



Étudiants ingénieurs en aérospatial

Mémoire de 3^e année

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

Auteurs :

M. AUDET Yoann
M. CHANDON Clément
M. DE CLAVERIE Chris
M. HUYNH Julien

Encadrant :

Pr. PESCHARD Cédric

Version 1.0 du
9 juin 2019

Remerciements

Table des matières

1	Introduction	1
2	Développement théorique des méthodes de descente	2
2.1	Retour sur les méthodes de descente	2
2.1.1	Motivation de l'étude et de l'intérêt	2
2.1.2	Encadrement du problème	2
2.1.3	Intérêt d'une fonction \mathcal{C}^2 sur \mathbb{R}^n	3
2.1.4	Les méthodes de gradients	3
2.2	Introduction aux espaces de Krylov	5
2.2.1	Définition des espaces de Krylov	5
2.2.2	Quelques propriétés	6
2.2.3	Un premier algorithme : GMRES	7
2.3	L'approche par les espaces de Krylov du gradient conjugué	11
2.3.1	Définition du gradient conjugué	11
2.3.2	Propriétés de l'implémentation du gradient conjugué	12
2.3.3	Algorithme du gradient conjugué	14
2.3.4	Quelques mots sur le gradient conjugué non-linéaire	16
2.4	D'autres algorithmes de minimisation	17
2.4.1	Les algorithmes de Newton	17
2.4.2	Les algorithmes de Quasi-Newton	18
3	Applications des méthodes de descente au deep learning	20
3.1	Introduction et Motivation	20
3.1.1	Intelligence artificielle, machine learning et deep learning	20
3.2	Architecture d'un réseau de neurones	22
3.2.1	Détail d'un neurone	23
3.2.2	Fonction d'activation	24
3.2.3	Fonction erreur	24
3.3	Application des algorithmes	25
4	Conclusion	29

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 c'est équivalent), 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 (par exemple les méthodes du types BFGS telles que L-BFGS, très utilisé dans l'industrie).

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 :

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

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 \mathbb{R}^n . Mathématiquement, ces deux conditions se traduisent par :

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

et

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

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 \mathcal{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 et de classe \mathcal{C}^2 .

Cela a son importance puisque dans bon nombre de cas, nous effectuerons un développement limité sur la fonction f , localement, afin d'obtenir une forme quadratique plus un reste qui dépendra du 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 $\operatorname{argmin}_x f(x)$ 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 recherchée 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

```

1: procedure MÉTHODE DU GRADIENT À PAS FIXE( $x_0, n_{Max}, \rho$ )
2:   Soit  $i = 1$ 
3:   while  $\|\nabla f(x^{(i)})\| > tol \ \& \ i < n_{Max}$  do
4:      $d^{(i)} = -\nabla f(x^{(i)})$ ;
5:      $x^{(i+1)} = x^{(i)} + \rho d^{(i)}$ ;
6:      $i = i + 1$ ;
7:   end while
8: 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 itération.

Algorithm 2 Méthode du gradient à pas optimal

```

1: procedure MÉTHODE DU GRADIENT À PAS OPTIMAL( $x_0, n_{Max}, \rho_0$ )
2:   Soit  $i = 1$ 
3:   while  $\|\nabla f(x^{(i)})\| > tol \ \& \ i < n_{Max}$  do
4:      $d^{(i)} = -\nabla f(x^{(i)})$ ;
5:      $\rho^{(i)} = rechercheDuPas(x^{(i)}, d^{(i)}, \rho_0, tolR)$ 
6:      $x^{(i+1)} = x^{(i)} + \rho^{(i)} d^{(i)}$ ;
7:      $i = i + 1$ ;
8:   end while
9: end procedure

```

L'algorithme de recherche du pas est :

Algorithm 3 Recherche du pas inexact

