Jérôme Malick

mars 2020, période coronavirus – mis à jour janvier 2021

Notes de cours d'optimisation en 2A de l'Ensimag

Ces notes couvrent une partie du cours d'optimisation en 2A de l'ENSIMAG, cours appelé officiellement "Optimisation Numérique" mais qui consiste en un cours d'introduction à l'optimisation continue et son application en apprentissage automatique et science des données. Nous allons utiliser ces notes comme en complément du cours hebdomadaire.

1 Introduction

L'optimisation est une discipline des mathématiques appliquées, qui se retrouve au coeur des méthodes d'analyse de données. Par exemple, en apprentissage statistique, des algorithmes d'optimisation passent sur de grands volumes de données pour calculer les paramètres des modèles stochastiques utilisés ensuite pour prédire, classifier ou décider.

L'optimisation mathématique a plusieurs aspects interagissants : la formulation des problèmes eux-même, la conception d'algorithmes pour les résoudre, l'étude de leur propriétés théoriques, leur implémentation et leur mise en oeuvre pratique. Chacun de ces points fait l'objet de livres entiers et de nombreux articles de recherche ; ici, nous n'allons qu'effleurer ces questions dans la perspective de l'analyse de données. Nous allons insister sur la structure particulière des problèmes d'optimisation, apparaissant en recommendation, classification et régression, et impliquant des données de grandes tailles, éventuellement stockées sur différentes machines. Nous rappellerons les méthodes de base de l'optimisation et les idées qui permettent aux algorithmes de passer à l'échelle.

Pour approfondir les notions présentées dans cette courte introduction, on renvoie au cours complet et/ou aux livres ou articles suivants : [BV04] pour la modélisation en optimisation, [BCN17] pour une vue d'ensemble de l'optimisation pour l'apprentissage, [HUL01] pour l'analyse analyse convexe, [Nes13] pour les algorithmes du premier ordre, [Bub15] pour l'étude des algorithmes d'optimisation pour l'apprentissage.

2 Apprentissage et optimisation

La chaine de traitement de données se résume par trois grandes étapes : observer le monde (collecter des données), proposer des modèles (conception et apprentissage), tester sur de nouvelles données (estimer l'erreur a posteriori). La phase d'apprentissage consiste à chercher les paramètres d'un modèle stochastique expliquant *au mieux* les données, et c'est ici qu'apparaît l'optimisation, lors du calcul des paramètres optimaux. Nous illustrons ceci dans contexte général d'apprentissage supervisé. L'apprentissage non-supervisé ou semi-supervisé (ou même pour d'autres techniques d'apprentissage) mène à des problèmes du même type.

Notations, définitions Commençons par définir quelques termes et notations. Dans ce cours, la variable d'optimisation est un vecteur de taille d qui sera notée $x \in \mathbb{R}^d$ (en statistique, c'est plutôt β, en apprentissage ω, en deep learning θ). On note le produit scalaire canonique de l'espace $\mathbb{R}^d$ (somme des produit termes à termes de deux vecteurs) par

$$\langle x, y \rangle = \sum_{j=1}^{d} x_j y_j \qquad \text{pour } x \in \mathbb{R}^d \text{ et } y \in \mathbb{R}^d.$$

On utilise deux normes dans $\mathbb{R}^d$, la norme ℓ_2 (la norme euclidienne associée au produit scalaire) et la norme ℓ_1, définies respectivement par

$$\|x\|_2 = \sqrt{\langle x, x \rangle} = \Big(\sum_{j=1}^{d} {x_j}^2 \Big)^{\frac{1}{2}} \qquad \text{et} \qquad \|x\|_1 = \sum_{j=1}^{d} |x_j| \qquad \text{pour } x \in \mathbb{R}^d.$$

Pour les problèmes d'optimisation qui font plus particulièrement l'objet de la section 3, nous utilisons la terminologie suivante. On considère une fonction $f\colon \mathbb{R}^d \to \mathbb{R}$ qu'on appelle *fonction-objectif* ou *critère*, et un ensemble $C \subset \mathbb{R}^d$ qu'on appelle *ensemble des contraintes* ou *ensemble réalisable*. Le problème de trouver la valeur optimale $f_\star \in \mathbb{R}$ et une solution optimale $x_\star \in C$ telle que

$$f_\star = f(x_\star) \leqslant f(x) \qquad \text{pour tout } x \in C$$

est un problème d'optimisation, que l'on écrit sous la forme suivante :

$$\min_{x \in C} \quad f(x). \tag{1}$$

Contexte d'apprentissage supervisé Plaçons-nous dans un cadre standard d'apprentissage supervisé. Après une phase préliminaire de collecte, stockage, et nettoyage des données, nous disposons un ensemble de données d'entrainement constitué de n observations sous forme de couples (a_i, y_i) avec un vecteur de *caractéristique* $a_i \in \mathbb{R}^m$ et une *attribut* $y_i \in \mathbb{R}$ (appelée aussi *étiquette*, *score* ou *label*). Pour fixer les idées, on peut penser que a_i représente les réponses d'un étudiant à un questionnaire sur sa vie personnelle et y_i représente sa note à l'examen final (ainsi $y_i \in [0, 20]$) ou alors l'information si il valide son année (dans ce cas, y_i est binaire). Nous souhaitons alors utiliser ces données pour *prédire* l'étiquette associée à une nouvelle donnée de caractéristique a. Pour l'exemple des notes des étudiants, on voudrait avoir, à partir des notes des étudiants de l'année précédente, une prédiction des notes finales des nouveaux étudiants. En général, le problème de prédiction s'appelle *régression* dans le cas où $y_i \in \mathbb{R}$, et *classification* dans le cas où les y_i prennent un nombre fini de valeurs (par exemple, classification binaire si $y_i \in \{-1, 1\}$).

En apprentissage supervisé, cette prédiction se fait par un modèle stochastique calibré sur les données d'entrainement. On choisit généralement un modèle avec une forme fixée dépendante d'un vecteur de paramètres que l'on note $x \in \mathbb{R}^d$. Une fonction de prédiction générale est ainsi une fonction $h(\cdot, x)$ paramétrée par x qui propose une prédiction $b = h(a, x) \in \mathbb{R}$ pour une nouvelle donnée a. Voici deux exemples fondamentaux de fonctions de prédiction :

- Prédiction linéaire : $h(a, x) = \langle x, a\rangle$ dans le cas le plus simple (ou $h(a, x) = \langle x, \phi(a)\rangle$ pour des modèles « à noyaux » plus riches) ;
- Prédiction (hautement) non-linéaire par réseau de neurones :

$$h(a, x) = \langle x_m, \sigma(\langle x_{m-1}, \sigma(\cdots \langle x_2, \sigma(\langle x_1, a\rangle)\rangle \cdots)\rangle\rangle$$

  où la prédiction est calculée en appliquant une succession de composition de combinaisons linéaires et d'une fonction d'activation non-linéaire (comme la fonction sigmoid $\sigma(z) = 1/(1 + \exp(-z))$ ou la fonction ReLU $\sigma(z) = \max\{0, z\}$).

