Étudiants ingénieurs en aérospatial

Mémoire de 3e année

Théorie et Benchmark des méthodes de descente en vue d'une application au machine learning

Auteurs :
M. Audet Yoann
M. Chandon Clément
M. De Claverie Chris
M. Huynh Julien

Encadrant :
Pr. Peschard Cédric

Version 1.0 du
5 juin 2019

Remerciements

Table des matières

Chapitre 1

Introduction

L'optimisation différentiable est une branche assez spécifique des mathématiques qui est importante dans de nombreux domaines où l'on souhaite maximiser ou minimiser des fonctions, c'est-à-dire, résoudre un problème d'optimisation.

Ces problèmes ont un champ d'application très large comme par exemple en algèbre linéaire où l'on souhaite résoudre des problèmes du type $Ax = b$. Mais également dans des problèmes plus concrets et physiques comme la conception optimale de systèmes ou dans la recherche opérationnelle. Ainsi, il est essentiel de trouver des méthodes efficaces pour résoudre ces problèmes. Il existe une grande variété d'algorithmes permettant de minimiser (ou maximiser puisque cela revient au même), une fonction différentiable donnée.

Si les algorithmes les plus communs sont plus souvent des méthodes de descente de gradient, d'autres algorithmes peuvent être plus efficaces dans des conditions spécifiques, c'est le cas des algorithmes de Newton et de Quasi-Newton (L-BFGS par exemple).

Nous nous intéresserons tout d'abord à la théorie justifiant ces méthodes avant de les appliquer à un domaine pour lequel l'intérêt ressurgit depuis quelques années : le Machine Learning.

Chapitre 2

Développement théorique des méthodes de descente

Le but de ce rapport est d'étudier les différentes possibilités afin de minimiser une fonction. Le problème se formule de la manière suivante :

$$\underset{x}{\operatorname{argmin}}\, f(x) \tag{2.1}$$

Nous utiliserons des méthodes de descente afin de résoudre ce type de problème mais avant tout, il est nécessaire de définir l'intérêt de résoudre un tel type de problème puis de l'encadrer.

2.1 Retour sur les méthodes de descente

2.1.1 Motivation de l'étude et de l'intérêt

Lorsque nous traitons d'optimisation, la fonction dont nous cherchons généralement l'argmin, est une fonction objectif. Ces fonctions à minimiser sont présentes dans de nombreux domaines très différents dont l'économie, la mécanique, la production industrielle ou dans la gestion de flux et de réseaux. Les applications sont nombreuses. Il est donc important de pouvoir trouver des méthodes efficaces de résolution de ce problème général.

2.1.2 Encadrement du problème

On cherche donc l'argument x tel que $f(x)$ soit minimal, f étant une fonction deux fois différentiable sur $\mathbb{R}^n$. On se donne deux conditions essentielles car sans celles-ci, l'étude devient beaucoup plus complexe.

On pose donc que la fonction f que l'on cherche à minimiser est une fonction strictement convexe afin d'utiliser les outils de l'analyse convexe. De plus, on suppose que f est coercive sur l'ensemble d'étude R^n. Mathématiquement, ces deux conditions se traduisent par :

$$\forall t \in]0;1[, \forall (x,y) \in \mathbb{R}^2, x \neq y, f(tx + t(1-t)y) < tf(x) + (1-t)f(y)$$

et

$$\lim_{|x| \to +\infty} f(x) = +\infty$$

La coercivité permet d'assurer l'existence d'une solution et la stricte convexité de limiter le nombre de solution à une, au plus. En posant ces deux conditions, notre problème d'optimisation aura une solution unique que l'on notera x^*.

2.1.3 Intérêt d'une fonction $\mathscr{C}^2$ sur $\mathbb{R}^n$

Dans ce mémoire nous nous intéressons uniquement aux fonctions objectif qui sont deux fois différentiables sur $\mathbb{R}^n$, voir de classe $\mathscr{C}^2$.

Cela a son importance puisque dans bons nombres de cas, nous effectuerons un développement de limité sur la fonction f, localement, afin d'obtenir une forme quadratique plus un reste qui dépendra de choix que nous effectuerons (Reste de Young, reste intégral ou reste de Taylor-McLaurin).

Notre problème d'optimisation différentiable se rapprochera donc d'un problème de minimisation quadratique et nous pourrons ainsi trouver un minimiseur local qui, dans le cadre d'une fonction strictement convexe et coercive, sera le minimum global unique recherché.

2.1.4 Les méthodes de gradients

Les méthodes de gradients sont des méthodes de descentes basées sur la connaissance du gradient de la fonction objectif. Nous verrons trois méthodes de descente de gradient classiques qui sont :

1. Le gradient à pas fixe
2. Le gradient à pas optimal
3. Le gradient conjugué

Principes

L'idée derrière ces méthodes est de trouver $\text{argmin}_x f(x)$ à partir d'un point initial puis d'en choisir une direction et un pas de descente. L'efficacité de l'algorithme sera

grandement influencée par le pas de descente. Le choix de ce paramètre donne naissance aux trois méthodes que nous étudions.

