie de filtres.

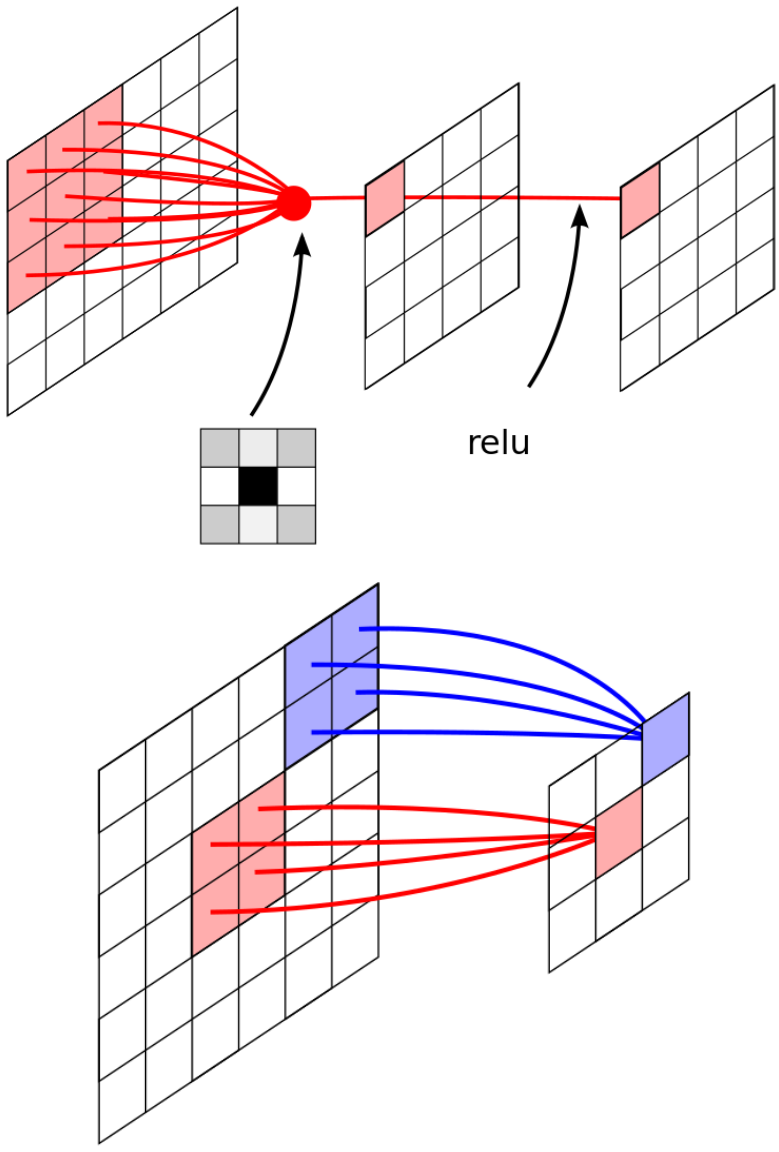

FIGURE 3 – Illustration du neurone convolutif (en haut) et du pooling (en bas)

Typiquement, il est très difficile de faire un TP réaliste d'apprentissage profond pour cette raison : l'apprentissage profond nécessite beaucoup de puissance de calcul (de la puissance bas coût car massivement parallèle mais de la puissance quand même). Dans le cours, on utilisera le service *google colab* car tout est bien installé mais la puissance reste limitée.

Cette idée de filtre hiérarchique est aussi un guide sur la conception du réseau : globalement, trop de neurones n'est pas trop bon, mais, il convient de les répartir entre les couches pour permettre un bon équilibre entre des filtres proches du signal et des filtres haut niveau. Plus la tache est proche du signal, plus les neurones sont dans les couches basses, et inversement (bien qu'il ne faille pas non plus ne pas avoir de neurones dans les premières couches dans le cas contraire).

3.4 Généralité et spécificités

Il est important de bien faire la différence entre ce qui est général aux réseaux de neurones et à toutes les petites choses qui font qu'il y a aujourd'hui au moins 1000 publication scientifique par mois relative à l'apprentissage profond. Rien que dans pytorch (pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity) il y a plein de fonction d'activation utilisable à la place de relu : il y a elu, leaky-relu, sigmoide, arctan, hard sigmoid, hard arctan, prelu, relu6, rrelu, celu, selu, gelu, hard shirk, soft shirk, log sigmoid, soft sign, tanh, tanhshirk ! Et ça ce n'est que celle qui sont suffisamment utilisées pour être intégré, il y a aussi toutes celles testées uniquement dans un article par deux auteurs !