Apprendre, c'est optimiser La phase d'apprentissage consiste à calculer le paramètre $x \in \mathbb{R}^d$ du modèle qui explique *au mieux* les données disponibles. Cela signifie que la prédiction du modèle doit *être proche* des vraies étiquettes sur les données connues

$$h(a_i, x) \simeq y_i \qquad \text{pour } (a_i, y_i) \text{ données d'entrainement} \tag{2}$$

et aussi pouvoir bien se généraliser pour des données inconnues. On mesure la différence (2) entre la prédiction et l'étiquette avec une fonction, appelée *fonction perte*, $\ell\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, dont la plus simple et connue est la perte des moindres carrées $\ell(y, z) = \frac{1}{2}(y - z)^2$. D'autres fonctions pertes sont très utilisées aussi, comme $\ell(y, z) = |y - z|$ en régression robuste, $\ell(y, z) = \log(1 + \exp(y\, z))$ en régression logistique, ou $\ell(y, z) = \max\{0, 1 - y\, z)$ en classification par exemple. Il est naturel de s'intéresser alors au paramètre qui réalise le moins d'erreurs (2) en moyenne, au sens de la mesure choisie ℓ, c'est-à-dire $x \in \mathbb{R}^d$ solution du problème d'optimisation

$$\min_{x \in \mathbb{R}^d} \quad \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(a_i, x)). \tag{3}$$

Point de vue statistique Le problème (3) admet une interprétation statistique dans le cas où les données sont des réalisations d'une variable aléatoire. En effet, si on suppose que les données (a_i, y_i) sont un échantillon des observations indépendantes et identiquement distribuées d'un vecteur aléatoire $(a, y) \in \mathbb{R}^{d+1}$, le problème d'optimisation sous-jacent consiste en la minimisation de l'espérance de l'erreur, qui s'écrit

$$\min_{x \in \mathbb{R}^d} \quad \mathbb{E}[\ell(y, h(a, x))] \tag{4}$$

où l'espérance est prise sur la probabilité $\mathbb{P}$ de (a, y). La perte peut aussi être une estimation de maximum de vraisemblance $-\log \mathbb{P}_x\big((a, y_i)\big)$ pour une distribution $\mathbb{P}_x$ paramétrisée par le vecteur à estimer. La fonction objectif de (3) s'interprète alors simplement comme l'observation empirique de l'espérance de l'erreur de (4). En théorie, par la loi des grands nombres, l'espérance empirique va converger l'espérance lorsque le nombre de données grandit. En pratique, les données sont en nombre fini et l'hypothèse de variable sous-jacente est discutable, et donc nous travaillons directement sur l'espérance empirique. Mais il faut retenir de cette situation idéale que la minimisation de (3) n'est pas une fin en soit car elle néglige l'aspect statistique sur les données. Il faut donc prévoir des mécanismes pour pouvoir contrôler l'erreur totale, qui est la somme de l'erreur d'optimisation (la différence entre la solution optimale (3) et l'itéré courant d'un algorithme d'optimisation) et de l'erreur statistique (la différence entre la solution optimale (3) et celle de (4)).

Minimisation du risque empirique régularisé Plutôt que de considérer directement (3), il est préférable d'ajouter dans l'objectif une fonction de régularisation $r \colon \mathbb{R}^d \to \mathbb{R}_+$ pondérée par un paramètre $\lambda > 0$, ce qui amène au problème d'optimisation

$$\min_{x \in \mathbb{R}^d} \quad f(x) \;=\; \frac{1}{n} \sum_{i=1}^{n} \ell\big(y_i, h(a_i, x)\big) \;+\; \lambda\, r(x). \tag{5}$$

Par exemple, dans le cas d'un modèle linéaire $h(a, x) = \langle a, x \rangle$, d'une perte par moindres carrés, et d'une régularisation par norme 2 (appelée aussi régularisation de Tikhonov), l'apprentissage revient au problème

$$\min_{x \in \mathbb{R}^d} \quad \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} (y_i - \langle a_i, x \rangle)^2 + \frac{\lambda}{2} \|x\|_2^{\,2} \;=\; \frac{1}{2n} \|Ax - y\|_2^{\,2} + \frac{\lambda}{2} \|x\|_2^{\,2} \tag{6}$$

où on rassemble toutes les données dans une matrice $A \in \mathbb{R}^{n \times d}$ dont les lignes sont les $a_i^\top$ et le vecteur $y \in \mathbb{R}^n$ des y_i. Insistons sur le fait que le paramètre de régularisation $\lambda > 0$ est un paramètre du problème d'optimisation, mais pas un paramètre du modèle d'apprentissage comme $x \in \mathbb{R}^d$.

On remarque les deux choix extrêmes : avec λ proche de 0, on souhaite coller aux données d'entrainement ; à l'inverse, λ très grand permet de diminuer l'importance des données. Entre ces deux choix, la présence de la régularisation r a plusieurs intérêts :

- apprendre de manière plus raisonnée : éviter le sur-apprentissage sur les données connues pour mieux généraliser aux nouvelles données ;
- promouvoir une structure particulière sur le paramètre d'apprentissage, comme par exemple de la parcimonie avec $r(x) = \|x\|_1$;
- améliorer la résolution numérique du problème d'optimisation, comme par exemple en améliorant le conditionnement avec la norme $r(x) = \|x\|_2^{\,2}$ (voir section 3.3).

Différentiabilité et gradient Terminons cette formalisation des problèmes d'optimisation en apprentissage par une remarque importante en vue de la résolution numérique. Certaines modélisations sous la forme (5) considèrent un modèle h, une perte ℓ et une régularisation r, tout trois différentiables, ce qui implique que la fonction objectif f est différentiable aussi. Souvent, le gradient $\nabla f(x)$ est de plus *facilement calculable*, ce qui signifie que l'on dispose d'une expression explicite simple du gradient, ou d'un algorithme efficace de calcul de dérivées itératives (appelée « différentiation automatique » en optimisation ou « propagation inverse » en apprentissage). Par exemple, la fonction-objectif du problème de régression ridge (6) est différentiable et on a

$$\nabla f(x) = \left(\frac{1}{n} A^\top A + \lambda \mathrm{Id} \right) x - \frac{1}{n} A^\top y \tag{7}$$

dont le coût de calcul est dominé par le coût de la multiplication de la matrice A et sa transposée $A^\top$. Cependant les fonctions-objectifs des problèmes d'apprentissage ne sont pas toujours différentiables : en fonction de la nature des données, la modélisation peut privilégier des pertes non-différentiables ou des régularisations non-différentiables ; donnons deux exemples classiques :

- la classification par « SVM » (support vector machines) dont la perte admet une cassure à cause du max
$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - y_i {a_i}^\top x\} \;+\; \frac{\lambda}{2}\|x\|_2^2; \tag{8}$$
- la régression « lasso » utilisant la norme $\|\cdot\|_1$ pour induire des solutions optimales creuses (c'est-à-dire avec beaucoup de $x_i = 0$), mais qui est justement non-différentiable quand une composant x_i s'annule
$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{2n}\|Ax - y\|_2^2 \;+\; \lambda\|x\|_1\,. \tag{9}$$

Bilan sur les problèmes d'optimisation en apprentissage Les problèmes d'optimisation en apprentissage sont typiquement de la forme

$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{n}\sum_{i=1}^{n} f_i(x) + r(x); \tag{10}$$

la minimisation se fait sur x, le paramètre du modèle que l'on souhaite calculer ; la fonction de régularisation $r : \mathbb{R}^d \to \mathbb{R}$ permet de "mieux" apprendre ; les fonctions $f_i : \mathbb{R}^d \to \mathbb{R}$ mesurent la qualité de prédiction associé à x sur un bloc de données. C'est ce que nous venons de voir, dans un cadre d'apprentissage supervisé, avec la minimisation du risque empirique régularisé (5).

En général, la difficulté particulière des problèmes d'optimisation issus de l'analyse de données ne vient pas de la formulation (10) en elle-même, mais de la grande taille : grands volumes de données (n grand, *big data*) et/ou grande complexité des modèles (d grand ou h complexe, *big models*). Ainsi la résolution de (10) avec n et d fait l'objet des sections suivantes : nous allons tout d'abord revoir, dans la section 3, les idées de base de l'optimisation, puis voir, dans la section 4 comment elles s'adaptent pour minimiser de grandes sommes finies comme (10).

3 Introduction à l'optimisation

Cette section donne une vue globale des fondamentaux de l'optimisation. La présentation insiste sur les idées en cachant sous le tapis les détails mathématiques et algorithmiques ; l'objectif étant uniquement de donner des éléments pour mieux comprendre les méthodes utilisées en apprentissage.

3.1 Problèmes d'optimisation

Au-delà de son rôle en une analyse de données, l'optimisation apparait dans de nombreuses disciplines scientifiques et dans les sciences de l'ingénieur au sens large (industrie, services...). Insistons cependant sur le fait que les problèmes n'apparaissent rarement déjà formalisés : une étape de modélisation est primordiale, comme illustré à la section précédente. Dans cette section, nous nous plaçons après cette étape de modélisation et nous considérons un problème optimisation, sous la forme générale (1). Commençons par rappeler quelques idées à garder à l'esprit.