Méthode du gradient à pas fixe

La méthode du gradient à pas fixe est la plus simple à mettre en place. Cependant, elle peut être dangereuse puisque l'on peut se dire qu'un pas très grand permettra d'atteindre la solution recherché plus rapidement. Cela induit en erreur puisque dans un tel cas (grand pas), il est possible d'osciller autour de la solution sans jamais l'atteindre à un seuil de précision convenable. L'algorithme est le suivant :

Algorithm 1 Méthode du gradient à pas fixe

procedure MÉTHODE DU GRADIENT À PAS FIXE(x_0, n_{Max}, ρ)
  Soit $i = 1$
  while $||\nabla f(x^{(i)})|| > tol$ & $i < n_{Max}$ **do**
    $d^{(i)} = -\nabla f(x^{(i)})$;
    $x^{(i+1)} = x^{(i)} + \rho d^{(i)}$;
    $i+ = 1$;
  end while
end procedure

Méthode du gradient à pas optimal

Une manière d'améliorer les performances de l'algorithme serait d'y faire une recherche de pas optimal à chaque tour.

Algorithm 2 Méthode du gradient à pas optimal

procedure MÉTHODE DU GRADIENT À PAS OPTIMAL(x_0, n_{Max}, ρ_0)
  Soit $i = 1$
  while $||\nabla f(x^{(i)})|| > tol$ & $i < n_{Max}$ **do**
    $d^{(i)} = -\nabla f(x^{(i)})$;
    $\rho^{(i)} = recherchedu Pas(x^{(i)}, d^{(i)}, \rho_0, tolR)$
    $x^{(i+1)} = x^{(i)} + \rho^{(i)} d^{(i)}$;
    $i+ = 1$;
  end while
end procedure

L'algorithme de recherche du pas est :

Algorithm 3 Recherche du pas inexact

procedure RECHERCHE DU PAS INEXACT($x^{(i)}, d^{(i)}, \rho_0, tolR$)
  Soit $j = 1$, $\rho^{(0)} = 10\rho^{(1)}$
  while $||\rho^{(j)} - \rho^{(j-1)}|| > tolR$ & $j < maxIt$ **do**
    $\varphi^{'}(\rho^{(j)}) = d^{(i)^T} \nabla f(x^{(i)} + \rho^{(j)} d^{(i)})$;
    $\varphi^{''}(\rho^{(j)}) = d^{(i)^T} H_f(x^{(i)} + \rho^{(j)} d^{(i)}) d^{(i)}$;
    $\rho^{(j+1)} = \rho^{(j)} - \frac{\varphi^{'}(\rho^{(j)})}{\varphi^{''}(\rho^{(j)})}$
  end while
end procedure

Méthode du gradient conjugué

La méthode du gradient conjugué est la version la plus élaborée de ces trois méthodes de descentes du gradient. Ainsi, nous l'étudierons plus en détail et la justifierons dans la partie suivante grâce aux espaces de Krylov.

2.2 Introduction aux espaces de Krylov

Dans cette partie, nous allons introduire un outil mathématique important qui permet la justification de la méthode du gradient conjugué : les espaces de Krylov.

2.2.1 Définition des espaces de Krylov

Afin d'introduire les espaces de Krylov, nous allons prendre un exemple simple : celui d'un système linéaire. On cherche donc x qui vérifie l'équation :

$$Ax = b \tag{2.2}$$

On définit le vecteur résidu à l'itération k comme étant :

$$r^k \triangleq b - Ax^k = -A(-A^{-1}b + x^k) = -A(-x^* + x^k) \tag{2.3}$$

Où x^* est la solution réelle du système. Alors, nous pouvons écrire la solution au rang k+1, comme celle au rang k plus le résidu :

$$\begin{aligned} x^{k+1} &= x^k + r^k && (2.4)\\ \Leftrightarrow x^{k+1} - x^* &= x^k - x^* + r^k && (2.5)\\ \Leftrightarrow -A(x^{k+1} - x^*) &= -A(x^k - x^*) - Ar^k && (2.6)\\ \Leftrightarrow r^{k+1} &= r^k - Ar^k && (2.7) \end{aligned}$$

Dans cette dernière équation récursive, nous pouvons voir que r^{k+1} est une combinaison linéaire des vecteurs r^k précédents. Ainsi :

$$r^k \in Vect\{r^0, Ar^0, ..., A^k r^0\} \tag{2.8}$$

Cela implique directement :

$$x^k - x^0 = \sum_{i=0}^{k-1} r^i \tag{2.9}$$

Donc il vient que :

$$x^k \in x^0 + Vect\{r^0, Ar^0, ..., A^k r^0\} \tag{2.10}$$

Où $Vect\{r^0, Ar^0, ..., A^k r^0\}$ est le k-ième espace de Krylov généré par A à partir de r^0 noté $\mathscr{K}_k(A, r^0)$

2.2.2 Quelques propriétés

Maintenant que nous avons une définition de ces espaces, nous allons montrer plusieurs propriétés pour ensuite construire l'algorithme et l'implémenter. Le premier fait remarquable des espaces de Krylov est que par construction, nous avons :

$$\mathscr{K}_{k-1}(A, r^0) \in \mathscr{K}_k(A, r^0) \tag{2.11}$$

Ensuite, il est important de montrer que la solution que l'on cherche appartient à cet espace. Nous reprenons donc notre système linéaire possédant une unique solution :

$$Ax = b \tag{2.12}$$

D'après la définition du problème, la matrice A est inversible. Nous supposons que l'on a le polynôme caractéristique de A :

$$P(\lambda) = \sum_{j=0}^{n} \alpha_j t^j \Rightarrow P(0) = \alpha_0 = det(A) \neq 0 \tag{2.13}$$

Par le théorème de Cayley-Hamilton [1], nous pouvons obtenir la valeur de A^{-1} :

$$P(A) = \alpha_0 I + \alpha_1 A + ... + \alpha_n A^n = 0 \tag{2.14}$$
$$\alpha_0 A^{-1} A + \alpha_1 A + ... + \alpha_n A^n = 0 \tag{2.15}$$
$$(\alpha_0 A^{-1} + \alpha_1 + ... + \alpha_n A^{n-1}) A = 0 \tag{2.16}$$
$$\alpha_0 A^{-1} + \alpha_1 + ... + \alpha_n A^{n-1} = 0 \tag{2.17}$$

