Université de Versailles
Saint-Quentin-en-Yvelines
Institut des Sciences et Techniques des Yvelines

Rapport de projet

Développement d'une bibliothèque de machine learning

Année scolaire 2020-2021

Etudiant :
Aurélien Delval
IATIC5
aurelien.delval@uvsq.fr

Professeur encadrant :
Pablo de Oliveira Castro
pablo.oliveira@uvsq.fr

17 février 2021

Résumé

L'objectif de ce projet est le développement (puis la mise en pratique) d'une bibliothèque de machine learning se concentrant plus particulièrement sur l'implémentation des réseaux de neurones. Le présent rapport se propose donc dans un premier temps de poser les bases théoriques de ces deux concepts, puis d'apporter au fur et à mesure de nouvelles notions et optimisations qui permettront dans la pratique d'obtenir de meilleurs résultats avec notre bibliothèque. Au fur et à mesure que de nouveaux concepts seront introduits, on proposera donc une explication du code correspondant, afin que ce rapport serve de guide pour comprendre le fonctionnement non seulement de notre bibliothèque, mais aussi de toutes celles qui sont apparues depuis le développement du machine learning ces dernières années. On proposera aussi des exemples d'applications pratiques, qui valideront notre implémentation et donneront un aperçu des nombreuses possibilités offertes par les réseaux neuronaux aujourd'hui. L'ambition de ce projet n'est cependant pas tant de résoudre des problèmes complexes que d'exposer la théorie derrière le machine learning et les réseaux de neurones, puis de voir comment il est possible de mettre en place ces concepts pour une utilisation pratique (et bien que les exemples d'application en constituent une partie importante). On préfèrera donc discuter longuement de la théorie pour adopter l'approche la plus efficace et générale plutôt que de passer directement à une implémentation naïve et plus rapide.

Enfin, concernant le rendu technique qui fait l'objet de ce rapport, le choix du langage s'est orienté vers le C++, et le code source a été entièrement rédigé et commenté en anglais, par souci de cohérence avec la syntaxe du langage. Il peut être consulté au lien suivant :

`https://github.com/PurplePachyderm/tensorslow`

Table des matières

Chapitre 1

Introduction théorique

1.1 Introduction au machine learning

1.1.1 Définition

Le *machine learning* (ou en français *apprentissage automatique*) est une branche de l'intelligence artificielle dédiée aux algorithmes capables de s'améliorer eux-mêmes avec l'expérience. Cela permet d'aboutir à des systèmes capables d'accomplir des tâches pour lesquelles ils n'ont pas été explicitement programmés. Cette définition reste cependant assez vaste et englobe ainsi de nombreuses techniques, dont seulement une infime partie sera traitée dans ce rapport. [1]

1.1.2 L'inférence statistique

Le procédé général utilisé par les algorithmes de machine learning porte le nom d'*inférence statistique*. Il s'agit d'un ensemble de méthodes permettant de procéder par induction pour prédire les caractéristiques d'une population à partir d'un échantillon de celle-ci. Si de nombreuses méthodes de plus en plus complexes ont été développées au fur et à mesure des années, un des premiers exemples connus est propablement celui de la régression linéaire, une méthode développée par Francis Galton dès le début du XIXème siècle. Cette méthode consiste à prendre une série de points, et à déterminer les coefficients de l'équation de la droite passant le plus près de tous ces points en moyenne. Plus précisément, si on a une droite d'équation $y = ax + b$ et une série de n points $\{x_i, y_i\}_{i=1}^{n}$,on cherchera à miniser la quantité suivante, que l'on pourrait appeler le *coût du modèle* :

$$c = \sum_{i=1}^{n} (a.x_i + b - y_i)^2 \tag{1.1}$$

Graphiquement, la corrélation entre une variable X et une variable Y peut alors être représentée par la droite suivante :

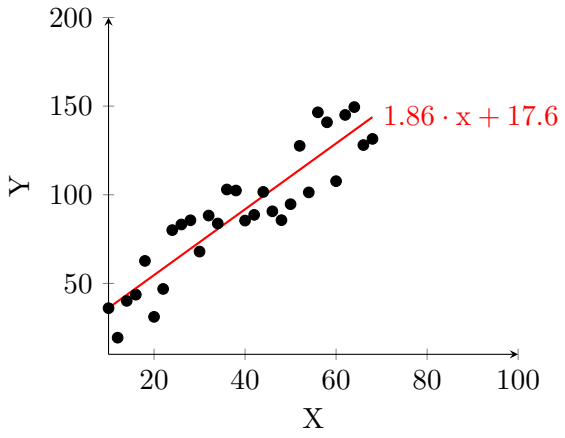

FIGURE 1.1 – Régression linéaire d'une série de données

Il s'agit probablement ici de l'exemple le plus simple de *régression*, car l'équation de la droite étant de la forme $y = ax+b$, on ne peut agir que sur les paramètres a et b (par exemple avec la méthode des moindres carrés). Cependant, l'ensemble des problématiques de machine learning peut être ramené à ce type de problème : ajuster les paramètres d'un modèle mathématique afin de formuler les meilleures prédictions possibles par la suite. [2, p. 9-10]

1.1.3 Les différentes approches du machine learning

Comme nous venons de l'évoquer (1.1.1), le machine learning est un domaine de recherche très vaste. Ainsi, il comprend de nombreuses approches algorithmiques, lesquelles se différencieront principalement par leurs domaines d'applications ainsi que les données sur lesquelles elles travaillent en entrée et en sortie. Globalement, les algorithmes de machine learning fonctionnent toujours en deux phases : une phase d'apprentissage, pendant laquelle l'algorithme affine son modèle, et une phase de prédiction où on se contente d'utiliser notre modèle pour calculer une sortie.

Nous avons donc évoqué la notion de modèle mathématique, il en existe plusieurs types en machine learning dont voici quelques-uns des plus importants (cette liste est très loin d'être exhaustive) :

— **la régression analytique** : nous l'avons déjà abordé à travers l'exemple de la régression linéaire (1.1.2). Si la régression linéaire est la méthode la plus simple et la plus courante, il en existe d'autres où on essaiera d'établir des relations plus complexes entre différentes variables (régression polynomiale, etc...).
— **les arbres de décision** : il s'agit d'une structure en arbre dont les décisions possibles sont modélisées par les noeuds, et dont les branches sont des combinaisons de variables d'entrée décrivant comment aboutir à ces décisions. L'arbre est alors construit pendant la phase d'apprentissage. Il existe plusieurs méthodes pour cela, mais l'idée générale est de partitionner successivement les données selon la valeur d'une variable d'entrée bien précise, jusqu'à les avoir suffisament réduites pour prendre une décision (c'est à dire quand le parcours de l'arbre est terminé).
— **les réseaux de neurones** : il s'agit de l'objet de ce projet, et sûrement d'un des domaines du machine learning pour lequel la recherche est la plus active. Il en existe de nombreux types que nous ne pourrons pas tous couvrir, mais l'on commencera à détailler leur fonctionnement dans la partie 1.2, et tout au long de ce rapport.

Enfin, il est aussi intéressant de noter que la phase d'aprentissage peut se faire de différentes façons. Principalement, on distingue trois types d'apprentissage (ce ne sont cependant pas les seuls) :

— **l'apprentissage supervisé** : cette méthode nécessite de fournir à la fois des données d'entrée mais aussi la sortie désirée pour ces données. L'algorithme sera alors capable de constater l'écart entre sa prédiction et la réalité, et d'ajuster son modèle en conséquence.
— **l'apprentissage non-supervisé** : ici, on fournit seulement un jeu de données à l'algorithme, sans la sortie désirée. Ce dernier sera alors capable de découvrir lui-même les différentes catégories dans lesquelles se rangeront les données, en identifiant les similarités qui peuvent exister entre les différentes entrées.
— **l'apprentissage par renforcement** : il s'agit ici de jouer sur le comportement d'un *software agent* (ou agent logiciel) dans son environnement. En fonction des différents actions que le programme va

exécuter au sein de cet environnement, il sera soit récompensé, soit pénalisé, et adaptera son modèle en fonction du feedback qui lui est donné.

Remarque : Les réseaux de neurones, qui feront l'objet de la prochaine section 1.2, sont capables de fonctionner avec ces trois modes d'apprentissage. Cependant, on ne se concentrera dans ce rapport que sur des exemples d'apprentissage supervisé.

1.1.4 Quelques exemples d'applications

Le machine learning et particulièrement les réseaux de neurones sont des domaines de recherche ayant connu une explosion ces dernières années, entre autres grâce à la puissance des machines récentes. Leurs applications sont ainsi devenues nombreuses. Si il serait long et laborieux de toutes les citer, en voici quelques-unes particulièrement intéressantes, dont certaines nous intéresseront dans la suite de ce projet :

- **vision par ordinateur** : les algorithmes de machine learning sont de plus en plus performants dans ce domaine. Ils sont notamment capables d'effectuer de la classification d'image ou de l'extraction de features avec une grande efficacité. Un problème classique est de résoudre la base de données MNIST [3], qui regroupe des milliers d'images de chiffres manuscrits, en les classifiant correctement. C'est un problème plutôt simple à résoudre, et il s'agira donc de la première application que l'on fera de notre bibliothèque. Plus généralement, c'est sur ce genre de problématiques que l'on fera le choix d'orienter notre travail et de tester notre bibliothèque.
- **traitement automatique des langues** : ou *natural language processing* (NLP). Il s'agit de permettre à des algorithmes de traiter et d'analyser le langage naturel. Encore une fois ce sont les algorithmes fonctionnant sur le principe du machine learning qui sont de loin les plus performants dans ce domaine. On peut par exempler penser au modèle GPT-3 [4] développé par OpenAI et sorti en juin 2020, qui est à ce jour un des plus avancés. Les applications du NLP sont diverses, et englobent par exemple la génération de texte ou la traduction.
- **intelligence artificielle pour les jeux** : il existe de nombreux jeux dans lesquels les algorithmes de machine learning peuvent être très efficaces. On pense notamment aux algorithmes AlphaZero et AlphaGo développés par Google DeepMind, respectivement pour les

jeux d'échecs et de Go. Ces algorithmes utilisent souvent des arbres de Monte Carlo pour évaluer différents coups, et sont capables de viser directement les plus intéressants grâce à l'utilisation de réseaux profonds (cette notion sera définie plus tard).

1.2 Les réseaux de neurones

La partie suivante consistera en une introduction théorique aux réseaux de neurones. Elle posera les bases suffisantes pour se faire une idée globale de leur fonctionnement, et commencer leur implémentation plus tard dans le projet [5].

Remarque : Il important de noter qu'il existe de nombreuses structures de réseaux de neurones, mais que nous ne décrirons pour l'instant que l'une des plus simples, celle du perceptron multicouche (ou en anglais *multilayer perceptron*, noté MLP). Bien qu'assez simple à comprendre, ce type de réseau permet de résoudre des problèmes réalistes, et sera par la suite une base pour définir des structures plus complexes.

1.2.1 Concept du neurone

Les neurones sont les objets élémentaires sur lesquels nous allons travailler, et représentent le coeur de la théorie qui va être développée dans cette partie. Insipirés des neurones biologiques, ils peuvent être vus comme une fonction mathématique simple ayant plusieurs entrées et une unique sortie réelle appartenant la plupart du temps à l'intervalle $[0, 1]$, que l'on appelle souvent l'activation. Cependant, il ne s'agit pas de la manière la plus intuitive d'introduire ce concept. Considérons donc dans un premier temps le schéma suivant, rappellant la structure d'un neurone biologique et de ses connexions :

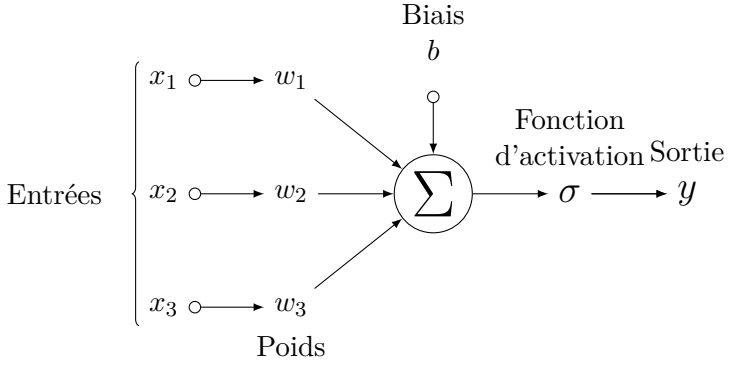

FIGURE 1.2 – Schéma d'un neurone artificiel

Les paramètres de cette fonction sont donc : n entrées $\{x_1, x_2, ..., x_n\}$ auxquelles on associe n poids $\{w_1, w_2, ..., w_n\}$ qui correspondent à des coefficients que l'on affecte aux entrées (sur le schéma $n = 3$). Après avoir calculé la somme des entrées coefficientées, on ajoute aussi le biais b. Rappellons que la sortie d'un neurone est idéalement dans l'intervalle $[0, 1]$, et rien ne nous garantit que la valeur de cette somme est bien comprise dedans (elle peut tout aussi bien être supérieure à 1 ou négative). C'est pour cela que l'on associe aussi à notre neurone une *fonction d'activation*, que l'on notera f_{act} et qui devra répondre à la condition suivante :

$$f_{act} : \mathbb{R} \to [0, 1] \tag{1.2}$$

Il existe plusieurs fonctions de ce type, et parmi elles la fonction sigmoïde qui a été traditionnellement beaucoup utilisé à cet effet. On la définit comme suit :

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{1.3}$$

Remarque : Aujourd'hui, il est commun d'utiliser des fonctions tendant vers l'infini car elles sont plus rapides à calculer et permettent d'obtenir de meilleurs résultats. On parle alors de *fonction de rectification* (ou *rectifier*). Cela nécessite parfois une étape de rescaling : après avoir calculé les valeurs de sorties de plusieurs couches (voir partie 1.2.2), on utilise une fonction d'activation telle que σ afin de recadrer toutes les valeurs dans l'intervalle désiré. L'exemple le plus courant de fonction de rectification est $ReLU(x) =$

$max(0, x)$. Ainsi, la règle d'avoir une sortie appartenant à $[0, 1]$ n'est plus considérée comme absolue aujourd'hui.

Pour récapituler, la fonction f représentée par un neurone s'écrit donc :

$$f : \begin{cases} \mathbb{R} \to [0, 1] \\ x \mapsto f_{act}(\sum_{i=1}^{n} x_i w_i \ + b) \end{cases} \tag{1.4}$$

Remarque : Le modèle de neurone que nous venons de décrire n'est pas celui qui a été théorisé en premier. Il s'agit en fait d'une version modifiée du *perceptron*, qui a été décrit pour la première fois en 1962 par Frank Rosenblatt [2, p. 119-120]. Ce dernier repose sur le même principe, mais a une sortie binaire (0 ou 1) et pas de fonction d'activation. En lui associant un ensembles d'entrées $\{x_1, x_2, ..., x_n\}$, la fonction correspondante peut s'écrire :

$$f(x) = \begin{cases} 1 & \sum_{i=1}^{n} x_i \ + b > 0 \\ 0 & sinon \end{cases} \tag{1.5}$$

Ce modèle ne dispose pas non plus de poids, toutes les entrées ont donc "la même importance".

Maintenant que nous avons défini formellement le concept de neurone, nous allons nous intéresser dans la partie suivante à une manière de les combiner pour former un réseau de neurones.

1.2.2 Structure d'un perceptron multicouche (MLP)

Comme nous l'avons dit (1.2), il existe de nombreuses structures de réseaux de neurones, c'est à dire de façons d'organiser et de connecter entre eux un grand nombre de neurones. Le modèle du MLP que nous allons décrire ici se compose de la manière suivante :

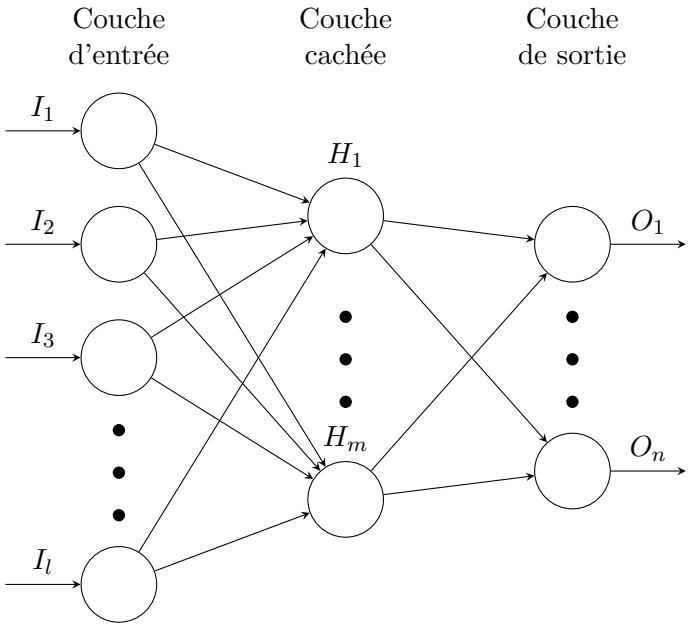

FIGURE 1.3 – Schéma d'un MLP

Comme on peut le voir, ce modèle repose sur le principe de *couches* : on distingue 3 types de couches, à savoir les couches d'entrées et de sortie, et les couches cahées. La couche d'entrée est particulière puisque comme celle-ci est censée nous fournir nos données d'entrée, elle n'est techniquement pas composée de neurones. On peut simplement la considérer comme un vecteur de réels appartenant à l'intervalle $[0, 1]$. Enfin, la couche de sortie représente la prédiction de notre réseau. Si le nombre de neurones sur ces deux couches devra être adapté au problème que l'on veut résoudre, une certaine liberté est laissée pour dimensionner les couches cachées (en taille et en nombre), qui sont censées procéder au "traitement intermédiaire" des données d'entrée. Dans le cadre de MNIST (dont nous avons parlé précédemment en 1.1.4), on peut imaginer que chaque entrée représente un pixel de l'image à analyser, et que l'on ait 10 sorties indiquant la probabilité de chaque chiffre. Les couches cachées pourraient alors être plus ou moins grandes ou nombreuses.

Remarque : Même si ce schéma ne présente qu'une seule couche cachée, il est tout à fait possible d'en combiner plusieurs à la suite. Le réseau devient alors *profond*, et on parle de *deep learning*.

Formellement, une couche est un ensemble de neurones non connectés entre eux. Cependant, deux couches peuvent être connectées entre elles. Dans ce cas, on considèrera pour l'instant que les entrées de chacun des neurones de la deuxième couche seront les sorties de tous les neurones de la première (on parle alors de couches *fortement connectées*, ou *denses*). En reprenant l'expression de la fonction d'un neurone 1.4 donnée dans la partie précédente, et en l'utilisant pour exprimer la valeur H_i d'un neurone de la couche cachée de notre schéma :

$$H_i = f_{act}(\sum_{j=1}^{l} I_j w_{j,i} \ + b_i) \tag{1.6}$$

où les $w_{j,i}$ sont les poids du neurone H_i affectés à chacune des entrées I_j et où b_i est le biais associé au neurone H_i.