```

1: procedure RECHERCHE DU PAS INEXACT( $x^{(i)}, d^{(i)}, \rho_0, tolR$ )
2:   Soit  $j = 1, \rho^{(0)} = 10\rho^{(1)}$ 
3:   while  $\|\rho^{(j)} - \rho^{(j-1)}\| > tolR \ \& \ j < maxIt$  do
4:      $\varphi'(\rho^{(j)}) = d^{(i)T} \nabla f(x^{(i)} + \rho^{(j)} d^{(i)})$ ;
5:      $\varphi''(\rho^{(j)}) = d^{(i)T} H_f(x^{(i)} + \rho^{(j)} d^{(i)}) d^{(i)}$ ;
6:      $\rho^{(j+1)} = \rho^{(j)} - \frac{\varphi'(\rho^{(j)})}{\varphi''(\rho^{(j)})}$ 
7:   end while
8: 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 :

$$Ax = b \quad (2.2)$$

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

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

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 :

$$x^{k+1} = x^k + r^k \quad (2.4)$$

$$\Leftrightarrow x^{k+1} - x^* = x^k - x^* + r^k \quad (2.5)$$

$$\Leftrightarrow -A(x^{k+1} - x^*) = -A(x^k - x^*) - Ar^k \quad (2.6)$$

$$\Leftrightarrow r^{k+1} = r^k - Ar^k \quad (2.7)$$

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

$$r^k \in Vect\{r^0, Ar^0, \dots, A^k r^0\} \quad (2.8)$$

Cela implique directement :

$$x^k - x^0 = \sum_{i=0}^{k-1} r^i \quad (2.9)$$

Donc il vient que :

$$x^k \in x^0 + Vect\{r^0, Ar^0, \dots, A^k r^0\} \quad (2.10)$$

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

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 :

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

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 :

$$Ax = b \quad (2.12)$$

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

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

Par le théorème de Cayley-Hamilton¹, nous pouvons obtenir la valeur de A^{-1} :

$$P(A) = \alpha_0 I + \alpha_1 A + \dots + \alpha_n A^n = 0 \quad (2.14)$$

$$\alpha_0 A^{-1} A + \alpha_1 A + \dots + \alpha_n A^n = 0 \quad (2.15)$$

$$(\alpha_0 A^{-1} + \alpha_1 + \dots + \alpha_n A^{n-1}) A = 0 \quad (2.16)$$

$$\alpha_0 A^{-1} + \alpha_1 + \dots + \alpha_n A^{n-1} = 0 \quad (2.17)$$

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

Ce qui donne finalement :

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

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

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

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

$$x^* \in \mathcal{K}(A, b) \quad (2.20)$$

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 $x_k \in \mathcal{K}_k(A, b)$ 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'Arnoldi² 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^* :

$$Ax = b \quad (2.21)$$

Le résidu initial est :

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

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

$$\langle b - Ax^*, v \rangle = 0, \forall v \in \mathbb{R}^n \Leftrightarrow b - Ax^* \perp \mathbb{R}^n \quad (2.23)$$

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 :

$$b - Ax \perp \mathbf{K} \Rightarrow \langle Au, v \rangle = \langle b, v \rangle, v \in \mathbf{K} \quad (2.24)$$

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 :

$$b - Ax \perp \mathbb{L} \Rightarrow \langle Au, v \rangle = \langle b, v \rangle, v \in \mathbb{L} \quad (2.25)$$

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} :

$$V = (V_1 \ V_2 \ \dots \ V_m) \quad (2.26)$$

$$W = (W_1 \ W_2 \ \dots \ W_m) \quad (2.27)$$

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

$$v = Vy \quad (2.28)$$

$$w = Wz \quad (2.29)$$

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 :

$$\langle Av, w \rangle = \langle r^0, w \rangle \ \forall w \in L \quad (2.30)$$

$$\langle AVy, Wz \rangle = \langle r^0, Wz \rangle \ \forall z \in \mathbb{R}^m \quad (2.31)$$

$$Avy = r^0 \quad (2.32)$$

$$y = (W^t AV)^{-1} W^t r^0 \quad (2.33)$$

Nous obtenons donc la solution :

$$x = x_0 + V(W^t AV)^{-1} W^t r^0 \quad (2.34)$$

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 \mathbf{K} et \mathbb{L}
- chercher la nouvelle itération dans $u + \mathbf{K}$ 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 $y = (W_m^t AV_m)^{-1} W_m^t r$
5. Créer la nouvelle solution : $x = x + V_m y$