1. http://www.logique.jussieu.fr/~alp/Cayley_Hamilton.pdf

Ce qui donne finalement :

$$A^{-1} = -\frac{1}{\alpha_0} \times \sum_{j=0}^{n-1} \alpha_{j+1} A^j \tag{2.18}$$

Or, la solution du problème 2.12 est : $x^* = A^{-1}b$. Ce qui peut s'écrire de la façon suivante :

$$x^* = -\frac{1}{\alpha_0} \sum_{j=0}^{n-1} \alpha_{j+1} A^j b \tag{2.19}$$

Ce vecteur appartient clairement à l'espace de Krylov défini par :

$$x^* \in \mathscr{K}(A, b) \tag{2.20}$$

2.2.3 Un premier algorithme : GMRES

Un des algorithmes utilisant les espaces de Krylov est l'algorithme GMRES qui se trouve ci-dessous en python. Cette méthode repose sur les espaces de Krylov. En effet, dans cet algorithme nous choisissons $x_k \in \mathscr{K}_k(A, b)$ tel que l'on a $||b - Ax_k||_2$ qui est minimal. Nous nous reposons sur une méthode de projection. L'algorithme utilise aussi sur le procédé d'Arnoldi [2] qui nous permet de créer une base orthonormée sur le sous-espace de Krylov. Dans la suite de ce chapitre, nous allons étudier ces composantes permettant d'arriver à l'écriture de notre algorithme.

Généralité

Nous avons le système linéaire suivant qui possède une unique solution x^* :

$$Ax = b \tag{2.21}$$

Le résidu initial est :

$$r_0 = b - Ax^{(0)} \tag{2.22}$$

L'algorithme GMRES repose sur une méthode de projection. On écrit donc le système sous sa forme résiduelle pour faire apparaître une projection :

$$< b - Ax^*, v >= 0, \forall v \in \mathbb{R}^n \Leftrightarrow b - Ax^* \perp \mathbb{R}^n \tag{2.23}$$

Le but est maintenant de construire des solutions par itérations. On se place dans un sous-espace $\mathbf{K}$ et on essaie de trouver dans ce sous-espace x tel que :

$$b - Ax \perp \mathbf{K} \Rightarrow < Au, v >=< b, v >, v \in \mathbb{K} \tag{2.24}$$

2. https://docplayer.fr/215470-Calcul-matriciel-et-systemes-lineaires.html (page 36)

On généralise cette méthode grâce à deux sous-espaces : $\mathbf{K}$ et $\mathbb{L}$. On cherche $x \in u_0 + \mathbf{K}$ avec :

$$b - Ax \perp \mathbb{L} \Rightarrow < Au, v >=< b, v >, v \in \mathbb{L} \tag{2.25}$$

C'est la condition de Petrov-Galerkin.

Sous la forme matricielle, nous allons créer deux matrices V et W, de taille $n \times m$, définies par les vecteurs des bases de $\mathbf{K}$ et $\mathbb{L}$:

$$V = (V_1\ V_2\ ...\ V_m) \tag{2.26}$$
$$W = (W_1\ W_2\ ...\ W_m) \tag{2.27}$$

Soit $(y, z) \in (\mathbb{R}^m)^2$. Nous pouvons définir $v \in \mathbf{K}$ et $w \in \mathbb{L}$:

$$v = Vy \tag{2.28}$$
$$w = Wz \tag{2.29}$$

Il nous faut maintenant trouver la solution de $x = x_0 + Vy \in \mathbf{K}$. Alors, grâce à 2.23 et 2.25, nous écrivons :

$$< Av, w > = < r0, w > \forall w \in L \tag{2.30}$$
$$< AVy, Wz > = < r0, Wz > \forall z \in \mathbb{R}^m \tag{2.31}$$
$$Avy = r0 \tag{2.32}$$
$$y = (W^tAV)^{-1}W^tr^0 \tag{2.33}$$

Nous obtenons donc la solution :

$$x = x_0 + V(W^tAV)^{-1}W^tr^0 \tag{2.34}$$

Nous pouvons donc maintenant expliquer ce que notre méthode de projection va faire. Il faut savoir qu'une méthode de projection est itérative et va à chaque itération faire deux choses :

— construire les sous-espaces K et L

— chercher la nouvelle itération dans $u + K$ par le procédé exposé ci-dessus.

Voici donc le principe général des méthodes de projection :

1. Choisir $\mathbf{K}^m$ et $\mathbb{L}^m$: deux sous-espaces de $\mathbb{R}^n$ de même dimension m
2. Construire V_m et W_m
3. Calculer le résidu : $r = b - Ax^{(k)}$
4. Résoudre le système $y = (W_m^tAV_m)^{-1}W_m^tr$
5. Créer la nouvelle solution : $x = x + V_my$

Pour créer le sous-espace L, il y a deux méthodes :

— $L = K$ donne lieu à la méthode du gradient conjugué (expliqué ci-dessous)

— $L = AK$ donne lieu à la méthode GMRES.