Remarque : Il existe une manière plus simple d'exprimer directement la relation entre deux couches reliées (sans exprimer la valeur de chacun des neurones) en faisant appel à l'algèbre linéaire. En effet, si l'on considère l'ensemble des I_j comme un seul vecteur colonne de taille l et les $w_{i,j}$ comme une matrice de taille $m \times l$ on remarque que la somme des entrées coefficientées pour l'ensemble des H_i s'exprime sous la forme d'un produit matriciel :

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,l} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,l} \end{bmatrix} \cdot \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_l \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{l} I_j w_{j,1} \\ \vdots \\ \sum_{j=1}^{l} I_j w_{j,m} \end{bmatrix} \tag{1.7}$$

Il ne reste plus qu'à ajouter les biais de la couche H comme un vecteur colonne de taille m et à appliquer la fonction d'activation à chacun des termes pour obtenir l'expression de toute la couche H_i :

$$H_i = f_{act}\left(\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,l} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,l} \end{bmatrix} \cdot \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_l \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \right) \tag{1.8}$$

Cette façon d'écrire les relations entre couches est non seulement intéressante du point de vue de la notation, mais permettra aussi une optimisation de l'implémentation par la suite (les BLAS permettent par exemple d'effectuer cette opération de manière très efficace).

Encore une fois, un réseau de neurone peut être vu comme une fonction, même si celle-ci est très complexe. Ses entrées sont l'ensemble des neurones de la couche d'entrée et ses sorties ceux de la couche de sortie. Ainsi, les paramètres sur lesquels nous pouvons jouer pour influencer l'efficacité du modèle constitué par le réseau sont l'ensemble des poids et des biais des couches cachées et de sortie. Cette structure sous forme de couches donne en pratique de très bons résultats, même si il est difficile de savoir comment se comporte un réseau entraîné : comme il existe de nombreux paramètres à modifier, et que la phase d'entraînement se fait souvent sur de nombreux exemples, la complexité d'un réseau (même relativement petit) est souvent bien trop grande pour que l'on puisse humainement comprendre comment ce dernier peut arriver à telle ou telle prédiction. Une fois la phase d'entraînement terminée, les réseaux de neurones sont donc souvent des boîtes noires, même si il existe des méthodes aidant au choix d'un type de réseau, à son dimensionnement, etc... Nous nous intéressons justement à la phase d'entraînement dans la partie suivante.

1.2.3 Fonctions de coût et optimisation

Si nous avons vu la structure classique d'un réseau de neurone, et comment ces derniers sont capables de formuler leurs prédictions, il reste encore à définir comment les entraîner. Dans cette partie, nous décrirons le déroulement d'une phase classique d'apprentissage supervisé.

Remarque : Cette partie a pour objectif de présenter le processus d'apprentissage de manière très générale, y compris pour des modèles mathématiques n'étant pas des réseaux de neurones. Nous rentrerons plus en détail dans la suite du rapport (chapitre 4), notamment en décrivant des algorithmes d'optimisation.

Concrètement, on n'a souvent au départ que très peu d'indications (ou aucune) sur comment ajuster les différents poids et biais de notre réseau. On doit donc commencer par initialiser chacun d'eux aléatoirement, même si des techniques existent pour déterminer des valeurs initiales intéressantes. On commencera ensuite à fournir à notre réseau des données d'entraînement, avec la sortie attendue pour chaque exemple (un vecteur de même taille que la couche de sortie).

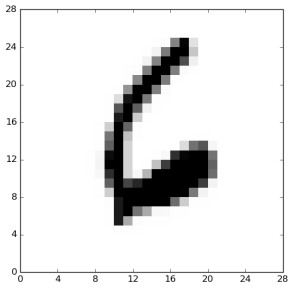

Image d'entrée (28×28 px $= 784$ entrées)

$$y^{pred} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Vecteur de sortie attendu

FIGURE 1.4 – Exemple d'entrée et de sortie attendue pour MNIST

Une fois la sortie réelle $\hat{y}$ calculée, on va la comparer à notre sortie attendue y^{pred} en calculant les différences terme à terme (mises au carré pour éliminer le signe). On obtient le vecteur $\vec{c}$:

$$\vec{c} = \begin{pmatrix} (\hat{y}_1 - y_1^{pred})^2 \\ (\hat{y}_2 - y_2^{pred})^2 \\ \vdots \\ (\hat{y}_n - y_n^{pred})^2 \end{pmatrix} \tag{1.9}$$

Afin d'exprimer simplement l'écart entre les sorties réelles et attendues, on va définir le *coût* du modèle comme étant la somme des éléments de $\vec{c}$ (notés c_i) :

$$c = \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (\hat{y}_i - y_i^{pred})^2 \tag{1.10}$$

On note la similarité avec l'équation 1.1 qui évaluait la qualité d'une régression linéaire. Cela nous permet d'obtenir une mesure scalaire pour le coût de notre modèle. Le but de la phase d'entraînement, c'est à dire faire tendre le valeur moyenne de c vers 0, est alors formalisé.

Remarque : Il existe de nombreux moyen de définir la fonction de coût d'un réseau. L'exemple décrit ici est celui de la fonction L2 (*least square error*) déjà utilisée dans la régression linéaire, mais il ne s'agit pas du seul choix possible. L'idée générale étant de calculer la norme du vecteur $|\hat{y} - y^{pred}|$, il est possible de choisir parmi de nombreuses fonctions de normes selon

notre problème. Softmax est une fonction alernative à L2 et souvent utilisée à cette effet [6].

La fonction de coût totale du réseau a donc de nombreuses entrées, qui sont non seulement celles du réseau mais aussi les poids et biais. C'est sur ces deux derniers que l'on va pouvoir agir pour trouver un minimum à cette fonction. En effet, si on arrive à faire cela sur une longue phase d'entraînement pour approcher un minimum de la fonction de coût, on aura un réseau dont les prédictions seront quasi-optimales (à supposer que le choix et le dimensionnement du modèle aient été faits judicieusement). La fonction de coût étant généralement très complexe, même pour des réseaux simples, il est pratiquement impossible d'en déterminer analytiquement un minimum. Plusieurs méthodes d'approximation, faisant appel à des *algorithmes d'optimisation* (ou *optimiseurs*, qui seront discutées plus tard) existent, et reposent toutes sur le même principe : calculer le gradient $\nabla(c)$ du coût. Celui-ci nous renseigne alors sur ses dérivées partielles par rapport à nos paramètres, et donc sur les modifications à appliquer à ceux-ci pour s'approcher d'un minimum (on va suivre la direction inverse du gradient). C'est ce qu'on appelle la *rétropropagation* (ou *backpropagation*). Le calcul des dérivées partielles (très complexes à exprimer analytiquement) peut se faire avec une méthode appellée *différentiation automatique* (AD), que nous décrirons au chapitre 2, et qui constituera la première étape de notre implémentation. Il est courant d'organiser nos données d'entrées en petits groupes (*batches*), de prendre la moyenne des gradients d'un batch, puis d'ajuster nos paramètres selon la méthode d'optimisation que l'on a choisie. Une fois qu'un observe une stagnation de ce taux à une valeur suffisament basse, on peut considérer que la phase d'entraînement est terminée, et que notre réseau est capable d'effectuer des prédictions efficacement.

Maintenant que nous avons posé les bases théoriques du machine learning, nous pouvons commencer l'implémentation des différents concepts abordés. La théorie qui vient d'être exposée sera tout de même détaillée et enrichie tout au long de ce rapport, notamment en décrivant les méthodes d'optimisations de modèles, et en proposant des architectures de réseaux plus complexes, permettant d'obtenir de meilleurs résultats ou donnant accès à de nouvelles applications.

Chapitre 2

La différentiation automatique

Comme vu en partie 1.2.3 , une partie importante de l'implémentation des réseaux de neurones (et plus généralement de nombreux modèles de machine learning) est le calcul du gradient de leur fonctions de coût. Une des méthodes les plus efficaces pour cela est la différentiation automatique. Nous commencerons par expliquer le principe de cette méthode [7], puis détaillerons comment l'implémenter et ainsi obtenir une base pour notre bibliothèque.

2.1 Principe de fonctionnement

2.1.1 Les méthodes de calcul de dérivées

On peut distinguer trois grandes méthodes de calcul de dérivées. Pour bien aborder la différentiation automatique, il peut être utile de les passer en revue, afin de mieux comprendre le principe sur lequel elle repose.

Tout d'abord, la méthode la plus évidente est la différentiation analytique. Il s'agit de manipuler l'expression de la fonction dont on veut calculer la dérivée en utilisant les règles de différentiation. Cela nous permet d'obtenir une expression analytique de la dérivée, et donc de pouvoir la calculer extrêmement facilement (il suffit d'évaluer la fonction à laquelle on a abouti). Si il est facile de faire cela manuellement pour des fonctions simples, voire possible à automatiser (c'est ce que font par exemple le framework de machine learning Theano ou le moteur de calcul WolframAlpha), le temps nécessaire pour exprimer cette dérivée peut devenir énorme. De plus,

la taille des expressions obtenues peut augmenter de manière exponentielle, ce qui augmentera aussi le temps d'évaluation (ce phénomène porte le nom d'*expression swell*). Ces défauts combinés à la dificulté d'implémenter cette méthode n'en font pas le meilleur choix dans notre contexte.

Une autre méthode est la différentiation numérique. Cette dernière reprend la définition même d'une dérivée pour éviter de formuler mathématiquement son expression. Si on a une fonction f évaluée en x, sa dérivée est :

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{2.1}$$

L'idée de cette méthode est donc d'évaluer la pente entre deux points très proche de la fonction. Cependant, on sera limité en pratique par la précision avec laquelle sont encodés les flottants, et il sera impossible de faire tendre ϵ vers 0 passé un certain point. Ainsi, si ϵ est trop grand, le calcul sera par nature imprécis, et si il est trop faible, on souffrira des erreurs de troncation. La différentiation numérique ne permet donc pas une évaluation très précise des dérivées, particulèrement pour des fonctions aux pentes élevées. Il ne s'agit donc pas non plus de la méthode que nous allons favoriser.

La différentiation automatique n'évalue pas non plus l'expression de la dérivée, mais permet en théorie d'obtenir sa valeur exacte. Le principe repose sur l'utilisation du théorème de dérivation des fonctions composées (*chain rule*) [8]. Pour rappeler ce dernier, si f et g sont deux fonctions différentiables, alors :

$$(f \circ g)' = (f' \circ g).g' \tag{2.2}$$

Ce théorème peut être réécrit avec la notation de Leibniz, plus proche de celle des dérivées partielles que nous utiliserons par la suite. Si x, y et z sont trois variables, avec z dépendant de y et y dépendant de x :

$$\frac{dz}{dx} = \frac{dz}{dy}.\frac{dy}{dx} \tag{2.3}$$

L'idée est de considérer la fonction que l'on veut différencier comme une composition de fonctions simples dont on connaît les dérivées. En utilisant 2.3, on peut calculer les dérivées de chacune des fonctions intermédiaires en un point donné. Ces fonctions simples peuvent être les opérateurs arithmétiques, ou des fonctions usuelles telles que le logarithme, les fonctions trigonométriques, etc... En procédant étape par étape, on obtient la dérivée qui nous intéresse. Le temps de calcul nécessaire pour réaliser un tel procédé reste relativement

limité, et on est alors capable d'obtenir la valeur exacte de la dérivée (sans pour autant en formuler une expression analytique). En pratique, le temps nécessaire pour évaluer une dérivée via la différentiation automatique est comparable à celui nécessaire pour évaluer la fonction de base, ce qui en fait une solution très efficace dans de nombreuses applications. Il existe deux façons de mettre en oeuvre cette technique, que nous allons détailler dans les deux parties suivantes (2.1.2 et 2.1.3).

2.1.2 Mode forward

Le mode vers l'avant (ou *forward*) est sûrement le plus intuitif et simple à aborder dans un premier temps. Illustrons-le sur une fonction d'exemple :

$$f : \begin{cases} \mathbb{R}^2 \to \mathbb{R} \\ (x_1, x_2) \mapsto x_2.cos(x_1) + e^{x_2} \end{cases} \tag{2.4}$$

Cette fonction particulière est très simple, et on peut donc exprimer analytiquement ses dérivées partielles sans problèmes :

$$\begin{cases} \frac{\partial f}{\partial x_1} = -x_2 sin(x_1) \\ \frac{\partial f}{\partial x_2} = cos(x_1) + e^{x_2} \end{cases} \tag{2.5}$$

Cependant, la différentiation automatique nous permet d'évaluer $\frac{\partial f}{\partial x_1}$ et de $\frac{\partial f}{\partial x_2}$ en un point donné sans même connaître leurs expressions. Pour mieux comprendre ce processus, commençons par construire le graphe de calcul de f :

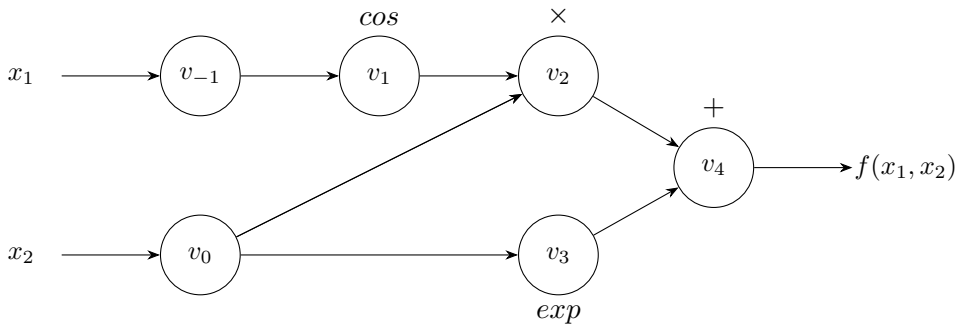

FIGURE 2.1 – Graphe de calcul de f

Sur ce graphe :

— les variables v_{-1} et v_0 sont les entrées de f (respectivement x_1 et x_2)
— les variables v_1 à v_4 sont les variables intermédiaires, v_4 étant la sortie de la fonction, c'est à dire $f(x_1, x_2)$

Chaque nouveau noeud correspond à une variable intermédiaire pour évaluer la valeur de la fonction, et cette représentation sous forme de graphe permet de voir facilement dans quel ordre les "fonctions élémentaires" sont évaluées conformément à la priorité des opérateurs. A titre d'exemple, évaluons f et ses dérivées partielles pour $x_1 = \pi$ et $x_2 = 1$:

$$\begin{cases} f(\pi, 1) = 1.cos(\pi) + e^1 = e - 1 \\ \frac{\partial f}{\partial x_1} = -1.sin(\pi) = 0 \\ \frac{\partial f}{\partial x_2} = cos(\pi) + e^1 = e - 1 \end{cases} \tag{2.6}$$

Voyons maintenant comment faire cela avec la différentiation automatique en mode forward en se basant sur le graphe de calcul 2.1 :

Evaluation de f	Evaluation de $\frac{\partial f}{\partial x_1}$	Evaluation de $\frac{\partial f}{\partial x_2}$
$v_{-1} = x_1 = \pi$ $v_0 = x_2 = 1$	$\dot{v_{-1}} = \dot{x_1} = 1$ $\dot{v_0} = \dot{x_2} = 0$	$\dot{v_{-1}} = \dot{x_1} = 0$ $\dot{v_0} = \dot{x_2} = 1$
$v_1 = cos(v_{-1}) = -1$ $v_2 = v_0.v_1 = -1$ $v_3 = e^{v_0} = e$	$\dot{v_1} = -\dot{v_{-1}}.sin(v_{-1}) = 0$ $\dot{v_2} = \dot{v_0}.v_1 + v_0.\dot{v_1} = 0$ $\dot{v_3} = \dot{v_0}.e^{v_0} = 0$	$\dot{v_1} = -\dot{v_{-1}}.sin(v_{-1}) = 0$ $\dot{v_2} = \dot{v_0}.v_1 + v_0.\dot{v_1} = -1$ $\dot{v_3} = \dot{v_0}.e^{v_0} = e$
$v_4 = v_2 + v_3 = e - 1$	$\dot{v_4} = \dot{v_2} + \dot{v_3} = 0$	$\dot{v_4} = \dot{v_2} + \dot{v_3} = e - 1$

TABLE 2.1 – Evaluation de f et de ses dérivées partielles en mode forward

On retrouve bien les résultats obtenus avec les expressions analytiques non seulement de f, mais aussi de $\frac{\partial f}{\partial x_1}$ et $\frac{\partial f}{\partial x_2}$. Dans le cas général, on peut réutiliser l'équation 2.3 pour exprimer la dérivée partielle d'une variable z dépendant d'un ensemble $\{y_i\}_{i=1}^n$ de variables par rapport à x [9] :

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} . \frac{\partial y_i}{\partial x} \tag{2.7}$$

On remarque alors que les expressions des $\dot{v_1}$ à $\dot{v_4}$ dans le tableau 2.1 peuvent toutes être rapportées à cette équation.

Remarque : Pour terminer sur le mode forward, notons que ce dernier nous permet de calculer les dérivées partielles de toutes les variables de sortie par rapport à une variable d'entrée (dans notre exemple, nous avons donc dû faire deux passages, pour x_1 puis x_2). Dans le cas général où $f : \mathbb{R}^m \to \mathbb{R}^n$, un passage en mode forward nous donne alors une colonne de la matrice jacobienne de f lorqu'elle est évaluée en un point a :

$$J_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}_{x=a} \tag{2.8}$$

2.1.3 Mode backward

Dans le mode forward, on parcourt notre graphe de calcul "vers l'avant" lorsque l'on évalue une dérivée partielle, donc de la même manière que pour évaluer la fonction de base. L'intuition du mode backward est de calculer la dérivée une fois que la fonction a été évaluée, en "remontant" le graphe vers l'arrière. Plus formellement, le mode backward repose sur une reformulation de l'équation 2.7. Cette fois, on cherche à exprimer la dérivée partielle par rapport à z d'une variable x dont dépend un ensemble $\{y_i\}_{i=1}^n$ de variables [9] :

$$\frac{\partial x}{\partial z} = \sum_{i=1}^{n} \frac{\partial y_i}{\partial z} . \frac{\partial x}{\partial y_i} \tag{2.9}$$

Dans cette équation, on exprime la dérivée partielle d'une variable non plus en fonction des variables dont elle dépend, mais en fonction des variables qui dépendent d'elles. Appliquons cette nouvelle méthode sur l'exemple de la partie 2.1.2 :

Evaluation de f	Evaluation des $\frac{\partial f}{\partial x_i}$
$v_{-1} = x_1 = \pi$	$\bar{v_{-1}} = \bar{v_1} . \frac{\partial v_1}{\partial v_{-1}} = 0$
$v_0 = x_2 = 1$	$\bar{v_0} = \bar{v_3} . \frac{\partial v_3}{\partial v_0} + \bar{v_2} . \frac{\partial v_2}{\partial v_0} = e - 1$
$v_1 = cos(v_{-1}) = -1$	$\bar{v_1} = \bar{v_2} . \frac{\partial v_2}{\partial v_1} = -1$
$v_2 = v_0 . v_1 = -1$	$\bar{v_2} = \bar{v_4} . \frac{\partial v_4}{\partial v_2} = -1$
$v_3 = e^{v_0} = e$	$\bar{v_3} = \bar{v_4} . \frac{\partial v_4}{\partial v_3} = e$
$v_4 = v_2 + v_3 = e - 1$	$\bar{v_4} = 1$

TABLE 2.2 – Evaluation de f et de ses dérivées partielles en mode backward

Par rapport au mode forward, l'évaluation de f ne change donc pas. Cependant, le calcul des dérivées partielles partira de $\frac{\partial f}{\partial v_4} = \frac{\partial v_4}{\partial v_4} = \bar{v_4} = 1$ afin de calculer les autres $\frac{\partial v_4}{\partial v_i} = \bar{v_i}$. Quand on sera arrivé à $\bar{v_{-1}}$ et $\bar{v_0}$, on aura calculé $\frac{\partial f}{\partial x_1}$ et $\frac{\partial f}{\partial x_2}$. Comme f n'avait ici qu'une sortie scalaire, obtenir toutes les dérivées partielles n'a nécessité qu'un seul passage. Notons cependant que le mode backward a toujours le désavantage de demander plus de mémoire que le mode forward, car il est nécessaire de stocker toutes les dérivées partielles jusqu'au passage en arrière.

