INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

PIR

Projet d'Introduction à la Recherche

présenté par

Victor BERTRET

Élève ingénieur de l'INSA Rennes

Spécialité Génie Mathématiques

Année universitaire 2020 - 2021

Tensor Networks and Kernels for Supervised Machine Learning

Tuteur

Jeremy COHEN

PIR soutenu le 18/12/2020

Résumé/Abstract

Résumé L'objectif de cette étude vise à reproduire l'expérience réalisée par Miles Stoudenmire et David Schwab décrite dans la publication "Supervised Learning with Tensor Networks" [1]. Pour ceci, nous nous sommes intéressés à la classification d'images de la base de données MNIST [4] qui contient des chiffres manuscrits en utilisant un réseau de tenseurs. Pour atteindre nos objectifs, nous avons utilisé une décomposition d'un tenseur à partir de laquelle nous avons développé des algorithmes de Machine Learning basiques, comme la descente de gradient, mais aussi des algorithmes plus sophistiqués inspirés de la physique. Néanmoins, l'objectif n'a pas été totalement atteint, nous n'avons pas réussi à classifier toutes les images de la base MNIST. Nous avons pu constater que nos algorithmes d'apprentissage n'étaient pas adaptés à des images de grande dimension. Cependant, nous avons trouvé de nouvelles pistes à explorer dans une suite potentielle à cette étude.

Abstract The aim of this study is to reproduce the experience from Miles Stoudenmire and David Schwab described in the publication "Supervised Learning with Tensor Networks" [1]. In this paper, we studied the classification of images from the MNIST [4] database which contains handwritten digits using a tensor network. To achieve our goals, we proposed a tensor decomposition from which we developed basic Machine Learning algorithms, such as gradient descent, but also more sophisticated physics-inspired algorithms. However, the goal was not completely achieved, we did not succeed in classifying all the images of MNIST database. We observed that our learning algorithms were not adapted to large-dimensional images. However, we have found new perspective to explore in a potential new study.

Table des matières

Liste et sens des abréviations techniques

- **SVM :** Support Vector Machine
- **SVD :** Décomposition en valeur singulière
- **MPS :** Matrix Product State
- **DMRG :** Density Matrix Renormalization Group
- **IA :** Intelligence artificielle
- **MNIST :** Mixed National Institute of Standards and Technology

Introduction

Depuis quelques années, l'abondance de données disponibles et l'accélération du calcul des ordinateurs, ont permis un développement important des algorithmes de machine learning dans différents domaines de la société telle que la classification d'images, la prévision de données, l'analyse de texte ... Tous ces ensembles de données contiennent de nombreuses informations que les algorithmes doivent alors faire ressortir.

L'objectif de cette étude est de reproduire l'expérience réalisée par Miles Stoudenmire et David Schwab décrite dans la publication "Supervised Learning with Tensor Networks" [1]. Ce projet a été réalisé au sein de l'équipe Panama du laboratoire de l'IRISA. Mon étude a été encadrée par Mr Jéremy Cohen que je remercie pour sa disponibilité et ses précieux conseils tout au long de mes recherches.

Dans cette étude, nous allons nous intéresser à la classification d'images de la base de données MNIST [4]. Celle-ci contient 70 000 images annotées de 784 pixels de chiffres manuscrits. Pour réaliser cette classification, nous allons utiliser un réseau de tenseurs.

FIGURE 1 – Représentation de la base de données MNIST

Dans ce rapport, nous commencerons par présenter les pre-réquis techniques nécessaires à la compréhension de notre étude. Ensuite, nous détaillerons le déroulement de notre réalisation. D'une part, nous utiliserons un réseau de tenseurs sans aucune structure particulière pour tenter de résoudre cette tâche de classification. D'autre part, nous testerons une autre méthode en imposant une structure particulière au réseau de tenseurs. Enfin, nous comparerons les différents algorithmes développés sur la base de données MNIST.

Chapitre 1

Pré-requis techniques

Commençons par développer quelques points techniques importants.

1.1 Présentation Calcul Tensoriel

Comme nous l'avons expliqué dans l'introduction, notre fonction f va être représentée par des tenseurs. La première sous-partie de ce rapport va donc s'orienter autour de la signification d'un tenseur et de l'introduction de quelques éléments théoriques qui vont nous servir par la suite.

1.1.1 Représentation tenseur

Tout d'abord, définissons le terme tenseur. Pour faire simple, sans aller dans les détails de la physique, un tenseur est simplement un tableau à n dimensions. On dit alors qu'il est d'ordre n .

Pour revenir à des représentations plus familières, nous pouvons considérer des tenseurs avec des ordres faibles. Un tenseur d'ordre 0 est un scalaire, un tenseur d'ordre 1 un vecteur et enfin un tenseur d'ordre 2 une matrice.

En revanche, dès que l'on a des tenseurs qui sont de plus en plus gros(en terme de modes) , cela peut être compliqué pour les représenter ou pour réaliser des calculs. Pour cela, des règles ont été définies. Nous représentons un tenseur par des ronds, et les indices des tenseurs sont notés par des lignes reliant de ces formes. Sur l'image ci-dessous, nous pouvons observer les représentations des tenseurs d'ordre allant de 1 à 3.

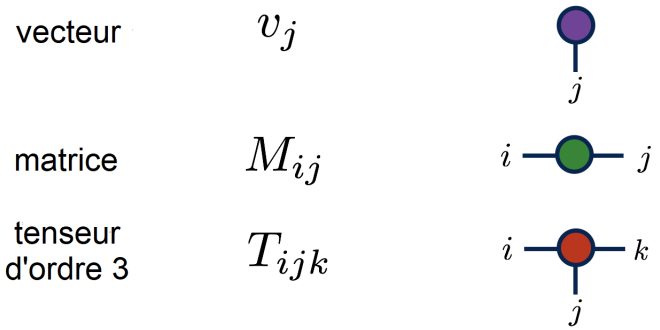

FIGURE 1.1 – Représentation de différents tenseurs
extrait de https ://tensornetwork.org

On remarque sur les images ci-dessus que l'on utilise des indices. Ceux-ci sont choisis arbitrairement mais ils doivent respecter une règle : un tenseur doit avoir des indices différents pour chaque ligne. De plus, pour simplifier la représentation, ils sont parfois omis.

Maintenant, après avoir défini ce que sont les tenseurs, on peut s'intéresser au nombre d'éléments composant un tenseur. En effet, chaque indice peut avoir une dimension différente. Si on considère un tenseur d'ordre N, avec N indices de mêmes dimensions d, alors ce dernier possède d^N éléments. Le nombre de paramètres d'un tenseur augmente donc de façon exponentielle avec le nombre d'indice.

Par la suite, notre objectif va être de trouver les meilleurs paramètres possibles pour un tenseur, par conséquent nous allons devoir être attentif au nombre de paramètres et donc aux dimensions et au nombre d'indices.

Ensuite, nous allons nous intéresser aux calculs réalisables avec des tenseurs.

1.1.2 Calcul Tensoriel

Maintenant que nous savons clairement représenter un tenseur, intéressons-nous aux différents type de calculs que nous pouvons réaliser entre eux.

Opérations usuelles Les premiers calculs les plus simples sont la multiplication par un scalaire et la somme de 2 tenseurs. Néanmoins, il faut faire attention pour la somme à ce que les 2 tenseurs possèdent la même dimension pour chaque indice. Voici un exemple avec des tenseurs d'ordre 2 :

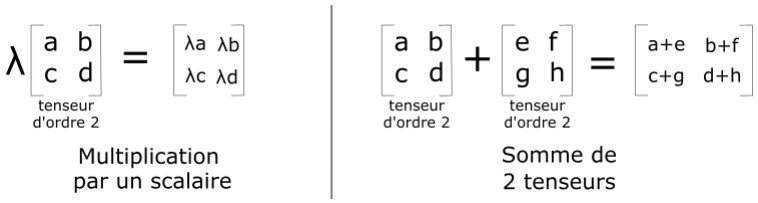

FIGURE 1.2 – Opérations usuelles entre tenseurs

Contraction Nous pouvons maintenant introduire la contraction de 2 tenseurs. Ceci est une sorte de généralisation du produit scalaire à un tableau à n dimensions. Considérons un tenseur $A_{\alpha_1,...,\alpha_{N_1},\gamma}$ d'ordre N_1+n et un tenseur $B_{\beta_1,...,\beta_{N_2},\gamma}$ d'ordre N_2+n avec γ composé de n indices et prenant ces valeurs dans $\mathcal{G}$.

En contractant $A_{\alpha_1,...,\alpha_{N_1},\gamma}$ et $B_{\beta_1,...,\beta_{N_2},\gamma}$, on obtient alors un nouveau tenseur $C_{\alpha_1,...,\alpha_{N_1},\beta_1,...,\beta_{N_2}} = A_{\alpha_1,...,\alpha_{N_1}}.B_{\beta_1,...,\beta_{N_2}}$ avec un ordre égale à N_1+N_2. Il est défini de la manière suivante :

$$C_{\alpha_1,...,\alpha_{N_1},\beta_1,...,\beta_{N_2}} = \sum_{\gamma\in\mathcal{G}} A_{\alpha_1,...,\alpha_{N_1},\gamma} B_{\beta_1,...,\beta_{N_2},\gamma}$$

Nous avons alors une définition calculatoire de la contraction. En revanche, nous utilisons très peu cette notation, nous préférons utiliser une notation graphique, comme dans la partie 1.1.1 avec les tenseurs, inspirée cette fois des sommations d'Einstein : la connexion de deux lignes d'indices implique une contraction, ou une sommation sur les indices connectés. Cette règle graphique va nous permettre de simplifier considérablement nos calculs. Voici un exemple simple de quelques représentations graphiques et leurs opérations associées :

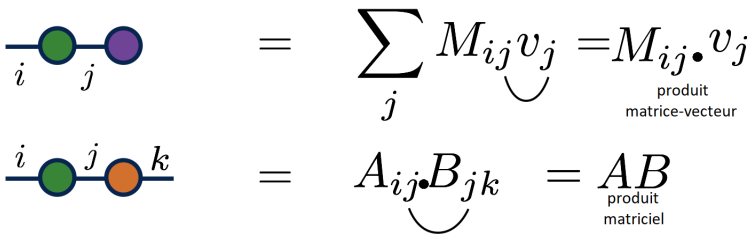

FIGURE 1.3 – Représentation graphique contraction
adapté de https ://tensornetwork.org

Produit tensoriel La dernière opération que nous devons définir est le produit tensoriel appelée aussi produit extérieur entre 2 tenseurs. Cette opération se note "$\otimes$". Considérons deux tenseurs $A_{\alpha_1,...,\alpha_{N_1}}$ d'ordre N_1 et $B_{\beta_1,...,\beta_{N_2}}$ d'ordre N_2 avec $\{\alpha_1,...,\alpha_{N_1}\}$ et $\{\beta_1,...,\beta_{N_2}\}$ prenant respectivement leurs valeurs dans $\mathcal{A}$ et $\mathcal{B}$. Le produit tensoriel de $A_{\alpha_1,...,\alpha_{N_1}}$ et $B_{\beta_1,...,\beta_{N_2}}$ donne un troisième tenseur $C_{\alpha_1,...,\alpha_{N_1},\beta_1,...,\beta_{N_2}} = A_{\alpha_1,...,\alpha_{N_1}} \otimes B_{\beta_1,...,\beta_{N_2}}$ d'ordre N_1+N_2 défini de la manière suivante :

$$\forall(\{\alpha_1,...,\alpha_{N_1}\},\{\beta_1,...,\beta_{N_2}\}) \in \mathcal{A}\times\mathcal{B}, C_{\alpha_1,...,\alpha_{N_1},\beta_1,...,\beta_{N_2}} = A_{\alpha_1,...,\alpha_{N_1}} B_{\beta_1,...,\beta_{N_2}}$$

Le produit tensoriel peut aussi être représenté sous forme graphique. Pour cela, on vient aligner les tenseurs côte à côte sans les relier par une droite comme illustré ci -dessous :

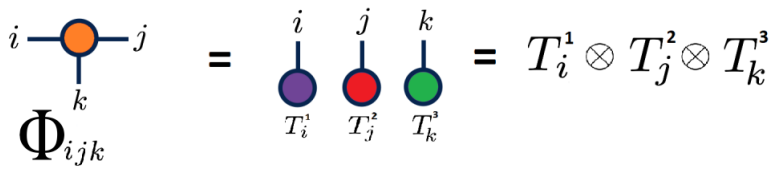

FIGURE 1.4 – Représentation graphique produit tensoriel

1.1.3 Decomposition MPS

Pour terminer cette partie sur les tenseurs et le calcul tensoriel, intéressons-nous aux décompositions possibles d'un tenseur. En effet, dans la partie 1.1.1, nous avons pu observer que le nombre de paramètres d'un tenseur augmentait de façon exponentielle avec son ordre. Par conséquent, si notre tenseur a un ordre élevé, son nombre d'éléments va exploser.

Nous avons de nombreux paramètres dans un tenseur non structuré. L'objectif de ces techniques est donc de trouver une décomposition du tenseur initial en une formule de tenseurs avec simultanément des ordres moins élevés et moins de paramètres. Dans notre cas, nous parlerons seulement de la décomposition MPS(Matrix Product State) car c'est la seule que nous utiliserons et c'est la plus développée.

La décomposition MPS d'un tenseur d'ordre N est une factorisation en une chaîne de contractions de N tenseurs d'ordre 3.

Si nous représentons la décomposition MPS d'un tenseur d'ordre 6 graphiquement, nous obtenons alors le graphique suivant :

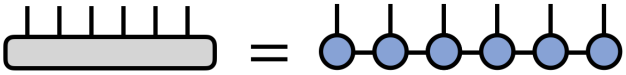

FIGURE 1.5 – Décomposition MPS d'un tenseur d'ordre 6
extrait de [1]

Alternativement, on peut aussi l'exprimer sous sa formule mathématique de la manière suivante :

$$T^{s_1,s_2,s_3,s_4,s_5,s_6} \approx \sum_{\{\alpha\}} A^{s_1}_{\alpha_1} . A^{s_2}_{\alpha_1\alpha_2} . A^{s_3}_{\alpha_2\alpha_3} . A^{s_4}_{\alpha_3\alpha_4} . A^{s_5}_{\alpha_4\alpha_5} . A^{s_6}_{\alpha_5} \quad (1)$$

Tous les tenseurs A sont différents mais nous les notons avec la même lettre et les distinguons seulement avec leurs indices qui sont différents. Cette décomposition est une approximation du tenseur initial mais si nous prenons une dimension assez grande pour les α alors cette décomposition est exacte et le tenseur initial peut-être exactement représenté par sa décomposition MPS.

C'est pourquoi les dimensions des α sont très importantes dans la décomposition, on appelle la dimension de α la "bond dimension" en anglais ce qui donne "dimension de liaison" en français. Elles peuvent être différentes pour chaque liaison et représente donc un paramètre de la décomposition.

Si nous avons un tenseur d'ordre N dont la dimension de chaque indice est d, alors si chaque "dimension de liaison" est supérieure ou égale à $d^{N/2}$ alors la décomposition est exacte mais ce n'est pas le plus important dans notre cas. En effet, nous préférons avoir une forme approchée de W mais diminuer grandement le nombre de paramètres. Dans ce cas, si on note m la "dimension de liaison" qui est la même pour toutes les liaisons, alors le nombre de paramètres est de Ndm^2 au lieu de d^N pour le tenseur initial.

Nous remarquons alors que cette décomposition est très intéressante, elle nous permet de passer d'un nombre de paramètres augmentant de manière exponentielle avec N à un nombre de paramètres augmentent linéairement avec N.

Par exemple, avec $N = 10$, $d = 2$ et $m = 5$, nous passons de 1024 à 500 paramètres, nous divisons donc par 2 le nombre de paramètres.

1.2 Apprentissage supervisé

Dans cette deuxième sous-partie sur les pré-requis techniques, nous allons nous focaliser sur l'apprentissage supervisé.

1.2.1 Définition

Commençons,tout d'abord, par définir l'apprentissage automatique :

Apprentissage automatique L'apprentissage automatique consiste à programmer des algorithmes, qui grâce à des données d'entrée, permettent de résoudre au mieux un problème à travers des méthodes statistiques.

Lors de la phase d'apprentissage, le modèle apprend à réaliser une tâche particulière grâce des données d'entrainement. Ensuite, nous pouvons utiliser ce modèle pour réaliser la tâche apprise sur un autre jeu de données. L'apprentissage automatique permet de réaliser des tâches allant de la reconnaissance faciale à la génération de texte automatique.

L'apprentissage supervisé est un cas particulier de l'apprentissage automatique. Il consiste à résoudre un problème grâce à des données annotées par un label. Pour représenter les données d'entrée, nous avons à notre disposition N_t données de dimensions N : $x_i = (x_i^1, ..., x_i^N)$ associées à un label y_i avec $i \in \{1, .., N_t\}$.

Pour chaque donnée, les composantes $x^1, ..., x^N$ sont appelées les descripteurs. Par exemple, dans le cadre d'une image en noir et blanc, celles-ci peuvent représentés les niveaux de gris des N pixels de l'image. Les labels eux représentent les catégories associées à chaque exemple : continus dans le cas d'une régression ou discrets dans le cadre d'une classification.