La méthode du gradient conjugué ayant été vue en cours, nous avons choisi la deuxième méthode de génération de L. Dans les deux cas cela revient à résoudre le problème des moindres carrés :

$$\begin{cases} \text{On cherche } x^{(k)} \in x_0 + \mathbf{K}_m \text{ tel que :} \\ \underset{v \in \mathbf{K}_m}{min} \ \frac{1}{2}\|A\,(x_0 + v) - b\|_2^2 \end{cases} \tag{2.35}$$

La démonstration liant les deux problèmes ne sera pas réalisée ici. [3]

Explication de l'algorithme GMRES

Cet algorithme se base dans un premier temps sur les itérations d'Arnoldi. Ici nous ne présenterons que la version qui est moins sensible aux erreurs d'arrondi car notre but est de justifier l'implémentation numérique de l'algorithme. Ce procédé d'Arnoldi a pour but de créer une base orthonormée du sous-espace de Krylov. Le procédé se résume par :

1. $v_1 = \frac{r_0}{||r_0||}$
2. Pour j = 1, ..., m ; on calcule : $w = Avj$
3. Pour i = 1, ..., j ; on calcule : $w = w - (w, v_i)v_i$
4. $\bar{v}_{j+1} = \frac{w}{||w||}$

Nous introduisons alors un objet : $\overline{H}_m = (h_{ij})$ la matrice de Hessenberg :

$$h_{ij} = \begin{cases} < Av_j, v_i >, \ i \leq j \\ ||v_{j+1}||, \ i = j + 1 \\ 0, \ sinon \end{cases} \tag{2.36}$$

Cet objet permet d'écrire de façon matricielle notre procédé car :

$$AV_m = V_{m+1}\overline{H}_m \tag{2.37}$$

3. Elle se trouve ici : `https://www2.mat.ulaval.ca/fileadmin/Cours/MAT-17992/notes_gmres.pdf`

et :

$$\bar{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j}v_i \tag{2.38}$$

$$\Leftrightarrow Av_j = \bar{v}_{j+1} + \sum_{i=1}^{j} h_{i,j}v_i \tag{2.39}$$

$$\Leftrightarrow Av_j = h_{j+1,j}v_{j+1} + \sum_{i=1}^{j} h_{i,j}v_i \tag{2.40}$$

Par ce procédé, nous arrivons à construire une base orthonormée de l'espace de Krylov.

Maintenant que nous avons cette base, nous pouvons résoudre le problème des moindres carrés exprimé ci-dessus (cf. équation 2.35). Nous obtenons les propriétés suivantes (qui ne seront pas démontrées) :

— $||r_{m+1}|| \leq ||r_m||$

— L'algorithme converge en n itérations au maximum car $\mathscr{K}_m(A, r0) = \mathbb{R}^n$ [4]

Voici le code et le pseudo-code de l'algorithme :

```
def GMRES(A, b, espilon):
        max_iter = A.shape[0] #Number of maxiter
        mat_q = np.zeros((max_iter, max_iter + 1))
        mat_h = np.zeros((max_iter + 1, max_iter))
        norm_b = np.linalg.norm(b)
        be1 = np.zeros(max_iter + 1)
        be1[0] = norm_b
        mat_q[:, 0] = 1 / np.linalg.norm(b) * b.T # On définit ici que
        ↪  l'on a forcément x0 = 0 (r0 = b)
        for j in range(max_iter):
                mat_q[:, j+1] = A @ mat_q[:, j]
                for i in range(j+1):
                        mat_h[i, j] = mat_q[:, i] @ mat_q[:, j + 1]
                        mat_q[:, j+1] -= mat_h[i, j] * mat_q[:, i]
                        mat_h[j + 1, j] = np.linalg.norm(mat_q[:, j + 1])
                        mat_q[:, j + 1] /= mat_h[j + 1, j]
                y = np.linalg.lstsq(mat_h, be1, rcond=None)[0]
                residue = np.linalg.norm(y) / norm_b
                if residue < espilon:
```

4. La démonstration se trouve ici : https://www.math.kth.se/na/SF2524/matber15/gmres.pdf

```
19                          return mat_q[:max_iter, :max_iter] @ y, residue
20          return mat_q[:max_iter, :max_iter] @ y
```

2.3 L'approche par les espaces de Krylov du gradient conjugué

2.3.1 Définition du gradient conjugué

Grâce aux espaces de krylov, nous sommes capable d'améliorer les algorithmes de gradient qui sont présentés ci-dessus. Nous partons donc d'un algorithme du gradient basique :

$$\begin{cases} x_0 \in \mathbb{R} \text{ le choix initial} \\ x_{k+1} = x_k + \alpha_k(b - Ax_k) \end{cases} \tag{2.41}$$

Nous définissons alors le vecteur résidu $r_k = b - Ax_k$ appartenant à l'espace de Krylov d'ordre k et définit par le résidu à l'origine : r_0. De ce fait, il vient que x_{k+1} appartient à l'espace affine composé par le point x_0 et le k$^{\text{ieme}}$ espace de krylov : $\mathscr{K}_k$. Afin de simplifier notre exposé, nous supposons que nous avons affaire à des matrices symétriques définies positives.