Pour créer le sous-espace \mathbb{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 :

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

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

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

$$h_{ij} = \begin{cases} \langle Av_j, v_i \rangle, & i \leq j \\ \|v_{j+1}\|, & i = j + 1 \\ 0, & \text{sinon} \end{cases} \quad (2.36)$$

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

$$AV_m = V_{m+1}\bar{H}_m \quad (2.37)$$

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

et :

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

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

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

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 $\mathcal{K}_m(A, r_0) = \mathbb{R}^n$

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

```

1  def GMRES(A, b, epsilon):
2      max_iter = A.shape[0] #Number of maxiter
3      mat_q = np.zeros((max_iter, max_iter + 1))
4      mat_h = np.zeros((max_iter + 1, max_iter))
5      norm_b = np.linalg.norm(b)
6      be1 = np.zeros(max_iter + 1)
7      be1[0] = norm_b
8      mat_q[:, 0] = 1 / np.linalg.norm(b) * b.T # On définit ici que
9      ↪ l'on a forcément x0 = 0 (r0 = b)
10     for j in range(max_iter):
11         mat_q[:, j+1] = A @ mat_q[:, j]
12         for i in range(j+1):
13             mat_h[i, j] = mat_q[:, i] @ mat_q[:, j + 1]
14             mat_q[:, j+1] -= mat_h[i, j] * mat_q[:, i]
15             mat_h[j + 1, j] = np.linalg.norm(mat_q[:, j + 1])
16             mat_q[:, j + 1] /= mat_h[j + 1, j]
17         y = np.linalg.lstsq(mat_h, be1, rcond=None)[0]
18         residue = np.linalg.norm(y) / norm_b
19         if residue < epsilon:

```

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 :

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

Nous définissons alors le vecteur résidu $r_k = b - Ax_k$ appartenant à l'espace de Krylov d'ordre k et défini 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 $k^{\text{ième}}$ espace de krylov : \mathcal{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 intéressants :

— Le premier repose sur le principe d'orthogonalisation :

$$\exists x_{k+1} \in [x_0 + \mathcal{K}_k], r_{k+1} \perp \mathcal{K}_k \quad (2.42)$$

— Le principe de minimisation :

$$\exists x_{k+1} \in [x_0 + \mathcal{K}_k], \frac{1}{2} \langle Ax, x \rangle - \langle b, x \rangle \text{ est minimal} \quad (2.43)$$

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 :

$$d_k = x_{k+1} - x_k \quad (2.44)$$

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

$$\mathcal{K}_k = \text{vect} r_0, \dots, r_k = \text{vect} d_0, \dots, d_k \quad (2.45)$$

— Tous les vecteurs de la suite sont orthogonaux :

$$\forall 0 \leq l < k \leq n-1, \langle r_k, r_l \rangle = 0 \quad (2.46)$$

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

$$\forall 0 \leq l < k \leq n-1, \langle A d_k, d_l \rangle = 0 \quad (2.47)$$

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 algorithme possédant 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 une propriété importante lors de l'implémentation logicielle 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} :

$$x_{k+1} = x_0 + \alpha_0 p_0 + \dots + \alpha_k p_k \quad (2.48)$$

$$= x_0 + \gamma_0 r_0 + \gamma_1 A r_0 + \dots + \gamma_k A^k r_0 \quad (2.49)$$

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 reconnaît un polynôme dépendant de A :

$$P_k(A) = \gamma_0 I + \gamma_1 A + \dots + \gamma_k A^k \quad (2.50)$$

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

$$x_{k+1} = x_0 + P_k(A)r_0 \quad (2.51)$$

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

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

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

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 $f(x) = \frac{1}{2} x^T A x - b^T x$. 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 résidu 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}^n :