Dans notre cas, nous nous intéresserons à la classification et nous allons considérer un ensemble L de N_l classes différentes. Voici un exemple de la tâche que nous allons devoir réaliser :

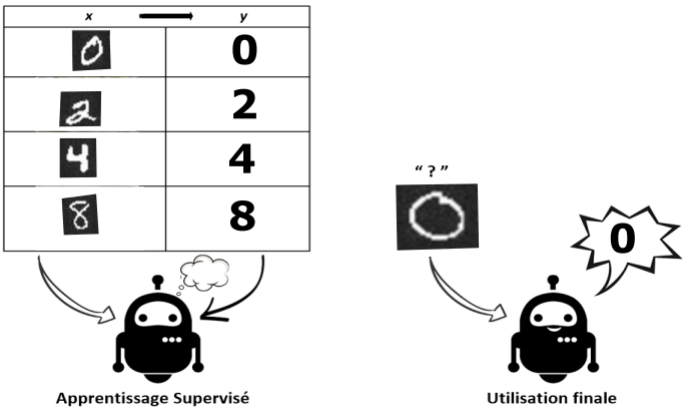

FIGURE 1.6 – Apprentissage supervisé MNIST

Nous allons maintenant détailler les étapes importantes d'une méthode d'apprentissage supervisé.

1.2.2 Déroulement

Notre objectif lors de cette étude est de classifier des images dans les bonnes classes. L'apprentissage supervisé peut se découper en 4 étapes :

• **Définition d'un ensemble de données** Dans notre cas, les descripteurs représentent les pixels des images allant de 0 pour blanc à 1 pour noir. Ensuite, nous allons choisir un système de "one hot encoding" pour les y. C'est à dire que chaque y^i est un vecteur et si L est le bon label pour x_i alors $y_i^L = 1$ et $y_i^l = 0$ pour tous les autres labels l. y_i représente les probabilités que x_i appartient à chaque label. Par conséquent, il est normal de fixer cette probabilité à 1 pour le bon label et 0 pour les autres.

• **Définition du modèle** Afin de résoudre un problème particulier, il faut définir un modèle et donc dans le cas d'une classification, une fonction de décision $f^l(x)$. Nous devons paramétrer cette fonction. Il existe différents types de paramétrage comme les réseaux de neurones, les SVMs(Support Vector Machine), les arbres de décisions. Dans cette étude, comme le titre le souligne, nous allons nous intéresser aux réseaux de tenseurs.

• **Définition de la fonction de coût** Pour qu'un modèle puisse réaliser une tâche particulière, nous devons être en capacité de mesurer celui-ci à travers sa fonction de décision. Pour ce faire, nous allons définir une fonction de coût. Ici, nous partirons sur une fonction de coût quadratique :

$$C(W) = \frac{1}{2} \sum_{n=1}^{N_T} \sum_{l} (f^l(X_n) - y_n^l)^2 \tag{2}$$

La fonction est nulle lorsque $f^l(X_n) = y_n^l$. La fonction de coût permet de mesurer la qualité de la réalisation de la tâche par la fonction de décision. Plus le coût est faible, plus la réalisation de la tâche est bonne. Dans notre cas, la réalisation parfaite de la classification d'une image se réalise lorsque la fonction de décision renvoie le label associé à l'image. Nous appelons parfois la valeur de la fonction de coût l'erreur.

• **Définition et utilisation d'une algorithme d'apprentissage** Nous devons chercher les paramètres du modèle qui minimisent la fonction de coût. C'est la phase d'apprentissage. Pour cela, de nombreux algorithmes ont été développés. Nous allons dans notre cas nous focaliser sur la méthode de descente de gradient que nous expliquerons dans la prochaine section. Cette étape permet d'apprendre au modèle comment résoudre le problème et dans notre cas comment classifier les images.

Pour résumer, une méthode d'apprentissage supervisé se déroule de la manière suivante :
- Définir l'ensemble de données
- Définir un modèle pour la fonction décision f^l
- Définir une fonction de coût
- Définir et utiliser un algorithme d'apprentissage pour minimiser la fonction de coût

Enfin, une fois le modèle entrainé, nous pouvons classifier les images d'un autre ensemble de données x_i avec $i \in \{1, .., N_{t_2}\}$ dans une classe de la manière suivante :

$$\underset{l \in \mathcal{L}}{\operatorname{argmax}} f^l(x_i) \tag{3}$$

1.2.3 Gradient

Pour finir sur l'apprentissage supervisé, nous allons vous présenter un algorithme d'apprentissage : la descente de gradient.

Pour réaliser une descente de gradient, nous devons commencer par calculer le gradient. Le gradient d'une fonction f en un point x est l'évolution de la fonction au voisinage de ce dernier, c'est à dire vers quel sens aller pour avoir une valeur plus élevée de la fonction. On le note $\frac{\partial f}{\partial x}$.

Afin d'avoir une représentation plus imagée, prenons l'exemple du relief d'une montagne. Si nous prenons la fonction qui représente l'altitude par rapport à notre position, alors le gradient de cette dernière va être la direction à prendre pour gagner le plus rapidement de l'altitude dans les minutes qui vont venir. Voici une image qui illustre la situation que nous venons de présenter :

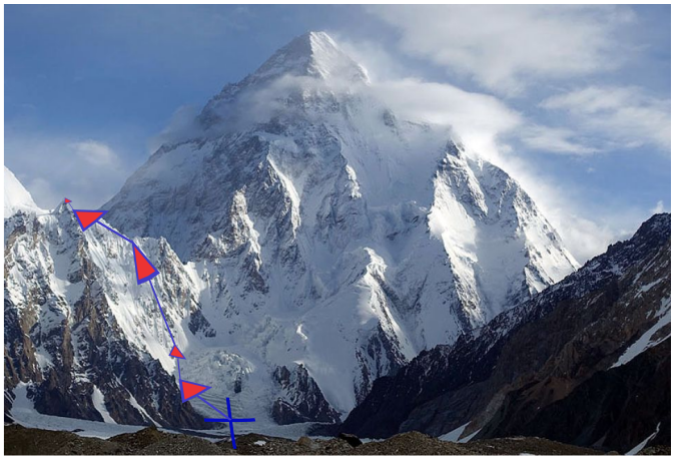

FIGURE 1.7 – Explication Gradient

Nous observons donc sur l'image que si nous répétons cette opération plusieurs fois, nous allons arriver à un sommet donc à un maximum. En revanche, ce n'est pas forcément le point le plus haut de la chaine de montagne comme nous observons sur la photo. Si le sommet sur lequel nous arrivons est le plus haut de toute la chaîne de montagne, on l'appelle le maximum global et dans le cas ci-dessus sur la photo, un maximum local.

Mathématiquement, un pas de gradient de la fonction f au point x, est défini par :

$$x = x + \alpha \frac{\partial f}{\partial x}, \text{avec } \alpha \in \mathbb{R}$$

Enfin, lorsque nous prenons plusieurs fois la direction opposée au gradient, nous appelons cela une descente de gradient, celle-ci permet de converger vers un minimum local ou global.

Pour converger vers un minimum global, il faut essayer d'avoir la fonction la plus lisse possible.

Chapitre 2

Premier Modèle

Nous avons maintenant terminé de définir toutes les notions sur les tenseurs dont nous allons avoir besoin par la suite. Dans cette 2ème partie, nous allons donc pouvoir commencer par une première version d'un modèle de réseau de tenseurs.

2.1 Paramétrisation de f

Premièrement, nous devons décider d'une paramétrisation de f afin de l'optimiser par la suite.

Pour cela, nous allons fonctionner en 2 étapes. Nous commencerons par envoyer nos vecteurs d'entrée x dans un espace de dimension plus grand grâce à une fonction Φ puis nous définirons un tenseur W afin de classifier nos entrée par la fonction de décision suivante :

$$\forall l \in \mathcal{L}, f^l(x) = W^l.\Phi(x) \tag{4}$$

Paramétrisation de Φ Nous devons donc commencer par créer une fonction plongement Φ qui permettra d'envoyer nos entrées dans un espace de dimension plus grand.

Dans la physique, les réseaux de tenseurs sont utilisés par la combinaison de N systèmes en réalisant un produit tensoriel des vecteurs correspondant à chaque système. Par conséquent, par analogie, nous définissons la fonction Φ de la manière suivante :

$$\forall i \in \{1, ..., N_t\}, \Phi^{s_1 s_2 ... s_N}(x_i) = \phi^{s_1}(x_i^1) \otimes \phi^{s_2}(x_i^2) \otimes ... \otimes \phi^{s_N}(x_i^N) \tag{5}$$

avec chaque $\phi^{s_j}(x_i^j)$ $(j = 1, ..., N)$, une application locale envoyant chaque x_i^j dans un espace de dimension d.

Par conséquent les indices s_j vont de 1 à d. $\Phi^{s_1 s_2 ... s_N}(x_i)$ est un produit tensoriel de N tenseurs d'ordre 1 de dimension d donc un tenseur d'ordre N avec d^N éléments. Ici, nous avons pris la décision d'appliquer la même fonction à tous les pixels, nous aurions pu décider d'avoir des fonctions différentes pour chaque pixel. On observe ci dessous sa représentation graphique avec N=6 :

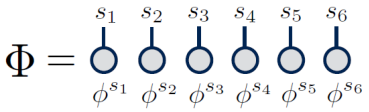

FIGURE 2.1 – Représentation de Φ
extrait de [1]

Nous avons donc défini une forme globale pour Φ mais pas encore la forme de chaque $\phi^{s_j}(x_i^j)$. Nous aurions pu ne pas lui donner de forme particulière et au contraire l'optimiser comme W^l mais dans ce rapport nous avons pris le choix d'instaurer à ϕ une fonction particulière.

Nous considérons alors maintenant que toutes les entrées seront des images en noir et blanc de N pixels avec des valeurs allant de 0 pour blanc à 1 pour noir. Nous définissons alors pour chaque i$\in \{1, ..., N_t\}$ et chaque j$\in \{1, ..., N\}$:

$$\phi^{s_j}(x_i^j) = [\cos(\frac{\pi}{2}x_i^j), \sin(\frac{\pi}{2}x_i^j)] \tag{6}$$

Cette application permet de transformer chaque pixel en un vecteur normalisé de dimension d=2 comme nous pouvons l'observer sur l'image ci-dessous. Dans les publications [3] et [2], l'utilité de la fonction ϕ a été discuté. Cette application est d'ailleurs inspiré du modèle statistique d'*Ising*, beaucoup utilisé en physique statistique.

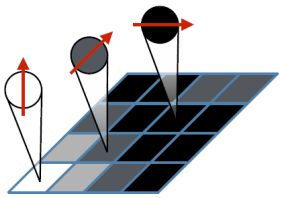

FIGURE 2.2 – Représentation de quelques applications de ϕ à une image de 16 pixels
extrait de [1]

Ce n'est surement pas le meilleur choix possible mais nous avons décidé de fixer ϕ afin de simplifier le réseau de tenseurs par la suite.

Représentation de f^l Nous connaissons maintenant la forme de la fonction Φ. En effet, c'est un tenseur d'ordre N. Par conséquent, comme nous réalisons la contraction de W^l avec Φ afin d'obtenir la fonction f^l qui est un tenseur de dimension 1, on considère que W^l est un tenseur d'ordre N+1. Nous rajoutons une dimension de plus qui correspond à la dimension de l. Ce dernier peut donc être représenté de la manière suivante :

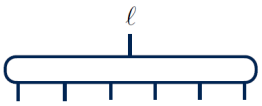

FIGURE 2.3 – Représentation de W^l
extrait de [1]

Notre tenseur W possède donc $2^N * N_l$ paramètres que nous allons devoir optimiser afin de trier les images. On peut enfin terminer sur cette sous-partie en représentant la fonction de décision :

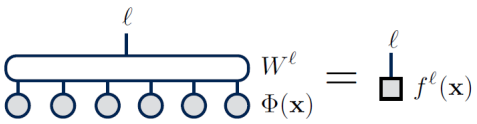

FIGURE 2.4 – Représentation de f^l

2.2 Définition du premier modèle

Nous pouvons passer à la définition du premier modèle.

Pour l'ensemble de données, nous utilisons celui défini dans la partie 1.1. De même, pour la fonction de coût, nous utilisons la fonction de coût quadratique défini par l'équation (2). De plus, nous avons explicité la fonction de décision f^l lors de la sous-partie précédente. Il nous reste donc à définir l'algorithme d'apprentissage.

Pour cela , nous allons utiliser une descente de gradient. On doit donc tout d'abord commencer par obtenir une formule pour le calcul du gradient de la fonction C(W) par rapport à W. D'après (7), on peut écrire :

$$\begin{aligned} C(W) &= \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l}(f^l(X_n) - y_n^l)^2 \\ &= \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l}(W^l.\Phi(X_n) - y_n^l)^2 \\ &= \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l}(\sum_{s_1,...,s_N} W^l_{s_1...s_N}\phi^{s_1}(x_1)...\phi^{s_N}(x_N) - y_n^l)^2 \end{aligned}$$

et si on fixe $s_1 = t_1, ..., s_N = t_N$ ainsi que $l = L$ ...

$$\begin{aligned} \frac{\partial C(W)}{\partial W^L_{t_1...t_N}} &= \sum_{n=1}^{N_T}(\sum_{s_1,...,s_N} W^L_{s_1...s_N}\phi^{s_1}(x_1)...\phi^{s_N}(x_N) - y_n^L)\phi^{t_1}(x_1)...\phi^{t_N}(x_N) \\ &= \sum_{n=1}^{N_T}\sum_{s_1,...,s_N} W^L_{s_1...s_N}\phi^{s_1}(x_1)...\phi^{s_N}(x_N)\phi^{t_1}(x_1)...\phi^{t_N}(x_N) - \sum_{n=1}^{N_T} y_n^L\phi^{t_1}(x_1)...\phi^{t_N}(x_N) \end{aligned}$$

Nous pouvons enfin utiliser les opérations sur les tenseurs afin d'obtenir cette dernière formule :

$$\frac{\partial C(W)}{\partial W^l} = \sum_{n=1}^{N_T}(W^l.\Phi(x_n)\Phi^T(x_n) - y_n \otimes \Phi(x_n)) \tag{7}$$

avec $\Phi(x_n)\Phi^T(x_n) = \phi(x_n^1)\phi^T(x_n^1) \otimes ... \otimes \phi(x_n^N)\phi^T(x_n^N)$.

Maintenant,après avoir calculé le gradient, il nous reste à définir comment allons-nous minimiser la fonction de coût ? Dans notre cas, nous allons faire très simple : nous allons utiliser une descente de gradient à pas fixe comme défini dans la section 1.2.3.

C'est à dire à chaque itération de l'algorithme, nous allons calculer le gradient de la fonction de cout grâce à l'équation ci dessus (7) puis nous redéfinirons W^l comme $W^l = W^l - \alpha\frac{\partial C(W)}{\partial W^l}$ avec un alpha, un nombre réel fixé strictement positif. Ce dernier est donc un hyperparamètre que nous devons choisir. Nous testerons son impact lors des applications numériques.

Nous avons donc réunis toutes les informations nécessaires afin d'implémenter l'algorithme. Voici son pseudo-code :

Algorithm 1 Optimiser W

Entrée: `tenseur` W^l, `pas fixe` α, `précision p`
Procédure:
  $erreur \leftarrow C(W)$
  Tant que $erreur > p$ **faire**
    $W^l \leftarrow W^l - alpha\frac{\partial C(W)}{\partial W^l}$
    $erreur \leftarrow C(W)$
  fin Tant que

2.3 Applications Numériques

Nous avons donc une première version de notre algorithme. Notre objectif maintenant est de le tester numériquement.

Pour cela, nous créons une base d'images en noir et blanc de 9 pixels aléatoirement puis nous affectons la moitié des images au label 0 puis l'autre moitié au label 1.

Nous avons généré les paramètres de W de manière aléatoire selon des tirages d'une loi normale centrée réduite. Voici nos résultats :

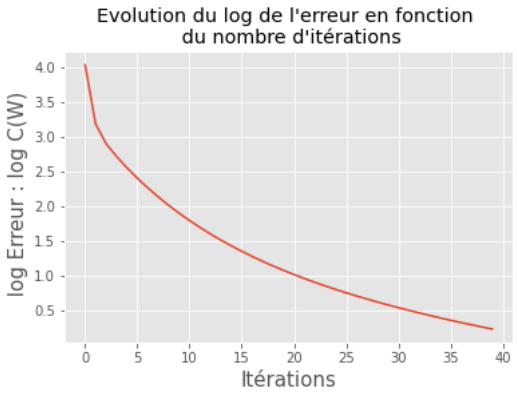

FIGURE 2.5 – Evolution de l'erreur

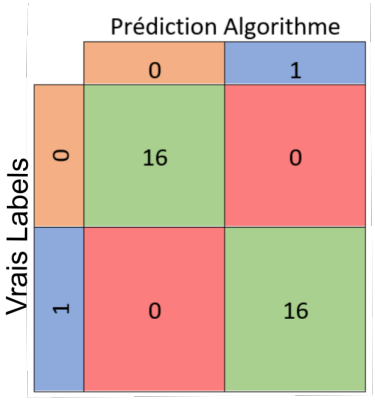

FIGURE 2.6 – Matrice de confusion

Nos résultats sont encourageants. En effet, nous remarquons que l'erreur diminue à chaque itération et tends vers 0. De plus, après les 40 itérations, on remarque que toutes nos images sont bien triées.