Dans la méthode du gradient conjugué, nous n'allons pas choisir la définition comme dans les autres méthodes de gradient mais nous avons d'autres critères qui sont plus intéressant :

— Le premier repose sur le principe d'orthogonalisation :

$$\exists x_{k+1} \in [x_0 + \mathscr{K}_k], r_{k+1} \perp \mathscr{K}_k \tag{2.42}$$

— Le principe de minimisation :

$$\exists x_{k+1} \in [x_0 + \mathscr{K}_k], \frac{1}{2} < Ax, x > - < b, x > \text{ est minimal} \tag{2.43}$$

Dans le cas d'une matrice symétrique définie positive, les deux choix ci-dessus donnent la même solution x_{k+1} d'où le choix de A symétrique définie positive.
Définissons ensuite le vecteur direction :

$$d_k = x_{k+1} - x_k \tag{2.44}$$

De part la construction de d, il est possible de déduire quelques propriétés :

— L'espace de de Krylov est une combinaison linéaire des vecteurs r_k et d_k

$$\mathscr{K}_k = vectr_0, ..., r_k = vectd_0, ..., d_k \tag{2.45}$$

— Tous les vecteurs de la suite sont orthogonaux :

$$\forall 0 \leq l < k \leq n-1, < r_k, r_l >= 0 \tag{2.46}$$

— Les vecteurs de la suite (d_k) sont conjugué selon le produit vectoriel défini par A :

$$\forall 0 \leq l < k \leq n-1, < Ad_k, d_l >= 0 \tag{2.47}$$

C'est grâce à cette dernière propriété que le gradient conjugué est appelé ainsi.

2.3.2 Propriétés de l'implémentation du gradient conjugué

Le gradient conjugué est aujourd'hui très utilisé dans différents domaines car c'est un algorithmes possédants des propriétés très intéressantes. La première d'entre-elles concerne le calcul des vecteurs de direction. En effet, nous avons vu plus haut, lors de la construction de la méthode, que chaque vecteur est un conjugué de chacun des vecteurs précédents. C'est un propriété important lors de l'implémentation logiciel de la technique puisque pour calculer le vecteur d_k nous n'avons besoin que de d_{k-1}.

Pour toutes méthodes numériques, il nous est important de montrer son utilité en étudiant la convergence : une méthode qui diverge est inutilisable et une méthode qui converge très lentement n'est guère plus utile. C'est pour cela que dans la suite de cette partie nous nous intéressons aux taux de convergence de notre gradient conjugué.

Tout d'abord, nous rappelons que le gradient conjugué converge en n itération au maximum. En effet, la solution de notre problème appartient à l'espace de Krylov dans lequel nous cherchons la solution. La convergence n'est donc pas un problème. Ce qui peut poser des problèmes c'est la vitesse à laquelle nous allons vers cette solution. Pour cela, nous partons de l'expression de x_{K+1} :

$$\begin{aligned} x_{k+1} &= x_0 + \alpha_0 p_0 + ... + \alpha_k p_k & (2.48)\\ &= x_0 + \gamma_0 r_0 + \gamma_1 A r_0 + ... + \gamma_k A^k r_0 & (2.49) \end{aligned}$$

Pour trouver ces formules, nous appliquons juste la définition du gradient : nous partons d'un point x_0 et à chaque itération k nous avançons dans la direction p_k avec un pas de α_k.

Dans cette expression on reconnait un polynôme dépendant de A :

$$P_k(A) = \gamma_0 I + \gamma_1 A + ... + \gamma_k A^k \tag{2.50}$$

Ce qui simplifie l'expressions de x_{k+1} en :

$$x_{k+1} = x_0 + P_(A) r_0 \tag{2.51}$$

En utilisant la norme définit par le produit scalaire avec la matrice A, nous avons :

$$\frac{1}{2}||x - x*||_A^2 = \frac{1}{2}(x - x^*)^T A(x - x^*) \quad (2.52)$$

$$= \frac{1}{2}(x^T Ax + x^{*T} Ax^* - x^T Ax^* - x^{*T} Ax) \quad (2.53)$$

Or par définition, A est symétrique, $b = Ax^*$ et en utilisant la propriété de commutation du produit scalaire, nous avons :

$$\frac{1}{2}||x - x*||_A^2 = f(x) - f(x^*) \quad (2.54)$$

avec $f(x) = \frac{1}{2}x^T Ax - b^T x$ À chaque nouvelle itération nous allons donc minimiser cette expression pour tout x habitant dans l'espace de Krylov :

$$\min_{P_k} ||x_0 + P_k(A)r_0 - x^*||_A \quad (2.55)$$

En introduisant cette minimisation dans l'expression du résidue et le fait que $r_0 = A(x_0 - x^*)$, il vient :

$$x_{k+1} - x^* = x_0 + P_k^*(A)r_0 - x^* = [I + P_k^*(A)A](x_0 - x^*) \quad (2.56)$$

On décompose A grâce à ses valeurs propres :

$$A = \sum_{i=1}^{n} \lambda_i v_i v_i^T \quad (2.57)$$

Et comme les vecteurs propres v_i de A décrivent $\mathbb{R}$:

$$x_0 - x^* = \sum_{i=1}^{n} \xi_i v_i \quad (2.58)$$

Nous introduisons ici une propriété de $P_k(A)$ du fait de sa construction :

$$P_k(A)v_i = P_k(\lambda_i)v_i \forall i = 1, 2, ..., n \quad (2.59)$$

Ainsi, en utilisant les équations ci-dessus :

$$x_{k+1} - x^* = \sum_{i=1}^{n} [1 + \lambda_i P_k^*(\lambda_i)]\xi_i v_i \quad (2.60)$$

On rappelle cette propriété du calcul d'une norme selon une matrice A :

$$||z||_A^2 = \sum_{i=1}^{n} \lambda_i (v_i^T z)^2$$

Ce qui nous permet d'écrire :

$$||x_{k+1} - x^*||_A^2 = min_{P_k} \sum_{i=1}^{n} \lambda_i [1 + \lambda_i P_k(\lambda_i)]^2 \xi^2 \tag{2.61}$$

Le minimum apparaît du fait que le polynôme généré est optimal au regard de la minimisation réalisé dans l'espace de Krylov (Propriété non démontrée ici).