Les deux modes de la différentiation automatique ayant été passés en revue, nous pouvons maintenant conclure sur l'efficacité de ces derniers. Nous avons vu en partie 2.1.2 que le mode forward permettait d'évaluer les dérivées partielles de toutes les sorties par rapport à une variable d'entrée donnée, c'est à dire de nous donner une colonne du jacobien de notre fonction. A l'inverse, le mode forward calcule les dérivées partielles d'une sortie par rapport à toutes les entrées, et donne ainsi une ligne du jacobien. Ainsi, pour $f : \mathbb{R}^m \to \mathbb{R}^n$:

— si $n > m$, il est plus facile de calculer J_f avec le mode forward (m passages)
— si $n < m$, il est plus facile de calculer J_f avec le mode backward (n passages)

Pour remettre cela dans notre contexte, il sera ainsi bien plus simple d'exprimer le gradient de la fonction de coût à l'aide du mode backward. Nous avons effectivement vu que celle-ci calculait la norme de la différence entre la sortie de notre modèle, et la sortie attendue. Il s'agit donc d'un scalaire, et l'entièreté du gradient pourra être évaluée en seulement un passage. Dans la prochaine partie, nous nous intéresserons donc à l'implémentation de cette technique en C++. Celle-ci constituera la première étape du développement de notre bibliothèque, et sera intégrée avec les autres composants au fur et à mesure de leur développement.

2.2 Implémentation

Dans cette section, nous détaillerons une méthode d'implémentation de la différentiation automatique en mode backward. Dans un premier temps, nous expliquerons son principe de fonctionnement, puis détaillerons le code C++ correspondant.

2.2.1 Listes de Wengert

La première problématique à laquelle nous serons confrontés sera la construction du graphe de calcul (comme celui vu en 2.1). La deuxième sera de rendre notre implémentation la plus transparente possible : dans l'idéal, on voudrait réaliser une suite de calculs sans se préoccuper de la différentiation automatique, et être capable de différencier le résultat via un simple appel de fonction par exemple.

Une solution permettant de répondre à ces deux exigences serait de construire le graphe au fur et à mesure que nous effectuons les calculs. Ce dernier pourra alors être stocké sous la forme d'une *liste de Wengert*, parfois appelée *tape* [9].

Le principe est de stocker les noeuds du graphe de calcul sous la forme d'un tableau auquel on ajoutera en dernière position les nouveaux noeuds, au fur et à mesure que les calculs sont effectués. Chaque noeud comprendra une référence vers l'indice du ou des noeud(s) dont il dépend, ainsi que les dérivées partielles par rapport aux opérandes (correspondant à ces noeuds). Par exemple, dans le cas de l'opération $y = f(a, b)$, le noeud créé contiendra les indices des noeuds associés à a et b, ainsi que les dérivées partielles $\frac{\partial f(a,b)}{\partial a}$ et $\frac{\partial f(a,b)}{\partial b}$. Le résultat, qui n'est pas nécessaire au passage en arrière, n'est donc pas stocké dans le noeud en lui même. Graphiquement, une liste de Wengert pourrait être représentée de la manière suivante (considérons par exemple le calcul $\sqrt{(a+b)\times c}$) :

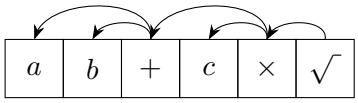

FIGURE 2.2 – Exemple de liste de Wengert

Cette méthode a l'avantage d'être très efficace en mémoire (pour une implémentation du mode backward). Cependant, dans le cas où l'on voudrait désallouer une partie du tableau, cela ne pourrait pas se faire de manière triviale (il faudrait ajuster tous les indices/pointeurs des noeuds dépendant des noeuds supprimés).

Afin de construire cette liste, nous définirons notre propre type de données : ce dernier contiendra une référence vers sa liste de Wengert et son index

dans cette dernière. Nous pourrons alors surcharger les opérateurs basiques et développer nos propres fonctions pour automatiser des calculs plus complexes. Pour chaque opérateur ou fonction, on aura donc un nouveau "type" de noeud, et il conviendra de rajouter le nouveau noeud à la liste de Wengert (avec les dérivées partielles des opérandes) avant de retourner le résultat du calcul.

Enfin, pour calculer un gradient, il suffira de créer un nouveau tableau rempli de zéros (de même taille que la liste de Wengert W), et de "remonter le graphe" depuis la fin de la liste (ou au moins depuis le noeud par rapport auquel est calculé le gradient) pour incrémenter chaque dérivée partielle en fonction des dépendances stockées dans son noeud, et ce en suivant le théorème de dérivation des fonctions composées (comme vu en 2.9). Formellement, cet algorithme calculant le gradient par rapport à une variable v de noeud d'indice k peut alors s'écrire comme suit :

Algorithme 1 : Calcul de gradient depuis une liste de Wengert

Entrée : liste de Wengert W, variable v associée au noeud W_k
Résultat : gradient ∇

$\nabla \leftarrow \{0\}_0^{|W|}$
$\nabla_k \leftarrow 1$
`/* Parcours depuis la fin */`
pour $i \leftarrow k$ **à** 1 **faire**
  $C \leftarrow enfants(W_k)$
  pour $j \leftarrow 0$ **à** $|C|$ **faire**
    `/* Indice dans` C `!= Indice dans` W `ou` ∇ `*/`
    $j' \leftarrow indice_{\nabla}(C_j)$
    $\nabla_{j'} \leftarrow \nabla_{j'} + \frac{\partial C_i}{\partial W_k} \cdot \frac{\partial C_j}{\partial C_i}$
  fin
fin

2.2.2 Implémentation en C++

Maintenant que nous avons posé les bases de notre implémentation, commençons naïvement par écrire une première version du code en C++. Le code présenté dans cette partie ne sera cependant pas celui retenu dans la bibliothèque, car nous présenterons dans la prochaine partie une manière de l'améliorer dans notre contexte. Le principe restera tout de même identique, et cette version du code sera donc la plus simple à appréhender dans un premier temps.

Nous aurons donc besoin de créer quatre classes : `ts::WengertList`, `ts::Node`, `ts::Variable` et `ts::Gradient`. `ts::WengertList` contiendra un `std::vector` de `ts::Node` qui sera donc notre liste. Un `ts::Node` devra alors contenir les dépendances qui lui sont associées :

```
template <typename T>
class ts::Node {
protected:
    // Represents an input variable
    Node();

    // Represents a unary operator
    Node(T xVal, int xDep);

    // Represents a binary operator
    Node(T xVal, int xDep, T yVal, int yDep);

    std::vector<int> dependencies{};
    std::vector<T> values{};

    // ...
};
```

Listing 2.1 – Définition de `ts::Node`

Comme on peut le voir dans les constructeurs, la création d'un noeud ayant des dépendances nécessite d'avoir calculé les dérivées partielles au préalable. On remarque aussi que l'intégralité de la classe est protégée, car les noeuds ne sont pas censés être visibles à un utilisateur de la bibliothèque : les noeuds seront tous créés automatiquement, et les dépendances / valeurs ne seront nécessaires que dans la fonction de calcul du gradient.

Intérressons-nous maintenant aux fonctions de calcul des opérations élémentaires. La classe `ts::Variable` encapsule un type numérique `T`, et nous permettra ainsi de surcharger nos opérateurs et de créer des fonctions dédiées pour d'autres opérations. Toutes ces fonctions reposeront sur le même principe, qui est de calculer le résultat de l'opération, les dérivées partielles, puis avec ces informations, d'ajouter le nouveau noeud à la liste avant d'enfin retourner le résultat. A titre d'exemple, voici le code associé à l'opérateur `*` de multiplication :

```cpp
template <typename T>
ts::Variable<T> ts::operator*(const ts::Variable<T>
    &x, const ts::Variable<T> &y){

    if(x.wList != y.wList) {
        // Return "empty" variable (no wList set)
        return ts::Variable<T>(0, NULL);
    }


    // a = x * y
    // da / dx = y
    // da / dy = x

    std::shared_ptr<ts::Node<T>> nodePtr (
        new ts::Node<T>(
            y.value, x.index,
            x.value, y.index
        )
    );

    return ts::Variable<T>(x.value * y.value,
        x.wList, nodePtr);
}
```

Listing 2.2 – Surcharge de l'opérateur *

On retrouve bien les étapes que nous avons détaillé. On note aussi qu'il est utile de vérifier que les deux opérandes appartiennent bien à la même liste de Wengert, comme cela est fait au début de la fonction. Evidemment, cet exemple est assez simple : les dérivées partielles des opérandes d'une multiplication sont triviaux à obtenir, on peut donc directement créer le noeud à ajouter. Pour certaines opérations, le calcul de ces dérivées sera moins évident, mais cet exemple permet de bien voir la construction des fonctions d'opérations élémentaires.

Enfin, nous allons voir comment implémenter le calcul du gradient en parcourant la liste depuis sa fin. Cela correspond ainsi à l'implémentation de l'algorithme 1 :

```cpp
template <typename T>
ts::Gradient<T> ts::Variable<T>::grad() {

    std::vector<T>
       derivatives(wList->nodes.size(),0);

    // Initialize gradient of self with respect to
       itself
    derivatives[index] = 1.0;

    // Iterate over the Wengert list backwards
    for (unsigned i=wList->nodes.size(); i-->0;) {

        std::shared_ptr<ts::Node<T>> node =
           wList->nodes[i];

        // Increment parent nodes
        for(unsigned j = 0; j <
           node->dependencies.size(); j++) {
            derivatives[node->dependencies[j]] +=
            derivatives[i] * node->values[j];
        }
    }

    return ts::Gradient<T>(derivatives);
}
```

Listing 2.3 – Fonction de calcul du gradient

Comme le calcul du gradient doit se faire par rapport à une variable bien précise, cette fonction est implémentée sous la forme d'une méthode de la classe `ts::Variable`. Comme nous l'avons dit, on commence donc par initialiser un vecteur dont les éléments sont nuls, et dont la taille correspond à la liste de Wengert (les dérivées partielles de chaque élément de la liste seront ainsi obtenues en accédant à l'élément du gradient de même indice). La première dérivée partielle peut être obtenue de manière triviale : si l'on a un ensemble de variables $\{y_i\}_{i=1}^{n}$, et que l'on calcule un gradient ∇ par rapport à la variable y_i, alors l'élément i du gradient sera :

$$\nabla_i = \frac{\partial y_i}{\partial y_i} = 1 \tag{2.10}$$

On peut donc initialiser à 1 l'élément du vecteur du gradient correspondant à notre variable de référence (`derivatives[index]`). Puis, on parcourt vers l'arrière la liste de Wengert `wList->nodes`, en incrémentant chacune des dépendances de nos noeuds. Ici, par rapport à l'équation 2.9 (notre réécriture du théorème de dérivation des fonctions composées), le terme $\frac{\partial y_i}{\partial z}$ correspond à `derivatives[i]`, et $\frac{\partial x}{\partial y_i}$ correspond donc à `node->values[j]`.

Remarque : Comme la liste de Wengert est construite dans l'ordre dans lequel sont effectués les calculs, si l'on veut calculer l'élément ∇_i dépendant de ∇_j, avec $j > i$, on est sûrs que la valeur de ∇_j est déjà bonne au moment de l'évaluation de ∇_i.

Une fois que le parcours du tableau est terminé, on retourne le gradient dans un `ts::Gradient`, qui permet simplement d'encapsuler et de protéger l'accès au vecteur contenant toutes nos dérivées partielles. Pour finir, voyons un exemple d'utilisation du code que nous venons de présenter :

```
1  ts::WengertList<float> wList;
2
3  ts::Variable<float> a = ts::Variable<float>(10,
       &wList);
4  ts::Variable<float> b = ts::Variable<float>(4,
       &wList);
5  ts::Variable<float> c = a * b;
6
7  ts::Variable<float> d = ts::Variable<float>(2,
       &wList);
8  ts::Variable<float> e = c + d;
9
10 ts::Gradient<float> = e.grad();
11
12 float gradA = grad.getValue(a); // 4
13 float gradB = grad.getValue(b); // 10
14 float gradD = grad.getValue(d); // 1
```

Listing 2.4 – Exemple d'utilisation de la différentiation automatique

2.3 Généralisation aux tenseurs

2.3.1 Intérêt

L'implémentation de la différentiation automatique que nous avons actuellement est parfaitement fonctionnelle, et pourrait tout à fait être utilisée dans le cadre de notre bibliothèque. Cependant, nous avons vu en partie 1.2 qu'il est bien plus simple de ramener le calcul de la sortie d'un réseau de neurones à un ensemble d'opérations sur des matrices et vecteurs. Ainsi, en généralisation notre moteur de différentiation automatique aux matrices (voire plus généralement aux tenseurs), nous pourrions simplifier grandement nos graphes de calcul. Sans rentrer dans des calculs détaillés, un réseau de neurones ayant plusieurs centaines de paramètres (ce qui est relativement faible pour beaucoup d'applications réelles) donnerait un graphe de calcul ayant encore plus de noeuds (facilement de l'ordre de plusieurs milliers). En construisant un graphe de calcul avec des tenseurs, on pourrait par exemple résumer une couche entièrement connectée à seulement une matrice de poids et un vecteur de biais, ce qui donnerait un graphe ayant quelques dizaines de noeuds tout au plus. L'intérêt d'une telle optimisation devient alors évident.

Pour faire cela, il nous faudra changer le type encapsulé dans `ts::Variable` par un type représentant un tenseur. Nous choisirons d'utiliser la bibliothèque Eigen, dont la documentation est disponible à l'adresse `http://eigen.tuxfamily.org`. Eigen est une bibliothèque d'algèbre linéaire offrant de très bonnes performances, et qui constituera pour nous une très bonne base pour effectuer de nombreuses opérations vectorielles et matricielles. Nous choisirons donc d'encapsuler le type `Eigen::Array`, qui permet de stocker un tableau 2D dense. Il est souvent privilégié pour des opérations terme à terme, et doit donc être différencié du type `Eigen::Matrix`. Cependant, il est possible d'effectuer la conversion entre ces deux types en $O(1)$, ce choix n'affectera donc pas les performances finales, et permettra surtout un plus grand confort pendant le développement, puisque nous serons plus souvent amenés à utiliser les opérations de `Eigen::Array`. Ainsi, on se limitera à la représentation de tenseurs d'ordre 2, même si cela sera suffisant dans la plupart des cas (il sera de toute façon toujours possible de se rapporter à un cas où l'on manipule des tenseurs d'ordre 2). Les vecteurs sont eux considérés comme des cas spéciaux de `Eigen::Array` ou de `Eigen::Matrix` dans Eigen, ils n'ont donc pas de type dédié. Un vecteur sera alors représenté comme une matrice de forme $1 \times n$ ou $n \times 1$, et un scalaire sera de forme 1×1. Ainsi, notre classe `ts::Variable` sera renommée en `ts::Tensor`.

2.3.2 Opérations terme à terme

Un problème se pose cependant avec l'approche que nous avons adoptée. La dérivée d'un tenseur d'ordre 2 par un autre tenseur non scalaire donne un tenseur d'ordre supérieur à 2 [10]. En toute rigueur, nous devrons donc nous limiter à calculer des gradients par rapport à des scalaires, ce qui est justement le cas lors du calcul du gradient d'une fonction de coût. Cela ne posera ainsi pas de problème majeur par la suite, et sera suffisant pour la grande majorité des applications réalistes.

Dans un premier temps, considérons uniquement les opérations terme à terme, puisque ce sont globalement les plus simples à différencier. Dans ce cas précis, nous nous permettrons d'ailleurs de déroger à la définition stricte de la dérivation de tenseurs que nous venons d'évoquer, et considèrerons que la dérivée d'une matrice par rapport à une autre matrice reste une matrice si seules des opérations terme à terme ont été effectuées. Plus précisément, on prendra la définition suivante : si A, B et C sont trois matrices de forme $n \times m$, et F une opération terme à terme telle que :

$$\begin{aligned} F(A,B) &= F\left(\begin{bmatrix} a_{(1,1)} & \cdots & a_{(1,m)} \\ \vdots & \ddots & \vdots \\ a_{(n,1)} & \cdots & a_{(n,m)} \end{bmatrix}, \begin{bmatrix} b_{(1,1)} & \cdots & b_{(1,m)} \\ \vdots & \ddots & \vdots \\ b_{(n,1)} & \cdots & b_{(n,m)} \end{bmatrix}\right) \\ &= \begin{bmatrix} f(a_{(1,1)}, b_{(1,1)}) & \cdots & f(a_{(1,m)}, b_{(1,m)}) \\ \vdots & \ddots & \vdots \\ f(a_{(n,1)}, b_{(n,1)}) & \cdots & f(a_{(n,m)}, b_{(n,m)}) \end{bmatrix} \\ &= C \end{aligned} \tag{2.11}$$

Alors, la dérivée partielle de C par rapport à A est :

$$\frac{\partial C}{\partial A} = \begin{bmatrix} \frac{\partial f(a_{(1,1)}, b_{(1,1)})}{\partial a_{(1,1)}} & \cdots & \frac{\partial f(a_{(1,m)}, b_{(1,m)})}{\partial a_{(1,m)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(a_{(n,1)}, b_{(n,1)})}{\partial a_{(n,1)}} & \cdots & \frac{\partial f(a_{(n,m)}, b_{(n,m)})}{\partial a_{(n,m)}} \end{bmatrix} \tag{2.12}$$

... et il en va de même pour $\frac{\partial C}{\partial B}$. Ce détail n'aura pas une grande importance dans la plupart des applications réelles, mais permet plus de flexibilité dans le cas où on ne s'intéresserait qu'à des opérations terme à terme. Cela explique aussi la manière dont a été implémenté le mécanisme de "garde" pour éviter le calcul d'un gradient dont les termes dépasseraient l'ordre 2 :

```
template <typename T>
ts::Gradient<T> ts::Tensor<T>::grad() {
    // 2 possibilities :
    // - All operations are element wise, so we
       allow this tensor not to be a scalar
    // - Some operations change shapes of tensors,
       we only allow this tensor to be scalar

    // Making sure that we're not in case 2 with a
       non-scalar tensor
    if(!wList->elementWiseOnly && value.rows() != 1
       && value.cols() != 1) {
        return ts::Gradient<T>({});
%   }

    // ...
```

Listing 2.5 – Mécanisme de garde du gradient

Le champ `elementWiseOnly` de `ts::WengertList` est un booléen initialisé à 1. Dans le cas où l'on effectuerait une opération modifiant la forme d'un tenseur, cette valeur serait mise à 0. Si le tenseur sur lequel est appelé la méthode n'es pas un scalaire, on retournerait alors un gradient vide indiquant l'impossibilité d'effectuer le calcul (en tout cas dans le cadre de notre bibliothèque).