Résoudre un problème d'optimisation Il est extrêmement rare que l'on puisse trouver une expression explicite des solutions optimales $f_\star$ et $x_\star$. Dans quelques cas particuliers simples, des solutions explicites peuvent tout de même être obtenues en résolvant explicitement des équations satisfaites par les solutions. On les appellent les *conditions d'optimalité*, qui sont, dans le cas particulier de f différentiable sans contrainte ,

$$x \text{ est un minimum (local) de } f \text{ différentiable } \implies \nabla f(x) = 0. \tag{11}$$

En utilisant cette propriété, on montre par exemple que la solution optimale de (6)

$$x_\star = (A^\top A + n\lambda \mathrm{Id})^{-1} A^\top y$$

et peut ainsi se calculer par la résolution du système linéaire.

Dans la plupart des cas, on approche donc numériquement $f_\star$ et $x_\star$ par un algorithme itératif (voir section 3.3). On cherche ainsi une solution approchée à une précision ε ; ce qui signifie que l'on cherche $\bar{x} \in C$ tel que $\|\bar{x} - x_\star\| \leqslant \varepsilon$, ou $f(\bar{x}) - f_\star \leqslant \varepsilon$, ou bien que les conditions d'optimalité sont vérifiées à ε près (par exemple, $\|\nabla f(\bar{x})\| \leqslant \varepsilon$ dans le cas sans contraintes). Notons qu'en appentissage, avoir une précision ε très petite ne fait aucun sens, d'après les discussions de la section précédente.

Dernière généralité à ne pas oublier : il est très difficile, voir impossible, de résoudre – même approximativement – un problème d'optimisation général (1) avec f et C quelconques. Il est essentiel de disposer de structures mathématiques pour envisager résoudre les problèmes associés. Dans notre cas, deux propriétés importantes sont les propriétés différentiabilité et de convexité, que nous allons revoir rapidement à la section suivante.

En théorie L'optimisation mathématique est une discipline des mathématiques appliquées qui étudie les algorithmes qui calculent des solutions approchées à (1). On cherche à établir des garanties de convergence, des garanties d'efficacité numérique, et des garanties statistiques sur la solution obtenue. Cet aspect théorique n'est pas développé ici dans ce cours qui se veut introductif. Nous ne faisons que mentionner, en section 3.3, quelques résultats sur l'analyse de convergence des algorithmes, qui est une question centrale : il s'agit de montrer la convergence vers une solution optimale et aussi la vitesse de convergence, afin d'avoir des estimations de la complexité théorique des algorithmes dans certaines situations particulières.

En pratique Il existe de nombreux algorithmes et des logiciels spécialisés pour certaines classes de problèmes d'optimisation. Il existe aussi des logiciels « modeleurs » qui font une interface entre les utilisateurs et les logiciels optimisation, permettant leur utilisation même aux non-experts. Par exemple, en science des données, la bibliothèque `scikitlearn` (en python) rassemble des méthodes pour l'apprentissage utilisant un format unique qui permet de les utiliser avec simplement une connaissance limitée des algorithmes d'optimisation sous-jacents. En général, il faut absolument se servir de ces algorithmes et ces logiciels généralistes lors de la modélisation des problèmes et pour résoudre les problèmes pour lesquelles la performance numérique n'est pas la priorité. Pour les problèmes de grandes tailles ou dans un cadre exigeant, il est par contre fondamental de développer des algorithmes spécifiques exploitant la structure particulière des problèmes visés. Dans tous les cas, il est utile de connaître les idées de base de l'optimisation, pour réagir en cas de soucis avec les logiciels généralistes, ou pour envisager un réglage des paramètres des algorithmes. C'est justement l'objectif de ce cours de présenter ces idées de bases dans un cadre d'analyse de données.

3.2 Analyse convexe pour impatients

Les problèmes d'optimisation convexe forment une classe de problèmes que l'on peut bien étudier et résoudre – en théorie et en pratique. De plus, les méthodes efficaces dans le cas non-convexe sont souvent des extensions de méthodes convexes ou utilisent des sous-problèmes convexes. Nous présentons ici, de manière le plus légère possible, le minimum d'analyse convexe pour comprendre les algorithmes et les modèles en science des données.

Convexité et différentiabilité Une fonction $f\colon \mathbb{R}^d \to \mathbb{R}$ est dite convexe si elle vérifie pour tous x et y

$$f(\alpha x + (1-\alpha)y) \leqslant \alpha f(x) + (1-\alpha) f(y) \qquad \text{pour tout } \alpha \in [0,1].$$

Cette inégalité signifie géométriquement que le graphe de f est en-dessous du segment rejoignant les points $(x, f(x))$ et $(y, f(y))$ dans $\mathbb{R}^d \times \mathbb{R}$, comme l'illustre la figure 1 avec $d = 1$.

Par exemple $f(\omega) = \exp(\omega)$ et $f(\omega) = \omega \log(\omega)$ (pour $\omega > 0$) sont des fonctions convexes d'une variable réelle ; de même que /les fonctions pertes $\ell(y, \cdot)$ introduites à la section 2. Les fonctions affines et les normes (comme $\|\cdot\|_2$ et $\|\cdot\|_1$) sont convexes en toute dimension. Il est souvent simple de montrer

qu'une fonction f est convexe, sans revenir à la définition, en montrant que f est construite à partir de fonctions convexes de base et d'opérations qui préservent la convexité. Notamment, f est convexe si elle est :

– une somme positive de fonctions convexes

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) \quad \text{avec } \alpha_1, \alpha_2 \geqslant 0,$$

– une composition d'une fonction convexe avec une application affine

$$f(x) = g(Ax + b),$$

– un maximum de fonctions convexes

$$f(x) = \max_i f_i(x).$$

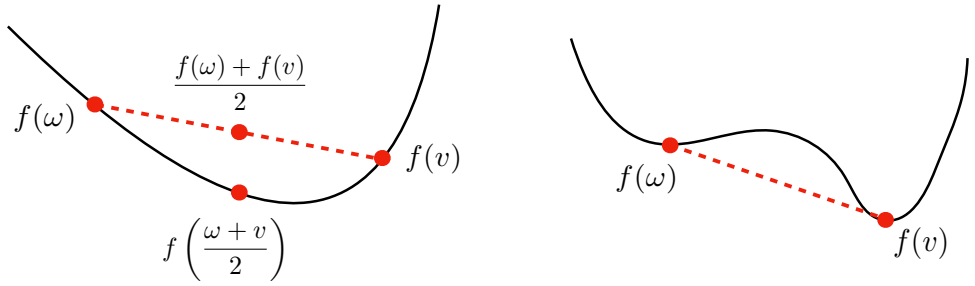

Figure 1 Illustration de la convexité avec des fonctions de $\mathbb{R}$ dans $\mathbb{R}$: à gauche, une fonction convexe, et à droite une fonction non-convexe (dont on note qu'elle admet un minimum local en y qui n'est pas un minimum).

La convexité entraîne des propriétés particulières sur les gradients (et les matrices hessiennes) des fonctions convexes différentiables [1]. Par exemple, une propriété caractéristique des fonctions convexes est la suivante (illustrée sur la figure 2)

$$f(x) + \langle \nabla f(x), (y - x) \rangle \leqslant f(y) \qquad \text{pour tous } x \text{ et } y. \tag{12}$$

Cette propriété implique, en particulier, qu'un minimum local ou même un point critique d'une fonction convexe est en fait un minimum global : on voit aisément que la propriété $\nabla f(x) = 0$ implique que x est un minimum global de f. En termes mathématiques : la condition d'optimalité (11), toujours nécessaire, est aussi suffisante dans le cas convexe, et caractérise en fait un minimum global.