Nous avons donc exprimé la converge de notre algoritme. Le taux de convergence du gradient conjugué est donné par :

$$\min_{P_k} max_{i \leq i \leq n} [1 + \lambda_i P_k(\lambda_i)]^2 \tag{2.62}$$

Nous en déduisons les propriétés suivantes (non démontrées) :

— Si A possède r $<$ n valeurs propres distinctes alors le gradient conjugué va se terminer avec au plus r itérations

— Soit λ_i les valeurs propres de A. Nous avons alors :

$$||x_{k+1} - x^*||_A^2 \leq \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1} \right)^2 ||x_0 - x^*||_A^2 \tag{2.63}$$

Grâce à toutes ses propriétés, on peut montrer que le gradient conjugué est bien meilleurs que les méthodes de descentes classiques. En effet, il converge bien plus vite et nous sommes sûr qu'il atteindra la solution de notre problème.

2.3.3 Algorithme du gradient conjugué

Voici la présentation de l'algorithme du gradient conjugué défini ci-dessus :

Algorithm 4 Méthode du gradient conjugué

Require: x_0

$r_0 = Ax_0 - b$, $p_0 = -r_0$, $k = 0$
while r_0 **do**
  $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k + \alpha_k A p_k$
  $\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$
  $k = k + 1$
end while

Bien entendu, cet algorithme peut être amélioré par plusieurs techniques de préconditionnement. Voici le code python qui correspond à l'implémentation de ce pseudo-code :

```
def gradient_conjugue(A, b, x0, tol, i_max):
        print("Méthode du gradient conjugué")
        i = 1
        x_n = x0
        x = list()
        d = 0
        r = 0
        while(1):
                r_old = r
                r = A @ x_n - b
                if i == 1:
                        d = -r
                else:
                        beta = np.linalg.norm(r) ** 2 /
                         ↪  np.linalg.norm(r_old) ** 2
                        d = -r + beta * d
                        p = r.T @ r / (d.T @ A @ d)
                        x_n = x_n + p * d
                        x.append(x_n)
                i += 1
                if np.linalg.norm(r) < tol or i > i_max:
                        return x_n, x, i
```

2.3.4 Quelques mots sur le gradient conjugué non-linéaire

Le but est ici de voir si l'on peut améliorer l'algorithme ci-dessus pour l'appliquer à des fonctions quadratiques non convexes voir non linéaires. Nous n'allons présenter ici que la méthode de Fletcher-Reeves. Cette méthode consiste en deux petits changement dans l'algorithme du CG.

La première modification consiste à changer le calcul du pas de descente. En effet, comme nous ne sommes plus dans le cas quadratique, il est beaucoup moins facile de trouver le pas de descente. Afin de trouver ce pas optimal, il nous faut minimiser la fonction partielle dans la direction p_k. De plus, le résidue qui était jusqu'à lors le gradient de la fonction quadratique devient ici le gradient de la fonction non-linéaire. Cette partie n'étant qu'une ouverture à l'étude non-linéaire, nous allons uniquement détailler sommairement comment choisir le nouvel α et présenter un algorithme.

Tout d'abord, voici l'algorithme de Fletcher-Reeves :

Algorithm 5 Méthode du gradient conjugué

Require: x_0
1: $f_0 = f(x_0)$ et $\nabla f_0 = \nabla f(x_0)$
2: Définir : $p_0 \triangleq -\nabla f_0$ et $k = 0$
3: **while** $\nabla f_k \neq 0$ **do**
4: Calculer α_k
5: $x_{k+1} = x_k + \alpha_k p_k$
6: Calculer ∇f_{k+1}
7: $\beta_{k+1}^{FR} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$
8: $p_{k+1} = -\nabla f_{k+1} + \beta_{k+1}^{FR} p_k$
9: $k = k + 1$
10: **end while**

2.4 D'autres algorithmes de minimisation

Dans cette partie, nous nous intéressons aux algorithmes de Newton qui sont d'autres algorithmes permettant une nouvelle approche des problèmes de minimisation.

2.4.1 Les algorithmes de Newton

Le but d'un algorithme de Newton est d'approximer notre fonction par le développement de Taylor de celle-ci à l'ordre 2. On peut donc dès lors remarquer que nous aurons besoin de la seconde dérivée de cette fonction (ou de la Hessienne en dimension supérieure). Le

développement de Taylor est :

$$f(x+h) = f(x) + < h, \nabla f(x) > + \frac{1}{2} < h, H_f(x)h > + o(||h||^2) \tag{2.64}$$

Le principe est assez simple : on se trouve au point x et nous allons choisir le point x+h comme prochain point tel que ce nouveau point minimise le développement de Taylor ci-dessus. Ce développement de Taylor s'apparente à une fonction quadratique.

Dans ce type de méthode, nous choisissons notre direction de descente de la manière suivante :

$$d_k = -\nabla^2 f_k^{-1} \nabla f_k \tag{2.65}$$

Bien entendu, dans la réalité, nous ne calculons pas l'inverse de la hessienne mais nous résolvons un système linéaire. Cette méthode peut donc paraître un peu lourde. De plus, nous avons une contrainte sur la matrice Hessienne. En effet, celle-ci doit absolument être définie positive. Cela vient du fait que l'on cherche un minimum et qu'il faut absolument que cette matrice soit non-singulière pour pouvoir trouver une nouvelle direction de descente. Si cette matrice ne l'est pas, il est possible de la modifier par l'approche de Goldfeld, Quandt et Trotter :

$$\hat{H}_k = \frac{H_k + \beta I}{1 + \beta} \tag{2.66}$$

Une seconde approche serait de transformer la hessienne :

$$\hat{H}_k = U^T H_k U + \epsilon \tag{2.67}$$

Avec ϵ une matrice diagonale. Le but est de choisir U telle que $\hat{H}_k$ soit diagonale car cela simplifie les calculs de l'algorithme. Le problème de cette dernière méthode est qu'il faut beaucoup de temps de calcul pour trouver la matrice U.