Pour le reste, le calcul d'opérations terme à terme reste assez trivial, et ne présente pas de grande particularité par rapport à notre implémentation sur des scalaires. A titre de comparaison (par rapport au listing 2.2), voici le code du produit terme à terme (ou produit de Hadamard) :

```cpp
template <typename T>
ts::Tensor<T> ts::operator*(const ts::Tensor<T> &x,
    const ts::Tensor<T> &y){
    if(
        x.wList != y.wList ||
        x.value.rows() != y.value.rows() ||
        x.value.cols() != y.value.cols()
    ) {
        return ts::Tensor<T>(Eigen::Array<T, 0,
            0>(), NULL);
    }

    // a = x * y
    // da / dx = y
    // da / dy = x
    std::shared_ptr<ts::Node<T>> nodePtr (
        new ts::ElementWiseNode<T>(
            {x.value.rows(), x.value.cols()},
            y.value, x.index,
            x.value, y.index
        )
    );

    return ts::Tensor<T>(x.value * y.value,x.wList,
        nodePtr);
}
```

Listing 2.6 – Produit de Hadamard (opérateur `*`)

La principale différence vient surtout du fait que par sécurité, nous décidons maintenant de stocker dans chaque noeud la forme du tenseur correspondant. Cela est pour l'instant plus une précaution qu'autre chose, mais s'avèrera cependant nécessaire lorsque nous nous intéresserons à certaines opérations modifiant les formes des tenseurs. Le noeud est alors un `ts::ElementWiseNode`, héritant de `ts::Node`, mais nous reviendrons sur ce point dans la partie suivante. Aussi, il est maintenant nécessaire au début de la fonction de vérifier que les deux opérandes ont la même forme, sans quoi il sera impossible d'effectuer le calcul. Pour le reste, il suffit d'adapter notre code en replaçant le type `T` par un `Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic>` (tableau de taille dynamique dont les éléments sont de

type `T`).

2.3.3 Produit matriciel et norme

La plus grande difficulté pour généraliser la différentiation automatique viendra donc des opérations qui modifient la taille des tenseurs. En effet, pour incrémenter les dérivées partielles lors du passage en arrière, il suffisait jusqu'à maintenant d'utiliser un produit terme à terme. Cependant, quand les deux termes servant à calculer l'incrément sont de forme différente, cela est plus compliqué. Le calcul des dérivées partielles de ce type d'opération n'est souvent pas trivial, et il faudra donc procéder au cas par cas pour le calcul de l'incrément. C'est pour cela que (comme nous l'avons vu dans le listing 2.6), la classe `ts::Node` servira maintenant de classe virtuelle pour définir différents types de noeuds. Chaque type aura sa propre méthode `incrementGradient`, pour laquelle le mode de calcul de l'incrément sera adapté. Dans un premier temps, nous ne nous intéresserons qu'au produit de matrice et à la norme 2. Ces opérations seront nécessaires pour développer le modèle du MLP. Le produit matriciel sera utilisé dans le calcul des poids (et donc plus généralement dans le calcul de couches fortement connectées, même dans d'autres types de réseaux), et la norme 2 pourra être utilisée pour la fonction de coût de tous les types de réseaux.

Dans un premier temps donc, voyons comment différencier un produit de matrices [11]. Pour cela, plaçons-nous dans un cas très général. On considère une suite de calculs aboutissant à un résultat scalaire S_O, et dont A et B sont des valeurs intermédiaire matricielles. Maintenant, soit une étape intermédiaire de la forme $C = f(A, B)$. Alors, $\bar{A}$ est la dérivée partielle de S_O par rapport à tous les éléments de A (de même pour $\bar{B}$ et $\bar{C}$). $\bar{A}$, $\bar{B}$ et $\bar{C}$ sont de même forme que A, B et C respectivement. Si A et B ne sont pas utilisés dans d'autres calculs intermédiaires (en d'autres terme, on ne s'intéresse qu'à la dérivation de f), alors :

$$\begin{cases} \bar{A} = \left(\frac{\partial f}{\partial A}\right)^T \bar{C} \\ \bar{B} = \left(\frac{\partial f}{\partial B}\right)^T \bar{C} \end{cases} \tag{2.13}$$

Dans le cas où f correspond au produit matriciel, 2.13 devient :

$$\begin{cases} \bar{A} = \bar{C}.B^T \\ \bar{B} = A^T.\bar{C} \end{cases} \tag{2.14}$$

Si A est de forme $l \times m$, B de forme $m \times n$, alors C est de forme $l \times n$, A^T de forme $m \times l$ et B^T de forme $n \times m$. $\bar{A}$ s'obtient bien par le produit de matrices de forme $l \times n$ et $n \times m$, et $\bar{B}$ par le produit de matrices de forme $m \times l$ et $l \times n$. Bien sûr, dans le cadre de la différentiation automatique en mode backward, $\bar{C}$ est déjà connue lorsque l'on calcule 2.14.

Pour revenir à notre moteur de différentiation automatique, voici comment le produit matriciel sera implémenté :

```
template <typename T>
ts::Tensor<T> ts::matProd(const ts::Tensor<T> &x,
    const ts::Tensor<T> &y) {
    if(x.wList != y.wList || x.value.cols() !=
        y.value.rows()) {
        return ts::Tensor<T>(Eigen::Array<T, 0,
            0>(), NULL);
    }

    // The gradient will have to be computed for a
        scalar
    x.wList->elementWiseOnly = false;

    // a = x.y
    // dx = y^T (transposed)
    // dy = x^T

    std::shared_ptr<ts::Node<T>> nodePtr (
        new ts::MatProdNode<T>(
            {x.value.rows(), y.value.cols()},
            y.value.matrix().transpose(), x.index,
            x.value.matrix().transpose(), y.index,
            {x.value.rows(), x.value.cols()},
                {y.value.rows(), y.value.cols()}
        )
    );

    return ts::Tensor<T>( x.value.matrix() *
        y.value.matrix(), x.wList, nodePtr);
}
```

Listing 2.7 – Produit matriciel

Tout d'abord, on vérifie que les deux matrices peuvent être multipliées entre elles, c'est à dire que `x.value.cols()` et `y.value.rows()` ont la même valeur. Ensuite, on marque la liste de Wengert comme ne comportant pas que des opérations terme à terme, en passant `elementWiseOnly` à 0. Puis, on crée notre noeud de type `ts::MatProdNode`, en stockant dans celui-ci les transposées `y.value.matrix().transpose()` et `x.value.matrix().transpose()` des opérandes. Enfin, et comme déjà vu précédemmet, on retourne le `ts::Tensor` avec le résultat du produit matriciel. On note aussi que cette fonction constitue un très bon exemple de conversion entre `Eigen::Array` et `Eigen::Matrix`.

La dérivée partielle d'une variable utilisée dans un produit de matrices s'incrémentera alors comme suit, dans la méthode `ts::MatProdNode<T>::incrementGradient` :

```
    // ...

    Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic>
       increment;

    // Incremening x
    if(
        xSize[1] == this->values[j].rows() &&
        xSize[0] == this->values[j].cols()
    ) {
        increment = (this->values[j].matrix() *
           childDerivative.matrix()).array();
    }

    // Incrementing y
    else if(
        ySize[1] == this->values[j].rows() &&
        ySize[0] == this->values[j].cols()
    ) {
        increment = (childDerivative.matrix() *
           this->values[j].matrix() ).array();
    }

    return increment;
}
```

Listing 2.8 – Incrément d'un noeud `ts::MatProdNode`

Le cas du produit matriciel est peut être un des plus délicats : en effet, cette opération n'étant pas commutative, l'ordre des opérandes dans l'équation 2.14 n'est pas le même selon que l'on incrémente la dérivée de A ou B. Pour choisr le bon ordre, il faudra alors comparer la taille de `childDerivative` ($\bar{C}$ dans 2.14) avec celles des `this->values[j]` (A^T et B^T dans 2.14). Il est ainsi possible de savoir sur quel opérande nous travaillons, et de se placer dans le bon cas de figure.

Remarque : Dans le cas du produit matriciel, les valeurs stockées dans le noeud correspondant ne seront techniquement pas les dérivées partielles de l'opération. Cependant, cela n'a pas réellement d'importance, puisque ces valeurs sont uniquement utilisées dans le cadre de la méthode `incrementGradient`. Plus généralement, cela sera vrai pour la grande majorité des opérations non terme à terme.

Pour finir, abordons rapidement la différentiation de la norme 2. Dans notre implémentation, nous avons généralisé la définition de la norme 2 aux matrices. Si A est une matrice de forme $m \times n$, la norme 2 de A sera :

$$\|A\|_2 = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2 \tag{2.15}$$

La sortie $\|A\|_2$ de la fonction de norme 2 étant un scalaire, il est plus facile d'exprimer directement la dérivée $\bar{A}$. Cette dernière est composée de la dérivée partielle de $\|A\|_2$ par rapport à chacun des éléments $a_{i,j}$ de A. Chacun des éléments n'apparaissant qu'une fois, il est assez trivial de les exprimer. En effet, $\forall i \in \{1, ..., m\}, \forall j \in \{1, ..., n\}$:

$$\frac{\partial \|A\|_2}{\partial a_{i,j}} = 2a_{i,j} \tag{2.16}$$

D'où :

$$\|\bar{A}\|_2 = \begin{bmatrix} 2a_{1,1} & \cdots & 2a_{1,n} \\ \vdots & \ddots & \vdots \\ 2a_{m,1} & \cdots & 2a_{m,n} \end{bmatrix} \tag{2.17}$$

Comme $\overline{\|A\|}_2$ est scalaire, le calcul de l'incrément se fera très facilement en calculant le produit scalaire-matrice $\overline{\|A\|}_2 . \frac{\partial f}{\partial A}$ où f est la fonction de norme 2. Voici la méthode `incrementGradient` associée, qui pourra être réutilisée pour le calcul de tous les types de norme :

```
    //...

    Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic>
       increment;
    increment = this->values[j] *
       childDerivative(0, 0);

    return increment;
}
```

Listing 2.9 – Incrément d'un noeud `ts::ScalarNode`

Pour conclure, ce chapitre nous aura permis de développer un moteur de différentiation automatique complet et à la portée très générale, mais répondant néanmoins aux besoins spécifiques de notre bibliothèque dans le cadre du machine learning. Nous nous sommes en effet concentrés sur l'implémentation du mode backward, qui sera le seul a avoir un intérêt pour nous, et avons généralisé celui-ci aux opérations sur des tenseurs. L'utilisation de la bibliothèque Eigen nous offrira alors de nombreux avantages pour la suite : outre ses très bonnes performances, elle met à notre disposition de nombreuses opérations sur des matrices et tableaux 2D. Cela nous facilitera le travail de développement, et nous permettra comme nous en avons discuté de grandement simplifier nos implémentations des réseaux de neurones et les graphes de calcul qui leurs sont associés. Nous avons donc ici une très bonne base, qui nous servira à calculer de manière rapide et presque transparente les gradients de toutes nos fonctions de coût (pour peu que l'on se donne la peine d'implémenter les fonctions intermédiaires dont on pourrait avoir besoin). Ce chapitre n'a donc présenté que quelques unes des fonctions présentes dans la bibliothèque finale, et le travail qu'il présente a pour vocation d'être enrichi dans la suite de ce rapport, au fur et à mesure que nous aborderons de nouvelles opérations.

Chapitre 3

Implémentation des réseaux de neurones

3.1 Gestion des modèles

Nous avons évoqué au chapitre 1 la notion de modèle, et comment le calcul de leur gradient pouvait permettre leur optimisation. Maintenant que nous avons mis au point les outils de calcul (et de dérivation) nécessaires, nous pouvons passer à leur implémentation dans la bibliothèque, et plus particulièrement à celle des réseaux de neurones qui nous intéressent dans le cadre de ce rapport.

3.1.1 Intérêt de l'approche par modèle

Avant de rentrer dans les détails du code, comprenons bien ce que sera un "modèle" dans la bibliothèque, et quel est l'intérêt d'une telle approche. Nous avons expliqué au chapitre 1 qu'une bibliohèque de machine learning a pour tâche d'utiliser des algorithmes d'optimisation pour ajuster les paramètres d'un grand nombre de modèles (qui sont alors simplement vus comme des fonctions). Afin de combiner facilement n'importe quel type de modèle avec n'importe quel algorithme d'optimisation, il est donc important de bien différencier ces deux notions au niveau du code. Ainsi, il apparaît judicieux de définir un modèle comme étant une classe abstraite (que l'on nommera `ts::Model`) dont on pourra calculer la sortie avec une unique méthode. Cela permettra à nos algorithmes d'optimisation, quand nous les implémenterons au chapitre 4, de travailler de la même manière quel que soit le modèle à optimiser, puisque les calculs seront encapsulés à l'intérieur

de la classe dédiée.

3.1.2 Classe `ts::Model`

Comme nous l'avons dit plus haut, le but de cette classe abstraite sera de permettre l'encapsulation de n'importe quels modèles mathématiques, ainsi que de stocker les paramètres de ces derniers (entre autres pour permettre leur optimisation/ajustement par la suite). La contrainte principale que nous aurons sera de permettre avec une même syntaxe le calcul de tous les types de modèles : en d'autres termes, il nous faudra définir une méthode virtuelle de calcul qui sera utilisée par tous les modèles (qui auront donc la même entrée, à savoir un `ts::Tensor`). Voici la définition de la classe `ts::Model` :

```
template <typename T>
class ts::Model {
private:

public:
    ts::WengertList<T> wList;

    // Call the WengertList toggleOptimize method
    void toggleOptimize(ts::Tensor<T> * tensor,
       bool enable);

    // Helper function to optimize the whole model
    virtual void toggleGlobalOptimize(bool enable)
       = 0;

    // General method for computing the model
       forward pass
    virtual ts::Tensor<T> compute(ts::Tensor<T>
       input) = 0;

    // Serializes / parses model into / from a file
    virtual void save(std::string filePath) = 0;
    virtual void load(std::string filePath) = 0;
};
```

Listing 3.1 – Définition de `ts::Model`

Tout d'abord, un modèle doit disposer de sa propre liste de Wengert. Celle-ci contiendra au moins les noeuds des paramètres d'un modèle, et se remplira lors d'un appel à la méthode `compute`, qui calcule donc la sortie du modèle. Il sera alors conseillé de nettoyer la liste entre chaque appel à `compute` grâce à la méthode correspondante de `ts::WengertList` (on ne supprimera que les noeuds correspondant au résultat et aux calculs intermédiaires, les noeuds "d'entrée" n'ayant pas de dépendance dans la liste seront conservés, puisque ce seront en réalité les paramètres du modèle).

Intéressons-nous maintenant aux méthodes `toggleOptimize`, et à la notion de *noeud optimisable*. On va définir un noeud optimisable comme un noeud d'entrée dont la variable qu'il représente pourra être modifiée par un algorithme d'optimisation.

Remarque : Il est important de noter qu'un noeud optimisable sera toujours un paramètre du modèle et donc un noeud d'entrée (cela n'aurait pas de sens d'optimiser une variable qui ne ferait pas partie du modèle), mais qu'un noeud d'entrée ne sera pas nécessairement optimisable (dans le cas général, on pourrait imaginer avoir besoin d'une variable avec une valeur bien précise, et donc ne pas vouloir que de celle-ci soit modifiée durant la phase d'entraînement).

Ainsi, cette distinction nous amènera à définir un type `ts::InputNode` pour les noeuds d'entrée. Ces derniers seront créés lorsque l'on ajoutera les paramètres de notre modèle dans le constructeur, et disposeront d'un champ `optimizedTensor` de type `ts::Tensor*`. Ce type de noeud fera ainsi exception à la règle que nous avions donné en 2.2, selon laquelle les variables contiennent une référence vers leurs noeuds dans la liste de Wengert, mais pas inversement. En effet, comme nous l'avons souligné, nos algorithmes d'optimisation devront être capables de travailler sur n'importe quel type de modèle. Mais si l'on définit les paramètres comme des champs de la classe, il sera impossible pour les algorithmes d'optimisation d'y accéder pour modifier leurs valeurs. A la place, la liste de Wengert nous fournira une méthode "universelle" d'accès aux paramètres (optimisables) du modèle, tout en nous laissant la liberté de définir les paramètres de nos modèles comme bon nous semble. L'optimisation d'un noeud pourra alors être activée ou désactivée. On propose ainsi dans `ts::Model` les méthodes `toggleOptimize` et `toggleGlobalOptimize` qui permettront respectivement d'activer l'optimisation d'un noeud ou d'un ensemble de noeuds prédéfini (idéalement, cela correspondra à l'ensemble des noeuds que l'on souhaitera rendre optimisables dans une situation réaliste).

Enfin, les méthodes `save` et `load` permettent de serialiser et de parser le modèle, et seront détaillées en partie 3.4.

3.2 Implémentation du MLP

La structure du perceptron multicouche ayant déjà été écrite en 1.2.2, nous ne reviendrons pas dessus dans cette partie. Présentons donc directement le code définissant la classe `ts::MultiLayerPerceptron`, héritant de `ts::Model` :

```
template <typename T>
class ts::MultiLayerPerceptron : public
   ts::Model<T> {
private:

public:
    MultiLayerPerceptron(unsigned inputSize,
       std::vector<unsigned> layers);

    ts::Tensor<T> (*activationFunction)(const
       ts::Tensor<T>&) = &(ts::sigmoid);

    std::vector<ts::Tensor<T>> weights = {};
    std::vector<ts::Tensor<T>> biases = {};

    void toggleGlobalOptimize(bool enable);

    ts::Tensor<T> compute(ts::Tensor<T> input);

    void save(std::string filePath);
    void load(std::string filePath);
};
```

Listing 3.2 – Définition de `ts::MultiLayerPerceptron`

Dans le cas du MLP, les seuls paramètres à stocker sont les poids et les biais de chaque couche. On décide de les stocker dans deux `std::vector` séparés, dont un élément i correspond soit aux poids, soit aux biais de la couche i. Les `std::vector` `weights` et `biases` seront donc normalement de même taille.

Comme on peut le voir dans le prototype du constructeur, les seules informations nécessaires pour initialiser les poids et biais sont les tailles successives de chaque couche. Cette syntaxe permet de créer et de dimensionner des réseaux de manière très simple, le constructeur se chargeant de générer et de remplir aléatoirement les matrices de poids et les vecteurs de biais. Si on souhaite avoir des valeurs initiales bien particulières, rien n'empêchera de les modifier manuellement.

Avant de détailler le calcul du MLP, attardons-nous sur l'initialisation des poids et biais. Dans le constructeur, on dispose des tailles de chaque couche successive. Mettons que celles-ci soient stockées dans un vecteur $l \in \mathbb{N}^n$, avec $l_i \neq 0, \forall i \in \{1, ..., n\}$. On considèrera que l_1 le premier élément de l est la taille de la couche d'entrée (nous avons juste décidé de le mettre à part dans les paramètres du constructeur). Alors, $\forall i \in \{2, ..., n\}$, les poids de la couche i seront de taille $l_i \times l_{i-1}$, et les biais de taille $l_i \times 1$. Dans le constructeur, il suffira donc de parcourir l à partir de l_2 pour générer les vecteurs `weights` et `biases` de taille $n - 1$.