Dans le cours, je parle que du relu (dans le TP il y a du leaky-relu). Mais ce n'est pas pour omettre le reste. C'est juste qu'aucune de ces non linéarité n'est vraiment meilleures que les autres, ça dépend de la base, du réseau associé, de la façon d'optimiser. Ce qui est sur c'est qu'avant on utilisait plutôt des sigmoid parce que c'est régulier et qu'aujourd'hui presque tout le monde utilise des relu parce que ça se calcule vite. Mais il n'y a pas à mon sens d'intérêt à vous détailler toutes ces fonctions. Si vous voulez creuser plus, il faut devenir chercheur en apprentissage par ordinateur, vous pourrez alors développer vos propres intuitions sur quelle est la bonne fonction d'activation étant donné un contexte.

3.5 Bilan

L'objectif est de disposer d'un ordinateur qui peut calculer la fonction y que l'on ne sait pas définir mais qu'un humain peut évaluer. On tire des données $x_1, ..., x_N$ selon une probabilité P et on utilise un humain pour connaître $y(x_1), ..., y(x_N)$ et ainsi avoir une base d'apprentissage.

Avec cette base on peut construire un modèle f.

Ce qui marche le mieux aujourd'hui c'est d'utiliser pour un réseau de neurones f paramétré par un ensemble de poids w. Si l'entrée est vectorielle, ce sera typiquement un perceptron multi couches, si l'entrée est une image se sera

un réseau convolutif. C'est à dire dans les 2 cas, des opérateurs linéaires combinés avec des non linéarité (par exemple relu au milieu). **L'apprentissage proprement dit consiste à trouver des poids w pertinents.**

La performance de f peut être mesurée sur des données de test $\chi_1, ..., \chi_M$ par $\frac{1}{M} \sum_{m \in \{1,...,M\}} \frac{|\text{signe}(f(x)) - y(x)|}{2}$.

4 Comment optimise-t-on un réseau en pratique

4.1 Optimisation vs Apprentissage

L'étape d'apprentissage a proprement parlé c'est de l'optimisation : on se fixe un critère la fonction de coût qui dépend de la base d'apprentissage, et dont les variables sont les poids w du réseau f, et, on minimise cette fonction de coût. À la fin de l'apprentissage les poids sont fixés, et on utilise le réseau uniquement en inférence pour calculer $f(x, w)$. Attention, seul les poids sont classiquement optimisés la structure du réseau est sélectionnée à la main (bien que quelques papiers sur l'optimisation de la structure existent).

Point très important : la phase d'apprentissage c'est de l'optimisation **mais** ce qui compte ce n'est **pas** de bien optimiser, c'est d'avoir une bonne performance en test (qu'on a pas au moment de l'apprentissage). Donc, même si ce qu'on présente après ne vous convient pas en terme d'optimisation parce que vous pensez pouvoir faire mieux, gardez à l'esprit qu'une meilleure optimisation n'est pas forcément meilleure en test, et, donc ne sera pas utilisé si en plus plus lente et etc.

4.2 Optimisation parfaite

Cela dit, imaginons qu'on ait toute la puissance qu'on veut. Nous avons donc $x_1, ..., x_N$ et $y(x_1), ..., y(x_N)$ et on veut trouver w tel que f paramétré par w soit pertinente en test. Mais, le test on le connaît pas (le fait qu'on ait toute la puissance qu'on veut n'y change rien). Donc ce qu'on va faire c'est au moins essayer d'être bon sur la base d'apprentissage (plus éventuellement des termes de régularités dont on pense qu'ils augmentent la performance en test, vous verrez cela durant le cours SVM et PAC).

Par exemple, notre problème d'optimisation pourrait être : **Existe-t-il w tel que**

$$\forall n \in \{1, ..., N\},\ \text{signe}(f(x_n, w)) = y(x_n)$$

Remarquons tout d'abord qu'on peut écrire ce problème

$$\exists ? w\ /\ \forall n \in \{1, ..., N\},\ y(x_n) f(x_n, w) > 0$$

ou encore en introduisant $h(x) = 0$ *si* $x > 0$ *et* 1 *sinon*

$$\exists ? w\ /\ \sum_{n \in \{1,...,N\}} h(y(x_n) f(x_n, w)) = 0$$

Est ce qu'on sait résoudre ce problème ? Je n'ai pas vraiment de réponse à cette question pour un réseau classique car l'optimisation fait apparaître des produits de variable et pourrait être indécidable. Ce qui est sur c'est que si on restreint toutes les couches sauf la première à avoir des poids dans $\{-1, 0, 1\}$ alors le problème n'est plus qu'un programme linéaire en nombre entier (je présuppose que vous avez déjà eu des cours de recherche opérationnel, ce n'est de toute façon que pour votre culture) ! En effet, on peut remarquer que $\lambda relu(w.x) = \text{signe}(\lambda) relu(|\lambda| w.x)$ donc la contrainte que les neurones hors première couche ont des poids dans $\{-1, 0, 1\}$ ne restreint pas tant la généralité car on peut propager leur poids vers les premières couches du réseaux (mais ça impose de de dupliquer la première couche donc ce n'est pas vraiment le même réseau).