Nous allons énoncer deux théorèmes sur la convergence de cette méthode :

Théorème 2.1
On suppose que l'on possède un point $\hat{x} \in \mathbb{R}^n$ tel que :

- *C'est un point critique*
- *$H_f(\hat{x})$ est non-singulière*
- *$H_f(\hat{x})$ est lipschitzienne au voisinage de $\hat{x}$*

Il existe $\varepsilon > 0$ tel que la méthode de Newton converge quadratiquement vers $H_f(\hat{x})$

Théorème 2.2
Soit f une fonction C^2 et soit $x_0 \in \mathbb{R}^n$. On suppose que f est convexe et que la matrice

Hessienne possède de bonnes propriétés : $0 < cI_n < H_f(x) < KI_n$. *Alors la fonction f admet un unique minimiseur et la méthode de Newton converge vers celui-ci.*
Si H_f *est M-Lipschitzienne dans le voisinage de ce minimiseur :*

— *Le pas* α_k *converge vers 1,*

— *L'algorithme possède une convergence quadratique.*

Néanmoins, il existe plusieurs problèmes avec l'utilisation informatique de cette méthode de Newton :

— Il faut calculer la matrice Hessienne ce qui peut devenir très vite très coûteux.

— L'inversion de cette hessienne est aussi très coûteuse et très lente en fonction de la taille de celle-ci

— La condition qui oblige la hessienne à être définie positive à chaque étape est très restrictive.

C'est pour cela que nous avons élaboré les méthodes de Quasi-Newton.

2.4.2 Les algorithmes de Quasi-Newton

Les algorithmes de Quasi-Newton se basent sur les algorithmes de Newton. Cependant, l'amélioration vient du fait qu'ils ne nécessitent pas la dérivée seconde de la fonction que l'on cherche à minimiser. En effet, nous allons choisir une direction de descente de la manière suivante :

$$d_k = -B^{-1}\nabla f_k \tag{2.68}$$

Avec B une matrice définie positive qui est recalculée à chaque itération afin d'approximer la valeur de la hessienne. L'algorithme de Quasi-Newton que nous allons étudier ci-dessous est la méthode BFGS. Il existe plusieurs dérivées de cette méthode mais elle repose sur ce même principe de base. La partie théorique étant longue, nous allons dans la suite présenter le pseudo-code de la méthode et quelques-unes de ses propriétés. Avant d'exposer l'algorithme, il faut savoir que dans cet algorithme, nous approximons la hessienne grâce à la formule :

$$H_{k+1} = (I - \rho_k s_k y_k^T)H_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.69}$$

Avec :

— $s_k = x_{k+1} - x_k$

— $y_k = \nabla_{k+1} - \nabla_k$

— $rho_k = \frac{1}{y_k^T s_k}$

Algorithm 6 Quasi-Newton BFGS

Require: x_0, $\varepsilon > 0$ comme tolérance et H_0 une approximation de la hessienne.

1: $k = 0$
2: **while** $||\nabla f_k|| > \varepsilon$ **do**
3: $p_k = -H_k \nabla f_k$
4: $x_{k+1} = x_k + \alpha_k p_k$
5: $s_k = x_{k+1} - x_k$
6: $y_k = \nabla_{k+1} - \nabla_k$
7: Calculer la nouvelle approximation de la Hessienne
8: k = k+1
9: **end while**

Cette algorithme possède un taux de convergence qui est super linéaire. De plus, nous pouvons énoncer le théorème suivant (même si nous n'allons pas le démontrer) :

Théorème 2.3
Soit f une fonction C^2. On se donne $x_0 \in \mathbb{R}^n$ et on suppose $S_0 = x \in \mathbb{R}^n, f(x) \leq f(x_0)$ est convexe et que l'approximation de notre matrice Hessienne ne diverge pas : $0 < cI_n < H_f(x) < KI_n$. Alors, la suite des valeurs (x_k) solutions de notre problème de minimisation converge vers x^*, l'unique minimiseur. De plus, si H_f est Lipschitzienne dans un voisinage de x^*, alors la convergence est super-linéaire

Chapitre 3

Applications des méthodes de descente au deep learning

3.1 Introduction et Motivation

L'intelligence artificielle représente un ensemble de théories et techniques dont le but est de simuler l'intelligence chez la machine. C'est un concept datant des années 1950 mais qui revient à la mode depuis le début du XIXe siècle, à la fois grâce à des avancées technologiques nombreuses mais également dans les mentalités par le biais de films de science fiction. On peut notamment citer la trilogie Matrix, décrivant un monde dystopique où les machines dominent les hommes après avoir dépassé un stade d'intelligence inattendu. Cependant malgré le fait que l'intelligence artificielle a beaucoup évolué ces dernières années, nous sommes encore loin de ce résultat. Actuellement, les appareils que nous utilisons sont généralement limités à un domaine, par exemple la traduction, la navigation... car les systèmes dit intelligents doivent passer par une phase d'apprentissage. Mais cela pourrait bien changer avec le développement du deep learning ou apprentissage profond.