Une fois que les poids et biais sont initialisés, il est possible de calculer la sortie du réseau. Cela se fait assez simplement en reprenant l'équation 1.8, il suffit pour chaque couche H_i :

1. de calculer le produit matriciel des poids w_i et de la sortie de la couche précédente (ou de l'entrée du réseau dans le cas où on calcule la sortie de la première couche cachée)
2. de rajouter les biais b_i
3. de passer le résultat dans la fonction d'activation f_{act} pour obtenir H_i

Formellement, pour un MLP de n couches (sans compter la "couche" d'entrée), l'algorithme de calcul est le suivant :

Algorithme 2 : Calcul de la sortie d'un MLP

Entrée : vecteur d'entrée v, matrices de poids $\{w\}_{i=0}^{n}$, vecteurs de biais $\{b\}_{i=0}^{n}$, fonction f_{act}
Résultat : vecteur H_n

$H_0 \leftarrow v$
pour $i \leftarrow 1$ **à** n **faire**
  $H_i \leftarrow f_{act}(w_i.H_{i-1} + b_i)$
fin

Pour finir sur le modèle du MLP, l'algorithme 2 permettant le calcul de sa sortie correspond à la méthode `ts::MultiLayerPerceptron::compute` :

```
template <typename T>
ts::Tensor<T>
   ts::MultiLayerPerceptron<T>::compute(ts::Tensor<T>
   input) {

    if(
    input.getValue().rows() !=
       weights[0].getValue().cols() ||
    input.getValue().cols() != 1
    ) {
        return ts::Tensor<T>(Eigen::Array<T, 0,
           0>(), NULL);
    }

    if(weights.size() != biases.size()) {
        return ts::Tensor<T>(Eigen::Array<T, 0,
           0>(), NULL);
    }

    for(unsigned i=0; i<weights.size(); i++) {
        if(i < weights.size() - 1) {
            input =
               (*activationFunction)(matProd(weights[i],
               input) + biases[i]);
        }
        else {
            input =
               (*finalActivation)(matProd(weights[i]
               input) + biases[i]);
        }
    }

    return input;
}
```

Listing 3.3 – Méthode de calcul de `ts::MultiLayerPerceptron`

L'algorithme en lui même est donc assez simple, les seules différences sont que dans le code, on s'assure avant de procéder au calcul que les données d'entrée semblent bien avoir les bonnes tailles, et que l'on laisse la possibilité de choisir une fonction d'activation différente pour la dernière couche (typiquement ou pourra utiliser des ReLU dans tout le réseau, et σ sur la dernière couche pour s'assurer que les valeurs sont dans l'intervalle $[0, 1]$).

La classe `ts::MultiLayerPerceptron` constitue ainsi notre première implémentation d'un réseau de neurones. Grâce à notre système de différentiation automatique, il est aussi possible d'effectuer des calculs de gradient sur ses instances d'une manière extrêmement simple. Cela permettra (comme nous le verrons au chapitre 4 portant sur l'optimisation) de mettre en place des phases d'entraînement pour ajuster les paramètres, et donc les prédictions de nos réseaux.

Remarque : Pour un exemple d'application du MLP (sur le problème MNIST), il est possible de consulter directement la section 5.1.1. La lecture du chapitre 4 est cependant conseillée.

3.3 Réseaux neuronaux convolutifs (CNN)

Nous nous sommes jusqu'à présent concentrés sur le modèle du MLP, qui en plus de constituer un très bon exemple introductif, permet malgré sa simplicité de résoudre un ensemble de problèmes dans des domaines très variés. Il existe cependant de nombreuses architectures de réseaux neuronaux, et donc autant de choix pour en présenter une nouvelle. On décide dans ce rapport de continuer les travaux réalisés sur le MLP en s'intéressant aux réseaux neuronaux convolutifs (*Convolutional Neural Networks*, ou CNN). En partie 5.1.1, on donne un exemple d'application du MLP sur une problématique de classification d'images (qui aujourd'hui reposent énormément sur les réseaux de neurones, et nous allons le voir, sur les CNN en particulier). MNIST est cependant un des exemples les plus simples qui soient, et sur des applications de reconnaissance plus "réalistes" ou tout du moins plus complexes, cette structure de réseau pourra montrer ses limites. Sa nature fortement connectée la rend notamment victime de problèmes de *surapprentissage* (ou *overfitting*) [12]. Les CNN y sont beaucoup moins sujets et sont donc aujourd'hui une solution classique pour résoudre efficacement des problématiques de reconnaissance d'image. L'intuition est d'utiliser des *couches de convolution*, afin de simplifier une image en un ensemble de features abstraites, puis

de passer la sortie des couches de convolution dans des couches fortement connectées à la manière d'un MLP pour travailler sur l'image simplifiée.

Avant d'étudier plus en détail la construction de ces réseaux en partie 3.3.4, intéressons-nous à l'opération de convolution dont les CNN tirent leur nom.

3.3.1 Opération de convolution

L'opération de convolution a une définition très générale, qui s'applique à l'origine sur deux fonctions. Formellement, si f et g sont deux fonctions à valeur réelle, alors leur convolution est une troisième fonction définie comme [13] :

$$(f \star g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{3.1}$$

De manière plus intuitive, cette opération mesurera la similitude qu'il existe entre f et g. Ainsi, en faisant "glisser" la fonction g (en faisant varier τ dans 3.1), on sera en mesure d'évaluer en différents points la similitude des deux fonctions. g agit alors de manière similaire à un filtre (c'est un terme qui est d'ailleurs assez intuitif, et donc souvent repris).

Dans notre contexte, à savoir la reconnaissance d'image, cette fonction devra être adaptée. Plus précisément, plutôt que de travailler sur des fonctions continues, on travaillera sur des matrices (que l'on pourrait assimiler à des fonctions de $\mathbb{N}^2 \to \mathbb{R}$). Ainsi, f correspondra à une matrice de base M, de dimensions $(M_x \times M_y)$. Le filtre g sera une deuxième matrice que l'on appellera k (le terme *kernel* de convolution sera retenu dans la suite) de dimensions $(k_x \times k_y)$, avec $k_x \leq M_x$ et $k_y \leq M_y$. En effet, plutôt que de faire une calcul d'intégral, on va simplement faire glisser k sur toutes les positions possibles dans M. Par *positions possibles*, on entend tous les (i, j) tels que :

$$\begin{cases} i + k_x - 1 \leq M_x \\ j + k_y - 1 \leq M_y \end{cases} \tag{3.2}$$

Autrement dit, les (i, j) correspondent aux coordonnées d'un coin supérieur gauche de la matrice k à partir duquel k sera contenue dans M :

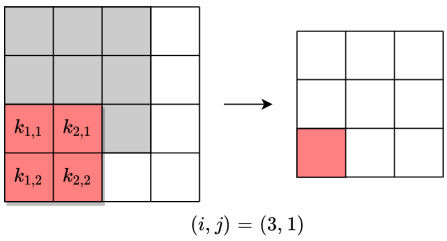

FIGURE 3.1 – Convolution de matrices

La figure 3.1 donne un exemple avec M de taille 4×4 et k de taille 2×2. Les cases grises représentent les positions (i, j) possibles, et les cases rouges indiquent les éléments de k lorsque $(i, j) = (3, 1)$. $(3, 1)$ est donc une position possible, ce qui ne serait pas le cas de $(4, 1)$ par exemple. Enfin, la matrice de droite est le résultat de la convolution, sur laquelle on a indiqué le coefficient correspondant à la sous matrice de coordonnées $(3, 1)$.

Pour chacun des couples (i, j) possibles, on calculera la somme des éléments du produit terme à terme ente k et la sous-matrice de M correspondant aux éléments de même positions que k (donc aussi marqués en rouge sur la figure). Ainsi, toute la matrice marquée en rouge ne donnera qu'un seul élément de la matrice de sortie. Formellement, en se basant sur l'équation 3.2, les valeurs possibles de i sont $\{1, ..., M_x - k_x + 1\}$ et celles de j sont $\{1, ..., M_y - k_y + 1\}$. Par conséquent, la matrice résultant de la convolution de M par k sera de taille $(M_x - k_x + 1 \times M_y - k_y + 1)$. On définira alors la convolution de M par k comme la matrice $M \star k = C$ dont les coefficients $C_{i,j}$ sont, $\forall i \in \{1, ..., M_x\}$ et $\forall j \in \{1, ..., M_y\}$:

$$C_{i,j} = \sum_{i'=0}^{k_x-1} \sum_{j'=0}^{k_y-1} M_{(i+i',j+j')}.k_{(i'+1,j'+1)} \tag{3.3}$$

On parlera alors de *convolution matricielle* [14]. On remarque la similitude de cette définition avec l'équation 3.1. Intuitivement, on peut imaginer que l'on va faire glisser un filtre permettant de détecter un motif (par exemple des boucles ou des lignes droites dans MNIST), et qu'à partir de la position de ces motifs, on sera en mesure de classifier notre image (deviner quel chiffre est représenté à partir des positions des boucles et lignes droites). Evidemment, ce n'est pas comme ça que procédera un réseau entraîné, et il sera souvent pratiquement impossible de donner une "explication" à la forme que vont prendre les filtres de convolution quand ceux-ci seront optimisés.

Cependant, cette idée permet de se représenter l'intérêt de la convolution dans de la classification d'image : simplifier les données sur lesquelles on travaille en détectant un ensemble de features (à partir desquelles on pourra extraire d'autres features de niveau encore plus élevé), plutôt que d'utiliser directement des couches fortement connectées sur l'image d'entrée.

3.3.2 Convolution multicanal

L'opération de convolution que nous venons de définir fonctionne parfaitement, mais a le défaut de ne pouvoir utiliser qu'un seul kernel de convolution, et donc de ne pouvoir détecter qu'un seul type de features sur l'image d'entrée. Dans une application réaliste, il sera nécessaire d'extraire plusieurs types de features d'une seule image. Par conséquent il sera nécessaire de réaliser plusieurs convolutions sur une même matrice de base, et l'on obtiendra alors plusieurs matrices de convolution. De plus, nous avons évoqué en début de partie 3.3 que nous aurons le besoin d'utiliser plusieurs couches de convolution consécutives, et donc de combiner plusieurs opérations de convolution à la suite. Cela nous amène à définir dans cette partie les notions de *canal* (ou *channel*) et de *convolution multicanal*[15].

Cette opération permettra de travailler simultanément sur plusieurs matrices de base, auxquelles on appliquera plusieurs kernels. Ainsi, une convolution multicanal se fera sur un certain nombre de channels d'entrée et de sortie. Si on a c_i channels d'entrée, et c_o channels de sortie, alors on utilisera $c_i.c_o$ kernels sur c_i matrices de base, pour obtenir c_o matrices en sortie. On repèrera alors les matrices d'entrées par un indice dans $\{1, ..., c_i\}$, les kernels par deux indices dans $\{1, ..., c_o\}$ et $\{1, ..., c_i\}$, et les matrices par un indice dans $\{1, ..., c_o\}$. Alors, $\forall j \in \{1, ..., c_o\}$, les matrices du channel de sortie seront :

$$C_j = \sum_{i=1}^{c_i} M_i \star k_{j,i} \tag{3.4}$$

Graphiquement, on peut représenter cette opération comme dans le schéma ci-dessous. Dans celui-ci, on effectue l'opération avec 3 channels d'entrée (par exemple pour une image RGB) et 2 channels de sortie. Les matrices résultant des convolutions avec les kernels gris seront sommées pour obtenir la matrice de sortie grise, le principe étant le même pour les matrices oranges. Les couleurs des bordures des kernels indiquent sur quelle matrice de base ils seront apppliqués :

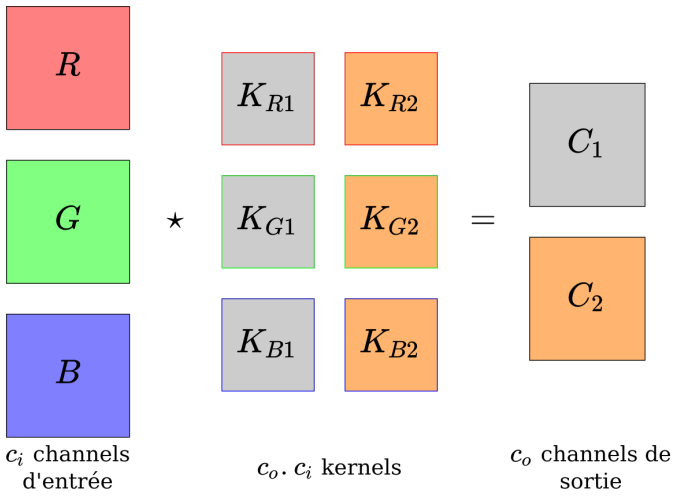

FIGURE 3.2 – Convolution multicanal

Cette opération nous servira directement dans les couches de convolution des CNN, dont nous verrons la strucuture détaillée dans la partie suivante.

Remarque : On pourrait utiliser une notation avec des tenseurs pour les convolutions multicanal. Ainsi, les channels d'entrée seraient stockés dans un tenseur de dimension $c_i \times M_x \times M_y$, et les kernels dans un tenseur de dimension $c_o \times c_i \times k_x \times k_y$. La sortie de l'opération serait alors de dimension $c_o \times (M_x - k_x + 1) \times (M_y - k_y + 1)$. Cependant, on a choisi dans notre bibliothèque de se limiter à des tenseurs d'ordre 2 représentés par des `Eigen::Array`, ce qui rend cette façon de voir les données difficiles à implémenter. La partie suivante se concentrera cependant sur une optimisation des convolutions multicanal permettant de représenter les kernels et l'image d'entrée comme de simples matrices.

3.3.3 Méthode im2col

La façon dont nous avons présenté le calcul d'une opération de convolution (y compris multicanal) est la plus intuitive, et donc probablement la plus intéressante à aborder dans un premier temps. Cependant, elle a le désavantage d'être très couteuse en terme de temps à cause de la manière dont on accède aux données. Nous présenterons donc dans cette partie la méthode "im2col" (pour *image-to-column*). Son principe est de calculer une convolution multicanal en ré-arrangeant les données pour ramener

l'opération à un produit matriciel. Même si les données écrites sous cette forme seront plus volumineuses, le fait de calculer un produit de matrice permettra une bien meilleure vectorisation du code. Cela compensera largement l'augmentation de la taille des données et le temps passé à les manipuler [16].

L'intuition de cette méthode est donc d'organiser différement les coefficients non seulement des matrices d'entrée d'une convolution multicanal, mais aussi des kernels de convolution. Cela nécessitera d'introduire de la redondance, mais permettra d'écrire l'opération comme un produit de matrices. Les kernels seront toujours de la même forme, et il suffira donc de les générer avec la forme adéquate dès le départ. Cependant, il sera nécessaire à l'entrée de chaque couche de convolution de redimensionner nos matrices d'entrée de manière appropriée. C'est cette opération préliminaire qui porte le nom de im2col. En voici une représentation graphique pour 3 channels d'entrée (similaire à la figure 3.2) et pour des kernels de taille 2×2 : [17]

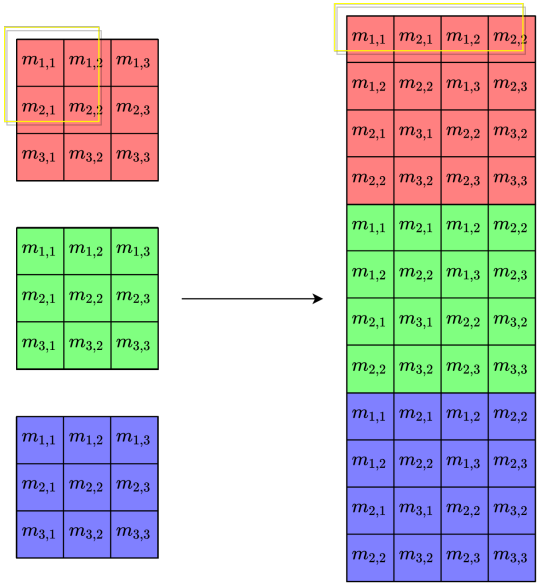

FIGURE 3.3 – Transformation im2col

Comme on peut le voir, le principe de im2col est de considérer chacune des sous matrices auxquelles on appliquerait un filtre dans une convolution "classique", et de convertir celles-ci en vecteurs lignes. Cela est fait

en column-major sur le schéma, mais il ne s'agit pas d'une obligation. Ces vecteurs seront ensuite combinés verticalement pour former une nouvelle matrice regroupant tous les channels. Soit M cette nouvelle matrice. Si la taille des i matrices de base est $m_x \times m_y$ et si celle des kernels est $k_x \times k_y$, alors M sera de taille $ik_xk_y \times (m_x - k_x + 1)(m_y - k_y + 1)$

Les kernels devront donc être agencés de manière à admettre un produit matriciel avec M. Dans la pratique, on génèrera directement nos kernels sous cette forme, ce qui ne posera pas de problème. Voici à quoi ressemblera la matrice de kernels comparé à l'agencement "classique" :

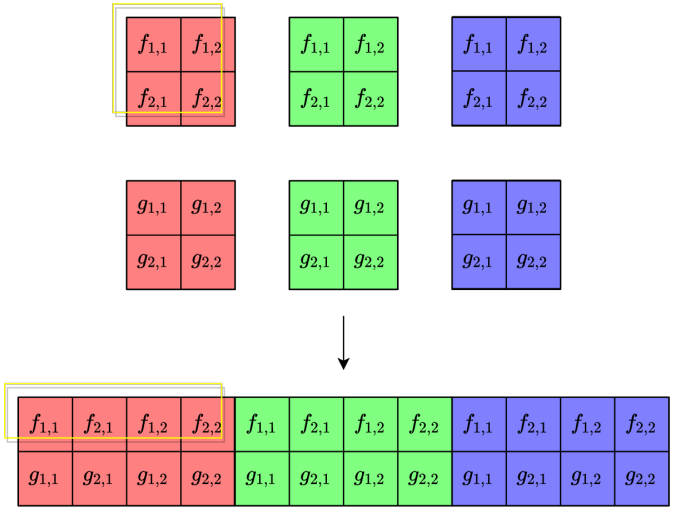

FIGURE 3.4 – Kernels pour une convolution im2col

Pour une couche de convolution de i channels d'entrée et de o channels de sortie utilisant des kernels de dimension $k_x \times k_y$, il suffira donc simplement de générer une matrice de taille $o \times k_xk_yi$. Cette matrice sera notée K.

Maintenant que nous disposons de la matrice de base M et de la matrice de kernels K, il suffit de calculer le produit matriciel pour obtenir le résultat de la convolution. Ce dernier sera noté $C = K.M$ et sera de taille $o \times (m_x - k_x + 1)(m_y - k_y + 1)$. Il suffira alors de redimensionner M en o matrices $m_x - k_x + 1 \times m_y - k_y + 1$ pour obtenir le résultat de la convolution sous la forme "classique". Il s'agit en quelques sortes de l'opération inverse de im2col, et porte donc parfois le nom de col2im. Cette opération ne comporte pas

de difficulté majeure : chaque ligne sera redimensionnée pour correspondre à une matrice de sortie, il faudra seulement prêter attention à faire cela de la même manière que dans im2col : si les sous-matrices avaient été ajoutées dans l'ordre column-major (comme en figure 3.3), il faudra considérer les lignes de M comme une matrice column-major.

L'opération col2im nous permet ainsi d'obtenir nos matrices de sortie comme si la convolution avait été calculée de manière naïve. Cela nous permet de les utiliser dans d'autres types d'opérations, y compris dans les im2col des prochaines couches de convolution. Nous disposons ainsi d'une manière optimisée de calculer des convolutions multicanal, ce qui va nous permettre de nous intéresser à l'implémentation des réseaux à convolution en eux-mêmes.

Remarque : Pour une comparaison des temps de calculs entre l'approche classique et im2col, consulter la partie 5.2.4.

