POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Exploiting the Partially-Separable Structure in Quasi-Newton Methods for Unconstrained Optimization and Deep Learning
Auteur: Author:	Paul Raynaud
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Raynaud, P. (2024). Exploiting the Partially-Separable Structure in Quasi-Newton Methods for Unconstrained Optimization and Deep Learning [Ph.D. thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58354/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/58354/
Directeurs de recherche: Advisors:	Dominique Orban, & Jean Bigeon
Programme: Program:	Doctorat en mathématiques

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL ET UNIVERSITÉ GRENOBLE ALPES
affiliée à l'Université de Montréal

Exploiting the Partially-Separable Structure in Quasi-Newton Methods for Unconstrained Optimization and Deep Learning

PAUL RAYNAUD
Département de mathématiques et génie industriel
Polytechnique Montréal
et
Département de génie industriel
Université Grenoble Alpes

Thèse en cotutelle présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques et génie industriel

Mai 2024

© Paul Raynaud, 2024.

POLYTECHNIQUE MONTRÉAL ET UNIVERSITÉ GRENOBLE ALPES
affiliée à l'Université de Montréal

Cette thèse intitulée :

Exploiting the Partially-Separable Structure in Quasi-Newton Methods for Unconstrained Optimization and Deep Learning

présentée par **Paul RAYNAUD**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Sébastien LE DIGABEL, président
Jean BIGEON, membre et codirecteur de recherche
Dominique ORBAN, membre et codirecteur de recherche
Pierre LEMAIRE, membre
Margherita PORCELLI, membre externe
Sébastien BOURGUIGNON, membre externe

DEDICATION

À ma maman,
elle qui m'a montré l'exemple :
flancher n'est pas une option.

ACKNOWLEDGEMENTS

Je tiens tout d'abord à remercier mes (co-)directeurs de recherche : Jean Bigeon et Dominique Orban. Ces quatre années de doctorat passées ensemble auront été riches d'enseignements, autant du point de vue technique qu'humain. Nos innombrables échanges m'auront montré ce qu'est l'excellence académique de la recherche et m'auront préparé pour les tâches auxquelles je serai confronté dans les futurs chapitres de ma vie.

Je tiens également à exprimer ma gratitude envers Sébastien Le Digabel, Margherita Porcelli, Sébastien Bourguignon et Pierre Lemaire qui ont accepté d'être les membres de mon jury. Je n'oublie pas Jean-Marie Flauss, Christophe Jermann et Raphaël Chenouard dont les présences à mes CSI auront jalonné ma thèse.

Cette thèse n'aurait pas été la même si elle n'avait pas été une cotutelle entre l'école Polytechnique Montréal et l'université Grenoble-Alpes, représentées respectivement par le GERAD et GSCOP. Pour cette raison, je tiens à remercier chaleureusement le personnel administratif du département MAGI et du GERAD ainsi que celui de l'école doctorale IMEP2 et de GSCOP. Également, je souhaite remercier sincèrement l'alliance HUAWEI/NSERC-CRSNG ainsi que Grenoble INPG-SA pour leur soutien financier tout au long de ma thèse.

Cette expérience en cotutelle m'a offert l'opportunité de découvrir Montréal, une ville que j'ai apprise à chérir, tantôt de glace tantôt incandescente, et, où parmi tout et son contraire, j'ai réussi à aimer autrui et à me construire seul. Évidemment, cette ville serait différente sans les personnes formidables qui l'habitent.

Je pense en premier à mes compagnons de JSO : Tangi, Dominique M., Antonin et bien entendu Alexis, un ami doublé d'un représentant syndical d'exception. Je ne peux également pas oublier de mentionner les volleyeurs : Franck, Charley, Justas et Vicky, l'été s'annonce chauuuddd !! Je sens que les Grenoblois pourraient se sentir jaloux, bien sûr que je ne vous oublie pas Yara, Hugo et Boubou, sans oublier Jules le dernier doigt de la main.

Et sans surprise la belle gang du Datcha : Adrien, mon premier Alsacien québécois, Qi Hao, le retardataire invétéré mais bout en train inégalé, Laurianne, la chokeuse régulière aux anecdotes troublantes qu'on prend toujours plaisir à voir ; puis mes deux piliers : Geoffroy, mon prudent mais meilleur partenaire de pepper, et Romain, ma bouée et mon phare dans l'océan, merci à vous deux de m'avoir montré le chemin à suivre.

Finalement, j'ai une pensée profonde pour ma mère et ma sœur. Bien que nous ne soyons plus sur le même continent, je pense que nous n'avons jamais été aussi proches, je vous aime.

RÉSUMÉ

Cette thèse explore l'utilisation de la structure partiellement-séparable pour l'optimisation continue sans contrainte, en particulier pour les méthodes quasi-Newton et l'entraînement de réseaux de neurones.

Une fonction partiellement-séparable est la somme de fonctions éléments, chacune de dimension inférieure à celle du problème total. Le Hessien peut être agrégé en approximant séparément le Hessien de chaque fonction élément avec une matrice dense. Cela préserve la structure creuse du Hessien, contrairement aux méthodes quasi-Newton à mémoire limitée. En pratique, ces méthodes nécessitent moins d'itérations qu'une méthode quasi-Newton à mémoire limitée et peuvent être parallélisées en distribuant les calculs liés aux fonctions éléments.

Cependant, la revue de littérature de la thèse révèle certaines limites lorsque la dimension des fonctions éléments est grande. De plus, le seul logiciel d'optimisation libre exploitant la structure partiellement-séparable[1] est inutilisable pour les utilisateurs inexpérimentés sans l'utilisation d'un langage de modélisation commercial.

Dans cette thèse, des solutions sont proposées pour remédier à ces lacunes, ainsi qu'une application des concepts d'optimisation partiellement-séparable à l'apprentissage supervisé d'un réseau de neurones.

La première contribution est une suite logicielle libre basée sur une détection automatique de la structure partiellement-séparable d'un problème, c'est-à-dire qu'elle retrouve chaque fonction élément de dimension réduite. Elle alloue des structures de données partitionnées nécessaires pour stocker les dérivées (approximées) et définit des méthodes d'optimisation quasi-Newton (à mémoire limitée) partitionnées. L'ensemble de la suite est intégré à l'écosystème JuliaSmoothOptimizers[2], qui rassemble de nombreux outils pour l'optimisation continue, notamment des algorithmes d'optimisation qui peuvent exploiter la séparabilité partielle détectée.

Dans la deuxième contribution, l'approximation d'un Hessien élément par une matrice dense est remplacée par un opérateur linéaire quasi-Newton à mémoire limitée. Par conséquent, le coût en mémoire pour stocker et effectuer un produit entre l'approximation du Hessien et un vecteur n'est plus quadratiquement lié à la dimension des fonctions éléments. Une telle approximation partitionnée à mémoire limitée est alors applicable lorsque les fonctions éléments sont grandes, contrairement à une approximation partitionnée dense. La norme

[1] LANCELOT [33]
[2] détaillé dans [110]

de chaque nouvel opérateur quasi-Newton partitionné à mémoire limitée peut être bornée, garantissant que sa méthode de région de confiance relative converge. De plus, les résultats numériques montrent que ces méthodes surpassent les méthodes quasi-Newton partitionnées ou à mémoire limitée lorsque les fonctions éléments sont larges.

La dernière contribution examine l'exploitation de la structure partiellement-séparable dans l'entraînement supervisé d'un réseau de neurones. En général, le problème d'optimisation associé à l'entraînement n'est pas partiellement-séparable. Par conséquent, une fonction de perte partiellement-séparable et une architecture de réseau de neurones partitionnée sont introduites pour rendre l'entraînement partiellement-séparable. Les résultats numériques montrent qu'une combinaison de ces deux contributions est compétitive avec les architectures et fonctions de perte usuelles. L'entraînement de cette combinaison peut être réalisé en parallèle, de manière complémentaire aux techniques d'apprentissage supervisé parallèles existantes. Plus spécifiquement, les calculs de chaque fonction de perte élément peuvent être distribués à un travailleur qui opère uniquement sur un fragment du réseau de neurones. Le problème d'entraînement partiellement-séparable possède des larges éléments pour lesquels un entraînement quasi-Newton partitionné à mémoire limitée est proposé. Cet entraînement est empiriquement montré comme compétitif avec les méthodes d'entraînement de l'état de l'art et légèrement inférieur à Adam.

ABSTRACT

This thesis investigates the use of the partially-separable structure for continuous unconstrained optimization, in particular for quasi-Newton methods and neural network training.

A partially-separable function is the sum of element functions, each of lower dimension than the total problem. The Hessian can be aggregated by separately approximating the Hessian of each element function with a dense matrix. This preserves the sparsity pattern of the Hessian, in contrast to limited-memory quasi-Newton methods. In practice, these methods require fewer iterations than a limited-memory quasi-Newton method and can be parallelized by distributing the computations related to the element functions.

However, the literature review of the thesis reveals some limitations, particularly when the dimension of the element functions is large. Additionally, the only open-source optimization software exploiting the partially-separable structure[3] is unusable for inexperienced users without using a commercial modelling language.

In this thesis, solutions are proposed to address these shortcomings, along with an application of partially-separable optimization concepts to supervised learning of a neural network.

The first contribution is an open source software suite based on an automatic detection of the partially-separable structure of a problem, i.e., retrieves each reduced-dimensional element function. It allocates partitioned data structures necessary for storing (approximated) derivatives and defines (limited-memory) partitioned quasi-Newton optimization methods. The entire suite is integrated into the JuliaSmoothOptimizers[4] ecosystem, which gathers numerous tools for continuous optimization, including optimization algorithms that can exploit the detected partial separability.

In the second contribution, the approximation of an element Hessian by a dense matrix is replaced with a limited-memory quasi-Newton linear operator. As a result, the memory cost for storing and performing a Hessian-approximation vector product is no longer quadratically related to the dimension of the element functions. Such a limited-memory partitioned approximation is then applicable when the element functions are large, conversely to a dense partitioned approximation. The norm of each new limited-memory partitioned quasi-Newton operator can be bounded, ensuring that its relative trust-region method converges. Additionally, numerical results show that these methods outperform partitioned or limited-memory quasi-Newton methods when the elements are large.

[3] LANCELOT [33]

[4] detailed in [110]

The final contribution examines the exploitation of the partially-separable structure in the supervised training of a neural network. In general, the optimization problem associated with training is not partially-separable. Therefore, a partially-separable loss function and a partitioned network architecture are introduced to make the training partially-separable. Numerical results show that a combination of these two contributions is competitive with state-of-the-art architectures and loss functions. The training of this combination can be carried out in parallel, complementarily to existing parallel supervised learning techniques. More specifically, the calculations of each element loss function can be distributed to a worker who only needs to operate a fragment of the neural network. The partially-separable training problem possesses large elements for which a limited-memory partitioned quasi-Newton training is proposed. This training is empirically shown to be competitive with state-of-the-art training methods and slightly inferior to Adam.

RÉSUMÉ DE LA THÈSE

Contexte

Cette thèse est centrée sur l'utilisation de la structure partiellement-séparable durant la minimisation d'un problème continu sans contrainte :

$$\min_{x \in \mathbb{R}^n} f(x)$$

où $f : \mathbb{R}^n \to \mathbb{R} \in \mathcal{C}^2$ est *partiellement-séparable*, i.e., f somme des fonctions éléments de moindre dimension :

$$f(x) = \sum_{i=1}^{N} \widehat{f}_i(\widehat{x}_i), \quad \widehat{x}_i \ := U_i x, \tag{1}$$

où chaque $U_i \in \mathbb{R}^{n_i \times n}, n_i < n$ sélectionne les variables paramétrant la fonction élément $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$. Le gradient et le Hessien s'expriment de la forme suivante :

$$\nabla f(x) = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i(\widehat{x}_i), \quad \nabla^2 f(x) = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_i) U_i,$$

où $\nabla \widehat{f}_i \in \mathbb{R}^{n_i}$ et $\nabla^2 \widehat{f}_i \in \mathbb{R}^{n_i \times n_i}$ sont respectivement le gradient et le Hessien de la fonction élément $\widehat{f}_i$. [76, Théorème 1] justifie l'intérêt de la séparabilité partielle en démontrant que chaque fonction possédant un Hessien creux est partiellement-séparable. La nature partitionnée des dérivées sera exploitée par les méthodes quasi-Newton qui seront ensuite appliquées à l'entraînement d'un réseau de neurones. Note: la notation donnée par (1) est complète, mais diffère de celle initiallement proposée au début du chapitre 1. Dans le chapitre 1, les notions liées à la séparabilité partielle sont introduites pas à pas, partant ainsi d'une définition plus générale qui est affinée par la suite pour trouver (1).

De manière générale, l'utilisation du Hessien par les méthodes de Newton est considérée comme trop coûteuse pour un problème de grande dimension. Les méthodes quasi-Newton permettent de pallier ce problème en approximant de manière itérative $\nabla^2 f(x_k)$ par des matrices $B_k = B_k^\top$ mises à jour par les évaluations de $\nabla f(x_k)$ [23, 41, 101]. Historiquement, l'approximation quasi-Newton d'un Hessien est réalisée par le biais d'une matrice dense, rendant l'approximation inapplicable pour un problème de grande dimension. Les approximations quasi-Newton sont néanmoins applicables à ces problèmes par l'introduction des variantes à mémoire limitée. Ces variantes approximent à partir d'un faible nombre de paires de vecteurs $(x_{k+1} - x_k, \nabla f(x_{k+1}) - \nabla f(x_k))$ le produit Hessien-vecteur par un opérateur linéaire $v \to B_k v \approx v \to$

$\nabla^2 f(x_k)v$ [45, 101]. Cependant, la construction d'une approximation quasi-Newton ne permet pas de conserver la structure creuse du Hessien. Lorsque la fonction f est partiellement-séparable, on peut faire l'approximation

$$B_k = \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i, \tag{2}$$

en agrégeant les approximations du Hessien de chaque fonction élément $\widehat{B}_{i,k} = \widehat{B}_{i,k}^\top \approx \nabla^2 \widehat{f}_i(\widehat{x}_{i,k}) \in \mathbb{R}^{n_i \times n_i}$, $\widehat{x}_{i,k} \coloneqq U_i x_k$. Ces méthodes quasi-Newton partitionnées sont applicables aux problèmes de grandes dimensions et conservent la structure creuse du Hessien [28, 75, 76, 104, 107], contrairement à une méthode quasi-Newton à mémoire limitée. En pratique, ces méthodes réalisent moins d'itérations qu'une méthode quasi-Newton à mémoire limitée [75, 101, 105, 148], et sont facilement parallélisables en distribuant les calculs attribués à chaque fonction élément [27, 58, 91, 99, 116, 136].

Le chapitre 2 rappelle des concepts liés aux méthodes quasi-Newton et le chapitre 3 est un état de l'art sur la minimisation partiellement-séparable. En particulier, la section 3.1 détaille les propriétés d'une fonction partiellement-séparable et les illustre par des exemples. La section 3.2 décrit les principales méthodes quasi-Newton partitionnées ainsi que leurs résultats de convergence. Par la suite, les sections 3.3 et 3.4 regroupent différentes méthodes d'optimisation continue exploitant la séparabilité partielle. Les sections 3.5 à 3.7 précisent comment l'exploitation de la séparabilité partielle permet l'implémentation de routines numériques spécifiques, aboutissant à des logiciels dédiés. La section 3.8 présente plusieurs méthodes d'optimisation sans dérivées incorporant la structure partiellement-séparable. La section 3.9 conclut par une analyse critique de l'état de l'art.

L'analyse critique relève deux principaux facteurs limitant le développement des méthodes quasi-Newton partitionnées. Premièrement, les $\widehat{B}_{i,k}$ sont denses, ce qui nécessite que les n_i soient petits pour mémoriser B_k avec peu de mémoire. Lorsque les n_i sont larges, mémoriser chaque $\widehat{B}_{i,k}$ devient coûteux. Ainsi, lorsque f possède des fonctions éléments de grande dimension, l'approximation B_k d'une méthode quasi-Newton partitionnée devient (très) coûteuse en mémoire. Deuxièmement, il manque un logiciel libre détectant automatiquement la séparabilité partielle d'une fonction f. En effet, le logiciel libre Fortran LANCELOT [33] extrait la séparabilité partielle du format SIF (Standard Input Format [34]), dont la modélisation nécessite le renseignement de la structure par l'utilisateur; tandis que AMPL [62] est un langage de modélisation commercial. Cette thèse apporte une solution pour chacun de ces problèmes, ainsi qu'une application des concepts de séparabilité partielle à l'apprentissage profond. Le chapitre 4 propose une méthode quasi-Newton partitionnée à mémoire limitée

supportant les fonctions éléments de grande dimension. Le chapitre 5 applique les méthodes développées durant le chapitre 4 à l'entraînement d'un réseau de neurones dont le problème d'entraînement partiellement-séparable possède des fonctions éléments de grande dimension. Le chapitre 6 présente l'ensemble du code produit nécessaire aux contributions des chapitres 4 et 5.

Méthodes quasi-Newton partitionnée à mémoire limitée

Le chapitre 4 introduit les méthodes quasi-Newton partitionnées à mémoire limitée afin de minimiser une fonction partiellement-séparable ayant des fonctions éléments de grande dimension. Dans ces méthodes présentées à la section 4.1, le produit Hessien-vecteur de chaque fonction élément est approximé par un opérateur quasi-Newton à mémoire limitée $v \to \widehat{B}_i U_i v$. L'agrégation des résultats de tous les $v \to \widehat{B}_{i,k} U_i v$ permet de calculer $v \to B_k v$. L'opérateur $v \to B_k v$ permet de définir une approximation quadratique

$$m_k(s) \coloneqq f(x_k) + \left(\sum_{i=1}^{N} \nabla \widehat{f}_i(\widehat{x}_{i,k}) \right)^\top s + \frac{1}{2} s^\top \underbrace{\left(\sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i \right)}_{B_k} s, \quad m_k(s) \approx f(x_k + s). \quad (3)$$

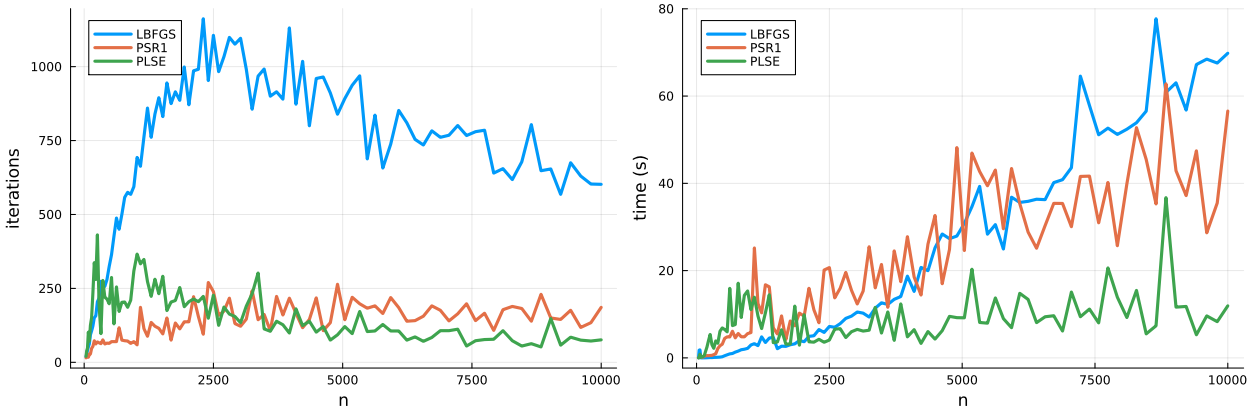
La minimisation du sous-problème $\min_{s \in \mathbb{R}^n} m_k(s)$ détermine à chaque itération la direction de descente d'une recherche linéaire lorsque $B_k \succ 0$, ou le pas d'une région de confiance en ajoutant la contrainte $\|s\| \leq \Delta_k, \Delta_k > 0$. En pratique, ce sous-problème est résolu par la méthode du gradient conjugué tronqué [84, 143] en employant uniquement $v \to B_k v$. Il en résulte trois nouvelles méthodes de région de confiance quasi-Newton partitionnées à mémoire limitée. La première, nommée PLBFGS, approxime chaque $v \to \widehat{B}_{i,k} U_i v$ par un opérateur linéaire à mémoire limitée quasi-Newton BFGS (LBFGS) [20, 21, 23, 59, 67, 101, 114, 141]. La seconde, nommée PLSR1, approxime chaque $v \to \widehat{B}_{i,k} U_i v$ par un opérateur linéaire à mémoire limitée quasi-Newton SR1 (LSR1) [23, 41, 102]. La troisième, nommée PLSE, approxime chaque $v \to \widehat{B}_{i,k} U_i v$ par un opérateur LSR1 lorsque les conditions numériques pour employer un opérateur LBFGS ne sont pas satisfaites. Les trois méthodes PLBFGS, PLSR1 et PLSE sont démontrées comme convergentes par la même preuve, détaillée dans la section 4.2. La preuve consiste à borner $\|B_k\|$ à chaque itération k d'une méthode de région de confiance afin de garantir la convergence globale. Pour réaliser cela, chaque $\|\widehat{B}_{i,k}\|$ est borné par l'application de garde-fous numériques supplémentaires lors de la mise à jour de $\widehat{B}_{i,k}$ à chaque itération.

Les résultats numériques présentés section 4.3 comparent plusieurs méthodes quasi-Newton sur des problèmes partiellement-séparables. Afin de minimiser une fonction partiellement-

séparable possédant des fonctions éléments de grande dimension, on considère la fonction

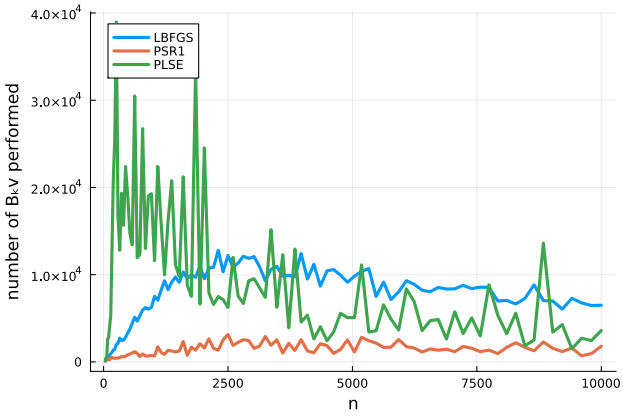
$$f^{\text{limit}} = \sum_{j=1}^{\sqrt{n}-3} \frac{\left(\sum_{i=(j-1)\sqrt{n}}^{(j+2)\sqrt{n}} i x_i \right)^2}{1 + x_j^2} + \sum_{j=1}^{\sqrt{n}-5} \frac{\left(\sum_{i=(j-1)\sqrt{n}+5}^{(j+4)\sqrt{n}+5} i x_i \right)^2}{1 + x_j^2}, \tag{4}$$

pour laquelle les n_i augmentent à mesure que n croît. La fig. 0.1 indique le temps, les itérations et le nombre de produits $B_k v$ nécessaire avant que chaque méthode atteigne un point stationnaire de f^{limit}, i.e. $\|\nabla f(x_k)\| \leq \epsilon \min(1, \|\nabla f(x_0)\|)$. Trois méthodes de région de confiance quasi-Newton sont comparées : PLSE, PSR1 et LBFGS. LBFGS est une méthode quasi-Newton à mémoire limitée. PSR1 est une méthode approximant chaque $\nabla^2 \widehat{f}_i(\widehat{x}_{i,k})$ par une matrice dense $\widehat{B}_{i,k}$, mise à jour à chaque itération par la formule SR1 (2.15). PLSE est une des méthodes quasi-Newton partitionnée à mémoire limitée.



(a) Iterations.

(b) Temps en secondes.



(c) Nombre de produits $B_k v$.

Figure 0.1 Performance des méthodes quasi-Newton minimisant f^{limit} (4).

La fig. 0.1a illustre que PSR1 et PLSE nécessitent moins d'itérations pour minimiser (4) que LBFGS. Le nombre d'itérations requis par LBFGS est entre trois et cinq fois supérieur à celui de PSR1 ou de PLSE, peu importe la dimension de n. La fig. 0.1b indique le temps avant convergence. Ici, PLSE domine PSR1 et LBFGS, qui ont respectivement un temps de convergence quatre et cinq fois plus important que PLSE lorsque $n = 10\,000$. De plus, le temps de résolution de PLSE semble indépendant de la dimension. Finalement, la fig. 0.1c illustre graphiquement le nombre de produits $B_k v$ nécessaires avant convergence, sachant que $B_k v$ est l'opération la plus coûteuse de chaque méthode. PSR1 est la méthode nécessitant le moins de produits $B_k v$, suivie par PLSE et LBFGS. Cependant, PSR1 nécessite plus de temps que PLSE pour calculer un produit $B_k v$ lorque n grandit, en raison de l'approximation de chaque $\nabla^2 \widehat{f}_i$ par une matrice dense. La fig. 0.1 indique PLSE comme la méthode la plus adaptée pour résoudre un problème partiellement-séparable ayant des fonctions éléments de grande dimension.

Les caractéristiques principales d'une méthode quasi-Newton partitionnée à mémoire limitée sont :

- une approximation de $\nabla^2 f(x_k)$ creuse, contrairement aux méthodes quasi-Newton à mémoire limitée;
- un espace mémoire raisonnable et un opérateur $v \to B_k v$ efficace lorsque les n_i sont grands, contrairement aux méthodes quasi-Newton partitionnées standards.

L'application de la séparabilité partielle à l'apprentissage profond

L'entraînement supervisé d'un réseau de neurones se base sur un jeu de données $(\mathcal{X}, \mathcal{Y})$ défini au préalable. À chaque itération, il se formule :

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(X, Y; w) \tag{5}$$

où $\mathcal{L}$ est la fonction de perte estimant la performance d'un réseau de neurones paramétré par les poids/variables w selon le lot (X, Y) issu du jeu de données $(\mathcal{X}, \mathcal{Y})$. Lorsque l'objectif de (5) est de classifier une entrée parmi un ensemble de classes donné, la fonction de perte $\mathcal{L}$ se base sur l'évaluation du réseau de neurones, considéré comme une fonction de scores $c : \mathbb{R}^n \to \mathbb{R}^C$, où C est le nombre de classes prédéfini à l'avance. La section 2.2 rappelle ces notions fondamentales.

Le chapitre 5 propose un problème d'entraînement (5) partiellement-séparable pouvant être minimisé par une méthode quasi-Newton partitionnée à mémoire limitée. Cependant, toutes

les fonctions de perte $\mathcal{L}$ ne sont pas partiellement-séparables. La section 5.1 propose une nouvelle fonction de perte :

$$\mathcal{L}^{\text{PSL}}(X,Y;w) \coloneqq \sum_{p=1}^{C}\sum_{j=1\neq k}^{C} h_{p,j}(X,Y;w), \quad h_{p,j}(X,Y;w) \coloneqq \frac{1}{L}\sum_{l=1}^{L} \delta_{p,j}(y^{(l)})\; e^{c_j(x^{(l)};w)-c_p(x^{(l)};w)}, \tag{6}$$

où $p_i(x^{(l)};w) \coloneqq \frac{exp(c_i(x^{(l)};w))}{\sum_{j=1}^{C} exp(c_j(x^{(l)};w))}$, $(x^{(l)},y^{(l)})$ est un des L couples de donnée-observation issu de (X,Y) et $\delta_{p,j}(y^{(l)}) = 1$ si $y^{(l)} = p$ ou 0 sinon. En utilisant $\mathcal{L}^{\text{PSL}}$, le problème (5) se reformule comme un problème partiellement-séparable, voir les détails section 5.1. En supplément, afin de réduire le sous-ensemble des variables dont dépend $h_{p,j}(X,Y;w)$, la section 5.2 introduit le concept d'architecture partitionnée. L'ingrédient clé de cette architecture est la couche séparable, illustrée fig. 0.2, qui est différente de la couche dense et de convolution rappelées section 2.2.2. L'empilement de couches séparables produit une architecture partitionnée. La fig. 0.3 illustre une architecture LeNet [96] et une architecture partitionnée, dénotée PSNet. L'emploi des couches séparables permet aux scores c_j de dépendre d'un

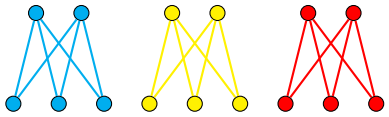


Figure 0.2 Une couche séparable, 9×6, considérant $C = 3$ classes.

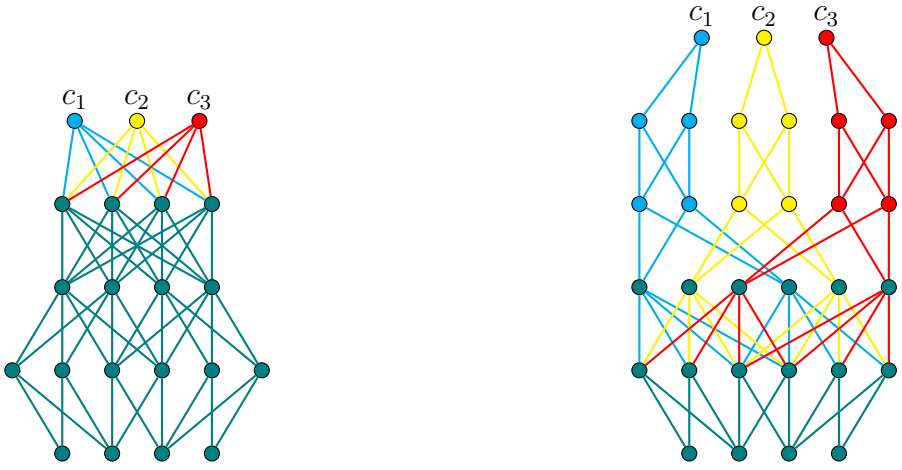


Figure 0.3 Dépendance des poids de chaque score, pour des réseaux LeNet (gauche) et PSNet (droite) simplifiés.

sous-ensemble réduit des variables du réseaux, indentifiables respectivement par la couleur

bleu, jaune et rouge. Les variables communes à tous les scores sont en verts. Contrairement à PSNet, les variables communes prédominent pour LeNet.

Les figs. 0.4 and 0.5 comparent les quatre couples architecture-fonction-de-perte possibles à partir des architectures LeNet et PSNet et des fonctions de perte $\mathcal{L}^{\mathrm{PSL}}$ et $\mathcal{L}^{\mathrm{NLL}}$ (2.2.3). Les spécifications des architectures LeNet et PSNet employées sont renseignées section 5.2.4. Chaque combinaison est entraînée par la méthode du premier ordre Adam [93] sur les jeux de données MNIST [97] et CIFAR10 [94], détaillés section 2.2.1. Les figs. 0.4 and 0.5 montrent

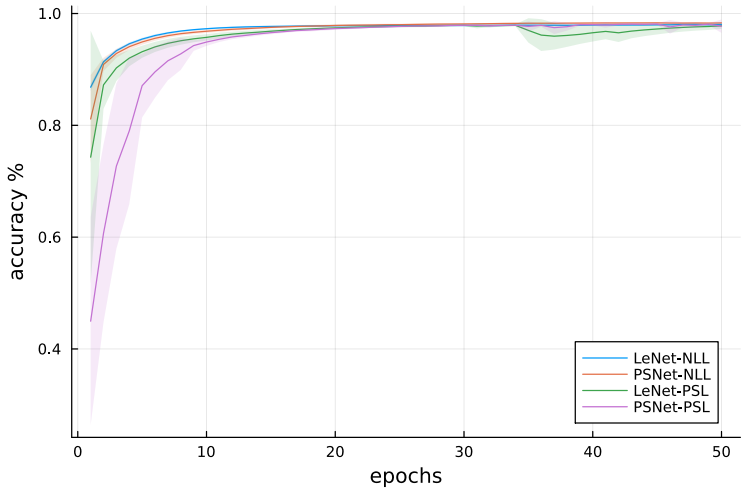


Figure 0.4 Précision au cours des époques de LeNet et PSNet pour $\mathcal{L}^{\mathrm{NLL}}$ et $\mathcal{L}^{\mathrm{PSL}}$ sur MNIST $L = 100$.

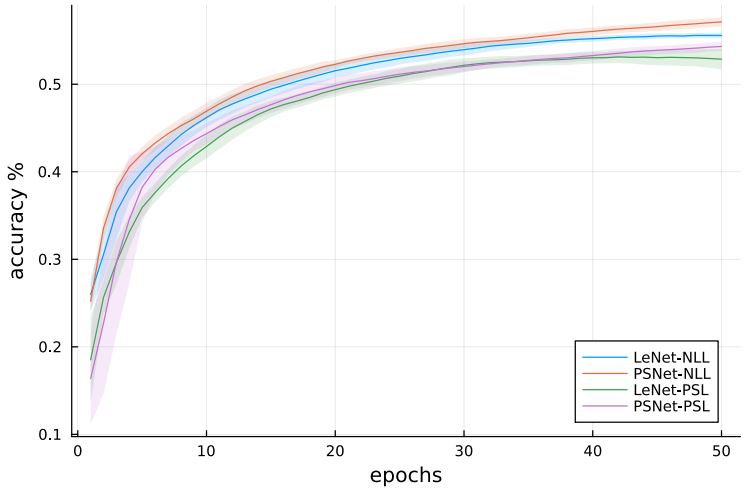


Figure 0.5 Précision au cours des époques de LeNet et PSNet pour $\mathcal{L}^{\mathrm{NLL}}$ et $\mathcal{L}^{\mathrm{PSL}}$ sur CIFAR10 $L = 100$.

que la combinaison de PSNet et de $\mathcal{L}^{\text{PSL}}$ est compétitive avec la combinaison de LeNet et de $\mathcal{L}^{\text{NLL}}$. La performance de PSNet dans la fig. 0.5 s'explique car l'architecture possède plus de couches de convolution que LeNet. La couche de convolution est un des concepts clés de la réussite actuelle des réseaux de neurones en classification d'images. Pour autant, PSNet possède moins de variables que LeNet. La combinaison de PSNet et de $\mathcal{L}^{PSL}$ permet de décrire un nouveau schéma de parallélisation, défini en détails section 5.2.5.

Les méthodes quasi-Newton partitionnées à mémoire limitée entraînant un réseau de neurones sont définies par l'algorithme 5.3.1. Elles sont similaires aux méthodes PLBFGS, PLSR1 et PLSE du chapitre 4. Les figs. 0.6 and 0.7 comparent la précision de différentes méthodes d'entraînement pour une architecture PSNet utilisant $\mathcal{L}^{\text{PSL}}$. Les méthodes d'entraînement considérées sont : PLBFGS, PLSR1, PLSE, LBFGS [4, 25, 26], Adam et SGD [96].

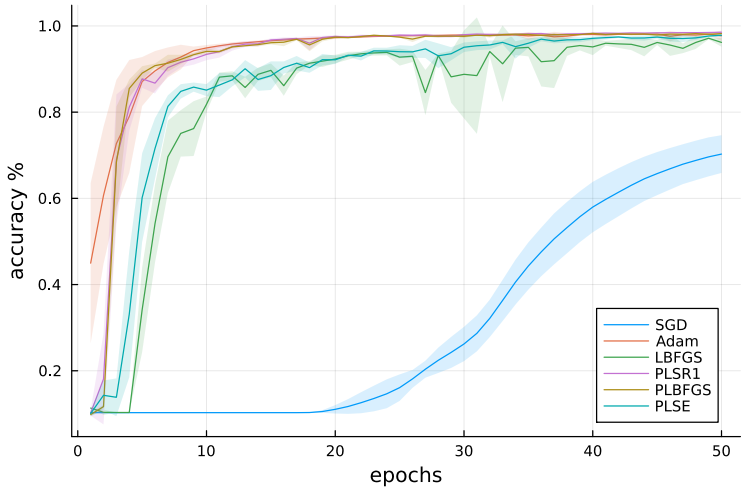


Figure 0.6 Précision au cours des époques pour différentes méthodes d'entraînement considerant PSNet et $\mathcal{L}^{\text{PSL}}$ sur MNIST, $L = 100$.

Les figs. 0.6 and 0.7 démontrent que la précision d'un entraînement avec PLBFGS ou PLSR1 est supérieure à celle de PLSE, LBFGS et SGD, mais reste inférieure à celle d'Adam. Les limites de PLBFGS, PLSR1 et PLSE sont énumérées section 5.4.

Logiciels exploitant la séparabilité partielle

Le chapitre 6 énumère et détaille l'ensemble du code implémenté durant cette thèse. Le code est divisé dans des modules Julia [5] qui implémentent les contributions des chapitres 4 et 5. Ces modules sont intégrés à l'écosystème JuliaSmoothOptimizers (JSO) [110] regroupant des outils pour l'optimisation continue. Chaque module est public et bénéficie de l'intégration

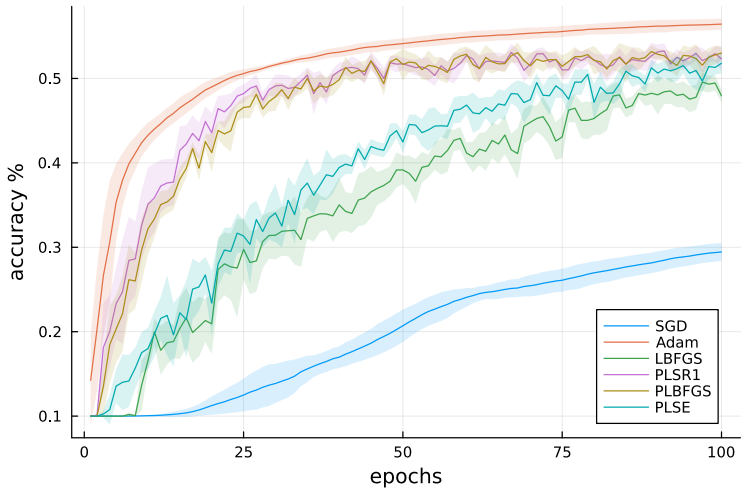


Figure 0.7 Précision au cours des époques pour différentes méthodes d'entraînement considerant PSNet et $\mathcal{L}^{\text{PSL}}$ sur CIFAR10, $L = 100$.

continue, comprenant : une documentation générée automatiquement, une validation par des tests, et des relectures extérieures afin de valider une modification importante. La table 6.1 indique pour chaque module : la version, le pourcentage de code couvert par les tests, le nombre de lignes, et sa section associée. Le chapitre 6 est divisé en deux parties.

Les sections 6.3 à 6.6 définissent une suite logicielle permettant notamment de détecter automatiquement la structure partiellement-séparable d'un problème, c'est-à-dire la récupération de chaque fonction élément $\widehat{f}_i$ de dimension réduite. Suite à cela, les structures de données partitionnées nécessaire à la mémorisation ou l'approximation des dérivées sont allouées, ce qui permet la définition de plusieurs modèles d'optimisation quasi-Newton partitionnés (à mémoire limitée). Chacun de ces modèles satisfait l'interface des modèles d'optimisation sur laquelle se basent les implémentations abstraites des méthodes d'optimisation de JSO. Ainsi, l'implémentation de la région de confiance de JSO exploite les approximations quasi-Newton (à mémoire limitée) $B_k \approx \nabla^2 f(x_k)$ et met en oeuvre les algorithmes 3.2.1 et 4.1.1 définis aux sections 3.2 et 4.1. Les modules décrits dans les sections 6.3 à 6.6 génèrent les résultats numériques du chapitre 4, incluant la fig. 0.1. Les dépendances logicielles entre ces modules sont illustrées par la fig. 6.1.

Les sections 6.7 et 6.8 se concentrent sur l'apprentissage profond. Le module défini section 6.7 propose une interface entre un problème d'entraînement d'un réseau de neurones et un modèle d'optimisation de JSO. Par conséquent, un réseau de neurones peut être entraîné par un algorithme d'optimisation développé au sein de JSO. Le module défini section 6.8 implémente la détection de la structure partiellement-séparable d'une fonction de perte, les architectures

partitionnées et l'entraînement quasi-Newton partitionné à mémoire limitée. Ce module produit les résultats numériques de la section 5.3, incluant les figures 0.6, 0.4, 0.5 et 0.7.

Conclusion

La revue de littérature dédiée à la séparabilité partielle souligne deux lacunes :

- les méthodes quasi-Newton partitionnées sont inefficaces pour minimiser une fonction partiellement-séparable ayant des fonctions éléments de grande dimension ;
- l'absence de logiciel libre détectant automatiquement la séparabilité partielle.

Le chapitre 4 propose une solution pour minimiser efficacement des problèmes dont les fonctions éléments sont de grande taille. Chaque hessien d'une fonction élément est approximé par un opérateur quasi-Newton à mémoire limitée. L'approximation de cette solution permet d'obtenir une approximation creuse de $\nabla^2 f(x_k)$, nécessitant un espace mémoire raisonnable et un opérateur $v \to B_k v$ efficace lorsque les n_i sont grands, contrairement aux méthodes quasi-Newton partitionnées standards.

Le chapitre 5 applique les concepts liés à la séparabilité partielle durant l'entraînement supervisé d'un réseau de neurones. La section 5.1 introduit une fonction de perte partiellement-séparable $\mathcal{L}^{\text{PSL}}$. La section 5.2 introduit les architectures partitionnées PSNet. La combinaison de PSNet et $\mathcal{L}^{\text{PSL}}$ est compétitive avec des architectures et des fonctions de perte de l'état-de-l'art. De plus, cette combinaison est entraînable par des méthodes quasi-Newton partitionnées à mémoire limitée. Ces méthodes sont compétitives avec les méthodes issues de la littérature, seulement inférieures à Adam.

Le chapitre 6 présente les modules Julia développés durant cette thèse permettant de détecter et d'exploiter automatiquement la structure partiellement-séparable. Ces modules produisent les résultats numériques des chapitres 4 et 5. Enfin, les limitations de nos travaux de recherche sont énumérées dans la section 7.2 et les travaux futurs dans la section 7.3.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

RÉSUMÉ DE LA THÈSE ix

TABLE OF CONTENTS xix

LIST OF TABLES xxiii

LIST OF FIGURES xxiv

LIST OF SYMBOLS AND ACRONYMS xxvi

LIST OF APPENDICES xxviii

CHAPTER 1 INTRODUCTION AND CONTEXT 1
1.1 Context 1
1.2 Problematics 3
1.3 Outline 4

CHAPTER 2 BACKGROUND 6
2.1 Concepts related to quasi-Newton methods 6
2.1.1 Line search methods 6
2.1.2 Trust-region methods 8
2.1.3 Quasi-Newton methods 10
2.1.4 Conjugate gradient and the truncated variant 13
2.1.5 Computing derivatives 16
2.2 Supervised learning and parallelization of neural network training 19
2.2.1 Dataset 20
2.2.2 Neural network architecture 21
2.2.3 Loss function 25

2.2.4 Neural network training . . . 26
2.2.5 Neural network optimizers . . . 26
2.2.6 Neural network parallelization . . . 28

CHAPTER 3 LITERATURE REVIEW . . . 30
3.1 Partially-separable structure . . . 30
3.1.1 Partially-separable function . . . 30
3.1.2 Derivatives of a partially-separable function . . . 34
3.1.3 Group partial separability . . . 37
3.1.4 Other definitions of partial separability . . . 38
3.2 Partitioned quasi-Newton methods . . . 40
3.2.1 Partitioned update for nonsmooth element functions . . . 44
3.2.2 Partitioned Quasi-Newton vs. Limited-Memory Quasi-Newton . . . 45
3.2.3 Partitioned update without element gradients . . . 46
3.2.4 Partitioned quasi-Newton trust-region . . . 47
3.3 Optimization methods enhancing partial separability exploitation . . . 48
3.3.1 Structured trust-region methods . . . 48
3.3.2 Partial separability exploitation to solve quadratic subproblem . . . 50
3.3.3 Enhance the performance of partitioned quasi-Newton methods . . . 52
3.4 Partially-separable problems with dedicated methods . . . 57
3.4.1 Large scale nonlinear network problems . . . 57
3.4.2 Signal reconstruction . . . 59
3.5 Computation of the partitioned derivatives . . . 60
3.5.1 Straightforward procedure when $\widehat{f}_i$ is available . . . 61
3.5.2 Exploiting directional derivatives . . . 61
3.6 Partitioned quasi-Newton software . . . 63
3.6.1 PSPMIN . . . 64
3.6.2 Revenge of the SIF . . . 66
3.6.3 LANCELOT and GALAHAD . . . 68
3.6.4 The AMPL modeling language . . . 69
3.6.5 Large-scale algorithms (LSA) . . . 70
3.7 Parallelization of partially-separable methods . . . 71
3.7.1 Synchronous methods . . . 71
3.7.2 Asynchronous methods . . . 76
3.8 Derivative-free methods . . . 78
3.8.1 Quadratic interpolation exploiting partial separability . . . 79

3.8.2 Pattern searches . . . 83
3.8.3 Partitioned crossover operator . . . 86
3.9 Critical analysis . . . 86

CHAPTER 4 Limited memory variant of partitioned quasi-Newton methods . . . 88
4.1 Partitioned limited-memory quasi-Newton trust-region . . . 89
4.2 Global convergence proof . . . 90
4.3 Numerical results . . . 98
4.3.1 Comparing quasi-Newton methods . . . 100
4.3.2 Experiments with limited-memory partitioned quasi-Newton methods . 104
4.4 Conclusion . . . 106

CHAPTER 5 Partially-separable learning . . . 107
5.1 Partially-separable loss function . . . 108
5.2 Separable layer and partitioned architecture . . . 109
5.2.1 The issue of standard architectures . . . 109
5.2.2 Separable layer . . . 110
5.2.3 Partitioned architecture . . . 110
5.2.4 Numerical results . . . 111
5.2.5 Parallel partitioned architecture computations . . . 115
5.2.6 Dropout consequences . . . 116
5.3 Limited-memory partitioned quasi-Newton training . . . 117
5.3.1 Numerical results . . . 119
5.3.2 A parallel partitioned Hessian-vector product . . . 120
5.4 Limitations and open problems . . . 121

CHAPTER 6 Software packages . . . 123
6.1 Programming environment and metadata . . . 123
6.2 JuliaSmoothOptimizers (JSO) ecosystem architecture . . . 124
6.3 ExpressionTreeForge.jl . . . 126
6.4 PartitionedStructures.jl . . . 128
6.5 PartitionedVectors.jl . . . 131
6.6 PartiallySeparableNLPModels.jl . . . 133
6.7 KnetNLPModels.jl . . . 134
6.7.1 Summary . . . 134
6.7.2 Training a neural network with KnetNLPModels.jl and JSO solvers . . 135
6.8 PartitionedKnetNLPModels . . . 137

6.9 Conclusion 138

CHAPTER 7 CONCLUSION 139
7.1 Summary of Works 139
7.2 Limitations 140
7.3 Future Research 142

REFERENCES 144

APPENDICES 158

LIST OF TABLES

Table 4.1 Instance details of f^{limit} for $n \in \{36, 625, 2500, 10000\}$ 104
Table 5.1 Architecture details 114
Table 6.1 Julia modules metadata 124
Table A.1 Partial separability details of the partially-separable problems set . . . 159

LIST OF FIGURES

Figure 0.1 Performance des méthodes quasi-Newton minimisant f^{limit} (4). xii
Figure 0.2 Une couche séparable, 9×6, considérant $C = 3$ classes. xiv
Figure 0.3 Dépendance des poids de chaque score, pour des réseaux LeNet (gauche) et PSNet (droite) simplifiés. xiv
Figure 0.4 Précision au cours des époques de LeNet et PSNet pour $\mathcal{L}^{\text{NLL}}$ et $\mathcal{L}^{\text{PSL}}$ sur MNIST $L = 100$. xv
Figure 0.5 Précision au cours des époques de LeNet et PSNet pour $\mathcal{L}^{\text{NLL}}$ et $\mathcal{L}^{\text{PSL}}$ sur CIFAR10 $L = 100$. xv
Figure 0.6 Précision au cours des époques pour différentes méthodes d'entraînement considerant PSNet et $\mathcal{L}^{\text{PSL}}$ sur MNIST, $L = 100$. xvi
Figure 0.7 Précision au cours des époques pour différentes méthodes d'entraînement considerant PSNet et $\mathcal{L}^{\text{PSL}}$ sur CIFAR10, $L = 100$. xvii
Figure 2.1 Automatic differentiation for $f(x) = (4.5 - (x_1x_2 + x_3x_4))^2$ 19
Figure 2.2 Activation function for a dense layer 22
Figure 2.3 Dense neural network and notation 22
Figure 2.4 Convolutional layer and notation (without activation function) 24
Figure 2.5 Max-pooling 24
Figure 3.1 Automatic differentiation for one group of a group partially-separable function 39
Figure 3.2 Arrow Hessian pattern (left) and its factor (right) 52
Figure 3.3 Tridiagonnal Hessian pattern (left) and its factor (right) 60
Figure 4.1 Iteration (left) and time (right) performance profiles for Newton and quasi-Newton methods. 101
Figure 4.2 Iteration (left) and time (right) performance profiles for partitioned quasi-Newton methods. 101
Figure 4.3 Iteration (left) and time (right) performance profiles for limited-memory partitioned quasi-Newton methods. 103
Figure 4.4 Summary of iteration (left) and time (right) performance profiles for (limited-memory) partitioned quasi-Newton methods 103
Figure 4.5 Metrics of quasi-newton methods solving (4.10). 105
Figure 5.1 Weight dependencies of simplified LeNet scores 109
Figure 5.2 A separable layer, 9×6, considering $C = 3$ groups 110
Figure 5.3 Weight dependencies of simplified PSNet scores 111

Figure 5.4 LeNet and PSNet training accuracies over epochs on MNIST, considering minibatches of size 20. . . . 112
Figure 5.5 LeNet and PSNet training accuracies over epochs on MNIST, considering minibatches of size 100. . . . 112
Figure 5.6 LeNet and PSNet training accuracies over epochs on CIFAR10, considering minibatches of size 20. . . . 113
Figure 5.7 LeNet and PSNet training accuracies over epochs on CIFAR10, considering minibatches of size 100. . . . 114
Figure 5.8 Partitioned neural network paired with $\mathcal{L}^{\text{PSL}}$ computation distribution . 117
Figure 5.9 Weight dependencies of PSNet scores when the first convolutional layer is dropped out (dotted deactivated weights) . . . 117
Figure 5.10 Comparison of optimizer accuracies over epochs during PSNet training when minimizing $\mathcal{L}^{\text{PSL}}$ (5.1a) on MNIST. . . . 120
Figure 5.11 Comparison of optimizer accuracies over epochs during PSNet training when minimizing $\mathcal{L}^{\text{PSL}}$ (5.1a) on CIFAR10. . . . 120
Figure 6.1 Abstract type and interface dependencies of JSO related to partial separability. The text and the boxes in red denote the code developed during the thesis. . . . 127
Figure 6.2 Automatic partial separability detection . . . 129
Figure 6.3 Bounds and convexity status of element functions . . . 129
Figure 6.4 Weight dependencies for the score c_1 (left) and for the element function $\widehat{h}_{1,2}$ (right) . . . 138

LIST OF SYMBOLS AND ACRONYMS

(PL)BFGS	(Limited-memory Partitioned) Broyden Fletcher Goldfarb Shanno
(PL)SR1	(Limited-memory Partitioned) Symmetric rank one
(PL)SE	(Limited-memory Partitioned) Secant Equation
(P)PSB	(Partitioned) Powell-symmetric-Broyden
S-PSB	Sparse PSB
JSO	JuliaSmoothOptimizers
SIF	Standard input format
AMPL	A Modeling Language for Mathematical Programming
CUTE(st)	Constrained and Unconstrained Testing Environment (with safe threads)
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
ELSO	Environments for Large-Scale Optimization
BFO	Brute Force Optimizer
PCDM	Partitioned Coordinate Descent Methods
APPROX	Accelerated Parallel and PROXimal
LSNNO	large-scale Nonlinear Network Optimization
HAMSI	Hessian Approximated Multiple Subsets Iteration
SGD	Stochastic gradient descent
RMSprop	Root Mean Squared Propagation
Adam	Adaptive moment estimation
AdaGrad	Adaptive Gradient algorithm
ADIFOR	Automatic Differentiation library in Fortran
GPU	Graphical Processing Unit
AD	Automatic differentiation
LTS	Lower Triangular Substitution
DAG	Directed Acyclic Graph
ED	Element Dimension, i.e. n_i
EC	Element Contribution

Notation

Lowercase Latin letters such as s, x and y denote vectors, except n which is the dimension of the problem. e_i refers to the i-th euclidean vector and $\mathbb{1} = (1, 1, \ldots, 1)$. Uppercase Latin letters such as B and H denote matrices. Lowercase Greek letters such as α and λ denote

scalars. Throughout, I_n denotes the identity matrix of size n. When there is no ambiguity, I denotes the identity matrix of appropriate size. $B \succ 0$ means that B is positive definite. When $\|\cdot\|$ is not specified, it refers to the Euclidean norm.

Any object $\widehat{}_i$ refers to an object related to the element function $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$ instead of $f : \mathbb{R}^n \to \mathbb{R}$, e.g., $\nabla \widehat{f}_i$ is the i-th element gradient. Consequently, the size of $\widehat{}_i$ is related to n_i instead of n, and $n_i^{\max} = \max_{1 \le i \le N} n_i$. When the k-th iterate must be specified, the subscript k is added $\widehat{}_{i,k}$. Here is a list of all the related notation:

- $\widehat{s}_i \coloneqq U_i s, \ s \in \mathbb{R}^n$;
- $\widehat{y}_i \coloneqq \nabla \widehat{f}_i(\widehat{x}_i + \widehat{s}_i) - \nabla \widehat{f}_i(\widehat{x}_i) \in \mathbb{R}^{n_i}$;
- $\widehat{B}_i \approx \nabla^2 \widehat{f}_i \in \mathbb{R}^{n_i \times n_i}$;
- $\widehat{z}_i \coloneqq \widehat{y}_i - \widehat{B}_i \widehat{s}_i$.

$B_k^{(0)}$ is the initializer of a limited-memory quasi-Newton operator at the iteration k. The limited-memory operator resulting from j quasi-Newton updates is denoted $B_k^{(j)}$, while $\widehat{B}_{i,k}^{(j)}$ is its elemental counterpart.

LIST OF APPENDICES

Appendix A Partially-separable problem structures 158
Appendix B Conjugate gradient replacement 160

CHAPTER 1 INTRODUCTION AND CONTEXT

1.1 Context

This thesis focuses on the minimization of large scale unconstrained continuous optimization problems, such as :

$$\min_{x \in \mathbb{R}^n} f(x) \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R} \in C^2$ possesses a *partially-separable* structure:

$$f(x) = \sum_{i=1}^{N} f_i(x), \quad f_i : \mathbb{R}^n \to \mathbb{R}, \tag{1.2}$$

considering that each *element function* $f_i(x)$ depends on a subset of the decision variables. To avoid introducing too many notations, (1.2) is a simplified definition of partial separability. A complete definition of the partially-separable function f specifying the lower dimension of the element functions is given Section 3.1.1, by (3.3). Partially-separable problems appear in several fields, e.g., variational calculus, including optimal command, sparse least-square problems, resource allocation problems, urban traffic network analysis and matrix completion problems [27, 75, 91, 151]. For example, the variational calculus "shortest curve" problem:

$$\begin{array}{ll} \min\limits_{x} & \int_{t_0}^{t_1} \sqrt{1 + \dot{x}^2} dt \\ \text{s.t.} & x(t_0) = a, \\ & x(t_1) = b, \end{array}$$

discretized with the trapezoidal rule results in

$$\begin{array}{ll} \min\limits_{x \in \mathbb{R}^n} & \frac{h}{2}\sqrt{1 + \frac{x_2 - x_1}{h}} + \sum\limits_{i=2}^{n-1} h\sqrt{1 + \frac{x_{i+1} - x_{i-1}}{2h}} + \frac{h}{2}\sqrt{1 + \frac{x_n - x_{n-1}}{h}}, \\ \text{s.t.} & x_1 = a, \\ & x_n = b, \end{array}$$

reformulated as the partially-separable unconstrained optimization problem:

$$\min_{x \in \mathbb{R}^{n-2}} \frac{h}{2}\sqrt{1 + \frac{x_2 - a}{h}} + h\sqrt{1 + \frac{x_3 - a}{2h}} + \left(\sum_{i=3}^{n-2} h\sqrt{1 + \frac{x_{i+1} - x_{i-1}}{2h}}\right) + h\sqrt{1 + \frac{b - x_{n-2}}{2h}}, + \frac{h}{2}\sqrt{1 + \frac{b - x_{n-1}}{h}}$$

by substituting $x_1 = a$ and $x_n = b$ in the objective, i.e. $x = (x_2, x_3, \ldots, x_{n-1})^\top \in \mathbb{R}^{n-2}$. Consequently, N, the number of element function, equals n and each element function is at

most size 2, i.e., $n_i \leq 2$, $\forall 1 \leq i \leq N$, forming a tridiagonal Hessian matrix.

Both first order and second order methods are candidate to minimize such problems. The former methods rely only on the gradient ∇f while the latter ones incorporate curvature from the Hessian $\nabla^2 f$ to minimize f. Second order methods offer better convergence results than first order methods, but require supplementary computational resources. As this thesis focuses on large scale optimization, i.e., where n is large, the computation of any Hessian related information is considered too costly. This led to the development of quasi-Newton methods, which update iteratively an approximation $B \approx \nabla^2 f$ from gradient evaluations computed across the iterations. At each iteration, a quadratic approximation of f is built without incurring Hessian computation. Initially, any quasi-Newton method was supported by a dense matrix, which eventually raised an issue for large problems, as storing $\frac{n(n+1)}{2}$ terms becomes unaffordable. Thus, limited-memory quasi-Newton variants became popular to minimize large problems, being based on a cheap linear operator $v \to Bv \approx v \to \nabla^2 f v$ allowing to solve a quadratic subproblem with the (truncated) conjugate gradient method [84, 143].

By construction, every quasi-Newton update produces a dense Hessian approximation, and therefore, cannot exploit the sparse structure that the real Hessian may have. Although sparse quasi-Newton methods exist, their numerical results are originally unsatisfactory [146] (1977). Few years later, in 1982, Griewank and Toint demonstrated that any function having a sparse Hessian is partially-separable [76, Theorem 1]. At the same time, they presented the partitioned quasi-Newton methods, where $\nabla^2 f \approx B = \sum_{i=1}^{N} B_i$ considering $B_i \approx \nabla^2 f_i$. As each f_i depends on a subset of variables, each B_i can be stored as a dense matrix of small size. Consequently, the storage of the partitioned B is generally a lot smaller than a dense matrix and $Bv = \sum_{i=1}^{N} B_i v$ can aggregate efficiently the contributions of every $B_i v$. Usually, the partially-separable discretized problems possess element function depending on very few variables. Moreover, a finer discretization results in a larger problem f with a sparser $\nabla^2 f$, making them ideal candidates for partitioned quasi-Newton methods. When f is partially-separable, partitioned quasi-Newton methods outperform any other quasi-Newton variants [101, 148].

This thesis continues the work started four decades ago about the exploitation of partial separability and tries to rejuvenate some of its ideas. In particular, partial separability concepts are applied to deep learning training problems which exhibit a structure similar to partial separability. This structure has never been exploited and could introduce a new parallel scheme, which is a critical issue for this growing area of research.