À ce moment, là on peut juste introduire une variable binaire pour chaque relu/donnée qui code sont état sur cette donnée : $relu(a) = b$ peut se coder par les contraintes suivantes : $b \leq a + M\alpha$, $b \geq a$, $b \geq 0$, $b \leq M(1 - \alpha)$ et $\alpha \in \{0, 1\}$ avec M un grand nombre.

Donc étant donnée $x_1, ..., x_N$ et $y(x_1), ..., y(x_N)$ et un réseau f dont on contraint toutes les couches sauf la première à avec des poids dans $\{-1, 0, 1\}$, on peut écrire un programme formelle PLNE qui encode

$$\exists ? w \ / \sum_{n \in \{1,...,N\}} h(y(x_n) f(x_n, w)) = 0$$

Est ce que ça marche en pratique : **Non, la taille des problèmes qu'on traite en vision par ordinateur est 10000 fois supérieur à ce qu'on sait traiter en PLNE !**

Donc, on ne va pas chercher à résoudre exactement notre problème : on va approximer la fonction h et utiliser une descente de gradient pour trouver des poids pas trop stupides. Ce n'est pas grave : ce qui compte de toute façon c'est la performance en test.

4.3 Descente de gradient

4.3.1 Descente de gradient simple

Si F est une fonction dérivable de $\mathbb{R}^D$ dans $\mathbb{R}$ alors

$$\forall u, h \in \mathbb{R}^D, \ F(u + h) = F(u) + \nabla F_u | h + o(h)$$

avec $ho(h) \underset{h \to 0}{\to} 0$ (notation petit o classique)

Donc si $\nabla F_u \neq 0$ alors il existe $\lambda > 0$ tel que $F(u - \lambda \nabla F_u) < F(u)$

Algorithmiquement la descente de gradient correspond à

input : F, u_0

1. $u = u_0$
2. calculer ∇F_u
3. si $\nabla F_u \approx 0$ ou early stopping alors sortir
4. $\lambda = 1$
5. tant que $F(u - \lambda \nabla F_u) \geq F(u)$ faire $\lambda = 0.5\lambda$
6. $u = u - \lambda \nabla F_u$
7. go to 2

4.3.2 Fonction de coût

On peut donc appliquer cet algorithme à $\sum_{n\in\{1,...,N\}} h(y(x_n)f(x_n,w))$ non pas pour espérer obtenir 0 mais pour obtenir des poids pas stupides.

Cela dit la fonction $h(x) = 0$ *si* $x > 0$ *et* 1 *sinon* ne permet pas d'utiliser la descente de gradient puisqu'elle n'est pas dérivable (on n'as pas vraiment besoin que la fonction soit vraiment dérivable mais h elle a une dérivée nulle partout sans être nulle, c'est vraiment un problème).

Donc, on va devoir utiliser autre chose : ce qu'on veut c'est une fonction positive qui est de plus en plus petite quand $y(x)f(x,w)$ est grand. La fonction utilisée aujourd'hui est la cross entropy. Cependant, elle met en jeu des logarithme. Donc dans le TP nous utiliserons une fonction plus simple la hinge loss (très utilisé il y a 10 ans dans les SVM) : $g(x) = relu(1-x)$ (voir illustration ici : rohanvarma.me/Loss-Functions)

Donc on va chercher à minimiser $\sum_{n\in\{1,...,N\}} g(y(x_n)f(x_n,w))$ ce qui devrait conduire à ce que les $g(y(x_n)f(x_n,w))$ soient positifs et donc à ce que les $y(x_n)f(x_n,w)$ soient grands (c'est ce qu'on veut).

4.3.3 Descente de gradient stochastique

Malheureusement, quand N est grand, c'est trop long juste de calculer $\sum_{n\in\{1,...,N\}} g(y(x_n)f(x_n,w))$!

Heureusement, on a la descente de gradient stochastique : cela consiste à essayer de minimiser $\sum_{n=1}^{N} q_n(u)$ en appliquant la descente de gradient à des sous sommes de cette somme (soit élément par élément soit par paquet). À noter la descente de gradient stochastique converge en espérance vers le minimum dans le cas convexe ce qui ne s'applique évidemment pas ici, mais indépendamment, en pratique ça marche !

Au final, la méthode de l'état de l'art pour optimiser un réseau de neurones c'est donc :

input : $x_1, y_1, ..., x_n, y_n$, θ_0

1. $\theta = \theta_0$
2. $iter = 0$
3. tirer i au hasard dans 1,...,n
4. $partial\_loss = relu(1 - y_i reseau_\theta(x_i))$
5. calculer $\nabla partial\_loss_\theta$
6. $\theta = \theta - \lambda_{iter} \nabla partial\_loss_\theta$
7. $iter = iter + 1$
8. si condition d'arrêt alors sortir
9. go to 3

avec θ l'ensemble de tous les poids du réseau !