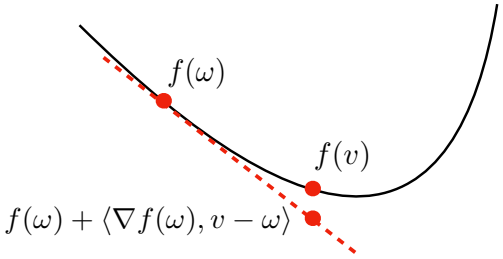

Figure 2 Illustration de l'inégalité pour une fonction convexe de $\mathbb{R}$ dans $\mathbb{R}$: 'une fonction convexe (différentiable) est toujours au dessus de ses tangentes.

1. Il existe toute une théorie qui étudie le cas des fonctions non-différentiables, avec en particulier un objet qui explique le premier ordre des fonctions convexes (le sous-différentiel et les sous-gradients). Nous ne parlons pas de ces notions ici.

Convexité forte et différentiabilité forte Certaines fonctions convexes ont des propriétés supplémentaires utiles pour l'optimisation. Parmi ces propriétés, la convexité forte mérite une attention particulière, car elle garantit l'existence et l'unicité d'une solution optimale, ainsi que de meilleures propriétés des algorithmes.

On dit qu'une fonction est fortement convexe si « elle reste convexe quand on lui enlève un peu de convexité quadratique » ; plus précisément f est fortement convexe (de module $\mu > 0$) si la fonction $x \mapsto f(x) - \mu\|x\|_2^2$ est toujours convexe. On démontre alors aisément qu'une fonction fortement convexe admet un unique minimum. Il est fréquent en analyse de données que les problèmes d'optimisation convexe soient construits pour être en fait fortement convexes, en choisissant lors de la modélisation d'ajouter d'une régularisation par la norme ℓ_2. C'est par exemple le cas pour les moindres carrés avec régularisation ℓ_2 de (6) où la fonction est fortement convexe de module $\mu = \lambda$. Notons aussi qu'une fonction non-différentiable peut aussi être fortement convexe, comme par exemple $R(x) = \|x\|_1 + \frac{\mu}{2}\|x\|_2^2$ appelé parfois la régularization « elastic-net ».

Une autre propriété très utile en optimisation est la différentiabilité forte de l'objectif, qui correspond intuitivement à ce que la fonction ne puisse pas « changer de forme trop rapidement ». Plus précisément, on dit que f est dite L-fortement différentiable (ou L-lisse) si elle est de classe C^1 avec un gradient Lipschitz de constante $L > 0$, c'est-à-dire

$$\|\nabla f(x) - \nabla f(y)\| \leqslant L\|x - y\| \qquad \text{pour tous } x \text{ et } y \text{ dans } \mathbb{R}^d.$$

Par exemple, on montre aisément que la fonction-objectif de (6) est fortement différentiable avec une constante L inférieure à la somme de λ et de la plus grande valeur propre de la matrice $A^\top A$. Notons que l'impact de ces deux notions [2] de convexité/différentiabilité forte porte sur les résultats mathématiques et aussi sur le comportement des algorithmes, comme nous allons le voir dans la section suivante.

3.3 Algorithmes d'optimisation

Cette section donne une vue d'ensemble des méthodes du premier ordre en optimisation. Nous insistons sur la méthode de descente de gradient et ses généralisations pour les problèmes d'optimisation, qui ont typiquement la forme suivante

$$\begin{cases} \min & f(x) + g(x) \\ x \in \mathbb{R}^d \end{cases} \tag{13}$$

où la fonction-objectif est une somme de deux fonctions de natures différentes, comme pour (5) par exemple. La présentation des algorithmes se déroule en trois parties traitant tour à tour le cas d'une seule fonction différentiable, puis d'une somme de fonctions différentiables et non-différentiables, et enfin d'une somme de fonctions non-différentiables.

Fonction-objectif différentiable : algorithme du gradient Commençons par le cas basique où $g = 0$ et f est différentiable. Géométriquement, le gradient $\nabla f(x)$ en un point x donne la direction de la plus forte pente locale autour de x. Il est donc naturel de suivre cette direction pour essayer de diminuer la valeur f ; c'est ce que fait l'algorithme du gradient dont l'itération s'écrit

$$x_{k+1} = x_k - \gamma_k \nabla f(x_k). \tag{14}$$

Ici k est le compteur de l'itération et γ_k est un pas de descente, choisi pour garantir la convergence de l'algorithme. Le choix de ce pas de descente (appelé *taux d'apprentissage* dans les applications en apprentissage) est fondamental pour l'efficacité de l'algorithme : pour des mauvais choix de pas la

2. Mentionnons que les deux notions de convexité forte et différentiabilité forte sont reliées par une jolie *dualité*. Nous disposons en effet du résultat suivant

$$f \text{ est } L\text{-fortement différentiable} \iff f^* \text{ est } 1/L\text{-fortement convexe}$$

mettant en jeu la fonction convexe conjuguée f^* définie comme le résultat du problème d'optimisation suivant (paramétré par y) $f^*(y) = \max_x \langle x, y\rangle - f(x)$.

méthodes peut être extrêmement lente ou même ne pas converger (regarder par exemple la minimisation de $f(x) = x^2$ avec des pas constants $\gamma_k = 1$; regarder aussi $\gamma_k = 2$ et $\gamma_k = 1/2$). En pratique le coût en temps de calcul de cette iteration se réduit essentiellement au coût de calcul du gradient.

Cet algorithme est suffisamment simple pour admettre une analyse mathématique complète. En effet, il est possible d'analyser le nombre d'itérations nécessaires pour atteindre une solution approchée à ε près, pour toute fonction convexe f satisfaisant certaines hypothèses. Une hypothèse standard, vérifiée dans presque toutes les applications en analyse de données, est que la fonction objectif f est fortement différentiable avec une constante L que l'on peut parfois la calculer a priori et souvent estimer au fil des itérations des algorithmes. Sous cette hypothèse, on peut alors démontrer que la convergence est de l'ordre de $1/k$ en pire cas ; plus précisément, pour toute fonction f L-fortement différentiable et tout point initial x_0, l'algorithme du gradient (14) avec un pas constant $\gamma_k = 1/L$ produit à l'itération k un point x_k satisfaisant

$$f(x_k) - f_\star \leqslant \frac{2}{k+4} \|x_0 - x_\star\|_2{}^2.$$

Si la distance de l'itéré initial x_0 à une solution optimale $x_\star$ peut être bornée par une constante D, on obtient la garantie d'avoir une solution approchée à ε près, en $2D/\varepsilon - 4$ itérations (et ce, quelque soit f).

Sans aucun réglage supplémentaire, cet algorithme s'avère même plus rapide si la fonction est de plus μ-fortement convexe : l'algorithme du gradient (toujours avec le même pas $\gamma_k = 1/L$) converge *de manière exponentielle* vers l'unique solution optimale :

$$\|x_k - x_\star\| \leqslant (1 - \mu/L)^k \|x_0 - x_\star\|_2 \leqslant e^{-k\mu/L} \|x_0 - x_\star\|_2 \tag{15}$$

Ce résultat théorique révèle le ratio $\kappa = \mu/L \leqslant 1$ qui joue le rôle de *conditionnement* pour le problème de minimisation (d'une fonction μ-convexe et L-lisse) ; comme on le voit sur le dessin de la figure 3. Dans le cas d'une fonction quadratique $f(x) = x^\top Q x$, on montre que κ correspond au ratio des valeurs propres extrêmes de la matrice $\kappa = \lambda_{\min}(Q)/\lambda_{\max}(Q)$.