1.2 Problematics

This thesis is partly motivated by the shortcomings identified during the literature review and by the application of partial separability concepts to supervised learning, resulting in three works.

The first project studies and completes the literature review. It provides an overview of the schemes and algorithms exploiting partial separability. The critical analysis concluding the literature review exposes what the current partitioned quasi-Newton algorithms lack. The both next projects are based on the conclusion drawn from this literature review.

The second project seeks to remedy two issues of partitioned quasi-Newton methods: providing an open source package detecting automatically the partially-separable structure, and designing an optimization method that manages large element functions. Currently, the sole partitioned quasi-Newton Fortran solver LANCELOT [33] is based either on the Standard input format (SIF) or AMPL [62], a commercial modelling language. Therefore, a new open source software is needed before implementing methods that exploit partial separability. The automatic detection of partial separability is based on tree-walking algorithms applied to the computational graph of a function, similarly to [64]. Consequently, the first step is to retrieve the expression tree from a modelling language, isolate element functions and provide routines for evaluating them and their derivatives. Moreover, the implementation of partitioned quasi-Newton methods requires the definition of partitioned data structures for storing element gradients and Hessians. The resulting software must be incorporated in the JuliaSmoothOptimizers (JSO) ecosystem. It must enable the JSO interface-based solvers to exploit efficiently a partitioned quasi-Newton operator. Once this software is implemented, the support of large element functions expands the range of problems for which partial separability can be exploited. Any problem having a sparse Hessian becomes a candidate problem for JSO partitioned quasi-Newton methods, without informing its partial separability.

The third project studies the application of partial separability concepts in a supervised learning context. Generally, the optimization problem related to a neural network training is generally not partially-separable. In order to be partially-separable, a new partitioned neural network architecture must be used as well as a loss function maintaining its structure. The resulting training problem is (extremely) large and possesses large element functions. There are two difficulties in implementing these concepts. First, there are no methods to automatically compute the partial separability of such a training problem. Second, the routines computing the derivatives of the loss function are not designed to compute the derivatives of the element functions. Nonetheless, the algorithms exploiting partial separability often

result in straightforward parallel implementations, which is of interest in a supervised learning context.

1.3 Outline

This thesis recalls in Chapter 2 some concepts about numerical quasi-Newton optimization methods and supervised learning. Although partial-separability is not mentioned during this chapter, those ideas will be reused later while exploiting partial-separability. The reader familiar with both quasi-Newton optimization concepts, described in Section 2.1, and supervised learning, depicted in Section 2.2, can go straight to the Chapter 3, which is a literature review about partially-separable optimization.

A full definition of the partially-separable structure with several examples is provided in Section 3.1. The Section 3.2 details fully the partitioned quasi-Newton methods. Both the Section 3.3 and Section 3.4 encompass continuous methods exploiting partial separability. Next, the Sections 3.5 to 3.7 enumerate schemes and implementations exploiting efficiently the partially-separable structure. The Section 3.8 presents several derivative-free methods incorporating partial separability. The Section 3.9 concludes the chapter with a critical analysis about partitioned quasi-Newton methods. The survey written by Bigeon, Orban, and Raynaud [9] inspired this chapter.

The Chapter 4 introduces new limited-memory partitioned quasi-Newton methods in Section 4.1. The Section 4.2 proves their global convergence while the Section 4.3 displays their numerical results. This contribution adapts to this thesis the research conducted by Bigeon, Orban, and Raynaud [8].

The Chapter 5 gathers the deep learning contributions exploiting partial separability. The training must consider a partially-separable loss function as described in Section 5.1, and possess a partitioned architecture as developed in Section 5.2. If such requirements are fulfilled, a new parallelization scheme is possible and detailed in Section 5.2. The Section 5.3 showcases results of the limited-memory partitioned quasi-Newton training. The Section 5.4 concludes with their limitations. Raynaud, Orban, and Bigeon [135] introduce the partially-separable loss function and the partitioned architecture, while Raynaud, Orban, and Bigeon [134] present the limited-memory partitioned quasi-Newton trainings.

All the pieces of code that are developed for the previous chapters are gathered in Chapter 6. The Section 6.1 gives a quantitative overview of the work accomplished. All the code produced is integrated into the JSO environment briefly summarized in Section 6.2. The Sections 6.3 to 6.6 enumerate and detail the pieces of software designed for partially-separable optimization,

i.e., related with Chapters 3 and 4. The Sections 6.7 and 6.8 focus on deep learning and implement the Section 2.2 and Chapter 5.

Finally, the conclusion and future works are discussed in Chapter 7.

CHAPTER 2 BACKGROUND

This chapter lays the ground for the contributions presented in Chapter 4 and Chapter 5, and introduces the constraints that the softwares implemented in Chapter 6 must incorporate. In particular, this chapter recalls the concepts related to quasi-Newton methods for unconstrained optimization Section 2.1 and the basics for supervised learning Section 2.2. Those two sections contain no reference to the partially-separable structure. Additionally, the notions and the notation used try to match the popular notation employed nowadays and will be reused later in the thesis. A familiar reader with inexact quasi-Newton methods may progress to the Chapter 3 which presents a literature review about the optimization of partially-separable problems.

2.1 Concepts related to quasi-Newton methods

Those notions are popular in continuous optimization and have been extensively studied in books such as Numerical Optimization by Nocedal and Wright [115] and Trust-region methods by Conn et al. [39]

In nonlinear optimization, most methods seek to reduce iteratively the value of the objective function. At every iteration k, an approximated solution of a simpler subproblem related to x_k finds the step s_k, in turn used to determine the next iterate $x_{k+1} = x_k + s_k$. In this thesis, we consider mainly two families of methods: the line searches, presented in Section 2.1.1, and the trust-region methods introduced in Section 2.1.2. The description of both families anticipates the exploitation of a quadratic approximation of $f(x_k)$ to generate s_k. The Section 2.1.3 recalls the secant quasi-Newton methods, employed to build a quadratic approximation of f by approximating $\nabla^2 f$ from gradient evaluations, as well as their numerical implementations. Finally, the Section 2.1.4 presents the (truncated) conjugate gradient—an iterative matrix-free method solving inexactly the quadratic subproblem induced by a quasi-Newton line search or a quasi-Newton trust-region.

2.1.1 Line search methods

A line search method is divided in two steps. First, given x_k it generates a direction search d_k, which most of the time will be a descent direction, that is such that

$$\nabla f(x_k)^\top d_k < 0.$$

To fulfill this criterion, the most straightforward candidate is the steepest descent $d_k = -\nabla f(x_k)$ making $\nabla f(x_k)^\top d_k = -\|\nabla f(x_k)\|^2$. Second, it solves a one dimensional minimization problem along the line span by d_k

$$\arg\min_{\alpha \in \mathbb{R}^+} \phi(\alpha) = f(x_k + \alpha d_k). \tag{2.1}$$

The solution resulting of (2.1) allows setting $s_k = \alpha d_k$ and updating $x_{k+1} = x_k + s_k$. Those two steps are repeated until convergence occurs, which is numerically reached when either an *absolute criterion* $\|\nabla f(x_k)\| \leq \epsilon_1$ or a *relative criterion* $\|\nabla f(x_k)\| \leq \epsilon_2 \|\nabla f(x_0)\|$ holds, considering $\epsilon_1, \epsilon_2 > 0$ small, e.g. 10^{-6}.

When α solves (2.1), the method is referred as an *exact* line search. Finding such an α may be computationally intensive, and, in order to spare computational effort, it is frequent to get an approximate solution of (2.1). Those methods are *inexact* line searches and seek to obtain a *sufficient decrease*, such as the *Armijo condition*:

$$f(x_k + \alpha d_k) \leq f(x_k) + \tau_1 \alpha \nabla f(x_k)^\top d_k, \quad 0 < \tau_1 < 1, \tag{2.2}$$

ensuring that α brings a decrease which is at least a fraction of the directional derivative, implying $f(x_{k+1}) < f(x_k)$.

The Armijo condition can be enforced with the *curvature* condition

$$\nabla f(x_k + \alpha d_k)^\top d_k \geq \tau_2 \nabla f(x_k)^\top d_k, \quad \tau_2 > 0, \tag{2.3}$$

to avoid steps not exploiting the slope ahead of them. Together, the Armijo and the curvature conditions form the *Wolfe* conditions, which are a standard to prove the global convergence for several line search methods. However, the curvature condition requires the computation of $\nabla f(x_k + \alpha d_k)$ for every α tested, which can be expensive for large problems. Hence, to avoid multiple gradient evaluations, an alternative is the backtracking line search, described in Algorithm 2.1.1. In addition of satisfying the Armijo condition, the Algorithm 2.1.1

Algorithm 2.1.1 Backtracking Line Search [115, Algorithm 3.1]

Choose $\bar{\alpha} > 0$, $\alpha = \bar{\alpha}$, $0 < \rho < 1$, $0 < \tau_1 < 1$
while the Armijo condition (2.2) is not satisfied **do**
  $\alpha = \rho\alpha$
end while

ensures that α is not too short, since it collects the first step that is not too long, i.e., which satisfies (2.2). Such line search can be proven globally convergent, that is under suitable

conditions

$$\liminf_{k\to\infty} \|\nabla f(x_k)\| = 0,$$

ensuring the method to eventually reach a stationary point.

A variant of the Algorithm 2.1.1 is the two ways backtracking line search that allows α to grow as long as (2.2) remains satisfied[1]. When $\alpha > \alpha_0$ fails (2.2), then the previous value of α is taken. Conversely, when $\alpha = \alpha_0$ fails, then α is determined by the Algorithm 2.1.1.

2.1.2 Trust-region methods

Unlike the line search method, the trust-regions methods are not confined to one (or several) directions. Instead, the model $m_k(s)$ approximates $f(x_k + s)$ in a close trusted neighborhood of the current iterate:

$$\begin{array}{rl} \arg\min_{s\in\mathbb{R}^n} & m_k(s) \approx f(x_k + s) \\ s.t. & \|s\| \leq \Delta_k \end{array}, \tag{2.4}$$

where Δ_k parametrizes the *trust-region* size. Usually, $m_k(s)$ approximates $f(x_k + s)$ in the sense of

$$m_k(0) = f(x) \quad \text{and} \quad \nabla m_k(0) = \nabla f(x_k).$$

Several choices of norm can be chosen for $\|s\| \leq \Delta_k$ in (2.4) [39, Sections 7.7 and 7.8], but this thesis only considers the euclidean norm $\|.\|_2$. Hence, the trust-region is a ball centred in x_k with a radius of Δ_k.

The radius Δ_k is eventually updated at every iteration, i.e., increased or decreased, depending on the fitness between $m_k(s)$ and $f(x_k + s)$, computed by:

$$\rho_k := \frac{f(x_k) - f(x_k + s)}{m_k(0) - m_k(s)}. \tag{2.5}$$

If $\rho_k \leq 0$, then the approximate solution s results in $f(x_k + s) \geq f(x_k)$, since $m_k(s) < m_k(0)$ by definition. In such a case, $m_k(s)$ is clearly unable to approximate properly $f(x_k + s)$ in the entirety of the trust-region, and therefore, Δ_{k+1} is chosen smaller than Δ_k. Conversely, when $\rho_k > 0$ then $f(x_k + s) < f(x_k)$. Additionally, when ρ_k tends to 1, then $m_k(s)$ tends to match $f(x_k + s)$, which confers more confidence toward the model m_k within the trust-region and resulting in a trust-region expansion for the next iteration. To summarize, after computing

[1] A simpler implementation of Algorithm 3.5 (p. 60) from [115]

ρ_k, Δ_k is updated as

$$\Delta_{k+1} \in \begin{cases} [\gamma_3\Delta_k,\, \gamma_4\Delta_k] & \text{if } \rho_k \geq \eta_2, \\ [\gamma_2\Delta_k,\, \Delta_k] & \text{if } \eta_1 \leq \rho_k < \eta_2, \\ [\gamma_1\Delta_k,\, \gamma_2\Delta_k) & \text{if } \rho_k < \eta_1, \end{cases} \tag{2.6}$$

where $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$ and $0 < \eta_1 \leq \eta_2 < 1$.

Similar to a line search, it is not necessary to solve (2.4) exactly for the trust-region method to achieve global convergence. When the model m_k is quadratic

$$m_k(s) = f(x_k) + \nabla f(x_k)^\top s + \tfrac{1}{2} s^\top B_k s \approx f(x_k + s), \quad \text{where} \quad B_k \approx \nabla^2 f(x_k), \tag{2.7}$$

the step only needs to produce a sufficient decrease in the sense that

$$m_k(0) - m_k(s) \geq \tau \, \|\nabla f(x_k)\| \, \min\left(\frac{\|\nabla f(x_k)\|}{1 + \|B_k\|}, \, \Delta_k \right), \quad 0 < \tau < 1, \tag{2.8}$$

to globally converge [39, Theorem 6.4.6]. By considering $\tau = \frac{1}{2}$, this decrease is verified by the *Cauchy point*:

$$s^C = -\alpha \frac{\Delta_k}{\|\nabla f(x_k)\|} \nabla f(x_k), \tag{2.9}$$

where

$$\alpha = \begin{cases} 1 & \text{if } \nabla f(x_k)^\top B_k \nabla f(x_k) \leq 0, \\ \min\left(\dfrac{\|\nabla f(x_k)\|^3}{\Delta_k \nabla f(x_k)^\top B_k \nabla f(x_k)}, 1 \right) & \text{otherwise,} \end{cases} \tag{2.10}$$

which is the exact solution for the steepest descent direction m_k (2.7) considering $\|s\| \leq \Delta_k$. The computation of s^C has the advantage of being cheap, and can be a default output when more sophisticated methods fail, providing in any case a solution ensuring (2.8). Theoretically, when any method solving (2.4) guarantees s to satisfy (2.8), then the trust-region method is globally convergent as the line search does (2.1.1). In particular, fixing $\eta_1 > 0$ obliges s to bring a decrease to f, and make the iterates of the trust-region globally converge [39, Theorem 6.4.6]:

$$\lim_{k \to \infty} \|\nabla f(x_k)\| = 0. \tag{2.11}$$

The scheme of a quadratic trust-region method is summarized in Algorithm 2.1.2, modelled after [39, Algorithm 6.1.1].

If $B_k = \nabla^2 f(x_k)$, then Algorithm 2.1.2 is a Newton trust-region method. The next section discusses how approximate $\nabla^2 f(x_k)$ with B_k. By doing so, it avoids evaluating $\nabla^2 f(x_k)$, which is generally assumed too costly.

Algorithm 2.1.2 Quadratic Trust-Region Algorithm

Choose $x_0 \in \mathbb{R}^n$, $\Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$
Choose $B_0 = B_0^\top \approx \nabla^2 f(x_0)$. *Initial approximation*
for $k = 0, 1, \ldots$ **do**
  Compute an approximate solution s_k of (2.4) that satisfies (2.8) considering m_k in (2.7).
  Compute the ratio ρ_k (2.5)
  if $\rho_k \geq \eta_1$ **then** *successful step*
    set $x_{k+1} = x_k + s_k$
  else *unsuccessful step*
    set $x_{k+1} = x_k$
  end if
  Update the trust-region radius according to (2.6)
end for

2.1.3 Quasi-Newton methods

Quasi-Newton methods, pioneered by Davidon [41] in 1959, seek to improve upon the convergence properties of first-order methods without incurring the cost of evaluating second derivatives exactly, as in Newton's methods. They do so by iteratively updating an Hessian approximation. Among quasi-Newton methods, *secant* methods have proven to be particularly effective and update the Hessian approximation based on local information, including: the last step s and an approximation of the function curvature along s from first-order information only. Some secant methods have properties ensuring fast local convergence; although convergence typically occurs at a superlinear rate instead of the quadratic rate of Newton's methods [45]. Secant methods begin with an initial symmetric approximation $B_0 \approx \nabla^2 f(x_0)$ at the initial guess x_0, and seek for an updated approximation satisfying the *secant equation*

$$B_{k+1} s_k = y_k, \quad k \geq 0 \tag{2.12}$$

where $s_k \coloneqq x_{k+1} - x_k$ is the last step taken, and $y_k \coloneqq \nabla f(x_{k+1}) - \nabla f(x_k)$. The secant equation results from the fact that

$$m_{k+1}(s) = f(x_{k+1}) + \nabla f(x_{k+1})^\top s + \tfrac{1}{2} s^\top B_{k+1} s,$$

and that $\nabla m_{k+1}(-s_k)$ is known to be $\nabla f(x_k)$, calculated at the previous iteration. Hence, a condition onto B_{k+1} can be expressed as:

$$\nabla m_{k+1}(-s_k) = \nabla f(x_{k+1}) - B_{k+1} s_k = \nabla f(x_k),$$

which can be reformulated as :

$$B_{k+1}s_k = \nabla f(x_{k+1}) - \nabla f(x_k) = y_k.$$

Because (2.12) in itself does not determine B_{k+1}, additional conditions are imposed. Among others, B_{k+1} should remain symmetric, and $B_{k+1} - B_k$ should be of rank-1 or rank-2. This section does not intend to give a complete account of secant methods, even less of quasi-Newton methods; consequently, the interested reader may refer to [45] for more details. For reference, the most commonly used secant method in practice is the BFGS method—Broyden [20], Fletcher [59], Goldfarb [67], Shanno [141]— which is based on the following formula update

$$B_{k+1}^{\text{BFGS}} = B_k - \frac{(B_k s_k)(B_k s_k)^\top}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}. \tag{2.13}$$

The BFGS update preserves the positive definiteness of B_k as long as the curvature condition

$$s_k^\top y_k > 0 \quad \text{or} \quad s_k^\top y_k \geq \epsilon > 0 \ \text{ in practice}, \tag{2.14}$$

is fulfilled, which automatically holds when the process determining s_k verifies (2.3), e.g. a Wolfe line search satisfying both (2.2) and (2.3). When B_k does not need to be positive definite, one can use the SR1 update formula [41]:

$$B_{k+1}^{\text{SR1}} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^\top}{(y_k - B_k s_k)^\top s_k}. \tag{2.15}$$

To avoid numerical instability, it is customary to only perform the SR1 update provided that

$$|s_k^\top (y_k - B_k s_k)| \geq \epsilon_2 \|s_k\| \|(y_k - B_k s_k)\|, \quad \epsilon_2 > 0. \tag{2.16}$$

As f may not be convex, having B_k not necessarily positive definite may result in a more realistic Hessian approximation. When B_k possesses negative curvatures, minimizing (2.7) is impossible as m_k is not bounded. As a consequence, a line search method must be adapted if one seeks for the line search to stop. Conversely, a trust-region guarantees $\|s\| \leq \Delta_k$, which limits the size of s and the influence of the quadratic term of m_k onto the result s. Nevertheless, the negative curvature case should be handled properly when solving (2.4). The next section describes a method solving both line search and trust-region methods when the quadratic subproblem is not convex. The adaptation of the Algorithm 2.1.2 for the needs of quasi-Newton methods is described in Algorithm 2.1.3.

An important point to keep in mind is that there is no particular reason to expect the vectors

Algorithm 2.1.3 Quasi-Newton Trust-Region Algorithm

Choose $x_0 \in \mathbb{R}^n$, $\Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$
Choose $B_0 = B_0^\top \approx \nabla^2 f(x_0)$. *Initial approximation*
for $k = 0, 1, \ldots$ **do**
  Compute an approximate solution s_k of (2.4) that satisfies (2.8) considering m_k (2.7).
  Compute the ratio ρ_k (2.5)
  if $\rho_k \geq \eta_1$ **then** *successful step*
    set $x_{k+1} = x_k + s_k$
    update B_k using (2.13) or (2.15) with $y_k \coloneqq \nabla f(x_{k+1}) - \nabla f(x_k)$ and s_k
  else *unsuccessful step*
    set $x_{k+1} = x_k$
  end if
  Update the trust-region radius according to (2.6)
end for

$B_k s_k$ and y_k to be sparse. As a consequence, even though B_0 and $\nabla^2 f(x_k)$ may be sparse, B_{k+1} will typically be dense. Thus, in the case of a sparse problem f, the structure of B_{k+1} bears no resemblance to that of $\nabla^2 f(x_{k+1})$. A symmetric dense matrix $B_k \in \mathbb{R}^{n \times n}$ is the simplest data structure allowing the memorization of all the quasi-Newton updates. Unfortunately, a dense matrix dashes any hopes of efficient storage for large-scale problems.

To overcome this issue, several authors proposed some solutions. Powell [127] proposed updates that take sparsity into account and do not modify structural zeros. Toint [146] shows that performing those updates involves the solution of sparse symmetric and positive-definite linear systems. Shanno [140] derives sparse updates as solutions to norm-minimization problems, analogously to the way standard updates such as (2.13) are derived, and, in particular, reports promising numerical performance with the sparse BFGS update. However, the numerical performances of those methods remain unsatisfactory [146] and were quickly overshadowed by other methods.

2.1.3.1 Limited-memory variant

In parallel of sparse quasi-Newton approximations, limited-memory approximations emerged [21, 23, 101, 102, 114]. The limited-memory variants of BFGS and SR1 are commonly referred to as LBFGS and LSR1. They take advantage of the low-rank's updates to store only the $m \geq 1$ most recent pairs $\{s_j, y_j\}$ and apply (2.13) or (2.15) implicitly to update an initial approximation $B_k^{(0)} = (B_k^{(0)})^\top$ ($\succ 0$ for LBFGS) chosen afresh at each iteration [114]. A simple loop can be devised to compute operator-vector products $B_k v$ by scanning the pairs $\{s_j, y_j\}$ in store and $B_k^{(0)}$. B_k is a *linear operator*, representing a linear application $v \to B_k v$ without the intrinsic cost of a matrix. Practically, limited-memory variants allow users to

predefine the storage requirements in $\Theta(2mn)$, where m is the memory set beforehand, and efficiently compute operator-vector products without explicitly forming B_k. In these methods, B_k incorporates a computationally cheap initializer $B_k^{(0)}$, e.g. $B_k^{(0)} = \lambda_k I$ with λ_k equals to 1 or $y_k^\top y_k$. B_k sums $B_k^{(0)}$ to m quasi-Newton updates $X_l X_l^\top,\ 1 \leq l \leq m, X_j \in \mathbb{R}^{n\times 1}$ or $\mathbb{R}^{n\times 2}$, where each X_l represents a quasi-Newton update. Therefore, the complexity of the product $B_k v$ is computed by summing all $X_l X_l^\top v$, setting the complexity of $B_k v$ to $\Theta(mn)$ or $\Theta(2mn)$ whether $X_j \in \mathbb{R}^{n\times 1}$, e.g. (2.15), or $X_j \in \mathbb{R}^{n\times 2}$, e.g. (2.13). However, the computation of new vectors $B_k s_k$ every time a new pair (s_k, y_k) replaces the oldest pair retained (s_{k-m}, y_{k-m}) makes the update of all X_l in $\Theta(\frac{3}{2}m^2 n)$. The complexity of the update can be reduced to $\Theta(2mn)$ by the implementation of a block representation of LBFGS or LSR1, a variant where B_k is written as the result of several matrix-matrix products [23]. Later in the thesis, the sum of $B_k^{(0)}$ and the first j quasi-Newton updates $B_k^{(0)} + \sum_{l=1}^{j} X_l X_l^\top$ will be denoted as $B_k^{(j)},\ 1 \leq j \leq m$.

A line search or a trust-region integrates a quasi-Newton approximation B_k by considering m_k as a quadratic approximation of $f(x_k)$ such as (2.7). Therefore, both Algorithm 2.1.1 and Algorithm 2.1.2 only need to get an initial guess $B_k^{(0)}$ and update B_k when the step s_k has been accepted, after computing y_k and verifying that the numerical safeguards of the chosen quasi-Newton formula hold. The next section details a method using only $B_k v$ products to solve a symmetric positive definite linear system, and one variant devised for the minimization of a quadratic subproblem (2.4) or (2.1) where B_k does not need to be positive definite.

Finally, it is worth noting that Liu and Nocedal [101] developed the most successful implementation among limited-memory quasi-Newton methods and that it does not approximate $\nabla^2 f(x_k)$. Instead, this line search method updates directly $H_k^{\text{BFGS}} \approx (\nabla^2 f(x_k))^{-1}$ instead of B_k^{BFGS} and therefore obtains $s = -H\nabla f(x_k)$ without solving a linear system. A similar method using H_k will be used later in Chapter 4 and Chapter 5 for numerical comparisons.

2.1.4 Conjugate gradient and the truncated variant

The efficiency of a trust-region method or a line search relies heavily on the procedure used to compute respectively s_k or d_k. For this thesis, both s_k or d_k are computed with the truncated conjugate gradient method [143], which is a variant of the conjugate gradient method [84].

The conjugate gradient method is an iterative method solving $Ax = b$ where $A = A^\top \succ 0$ [84]. Furthermore, it can be seen as a method solving a quadratic strictly convex problem $q(x) = \frac{1}{2}x^\top Ax - b^\top x$ as $\nabla q(x) = Ax - b = 0$ describes the minimum of q. Whereas direct methods—factorizing A by a product of structured matrices before solving $Ax = b$—require access to the matrix A, the conjugate gradient only needs the linear application $v \to Av$. This

feature synergizes well with limited-memory quasi-Newton operators and allow the resolution of large linear systems with little storage. In the case A is not positive definite, $\nabla q(x) = 0$ does not characterize a minimum of (2.7), and does not necessarily return a descent direction for a line search (2.1) or a decreasing candidate within the trust-region (2.4). After describing how the conjugate gradient operates for $A \succ 0$, the adaptions for quadratic non-convex inexact quasi-Newton methods are discussed.

The conjugate gradient method performs successive exact line searches onto the quadratic problem q along the *conjugate directions* p_i:

$$p_i^\top A p_j = 0, \forall i \neq j.$$

After an exact line search, it sets the next iterate

$$x_{k+1} = x_k + \alpha_k p_k, \quad \alpha_k = -\frac{r_k^\top p_k}{p_k^\top A p_k}, \quad r_k = \nabla q(x_k) = A x_k - b.$$

Iteratively, r_k can be updated as $r_{k+1} = r_k + \alpha_k A p_k$. Unlike a coordinate exact line search, which performs successive exact line searches along e_i without guarantying convergence in a finite number of iterations, the use of conjugate directions guarantees to converge after n iterations in exact arithmetic [84]. The miracle of the conjugate gradient iterates is to generate any conjugate direction p_k from x_k, A and p_{k-1}

$$p_k = -r_k + \beta_k p_{k-1}, \quad \beta_k = \frac{r_k^\top A p_{k-1}}{p_{k-1}^\top A p_{k-1}},$$

and thus, keeping the method's storage minimal as all p_{k-i}, $1 \leq i \leq k-2$ are ignored. The Algorithm 2.1.4 summarizes the conjugate gradient method described above.

The conjugate gradient possesses several properties:

- the residuals r_i are mutually orthogonal. Therefore, α_k can be simplified to $\frac{r_k^\top r_k}{p_k^\top A p_k}$;
- in exact arithmetic, the methods converges in n or r iterations, where r is the number of distinct eigenvalues of A.

Furthermore, by setting x_0 to 0 the norm of x_k increases monotonically

$$\|x_{k+1}\|^2 = \|x_k\|^2 + 2\alpha_k x_k^\top p_k + \alpha_k^2 \|p_k\|^2,$$

knowing that $x_k^\top p_k$ is proven to be superior to 0 [115, Theorem 7.3].

Algorithm 2.1.4 Conjugate Gradient Method [84]

$A = A^\top \succ 0, b$ *Input*
initial guess x_0, e.g. $x_0 = 0 \in \mathbb{R}^n$
$r_0 = Ax_0 - b, p_0 = -r_0$
while $r_k \neq 0$ **do**
  $\alpha_k = -\frac{r_k^\top p_k}{p_k^\top A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k + \alpha_k A p_k$
  $\beta_{k+1} = \frac{r_{k+1}^\top A p_k}{p_k^\top A p_k}$
  $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$
  $k = k + 1$
end while

The Algorithm 2.1.4 may also integrate a preconditioner to improve the condition number of the linear system and gather its eigenvalues. The new linear system solved is:

$$C^{-\top} A C^{-1} \bar{x} = C^{-\top} b, \quad \text{considering} \quad \bar{x} = Cx.$$

For practical efficiency, a favorable trade-off must be found. Indeed, the computational effort gained from the reduction of iterations must compensate the introduction of computation into the Algorithm 2.1.4 made to incorporate a preconditioner C.

An important case appears when A is not positive definite, then computing α_k or β_k may be infeasible. Therefore, the conjugate gradient method must be adapted for the line search and the trust-region contexts, which seek to approximately minimize a quadratic approximation (2.7).

2.1.4.1 Line search adaptation

For an inexact line search, the determination of d_k by minimizing m_k (2.7) is equivalent to $B_k d_k = -\nabla f(x_k)$ when $B_k \succ 0$, which is covered by Algorithm 2.1.4. Conversely, when $B_k \not\succ 0$, one of the conjugated direction may lie on a negative curvature $p_i B_k p_i \leq 0$. In that case, instead of setting α_k to ∞, d_k is set to the previous conjugated direction p_{i-1}, which is a descent direction. If the first conjugate gradient direction generated finds a negative curvature, i.e. $p_1 B_k p_1 \leq 0$, then $d_k = p_0 = -\nabla f(x_k)$. Additionally, instead of iterating until $\|r_k\| = 0$, the conjugate gradient method loops until $\|r_k\| \leq \eta_k \|\nabla f(x_k)\|$, which prevents numerical issues. By setting $\eta_k = \min(0.5, \sqrt{\|\nabla f(x_k)\|})$ the inexact line search guarantees a superlinear convergence rate[2].

[2] [115] p.169 before Algorithm 7.1

2.1.4.2 Trust-region adaptation

For the trust-region context, Steihaug developed an efficient variant of the Algorithm 2.1.4 named *truncated-conjugate gradient* [143, 147]. The idea is to set x_0 to 0 —making its norm increasing monotonically— and check after each iteration if the current iterate remains in the trust-region, i.e. if $\|x_k\| \leq \Delta$ holds or not. When the minimization along p_k leads to a next iterate $x_k + \alpha_k p_k$ outside the trust-region, then x_{k+1} is set to $x_k + \alpha_k p_k$ such that $\|x_k + \alpha_k p_k\| = \Delta$. Such cases may appear when p_k is a non-positive curvature of B_k, i.e. $p_k^\top B_k p_k \leq 0$, or simply by accepting a larger step $\alpha_k p_k$ than the trust-region permits.

By design, the point found by the truncated conjugate gradient method will bring a decrease at least equal to the Cauchy's point. In the worst case scenario, the algorithm stops during the minimization of p_1 either because $p_1^\top B_k p_1 \leq 0$ or $\|\alpha_1 p_1\| > \Delta$, then the algorithm terminates and returns x_1 set to the Cauchy point s^C (2.9). If the minimization along p_1 is a success, the minimization of subsequent p_k will ensure that $m_k(p_k) \leq m_k(p_1) \leq m_k(s^C)$, making them satisfy the sufficient decrease condition (2.8). Hence, a quadratic trust-region algorithm using the truncated conjugate gradient converges.

2.1.5 Computing derivatives

Most of smooth optimization methods compute derivatives either to find iteratively the next iterate or to check if the current iterate x_k is a local optimum. There exist several ways to numerically approximate one partial derivative of f:

$$\frac{\partial f(x)}{\partial x_i} \coloneqq \lim_{\epsilon \to 0} \frac{f(x + \epsilon e_i) - f(x)}{\epsilon}, \quad f : \mathbb{R}^n \to \mathbb{R} \in \mathcal{C}^1. \tag{2.17}$$

The most applicable method is the finite difference, which only require evaluations of f to approximate $\frac{\partial f}{\partial x_i}$. To do so, it first evaluates $f(x)$ and $f(x + \epsilon e_i)$ to thereafter compute

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + \epsilon e_i) - f(x)}{\epsilon}, \quad \epsilon > 0.$$

Unfortunately, finite difference has a severe drawback since ϵ is limited by the machine precision utilized (mainly floating point reals) and therefore cannot infinitely approach 0 as in (2.17). This limitation introduces round off errors during the approximation of a partial derivative. Round off errors are accentuated as the storage of the floating arithmetic variables shrinks, to eventually become the most prevalent contribution of the partial derivative approximation.

When the final user knows and can access the expression tree of f, he can apply other methods

requiring usually less computational resources and diminishing round off errors. One method is the symbolic differentiation, which constructs a new expression tree for each partial derivative of f by applying the derivation rules recursively on the nodes composing the expression tree of f. However, when the expression tree of $f : \mathbb{R}^n \to \mathbb{R}^m$ is consequent, storing either n or m expression trees of similar size to f may be unrealistic.

Alternatives to symbolic differentiation exist, headed by automatic differentiation (AD), which computes on the fly the derivatives of f by going through its expression tree. Unlike symbolic differentiation, AD produces numerical values of derivatives without creating explicitly its analytical expression. Without fully detailing AD, the remaining of the section briefly recalls how the modes of AD work and their advantages. Numerous literature exists on AD, such as [64, 73] or [115, Section 8.2], that an interested reader may consult for more insights.

AD merely applies the principle of the chain rule onto $f : \mathbb{R}^n \to \mathbb{R}^m$ that is a sequence of differentiable operators. For example, suppose $f(x) = g(h(x))$, where $g : \mathbb{R}^p \to \mathbb{R}^m$ and $h : \mathbb{R}^n \to \mathbb{R}^p$, then the generalized chain rule is:

$$\nabla f(x) = \begin{pmatrix} \sum_{i=1}^{p} \frac{\partial g_1}{\partial h_i} \nabla h_i(x)^\top \\ \vdots \\ \sum_{i=1}^{p} \frac{\partial g_m}{\partial h_i} \nabla h_i(x)^\top \end{pmatrix}, \tag{2.18}$$

which explicit ∇f from the variations of g compared to those of h, i.e. $\frac{\partial g}{\partial h_i}$, and ∇h. The chain rule can be applied recursively on the sequence of operations forming f in two different fashions. It may start either from the leaves of the expression tree, i.e. the variables, or from its root after a proper evaluation of f, resulting respectively in: the *forward* mode and the *reverse* mode. The machine learning community refers to the reverse mode as the *back propagation*.

2.1.5.1 Automatic differentiation: forward mode

The forward mode initiates from the leaves with a seed vector $s \in \mathbb{R}^n$ for which the m directional derivatives of f will be computed. Thereafter, it propagates the computation of directional derivatives through the intermediate nodes of f up to the root, making the directional derivatives of every sub function composing f available, e.g. $h(x)$ or $g(x)$. In general $s_i = e_i,\ i = 1, ..., n$, and each s_i allows the computation of m partial derivatives $(\frac{\partial f_1}{\partial x_i}, \ldots, \frac{\partial f_m}{\partial x_i})$ in one sweep. As a result, the seed matrix $S = I_n$ permits to compute every partial derivative from f as in (2.18). The implementation of the forward mode AD can be done by using dual numbers, implemented by overloading all the operators that f uses and

without requiring supplementary storage.

2.1.5.2 Automatic differentiation: reverse mode

The reverse mode operates in two stages, the first one evaluates $f(x)$ starting from the leaves of the expression tree set to x, while the second computes the partial derivatives starting from the root. During the (second) backward pass, each node accumulates the contributions it has made to its direct predecessor nodes (closer to the root) into its *adjoint* value. The contribution depends on the derivative rule set for the operation that the predecessor node represents as well as the predecessor adjoint value. By storing adjoint values, the reverse pass needs to store a tape shaped as the expression tree of f, which can be an issue for a large numerical process. Usually, a modelling software integrates right from the start a suitable storage for adjoint values to the expression tree of f, to keep their memorization affordable. Initially, all adjoint values are set to 0 except for the root node, corresponding to the f_j of interest which is set to 1 and initiates the backward pass. One reverse mode sweep computes the gradient of f_j: $(\frac{\partial f_j}{\partial x_1}, \ldots, \frac{\partial f_j}{\partial x_n})$. Therefore, m backward passes are needed to compute all partial derivatives from f as in (2.18). The Figure 2.1 illustrates an example of the reverse mode for $f(x) = (4.5 - (x_1 x_2 + x_3 x_4))^2$. In this graph, each node is identified by a number, e.g. the node ⑨ represents the difference between the node ⑦ and node ⑧. In addition, each node contains: two values red (left) and blue (bottom right), which respectively inform about the node value resulting from the forward pass and the adjoint value computed during the reverse pass assuming $x_1 = 1.5$, $x_2 = 2$, $x_3 = 2.5$ and $x_4 = 3$. The forward values propagated from the leaves depend on the operators they go across. The reverse values $\frac{\partial f}{\partial ①}$ for all $1 \leq l \leq 10$ are computed subsequently, ending with $\frac{\partial f}{\partial ①}$, $1 \leq l \leq 4$ which are the partial derivatives of interest. Later, the Figure 3.1 illustrates another example of reverse automatic differentiation for a (group) partially-separable function.

More generally, the time for computing one forward or reverse sweep is roughly a multiplication by five times of the computation time needed to evaluate f [73]. Therefore, one should prefer the reverse mode when $m \ll n$ and the forward mode if $n \ll m$. Hence, if $m = 1$, one should prefer the reverse mode, as long as the tape's storage is not an issue. When n and m are close, both modes have comparable performance.

This section has presented the main concepts related to the quasi-Newton methods. In particular the line search and the trust-region methods, which can both employ a quadratic approximation of $f(x_k)$ to model a subproblem at each iteration. To avoid the computation of the Hessian, the quadratic approximation relies on a quasi-Newton approximation of the Hessian and the computation of derivatives provided by the automatic differentiation. The

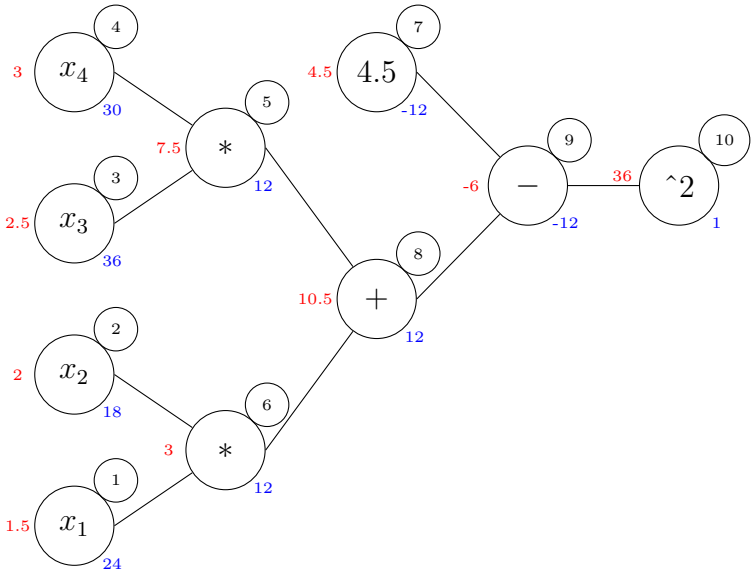


Figure 2.1 Automatic differentiation for $f(x) = (4.5 - (x_1x_2 + x_3x_4))^2$

resulting quadratic subproblem can be solved with the (truncated) conjugate gradient method. Altogether, these concepts may define a convergent quasi-Newton method.

The next section recalls the basis for supervised learning. Notably, the management of the dataset, the neural network architecture and the training formulated as an optimization problem.

2.2 Supervised learning and parallelization of neural network training

This section describes the key components composing the training of a multiclass classification neural networks, whose purpose is to classify any input into one of the C classes set in advance [48]. In practice, a neural network architecture calculates a score c_j for each class. Consequently, a neural network can be conceptualized as a function $c : \mathbb{R}^n \to \mathbb{R}^C$ that will try to classify any given input that conforms to its architectural specifications. Among the C scores, it selects the class with the highest score $\arg\max_{j=1,\dots,C} c_j$. Those problems are a part of the supervised learning, i.e., where the neural network is trained from a dataset. In this thesis, the topic of interest is the computer vision, considering pictures as neural network inputs. Before introducing a neural network training problem, three ingredients are needed:

- a dataset, which is problem specific, as recalled in Section 2.2.1;
- a neural network architecture, described in Section 2.2.2 with a focus on how computations are propagated through the neural network from the inputs of the dataset;

- a loss function, which quantifies the correctness of the class(es) determined by the neural network for one or several inputs, as presented in Section 2.2.3.

The loss function will finally be used in Section 2.2.4 as the objective function of the optimization problem training the neural network.

Note: The variables of a neural network are denoted with w instead of x, which is commonly used for the neural network inputs.

2.2.1 Dataset

A dataset consists of a collection of observations denoted as $\mathcal{X}$, paired with their corresponding labels $\mathcal{Y}$. Here, each $x \in \mathcal{X}$ represents an individual observation, and its corresponding label $y \in \mathcal{Y}$ is characterized by $1 \leq y \leq C$, $y \in \mathbb{N}$. A dataset contains L pairs of observation-label $\{(x^{(l)}, y^{(l)})\}_{l=1}^{\mathsf{L}}$. Any input $x^{(l)}$ can be vectorized to become an input of the neural network architecture, i.e., $x^{(l)} \in \mathbb{R}^d$.

Usually, the dataset is split into two segments: the training dataset and the test dataset. The training dataset is used to train the neural network while the test dataset tests the real effectiveness of the neural network on independent data. In particular, the percentage of correct recognition over the entire test dataset can assess the neural-network's capacity, this is the *accuracy*.

Later on, our numerical results focus on two specific datasets: MNIST [97] and CIFAR10 [94], both encompassing labelled pictures of ten distinct classes ($C = 10$). MNIST comprises grayscale images of handwritten digits, each having dimensions of 28×28 pixels. Conversely, CIFAR10 consists of color images (i.e. an RGB image) with dimensions of 32×32 pixels. Although both datasets have the same range of labels ($C = 10$), they differ on the dimension of the vectorized x. Consequently, an architecture made for MNIST can not be used to classify CIFAR10 pictures. Lastly, the training dataset of MNIST encompasses 60 000 labelled pictures while the test dataset gathers 10 000 labelled pictures. Similarly, CIFAR10 regroups 50 000 labelled pictures in its training dataset and 10 000 in its test dataset.

When the optimizer has gone through the complete training dataset once, the optimizer is said to have reach one *epoch*. The capacity of a neural network is frequently measured depending on how many data the neural network has seen (scaled on epochs).

2.2.2 Neural network architecture

A neural network architecture is composed of K layers. The k-th layer aggregates a collection of neurons V^k and a connection with the previous layer outputs. The i-th neuron of the k-th layer is V_i^k. Every neuron is an intermediary node of the computational graph that the neural network represents. The set of all neurons is denoted as $V = \cup_{k\in\{1,\dots,K\}} V^k$ the union of all neuron layers. The neurons composing V^0 are particular, since their values are directly set from the input, i.e. $|V^0| = d$ and $V_i^0 = x_i$.

The connection between the neuron layers V^{k-1} and V^k is denoted W^k, completed with a bias b^k. The bias can be seen as an additional unitary neuron from V^{k-1} connected with the neurons of V^k, e.g. W^0 is an identity function. Both dimensions of W^k and b^k depend on $|V^{k-1}|$, $|V^k|$ and how the neurons are connected. The variable vector $w \in \mathbb{R}^n$ aggregates a vectorized version of all W_k and b_k. There exist many ways to connect layers, and, most of them make intervene variables, also known as weights. Consequently, the modification of variables parametrizing W^k will change the neuron evaluations of the subsequent $V^{k+j},\ 0 \leq j \leq K-k$. In particular, the training seeks to modify variables until the neural network meets the expectations set. All connections between layers are not exhaustively recalled here, but two of them are detailed, as they will be used later in Chapter 5.

A *dense* layer links all neurons from V^{k-1} to all neurons of V^k. To simplify further equations, W^k is written as a matrix $W^k \in \mathbb{R}^{|V^k|\times|V^{k-1}|}$, where $W_{i,j}^k$ is the variable between V_j^{k-1} and V_i^k, while b^k is a vector $b^k \in \mathbb{R}^{|V^k|}$. As a consequence, all the variables between the neurons of V^{k-1} and V_i^k are gathered by the i-th line of W^k, i.e. $W_{i,:}^k = W^k e_i$, completed with the bias b_i^k. The value of $V_i^k(x;w)$ is then computed from $V^{k-1}(x;w)$, $W_{i,:}^k$ and b_i^k through the mean of an *activation function*, such as the sigmoid:

$$V_i^k(x;w) \coloneqq \frac{1}{1+e^{-(W_{i,:}^k V^{k-1}(x;w)+b_i^{k-1})}}.$$

Other activation functions exist, but this thesis considers mostly the sigmoid as it is infinitely derivable. The Figure 2.2 graphically highlights the variables from W^k necessitated to compute $V_i^k(x;w)$ from an activation function. Additionally, the notation and an architecture composed of dense layers is illustrated graphically Figure 2.3. To make this illustration complete, the architecture displayed integrates a loss function: the negative log-likelihood (composed of the softmax function) described in Section 2.2.3. When the neural network is composed only of dense layers, then $n = \sum_{k=1}^{K} \left(|V^k|+1\right) \times |V^{k-1}|$. This makes the amount of variable increasing quadratically depending on the size of the neuron layers, which rapidly becomes unbearable for any large neural network architecture.

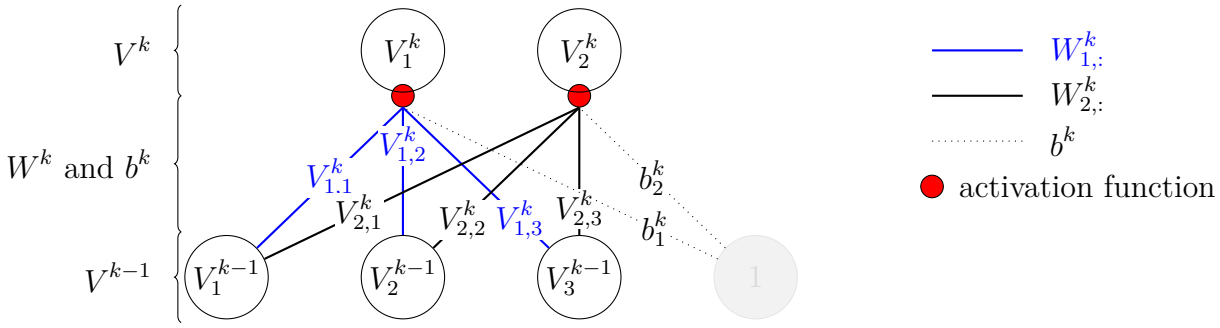


Figure 2.2 Activation function for a dense layer

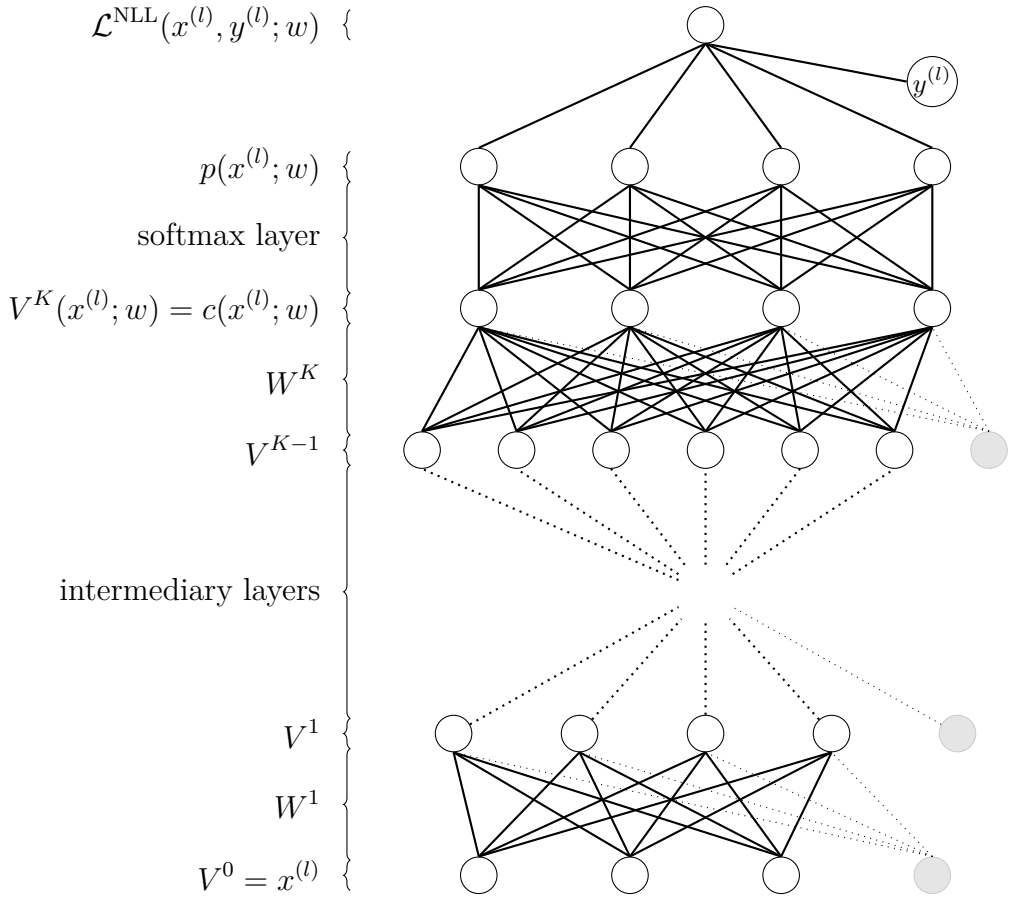


Figure 2.3 Dense neural network and notation

To remedy this issue, Lecun et al. [96] proposed an alternative for computer vision: the *convolutional layer*, which requires fewer variables for the same practical efficiency. Unlike a dense layer, a convolutional layer integrates the spatial information entangled in an input, i.e., the distance between pixels contains information for an image. Practically, a convolutional layer applies one or several kernels, i.e. filters, to a spatially organized input. Every kernel is applied successively onto different input's subsets that completely cover the input and capture its local features. The kernel application and the notation enumerated in the next paragraph are illustrated by the Figure 2.4. Every kernel applies to a channelled input and results in one channel of the output. Any input is channelled, for example, a grayscale image has a one channel (gray shades) while an RGB color image has 3 channels: red shades, green shades and blue shades.

Formally, a convolutional layer takes an input with p_1 channels and applies p_2 kernels to return an intermediate image of p_2 channels. Every kernel is parametrized by p_1, its height h_1, width h_2, a stride p_3 and the size of an artificial border p_4 added on the input. Hence, a kernel contains p_1 channels and each of them is a matrix $K \in \mathbb{R}^{h_1 \times h_2}$. The kernel application traverses the channeled input respecting the stride p_3 and input dimensions. Concretely, each input's channel selects $h_1 \times h_2$ connected components represented by the matrix $J^l \in \mathbb{R}^{h_1 \times h_2}$, $1 \leq l \leq p_1$, Then, each kernel channels K^l multiply component wisely J^l, i.e. $\left(\sum_{l=1}^{p_1} \sum_{i=1}^{h_1} \sum_{j=1}^{h_2} K_{i,j}^l J_{i,j}^l\right) + b$, where b is the bias. Consequently, a convolutional layer contains $(h_1 \times h_2 \times p_1 + 1) \times p_2$ variables, where $+1$ is the bias.

To reduce the size of the intermediate image, a *pooling* layer can be applied, parameterized by its height h_3, its width h_4, and a stride p_5. The main pooling layers include *max pooling*, *min pooling*, and *average pooling*, which respectively take the maximum, minimum, and average of a subset of $h_3 \times h_4$ connected components from the intermediate image. The Figure 2.5 illustrates the max pooling onto one channel of the intermediate image resulting from the Figure 2.4. To perpetuate the positivity nature of a pixel (intensity ≥ 0), it is common to apply an activation function, such as the sigmoid function (2.2.2) on the result of the convolutional layer. The Figure 2.5 does not apply an activation function to maintain simple numerical values.

Convolutional layers are popular because they isolate local features of an image. Also, they require minimal preprocessing and are robust to translations of the neural network input $x^{(l)}$, unlike dense layers.

The evaluation of a neural network architecture evaluates successively all its layers, regardless of their type, starting with $V^0(x; w) = x$, where x is the input and w the variables parametrizing the neural network. Consequently, if x is a grayscale picture, V^0 contains as many neurons

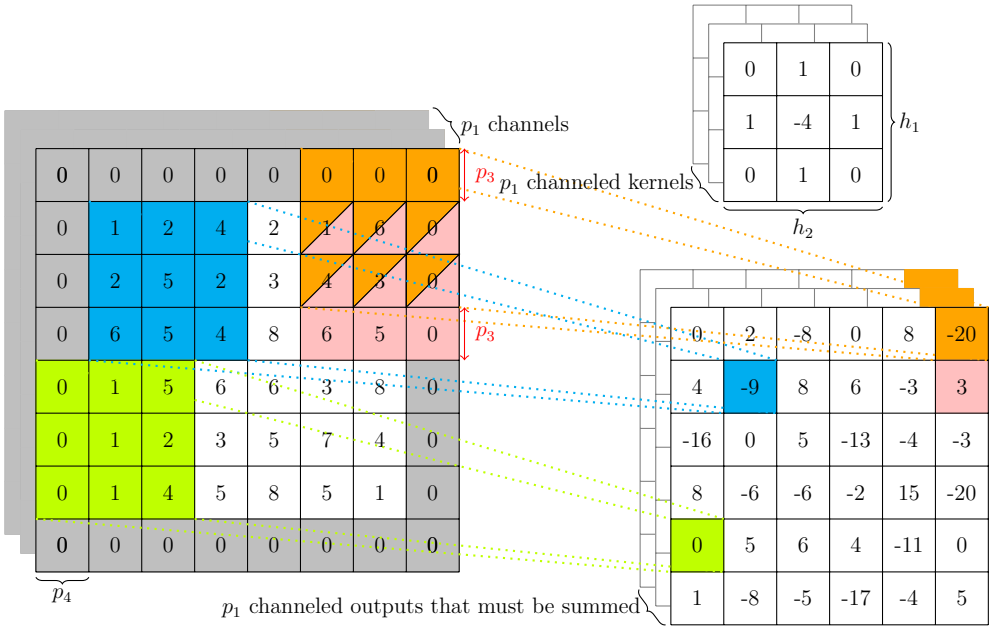


Figure 2.4 Convolutional layer and notation (without activation function)

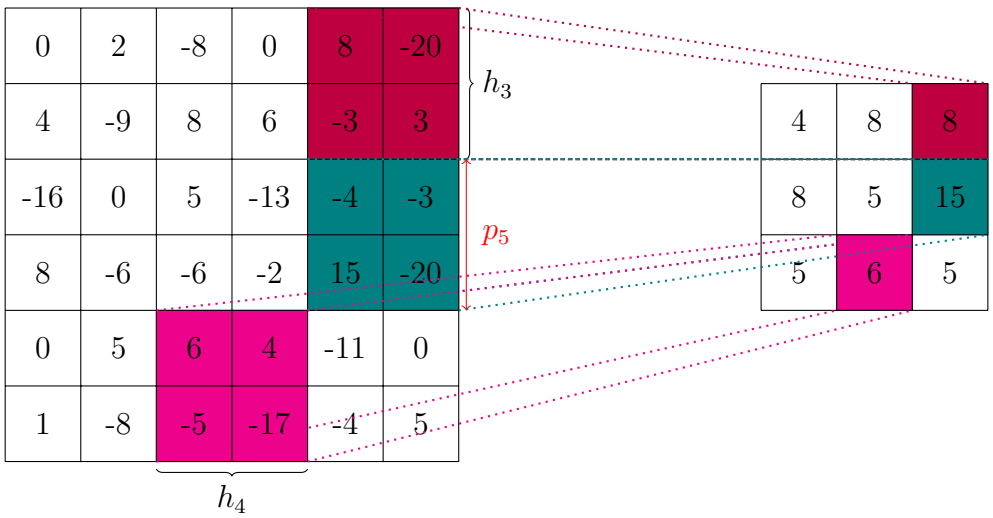


Figure 2.5 Max-pooling

as pixels encompassed in x. The output resulting from the evaluation of V^k is denoted as $V^k(x;w) = (V_1^k(x;w), V_2^k(x;w), \ldots, V_{|V^k|}^k(x;w))$ (w is used for simplicity, only a subset of w is used for evaluating V^k except for V^K). Finally, the output from the neural network architecture $c(x;w) \in \mathbb{R}^C$ is equivalent to $V^K(x;w)$.

2.2.3 Loss function

Given an observation x and a label y, the *loss function* $\mathcal{L}$ seeks to assess if $\arg\max_{1\leq j\leq C} c_j(x;w)$ matches the label y. A popular example of loss function is the *negative log likelihood* $\mathcal{L}^{\text{NLL}}$, based on the *softmax* layer $p(x;w)$ such that:

$$p_i(x;w) \coloneqq \frac{exp(c_i(x;w))}{\sum_{j=1}^{C} exp(c_j(x;w))}, \quad \forall 1 \leq i \leq C, \quad 0 \leq p_i(x;w) < 1, \tag{2.19}$$

which normalizes the scores c_i and provide a probabilistic interpretation, i.e., $\sum_{i=1}^{C} p_i(x,w) = 1$. The negative log likelihood function is applied to a pair (x,y) as

$$\mathcal{L}^{\text{NLL}}(x,y;w) \coloneqq -\log(p_y(x;w)) \ > 0,$$

and can be generalized for a dataset $(\mathcal{X},\mathcal{Y})$ as

$$\mathcal{L}^{\text{NLL}}(\mathcal{X},\mathcal{Y};w) = \tfrac{1}{\mathsf{L}} \sum_{l=1}^{\mathsf{L}} \mathcal{L}^{\text{NLL}}(x^{(l)},y^{(l)};w) = -\tfrac{1}{\mathsf{L}} \sum_{l=1}^{\mathsf{L}} \log(p_{y^{(l)}}(x^{(l)};w)).$$

The value of $\mathcal{L}^{\text{NLL}}$ increases exponentially when the normalized score of the class targeted $p_y(x;w)$ moves away from 1. Such a case occurs when the wrong class is determined $\max_{1\leq j\leq C} c_j(x;w) > c_y(x;w)$ or when the other scores are large enough to diminish $p_y(x;w)$.

The loss function tops the neural network architecture and becomes the root of its expression tree, as illustrated in Figure 2.3. The gradient of $\mathcal{L}$ is generally computed by backpropagation, a synonym of the reverse automatic differentiation, see Section 2.1.5.

The next sections detail:

- the formulation of the optimization problem training a neural network architecture;
- how the hardware limitation forces the resulting optimization problem to be structurally stochastic;
- the most popular optimizers existing to train a neural network;

- how the computation may be distributed either to speed-up training or share the computational effort between several devices/workers;
- one particular problematic known as *federated learning*, which seeks to push the training of a neural network toward *edge devices*–the final user devices, each having limited resources–.

2.2.4 Neural network training

To summarize the previous section, a neural network works properly when the appliance of the loss function to an observation-label couple $\mathcal{L}(x, y; w)$ or the entire dataset $\mathcal{L}(\mathcal{X}, \mathcal{Y}; w)$ is small. Therefore, the minimization of $\mathcal{L}(\mathcal{X}, \mathcal{Y}; w)$:

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(\mathcal{X}, \mathcal{Y}; w), \tag{2.20}$$

will modify w to diminish the loss function value over the entire dataset, which should result in a misclassification reduction, i.e., increase the neural-network's accuracy.