3.3.4 Structure et implémentation des CNN

Comme expliqué en début de partie 3.3, le principe d'un CNN sera d'utiliser des opérations de convolution avant les couches fortement connectées. Cependant, les convolutions multicanal de la première partie du réseau sont combinées avec plusieurs autres opérations. Voici une représentation graphique des différents types de couche qui composent un CNN :

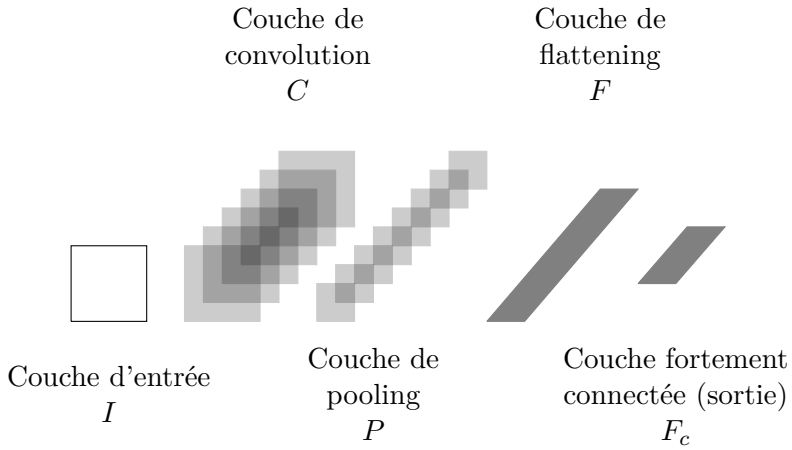

FIGURE 3.5 – Schéma d'un CNN

Voici une explication des opérations successives que l'on trouve dans un réseau à convolution :

— couche de convolution C : il s'agit d'une convolution multicanal comme vue en parties 3.3.2 et 3.3.3. Comme dans les couches fortement connectées d'un MLP, on ajoutera des biais aux résultats de la convolution, et on appliquera une fonction d'activation ou de rectification (typiquement ReLU, les fonctions d'activation comme σ ayant tendance à saturer dans ce type de couches).
— couche de pooling P : cette opération consiste à diviser une matrice en plusieurs petits "pools", ou plus simplement en sous-matrices disjointes (cela est donc différent des sous-matrices considérées dans une convolution). Pour chacun des pools, on va sélectionner le maximum et le conserver dans la matrice résultante. Cette opération porte le nom de "max pooling", mais il existe des variantes comme le "mean pooling" consistant à prendre la moyenne des coefficients de chaque pool. Si les pools sont de taille $p_x \times p_y$, une matrice de taille $m_x \times m_y$ sera alors redimensionnée en une matrice $\frac{m_x}{p_x} \times \frac{m_y}{p_y}$ après une opération de pooling. A moins de rajouter du padding, m_x et m_y doivent donc être divisibles respectivement par p_x et p_y. Ces couches ont pour utilité de réduire le volume de données en ne conservant que les features les plus significatives.
— couche de flattening F : il s'agit simplement d'une opération consistant à redimensionner les matrices des couches de convolution en un unique vecteur colonne. Le but de cette couche est de préparer les données aux couches fortement connectées.
— couche de fortement connectée F : enfin, on entre dans les couches fortement connectées en utilisant comme entrée le résultat de la couche de flattening. Ces couches sont exactement les mêmes que celles d'un perceptron multicouche.

Remarque : Le schéma ci-dessus ne présente qu'une seule couche de convolution et une seule couche de pooling. Dans la pratique, il est courant d'alterner successivement les couches de convolution puis de pooling. Cependant, il est aussi possible de sauter une ou plusieurs couches de pooling pour combiner directement plusieurs convolutions à la suite. La couche de flattening sera toujours unique, et on pourra mettre autant de couches fortement connectées qu'on le souhaite.

L'implémentation des CNN demande passe donc par celle de nombreuses opérations de redimensionnement (im2col et col2im évidemment, mais aussi

le flattening, la séparation de matrices, ...). Ces dernières ne sont pas nécessairement très complexes et nous ne nous attarderons pas dessus. A la place, voici directement le prototype du constructeur de la classe `ts::ConvolutionalNetwork` afin de comprendre la construction des CNN dans la bibliothèque :

```
ConvolutionalNetwork(
    std::vector<unsigned> inputSize,
    ChannelSplit splitDirection,
    unsigned inputChannels,
    std::vector<std::vector<unsigned>> convLayers,
    std::vector<std::vector<unsigned>>
       poolingLayers,
    std::vector<unsigned> denseLayers
);
```

Listing 3.4 – Prototype du constructeur de `ts::ConvolutionalNetwork`

Ici, `inputSize` est un vecteur de taille 2 indiquant les dimensions de la matrice d'entrée. Cette dernière pourra être séparée en un nombre `inputChannels` de channels de taille égale selon la valeur de `splitDirection` (il s'agit d'un type enum). Une fois les channels obtenus, les couches de convolution et de pooling sont décrites par `convLayers` et `poolingLayers` respectivement (les éléments de `convLayers` donnent la taille des kernels et le nombre de channels de sortie de la couche, et les éléments de `poolingLayers` donnent la taille des pools à former). Enfin, `denseLayers` décrit la structure des couches fortement connectées comme dans un MLP.

Maintenant, intéressons-nous à la fonction de calcul de `ts::ConvolutionalNetwork`. Cette dernière étant relativement longue, on va s'intéresser uniquement à sa partie la plus importante, à s'avoir l'évaluation des couches de convolution. Le code suivant est ainsi extrait de la méthode `compute` :

```
// 1) Convolution / pooling computation loop
for(unsigned i=0; i<convKernels.size(); i++) {
    // Compute the im2col multichannel convolution
    input = ts::im2col(inputVec, kernelDims[i]);
    input =
       (*convActivation)(matProd(convKernels[i],
       input) + convBiases[i]);
    inputVec = ts::col2im(input,  outputDims[i]);

    // A pooling layer of size 0 means we want to
       skip it
    if(pooling[i][0] != 0 || pooling[i][1] != 0) {
        for(unsigned j=0; j<inputVec.size(); j++) {
            inputVec[j] =
               ts::maxPooling(inputVec[j],
               pooling[i]);
        }
    }
}
```

Listing 3.5 – Calcul des couches de convolution

Au début de cette boucle, on dispose des channels d'entrée dans le vecteur `inputVec`. Les dimensions des kernels auront été stockées dans le vecteur `kernelDims`, ce qui nous permet de procéder à une opération im2col donnant une matrice dont les dimensions permettront le produit matriciel avec les kernels. De la même manière, on a enregistré les dimensions des sorties de chaque couche dans `outputDims`, ce qui permet de faire appel à col2im en obtenant des matrices de la taille souhaitée. On remarque que par souci de simplicité, la fonction d'activation et les biais son directement appliqués sur la matrice résultant du produit entre les `convKernels[i]` et `input`. Cela évite de multiplier le nombre d'opérations et donc les insertions de noeuds dans le graphe de calcul. Enfin, on passe à la couche de pooling. Par convention, si les dimensions de celles-ci sont $\{0, 0\}$, cela veut dire que l'on souhaite passer l'étape de pooling. Si les dimensions sont non nulles, on doit cependant procéder à l'opération de max-pooling sur chacun des channels. En effet, il n'est pas possible de l'appliquer directement après le produit matriciel comme auparavant, car cette opération repose sur la localité des coefficients dans la matrice (ou alors il faudrait développer une fonction adaptée aux convolutions im2col, ce qui empêcherait cependant de l'utiliser

dans un autre contexte).

Remarque : Pour un exemple d'application de CNN, il est possible de consulter directement la section 5.1.2. La lecture du chapitre 4 portant sur l'optimisation est cependant conseillée.

3.4 Sauvegarde d'un modèle

Nous avons jusqu'à présent détaillé le fonctionnement de la classe `ts::Model`, ainsi que les structures de différents types de réseaux de neurones qui ont été implémentés à l'aide de cette classe. Cependant, une fois que les valeurs d'un modèle ont été précisément ajustées, par exemple à la suite d'une phase d'entraînement, il peut être intéressant de sauvegarder ces dernières dans un fichier externe. En effet, il est difficilement envisageable de procéder de nouveau à la phase d'entraînement à chaque utilisation d'un modèle, cette étape pouvant être assez longue. Le besoin de développer un système permettant de serialiser / parser un modèle pour une utilisation ultérieure apparaît ainsi évident.

Evidemment, chaque type de modèle aura des paramètres différents, et chaque instance d'un même type aura des dimensions bien particulières (nombre de couches, taille de ces couches, ...). Le but du système de sauvegarde sera ainsi de proposer des fonctions à la portée générale et permettant, pour chaque modèle, d'écrire les fonctions de parsing et de serialization de manière très simplifiée.

Plus précisément, le principe de ce système reposera sur des `std::ofstream` et `std::ifstream` afin de lire et écrire les valeurs du modèle dans des fichiers externes. Pour parser un `ts::Tensor`, on va réutiliser l'opérateur `<<` des `Eigen::Array` encapsulés. La sortie de cet opérateur sera légèrement modifiée afin de replacer toutes les valeurs sur une même ligne. Sans rentrer dans les détails, la fonction `ts::serializeTensor(ts::Tensor<T> &tensor)` prendra en entrée une référence d'un tenseur et retournera une chaîne de caractères de trois lignes indiquant successivement le nombre de lignes, le nombre de colonnes, et les coefficients. Inversement, la fonction `ts::Tensor<T> ts::parseTensor(std::ifstream &in, ts::WengertList<T> * wList)` lira les trois premières lignes d'un `std::ifstream` et essayera de reconstruire un `ts::Tensor` gràce à l'opérateur `>>` des `Eigen::Array`, puis de l'ajouter à la liste de Wengert. Ces fonctions pourront elles mêmes être combinées pour sauvegarder ou lire en une seule fois un vecteur de

`ts::Tensor`. C'est ce que font les fonctions `ts::serializeTensorsVector` et `ts::parseTensorsVector`, dont nous allons voir un exemple d'utilisation.

Remarque : Il aurait été possible de sauvegarder et de lire les modèles directement avec leurs valeurs au format binaire. Cela serait bien plus efficace, mais auait cependant un désavantage majeur : en fonction de la machine utilisée pour lire un fichier obtenu avec cette méthode, restaurer le modèle ne serait pas forcément possible. Le but étant aussi de pouvoir partager un modèle, on préfèrera travailler avec des chaînes de caractère.

Ces fonctions seront appelées dans les métohdes `save` et `load` de `ts::Model`. Afin d'illustrer leur utilisation, reprenons l'exemple du perceptron multicouche. Comme vu en partie 3.2, ce dernier possède un ensemble de poids et de biais, qui sont stockés séparément sous forme de 2 vecteurs de même taille. Voici comment, en quelques lignes, il est possible de serialiser l'ensemble de ces coefficients (quels que soit le nombre de couches ou leurs dimensions) :

```
1 template <typename T>
2 void ts::MultiLayerPerceptron<T>::save(std::string
     filePath) {
3     std::ofstream out(filePath);
4     out << ts::serializeTensorsVector(weights);
5     out << ts::serializeTensorsVector(biases);
6     out.close();
7 }
```

Listing 3.6 – Serialisation d'un `ts::MultilayerPerceptron`

Il suffit comme on peut le voir de renseigner un chemin pour la sauvegarde (sous forme de chaîne de caractère), puis deux simples appels de fonction sur les `weights` et `biases` permettent de les sauvegarder, en redirigeant la sortie vers le `std::ofstream`. La fonction de parsing est assez similaire :

```cpp
template <typename T>
void ts::MultiLayerPerceptron<T>::load(std::string
    filePath) {
    weights = {};
    biases = {};
    this->wList.reset();

    std::ifstream in(filePath);
    weights = ts::parseTensorsVector(in,
        &(this->wList));
    biases = ts::parseTensorsVector(in,
        &(this->wList));
    in.close();
}
```

Listing 3.7 – Parsing d'un `ts::MultilayerPerceptron`

Ici, on réinitialise les poids et biais, puis la liste de Wengert pour s'assurer que le modèle sera bien vide au moment de réécrire par dessus. Pour le reste, le fonctionnement est assez similaire puisque l'on se contente encore une fois d'appeler nos fonctions générales sur les champs qui nous intéressent dans ce cas précis. L'idée de cet exemple est de montrer qu'un utilisateur qui voudrait implémenter son propre modèle pourrait très facilement écrire les fonctions `save` et `load` correspondantes à l'aide des outils qui lui sont fournis.

Pour conclure sur la classe `ts::Model`, cette dernière remplit son objectif qui était de proposer une base à la portée très générale pour l'implémentation de tous types de modèles. Cette classe nous permettra ainsi facilement de définir des paramètres, de les sauvegarder et de les charger, ainsi que de calculer la sortie du modèle à partir ce ces derniers. Chaque modèle sera ainsi compatible avec le reste de notre bibliothèque, que ce soit avec le moteur de différentiation automatique, ou avec le système d'optimisation qui sera détaillé au chapitre suivant.

Chapitre 4

Algorithmes d'optimisation

Dans le chapitre précédent, nous avons détaillé différentes structures de réseaux de neurones, et avons mis en place un système permettant d'implémenter facilement de nouveaux types de modèle. Ce chapitre sera dédié à la phase d'entraînement qui a été évoquée au chapitre 1. Le but sera ainsi d'exploiter le mécanisme de calcul de gradient du chapitre 2 afin d'ajuster et d'optimiser les paramètres de nos modèles. Comme cela a été dit, il existe plusieurs algorithmes d'optimisations remplissant ce rôle. Dans un premier temps, nous en présenterons deux afin de comprendre leur fonctionnement, tout en détaillant la manière dont sont implémentés ces algorithmes dans la bibliothèque. Le but est à la fin de pouvoir optimiser n'importe quel modèle à l'aide de n'importe quelle méthode d'optimisation, et ce sans nécessiter de travail d'adaptation particulier.

4.1 Descente stochastique de gradient

4.1.1 Principe de l'algorithme

L'algorithme d'optimisation le plus simple est probablement celui de la descente stochastique de gradient (ou SGD, pour *Stochastic Gradient Descent*). Les algorithmes plus avancés reprennent le même principe de descente de gradient, et il constitue donc une excellente introduction aux algorithmes d'optimisation. [2, p. 73]

Avant de rentrer dans les détails de cet algorithme, rappelons rapidement le déroulement de la phase d'apprentissage. Le principe de cette dernière sera de calculer tout simplement la sortie du modèle, puis, en effectuant la différence entre celle-ci et la sortie attendue, d'exprimer la valeur de la

fonction de coût. L'idée sera alors de calculer le gradient de cette dernière afin de savoir dans quel sens (à l'opposé du gradient), et à quel point modifier les paramètres du modèle pour affiner sa sortie. On répète ensuite ce processus de nombreuses fois et sur différents exemples jusqu'à ce que la fonction de coût converge vers un minimum. La figure suivante donne une représentation graphique de l'intuition de l'algorithme :

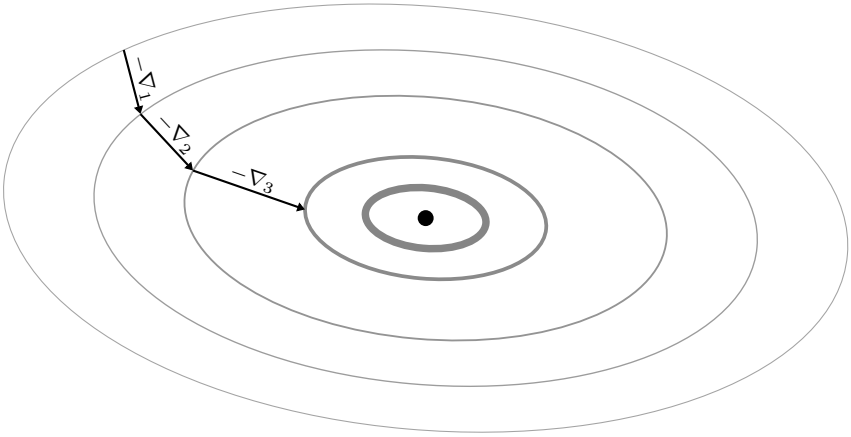

FIGURE 4.1 – Représentation de la descente de gradient

La question est donc de savoir de combien exactement il faut ajuster les paramètres. La solution très simple proposée par la descente stochastique de gradient est de définir un *taux d'apprentissage* η, qui est une valeur scalaire par laquelle on multipliera les éléments du gradient pour obtenir l'incrément de chaque paramètre. Ainsi, lorsque l'on cherche à optimiser une fonction f à l'aide d'une entrée x, ses paramètres θ_t après t étapes sont donnés par :

$$\theta_t = \theta_{t-1} - \eta.\nabla f_t(\theta_{t-1}, x) \tag{4.1}$$

Cependant, ces incréments ne seront pas appliqués (en tout cas dans le cas général) pour chaque calcul de gradient. Cela est dû à plusieurs facteurs, mais notamment au fait que se baser sur un unique exemple pour mettre à jour les valeurs de notre modèle le rendrait sensible au "bruit", c'est à dire aux particularités d'une seule image qui ne refléteraient pas une caractéristique récurrente dans notre jeu de données. A la place, on organisera nos données en "batches", qui sont un petit ensemble d'instances de ces données. Ainsi, l'incrément sera donné par la moyenne des gradients

d'un batch, toujours multipliée par le taux d'apprentissage. En reprenant l'équation 4.1, et pour un batch b, alors les paramètres θ_t après t batches seront :

$$\theta_t = \theta_{t-1} - \eta \frac{\sum_{i=1}^{|b|} \nabla f_t(\theta_{t-1}, b_i)}{|b|} \tag{4.2}$$

Le principe de l'algorithme sera dont d'appliquer cette formule à l'ensemble des batches constituant notre jeu de données. En plus de cela, on pourra définir un nombre *d'epochs*, qui correspondra au nombre de fois que l'on répètera le procédé. Ainsi, l'algorithme SGD s'écrit de la manière suivante :

Algorithme 3 : Descente stochastique de gradient

Entrée : fonction f, paramètres θ_0, ensemble de batches B, nombre d'epochs e, taux d'apprentissage η
Résultat : paramètres θ_t

```
pour i ← 1 à e faire
    pour t ← 1 à |B| faire
        b ← B_t
        ∇_sum ← {0}_{k=1}^{|θ_0|}
        pour j ← 1 à |b| faire
            ∇_sum ← ∇_sum + ∇f_t(θ_{t-1}, b_j)
        fin
        θ_t ← θ_{t-1} − η ∇_sum / |b|
    fin
fin
```

La structure en trois boucles de cet algoritmhe (respectivement pour les epochs, baches, et instances de données d'entrée) sera à la base de tous les algorithmes d'optimisation. La variable ∇_{sum}, qui sert à stocker la somme des gradients d'un batch a quant à elle une grande importance dans l'algorithme et correspond à ce que l'on appelle un *accumulateur de gradient*. Il s'agit d'une notion qui sera au coeur de notre implémentation en partie 4.1.2.

Pour terminer sur l'algorithme SGD, le taux d'apprentissage peut prendre de nombreuses valeurs selon le contexte, mais sera souvent proche de 0 (de l'ordre de 0.01 ou 0.001 par exemple). Il est important de l'initialiser intelligement pour maximiser l'efficacité de la phase d'entraînement : si il est trop

faible la fonction de coût convergera très lentement, mais si il est trop élevé on risque d'osciller autour d'un minimum local sans jamais l'approcher de suffisament près.

Remarque : L'algorithme présenté ci-dessus est assez général dans la mesure où il intègre la notion de batch (cette méthode en particulier est parfois appelée *mini-batches*). Il est cependant possible de l'utiliser "sans batch" en mettant leur taille à 1. A l'inverse, une autre approche possible est de ne constituer qu'un seul batch de taille conséquente et de ne mettre à jour le gradient qu'une seule fois par epoch. On préfèrera alors choisir un nombre d'epochs plus élevé.