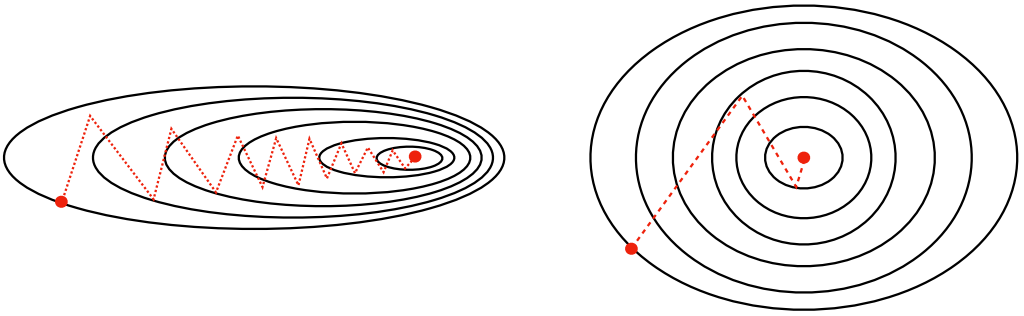

Figure 3 Lignes de niveau de deux fonctions de $\mathbb{R}^2$ dans $\mathbb{R}$ avec les itérés de l'algorithmes du gradient, illustrant la vitesse de convergence (15) et le rôle du conditionnement $\kappa = \mu/L$ Quand κ est grand (comme à droite), le problème est bien conditionné et l'algorithme converge vite ; plus le ratio est petit (comme à gauche), plus le problème est dur, et plus l'algorithme patine.

Ingrédients de base et recettes avancées L'algorithme de gradient est ainsi simple à comprendre et à coder. C'est en fait archétype des algorithmes d'optimisation, avec deux ingrédients essentiels : le pas de descente γ_k et la direction de descente donnée ici par le gradient. Chacun de ces deux ingrédients peut être amélioré :

– L'algorithme peut converger plus vite en choisissant des pas de descente particuliers pour une fonction ou une classe de fonctions. Par exemple, on peut améliorer la vitesse (15) dans le cas fortement

convexe en choisissant un pas dépendant de la constante de convexité forte [3]. En général, on peut encore mieux en calculant des γ_k adaptatifs, spécifique à une fonction et une itération particulière ; c'est ce qu'on appelle *recherche linéaire* en optimisation.

- D'autres directions de descente de descente sont meilleures et permettent d'éviter l'effet de zig-zag que l'on voit sur les figures ci-dessus. L'idée est d'incorporer dans l'algorithme de l'information supplémentaire sur le second-ordre de f, comme dans la méthode de gradient accéléré par inertie ou la méthode de Newton.

En particulier, les méthode de quasi-Newton (BFGS) à mémoire limitée, combinant les deux aspects ci-dessus (pas adaptatif et information supplémentaire), sont reconnues comme efficaces et sont utilisées depuis longtemps pour des problèmes avec données météorologiques. En général, les algorithmes du second-ordre (comme des méthodes de type Newton), incorporant de l'information sur la courbure de la fonction, demandent moins d'itérations pour atteindre une bonne précision. En contre-partie, elles nécessitent souvent de connaitre et de manipuler cette l'information supplémentaire (avec notamment des calculs plus coûteux à chaque itération), ce qui n'est rentable que si on exploite la structure particulière des problèmes d'optimisation en science des données (de grande taille et n'exigeant pas une précision fine).

Fonction composée : algorithme de gradient-proximal Considérons maintenant le problème (13) où f est différentiable et g est une fonction convexe non-différentiable simple. Dans ce cas, le gradient de la fonction-objectif n'existe plus partout et ainsi une application directe de l'algorithme du gradient (14) n'est pas possible (penser par exemple a la minimisation de la valeur absolue dans $\mathbb{R}$). Le fait que g soit « simple » permet de quand même définir un algorithme de premier ordre qui exploite le fait que l'objectif est une somme.

On dit que g est *prox-simple* si on peut aisément calculer (via une expression explicite peu coûteuse ou une sous-routine très efficace) le résultat du problème d'optimisation suivant, pour un point x fixé

$$\min_{z\in\mathbb{R}^d} \quad g(z) + \frac{1}{2}\|z-x\|^2. \tag{17}$$

Les exemples en apprentissage de telle fonctions sont légions : régularisation par norme ℓ_1 (qui correspond à la valeur absolue [4] en dimension 1), norme ℓ_1 par blocs, variation totale en une dimension, contraintes de type boite ou simplexe, ou avec peu de contraintes linéaires. Observons sur (17) que le terme quadratique additionnel garantit qu'il existe une unique solution pour tout paramêtre x ; ce qui permet de définir l'opérateur proximal de g noté $\mathrm{prox}_g\colon \mathbb{R}^d \to \mathbb{R}^d$ qui à x associe cette unique solution de (17).

L'algorithme de gradient-proximal consiste à faire un pas de gradient sur f suivi d'une correction par l'opérateur proximal associé à g, ce qui donne l'itération suivante :

$$x_{k+1} = \mathrm{prox}_{\gamma_k g}(x_k - \gamma_k \nabla f(x_k)). \tag{19}$$

3. En prenant un pas constant $\gamma_k = 2/(L+\mu)$ dépendant à la fois de L la constante de différentiabilité forte et de μ la constante de convexité forte, la vitesse exponentielle de algorithme du gradient peux être améliorée en

$$\|x_k - x_\star\| \leqslant \big(\frac{\kappa-1}{\kappa+1}\big)^k \|x_0 - x_\star\|_2 \leqslant \exp(-4k/(\kappa+1)))\|x_0 - x_\star\|_2. \tag{16}$$

Le gain est simplement de remplacer κ par $(\kappa+1)/4$, ce qui n'est pas spectaculaire, mais permet de retrouver dans le cas particulier de la minimisation de fonction quadratique un résultat bien connu en analyse numérique matricielle.

4. Par exemple, la fonction valeur absolue, non-dérivable en 0, est prox-simple : en explicitant les trois cas ($z=0$, $z>0$ et $z<0$), on montre que l'unique solution de (17) s'écrit

$$\mathrm{prox}_{|\cdot|}(\omega) = \begin{cases} 0 & \text{si } -1 < \omega < 1 \\ \omega - 1 & \text{si } \omega > 1 \\ w + 1 & \text{si } \omega < -1 \end{cases}$$

qui définit un opérateur de $\mathbb{R}$ dans $\mathbb{R}$, souvent appelé "soft-thresholding". On en déduit que la norme $\|\cdot\|_1$ est prox-simple aussi, avec

$$\mathrm{prox}_{\|\cdot\|_1}(x) = (\mathrm{prox}_{|\cdot|}(x_i))_{i=1,\dots,d} \ \in \mathbb{R}^d. \tag{18}$$

On voit ainsi que cet opérateur a pour effet de produire des vecteurs avec des composantes nulles (en annulant $x_i \in [-1,1]$). Cet effet explique l'intérêt de la régularisation par la norme 1 pour sélectionner les attributs les plus importants.

Notons que pour $g = 0$, l'opérateur proximal est l'identité, et on retombe sur l'algorithme du gradient discuté précédemment. Il se trouve que l'analyse de convergence de cet algorithme (ainsi que de ces extensions et accélérations) donne des résultats très similaires à ceux de l'algorithme du gradient rappelés juste avant.

Plaçons-nous finalement dans le cas général de (13) quand f et g sont tous deux est non-différentiables et prox-simples. Comme l'opérateur proximal d'une somme n'est pas simple en général, nous devons reformuler le problème pour pouvoir exploiter chaque opérateur proximal indépendement. La manipulation consiste à ajouter une variable supplémentaire $z \in \mathbb{R}^d$ pour reformuler (13) comme

$$\begin{cases} \min & f(x) + g(y) \\ & x = y. \end{cases}$$

Intuitivement, on peut alors faire une itération proximale sur chaque variable en ajoutant une étape qui connecte les deux, ce qui donne :

$$\begin{cases} x_{k+1} = \mathrm{prox}_{f/\rho}(-x_k + u_k/\rho) \\ y_{k+1} = \mathrm{prox}_{g/\rho}(x_k + u_k/\rho) \\ u_{k+1} = u_k + \rho(x_{k+1} - y_{k+1}) \end{cases} \tag{20}$$

pour $\rho > 0$ quelconque fixé. Cette itération correspond à celle d'un algorithme appelé ADMM (pour *alternating direction method of multipliers*[5]) dans un cas particulier.