En revanche, même si nos résultats sont très bons, le nombre d'images et de pixels n'étaient pas élevées. De plus, le nombre de pixels par image est lié au nombre de paramètres du tenseur W. Nous avons vu dans la partie 2.1 que le nombre de paramètres de W est égale à $2^N * N_l$ soit dans notre exemple $2^9 * 2 = 1024$ paramètres. Si nous multiplions par 10 le nombre de pixels, alors le nombre de paramètres de W passe à $2^{90} * 2 = 2.4758801e + 27$ paramètres. Nous voyons toute suite que cela explose vers l'infini.

Nous allons alors regarder numériquement si nous pouvons toujours utiliser l'algorithme. Pour cela, nous mesurons le temps d'exécution d'une itération et le nombre de paramètres de W pour des valeurs de N croissantes :

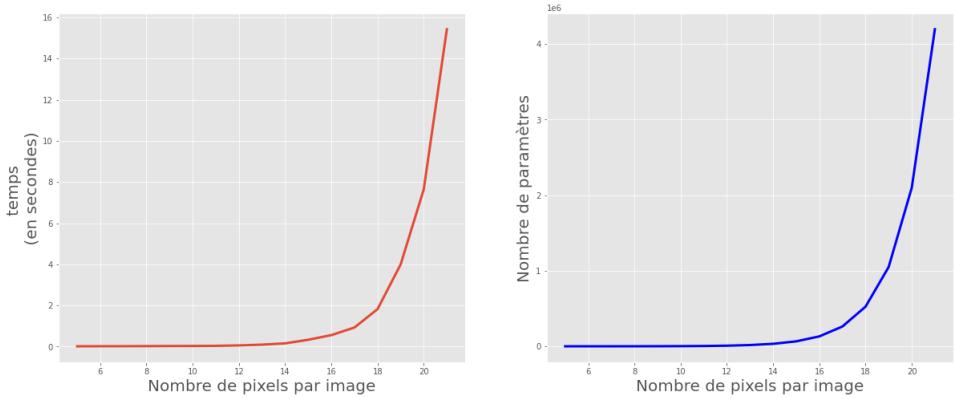

FIGURE 2.7 – Evolution temps d'execution et nombre de paramètres en fonction N

Nous observons bien sur ces 2 graphiques que lorsque que le nombre de paramètres de W explose de manière exponentielle, le temps d'exécution suit la même évolution. Avec seulement 21 pixels, une itération du programme prend plus de 20 secondes. Nous ne pouvons pas nous permettre d'attendre autant de temps. En effet, dans la base d'images MNIST, les images possèdent 780 pixels et au vu de l'évolution exponentielle du temps d'exécution, une itération du programme prendrait trop de temps. Nous devons modifier notre algorithme afin qu'il fonctionne avec des images plus grandes.

Avant de passer à la réalisation d'un deuxième algorithme, nous pouvons expliquer le temps d'exécution du programme par l'opération de contraction $W^l.\Phi$ qui est utilisé dans le gradient ainsi que dans le calcul de la fonction de décision.

2.4 Détermination du coût du calcul $W^l.\Phi$

Nous nous intéressons à calculer le coût de la contraction entre W^l et Φ. Rappelons tout d'abord que W^l est un tenseur d'ordre N+1 et $\Phi(x_i)$ un tenseur d'ordre N. L'opération entre ces 2 tenseurs donnent donc un vecteur de longueur N_l.

Dans notre cas, nous n'allons pas former le grand tenseur Φ mais garder les $\phi^{s_1}, ..., \phi^{s_N}$ afin d'avoir $2 * N$ paramètres au lieu de 2^N. On doit donc réaliser l'opération de la figure 2.4.

Pour cela, nous contractons le gros tenseur W^l, avec chaque ϕ^{s_i} un par un $\forall i \in \{1, ..., N\}$. Si nous considérons que nous commençons par contracter W^l avec ϕ^{s_1}, nous obtenons un nouveau tenseur T définit de la manière suivante :

$$\forall s_2, ..., s_N, l,\ T^l_{s_2,...,s_N} = \sum_{\{s_i\} \neq s_1} W^l_{s_1,...,s_N} \phi^{s_1}$$

Cette opération nécessite $d^N N_l$ multiplications et autant d'additions. Nous contractons ensuite ϕ^{s_2} avec le nouveau tenseur T ce qui donne $2d^{N-1}N_l$ opérations et ainsi de suite.

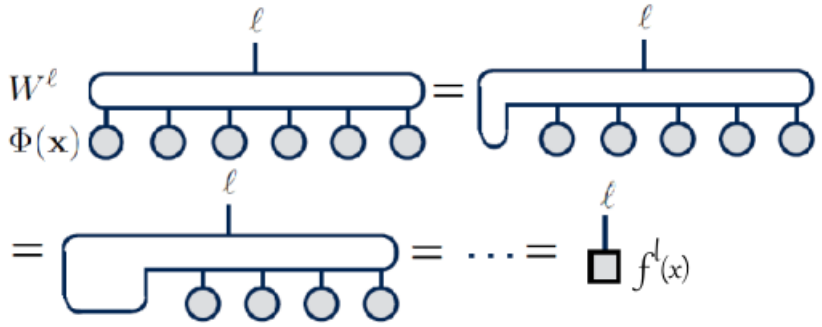

FIGURE 2.8 – Visualisation Contraction W^l et Φ avec N=6

Le nombre d 'opérations total est égale $2 * \sum_{k=1}^{N} N_l d^k = 2d\frac{1-d^N}{1-d}$ soit environ 2^{N+3} dans l'exemple de l'application numérique. Comme le nombre de paramètres, nous pouvons alors constater que le nombre d'opérations augmentent aussi de façon exponentielle avec N.

Notre objectif maintenant est donc de réussir à ce que le nombre de paramètres de W^l et que le nombre d'opérations pour construire f^l n'évoluent pas de façon exponentielle avec N.

Chapitre 3

Decomposition MPS du tenseur W

Nous avons pu observer dans la partie précédente qu'apprendre le tenseur W^l sans imposer de structure particulière le rendait inutilisable. Par conséquent, nous décidons de réaliser une approximation du tenseur W avec une décomposition MPS de W^l comme défini dans la partie 1.1.3. Notre tenseur W est définit maintenant par la formule suivante d'après (3) :

$$\forall s_1, ..., s_N, l,\ W^l_{s_1,...,s_N} = \sum_{\alpha_1,...,\alpha_{N-1}} A^{\alpha_1}_{s_1} ... A^{l;\alpha_i\alpha_{i+1}}_{s_i} ... A^{\alpha_{N-1}}_{s_N}$$

Nous positionnons l'indice l pour le moment sur un tenseur A quelconque.

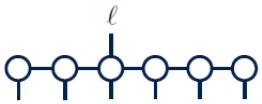

FIGURE 3.1 – Nouvelle forme de W
extrait de [1]

Comme nous avons vu dans la partie 1.1.3, la décomposition va permettre de diminuer le nombre de paramètres. En effet, on passe de $d^N * N_l$ paramètres à $(N-1)*d*m^2 + d*m^2 * N_l$ paramètres. Le nombre de paramètres évolue maintenant linéairement avec N. En revanche,qu'en est-il du nombre d'opérations lors de la contraction de W^l avec Φ ?

3.1 Détermination du coût du calcul $W^l.\Phi$ avec une décomposition MPS

Pour calculer le nombre d'opérations, nous considérons que la dimension de liaison m est la même pour tous les α. De plus, nous considérons que le tenseur qui porte la dimension de l est le dernier tenseur.

Cette fois-ci, la contraction de W^l n'a pas la même forme, on peut donc avoir une stratégie différente afin de calculer la contraction. Premièrement, comme précédemment, nous commençons par les tenseurs avec les plus petits numéros d'indices soit ϕ^s_1 et $A^{\alpha_1}_{s_1}$.

Nous séparons la contraction en 2 étapes.

- Nous contractons le tenseur $A^{\alpha_1}_{s_1}$ avec ϕ^s_1. Cette opération nécessite $2 * m * d$ additions et multiplications. Elle nous donne un nouveau tenseur A^{α_1}

- Nous contractons A^{α_1} avec $A^{\alpha_1,\alpha_2}_{s_2}$. Cette opération nécessite $2*d*m^2$ additions et multiplications. Nous obtenons ainsi un nouveau tenseur qui a le même forme que $A^{\alpha_1}_{s_1}$, on peut donc ensuite répéter les opérations jusqu'au dernier tenseur.

Le dernier tenseur possède une dimension en plus, celle de l. Par conséquent, lorsque nous réalisons la 2ème étape pour l'avant dernier tenseur, cela nécessite $2 * m * d * N_l$ opérations. Enfin, pour le dernier tenseur, nous réalisons seulement la 1ère étape avec $2 * d * N_l$ opérations.

Au finale, on obtient un total de $2 * (N-1) * (d * m^2 + m * d) + 2 * N_l * (m * d + d)$ opérations. Voici un schéma résumant le processus :

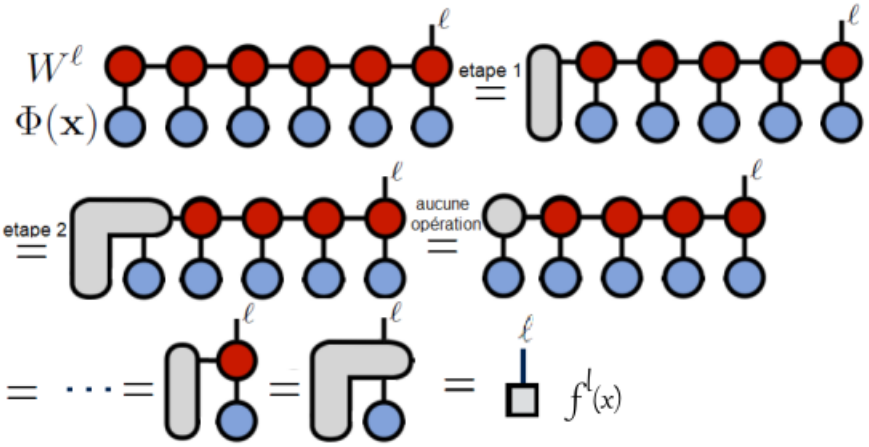

FIGURE 3.2 – Visualisation Contraction W^l sous forme MPS et Φ avec $N = 6$

Grâce à cette décomposition MPS, nous remarquons cette fois que le nombre d'opérations impliqué dans la contraction entre W^l et Φ évolue linéairement avec N. En utilisant une décomposition MPS, nous avons donc réussi à ce que le nombre de paramètres de W^l ainsi que le nombre d'opérations impliquées dans la contraction de W^l et Φ évolue linéairement avec N. C'est une bonne nouvelle, cela va sûrement nous permettre d'avoir un algorithme utilisable avec un N grand.

En revanche, l'algorithme que nous avions défini dans la partie 2 ne peut plus fonctionner. En effet, W^l n'a plus la même forme et on doit donc maintenant optimiser chaque tenseur A. Nous devons donc définir un nouvelle algorithme. Nous avons le choix entre un algorithme de Machine Learning standard avec une descente de gradient ou des algorithmes venant de la physique un peu plus particuliers. Les 2 prochaines sous-parties présentent les 2 algorithmes différents.

3.2 Descente de gradient avec pas fixe

Cette deuxième sous-partie est dédiée à une descente de gradient avec pas fixe pour optimiser W sous forme MPS. Comme dans la section précédente, nous fixons la dimension de l au dernier tenseur.

Tout d'abord, pour optimiser W, nous devons maintenant optimiser chaque tenseur A. Nous avons le choix d'optimiser tous les A à chaque itération ou bien d'optimiser un tenseur A par itération. Nous décidons d'optimiser un tenseur par itération. Cela permet la plupart du temps de converger plus vite vers un minimum local de la fonction de coût. En revanche, à la fin, la convergence est lente lorsque nous sommes proche du minimum.

En ce qui concerne la fonction de coût, nous gardons celle que nous avions défini à l'équation (2), c'est à dire une fonction de coût quadratique. Nous devons donc maintenant calculer le gradient de la fonction de coût par rapport à chaque tenseur A.

Nous commençons alors par développer C :

$$C(W) = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{2}(f^l(X_n) - y_n^l)^2 = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{2}(W^l.\Phi(X_n) - y_n^l)^2$$

$$C(W) = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{2}(\sum_{s_1,...,s_N}(\sum_{\alpha_1,...,\alpha_{N-1}} A_{s_1}^{\alpha_1}...A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{l;\alpha_{N-1}})\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N) - y_n^l)^2$$

Considérons maintenant que nous voulons calculer le gradient par rapport à $A_{s_1}^{\alpha_1}$. Nous pouvons alors réécrire $C(W)$ de la manière suivante :

$$C(W) = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{N_l}(\sum_{s_1,\alpha_1} A_{s_1}^{\alpha_1}(\sum_{\alpha_2,...,\alpha_{N-1}s_2,...,s_N} A_{s_2}^{\alpha_1,\alpha_2}...A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{l;\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)) - y_n^l)^2$$

$$C(W) = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{N_l}(\sum_{s_1,\alpha_1} A_{s_1}^{\alpha_1}\tilde{\Phi}_{s_1}^{l;\alpha_1}(X_n) - y_n^l)^2 = C(W) = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l=1}^{2}(A_{s_1}^{\alpha_1}.\tilde{\Phi}_{s_1}^{l;\alpha_1}(X_n) - y_n^l)^2$$

avec $\forall s_1, \alpha_1, l : \tilde{\Phi}_{s_1}^{l;\alpha_1}(X_n) = \sum_{\alpha_2,...,\alpha_{N-1}s_2,...,s_N} A_{s_2}^{\alpha_1,\alpha_2}...A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{l;\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_n^N}(x_N)$

Nous retrouvons alors la même forme que la premier algorithme avec $\tilde{\Phi}$ au lieu de Φ. Nous en déduisons alors que nous pouvons écrire le gradient sous forme vectorisé de la manière suivante :

$$\frac{\partial C(W)}{\partial A_{s_1}^{\alpha_1}} = \sum_{n=1}^{N_T}(A_{s_1}^{\alpha_1}.\tilde{\Phi}_{s_1}^{l;\alpha_1}(X_n) - y_n^l).\tilde{\Phi}_{s_1}^{l;\alpha_1}(X_n) \tag{8}$$

Nous avons trouvé une formule pour la gradient de C(W) par rapport au premier tenseur $A_{s_1}^{\alpha_1}$. Afin de trouver une formule pour les autres tenseurs, nous répétons les même étapes et nous trouvons les formules suivantes :

- $\forall i \in \{1, ..., N-1\}$,

$$\frac{\partial C(W)}{\partial A_{s_i}^{\alpha_{i-1},\alpha_i}} = \sum_{n=1}^{N_T}(A_{s_i}^{\alpha_{i-1},\alpha_i}.\tilde{\Phi}_{s_i}^{l;\alpha_{i-1},\alpha_i}(X_n) - y_n^l).\tilde{\Phi}_{s_i}^{l;\alpha_{i-1},\alpha_i}(X_n) \tag{9}$$

avec $\forall s_i, \alpha_{i-1}, \alpha_i : \tilde{\Phi}_{s_i}^{l;\alpha_{i-1},\alpha_i}(X_n) = \sum_{\{\alpha\}\neq\alpha_{i-1},\alpha_i\{s\}\neq s_i} A_{s_1}^{\alpha_1}...A_{s_{i-1}}^{\alpha_{i-2},\alpha_{i-1}}A_{s_{i+1}}^{\alpha_i,\alpha_{i+1}}...$
$A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{l;\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

- pour le dernier tenseur,

$$\frac{\partial C(W)}{\partial A_{s_N}^{l;\alpha_{N-1}}} = \sum_{n=1}^{N_T}(A_{s_N}^{l;\alpha_{N-1}}.\tilde{\Phi}_{s_N}^{\alpha_{N-1}}(X_n) - y_n^l) \otimes \tilde{\Phi}_{s_N}^{\alpha_{N-1}}(X_n)) \tag{10}$$

avec$\forall s_N, \alpha_{N-1} : \tilde{\Phi}_{s_N}^{\alpha_{N-1}}(X_n) = \sum_{\{\alpha\}\neq\alpha_{N-1}\{s\}\neq s_N} A_{s_1}^{\alpha_1}...A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

Tous ces calculs ne sont pas forcément faciles à comprendre, il y a beaucoup d'indices et nous nous perdons vite. C'est pourquoi, nous préférons souvent utiliser des représentations graphiques. Voici une représentation graphique qui explique les étapes à réaliser afin d'obtenir une forme particulière pour ensuite calculer le gradient pour des tenseurs qui ne sont pas aux extrémités :

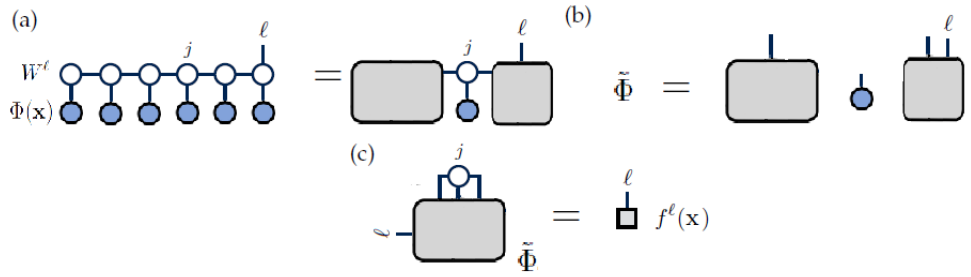

FIGURE 3.3 – Explication calcul gradient

avec a)Réorganisation contraction W^l et Φ, b)Définition de $\tilde{\Phi}$ et c) Nouvelle forme de f^l

Nous avons donc nos formules de gradient.