However, datasets have expanded to the point where they can no longer be loaded into a single hardware. For example, the ImageNet dataset comprises over 14 million images [138]. As the evaluation of $\mathcal{L}(\mathcal{X}, \mathcal{Y}; w)$ is considered impossible, the training relies in practice on *minibatches* (X, Y), which are subsets of the entire dataset $(X, Y) \subseteq (\mathcal{X}, \mathcal{Y})$, $|(X, Y)| = L$ where $L < \mathsf{L}$. The minibatch training problem considering $(X, Y) = \left\{(x^{(l)}, y^{(l)})\right\}_{l=1}^{L}$ is then

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(X, Y; w) = \frac{1}{L} \sum_{l=1}^{L} \mathcal{L}(x^{(l)}, y^{(l)}; w). \tag{2.21}$$

To go through the entire dataset, a new random minibatch is loaded at each iteration of the optimizer. Two different minibatches $(X^{(1)}, Y^{(1)})$ and $(X^{(2)}, Y^{(2)})$ result in different values $\mathcal{L}(X^{(1)}, Y^{(1)}; w)$ and $\mathcal{L}(X^{(2)}, Y^{(2)}; w)$, also different to $\mathcal{L}(\mathcal{X}, \mathcal{Y}; w)$. The difference between $\mathcal{L}(X, Y; w)$ and $\mathcal{L}(\mathcal{X}, \mathcal{Y}; w)$ can be considered as a stochastic noise. Hence, selecting random minibatches prevents the resolution of (2.20) by successive resolutions of (2.21) to be deterministic. Therefore, the optimizers requiring minibatches are referred as stochastic minibatch methods.

2.2.5 Neural network optimizers

The foundation of stochastic optimization was laid by Robbins and Monro [137]. The method root lies in gradient loss computation from data subsets, commonly known since as the

stochastic gradient descent (SGD) [96]. SGD represents the prototype among gradient-based methods, where weight updates rely on scalar adjustments derived of first partial derivatives. Generally, gradient based methods are such

$$w_i^{(k+1)} = w_i^{(k)} + \alpha_i^{(k)}(\nabla \mathcal{L}(X^{(k)}, Y^{(k)}; w))_i, \quad k \geq 0, \quad 1 \leq i \leq n, \tag{2.22}$$

where k represents the iteration's index, and $w_i^{(k)}$ is the i-th weight at the k-th iteration. Consequently, optimizers that train a neural network incorporate this stochastic nature by dynamically adapting $a^{(k)} = (\alpha_1^{(k)}, \ldots, \alpha_n^{(k)})^\top$. While not aiming for an exhaustive review of all stochastic gradient-based methods, we list some well-known approaches below. For more comprehensive information, interested readers are directed to surveys on this topic [3, 15]. Momentum was introduced by [112], followed by more recent techniques like AdaGrad [50], RMSProp [85], and Adam [93]. The latter amalgamates AdaGrad and RMSProp to adjust $a^{(k)}$ based on estimates of the first and second gradient moments. Adam will be used in Section 5.2.4 and Section 5.3.1 for comparing neural network trainings. For this reason, the Algorithm 2.2.1 summarizes the algorithm Adam which minimizes the loss function $\mathcal{L}(X, Y; w)$ where w is the weight vector that parametrizes a neural network. The default values for $\alpha, \beta_1, \beta_2, \epsilon$ are usually

Algorithm 2.2.1 Adam [93]

$\alpha > 0$, $\beta_1, \beta_2 \in (0, 1)$
$k = 0$, $m^{(0)} = 0 \in \mathbb{R}^n$, $v^{(0)} = 0 \in \mathbb{R}^n$, $w^{(0)} \in \mathbb{R}^n$
for $k = 0, 1, \ldots$ **do**
    Select a new minibatch $(X^{(k)}, Y^{(k)})$
    $k = k + 1$
    $g^{(k)} = \nabla L(X^{(k)}, Y^{(k)}; w^{(k)})$ *loss function gradient*
    $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1) g^{(k)}$ *Update biased first moment estimate*
    $v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2) g^{(k)}.^2$ *Update biased second raw moment estimate*
    $\bar{m}^{(k)} = \frac{m^{(k)}}{1-\beta_1^k}$ *Compute bias-corrected first moment estimate*
    $\bar{v}^{(k)} = \frac{v^{(k)}}{1-\beta_2^k}$ *Compute bias-corrected second raw moment estimate*
    $w^{(k)} = w^{(k)} - \alpha \bar{m}^{(k)}./(\sqrt{.}\bar{v}^{(k)}. + \epsilon)$ *Update of the weights*
end for

$\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [93]. Also, note that the operations $./$, $.^2$ and $\sqrt{.}$ are element-wise operations applied onto a vector. For example, suppose $g = (g_1, g_2, \ldots, g_n)^\top$, then $g.^2 = (g_1^2, g_2^2, \ldots, g_n^2)^\top$.

Gradient methods have the advantage of requiring fewer memory, unlike basic quasi-Newton methods which rely on the support of a dense matrix. Therefore, only the limited-memory quasi-Newton methods remain applicable to minimize such large optimization problems.

Similarly to the stochastic gradient-based methods, the stochastic limited-memory quasi-Newton methods replace the gradient with stochastic gradient, notably to compute y during quasi-Newton updates (2.13), (2.15), see [4, 25, 26].

Within a stochastic setup, both gradient-based methods and limited quasi-Newton methods achieve sub-linear convergence rates [15]. However, quasi-Newton methods are likely to enhance the hidden constant behind the theoretical asymptotic convergence results [15]. Generally, the hyperparameters of a quasi-Newton method are less sensible than those of gradient-based methods, due to their scale invariant properties. In practice, the gradient-based method Adam remains the most efficient method to train a neural network (at the time of writing).

2.2.6 Neural network parallelization

In addition of the optimizer used, the architecture plays a pivotal role in practical performance, as larger architectures often lead to enhanced results [29]. As a result, training computationally expensive deep neural networks in a reasonable time requires parallel optimization methods that scale to adequate computational resources. In the context of supervised learning, two problems arise. First, the sheer size of training datasets precludes their simultaneous evaluation, requiring the use of minibatches, each containing a fraction of the original dataset. Second, with the continuous expansion of neural network sizes, a solitary hardware component might prove insufficient for storing or training a neural network [43], e.g. GPT-4 is estimated to be parametrized with approximately 10^{12} variables. Several techniques coexist to remedy those issues.

The first parallel scheme, called *data parallelism*, requires the exploitation of a graphical processing unit (GPU). The GPU takes advantage of the minibatch evaluation to dispatch efficiently the computation related to the observation-label couples across its numerous cores [29, 43, 130]. By doing so, the larger the minibatch is, the greater the speed-up is. However, for a single epoch, there is generally more progress by considering smaller minibatches.

When the architecture is too large, a natural idea is to fragment the neural network training, which has produced multiple schemes and implementations, from which two emerge: model parallelism and tensor parallelism. The idea of *model parallelism* is that each worker is tasked with storing a specific neuron layer [83, 87]. To achieve the computation of the loss function and its derivatives, both forward pass and reverse passes are adapted to transfer layer computation outputs to the worker related with its subsequent layer. Conversely, *tensor parallelism* [6, 98] operates by assigning to each worker a slice of one or several layers. Once

all workers have computed the slices pertaining to a given layer, the results are shared among all the workers to enable the next layer evaluation. Both approaches effectively fragment neural network training by incorporating communication among workers. However, in an unfavorable setup, communication costs can significantly impair practical performance [3].

Finally, *Hybrid parallelism* integrates all previous strategies to propose techniques that enhance practical efficiency. For instance, both model parallelism and tensor parallelism can harness data parallelism by appropriately managing minibatches. Adapting model parallelism is straightforward, but for tensor parallelism, the same minibatch must be loaded onto multiple workers. To achieve competitive performance, implementations prioritize maintaining a balanced workload across workers depending on the computational intensity of layers [3].

CHAPTER 3 LITERATURE REVIEW

This chapter constitutes a literature review on the exploitation of partial separability in optimization, derived from Bigeon, Orban, and Raynaud [9]. The Section 3.1 recalls in details the definition of what a partially-separable function is, elaborates on the partitioned structure nature of its derivatives, defines a generalization of partial separability named *group-partial separability*, and provides other definitions of partial separability not related to the content of this thesis. Then, the Section 3.2 introduces the most well-known partitioned quasi-Newton methods and their respective theoretical results. The Section 3.3 groups the description of other continuous optimization methods that exploit partial separability by design, complemented by two methods dedicated to the particular partially-separable problems presented in Section 3.4. The Section 3.5 gathers various works explaining how partial separability helps in computing derivatives, which is a critical factor for efficient numerical methods. Thereafter, the Section 3.6 describes some pieces of software integrating partitioned quasi-Newton updates, based on what the previous sections present. Unlike the contributions encompassed in Section 3.6, which are mostly sequential implementations, the Section 3.7 summarizes several parallel methods or implementations exploiting the partially-separable structure to operate. Before concluding, the Section 3.8 breaks down the exploitation of partial separability for derivative-free methods. Finally, the Section 3.9 proposes a critical analysis of partially-separable optimization and its current limits, which guided the subsequent research of this thesis.

3.1 Partially-separable structure

This section defines thoroughly a partially-separable function, details the partitioned structure of its derivatives, presents a generalization of partial separability and enumerates close "partially-separable" definitions out of this thesis context.

3.1.1 Partially-separable function

Most of the time, the minimization method for $f : \mathbb{R}^n \to \mathbb{R}$ can take advantage of the underlying structure if it can be identified. One interesting information is the knowledge of invariant subspace for which the value of f remains constant. Such a subspace is called *nullspace* $\mathcal{N} \subseteq \mathbb{R}^n$:

$$f(x + s) = f(x), \forall\, x \in \mathbb{R}^n \text{ and } \forall\, s \in \mathcal{N}.$$

As such, that is neither an interesting nor a compelling example, but it takes its meaning when f may be written in the form

$$f(x) = \sum_{i=1}^{N} f_i(x), \quad f_i : \mathbb{R}^n \to \mathbb{R}, \quad i = 1, \ldots, N, \tag{3.1}$$

and each f_i has a nonempty nullspace $\mathcal{N}_i$. Such a situation could occur because each f_i only depends on a subset of variables, but the definition is more general in the sense that nullspaces do not need to be aligned with euclidean axis. Total separability is perhaps the simplest special case of (3.1), it corresponds to the case where each f_i only depends on x_i, $\mathcal{N}_i$ is then $Span(e_i)^\perp$:

$$f(x) = \sum_{i=1}^{n} f_i(x_i), \quad f_i : \mathbb{R} \to \mathbb{R}.$$

It may be generalized to block separable functions, where each f_i only depends on an orthogonal subset of variables:

$$f(x) = \sum_{i=1}^{N} f_i(x_{[i]}), \quad f_i : \mathbb{R}^{n_i} \to \mathbb{R},$$

where $x_{[i]}$ selects a subset of components of x. Consequently, $\mathcal{N}_i$ is spanned by at least all components of x not captured by $x_{[i]}$. Structurally, f_i is such that $\mathcal{N}_i \supseteq \bigcup_{j \in \{1,\ldots,N\} \setminus \{i\}} \mathcal{N}_j$ and $\sum_{i=1}^{N} n_i = n$.

Let's now consider a slightly more elaborate example:

$$f(x) = \sum_{i=1}^{n-1} f_i(x), \quad f_i(x) = (x_i + x_{i+1})^4, \tag{3.2}$$

the nullspace of f_i is $\mathcal{N}_i = \{s \in \mathbb{R}^n \mid s_i + s_{i+1} = 0\}$. The situation is similar if $f_i(x) = \sin(x_i + x_{i+1})$, except that due to periodicity, $\mathcal{N}_i$ can be described as $\{s \in \mathbb{R}^n \mid s_i + s_{i+1} = 0 \pmod{2\pi}\}$. The nullspace cannot always be stated so easily, and sometimes, we must fall back on a subset of $\mathcal{N}_i$ that has a simpler expression. In the examples above, one could choose $\mathcal{N}_i = \{s \in \mathbb{R}^n \mid s_i = s_{i+1} = 0\}$ or $\mathcal{N}_i = \{s \in \mathbb{R}^n \mid s_i = s_{i+1} = 0 \pmod{2\pi}\}$. Such subsets are sometimes called *trivial* nullspaces. In certain cases, finding a subset of the nullspace strictly larger than the trivial nullspace can be difficult, for example if $f_i(x) = x_i x_{i+1}$.

When searching for a minimizer of f, it is interesting for f_i to have a nullspace as large as possible, i.e., the contribution of f_i is confined to a small subset of $\mathbb{R}^n$. For the rest of the thesis, a partially-separable function can be described as

$$f(x) = \sum_{i=1}^{N} \widehat{f}_i(\widehat{x}_i), \quad \widehat{f}_i(\widehat{x}_i) = f_i(x), \quad \widehat{x}_i := U_i x, \tag{3.3}$$

where $U_i \in \mathbb{R}^{n_i \times n}$ and $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$. In such definition, $\widehat{f}_i$ depends only on a subset of variables, or, more precisely, on few linear combinations of a subset of variables captured by U_i. It completes (3.1), by specifying the nullspace of every $f_i(x) = \widehat{f}_i(\widehat{x}_i)$, which corresponds to $Range(U_i)^\perp$. Any object $\widehat{\ }_i$ refers to an object related to the element function $\widehat{f}_i$ instead of f, e.g., $\nabla \widehat{f}_i$ is the i-th element gradient. Consequently, the size of $\widehat{\ }_i$ is related to n_i instead of n.

If the j-th column of U_i has at least one nonzero element, then x_j participates in f_i and is called an *elemental variable* of f_i. The number of elemental variables of f_i is denoted n_i^E. Thus, one possible choice for U_i is the subset of rows of the identity corresponding to elemental variables, denoted U_i^E. For instance, if $f_i(x) = \sin(x_{12} - 2x_{23})(x_7 + x_{12} + x_{23})$, only $n_i^E = 3$ variables participate in f_i, and we may use $U_i = U_i^E$ as the $3 \times n$ matrix composed of rows 12, 23 and 7 of the $n \times n$ identity. The formulation of f_i depending on its elemental variables is

$$\widehat{f}_i^E(y_1, y_2, y_3) \coloneqq \sin(y_1 - 2y_2)(y_3 + y_1 + y_2),$$

so that

$$f_i(x) = \widehat{f}_i^E(U_i^E x), \quad U_i^E \coloneqq \begin{bmatrix} e_{12}^\top \\ e_{23}^\top \\ e_7^\top \end{bmatrix}.$$

For the decomposition (3.1) to be useful, it is necessary that $n_i^E < n$. However, it is also possible to write

$$f_i(x) = \widehat{f}_i^I(\widehat{x}^I), \quad \widehat{x}^I \coloneqq U_i^I x, \quad U_i^I \coloneqq \begin{bmatrix} e_{12}^\top - 2e_{23}^\top \\ e_7^\top + e_{12}^\top + e_{23}^\top \end{bmatrix},$$

where this time, $\widehat{f}_i^I$ depends on $n_i^I = 2$ variables and can be written $\widehat{f}_i^I(y_1, y_2) \coloneqq \sin(y_1)y_2$. Now, U_i^I selects several linear combinations of variables, such that $\widehat{x}_1^I \coloneqq x_{12} - 2x_{23}$ and $\widehat{x}_2^I \coloneqq x_7 + x_{12} + x_{23}$, which are called *internal* variables. Obviously, it is always possible to choose $U_i^I = U_i^E$ but exploiting internal variables is only useful if we identify a choice of U_i^I such that $n_i^I < n_i^E$.

Elemental variables describe the trivial nullspace of f_i whereas proper choices of internal variables describe larger subsets of the nullspace, and perhaps even the entire nullspace. Which representation choose between U_i^E or U_i^I is of importance, since it impacts the efficiency of routines exploiting partial separability. The Section 3.3.3, summarize the approach of Conn et al. [36] to answer this interrogation.

Returning to (3.2), the elemental variables of f_i are x_i and x_{i+1}, so that $n_i^E = 2$ and

$$\widehat{f}_i^E : \mathbb{R}^2 \to \mathbb{R}, \quad \widehat{f}_i^E(\widehat{x}_1^E, \widehat{x}_2^E) = (\widehat{x}_1^E + \widehat{x}_2^E)^4, \quad U_i^E := \begin{bmatrix} e_i^\top \\ e_{i+1}^\top \end{bmatrix}.$$

Alternatively, we may set $n_i^I = 1$ and

$$\widehat{f}_i^I : \mathbb{R} \to \mathbb{R}, \quad \widehat{f}_i^I(\widehat{x}_1^I) = (\widehat{x}_1^I)^4, \quad U_i^I := \left[e_i^\top + e_{i+1}^\top\right].$$

Therefore, any s in the trivial nullspace $\mathcal{N}_i^E$ is such that $U_i^E s = 0$, and any $s \in \mathcal{N}_i$ is such that $U_i^I s = U_i^E s = 0$.

Although matrices are convenient to define U_i in (3.3), an efficient computer representation of partial separability represents U_i with a linear operator, e.g., via an n_i-vector identifying the variables selected by U_i. The Section 3.1.2 presents the partitioned derivatives of f and details how their computations depend directly on U_i and n_i. Therefore, having operator-vector product $v \to U_i v$ more efficient than a dense matrix also helps during the assembly of element derivative contributions.

In the rest of the thesis, no distinction is made between an internal and an elemental function, gradient or Hessian, they will be referred as element function, element gradient or element Hessian, respectively $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$, $\nabla \widehat{f}_i \in \mathbb{R}^{n_i}$ and $\nabla^2 \widehat{f}_i \in \mathbb{R}^{n_i \times n_i}$. However, the pieces of software presented in Chapter 6 consider $\widehat{f}_i^E$ and U_i^E for which:

$$U_i^E = \begin{bmatrix} e_i^\top \\ e_j^\top \end{bmatrix}, \quad \text{is stored with the vector} \quad \texttt{Ui = [i, j]}$$

and $v \to U_i v$ selects the components `v[i]` and `v[j]` from the vector `v`.

The relevancy of partial separability in large-scale optimization originates from Griewank and Toint [76], which demonstrated:

Theorem 3.1.1 (derived from Griewank and Toint [76]). *If there exist* $1 \leq i \neq j \leq n$ *such that* $\frac{\partial^2 f}{\partial x_i \partial x_j} = 0$, $\forall x \in \mathbb{R}^n$ *then* f *is partially-separable.*

Theorem 3.1.1 makes any problem having a sparse Hessian partially-separable, and susceptible to consider the partitioned quasi-Newton methods described in Section 3.2. Originally, [76, Theorem 1] establishes the partial separability of f using the sparsity of higher order:

$$\frac{\partial^n f(x)}{\partial x_1 \partial x_2 \ldots \partial x_n} = 0, \ \forall x \in \mathbb{R}^n. \tag{3.4}$$

Consequently, if there is $i \neq j$, such that $\frac{\partial^2 f}{\partial x_i \partial x_j} = 0$, then f satisfies straightforwardly (3.4). Griewank and Toint [76] technically defined f as partially-separable if it satisfies:

$$\sum_{e \in E_0} (-1)^{|e|} f(x_e) = 0, \ \forall x \in \mathbb{R}^n, \tag{3.5}$$

considering the vertex set of the unit cube E_0, such that $\forall e \in E_0$, $e = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n)^\top$ where $\epsilon_i \in \{0, 1\}$ and $x_e = (\epsilon_1 x_1, \epsilon_2 x_2, \ldots, \epsilon_n x_n)^\top$ the projection of x onto the vertex e of the rectangle $R := \{(\alpha_1 x_1, \alpha_2 x_2, \ldots, \alpha_n x_n)^\top$, with $0 \leq \alpha_i \leq 1, \forall\, 1 \leq i \leq n\}$. Thus, $|E_0| = 2^n$, the metric $|e| = \sum_{i=1}^n \epsilon_i$ and the special vertex $e^1 = (1, 1, \ldots, 1)$ is such as $f(x_{e^1}) = f(x)$. By moving e^1 into the right side of (3.5), it simplifies as:

$$f(x) = (-1)^{n+1} \sum_{k=0}^{n-1} (-1)^k \sum_{e \in E_0, |e|=k} f(x_e).$$

Suppose

$$D^e f(x) = \frac{\partial^{|e|} f(x)}{\partial^{\epsilon_1} x_1 \partial^{\epsilon_2} x_2 \ldots \partial^{\epsilon_n} x_n},$$

then, for each e such that $D^e f \equiv 0$ the function $f(x_e)$ can be described as a linear combination of several $f_{\hat{e}}(x_{\hat{e}}) := f(x_{\hat{e}})$, where $\hat{e} \in E_e := \{\hat{e} : \hat{\epsilon}_i \leq \epsilon_i, \ 1 \leq i \leq n$ and $|\hat{e}| \leq |e|\}$. The linear decomposition into sub-functions of every $f(x_e)$ leads to:

$$f(x) = \sum_{\hat{e} \in E} f_{\hat{e}}(x_{\hat{e}}), \tag{3.6}$$

where $E \subseteq \bigcup_{e \in E_0} E_e \subseteq E_0$ a minimal vertex subset such that for any $\hat{e} \in E$, $D^{\hat{e}} f \not\equiv 0$. The formulation (3.6) is equivalent to (3.3) and led the way for studies analyzing f from $f_{\hat{e}}$ nullspaces. In particular, a trivial nullspace occurs for any function independent of certain variables, which is the case for every $f_{\hat{e}}$ since there is at least one $\hat{\epsilon}_i = 0$.

3.1.2 Derivatives of a partially-separable function

Partially-separable functions satisfying (3.3) yield structured derivative computations. If each $f_i \in \mathcal{C}^2$, then

$$\nabla f(x) = \sum_{i=1}^N \nabla f_i(x) = \sum_{i=1}^N U_i^\top \nabla \widehat{f}_i(U_i x) = \sum_{i=1}^N U_i^\top \nabla \widehat{f}_i(\widehat{x}_i), \tag{3.7a}$$

$$\nabla^2 f(x) = \sum_{i=1}^N \nabla^2 f_i(x) = \sum_{i=1}^N U_i^\top \nabla^2 \widehat{f}_i(U_i x) U_i = \sum_{i=1}^N U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_i) U_i. \tag{3.7b}$$

For example, suppose f such as:

$$f(x) = \frac{(x_1 x_3)^4}{x_2^2 + 1} + \frac{(x_3 x_5)^4}{x_4^2 + 1} + \exp((x_1 + x_3 + x_5)^2), \quad f : \mathbb{R}^5 \to \mathbb{R}, \tag{3.8}$$

which can be written as:

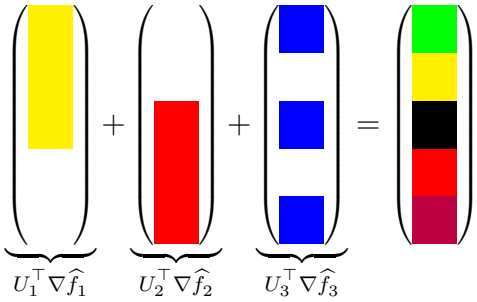
$$f(x) = \widehat{f}_1(x_1, x_2, x_3) + \widehat{f}_2(x_3, x_4, x_5) + \widehat{f}_3(x_1, x_3, x_5), \quad \widehat{f}_i : \mathbb{R}^3 \to \mathbb{R},$$

where

$$\widehat{f}_1(y_1, y_2, y_3) = \frac{(y_1 y_3)^4}{y_2^2 + 1}, \quad \widehat{f}_2(y_1, y_2, y_3) = \frac{(y_1 y_3)^4}{y_2^2 + 1}, \quad \widehat{f}_3(y_1, y_2, y_3) = \exp((y_1 + y_2 + y_3)^2).$$

Therefore, the gradient structure is:

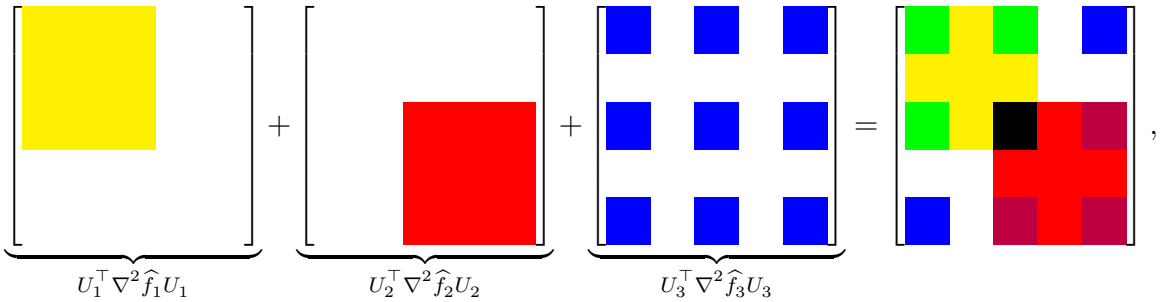
$$\nabla f(x) =$$



,

while the Hessian structure is:

$$\nabla^2 f(x) =$$



,

considering that the yellow, red and blue colors represent respectively the contributions from $\widehat{f}_1, \widehat{f}_2$ and $\widehat{f}_3$. and other colors express the combination of several element contributions for a single partial derivative. The same example is reused in Section 6.3 to describe how the partially-separable structure is automatically detected from an analytical expression stored as an expression graph.

From (3.7a), $\nabla f(x)$ can either be stored as an n-vector accumulating the contributions of each $\widehat{f}_i$, or as the set $\{\nabla \widehat{f}_i(\widehat{x}_i) \mid i = 1, \ldots, N\}$ encompassing N vectors of size n_i, $i = 1, \ldots, N$.

The second option has computational advantages. Firstly, when there are few component changes in x, recomputing ∇f only needs to update few $\nabla \widehat{f}_i(\widehat{x}_i)$. For example, suppose $f(x)$ as (3.8) and that every $\nabla \widehat{f}_i(U_i x)$ is known, then the computation of $\nabla f(x + e_1)$ only needs to recompute $\nabla \widehat{f}_1(U_1(x + e_1))$ and $\nabla \widehat{f}_3(U_3(x + e_1))$, which avoids the unnecessary computation of $\nabla \widehat{f}_2(U_2(x + e_1))$. Secondly, ∇f can be computed in parallel by dispatching $\nabla \widehat{f}_i(\widehat{x}_i)$ to several cores [99]. For more details, the Section 3.5 delves further into the computation of derivatives.

Similarly, instead of assembling $\nabla^2 f(x)$, it is possible to store separately the element Hessians $\{\nabla^2 \widehat{f}_i(\widehat{x}_i) \mid i = 1, \ldots, N\}$ as small dense matrices with their corresponding U_i. This partitioned storage makes possible the computation of Hessian-vector products by accumulating the $\nabla^2 \widehat{f}_i(\widehat{x}_i) U_i v$ contributions and the factorization of $\nabla^2 f(x)$ without ever assembling the matrix [52, 53]. Storing all $\nabla^2 \widehat{f}_i(\widehat{x}_i)$ requires $\frac{1}{2} \sum_{i=1}^{N} n_i(n_i + 1)$ real numbers, to which we must add the storage of U_i for $i = 1, \ldots, N$. Clearly, if the resulting overall storage is significantly smaller than $\frac{1}{2} n(n + 1)$, there is virtue to keeping the element Hessians unassembled. Keeping them unassembled also has the benefits that we mentioned for the gradient: only certain element Hessians need be recomputed after a sparse update of x, and their evaluation can be carried out in parallel.

Finally, (3.7b) suggests that an approximation $B_k = B_k^\top \approx \nabla^2 f(x)$ can be obtained in the form

$$B_k = \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i, \quad \text{where} \quad \widehat{B}_{i,k} = \widehat{B}_{i,k}^\top \approx \nabla^2 \widehat{f}_i(\widehat{x}_i). \tag{3.9}$$

Griewank and Toint [75] saw in (3.9) an opportunity to develop partitioned quasi-Newton updates and partitioned quasi-Newton methods, a topic that we develop further in Section 3.2.

The appliance of U_i and $U_i^\top$ in (3.7) scatters the contributions of the small element gradients $\nabla \widehat{f}_i(\widehat{x}_i)$ and element Hessians $\nabla^2 \widehat{f}_i(\widehat{x}_i)$ to the components of $\nabla f(x)$ and $\nabla^2 f(x)$ they contribute during assembly. The smaller n_i^E is compared to n, the sparser is the contribution of $\nabla \widehat{f}_i(x)$ and $\nabla^2 \widehat{f}_i(x)$ to $\nabla f(x)$ and $\nabla^2 f(x)$. However, it is not because f is partially-separable that $\nabla^2 f(x)$ is sparse. For instance, we may have the decomposition

$$f(x_1, x_2, x_3) = \widehat{f}_1(x_1, x_2) + \widehat{f}_2(x_1, x_3) + \widehat{f}_3(x_2, x_3),$$

which has $U_1^\top \nabla^2 \widehat{f}_1 U_1$, $U_2^\top \nabla^2 \widehat{f}_2 U_2$ and $U_3^\top \nabla^2 \widehat{f}_3 U_3$ all sparse, but $\nabla^2 f(x)$ dense:

$$\nabla^2 f(x) = \left[\begin{array}{ccc} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array}\right], \tag{3.10}$$

where yellow, red and blue are respectively the contributions of $\widehat{f_1}$, $\widehat{f_2}$ and $\widehat{f_3}$. However, as we elaborate in Theorem 3.1.1, the converse holds: if $\nabla^2 f(x)$ is sparse, then f is partially-separable.

3.1.3 Group partial separability

As the first version of the LANCELOT solver Conn et al. [33] and the SIF modeling language [34] were developed, further examined in Section 3.6.2 and Section 3.6.3, Conn et al. [33] generalize (3.1) and formulate the concept of a group-partially-separable function. Those functions are of the form

$$f(x) = \sum_{j=1}^{m} g_j(l_j(x) + \sum_{i=1}^{N_j} f_i(x)), \quad g \in \mathcal{C}^2, \tag{3.11}$$

where $g_j : \mathbb{R} \to \mathbb{R}$, is the j-th *group function*, l_j is the linear term of the j-th group, and f_i, $i = 1, \ldots, N_j$ are the nonlinear functions of the j-th group. The variables appearing in the j-th group can be accessed by U_j, while the nonlinear terms can use an internal representation based upon U_j to select their variables. As an extension to the SIF, Gould et al. [69] further refine (3.11) to expose quadratic terms explicitly, i.e.,

$$f(x) = \sum_{j=1}^{m} g_j(l_j(x) + \sum_{i=1}^{N_j} f_i(x) + c) + \tfrac{1}{2} x^\top H x, \tag{3.12}$$

where $H = H^\top$ and c is a constant.

A simple example about group partial separability is the matrix completion problem:

$$\min_{X \in \mathbb{R}^{m \times r}, Y \in \mathbb{R}^{r \times n}} f(X, Y) = \|(A - XY)_\Omega\|_F^2, \tag{3.13}$$

where $A \in \mathbb{R}^{m \times n}$ is the matrix with a known subset of its entries Ω that XY seeks to approximate. The projection operator $(.)_\Omega$ is defined as $[(Z)_\Omega]_{i,j} = Z_{i,j}$ if $(i, j) \in \Omega$ or 0 otherwise. Matrix completion problem can be reformulated as:

$$f(X, Y) = \sum_{(h,k) \in \Omega} g_{h,k}(X, Y) = \sum_{(h,k) \in \Omega} \left(A_{h,k} - \sum_{l=1}^{r} X_{h,l} Y_{l,k} \right)^2.$$

f sums $|\Omega|$ identical group functions $g_{h,k}(y) = y^2$. All group dimensions are $2r$, containing a constant term $A_{h,k}$ and the nonlinear terms $\sum_{l=1}^{r} X_{h,l} Y_{l,k}$. Suppose that $r = 1$ in (3.13) making X and Y two vectors and $g_{h,k}(x, y) = \left(A_{h,k} - xy \right)^2 : \mathbb{R}^2 \to \mathbb{R}$ applied onto X_h and

Y_k. The addition of the regularization terms $\|X\|_F^2 + \|Y\|_F^2$ to (3.11) furnishes the quadratic terms of (3.12).

Group partial separability can enhance automatic differentiation for computing ∇f. The Figure 3.1 illustrates the reverse mode for one group of the group partially-separable function $g_{h,k}$ presented (3.13) considering $r = 2$:

$$f(X,Y) = \sum_{(h,k)\in\Omega} \underbrace{\left(A_{h,k} - \sum_{l=1}^{2} X_{h,l}Y_{l,k}\right)^2}_{g_{h,k}}, \quad X \in \mathbb{R}^{m\times 2}, Y \in \mathbb{R}^{2\times n}. \tag{3.14}$$

The Figure 3.1 is intentionally similar with Figure 2.1, which details automatic differentiation. Similarly, each node is identified by a number and contains two values red (left) and blue (bottom right) which respectively inform about the node values resulting from a forward evaluation and the adjoint values. In this example, x_1, x_2, x_3 and x_4 are replaced with $Y_{2,j} = 1.5, X_{i,2} = 2, Y_{1,j} = 2.5$ and $X_{i,1} = 3$. Unlike the Figure 2.1, the group partial separability allows computing a group derivative $\nabla g_{h,k}(X,Y)$ without knowing the value of the complete expression tree, i.e. $f(X,Y)$, as $\frac{\partial f}{\partial g_{h,k}} = \frac{\partial f}{\partial ⑪}.\frac{\partial ⑪}{\partial ⑩} = 1$. Furthermore, $\frac{\partial ⑩}{\partial ⑨} = 2.⑨$ enables the computation of $\frac{\partial ⑩}{\partial ①}$, $1 \leq l \leq 8$ right after the evaluation of ⑨, i.e. $\frac{\partial ⑩}{\partial ⑧} = \frac{\partial ⑩}{\partial ⑨}.\frac{\partial ⑨}{\partial ⑧} = 1.2.⑨ = 12$. Lastly, the dotted curves indicate that a node could have edges related to other groups which do not appear to keep the example minimal. The accumulation of all the other groups' contributions computes ∇f and is equivalent to summing all $\nabla g_{h,k}$. In the particular case of (3.14), every group is identical. The tape described in Figure 3.1 is enough to compute all $\nabla g_{h,k}$, which avoids the storage of $|\Omega| - 1$ tapes. Practically, the values of the leaves $Y_{2,j}, X_{i,2}, Y_{1,j}, X_{i,1}$ must be modified accordingly and the indices of X and Y are likely to become irrelevant.

3.1.4 Other definitions of partial separability

There exist other concepts close to partial separability that Omidvar et al. [117] and Liang [100] exploit. Their definitions differ with the definition given in Section 3.1.1. Omidvar et al. [117] define a variable x_i as separable if it satisfies

$$\underset{x\in\mathbb{R}^n}{\operatorname{argmin}} f(x) = \left(\underset{x_i\in\mathbb{R}}{\operatorname{argmin}} f(\dots, x_i, \dots), \underset{\{x_j\}_{j\in\mathcal{J}}\in\mathbb{R}^{n-1}}{\operatorname{argmin}} f(\{x_j\}_{j\in\mathcal{J}}, \mathbf{x}_i)\right), \tag{3.15}$$

where $\mathcal{J} \coloneqq \{\{1,\dots,n\}\backslash\{i\}\}$ and $\mathbf{x}_i$ the i-th fixed variable. Note that a "totally separable" function $f(x) = \sum_{i=1}^n f_i(x_i)$ is a particular case of (3.15). More generally, Omidvar et al. [117]

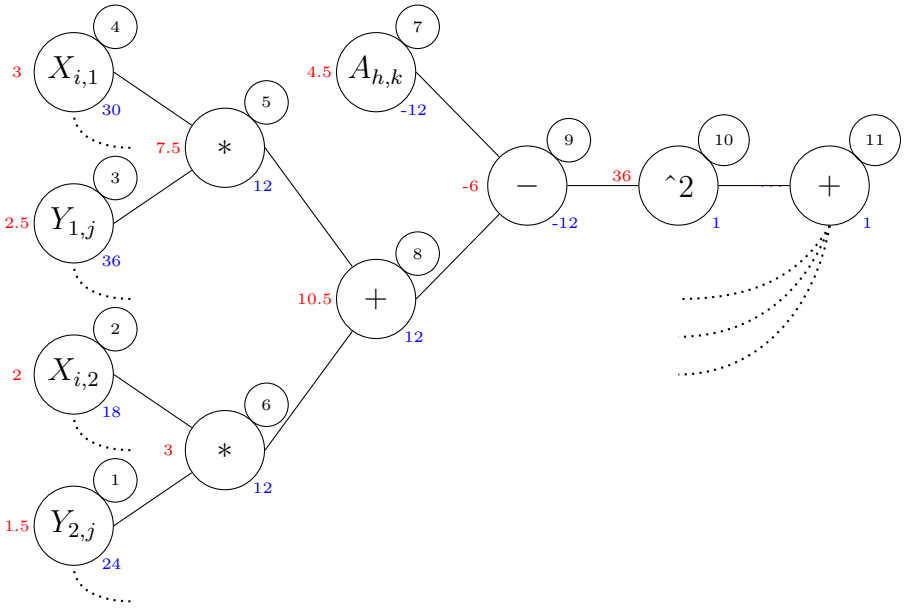


Figure 3.1 Automatic differentiation for one group of a group partially-separable function

define a partially-separable function as

$$\underset{x\in\mathbb{R}^n}{\operatorname{argmin}} f(x) = \left(\underset{x_{[1]}\in\mathbb{R}^{n_1}}{\operatorname{argmin}} f(x_{[1]}, \dots), \dots, \underset{x_{[m]}\in\mathbb{R}^{n_m}}{\operatorname{argmin}} f(\dots, x_{[m]}) \right), \tag{3.16}$$

where $x \in \mathbb{R}^n$ can be divided into m subcomponents $x = (x_{[1]}, \dots, x_{[m]})$, where $\sum_{i=1}^{m} n_i = n$. This definition strongly differs with (3.1), as f is not a sum and variables interact nonlinearly with each others which prevent (3.16) to be satisfied. Nevertheless, those definitions gave birth to several evolutionary algorithms [66, 91, 117, 154, 155] to solve (3.16).

A special case of such partial separability is partially additively separable $f(x) = \sum_{i=1}^{N} f_i(x_{[i]})$ where $x_{[i]}$ is a mutually exclusive decision vector. Partial additive separability is equivalent to $f(x) = \sum_{i=1}^{N} f_i(U_i x)$ where each element function uses an independent subset of variables, i.e., $U_i^\top U_j = 0,\ i \neq j,\ i, j = 1, 2, \dots, N$ and element Hessians are not overlapping. Such a problem can be viewed as separated subproblems, which enables straightforward parallel optimization methods.

In addition, Liang [100] gives a different definition of a separable function:

$$f(x) = f_1(x_1) f_2(x_2) \dots f_n(x_n),$$

which can be generalized into a partially-separable function as:

$$f(x) = \sum_{l=1}^{L} f_{l,1}(x_1) f_{l,2}(x_2) \dots f_{l,n}(x_n). \tag{3.17}$$

Such a structure appears in medical imaging, for which Haldar and Liang [80] provides a minimization method. Nonetheless, (3.17) partial separability also differs with (3.1) as each term of the sum contains every decision variables.

Finally, closer to what (3.1) considers, Bouzarkouna et al. [18] extend the definition of (3.1) by replacing U_i with a nonlinear operator $\Phi^i : \mathbb{R}^n \to \mathbb{R}^{n_i}$:

$$f(x) = \sum f_i(\Phi^i(x)).$$

To solve such problem, Bouzarkouna et al. propose a population based stochastic search rank-based algorithm, described in Section 3.8.

3.2 Partitioned quasi-Newton methods

The idea of partitioned quasi-Newton updates is to aggregate $\widehat{B}_{i,k} \approx \nabla^2 f_i(\widehat{x}_{i,k})$ to approximate $B_k \approx \nabla^2 f(x_k)$. If every element Hessian approximation update satisfies its element secant equation:

$$\widehat{B}_{i,k+1}\widehat{s}_{i,k} = \widehat{B}_{i,k+1}U_i s_k = \widehat{y}_{i,k} = \nabla \widehat{f}_i(\widehat{x}_{i,k} + \widehat{s}_{i,k}) - \nabla \widehat{f}_i(\widehat{x}_{i,k}), \tag{3.18}$$

then B_{k+1} also satisfies the global secant equation

$$B_{k+1}s_k = \sum_{i=1}^{N} U_i^\top \left(\widehat{B}_{i,k+1}U_i s_k\right) = \sum_{i=1}^{N} U_i^\top \left(\nabla \widehat{f}_i(\widehat{x}_{i,k+1}) - \nabla \widehat{f}_i(\widehat{x}_{i,k})\right) = \nabla f(x_{k+1}) - \nabla f(x_k) = y_k. \tag{3.19}$$

However, contrary to unstructured quasi-Newton methods, $\widehat{s}_{i,k}$ does not come from the minimization of $\widehat{f}_{i,k}$. Therefore, unlike a Wolfe (L)BFGS line search, the quasi-Newton's formula numerical safeguards (2.14) adapted for element Hessian approximation updates, parametrized by $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$, may not automatically hold.

The partitioned-BFGS update, commonly denoted as PBFGS [75, 76], supposes that every element Hessian is updated with the BFGS formula $\widehat{B}_{i,k+1} = \widehat{B}^{\text{BFGS}}_{i,k+1}$ from (2.13) where s_k and y_k are replaced with $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$. If every element curvature condition $\widehat{s}_{i,k}^\top \widehat{y}_{i,k} > 0$ holds for

any k and $\widehat{B}_{i,0} \succ 0$, then $\widehat{B}_{i,k} \succ 0$ and $B_k \succ 0$

$$v^\top B_k v = v^\top \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i v = \sum_{i=1}^{N} \underbrace{\widehat{v}_{i,k}^\top \widehat{B}_{i,k} \widehat{v}_{i,k}}_{>0} > 0, \ \forall v \in \mathbb{R}^n.$$

Conversely, if one or several element curvature conditions fail, the update of the corresponding $\widehat{B}_{i,k}$ would make them lose their positive definiteness and so forth making possibly $B_k \not\succ 0$. Fortunately, several strategies exist to preserve the positive definiteness of $\widehat{B}_{i,k+1}$, and therefore that of B_{k+1}. One may choose to: skip the update $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$, reinitialize $\widehat{B}_{i,k+1} = \widehat{B}_{i,\text{init}} \succ 0$, or apply a damped update [128] keeping $\widehat{B}_{i,k+1} \succ 0$. Nevertheless, the resulting B_{k+1} is unlikely to verify the secant condition.

Griewank and Toint [79] proposed the first known routine implementing PBFGS, named PSPMIN, described later in Section 3.6. The success of PBFGS [79, 101] led the way to the development of other partitioned quasi-Newton methods, for example PSR1 $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{SR1}}, \ \forall i$ [75, 76] and several others described later in the section.

To ensure the convergence, Griewank and Toint [77] describe a partitioned minimization method based on the assumption that every element function is convex, making f equally convex. Therefore, every $\widehat{B}_{i,k}$ is positive definite. The update formula is either BFGS (2.13) or DFP (Davidon [41] Fletcher and Powell [60]):

$$B_{k+1}^{\text{DFP}} = B_k - \frac{B_k s y_k^\top + y_k s^\top B_k}{y_k^\top s} + \frac{y_k y_k^\top}{y_k^\top s}(1 + \frac{s^\top B_k s}{y_k^\top s}), \tag{3.20}$$

another popular positive definite quasi-Newton update at that time.

In this context, Griewank and Toint [75] theoretically study the behaviour of both BFGS and DFP updates to determine which one is best suited for partitioned updates, considering that $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$ do not result from $\widehat{f}_i$ minimization. To do so they study the unstructured approximation $B_k \approx \nabla^2 f(x_k) \in \mathbb{R}^{n \times n}$, a sparse symmetric positive semi-definite matrix with an initial nullspace that may not correspond to the nullspace of $\nabla^2 f$. The main result of Griewank and Toint [75] is:

Proposition 3.2.1. *Let $\nabla^2 f$ be continuous and positive semi-definite at all x in a convex compact subset D of $\mathbb{R}^n$. Given any symmetric positive semi-definite $n \times n$ matrix B_k, let the set S consists of all $(s, y) \in \mathbb{R}^n \times \mathbb{R}^n$ such that, for some $x \in D$ with $x + s \in D$,*

$$(\nabla f(x + s) - \nabla f(x))^\top s > 0.$$

Then B_{k+1}^{BFGS} *defined by* (2.13) *is uniformly bounded over all* (s, y) *in* S *with* $s^\top B_k s \neq 0$. *For* B_{k+1}^{DFP} *defined by* (3.20) *to have the same property, it is necessary that for all* $x \in D$,

$$\mathrm{Null}(\nabla^2 f(x)) \subseteq \mathrm{Null}(B_k) \quad \text{or} \quad \nabla^2 f(x) = 0,$$

Furthermore, if, for some x *in* D,

$$\mathrm{Null}(B_k) \not\subseteq \mathrm{Null}(\nabla^2 f(x))$$

then there is $(s, y) \in S$ *such that*

$$rank(B_{k+1}^{\mathrm{DFP}}) = rank(B_k) + 1.$$

In view of the Proposition 3.2.1, B_{k+1}^{BFGS} remains well-defined and is uniformly bounded, despite $y_k^\top s_k$ or $s_k^\top B_k s_k$ being close to zero. In the cases $y_k^\top s_k \approx 0$ or $s_k^\top B_k s_k \approx 0$, it is still possible to use a restart scheme, a damped vector y_k or skip the update to keep the positive definiteness. On the other hand, DFP has a property increasing the nullspace of B_{k+1}, by adding $span(y_k)$ to $range(B_{k+1}^{\mathrm{DFP}})$ for matching the nullspace of $\nabla^2 f$. However, beside the numerical issues from starting with a nullspace of B_k too small, the process increasing the spanned space by B_{k+1}^{DFP} may result with $\|B_{k+1}^{\mathrm{DFP}}\|$ unbounded.

The analysis of Griewank and Toint unanimously favours the use of BFGS for B_k rather than DFP. The authors confirmed the Proposition 3.2.1 numerically by showing that PBFGS performs half as many iterations as PDFP and three times fewer gradient evaluations.

Later on, in order to prove convergence for partitioned quasi-Newton methods, [74, 77, 149] assume that all element functions f_i are convex, so is f. Griewank and Toint [77] derive the first local convergence theory for partitioned quasi-Newton methods, which, interestingly, does not require the limiting Hessian to be nonsingular, but Hölder continuous. Griewank and Toint employ an inexact partitioned variable metric (line search) method to establish Q-superlinear local convergence:

$$\lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^q} = \mu \quad \text{with} \quad q > 1,\ \mu \in (0, \infty) \text{ or } q = 1, \mu = 0, \tag{3.21}$$

as long as the element quasi-Newton updates are in the convex Broyden class. In order to reach the Q-superlinear convergence, the linear system iteratively determining the search direction must be solved exactly asymptotically. During the proof process, Griewank and Toint happen to bound $\{\|\widehat{B}_{i,k}\|\}_k$, $\{\|B_k\|\}_k$ and $\{\|B_k^{-1}\|\}_k$ using suitable numerical safeguards and positive semidefinite initialization of $\widehat{B}_{i,0} \succ 0$. To ensure the boundness of $\{\|\widehat{B}_{i,k}\|\}_k$,

$\{\|B_k\|\}_k$ and $\{\|B_k^{-1}\|\}_k$, the partitioned updates proposed skip the element quasi-Newton updates when they do not meet those numerical safeguards.

Later, Toint [149] extends the local convergence of Griewank and Toint [77] to establish a global convergence. Similarly to Griewank and Toint [77], Toint uses a line search where the search direction is determined with the conjugate gradient. The Hessian is also assumed to be Hölder continuous while every element Hessian approximation is updated with the BFGS formula. Contrary to [77], Toint [149]'s proof may prevent the update of every element at each iteration. Nonetheless, the remaining element Hessian approximation updates are enough to prove global convergence, as well as Q-superlinear local convergence, considering the same assumptions as [77].

Griewank [74] proposes another global convergence proof using an inexact PBFGS line search. Unlike Toint [149], BFGS is applied to every element function but all $\nabla^2 f_i$ are required to be Lipschitz continuous. $\widehat{B}_i$ is damped, i.e. $\widehat{y}_i$ is slightly modified, when its element curvature condition fails. Griewank establishes global convergence and local R-superlinear convergence:

$$\|x_k - x^*\| \leq \epsilon_k,$$

considering that ϵ_k converges Q-linearly, i.e., satisfies (3.21) with $q = 1$ and $\mu = (0, \infty)$. If every $\nabla \widehat{f}_i$ happens to be strictly differentiable [74, eq 2.5-2.6], which includes the case where $f \in C^1$, then the local convergence becomes almost Q-superlinear. Furthermore, when all $\nabla^2 \widehat{f}_i$ pass the *Dini's test* [74, eq 2.8-9]—originating from harmonic analysis "This relation can be used to bound the effects of the nonlinearities in g on the updating process" [90]—, which automatically holds when all $\nabla^2 \widehat{f}_i$ are Hölder continuous, then the convergence rate is Q-superlinear.

In a similar fashion to PBFGS, Cao and Yao [28] define a partitioned quasi-Newton update exploiting the Powell-symmetric-Broyden (PSB) formula [126]:

$$B_{k+1}^{\text{PSB}} = B_k + \frac{z_k s_k^\top + s_k z_k^\top}{\|s_k\|^2} - \frac{z_k^\top s_k . s_k s_k^\top}{\|s_k\|^4}, \quad z_k \coloneqq y_k - B_k s_k,$$

which may result with B_{k+1}^{PSB} not positive definite. Consequently, s^{PSB} issued from $B_k s^{\text{PSB}} = -\nabla f(x_k)$ may not be a descent direction. Therefore, Cao and Yao rely on a projected quasi-Newton line search method initiated by An et al. [2] to ensure that s lies in a descent direction. It linearly combines the steepest descent direction $-\nabla f(x_k)$ and s^{PSB}:

$$s = -\nabla f(x_k) + \lambda_k \left(I - \frac{\nabla f(x_k) \nabla f(x_k)^\top}{\|\nabla f(x_k)\|^2} \right) s^{\text{PSB}}, \quad B_k s^{\text{PSB}} = -\nabla f(x_k),$$

where

$$\lambda_k = \begin{cases} 0 & \text{if } \nabla f(x_k)^\top s_k^{\text{PSB}} = 0 \\ \min\left\{\lambda k^{\frac{1}{4}}, \dfrac{\|\nabla f(x_k)\|^2}{|\nabla f(x_k)^\top s_k^{\text{PSB}}|}\right\} & \text{otherwise,} \end{cases}$$

considering $\lambda > 0$ is practically set to 10^3 in the numerical experiments. Once s is found, an Armijo line search determines α_k to set $x_{k+1} = x_k + \alpha_k s$ and B_k is updated.

In the partitioned case, Cao and Yao [28] update every $\widehat{B}_{i,k}$ with the PSB formula:

$$\widehat{B}_{i,k+1}^{\text{PSB}} = \widehat{B}_{i,k} - \frac{\widehat{z}_{i,k}\widehat{s}_{i,k}^\top + \widehat{s}_{i,k}\widehat{z}_{i,k}^\top}{\|\widehat{s}_{i,k}\|^2} - \frac{\widehat{z}_{i,k}^\top \widehat{s}_{i,k}}{\|\widehat{s}_{i,k}\|^4}\widehat{s}_{i,k}\widehat{s}_{i,k}^\top, \ \widehat{z}_{i,k} := \widehat{y}_{i,k} - \widehat{B}_{i,k}\widehat{s}_{i,k},$$

where

$$\widehat{s}_{i,k} := \lambda_k \alpha_k U_i s^{\text{PPSB}} \quad \text{if} \quad \widehat{s}_{i,k} \neq 0,$$

considering

$$B_k s^{\text{PPSB}} = -\nabla f(x_k), \quad B_k = \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k}^{\text{PSB}} U_i.$$

Cao and Yao prove a global and a superlinear convergence under the assumptions that f is uniformly convex, twice continuously differentiable and that every $\nabla^2 \widehat{f}_i$ is Lipschitz continuous.

The PPSB method is compared to a PBFGS and a LBFGS line searches [75, 101] considering a set of 30 partially-separable problems described by [106] with a size's range from 10^3 to 10^6. Cao and Yao report that PPSB requires generally slightly less iterations, function and gradient evaluations than PBFGS and LBFGS. By taking fewer iterates and by having similar subsidiary cost than PBFGS, PPSB takes less CPU time than PBFGS. However, when only the CPU time matters, LBFGS performance is comparable to that of PPSB.

Griewank and Toint [75] stated that a straightforward partitioned PSB method (PPSB) may not bring benefit compared to a sparse quasi-Newton update. Those numerical results contradict this claim. Strangely enough, PBFGS performances are close to LBFGS performances, which is not replicated in other quasi-Newton comparisons, e.g. Liu and Nocedal [101], where PBFGS outperforms LBFGS.

3.2.1 Partitioned update for nonsmooth element functions

To deal with the nonsmoothness of $\widehat{f}_i$, Lukšan et al. [104] implement PSEN, a bundle method which relies on a partitioned quasi-Newton line search. Contrary to the partitioned method proofs presented earlier in this section, each element function needs only to be locally Lipschitz

continuous for the method to globally converge. Hence, instead of using the gradient, a (Clarke) subgradient is used which is a convex combination of the nearest subgradients computed [104, eq.7, step 7 in Alg. 1].

As a line search, the descent direction d_k is determined by solving the partitioned linear system $B_k d_k = -\nabla f(x_k)$, where B_k is partitioned as (3.9) and remains positive definite. To do so, Lukšan et al. compute the Cholesky factorization of B_k. When d_k happens to not be a descent direction, the element-Hessian approximations $\widehat{B}_{i,k}$ are updated with SR1 enforced with a safeguard ensuring that B_k remains positive definite, similarly to the curvature condition for BFGS. If the numerical safeguards fail, then the update is skipped, i.e. $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$.

Numerical results show a comparison of the partitioned line search with an unstructured bundle method and an unstructured proximal bundle method, on problems of size $50, 200$ and 1000. For most partially-separable problems tested, the partitioned line search converges to a stationary point within the fewest function/gradient evaluations. Moreover, it is the only method out of the three tested applicable for problems of size 1000. The implementation of the algorithm is available in the Fortran library LSA Lukšan et al. [105].

3.2.2 Partitioned Quasi-Newton vs. Limited-Memory Quasi-Newton

Toint [148] and Lukšan et al. [106] historically collect over two hundred partially-separable problems that may be useful to compare limited-memory and partitioned quasi-Newton implementations. Nowadays, the CUTEst collection [71] contains hundreds more. To the best of our knowledge, there is no recent exhaustive comparison between the limited-memory quasi-Newton and partitioned quasi-Newton methods. The comparisons come from Griewank and Toint [75], Griewank and Toint [79], Liu and Nocedal [101], Conn et al. [35] and Lukšan et al. [105]. Drawing general conclusions from those studies is difficult, mainly because the methods differ in nature—some are line search methods, while some are trust-region methods—as do the implementations and programming languages. However, they suggest that if $n_i \ll n$, N is moderate, and the overlap between element Hessians does not drag down memory requirements too much, then the partitioned methods typically outperform their full-space and limited-memory counterparts. The comparison by Liu and Nocedal [101] suggests that PBFGS outperforms LBFGS in cases such as those just described, but LBFGS remains significantly more general. The Section 4.3 illustrates this trend with the Figure 4.4 where all partitioned quasi-Newton methods outperform a LBFGS line search.

3.2.3 Partitioned update without element gradients

Up to now, the partitioned quasi-Newton methods presented have always had access to the element function gradients $\nabla \widehat{f}_i$, and thus to $\widehat{y}_i$. Malmedy and Toint [107] interest themselves to the case where the partially-separable structure is known –allowing a partitioned quasi-Newton approximation to exist– but only ∇f is accessible during the minimization process. Ultimately, the method must solve infinite-dimensional problems arising from the discretization of partial differential equations. Therefore, the authors used a recursive multilevel trust-region [72], whose recursively returns a finer solution $x \in \mathbb{R}^n$ of the discretized problem $f : \mathbb{R}^n \to \mathbb{R}$ for which n is getting increasingly bigger. Note that to match Malmedy and Toint [107] notation, each element function $f_i : \mathbb{R}^n \to \mathbb{R}$ is full size, but their corresponding vectors or matrix are sparse, e.g. $\nabla f_i \in \mathbb{R}^n$ or $B_{i,k} \in \mathbb{R}^{n \times n}$.

The approximation of an element Hessian is guided by several assumptions such as: $B_{i,k} = U_i^\top \widehat{B}_{i,k} U_i$, $B_{i,k} = B_{i,k}^\top$ and $B_{i,k}$ must satisfy the element secant equation utilizing an extrapolated $y_{i,k}$. Those different conditions formulate an optimization problem:

$$
\begin{aligned}
\min \quad & \tfrac{1}{2} \textstyle\sum_{i=1}^{N} \omega_i \|B_{i,k+1} - B_{i,k}\|_F^2 \\
s.t. \quad & B_{i,k+1} - B_{i,k} = (B_{i,k+1} - B_{i,k})^\top \; \forall \; i = 1, .., N \\
& B_{i,k+1} s_k = y_{i,k} \\
& I_i y_{i,k} = y_{i,k} \\
& J_i \bullet B_{i,k+1} = B_{i,k+1} \\
& \textstyle\sum_{i=1}^{N} y_{i,k} = y_k,
\end{aligned}
\tag{3.22}
$$

where:

- $\bullet$ is the Hadamard product;
- $J_i = {U_i^E}^\top \mathbb{1}_{n_i} \mathbb{1}_{n_i}^\top U_i^E$ is the identity of the Hadamard product, $\mathbb{1}_{n_i} = (1, 1, ..., 1)^\top \in \mathbb{R}^{n_i}$;
- $I_i = {U_i^E}^\top U_i^E \in \mathbb{R}^{n \times n}$ is a sparse diagonal matrix, whose diagonal components are set to one only for the variables parametrizing the i-th element function.

Regardless of the availability of $y_{i,k} = U_i^\top \widehat{y}_{i,k}$, the extrapolated $y_{i,k}$ satisfies $\sum_{i=1}^{N} y_{i,k} = y_k$.

The first order optimal condition of the Lagrangian problem from (3.22) leads to the PPSB update, a 3-steps algorithm [107, Algorithm 2.3]. First, it decomposes iteratively the step s into $\{s_i\}_{i=1}^N$ and form $S = \sum_{i=1}^{N} \omega_i^{-1} (\|s_i\| I_i + s_i s_i^\top)$, $\omega_i > 0$. Then, it solves the sparse linear system $Sc = y - Hs$, and decomposes c into $\{c_i\}_{i=1}^N$. Finally, it updates $B_{k+1}^{\text{PSPSB}} = \sum_{i=1}^{N} B_{i,k+1}^{\text{PSB2}} = \sum_{i=1}^{N} \left(B_{i,k} + \omega_i^{-1} (c_i s_i^\top + s_i c_i^\top) \right)$. Here, the denotation PSB differs from the one used in [28], and is therefore denoted by PSB2 while the partitioned approximation is PSPSB.