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

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

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

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 :

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

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

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

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

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 :

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

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

```

1:  $r_0 = Ax_0 - b$ ,  $p_0 = -r_0$ ,  $k = 0$ 
2: while  $r_0$  do
3:    $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$ 
4:    $x_{k+1} = x_k + \alpha_k p_k$ 
5:    $r_{k+1} = r_k + \alpha_k A p_k$ 
6:    $\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$ 
7:    $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$ 
8:    $k = k + 1$ 
9: 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 :

```

1  def gradient_conjugué(A, b, x0, tol, i_max):
2      print("Méthode du gradient conjugué")
3      i = 1
4      x_n = x0
5      x = list()
6      d = 0
7      r = 0
8      while(1):
9          r_old = r
10         r = A @ x_n - b
11         if i == 1:
12             d = -r
13         else:
14             beta = np.linalg.norm(r) ** 2 /
15                 ↪ np.linalg.norm(r_old) ** 2
16             d = -r + beta * d
17             p = r.T @ r / (d.T @ A @ d)
18             x_n = x_n + p * d
19             x.append(x_n)
20             i += 1
21         if np.linalg.norm(r) < tol or i > i_max:
22             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 changements dans l'algorithme du gradient conjugué.

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 résidu qui était jusqu'alors 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.

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  $\alpha_k$ 
5:    $x_{k+1} = x_k + \alpha_k p_k$ 
6:   Calculer  $\nabla f_{k+1}$ 
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
```

Pour définir le α , nous allons utiliser la condition forte de Wolfe :

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \quad (2.64)$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq -c_2 \nabla f_k^T p_k, 0 < c_1 < c_2 < \frac{1}{2} \quad (2.65)$$

Si l'on prends α satisfaisant cela, alors nous savons que les directions de descente vérifient :

$$-\frac{1}{1 - c_2} \leq \frac{\nabla f_k^T p_k}{\|\nabla f_k\|^2} \leq \frac{2c_2 - 1}{1 - c_2} \quad (2.66)$$

Ceci permet d'assurer que les valeurs p_k sont des directions de descente (Démonstration non-réalisée car longue et compliquée).

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 :

$$f(x+h) = f(x) + \langle h, \nabla f(x) \rangle + \frac{1}{2} \langle h, H_f(x)h \rangle + o(\|h\|^2) \quad (2.67)$$

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 :

$$d_k = -\nabla^2 f_k^{-1} \nabla f_k \quad (2.68)$$

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 :

$$\hat{H}_k = \frac{H_k + \beta I}{1 + \beta} \quad (2.69)$$

Une seconde approche serait de transformer la hessienne :

$$\hat{H}_k = U^T H_k U + \epsilon \quad (2.70)$$

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 :

$$d_k = -B^{-1}\nabla f_k \quad (2.71)$$

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 approximations la hessienne grâce à la formule :

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \quad (2.72)$$

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 algorithmes 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 XXI^e 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 :