Un des principes clés de l'algorithme est que nous n'avons pas besoin de recalculer $\tilde{\Phi}$ à chaque fois que nous passons au tenseur suivant. Définissons $\tilde{\Phi}^1$ et $\tilde{\Phi}^2$ les composantes respectivement à gauche et à droite du tenseur que nous sommes entrain d'optimiser.

Lors de l'optimisation de $A^{\alpha_1}_{s_1}$, nous contractons $A^{\alpha_{N-1}}_{s_N}$ et ϕ_{s_N} puis nous stockons la valeur. Nous contractons la valeur obtenue par $A^{\alpha_{N-2},\alpha_{N-1}}_{s_{N-1}}$ et $\phi_{s_{N-1}}$ pour ensuite la stocker. Itérativement, nous construisons $\tilde{\Phi}$ de $A^{\alpha_1}_{s_1}$ en stockant les valeurs intermédiaires. Comme lorsque nous avançons, nous modifions que les tenseurs qui sont à gauche du tenseur que nous optimisons, cette étape permet aussi de stocker tous les tenseurs $\tilde{\Phi}^2$ dont nous aurons besoin pour optimiser chaque tenseur A^α_s.

Pour $\tilde{\Phi}^1$, nous allons dans l'autre sens. Nous commençons avec aucun tenseur pour $A^{\alpha_1}_{s_1}$ et $A^{\alpha_1}_{s_1}$ pour $A^{\alpha_1,\alpha_2}_{s_2}$. Puis, dès que nous venons d'appliquer un pas de gradient à un tenseur, nous contractons ce nouveau tenseur avec $\tilde{\Phi}^1$ afin de construire le nouveau $\tilde{\Phi}^1$ pour le tenseur suivant.

Par la suite, nous utiliserons une descente de gradient simple avec un pas fixe. Voici un pseudo-code de cet algorithme :

Algorithm 2 Optimiser W sous forme MPS

Entrée: `tenseur` W^l, `pas fixe` α, `précision p`
Procédure:
  $erreur \leftarrow C(W)$
  Tant que $erreur > p$ **faire**
    Pour i=1,...,N **faire**
      $A^\alpha_s \leftarrow A^\alpha_s - \alpha \frac{\partial C(W)}{\partial A^\alpha_s}$
      $erreur \leftarrow C(W)$
    fin Pour
  fin Tant que

Nous avons maintenant notre deuxième algorithme que nous nommerons MPS-GD. Avant de le tester numériquement, nous allons définir un deuxième algorithme un peu moins standard.

3.3 Algorithme de Balayage

Nous sommes maintenant prêt pour définir un troisième algorithme. Cette fois-ci, nous allons utiliser un algorithme qui est inspiré de la DMRG (Density Matrix Renormalization Group). Cette méthode est à la base utilisée en physique pour calculer une forme MPS étant donné un tenseur W non structuré.

Ici, nous allons proposer une version similaire de cet algorithme qui balaye de gauche à droite la forme MPS de W, en minimisant itérativement la fonction de coût.

Tout d'abord, nous utilisons encore une fois une fonction de coût quadratique. En revanche, cette fois-ci, nous allons optimiser les tenseurs A_s^α 2 par 2. Cela va nous permettre de modifier la taille des dimensions de liaison au fur et à mesure que nous allons de gauche à droite le long de la forme MPS. Comme précédemment, nous fixons l sur un tenseur quelconque.

• Pour commencer, nous combinons 2 tenseurs $A_{s_j}^{\alpha_{j-1},\alpha_j}$ et $A_{s_{j+1}}^{\alpha_j,\alpha_{j+1}}$ en les contractant selon l'indice α_i pour former un tenseur $B_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}$ comme sur l'image ci-dessous :

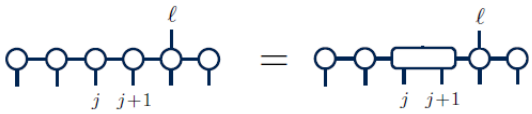

FIGURE 3.4 – Représentation étape 1 (DMRG)
extrait de [1]

Il faut faire attention ici. En effet, nous n'avons pas représenter les cas particuliers dans lesquels l appartient au jème ou (j+1)ème tenseur. Dans ce cas, le tenseur B a un ordre de plus qui correspond à l.

• Par la suite, nous devons calculer le gradient. Cela reste similaire à l'algorithme MPS-GD. En effet,nous reformulons la fonction f^l en définissant un tenseur $\tilde{\Phi}$ puis nous calculons le gradient. Nous obtenons beaucoup de formules différentes en fonction de l'emplacement ou si nous allons vers la droite ou la gauche.

•cas 1 : si le 1er tenseur est dans B

$$\frac{\partial C(W)}{\partial B_{s_1,s_2}^{\alpha_0}} = \sum_{n=1}^{N_T} (B_{s_1,s_2}^{\alpha_0}.\tilde{\Phi}_{s_1,s_2}^{l;\alpha_0}(X_n) - y_n^l).\tilde{\Phi}_{s_1,s_2}^{l;\alpha_0}(X_n) \tag{11}$$

avec $\forall l, s_1, s_2, \alpha_0 : \tilde{\Phi}_{s_1,s_2}^{l;\alpha_0}(X_n) = \sum_{\{\alpha\}\neq\alpha_0\{s\}\neq s_0,s_1} A_{s_3}^{\alpha_2,\alpha_3}...A_{s_i}^{l;\alpha_{i-1},\alpha_i}...A_{s_N}^{\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

•cas 2 : si l est porté par B

$$\frac{\partial C(W)}{\partial B_{s_j,s_{j+1}}^{l;\alpha_{j-1},\alpha_{j+1}}} = \sum_{n=1}^{N_T} (B_{s_j,s_{j+1}}^{l;\alpha_{j-1},\alpha_j}.\tilde{\Phi}_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}(X_n) - y_n^l) \otimes \tilde{\Phi}_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}(X_n) \tag{12}$$

avec $\forall s_j, s_{j+1}, \alpha_{j-1}, \alpha_{j+1} : .\tilde{\Phi}_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}(X_n) = \sum_{\{\alpha\}\neq\alpha_{j-1},\alpha_{j+1}\{s\}\neq s_j,s_{j+1}} A_{s_1}^{\alpha_1}...A_{s_{j-1}}^{\alpha_{j-2},\alpha_{j-1}}A_{s_{j+2}}^{\alpha_{j+1},\alpha_{j+2}}...$
$A_{s_N}^{\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

cas 3 : si le dernier tenseur est dans B

$$\frac{\partial C(W)}{\partial B_{s_{N-1},s_N}^{\alpha_{N-1}}} = \sum_{n=1}^{N_T} (B_{s_{N-1},s_N}^{\alpha_{N-1}}.\tilde{\Phi}_{s_{N-1},s_N}^{l;\alpha_{N-1}}(X_n) - y_n^l).\tilde{\Phi}_{s_{N-1},s_N}^{l;\alpha_{N-1}}(X_n) \tag{13}$$

avec $\forall s_{N-1}, s_N, \alpha_{N-1} : \tilde{\Phi}_{s_{N-1},s_N}^{l;\alpha_{N-1}}(X_n) = \sum_{\{\alpha\}\neq\alpha_{N-1}\{s\}\neq s_{N-1},s_N} A_{s_1}^{\alpha_1}...A_{s_i}^{l;\alpha_{i-1},\alpha_i}...$
$A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

•cas 4 : si aucunes des conditions précédentes n'est respectées

$$\frac{\partial C(W)}{\partial B_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}} = \sum_{n=1}^{N_T} (B_{s_j,s_{j+1}}^{\alpha_{j-1},\alpha_{j+1}}.\tilde{\Phi}_{s_j,s_{j+1}}^{l;\alpha_{j-1},\alpha_{j+1}}(X_n) - y_n^l).\tilde{\Phi}_{s_j,s_{j+1}}^{l;\alpha_{j-1},\alpha_{j+1}}(X_n) \tag{14}$$

avec $\forall s_j, s_{j+1}, \alpha_{j-1}, \alpha_{j+2} : \tilde{\Phi}_{s_j,s_{j+1}}^{l;\alpha_{j-1},\alpha_{j+1}}(X_n) = \sum_{\{\alpha\}\neq\alpha_{j-1},\alpha_{j+1}\{s\}\neq s_j,s_{j+1}} A_{s_1}^{\alpha_1}...A_{s_{j-1}}^{\alpha_{j-2},\alpha_{j-1}}A_{s_{j+2}}^{\alpha_{j+1},\alpha_{j+2}}...$
$A_{s_i}^{l;\alpha_{i-1},\alpha_i}...A_{s_{N-1}}^{\alpha_{N-2},\alpha_{N-1}}A_{s_N}^{\alpha_{N-1}}\phi^{s_1}(x_n^1)...\phi^{s_N}(x_n^N)$

Voici une représentation de $\tilde{\Phi}$ et de la nouvelle forme de f^l dans le cas n°2 :

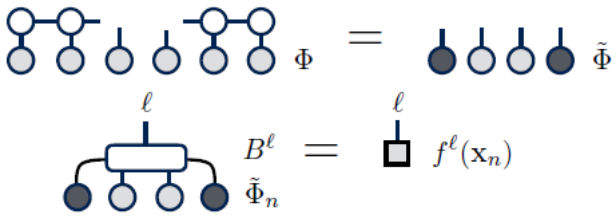

FIGURE 3.5 – Visualisation étape 2 (DMRG)
extrait de [1]

• Ensuite, nous appliquons une descente de gradient à pas fixe, c'est à dire $B = B - \alpha * \frac{\partial C(W)}{\partial B}$. Nous avons appliqué un pas de gradient à B. Néanmoins, nous devons maintenant séparer B en 2 afin de retrouver la forme initiale de W.

• Dans cette étape, nous séparons le tenseur B en 2 tenseurs en utilisant une SVD (Décomposition en valeurs singulières). Pour cela, nous considérons le tenseur comme une matrice en mettant en ligne les indices liés au tenseur de gauche et en colonne les indices liés au tenseur de droite. Nous obtenons ainsi la décomposition suivante du tenseur B (dans le cas ou l ne fait pas partie du tenseur B) :

$$B^{\alpha_{j-1},\alpha_{j+1}}_{s_j,s_{j+1}} = \sum_{\alpha_j,\alpha_{j'}} U^{\alpha j-1}_{s_j,\alpha_{j'}} S^{\alpha_j}_{\alpha_{j'}} V^{\alpha_j,\alpha j+1}_{s_{j+1}} \tag{15}$$

• Enfin, pour cette dernière étape il y a plusieurs variantes que nous allons appeler DMRG-MSENS, DMRG-MSENS et DMRG-EQUI. Dans les différentes versions, la différence va être le sens de la contraction avec S :

•**DMRG-MSENS :** $A^{\alpha_{j-1},\alpha_{j'}}_{s_j} = U^{\alpha j-1}_{s_j,\alpha_{j'}}$ et $A^{\alpha_{j'},\alpha_{j+1}}_{s_{j+1}} = S^{\alpha_j}_{\alpha_{j'}}.V^{\alpha_j,\alpha j+1}_{s_{j+1}}$ vers la droite
$A^{\alpha_{j-1},\alpha_j}_{s_j} = U^{\alpha j-1}_{s_j,\alpha_{j'}}.S^{\alpha_j}_{\alpha_{j'}}$. et $A^{\alpha_j,\alpha_{j+1}}_{s_{j+1}} = V^{\alpha_j,\alpha j+1}_{s_{j+1}}$ vers la gauche (les valeurs propres sont contractées avec le tenseur qui refera parti de B au pas de gradient suivant)

•**DMRG-SENSI :** $A^{\alpha_{j-1},\alpha_j}_{s_j} = U^{\alpha j-1}_{s_j,\alpha_{j'}}.S^{\alpha_j}_{\alpha_{j'}}$. et $A^{\alpha_j,\alpha_{j+1}}_{s_{j+1}} = V^{\alpha_j,\alpha j+1}_{s_{j+1}}$ vers la droite
$A^{\alpha_{j-1},\alpha_{j'}}_{s_j} = U^{\alpha j-1}_{s_j,\alpha_{j'}}$ et $A^{\alpha_{j'},\alpha_{j+1}}_{s_{j+1}} = S^{\alpha_j}_{\alpha_{j'}}.V^{\alpha_j,\alpha j+1}_{s_{j+1}}$ vers la gauche
(les valeurs propres sont contractées avec le tenseur qui ne refera pas parti de B au pas de gradient suivant)

•**DMRG-EQUI :** $A^{\alpha_{j-1},\alpha_j}_{s_j} = U^{\alpha j-1}_{s_j,\alpha_{j'}}.\sqrt{S^{\alpha_j}_{\alpha_{j'}}}$. et $A^{\alpha_{j'},\alpha_{j+1}}_{s_{j+1}} = \sqrt{S^{\alpha_j}_{\alpha_{j'}}}.V^{\alpha_j,\alpha j+1}_{s_{j+1}}$
Voici un schéma qui représente les 2 dernières étapes :

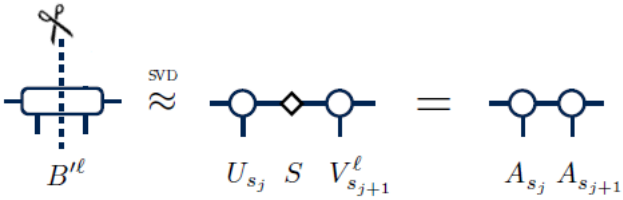

FIGURE 3.6 – Visualisation étape 3 et 4 (DMRG)
extrait de [1]

Après avoir respecté les 4 étapes : construction de B, construction de $\tilde{\Phi}$, pas de gradient et de découpage avec SVD, nous passons aux tenseurs suivants en balayant chaque tenseur de la forme MPS en allant vers la droite puis vers la gauche afin de revenir au premier tenseur.

Cet algorithme a plusieurs avantages :

- Nous pouvons choisir la dimension de liaison au fur et à mesure de l'algorithme. Par exemple, on peut garder seulement les premières valeurs propres de la SVD ou bien prendre celle qui sont supérieures à un certain seuil. Cela permet ainsi de contrôler le coût de l'algorithme.

- Comme avec MPS-GD, lorsque l'on passe au tenseur suivant, nous n'avons pas besoin de recalculer entièrement $\tilde{\Phi}$. Cela implique donc que chaque étape de l'algorithme ne dépend pas de la taille N des images. Par conséquent, le coût total de l'algorithme évolue linéairement avec N.

Voici le pseudo-code de l'algorithme :

Algorithm 3 Optimiser W sous forme MPS

Entrée: `tenseur` W^l, `pas fixe` α, `précision p, nom algorithme algo`
Procédure:
  $erreur \leftarrow C(W)$
  Tant que $erreur > p$ **faire**
    `sens` $\leftarrow$ `droite`
    Pour i=1,...,N-1,N-1,...,1 **faire**
      $B \leftarrow A^{\alpha}_{s_i}.A^{\alpha}_{s_{i+1}}$
      $B^i \leftarrow B - \alpha\frac{\partial C(W)}{\partial B}$
      $erreur \leftarrow C(W)$
      $(A^{\alpha}_{s_i}, A^{\alpha}_{s_{i+1}}) \leftarrow$ `SVDB`$(B, algo, sens)$
      Si $i == N-1$ **Alors**
        `sens` $\leftarrow$ `gauche`
      fin Si
    fin Pour
  fin Tant que

avec

Algorithm 4 `SVDB`

Entrée: `tenseur` B, `seuil`$\gamma = 10^{-10}$, `nombre m-max` $= 30$, `numéro algorithme algo, sens`
Procédure: $U, S, V \leftarrow$ `Singular Value Decomposition(B)`
  `dim-liaison` $\leftarrow \min(longueur(s), m - max, nombredes > gamma)$
  $S \leftarrow$ `les dim-liaison plus grandes valeurs propres de S`
  Si $algo ==$ `DMRG-MSENS` **Alors**
    Si $sens ==$ `droite` **Alors**
      $T^1 = U$ `et` $T^2 = S.V$
    else
      $T^1 = U.S$ `et` $T^2 = V$
    fin Si
  Sinon Si $algo ==$ `DMRG-SENSI` **Alors**
    Si $sens ==$ `droite` **Alors**
      $T^1 = U.S$ `et` $T^2 = V$
    else
      $T^1 = U$ `et` $T^2 = S.V$
    fin Si
  else
    $T^1 = U.\sqrt{S}$ `et` $T^2 = \sqrt{S}.V$
  fin Si
  return (T^1, T^2)

Nous pouvons maintenant tester numériquement les algorithmes MPS-GD, DMRG-MSENS ,DMRG-SENSI et DMRG-EQUI.