The numerical results consider fifteen problems in infinite-dimensional spaces and involve differential operators. All the following methods are based on the recursive multilevel trust-region method, and only the approximation of $\nabla^2 f$ changes. PSPSB is compared to:

- LTS (Lower Triangular Substitution): finite difference computation of $\nabla^2 f$ by exploiting only the sparsity pattern of the problem and gradient evaluations [40];
- LTS-O, a LTS-variant using optimal column groups [68];
- Sparse PSB method (S-PSB), [107, Algorithm 2.2], [146], which is closely related to PSPSB. The main difference between the two method is that PSPSB weights heavier the overlapping elements of the Hessian than S-PSB.

A first performance profile criterion counts the objective function evaluations to which is added five times the gradient evaluations before convergence. For this criteria, LTS-O and PSPSB are the two methods whose runs require the lesser function and gradient evaluations. LTS-O has a slight advantage as it solves all problems, while PSPSB fails for two problems. A second performance profile compares the CPU time and reports PSPSB to be slightly faster than S-PSB, but significantly slower than both LTS methods. These results are partly explained by the fact that LTS methods update the Hessian approximation periodically, rather than iteratively as PSPSB and S-PSB do. In addition, LTS methods only solve a triangular linear system while both PSB methods solve a sparse linear system at every update. Overall, the results uniformly favour LTS-O to solve the infinite-dimensional problems.

3.2.4 Partitioned quasi-Newton trust-region

To conclude this section, the Algorithm 3.2.1 presents a variant of the Algorithm 2.1.2 using a partitioned quasi-Newton approximation of $\nabla^2 f(x_k)$, similarly to PSPMIN [79] and LANCELOT [33]. The main difference is the (partitioned) update of the matrix B_k. Instead of using s and y, the partitioned quasi-Newton update is based on $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$. An implementation closely aligned with Algorithm 3.2.1 will be used in Section 4.3 for comparing numerically partitioned quasi-Newton and unstructured quasi-Newton trust-region methods. The Algorithm 3.2.1 has three variants, changing the way an element Hessian approximation is updated (Line 8):

- PBFGS, where $\widehat{B}_{i,k+1} = \widehat{B}^{\text{BFGS}}_{i,k+1}$ (2.13);
- PSR1, where $\widehat{B}_{i,k+1} = \widehat{B}^{\text{SR1}}_{i,k+1}$ (2.15);

- PSE, where $\widehat{B}_{i,k+1} = \widehat{B}^{\text{BFGS}}_{i,k+1}$ if $\widehat{s}_{i,k}^{\top}\widehat{y}_{i,k} > \epsilon$ or $\widehat{B}_{i,k+1} = \widehat{B}^{\text{SR1}}_{i,k+1}$ otherwise. By having more chances to update $\widehat{B}_{i,k}$, B^{PSE}_{k+1} is more likely to satisfy the secant equation.

All methods perform numerical safeguards when updating every element Hessian approximation. If the safeguards fail, then the update is skipped $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$.

Algorithm 3.2.1 Partitioned Quasi-Newton Trust-Region Algorithm

Choose $x_0 \in \mathbb{R}^n$, $\Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$, and $0 < \epsilon_1, \epsilon_2$.
Choose for every element $\widehat{B}_{0,i} = \widehat{B}_{0,i}^{\top} \approx \nabla^2 \widehat{f}_i(x_0)$. *initial approximation*
for $k = 0, 1, \ldots$ **do**
    Compute an approximate solution s_k of

$$\min_s m_k(s) \quad \text{s.t.} \|s\| \leq \Delta_k, \quad m_k(s) := f(x_k) + \nabla f(x_k)^{\top} s + \tfrac{1}{2} s^{\top} \left(\sum_{i=1}^{N} U_i^{\top} \widehat{B}_{i,k} U_i \right) s,$$

    bringing a sufficient decrease (2.8).
    Compute the ratio $\rho_k := \frac{f(x_k) - f(x_k + s)}{m_k(0) - m_k(s)}$.
    if $\rho_k \geq \eta_1$ **then** *successful step*
        set $x_{k+1} = x_k + s_k$
        update every $\widehat{B}_{i,k}$ with either (2.13) or (2.15) by replacing s and y respectively by $\widehat{s}_{i,k} := \widehat{x}_{i,k+1} - \widehat{x}_{i,k}$ and $\widehat{y}_{k_i} := \nabla \widehat{f}_i(\widehat{x}_{i,k+1}) - \nabla \widehat{f}_i(\widehat{x}_{i,k})$. If the safeguards (2.16) or (2.14) considering $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$ instead of s and y fail, then $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$.
    else *unsuccessful step*
        set $x_{k+1} = x_k$ and every $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$.
    end if
    Update the trust-region radius according to

$$\Delta_{k+1} \in \begin{cases} [\gamma_3 \Delta_k, \gamma_4 \Delta_k] & \text{if } \rho_k \geq \eta_2, \\ [\gamma_2 \Delta_k, \Delta_k] & \text{if } \eta_1 \leq \rho_k < \eta_2, \\ [\gamma_1 \Delta_k, \gamma_2 \Delta_k) & \text{if } \rho_k < \eta_1. \end{cases} \tag{3.23}$$

end for

3.3 Optimization methods enhancing partial separability exploitation

3.3.1 Structured trust-region methods

Conn et al. [37] describe a structured trust-region method for minimizing a partially-separable problem. The quadratic trust-region subproblem can be seen as the minimization of a partitioned quadratic model:

$$m_k(s) \coloneqq \sum_{i=1}^{N} \widehat{m}_{i,k}(\widehat{s}_i), \tag{3.24}$$

aggregating element quadratic models:

$$\widehat{m}_{i,k}(\widehat{s}_i) \coloneqq \widehat{f}_i(\widehat{x}_{i,k}) + \nabla \widehat{f}_i(\widehat{x}_{i,k})^\top \widehat{s}_i + \tfrac{1}{2}\widehat{s}_i^\top \widehat{B}_{i,k}\widehat{s}_i, \quad 1 \leq i \leq N.$$

Hence, Conn et al. [37] design a partitioned trust-region, where each element quadratic model is associated with an element trust-region:

$$\mathcal{B}_{i,k} := \{x \in \mathbb{R}^n \mid \|U_i(x - x_k)\| \leq \Delta_{i,k}\}, \tag{3.25}$$

where $\Delta_{i,k} > 0$ is the element trust-region radius. The overall trust-region

$$\mathcal{B}_k = \bigcap_{i=1}^{N} \mathcal{B}_{i,k}$$

may be asymmetric and may allow larger steps in some directions than in others.

To find a step s_k in $\mathcal{B}_k$, a first step s_k^1 is computed in the simpler, unstructured, trust-region

$$\mathcal{B}_k^{\min} = \mathcal{B}_k \cap \{x \in \mathbb{R}^n \mid \|U_i(x - x_k)\| \leq \Delta_k^{\min}\} \subset \mathcal{B}_k,$$

where

$$\Delta_k^{\min} = \min\{\Delta_{i,k} \mid i = 1, \ldots, N\} > 0.$$

Subsequently, the user may employ any procedure to extend $s_k^1 \in \mathcal{B}_k^{\min}$ to a step $s_k^2 \in \mathcal{B}_k$ as long as the latter satisfies a sufficient decrease condition.

Every iteration, each element radius $\Delta_{i,k}$ is updated individually based on the step acceptance and the decrease of its corresponding element function $\widehat{f}_i$. Such procedure may push radii of highly nonlinear element functions to remain small compared to other element radii. To compensate this side effect, a hybrid strategy includes an overall radius $\Delta_k > 0$ bounding element radii such as

$$\Delta_{i,k}^h := \max\{\Delta_k, \Delta_{i,k}\}.$$

Consequently, the element and overall trust-regions are redefined as

$$\mathcal{B}_{i,k} := \{x \in \mathbb{R}^n \mid \|U_i(x - x_k)\| \leq \Delta_{i,k}^h\}$$

and

$$\mathcal{B}_k^{\min} = \{x \in \mathbb{R}^n \mid \|U_i(x - x_k)\| \leq \Delta_k\}.$$

Conn et al. [37] propose a sounded framework to solve the trust-region subproblem by

aggregating the solutions of element trust-region subproblem results. This can be seen as an attempt to replace the truncated conjugate gradient by a method incorporating partial separability in its core. Nonetheless, the question of how aggregate the element trust-region solutions remains open. At the time of Conn et al. [37], the implementation of a structured trust-region was not competitive with an unstructured trust-region, making [37] lacks numerical results.

3.3.2 Partial separability exploitation to solve quadratic subproblem

3.3.2.1 Accelerate the conjugate gradient with a partitioned preconditioner

The partitioned-matrix-vector product $\nabla^2 f v$ or $B_k v$ aggregates the computation of $\nabla^2 \widehat{f}_i \widehat{v}_i$ or $\widehat{B}_i \widehat{v}_i$ to fasten conjugate gradient computation, see Section 2.1.4 and Section 3.1.2. However, the conjugate gradient performance can be worsened if the linear system is bad conditioned. Therefore, partial separability can be exploited to define dedicated preconditioners improving conjugate gradient runs. Such a preconditioner is used to disminish the conditioning number of the linear system, which ultimately reduces the iterations needed before the conjugate gradient method converges. Daydé et al. [42] developed and tested several element-by-element preconditioners for a partitioned matrix, i.e., each element Hessian $\widehat{B}_i$ has its own preconditioner. Those preconditioners are inspired from finite element methods and scale on n_i, making the storage reasonable.

The resulting numerical measures indicate that partitioned preconditioners best perform when the overlap between the element Hessians is scarse. As the overlap between elements increases, the performance of partitioned preconditioners becomes close to that of a diagonal preconditioner. Nonetheless, these element-by-element preconditioners remain better for ill-conditionned problems. Consequently, some of those preconditioners were added to the LANCELOT software—see Section 3.6.3.

Lastly, Daydé et al. observe that when two element functions significantly overlap, it may be advantageous to merge both element functions. This observation resulted in the design of two amalgamation algorithms, which tend to speed up both the preconditioning and the operator-vector products, similarly to the work of Conn et al. [33] described in Section 3.3.3.1.

3.3.2.2 Structured factorizations of B_k

Direct methods are alternatives to iterative methods, e.g., the conjugate gradient method, for solving a linear system $B_k s = -\nabla f(x_k)$, and therefore, finding a solution to a line search or a trust-region subproblem. To do so, direct methods find a factorization of B_k, such as

the Cholesky factorization $B_k = LL^T$, where L is a lower triangular matrix, if $B_k = B_k^T \succ 0$. Originally, Irons [89] described a *frontal* implementation of a Cholesky factorization for a sparse matrix $A = A^T \succ 0$ aggregating element contributions $\widehat{A}_i \in \mathbb{R}^{n_i \times n_i}$:

$$A = \sum_{i=1}^{N} A_i = \sum_{i=1}^{N} U_i^T \widehat{A}_i U_i \succ 0.$$

The frontal method takes advantage of the basic operation of the Gaussian elimination

$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj} / a_{kk}$$

which can be performed as soon as the row k —the pivot row— (and therefore column k) has been fully aggregated from some $\widehat{A}_i$. Once the contribution of each $\widehat{A}_i$ to row k of A has been taken into account, it does not need to wait for the rest of A to be assembled. The fully aggregated but not yet eliminated rows and columns of A are maintained in a data structure called the *front*, which must remain small compared to n for computations to be efficient. The chief difference between a frontal method using a sparse matrix or a partitioned matrix is the routine managing the front, which in the latter case must accumulate element contributions properly on the fly while it only requires data access for a sparse matrix.

Frontal methods originated from finite-element simulations in structural mechanics. They became quite popular in continuous optimization thanks in particular to the contributions of Duff and Reid [53], who generalize the approach of Irons [89] to symmetric indefinite systems by drawing inspiration from the dense symmetric indefinite factorization of Bunch and Parlett [22]. In particular, Duff and Reid describe the *multifrontal* method, which maintains multiple fronts simultaneously, and can therefore offer an opportunity for parallel computation. In later years, refinements of the frontal and multifrontal methods made them an essential component of any efficient large-scale optimization software and can be found in libraries such as MA57 [51], MA62 [54], MUMPS [1] or PARDISO [139]. They are used in such widely successful optimization libraries as IPOPT [153], GALAHAD [70], and KNITRO [24].

Like any other sparse factorization, frontal and multifrontal methods may suffer from fill-in, i.e., the factors may be significantly denser than A, see Figure 3.2 for example. Therefore, the first step is to identify a fill-reducing permutation before proceeding with the factorization. To completely avoid fill-in after permutation, Griewank and Toint [78] demonstrated that the pattern of A must be a perfect graph, i.e. having a permutation of A which is an overlapping block diagonal matrix. Furthermore, a good permutation for a (multi-)frontal method should also keep the front size(s) moderate while seeking to develop several fronts.

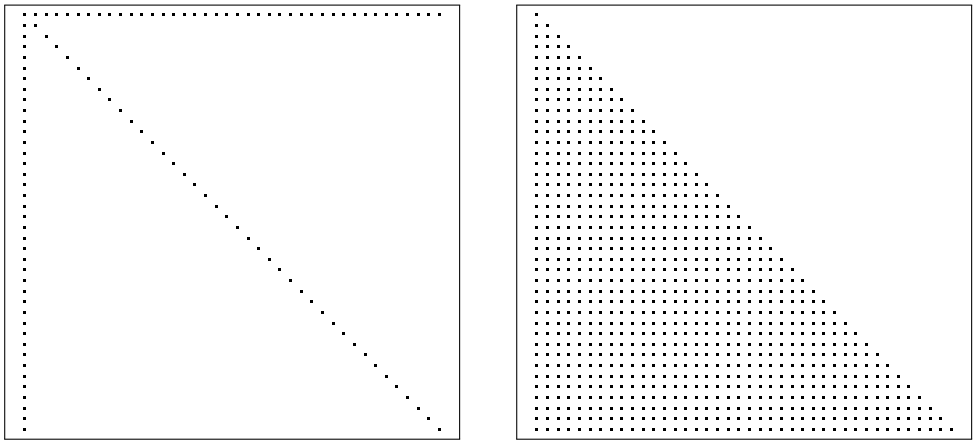


Figure 3.2 Arrow Hessian pattern (left) and its factor (right)

Conn et al. [35] evaluate the performance of a (multi-)frontal method for a partitioned matrix such as (3.7b) or (3.9) in a non linear optimization context. Conn et al. [35] exploit the multifrontal factorization implementation of [52] and conclude, on a relatively limited test set, that its performance is competitive with that of the truncated conjugate gradient method [143] in a trust-region context. They observe that the performance of the multifrontal method is excellent when the trust-region model is convex and the fill-in is moderate. However, its performance decreases in the presence of directions of negative curvature, i.e. $B \not\succ 0$, or when significant fill-in occurs.

3.3.3 Enhance the performance of partitioned quasi-Newton methods

This section shows how partial separability can be preprocessed to improve the performance of any partitioned quasi-Newton method detailed Section 3.2.

3.3.3.1 Merge element functions and choose U_i representation

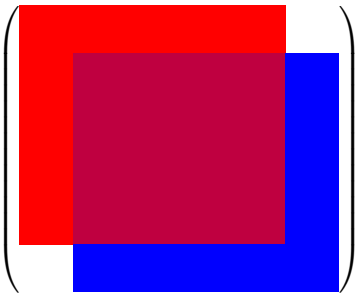
The software LANCELOT developed by Conn et al. [36] is a partitioned quasi-Newton trust-region method, solving at each iterate the trust-region subproblem mainly with the truncated conjugate gradient. Thus, beside the cost implied by computing every $\widehat{f}_i$ and $\nabla \widehat{f}_i$, the main cost of LANCELOT resides in conjugate gradient iterations, whose complexity scales on

$$B_k v = \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i v. \tag{3.26}$$

Investigating how to reduce the computations related to $B_k v$ means improving the trust-region method efficiency. Conn et al. remark that if two elements i and j widely overlap, merging

the elements i and j into a single element reduces the amount of computation needed by $B_k v$. Suppose an extreme example $B \in \mathbb{R}^{n \times n}$ aggregating $\widehat{B}_i \in \mathbb{R}^{n-1 \times n-1}$ and $\widehat{B}_j \in \mathbb{R}^{n-1 \times n-1}$. Hence, $\widehat{B}_i$ and $\widehat{B}_j$ overlap over $n-2$ variables. As B is a dense matrix, Bv requires n^2 floating point operations, or flops, while each $\widehat{B}_i U_i v$ or $\widehat{B}_j U_j v$ requires $(n-1)^2$ flops. Since $n^2 < 2(n-1)^2$, computing Bv requires fewer flops than aggregating $\widehat{B}_i U_i v$ and $\widehat{B}_j U_j v$. This question is related to how memory is managed. The same argument, i.e. $\frac{n(n+1)}{2} < n(n-1)$, favours storing B over separate $\widehat{B}_i$ and $\widehat{B}_j$, even if it means the loss of structural sparsity. An illustration of such an example considering $n = 6$ is as follows:

$$U_i^\top \widehat{B}_i U_i + U_j^\top \widehat{B}_j U_j = \left(\right.$$



$$\left. \right),$$

where the overlapping partial derivatives are in purple.

As a result, Conn et al. come up with a *merge* procedure, merging two elements together when the sparsity gain obtained by storing two distinct elements Hessians is inferior to the intrinsic overlap between elements. The merge procedure is recursively applied over all pairs of elements until a stable decomposition of f is found, which is a simple solution for a large-scale set covering problem, a NP-hard combinatorial problem. The downside of merging is the diminution of the number of elements, resulting in a partitioned matrix B_k denser and closer to an unstructured quasi-Newton approximation than before the merging.

In addition, Conn et al. [36] propose an *expand* procedure to chose whether an element Hessian $\widehat{B}_i \approx \nabla^2 \widehat{f}_i(\widehat{x}_k)$ should use its elemental $\widehat{B}_i^E$ or its internal $\widehat{B}_i^I$ Hessian approximation. This choice reflects on the flops (3.26) needs, as:

$$\widehat{B}_i^E = C_i^\top \widehat{B}_i^I C_i, \quad C_i \in \mathbb{R}^{n_i^I \times n_i^E}, \tag{3.27}$$

where C_i is a sparse matrix representing the linear combinations of variables appearing in $\widehat{f}_i$. C_i is composed of d_i nonzero components. Note that if $n_i^I > n_i^E$, then $\widehat{B}_i^I$ is memory and computationally counterproductive.

The cost of $\widehat{B}_i^E U_i^E v$ is ${n_i^E}^2$, as the linear operator U_i^E only selects pertinent variables in a straightforward and highly efficient way. On the contrary, $\widehat{B}_i^I U_i^I v$ depends on the sparsity of C_i, i.e. d_i, as well as $v \to \widehat{B}_k^I v$ which requires ${n_i^I}^2$ flops. Thus, Conn et al. [36] propose a

criterion expanding $\widehat{B}_i^I$ to $\widehat{B}_i^E$ only if $n_i^{E^2} < 2d_i + n_i^{I^2}$.

Merge cuts off the finer block representation to improve the efficiency of storage and computation by sacrificing sparsity. Expand is a trade-off between storage and computation efficiency. If the memory is an issue, expand should be used cautiously. Both procedures reformulate properly the partially-separable structure before starting the trust-region more efficiently. The results Conn et al. [36] present via LANCELOT show that a combination of merge and expand divides by two the computation while it multiplies by three the storage .

3.3.3.2 Decomposition of a partially-separable function into convex element functions

Griewank and Toint [77] first proved the local superlinear convergence of an inexact partitioned quasi-Newton line search method. To do so, every element function Hessian must be semi-definite positive when reaching an isolated minimizer of f making $\widehat{f}_i$ locally convex and leaving the indefinite case unsupported. From this point, Griewank and Toint [78] study the structure prerequisites and try to convexify every element function. Their approach shifts (artificial) quadratic terms between element functions to force every $\widehat{f}_i$ to be locally convex. As mentioned previously, there is almost an infinite number of decompositions for a partially-separable function f. Therefore, before shifting quadratic terms, Griewank and Toint study thoroughly the partially-separable decomposition, resulting in a procedure comparing two partially-separable structures.

A partially-separable structure may be described as a collection of element ranges derived from the element functions considered. The i-th element range is equivalent to $Span(U_i^\top)$, with an additional constraint on the element function decomposition that $Span(U_i^\top) \not\subset Span(U_j^\top), \forall i \neq j$. This means that

$$U_i = \begin{bmatrix} e_1^\top \\ e_2^\top \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} e_1^\top \\ e_2^\top \\ e_3^\top \end{bmatrix}$$

cannot coexist. Suppose now two collections of element ranges $\{\mathcal{J}_j\}_{j=1}^{N_\mathcal{J}}$ and $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$ of f. $\{\mathcal{J}_j\}_{j=1}^{N_\mathcal{J}}$ is said to be included in $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$ if, for any $j \in \{1, \ldots, N_\mathcal{J}\}$, there exists $i \in \{1, ..., N_\mathcal{R}\}$ such that $\mathcal{J}_j \subseteq \mathcal{R}_i$. In that case, $\{\mathcal{J}_j\}_{j=1}^{N_\mathcal{J}}$ is said to be *finer* than $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$. Conversely, $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$ is *coarser* than $\{\mathcal{J}_j\}_{j=1}^{N_\mathcal{J}}$ as long as $\{\mathcal{J}_j\}_{j=1}^{N_\mathcal{J}}$ and $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$ are different. Note that there may exist almost an infinity of finest partially-separable structures. From theses definitions, Griewank and Toint characterize the totally convex separability structure,

an additional structure ensuring the convexification of element functions when $\nabla^2 f$ is positive definite.

Theorem 3.3.1 (Theorem 1 [78]). *Any totally convex separability structure $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$ must be maximal in that the element ranges of $\mathcal{R}_i$ are exactly the maximal generator ranges of the associated Hessian space $\boldsymbol{H}$. Furthermore, any other partial separability structure with the same Hessian space must be finer than $\{\mathcal{R}_i\}_{i=1}^{N_\mathcal{R}}$.*

Without detailing futher the Theorem 3.3.1, it implies that only the coarsest partially-separable structure can be totally convex. The consequence of the Theorem 3.3.1 is rather disappointing since one would like to work on the finest partially-separable structure due to its sparsest Hessian. However, it allows quadratic shifting terms according to $\{\mathcal{R}_i\}_{i=1}^{m}$ on which the appliance of PBFGS can match the theoretical convergence [77]. Another interesting property of a totally convex function f is the perfect graph pattern of $\nabla^2 f$ due to the coarser partially-separable decomposition. As a consequence, $\nabla^2 f$ or B_k has Cholesky factors without fill-in [78, Theorem 4].

The main issue of the convexification approach is that the shifting must be done at every iteration, which may be computationally intensive. The convexification strategies tested at the time of [78] were too costly, and resulted in slower convergence than a PBFGS applied on f without shifting.

3.3.3.3 New basis to express sparser partially-separable polynomials

Kim et al. [92] are interested in solving a sparse semidefinite positive relaxation of large-scale polynomial optimization problems. As a sum of monomials, polynomials are disposed to be partially-separable functions as long as every monomial does not involve all decision variables. Kim et al. [92] seeks to find a non-singular linear transformation P of decision variables to improve the partially-separable structure of the objective $g(x) = f(Px)$. Kim et al. aim to increase the sparsity of $\nabla^2 g$ compared to $\nabla^2 f$ and get sparser Cholesky factors for $\nabla^2 g$ than $\nabla^2 f$ in order to solve more efficiently the original problem. By increasing the Hessian's sparsity, sparse Newton or partitioned quasi-Newton methods have faster computation and cheaper storage, which can extend the set of problems practically solvable. In addition, the procedure seeks to return a Hessian structure such that the Cholesky factorization has no fill-in, i.e., a maximal partially-separable structure in the sense of [78],

The next example illustrates an ideal case of transformation P. Suppose the polynomial

$$f(x) = \sum_{i=1}^{n-1} (x_i - x_{i+1})^3 + (\sum_{i=1}^{n} x_i)^4.$$

The term $\sum_{i=1}^{n-1}(x_i - x_{i+1})^3$ induces a tridiagonal Hessian, while $(\sum_{i=1}^{n} x_i)^4$ induces a dense Hessian.

Now, consider the non-singular transformation

$$P = \begin{pmatrix} 1 & 0 & & \cdots\cdots & & 0 \\ -1 & 1 & 0 & & \cdots & \vdots \\ 0 & -1 & 1 & 0 & . & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & . & 0 & -1 & 1 & 0 \\ 0 & & \cdots & 0 & -1 & 1 \end{pmatrix}, \quad P^{-1} = \begin{pmatrix} 1 & 0 & & \cdots\cdots & & 0 \\ \vdots & 1 & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & 0 \\ 1 & & \cdots\cdots & & & 1 \end{pmatrix},$$

whose application onto x (i.e. $x = Pz$) returns:

$$\begin{aligned} x_1 &= z_1 \\ x_i &= z_i - z_{i-1} \quad 2 \leq i \leq n. \end{aligned}$$

Transposing Pz to f returns:

$$\begin{aligned} g(z) &= f(Pz), \\ &= (z_1 + (z_2 - z_1))^3 + \textstyle\sum_{i=2}^{n-1} \Big((z_i - z_{i-1}) + (z_{i+1} - z_i)\Big)^3 + \Big(z_1 + \sum_{i=2}^{n}(z_i - z_{i-1})\Big)^4, \\ &= z_2^3 + \textstyle\sum_{i=2}^{n-1}(z_{i+1} - z_{i-1})^3 + z_n^4. \end{aligned}$$

The element functions of g depend on at most two decision variables, making $\nabla^2 g$ tridiagonal.

In order to find P, the first step is to retrieve the nullspace of any polynomial $p(x) : \mathbb{R}^n \to \mathbb{R}$, i.e., finding w such that:

$$p(x + \lambda w) - p(x) = 0, \ \forall \lambda \in \mathbb{R} \text{ and } \forall x \in \mathbb{R}^n. \tag{3.28}$$

By factorizing (3.28) terms with λ, a new linear equation system appears, whose solutions form the polynomial nullspace. An example of such a factorization is given in [92, Section 2.3]. Thereafter, the columns of P are selected from the nullspace basis resulting of the polynomial nullspace.

In order to assess the quality of g, Kim et al. formalize the correlative sparsity pattern E:

$$\begin{aligned} K(h) &= \{j \in \{1, ..., n\} \mid e_j \in nullspace(h)\}, \\ E(\{\widehat{g}_i\}_{i=1}^{N}) &= \textstyle\bigcup_{j \in \{1,...,N\}} (\{1, ..., n\} \backslash K(\widehat{g}_j)) \times (\{1, ..., n\} \backslash K(\widehat{g}_j)), \end{aligned}$$

as an indicator of $\nabla^2 g$ sparsity and that of $\nabla^2 g$ Cholesky factors. In practice, E enables the extraction of the adjacency matrix of an undirected graph, similarly to $\sum_{i=1}^{N} U_i^\top U_i$. The cardinality $\#E(g)$ measures the sparsity of $\nabla^2 g$, since $\frac{\partial^2 g}{\partial x_i \partial x_j} = 0, \forall (i,j) \notin E$. Therefore, if $\#E(g) \ll n^2$ then $\nabla^2 f$ is (very) sparse. Moreover, as stated in Griewank and Toint [78], if the graph induced by $E(g)$ is chordal, $\nabla^2 g$ is an overlapping block diagonal matrix with Cholesky factors as sparse as $\nabla^2 g$.

Kim et al. find a transformation P by resolving a combinatorial problem with a greedy algorithm. The problem looks for transformed decision variables which are invariant for most transformed element functions. The relevancy of transformed decision variables is assessed with E. The problem is solved in two parts. The first one solves approximately the combinatorial problem, while the second checks for the feasibility of the solution determined.

The numerical results state that after the application of P, the transformed polynomials have: smaller monomial (element) dimensions, a sparser Hessian and sparser Cholesky factors. However, despite solving more efficiently the subsequent transformed polynomial, finding the transformation P takes more CPU time than solving the original polynomial. Nonetheless, the approach remains relevant as the transformed problem allows dealing with larger polynomial problems which could not have been solved without the sparsification resulting from P.

3.4 Partially-separable problems with dedicated methods

This section compiles two partially-separable problems for which partial separability allows the design of dedicated methods.

3.4.1 Large scale nonlinear network problems

In Toint and Tuyttens [151], the authors are interested in solving a large scale nonlinear network problem:

$$\begin{array}{ll} \min\limits_{x \in \mathbb{R}^n} & f(x) = \sum_{i=1}^{N} \widehat{f}_i(U_i x) \\ \text{s.t.} & Ax = b \\ & l \leq x \leq u \end{array} , \tag{3.29}$$

where f is a partially-separable function, $A \in \mathbb{R}^{m \times n}$ $(m \leq n)$ is a node-arc incidence matrix, b is the supply/demand vector and $l, u \in \mathbb{R}^n$ are the upper and lower bounds for the flow of the oriented network that A models. Such models gather a large set of problem: energy distribution problem, hydroeletric power management or urban traffic network analysis [151].

To keep the sequence of iterative solutions feasible, the step taken at each iteration lies in

$Null(A)$ spanned by Z. To characterize Z, A is divided in three $A = (B\,S\,N)$ (*basic*, *superbasic* and *nonbasic* variables) where B is a square nonsingular matrix, enabling a straightforward choice for $Z = (-B^{-1}S\ I\ 0)^\top$. Similarly, the step s is divided in three $s = (s_\mathcal{B}\, s_\mathcal{S}\, s_\mathcal{N})^\top$. As the steps minimizing the quadratic approximation of f (3.29) lie in $span(Z)$, at each iteration the linear system:

$$(Z^\top GZ)s_\mathcal{S} = -Z^\top \nabla f(x), \quad G = \sum_{i=1}^{N} U_i \widehat{G}_i U_i, \quad \widehat{G}_i \approx \nabla^2 f_i(x), \tag{3.30}$$

$$Bs_\mathcal{B} = -Ss_\mathcal{S}, \tag{3.31}$$

$$s_\mathcal{N} = 0, \tag{3.32}$$

must be solved. Concretely, $s_\mathcal{S}$ tries to minimize the quadratic approximation of f while $s_\mathcal{B}$ focuses on satisfying $As = 0$ to preserve feasibility.

Toint and Tuyttens choose to solve (3.30) using the truncated conjugate gradient method [84, 143] which does not form $Z^\top GZ$, a potentially dense matrix, and requires only $v \to (Z^\top GZ)v$ and exploits the partitioned structure of G. After determining $s_\mathcal{S}$ from (3.30), a line search is performed to find α such that $\alpha s_\mathcal{S}$, $s_\mathcal{B}$ and $s_\mathcal{N}$ satisfy the constraints of (3.29). To do so, the projection $P(x_\mathcal{S} + \alpha s_\mathcal{S})$ tries if $x_\mathcal{S} + \alpha s_\mathcal{S}$ satisfies the upper/lower bounds constraints. The step size α is increased iteratively until $x_\mathcal{S} + \alpha s_\mathcal{S}$ first reaches the upper/lower bound constraints. Then, it computes $s_\mathcal{B} = -\alpha B^{-1} S s_\mathcal{S}$ and sets $s_\mathcal{N}$ to 0. Those two phases are repeated until the overall step $(s_\mathcal{B}, \alpha s_\mathcal{S}, 0)^\top$ parametrized by α is feasible and achieves a sufficient decrease of f.

The mandatory decrease can be computed from $s_\mathcal{S}$, as $s_\mathcal{S}$ contains the only variables able to bring a decrease:

$$f(x + (s_\mathcal{B}, \alpha s_\mathcal{S}, 0)^\top) - f(x) \leq \mu_1 (Z^\top \nabla f(x_k))^\top s_\mathcal{S}, \quad 0 \leq \mu_1 \leq \tfrac{1}{2}.$$

In the particular case where superbasis independent sets exist, then $Z^\top GZ$ becomes block diagonal and each independent set may have its own (cheaper) block-optimization process.

The numerical results show a comparison of four variants using different methods for approximating the element Hessians $\widehat{G}_i$: an analytic second derivatives, a finite difference estimation, a quasi-Newton update —BFGS until a negative curvature is found to switch over SR1— and the Dembo method [44], which consists in the linear operator:

$$U_i^\top \nabla^2 \widehat{f}_i(x) U_i s \approx U_i^\top \widehat{G}_i U_i s = U_i^\top (\nabla \widehat{f}_i(U_i(x + \epsilon s)) - \nabla \widehat{f}_i(U_i x)),$$

where ϵ is an appropriate step length, e.g. 10^{-6}.

All those variants are implemented by the Fortran subroutine LSNNO [152]. The most effective variant, considering both CPU time and iterations, is the one exploiting analytic second derivatives. When they are inaccessible and if the gradient is computationally expensive, the partitioned quasi-Newton method would be preferred over the finite difference element Hessian approximation. The performance of the Dembo's variant varies strongly depending on the problem considered. By approximating every Gv with a gradient computation, each outer iteration calculates as many gradients as conjugate gradient iterations needed. On most problems studied in [152], the Dembo's variant performs far more conjugate gradient iterations, and therefore, requires far more gradient evaluations. Nonetheless, it does not reflect linearly on CPU time, even though, it usually converges slower than the three other variants.

3.4.2 Signal reconstruction

Hamam and Romberg [81] formulate a partially-separable problem to reconstruct a streaming signal. The model considers that every element function is convex, $\mathcal{C}^2$ and depends on two element variables. Specifically, the signal is reconstructed iteratively by minimizing a sequence of partially-separable subproblems:

$$J_n(x) = \sum_{i=1}^{n} l_i(x_{i-1}, x_i),$$

where l_i may be a least-squared loss on observed data over a time frame parametrized by x_{i-1} and x_i (starting from x_0). J_{n+1} is formed by adding the element loss $l_{i+1}(x_i, x_{i+1})$ to J_n, which has for effect to continue the signal reconstruction process over a new time frame. Structurally, $\nabla^2 J_n$ is a tridiagonal matrix, formed by overlapping element Hessians of size 2. Therefore, $\nabla^2 J_n$ pattern forms a chordal graph and may be LU-factorized without fill-in by bi-diagonal triangular matrices [78]. The structures of the Hessian and its factor are illustrated in Figure 3.3

To minimize the sequence of J_n, Hamam and Romberg present a general algorithm for any problem having a block tridiagonal Hessian (including tridiagonal Hessian). The algorithm relies on the proof that if $\nabla^2 J_n$ and $\nabla^2 J_{n+1}$ are diagonal dominant matrices, then the solution vector ${x^{J_n}}^* \in \mathbb{R}^{n+1}$ of J_n is only weakly coupled to the solution ${x^{J_{n+1}}}^* \in \mathbb{R}^{n+2}$ of J_{n+1}, i.e., as $n - i$ increases, ${x_i^{J_n}}^* - {x_i^{J_{n+1}}}^*$ decreases exponentially [81, theorem 3.3]. Intuitively, the more ${x_i^{J_n}}^*$ parametrizes a distant time of the signal's reconstruction J_n, the less ${x_i^{J_n}}^*$ value is related to that of ${x_n^{J_n}}^*$.

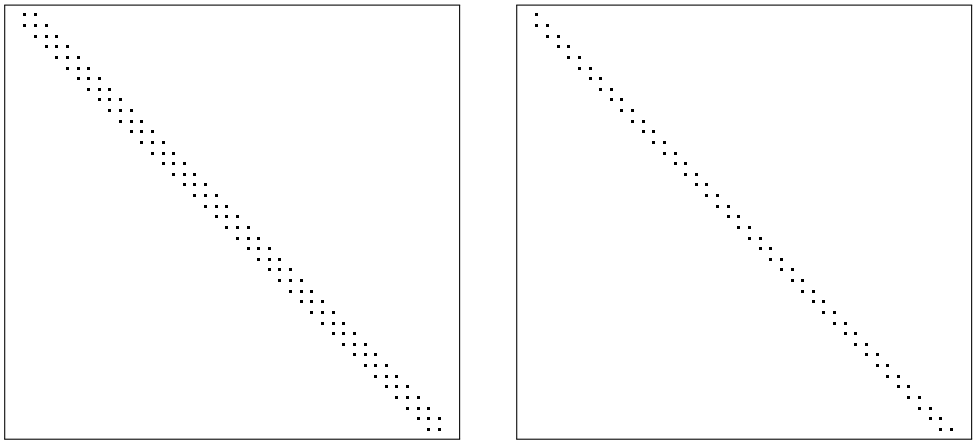


Figure 3.3 Tridiagonnal Hessian pattern (left) and its factor (right)

From this observation, $x^{J_n{}^*}$ resulting from the minimization of J_n is recycled to warm start the minimization of J_{n+1}. Thereafter, J_{n+1} is minimized iteratively by solving the Newton linear system $\nabla^2 J_{n+1}(x)s = -\nabla J_{n+1}(x)$ until $\|\nabla J_{n+1}(x)\|$ is sufficiently low. The linear system is solved by exploiting a LU factorization of $\nabla^2 J_n$, for which a forward-backward process computing s on the fly is proposed. As $x_i^{J_n{}^*} - x_i^{J_{n+1}{}^*}$ decreases exponentially when i is far from n, Hamam and Romberg ultimately update only the last components of $x^{J_n{}^*}$ to find $x^{J_{n+1}{}^*}$, i.e., $x_i^{J_{n+1}{}^*} = x_i^{J_n{}^*}$, $\forall 1 \leq i \leq n-m$, parametrized by the buffer size m. It takes advantage of the forward-backward process to cut off computation related to the Newton update beyond the cache. As a consequence, the update's complexity is independent of the dimension of $\nabla^2 J_n$ and remains constant even though n increases.

The numerical results explore the reconstitution of an intensity function (denoted as $\lambda(t)$) of a non-homogeneous Poisson process. [81, Figure 1] illustrates how the signal reconstruction error at a given time diminishes as new J_{n+k}, $k > 0$ are successively minimized. [81, Figure 2] empirically corroborates how the buffer's size growth causes the error of the solution to decrease (exponentially) before it reaches a threshold, beyond which, updating variables becomes irrelevant.

3.5 Computation of the partitioned derivatives

The computation of derivatives is mandatory for minimizing a nonlinear problem with quasi-Newton methods. It serves many purposes, such as assessing if a critical point has been reached, computing a descent direction or performing a quasi-Newton update. All those cases are illustrated by the Algorithm 2.1.3. For performing partitioned quasi-Newton updates, one must compute (in most cases) the element gradients $\nabla \widehat{f}_i$. This section intends to list

the schemes proposed over the years to compute the element gradients and the partitioned Hessian-vector product.

3.5.1 Straightforward procedure when $\widehat{f}_i$ is available

When $f : \mathbb{R}^n \to \mathbb{R}$ is partially-separable and each $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$ is individually accessible, both automatic differentiation modes can be applied directly onto $\widehat{f}_i$ for computing element gradients $\nabla \widehat{f}_i$ of size n_i. In the case where n_i is much smaller than n, the difference between both forward and reverse modes disminishes, and should theoretically take at most $5.n_i^{\max}$ the time of one f evaluation.

Unlike the forward mode that benefits indirectly the small dimensions n_i, the reverse mode does not take as much advantage of the partial separability to speed up its computation. However, instead of keeping the tape of f, it keeps one smaller tape for every $\widehat{f}_i$. In practice, several element functions may be identical, but applied onto different variables, i.e. $\widehat{f}_i(U_i x) = \widehat{f}_j(U_i x)$. Consequently, the reverse mode only needs to store the distinct element function tapes instead of every element function tape, which may reduce significantly the storage requirement as well as reducing the computational resource needed to build the tapes.

In addition, the computation of the Hessian-vector product $v \to \nabla^2 f(x)v$ in large problems can benefit from the partial separability of f:

$$\nabla^2 f(x)v = \left(\sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_i) U_i \right) v = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_i) \widehat{v}_i.$$

The Hessian-vector product can be computed efficiently by using sequentially the reverse mode and the forward mode [73]. Applying this technique to every $\widehat{f}_i$ permits the computation of every $\nabla^2 \widehat{f}_i(\widehat{x}_i)\widehat{v}_i$ which can be accumulated to form $\nabla^2 f(x)v$. These approaches compute $\nabla \widehat{f}_i(\widehat{x}_i)$ or $\nabla^2 \widehat{f}_i(\widehat{x}_i)\widehat{v}_i$ independently, and thus, can be parallelized straightforwardly before accumulating sequentially the element contributions [32].

The next section explains the computation of the partitioned gradient when only f can be evaluated, i.e., $\widehat{f}_i$ cannot be evaluated individually. This situation occurs when f results from a "black-box" numerical simulation for which the partially-separable structure is known.

3.5.2 Exploiting directional derivatives

In the case all $\widehat{f}_i$ are evaluated at once, the partitioned derivatives can be computed by applying a technique originated from sparse Jacobian computation that employs directional derivatives [40]. To apply this technique, f such as (3.1) must be reformulated as $F : \mathbb{R}^n \to$

$\mathbb{R}^N$:

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_N(x) \end{pmatrix}, \quad f(x) = \mathbb{1}^\top F(x), \quad \nabla f(x) = \mathbb{1}^\top \nabla F(x), \quad \nabla F(x) = \begin{pmatrix} \nabla f_1(x)^\top \\ \nabla f_2(x)^\top \\ \vdots \\ \nabla f_N(x)^\top \end{pmatrix}.$$

Since each f_i depends only on few variables, the Jacobian ∇F is (very) sparse. Suppose the example $f(x) = \widehat{f}_1(x_1, x_3) + \widehat{f}_2(x_1, x_4) + \widehat{f}_3(x_2, x_3) + \widehat{f}_4(x_2, x_4) + \widehat{f}_5(x_3)$, then the sparsity pattern of ∇F is:

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \\ f_4(x) \\ f_5(x) \end{pmatrix}, \quad \nabla F = \begin{pmatrix} \square & & \triangle & \\ \square & & & \diamond \\ & \lozenge & \triangle & \\ & \lozenge & & \diamond \\ & & \triangle & \end{pmatrix}, \tag{3.33}$$

where $\square, \triangle, \lozenge$ and $\diamond$ represent non-zero components.

Usually, finite difference and forward automatic differentiation compute partial derivatives considering a seed matrix. For example, by carrying out the direction $s = e_1$, both finite difference and forward automatic differentiation estimate the first column of ∇F, i.e. $\square$ (3.33). Hence, the seed is generally set to $S = I_n$. Curtis et al. [40] propose to accelerate the computation of ∇F by finding sets of orthogonal columns of ∇F. A set of orthogonal columns allows a safe evaluation of several element function derivatives with one direction summing several e_i, where i are the orthogonal column indices. The sets of orthogonal columns are found with a column colouring performed onto the sparsity pattern of ∇F. Since the sparsity pattern of ∇F remains identical over the iterations, the colouring is performed only once. After the colouring, the directions determined are gathered in a *compressed* matrix S_c. For example, considering (3.33):

$$\begin{pmatrix} \square & & \triangle & \\ \square & & & \diamond \\ & \lozenge & \triangle & \\ & \lozenge & & \diamond \\ & & \triangle & \end{pmatrix}, \quad \text{leads to } S_c = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

Thereafter, the direction $s_1 = e_1 + e_2$ computes $\frac{\partial f_j}{\partial x_1}$, $j = 1, 2$ and $\frac{\partial f_j}{\partial x_2}$, $j = 3, 4$, while $s_2 = e_3 + e_4$ computes $\frac{\partial f_j}{\partial x_3}$, $j = 1, 3, 5$ and $\frac{\partial f_j}{\partial x_4}$, $j = 2, 4$. Similar approaches exist to construct either a partitioned interpolation of $f, \nabla f$ and $\nabla^2 f$ [31], or to generate a direction search s minimizing

the number of element-function calls to evaluate $f(x+s)$ [125], both methods are detailed in Section 3.8.

The amount of element gradients evaluated simultaneously depends on the cardinal of each orthogonal column set. Therefore, for best efficiency, the colouring should be balanced. Overall, the reduction obtained depends on the Jacobian sparsity induced by the partially-separable structure of f. The most extreme case is total separability, i.e. $U_i = e_i^\top,\ 1, \ldots, N$, making ∇F diagonal, where one adapted $s = \sum_{i=1}^{n} e_i$ is enough to compute the whole Jacobian.

Coleman and Moré [30] applied this scheme using finite differences while Bischof and El-khadiri [12] used it with forward automatic differentiation. Bischof and El-khadiri [12] contribution extends ADIFOR, an automatic differentiation library in Fortran, to compute efficiently the partitioned gradient without having individual access to every $\widehat{f}_i$. The approach is reported to not consume additional storage, while numerical results relate performance few times slower than a hand coded gradient for very sparse problems. Bischof et al. [13] prolong [12] with the library SparsLinC, which integrates a *sparse* approach that does not require the user to inform partial separability to compute a sparse ∇F, i.e., the U_i are unknown. The numerical results report that the sparse approach computes the sparse Jacobian 10 times slower than the method developed in [12]. Finally, Bouaricha and Moré [16] combine the ideas from [12] and [13] to create a *hybrid* approach that utilizes the sparse approach to find the Jacobian sparsity in the early iterates and then determines a colouring of ∇F for subsequently applying the compressed approach during the following iterations. The hybrid approach is integrated in the ELSO framework [16]. As the extra cost of *sparse* iterates is flooded among all iterates, the hybrid variant performance is similar to the compressed variant while benefiting the usability of the sparse approach.

3.6 Partitioned quasi-Newton software

The main objectives of exploiting partial separability is to develop efficient numerical implementations while performing a partitioned quasi-Newton approximation of the Hessian. It is therefore natural that several software libraries emerged over the years that make it possible to model and solve large problems faster than implementations that do not exploit the partially-separable structure. In this section, we review a number of such libraries and their behaviour in solving large-scale optimization problems. The review is organized in approximate historical order of appearance.

3.6.1 PSPMIN

One of the earliest software implementations that exploits the partially-separable structure is the PSPMIN routine of Griewank and Toint [79], which implements a partitioned quasi-Newton method described by Griewank and Toint [75, 76, 77] and Toint [149] for bound-constrained optimization. PSPMIN extends the research implementation of Griewank and Toint [77] to nonconvex decompositions, i.e., decompositions of the form (3.1) where the element functions f_i are not necessarily convex, and to problems with bound constraints.

PSPMIN is a trust-region method in which a quasi-Newton quadratic model of the objective is updated at each iteration and used to compute a step. The step is first computed as the solution of the partitioned quasi-Newton equation $B_k s = -\nabla f(x_k)$ by the conjugate gradient method [84] with diagonal preconditioner and adaptive stopping tolerance. The truncation strategy of Steihaug [143] is applied if a direction of negative curvature is found during the conjugate gradient iterations, see Section 2.1.4 for more details. However, if no negative curvature direction is found, the trust-region constraint is not enforced so as to allow the use of a full quasi-Newton step when the tolerance shrinks. A line search is subsequently performed along the projection of the step into the trust-region so as to ensure sufficient decrease. The element Hessian approximations are then updated in such a way that the secant equation (2.12) remains satisfied. PSPMIN capitalizes on the partially-separable structure to save function and gradient evaluations by only evaluating elements whose variables changed significantly during the most recent iteration. Griewank and Toint note that the quasi-Newton approximations of the element functions are typically singular beyond the fact that they only depend on a subset of variables, and make provision for their efficient storage. They also observe that several element Hessians may share the same range space. Thus, instead of storing dense element matrices B_i^E of size n_i^E (details Section 3.1.1), they write $B_i^E = C_i^\top B_i^I C_i$ and store the dense internal element matrices B_i^I of size $n_i^I = \text{rank}(B_i)$, which is the same equation as (3.27). The matrix B_i^I represents a particular choice of internal variables for $\widehat{f}_i$, making $C_i = U_i^I (U_i^E)^\top \in \mathbb{R}^{n_i^I \times n_i^E}$. Whether one chooses B_i^E or B_i^I is studied in details by [36], which is summarized in Section 3.3.3.1. The matrices C_i are not stored, but the user is required to provide functions to compute matrix-vector products with C_i and $C_i^\top$ and to solve systems with coefficients C_i and $C_i^\top$. Each individual B_i^I is updated using the BFGS formula until the curvature condition (2.14) fails, after which B_i^I is updated using the SR1 formula. The update is skipped if (2.14) fails and the SR1 denominator is too close to zero.

PSPMIN is also available as subroutine VE08, which is part of the HSL Mathematical Software Library [86], formerly the Harwell Subroutine Library. The user is required to supply functions to evaluate each $\widehat{f}_i$ and $\nabla \widehat{f}_i$. If exact derivatives are not provided, they are approximated

by way of finite differences. Even when exact derivatives are available, finite differences may be used to compute an initial quasi-Newton approximation, a variant that is reported to be efficient on average, though slightly less robust than a simple diagonal initialization on the problems tested by Griewank and Toint. PSPMIN performed favorably compared to earlier methods that employed *sparse quasi-Newton updates* in an attempt to solve large problems with sparse Hessians, such as those of Shanno [140]. Even though the problems tested were rather small by today's standards, there is no doubt that PSPMIN ignited further research into efficient computational procedures for large-scale problems and indicated that exploiting partial separability was an effective way to do so.

Clearly, an algorithm such as PSPMIN could in theory be accelerated if computations on the element functions are dispatched to different processors. However, we only expect a speedup if those computations are sufficiently costly, otherwise, the communication costs between processors will dash any hopes of time savings. Griewank and Toint comment on those aspects, although [79] does not provide a parallel implementation of PSPMIN.

A variant of PSPMIN for large-scale nonlinear least-squares problems is available as subroutine VE10 in the HSL Mathematical Software Library [86], and is described by Toint [150]. VE10 is designed for problems of the form

$$f(x) := \tfrac{1}{2}\|r(x)\|^2 = \tfrac{1}{2}\sum_{i=1}^{m} r_i(x)^2, \quad r(x) = \begin{bmatrix} r_1(x) \\ \dots \\ r_m(x) \end{bmatrix}, \quad r : \mathbb{R}^n \to \mathbb{R}^m,$$

subject to bound constraints. It is able to employ separate models for each element residual $r_i(x)$, and updates an approximation

$$\nabla r_i(x) \nabla r_i(x)^\top + r_i(x) B_i,$$

where $B_i = B_i^\top \approx \nabla^2 r_i(x)$ is either zero, which corresponds to a Gauss-Newton model, or is updated by way of the BFGS formula if the curvature condition $s^\top(\nabla r_i(x+s) - \nabla r_i(x)) > 0$ is satisfied, and a safeguarded symmetric rank-1 update otherwise. Like PSPMIN, VE10 is a trust-region method, in which a search direction is computed using the truncated conjugate gradient method of Steihaug [143]. A line search is subsequently performed to ensure satisfaction of the Wolfe conditions. Toint observes that a full quasi-Newton model is more efficient and reliable than a pure Gauss-Newton model on average, but that a pure Gauss-Newton model is far more efficient on a subset of problems. This observation motivates the design of a strategy to select a model type for each element function individually at each iteration, but the numerical performance is disappointing due to frequent changes between

model types. Toint experiments with strategies that tend to prefer element quasi-Newton models and his *biased best fit overall* strategy compares favorably with NL2SOL [46, 47], which employs a dense overall quasi-Newton model.

3.6.2 Revenge of the SIF

Although Griewank and Toint [79] state that inputting the partially-separable structure into PSPMIN is "usually very easy",[1] it remains inconvenient when modeling a problem manually, and may be better suited to situations where the user injects a partially-separable problem from another program into PSPMIN. When modeling by hand, say when experimenting with various formulations of a problem, specifying the explicit partially-separable structure can be tedious. Toint [148] released a report with a complete description of the test set used in the experiments of Griewank and Toint [79]. He provides an analytic statement of each of the 50 problems along with its Fortran implementation. It is not difficult to see how enriching such a test set, or correcting a bug, is error prone and cumbersome, and how specifying large problems can be considerably more difficult than specifying small problems. However, for purposes of benchmarking, such a test set is precious. A step towards making it easier to model problems and input the partially-separable structure into routines that can exploit it was made with the creation of the SIF: the Standard Input Format, initially described by Conn et al. [34], who readily acknowledge[2]

> "[...] the very idea of defining a *Standard Input Format* (SIF) was forced on the authors by the sheer difficulty they experienced in specifying large problems."

The SIF is inspired from the earlier MPS, or *Mathematical Programming System*; a specification format designed to represent, archive and share linear and mixed-integer linear optimization problems. Early versions of MPS were set up for punch cards and users would represent problems in column-oriented fashion with different data occurring in different columns on the card. Although the MPS later developed a "free form", where columns no longer needed to be respected, the SIF preserved the column-oriented problem representation.

The SIF lets users represent problem functions in group-partially-separable form —see Section 3.1.3. To a user accustomed to using algebraic modeling languages, the SIF may appear as a program in assembly language would appear to a user of a high-level interpreted language. Algebraic modeling languages such as AMPL[3] [62] and GAMS[4] [19] were already in use when

[1] Bottom of p. 208.
[2] Bottom of p. 14.
[3] A Modeling Language for Mathematical Programming
[4] The General Algebraic Modeling System

the SIF was developed but they did not acquire the capability to detect the partially-separable structure automatically until several years later—see Section 3.6.4. In addition, the SIF was in the public domain.

Problems modeled in the SIF must be translated into a set of Fortran subroutines that fully decribe the group-partially-separable structure of each problem function: the groups, the transformations from elemental to internal variables, the element functions, and so forth. That is the task of the *decoder*. The resulting Fortran subroutines can subsequently be compiled and linked together with the main solver. To this day, the process remains slightly cumbersome as a separate executable must be compiled and linked for each problem. There is no support for shared libraries. We note in passing that the recent interface between the SIF decoder and the Julia [5] programming language implemented by Orban et al. [118] does make provision for shared libraries.

Bongartz et al. [14] recognized that not all solvers are written with the SIF in mind and assembled the Constrained and Unconstrained Testing Environment (CUTE): a library of tools to facilitate high-level interaction with problems modeled in SIF and translated by the decoder, including: evaluating the objective and its derivatives, evaluating the Lagrangian and its derivatives, multiplying the constraint Jacobian by a vector, and so forth. CUTE also gives access to the collection of problems modeled in SIF and to a tool used to select a subset of problems matching a number of requirements. Conn et al. [34] report that at the time of its inception, more than five hundred problems had been modeled in SIF, including the PSPMIN problems described by Toint [148]. At the time of the writing of [14], the collection contained 738 problems. More importantly, CUTE also featured interfaces to ten optimization packages along with scripts that took care of compilation and linking, and let the user run a solver on a test problem from the command line or from a script. Finally, CUTE also contained a MATLAB interface to the tools that let users write solvers or simply interact with models in a high-level language.

There is no denying that the SIF resulted in the first significant collection of test problems for large-scale nonlinear optimization and remains one of the main interfaces of continuous optimization solvers. Today, highly successful solvers such as IPOPT [153], KNITRO [24] and SNOPT [65] all have a CUTE interface. Gould et al. [69] released CUTEr, the next iteration of CUTE, which, among other features, provided interfaces to 13 additional solvers, extensions of the SIF to model quadratic optimization problems explicitly in a way that is compatible with the QPS format—an MPS extension to quadratic and mixed-integer quadratic optimization problems [109]—support for parameters in SIF files, and facilities to interface with solvers written in Fortran 95 and C/C++. The most recent iteration is named CUTEst [71] and

features Fortran 2003 support, dynamic allocation, and approximately 200 new problems compared to CUTEr. At the time of this writing, the CUTEst collection[5] features nearly $1,500$ problems.

3.6.3 LANCELOT and GALAHAD

LANCELOT was probably the first implementation of a large-scale optimization solver that built upon the partially- and group-partially-separable structure of problems, and it did so via the SIF modeling system. LANCELOT is designed for general nonlinear constrained problems in the form

$$\underset{x}{\text{minimize}}\ f(x) \quad \text{subject to } c(x) = 0,\ \ell \leq x \leq u,$$

where both the objective and constraints $c : \mathbb{R}^n \to \mathbb{R}^m$ are assumed to be twice-continuously differentiable and group-partially-separable with given structure. The equality constraints are taken into account by way of an augmented Lagrangian algorithm, i.e., LANCELOT solves a sequence of problems of the form

$$\underset{x}{\text{minimize}}\ L(x; y, \rho) \quad \text{subject to } \ell \leq x \leq u,$$

for fixed values of the parameters $y \in \mathbb{R}^m$ and $\rho > 0$, where $L(x; y, \rho) \coloneqq f(x) - y^\top c(x) + \frac{1}{2}\rho\|c(x)\|^2$ is the augmented Lagrangian. Suppose $f(x) = \sum_{i=1}^{N_0} \widehat{f}_i(U_i x)$ with

$$c_j(x) = \sum_{i=1}^{N_j} \widehat{c}_{j,i}(U_{j,i}x) = 0, \quad \forall\ 1 \leq j \leq m,$$

where $U_{j,i}$ collects the variables parametrizing $c_{j,i}$, then

$$L(x; y, \rho) = \sum_{i=1}^{N_0} \widehat{f}_i(U_i x) + \sum_{j=1}^{m}\sum_{i=1}^{N_j} y_j \widehat{c}_{j,i}(U_{j,i}x) + \sum_{j=1}^{m}\sum_{i_1=1}^{N_j}\sum_{i_2=1}^{N_j} \underbrace{\rho \widehat{c}_{j,i_1}(U_{j,i_1}x)\widehat{c}_{j,i_2}(U_{j,i_2}x)}_{\widehat{h}_{j,i_1,i_2}(U_{j,i_1,i_2}x)},$$

where U_{j,i_1,i_2} combines U_{j,i_1} and U_{j,i_2}. Knowledge of the group-partially-separable structure of f and c translates to knowledge of that of $L(x; y, \rho)$, so that its values and derivatives may be evaluated or approximated efficiently. For example, suppose:

$$\widehat{h}_{j,i_1,i_2}(x_1, x_2, x_3) = \widehat{c}_{j,i_1}(x_1).\widehat{c}_{j,i_2}(x_2, x_3),$$

[5] `https://bitbucket.org/optrove/sif`

and suppose that $\widehat{c}_{j,i_1}(x_1) = 4x_1$ and $\widehat{c}_{j,i_2}(x_2, x_3)$ is a nonlinear function. Then, the partial derivatives of $\widehat{h}_{j,i_1,i_2}$ are:

$$\frac{\partial \widehat{h}_{j,i_1,i_2}}{\partial x_1} = 4\widehat{c}_{j,i_2}(x_2, x_3), \quad \frac{\partial \widehat{h}_{j,i_1,i_2}}{\partial x_2} = \widehat{c}_{j,i_1}(x_1).\frac{\partial \widehat{c}_{j,i_2}(x_2, x_3)}{\partial x_2} = 4x_1.\frac{\partial \widehat{c}_{j,i_2}(x_2, x_3)}{\partial x_2},$$

and

$$\frac{\partial \widehat{h}_{j,i_1,i_2}}{\partial x_3} = \widehat{c}_{j,i_1}(x_1).\frac{\partial \widehat{c}_{j,i_2}(x_2, x_3)}{\partial x_3} = 4x_1.\frac{\partial \widehat{c}_{j,i_2}(x_2, x_3)}{\partial x_3}.$$

Even without the linear nature of $\widehat{c}_{j,i_1}$, if the variables of $\widehat{c}_{j,i_1}$ and $\widehat{c}_{j,i_2}$ are not overlapping, then $\nabla \widehat{h}_{j,i_1,i_2}$ is mainly a recombination of $\nabla \widehat{c}_{j,i_1}$ and $\nabla \widehat{c}_{j,i_2}$. Also, when computing $\rho \nabla \widehat{h}_{j,i_1,i_2}$, ρ can be simply dispatched after aggregating the contributions of each $\frac{\partial \widehat{h}_{j,i_1,i_2}}{\partial x_i}$.