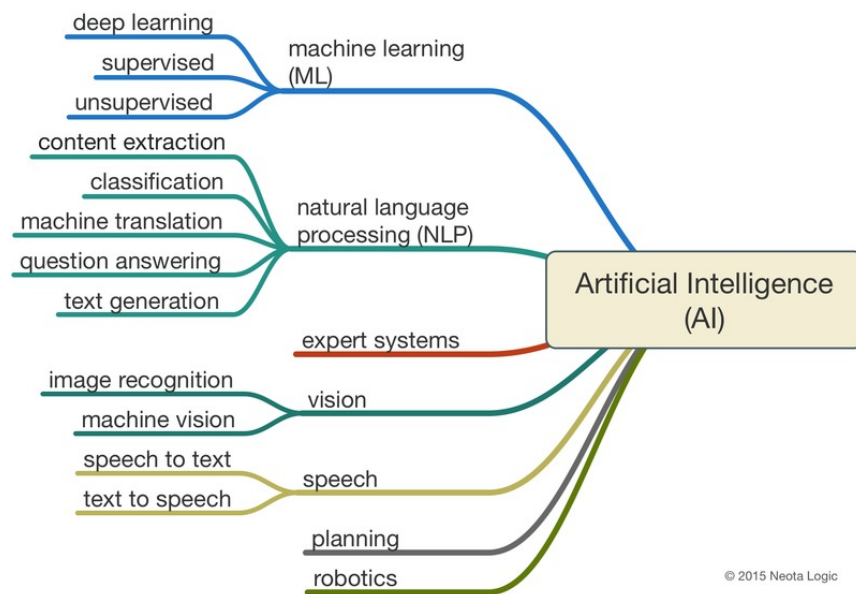


FIGURE 3.1 – Approches de l'IA

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 constituants 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 reconnaître et à reproduire. On dit qu'on "entraîne" la machine. Il faut pour cela lui fournir de grands jeux de données, c'est là qu'intervient 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'entraîner 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

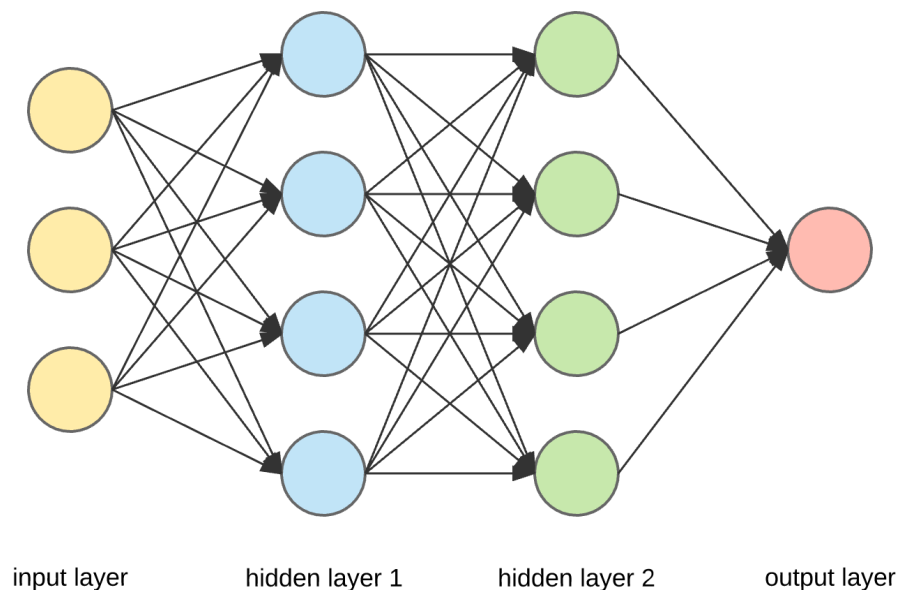


FIGURE 3.2 – Architecture d'un réseau de neurones

Les réseaux de neurones sont formés de plusieurs couches successives, comme on peut le voir sur l'image ci-dessus. Sur ce schéma, les nœuds représentent les neurones tandis que les flèches illustrent les "poids". La première couche (en jaune) est la couche d'entrée, elle contient les informations à "traiter". Tout au bout de la chaîne on a une couche de sortie, elle est ici composée d'un seul neurone mais il peut y en avoir plusieurs. Entre les entrées et sorties, on trouve des couches dites "cachées". C'est là que se font tous les calculs. Le réseau de neurone est parcouru dans les deux sens : d'abord de la couche "entrée" à la couche "sortie" (phase de prédiction), puis dans le sens opposé (phase d'apprentissage ou *backpropagation*). Lors de la première phase avant, les valeurs d'entrée sont généralement initialisées de manière aléatoire, puis les valeurs des neurones des couches suivantes sont calculées comme détaillé plus loin. Intéressons-nous donc à un seul neurone pour mieux comprendre son fonctionnement.

3.2.1 Détail d'un neurone

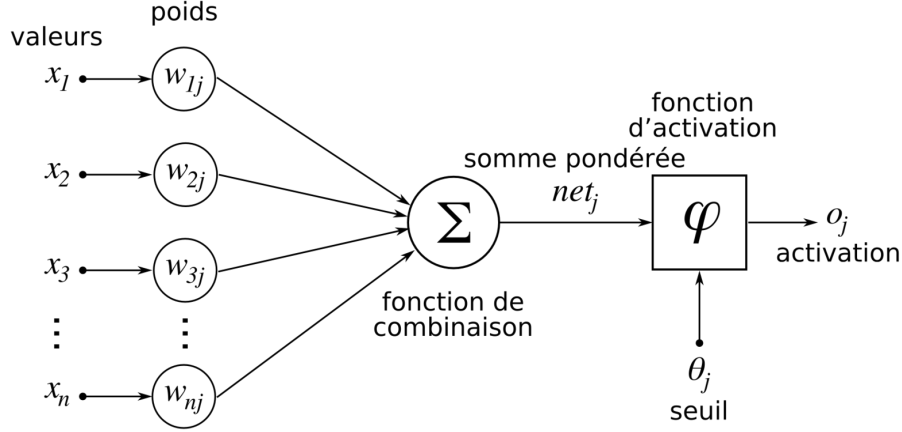


FIGURE 3.3 – Neurone détaillé

Chaque neurone prend une valeur entre 0 et 1, déterminée par ce qu'il reçoit en entrée et par une fonction d'activation. On voit, à gauche de l'image, des valeurs $x_1, x_2, x_3, \dots, x_n$. Ce sont les valeurs déduites de la couche précédente. Les poids, notés ici w représentent l'impact qu'ont ces valeurs sur le neurone. Vient ensuite la fonction d'activation. Son rôle est "d'écraser" la valeur de la somme pondérée ($\sum_{k=0}^N x_k w_k$) pour qu'elle appartienne à un intervalle voulu. On la notera φ .

Avant de commencer la partie mathématique, voici un récapitulatif des indices et exposants utilisés dans les équations ci-dessous :

$$\left\{ \begin{array}{l} L : \text{numéro de couche, la couche d'entrée est } L = 1 \\ k : \text{numéro du neurone de la couche d'entrée} \\ K : \text{nombre de neurones de la couche d'entrée (si on ne compte pas le biais noté } x_0) \\ n : \text{numéro du neurone de la couche } L \\ N : \text{nombre de neurones de la couche } L \text{ (si on ne compte pas le biais noté } a_0^{(L)}) \\ m : \text{numéro du neurone de la couche } L - 1 \\ M : \text{nombre de neurones de la couche } L - 1 \text{ (si on ne compte pas le biais noté } a_0^{(L-1)}) \end{array} \right.$$

On peut donc écrire que la valeur du premier neurone de la première couche cachée est :

$$a_1^{(2)} = \varphi \left(\sum_{k=0}^K x_k w_{k1}^{(1)} \right)$$

On peut le généraliser au $n^{\text{ème}}$ neurone de cette couche :

$$a_n^{(2)} = \varphi \left(\sum_{k=0}^K x_k w_{kn}^{(1)} \right)$$

Si l'on passe à la couche suivante, le premier neurone celle-ci s'écrit comme la somme pondérée des valeurs des neurones de la couche précédentes multipliées par leur poids sur ce neurone :

$$a_1^{(3)} = \varphi \left(\sum_{i=0}^N a_i^{(2)} w_{i1}^{(2)} \right)$$

C'est donc une somme de sommes et pour le $n^{\text{ème}}$ neurone de la 2^e couche cachée, elle vaut :