3.4 Applications numériques

Nous allons maintenant pouvoir tester numériquement nos nouveaux algorithmes.

Les 4 nouveaux algorithmes possèdent des hyperparamètres en commun : pas de la descente de gradient α m la dimension de liaison.

En revanche, DMRG-MSENS ,DMRG-SENSI et DMRG-EQUI en possèdent plus : $m-max$ la de dimension de liaison la plus grande et γ la borne inférieure d'acceptation des valeurs propres. Dans le cadre de notre première exemple simple, nous fixons m-max=10 et $\gamma = 10^{-10}$.

Pour cette première application numérique des nouveaux algorithmes, nous allons utiliser la même approche que pour le 1. En effet, nous allons créer une base d'images en noir et blanc de 9 pixels aléatoirement puis nous affectons la moitié des images au label 0 puis l'autre moitié au label 1. Nous avons généré les paramètres de W de manière aléatoire selon des tirages d'une loi normale centrée réduite et nous prenons m=10 comme dimension de liaison initiale de la décomposition MPS. Voici nos résultats pour les 3 différents algorithmes DMRG-MSENS ,DMRG-SENSI et DMRG-EQUI :

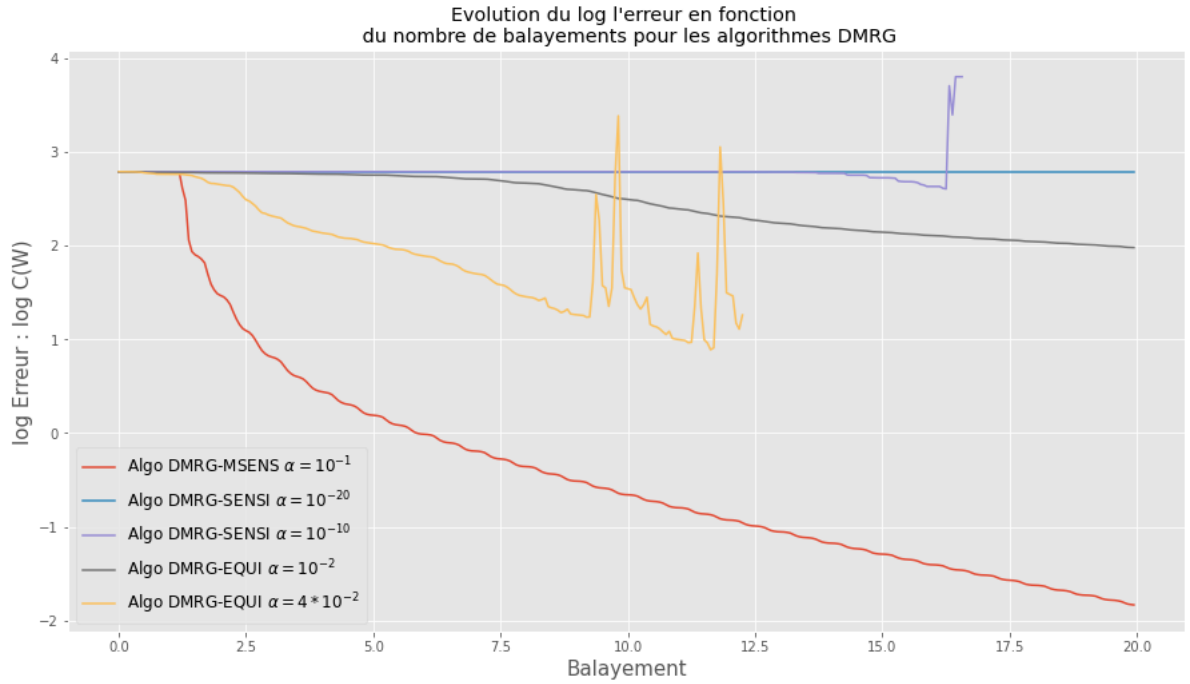

FIGURE 3.7 – Evolution de l'erreur (algorithmes DMRG)

Grâce à cette application numériques, on peut observer de nombreuses choses :

- Le programme dont l'erreur converge le plus vite vers 0 est le programme DMRG-MSENS.

- Les programmes DMRG-SENSI et DMRG-EQUI ne sont pas très stables. En effet, nous n'arrivons pas à trouver un pas fixe qui permet à l'erreur de converger vers 0. Lorsque notre pas est trop grand, l'erreur explose vers $+\infty$ comme nous pouvons le voir sur le graphique ci-dessus(nous avons arrêté d'afficher l'erreur lorsque celle-ci explose). Pour ces 2 programmes, il faudrait donc implémenter des méthodes de pas adaptatifs afin que le pas diminue lorsque l'erreur s'approche de 0.

- Pour le programme DMRG-MSENS, nous avions le même problème. Nous avons donc fait une petite modification au programme. Lors du premier aller-retour, nous réalisons seulement la SVD mais on ne réalise pas de descente de gradient. En effet, la méthode 3.1 associe toujours les valeurs propres au tenseurs que nous allons modifier par la suite. Par conséquent, tous les grands nombres sont contenues dans le tenseur que nous allons optimiser par la suite et ne vont donc pas intervenir directement dans le calcul du gradient.

Voici nos résultat pour le programme MPS-GD avec $\alpha = 10^{-2}$:

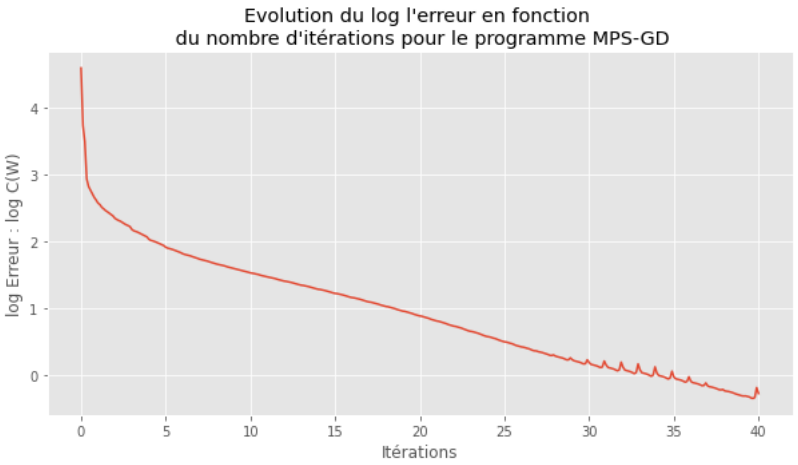

FIGURE 3.8 – Evolution de l'erreur (MPS-GD)

MPS-GD nous donne de bons résultats. Il converge assez vite vers 0 comme l'algorithme DMRG-MSENS.

Nous décidons par la suite de s'intéresser seulement aux algos MPS-GD et DMRG-MSENS . Avec l'algorithme 1, nous avions pu voir que les itérations prenaient de plus en plus de temps lorsque N augmentait. Nous allons donc regarder la même chose pour ces 2 algorithmes. Pour cela, nous allons mesurer le temps d'exécution d'une itération pour différentes valeurs de N. Voici nos résultats :

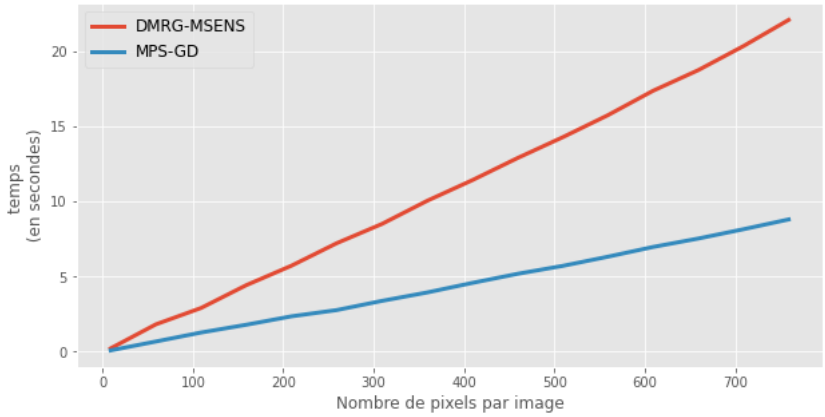

FIGURE 3.9 – Evolution du temps en fonction du nombre de pixels

Nous observons cette-fois ci que l'évolution est linéaire par rapport à N. C'est une bonne nouvelle et nous allons donc pouvoir tester nos 2 programmes sur les données MNIST. MPS-GD est un peu plus rapide que DMRG-MSENS mais cela ne veut rien dire car nous ne savons pas à quelle vitesse ils convergent vers 0.

Avant de tester sur les données MNIST, nous allons refaire l'expérience précédente avec des données aléatoires mais cette fois avec 100 pixels pour voir l'évolution de l'erreur pour les 2 algorithmes.

Commençons par observer les résultats pour MPS-GD.

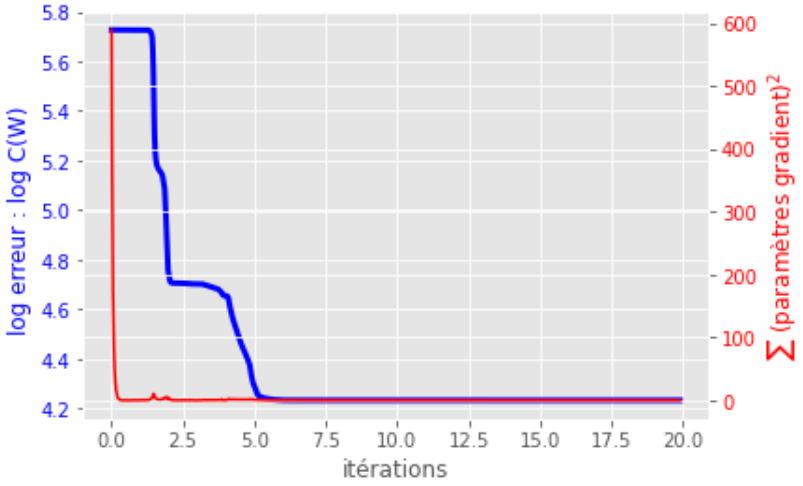

FIGURE 3.10 – Evolution de l'erreur avec N=100 algorithme DRMG-MSENS

Nous remarquons que l'erreur commence par diminuer. Malheureusement, à partir d'un certain moment, l'erreur commence à stagner et à ne pas diminuer. Pour mieux comprendre ce phénomène, nous avons affiché sur le même graphe la somme de tous les paramètres du gradient au carré. Nous remarquons alors que ceux-ci tendent vers 0. C'est pourquoi l'erreur ne diminuent plus. Ce phénomène s'appelle "vanishing gradient". Ceci peut s'expliquer par les décisions que nous avons prises lors de la SVD. En effet, comme nous mettons tous les grands nombres dans le tenseur que nous allons optimiser par la suite, ils n'apparaissent pas dans le calcul du gradient. Par conséquent, il ne reste que les petits nombres qui multipliés un par un tendent vers 0.

Il existe différentes manière de résoudre le problème. Nous pouvons utiliser des techniques de pas adaptatifs qui permettraient d'augmenter le pas lorsque la valeur du gradient diminue ou bien utiliser d'autres techniques de gradient. Dans le cadre de ce rapport, nous ne développerons pas ces méthodes.

Nous testons alors MPS-GD sur ces mêmes données. Nous obtenons les résultats suivants :

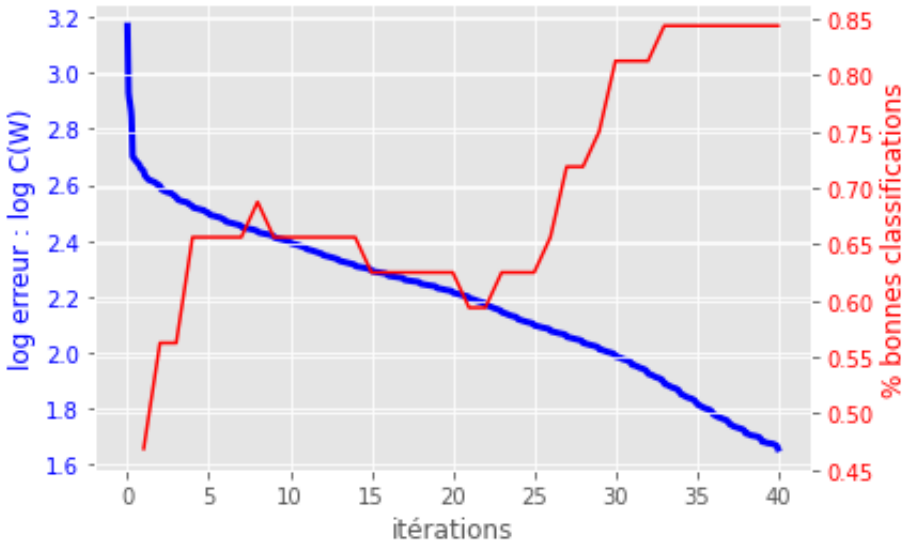

FIGURE 3.11 – Evolution de l'erreur avec N=100 algorithme MPS-GD

Cette fois-ci, nous avons pas de problème de "vanishing gradient". L'erreur diminue bien et le pourcentage de réussite au niveau de la classification augmente au fur et à mesure.

Il nous reste donc plus qu'à tester le programme MPS-GD avec MNIST.

Chapitre 4

Applications aux données MNIST

Notre objectif est maintenant de regarder si le programme MPS-GD est capable de classifier les images de MNIST.

MNIST est une base d'images d'entiers manuscrits, avec pour labels, les entiers correspondants à chaque image [4].

Pour commencer, nous allons nous intéresser seulement à une partie des données. En effet, MNIST regroupe 60 000 images pour l'entraînement, un entraînement total sur l'ensemble des données prendraient beaucoup de temps.

Nous nous limiterons donc à classer seulement les images manuscrites de 0 et de 1 sur un ensemble d'une dizaine d'images. Voici par exemple les 2 premières images de nos données :

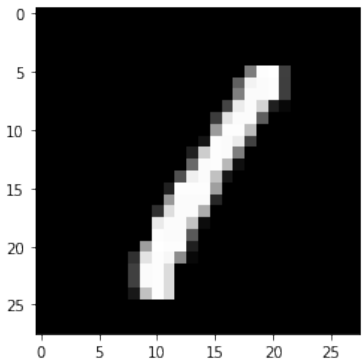

FIGURE 4.1 – Image d'un 1 manuscrit

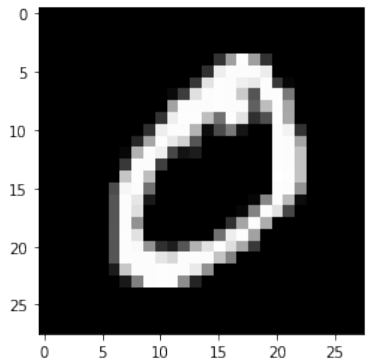

FIGURE 4.2 – Image d'un 0 manuscrit

Pour cela, nous utilisons l'algorithme MPS-GD. Les seuls hyperparamètres du modèle sont α, le pas de la descente, ainsi que m, la dimension de liaison. Dans notre cas, nous prenons m=10 et $\alpha = 10^{-10}$.

Nous initialisons tous les paramètres des A_s^α de la décomposition MPS par des tirages de loi normale centrée réduite avec m=10. Il est difficile d'obtenir de bonnes valeurs initiales pour la fonction f^l. En effet, N étant grand, il y a de grande variation dans les sorties de la fonction de décision.

Nous essayons d'avoir des valeurs initiales ni trop grandes, ni trop petites en appliquant un terme compensateur à tous les paramètres.