Terminons en disant que si les fonctions non-différentiables apparaissant dans (13) ne sont pas prox-simples, d'autres familles d'algorithmes existent, parmi lesquelles les plus populaires en analyse de données sont :

- les méthodes de gradient conditionnel (appelé aussi Franck-Wolfe) ;
- les méthodes proximales dans des géométries favorables (utilisant la notion de *divergence de Bregman* pour mesurer l'éloignement).

Bilan sur les algorithmes d'optimisation de cette section Les algorithmes du premier ordre présentées dans cette section sont simples et permettent de présenter de nombreux aspects de l'optimisation (descente, pas, propriétés de convergence, rôle de la convexité, de la différentiabilité...). Ces algorithmes ont aussi de bonnes propriétés pour les passages à l'échelle présentées dans la section suivante. Notons enfin qu'ils se trouvent remarquablement robustes à l'utilisation d'approximation des gradients et des opérateurs proximaux, et s'accommodent des techniques de randomisation (par moyennage, échantillonnage, ou tirage de Monte-Carlo...).

4 Optimisation en science des données

Comme vu à la section 2, de nombreux problèmes d'optimisation en science des données s'écrivent comme la minimisation de sommes de fonctions du type

$$\min_{x \in \mathbb{R}^d} \quad \frac{1}{n} \sum_{i=1}^{n} f_i(x) + r(x). \tag{21}$$

Les fonctions $f_i \colon \mathbb{R}^d \to \mathbb{R}$ sont des fonctions similaires, concernant chacune une partie des données : typiquement, f_i mesure la qualité de la prédiction associée à x sur un bloc de données. Pour fixer les idées, on peut garder à l'esprit la version basique $f_i(x) = \ell\big((a_i, y_i), x\big)$ pour laquelle (10) correspond à (5). Dans le cas de blocs de données (comme par exemple lorsque les données sont stockées sur différentes machines), on se trouve alors avec des fonctions du type

$$f_i(x) = \frac{1}{|B_i|} \sum_{j \in B_i} \ell((a_j, b_j), x), \quad \text{où } B_i \text{ désigne un bloc de données.}$$

5. La variable u s'interprète en effet comme une variable duale. Une justification propre de cet algorithme utilise la dualité de l'optimisation convexe.

Ce problème d'optimisation (21) pourrait être vu comme étant de la forme (13) avec

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \quad \text{et} \quad g(x) = r(x) \tag{22}$$

et ainsi pourrait être résolu par les algorithmes dans la section 3.3. Malheureusement, une utilisation directe de ces algorithmes n'est pas approprié dans le cas de données volumineuses, pour lesquelles certaines opérations élémentaires, demandant l'utilisation de toutes les données, s'avèrent très coûteuses.

Illustrons ceci dans le cas où les f_i sont différentiables et r prox-simple, comme par exemple pour (9). On peut utiliser l'algorithme du gradient proximal (19) pour résoudre le problème (21), ce qui donne l'itération

$$x_{k+1} = \text{prox}_{\gamma_k r}\left(x_k - \frac{\gamma_k}{n}\sum_{i=1}^{n}\nabla f_i(x_k)\right). \tag{23}$$

Pour de nombreuses régularisations classiques, le calcul de l'opérateur proximal n'est pas trop coûteux, ni en temps de calcul, ni en mémoire (voir par exemple $\text{prox}_{\|\cdot\|_1}$ en (18)). La principale opération de cette itération est le calcul du gradient de la somme qui requiert une passe sur toutes les n données pour calculer les gradients $\nabla f_i(x_k)$ (auquel s'ajoute le coût de sommer n vecteurs de $\mathbb{R}^d$). C'est cette opération qui est coûteuse quand n et d sont grands, et qui rend cet algorithme peu efficace.

L'idée pour passer à l'échelle sur des données volumineuses est de traiter chaque bloc de données indépendemment, plutôt que de considérer toutes les données en même temps. Cette stratégie se dérive en deux types [6] de méthodes : les algorithmes incrémentaux qui exploitent chaque bloc de données un par un (section 4.1) et les algorithmes distribués, adaptés au cas où les données sont ensemble mais stockées sur des machines différentes (section 4.2).

4.1 Algorithmes incrémentaux

Les algorithmes incrémentaux opèrent une mise à jour à chaque lecture d'un bloc de données (au lieu de tous les utiliser en même temps). Nous présentons le plus simple d'entre eux, l'algorithme du gradient incrémental ou stochastique, puis nous discutons ses extensions.

Une donnée, une itération Plaçons-nous dans cette section dans le cas basique de la minimisation de fonctions différentiables ($r = 0$ dans (21)). Dans ce cas, on pourrait utiliser l'algorithme du gradient

$$x_{k+1} = x_k - \frac{\gamma_k}{n}\sum_{i=1}^{n}\nabla f_i(x_k). \tag{24}$$

Comme expliqué précédemment pour (23), l'inconvénient de cet algorithme est qu'il nécessite de faire, à chaque itération, un passage sur toutes les données (pour calculer tous les $\nabla f_i(x_k)$). L'idée est alors de faire une itération en utilisant un seul bloc de données et ainsi un seul $\nabla f_i(x_k)$.

A l'itération k, on choisit un indice i_k, on calcule $\nabla f_{i_k}(x_k)$ sur le bloc de données correspondant à i_k, et on procède à la mise à jour

$$x_{k+1} = x_k - \gamma_k \nabla f_{i_k}(x_k) \qquad i_k \in \{1, \dots, n\}. \tag{25}$$

Cette itération (25) correspond à l'algorithme du gradient incrémental, dépendant du choix de i_k (qui peut faire de manière cyclique, gloutonne ou aléatoire). En exploitant la redondance [7] inhérente des données, cet algorithme permet d'obtenir une itération de minimisation pour un coût de calcul nettement moindre que dans (24) : le calcul de $\nabla f_{i_k}(x_k)$ est bien moins coûteux que celui du gradient entier $\nabla f(x)$ (typiquement n fois moins coûteux).

6. Nous séparons les deux types pour en présenter les idées sous-jacentes ; dans les applications pratiques, il y a souvent un mélange des deux.

7. Considérons une situation dégénérée qui permet de bien comprendre l'idée. Supposons que notre ensemble de données est constitué de dix copies identiques du même bloc de données ; autrement dit, qu'il y a dix f_i tous égaux. Dans ce cas, l'algorithme (25) donne, pour un coût de calcul n fois plus petit, exactement la même itération que l'algorithme du gradient (24). Evidemment, en pratique les données ne sont pas dupliquées comme dans cette situation. Mais beaucoup de problèmes d'analyse de données contiennent une bonne partie de données redondantes, ce qui rend cette approche intéressante.

Optimisation par gradient stochastique Dans le cas où i_k est aléatoire et choisi uniformément dans $\{1, \dots, n\}$, l'itération (24) s'appelle aussi gradient stochastique : on observe en effet que l'espérance de la direction de descente (aléatoire) correspond au gradient

$$\mathbb{E}[\nabla f_{i_k}(x_k)] = \nabla f(x_k). \tag{26}$$

Utiliser des directions de descente aléatoires amène une décroissance aléatoire (voir la figure 4). On peut tout de même contrôler la convergence un prenant un pas γ_k qui tend vers 0 mais pas trop vite, typiquement $\gamma_k = 1/k$. On démontre alors la convergence (presque sûre) vers un point $\bar{x}$ tel que $\nabla f(\bar{x}) = 0$ (qui est un minimum si f est convexe).

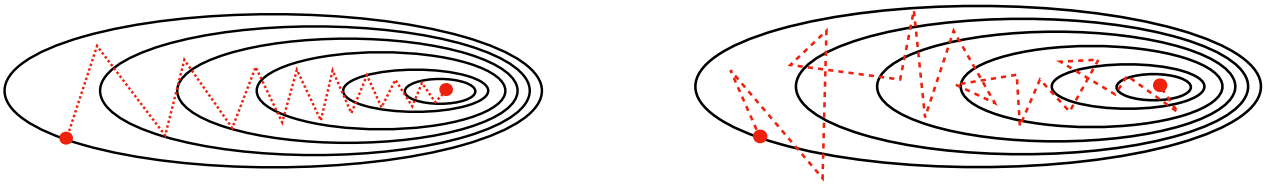

Figure 4 Itérations des algorithmes du gradient (à gauche) et gradient stochastique (à droite) sur les lignes de niveau d'une fonction quadratique. L'espérance des directions de descente correspond à la direction du gradient, et on récupère des garanties de convergence si on peut aussi contrôler la variance des directions.