Each bound-constrained subproblem above is solved inexactly using a trust-region method [39]. At each trust-region iteration, a quadratic model $m(s) \approx L(x + s; y, \rho)$ is approximately minimized inside the intersection of an ℓ_∞-norm ball and the box $\ell \leq x + s \leq u$ using a projected search procedure named SBMIN. The model m employs either exact second derivatives or quasi-Newton approximations. In the course of the minimization of the model, SBMIN requires the solution of Newton-type systems $Bs = -\nabla L(x, y, \rho)$, where $B = B^\top \approx \nabla^2 L(x; y, \rho)$ may be indefinite, semi-definite or definite. One option is to use an iterative method to compute s, and SBMIN implements the conjugate gradient method [84] with a choice of preconditioners. Another option is to factorize B. Among other choices, SBMIN uses the sparse multifrontal factorization package MA27 of Duff and Reid [52, 53]. MA27 required B to be assembled prior to analysis and factorization. It is not until subroutine MA62 of Duff and Scott [54] that the factorization of symmetric matrices in finite-element format became available, although MA62 is a frontal method and is limited to positive-definite systems as it does not perform numerical pivoting.

The GALAHAD library [70] features an updated version of LANCELOT. From the point of view of partial separability, one of the main additions is the use of structured trust-regions [37] —Section 3.3.1.

3.6.4 The AMPL modeling language

In the AMPL modeling language, developed by Fourer et al. [61] as an algebraic modeling language for optimization, nonlinear objectives and constraints are modeled by combining user-defined variables using predefined operators and functions. The user may define expressions that appear several times in the statement of a problem, or that help make the model more readable. Expressions and combinations of expressions are translated to an internal format

comprised, among others, of a directed acyclic graph (DAG) indicating how constants and variables are combined to form those expressions. The leaves of the graph are those constants and variables, while intermediate nodes represent operators and functions used to combine them. Such format is often used in modeling languages but also in other areas, such as, for instance, compilers. The immediate benefit of the DAG is that once values are assigned to variables, they can be propagated through the graph to evaluate each subexpression, i.e., to assign a value to each intermediate node representing the value of the expression represented by the subgraph of its descendants. The same structure can be used to propagate derivatives via AD in forward or reverse mode, which has long been a key feature of AMPL for smooth optimization. Gay [64] briefly reviews how AD works in AMPL and extends tree walks to automatically identify the partially-separable structure, which greatly simplifies the user's task. The strategy is to walk the DAG and accumulate linear subexpressions until a nonlinear operation occurs. Those linear subexpressions determine rows of the U_i in (3.3) while the nonlinear operations determine $\widehat{f}_i$. Linear terms are subsequently normalized and inserted into a hash table so duplicates can be found efficiently. Gay [64] determines that for a specific protein-folding problem, the time spent detecting the partially-separable structure approximately amounts to the time to set up the data structures to evaluate functions and gradients, and notes that computing a Hessian by accumulating the Hessian of each $\widehat{f}_i$ can be much faster than by using straightforward reverse-mode AD on the DAG of f. One of the outcomes of the automatic detection of the partially-separable structure in AMPL is an interface to LANCELOT.[6]

3.6.5 Large-scale algorithms (LSA)

Later in the 2000's, Lukšan et al. [105] implemented several methods to solve "large-scale unconstrained and box constrained optimization and large-scale systems of nonlinear equations" [105]. In total, Lukšan et al. developed fourteen basic Fortran subroutines gathered in the library LSA. Among these fourteen subroutines, three line searches exploit the partially-separable structure of f: PSED, PSEC, and PSEN [105, Section 4] which are inspired of the works from Griewank and Toint [75, 76]. Both PSED and PSEC are counterparts of the VE08 and VE10 routines. The latter method, PSEN, is designed to solve partially-separable functions having nonsmooth element functions [104] and is summarized in Section 3.2.1. PSED and PSEN solve the linear system derived from the quadratic subproblem with a direct method while PSEC exploits an iterative method. The contribution of Lukšan et al. is completed with the description of 82 partially-separable problems [106] used to furnish

[6] See http://netlib.org/ampl/solvers/lancelot/index.html

numerical results.

Other pieces of software exploiting the partial separability exist. However, as they are not focused around partitioned quasi-Newton methods, they are described elsewhere through the present literature review. Notably:

- LSNNO [152], detailed in Section 3.4.1, is a Fortran routine based on the SIF format designed to solve large scale nonlinear network problems;
- ELSO [16], summarized in Section 3.5.2, specify how partial separability can efficiently compute partitioned derivatives using directional derivatives;
- BFO [124, 125], presented in Section 3.8, is a derivative-free pattern search implemented in Matlab. BFO exploits the partially-separable structure in both its search step and poll step;
- CMA-ES has a variant exploiting partial separability named P-sep lmm-CMA-ES [18], as detailed in Section 3.8. P-sep lmm-CMA-ES assembles small element meta-models of size n_i to break the dimensional curse inherent to an unstructured meta-model of size n. CMA-ES is available in several languages: C(++), Fortran, Java, Matlab, R, Scilab and Python, the latter having additional feature unavailable in other languages.

3.7 Parallelization of partially-separable methods

In addition of looking for optimization methods converging in lesser iterations, optimization scientists are always interested in finding ways to reduce the total time needed to find a satisfactory solution. While programming started mostly with sequential computation, the rise of efficient graphic cards and parallelized architecture made distributed computation one of the most active area of research in optimization. The most known straightforward and scalable problems exploiting parallelization are the (block-)separable problems, which are a special case of partial separability. Hence, this section develops methods where partial separability is exploited to leverage additional ways of parallelization, split between the synchronous and asynchronous methods. Furthermore, some insights on the known limitation such as: communication between processors, synchronization of computation or the task's granularity of the distributed process will be discussed.

3.7.1 Synchronous methods

Lescrenier [99] details how partial separability allows an inexact line search exploiting a partitioned quasi-Newton Hessian approximation to be parallelized on several processors.

Every iteration of the inexact line search consists mainly in four phases: the evaluation of the gradient $\nabla f(x_k)$, an inexact solution of $B_k s = -\nabla f(x_k)$, the step size α determined through $f(x_k + \alpha s)$ evaluations and the update of the partitioned matrix B_k. Each one of those methods involves either a partially-separable function or a partitioned data structure. Therefore, it can distribute computation over its element functions or element data structures. To measure how much scalable a parallel implementation is, Lescrenier defines the *efficiency* as the average time of a processor's activity during a given method, evaluating indirectly communication and sequential aggregation processes.

To distribute the computational effort of f and ∇f, each processor handles the evaluation of one or several $\widehat{f}_i$ or $\nabla \widehat{f}_i$. Once, $\widehat{f}_i$ and $\nabla \widehat{f}_i$ are calculated, the element contributions are aggregated sequentially, due to overlap between variables. As an inexact line search, $B_k s = -\nabla f(x_k)$ is inexactly solved by the truncated conjugate gradient method [84, 143]. During this phase, the partitioned matrix vector product $B_k v = \sum_{i=1}^{N} U_i^\top \widehat{B}_{i,k} U_i v$ is distributed across processors, each performing several element matrix vector products $\widehat{B}_{i,k} U_i v$, later aggregated using $U_i^\top$ to form $B_k v$. The limits of these three phases is the synchronization needed to aggregate the result of size n, making the method not fully parallel. Furthermore, the more processors there are, the more synchronization prevents each method's efficiency to reach 1. The numerical experiments consider up to five processors and display the efficiency measured for all the methods. The efficiencies of those three methods decrease for any new processor added, to reach between 0.55 and 0.6 when considering five processors.

The last method, the partitioned quasi-Newton update of B_k, distributes across processors every element update, which computes $\widehat{y}_{i,k}$ (3.18), $\widehat{s}_{i,k}$ and performs the quasi-Newton update onto $\widehat{B}_{i,k}$. The partitioned update is the method that best parallelizes out of the four by having no sequential aggregation and obtain an efficiency of 0.8 for 5 processors. Finally, when f is separable, the method becomes almost fully separable since the variables do not overlap.

Okoubi and Koko [116] define a parallel projected gradient (Nesterov) algorithm to solve smooth convex partially-separable problems. The algorithm combines two steps. The first one solves element subproblems separately while the second aggregates their solutions to update x. Each element problem is a quadratic approximation depending on the element gradient Lipschitz constant L_i. To define properly the problem solved at each iterate, Okoubi and Koko [116] divide $x \in \mathbb{R}^n$ into an N-vector $x_{[]} \coloneqq (x_{[1]}, x_{[2]}, ..., x_{[N]}) \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \times \cdots \times \mathbb{R}^{n_N}$, where $x_{[1]} \coloneqq U_1 x$. $x_{[]}$ represents the memory maintained in parallel during the optimization process. When f is not separable, $x_{[i]}$ may contain variables used by several other element

functions. Hence, suppose $1 \leq i \neq j \leq N$ such that $U_i U_j^\top \neq 0$, then $x_{[i]}$ can be seen as:

$$x_{[i]} = \begin{pmatrix} x_{[ii]} \\ x_{[ij]} \end{pmatrix}$$

where $x_{[ii]}$ refers to the pure variables of $\widehat{f}_i$ while $x_{[ij]}$ selects the shared variables between $\widehat{f}_i$ and $\widehat{f}_j$. From this formulation, the following problem is solved at each iteration:

$$\min_{x \in \mathbb{R}^n} f(x) = \sum_{i=1}^{N} \widehat{f}_i(x_{[i]}) \tag{3.34a}$$

$$x_{[ij]} - x_{[ji]} = 0, \forall i \in \{1, ..., N\},\ \forall j \in M_i, \tag{3.34b}$$

$$x_{[i]} \in \mathbb{R}^{n_i}, \forall \in \{1, ..., N\}, \tag{3.34c}$$

where $M_i \coloneqq \{j = 1, \ldots, N \mid U_i U_j^\top \neq 0\}$ is the set of element function indices sharing variables with $\widehat{f}_i$. By imposing the constraints $x_{[ij]} - x_{[ji]} = 0$, the variables shared by several element functions (i.e. scattered across many $x_{[i]}$) must be all equal. Okoubi and Koko [116] reformulated (3.34) as

$$\min_{x_{[]}} \sum_{i=1}^{N} \widehat{f}_i(x_{[i]}) + \sum_{i=1}^{N} g_i(x_{[i]}), \tag{3.35}$$

where $g_i : \mathbb{R}^{n_i} \to \mathbb{R}$ is the characteristic function of the set $K_i := \{x_{[]} \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \times \cdots \times \mathbb{R}^{n_N} \mid x_{[ij]} - x_{[ji]} = 0, \forall j \text{ such as } U_i U_j^\top \neq 0\}$. Concretely, $g_i(x_{[i]})$ values 0 if $x_{[]} \in K_i$ and values $+\infty$ otherwise.

To solve (3.35) considering the current point $y_{[]}$, every element $y_{[i]}$ solves the following subproblem :

$$\arg\min_{y_{[i]}} g_i(y_{[i]}) + \frac{L_i}{2} \|y_{[i]} - (x_{[i]} - L_i^{-1} \nabla \widehat{f}_i(x_{[i]}))\|^2,$$

leading to the element step $s_{[i]} = x_{[i]} - L_i^{-1} \nabla \widehat{f}_i(x_{[i]})$, where L_i is the Lipschitz's constant of $\nabla \widehat{f}_i$. Then, every element step is projected on K_i, which corresponds to aggregate the element steps $s_{[i]}$ as follows:

$$s^+_{[ii]} = s_{[ii]}, \quad s^+_{[ij]} = \frac{1}{|M_i| + 1} \left(\sum_{j \in M_i} \frac{1}{|M_i|} s_{[ij]} + s_{[ji]} \right).$$

The term $s^+_{[ij]}$ is the barycenter of all element contributions involved in $s_{[ij]},\ j \in M_i$. For example if $|M_i| = 1$ then $s^+_{[ij]} = \frac{s_{[ij]} + s_{[ji]}}{2}$, making $s^+_{[ij]} = s^+_{[ji]}$ and $U_i s^+ = s^+_{[i]}$. Before the next iteration starts, $x_{[i],k+1} = x_{[i],k+1} + \alpha_k s_{[i]}$ is updated considering $\alpha_k > \alpha_{k-1} > \cdots > \alpha_0 = 1$, which could be restarted when f fails to decrease. The numerical results on quadratic

partially-separable problems plebiscite the restart strategy, notably when the number of processors is small and the element-gradient Lipschitz constants are not clustered.

Note that if the architecture has less than N processors available, some element functions may be merged homogeneously together until their number matches the amount of processors available.

Another parallel algorithm originates from Kaya et al. [91] and results in HAMSI, which is designed to solve the matrix completion problem (3.13). This algorithm proposes to split f into groups of non-overlapping element functions $f^{(i)}$ and minimize iteratively each of them periodically. Therefore, the method first determines subset of element functions where none of the element functions overlap, for example:

$$\begin{aligned} f(x) &= \widehat{f}_1(x_1, x_2) + \widehat{f}_2(x_2, x_3) + \widehat{f}_3(x_3, x_4) + \widehat{f}_4(x_4, x_5), \\ &= f^{(1)}(x) + f^{(2)}(x), \\ f^{(1)}(x) &= \widehat{f}_1(x_1, x_2) + \widehat{f}_3(x_3, x_4), \\ f^{(2)}(x) &= \widehat{f}_2(x_2, x_3) + \widehat{f}_4(x_4, x_5). \end{aligned} \tag{3.36}$$

The decomposition of f returns $f^{(1)}$ and $f^{(2)}$ as (block) separable functions. Then, at each iteration, one $f^{(i)}$ is periodically solved using a block-separable optimization method. To solve each subproblem, Kaya et al. [91] build a quadratic approximation of $f^{(i)}$. However, contrary to the Section 3.2, no $\nabla^2 \widehat{f}_j$ is approximated individually. Instead, a block-LBFGS approximation H of $\nabla^2 f^{-1}(x_k)$ is built iteratively, which provides an efficient operator-vector product with sparse vectors or sparse matrices [23]. Note that the pair s, y updating H comes from different $f^{(i)}$ evaluated successively. H is then employed for approximating the Hessian of one element function subset $f^{(i)}$. As $f^{(i)}$ is block-separable, Kaya et al. propose to approximate each block of the Hessian with $\widehat{H}_j = U_j H U_j^\top \approx \nabla^2 \widehat{f}_j^{-1}$, before solving independently the resulting element quadratic problems of smaller size. While this method strongly differ with the content of the Section 3.2, this is no surprise that the method works as the subproblem solution is a descent direction.

In order to scale uniformly the minimization of non-overlapping element function subsets, the computational effort must be distributed fairly among the processors. To do so, Kaya et al. put an intense effort to find balanced non-overlapping element-function subsets. Thus, the size of the independent variable subsets found after a proper colouring of the bipartite graph must be homogeneous, e.g., f from (3.36) is decomposed as $f^{(1)}$ and $f^{(2)}$ both having two element functions (of identical size). After testing some colouring, [91] reports STRATA-B as the best grouping as it achieves nearly a linear speed-up depending on the number of threads. In practice, the difference between STRATA, the naive implementation, and STRATA-B becomes

larger as the problem dimension grows. For the largest instances tested, STRATA-B is three times faster than STRATA. Additionally, HAMSI is compared to a handmade minibatch gradient descent (mb-GD) method, which is outperformed, whatever the grouping strategy chose.

Toward asynchronous methods

Richtárik and Takáč [136] proposed a partitioned coordinate descent method (PCDM) to parallelize the minimization of partially-separable problems. This method is designed to solve

$$\min_{x\in\mathbb{R}^n} f(x)+\psi(x), \quad f(x)=\sum_{i=1}^{N} f_i(x),$$

where ψ is a block separable regularizer, e.g. the $L1$ norm, and f_i depends on a subset of variables. Both f and ψ are convex. Additionally, the decision variables are divided into distinct blocks:

$$x=(x^{(1)},x^{(2)},\ldots,x^{(j_{max})})\in\mathbb{R}^n, \quad x^{(j)}=U_j^\top x,\ U_j^\top\in\mathbb{R}^{n_j\times n},$$

Observe that $U=[U_1...U_{j_{max}}]$ is a permutation of I. Consequently, $U_{j_1}^\top U_{j_2}=0,\ \forall j_1\neq j_2$ and $\sum_{j=1}^{N} n_j=n$. Therefore, every element function depends on variables selected from several U_j. The number of U_j each element function depends on is referred as the *separability degree*. This definition differs with U_i as described by (3.3), which selects the (linear combination of) variables parametrizing $\widehat{f}_i$.

PCDM is a method supporting proximal operators and utilizing randomized blocks of variables to parallelize computation. At every iteration, a subset $S_k\subseteq\{1,\ldots,j_{max}\}$ is randomly pulled. Hence, it will iteratively update only $x^{(j)},\forall j\in S_k$ from the function $h(x)$:

$$h(x)=\arg\min_{h\in\mathbb{R}^n} f(x)+\nabla f(x)^\top h+\tfrac{\beta}{2}\|h\|_w^2+\psi(x+h),$$

where $\beta>0$ is fixed depending on the separability degree of f and $w\in\mathbb{R}^{n_{j_{max}}}$ is chosen from the block Lipschitz's constants $L=[L_1,\ldots,L_{j_{max}}]$ defined as:

$$\|U_j^\top\left(\nabla f(x+U_jt)-\nabla f(x)\right)\|_{(j)}\leq L_j\|t\|_{(j)}, \quad \forall t\in\mathbb{R}^{n_j},$$

where

$$\|v\|_{(j)}=v^\top A_j v, \quad v\in\mathbb{R}^{n_j}, \quad A_j\in\mathbb{R}^{n_j\times n_j}\succ 0,$$

making the function $h(x)$ minimizing a block separable problem. Therefore, $h(x)$ can be

distributed across the block variables selected by S_k safely. Each variable block gets

$$h(x)^{(j)} = \arg\min_{h^{(j)} \in \mathbb{R}^{n_j}} \nabla f(x)^\top U_j h^{(j)} + \frac{\beta w_j}{2} \|h^{(j)}\|^2_{(j)} + \psi(x^{(j)} + h^{(j)}).$$

Later on, Fercoq and Richtárik [56, 57] proposed a distributed block coordinate descent optimization method APPROX (Accelerated, Parallel, and PROXimal) as an upgrade of PDCM. Specifically, [56] integrates an accelerated variant to achieve the $O(1/k^2)$ convergence result that Nesterov [113] originally obtained. Thereafter, Fercoq and Richtárik [57] propose an asynchronous implementation of APPROX, which is reported as 5 times faster than the initial implementation from [56]. In addition, it introduces new step sizes and different samplings of S_k. Finally, Mareček et al. [108] refined specific implementations for two particular problems: solving sparse least squares and training support vector machines for which they report promising results.

Despite that neither PCDM nor APPROX minimize f by exploiting its partial separability, the notion of partially-separable degree permits a bound on the number of iterations performed. That bound is reached when the number of (block) variables matches the number of processors available.

3.7.2 Asynchronous methods

Synchronization and communication processes prevent synchronous methods to be completely parallel. Hence, asynchronous methods, e.g. Fercoq and Richtárik [57], seem to be the key to get fully parallel methods.

In order to minimize a partially-separable function in parallel, a natural idea is to divide element functions into subsets which are minimized independently. However, having variables appearing in several subsets prevent generally any algorithm to reach a minimizer only by minimizing individually these subsets. A necessary condition is then to have subsets of element-functions depending on distinct variables, i.e., which do not interfere with any variable of other subsets of element-functions. Therefore, to minimize $x_i \in \mathbb{R}$, every element function depending on x_i must be identified:

$$E_i \coloneqq \{j \in \{1, \ldots, N\} \ \mid \ U_j e_i \neq 0\}.$$

Then, every variable on which element function from E_i depends on must be locked:

$$O_i \coloneqq \{j \in \{1, \ldots, n\} \ \mid \ \exists\, l \in E_i \ : \ U_l e_j \neq 0\}.$$

Hence, an independent subproblem minimizing x_i appears by locking $|O_i|$ variables:

$$\min_{x_i \in \mathbb{R}} \bar{f}(x_i) = \sum_{i \in E_i} \bar{f}_i(x_i), \quad \bar{f}_i(x_i) \coloneqq \widehat{f}_i(U_i \bar{x}),$$

where

$$\bar{x} \in \mathbb{R}^n, \quad \bar{x}_i = x_i \text{ and } \bar{x}_j \text{ remain fixed } \forall j \in O_i \backslash \{i\}.$$

The same scheme can be extended to minimize all $x_i,\ i \in S \subseteq \{1, \ldots, n\}$ a subset of variable indices. In this case:

$$E_S \coloneqq \bigcup_{i \in S} E_i, \quad O_S \coloneqq \bigcup_{i \in S} O_i,$$

creating the subproblem:

$$\min_{x_S \in \mathbb{R}^{|S|}} \bar{f}(x_S) = \sum_{i \in E_S} \bar{f}_i(x_S), \quad \bar{f}_i(x_S) \coloneqq \widehat{f}_i(U_i \bar{x}),$$

where

$$\bar{x} \in \mathbb{R}^n, \quad \bar{x}_i = x_i,\ \forall i \in S \text{ and } \bar{x}_j \text{ remain fixed } \forall j \in O_S \backslash S.$$

Note that $\bar{x}$ is in $\mathbb{R}^n$ to simplify the equations by using U_i, but $\bar{x}$ could be in $\mathbb{R}^{|O_i|}$ or $\mathbb{R}^{|O_S|}$.

Fischer and Helmberg [58] use this procedure within a bundle method minimizing a subset of variables at each iterate. The algorithm is originally designed for any unstructured problem. But, an unstructured problem induces a subproblem in $\mathbb{R}^n$ which requires every variable to be locked, dashing any hope of parallelization. Partial separability reduces the subproblem to a subset of variables O_S and element functions E_S. Hence, by defining orthogonal variable subsets which can be minimized independently, the method becomes parallel in a straightforward way. One process selects S such that O_S is free, lock O_S, minimizes the subproblem of dimension $|S|$ from E_S, updates x_S and frees the variables O_S. After the proper selection of S_1 from the first process, a second process can start selecting S_2 such that O_{S_2} does not overlap on O_{S_1}, and so forth. The method minimizes successive independent subproblems until it converges. The use of independent variable subsets avoid complete data synchronization, making the method asynchronous. Fischer and Helmberg apply the algorithm to the Lagrangian relaxation of a train timetabling problem in time expanded networks. The speed-up obtained is reported to depend on the interdependencies between constraints. In particular, long-distance trains, interacting with a lot of other trains and stations, decreases the speed-up obtained.

Lastly, Cannelli et al. [27] presented an asynchronous and distributed algorithm for optimizing multi-agent systems by exploiting their partial separability. In a multi-agent system, the objective is related to a network having agents a_i, $1 \leq i \leq N$ whose inner-variables are

selected by U_{a_i}, i.e. $U_{a_i}U_{a_j}^\top = 0$, $\forall 1 \leq i \neq j \leq N$. The objective sums agent objectives, each of which depends on the variables from the agent and its neighbours, gathered in $\mathfrak{N}_i$ (note that $a_i \in \mathfrak{N}_i$):

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^{N} h_i(U_{a_i}x) + g_i(\bigcup_{j\in\mathfrak{N}_i} U_{a_j}x), \tag{3.37}$$

considering $\bigcup_{j\in\mathfrak{N}_i} U_{a_j} = \begin{bmatrix} U_{a_{j_1}}^\top & U_{a_{j_2}}^\top & \dots & U_{a_{j_{\max}}}^\top \end{bmatrix}^\top$, where $a_{j_1}, a_{j_2}, \dots, a_{j_{\max}} \in \mathfrak{N}_i$. Hence, $\bigcup_{j\in\mathfrak{N}_i} U_{a_j}x$ selects a subset of variables, depending on the network sparsity. Problems formulated as (3.37) appear notably in: network utility maximization and resource allocation problems, state estimation in power network, cooperative localization in wireless networks, map building in robotic networks and machine learning [27]. The formulation (3.37) leads to:

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^{N} \widehat{f}_i(\bigcup_{j\in\mathfrak{N}_i} U_{a_j}x), \quad \widehat{f}_i(\bigcup_{j\in\mathfrak{N}_i} U_{a_j}x) \coloneqq h(U_{a_i}) + g(\bigcup_{j\in\mathfrak{N}_i} U_{a_j}x).$$

The optimization of each agent is dispatched to a single process minimizing $U_{a_i}x$. Each agent solves a convex approximation of $\widehat{f}_i$ considering possibly outdated values for $U_{a_j}x$, $j \in \mathfrak{N}_i\backslash\{a_i\}$ [27, eq. 2-3]. The newer values of $U_{a_i}x$ are transmitted to the neighbour's agents $a_j, j \in \mathfrak{N}_i\backslash\{a_i\}$ periodically. Cannelli et al. [27] propose two protocols:

- protocol (a) updates every neighbour after any change in $U_{a_i}x$;
- protocol (b) updates each neighbour every $|\mathfrak{N}_i|$ agent iterations (cyclic).

The asynchronous algorithm is proved to converge sublinearly with a near linear speed-up according to the number of agents. Also, it is reported to reduce the time lost in communication processes and in synchronization compared to the synchronous variant. Moreover, a beneficial side effect is the algorithm's robustness about packet losses.

Numerical results compare the two communication protocols on distributed matrix completion problem (3.13). Protocol (b) is demonstrated to perform better than protocol (a) in the light of communication exchanges, which is the bottleneck for parallel methods.

3.8 Derivative-free methods

The exploitation of partial separability is not restricted to continuous optimization only. This section intends to provide some insight on how derivative free optimization exploits the partially-separable structure to evaluate f, or approximate ∇f or $\nabla^2 f$ using: finite difference, Newton polynomial or other interpolation methods. Exploiting the structure can drastically

reduce both the amount of f evaluations and/or the storage usually required. Lastly, we summarize a genetic algorithm employing a partitioned crossover operator derived from the partially-separable structure of f.

3.8.1 Quadratic interpolation exploiting partial separability

Colson and Toint [31] minimize a partially-separable function (3.1) with a quadratic trust-region method which does not access derivatives. Hence, at each iterate, the trust-region subproblem aggregates all element-function quadratic interpolations:

$$\begin{array}{rl} \min_{s\in\mathbb{R}^n} & m_k(s) = \sum_{i=1}^{N} m_{i,k}(s) = \sum_{i=1}^{N} \widehat{m}_{i,k}(\widehat{s}_i) \\ \text{s.t.} & \|s\| \leq \Delta_k \end{array}, \tag{3.38}$$

where $\widehat{m}_{i,k}(\widehat{s}_i)$ is an interpolating quadratic polynomial of $\widehat{f}_i(\widehat{x}_{i,k} + \widehat{s}_i)$ such as:

$$\widehat{m}_{i,k}(\widehat{s}_i) := \widehat{f}_i(\widehat{x}_{i,k}) + \widehat{g}_{i,k}^{\top}\widehat{s}_i + \frac{1}{2}\widehat{s}_i^{\top}\widehat{B}_{i,k}\widehat{s}_i \ : \mathbb{R}^{n_i} \to \mathbb{R},$$

considering that both $\widehat{g}_{i,k}$ and $\widehat{B}_{i,k}$ approximate respectively $\nabla\widehat{f}_i(\widehat{x}_{i,k})$ and $\nabla^2\widehat{f}_i(\widehat{x}_{i,k})$.

Each $\widehat{m}_{i,k}$ is built from a subset of points that the i-th element function visited Y_i and their associated values $\{\widehat{f}_i(\widehat{y}_j)\}_{\widehat{y}_j\in Y_i}$, with $|Y_i| \leq p_i = n_i + \frac{1}{2}n_i(n_i+1)$. To ensure a unique interpolating quadratic polynomial, i.e., a unique pair $\widehat{g}_i, \widehat{B}_i$, the points $\widehat{y} \in Y_i$ must verify a geometric and practical property known as *poisedness*. Poisedness ensures that points of Y_i "do not collapse in a lower dimensional space or do not lie on the same quadratic curve from which an infinity of interpolating quadratic polynomial may be found" [31]. Nevertheless, if $|Y_i| < p_i$, making Y_i not poised, at least one polynomial interpolation exists for which $\widehat{m}_{i,k}(\widehat{y}) = \widehat{f}_i(\widehat{y})$, $\forall\widehat{y} \in Y_i$. The poisedness of Y_i can be improved by solving two trust-region sub-problems, as [31, Section. 2] points out using the method introduced in [39, Chapter 7]. Colson and Toint [31] used Newton fundamental polynomials [38] to interpolate $\widehat{g}_i, \widehat{B}_i$ from Y_i. In that case, poisedness is theoretically ensured as long as nonnormalized Newton fundamental polynomials and their interpolation points are different from zero.

The trust-region method starts from an initial point x_0, conditioning each element quadratic interpolation to $Y_i = \{\widehat{x}_{i,0}\}$. Then, at every new iterations, if $|Y_i| < p_i$ and $\widehat{x}_{i,k+1}$ does not break poisedness, then $\widehat{x}_{i,k+1}$ is simply added to Y_i. When $|Y_i| = p_i$, the updating process may replace one $\widehat{y}$ of Y_i by $\widehat{x}_{i,k+1}$ if the resulting Y_i is best poised or if $\widehat{x}_{i,k+1}$ is closer to the trust-region than $\widehat{y}$. Unlike partitioned quasi-Newton updates, every Y_i can capitalize on $\widehat{f}_i(\widehat{x}_{i,k} + \widehat{s}_{i,k})$ to improve the poisedness of Y_i whether s coming from (3.38) is successful or

not. It comes handy for managing a trust-region, where keeping a proper geometry becomes difficult as Δ_k decreases. In particular, the bad geometries of Y_i may result in an unsuccessful s. Hence, the poisedness of Y_i must be enhanced every time Δ_k decreases, by computing new points if necessary. If a new best point is found by accident during the management of some Y_i, then the method continues from this point. Finally, to terminate/converge in a derivative-free context, the trust-region relies on $\|g_k\| \leq \epsilon,\ \epsilon > 0$ completed with:

$$f(x) - m_k(x) \leq \kappa\Delta^2,\ \forall x \text{ such that } \|x - x_k\| \leq \Delta,\ \kappa > 0,$$

which guarantees the validity of the quadratic model in the local ball, as described [39, p.308].

Colson and Toint [31] develop two approaches to update Y_i depending on the assumptions:

- assumption I: each element function can be evaluated separately;
- assumption G: all element functions are evaluated simultaneously, i.e., $\{\widehat{f}_i(\widehat{x}_{i,k})\}_{i=1}^N$ at once.

In the case of assumption I, the management of Y_i is straightforward since the poisedness of each element can be managed independently by evaluations of $\widehat{f}_i$. Conversely, for the assumption G, each $\{\widehat{f}_i(U_i x)\}_{i=1}^N$ evaluation must try to poise as many Y_i as possible. Therefore, to make every $\{\widehat{f}_i(U_i x)\}_{i=1}^N$ profitable, Colson and Toint seek to identify element-function subsets without overlapping element variables. The idea to poise a many Y_i geometry within the minimal amount of $\{\widehat{f}_i(U_i x)\}_{i=1}^N$ evaluations is inspired from Curtis et al. [40] and defines:

$$f_0(x) = \begin{pmatrix} \widehat{f}_{l_1}(U_{l_1}x) \\ \widehat{f}_{l_2}(U_{l_2}x) \\ \vdots \\ \widehat{f}_{l_{\#L}}(U_{l_{\#L}}x) \end{pmatrix}, \quad \nabla f_0(x) = \begin{pmatrix} \nabla\widehat{f}_{l_1}(U_{l_1}x)^\top \\ \nabla\widehat{f}_{l_2}(U_{l_2}x)^\top \\ \vdots \\ \nabla\widehat{f}_{l_{\#L}}(U_{l_{\#L}}x)^\top \end{pmatrix},$$

where $L = \{l_1, l_2, \ldots, l_{\#L}\} \subseteq \{1, 2, \ldots, N\}$ informs the set of element function interpolations requiring a geometry improvement. A proper row coloring of $\nabla f_0(x)$ defines orthogonal columns. Each color describes a subset $\mathcal{G} \subseteq L$ of non-overlapping element functions such that $\forall i, j \in \mathcal{G}^2,\ i \neq j :\ U_i U_j^\top = 0$. Such a group of non-overlapping element functions allows to improve the poisedness of all $Y_i,\ i \in \mathcal{G}$ with a single evaluation of $\{\widehat{f}_i(U_i x)\}_{i=1}^N$. A similar scheme is exploited in automatic differentiation to accelerate forward automatic differentiation [12], see Section 3.5.2 for more details about the coloring.

Both methods I and G outperform the unstructured methods in terms of function evaluations (especially the method I). Moreover, the method I approximates $\nabla^2 f(x_k)$ more accurately

than method G and seems to require an amount of f evaluations depending on the overlapping between element functions instead of n. The work of [31] was first implemented in Fortran with PSDFO, and renew 20 years later in Matlab with the Brute Force Optimizer (BFO) solver [125].

Similarly to Colson and Toint [31], CMA-ES [82], a population based stochastic search rank-based algorithm, integrates a variant exploiting the partial separability of f when the structure is known. To enhance the speed convergence of an unstructured function f, a first variant incorporates a metamodel of f: lmm-CMA-ES [17]. We focus on a lmm-CMA-ES's variant, named P-SEP lmm-CMA-ES Bouzarkouna et al. [18], which integrates an idea similar to method I [31] to form a partitioned metamodel where each element function possesses a metamodel.

At each iteration (or generation) of the standard CMA-ES algorithm, a new population of λ points is sampled. These points are then ranked depending on how close they are from the current optimum. The new population is generated by summing the current estimate of the optimum and random vectors —with independent multivariate normal distributions, considering a zero mean vector and a covariance matrix— coupled to an adaptive step length. Finally, the estimate of the optimum is iteratively updated with the weighted mean of the best individuals, the step length and the covariance matrix.

Instead of using only the default generation, lmm-CMA-ES exploits the archive points, named *training set*, as well as their objective function evaluations to build an unstructured quadratic metamodel. For any point $q \in \mathbb{R}^n$, a metamodel $\bar{f}(q, b)$ seeks to approximate $f(q)$:

$$\bar{f}(x, b) = b^\top (x_1^2, \ldots, x_n^2, x_1 x_2, \ldots, x_{n-1} x_n, x_1, \ldots, x_n, 1), \quad b \in \mathbb{R}^{\left(\frac{n(n+3)}{2}+1\right)}, \tag{3.39}$$

from the k nearest points of q within the training set. The vector b is deduced by minimizing a subproblem close to a nonlinear least square problem, integrating the Mahalanobis distance (related to the covariance matrix) between q and the considered points of the training set. Take note that every time q changes or a new point enters the training set then b must be redefined. To construct a quadratic metamodel, the training set must contain at least $k_{min} = \frac{n(n+3)}{2} + 1$ points, while using $k = 2k_{min}$ points is recommended. By default, the standard CMA-ES strategy is used until k_{min} points are accumulated.

Back to the outer iteration, the metamodel evaluates every point from the new population. As a result, every point gets a value from $\bar{f}$ leading to an approximate ranking without evaluating f. Hence, only the best ranked points are truly evaluated, helping in the refinement of new metamodels to best approximate the remaining population points. If the ensuing ranking

(based on f) remains mostly the same, then it prevents the rest of the population to be evaluated. Conversely, when the ranking among the best points changes, then the next best points are evaluated until no significant change is observed. Additionally, the initial amount of points truly evaluated changes over iterations depending on the number of inner iterations previously required before breaking the inner loop. If few iterations were needed, the amount of points that must be evaluated decreases, otherwise it stagnates or increases. Also, note that any f evaluation enters the training set.

Contrary to [75–77], Bouzarkouna et al. [18] define a partially-separable function as:

$$f(x) = \sum_{i=1}^{N} \widehat{f}_i(\phi_i(x)),$$

where $\phi_i(x) : \mathbb{R}^n \to \mathbb{R}^{n_i}$ could be a nonlinear mapping. Bouzarkouna et al. motivations for ϕ_i to exist are the need of distances, between (petrol) well coordinates (the variables), which involves nonlinearity. Beside the difference between ϕ_i and U_i, the idea stays the same. P-SEP lmm-CMA-ES adapts $\bar{f}$ (3.39) to accumulate element metamodels:

$$\bar{f}(x) = \sum_{i=1}^{N} \bar{\widehat{f}}_i(\phi_i(x), b_i),$$

where $\bar{\widehat{f}}_i(\phi_i(x), b_i)$ replicates to $\widehat{f}_i$ what (3.39) does to f. The element covariance matrix, mandatory for computing b_i, is built iteratively from the evaluations of $\widehat{f}_i(\phi_i(x))$ or from the approximations $\bar{\widehat{f}}_i(\phi_i(x), b_i)$ if not all best individuals are evaluated.

The numerical results of P-sep lmm-CMA-ES [18] show a speed-up factor from 4.5 up to 15 between the standard CMA-ES and P-SEP lmm-CMA-ES for a Rosenbrock's function. On the same problems, the unstructured variant lmm-CMA-ES obtains a speed-up factor of 3. Furthermore, the bigger the instance is, the bigger the speed-up from P-sep lmm-CMA-ES is, unlike lmm-CMA-ES whose speed-up decreases as the dimension increases. Another experiment reveals that changing $n_i = 2$ to $n_i = 4$ for all element functions reduces the speedup factor from 8.6 to 2.2. Moreover, the larger the population is, the fewer outer-iterations are needed. Usually, iterations are reduced by a factor 1.5 to 2. Additionally, in particular cases, P-sep lmm-CMA-ES achieves the maximal theoretical speed-up, which is equal to the population's size. Bouzarkouna et al. conclude by stating that a partitioned metamodel breaks the curse of dimensionality related to meta-models (k being too big) for large partially-separable problems when every n_i remains small enough.

3.8.2 Pattern searches

Price and Toint [129] describe a pattern search exploiting the partial separability to reduce the computational effort put during the poll step. At each iteration of a pattern search, the poll step considers a grid of several points (partly aligned on the euclidean axis), all of which are evaluated by the objective function. Thereafter, the pattern search selects the new best solution if there is any and starts a new grid. If no such solution exists, the current solution remains, and the grid length shrinks. To be more flexible, the grid's size considered may differ depending on the directions considered, but further equations will not mention it for sake of simplicity. The pattern search converges when no evaluation of the grid points yields a decrease and that the step length is smaller than the tolerance set prior to the execution.

The pattern search exploits partial separability to reduce as much as possible the number of element function $\widehat{f}_i$ calls needed to evaluate the grid points. Indeed, the evaluation of f can be seen as the evaluation of its N element functions, for which we suppose a unitary cost. Thus, a grid evaluation without integrating partial separability necessitates $|V|N$ element function calls, where V is the set of directions considered.

The main idea of Price and Toint [129] is to tailor the direction set V according to the partial separability to reduce the element function calls needed to evaluate the grid. The set V is formed by the euclidean axes $\{e_1, \ldots, e_n\}$ to which are added $e_{n+j} = -\left(\sum_{i \in S_j} e_i\right),\ 1 \leq j \leq p$, where each S_j is defined after analyzing the partial-separability of f. Each S_j represents a subset of *non-interacting* variables, i.e., appearing exactly on the same element functions. For example, consider:

$$f(x) = \widehat{f}_1(x_1, x_2, x_3) + \widehat{f}_2(x_1, x_2, x_4, x_5) + \widehat{f}_3(x_4, x_5), \tag{3.40}$$

then

$$\begin{array}{ll} x_1 \text{ appears in} & \widehat{f}_1,\ \widehat{f}_2, \\ x_2 \text{ appears in} & \widehat{f}_1,\ \widehat{f}_2, \\ x_3 \text{ appears in} & \widehat{f}_1, \\ x_4 \text{ appears in} & \widehat{f}_2,\ \widehat{f}_3, \\ x_5 \text{ appears in} & \widehat{f}_2,\ \widehat{f}_3, \end{array} \tag{3.41}$$

resulting in $S_1 := \{1, 2\}, S_2 := \{3\}$ and $S_3 := \{4, 5\}$. Hence, for f such as (3.40):

$$V = \{e_1, e_2, e_3, e_4, e_5, -(e_1 + e_2), -e_3, -(e_4 + e_5)\}.$$

Structurally, every e_i has its index i covered by one of the subspace S_j. Therefore, each S_j can be held responsible to evaluate $f(x + v)$ where $v \in \{e_i | \, i \in S_j\} \cup \{-\sum_{i \in S_j} e_i\}$ by making

the computation related to its corresponding element functions. Note that $v_i^\top v_j = 0$ for any $v_i \in S_i$ and $v_j \in S_j \neq S_i$. In the case of (3.40), S_1 and S_3 require two element functions while S_2 depends only on $\widehat{f}_1$, so that every $f(x+v)$, $v \in V \backslash \{e_3, -e_3\}$ requires two element function evaluations instead of three (3.41). Therefore, the grid evaluation, i.e. V, is done by 14 element-function evaluations while it would require 24 element-function evaluations with an unstructured procedure.

Contrary to the default unstructured search, the decreasing points resulting from the evaluations of $v \in S_j, 1 \leq j \leq p$ must be aggregated. Hence, Price and Toint propose a greedy procedure aggregating all sub-steps $s_j \in \mathbb{R}_n$, to form s. The greedy iterative algorithm starts by selecting $\bar{s}_0$ from $\bar{S}_0 \subseteq \bigcup_{j \in \{1,\ldots,p\}} S_j$, where every $v \in \bar{S}_0$ brings a decrease to the sum of element functions affiliated to its corresponding S_j. Then $\bar{S}_0$ is updated to $\bar{S}_1$ by cutting off $\bar{s}_0$, as well as all the others sub-steps whose their decrease would be invalidated by accepting the change of $\bar{s}_0$. For example, in (3.41), any solution of S_2 would interfere with any solution of S_1, since x_3 is used by both $\widehat{f}_1$ and $\widehat{f}_2$. The greedy algorithm loops over $\bar{S}_l$ until $\bar{S}_l$ becomes empty.

As a result, the step s, which sums the successive $\bar{s}_l$ may be sparse, since the solution of some S_j are likely to not be used. However, each $\bar{s}_l$ guarantees to bring a decrease for its affiliated element functions, making s summing several decreases despite the sparse update. Several strategies to choose which $\bar{s}_l$ must be taken next exist, e.g., choosing $v \in \bar{S}_l$ which brings the biggest decrease. But regardless of the strategy used, the overall scheme that ensures successive decreases remains.

The more dramatic improvement resulting from the partitioned poll step appears when f is *totally separable* $f = \sum_{i=1}^{n} \widehat{f}_i(x_i)$ $(N = n)$. In such case, evaluating $f(x + e_i), \forall\, 1 \leq i \leq n$ without exploiting the structure would require n^2 element-function calls while it requires only $2n$ element function calls when the structure is exploited, i.e., $S_j = \{i\},\ 1 \leq j \leq n$.

Brute Force Optimizer (BFO), developed by Porcelli and Toint [124] is a direct-search derivative free Matlab optimizer for bound constrained problems, who is supporting both continuous and discrete variables. Porcelli and Toint [125] extends [124] in a gray-box context where variables are only continuous. The extension exploits the *coordinate partial separability* of a problem, i.e., knowing that $\widehat{f}_i$ exist as well as their respective U_i^E. The extension considers that each $\widehat{f}_i$ may be evaluated separately, but $\nabla \widehat{f}_i$ remains inaccessible. Porcelli and Toint [125] combine the ideas from Colson and Toint [31] and Price and Toint [129] to take advantage of partial separability during both the poll step and the search step.

Similarly to Price and Toint [129], the partially-separable structure is preprocessed to determine subspaces of independent (or non-interacting) variables and the element functions they depend

on. However, the employment of those subspaces during the poll step differs from [129]. In particular, the positive basis giving the poll directions is not fixed along the iterates. Also, rather than aggregating sub-steps, and potentially neglecting some others, the poll step generates structured orthogonal directions in $\mathbb{R}^n$ from the independent variables. That way, each orthogonal direction can combine several euclidean basis vectors while minimizing the number of element function calls. For flexibility, each independent variable subspace considers its own stepsize, which is iteratively updated depending on its local achievements.

The first phase of the poll step is achieved when every stepsize is beneath the threshold set in advance. It corresponds to the fact that no further improvement can be obtained from every independent variable subspace. Nevertheless, the random poll directions remain heavily structured by independent variables, making an improvement of f still possible. Therefore, to guarantee convergence, a second unstructured poll step is performed. To avoid n evaluations of f as a typical unstructured poll step would operate, Porcelli and Toint propose to evaluate f on a smaller set of orthogonal directions designed to break the hidden structure of the first poll step. Once the convergence is disproved by the second poll step, the first poll step starts again from the new best point.

In addition of the poll step, the search step can exploit partial separability to build partitioned interpolation models similarly to Colson and Toint [31]. Those models may be linear or quadratic, considering either a diagonal Hessian or a full Hessian approximation. Each one of those variants requires respectfully: $n_i + 1, 2n_i + 1$ and $\frac{(n_i+1)(n_i+2)}{2}$ points for each element function $\widehat{f}_i$. In order to manage the bound constraints, the interpolated model in embedded in a trust-region method.

The numerical results indicate that partial separability is best exploited in the poll step rather than in the search step. The computational gain from the poll step using partial separability outperforms an unstructured pattern search method, especially when the problem's size grows. Moreover, the partitioned polling step seems to converge with a number of f evaluations independent of n. The second phase of the poll step and the breakage of the hidden structure are illustrated graphically [125, Section 4], but it does not delay the global convergence by an order of magnitude. However, the benefit of the partitioned quadratic approximations used in the search step is unclear. The performance and data profiles inform that the search step variant performs slightly better on small problems, but this gain vanishes for medium and large problems. Nonetheless, for few specific problems [125, Table 1.3], the partitioned search step can significantly reduce the number of f evaluations. In addition, a partitioned quadratic approximation is computationally intensive, which prevents running such method for most large problems [125, Table 1.3]. Overall, Porcelli and Toint explain that the choice

of using a partitioned search step is problem specific and preconize to test with or without it for each problem.

3.8.3 Partitioned crossover operator

In [55], Durand and Alliot seek to solve an air traffic conflict problem, whose formulation happens to be partially-separable, with a genetic algorithm. A typical genetic algorithm infants a new solution x^{child} from two valid solutions $x^{parent1}$ and $x^{parent2}$ by the means of a crossover operator. Hence, Durand and Alliot [55] propose a dedicated partitioned crossover operator based on the partial separability of f. The specific crossover operator is based upon a local fitness function:

$$G_i(x) = \sum_{j \in Q_i} \frac{f_j(\widehat{x}_j)}{n_j}, \quad f_j : \mathbb{R}^{n_j} \to \mathbb{R},$$

where $Q_i \coloneqq \{j \in \{1, \ldots, N\} \mid U_j e_i \neq 0\}$ is the set of element functions parametrized by x_i. The adapted crossover operator compares the local fitness of each component of the parents to create x^{child}. The rules are as follows:

- if $G_i(x^{parent1}) < G_i(x^{parent2}) - \Delta$ then $x_i^{child} = x_i^{parent1}$;
- else if $G_i(x^{parent1}) > G_i(x^{parent2}) + \Delta$ then $x_i^{child} = x_i^{parent2}$;
- otherwise, $|G_i(x^{parent1}) - G_i(x^{parent2})| \leq \Delta$, then, choose randomly between $x_i^{parent1}$ and $x_i^{parent2}$ or make a linear combination of both if x_i is continuous;

considering $\Delta > 0$. A statistical study compares the children's fitness produced by either an unstructured crossover operator or a partitioned crossover operator on a bit string function. [55, Figures 1 and 2] show the superiority of the partitioned operator over the unstructured operator, since the former has significantly higher probabilities to produce better (fitted) children. Moreover, the partitioned operator finds more global optima, and finds them faster than the unstructured operator [55, Figure 5].

3.9 Critical analysis

This chapter assembles most of the literature about partially-separable functions. As this section comes to an end, it is time to draw some conclusions about the exploitation of the partial separability for continuous optimization.

In general, optimization methods can take advantage of partial separability when it exists. In this chapter, every method exploiting the partially-separable structure outperforms its

unstructured counterpart. This is particularly true in the case of partitioned quasi-Newton methods over limited-memory quasi-Newton methods. The dominance of partitioned quasi-Newton methods can mostly be explained by the replication of the Hessian's sparsity, unlike limited-memory quasi-Newton methods.

Therefore, as long as the Hessian is sparse and that the storage of its sparse matrix is affordable, then partitioned quasi-Newton methods are applicable, since they have a similar memory storage requirement. Also, the best application cases of partial separability occur when f is (block) separable. Without overlapping, storing element Hessian approximations is as costly as storing the sparse Hessian matrix. Moreover, in such a case, a factorization can be performed without fill-in and straightforward asynchronous parallel methods can be implemented.

Partitioned quasi-Newton methods start to face issues when the element function dimensions are large. In such a case, storing the sparse matrix of the Hessian or its partitioned quasi-Newton approximation usually becomes unaffordable. Additionally, given a constant amount of element functions, the larger element function dimensions are, the more element function overlaps are likely to occur; in general, the methods exploiting partial separability tend to be more efficient when the overlapping between element functions is small. To answer this issue, the Section 3.3.3 provides partial answers, e.g., merge the overlapping element functions may reduce the memory storage and the computational effort required. However, in doing so, the partitioned quasi-Newton methods approximate less element function Hessians, making the partitioned Hessian approximation less sparse than the real Hessian, and it reduces the rank of the partitioned quasi-Newton update as well. Finally, the conjugate gradient method emerges as the most suited method for solving a partitioned linear system, since it takes advantage of the faster evaluation of the partitioned-matrix-vector and its potential parallel evaluation. Unfortunately, it does not exploit the partial separability for generating the conjugate directions, and it requires synchronization at each iteration.

The Chapter 4 addresses the issue related to the large element function dimensions. Its contribution is applied later in Chapter 5 for training deep neural networks by minimizing a partially-separable function with large element functions. The remaining issue that is the replacement of the conjugate gradient for solving the quadratic subproblem at each iteration of a quasi-Newton method is discussed in Chapter 7 and in Appendix B.

CHAPTER 4 Limited memory variant of partitioned quasi-Newton methods

This chapter presents a solution to the issue that partitioned quasi-Newton methods face when the dimension of an element function is large. Such a situation can occur for the matrix completion problem (3.13) when r becomes large, making $\widehat{B}_i \in \mathbb{R}^{2r \times 2r}$ undergoes the same issue that quasi-Newton methods face for large problems, i.e., store a large dense matrix. Hence, this chapter contribution approximates each $\widehat{v}_i \to \nabla^2 \widehat{f}_i \widehat{v}_i$ with a limited-memory quasi-Newton operator LBFGS or LSR1, as recalled in Section 2.1.3.1. This approach keeps the sparsity structure of $\nabla^2 f$, reduces the storage of each $\widehat{B}_i$ and decreases the complexity of the partitioned-matrix-vector product $v \to Bv$ from $\Theta(\sum_{i=1}^{N} \frac{n_i(n_{i+1})}{2})$ to $\Theta(\sum_{i=1}^{N} 2mn_i)$, where m is the memory parameter.

The Section 4.1 presents the details of the new limited-memory partitioned quasi-Newton trust-region algorithms. The Section 4.2 proves that every trust-region algorithm is globally convergent. The proof needs to bound $\|B_k\|$, which, in turn, necessitates to bound all $\|\widehat{B}_i\|$ across all iterations. Finally, the Section 4.3 studies the practical performance of the limited-memory partitioned quasi-Newton trust-region algorithms, whose implementations are detailed in Chapter 6. First, the Section 4.3.1 compares:

- two Newton trust-region methods. One method exploits the partial separability for computing derivatives with automatic differentiation while the other does not;
- one LBFGS line search, for which $H_k \approx \nabla^2 f(x_k)^{-1}$;
- several partitioned quasi-Newton trust-region methods;
- several limited-memory partitioned quasi-Newton trust-region methods, introduced in Section 4.1;

on a test set of partially-separable problems of size $n = 5000$. Second, the best methods out of the Section 4.3.1 are compared in Section 4.3.2 on a partially-separable problem for which n_i increases as n grows. In this last comparison, the limited-memory partitioned quasi-Newton method outperforms the limited-memory or partitioned quasi-Newton methods it is compared with. Bigeon, Orban, and Raynaud [8] originate most of the content of this chapter as well as its related implementation in Chapter 6.

4.1 Partitioned limited-memory quasi-Newton trust-region

All the partitioned quasi-Newton methods discussed in Chapter 2 utilize dense matrices to store all element Hessian approximations $\widehat{B}_i$. The storage cost and the complexity of the partitioned-matrix-vector product for such a partitioned matrix is given by

$$\Theta\left(\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2}\right),$$

which is smaller than $\frac{n(n+1)}{2}$ if n_i and the number of elements N is such that

$$N \leq \frac{n(n+1)}{\left(n_i^{\max}(n_i^{\max}+1)\right)}, \quad n_i^{\max} = \max_{1\leq i\leq N} n_i, \tag{4.1}$$

which implies

$$\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2} \leq N\frac{n_i^{\max}(n_i^{\max}+1)}{2} \leq \frac{n(n+1)}{2},$$

ensuring that storing all individual $\widehat{B}_i$ is cheaper than storing a dense matrix B_k. As the size of the elements increases, the storage of N dense matrices $\widehat{B}_i$ becomes expensive, especially if N is large. The partitioned quasi-Newton methods encounter the same issue faced by unstructured quasi-Newton methods. In such cases, the use of traditional partitioned quasi-Newton methods may not be feasible.

This chapter proposes viable partitioned quasi-Newton operators for large-element partially-separable functions by approximating each $\nabla^2 \widehat{f}_i$ with a quasi-Newton linear operator $\widehat{B}_i$, either LBFGS or LSR1, detailed in Section 2.1.3.1. By replacing a dense matrix with a limited-memory linear operator, the cost of each $\widehat{B}_i$ drops from $\Theta(\frac{n_i(n_i+1)}{2})$ to $\Theta(2mn_i)$, where $1 \leq m \leq 20$ is a typical memory factor. When aggregated together, the memory cost and the complexity of the partitioned-matrix-vector product are reduced to

$$\Theta\left(2\sum_{i=1}^{N} mn_i\right).$$

Three new methods derive from this scheme:

- PLBFGS approximates each $\nabla^2 \widehat{f}_i$ with a LBFGS operator. Among others, this method is well-suited for problems having every $\widehat{f}_i$ convex, since $\widehat{B}_i$ will preserve the positive definiteness of $\nabla^2 \widehat{f}_i$.
- PLSR1 approximates each $\nabla^2 \widehat{f}_i$ with a LSR1 operator. This method is suitable when

some $\widehat{f}_i$ are nonconvex, i.e. $\nabla^2 \widehat{f}_i \not\succ 0$.

- PLSE[1] combines both LBFGS and LSR1 operators for approximating $\nabla^2 \widehat{f}_i$ to best satisfy the secant equation (3.19). By default, $\widehat{B}_i$ will use a LBFGS update. If the safeguard verifying the secant equation fails, i.e. $s^\top y < \epsilon$, $\epsilon > 0$, then an LSR1 update is performed. If the numerical safeguard of LSR1 $|s_k^\top z_k| \geq \epsilon \|s_k\| \|z_k\|$ also fails, then $\widehat{B}_i$ is left unchanged. By allowing $\widehat{B}_i$ to be updated by either LBFGS or LSR1, every $\widehat{B}_i$ is more likely to be updated, making B partitioned such as (3.9) best satisfy (3.19).

The Algorithm 4.1.1 is a specification of the quadratic trust-region Algorithm 2.1.3 considering that B_k is a limited-memory partitioned quasi-Newton operator B_k^{PLBFGS}, B_k^{PLSR1}, or B_k^{PLSE}. PLBFGS, PLSR1 and PLSE only differ during the update of B_k, i.e., during the update of every $\widehat{B}_{i,k}$.

All PLBFGS, PLSR1 or PLSE operators have the advantages to :

- keep B_k as sparse as $\nabla^2 f$;
- support large element functions, making the storage of B_k in $\Theta\left(\sum_{i=1}^N 2mn_i\right)$ instead of $\sum_{i=1}^N \frac{n_i(n_i+1)}{2}$. Similarly to (4.1), $\sum_{i=1}^N 2mn_i^{\max} \leq \frac{n(n+1)}{2}$ as long as $N \leq \frac{n(n+1)}{4n_i^{\max}m}$;
- perform an efficient partitioned-matrix-vector product $v \to B_k v$ in $\Theta\left(\sum_{i=1}^N 2mn_i\right)$.

The next section proves the global convergence of the Algorithm 4.1.1 and Section 4.3 shows numerical comparisons of PLBFGS, PLSR1 and PLSE with limited-memory and standard partitioned quasi-Newton methods.

4.2 Global convergence proof

The convergence analysis presented in this section focuses on the convergence to first-order critical points of the Algorithm 4.1.1. It assumes several assumptions:

Assumption 4.2.1 (Assumptions on the problem)**.** *The objective function f is bounded below on $\mathbb{R}^n$. f is partially-separable as defined in* (3.3)*, and each element function $\widehat{f}_i$ is twice continuously differentiable on $\mathbb{R}^{n_i}$.*

Consequently, f is twice continuously differentiable on $\mathbb{R}^n$.

[1] SE stands for secant equation

Algorithm 4.1.1 Limited-Memory Partitioned Quasi-Newton Trust-Region Algorithm (PLBFGS, PLSR1, PLSE)

Choose $m \in \mathbb{N}^*$, $x_0 \in \mathbb{R}^n$, $\Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$, $0 < \tau < 1$ and $0 < \epsilon_1, \epsilon_2$.

Choose for every element i a linear operator LBFGS or LSR1 and $\widehat{B}_{0,i} = \widehat{B}_{0,i}^\top \approx \nabla^2 \widehat{f}_i(x_0)$. *initial approximation*

for $k = 0, 1, \ldots$ **do**

Compute an approximate solution s_k of

$$\min_s m_k(s) \quad \text{s.t.} \|s\| \leq \Delta_k, \quad m_k(s) := f(x_k) + \nabla f(x_k)^\top s + \tfrac{1}{2} s^\top \left(\sum_{i=1}^N U_i^\top \widehat{B}_{i,k} U_i \right) s,$$

bringing a sufficient decrease (2.8):

$$m_k(0) - m_k(s) \geq \tau \, \|\nabla f(x_k)\| \, \min\left(\frac{\|\nabla f(x_k)\|}{1 + \|B_k\|}, \, \Delta_k \right), \quad 0 < \tau < 1.$$

The step s_k can be computed with the truncated conjugate gradient method detailed in Section 2.1.4.