Nous obtenons après 600 cycles une erreur de 27,19 avec une diminution de 0.01 par cycle. Voici nos résultats :

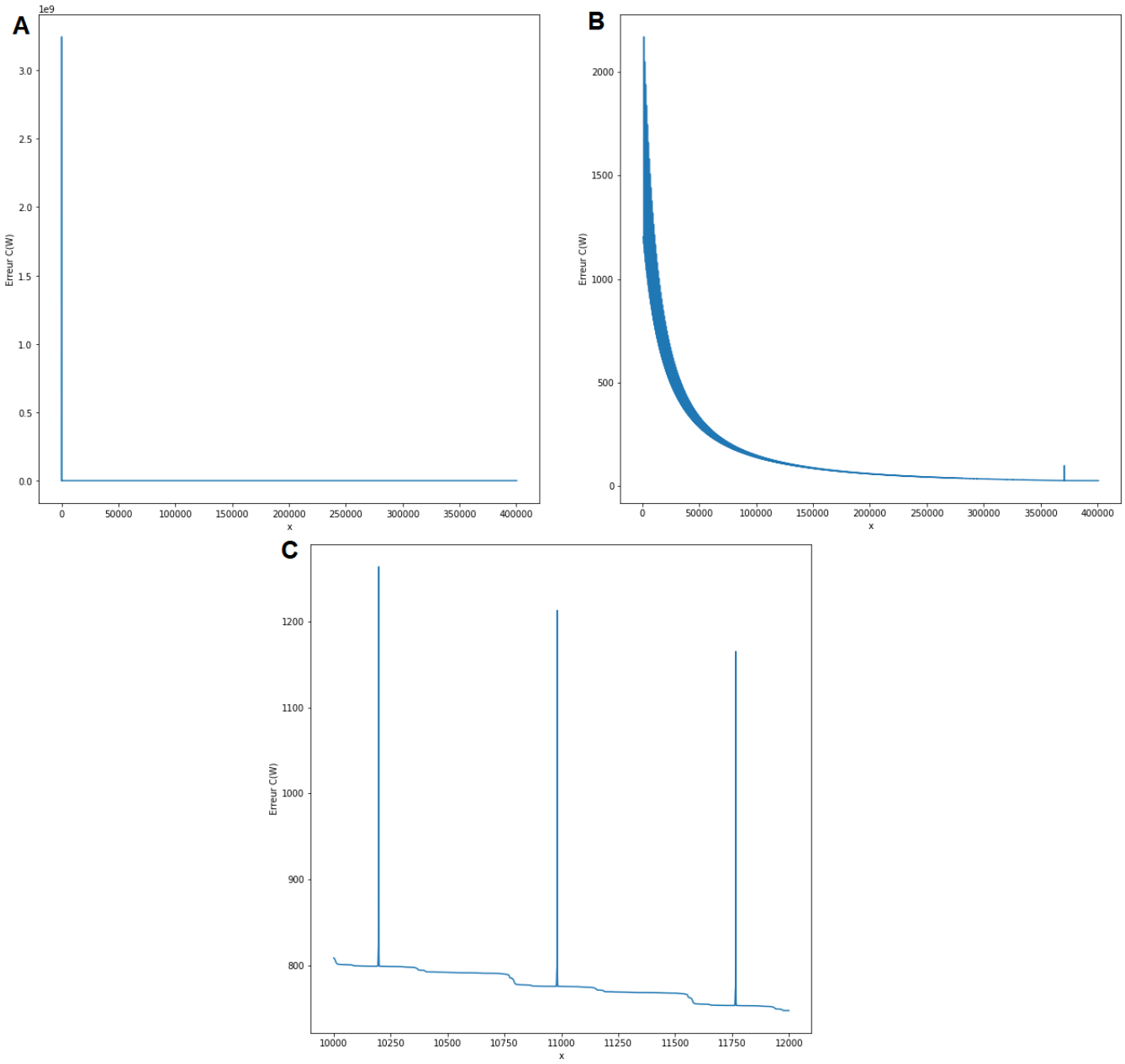

FIGURE 4.3 – Evolution de l'erreur de MPS-GD avec MNIST (images de 784 pixels)

Nos résultats sont décevants. Sur le graphique A, nous pouvons observer l'évolution globale de l'erreur. Le graphique B représente un zoom après les 1000 premières itérations et enfin le graphique C met en évidence les 10 000 et 12 000 itérations.

Sur les 3 graphiques, une itération correspond à un pas de gradient. Voici nos constations :

- L'erreur diminue au fur et à mesure des itérations comme nous pouvons le voir sur le graphique A. Cela est déjà une bonne chose et montre qu'il n'y a surement pas d'erreur dans les formules.

- L'erreur diminue beaucoup lors des premières itérations puis diminue plus doucement. Nous pouvons observer cela en comparant les graphiques A et B. En effet, sur le graphique A, nous observons la diminution générale de l'erreur mais lorsqu'on nous faisons un zoom après les 1000 premières itérations, la diminution est beaucoup plus faible.

- Au bout des 600 itérations, l'erreur diminue très peu. Nous pourrions penser qu'augmenter le pas de la descente est une solution. Néanmoins, cela ne fonctionne pas. En effet, nous observons sur le graphique C, qu'il y a de temps en temps des pics au niveau de l'erreur. Nous retrouvons ces pics avec une période d'environ 800 itérations. Ces pics peuvent s'expliquer par une descente de gradient

avec un pas trop élevé pour un certain tenseur. Ce tenseur vient donc bloquer toute la descente car à cause de lui, nous ne pouvons pas augmenter le pas de la descente afin que la fonction de coût converge plus vite vers 0.

Pour résoudre ce problème, nous allons diminuer la taille des images en passant de 28x28 à 14x14 pixels par image.

Pour diminuer la taille de l'image, nous réalisons la moyenne sur des groupes de 4 pixels. Cela nous permet d'obtenir de nouvelles images de 196 pixels. Cette fois-ci, nous allons réaliser notre test sur un sous-ensemble de 50 images. Nous obtenons alors le résultat suivant :

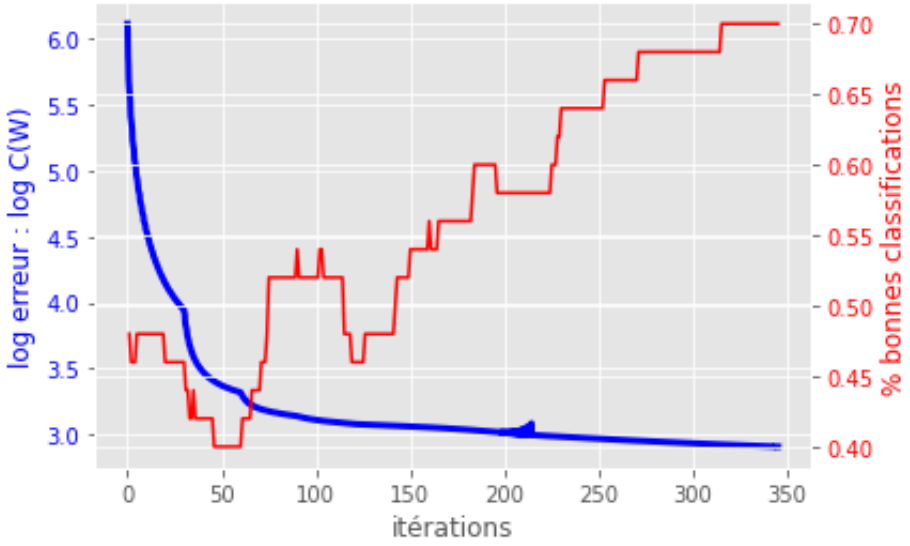

FIGURE 4.4 – Evolution de l'erreur de MPS-GD avec MNIST (images de 196 pixels)

Nous remarquons cette-fois ci que l'algorithme réagit mieux. En effet, l'erreur diminue beaucoup et le taux de bonnes classifications augmente même si la fonction de coût prend du temps à converger vers 0.

Pour tester l'algorithme, nous allons calculer le taux d'images bien classées dans une base de données qui n'a pas été utilisée lors de l'apprentissage. Pour cela, nous sélectionnons donc 2115 images différentes des 50 de la phase d'apprentissage et nous trouvons un taux d'images bien classées de 19.9%. Ce résultat est très faible mais il s'explique très bien.

En effet, nous avons utilisé que 50 images pour former notre modèle alors que le tenseur W possède au total 38860 paramètres. Notre modèle est donc face à un problème de surapprentissage ou surajustement. Nous ne donnons pas assez de données au modèle pour qu'il apprenne de manière générale comment classifier des images de 0 et de 1 manuscrits. Dans notre cas, il a appris à classer seulement les images utilisées lors de la phase d'apprentissage.

Afin de résoudre ce problème, il faudrait entrainer notre modèle avec plus de données. Malheureusement, la fonction de coût converge trop lentement vers 0 et la durée des itérations augmentent avec le nombre de données. Nous devons donc trouver un nouvel algorithme ou une modification de MPS-GD ou de DMRG-MSENS afin que la fonction de coût converge plus vite vers 0 ou bien rajouter un terme de régularisation ...

Chapitre 5

Conclusion

En conclusion, pour atteindre notre objectif de classification d'images avec des tenseurs, nous avons franchi plusieurs étapes.

Tout d'abord, nous avons créé un réseau de tenseurs sans aucune structure particulière. Néanmoins, nous nous sommes vite rendu compte que l'algorithme lié à ce modèle était fonctionnel mais ne pouvait pas être retenu à cause de son temps d'exécution.

Par la suite, nous avons donné une forme particulière à notre tenseur W^l en appliquant une décomposition MPS. Nous avons ainsi crée 4 nouveaux algorithmes : MPS-GD , DMRG-MSENS , DMRG-SENSI et DMRG-EQUI. Parmi ceux-ci, nous avons décidé de garder seulement les 2 premiers. En effet, ils étaient plus stables et la fonction de coût convergeait plus rapidement vers 0.

Nous avons donc testé MPS-GD et DMRG-MSENS sur des images de grandes tailles et remarqué que l'algorithme DMRG-SENS ne fonctionnait pas correctement.

Finalement, nous avons testé l'algorithme MPS-GD sur la base de données MNIST qui contient des images de très grande taille, le résultat n'était pas satisfaisant. Nous avons donc décidé de diviser par 2 le nombre pixels horizontales et verticales de ces images. Suite à cette modification, l'algorithme MPS-GD fonctionnait mieux mais la fonction de coût convergeait très lentement vers 0 avec un grand ensemble de données. Nous pouvons donc conclure que la méthode réalisée sur les données d'entrainement n'est pas applicable pour classifier les images de la base de donnée MNIST.

Pour résoudre ce problème, d'autres solutions s'offrent à nous et pourraient être testées dans la continuité de ce projet. Nous pourrions, par exemple, utiliser une méthode des moindres carrés alternés afin de ne pas avoir à choisir le pas de la descente ou bien utiliser une pénalisation de ridge afin de rendre notre fonction de coût plus facile à minimiser ...

Bibliographie

[1] Miles Stoudenmire et David Schwab. Supervised learning with tensor networks. *Advances in Neural Information Processing Systems 29, 4799*, Thu, 18 May 2017.

[2] Vladimir Vapnik. The nature of statistical learning theory. *Springer-Verlag New York*, 2000.

[3] W Waegeman. A kernel-based framework for learning graded relations from data. *Fuzzy Systems, IEEE Transactions*, Dec 2012.

[4] Corinna Cortes et Christopher JC Burges. Yann Lecun. Mnist handwritten digit database. `http://yann.lecun.com/exdb/mnist/`.

Table des figures

Annexe A

Code Algorithme 1

```
# -*- coding: utf-8 -*-

###### Importation module ######
import numpy as np
import matplotlib.pyplot as plt
import tensorly as tl
from matplotlib import style
import time

###### Fonction mapping ######
def phi(x):
    return np.array([np.cos((np.pi*x)/2),np.sin((np.pi*x)/2)])

#Creation dataset
Nt=32 #Nombre de training example
N=9 #Nombre de pixel par image

#Creation des images
data=np.random.random((Nt,3,3))

#Creation des labels
y=np.zeros((Nt,2)) ; y[0:16,0] = 1 ; y[16:Nt,1]=1

W=tl.tensor(np.random.randn(2,2,2,2,2,2,2,2,2,2))

alpha=0.1 #le pas de la descente de gradient
err=[]
epoch=40

###### Programme 1 ######

for z in range(epoch):
  gradW=np.zeros(W.shape) #On fixe le tenseur du gradient de W a 0
  cout = 0 #On fixe la fonction de cout a 0
  for n in range(Nt):

    ### Calcul du gradient ###
    temp1=phi(data[n].reshape(-1,))  #Pour chaque image,
    #on calcule la transformations des pixels par la phi
    Wtemp=W
    #Calcul de la premiere partie gradient
    for i in range(temp1.shape[1]):
```

```
        phi_t=temp1[:,i].reshape(2,1)
        temp2=np.dot(phi_t,phi_t.T)
        Wtemp=tl.tenalg.mode_dot(Wtemp,temp2,i+1)

    #Calcul de la deuxieme partie gradient
    phi_t=y[n,:]
    for i in range(N):
        phi_t=np.tensordot(phi_t,temp1[:,i],0)

    #Ajout du terme du gradient associe a chaque image
    gradW+=Wtemp-phi_t

    ### Calcul du cout ###
    temp1=phi(data[n].reshape(-1,))
    Wtemp=W
    for i in range(temp1.shape[1]):
        phi_t=temp1[:,i]
        Wtemp=tl.tenalg.mode_dot(Wtemp,phi_t,1)
    temp2 = Wtemp-y[n,:]
    cout+= sum([i**2 for i in temp2])

  #Realisation d'une descente de gradient
  W=W-alpha*gradW
  err.append( (1/2)*cout )
print(f"L erreur finale est de {err[len(err)-1]}.")
```

Annexe B

Code Algorithme MPS-GD

```
# -*- coding: utf-8 -*-
##### Importation module #####
import numpy as np
import matplotlib.pyplot as plt
import tensorly as tl
import time
import math
from tensorflow.keras.datasets import mnist
from matplotlib import style

##### Fonction #####

#Fonction mapping
def phi(x):
    return np.array([np.cos((np.pi*x)/2),np.sin((np.pi*x)/2)])

##### Creation Dataset #####

nbTraining=32 #Nombre de training example
N=100 #Nombre de pixel par image

#Creation des images
training_data=np.random.random((nbTraining,N))

#Creation des labels
training_y=np.zeros((nbTraining,2)) ; training_y[0:16,0] = 1 ;
training_y[16:nbTraining,1]=1

##### Algo MPS-GD #####

##Construction des A et de W
W=[] #dim0 : A
dimalpha=10
diml=2
mfact=0.33


W.append(np.random.randn(2,dimalpha)*mfact)
#dim1 s(1), dim2 alpha(1)
for i in range(1,N-1):
  W.append(np.random.randn(2,dimalpha,dimalpha)*mfact)
  # dim1 : s(i) , dim2 alpha(i) ,dim3 alpha(i+1)
```

```
W.append(np.random.randn(2,dimalpha,diml)*mfact)
# dim1 : s(N) , dim2 alpha(N) ,dim3 l
W_init = W

##Algo pour A
t1=time.time()
Phi_tilde1_table=[0]*nbTraining
Phi_tilde2_table=[[0 for i in range(N-1)]
    for j in range(nbTraining)]
err=[] #tofix
r=0
maxiter=40
alpha=10**(-3)
alpha0=alpha
prec = 0.1
resultat_tab=[]
W = W_init.copy()
while r<maxiter:
    r=r+1
    epoch = N
    for e in range(epoch):
        gradW=0
        cout=0
        sel=e%N
        for n in range(nbTraining):
          ### Creation de phi_tilde ###

            img=training_data[n].reshape(-1,)

            A_tilde=W.copy() ; A=A_tilde.pop(sel)
            si=phi(img[sel]) ; Phi=phi(np.delete(img,sel))

            #Construction de phi_tilde1
            if(sel==1):
                Phi_tilde1_table[n]=contractMPS(
                        A_tilde[0:sel],Phi[:,:sel])
                Phi_tilde1=Phi_tilde1_table[n]

            if(sel!=0 and sel!=1):
                Phi_tilde1_table[n]=tl.tenalg.
                    mode_dot(A_tilde[sel-1]
                ,Phi_tilde1_table[n],1)
                Phi_tilde1_table[n]=tl.tenalg.
                    mode_dot(Phi_tilde1_table[n],
                                Phi[:,sel-1],0)
                Phi_tilde1=Phi_tilde1_table[n]

              #Construction de phi_tilde2
            if(sel==0):
                Phi_tilde2_table[n][N-2]=tl.tenalg.
                    mode_dot(A_tilde[N-2],
                                Phi[:,N-2],0)
                for i in range(N-3,-1,-1) :
                    Phi_tilde2_table[n][i]=tl.tenalg.
                    contract(A_tilde[i],
                    2,Phi_tilde2_table[n][i+1],0)
```

```
                    Phi_tilde2_table[n][i]=tl.tenalg.
                    mode_dot(
                Phi_tilde2_table[n][i],Phi[:,i],0)
                Phi_tilde2=Phi_tilde2_table[n][0]

            if(sel!=0  and sel!=(N-1)):

                Phi_tilde2=Phi_tilde2_table[n][sel]

            ### Calcul Gradient et Cout ###
            if(sel==0):
                Phi_tilde=np.multiply.outer(si,Phi_tilde2)
        #Phi_tilde2(alpha1,l) => Phi_tilde(s1,alpha1,l)
                fl=tl.tenalg.contract(A,(0,1),
                        Phi_tilde,(0,1))-training_y[n,:]
        # => vecteur de taille l
                gradW+=tl.tenalg.mode_dot(Phi_tilde,fl,2)
        # => gradW(s1,alpha1)
                cout+= sum([i**2 for i in fl])
            elif(sel==N-1):
                Phi_tilde=np.multiply.outer(si,Phi_tilde1)
        #Phi_tilde1(alpha(N-1)) => Phi_tilde(sN,alpha(N-1))
                fl=tl.tenalg.contract(A,(0,1),
                        Phi_tilde,(0,1))-training_y[n,:]
        #=> vecteur de taille l
                gradW+=np.multiply.outer(Phi_tilde,fl)
        # => gradW(sN,alpha(N-1),l)
                cout+= sum([i**2 for i in fl])
            else:
                Phi_tilde=np.multiply.outer(si,
                                    Phi_tilde1)
        #=> Phi_tilde1(alpha(i-1)) => Phi_tilde1(si,alpha(i-1))
                Phi_tilde=np.multiply.outer(Phi_tilde,
                                        Phi_tilde2)
        # Phi_tilde2(alpha(i),l) => Phi_tilde(si,alpha(i-1),alpha(i),l)
                fl=tl.tenalg.contract(A,(0,1,2),
                                Phi_tilde,(0,1,2))-training_y[n,:]
                # => vecteur de taille l
                gradW+=tl.tenalg.mode_dot(Phi_tilde,fl,3)
                # => gradW(si,alpha(i-1),alpha(i))
                cout+= sum([i**2 for i in fl])

        W[sel]=W[sel]-alpha*gradW
        err.append( (1/2)*cout )
    if r%1==0:
        print('epoch ', r*epoch, ' error ',
              err[len(err)-1], ' stepsize ', alpha)
        resultat_tab.append(resultat())
print(time.time()-t1)

##### Importation MNIST #####
(train_data, train_y), (X_test, Y_test) = mnist.load_data()

nbLabel=10 #10 classes differentes allant de 0 a 9
N=28*28 #nombre de pixels par image