3.1.1 Intelligence artificielle, machine learning et deep learning

Mais qu'est-ce que l'apprentissage profond ? C'est en fait un sous-domaine du machine learning, lui même étant une branche de l'intelligence artificielle. Voici une représentation sous forme d'arbres permettant de visualiser différentes approches de l'IA :

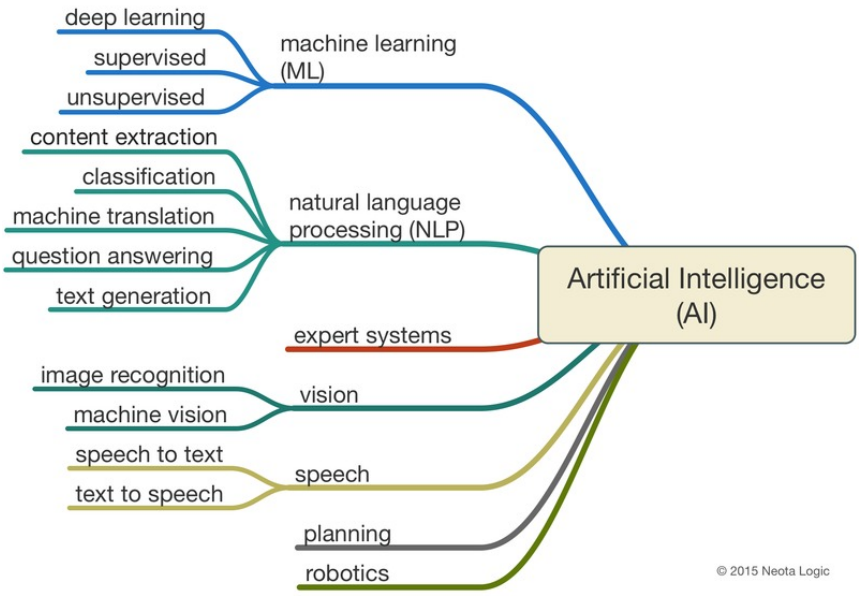

FIGURE 3.1 – Approches de l'IA

Pour mieux comprendre ce que sont machine learning et deep learning commençons par les *systèmes experts*. Ce sont des logiciels très spécialisés dont le but est de reproduire la "pensée" d'un expert dans un domaine. Cela permet de remplacer une personne pour une tâche bien précise. Par exemple, le premier système expert mis au point, Dendral (en 1965) permettait d'identifier les consitituants chimiques d'un matériau. Les systèmes experts reposent sur des algorithmes "simples", constitués de conditions "if... then...", pouvant être assimilés à des arbres de décisions. Ils sont donc utilisables uniquement pour des tâches précises et déterminées à l'avance et ne peuvent sortir de leurs bornes d'application. L'apprentissage de la machine est manuelle, c'est l'homme qui doit implémenter toutes les possibilités lui-même.

Le machine learning apporte une solution à cela. Dorénavant, l'ordinateur apprend par lui-même. L'homme lui apprend à reconnaitre et à reproduire. On dit qu'on "entraine" la machine. Il faut pour cela lui fournir de grands jeux de données, c'est là qu'intevient le domaine du big data et c'est une des raisons pour lesquelles le développement du machine learning s'est accéléré ces dernières années. Grâce aux données d'entraînement, le programme constitue des données statistiques sur lesquelles il s'appuie ensuite pour faire ses prévisions.

Enfin, le deep learning est constitué d'algorithmes permettant au logiciel de s'entrainer lui même. Cette dernière approche est basée sur ce que l'on appelle des "réseaux de neurones" à multiples couches, imitant (plus ou moins) le fonctionnement du cerveau humain. Plus ils reçoivent de données, plus ces réseaux de neurones sont performants.

3.2 Architecture d'un réseau de neurones

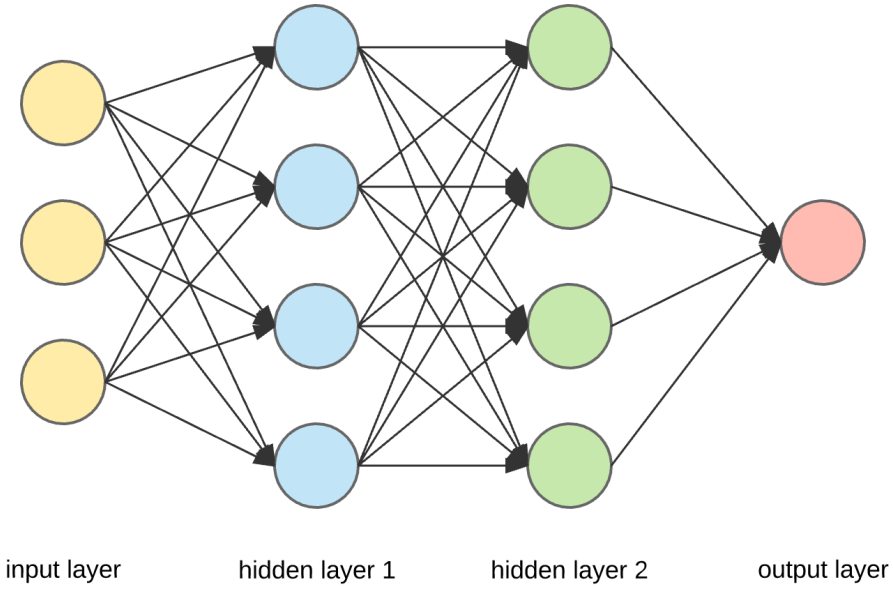

FIGURE 3.2 – Architecture

3.3 Application des algorithmes

Chapitre 4

Conclusion

Table des figures