4.4 Calcul du gradient

Il ne reste qu'à savoir calculer les gradients et on aura tous les ingrédients (il ne manquera que les subtilités des 1000 articles par mois). Malheureusement, cette partie est pénible. Elle est pénible et inutile en pratique car les moteurs de réseau de neurones pytorch (facebook) et tensorflow (google) permettent de faire ça automatiquement sans savoir ce qu'on fait. Néanmoins, c'est comme les listes en c++, on utilise toujours celle de la librairie standard en pratique mais l'avoir vu une fois en cours ça permet de comprendre la notion de pointeurs. C'est donc mon devoir d'enseignant de vous montrer qu'il n'y a aucune magie et que c'est juste la dérivation d'une fonction composée... Le 2ème TP est même sur ça mais rassurer vous en dehors de ce TP vous utiliserez toujours pytorch !

Donc on veut calculer $\frac{\partial loss_i}{\partial w_{t,i,j}}$ avec $loss_i = relu(1 - y_i f(x_i, w))$ et $w_{t,i,j}$ le poids du neurones i de la couche t relié au neurones j de la couche $t-1$.

Ça se fait en 2 étapes : d'abord, on peut remarquer que

$$\frac{\partial loss_i}{\partial w_{t,i,j}} = \frac{\partial loss_i}{\partial x_{t,i}} \frac{\partial x_{t,i}}{\partial w_{t,i,j}} = \frac{\partial loss_i}{\partial x_{t,i}} relu(x_{t-1,j})$$

avec $x_{t,i}$ le neurone t de la couche i et $x_{t-1,j}$ le neurone $t-1$ de la couche j.

En effet $x_{t,i} = b + \sum_j w_{t,i,j} relu(x_{t-1,j}$ c'est la définition d'un neurone voir la section 3.1.1 pour rappel.

Ensuite, si $p(x) = u(v(x), w(x))$ alors $\frac{\partial p}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial w}\frac{\partial w}{\partial x}$ **ATTENTION :** ça n'a rien à voir avec le fait que les réseaux de neurones soit des opérateurs linéaires c'est à dire des sommes - là c'est un fait général qui est due au fait que la dérivée est un opérateur linéaire mais c'est vrai pour toutes fonctions p, u, v, w !

Spécifiquement appliquer à nos neurones, la dérivée par rapport au neurone $x_{t,j}$ peut donc s'exprimer vis à vis des neurones $x_{t+1,i}$ puisqu'un neurone ne dépend que de la couche d'avant et n'influe que sur la couche d'après ! Donc

$$\frac{\partial loss_i}{\partial x_{t,j}} = \sum_i \frac{\partial loss_i}{\partial x_{t+1,i}} \frac{\partial x_{t+1,i}}{\partial x_{t,j}} = \sum_i \frac{\partial loss_i}{\partial x_{t+1,i}} w_{t+1,i,j} relu'(x_{t,j})$$

En combinant le passage de w vers x et la récurrence sur x, il est possible de calculer tous les $\frac{\partial loss_i}{\partial w_{t,i,j}}$ en 1 passage de la donnée dans le réseau. Précisément, il faut 1 passage pour calculer les valeurs (*forward*) et un passage dans l'autre sens pour les dérivées (*backward*).

5 Bilan global

L'objectif est de disposer d'un ordinateur qui peut calculer la fonction y que l'on ne sait pas définir mais qu'un humain peut évaluer.

On tire des données $x_1, ..., x_N$ selon une probabilité P et on utilise un humain pour connaître $y(x_1), ..., y(x_N)$ et ainsi avoir une base d'apprentissage.

Indépendamment, on choisit une architecture de réseau de neurones f (guidée par l'état de l'art).

Ensuite, on minimise la fonction de coût $\sum_{n\in\{1,...,N\}} g(y(x_n)f(x_n,w))$ avec par exemple $g(t) = relu(1-t)$, ce qui permet d'optimiser les poids de f sur la base d'apprentissage (ce qui tend à minimiser l'erreur sur la base d'apprentissage).

Une fois l'apprentissage effectué on dispose de poids w bon sur la base d'apprentissage. Il faut alors qualifier l'approximation en mesurant la performance de f sur des données de test $\chi_1, ..., \chi_M$ mesurée par $\frac{1}{M}\sum_{m\in\{1,...,M\}} \frac{|\text{signe}(f(x))-y(x)|}{2}$.

Au vue du niveau d'erreur, de la criticité du problème (et du coût humain pour calculer y), c'est à la société de décider ou non de l'applicabilité réelle de l'approximation f créée.