```

```
#Separation dataset
nbTraining=len(train_data)
nbTest=len(Y_test)

#Copie donnee et transformation y
training_data=train_data.copy()
training_y=np.zeros((nbTraining,nbLabel))
for i in range(nbTraining):
    training_y[i,train_y[i]]=1
test_data=X_test.copy()
test_y=np.zeros((nbTest,nbLabel))
for i in range(nbTest):
    test_y[i,Y_test[i]]=1

#Recuperation que des labels 0 ou 1
label=[0,1]
nbLabel=len(label)
index=[]
for lab in label:
    for i in range(nbTraining):
        if training_y[i][lab]==1 : index.append(i)
index.sort()
nbTraining=len(index)
training_data=training_data[index]/256
training_y=training_y[index]
training_y=training_y[:,0:nbLabel]

#Recuperation que des labels 0 ou 1
label=[0,1]
nbLabel=len(label)
index=[]
for lab in label:
    for i in range(nbTest):
        if test_y[i][lab]==1 : index.append(i)
index.sort()

nbTest=len(index)
test_data=test_data[index]/256
test_y=test_y[index]
test_y=test_y[:,0:nbLabel]

i=0
plt.imshow(test_data[i].reshape(28,28),cmap="gray")
test_y[i]

#### Transformation image MNIST ####
def transform_img(img):
  new_img=np.zeros((14,14))
  for i in range(0,28,2):
    for j in range(0,28,2):
      new_img[int(i/2),int(j/2) ] = (img[i,j] +
              img[i+1,j] + img[i,j+1]
              + img[i+1,j+1])/4
  return new_img
```

Annexe C

Code Algorithmes DMRGs

```
# -*- coding: utf-8 -*-

##### Importation module #####
import numpy as np
import matplotlib.pyplot as plt
import tensorly as tl
import time
import math
from tensorflow.keras.datasets import mnist
from matplotlib import style

##### Fonction #####

#Fonction mapping
def phi(x):
    return np.array([np.cos((np.pi*x)/2),np.sin((np.pi*x)/2)])

#ContractionTenseur sous forme MPS
def contractMPS(W,Phi):
    res=tl.tenalg.mode_dot(W[0],Phi[:,0],0)
    if(len(W)>1):
        if(len(W[0].shape)==2):
            res=tl.tenalg.contract(W[1],1,res,0)
        else:
            res=tl.tenalg.contract(W[1],1,res,1)
        res=tl.tenalg.mode_dot(res,Phi[:,1],0)

    for i in range(2,len(W)):
      res=tl.tenalg.contract(W[i],1,res,0)
      res=tl.tenalg.mode_dot(res,Phi[:,i],0)
    return(res)

#Score
def resultat():
    nb=0
    for n in range(nbTraining):
      img=training_data[n].reshape(-1,)
      Phi=phi(img)
      z=contractMPS(W,Phi)
      categorie=np.argmax(training_y[n])
      if(categorie==np.argmax(z)):
        nb+=1
```

```
43      return nb/nbTraining
44
45  #Differentes fonction SVD
46  def SVD_B(sel,pos0,B,maxalpha,cutoff):
47      dim=B.shape
48      if(sel==0 or (sel==1 and sel>pos0) ):
49          B=B.reshape(dim[0],dim[1]*dim[2])
50          u, s, v = np.linalg.svd(B,full_matrices=False)
51          bond_dim=min(np.sum(s>cutoff),maxalpha)
52          s=s[0:bond_dim]
53          if(sel>pos0):
54              v = v[0:bond_dim,:]
55              u =  u[:,0:bond_dim] @ np.diag(s)
56          else:
57              v = np.diag(s) @ v[0:bond_dim,:]
58              u =  u[:,0:bond_dim]
59          v= v.reshape(bond_dim,2,dim[2]).transpose((1,0,2))
60      elif( (sel==posL-1 and sel< pos0) or
61           (sel==posL and sel> pos0) ):
62          B=B.reshape(dim[0]*dim[1],dim[2]*dim[3]*dim[4])
63          u, s, v = np.linalg.svd(B,full_matrices=False)
64          bond_dim=min(np.sum(s>cutoff),maxalpha)
65          s=s[0:bond_dim]
66          if(sel>pos0):
67              v = v[0:bond_dim,:]
68              u =  u[:,0:bond_dim] @ np.diag(s)
69          else:
70              v = np.diag(s) @ v[0:bond_dim,:]
71              u =  u[:,0:bond_dim]
72          u = u.reshape(2,dim[1],bond_dim)
73          v= v.reshape(bond_dim,2,dim[3],
74                       dim[4]).transpose((1,0,2,3))
75      elif( (sel==N-2 and sel< pos0 )or sel==N-1):
76          B=B.reshape(dim[0]*dim[1],dim[2])
77          u, s, v = np.linalg.svd(B,full_matrices=False)
78          bond_dim=min(np.sum(s>cutoff),maxalpha)
79          s=s[0:bond_dim]
80          if(sel>pos0):
81              v = v[0:bond_dim,:]
82              u =  u[:,0:bond_dim] @ np.diag(s)
83          else:
84              v = np.diag(s) @ v[0:bond_dim,:]
85              u =  u[:,0:bond_dim]
86          u = u.reshape(2,dim[1],bond_dim)
87          v= v.reshape(bond_dim,dim[2]).transpose((1,0))
88      elif( (sel==posL and sel< pos0) or
89           (sel==posL+1 and sel > pos0) ):
90          B=B.reshape(dim[0]*dim[1]*dim[2],dim[3]*dim[4])
91          u, s, v = np.linalg.svd(B,full_matrices=False)
92          bond_dim=min(np.sum(s>cutoff),maxalpha)
93          s=s[0:bond_dim]
94          if(sel>pos0):
95              v = v[0:bond_dim,:]
96              u =  u[:,0:bond_dim] @ np.diag(s)
97          else:
98              v = np.diag(s) @ v[0:bond_dim,:]
```

```
            u =  u[:,0:bond_dim]
        u = u.reshape(2,dim[1],
            dim[2],bond_dim).transpose((0,1,3,2))
        v= v.reshape(bond_dim,
                    2,dim[4]).transpose((1,0,2))
    else:
        B=B.reshape(dim[0]*dim[1],dim[2]*dim[3])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
        if(sel>pos0):
            v = v[0:bond_dim,:]
            u =  u[:,0:bond_dim] @ np.diag(s)
        else:
            v = np.diag(s) @ v[0:bond_dim,:]
            u =  u[:,0:bond_dim]
        u = u.reshape(2,dim[1],bond_dim)
        v= v.reshape(bond_dim,2,dim[3]).transpose((1,0,2))
    return (u,v)

def creation_B_Atilde(W,sel,pos0):

        A_tilde=W.copy() ; B_vec=[]
        B_vec.append(A_tilde.pop(sel))
        if(sel<pos0) : B_vec.append(A_tilde.pop(pos0-1))
        else : B_vec.append(A_tilde.pop(pos0))

        ###Construction de B
        if(sel<pos0):
            if(sel==0):
                B=tl.tenalg.contract(B_vec[0],1,B_vec[1],1)
            else:
                B=tl.tenalg.contract(B_vec[0],2,B_vec[1],1)

        if(sel>pos0):
            if(sel==1):
                B=tl.tenalg.contract(B_vec[1],1,B_vec[0],1)
            else:
                B=tl.tenalg.contract(B_vec[1],2,B_vec[0],1)
        return (B,A_tilde)

def creation_phi_tilde(A_tilde,Phi,sel,pos0,n):
    Max=max(sel,poss[0])
    Min=min(sel,poss[0])
    Phi_tilde1=0 ; Phi_tilde2=0;

    ##Construction de Phi_tilde1 vers la droite
    if(sel==1 and sel<pos0):
        Phi_tilde1_table_to_right[n]=
            contractMPS(A_tilde[0:Min],Phi[:,:Min])
        Phi_tilde1=Phi_tilde1_table_to_right[n]

    if(sel!=0 and sel!=1 and sel<pos0):
        Phi_tilde1_table_to_right[n]=
            tl.tenalg.contract(A_tilde[Min-1],1,
            Phi_tilde1_table_to_right[n],0)
```

```
155         Phi_tilde1_table_to_right[n]=tl.tenalg.mode_dot(
156                 Phi_tilde1_table_to_right[n],Phi[:,Min-1],0)
157         Phi_tilde1=Phi_tilde1_table_to_right[n]
158 
159     ##Construction de Phi_tilde1 vers la gauche
160     if(sel==(N-1)):
161         Phi_tilde1_table_to_left[n][0]=
162         tl.tenalg.mode_dot(A_tilde[0],Phi[:,0],0)
163         for i in range(1,N-2) :
164             Phi_tilde1_table_to_left[n][i]=
165             tl.tenalg.contract(A_tilde[i],
166                     1,Phi_tilde1_table_to_left[n][i-1],0)
167             Phi_tilde1_table_to_left[n][i]=tl.tenalg.mode_dot(
168                     Phi_tilde1_table_to_left[n][i],Phi[:,i],0)
169         Phi_tilde1=Phi_tilde1_table_to_left[n][N-3]
170 
171     if(sel>pos0 and  not(sel==1 and sel>pos0)):
172         Phi_tilde1=Phi_tilde1_table_to_left[n][sel-2]
173 
174     ##Construction de Phi_tilde2 vers la droite
175     if(sel==0):
176         Phi_tilde2_table_to_right[n][N-3]=
177         tl.tenalg.mode_dot(A_tilde[N-3],
178                                 Phi[:,N-3],0)
179         for i in range(N-4,-1,-1) :
180             Phi_tilde2_table_to_right[n][i]=
181             tl.tenalg.contract(A_tilde[i],
182                         2,Phi_tilde2_table_to_right[n][i+1],0)
183             Phi_tilde2_table_to_right[n][i]=tl.tenalg.mode_dot(
184                     Phi_tilde2_table_to_right[n][i],Phi[:,i],0)
185         Phi_tilde2=Phi_tilde2_table_to_right[n][0]
186 
187     if(sel!=0 and sel!=(N-2) and sel<pos0):
188         Phi_tilde2=Phi_tilde2_table_to_right[n][sel]
189 
190     ##Construction de Phi_tilde2 vers la gauche
191     if(sel==N-2 and sel>pos0):
192         Phi_tilde2_table_to_left[n]=
193         tl.tenalg.mode_dot(A_tilde[N-3],
194                                 Phi[:,N-3],0)
195         Phi_tilde2=Phi_tilde2_table_to_left[n]
196 
197     if(sel != N-1 and sel!=N-2 and sel>pos0):
198         Phi_tilde2_table_to_left[n]=
199         tl.tenalg.contract(A_tilde[sel-1],2,
200                     Phi_tilde2_table_to_left[n],0)
201         Phi_tilde2_table_to_left[n]=
202         tl.tenalg.mode_dot(
203                 Phi_tilde2_table_to_left[n],Phi[:,sel-1],0)
204         Phi_tilde2=Phi_tilde2_table_to_left[n]
205 
206     return (Phi_tilde1,Phi_tilde2)
207 
208 def calcul_cout_gradient(B,Phi_tilde1,Phi_tilde2,
209                          si,y,sel,pos0):
210     ##Calcul Cout
```

```
    if(sel==0 or (sel==1 and sel>pos0) ):
        Phi_tilde=np.multiply.outer(si[:,1],Phi_tilde2)
        #Phi_tilde2(alpha2,l) => Phi_tilde(s2,alpha2,l)
        Phi_tilde=np.multiply.outer(si[:,0],Phi_tilde)
        #=> Phi_tilde(s1,s2,alpha2,l)
        fl=tl.tenalg.contract(B,(0,1,2),
                              Phi_tilde,(0,1,2))-y
        gradB=tl.tenalg.mode_dot(Phi_tilde,fl,3)
        #=> gradB(s1,s2,alpha2)
    elif( (sel==posL-1 and sel< pos0) or
          (sel==posL and sel> pos0) ):
        Phi_tilde=np.multiply.outer(si[:,0],Phi_tilde1)
        #=> Phi_tilde1(alpha(i-1)) => Phi_tilde1(si,alpha(i-1))
        Phi_tilde=np.multiply.outer(Phi_tilde,si[:,1])
        #=> Phi_tilde(si,alpha(i-1),s(i+1))
        Phi_tilde=np.multiply.outer(Phi_tilde,Phi_tilde2)
        #=>Phi_tilde(si,alpha(i-1),s(i+1),alpha(i+1))
        fl=tl.tenalg.contract(B,(0,1,2,3),
                              Phi_tilde,(0,1,2,3))-y
        gradB=np.multiply.outer(Phi_tilde,fl)
        #gradB(si,alpha(i-1),s(i+1),alpha(i+1),l)
    elif( (sel==posL and sel< pos0) or
          (sel==posL+1 and sel > pos0)  ):
        Phi_tilde=np.multiply.outer(si[:,0],Phi_tilde1)
        #=> Phi_tilde1(alpha(i-1)) => Phi_tilde1(si,alpha(i-1))
        Phi_tilde=np.multiply.outer(Phi_tilde,si[:,1])
        #=> Phi_tilde(si,alpha(i-1),s(i+1))
        Phi_tilde=np.multiply.outer(Phi_tilde,Phi_tilde2)
        #=>Phi_tilde(si,alpha(i-1),s(i+1),alpha(i+1))
        fl=tl.tenalg.contract(B,(0,1,3,4),
                              Phi_tilde,(0,1,2,3))-y
        Phi_tilde=np.multiply.outer(Phi_tilde,fl)
        #=>Phi_tilde(si,alpha(i-1),s(i+1),alpha(i+1),l)
        Phi_tilde=Phi_tilde.transpose((0,1,4,2,3))
        #=>Phi_tilde(si,alpha(i-1),l,s(i+1),alpha(i+1))
        gradB=Phi_tilde
    elif( (sel==N-2 and sel<pos0 ) or sel==N-1):
        Phi_tilde=np.multiply.outer(si[:,0],Phi_tilde1)
        #Phi_tilde1(alpha(N-2),l) =>Phi_tilde1(s(N-1),alpha(N-2),l)
        Phi_tilde=np.multiply.outer(Phi_tilde,si[:,1])
        #=> Phi_tilde(s(N-1),alpha(N-2),l,sN)
        fl=tl.tenalg.contract(B,(0,1,2),
                              Phi_tilde,(0,1,3))-y
        gradB=tl.tenalg.mode_dot(Phi_tilde,fl,2)
        #=> gradB(s(N-1),alpha(N-2),sN)
    else:
        Phi_tilde=np.multiply.outer(si[:,0],Phi_tilde1)
        #=> Phi_tilde1(alpha(i-1)) => Phi_tilde1(si,alpha(i-1))
        Phi_tilde=np.multiply.outer(Phi_tilde,si[:,1])
        #=> Phi_tilde(si,alpha(i-1),s(i+1))
        Phi_tilde=np.multiply.outer(Phi_tilde,Phi_tilde2)
        #=>Phi_tilde(si,alpha(i-1),s(i+1),alpha(i+1),l)
        if(sel>posL):
            fl=tl.tenalg.contract(B,(0,1,2,3)
            ,Phi_tilde,(0,1,3,4))-y
            gradB=tl.tenalg.mode_dot(Phi_tilde,fl,2)
```

```
267                 #gradB(si,alpha(i-1),s(i+1),alpha(i+1))
268             else:
269                 fl=tl.tenalg.contract(B,(0,1,2,3),
270                                       Phi_tilde,(0,1,2,3))-y
271                 gradB=tl.tenalg.mode_dot(Phi_tilde,fl,4)
272                 #gradB(si,alpha(i-1),s(i+1),alpha(i+1))
273         cout=sum([i**2 for i in fl])
274 
275         return (cout,gradB)
276 
277 def SVD_B2(sel,pos0,B,maxalpha,cutoff):
278         dim=B.shape
279         if(sel==0 or (sel==1 and sel>pos0) ):
280             B=B.reshape(dim[0],dim[1]*dim[2])
281             u, s, v = np.linalg.svd(B,full_matrices=False)
282             bond_dim=min(np.sum(s>cutoff),maxalpha)
283             s=s[0:bond_dim]
284             if(sel<pos0):
285                 v = v[0:bond_dim,:]
286                 u =  u[:,0:bond_dim] @ np.diag(s)
287             else:
288                 v = np.diag(s) @ v[0:bond_dim,:]
289                 u =  u[:,0:bond_dim]
290             v= v.reshape(bond_dim,2,
291                          dim[2]).transpose((1,0,2))
292         elif( (sel==posL-1 and sel< pos0)
293         or (sel==posL and sel> pos0) ):
294             B=B.reshape(dim[0]*dim[1],dim[2]*dim[3]*dim[4])
295             u, s, v = np.linalg.svd(B,full_matrices=False)
296             bond_dim=min(np.sum(s>cutoff),maxalpha)
297             s=s[0:bond_dim]
298             if(sel<pos0):
299                 v = v[0:bond_dim,:]
300                 u =  u[:,0:bond_dim] @ np.diag(s)
301             else:
302                 v = np.diag(s) @ v[0:bond_dim,:]
303                 u =  u[:,0:bond_dim]
304             u = u.reshape(2,dim[1],bond_dim)
305             v= v.reshape(bond_dim,2,dim[3]
306             ,dim[4]).transpose((1,0,2,3))
307         elif( (sel==N-2 and sel< pos0 )or sel==N-1):
308             B=B.reshape(dim[0]*dim[1],dim[2])
309             u, s, v = np.linalg.svd(B,
310                                     full_matrices=False)
311             bond_dim=min(np.sum(s>cutoff),maxalpha)
312             s=s[0:bond_dim]
313             if(sel<pos0):
314                 v = v[0:bond_dim,:]
315                 u =  u[:,0:bond_dim] @ np.diag(s)
316             else:
317                 v = np.diag(s) @ v[0:bond_dim,:]
318                 u =  u[:,0:bond_dim]
319             u = u.reshape(2,dim[1],bond_dim)
320             v= v.reshape(bond_dim,dim[2]).transpose((1,0))
321         elif( (sel==posL and sel< pos0)
322         or (sel==posL+1 and sel > pos0) ):
```

```
        B=B.reshape(dim[0]*dim[1]*dim[2],dim[3]*dim[4])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
        if(sel<pos0):
            v = v[0:bond_dim,:]
            u =  u[:,0:bond_dim] @ np.diag(s)
        else:
            v = np.diag(s) @ v[0:bond_dim,:]
            u =  u[:,0:bond_dim]
        u = u.reshape(2,dim[1],dim[2],
                      bond_dim).transpose((0,1,3,2))
        v= v.reshape(bond_dim,2,
                     dim[4]).transpose((1,0,2))
    else:
        B=B.reshape(dim[0]*dim[1],dim[2]*dim[3])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
        if(sel<pos0):
            v = v[0:bond_dim,:]
            u =  u[:,0:bond_dim] @ np.diag(s)
        else:
            v = np.diag(s) @ v[0:bond_dim,:]
            u =  u[:,0:bond_dim]
        u = u.reshape(2,dim[1],bond_dim)
        v= v.reshape(bond_dim,2,dim[3]).
        transpose((1,0,2))
    return (u,v)

def SVD_B3(sel,pos0,B,maxalpha,cutoff):
    dim=B.shape
    if(sel==0 or (sel==1 and sel>pos0) ):
        B=B.reshape(dim[0],dim[1]*dim[2])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
        v = np.sqrt(np.diag(s)) @ v[0:bond_dim,:]
        u =  u[:,0:bond_dim] @ np.sqrt(np.diag(s))
        v= v.reshape(bond_dim,2,dim[2]).transpose((1,0,2))
    elif( (sel==posL-1 and sel< pos0)
    or (sel==posL and sel> pos0) ):
        B=B.reshape(dim[0]*dim[1],dim[2]*dim[3]*dim[4])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
        v = np.sqrt(np.diag(s)) @ v[0:bond_dim,:]
        u =  u[:,0:bond_dim] @ np.sqrt(np.diag(s))
        u = u.reshape(2,dim[1],bond_dim)
        v= v.reshape(bond_dim,2,
                     dim[3],dim[4]).transpose((1,0,2,3))
    elif( (sel==N-2 and sel< pos0 )or sel==N-1):
        B=B.reshape(dim[0]*dim[1],dim[2])
        u, s, v = np.linalg.svd(B,full_matrices=False)
        bond_dim=min(np.sum(s>cutoff),maxalpha)
        s=s[0:bond_dim]
```

```
379             v = np.sqrt(np.diag(s)) @ v[0:bond_dim,:]
380             u =  u[:,0:bond_dim] @ np.sqrt(np.diag(s))
381             u = u.reshape(2,dim[1],bond_dim)
382             v= v.reshape(bond_dim,dim[2]).transpose((1,0))
383         elif( (sel==posL and sel< pos0)
384         or (sel==posL+1 and sel > pos0) ):
385             B=B.reshape(dim[0]*dim[1]*dim[2],dim[3]*dim[4])
386             u, s, v = np.linalg.svd(B,full_matrices=False)
387             bond_dim=min(np.sum(s>cutoff),maxalpha)
388             s=s[0:bond_dim]
389             v = np.sqrt(np.diag(s)) @ v[0:bond_dim,:]
390             u =  u[:,0:bond_dim] @ np.sqrt(np.diag(s))
391             u = u.reshape(2,dim[1],dim[2],
392                           bond_dim).transpose((0,1,3,2))
393             v= v.reshape(bond_dim,2,dim[4]).transpose((1,0,2))
394         else:
395             B=B.reshape(dim[0]*dim[1],dim[2]*dim[3])
396             u, s, v = np.linalg.svd(B,full_matrices=False)
397             bond_dim=min(np.sum(s>cutoff),maxalpha)
398             s=s[0:bond_dim]
399             v = np.sqrt(np.diag(s)) @ v[0:bond_dim,:]
400             u =  u[:,0:bond_dim] @ np.sqrt(np.diag(s))
401             u = u.reshape(2,dim[1],bond_dim)
402             v= v.reshape(bond_dim,2,dim[3]).transpose((1,0,2))
403         return (u,v)
404
405 ##### Creation Dataset #####
406
407 nbTraining=32 #Nombre de training example
408 N=100 #Nombre de pixel par image
409
410 #Creation des images
411 training_data=np.random.random((nbTraining,N))
412
413 #Creation des labels
414 training_y=np.zeros((nbTraining,2)) ; training_y[0:16,0] = 1
415  training_y[16:nbTraining,1]=1
416
417 ##Construction des A et de W
418 W=[] #dim0 : A
419 dimalpha=1
420 diml=2
421 mfact=0.950 # a adapter en fonction de la premiere valeur de f pour
422 #qu'elle soit dans un intervalle proche de 0
423
424 posL=math.floor(N/2) #Le tenseur central aura la dimension de l
425 W.append(np.ones((2,dimalpha))*mfact) #dim1 s(1), dim2 alpha(1)
426 for i in range(1,N-1):
427     if(i==posL):
428         W.append(np.ones((2,dimalpha,dimalpha,diml))*mfact)
429         # dim1 : s(i) , dim2 alpha(i) ,dim3 alpha(i+1) ,dim4 l
430     else:
431         W.append(np.ones((2,dimalpha,dimalpha))*mfact)
432         # dim1 : s(i) , dim2 alpha(i) ,dim3 alpha(i+1)
433 W.append(np.ones((2,dimalpha))*mfact)
434 # dim1 : s(N) , dim2 alpha(N)
```

```
W_init = W

##### Algo DMRG ######

# DRMG-MSENS #

Phi_tilde1_table_to_right=[0]*nbTraining
Phi_tilde1_table_to_left=[[0 for i in
            range(N-2)] for j in range(nbTraining)]
Phi_tilde2_table_to_right=[[0 for i in
            range(N-2)] for j in range(nbTraining)]
Phi_tilde2_table_to_left=[0]*nbTraining
cutoff=10*(-10) #seuil valeur propre
maxalpha=10 #bond dimension maximale
err=[] #tofix
epoch=0
maxcycle=20
alpha=10**(-1)
prec = 0.1
W = W_init.copy()
cycle=0
while cycle < maxcycle:
    epoch=epoch+2*(N-1)
    cycle+=1

    poss=[i for i in range(0,N)]+[i
          for i in range(0,N-1)][::-1]

    for e in range(2*(N-1)):
        gradB=0
        cout=0
        sel=poss.pop(0)
        Max=max(sel,poss[0])
        Min=min(sel,poss[0])

        #Construction de B et A_tilde
        (B,A_tilde)=creation_B_Atilde(W,sel,poss[0])

        for n in range(nbTest):
          ### Creation de phi_tilde ###

            img=test_data[n].reshape(-1,)

            si=phi(img[[Min,Max]])
            Phi=phi(np.delete(img,(sel,poss[0])))

            (Phi_tilde1,Phi_tilde2) =
            creation_phi_tilde(A_tilde,
            Phi,sel,poss[0],n)


            ##Calcul Cout
            (cout_ite,grad_ite)=calcul_cout_gradient(B,
            Phi_tilde1,Phi_tilde2,si,test_y[n,:],
            sel,poss[0])
            cout+=cout_ite ; gradB+=grad_ite
```

```
491
492         err.append( (1/2)*cout )
493         if(cycle!=1):
494           B=B-alpha*gradB
495         #SVD
496         (W[Min],W[Max])=SVD_B(sel,poss[0],B,
497         maxalpha,cutoff)
498
499     print('cycle ', cycle , ' error ',
500           err[len(err)-1], ' stepsize ', alpha)
501
502 # DMRG-SENSI #
503     ##Optimisation du gradient avec B
504 Phi_tilde1_table_to_right=[0]*nbTraining
505 Phi_tilde1_table_to_left=[[0 for i in range(N-2)]
506     for j in range(nbTraining)]
507 Phi_tilde2_table_to_right=[[0 for i in range(N-2)]
508     for j in range(nbTraining)]
509 Phi_tilde2_table_to_left=[0]*nbTraining
510 Wmemory=[]
511 cutoff=10*(-10) #seuil valeur propre
512 maxalpha=10 #bond dimension maximale
513 err=[] #tofix
514 epoch=0
515 maxcycle=20
516 alpha=10**(-10)
517 prec = 0.1
518 W = W_init.copy()
519 cycle=0
520 while cycle < maxcycle:
521     epoch=epoch+2*(N-1)
522     cycle+=1
523
524     poss=[i for i in range(0,N)]+
525     [i for i in range(0,N-1)][::-1]
526
527     for e in range(2*(N-1)):
528         gradB=0
529         cout=0
530         sel=poss.pop(0)
531         Max=max(sel,poss[0])
532         Min=min(sel,poss[0])
533
534         #Construction de B et A_tilde
535         (B,A_tilde)=creation_
536         B_Atilde(W,sel,poss[0])
537
538         for n in range(nbTraining):
539           ### Creation de phi_tilde ###
540
541             img=training_data[n].reshape(-1,)
542
543             si=phi(img[[Min,Max]])
544             Phi=phi(np.delete(img,(sel,poss[0])))
545
546             (Phi_tilde1,Phi_tilde2) =
```

```
547                 creation_phi_tilde(A_tilde,
548                 Phi,sel,poss[0],n)
549
550
551                 ##Calcul Cout
552                 (cout_ite,grad_ite)=
553                 calcul_cout_gradient(B,Phi_tilde1,
554                 Phi_tilde2,si,training_
555                 y[n,:],sel,poss[0])
556                 cout+=cout_ite ;
557                 gradB+=grad_ite
558
559             err.append( (1/2)*cout )
560             B=B-alpha*gradB
561             #SVD
562             (W[Min],W[Max])=SVD_B2(sel,poss[0],
563             B,maxalpha,cutoff)
564
565         print('cycle␣', cycle , '␣error␣',
566               err[len(err)-1], '␣stepsize␣', alpha)
567
568 # DMRG-EQUI #
569 ##Optimisation du gradient avec B
570 Phi_tilde1_table_to_right=[0]*nbTraining
571 Phi_tilde1_table_to_left=[[0 for i in range(N-2)]
572     for j in range(nbTraining)]
573 Phi_tilde2_table_to_right=[[0 for i in range(N-2)]
574     for j in range(nbTraining)]
575 Phi_tilde2_table_to_left=[0]*nbTraining
576 Wmemory=[]
577 cutoff=10*(-10) #seuil valeur propre
578 maxalpha=10 #bond dimension maximale
579 err=[] #tofix
580 epoch=0
581 maxcycle=20
582 alpha=10**(-2)
583 prec = 0.1
584 W = W_init.copy()
585 cycle=0
586 while cycle < maxcycle:
587     epoch=epoch+2*(N-1)
588     cycle+=1
589
590     poss=[i for i in range(0,N)]+
591     [i for i in range(0,N-1)][::-1]
592
593     for e in range(2*(N-1)):
594         gradB=0
595         cout=0
596         sel=poss.pop(0)
597         Max=max(sel,poss[0])
598         Min=min(sel,poss[0])
599
600         #Construction de B et A_tilde
601         (B,A_tilde)=creation_B_
602         Atilde(W,sel,poss[0])
```

```
603
604         for n in range(nbTraining):
605           ### Creation de phi_tilde ###
606
607             img=training_data[n].
608             reshape(-1,)
609
610             si=phi(img[[Min,Max]])
611             Phi=phi(np.delete(
612                     img,(sel,poss[0])))
613
614             (Phi_tilde1,Phi_tilde2) =
615             creation_phi_tilde(
616                     A_tilde,
617             Phi,sel,poss[0],n)
618
619
620             ##Calcul Cout
621             (cout_ite,grad_ite)=calcul_cout_gradient(B,
622             Phi_tilde1,
623             Phi_tilde2,si,training_y[n,:],sel,poss[0])
624             cout+=cout_ite ; gradB+=grad_ite
625
626         err.append( (1/2)*cout )
627         B=B-alpha*gradB
628         #SVD
629         (W[Min],W[Max])=SVD_B3(sel,poss[0],B,maxalpha,cutoff)
630
631     print('cycle ', cycle , ' error ',
632           err[len(err)-1], ' stepsize ', alpha)
```