Prendre des pas de plus en plus petits entraine néanmoins que cet algorithme converge lentement. Par apport à l'algorithme du gradient, on est ainsi dans la situation opposée concernant l'équilibre "coût de calcul" vs "vitesse de convergence" :

- pour l'algorithme du gradient (24), on a un coût d'itération proportionel à n, mais une vitesse de convergence de l'ordre de $e^{-k/\kappa}$) dans le cas fortement convexe (voir (15)) ;
- pour l'algorithme du gradient stochastique avec pas $\gamma_k = 1/k$, on a un coût d'itération indépendant de n, mais on démontre une vitesse de convergence de l'ordre κ/k uniquement.

Ainsi même si chaque itération est bien moins coûteuse que pour la méthode du gradient, l'algorithme gradient stochastique n'est pas efficace en fin de convergence. Son intérêt réside dans ces premières itérations : on observe en effet systématiquement une décroissance initiale rapide. En pratique, il est encore une fois fondamental de régler correctement [8] le pas de descente γ_k pour observer une bonne convergence empirique. Au delà de ces réglages empiriques, d'autres techniques de réduction de variance, ont été proposées pour améliorer les performances de l'algorithme de gradient stochastique.

Réduction de l'aléatoire par moyennes Il existe des alternatives à l'utilisation de pas décroissants pour contrôler la variance dans (26) par moyennage [9]. Une première technique classique est de considérer les moyennes des itérés successifs produits par (25). Ce moyennage

$$\bar{x}_k = \frac{1}{k} \sum_{\ell=1}^{k} x_\ell$$

amène une certaine stabilité de la suite $(\bar{x}_k)_k$ et permet ainsi de prendre des pas plus grands. Une seconde technique plus récente est de faire des moyennes des gradients successifs. Il s'agit de garder en mémoire certains gradients calculés aux itérations précédentes et de les combiner pour obtenir une meilleure descente. Par exemple, on peut garder en mémoire un gradient par fonction f_i : à l'itération k, on tire aléatoirement l'indice i_k, on calcule $\nabla f_{i_k}(x_k)$, et on met à jour la mémoire des n gradients stockés

$$g_i^k = \nabla f_i(x_k) \text{ si } i = i_k \quad \text{et} \quad g_i^k = g_i^{k-1} \text{ sinon.}$$

8. Deux recettes à garder en tête : prendre un pas initial modéré par la constante de Lipschitz pour garantir une décroissance dès les premières itérations ; ensuite prendre des pas le plus grands possible (dans les limites de résultats généraux de convergence ou mème au-delà si on a besoin de faire décroitre rapidement l'objectif).

9. Il existe aussi des méthodes de réduction de variance utilisant la dualité

L'itération de l'algorithme appelé SAG (pour "stochastic averaged gradient") consiste alors à prendre la moyenne des gradients en mémoire

$$x_{k+1} = x_k - \frac{\gamma_k}{n} \sum_{i=1}^{n} g_i^k.$$

On voit que cette itération peut se réécrire de la manière suivante en remplaçant la vecteur gradient i_k dans la la moyenne des g_i^k

$$x_{k+1} = x_k - \gamma_k \Big(\frac{1}{n} \sum_{i=1}^{n} g_i^k + \frac{1}{n} (\nabla f_{i_k}(x_k) - g_{i_k}^k) \Big)$$

De simples modifications de cette itération amènent à des algorithmes efficaces, ayant de plus de bonnes garanties théoriques. Par exemple [10], en supprimant le $1/n$ (ce qui s'interprète comme insister sur l'itération courante), on obtient l'itération de l'algorithme appelé SAGA

$$\begin{aligned} x_{k+1} &= x_k - \gamma_k \Big(\frac{1}{n} \sum_{i=1}^{n} g_i^k + (\nabla f_{i_k}(x_k) - g_{i_k}^k) \Big) \\ &= x_k - \gamma_k \nabla f_{i_k}(x_k) + \gamma_k \Big(\frac{1}{n} \sum_{i=1}^{n} g_i^k - g_{i_k}^k \Big) \end{aligned} \tag{27}$$

Avec une meilleure direction de descente stochastique que celle de (25), cet algorithme admet des pas de descente plus grands et s'avère ainsi plus rapide. Dans le cas μ-fortement convexe et L-fortement différentiable, on montre que (27) avec un pas constant $\gamma_k = \frac{1}{2(\mu d + L)}$ admet la convergence exponentielle suivante (en espérance car c'est un algorithme aléatoire), qui très proche de celle de l'algorithme du gradient (16),

$$\mathbb{E}[\|x_k - x_\star\|^2] \leqslant C \exp\left(\frac{-k\mu}{2(\mu d + L)} \right)$$

avec une constante C dépendant de x_0 et $x_\star$.

Bilan sur les algorithmes incrémentaux Les méthodes incrémentales, comme l'algorithme du gradient stochastique, sont efficaces lors des premières itérations pour faire décroitre l'objectif rapidement sans avoir à exploiter la totalité des données, mais elles ralentissent très vite à cause d'un taux d'apprentissage décroissant imposé par la théorie. En acceptant un coût supplémentaire en mémoire et en calcul, elles peuvent être modifiées pour accepter des pas constants et ainsi afficher une meilleure convergence en pratique et en théorie.

Ces méthodes marchent particulièrement bien pour minimiser des fonctions fortement convexes, ce qui incite à ajouter, si nécessaire, une régularisation quadratique dans les modèles d'apprentissage, ce qui revient à prendre $f_i(x) = \ell\big((a_i, y_i), x\big) + \lambda_2 \|x\|_2{}^2$. Dans tous les cas, le réglage empirique des différents paramètres des algorithmes (notamment du pas de descente) requiert une attention particulière car il impacte fortement le comportement observé des algorithmes.

A retenir : l'utilisation de l'aléatoire [11] dans ces algorithmes incrémentaux permet de récupérer de l'information pertinente en ne traitant qu'une partie des données. Néanmoins, cet aléa requiert de pouvoir accéder rapidement à n'importe quelle partie des données. Ceci implique des restrictions sur les systèmes informatiques : les différents machines de calcul doivent avoir une mémoire partagée contenant les données (en plus de ressources de calcul partagées). Si ce n'est pas le cas, d'autres algorithmes d'optimisation (déterministes) existent ; ils sont présentés à la section suivante.

10. Il existe d'autres méthodes par moyennage, dont une, appelée SVRG admet une implémentation avec faible mémoire, allégeant ainsi le coût supplémentaire en mémoire de n vecteurs de $\mathbb{R}^d$ des méthodes présentées ici.

11. D'autres méthodes aléatoires sont utiles en optimisation en grande dimension, comme par exemple : la mise à jour d'un bloc aléatoire de variables, l'approximation des gradients ou des opérateurs proximaux par des estimations stochastiques, ou l'accélération des routines de calcul matriciel par randomisation.

4.2 Algorithmes distribués

Dans cette section, nous considérons la situation où les données ne peuvent être stockées au même endroit, par exemple pour des raisons techniques (données trop volumineuses) ou des raisons juridiques (données privées, comme par exemple les données des téléphones portables). Dans ce cas, les données sont stockées sur différentes machines et la résolution d'un problème d'apprentissage (21) doit se faire avec les algorithmes d'optimisation distribuée. Les questions reliées aux communications entre machines deviennent alors prépondérantes pour l'efficacité de ces algorithmes.