Compute the ratio

$$\rho_k := \frac{f(x_k) - f(x_k + s_k)}{m_k(0) - m_k(s_k)}.$$

if $\rho_k \geq \eta_1$ **then** *successful step*

set $x_{k+1} = x_k + s_k$,

update every $\widehat{B}_{i,k}$ to:

$$\begin{array}{lll} B_{k+1}^{\text{PLBFGS}}: & \widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LBFGS}} & \text{if } \widehat{s}_{i,k}^\top \widehat{y}_{i,k} \geq \epsilon_1, \\ B_{k+1}^{\text{PLSR1}}: & \widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LSR1}} & \text{if } |\widehat{s}_{i,k}^\top \widehat{z}_{i,k}| \geq \epsilon_2 \|\widehat{s}_{i,k}\| \|\widehat{z}_{i,k}\|, \\ B_{k+1}^{\text{PLSE}}: & \widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LBFGS}} & \text{if } \widehat{s}_{i,k}^\top \widehat{y}_{i,k} \geq \epsilon_1, \\ & \widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LSR1}} & \text{if } \widehat{s}_{i,k}^\top \widehat{y}_{i,k} < \epsilon_1 \text{ and } |\widehat{s}_{i,k}^\top \widehat{z}_{i,k}| \geq \epsilon_2 \|\widehat{s}_{i,k}\| \|\widehat{z}_{i,k}\|, \end{array} \tag{4.2}$$

given $\widehat{s}_{i,k} := \widehat{x}_{i,k+1} - \widehat{x}_{i,k}$, $\widehat{y}_{k_i} := \nabla \widehat{f}_i(\widehat{x}_{i,k+1}) - \nabla \widehat{f}_i(\widehat{x}_{i,k})$ and $\widehat{z}_{i,k} := \widehat{y}_{i,k} - \widehat{B}_{i,k} \widehat{s}_{i,k}$.

When the numerical safeguards (4.2), (4.7) and (4.8) fail, then $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$ for B_{k+1}^{PLBFGS}, B_{k+1}^{PLSR1} or B_{k+1}^{PLSE}.

else *unsuccessful step*

set $x_{k+1} = x_k$ and every $\widehat{B}_{i,k+1} = \widehat{B}_{i,k}$.

end if

Update the trust-region radius according to

$$\Delta_{k+1} \in \begin{cases} [\gamma_3 \Delta_k, \, \gamma_4 \Delta_k] & \text{if } \rho_k \geq \eta_2, \\ [\gamma_2 \Delta_k, \, \Delta_k] & \text{if } \eta_1 \leq \rho_k < \eta_2, \\ [\gamma_1 \Delta_k, \, \gamma_2 \Delta_k) & \text{if } \rho_k < \eta_1. \end{cases} \tag{4.3}$$

end for

To prove global convergence, we need to establish [39, assumption AM.4d], which states:

$$\sum_{k=1}^{\infty} \frac{1}{\varphi_k} = +\infty, \tag{4.4}$$

where

$$\varphi_k := 1 + \max_{j=1,\dots,k} \|B_j\|. \tag{4.5}$$

The purpose of the proof is to show that the sequence $\sum_{k=1}^{\infty} \frac{1}{\varphi_k}$ diverges. To do so, we need to bound the norm of B_j at every iteration.

The triangular inequality decomposes $\|B_j\|$ as follows:

$$\|B_j\| \leq \|B_j^{(0)}\| + \|B_j - B_j^{(0)}\|,$$

where $B_j^{(0)}$ aggregates all the element initializers and $B_j - B_j^{(0)}$ accumulates the contributions of low-rank limited-memory quasi-Newton updates from every element. Those terms can be expressed as:

$$B_j^{(0)} = \sum_{i=1}^{N} U_i \widehat{B}_{j,i}^{(0)} U_i^\top \text{ and } B_j - B_j^{(0)} = \sum_{i=1}^{N} U_i^\top \left(\widehat{B}_{j,i}^{(m)} - \widehat{B}_{j,i}^{(0)} \right) U_i,$$

where $\widehat{B}_{j,i}^{(0)}$ represents the initial approximation of the i-th element Hessian operator at the iteration j, and $\widehat{B}_{j,i}^{(m)}$ is the complete linear operator considering at most m LBFGS or LSR1 updates, as described in Section 2.1.3.1 for the unstructured case.

By substituting these expressions into (4.5), we obtain:

$$\frac{1}{\varphi_k} \geq \frac{1}{\max_{j=1,\dots,k} 1 + \|B_j^{(0)}\| + \|B_j - B_j^{(0)}\|}.$$

The following proof is divided into two parts. The first part aims to prove that $\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|}$ diverges under reasonable assumptions. The second part focuses on bounding $\|B_k - B_k^{(0)}\|$ for any k. This is achieved by introducing suitable numerical safeguards for the pairs $\widehat{s}_{i,k}$ and $\widehat{y}_{i,k}$ updating each element quasi-Newton operator $\widehat{B}_{i,k}$ (4.2).

Suppose $\widehat{B}_{i,k}^{(0)} = \lambda_k I$, several choices of $\lambda_{i,k}$ exist at each iteration. Thus, bounding $\|B_k^{(0)}\|$ is not trivial without further assumptions. To let a degree of freedom on how each $\widehat{B}_{i,k}^{(0)}$ is chosen, and allow a growth of $\|B_k^{(0)}\|$, we make the following assumption

Assumption 4.2.2 (Maximal element initializer series diverges)**.**

$$\sum_{k=1}^{\infty} \frac{1}{\max_{i=1,\dots,N} \|\widehat{B}_{i,k}^{(0)}\|} = +\infty.$$

The choice $\lambda_{i,k} = 1$ trivially satisfies Assumption 4.2.2. However, when $\lambda_{i,k} = \frac{\widehat{y}_{i,k}^\top \widehat{y}_{i,k}}{\widehat{y}_{i,k}^\top \widehat{s}_{i,k}}$, additional conditions must be considered. For example, $\widehat{y}_i$ and $\widehat{s}_i$ should not be too much orthogonal such that $|\lambda_{i,k}| \leq c_1,\ c_1 > 0$.

Lemma 4.2.1. *If Assumption 4.2.2 holds, then*

$$\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} = +\infty. \tag{4.6}$$

Proof. As $B_k^{(0)} = \sum_{i=1}^N U_i^\top \widehat{B}_{i,k}^{(0)} U_i$, then

$$\|B_k^{(0)}\| \leq \sum_{i=1}^{N} \|U_i\|^2 \|\widehat{B}_{i,k}^{(0)}\| \leq N \max_{i=1,\dots,N} \|U_i\|^2 \max_{i=1,\dots,N} \|\widehat{B}_{i,k}^{(0)}\|,$$

where $N \max_{i=1,\dots,N} \|U_i\|^2 > 0$ is a constant with respect to k. Therefore, the Assumption 4.2.2 yields

$$\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} \geq \frac{1}{N \max_{i=1,\dots,N} \|U_i\|^2} \sum_{k=1}^{\infty} \frac{1}{\max_{i=1,\dots,N} \|\widehat{B}_{i,k}^{(0)}\|} = +\infty.$$

□

Now that (4.6) is satisfied, we focus on bounding $\|B_k - B_k^{(0)}\|$ by introducing numerical safeguards for the quasi-Newton update of each $\widehat{B}_{i,k}$. Those safeguards prevent the updates from introducing unbounded rank one terms.

Lemma 4.2.2 (Boundedness of an element quasi-Newton update)**.** *For any $\widehat{B}_{i,k}$ that accumulates m quasi-Newton updates (BFGS or SR1) satisfying the usual safeguards* (4.2) *and enforced with*

$$|\widehat{s}_{i,k}^\top (\widehat{y}_{i,k} - \widehat{B}_{i,k}\widehat{s}_{i,k})| \geq \omega_{\text{SR1}} \|\widehat{y}_{i,k} - \widehat{B}_{i,k}\widehat{s}_{i,k}\|^2, \text{ where } \omega_{\text{SR1}} > 0, \tag{4.7}$$

for SR1 and

$$|\widehat{y}_{i,k}^\top \widehat{s}_{i,k}| \geq \omega_{\text{BFGS}_1} \|\widehat{y}_{i,k}\|^2, \text{ and } |\widehat{s}_{i,k}^\top \widehat{B}_{i,k}\widehat{s}_{i,k}| \geq \omega_{\text{BFGS}_2} \|\widehat{B}_{i,k}\widehat{s}_{i,k}\|^2, \tag{4.8}$$

for BFGS, where $\omega_{\text{BFGS}_1}, \omega_{\text{BFGS}_2} > 0$*; then there exists* $\Omega^{\text{QNU}} > 0$[a] *such that*

$$\|\widehat{B}_{i,k}^{(j+1)} - \widehat{B}_{i,k}^{(j)}\| \leq \Omega^{\text{QNU}},$$

for all k*,* i *and* $0 \leq j \leq m-1$*.*

[a] QNU stands for quasi-Newton update.

Proof. Since the two quasi-Newton updates BFGS and SR1 require different numerical safeguards, the bounding of $\|\widehat{B}_{i,k}^{(j+1)^{\text{SR1}}} - \widehat{B}_{i,k}^{(j)}\|$ and $\|\widehat{B}_{i,k}^{(j+1)^{\text{BFGS}}} - \widehat{B}_{i,k}^{(j)}\|$ are distinguished.

Suppose that the j-th quasi-Newton formula updating $\widehat{B}_{i,k}^{(j)}$ to $\widehat{B}_{i,k}^{(j+1)}$ is SR1. By enforcing (4.2) with (4.7) at every iteration k, then:

$$\|\widehat{B}_{i,k}^{(j+1)^{\text{SR1}}} - \widehat{B}_{i,k}^{(j)}\| \leq \frac{\|\widehat{s}_{i,k-m+j} - \widehat{B}_{i,k-m+j}\widehat{y}_{i,k-m+j}\|^2}{|\widehat{s}_{i,k-m+j}^{\top}(\widehat{y}_{i,k-m+j} - \widehat{B}_{i,k-m+j}\widehat{s}_{i,k-m+j})|} \leq \omega_{\text{SR1}}^{-1}.$$

Suppose that the j-th quasi-Newton formula updating $\widehat{B}_{i,k}^{(j)}$ to $\widehat{B}_{i,k}^{(j+1)}$ is BFGS and that both (4.2) and (4.8) hold at every iteration k, then:

$$\begin{aligned}\|\widehat{B}_{i,k}^{(j+1)^{\text{BFGS}}} - \widehat{B}_{i,k}^{(j)}\| &\leq \frac{\|\widehat{B}_{i,k-m+j}\widehat{y}_{i,k-m+j,i}\|^2}{|\widehat{s}_{i,k-m+j}^{\top}\widehat{B}_{i,k-m+j}\widehat{s}_{i,k-m+j}|} + \frac{\|\widehat{y}_{i,k-m+j}\|^2}{|\widehat{y}_{i,k-m+j}^{\top}\widehat{s}_{i,k-m+j}|} \\ &\leq \omega_{\text{BFGS}_1}^{-1} + \omega_{\text{BFGS}_2}^{-1}.\end{aligned}$$

The different constants $\omega_{\text{BFGS}_1}^{-1}, \omega_{\text{BFGS}_2}^{-1}$ and ω_{SR1}^{-1} remain unchanged along the iterations. Therefore, $\|\widehat{B}_{i,k}^{(j+1)} - \widehat{B}_{i,k}^{(j)}\|$ is bounded by

$$\|\widehat{B}_{i,k}^{(j+1)} - \widehat{B}_{i,k}^{(j)}\| \leq \max(\omega_{\text{BFGS}_1}^{-1} + \omega_{\text{BFGS}_2}^{-1}, \omega_{\text{SR1}}^{-1}) = \Omega^{\text{QNU}},$$

whether $\widehat{B}_{i,k}$ represents a BFGS update or an SR1 update, for any iteration k and any element i. □

If an element Hessian approximation fails to satisfy the numerical safeguards, e.g. the curvature condition for a LBFGS operator, then the update for that element is skipped, and $\widehat{B}_{i,k}^{(j+1)} = \widehat{B}_{i,k}^{(j)}$. Alternatively, a damped $\tilde{\widehat{y}}_{i,k}$ that satisfies the numerical safeguards can be used. Either ways, the boundedness property stated in Lemma 4.2.2 still holds, ensuring that the norm of the element quasi-Newton update remains bounded.

Lemma 4.2.3 (Boundedness of an element limited-memory quasi-Newton operator)**.** *For any $\widehat{B}_{i,k}$, accumulating m quasi-Newton updates satisfying the numerical safeguards* (4.2)*,* (4.7) *and* (4.8)*, then there exists $\Omega^{\text{LM}} > 0$[a] such that*

$$\|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\| \leq \Omega^{\text{LM}},$$

for any iteration k and any element i, independently of the quasi-Newton updates performed by $\widehat{B}_{i,k}$.

[a] LM stands for limited-memory

Proof. We decompose $\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}$ as

$$\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)} = \widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(m-1)} + \widehat{B}_{i,k}^{(m-1)} - \widehat{B}_{i,k}^{(m-2)} + \cdots + \widehat{B}_{i,k}^{(1)} - \widehat{B}_{i,k}^{(0)},$$

as m individual updates. Thus, by applying m times Lemma 4.2.2, we find an upper-bound of $\|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\|$:

$$\|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\| \leq \sum_{j=1}^{m} \|\widehat{B}_{i,k}^{(j)} - \widehat{B}_{i,k}^{(j-1)}\| \leq \sum_{j=1}^{m} \Omega^{\text{QNU}} \leq m\Omega^{\text{QNU}} = \Omega^{\text{LM}},$$

for any iteration k and any element i. □

One interesting characteristic of both Lemma 4.2.2 and Lemma 4.2.3 is that they apply without any distinction to both BFGS and SR1 updates. This means that those lemmas naturally handle the changes in quasi-Newton updates that can be performed by a single element Hessian approximation $\widehat{B}_{i,k}$, as in the case of B_k^{PLSE}. Furthermore, Ω^{LM} also provides a bound for all limited-memory quasi-Newton operators $\widehat{B}_{i,k}$ that have accumulated less than m quasi-Newton updates. This situation occurs for every element during the $m-1$ first iterations of an optimization method, or when some element updates are skipped.

Lemma 4.2.4 (Boundedness of a limited-memory partitioned quasi-Newton update)**.** *For any limited-memory partitioned quasi-Newton operator B_k as described in Section 4.1 and Algorithm 4.1.1 that satisfies the numerical safeguards* (4.2)*,* (4.7) *and* (4.8)*, then there exists $\Omega > 0$*

$$\|B_k - B_k^{(0)}\| \leq \Omega,$$

for all k.

Proof. By definition,

$$B_k - B_k^{(0)} = \sum_{i=1}^{N} U_i^\top \left(\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)} \right) U_i.$$

Thus, the norm may be bounded as:

$$\|B_k - B_k^{(0)}\| \leq \sum_{i=1}^{N} \|U_i\|^2 \|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\| \leq \max_{i=1,\ldots,N} \|U_i\|^2 \sum_{i=1}^{N} \|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\|. \tag{4.9}$$

Lemma 4.2.3 provides an upper-bound for any $\|\widehat{B}_{i,k}^{(m)} - \widehat{B}_{i,k}^{(0)}\|$, which simplifies (4.9) to:

$$\|B_k - B_k^{(0)}\| \leq \max_{i=1,\ldots,N} \|U_i\|^2 \sum_{i=1}^{N} \Omega^{\mathrm{LM}} \leq N \max_{i=1,\ldots,N} \|U_i\|^2 \Omega^{\mathrm{LM}} = \Omega,$$

for any iteration k, since $N \max_{i=1,\ldots,N} \|U_i\|^2$ is constant. □

By assuming the Assumption 4.2.2, the Lemma 4.2.1 proves that $\sum_{k=1}^{\infty} \frac{1}{\|B_j^{(0)}\|}$ diverges. Furthermore, the Lemma 4.2.4 determines an upper bound for $\|B_k - B_k^{(0)}\|$ regardless of k. With both requirements covered, we prove now (4.4).

Theorem 4.2.1. *If Assumption 4.2.2 holds, then any limited-memory partitioned quasi-Newton trust-region method presented in Algorithm 4.1.1, namely: PLBFGS, PLSR1 or PLSE satisfies:*

$$\sum_{k=1}^{\infty} \frac{1}{\varphi_k} = +\infty,$$

as along as the safeguards (4.2)*,* (4.7) *and* (4.8) *are satisfied across the iterations.*

Proof. Lemma 4.2.4 informs that

$$\|B_k\| \leq \|B_k - B_k^{(0)}\| + \|B_k^{(0)}\| \leq \Omega + \|B_k^{(0)}\|.$$

Therefore,

$$\frac{1}{\|B_k\|} \geq \frac{1}{\Omega + \|B_k^{(0)}\|} \text{ and } \sum_{k=1}^{\infty} \frac{1}{\|B_k\|} \geq \sum_{k=1}^{\infty} \frac{1}{\Omega + \|B_k^{(0)}\|}.$$

By assuming the Assumption 4.2.2, the Lemma 4.2.1 guarantees

$$\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} = +\infty,$$

which gives straightforwardly

$$\sum_{k=1}^{\infty} \frac{1}{\Omega + \|B_k^{(0)}\|} = +\infty.$$

□

Theorem 4.2.1 proves [39, AM4.d] leaving only two assumptions to make for fulfilling the conditions of [39, Theorems 8.4.7].

Assumption 4.2.3. *[39, AN.1] There exists a constant $\kappa_{une} \geq 1$ such that, for all k,*

$$\frac{1}{\kappa_{une}} \|x\|_p \leq \|x\| \leq \kappa_{une} \|x\|_p,$$

for all $x \in \mathbb{R}^n$.

Assumption 4.2.4. *[39, AM.4f]*

$$\lim_{k \to \infty, k \in S} \varphi_k . \left(f(x_k) - f(x_{k+1})\right) = 0,$$

where S is the set of successful iterations.

From these two additional assumptions, the convergence theorem based on [39, Theorems 8.4.7] can be stated.

Theorem 4.2.2 (Boundedness of limited-memory partitioned quasi-Newton operators (PLBFGS, PLSR1 and PLSE))**.** *By assuming that Assumption 4.2.1 ($\widehat{f}_i \in \mathcal{C}^2$, $\forall i$), Assumption 4.2.2, Assumption 4.2.3 and Assumption 4.2.4 hold; the sequence of points x_k generated by each limited-memory partitioned quasi-Newton trust-region method, namely: PLBFGS, PLSR1 and PLSE formalized by the Algorithm 4.1.1 and enforced with the numerical safeguards* (4.2)*,* (4.7) *and* (4.8) *converges*

$$\lim_{k \to \infty} \|\nabla f(x_k)\| = 0.$$

Proof. By assuming Assumption 4.2.1 and choosing B_k as a limited-memory partitioned linear operator makes $m_k \in \mathcal{C}^2$ and $f(x_k) = m_k(0)$. Consequently, several assumptions related to trust-region convergence, namely [39, AF.1-2, AM.1, AM.2, AM.3], are covered. Furthermore, solving (2.4) with the truncated conjugate gradient [144] covers [39, AA.1], whereas γ_3 and γ_4 choices from the Algorithm 4.1.1 fulfill [39, AA.4]. Finally, Theorem 4.2.1 covers [39, AM.4d]

and Assumption 4.2.4 completes the assumptions made for [39, theorem 8.4.7]: AF.1-3, AM.1, AM.2, AM.3, AA.1, AA.4, AM.4d. As a result,

$$\lim_{k\to\infty} \|\nabla f(x_k)\| = 0.$$

□

The Theorem 4.2.2 ensures the convergence of PLBFGS, PLSR1 and PLSE to a stationary point.

4.3 Numerical results

This section compares several trust-region methods described in the previous sections on a set of 65 partially-separable problems from Orban et al. [123]. The details of the partially-separable structure of these problems are provided in Appendix A. The numerical results compare several variants of a trust-region method:

- Newton variants, similar to the Algorithm 2.1.2 where $B_k = \nabla^2 f(x_k)$;
- quasi-Newton variants, similar to the Algorithm 2.1.3;
- partitioned quasi-Newton variants, similar to the Algorithm 3.2.1;
- limited-memory partitioned variants, similar to the Algorithm 4.1.1.

All the methods tested result from the same trust-region implementation. It slightly differs from precedent trust-region algorithms by adding a backtracking line search (Algorithm 2.1.1) along s_k when s_k is an unsuccessful step, as proposed in [39, section 10.3.2]. As a consequence, the differences between all variants are the approximation $B_k \approx \nabla^2 f(x_k)$ and the numerical routines involved to compute $f, \nabla f$ and $v \to B_k v$. The Julia modules implemented and used for producing those numerical results are detailed in Chapter 6. The following list details all the methods and their specifics:

- (Newton) **PHv** and **Hv**: both methods compute $B_k v = \nabla^2 f(x_k)v$ using different approaches. **Hv** uses reverse and forward automatic differentiation on f to compute $\nabla^2 f(x_k)v$ [73] while **PHv** aggregates the contributions of $\nabla^2 \widehat{f}_i(\widehat{x}_{i,k})U_i v$ computed with reverse and forward automatic differentiation applied onto $\widehat{f}_i$, see Section 3.5.
- (Quasi-Newton) **LBFGS_TR** and **LSR1_TR**: these methods consider B_k as a LBFGS operator or a LSR1 operator, respectively, see Section 2.1.3.1.

- Partitioned quasi-Newton methods consider B_k as a sum of element quasi-Newton operators, i.e., $B_k v = \sum_{i=1}^{N} U_i \widehat{B}_{i,k} U_i v$. The methods below consider each $\widehat{B}_{i,k}$ as a dense matrix, the methods are:
  - **PBFGS**: each $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{BFGS}}$ if $\widehat{y}_{i,k}^{\top} \widehat{s}_{i,k} \geq \epsilon_1$.
  - **PSR1**: each $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{SR1}}$ if $|\widehat{s}_{i,k}^{\top} \widehat{z}_{i,k}| \geq \epsilon_2 \|\widehat{s}_{i,k}\| \|\widehat{z}_{i,k}\|$, where $\widehat{z}_i \coloneqq \widehat{y}_{i,k} - \widehat{B}_{i,k} \widehat{s}_{i,k}$, and $\epsilon_2 > 0$.
  - **PCS**: the convex element functions are updated with BFGS, i.e. $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{BFGS}}$ while the nonconvex element functions are updated with SR1, i.e. $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{SR1}}$. The recognition of convexity is detailed later in Section 6.3.
  - **PSE**: $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{BFGS}}$ if $\widehat{y}_{i,k}^{\top} \widehat{s}_{i,k} > \epsilon_1$, otherwise $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{SR1}}$.
- Limited-memory partitioned quasi-Newton methods are similar to the partitioned quasi-Newton methods but every $\widehat{B}_{i,k}$ is a linear operator, the methods are:
  - **PLBFGS**: each $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LBFGS}}$ when $\widehat{y}_{i,k}^{\top} \widehat{s}_{i,k} \geq \epsilon_1$;
  - **PLSR1**: each $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LSR1}}$ when $|\widehat{s}_{i,k}^{\top} \widehat{z}_{i,k}| \geq \epsilon_2 \|\widehat{s}_{i,k}\| \|\widehat{z}_{i,k}\|$;
  - **PLSE**: some $\widehat{B}_{i,k}$ are LBFGS operators while the rest are LSR1 operators. Similarly to **PSE**, if $\widehat{y}_{i,k}^{\top} \widehat{s}_{i,k} > \epsilon_1$, then the pair $\widehat{s}_{i,k}, \widehat{y}_{i,k}$ will be used to perform $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LBFGS}}$, otherwise it will perform $\widehat{B}_{i,k+1} = \widehat{B}_{i,k+1}^{\text{LSR1}}$.

  If the numerical safeguards fail during an element quasi-Newton update, then the element update is skipped.

To reproduce the numerical comparisons discussed in Chapter 3, a **LBFGS** line search method is added. This **LBFGS** implementation is similar to Liu and Nocedal [101] employing the linear operator H_k approximating $\left(\nabla^2 f(x_k)\right)^{-1}$, i.e. $H_k = B_k^{-1}$. The results of the comparisons are illustrated with Dolan and Moré performance profiles [49], considering criteria such as computation time and number of iterations before achieving first-order convergence.

In Section 4.3.2, the best methods out of the Section 4.3.1 are compared on a partially-separable problem with element functions that become larger as the problem dimension grows. This study compares how the nature of an element Hessian approximation $\widehat{B}_{i,k}$, i.e., a dense matrix or a limited-memory operator, impacts the problem resolution. This scenario assesses the relevancy of limited-memory partitioned quasi-Newton Hessian approximations for partially-separable functions with large element functions, which is the main issue for classical partitioned quasi-Newton methods.

4.3.1 Comparing quasi-Newton methods

All numerical experiments in this chapter were conducted on an Intel(R) Xeon(R) Gold 6126 CPU 2.60GHz architecture. In the test set considered, the partially-separable objectives have element functions of relatively small sizes compared to the overall problem size, e.g., $2 = n_i \ll n = 5000$. The profiles are built considering a time budget and an iteration budget. For example, if a *solver*, i.e., an optimization method, runs for more than one hour or exceeds 50 000 evaluations of the objective, the solver stops. Such failed executions do not meet the absolute or relative first-order convergence criteria, which are respectively defined as $\|\nabla f(x_k)\| \leq 10^{-6}$ and $\|\nabla f(x_k)\| \leq 10^{-6}\|\nabla f(x_0)\|$, considering the initial point x_0. Therefore, the problem is not counted as solved for this particular solver.

The Figure 4.1 presents the performance profiles for Newton and quasi-Newton methods, and aims to reproduce a state-of-the-art methods comparison. An attentive reader may find surprising that two methods performing the same iterations: **PHv** and **Hv**, result in two clearly different curves. This difference is most likely not only explained by the floating arithmetic difference between implementations. Although floating divergences can occur, most of the problems solved within the allowed budgets result in identical iteration counts. However, the partitioned Hessian-vector product of **PHv** outperforms the Hessian-vector product of **Hv**, making **PHv** faster to solve the trust-region subproblem (2.4) than **Hv**. Concretely, **Hv** fails to solve some problems given the time budget unlike **PHv**. Consequently, **PHv** solves 84% of the test set while **Hv** solves 61% of the test set. **PHv** and **Hv** have the best iteration records for 52% of the test set out of all methods. Hence, it produces parallel curves between the two methods on the iteration performance profile, ending with a 23% difference of problems solved. The parallel curves break at some points, due to the existing floating arithmetic differences between the methods and due to problems for which **PHv** is not the best method (in iteration). The two quasi-Newton methods **LBFGS** and **LBFGS_TR** exhibit similar iteration performance, solving both 80% of the test set. Among the solved problems, **LBFGS_TR** solves 93.75% of them faster than **LBFGS**. Lastly, **LSR1_TR** solves only 58% of the test set. Overall, Newton methods require fewer iterations. However, due to $\nabla^2 f(x_k)v$ computational effort, fewer iterations do not reflect in smaller resolution time. Although **PHv** displays the advantages of exploiting partial separability over **Hv**, **LBFGS** methods solve most of the problems in less time than **PHv**.

The Figure 4.2 shows comparisons of **PBFGS**, **PSR1**, **PCS**, and **PSE** in terms of time and iterations. The iteration performance profile distinguishes two groups of methods. First: **PSE** and **PBFGS** and second: **PSR1** and **PCS**. The two methods of each group solve roughly the same percentage of problems and display similar performance curves. The first group seems to

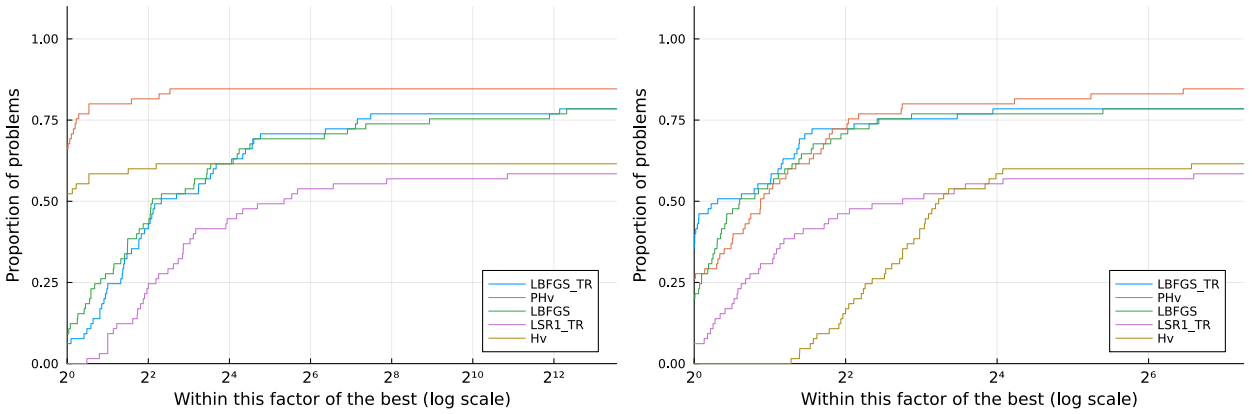


Figure 4.1 Iteration (left) and time (right) performance profiles for Newton and quasi-Newton methods.

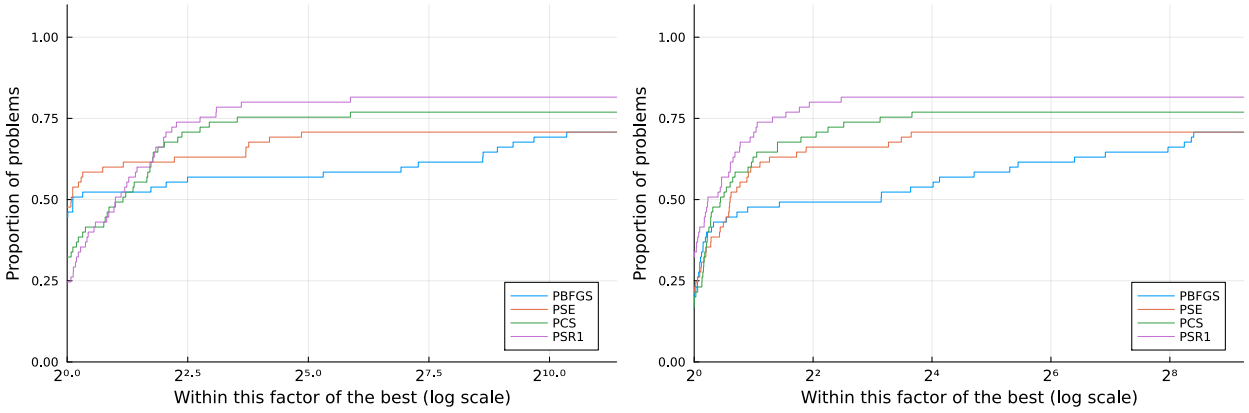


Figure 4.2 Iteration (left) and time (right) performance profiles for partitioned quasi-Newton methods.

solve more than 50% of the test set with fewer iterations than the second group, which happens to dominate the remaining 25% problems of the test set. The best method of each group: **PSE** and **PSR1** solve respectively 70% and 82% of the test set with respect to the given budgets. The record of time consumption for partitioned quasi-Newton methods Figure 4.2 indicates that the second group generally out speeds group one in time resolution, and **PSR1** emerges as the fastest method seemingly on all problems. This observation can be justified by the fact that both a $\widehat{B}^{\text{BFGS}}_{i,k+1}$ update and a product $v \to \widehat{B}^{\text{BFGS}}_{i,k+1}v$ require roughly the double of computational effort than a $\widehat{B}^{\text{SR1}}_{i,k+1}$ update and a product $v \to \widehat{B}^{\text{SR1}}_{i,k+1}v$ need. As **PCS** and **PSR1** mostly perform $\widehat{B}^{\text{SR1}}_{i,k+1}$ while **PBFGS** and **PSE** mostly perform $\widehat{B}^{\text{BFGS}}_{i,k+1}$, **PCS** and **PSR1** are less computationally intensive.

The Figure 4.3 shows comparisons of the limited-memory partitioned quasi-Newton methods: **PLBFGS**, **PLSR1**, and **PLSE**. Among them, **PLSE** dominates by necessitating fewer iterations and less time before convergence than **PLSR1** and **PLBFGS**. **PLSE** solves 77% of the problems, **PLSR1** 68% and **PLBFGS** 57%. Nonetheless, having fewer iterations does not translate directly to shorter runtime. This difference in performance can be attributed to the change of quasi-Newton linear operator between $\widehat{B}_{i,k}$ and $\widehat{B}_{i,k+1}$ for **PLSE**, which requires additional allocations and affects the overall runtime efficiency. It is worth noting that both **PBFGS** and **PLBFGS** struggle to perform well in those experiments. The ineffectiveness of partitioned (L)BFGS element updates may be due to two reasons. First, the element functions are not necessarily convex. Second, more truncated conjugate gradient iterations may be needed for solving the convex trust-region subproblem when the solution is inside the trust-region, i.e., there is no negative curvature to cut off conjugate gradient iterations.

The Figure 4.4 provides a comprehensive comparison by gathering the most significant methods from the previous profiles: **PHv**, **LBFGS**, **PSE**, **PSR1**, **PLSE**. In the iteration performance profile, three trends coexist: first **PHv**, second the partitioned quasi-Newton methods, and lastly, **LBFGS**. **PHv** dominates the iteration performance profile. The partitioned quasi-Newton methods all require fewer iterations than **LBFGS**, except perhaps **PSE**.

The time performance profile records that both **PHv** and **LBFGS**, despite being the best and worst method in terms of iterations, have similar time performance. Partitioned quasi-Newton methods dominate, led by the dense versions **PSR1** and **PSE**, followed by **PLSE**.

The results displayed in Figure 4.4 suggest to use **PSR1** or **PSE** in case f is partially-separable. Although the partially-separable problems used in this section do not favour limited-memory partitioned quasi-Newton methods, i.e., n_i are small, **PLSE** remains a more efficient method than **PHv** or **LBFGS**. The next section will illustrate **PLSE**'s features when element sizes grow larger.

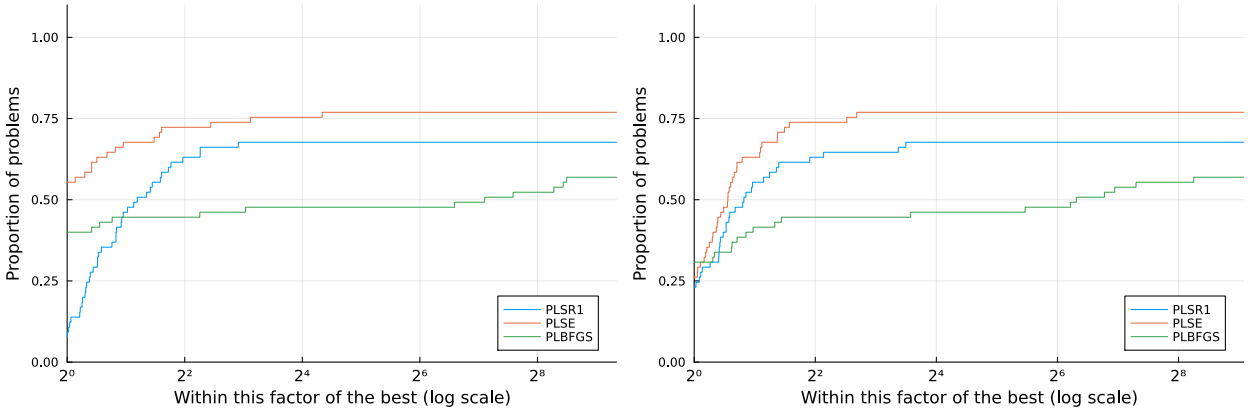


Figure 4.3 Iteration (left) and time (right) performance profiles for limited-memory partitioned quasi-Newton methods.

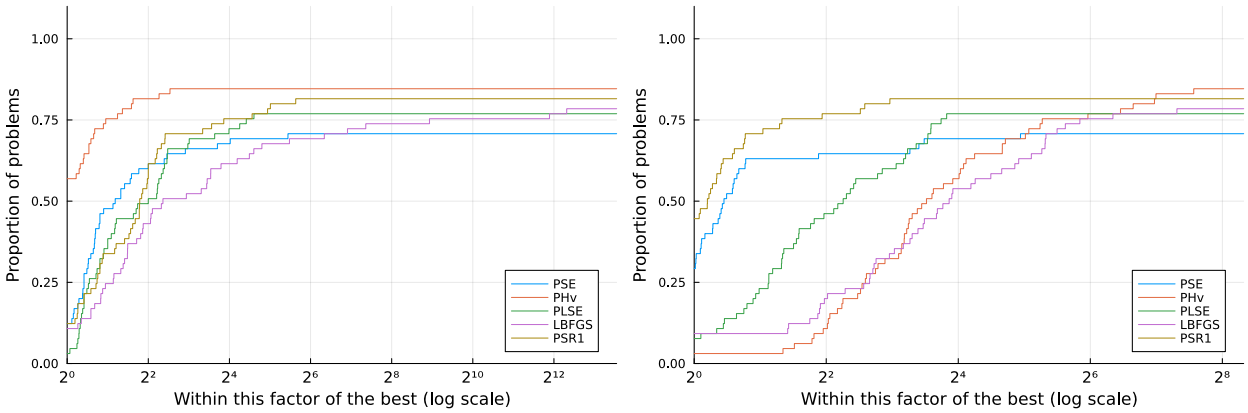


Figure 4.4 Summary of iteration (left) and time (right) performance profiles for (limited-memory) partitioned quasi-Newton methods

4.3.2 Experiments with limited-memory partitioned quasi-Newton methods

In this section, the numerical experiments consider a partially-separable function that is designed to gradually increase n_i and N as n grows larger. The function is formulated as

$$f^{\text{limit}} = \sum_{j=1}^{\sqrt{n}-3} \frac{\left(\sum_{i=(j-1)\sqrt{n}}^{(j+2)\sqrt{n}} ix_i\right)^2}{1+x_j^2} + \sum_{j=1}^{\sqrt{n}-5} \frac{\left(\sum_{i=(j-1)\sqrt{n}+5}^{(j+4)\sqrt{n}+5} ix_i\right)^2}{1+x_j^2}. \tag{4.10}$$

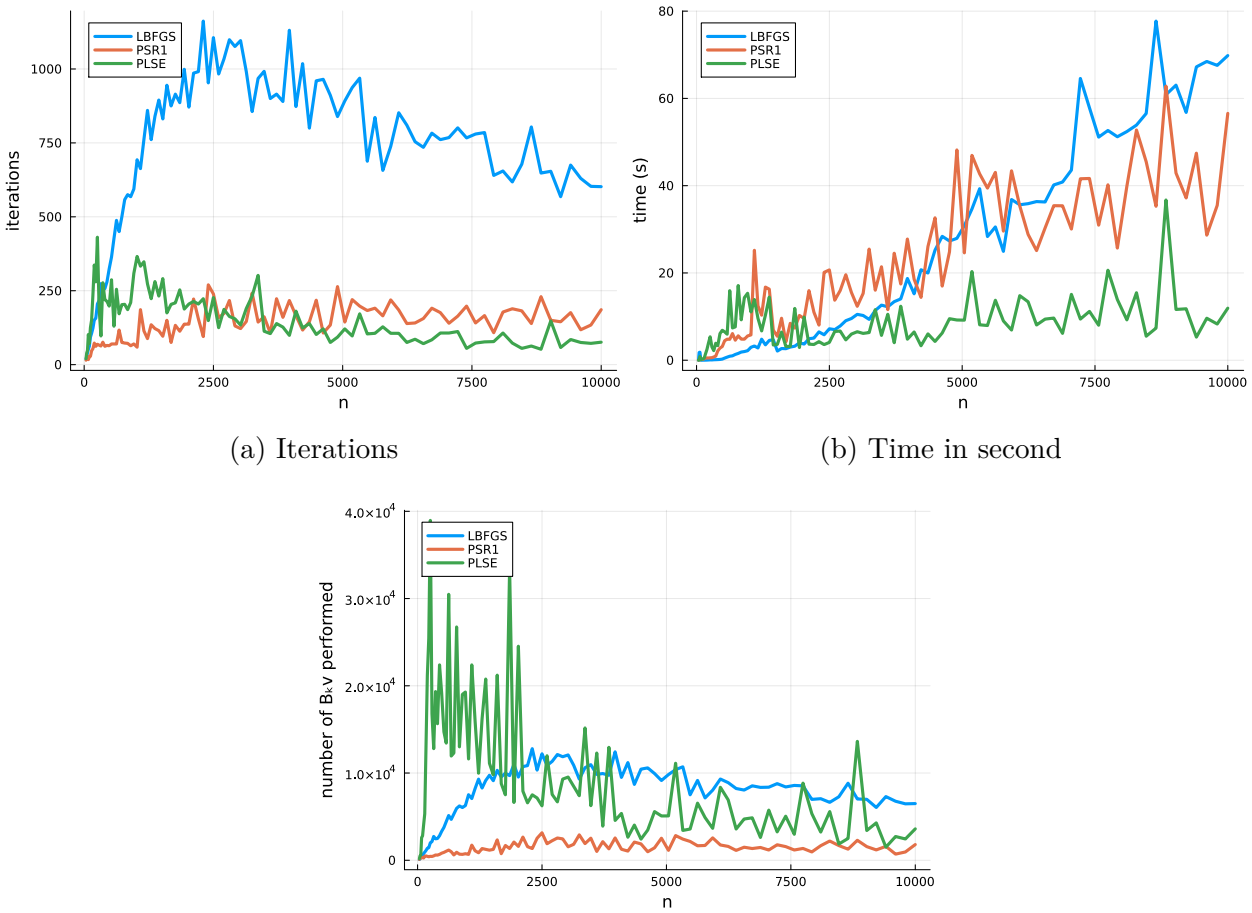
f^{limit} has $N \approx 2\sqrt{n} - 8$ element functions, and challenges the standard partitioned quasi-Newton methods since storing large dense matrices $\widehat{B}_i$ becomes memory intensive. The use of $\sqrt{n}$ for indexing the sum in (4.10) suppose to round $\sqrt{n}$ to a near integer. By doing so, for all n such that $\beta^2 < n < (\beta+1)^2$, $f(n)$ is either $f(\beta^2)$ or $f((\beta+1)^2)$. Therefore, only $n \in \{\beta^2 \mid \beta \in \mathbb{N}^*\}$ are relevant for modelling functions f of different dimensions. The Table 4.1 gives some measures for f^{limit} depending on $n \in \{36, 625, 2500, 10000\} = \{\beta^2 \mid \beta \in \{6, 25, 50, 100\}\}$. ED stands for element dimension, i.e. n_i, and EC stands for element contribution, which counts the number of element functions in which each variable occurs.

Table 4.1 Instance details of f^{limit} for $n \in \{36, 625, 2500, 10000\}$

size (n)	N	min. ED	mean ED	max ED	mean EC	max EC
36	4	18	21.75	31	2.42	4
625	42	75	100.24	127	6.73	9
2500	92	150	200.38	252	7.37	9
10000	192	300	400.44	502	7.69	9

The Figure 4.5 displays a comparison of the best method from each quasi-Newton trust-region family established in the previous section, namely: **LBFGS**, **PSR1** and **PLSE**. The Figure 4.5a shows iterations, the Figure 4.5b records time and Figure 4.5c exposes the number of products $B_k v$ required before each method reaches a first-order convergence. In Figure 4.5a, both **PSR1** and **PLSE** require fewer iterations than **LBFGS**. Both methods expose an iteration count seemingly independent of n, at least three times smaller than **LBFGS**. The superior performance of partitioned methods in terms of iterations can be attributed by the better approximations B_k of $\nabla^2 f(x_k)$, which at least, maintains the sparsity structure of $\nabla^2 f(x_k)$.

The Figure 4.5b records the convergence time, and exhibits different behaviours compared to the iteration counts from Figure 4.5a. For **LBFGS**, whereas the iteration count seems to stagnate as n increases, the convergence time continues to grow with a linear rate, up to



(a) Iterations

(b) Time in second

(c) Number of $B_k v$ performed

Figure 4.5 Metrics of quasi-newton methods solving (4.10).

reach more than five **PLSE**'s time for $n = 10\,000$. **PSR1**'s time also increases with n, having identical time than **PLSE** for $n \leq 2500$ and reaching four **PLSE**'s time for $n = 10\ 000$. Conversely, **PLSE**'s time remains mostly constant. It is worth noting that spikes in the iteration counts from the Figure 4.5a are not always echoed by spikes in Figure 4.5b. This is due to the fact that most of the computational effort of a trust-region method comes from the truncated conjugate gradient method. Although the amount of products $B_k v$ performed usually scales with ∇f, it is not always the case. Therefore, in some cases, the time spent for computing products $B_k v$ overshadows the time spent for evaluating f or ∇f. This particular behaviour is illustrated in Figure 4.5 for $n = 7744 = 88^2$. While the number of iterations remains similar to $n = 7569 = 87^2$, the number of products $B_k v$ performed doubles, which reflects in Figure 4.5b.

The limitation of the standard partitioned quasi-Newton methods for this type of partially-

separable function lies in the cost of storing and manipulating the dense matrices $\widehat{B}_{i,k}$. Hence, as n_i grows, the storage and computational costs for operating B_k increase quadratically. Conversely, limited-memory partitioned quasi-Newton methods, such as **PLSE**, overcome this limitation by employing linear operators for approximating $\widehat{v}_i \to \nabla^2 \widehat{f}_i(\widehat{x}_i)\widehat{v}_i$ and best perform when n_i is large.

The Figure 4.5b indicates that the time growth of **PLSE** for minimizing f^{limit} is almost null compared to **PSR1**, even though **PLSE** may perform more $B_k v$ products than **PSR1** (Figure 4.5c). This study determines that limited-memory partitioned quasi-Newton methods are better suited for partially-separable functions having large element functions. The key features of **PLBFGS**, **PLSR1** and **PLSE** are: a sparse Hessian approximation, a reasonable memory usage and an efficient operator-vector product $v \to B_k v$.

4.4 Conclusion

This chapter presented several limited-memory partitioned quasi-Newton operators to approximate the Hessian of the objective function. While a partitioned quasi-Newton operator approximates each element Hessian by a dense matrix, its limited-memory variant approximates each element Hessian by a limited-memory quasi-Newton operator. The resulting partitioned-matrix possesses a faster partitioned-matrix-vector product and alleviates the memory requirement from $\Theta\left(\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2}\right)$ to $\Theta\left(\sum_{i=1}^{N} mn_i\right)$. As a consequence, a partially-separable function with large element functions can be efficiently minimized.

The next chapter transfers the concepts of partial separability to deep learning, resulting in the formulation of a partially-separable training problem with large element functions. This new training problem is later minimized with a limited-memory partitioned quasi-Newton method.

CHAPTER 5 Partially-separable learning

In this chapter, the concepts of partial separability are transferred to deep learning. This contribution is divided in three parts. The Julia modules implementing those parts are detailed in Chapter 6.

The first part, detailed in Section 5.1, focuses on the impact that the loss function employed has onto the computation of the loss function derivatives. When the loss function exhibits partial separability, the gradient of the loss function sums the contributions of element loss functions, each of smaller dimension. This approach lends itself to parallelization within a master(s)-workers framework by assigning one or several element functions to each worker. Unlike model parallelism or tensor parallelism, recalled in Section 2.2.6, it does not necessitate communication between workers after the evaluation of every layer and solely aggregates the element loss contributions. Consequently, it can seamlessly integrate the other parallelism schemes introduced in Section 2.2.6.

The second part studies in Section 5.2 the effective exploitation of a partially-separable loss function during the neural network training, which heavily depends on the neural network's architecture. Generally, conventional architectures fail to exploit partial separability as their loss element function dimensions approach the overall neural network dimension, i.e., $n_i \approx n$. Hence, Section 5.2 introduces the separable layer, designed to reduce the loss function dimensions in comparison to the total neural network dimension. As a consequence, a partitioned neural network, i.e., stacking several separable layers, that employs a partially-separable loss function can be theoretically trained over several workers, each containing only a small fragment of the neural network. Numerical results show that a combination of a partially-separable loss function and a partitioned architecture has a comparable performance to that of a standard architecture paired with a conventional loss function. Raynaud, Orban, and Bigeon [135] introduce most of the content of those two first parts, whereas Raynaud, Orban, and Bigeon [134] focus on the third and last part.

The third part, presented in Section 5.3, implements a limited-memory partitioned quasi-Newton training. In the continuation of the Chapter 4, each element loss function Hessian is approximated with a LBFGS or a LSR1 operator. Unlike the Chapter 4, the optimization method is a line search instead of a trust-region method. The numerical results presented in Section 5.3.1 show that limited-memory partitioned quasi-Newton trainings are competitive with state-of-the-art trainings, including a LBFGS training.

The current limitations and the remaining open problems are discussed in Section 5.4.

5.1 Partially-separable loss function

Before investigating on partially-separable loss functions, one must observe that most loss functions are depending nonlinearly on all scores c_i. This is the case for $\mathcal{L}^{\text{NLL}}(X, Y; w)$:

$$\mathcal{L}^{\text{NLL}}(X, Y; w) = \tfrac{1}{L} \sum_{l=1}^{L} \underbrace{-\log(p_{y^{(l)}}(x^{(l)}; w))}_{\mathcal{L}^{\text{NLL}}(x^{(l)}, y^{(l)}; w)}, \quad p_i(x^{(l)}; w) := \tfrac{exp(c_i(x^{(l)}; w))}{\sum_{j=1}^{C} exp(c_j(x^{(l)}; w))}.$$

Therefore, $\mathcal{L}^{\text{NLL}}(x^{(l)}, y^{(l)}; w)$ does not fulfill the criteria of an element function, as its dimension matches that of $\mathcal{L}^{\text{NLL}}(X, Y; w)$, i.e., $\mathcal{L}^{\text{NLL}}(x^{(l)}, y^{(l)}; w) : \mathbb{R}^{n^{\text{NLL}}} \to \mathbb{R}$ is such that $n = n^{\text{NLL}} \not< n$.

To define an element loss of smaller dimension, it must depend on a subset of the scores c_j, which structurally rely on a subset of weights, i.e. $\widehat{c}_j : \mathbb{R}^{n_j} \to \mathbb{R}$ parametized by U_j (Figure 5.1 and Figure 5.3). To remedy this issue, the partially-separable loss (PSL) sums element loss functions depending on a couple of scores:

$$\mathcal{L}^{\text{PSL}}(X, Y; w) := \frac{1}{L} \sum_{l=1}^{L} \sum_{j=1}^{C} e^{c_j(x^{(l)}; w) - c_{y^{(l)}}(x^{(l)}; w)}, \tag{5.1a}$$

$$= \sum_{p=1}^{C} \sum_{j=1 \neq k}^{C} h_{p,j}(X, Y; w), \tag{5.1b}$$

$$h_{p,j}(X, Y; w) := \frac{1}{L} \sum_{l=1}^{L} \delta_{p,j}(y^{(l)})\ e^{c_j(x^{(l)}; w) - c_p(x^{(l)}; w)}, \tag{5.1c}$$

where $\delta_{p,j}(y^{(l)}) = 1$ if $y^{(l)} = p$, and 0 otherwise. The element loss function $h_{p,j}$ relies solely on the weights that parametrize the two scores c_p and c_j:

$$h_{p,j}(X, Y; w) = \widehat{h}_{p,j}(X, Y; U_{p,j} w), \quad U_{p,j} \in \mathbb{R}^{n_{p,j} \times n}. \tag{5.2}$$

$U_{p,j}$ is the linear operator combining the weights selected by both U_p and U_j. As a consequence, $n_{p,j} \leq n_p + n_j$ and $n_{p,j} \leq n$. Structurally, $n_{p,j} = n_p + n_j$ when there is no weight overlap between c_j and c_p, and $n_{p,j} = n$ only if $C = 2$. If the neural network gets a maximum score c_j different from $c_{y^{(l)}}$ for a given input $x^{(l)}$, then $e^{c_j(x^{(l)}; w) - c_{y^{(l)}}(x^{(l)}; w)}$ will return a large value. Hence, minimizing (5.1a) reduces misclassification.

The $\mathcal{L}^{\text{PSL}}$ loss sums $N = C^2 - C$ element functions $\widehat{h}_{p,j}$, each of which calculates a loss between a pair of distinct classes c_p and c_j ($p \neq j$). In cases where the neural network is symmetric, there exists a common set of weights on which every element function depends (although this set could be empty). Such a simplified symmetric neural network having common weights is

illustrated in Figure 5.3. Additionally, all element losses perform the same computation in a symmetric neural network, but with different weights as parameters—$n_{p_1,j_1} = n_{p_2,k_2}$ while $U_{p_1,j_1} \neq U_{p_2,j_2}$, for all $(p_1, j_1) \neq (p_2, j_2)$. The specific dimensions of each $\widehat{h}_{p,j}$ depends upon the layers composing the neural network architecture. The Section 5.2 develops deeper this subject and introduces the concept of a *separable* layer to make $n_{p,j}$ a smaller fraction of n.

5.2 Separable layer and partitioned architecture

5.2.1 The issue of standard architectures

As previously discussed, a partially-separable loss function can be employed with any type of multiclass classification neural network. This section explains why most architectures fail to exploit partially-separable loss functions. For instance, the Figure 5.1 showcases

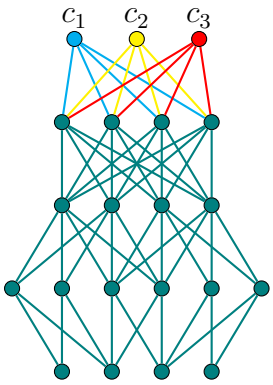


Figure 5.1 Weight dependencies of simplified LeNet scores

the weight dependencies of each score within an overly simplified LeNet [96] architecture. This architecture incorporates two convolutional layers, defined in Section 2.2.2, consisting respectively of two and three kernels. Subsequently, two dense layers are applied. The setup is tailored for a vectorized input of size four and generates $C = 3$ class scores. Distinct colors—blue, yellow, and red—represent weights that exclusively parameterize a single score. Meanwhile, common weights shared among all scores are indicated in green. In this example, the distinct colors are solely utilized to differentiate weights forming the last layer, i.e. prior the loss function layer, e.g. softmax in Figure 2.3. All weights beneath this layer are common weights (in green). Generally, the application of a dense or a convolutional layer onto a given layer makes all the precedent weights (from bottom to top) common for all scores c_j. As a result, any neural network topped by a dense layer generates scores depending on the vast majority of the network, i.e., $n_j \approx n$, $\forall\, 1 \leq j \leq C$. Consequently, $n_{p,j} \geq max(n_j, n_p)$ tends to approach n.

5.2.2 Separable layer

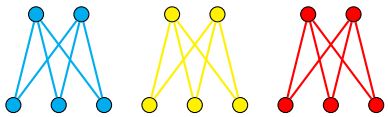


Figure 5.2 A separable layer, 9×6, considering $C = 3$ groups

The concept of *separable* layers is introduced to mitigate n_j in comparison to n, such as depicted in Figure 5.2. This layer divides the neurons from two consecutive layers into C groups, with each group being fully connected to its counterpart in the subsequent layer, similarly to [145]. A separable layer employs C times fewer weights than a fully connected layer. Structurally, it propagates the group weight dependencies from one layer to its corresponding group in the next layer, to which are added the weights parametrizing the separable layer itself. Unlike dense or convolutional layers, it avoids spreading weight dependencies of the precedent layer across all neurons of the subsequent layer. However, here are some relevant practical considerations:

- if a layer relies nonlinearly on all the outputs from a stack of separable layers, then the partitioned structure introduced by the stack of separable layers vanishes;
- the output dimension of the layer preceding the initial separable layer must be a multiple of C. Consequently, dense layers will necessitate $\alpha * C$ neurons, while convolutional layers will require $\alpha * C$ kernels, where $\alpha \in \mathbb{N}^*$. Each group within a separable layer in based on the output of either α neurons or α kernels.

5.2.3 Partitioned architecture

When multiple separable layers are successively stacked, they form a highly structured neural network which will be referred as a partitioned architecture, or more succinctly *PSNet*. The Figure 5.3 illustrates a simplified PSNet architecture composed of the same two convolutional layers, similarly to the Figure 5.1, followed by three separable layers. Input and output configurations remain consistent with Figure 5.1. By concentrating the weight overlaps within the first layer, both $\frac{n_j}{n}$ and $\frac{n_{p,j}}{n}$ become smaller.

The loss function $\mathcal{L}^{\text{PSL}}(X, Y; w)$ achieves minimal common weights when all layers beyond the first one are separable. In this ideal scenario, $U_{p_1,j_1} U_{p_2,j_2}^{\top} = 0$ holds true whenever $p_2 \neq p_1 \neq j_2$ and $p_2 \neq j_1 \neq j_2$. Moreover, each score and element loss function are parametrized respectively by $\frac{n}{C}$ and $\frac{2n}{C}$ variables, representing the smallest fractions n_j and

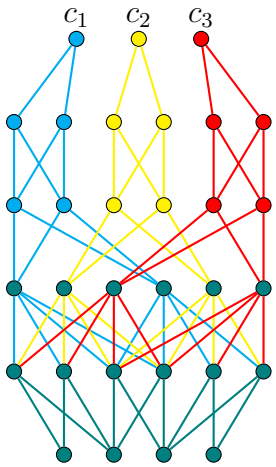


Figure 5.3 Weight dependencies of simplified PSNet scores

$n_{p,j}$ can respectively attain. When the number of classes grows, the count of element loss functions $N = C^2 - C$ increases while $\frac{n}{C}$ and $\frac{2n}{C}$ decrease if the architecture conserves identical element functions $\widehat{h}_{p,j}$. Additional insights of the new architecture PSNet, can be found in Section 5.2.4, e.g., practical values for n_j, $n_{p,j}$, $\frac{n_j}{n}$ and $\frac{n_{p,j}}{n}$.

5.2.4 Numerical results

This section provides comparisons between combinations of a LeNet or a PSNet architecture with either the $\mathcal{L}^{\text{NLL}}$ or the $\mathcal{L}^{\text{PSL}}$ loss function. All pairs of architecture and loss function are trained using the same optimizer Adam [93], defined in Algorithm 2.2.1. Adam trainings consider minibatches of size 20 or 100. The trainings run on an Nvidia A100 Tensor Core GPU and involve relatively small architectures. The subsequent results serve as a proof of concept to demonstrate the effectiveness of PSNet and $\mathcal{L}^{\text{PSL}}$.

Our focus centers on two specific datasets: MNIST [97] and CIFAR10 [94], both encompassing ten distinct classes, i.e. $C = 10$. MNIST comprises grayscale images of handwritten digits, each having 28×28 pixels. Conversely, CIFAR10 consists of color images, i.e., three channel inputs, each of 32×32 pixels.

The architectures presented in both figures are constructed using Knet.jl [156], while the datasets are sourced from MLDatasets.jl [95]. The Figures 5.4 to 5.7 compile the training of architecture-loss pairs: PSNET-NLL, PSNET-PSL, LeNet-NLL and LeNet-PSL. Each curve displayed is the mean of 10 trainings surrounded by its respective standard deviation error. The code generating those numerical results is detailed in Section 6.7.