4.1.2 Classe `ts::Optimizer` et implémentation

La principale difficulté dans l'implémentation des algorithmes de descente de gradient dans la bibliothèque sera l'accumulateur de gradient. En effet, la seule manière que nous avons d'accéder aux paramètres du modèle est la liste de Wengert. Or tous les éléments de cette liste ne sont pas forcément optimisables ni des variables d'entrée, et ne devraient donc pas être stockés dans l'accumulateur de gradient. Cela va donc nous amener à travailler avec deux systèmes d'indices différents pour repérer nos paramètres : un pour le gradient ou la liste de Wengert (puisque ces derniers sont deux vecteurs de taille égale ou chaque élément a le même indice), et un pour l'accumulateur de gradient. La différence sera que le systèmes d'indices de l'accumulateur de gradient ne comportera pas les paramètres non-optimisables ou les noeuds intermédiaires, on conservera cependant l'ordre des éléments du fait que la liste de Wengert sera lue dans l'odre pour la création de l'accumulateur.

Cela nous amène à définir dans le code les classes `ts::GaElement` et `ts::GradientAccumulator`. Un `ts::GaElement` correspondra alors à une variable optimisable d'un modèle et contiendra la somme de ses gradients, anisi que son indice dans la liste de Wengert. La classe `ts::GradientAccumulator` contiendra alors un vecteur de `ts::GaElement`. Voici son constructeur, qui à partir d'un modèle, se charge de construire ce vecteur :

```cpp
template <typename T>
ts::GradientAccumulator<T>::GradientAccumulator(ts::Model<T>
    &model) {

    // Reset wengertList in case it has been used
       before
    model.wList.reset();

    for(unsigned i=0; i<model.wList.nodes.size();
       i++) {

        std::shared_ptr<ts::InputNode<T>> inputPtr =
        std::static_pointer_cast<ts::InputNode<T>>(
            model.wList.nodes[i]
        );

        // Check if it is associated with a tensor
           (== optimizable)
        if(inputPtr->optimizedTensor != NULL) {

            // Then append it to the gradient
               accumulator
            elements.push_back(ts::GaElement<T>(
                inputPtr->optimizedTensor
            ));
        }
    }
}
```

Listing 4.1 – Constructeur de `ts::GradientAccumulator`

On s'assure au début que la liste de Wengert du modèle ne contient que des noeuds d'entrée (au cas où celle-ci aurait déjà servi pour des calculs). Puis, on regarde pour chacun des noeuds d'entrée restants si ils sont optimisables (c'est à dire qu'ils on une référence vers un `ts::Tensor`). Si c'est le cas, un `ts::GaElement` sera créé pour lui.

Une fois que l'accumulateur de gradient est créé et initialisé à 0, il faut être capable de l'incrémenter. Cela se fait très simplement, et c'est ici que l'on fait appel aux deux systèmes d'indices :

```cpp
template <typename T>
void
   ts::GradientAccumulator<T>::increment(ts::Gradient<T>
   &gradient) {
    // Increment all elements of gradAccumulator
       according to gradient

    for(unsigned i=0; i<elements.size(); i++) {
        // We use two different indices systems here
        // (one for the wList/grad and one for the
           gradient accumulator)
        elements[i].gradSum +=
           gradient.derivatives[elements[i].index];
    }
}
```

Listing 4.2 – Constructeur de `ts::GradientAccumulator`

Plutôt que de faire une itération sur tout le gradient, on va plutôt se concentrer uniquement sur les éléments qui sont optimisables, et parcourir les éléments de l'accumulateur de gradient. Pour chacun d'eux, on peut accéder au gradient correspondant avec l'indice `elements[i].index`, et procéder à l'incrément (il s'agit d'une simple addition terme à terme sur des `Eigen::Array`).

Maintenant que nous disposons d'un système capable de stocker les gradients des paramètres optimisables, nous pouvons définir la classe `ts::Optimizer`. Cette dernière est une classe abstraite qui dispose d'une méthode `run` prenant en paramètre un `ts::Model` à optimiser, et un jeu de données d'entraînement. Ce jeu de donnés doit contenir pour chaque instance les entrées du modèle et la sortie attendue. Pour simplifier la manipulation de ces données, on définit la classe `ts::TrainingData` :

```
template <typename T>
class ts::TrainingData {
private:

public:
    TrainingData(
        Eigen::Array<T, Eigen::Dynamic,
           Eigen::Dynamic> newInput,
        Eigen::Array<T, Eigen::Dynamic,
           Eigen::Dynamic> newExpected
    );

    Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic>
       input;
    Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic>
       expected;
};
```

Listing 4.3 – Constructeur de `ts::TrainingData`

La méthode `run` de `ts::Optimizer` aura ainsi besoin d'un vecteur 2D de `ts::TrainingData` qui correspondra à l'organisation en batch de l'algorithme 3. Cette méthode reprend donc la structure de cet algorithme. Sans donner le code en entier, voici la partie correspondant au calcul pour une instance des données dans la classe `ts::GradientDescentOptimizer` correspondant à l'algorithme SGD :

```
ts::Tensor<T> input = ts::Tensor<T>(
    batches[j][k].input, &(model.wList)
);
ts::Tensor<T> expected = ts::Tensor<T>(
    batches[j][k].expected, &(model.wList)
);

// Compute model and norm
ts::Tensor<T> output = model.compute(input);
ts::Tensor<T> norm = (*this->normFunction)(output -
   expected);

// Get gradient and increment gradient accumulator
ts::Gradient<T> gradient = norm.grad();
this->gradAccumulator.increment(gradient);

model.wList.reset();
```

Listing 4.4 – Calcul du gradient dans SGD

Ici, on commence par créer des `ts::Tensor` à partir du `ts::TrainingData` concerné (cela est nécessaire pour le calcul du modèle et du gradient). Puis, après avoir obtenu la sortie du modèle, on calcule la différence avec la sortie attendue, et on récupère le gradient de la norme. Enfin, l'accumulateur de gradient est incrémenté et la liste de Wengert réinitialisée pour les prochains calculs.

Pour finir sur la descente stochastique de gradient, chaque `ts::Optimizer` dispose aussi d'une méthode `updateModel` qui permet de mettre à jour les coefficients du modèle optimisé selon les valeurs de l'accumulateur de gradient. Cette méthode est typiquement appelée à la fin de chaque batch. La voici pour `ts::GradientDescentOptimizer` :

```
template <typename T>
void ts::GradientDescentOptimizer<T>::updateModel(
    ts::Model<T> &model, unsigned batchSize
) {
    for(unsigned i=0;
       i<this->gradAccumulator.elements.size();
       i++) {
        this->gradAccumulator.updateTensor(
            model, i,
            learningRate *
               this->gradAccumulator.elements[i].gradSum
               / batchSize
        );
    }
}
```

Listing 4.5 – Mise à jour du modèle dans SGD

Cette méthode parcourt encore une fois l'accumulateur de gradient. `updateTensor` encapsule la mise à jour des valeurs du modèle avec la valeur passée en paramètre. On reconnaît bien l'équation 4.2.

Cette partie aura permis de présenter non seulement la version la plus simple de l'algorithme de descente stochastique de gradient et son implémentation, mais servira aussi de base pour mettre au point des techniques d'optimisation bien plus avancées. C'est ce que nous verrons en partie suivante avec l'algorithme Adam.

4.2 Algorithme Adam

4.2.1 Principe de l'algorithme

L'algorithme Adam (pour *Adaptative Moment Estimation*) [18] reprend donc la structure de l'algorithme 3. Cependant, celui-ci propose une amélioration non négligeable en palliant au principal problème de la SGD de base, à savoir que le taux d'apprentissage est une constante η. Comme évoqué précédemment, on préfèrera la plupart du temps avoir un taux d'apprentissage élevé au début de l'algorithme, afin de s'approcher grossièrement mais rapidement d'un minimum, puis un taux faible à la fin pour apporter des ajustements très précis. Plusieurs algorithmes ont proposé des solutions

pour faire varier ce taux de manière pertinente : parmi ceux-ci, Adam est aujourd'hui considéré comme une des méthodes standard d'optimisation.

On va donc se baser sur l'algorithme 3 pour décrire l'algorithme Adam. Sans rentrer dans des considérations techniques concernant la convergence de l'algorithme et justifiant son fonctionnement, l'idée est d'adapter le taux d'apprentissage en évaluant à chaque itération les premiers et seconds moments du gradient, et de leur appliquer à chacun un taux d'affaiblissement exponentiel, respectivement β_1 et β_2, tout en prenant en compte la valeur précédente du gradient. Voici comment l'algorithme se présente :

Algorithme 4 : Algorithme Adam

Entrée : fonction f, paramètres θ_0, ensemble de batches B, nombre d'epochs e, taux d'affaiblissement $\beta_1, \beta_2 \in [0, 1)$, pas α

Résultat : paramètres θ_t

$m \leftarrow \{0\}_1^{|\theta_0|}$
$v \leftarrow \{0\}_1^{|\theta_0|}$
$\beta'_1 \leftarrow \beta_1$
$\beta'_2 \leftarrow \beta_2$
pour $i \leftarrow 1$ **à** e **faire**
  pour $t \leftarrow 1$ **à** $|B|$ **faire**
    $b \leftarrow B_t$
    $\nabla_{sum} \leftarrow \{0\}_1^{|\theta_0|}$
    pour $j \leftarrow 1$ **à** $|b|$ **faire**
      $g \leftarrow \nabla f_t(\theta_{t-1}, b_j)$
      $m \leftarrow \beta_1.m + (1 - \beta_1).g$
      $v \leftarrow \beta_2.v + (1 - \beta_2).g^2$
      $\hat{m} \leftarrow \frac{m}{1-\beta'_1}$
      $\hat{v} \leftarrow \frac{v}{1-\beta'_2}$
      $\nabla_{sum} \leftarrow \nabla_{sum} + \frac{\hat{m}}{\sqrt{\hat{v}_t}+\epsilon}$
    fin
    $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\nabla_{sum}}{|b|}$
    $\beta'_1 \leftarrow \beta'_1.\beta_1$
    $\beta'_2 \leftarrow \beta'_2.\beta_2$
  fin
fin

Comme on peut le voir α correspond au taux d'apprentissage initial,

et remplace en quelques sortes η dans l'algorithme SGD. Ce dernier sera multiplié à chaque fois par les $\frac{\hat{m}}{\sqrt{\hat{v}_t}+\epsilon}$ (ici, ϵ prend une valeur très faible pour éviter les divisions par 0 si un élément de $\hat{v}_t$ est nul). Comme β_1' et β_2' sont mis à une puissance de plus à chaque batch, cette quantité par laquelle sera multipliée α va donc être amenée à diminuer comme on le souhaitait. Dans la pratique, α prendra donc une valeur similaire à η dans SGD, et les β_1 et β_2 prendront une valeur proche de 1 (particulièrement pour β_2). Les auteurs de [18] conseillent d'utiliser les valeurs suivantes pour chacun des paramètres :

$$\begin{cases} \alpha = 0,001 \\ \beta_1 = 0,9 \\ \beta_2 = 0,999 \\ \epsilon = 10^{-8} \end{cases} \tag{4.3}$$

α représentera alors "l'amplitude" initiale des ajustements apportés aux paramètres θ, alors que β_1 et β_2 influeront sur la vitesse à laquelle cette amplitude diminuera.

4.2.2 Implémentation

L'implémentation de Adam se fera encore une fois à partir de la classe `ts::Optimizer`. La classe `ts::AdamOptimizer` fonctionnera donc sur le même principe que `ts::GradientDescentOptimizer`, et nous ne détaillerons donc pas toute son implémentation (la méthode `run` notamment est très similaire, car elle reprend la structure en 3 boucles). Il peut cependant être intéressant de voir comment sont effectués les calculs des estimateurs de moments qui sont la spécificité de l'algorithme. Ces derniers sont effectués dans la méthode `computeIncrement`, spécifique à `ts::AdamOptimizer` et appelée pour chaque instance de données :

```cpp
template <typename T>
void ts::AdamOptimizer<T>::computeIncrement(
    std::vector< Eigen::Array<T, Eigen::Dynamic,
       Eigen::Dynamic> >& derivatives,
    std::vector<ts::GaElement<T>>& elements
) {
    unsigned iGrad;
    for(unsigned iAcc=0; iAcc<elements.size();
       iAcc++) {

        // Get index in the gradAccumulator system
        iGrad = elements[iAcc].index;

        // Compute biased moment estimates
        m[iAcc] = beta1 * m[iAcc] + (1-beta1) *
           derivatives[iGrad];
        v[iAcc] = beta2 * v[iAcc] + (1-beta2) *
           derivatives[iGrad] * derivatives[iGrad];

        // Compute bias-corrected moment estimates
        mHat[iAcc] = m[iAcc] / (1 - decayedBeta1);
        vHat[iAcc] = v[iAcc] / (1 - decayedBeta2);

        // Replace gradient with its corrected value
        // (since gradient is used in the
           gradAccumulator increment method)
        derivatives[iGrad] = mHat[iAcc] /
           (vHat[iAcc].sqrt() + epsilon);
    }
}
```

Listing 4.6 – Calcul des incréments du gradient dans Adam

Il est encore une fois nécessaire de jongler entre les deux systèmes d'indices (du gradient et de l'accumulateur de gradient) : les indices dans le gradient (ou la liste de Wengert) sont repérés par `iGrad` et ceux dans l'accumulateur sont repérés par `iAcc`. Pour le reste, on retrouve bien les formules décrites dans l'algorithme 4. Cette méthode modifiera directement les valeurs du gradient avant d'ajouter l'incrément dans la méthode `increment` de `ts::GradientAccumulator`.

Pour conclure, l'implémentation des algorithmes d'optimisation était le dernier élément manquant à notre bibliothèque pour la rendre fonctionnelle. Avec ce chapitre, tous les concepts permettant son fonctionnement ont été expliqués, et il va être possible de passer à la mise en pratique dès le chapitre suivant.

Chapitre 5

Applications de la bibliothèque

Nous disposons à présent d'une base robuste permettant de mettre en place très facilement de nombreux modèles mathématiques, et permettant l'entraînement de chacun d'eux par différents algorithmes d'optimisation. Nous pouvons donc considérer avoir atteint notre but, qui était de développer une bibliothèque générale et facilement extensible. Ainsi, les exemples qui seront présentés dans la partie suivante ont été choisis de manière arbitraire et ne représentent qu'une petite partie des possibilités offertes par les outils que nous avons développé. Les modèles, algorithmes d'optimisation et fonctions qui ont été implémentés jusqu'à maintenant ont pour but de rendre ces exemples possibles, et on pourrait imaginer étendre bien plus la bibliothèque. On se baserait cependant toujours sur les mêmes les concepts fondamentaux que ceux introduits jusqu'à maintenant et qui en constituent le coeur.

5.1 Exemples d'application

5.1.1 Résolution de MNIST

Le premier exemple sur lequel nous allons tester notre bibliothèque sera celui de MNIST, qui a déjà été évoqué tout au long de ce rapport. Ce jeu de données est constitué d'un ensemble de fichiers binaires. [3] détaille leur structure et contient un lien de téléchargement. Alternativement, la bibliothèque inclut un script `examples/get-mnist.sh` téléchargeant et décompressant ces fichiers directement dans le bon répertoire. Une grande partie du code

nécessaire à la résolution de MNIST sera donc l'écriture du code capable de lire ces fichiers, et d'écrire les données dans des `ts::InputData`.

Une fois les données lues, il faut ensuite définir le modèle qui sera utilisé. MNIST étant un problème assez simple à résoudre, un simple perceptron multicouche sera suffisant. Voici sa structure et la création de son optimiseur :

```
1  std::vector<unsigned> layers = {512, 128, 10};
2
3  ts::MultiLayerPerceptron<float> model(
4      EXPECTED_IMAGE_SIZE,
5      layers
6  );
7
8  ts::AdamOptimizer<float> optimizer;
9
10 std::vector<std::vector<std::vector< float >>>
       losses =
11 optimizer.run(model, trainingData);
```

Listing 5.1 – Définition et utilisation du MLP pour MNIST

Comme les images de MNIST sont de taille 28×28 et en niveau de gris, le nombre de neurones sur la couche d'entrée sera de 784 (`EXPECTED_IMAGE_SIZE` dans le code). On choisit d'utiliser deux couches cachées de 512 et 128 neurones. La couche de sortie aura toujours une taille de 10, puisque c'est le nombre de classes dans MNIST. Concernant les fonctions d'activation, ReLU a été utilisé, avec σ sur la dernière couche (ce sont les paramètres par défaut de la bibliothèque). Enfin, on utilisera l'optimiseur Adam avec les paramètres par défaut (pour une comparaison avec SGD sur l'exemple de MNIST, voir la partie 5.2.3). Les valeur des fonctions de coût de chaque instance sont stockées dans le vecteur 3 dimensions `losses` (classées par epochs, batches, puis instances de données). Lorsque l'on choisit d'utiliser 1000 batches de taille 5 sur 10 epochs, voici les résultats donnés par ce vecteur :

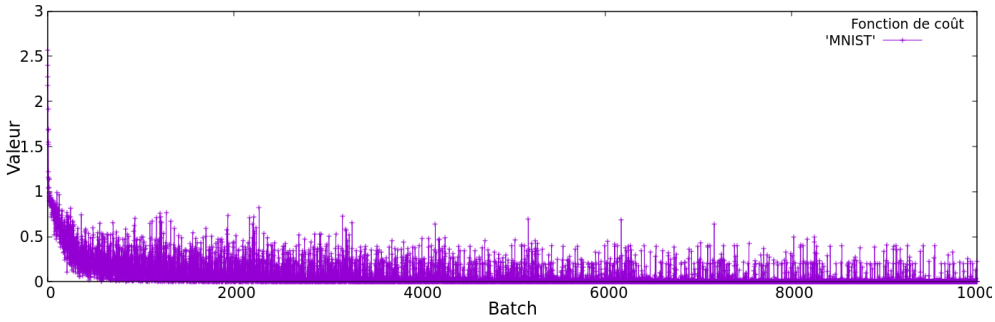

FIGURE 5.1 – Evolution de la fonction de coût de MNIST

Ici, chaque point représente la moyenne de la fonction de coût sur un batch. Comme souvent, cette dernière descend très rapidement au début, puis se stabilise au fur et à mesure. Si elle n'est pas forcément stable, on observe que les valeurs extrêmes sont de moins en moins élevées, même pour les derniers batches. Cela se traduira ainsi par une amélioration de la précision de quelques pourcents entre une phase d'entraînement de 10 batches et une plus courte de 5 batches par exemple (cela est possible grâce à l'affaiblissement du taux d'apprentissage de Adam, notamment).

La phase de test a été faite sur 300 instances. Le modèle a été capable de les classer correctement 289 fois, en commentant donc 11 erreurs. Cela donne ainsi une précision de 96,3%. C'est un résultat plutôt satisfaisant pour une modèle simple comme celui-ci, et dont la phase d'entraînement n'a duré que quelques minutes sur notre machine (l'état de l'art est à une précision de 99,84%, mais avec des modèles avancés comptant plus d'un milliard de paramètres [19]).

Remarque : Du fait que les coefficients du réseau sont initialisés de manière aléatoire, les résultats obtenus (même sur des donées identiques) ne seront pas nécessairement tojours les mêmes. Cependant, le nombre d'images utilisées pour la phase d'entraînement (5000) et les 300 tests réalisés devraient être suffisants pour se faire une bonne idée des résultats. Cette observation restera valable pour les exemples suivants.

Remarque : Le code complet de l'exemple MNIST se trouve dans le fichier `examples/mnist.cpp` de la bibliothèque.