Pour simplifier, plaçons-nous dans le cas où l'on dispose d'une architecture de calcul centralisée avec une machine-maître qui dirige m machines-esclaves ; et pour fixer les idées, on considère que $m = n$, c'est-à-dire qu'il y a autant de machines que de blocs de données (et de fonctions f_i). L'indice i réfère maintenant à la fois au bloc de donnée i comme précédemment, mais aussi à la machine qui le stocke.

Il se trouve que les algorithmes du premier ordre rappelés en section 3.3 s'étendent naturellement dans ce cadre distribué, où les calculs coûteux sont gérés en local sur chaque machine, puis les résultats sont reliés entre eux par une étape de consensus. Cette section présente brièvement les idées de ces algorithmes distribués, d'abord dans le cas où les f_i sont différentiables, puis dans le cas de f_i non-différentiables.

Calculer le gradient en parallèle Reprenons le problème d'optimisation (21), qui est de la forme (13) avec les fonctions (22). Supposons tout d'abord que les fonctions d'attache aux données f_i sont différentiables, ce qui implique que f l'est aussi avec

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x). \tag{28}$$

On peut alors utiliser l'algorithme du gradient proximal qui donne l'itération (23), dont l'opération coûteuse est le calcul de la somme des gradients. Dans le cadre distribué de cette section, cette itération se prête bien au calcul parallèle :

1. la machine-maitre diffuse la variable x_k aux machines-esclaves ;
2. le calcul des n gradients $\nabla f_i(x_k)$ dépendant des données se fait en parallèle sur ces n machines ;
3. la machine-maître centralise les résultats des calculs pour calculer x_{k+1}, en faisant la moyenne des gradients, suivie de l'opérateur proximal de r.

Ce schéma de calcul entre dans le cadre d'un formalisme *map-reduce* où la phase de "map" distribue le calcul coûteux sur toutes les machines et la phase de réduction synchronise les résultats de calculs en opérant uniquement des opérations simples. Soulignons de plus que l'algorithme ne communique que des résultats de calculs, et pas de données brutes.

Ce raisonnement, présenté ici dans un cadre centralisé, s'étend à des systèmes informatiques plus complexes. Dans le cas général, des logiciels big data permettent de faire ce map-reduce sans trop se soucier du système sous-jacent. Par exemple, `Spark` permet de programmer de la même manière le cas $m < n$ que le cas $n = m$ considéré ici.

Calculer l'opérateur proximal en parallèle (après reformulation) Toujours dans le cadre distribué avec le problème d'optimisation (21) (de la forme (13) avec (22)), supposons à présent que les fonctions f_i sont non-différentiables et prox-simples. Contrairement à la situation précédente, on n'est pas directement dans une situation de la section 3.3 : en effet, f n'est pas prox-simple, car l'opérateur proximal d'une somme ne correspond pas à la somme des opérateurs proximaux (contrairement aux gradients (28) dans le différentiable). On ne peut donc pas distribuer les calculs en appliquant directement l'algorithme ADMM (20) adapté à la minimisation de la somme de deux fonctions prox-simples. Une manière de contourner cette interdiction consiste à reformuler (21), comme suit, sous une forme distribuable impliquant des fonctions prox-simples.

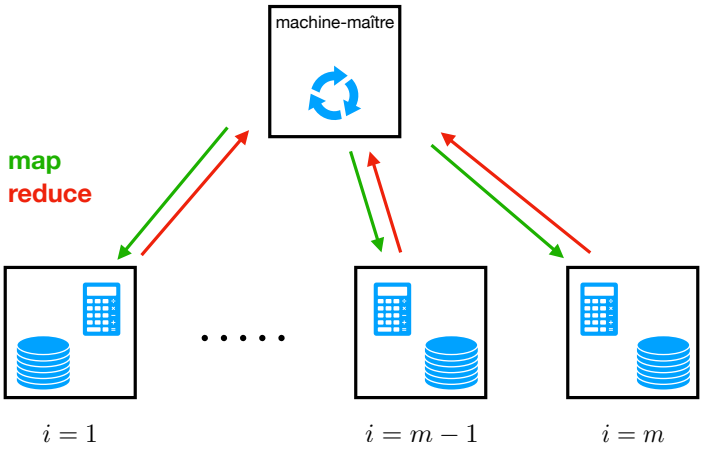

Figure 5 Schéma du cadre d'optimisation distribuée : une machine-maître coordonne les calculs effectués en parallèle sur les machines-esclaves, stockant chacune une partie des données. Soulignons que les algorithmes ne communiquent que des résultats de calcul, pas des données brutes.

En donnant à chaque f_i une copie de x, notée x_i, le problème (21) peut s'écrire de manière équivalente comme

$$\left\{ \begin{array}{ll} \min\limits_{(x_1,\dots,x_n)\in\mathbb{R}^{d\times n}} & \frac{1}{n}\sum\limits_{i=1}^{n} f_i(x_i) + r(x) \\ x_1 = \cdots = x_n. & \end{array} \right. = \frac{1}{n}\sum_{i=1}^{n} \big(f_i(x_i) + r(x_i)\big) \tag{29}$$

En introduisant la fonction indicatrice de consensus $\delta\colon \mathbb{R}^{d\times n} \to \mathbb{R}\cup\{+\infty\}$ (définie par $\delta\big((x_1,\dots,x_n)\big) = 0$ si $x_1 = \cdots = x_n$, et $+\infty$ sinon), on voit que ce problème est de la forme (13) avec

$$f\big((x_1,\dots,x_n)\big) = \frac{1}{n}\sum_{i=1}^{n} f_i(x_i)$$
$$g\big((x_1,\dots,x_n)\big) = \frac{1}{n}\sum_{i=1}^{n} r(x_i) + \delta\big((x_1,\dots,x_n)\big).$$

On montre, sans trop de difficultés, à partir de la définition (17), que ces deux fonctions sont prox-simples et que leur opérateurs proximaux "s'éclatent" sur chaque x_i et s'écrivent uniquement à l'aide des prox_{f_i} et de prox_r. On peut maintenant appliquer l'algorithme ADMM (20) avec les bonnes fonctions ; ce qui donne une itération éclatée sur chaque machine :

$$\left\{ \begin{array}{ll} (x_i)_{k+1} = \mathrm{prox}_{f_i/n\rho}(-y_k + (u_i)_k/\rho) & \text{pour toute machine } i \\ y_{k+1} = \mathrm{prox}_{r/n\rho}\big(\frac{1}{n}\sum_{i=1}^{n}((x_i)_{k+1} + (u_i)_k/\rho)\big) & \\ (u_i)_{k+1} = (u_i)_k + \rho((x_i)_{k+1} - y_{k+1}) & \text{pour toute machine } i. \end{array} \right.$$

Cet algorithme entre dans le cadre map-reduce, comme précédemment : la mise a jour des $(u_i)_k$ et $(x_i)_k$ correspondant au map, et celle de y_k au reduce. L'opération coûteuse qui dépend des donnés (le calcul des $\mathrm{prox}_{f_i/n\rho}$) est ainsi effectué efficacement en parallèle sur chaque machine.

Bilan sur les algorithmes distribués La distribution et la parallélisation des calculs dans les algorithmes d'optimisation du premier ordre se fait de manière naturelle dans un cadre d'une plateforme de calcul centralisée. On peut aussi étendre ces algorithmes à des architectures plus évoluées (avec des versions asynchrones ou décentralisées des algorithmes) ; les contraintes sur les communications entre machines sont alors importantes à prendre en compte. Soulignons que ces questions sur l'adaptation des algorithmes d'optimisation à l'hétérogénéité des plateformes de calculs sont au coeur de défis actuels en optimisation pour le big data.

Références

[BCN17] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv :1606.04838*, 2017.

[Bub15] Sébastien Bubeck. Convex optimization : Algorithms and complexity. *Foundations and Trends in Machine Learning*, 8(3-4) :231–357, 2015.

[BV04] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[HUL01] J.-B. Hiriart-Urruty and C. Lemaréchal. *Fundamentals of Convex Analysis*. Springer Verlag, Heidelberg, 2001.

[Nes13] Y. Nesterov. *Introductory lectures on convex optimization : A basic course*, volume 87. Springer Science & Business Media, 2013.