In Figures 5.4 and 5.5, both architectures are tailored for the MNIST dataset. The LeNet

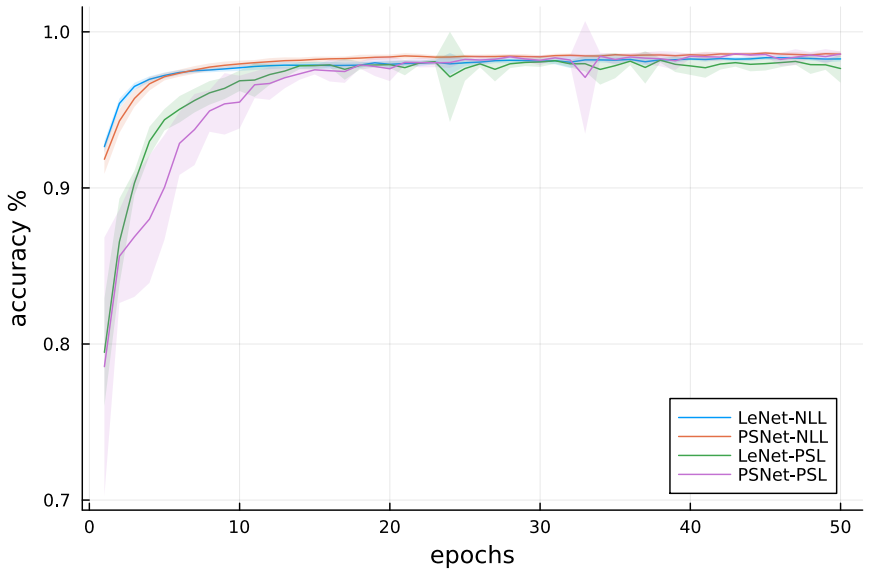


Figure 5.4 LeNet and PSNet training accuracies over epochs on MNIST, considering mini-batches of size 20.

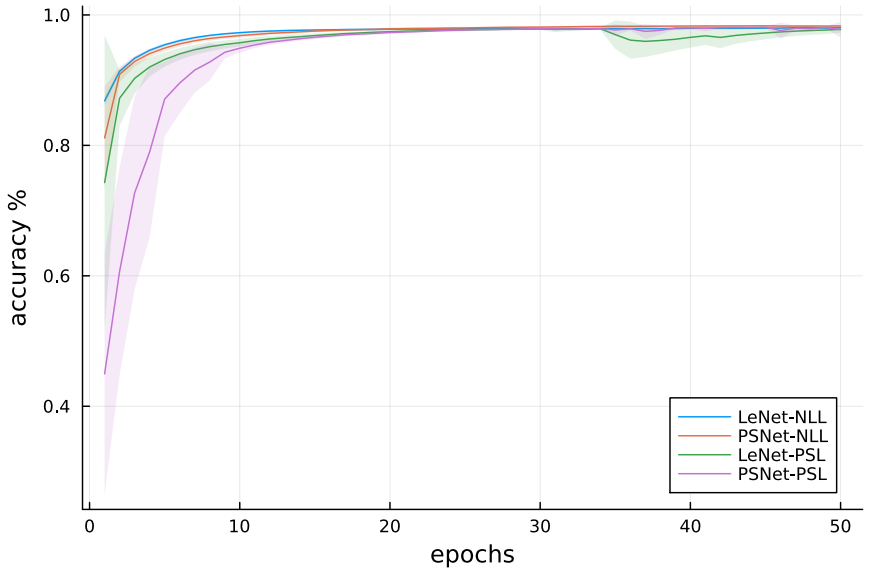


Figure 5.5 LeNet and PSNet training accuracies over epochs on MNIST, considering mini-batches of size 100.

architecture comprises two convolutional layers followed by three dense layers. Both convolutional layers employ 5×5 kernels and incorporate average pooling. The first convolutional layer employs 6 kernels, while the second employs 16 kernels. The subsequent fully connected layers contain respectively 120, 84, and 10 neurons.

On the other hand, PSNet consists of two convolutional layers followed by three separable layers. Both convolutional layers use 5×5 kernels and include average pooling. The first convolutional layer employs 40 kernels, while the second employs 30 kernels. As a result, the separable layers consist of 240, 150, and 10 neurons respectively.

The LeNet architecture is parameterized by 44426 weights, while the PSNet architecture is parameterized by 53780 weights. In PSNet, each score (resp. element loss) relies on 6340 (resp. 11588) weights. All PSNet scores have 1092 common weights at the root of the neural network. The Section 5.2.4 presents succinctly the specifics of both LeNet and PSNet architectures for the MNIST dataset.

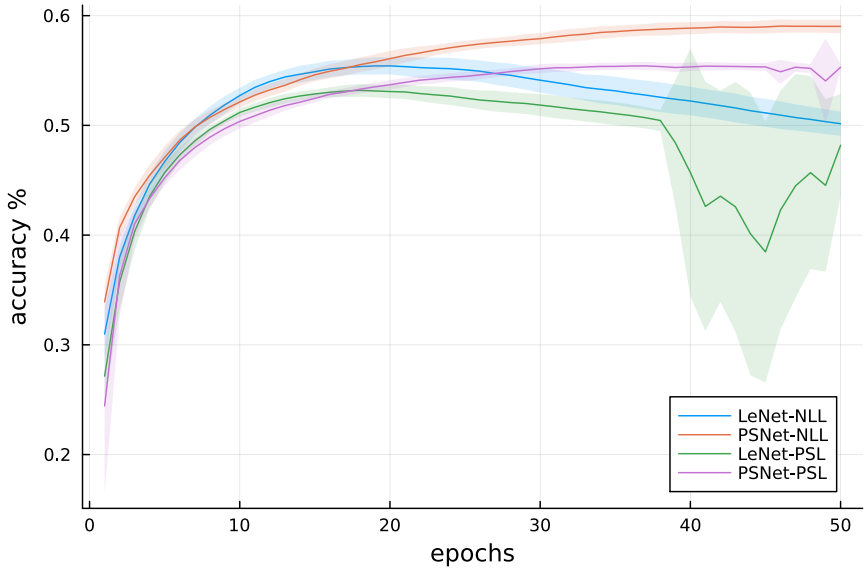


Figure 5.6 LeNet and PSNet training accuracies over epochs on CIFAR10, considering minibatches of size 20.

In Figures 5.6 and 5.7, the architectures depicted in the first figure have been modified to accommodate the input size for the CIFAR10 dataset. Both LeNet and PSNet have adjusted their first convolutional layers to handle three-channelled images. Additionally, the three dense layers of LeNet have been adapted to consist of 200, 100, and 10 neurons, respectively. In the case of PSNet, the number of kernels in the second convolutional layer has been increased from 40 to 60. Furthermore, the separable layers have been adjusted to be parameterized by 350, 150, and 10 neurons. With these adaptations, the modified LeNet and PSNet architectures are respectively parameterized by 103882 and 81750 weights. In PSNet, each score (resp. element loss function) depends on 12279 (resp. 19998) weights and all element loss functions share 4560 common weights. The Section 5.2.4 summarizes the specifics of all architectures.

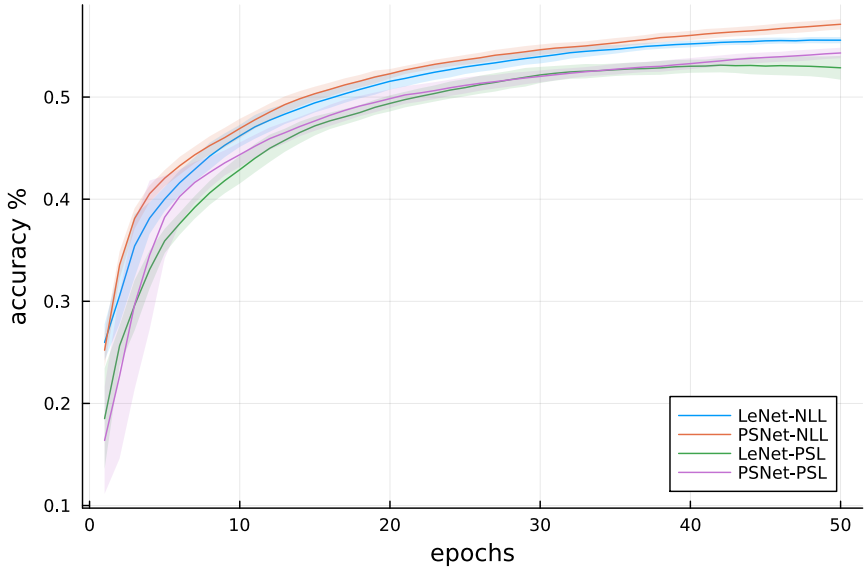


Figure 5.7 LeNet and PSNet training accuracies over epochs on CIFAR10, considering minibatches of size 100.

Table 5.1 Architecture details

LeNet			PSNet		
type	MNIST	CIFAR10	type	MNIST	CIFAR10
Conv	$5 \times 5 \times 1 \times 6$	$5 \times 5 \times 3 \times 6$	Conv	$5 \times 5 \times 1 \times 30$	$5 \times 5 \times 3 \times 30$
Conv	$5 \times 5 \times 6 \times 16$	$5 \times 5 \times 6 \times 16$	Conv	$5 \times 5 \times 30 \times 40$	$5 \times 5 \times 30 \times 60$
Dense	256×120	400×200	Separable	480×240	750×350
Dense	120×84	200×100	Separable	240×150	350×150
Dense	84×10	100×10	Separable	150×10	150×10
n	44426	103882	n	53780	81750
$n_j, n_{p,j}$	-	-	$n_j, n_{p,j}$	$6340, 11588$	$12279, 19998$
$\frac{n_j}{n}, \frac{n_{p,j}}{n}$	-	-	$\frac{n_j}{n}, \frac{n_{p,j}}{n}$	$0.11, 0.21$	$0.15, 0.24$

The comparison results presented in Figures 5.4 and 5.5 showcase the accuracy—the percentage of correct predictions on the test dataset—of the loss function PSL or NLL to train LeNet and PSNet onto the MNIST dataset. Regardless of the architecture employed, any training using PSL starts off slower compared to NLL, especially when using the minibatches of size 20. However, over time, PSL trainings tend to catch up with NLL trainings and approach asymptotically a similar performance. The standard deviation of training results is higher for PSL compared to NLL, a pattern that is consistent across both datasets, as demonstrated in Figures 5.6 and 5.7. The Figures 5.6 and 5.7 display the same comparisons for the CIFAR10 dataset. While the ranking between loss functions remains consistent, the architecture used

plays a more significant role than the chosen loss function. This phenomenon is noticeable for minibatches of size 100 and is particularly evident for minibatches of size 20. Nonetheless, for minibatches size of 20, the training of LeNet-PSL exhibits overfitting with a high standard deviation error. Raw results indicate consistent drops in accuracy (down to 20%) that are regained after a few epochs, recovering a stable 50% accuracy. Solving this overfitting issue is a topic for future research, but note that regularization techniques induce big changes in the partially-separable structure of the training, see for example Section 5.2.6.

Given that PSNet is composed of fewer weights than LeNet (81750 vs. 103882), it is surprising to witness the superiority of PSNet over LeNet. This difference can be attributed to the kernels added on the convolutional layers of PSNet, which are cornerstones of the computer vision current success. The addition of those kernels compensate the dimension growth of LeNet to accommodate CIFAR10. Separable layers indirectly induced this addition, since the accommodation to CIFAR10 required far fewer weights than for dense layers.

5.2.5 Parallel partitioned architecture computations

This section outlines a scalable strategy to efficiently distribute the computation of a partitioned neural network training using a partially-separable loss function. A straightforward approach is to allocate a single element function $\widehat{h}_{p,j}$ to each available worker. Since $\mathcal{L}^{\text{PSL}}$ consists of N element functions (5.1a), it ideally requires N workers. Each worker needs memory proportional to $\Theta(n_{p,j})$ to store its respective element function, which could be significantly smaller than n if separable layers are properly employed. During the forward pass, a worker computes $\widehat{h}_{p,j}$, and during the reverse pass, it computes $\nabla\widehat{h}_{p,j}$. Subsequently, the worker sends its computed contributions to the master node(s). The master node(s) maintain a vector g of size n, which is initially zeroed, and accumulate worker contributions. Each worker's contribution, of size $n_{p,j}$, is scattered across g based on $U_{p,j}$. If the total network's size n is too large to be accommodated by a single hardware unit, the vector g can be managed by multiple master nodes. For instance, $C+1$ master nodes can be utilized, with one storing the common weights to all scores, while the C remaining masters store the distinct weights of each score c_j. Additionally, the master nodes are responsible for updating iteratively w before transmitting $U_{p,j}w$ to the workers.

The straightforward strategy distributing every element loss function might saturate the available devices due to the quadratic growth of N with respect to C. The symmetry of partitioned architectures provides a solution to this issue by allowing a worker to easily switch the element function it manages. This symmetry arises from the fact that all scores c_j share the same underlying function but are parametrized by different U_j matrices. Consequently,

the results of two different element loss functions are governed by the same element loss function $\widehat{h}_{p,j}$, while being driven by distinct $U_{p,j}$ linear operators, i.e., $\widehat{h}_{p_1,j_1}(X, Y, U_{p_1,j_1}w) = \widehat{h}_{p_2,j_2}(X, Y, U_{p_1,j_1}w)$ where $(p_1, j_1) \neq (p_2, j_2)$. Hence, by substituting the loaded weights $U_{p_1,j_1}w$ with $U_{p_2,j_2}w$, a worker operates $\widehat{h}_{p_2,j_2}$ instead of $\widehat{h}_{p_1,j_1}$ and can therefore compute several multiple element loss functions and their derivatives. This trick significantly reduces the number of workers needed and facilitates the utilization of computational resource.

An essential point to consider is that there is no need to compute the element loss $\widehat{h}_{p,j}$ for the observations $x^{(l)}$ that are not labelled as $p \neq y^{(l)}$. Therefore, the data required for a worker only represents a fraction $\frac{1}{C}$ of the entire training dataset. Additionally, several element functions may be merged, for instance, a worker can cumulate $\widehat{h}_{p_1,j_1}$ and $\widehat{h}_{p_2,j_2}$. In that case, the neural network subpart handled by the worker may expand, and the labelled observations that the worker consider will combine p_1 and p_2. The only merge that does not augment the dimension is $\widehat{h}_{p,j} + \widehat{h}_{j,p}$, as the weights of these two element loss functions completely overlap. In the case where more than N workers are available, the same element function may be duplicated across multiple workers. In this scenario, each worker evaluates a disjoint subset of data to avoid duplicate computation. This parallel scheme based on $\mathcal{L}^{\text{PSL}}$ and a partitioned architecture can be combined with data parallelism, model parallelism, and tensor parallelism to further distribute computation.

The Figure 5.8 illustrates these notions for the PSNet example from the Figure 5.3. It denotes by $\mathbf{M}_i$ the i-th master node and by $(X_p^{(i)}, Y_p^{(i)})$ the $i - th$ minibatch from the dataset subset $(X_p, Y_p) \subset (X, Y)$ containing only the data labelled as p, i.e., $y = p,\ \forall y \in Y_p$.

Although the proposition of this section is interesting, this scheme was not tested in practice. It would have required a framework able to access several distant workers with the ability of evaluating the communication. Without such requirements, this proposition remains a theoretical idea to distribute neural network computation.

5.2.6 Dropout consequences

The introduction of techniques like the dropout [142] can enhance the partial separability of the loss function. Dropout randomly deactivates some weights during the training, affecting dynamically the loss function's structure. For a partially-separable loss function, the dropout impacts the subset of weights that each element function depends on. In an extreme case, where an entire layer is temporarily dropped out, the partitioned network may exhibit a separable behavior. The Figure 5.9 illustrates onto PSNet (Figure 5.3) the consequences of the dropout when it is applied on the layer encompassing all common weights. Each score c_{j_1} becomes independent of all other scores c_{j_2} ($U_{j_1}U_{j_2}^{\top} = 0$ for all $j_2 \neq j_1$) leading to

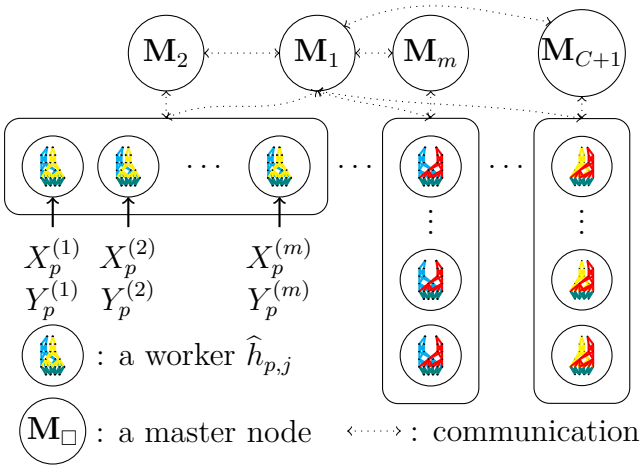


Figure 5.8 Partitioned neural network paired with $\mathcal{L}^{\text{PSL}}$ computation distribution

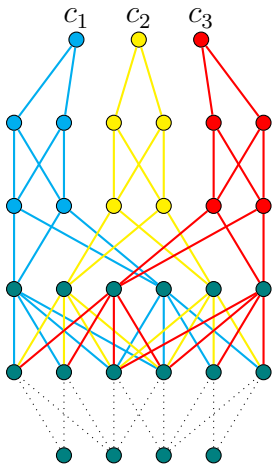


Figure 5.9 Weight dependencies of PSNet scores when the first convolutional layer is dropped out (dotted deactivated weights)

non-overlapping element functions; therefore $U_{p_1,j_1}U_{p_2,j_2}^\top = 0$ as long as $p_2 \neq j_1 \neq j_2$ and $p_2 \neq p_1 \neq j_2$. In the case where the complete neural network can be accommodated in a single hardware, all $\nabla\widehat{h}_{p,j}$ can be evaluated with fewer reverse passes.

5.3 Limited-memory partitioned quasi-Newton training

Second-order methods differ from gradient-based methods, which update weights with the sampled loss gradient $\nabla\mathcal{L}(X, Y; w)$, without seeking an approximation of $\nabla^2\mathcal{L}(X, Y; w)$. Quasi-Newton methods seek to mimic the Hessian $B \approx \nabla^2\mathcal{L}(X, Y; w)$ by updating B along iterations to minimize a quadratic approximation of $\mathcal{L}$. In this section, $\mathcal{L}^{\text{PSL}}(X, Y; w)$ is

minimized by a limited-memory partitioned quasi-Newton method. Unlike gradient based methods, a quasi-Newton method needs to compute a gradient difference at each iteration. Therefore, two gradient evaluations are needed for the same minibatch, e.g. (L)BFGS needs $y_k = \nabla\mathcal{L}^{\text{PSL}}(X, Y; w_{k+1}) - \nabla\mathcal{L}^{\text{PSL}}(X, Y; w_k)$, otherwise the meaning of y_k is lost.

Unlike quasi-Newton methods, any partitioned quasi-Newton method needs to compute the partitioned gradient, i.e., the computation of every element function gradient $\nabla\widehat{h}_{p,j}$. Unfortunately, the automatic differentiation (or backpropagation) engine employed to compute $\nabla\mathcal{L}^{\text{PSL}}$ is unable to store distinctly the contributions of all $\nabla\widehat{h}_{p,j}$ to $\nabla\mathcal{L}^{\text{PSL}}$. Practically, the computation of all $\nabla\widehat{h}_{p,j}$ falls backs on the computation of $N = C(C-1)$ element function gradients, each of which has a similar cost than computing $\nabla\mathcal{L}^{\text{PSL}}$. In other words, for a MNIST or a CIFAR10 architecture ($C = 10$), computing all $\nabla\widehat{h}_{p,j}$ costs $N = 90$ gradient evaluations of $\nabla\mathcal{L}^{\text{PSL}}$. As the gradient computation is usually the most computationally intensive operation of a training, a partitioned quasi-Newton training is at least N times slower than a gradient based method (by iteration). For example, one limited-memory partitioned quasi-Newton training applied on the MNIST's PSNet architecture for 100 epochs and with minibatches of size 100, as described in Section 5.2, takes 50 hours. Consequently, 500 GPU hours are needed for computing the mean of one training.

Although previous chapters focus on trust-region methods, here, the limited-memory partitioned quasi-Newton training implements an inexact line search (see Section 2.1.1). Initially, a trust-region method such as Algorithm 4.1.1 was exploited. However, early numerical results for trust-region methods show that if successive steps s_k were rejected, i.e. $\rho_k < 0$, then the trust-region radius shrinks and prevents larger subsequent steps. This may become an issue when considering the stochastic noise introduced by the minibatch evaluation, which may prevail on the (element) gradients real difference. A line search as no constraint such as $\|s\| \leq \Delta_k$, and therefore, is more likely to accept larger steps and resolve this trust-region issue. By doing so, it best exploits the computation of each partitioned gradient. The Algorithm 5.3.1 outlines an inexact line search method minimizing a partially-separable function f by exploiting a limited-memory partitioned quasi-Newton approximation of $\nabla^2 f$.

In order to avoid the introduction of too many notation indices in the limited-memory partitioned quasi-Newton line search algorithm, the following notation is employed:

- f and ∇f correspond respectively to the sampled loss function $\mathcal{L}^{\text{PSL}}(X, Y; w)$ and its gradient $\nabla\mathcal{L}^{\text{PSL}}(X, Y; w)$;
- $\widehat{f}_i$ refers to one element loss function $\widehat{h}_{p,j}$, and consequently U_i refers to $U_{p,j}$;
- $\widehat{y}_{i,k} = \nabla\widehat{f}_i(w_{k+1}) - \nabla\widehat{f}_i(w_k) = \nabla\widehat{h}_{p,j}(X, Y; U_{p,j}w_{k+1}) - \nabla\widehat{h}_{p,j}(X, Y; U_{p,j}w_k)$ computed

on the same minibatch X, Y;

which is similar to the Chapters 3 and 4 and the Algorithm 4.1.1. The detection of the element variables composing each $\widehat{h}_{p,j}$ is briefly detailed in Figure 6.4.

Algorithm 5.3.1 Inexact Partitioned Quasi-Newton Line Search

Choose $w_0 \in \mathbb{R}^n$, $\tau_1, \tau_2 \in \mathbb{R}^+$,
Choose $B_0 = B_0^\top = \sum_{i=1}^N U_i^\top \widehat{B}_{0,i} U_i \approx \nabla^2 f(w_0)$.
$k = 0$
repeat
    Compute an inexact solution d_k

$$\min_{d_k} m_k(d_k) = f(w_k) + \nabla f(w_k)^\top d_k + \tfrac{1}{2} d_k^\top B_k d_k$$

by using the conjugate gradient, such that $\|B_k d_k + \nabla f(w_k)\|_2 \leq \tau_1$.
    Perform a line search to retrieve $\alpha > 0$ such that

$$f(w_k) - f(w_k + \alpha s_k) \geq \alpha \tau_2 \nabla f(w_k),$$

for example Algorithm 2.1.1
    set $w_{k+1} = w_k + \alpha_k d_k$
    update every $\widehat{B}_{i,k}$ given $\widehat{s}_{i,k} = \alpha_k U_i d_k$ and $\widehat{y}_{i,k}$ satisfying $\widehat{B}_{i,k+1} \widehat{s}_{i,k} = \widehat{y}_{i,k}$ (4.2).
    $k = k + 1$
until convergence

5.3.1 Numerical results

In Figures 5.10 and 5.11, similarly to the Section 5.2.4, each curve represents the mean accuracy while its surrounding shaded region is its standard deviation error. Again, the results are produced in Julia [5], with gradient-based method implementations from Knet.jl [156]. All methods run on an Nvidia A100 Tensor Core GPU, using minibatches of size 100. MNIST trainings run for 50 epochs whereas CIFAR10 trainings run for 100 epochs.

The Figures 5.10 and 5.11 illustrate the comparison between several optimizers: SGD, Adam, LBFGS, PLBFGS, PLSR1 and PLSE. Adam (resp. SGD) learning rate is fixed at 0.001 (resp. 0.025), while $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All are employed to minimize $\mathcal{L}^{\text{PSL}}$ using a PSNet architecture over MNIST and CIFAR10 datasets. In this setup, Adam achieves the best accuracy, followed closely by PLSR1 and PLBFGS, and then finally LBFGS and SGD. SGD's slower convergence can be attributed to its small learning rate, which is chosen to prevent numerical instability during gradient computations.

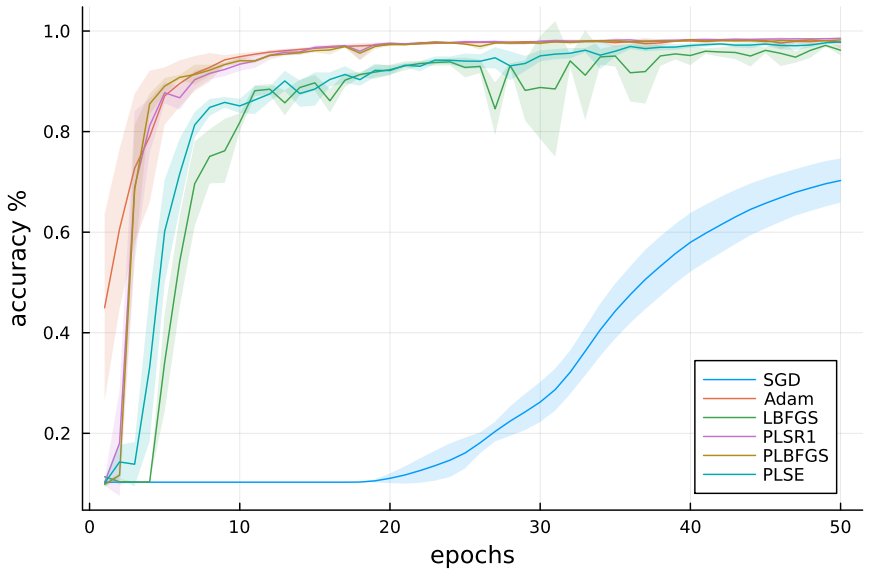


Figure 5.10 Comparison of optimizer accuracies over epochs during PSNet training when minimizing $\mathcal{L}^{\text{PSL}}$ (5.1a) on MNIST.

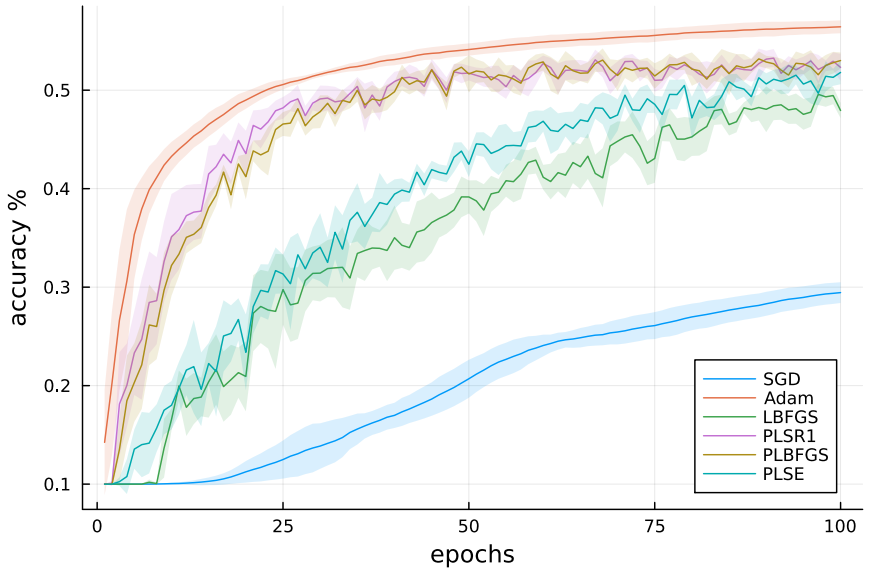


Figure 5.11 Comparison of optimizer accuracies over epochs during PSNet training when minimizing $\mathcal{L}^{\text{PSL}}$ (5.1a) on CIFAR10.

5.3.2 A parallel partitioned Hessian-vector product

This section embeds a limited-memory partitioned quasi-Newton approximation of $\nabla^2\mathcal{L}^{\text{PSL}}$ into the parallel scheme presented in Section 5.2.5. The partial separability of $\mathcal{L}^{\text{PSL}}$ structures

heavily the Hessian:

$$\nabla^2 \mathcal{L}^{\text{PSL}} = \sum_{p=1}^{C} \sum_{j=1 \neq k}^{C} U_{p,j}^\top \nabla^2 \widehat{h}_{p,j} U_{p,j}.$$

While directly computing and storing each element loss function Hessian might be impractical due to its size, calculating the Hessian-vector product remains accessible. In this scenario, the master node(s) aggregate the workers contributions $\nabla^2 \widehat{h}_{p,j}(U_{p,j} w) U_{p,j} v$ computed from a reverse-forward pass of automatic differentiation, see Section 2.1.5 and Section 3.5.

When every $\nabla^2 \widehat{h}_{p,j}$ is approximated by a limited-memory quasi-Newton operator, then

$$Bv = \sum_{p=1}^{C} \sum_{j=1 \neq k}^{C} U_{p,j}^\top \widehat{B}_{p,j} U_{p,j} v, \quad \widehat{B}_{p,j} \approx \nabla^2 \widehat{h}_{p,j}.$$

Similarly to the Hessian-vector product, this approach involves that each worker computes $\widehat{B}_{p,j} U_{p,j} v$ and sends its contributions to master(s) which will aggregate them using $U_{p,j}^\top$. Consequently, the conjugate gradient method [84] needs master(s) to synchronize all workers contributions at every iteration to compute effectively Bv.

5.4 Limitations and open problems

This chapter introduced three new concepts: a partially-separable loss function, a partitioned architecture and a limited-memory partitioned quasi-Newton training. Except for a partially-separable loss function which easily integrates in most neural-network modelling libraries, the two other concepts require consequent efforts for an effective implementation.

First, a partitioned architecture leverages a new scheme for parallel computation when the loss function is partially-separable. Specifically, in a computer vision context, each score is parametrized by a subset of neural-network weights. By combining a couple of scores, each element loss function requires only a fragment of the neural network to operate. Therefore, the computational effort related to the loss function or its derivatives can be distributed to working units, each managing one element function, i.e., a neural network fragment. Test this parallelization procedure requires access to one worker for each element function, e.g. $N = 90$ workers for MNIST and CIFAR architectures, as well as a complete framework aggregating worker results and measuring communication costs. Due to the lack of computational resource at our disposal and probably of expertise, those aspects have not been studied yet. Still, two particular setups are relevant depending on the computational capacity of the workers. If each worker is a powerful device, e.g. a GPU, then the partial separability parallelization should be compared with the other parallelization schemes presented in Section 2.2.6. If the workers

are countless less powerful devices, which is a federated learning context, then each worker can: accommodates the minibatch size it considers computing the element loss function and its derivatives, see Figure 5.8, and it can evaluate autonomously its neural-network fragment before sending its contributions to master(s). For both setups, the scalability and the communication must be evaluated and compared with state-of-the-art implementations.

Second, a limited-memory partitioned quasi-Newton training requires the computation of all element loss gradients. As mentioned in Section 5.2, the neural-network modelling libraries do not return the element loss function gradient contributions from the computation of the loss function gradient. Therefore, the solution implemented computes individually each element gradient at roughly the same cost of the total gradient, making the partitioned gradient cost N times that of the loss function gradient. Although the loss function gradient do not compute the element gradient contributions, they could be propagated with an adequate storage. Separate storage for element contributions is feasible, but would multiply the memory needed depending on the number of element functions contributing to each weight. Finally, the indexing operations on vectors or matrices are not scalable on GPU and practically slow down the efficiency of the GPU parallelization. Therefore, the two Julia modules `PartitionedStructures.jl` and `PartitionedVectors.jl`, described in the next chapter, should remove indexing operations from their routines to best scale on GPU.

These are the two major issues. Their resolutions could transform the contributions of this chapter into practically competitive training methods.

CHAPTER 6 Software packages

This chapter details the implementation of the methods defined in Chapter 4 and in Chapter 5. The Section 6.1 provides some information about the programming environment and some metadata on the code developed. The Section 6.2 gives an overview of the JuliaSmoothOptimizers (JSO) ecosystem, which proposes several tools for numerical continuous optimization. In particular, the Sections 6.3 to 6.6 explain how the (limited-memory) partitioned quasi-Newton methods are integrated in JSO. More specifically, the partially-separable structure is automatically detected by using the work from the Section 6.3. Then, the Sections 6.4 and 6.5 define the partitioned data structures needed for running a partitioned quasi-Newton minimization method, as well as the related operations mandatory for both Algorithm 3.2.1 and Algorithm 4.1.1. Finally, the Section 6.6 articulates the features developed in previous sections to provide a partitioned quasi-Newton model exploiting the partially-separable structure whenever it is advantageous.

The remaining sections are about the training of (partitioned) neural networks. The Section 6.7 presents an interface that formulates a neural network as an optimization model compatible with JSO's tools. Lastly, the Section 6.8 extends the contribution described in Section 6.7 to enable a limited-memory partitioned quasi-Newton training, i.e., an implementation of the Algorithm 5.3.1.

6.1 Programming environment and metadata

All the code presented in this chapter is implemented in Julia, a high-level programming language specifically designed for scientific computing [5]. Julia is known for its natural multi-precision support and its compatibility with interface-oriented code architectures. Unlike other popular high-level programming languages such as Python or Matlab, pure Julia code can achieve performance comparable to that of C/C++ [103].

Additionally, all the code developed by the JSO community is open source and follows continuous integration standards, which include: a documentation generated automatically, validation by tests, and pull-request review(s) before integrating significant code changes. The Table 6.1 summarizes for each Julia module some information: version, percentage of code covered by tests, number of lines and its related section.

Table 6.1 Julia modules metadata

Name	version	code coverage	lines	Section
ExpressionTreeForge.jl	0.1.6	81%	8000	6.3
PartitionedStructures.jl	0.1.6	97%	7000	6.4
PartitionedVectors.jl	0.1.2	99%	2000	6.5
PartiallySeparableNLPModels.jl	0.3.5	97%	4000	6.6
KnetNLPModels.jl :	0.2.3	79%	2 000	6.7
PartitionedKnetNLPModels.jl :	0.0.1	≈30%	2 000	6.8

6.2 JuliaSmoothOptimizers (JSO) ecosystem architecture

The JSO ecosystem gathers tools to democratize nonlinear optimization methods, to solve for example:

$$\min_{x \in \mathbb{R}^n} f(x).$$

This section discusses how the partial separability of f is integrated in the JSO ecosystem. In particular, how the partial separability knowledge is incorporated into the JSO models and the JSO multi-precision solvers. See Figure 6.1 for a graphical overview of the module dependencies within JSO.

A key component of the JSO ecosystem is NLPModels.jl [120], which provides an abstract interface for optimization models employed to implement JSO solvers. Any model that conforms to the NLPModel's interface must implement several methods, evaluating: the objective function, the gradient, the Hessian-vector product, or alternatively, the linear application $v \to B_k v \approx v \to \nabla^2 f(x_k)v$. In our context, the simplest type of model will be referred as *pure Julia's model*. Such a model considers a Julia function as the objective and computes derivatives with automatic differentiation techniques [73].

All JSO solvers based on the NLPModels's interface are gathered in JSOSolvers.jl [121]. The code of these solvers is generic, i.e., they minimize any model that respects the NLPModels's interface while leveraging its distinct features. The Sections 6.3 to 6.6 focus on TRUNK, a trust-region solver from JSO that closely aligns with Algorithm 2.1.2. The main difference is that a backtracking line search (Algorithm 2.1.1) is performed along the direction s determined by (2.4) when ρ_k is too small to accept $x_k + s$ as the next iterate. Nonetheless, the tools developed in Sections 6.3 to 6.6 are designed to be applicable to other JSO solvers as well.

By default, when minimizing a pure Julia's model with TRUNK, the exact Hessian is used. Therefore, TRUNK implements a Newton trust-region method. An alternative option for a pure Julia's model is to incorporate a LBFGS or a LSR1 operator to approximate

the Hessian [119, 122] (see the Section 2.1.3). This modification overwrites the Hessian operations and utilizes the limited-memory quasi-Newton operator instead. Consequently, the limited-memory quasi-Newton model specializes TRUNK into a limited-memory quasi-Newton trust-region solver.

The contribution from the Sections 6.3 to 6.6 is to detect automatically the partially-separable structure of a pure Julia's model to formulate a partially-separable model. By default, a partially-separable model uses $B_k = \nabla^2 f(x_k) = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_{i,k}) U_i$ by computing only the derivatives of the element functions, which reduces the computational effort to access $\nabla^2 f$ or $v \to \nabla^2 f v$. In addition, the user may incorporate one of the various partitioned quasi-Newton operators: PBFGS, PSR1, PSE, PCS, or the limited-memory variants: PLBFGS, PLSR1 and PLSE presented in Section 4.1, providing sparse $B_k \approx \nabla^2 f(x_k)$ or $v \to B_k v \approx v \to \nabla^2 f(x_k) v$. Partially-separable models not only overwrite the Hessian operations, but other NLPModel's methods that could benefit from partial separability, e.g., computing f or ∇f. As a result, TRUNK is specialized in as many partitioned quasi-Newton solvers as there are partitioned quasi-Newton models.

In order to ensure efficiency without sacrificing flexibility, JSO solvers pre-allocate all data structures to avoid runtime allocations. In particular, it requires a type deriving from `AbstractVector` that must store x_k, while assuming that the same type will also store $\nabla f(x_k)$, s_k, and y_k as defined in Algorithm 2.1.2. However, storing x_k (usually a `Vector`), $\nabla f(x_k)$ (the partitioned gradient), and y_k (the partitioned gradient difference) with a single data structure is quite challenging. Furthermore, the same data structure must parametrize the linear application $v \to B_k v$ and behave properly with the Krylov.jl's implementation of the truncated conjugate gradient method [111], which has its own solver structure. Those constraints led to the creation of the type `PartitionedVector`, deriving from `AbstractVector`, which behaves like a `Vector` for x_k and s_k while keeping track of element contributions for $\nabla f(x_k)$ and y_k. A `PartitionedVector` matches the needs of the NLPModel's interface and supports all operations required by TRUNK and the truncated conjugate gradient solvers. In practice, any partially-separable model properly instantiates the `PartitionedVectors`, the partitioned quasi-Newton operator and all solver structures needed for allocating and running both TRUNK and the truncated conjugate gradient methods.

The remainder of this section enumerates the JSO modules developed during this thesis that allow TRUNK to exploit partial separability:

- ExpressionTreeForge.jl [131], described in Section 6.3, automatically detects the partially-separable structure from a NLPModel. It supports pure Julia's models, JuMP mod-

els [88][1], as well as the native Julia type `Expr`;

- PartitionedStructures.jl [132], elaborated in Section 6.4, describes the partitioned structure implementations related to partitioned quasi-Newton approximations or partitioned gradients;
- PartitionedVectors.jl [133], detailed in Section 6.5, introduces the `PartitionedVector` type and its related methods, which are essential for TRUNK and the truncated conjugate gradient solvers. `PartitionedVector` and its related methods are based on PartitionedStructures.jl.
- PartiallySeparableNLPModels.jl [7] combines these modules (see Figure 6.1) to: detect partial separability, allocate properly the partitioned data structures and exploit partial separability to compute derivatives. As a result, it encompasses all the (limited-memory) partitioned quasi-Newton models with respect to the NLPModel's interface.

As a user, running TRUNK on any model from PartiallySeparableNLPModels.jl performs a trust-region method exploiting partial separability.

6.3 ExpressionTreeForge.jl

ExpressionTreeForge.jl is a toolbox dealing with an expression tree, which, in our unconstrained optimization context, represents the objective function. Each of ExpressionTreeForge's algorithms is based on a tree interface that JuMP's models adhere to. For any supported tree, each leave is either a variable or a constant node, while intermediate nodes (including the root) represent operators. ExpressionTreeForge provides several tree-walking algorithms, illustrated by Figure 6.2 and Figure 6.3, which offer:

- an automatic detection of f partial separability, by recursively removing the additive operators from the root of the expression tree. Then, it identifies the subtrees corresponding to element functions $f_i : \mathbb{R}^n \to \mathbb{R}$;
- the inference of U_i and n_i from each f_i to define the reduced dimension element function $\widehat{f}_i(\widehat{x}_i) = f_i(x),\ \widehat{x}_i \coloneqq U_i x \in \mathbb{R}^{n_i}$. U_i selects the subset of variables appearing in f_i as a linear operator, each row of which is a Euclidean basis vector. In practice, U_i is a vector of integers indicating the indices of the variables that parameterize f_i. Consequently, the variable indices of f_i expression tree are adjusted according to U_i to construct $\widehat{f}_i$. For example, $f_i(x) = x_i x_{i+1}$ becomes $\widehat{f}_i(x_1, x_2) = x_1 x_2$ with $U_i =$ `[i, i+1]`;

[1] a popular Julia modelling language

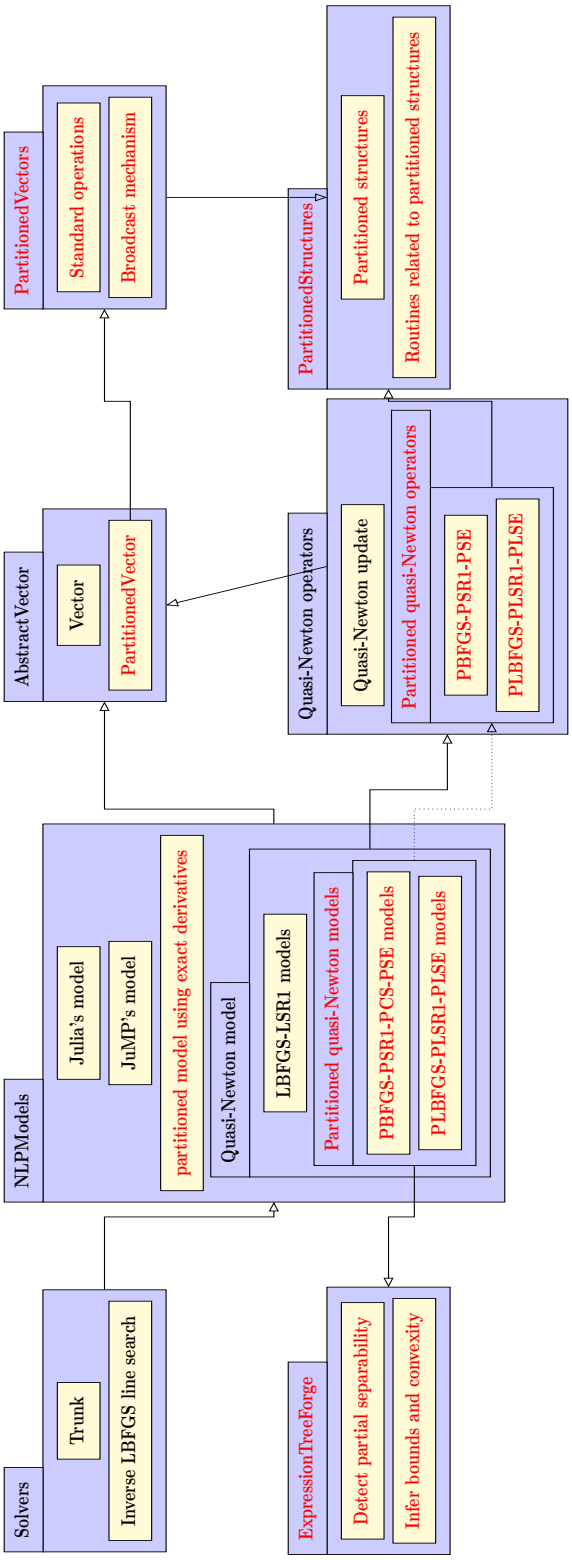


Figure 6.1 Abstract type and interface dependencies of JSO related to partial separability. The text and the boxes in red denote the code developed during the thesis.

- bound propagation and convexity detection for all nodes of an expression tree, e.g. $\widehat{f}_i$. Initially, leaf nodes are set to the bounds $(-\infty, +\infty)$ to fit the unconstrained optimization requirement. The bounds are then propagated to every intermediate node based on specific operator rules depending on the bounds of the children. Convexity detection follows a similar strategy, guided by operator rules from Fourer et al. [63], based on both children bounds and children convexity statuses. For example, suppose $f(y) = e^y$, then $f \circ g$ is convex only if g is convex [63].

The illustrations of the ExpressionTreeForge algorithms are based on the following example:

$$f(x) = \frac{(x_1 x_3)^4}{x_2^2 + 1} + \frac{(x_3 x_5)^4}{x_4^2 + 1} + \exp((x_1 + x_3 + x_5)^2), \tag{6.1}$$

for which the tree-walking algorithms are illustrated in Figure 6.2 and in Figure 6.3. Their results are:

$$\begin{gathered} \widehat{f}_1(y_1, y_2, y_3) = \widehat{f}_2(y_1, y_2, y_3) = \frac{(y_1 y_3)^4}{y_2^2 + 1}, \\ \widehat{f}_3(y_1, y_2, y_3) = \exp((y_1 + y_2 + y_3)^2), \\ U_1 = \begin{bmatrix} e_1^\top \\ e_2^\top \\ e_3^\top \end{bmatrix}, \; U_2 = \begin{bmatrix} e_3^\top \\ e_4^\top \\ e_5^\top \end{bmatrix}, \; U_3 = \begin{bmatrix} e_1^\top \\ e_3^\top \\ e_5^\top \end{bmatrix}. \end{gathered} \tag{6.2}$$

ExpressionTreeForge.jl identifies the partially-separable structure and returns all the reduced-size $\widehat{f}_i$ as well as their respective U_i. All $\widehat{f}_i$ and U_i are transmitted to ensuing modules to allocate partitioned data structures and define a partially-separable model.

6.4 PartitionedStructures.jl

PartitionedStructures.jl defines the partitioned vectors and the partitioned matrices, which are the data structures related to the partitioned nature of the derivatives (3.7). Partitioned data structures store separately either the element-vector contributions or the element-matrix contributions. A partitioned vector can also represent the linear applications $U_i v, \; \forall \; 1 \leq i \leq N$, or the result of a partitioned-matrix-vector product (more details in Section 6.5). The

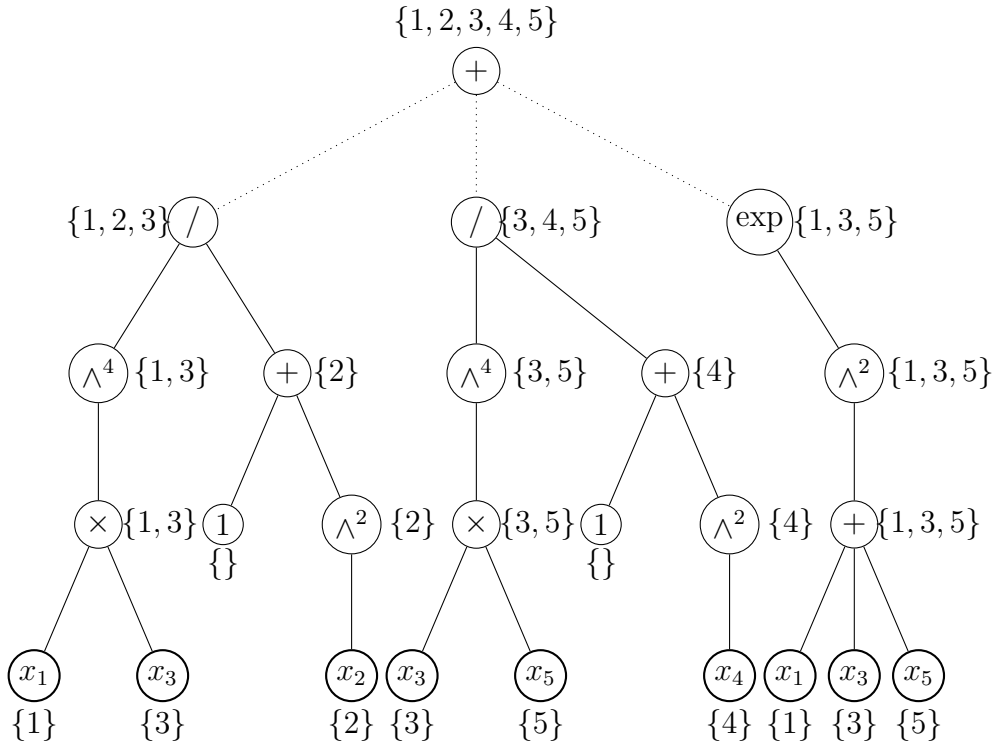


Figure 6.2 Automatic partial separability detection

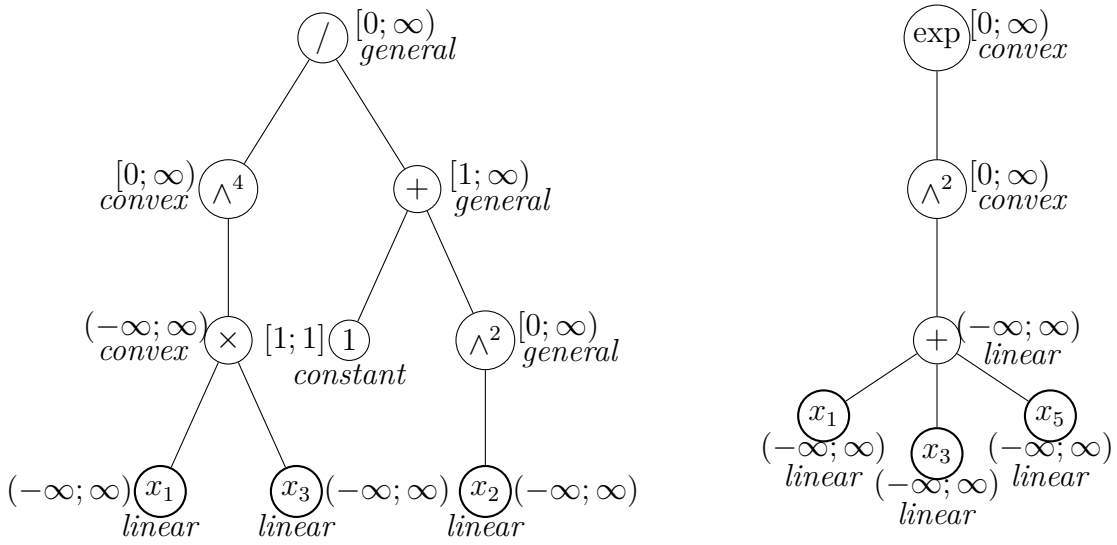
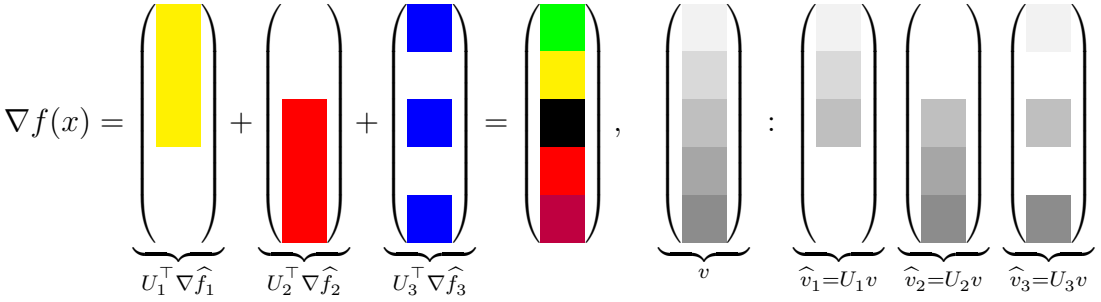
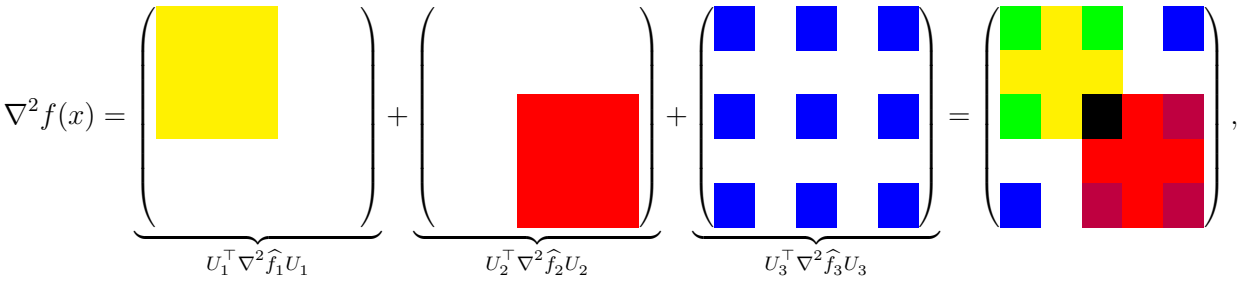


Figure 6.3 Bounds and convexity status of element functions

partitioned derivatives of f and $U_i v$ from of f (6.1) are of the form:



and



where yellow, red and blue represent respectively the contributions from $\widehat{f}_1, \widehat{f}_2$ and $\widehat{f}_3$. Other colors express the combination of several element contributions for a single partial derivative.

Additionally, PartitionedStructures.jl provides subroutines for performing: basic operations on partitioned vectors, the partitioned-matrix-vector product and the partitioned quasi-Newton updates. To facilitate their utilization, all partitioned quasi-Newton updates have a homogeneous interface, relying on a partitioned matrix and two partitioned vectors representing respectively all $\widehat{y}_{i,k} := \nabla \widehat{f}_i(\widehat{x}_{i,k+1}) - \nabla \widehat{f}_i(\widehat{x}_{i,k})$ and all $\widehat{s}_i := U_i s$.

A partitioned object (i.e. vector or matrix) consists of an ordered set of element objects and an index table that indicates which element object affects which x_j, $1 \leq j \leq n$. A partitioned vector stores a Julia `Vector` in case it must build in-place its associated vector, e.g., build ∇f from all $\nabla \widehat{f}_i$. The storage of such a `Vector` of size n is negligible as $\sum_{i=1}^{N} n_i \geq n$ by definition. Similarly, a partitioned quasi-Newton operator can aggregate element contributions to form either a SparseMatrix or a Matrix, primarily for tests. The main distinction between partitioned objects lies in the nature of their element objects, which can be: a `Vector`, a symmetric `Matrix`, or a `QuasiNewtonLinearOperator`. In addition, each element object maintains a list of variable indices, equivalent to U_i. Therefore, the memory footprint of the partitioned structures is not directly related to n like regular vectors or symmetric matrices. The memory storage required for a partitioned vector is $\Theta\left(\sum_{i=1}^{N} n_i\right)$ while a partitioned matrix takes $\Theta\left(\sum_{i=1}^{N} \frac{1}{2} n_i(n_i+1)\right)$ and its limited-memory variant needs $\Theta(\sum_{i=1}^{N} 2mn_i)$.

The simplest way to instantiate partitioned structures is by indicating the variable indices used by each $\widehat{f}_i$, i.e. U_i. This can take the form of a nested `Vector` of integer U encompassing every U_i. In the case of (6.2), for which $U_1, U_2, U_3 : \mathbb{R}^5 \to \mathbb{R}^3$, U can be defined as follows:

$$U = \texttt{[}U_1\texttt{,}U_2\texttt{,}U_3\texttt{]},\ \text{where } U_1 = \texttt{[1,2,3]},\ U_2 = \texttt{[3,4,5]},\ U_3 = \texttt{[1,3,5]}.$$

The value $n = 5$ can be added manually or automatically detected from the maximum value contained within all U_i. The tree walks of ExpressionTreeForge.jl detailed in Section 6.3 and displayed in Figure 6.2 provide all the information needed to create U, and therefore, any partitioned object.

6.5 PartitionedVectors.jl

PartitionedStructures.jl defines the partitioned data structures, in particular, the partitioned vectors. As PartitionedStructures.jl define an abstract type for all partitioned structures, a partitioned vector cannot be directly a subtype deriving from `AbstractVector`. Therefore, PartitionedVectors.jl wraps a partitioned vector from PartitionedStructures.jl into a new type `PartitionedVector` to fulfill the `AbstractVector` interface from Julia. Ultimately, the type `PartitionedVector` must support: in place broadcasted instructions and satisfy the quasi-Newton update interface to run properly the TRUNK and the truncated conjugate gradient methods. As TRUNK considers all $\nabla f, y$ and x, s from the Algorithm 2.1.2 as a single type deriving from `AbstractVector`, a `PartitionedVector` must have two distinct usages:

- Usage 1: storing distinct element vector values. In this case, the associated vector is built by aggregating the element vectors using $U_i^\top$. For example, the gradient vector $\nabla f = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i$ accumulates element-vector contributions $\nabla \widehat{f}_i$ with $U_i^\top$ to form ∇f.
- Usage 2: each element vector represents the linear applications U_i onto $v \in \mathbb{R}^n$, i.e., $\widehat{v}_i := U_i v,\ \forall\ 1 \leq i \leq N$. For example, if we consider U_i from (6.2) and $x = (1, 2, \sqrt{2}, \pi, 5)$, then the `PartitionedVector` of usage 2 results in $\widehat{x}_1 = (1, 2, \sqrt{2})$, $\widehat{x}_2 = (\sqrt{2}, \pi, 5)$, and $\widehat{x}_3 = (1, \sqrt{2}, 5)$. Unlike the first usage, the associated vector of the second usage does not aggregate element contributions through $U_i^\top$. Instead, if needed, the original vector v can be reconstructed by extracting every component v_j from element vectors for which $U_i e_j \neq 0$.

The first usage is mandatory to perform partitioned quasi-Newton updates, by computing $\widehat{y}_i$ from $\nabla \widehat{f}_i$, as well as storing all $\widehat{B}_{i,k}\widehat{v}_i$ results. The second usage is used to track the current

point x and step s, as well as performing element-matrix-vector product $\widehat{v}_i = U_i v$. Both usages must coexist during TRUNK and conjugation-gradient iterations. Additionally, for running TRUNK and conjugate gradient methods without additional runtime allocations, both usages necessitate in-place broadcasted methods, notably: `axby!, axpby!` and `mul!` which are discussed later.

To fulfill the `AbstractVector` interface, the indexing operation `pv[i]` returns the i-th element vector. For example, `pv[i] =` $\widehat{v}_i$ sets the i-th element vector of `pv` to the value of $\widehat{v}_i$, where $\widehat{v}_i$ is either an element vector or a `Vector` of size n_i. Additionally, basic element-wise operations such as addition, subtraction, and scalar multiplication are implemented for `PartitionedVectors`, enabling operations like `-(pv[1] + pv[1]) == -2 * pv[1]`.

A broadcasted operation applied onto a `PartitionedVector` is applied element-vector by element-vector. For example, `pv1 .= pv2` and `pv .= pv0 ; pv .= 2 .* pv ; pv[i] == 2 .* pv0[i]` are valid. However, two requirements are needed to perform properly broadcasted operations with `PartitionedVectors`, which are: all the `PartitionedVectors` involved must have the same usage (1 or 2) and an identical partitioned structure, i.e., identical U_i, $\forall i$. Broadcasted operations with `PartitionedVectors` of different usages may lead to an unclear semantic with a result that does not correspond to any meaningful partitioned operation. For example, if `partitioned_gradient` collects all $\nabla \widehat{f}_i$ (usage 1) and `Uiv` collects all $U_i v$ (usage 2), then `partitioned_gradient .+ Uiv` sets every element vector to $\nabla \widehat{f}_i + U_i v$ which is generally irrelevant.