$$a_n^{(3)} = \varphi \left(\sum_{i=0}^N \varphi \left(\sum_{k=0}^K x_k w_{kn}^{(1)} \right) a_i w_{i1}^{(2)} \right)$$

Finalement, pour n'importe quel neurone, on a :

$$a_n^{(L)} = \varphi \left(\sum_{m=0}^M \left[\dots \left[\varphi \left(\sum_{i=0}^N \left(\sum_{k=0}^K x_k w_{kn}^{(1)} \right) w_{in}^{(2)} \right) \right] \dots \right] w_{nm}^{(L)} \right)$$

3.2.2 Fonction d'activation

Nous n'avons pas encore parlé de la fonction d'activation, ici φ . Son rôle est de dire si un neurone doit être "actif" ou non, selon la valeur de la somme pondérée. Nous nous attarderons pas trop dessus mais il faut tout de même savoir quelle fonction prendre on utilise afin de la dériver par la suite¹. Une des fonctions d'activation les plus répandues est la fonction sigmoïde définie par $\sigma(x) = \frac{e^x}{1+e^x}$ et dont les valeurs sont comprises entre 0 et 1.

3.2.3 Fonction erreur

Après la chaque phase avant, on obtient des valeurs pour chacun des neurones de la couche de sortie. Cependant ces valeurs ont été calculées à partir d'entrées initialisées aléatoirement. Il faut donc comparer ces résultats avec ceux attendus. Plusieurs fonctions erreur permettent de faire cela. Une liste non exhaustive est disponible ici². On peut par

1. Les fonctions d'activations plus en détails : <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

2. <https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications>

exemple citer la fonction d'erreur quadratique :

$$C_{MST}(W, B, S^r, E^r) = 0.5 \sum_j (a_j^L - E_j^r)^2$$

où W est la matrice des poids du réseau de neurones, B contient les biais³, S^r , l'entrée d'une phase avant et E^r , la sortie attendue.

C'est lors de la phase d'apprentissage que l'on est face à un problème d'optimisation. L'objectif est ici de minimiser cette fonction erreur. On utilise pour cela une méthode de descente de gradient. Cependant, de cette manière nous ne sommes pas toujours certain de trouver le minimum global de la fonction. C'est pourquoi il faut parfois utiliser la fonction logarithme. C'est le cas de la *L2-regularized cost function* :

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\theta_{ji}^{(l)})^2$$

Pendant la phase arrière, les poids et biais sont mis à jour. La phase avant s'exécute à nouveau, recalculant les valeurs des différents neurones jusqu'à la couche de sortie. Cette alternance entre les deux phases se répète jusqu'à ce que l'erreur soit suffisamment faible ou lorsqu'une condition d'arrêt est atteinte.

3.3 Application des algorithmes

Nous voilà donc avec un ensemble de valeurs de sortie proches de nos attentes. Mais quelles attentes ? Que signifient ces valeurs ?

Nous avons choisi pour ce rapport de prendre l'exemple de la reconnaissance de vêtements. Nous une liste de catégorie (écharpe, blouson, pantalon, chaussure..) et le but est de savoir à quelle catégorie appartient un vêtement. On assigne à chaque sortie une catégorie. Si l'on veut faire deviner une écharpe alors toutes les sorties devront prendre la valeur 0 sauf la sortie correspondant aux écharpes qui devra être à 1. Maintenant que nous connaissons les sorties attendues, l'algorithme pourra calculer la valeur de la fonction erreur et tenter de la minimiser. Après plusieurs itérations on obtiendra alors une probabilité pour chaque catégorie, en espérant que la sortie associée à "écharpes" soit proche de 1 et supérieure à toutes les autres.

3. Pour en apprendre plus sur les biais :

<https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a>
<https://deeplylearning.fr/cours-theoriques/>

Pour coder l'algorithme en python, nous avons fait appel à la bibliothèque mathématique *Tensorflow*⁴. Le code est le suivant :