Annexe D

Synthèse des échanges avec des chercheurs

Dans le cadre du projet recherche, j'ai rencontré trois chercheurs dans le but d'approfondir mes connaissances sur les différents métiers de la recherche.

Tout d'abord, j'ai rencontré Clément Elvira, maître de conférence au sein de l'école Supélec travaillant dans le traitement du signal. Grâce à cette rencontre, j'ai pu en apprendre davantage sur le métier d'enseignant-chercheur. Effectivement, Clément exerce ses fonctions depuis le début de l'année, il avait donc une vision récente des étapes à franchir afin de devenir enseignant-chercheur.

Ensuite, j'ai discuté avec Axel Marmoret, doctorant à l'INRIA dans l'équipe Panama sur l'analyse de la musique et du son. Cette rencontre m'a permis d'en savoir plus sur le déroulement d'une thèse.

Enfin, j'ai contacté Moez Baccouche, chercheur IA chez Orange Data-IA dans le traitement audio. Moez a pu me donner une vision différente de la recherche dans le secteur du privée, ainsi qu'une vision plus générale de la recherche car il travaille dans la recherche depuis 11 ans.

Ci-après une synthèse de ces trois entretiens :

Les trois chercheurs que j'ai rencontrés m'ont conseillé de lire des articles disponibles en libre accès sur google scholar, HAAL et arXiv. Ces lectures permettent de rester informés des évolutions scientifiques. Il est important dans le métier d'enseignant chercheur d'être attentif à toutes ces évolutions. Les livres permettent également d'approfondir ses connaissances sur des sujets plus précis. Enfin, Moez Baccouche m'a conseillé de suivre sur Twitter des chercheurs en IA les plus connus. Ils publient régulièrement sur l'avancée de leurs recherches.

De plus, ces entretiens m'ont permis d'accroître mes connaissances sur les différences entre la recherche académique et la recherche dans le secteur privé. Selon eux, la recherche dans le privé est plus applicative, autrement dit, elle est moins théorique que la recherche dans le domaine public. Cependant, dans le secteur privé, on doit justifier que ce que l'ont produit va apporter une plus-value à l'entreprise, tandis que dans le domaine public, les chercheurs ont plus de liberté et contribuent davantage aux avancées scientifiques.

Pour travailler dans le domaine de la recherche, ils m'ont tous les trois fais comprendre qu'il était recommandé d'effectuer une thèse. Il peut être intéressant d'effectuer un stage dans le privé et un doctorat dans le public ou inversement, afin de pouvoir se faire une idée des différences entre les deux secteurs. En revanche, il est nécessaire d'être bien encadré lorsqu'on effectue une thèse cifre.

Ensuite, pour le métier d'enseignant chercheur, il faut trouver le juste équilibre entre les cours, les tâches administratives et la recherche afin de trouver un rythme, qui est propre à chacun. Ils me conseillent aussi d'écrire beaucoup d'articles et de participer à des conférences au début afin d'être visible et d'obtenir un poste plus facilement.

Enfin, le parcours scolaire n'a pas beaucoup d'importance. Tous les trois avaient des parcours totalement différents.