Also, broadcasting a constant must be used with care since `pv .= 1` sets every element vector `pv[i]` to $(1, 1, ..., 1)^\top \in \mathbb{R}^{n_i}$. This makes sense for a `PartitionedVector` of usage 2, i.e. $\widehat{v}_i = U_i v$, where

$$U_i \underbrace{(1, 1, ..., 1)^\top}_{\in \mathbb{R}^n} = \underbrace{(1, 1, ..., 1)^\top}_{\mathbb{R}^{n_i}},$$

for all i. However, for a `PartitionedVector` of usage 1, operations like `.=` α or `.+` α, where α is a scalar, do not have a clear meaning in terms of partitioned operations. The only exception is `pv .= 0`, which can be useful for initializing a `PartitionedVector` to zeros before populating it with actual values, e.g., `pv = similar(pv0) ; pv .= 0`.

`PartitionedVectors` of both usages support scalar product and norm methods, which are essential for the trust-region method (see Algorithm 3.2.1 and Algorithm 4.1.1). Some of these methods, such as the scalar product between `PartitionedVectors`, may exploit partial separability depending on the usage of the `PartitionedVectors`. Let's consider $v \coloneqq \sum_{i=1}^{N} U_i^\top \widehat{v}_i$ and a Vector w represented respectfully by a `PartitionedVector` of usage 1

and 2. In such case, the scalar product $v^\top w$

$$v^\top w = (\sum_{i=1}^{N} U_i^\top \widehat{v}_i)^\top w = \sum_{i=1}^{N} \widehat{v}_i^\top \underbrace{U_i w}_{\text{usage 2}} = \sum_{i=1}^{N} \widehat{v}_i^\top \widehat{w}_i,$$

can be computed by accumulating the scalar products of every pair of element vectors `v[i]` and `w[i]`. If both v and w are of the same usage, then it is easier to calculate the scalar product directly from the vector represented by either v or w, as explained earlier. Afterwards, the scalar product computes $v^\top w$ directly on the assembled vectors. Similarly, the norm is computed from the `Vector` represented by the `PartitionedVector`.

The next critical step is how the conjugate gradient method from Krylov.jl [111] exploits the partitioned-matrix-vector product with `PartitionedVectors`:

$$Bv = \sum_{i=1}^{N} U_i^\top \widehat{B}_i U_i v.$$

Computing Bv requires first `v[i] =` $\widehat{v}_i = U_i v$, which is known if `v` is a `PartitionedVector` of usage 2. Then, the linear application of $\widehat{B}_i$ is applied to `v[i]` to get `Bv[i] .=` $\widehat{B}_i$`v[i]`. Finally, the results are aggregated into $Bv = \sum_{i=1}^{N} U_i^\top$`Bv[i]`, making `Bv` a `PartitionedVector` of usage 1. Even though `Bv` is of usage 1, the solution of the trust-region subproblem `s` is of usage 2, since it must provide $\widehat{s}_i = U_i s$ during the partitioned quasi-Newton update. To transfer `Bv` to `s` safely, we overload the `axpy!(`α`,x,y)` method used in Krylov.jl, which is equivalent to `y .= x .*` α `.+ y`, $\alpha \in \mathbb{R}$, when y is of usage 1 and x is of usage 2. The overloaded `axpy!` method builds the associated `Vectors` of both `Bv` and `s` and then applies `axpy!` on both `Vectors` to store the result in `s`.

Note that `axpy!` and `axpby!`, i.e., `y .= x .*` α `.+ y .*` β, perform more computation for `PartitionedVectors` than for `Vectors`. As a matter of fact $n \leq \sum_{i=1}^{N} n_i$, therefore, dispatching `axpy!` or `axpby!` across element vectors requires more computational effort than for a `Vector`. Additionally, the execution of a matrix-vector product does not scale linearly with the size of the data structure. Similarly, applying one given operation on N vectors of size n_i is slower than applying the same operation on a single vector of size $\sum_{i=1}^{N} n_i$.

6.6 PartiallySeparableNLPModels.jl

PartiallySeparableNLPModels.jl articulates the modules presented in Sections 6.3 to 6.5 to define a partially-separable model or a partitioned quasi-Newton model. First, ExpressionTreeForge.jl detects the partially-separable structure. Then, PartitionedStructures.jl and

PartitionedVectors.jl allocate the adequate partitioned structures. Additionally, PartiallySeparableNLPModels.jl manages the computation of partitioned derivatives with reverse automatic differentiation applied to each element function. Therefore, as mentioned in Section 3.5, PartiallySeparableNLPModels.jl detects the duplication of element functions, such as $\widehat{f}_1$ and $\widehat{f}_2$ in (6.2), to minimize the number of tapes required for automatic differentiation. As a result, only $M \leq N$ distinct element functions remain, paired with a matching between U_i and those M element functions. Additionally, if the estimated memory of the partitioned matrix becomes too large, i.e., either $\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2}$ or $\sum_{i=1}^{N} mn_i$ is larger than $\frac{1}{2}n(n+1)$, then the partially-separable structure is ignored. In practice, it means reverting to an unstructured problem where $\widehat{f}_1 = f : \mathbb{R}^n \to \mathbb{R}$ and $U_1 = I \in \mathbb{R}^{n \times n}$. By making those adjustments, the memory footprint as well as the complexity of $v \to B_k v$ is reduced.

To summarize, PartiallySeparableNLPModels.jl defines several partitioned models, each approximating differently the Hessian:

- PBFGSNLPModel, PSR1NLPModel, PCSNLPModel, or PSENLPModel use a dense quasi-Newton matrix $\widehat{B}_{i,k} \approx \nabla^2 \widehat{f}_i$ as described in Section 3.2;
- PLBFGSNLPModel, PLSR1NLPModel, or PLSENLPModel use a new limited-memory quasi-Newton operator $\widehat{v}_i \to \widehat{B}_{i,k}\widehat{v}_i \approx \widehat{v}_i \to \nabla^2 \widehat{f}_i(\widehat{x}_{i,k})\widehat{v}_i$ as introduced in Section 4.1;
- PSNLPModel exploits both reverse and forward automatic differentiation to compute separately element Hessian-vector products $\nabla^2 \widehat{f}_i(\widehat{x}_{i,k})U_i v$ before it aggregates those contributions to build $\nabla^2 f v$.

As a result, partitioned quasi-Newton models specialize TRUNK into as many partitioned quasi-Newton solvers as there are models.

6.7 KnetNLPModels.jl

6.7.1 Summary

KnetNLPModels.jl is a Julia [5] module integrated in JSO. It is designed to bridge the gap between JSO optimization solvers and the deep neural network modeling framework Knet.jl [156]. While it is possible to write and integrate solvers directly into Knet.jl, separating them into standalone packages offers advantages in terms of modularity, flexibility and ecosystem interoperability. The decoupling of the modeling tool from the optimization solvers allows users and researchers to employ a wide variety of optimization solvers, including a range of existing solvers not traditionally applied to deep network training such as R2 [10, 11].

Knet.jl, as a standalone framework, does not have built-in interfaces with general optimization frameworks like JSO. However, it allows users to model deep neural network architectures and combine them with a loss function and a dataset from MLDataset.jl [95]. In particular, it provides pre-defined neural layers, such as dense layers, convolutional layers, see Section 2.2.2, and other complex layers. Additionally, it allows users to initialize the weights using various methods, such as uniform distribution. Moreover, Knet.jl offers a wide range of loss functions, e.g., negative log likelihood (2.2.3), and provide the flexibility for users to define their own loss functions according to their specific needs. Furthermore, it enables an efficient evaluation of the neural network outputs, the sampled loss and its derivatives, on both CPU and GPU. This flexibility allows the weights to be represented as either a `Vector` (for CPU) or a `CUVector` (for Nvidia GPU), with support for multiple floating-point systems. In addition, Knet.jl facilitates the definition of minibatches as iterators over the dataset, enabling efficient batch processing during training. This can also be used to evaluate the accuracy of the trained neural network on the test dataset. Lastly, it naturally supports various stochastic optimizers, such as: stochastic gradient [96], Nesterov acceleration [112], Adagrad [50], and Adam [93], which are used to produce numerical results.

KnetNLPModels.jl adopts the triptych of architecture, dataset, and loss function to model a neural network training problem as an unconstrained optimization problem with respect to the NLPModel's interface. Consequently, those models can be solved using first- and second-order solvers from JSOSolvers.jl. However, the JSO solvers are deterministic. To palliate this issue, they integrate a callback mechanism that allows the user to change the training minibatch and its size at each iteration, producing stochastic solvers. Lastly, note that contrary to usual stochastic optimizers, all methods in JSOSolvers enforce the decrease of a certain merit function.

6.7.2 Training a neural network with KnetNLPModels.jl and JSO solvers

The following section illustrates how to train a LeNet architecture [96] using JSO solvers. Assume that `LeNet` architecture is defined in Knet.jl using $\mathcal{L}^{\text{NLL}}$ (2.2.3), the MNIST dataset is loaded from MLDataset and the minibatch loaders `data_train, data_test` exist, such as in the KnetNLPModels.jl documentation. Then, the LeNet model is casted as an KnetNLPModel:

```
using KnetLNPModels

size_minibatch = 100
LeNetNLPModel = KnetNLPModel(
        LeNet;
```

```
        data_train,
        data_test,
        size_minibatch,
    )
```

`LeNetNLPModel` can utilize a solver from JSOSolvers to minimize the loss of LeNetNLPModel iteratively evaluated with sampled data. The modification of the training minibatch is accomplished with the callback mechanism incorporated in JSOSolvers, executing the callback function at the conclusion of each iteration. In the following code snippet, we demonstrate the execution of the R2 solver with a `callback` that changes the training minibatch at each iteration:

```
using JSOSolvers

max_time = 300. # run at most 5min
callback = (LeNetNLPModel,
                    solver,
                    stats) -> KnetNLPModels.minibatch_next_train!(LeNetNLPModel)

solver_stats = R2(LeNetNLPModel; callback, max_time)
test_accuracy = KnetNLPModels.accuracy(LeNetNLPModel)
```

Another JSO solver that can train `LeNetNLPModel` is the line search LBFGS solver:

```
solver_stats = lbfgs(LeNetNLPModel; callback, max_time)
```

To run a trust-region method, `LeNetNLPModel` integrates a LSR1 approximation of the Hessian and run `trunk`:

```
  using NLPModelsModifiers # contains LSR1Model

  lsr1_LeNet = LSR1Model(LeNetNLPModel)
  callback_lsr1 =
    (lsr1_LeNet, solver, stats) -> KnetNLPModels.minibatch_next_train!(
                                                    lsr1_LeNet.model
                                                  )
  solver_stats = trunk(lsr1_LeNet; callback = callback_lsr1, max_time)
```

To summarize, KnetNLPModels provides a user-friendly interface between the deep neural networks modelled from Knet and the NLPModel's interface. Furthermore, the introduction of the `callback` enables the deterministic solvers from JSO to seamless adapt for the stochastic context of supervised learning, with the same simplicity of usage. Whereas the current section focuses on standard neural network architectures, the next section presents `PartitionedKnetNLPModels.jl`, an extension of KnetNLPModels.jl dedicated to partitioned architectures and limited-memory partitioned quasi-Newton training, respectively introduced in Section 5.2 and Section 5.3.

6.8 PartitionedKnetNLPModels

The idea of this section is to extend a KnetNLPModel into a PartitionedKnetNLPModel (Partitioned-KnetNLPModel) and implement the limited-memory partitioned quasi-Newton training presented in Section 5.3. PartitionedKnetNLPModels.jl is an experimental code and the last module implemented during this thesis. As such, it is not fully integrated in the JSO ecosystem. Consequently, it cannot benefit from the trust-region solver `trunk` used previously for numerical results. Nevertheless, PartitionedKnetNLPModels.jl implements a two ways backtracking line search, following the Algorithm 5.3.1, which exploits a limited-memory partitioned quasi-Newton operator B_k to operate.

In order to exploit the partial separability, the first step is to compute the dependencies of all neural network scores c_j, to determine $U_{p,j}$ parametrizing $\widehat{h}_{p,j}$. The computation of the dependencies is propagated during a forward evaluation of the neural network. Generally, all neurons depend on the previous layer's neurons weight dependencies and the weights that parameterize the layer to which they belong. The only exceptions are the neurons from the first layer, since they depend only on the input given to the neural network; hence, their weight dependencies are null, making an empty set initiating the propagation toward the second layer. During the forward propagation, each neuron accumulates the weights it depends on from the previous layers. Therefore, at the end of the propagation, each neuron representing a score c_j compiles all the weights on which it depends. The Figure 6.4 illustrates those dependencies for a simplified PSNet.

In the case of $\mathcal{L}^{\text{PSL}}$ (5.1a), each element loss function $\widehat{h}_{p,j}$ (5.2) relies on the difference of a couple of scores. Therefore, $\widehat{h}_{p,j}$ is parametrized by the union of weight dependencies from both c_j and c_p to form $U_{p,j}$. Note that the partially-separable structure may differ depending on the partially-separable loss function chosen. Based on those $U_{p,j}$, the partitioned data structures needed for Algorithm 5.3.1 are allocated from PartiallySeparableStructures.jl and PartitionedVectors.jl either to store individually $\nabla\widehat{h}_{p,j}$ or approximate $\nabla^2\widehat{h}_{p,j}$. Although those

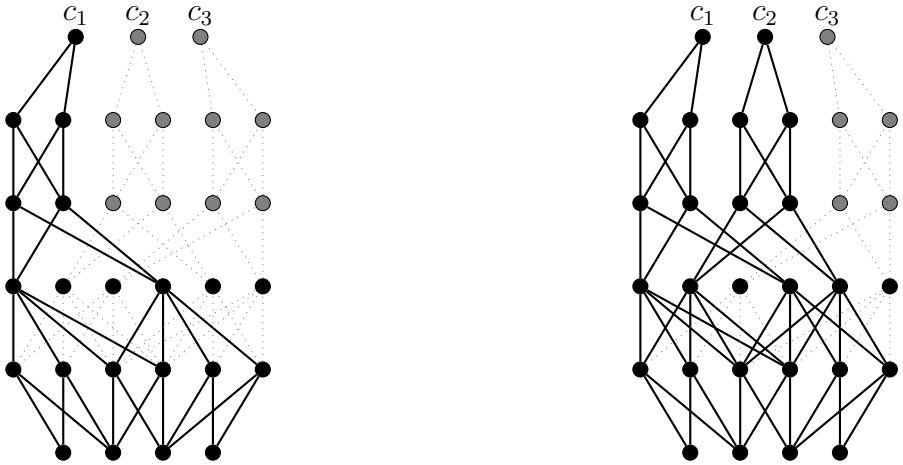


Figure 6.4 Weight dependencies for the score c_1 (left) and for the element function $\widehat{h}_{1,2}$ (right)

partitioned data structures and their routines do not fully support GPU computation, most of the runtime is monopolized by the computation of the partitioned gradient. PartitionedKnetNLPModels.jl computes the partitioned gradient by evaluating separately the gradient of each element function $\nabla\widehat{h}_{p,j}$. Despite computing the element function gradients on GPU, the computation of $N = C(C-1)$ gradients monopolizes the runtime. As the quality of the iterates can not compensate the time lost, limited-memory partitioned quasi-Newton optimizers are not competitive with first order optimizers when only training time is compared. Nevertheless, the numerical results shown Section 5.2.4 illustrate the accuracy over training dataset epochs for several optimizers which indicate that limited-memory partitioned quasi-Newton optimizers are competitive with first-order optimizers.

6.9 Conclusion

This chapter enumerates and details the Julia implementations resulting from the research activities conducted during this thesis. Some Julia modules are designed for unconstrained nonlinear optimization methods while others focus on deep learning. The former replicate state-of-the-art numerical results and effectively implement the limited-memory partitioned quasi-Newton methods. The latter implement the partitioned neural-network architecture, the partially-separable loss function, and several limited-memory partitioned quasi-Newton optimizers.

Implementing the concepts related to partial separability revealed difficulties not immediately apparent in theory. Those limitations are enumerated in Section 7.2, alongside a summary of this thesis works and future work, which, together, conclude this thesis.

CHAPTER 7 CONCLUSION

This manuscript presented several contributions thorough the previous chapters. This last chapter summarizes the contributions from each chapter in Section 7.1. The Section 7.2 enumerates the remaining limitations. To conclude, the Section 7.3 presents future work for enhancing the performance of the partitioned quasi-Newton methods.

7.1 Summary of Works

The Chapter 3 is a literature review about the minimization of partially-separable functions [9]. This chapter provides an overview of several methods from various fields of optimization, and, gives an idea about what can be expected from exploiting partial separability. It results in the identification of two shortcomings of partitioned quasi-Newton methods: the lack of an open source software package detecting automatically the partially-separable structure, and the absence of a method supporting large element functions.

The Chapter 4 remedies the second shortcoming by enabling partitioned quasi-Newton methods to manage large element functions [8]. This approach approximates every large element function Hessian by a quasi-Newton linear operator instead of a dense matrix. Although the Hessian matrix cannot be directly assembled, the approximated Hessian-vector product can be evaluated and used effectively by the (truncated) conjugate gradient method to solve a the quadratic subproblem. Moreover, the memory storage decreases significantly and one approximated Hessian-vector product requires less computational resource. A global convergence proof is provided for all newly introduced methods. The numerical results indicate that the limited-memory partitioned quasi-Newton methods outperform partitioned/limited-memory quasi-Newton methods for problems having large element functions.

The Chapter 5 applies the concepts akin to partial separability during the supervised training of a deep neural network [134, 135]. To do so, the Sections 5.2 and 5.3 respectively introduce a partitioned neural network architecture and a partially-separable loss function. By using both, the resulting training problem is partially-separable and can be minimized with a limited-memory partitioned quasi-Newton method. The numerical results obtained outperform the LBFGS optimizer and are comparable with the state of the art first order optimizers. The combination of both the partitioned architecture and the partially-separable loss function leverages a new way to distribute computation, where each worker operates on a neural-network's fragment.

All the pieces of software that produce the numerical results of the previous chapters are presented in Chapter 6 through several Julia modules. Most of those modules are designed to minimize general nonlinear partially-separable problems. In particular, they enable the automatic detection of partial separability and allocate the suitable partitioned data structures needed for partitioned quasi-Newton methods. Others modules are dedicated to supervised learning. The first one provides an interface between a neural network training problem and an optimization model, while the other implements the limited-memory partitioned quasi-Newton optimizers for training a neural network. Because they bridge the gap between theory and practice, the implementation of those Julia modules raises some issues that are enumerated in the next section.

7.2 Limitations

The contributions made in Chapter 4 and Chapter 5 both extend the application cases of partial separability. However, despite the global convergence proof of the limited-memory partitioned quasi-Newton methods presented in Section 4.2, a local rate of convergence such as Griewank [74] would be appreciated. Also, the implementations resulting from the Chapter 4 and Chapter 5 suffer some limitations which are partly described in Chapter 6.

First, the modules exploiting the partial separability to minimize a non-linear problem, e.g. `PartiallySeparableNLPModels.jl`, do not have all the features that LANCELOT has. In particular, they lack:

- constraints support;
- the internal representation of element functions, i.e., $f_i(x) = \widehat{f}_i^I(U_i^I x)$, where U_i^I is a sparse matrix selecting linear combinations of decision variables as internal variables, see Section 3.1.1;
- a complete merge strategy for problems having a lot of overlapping elements. Such a strategy may reduce the computational effort during partitioned quasi-Newton methods by merging elements, and therefore, cutting B_k sparsity. Although Conn et al. [36] explicit some criteria to choose whether two elements should be merged or not, those criteria should be adapted to consider the limited-memory quasi-Newton approximations of the element function Hessians. If preserving the sparsity prevails on reducing the computational resource for computing $v \to B_k v$, then a new ideal trade-off must be found.

Second, the PLSE method implemented in `PartiallySeparableNLPModels.jl` alternates between LBFGS and LSR1 operators to best approximate $\nabla^2 \widehat{f}_i(U_i x_k)$. The current state of implementation of quasi-Newton linear operators in JSO obliges the allocation of a new LBFGS or LSR1 operator whenever it switches from one to another. This implementation has two drawbacks:

- it induces allocations during runtime, and, it resets the memory of the newly allocated operator, i.e., it cannot use the pairs (s, y) from previous iterations. The difficulty lies in the fact that LBFGS and LSR1 do not compute the same vectors for performing the operator-vector product. LBFGS computes Bs and y, while LSR1 needs $y - Bs$;
- the current implementation of a quasi-Newton operator does not support GPU computation. This could be enabled with a block-LBFGS or a block-LSR1 implementation.

Third and more fundamentally, the exploitation of partial separability makes the numerical routines operate on smaller data structures. In general, in-place routines (at least in Julia) scale better for large data structures. Hence, in a sequential programming context, splitting the computation across small data structures increases the computation time. To minimize this drawback, the partitioned structures, especially the `PartitionedVectors`, could aggregate all the element contributions to a single large data structure, e.g., a `Vector` of size $\sum_{i=1}^{N} n_i$, since most operations are element-wise.

Likewise, the computation of partitioned derivatives scales better for large data structures. In addition of maintaining supplementary information for computing element derivatives during the reverse/backward propagation, the use of small tapes is less productive than one bigger tape. This particularity is true for the modelling modules about nonlinear optimization and deep learning. The best solution for this problem should be an end-to-end solver with its own modelling and automatic differentiation tools, implementing routines dedicated to the partially-separable structure.

Finally, the biggest upgrade that could help partitioned quasi-Newton methods to reach new heights would be the replacement of the truncated conjugate gradient to solve the partitioned quadratic subproblem. In spite of exploiting the partitioned structure through (approximated) Hessian-vector products to operate in parallel, it requires synchronization at each conjugate gradient iteration. The next section presents topics for future research, among which, an idea for replacing the conjugate gradient method.

7.3 Future Research

The several issues that the previous section addresses can all become future works. In particular, the features available in LANCELOT and currently missing could all be implemented by following the LANCELOT literature [33], summarized in Section 3.3.3.1 and Section 3.6.3. The rest of this section is decomposed in three independent themes, each improving the numerical effectiveness of partitioned quasi-Newton methods.

This first theme is about best modelling the partially-separable problems by a directed acyclic graph, which is closely related to automatic differentiation. It induces to work closely with a modelling language, such a JuMP [88], and adapt the directed acyclic graph to best exploit partial separability. In particular, it could factorize the identical element functions from the directed acyclic graph and generate a dedicated tape for the reverse automatic differentiation. Each vertex may have more than one adjoint value computed, i.e., one adjoint value for each element function using this vertex. Each variable vertex may have several adjoint values, corresponding to the element partial derivative contributions, whose sum is the total partial derivative. A direct extension would be the modelling of a (partitioned) neural network with this tool that could resolve the partitioned gradient issue.

The second theme focuses on the efficiency of the modules that allocate the partitioned data structures and their routines. To do so, the partitioned data structures must be tailored for the architecture on which they will be used, either a CPU or a GPU. A novel implementation of a `PartitionedVector` could use a `Vector` of size $\sum_{i=1}^{N} n_i$ where each element contribution is characterized by range of size n_i. This approach can be efficiently applied to GPU, by replacing the `Vector` to its GPU counterpart, e.g. `CuVector` (NVIDIA-GPU Vector). The partitioned matrix may also be tailored for GPU computation by replacing the `Matrix` with a `CuMatrix`. For larger element functions, the limited-memory quasi-Newton operators from JSO must be adapted for GPU. Once the development of a block-LBFGS implementation functioning with GPU is done, the limited-memory partitioned quasi-Newton operators will fully leverage GPU computation.

The last theme is the replacement of the (truncated) conjugate gradient method to solve a partitioned quadratic subproblem. As the conjugate gradient exploits the partitioned structure to eventually distribute its computation, this new method must be parallel. Ideally, this method requires less synchronization than the conjugate gradient. To do so, a two steps algorithm, similar to Okoubi and Koko [116], could be devised. First, it would solve the quadratic element subproblems and it would aggregate the element solutions found. Unlike Okoubi and Koko [116], which utilize the Lipschitz constant of the element function,

the element quadratic approximation $\widehat{m}_{i,k}$ (3.3.1) exploits the element Hessian approximation $\widehat{B}_{i,k}$,

The minimization of each $\widehat{m}_{i,k}$ can be done by several methods, e.g., the (truncated) conjugate gradient. However, the aggregation of element solutions is not the barycenter of element solution contributions, such as in [116], but a new quadratic subproblem for every overlap between element solutions. As the definition of this subproblem is technical, a simple example aggregating two element solutions is fully detailed in Appendix B. This simple scheme can be applied recursively onto every overlap between elements to produce a solution for the total quadratic subproblem m_k. Depending on the partially-separable structure, i.e., how element functions overlap, an order of resolution must be found. If several element functions overlap each others, then multiple aggregation subproblems can be carried out in parallel. Once they are done, the same process applies recursively onto the aggregated element solutions that still overlap.

REFERENCES

[1] P. Amestoy, I. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Method. Appl. M.*, 184(2):501–520, 2000.

[2] X. An, D. Li, and Y. Xiao. Sufficient descent directions in unconstrained optimization. *Comput. Optim. Appl.*, 48(3):515–532, 2011.

[3] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019. ISSN 0360-0300.

[4] A. S. Berahas, J. Nocedal, and M. Takáč. A multi-batch lbfgs method for machine learning, 2016.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.

[6] Z. Bian, Q. Xu, B. Wang, and Y. You. Maximizing parallelism in distributed training for huge neural networks, 2021.

[7] J. Bigeon, D. Orban, and P. Raynaud. PartiallySeparableNLPModels.jl: Automatic detection and exploitation of partial separability toward partitioned quasi-newton methods. `https://github.com/JuliaSmoothOptimizers/PartiallySeparableNLPModels.jl`, July 2022.

[8] J. Bigeon, D. Orban, and P. Raynaud. A framework around limited-memory partitioned quasi-newton methods. Cahier du GERAD G-2023-17, GERAD, Montréal, QC, Canada, May 2023.

[9] J. Bigeon, D. Orban, and P. Raynaud. A survey on partially separable structure in continuous optimization. Les Cahiers du GERAD G-2023-63, Groupe d'études et de recherche en analyse des décisions, GERAD, Montréal QC H3T 2A7, Canada, Dec. 2023.

[10] E. G. Birgin, J. Gardenghi, J. M. Martínez, S. A. Santos, and P. L. Toint. Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order regularized models. *Mathematical Programming*, 163:359–368, 2017.

[11] E. G. Birgin, J. L. Gardenghi, J. M. Martínez, S. A. Santos, and Ph. L. Toint. Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order

regularized models. *Mathematical Programming*, 163(1):359–368, May 2017. ISSN 1436-4646.

[12] C. Bischof and M. El-khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. Technical Report ANL/MCS-TM-163, Argonne National Laboratory, Argonne, Illinois, USA, 04 1994.

[13] C. Bischof, A. Bouaricha, P. Khademi, and J. Moré. Computing gradients in large-scale optimization using automatic differentiation. *INFORMS J. on Computing*, 9:185–194, 1997.

[14] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM T. Math. Software*, 21(1):123–160, 1995.

[15] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[16] A. Bouaricha and J. Moré. Impact of partial separability on large-scale optimization. *Comput. Optim. Appl.*, 7(1):27–40, 1997.

[17] Z. Bouzarkouna, D. Ding, and A. Auger. Using evolution strategy with meta-models for well placement optimization, 2010.

[18] Z. Bouzarkouna, A. Auger, and D. Ding. Local-meta-model CMA-ES for partially separable functions. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 869—-876, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305570.

[19] A. Brooke, D. Kendrick, and A. M. R. Raman. *GAMS: A User's Guide*. GAMS Development Corporation, 1998.

[20] C. G. Broyden. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA J. Appl. Math.*, 6(1):76–90, Mar. 1970. ISSN 0272-4960.

[21] A. Buckley and A. Lenir. QN-like variable storage conjugate gradients. *Math. Program.*, (27):155–175, 1983.

[22] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 8(4):639–655, 1971.

[23] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Math. Program.*, 63(1):129–156, Jan. 1994.

[24] R. H. Byrd, J. Nocedal, and R. A. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer Verlag, New York, 2006.

[25] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[26] R. H. Byrd, S. L. Hansen, J. Nocedal, and Y. Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[27] L. Cannelli, F. Facchinei, and G. Scutari. Multi-agent asynchronous nonconvex large-scale optimization. In *2017 IEEE 7th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 1–5, 2017.

[28] H. Cao and L. Yao. A partitioned PSB method for partially separable unconstrained optimization problems. *Appl. Math. Comput.*, 290:164–177, 2016.

[29] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, dec 2010.

[30] T. F. Coleman and J. J. Moré. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20(1):187–209, 1983. ISSN 00361429.

[31] B. Colson and Ph. L. Toint. Optimizing partially separable functions without derivatives. *Optim. Method Softw.*, 20(4–5):493–508, 2005.

[32] D. Conforti, L. De Luca, L. Grandinetti, and R. Musmanno. A parallel implementation of automatic differentiation for partially separable functions using pvm. *Parallel Comput.*, 22(5):643–656, 1996. ISSN 0167-8191.

[33] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. *Computing Methods in Applied Sciences and Engineering*, pages 42–54, 1990.

[34] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. LANCELOT*: a Fortran package for Large-scale Nonlinear Optimization (Release A)*. Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.

[35] A. R. Conn, N. I. M. Gould, M. Lescrenier, and Ph. L. Toint. Performance of a multifrontal scheme for partially separable optimization. In *Advances in Optimization and Numerical Analysis*, pages 79–96. Springer Verlag, 1994.

[36] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Improving the decomposition of partially separable functions in the context of large-scale optimization: a first approach. In *Large Scale Optimization*, pages 82–94. Springer Verlag, 1994.

[37] A. R. Conn, N. I. M. Gould, A. Sartenaer, and Ph. L. Toint. Convergence properties of minimization algorithms for convex constraints using a structured trust region. *SIAM J. Optim.*, 6(4):1059–1086, 1996.

[38] A. R. Conn, K. Scheinberg, and Ph. L. Toint. Recent progress in unconstrained nonlinear optimization without derivatives. *Math. Program.*, 79(1):397–414, Oct. 1997. ISSN 1436-4646.

[39] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust Region Methods*. SIAM, Philadelphia, USA, 2000. ISBN 0-89871-460-5. MPS-SIAM Series on Optimization.

[40] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *IMA J. Appl. Math.*, 13(1):117–119, 02 1974. ISSN 0272-4960.

[41] W. C. Davidon. Variable metric method for minimization. Report ANL-5990 Rev., Argonne National Laboratory, Argonne, Illinois, USA, 1959.

[42] M. Daydé, J. Y. L'Excellent, and N. Gould. Element-by-element preconditioners for large partially separable optimization problems. *SIAM J. Sci. Comput.*, 18(6):1767—-1787, Nov. 1997. ISSN 1064-8275.

[43] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng. Large scale distributed deep networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[44] R. S. Dembo. A primal truncated Newton algorithm with application to large-scale nonlinear network optimization. In K. Hoffman, R. Jackson, and J. Telgen, editors, *Computation Mathematical Programming*, pages 43–71. Springer Verlag, Heidelberg, Berlin, New York, 1987. ISBN 978-3-642-00933-4.

[45] J. E. Dennis Jr. and J. J. Moré. Quasi-Newton methods, motivation and theory. *SIAM Rev.*, 19(1):46–89, 1977.

[46] J. E. Dennis Jr., D. M. Gay, and R. E. Welsch. Algorithm 573: NL2SOL–an adaptive nonlinear least-squares algorithm [e4]. *ACM T. Math. Software*, 7(3):369—383, Sept. 1981. ISSN 0098-3500.

[47] J. E. Dennis Jr., D. M. Gay, and R. E. Welsh. An adaptive nonlinear least-squares algorithm. *ACM T. Math. Software*, 7(3):348—368, Sept. 1981. ISSN 0098-3500.

[48] T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *J. Artif. Int. Res.*, 2(1):263–286, jan 1995. ISSN 1076-9757.

[49] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming Studies*, 91(2):201–213, Jan. 2002. ISSN 1436-4646.

[50] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. 12, 2011. ISSN 1532-4435.

[51] I. S. Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM T. Math. Software*, 30(2):118—-144, June 2004.

[52] I. S. Duff and J. K. Reid. MA27–a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R10533, AERE Harwell Laboratory, Harwell, Oxfordshire, England, 1982.

[53] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM T. Math. Software*, 9(3):302—325, Sept. 1983. ISSN 0098-3500.

[54] I. S. Duff and J. A. Scott. A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications. *ACM T. Math. Software*, 25(4):404–424, Dec. 1999. ISSN 0098-3500.

[55] N. Durand and J. M. Alliot. Genetic crossover operator for partially separable functions. In *GP 1998, 3rd annual conference on Genetic Programming*, Madison, United States, 1998.

[56] O. Fercoq and P. Richtárik. Accelerated, parallel, and proximal coordinate descent. *SIAM J. Optim.*, 25(4):1997–2023, 2015.

[57] O. Fercoq and P. Richtárik. Optimization in high dimensions via accelerated, parallel, and proximal coordinate descent. *SIAM Rev.*, 58(4):739–771, 2016.

[58] F. Fischer and C. Helmberg. A parallel bundle framework for asynchronous subspace optimization of nonsmooth convex functions. *SIAM J. Optim.*, 24(2):795–822, 2014.

[59] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13 (3):317–322, 01 1970. ISSN 0010-4620.

[60] R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *Computer Journal*, 6(2):163–168, 08 1963. ISSN 0010-4620.

[61] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990.

[62] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole, Pacific Grove, second edition, 2002. `https://ampl.com/resources/the-ampl-book`.

[63] R. Fourer, C. Maheshwari, A. Neumaier, D. Orban, and H. Schichl. Convexity and concavity detection in computational graphs: Tree walks for convexity assessment. *INFORMS J. on Computing*, 22(1):26–43, 2010.

[64] D. M. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational differentiation: techniques, applications, and tools, Proceedings of the second international workshop on computational differentiation*, pages 173–184, Philadelphia, USA, February 1996. SIAM.

[65] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Rev.*, 47(1):99–131, 2005.

[66] E. Glorieux, B. Svensson, F. Danielsson, and B. Lennartson. Constructive cooperative coevolution for large-scale global optimisation. *J. Heuristics*, 23(6):449–469, 2017.

[67] D. Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–26, 1970. ISSN 00255718, 10886842.

[68] D. Goldfarb and Ph. L. Toint. Optimal estimation of jacobian and hessian matrices that arise in finite difference calculations. *Math. Comp.*, 43(167):69–88, 1984. ISSN 0025-5718.

[69] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM T. Math. Software*, 29(4):373–394, 2003.

[70] N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM T. Math. Software*, 29 (4):353–372, 2003.

[71] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Comput. Optim. Appl.*, 60(3):545–557, 2015.

[72] S. Gratton, A. Sartenaer, and Ph. L. Toint. Recursive trust-region methods for multiscale nonlinear optimization. *SIAM J. Optim.*, 19(1):414–444, 2008.

[73] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: recent developments and applications*, Mathematics and its applications, pages 83–108. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1989.

[74] A. Griewank. The global convergence of partitioned BFGS on problems with convex decompositions and lipschitzian gradients. *Math. Program.*, 50(1-3):141–175, 1991.

[75] A. Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numer. Math.*, 39:119–137, 1982.

[76] A. Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312. Academic Press, 1982. Publication editors : M.J.D. Powell.

[77] A. Griewank and Ph. L. Toint. Local convergence analysis for partitioned quasi-Newton updates. *Numer. Math.*, 39(3):429–448, 1982.

[78] A. Griewank and Ph. L. Toint. On the existence of convex decompositions of partially separable functions. *Math. Program.*, 28(1):25–49, 1984.

[79] A. Griewank and Ph. L. Toint. Numerical experiments with partially separable optimization problems. In D. F. Griffiths, editor, *Numerical Analysis*, pages 203–220, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-540-38881-4.

[80] J. Haldar and Z. Liang. Spatiotemporal imaging with partially separable functions: A matrix recovery approach. In *2010 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 716–719, Apr. 2010.

[81] T. Hamam and J. Romberg. Second-order filtering algorithm for streaming optimization problems. In *2019 IEEE 8th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 21–25, Dec. 2019.

[82] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, May 1996.

[83] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[84] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, 49(6):409–436, 1952.

[85] G. Hinton. Neural networks for machine learning, lecture 6a: Overview of mini-batch gradient descent. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, June 2012.

[86] HSL Mathematical Software Library. *A catalogue of subroutines (HSL 2000)*. AEA Technology, Harwell, Oxfordshire, England, 2000. `http://www.hsl.rl.ac.uk`.

[87] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[88] D. Iain, H. Joey, and L. Miles. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[89] B. Irons. A frontal solution program for finite element analysis. *Int. J. Numer. Meth. Eng.*, 2:5–32, 1970.

[90] Y. Katznelson. *An Introduction to Harmonic Analysis*. Cambridge Mathematical Library. Cambridge University Press, 3 edition, 2004.

[91] K. Kaya, F. Öztoprak, Ş. Birbil, A. Cemgil, U. Şimşekli, N. Kuru, H. Koptagel, and M. Öztürk. A framework for parallel second order incremental optimization algorithms for solving partially separable problems. *Comput. Optim. Appl.*, 72(3):675–705, 2019.

[92] S. Kim, M. Kojima, and Ph. L. Toint. Recognizing underlying sparsity in optimization. *Math. Program.*, 119(2):273–303, Jul 2009. ISSN 1436-4646.

[93] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[94] A. Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009.

[95] C. L. and C. S. Mldatasets.jl:, 2016.

[96] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 1558-2256.

[97] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[98] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[99] M. Lescrenier. Partially separable optimization and parallel computing. *Ann. Oper. Res.*, 14(1):213–224, 1988.

[100] Z. Liang. Spatiotemporal imaging with partially separable functions. In *2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 988–991, April 2007.

[101] D. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.*, 45(1-3):503–528, 1989.

[102] X. Lu. A computational study of the limited memory SR1 method for unconstrained optimization. *University of Colorado, Boulder*, 1996.

[103] M. Lubin and I. Dunning. Computing in operations research using julia. *INFORMS J. on Computing*, 27(2):238–248, 2015.

[104] L. Lukšan, , and J. Vlček. Variable metric method for minimization of partially separable nonsmooth functions. *Pacific Journal of Optimization*, 2, 01 2006.

[105] L. Lukšan, C. Matonoha, and J. Vlček. Algorithm 896: LSA: Algorithms for large-scale optimization. *ACM T. Math. Software*, 36(3), July 2009. ISSN 0098-3500.

[106] L. Lukšan, M. C., , and J. Vlček. Sparse test problems for unconstrained optimization. Technical Report 1064, Institute of Computer Science, Academy of Sciences of the Czech Republic, 01 2010.

[107] V. Malmedy and Ph. L. Toint. Approximating Hessians in unconstrained optimization arising from discretized problems. *Comput. Optim. Appl.*, 50(1):1–22, 2010.

[108] J. Mareček, P. Richtárik, and M. Takáč. Distributed block coordinate descent for minimizing partially separable functions. In M. Al-Baali, L. Grandinetti, and A. Purnama, editors, *Numerical Analysis and Optimization*, pages 261–288, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17689-5.

[109] I. Maros and C. Mészáros. A repository of convex quadratic programming problems. *Optim. Method Softw.*, 11(1–4):671–681, 1999.

[110] T. Migot, D. Orban, and A. S. Siqueira. The JuliaSmoothOptimizers ecosystem for linear and nonlinear optimization, 2021.

[111] A. Montoison, D. Orban, and contributors. Krylov.jl: A Julia basket of hand-picked Krylov methods. https://github.com/JuliaSmoothOptimizers/Krylov.jl, June 2020.

[112] Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

[113] Y. Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM J. Optim.*, 22(2):341–362, 2012.

[114] J. Nocedal. Updating quasi-Newton matrices with limited storage. *Math. Oper. Res.*, 35(151):773–782, 1980. ISSN 0025-5718.

[115] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, 2e edition, 2006.

[116] F. Okoubi and J. Koko. Parallel nesterov's method for large-scale minimization of partially separable functions. *Optim. Lett.*, 11:571–581, 2017.

[117] M. Omidvar, X. Li, and K. Tang. Designing benchmark problems for large-scale continuous optimization. *Inform. Sciences*, 316:419–436, 2015.

[118] D. Orban, A. S. Siqueira, and contributors. CUTEst.jl: Julia's CUTEst interface. `https://github.com/JuliaSmoothOptimizers/CUTEst.jl`, October 2020.

[119] D. Orban, A. S. Siqueira, and contributors. LinearOperators.jl. `https://github.com/JuliaSmoothOptimizers/LinearOperators.jl`, September 2020.

[120] D. Orban, A. S. Siqueira, and contributors. NLPModels.jl: Data structures for optimization models. `https://github.com/JuliaSmoothOptimizers/NLPModels.jl`, July 2020.

[121] D. Orban, A. S. Siqueira, and contributors. JSOSolvers.jl: JuliaSmoothOptimizers optimization solvers. `https://github.com/JuliaSmoothOptimizers/JSOSolvers.jl`, March 2021.

[122] D. Orban, A. S. Siqueira, and contributors. NLPModelsModifiers.jl: Model modifiers for nlpmodels. `https://github.com/JuliaSmoothOptimizers/NLPModelsModifiers.jl`, March 2021.

[123] D. Orban, A. S. Siqueira, and contributors. OptimizationProblems.jl: A collection of optimization problems in julia. `https://github.com/JuliaSmoothOptimizers/OptimizationProblems.jl`, July 2022.

[124] M. Porcelli and Ph. L. Toint. BFO, a trainable derivative-free brute force optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables. *ACM T. Math. Software*, 44(1), jun 2017. ISSN 0098-3500.

[125] M. Porcelli and Ph. L. Toint. Exploiting problem structure in derivative free optimization. *ACM T. Math. Software*, 48(1), feb 2022. ISSN 0098-3500.

[126] M. J. D. Powell. A new algorithm for unconstrained optimization. In J. Rosen, O. Mangasarian, and K. Ritter, editors, *Nonlinear Programming*, pages 31–65. Academic Press, 1970. ISBN 978-0-12-597050-1.

[127] M. J. D. Powell. A view of unconstrained optimization. In L. C. W. Dixon, editor, *Optimization in Action*, New York, 1976. Academic Press. Proceedings of the Conference on Optimization in Action Held at the University of Bristol in January 1975.

[128] M. J. D. Powell. Algorithms for nonlinear constraints that use Lagrangian functions. *Math. Program.*, 14(1):224–248, 1978.

[129] C. J. Price and Ph. L. Toint. Exploiting problem structure in pattern search methods for unconstrained optimization. *Optim. Method Softw.*, 21(3):479–491, 2006.

[130] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 873–880, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161.

[131] P. Raynaud, D. Orban, and J. Bigeon. ExpressionTreeForge.jl: A manipulator of expression trees. `https://github.com/JuliaSmoothOptimizers/ExpressionTreeForge.jl`, July 2022.

[132] P. Raynaud, D. Orban, and J. Bigeon. PartitionedStructures.jl: Partitioned derivatives storage and partitioned quasi-Newton updates. `https://github.com/JuliaSmoothOptimizers/PartitionedStructures.jl`, Month 2022.

[133] P. Raynaud, D. Orban, and J. Bigeon. PartitionedVectors.jl:. `https://github.com/JuliaSmoothOptimizers/PartitionedVectors.jl`, Month 2022.

[134] P. Raynaud, D. Orban, and J. Bigeon. Plsr1: A limited-memory partitioned quasi-newton optimizer for partially-separable loss functions. Les Cahiers du GERAD G-2023-41, Groupe d'études et de recherche en analyse des décisions, GERAD, Montréal QC H3T 2A7, Canada, Sept. 2023.

[135] P. Raynaud, D. Orban, and J. Bigeon. Partially-separable loss to parallellize partitioned neural network training. Les Cahiers du GERAD G-2023-36, Groupe d'études et de recherche en analyse des décisions, GERAD, Montréal QC H3T 2A7, Canada, Aug. 2023.

[136] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *Math. Program.*, 156(1–2):433–484, 2016.

[137] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[138] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3): 211–252, 2015.

[139] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001. ISSN 0167-739X. I. High Performance Numerical Methods and Applications. II. Performance Data Mining: Automated Diagnosis, Adaption, and Optimization.

[140] D. F. Shanno. On variable-metric methods for sparse Hessians. *Math. Comp.*, 34(150): 499–514, 190.

[141] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24(111):647–656, 1970. ISSN 00255718, 10886842.

[142] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[143] T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM J. Numer. Anal.*, 20(3):626–637, 1983.

[144] T. Steihaug, S. Hossain, and L. Hascoët. Structure in optimization: Factorable programming and functions. In E. Gelenbe and R. Lent, editors, *Computer and Information Sciences III*, pages 449–458, London, 2013. Springer London. ISBN 978-1-4471-4594-3.

[145] D. P. Sutton, M. C. Carlisle, T. A. Sarmiento, and L. C. Baird. Partitioned neural networks. In *2009 International Joint Conference on Neural Networks*, pages 3032–3037, 2009.

[146] Ph. L. Toint. On sparse and symmetric matrix updating subject to a linear equation. *Math. Comp.*, 31:954–961, 1977.

[147] Ph. L. Toint. Towards an efficient sparsity exploiting Newton method for minimization. In I. S. Duff, editor, *Sparse Matrices and Their Uses*, pages 57–88. Academic Press, London, England, 1981.

[148] Ph. L. Toint. Test problems for partially separable optimization and results for the routine PSPMIN. Technical Report 83/4, Department of Mathematics, University of Namur, Namur, Belgium, February 1983.

[149] Ph. L. Toint. Global convergence of the partitioned BFGS algorithm for convex partially separable optimization. *Math. Program.*, 36(3):290–306, 1986.

[150] Ph. L. Toint. On large scale nonlinear least squares calculations. *SIAM J. Sci. and Statist. Comput.*, 8(3):416–435, 1987.

[151] Ph. L. Toint and D. Tuyttens. On large scale nonlinear network optimization. *Math. Program.*, 48(1-3):125–159, 1990.

[152] Ph. L. Toint and D. Tuyttens. LSNNO, a FORTRAN subroutine for solving large-scale nonlinear network optimization problems. *ACM T. Math. Software*, 18(3):308–328, 1992. ISSN 0098-3500; 1557-7295/e.

[153] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program., Series A*, 106 (1):25–57, 2006.

[154] X. Wu, Y. Wang, J. Liu, and N. Fan. A new hybrid algorithm for solving large scale global optimization problems. *IEEE Access*, 7:103354–103364, 2019. ISSN 2169-3536.

[155] D. Yazdani, M. Omidvar, J. Branke, T. Nguyen, and X. Yao. Scaling up dynamic optimization problems: A divide-and-conquer approach. *IEEE T. Evolut. Comput.*, 24 (1):1–15, 2020.

[156] D. Yuret. Knet: beginning deep learning with 100 lines of julia. In *Machine Learning Systems Workshop at NIPS*, volume 2016, page 5, 2016.

APPENDIX A PARTIALLY-SEPARABLE PROBLEM STRUCTURES

This appendix details in Table A.1 the information of the partially-separable problems considered producing the performance profiles Section 4.3. The linear and constant element functions are merged together, and are therefore not counted. The term general is used for element functions which are not quadratic, and for element functions which are neither convex nor concave. In order to restrain column's heading sizes, consider:

- **med** for mean element dimension;
- **mad** for maximal element dimension;
- **mic** for minimal elemental contribution (for one decision variable);
- **mec** for mean elemental contribution (for one decision variable);
- **mac** for maximal elemental contribution (for one decision variable).

Table A.1 Partial separability details of the partially-separable problems set

name	n	N	M	quadratic	general	convex	concave	general	med	mad	mic	mec	mac
arwhead5000	5000	9999	3	0	4999	9999	5000	0	1.5	2	2	3	4999
bdqrtic5000	5000	9992	2	4996	4996	9992	0	0	3.0	5	1	6	4996
brybnd5000	5000	5000	7	0	5000	0	0	5000	7	7	2	7	7
chnrosnb5000	5000	9998	5000	4999	4999	4999	0	4999	1.5	2	1	3	3
clplatea5000	4900	19045	5	9522	9522	19045	1	0	2	2	0	7.7	8
clplateb4900	4900	19114	5	9522	9522	19114	70	0	2	2	0	7.7	8
clplatec4900	4900	19046	7	9522	9522	19046	2	0	2	2	0	7.7	8
cragglvy4900	4900	12245	5	2449	9796	7347	0	4898	1.6	2	2	4	4
dixmaane4900	4899	9799	6534	6532	3266	4900	1	4899	1.5	2	3	3.0	3
dixmaanf4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaang4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaanh4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaani4899	4899	9799	6534	6532	3266	4900	1	4899	1.5	2	3	3	3
dixmaanj4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaank4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaanl4899	4899	14697	6535	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaanm4899	4899	9799	9799	6532	3266	4900	1	4899	1.5	2	3	3	3
dixmaann4899	4899	14697	14697	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaano4899	4899	14697	14697	6532	8164	4900	1	9797	1.7	2	4	5	5
dixmaanp4899	4899	14697	14697	6532	8164	4900	1	9797	1.7	2	4	5	5
dixon3dq4899	4899	4899	2	4899	0	4899	0	0	2	2	1	2	2
dqdrtic4899	4899	4899	5	4899	0	4899	0	0	1.0	1	1	1.0	1
dqrtic4899	4899	4899	4899	0	4899	4899	0	0	1.0	1	1	1.0	1
edensch4899	4899	14695	4	4898	9796	9797	1	4898	1.3	2	2	4	4
engval14899	4899	9797	3	0	4898	9797	4899	0	1.5	2	1	3	3
errinros4899	4899	9796	4899	4898	4898	4898	0	4898	1.5	2	1	3	3
extrosnb4899	4899	4899	2	1	4898	1	0	4898	2	2	1	2	2
fletcbv24899	4899	14698	5	4900	4899	9799	4899	4899	1.3	2	4	4.0	4
fletcbv34899	4899	14698	4	4900	9798	4900	0	9798	1.3	2	4	4.0	4
freuroth4899	4899	9796	4	0	9796	0	0	9796	2.0	2	2	4	4
genhumps4899	4899	9797	3	4899	4898	4899	0	4898	1.5	2	2	3	3
genrose4899	4899	9797	3	4898	4898	4899	1	4898	1.5	2	1	3	3
genrose-nash4899	4899	9797	3	4898	4898	4899	1	4898	1.5	2	1	3	3
morebv4899	4899	4899	4899	0	4899	0	0	4899	3	3	2	3	3
ncb204899	4899	14667	4893	10	9768	9788	4889	4879	7.3	20	1	21.8	23
noncvxu24899	4899	9798	6	4899	4899	4899	0	4899	3	3	4	6	10
noncvxun4899	4899	9796	6	4898	4898	4898	0	4898	3	3	2	6	10
nondia4899	4899	4899	2	1	4898	1	0	4898	2	2	1	2	4899
nondquar4899	4899	4899	2	2	4897	4899	0	0	3	3	2	3	4898
penalty34899	4899	2454	6	2449	4	2450	1	4	9	4899	3	4.5	5
powellsg4899	4896	4896	4	2448	2448	4896	0	0	2	2	2	2.0	2
quartc4896	4896	4896	4896	0	4896	4896	0	0	1	1	1	1.0	1
sbrybnd4896	4896	4896	4896	0	4896	0	0	4896	7	7	2	7	7
schmvett4896	4896	14682	3	0	14682	0	0	14682	2.3	3	2	7	7
sinquad4896	4896	4896	3	0	4896	1	0	4895	3	3	1	3	4896
sparsine4896	4896	4896	4896	0	4896	0	0	4896	6	6	4	6	9
sparsqur4896	4896	4896	4896	0	4896	4896	0	0	6	6	4	6	9
spmsrtls4896	4897	9791	9791	0	9791	0	0	9791	2.2	3	3	4.3	5
srosenbr4897	4896	4896	2	2448	2448	2448	0	2448	1.5	2	1	1.5	2
tointgss4896	4896	4894	2	0	4894	0	0	4894	3.0	3	1	3	3
tquartic4896	4896	4895	2	1	4894	1	0	4894	2	2	0	2	4895
tridia4896	4896	4896	4896	4896	0	4896	0	0	2	2	1	2	2
vardim4896	4896	4898	3	4897	1	4898	0	0	3	4896	3	3.0	3
woods4896	4896	7344	5	4896	2448	4896	0	2448	1.6	2	2	2.5	3

APPENDIX B CONJUGATE GRADIENT REPLACEMENT

The formulation of a partitioned subproblem is clearly described by Okoubi and Koko [116], explicit by (3.34). Okoubi and Koko solve the partitioned subproblem in two steps. First, they solve separately the element subproblems and then aggregate their solutions to find a global solution. While the solution proposed by Okoubi and Koko [116] to solve element subproblems require only gradient information, it expresses an aggregation procedure taking the barycenter of element subproblem solutions. A similar approach can be applied for a partitioned quadratic subproblem.

In general, the exact solution of the partitioned linear system is not correlated to the element linear system solutions, even if all element matrices are positive definite. Nevertheless, both a line search or a trust-region subproblem only require the solution to be either a descent direction or a step comparable to the Cauchy point. Hence, it falls down to: formulate the partitioned quadratic subproblem, a solver for the element subproblems and an aggregation procedure that results in a total solution holding sufficient properties for convergence.

The rest of the section illustrates through a simple example an alternative method to solve a partitioned quadratic subproblem. The function f sums two overlapping element functions $f_i : \mathbb{R}^{n_i} \to \mathbb{R}$ and $f_j : \mathbb{R}^{n_j} \to \mathbb{R}$. At the k-th trust-region iteration, an element quadratic suproblem is:

$$\begin{array}{ll} \min_{\widehat{s}_{l,k}} & \widehat{f}_l(\widehat{x}_{l,k}) + \left(\nabla \widehat{f}_l(\widehat{x}_{l,k})\right)^\top \widehat{s}_{l,k} + \frac{1}{2}\widehat{s}_{l,k}^\top \widehat{B}_{l,k}\widehat{s}_{l,k} \\ \text{s.t.} & \|\widehat{s}_{l,k}\| \leq \Delta_{l,k} \end{array}$$

which can be solved with the truncated conjugate gradient method for example and returning either $\widehat{s}_{i,k}$ or $\widehat{s}_{j,k}$.

By setting one element solution, e.g. $\widehat{s}_{i,k}$, as an immutable part of the final solution, a new subproblem arise for minimizing the total quadratic subproblem. This aggregation process seeks to find $\widehat{s}_{j,k_{[j]}}$, i.e., the remaining variables of $\widehat{s}_{j,k}$ left to be minimized, the ones that are not overlapping with $\widehat{s}_{i,k}$. The total quadratic subproblem considering that s aggregates the result of $\widehat{s}_{i,k}$ and $\widehat{s}_{j,k_{[j]}} \in \mathbb{R}^{n-n_i}$ is

$$\begin{array}{ll} \min_{\widehat{s}_{j,k_{[j]}} \in \mathbb{R}^{n-n_i}} & \left(\sum_{l=1}^N U_l^\top \nabla \widehat{f}_l(\widehat{x}_{l,k})\right)^\top s + \frac{1}{2} s^\top \left(\sum_{l=1}^N U_l^\top \widehat{B}_l U_l\right) s \\ \text{s.t.} & \|\widehat{s}_{j,k_{[j]}}\| \leq \Delta_k \end{array} \quad \text{with} \quad s = \begin{bmatrix} \widehat{s}_{i,k} \\ \widehat{s}_{j,k_{[j]}} \end{bmatrix}.$$

Due to $\widehat{s}_{i,k}$ being a constant, several terms are constant, simplifying the problem to

$$\begin{aligned} \min_{\widehat{s}_{j,k_{[j]}} \in \mathbb{R}^{n-n_i}} \quad & \left(2\widehat{s}_{i,k_{[j]}}^{\top} \widehat{B}_{j_{[i,j]}} + \nabla \widehat{f}_j(\widehat{x}_{j,k})_{[j]}\right)^{\top} \widehat{s}_{j,k_{[j]}} + \widehat{s}_{j,k_{[j]}}^{\top} \widehat{B}_{j_{[j,j]}} \widehat{s}_{j,k_{[j]}} \\ \text{s.t.} \quad & \|\widehat{s}_{j,k_{[j]}}\| \leq \Delta_k \end{aligned}, \tag{B.1}$$

where $\widehat{s}_{i,k_{[j]}} = \widehat{s}_{j,k_{[i]}}$ is the part of $\widehat{s}_{j,k}$ overlapping and fixed by $\widehat{s}_{i,k}$. $\nabla \widehat{f}_j(\widehat{x}_{j,k})_{[j]}$ represents the element gradient components that $\widehat{s}_{j,k_{[j]}}$ interacts with in $\nabla f(x_k)^{\top} s$ and $\widehat{B}_j$ is block-decomposed as

$$\widehat{B}_j = \begin{bmatrix} \widehat{B}_{j_{[i,i]}} & \widehat{B}_{j_{[i,j]}} \\ \widehat{B}_{j_{[j,i]}} & \widehat{B}_{j_{[j,j]}} \end{bmatrix} \quad \text{where} \quad \begin{array}{c} \widehat{B}_{j_{[i,i]}} \in \mathbb{R}^{(n_j - n + n_i) \times (n_j - n + n_i)}, \quad \widehat{B}_{j_{[j,j]}} \in \mathbb{R}^{(n-n_i)\times(n-n_i)}, \\ \widehat{B}_{j_{[i,j]}} = \widehat{B}_{j_{[j,i]}}^{\top} \in \mathbb{R}^{(n_j - n + n_i)\times(n-n_i)}. \end{array}$$

$\widehat{B}_{j_{[i,i]}}$ gathers the components of $\widehat{B}_j$ overlapping with $\widehat{B}_i$; it is a vanishing constant term in (B.1). $\widehat{B}_{j_{[i,j]}}$ and $\widehat{B}_{j_{[j,i]}}^{\top}$ are multiplied by both $\widehat{s}_{i,k_{[j]}}$ and $\widehat{s}_{j,k_{[j]}}$, making $2\widehat{s}_{i,k_{[j]}}^{\top} \widehat{B}_{j_{[i,j]}} + \nabla \widehat{f}_j(\widehat{x}_{j,k})_{[j]}$ the linear factor of the aggregation problem. Finally, the quadratic terms of (B.1) result from $\widehat{B}_{j_{[j,j]}}$. This aggregation problem (B.1) can also be solved with the truncated conjugate gradient method.

The investigation on several instances of (B.1) shows potential. Frequently, the value found by this aggregated solution for the quadratic subproblem is (a lot) smaller than the one found from the unstructured conjugate gradient method. This scheme could be extended for more complex partitioned quadratic problems by solving recursively a combination of two element suproblems. However, several issues remain, such as: whether choose $\widehat{s}_{i,k}$ or $\widehat{s}_{j,k}$ as the initial solution, what properties the structure need to verify for ensuring a proper recursion between element solutions, and a thorough investigation about the impact of the overlapping size between elements must be done. Nonetheless, a trust-region method incorporating this alternative would benefit from the convergence proof of the partitioned trust-region method [37] with few efforts.