```
1 from __future__ import absolute_import, division, print_function
2
3 import tensorflow as tf
4 from tensorflow import keras
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9 def plot_image(i, predictions_array, true_label, img):
10 predictions_array, true_label, img = predictions_array[i], true_label[i],
    ↪ img[i]
11 plt.grid(False)
12 plt.xticks([])
13 plt.yticks([])
14
15 plt.imshow(img, cmap=plt.cm.binary)
16
17 predicted_label = np.argmax(predictions_array)
18 if predicted_label == true_label:
19 color = 'blue'
20 else:
21 color = 'red'
22
23 plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
24 100 * np.max(predictions_array),
25 class_names[true_label]),
26 color=color)
27
28
29 def plot_value_array(i, predictions_array, true_label):
30 predictions_array, true_label = predictions_array[i], true_label[i]
31 plt.grid(False)
32 plt.xticks([])
33 plt.yticks([])
34 thisplot = plt.bar(range(10), predictions_array, color="#777777")
35 plt.ylim([0, 1])
36 predicted_label = np.argmax(predictions_array)
```

4. www.tensorflow.org

```
37
38 thisplot[predicted_label].set_color('red')
39 thisplot[true_label].set_color('blue')
40
41
42 print(tf.__version__)
43
44 fashion_mnist = keras.datasets.fashion_mnist
45
46 (train_images, train_labels), (test_images, test_labels) =
    ↪ fashion_mnist.load_data()
47
48 class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
49 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
50
51 train_images = train_images / 255.0
52
53 test_images = test_images / 255.0
54
55 plt.figure()
56 plt.imshow(train_images[10])
57 plt.colorbar()
58 plt.grid(False)
59 plt.show()
60
61 plt.figure(figsize=(10, 10))
62 for i in range(25):
63     plt.subplot(5, 5, i + 1)
64     plt.xticks([])
65     plt.yticks([])
66     plt.grid(False)
67     plt.imshow(train_images[i], cmap=plt.cm.binary)
68     plt.xlabel(class_names[train_labels[i]])
69     plt.show()
70
71 model = keras.Sequential([
72     keras.layers.Flatten(input_shape=(28, 28)),
73     keras.layers.Dense(128, activation=tf.nn.relu),
74     keras.layers.Dense(10, activation=tf.nn.softmax)
75 ])
```

```
76
77 model.compile(optimizer='adam',
78               loss='sparse_categorical_crossentropy',
79               metrics=['accuracy'])
80
81 model.fit(train_images, train_labels, epochs=5)
82
83 test_loss, test_acc = model.evaluate(test_images, test_labels)
84
85 print('Test accuracy:', test_acc)
86 predictions = model.predict(test_images)
87
88 num_rows = 5
89 num_cols = 3
90 num_images = num_rows * num_cols
91 plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
92 for i in range(num_images):
93     plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
94     plot_image(i, predictions, test_labels, test_images)
95     plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
96     plot_value_array(i, predictions, test_labels)
97
98 plt.show()
```

Chapitre 4

Conclusion

Ce mémoire nous a permis de faire une présentation des différentes méthodes de descente utilisées afin de résoudre différents problèmes d'optimisation et notamment de minimisation. En fonction du problème étudié et de sa dimension, nous avons un certain nombre d'outils à notre disposition afin de trouver le minimiseur d'une fonction.

Nous avons donc développé principalement l'utilisation de la méthode du gradient-conjugué ainsi que les principales caractéristiques de celle-ci. Pour ce faire, nous avons établi et défini un outil mathématique très important en minimisation : les espaces de krylov qui nous permettent de définir d'importantes propriétés de convergence pour ces algorithmes.

Nous avons aussi cherché quelques pistes d'amélioration avec l'utilisation des algorithmes de Newton. Néanmoins, le calcul étant assez lourd à effectuer, nous avons exposé le principe des algorithmes de Quasi-Newton qui permettent d'approximer la matrice hessienne. Cela nous produit une famille d'algorithmes possédant les propriétés de ceux de Newton tout en étant moins lourds.

Ensuite, nous avons choisi d'appliquer ces études à un domaine très vaste et en expansion depuis quelques années. En effet, dès le début de notre étude des intelligences artificielles, nous nous sommes aperçus que ce n'était qu'un ensemble de mathématiques d'optimisation que nous utilisons dans différents buts. Nous avons donc appliqué cela à plusieurs problèmes.

Table des figures

3.1	Approches de l'IA	21
3.2	Architecture d'un réseau de neurones	22
3.3	Neurone détaillé	23