5.1.2 Résolution de CIFAR

Afin de tester notre impémentation des réseaux à convolution, tentons de résoudre un problème similaire à MNIST, mais légèrement plus complexe. Pour ce faire, nous téléchargerons la base de données CIFAR [20]). Celle-ci est constituée de 60 000 images de 32×32 pixels. A la différence de MNIST, ces images contiennent 3 channels RGB, elles sont donc représentées par 3072 valeurs d'entrée. Cela est considérablement plus que les 784 valeurs de MNIST, d'autant plus que les classes seront aussi plus difficiles à différencier (avions, voitures, oiseaux, chats, cerfs, chiens, grenouilles, chevaux, bateaux et camions). CIFAR est donc un autre problème classique, mais de part sa plus grande complexité, il est souvent résolu avec des CNN. Voici quelques exemples d'images issues de ce jeu de données :

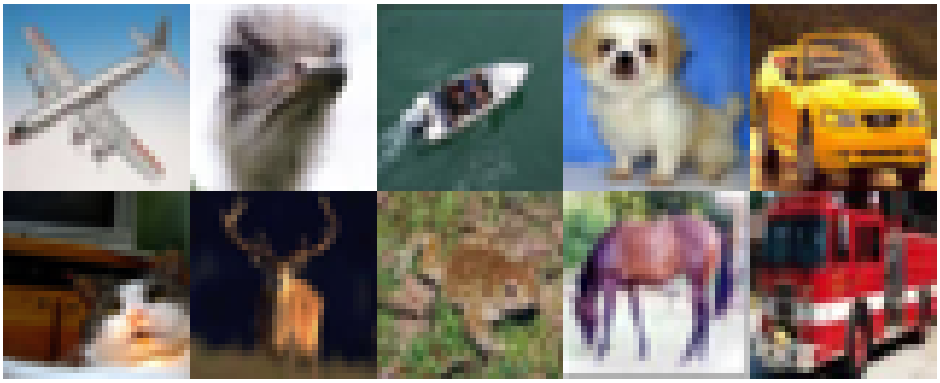

FIGURE 5.2 – Exemples d'images de CIFAR

De la même manière, ce jeu de données est contenu dans des fichiers binaires téléchargeables via le script `examples/get-cifar.sh`. Voici la structure du CNN que nous allons utiliser une fois les fichiers lus :

```
ts::ConvolutionalNetwork<float> model(
    // Input
    {IMAGE_HEIGHT, IMAGE_WIDTH},
    // Number of channels for input (3 for RGB)
    ts::ChannelSplit::SPLIT_HOR, 3,
    // Convolution / pooling
    {{3, 3, 32}, {5, 5, 32}},
    {{0,0}, {2, 2}},
    // Dense layers
    {256, 128, N_CLASSES}
);
```

Listing 5.2 – Définition du CNN pour CIFAR

Les données sont lues de manière à générer des tenseurs de taille 96×32. Chaque bloc de 32×32 contiendra les composantes d'une couleur. Pour cette raison, on définit un split vertical en 3 channels d'entrée. On aura deux couches de convolution de 32 channels avec des kernels de taille 3×3 et 5×5. Enfin, le réseau se termine avec 3 couches fortement connectées, la dernière étant aussi de taille 10 pour correspondre au nombre de classes de CIFAR. Comme pour MNIST, on utilise l'optimiseur Adam sur 1000 batches de taille 5, mais avec 15 epochs. Voici comment évolue la fonction de coût :

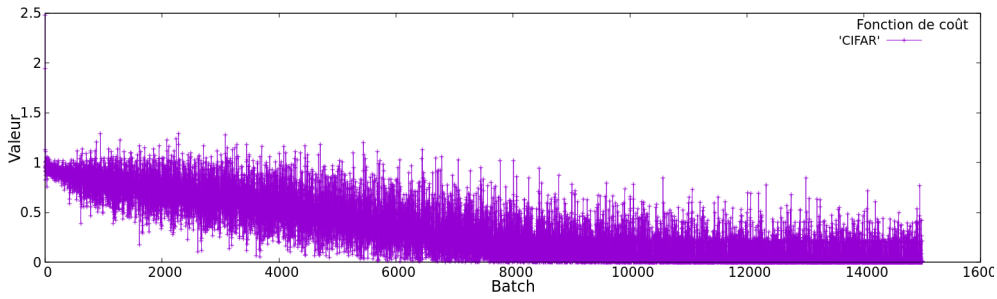

FIGURE 5.3 – Evolution de la fonction de coût de CIFAR

Sur les 300 test réalisés, le modèle a donné la bonne classe 148 fois, et a commis 152 erreurs. Cela donne une précision de 49,3%. Même si cela est bien inférieur aux résultats obtenus sur MNIST, et que l'entraînement du réseau a été bien plus long (on a utilisé 15 epochs, et le calcul d'un CNN est en règle générale bien plus coûteux en terme de calculs), une précision de 50%

constitue tout de même un gain considérable par rapport aux performances "baseline", qui seraient de 10%. Ainsi, cet exemple nous permet de valider notre implémentation des réseaux de neurones, même si l'utilisation d'une structure plus complexe et adaptée nous permettrait d'obtenir des meilleurs résultats, plus proches de ceux de MNIST. En effet, en comparaison avec 5.3, la fonction de coût du CNN semble converger bien plus lentement (c'est pour cette raison que l'on a choisi 15 epochs).

Pour conclure, et même si il est difficile de viser plus de 90% de précision sur CIFAR, une étude approfondie nous permettrait probablement d'améliorer notre modèle. Cependant les résultats obtenus nous indiquent que le fonctionnement de la bibliothèque est correct sur le fond, ce dont on voulait s'assurer avec cette application.

Remarque : Le code complet de l'exemple CIFAR se trouve dans le fichier `examples/cifar.cpp` de la bibliothèque.

5.1.3 Détection de véhicules

Enfin, le dernier exemple d'application reprendra le dataset CIFAR ainsi que le CNN qui y est défini pour procéder non plus à de la classification d'images, mais à de la classification d'objets. Cet exemple reposera sur un principe similaire à celui de l'algorithme R-CNN [21].

Le principe est le suivant : en se basant sur le dataset CIFAR, on entraînera un réseau à détecter des véhicules (il existe deux classes contenant des voitures et des camions). Pour les prédictions, on utilisera un autre jeu de données, Trafic-Net (`https://github.com/OlafenwaMoses/Traffic-Nethttps://github.com/OlafenwaMoses/Traffic-Net`) contenant des images de trafic d'une taille relativement faible (les dimensions dépassent rarement les 300 pixels). Encore une fois, on téléchargera le jeu de données avec notre script `examples/get-trafficnet.sh`. Même si les tailles des images ne sont pas constantes, il sera possible de sélectionner plusieurs régions de 32×32 pixels, et d'évaluer la probabilité qu'un véhicule s'y trouve. Cela sera d'ailleurs similaire à la manière dont on fait glisser un kernel lors d'une convolution matricielle.Le dataset Trafic-Net ne nous fournit pas de données qui permettraient de mesurerla précision de la détection de manières chiffrée, ce qui serait de toute façon relativement complexe (il ne s'agit pas simplement de vérifier la classe à laquelle appartient l'image). On se contentera donc de visualiser graphiquement nos résultats en dessinant sur l'image les régions qui contiendraient des véhicules.

Commençons donc par détailler la phase d'entraînement. Tout d'abord, il faudra adapter les données d'entraînement de nos modèle. Comme dit plus haut, on utilisera les classes correspondant aux voitures et aux camions pour la détection, notre réseau ne fera pas la différence entre les deux. Au lieu d'avoir une couche de sortie de taille 10 pour classifier l'image, on cherche simplement à savoir si un véhicule s'y trouve. La couche de sortie sera donc de taille 1. Comme pour toutes les phases d'entraînement, il faudra s'assurer que chaque classe (ici, un véhicule ou non) apparaît de manière de manière équiprobable dans nos données. On sélectionnera donc toutes les images de CIFAR appartenant aux classes voiture ou camion, et un quart des autres. La structure du CNN utilisé sera similaire à celle de l'exemple 5.1.2 :

```
ts::ConvolutionalNetwork<float> model(
    // Input
    {IMAGE_HEIGHT, IMAGE_WIDTH},
    // Number of channels for input (3 for RGB)
    ts::ChannelSplit::SPLIT_HOR, 3,
    // Convolution / pooling
    {{3, 3, 32}, {5, 5, 16}},
    {{0,0}, {2, 2}},
    // Dense layers (with output vector & not
       including first layer)
    {128, 64, N_CLASSES, 1}
);
```

Listing 5.3 – Définition du CNN pour la détection de véhicules

Ce réseau possède cependant quelques différences avec celui utilisé pour résoudre CIFAR. Tout d'abord, on a rajouté une couche dense de taille 1 en dernière position pour avoir une sortie de la bonne taille. De plus, l problème de classification que doit résoudre ce réseau étant plus simple, car composé de seulement une classe dont il faut évaluer la probabilité, on se permettra d'utiliser 16 channels seulement sur la deuxième couche de convolution. Surtout, cela nous permettra un gain de temps appréciable lors de la phase de prédiction, pour des raisons que nous évoquerons bientôt. A la fin de la phase d'entraînement, le réseau atteint une précision de 87% sur la classification (261 succès et 39 erreurs). On considère que la précision est un succès si la distance avec la valeur réelle (0 ou 1) est inférieure à 0,5. Pour le réutiliser lors de la phase de prédiction, on finit par le sauvegarder dans un fichier externe.

La classification d'une image se fait donc en calculant la sortie du réseau pour plusieurs de ses régions. Une méthode naïve consisterait à faire glisser cette région à la manière d'un kernel lors d'une convolution. Cela nécessiterait cependant de nombreux passages en avant avec notre CNN. Le calcul de la sortie d'un tel réseau étant plus long que les simples calculs efectués sur les kernels pendant une convolution, cette approche pourrait poser des problèmes en termes de performance. Même pour les images de taille assez faibles de Traffic-Net, le nombre de région à classifier serait trop élevé. Pour une image de 200×200 pixels, et avec nos régions de taille 32×32, on calculerait alors $(200 - 32 + 1) \times (200 - 32 + 1) = 28561$ pasages en avant.

Comme précisé au début de cette partie, il existe des algorithmes très avancés traitant de ces problématiques et détaillés en [21]. Par souci de simplicité, nous proposerons une solution imparfaite mais très facile à implémenter. Plutôt que de décaler chaque nouvelle région d'un seul pixel, on définira un "stride", correspondant à la distance séparant chaque région. Afin de ne pas trop affecter la qualité de la détection, il conviendra de ne pas choisir cette valeur trop élevée. Cela sera cependant suffisant pour réduire de manière considérable les calculs : dans notre exemple précédent, et pour des strides horizontaux et verticaux de 5, le nombre de régions à classifier passerait alors à $\frac{(200-32+1)}{5} \times \frac{(200-32+1)}{5} < 1143$. Evidemment, cela reste élevé, et cette solution n'est pas optimale, mais elle sera un bon compromis entre simplicité et efficacité.

Après avoir effectué tous les passages en avant, on obtient une "matrice de probabilité", dont chaque coefficient correspond à la probabilité qu'un véhicule se trouve dans la région associée. Pour formuler une prédiction finale concernant chaque région, on utilisera un *seuil de prédiction*. Si il est dépassé, alors on prédira un véhicule. A partir de la matrice de probabilité, on peut facilement retrouver la région concernée dans l'image de base, et dessiner cette dernière.

Dans la pratique, on choisira des slides de 5, et un seuil de détection de 0,999. Voici quelques exemples d'images sur lesquelles on a indiqué graphiquement les objets détectés :

FIGURE 5.4 – Exemples de détections de véhicules

Comme on peut le voir, notre système donne d'assez bons résultats sur un bon nombre d'images en étant capable de détecter des véhicules de tailles différentes en terme de pixels (même plus grands que nos régions de 32×32). Cependant, il se comporte moins bien sur certaines images, en ne détectant pas certains véhicules et en donnant parfois des faux positifs :

FIGURE 5.5 – Exemples de mauvaises détections

De plus, notre système se contente de mettre en valeur les régions pour lesquelles des véhicules sont détectés. Cependant, celles-ci concernent parfois le même véhicule qui sera donc reporté plusieurs fois.

Pour conclure, si ce dernier n'est pas parfait, notre système est tout de même capable de donner de bons résultats sur beaucoup d'images. On peut donc considérer cette expérience comme satisfaisante, les deux datasets choisis n'étant pas forcément prévus pour cette utilisation et notre CNN de base n'étant pas optimal. La détection d'objets peut devenir un sujet assez complexe, avec des algorithmes tels que R-CNN ou YOLO qui permettent des optimisations très importantes. Nous n'avons donc qu'effleuré le sujet dans

cette partie. Cela aura tout de même permis de mettre en place une application légèrement plus complexe que de la simple classification d'images, et qui s'approche de problématiques réelles apparaissant dans des domaines de recherche très actuels, tels que le développement de logiciels pour voitures autonomes par exemple.

Remarque : Le code de cet exemple est réparti en deux fichiers. La phase d'entraînement se trouve dans le fichier `examples/rcnn_train.cpp`, et la phase de prédiction dans `examples/rcnn_predict.cpp`.

5.2 Analyse des performances et résultats

Dans cette section, on se proposera d'étudier les performances de la bibliothèque, et de comparer l'efficacité de différents concepts introduits dans les chapitres précédents.

Remarque : Tous les benchmarks présentés dans les parties suivantes ont été réalisés sur un processeur Intel Core i5-9300H (8 cores @ 2,4 GHz) :

- L1d cache : 128 KiB
- L1i cache : 128 KiB
- L2 cache : 1 MiB
- L3 cache : 8 MiB

5.2.1 Parallélisation

Pour commencer, intéressons-nous à l'impact sur les performances que peut avoir la mise en parallèle de la bibliothèque. Eigen propose par défaut une parallélisation avec OpenMP de certains de ses algorithmes, y compris du produit de matrice qui constitue une part importante des calculs. On mesure ainsi les temps de calcul (relatifs pour chaque réseau) des MLP et CNN pour différents nombres de threads :

Nombre de threads	MLP	CNN
1	1,065	1,240
2	1	1,097
4	1,022	1
8	1,489	1,126

TABLE 5.1 – Temps de calcul relatifs selon le nombre de threads

Tout d'abord, on remarque que les performances semblent chuter lorsque l'on utilise 8 threads. Cela est dû au fait que notre processeur de test dispose en réalité de 4 coeurs physiques. L'utilisation de 8 threads nous fait donc utiliser deux fois plus de coeurs que ce dont on dispose réellement, ce qui explique ces mauvaises performances.

Pour le reste, on observe que le temps de calcul du MLP reste le même quelque soit le nombre de threads utilisés. On rappelle que seulement certaines opérations sur les `Eigen::Array` sont parallelisées. Or, l'utilisation typique de la librairie consistera plutôt en des appels répétés à ces opérations, sur des tableaux de taille relativement faibles. On touche ici au plus gros problème pour paralléliser efficacement notre librairie : l'overhead créé par la parallélisation est selon toute vraisemblance trop grand par rapport aux gains de performances qu'elle pourrait offrir.

Pour démontrer cela, le CNN a été dimensionné avec les plus grandes entrées que l'on a utilisé jusqu'à maintenant (128 channels d'entrée de même taille que pour CIFAR). Ainsi, les produits de matrices engendrés devraient être les plus longs que l'on a eu à calculer, et les avantages de la parallélisation devraient commencer à se faire sentir. Cela semble effectivement être le cas si l'on regarde les résultats. Ainsi, à moins d'utiliser des réseaux d'une taille assez importante, il sera difficile de paralléliser efficacement la librairie. Des essais pour paralléliser d'autres opérations ont été faits, mais ne présentaient encore une fois pas de gains de performances substantiels et n'ont donc pas été retenus.

Pour pallier à cela, il serait sûrement nécessaire de procéder à la parallélisation de la librairie à un niveau bien plus élevé. Par exemple, des versions parallèles de l'algorithme SGD existent [22]. En effet, à cause des dépendances existant entre les calculs dans les algorithmes d'optimisation, il est nécessaire de les adapter pour les paralléliser, ce qui ne peut ainsi pas se faire de manière triviale.

Pour conclure, le parallélisme devrait probablement être implémenté à un haut niveau pour avoir un réel intérêt. Cela demanderait cependant des efforts d'implémentation. Pour autant, on a tout de même conservé la parallélisation par défaut de Eigen, qui pourrait tout de même avoir un intérêt sur des réseaux de grande taille, et qui peut facilement être désactivée en changeant le nombre de threads que l'on souhaite utiliser.

5.2.2 σ vs ReLU sur un MLP

Comme nous l'avons déjà évoqué, les réseaux modernes n'utilisent plus que très rarement σ comme fonction d'activation, et prèfèrent souvent des fonctions d'activation telles que ReLU. Les raisons principales sont que les fonctions d'activation peuvent avoir tendance à "saturer" lorsqu'elles prennent une valeur trop élevée en entrée. Il devient alors difficile de tirer une information du neurone en question :

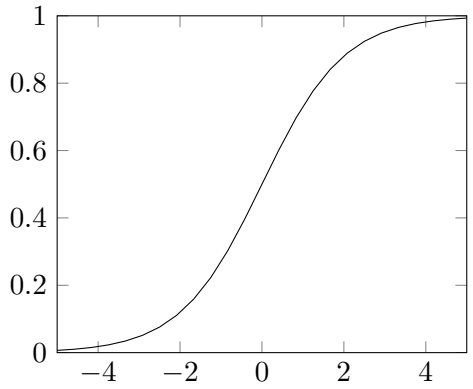

FIGURE 5.6 – Graphe de la fonction σ

On observe que la fonction s'approche très rapidement de 1, il serait alors facile d'atteindre une valeur extrême et de trop s'approcher de 1 ou de 0. A l'inverse, ReLU tend vers $+\infty$. Le problème de saturation n'existe plus, mais on risque cependant d'avoir des valeurs qui explosent après plusieurs couches :

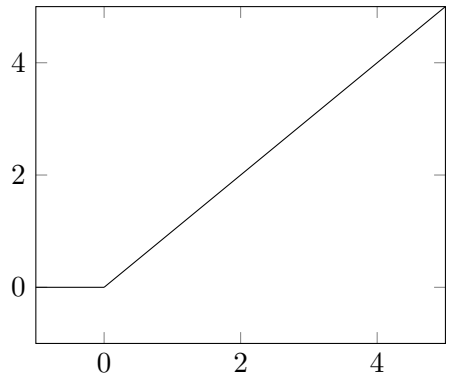

FIGURE 5.7 – Graphe de la fonction σ

Ce point n'a pas été abordé jusqu'ici, mais il existe des techniques permettant d'initialiser les coefficients et de pallier à ce problème, notamment les méthodes d'initialisation de Xavier et de He [23] (la bibliothèque utilise l'initialisaton de He par défaut pour les perceptrons multicouches et les réseaux à convolution). Un autre avantage de ReLU est que cette fonction est en théorie plus simple à calculer. Vérifions dans la pratique les temps de calcul obtenus pour ces deux fonctions sur des matrices carrées de différentes tailles (le benchmark inclut aussi la dérivation des fonctions pour se rapprocher d'une situation réaliste dans laquelle on entraînerait un réseau) :

Taille de la matrice	σ	ReLU
10	1,422	1
100	2,400	2,253
1000	2,505	2,371
5000	20,918	20,660

TABLE 5.2 – Temps de calcul relatifs de σ et ReLU

Comme on peut le voir, nos deux implémentations sont à peu près similaires en terme de temps de calcul, malgré un léger avantage pour ReLU. Considérons maintenant les résultats obtenus sur MNIST en 5.1.1 comme référence. Le modèle présenté (utilisant ReLU comme sa fonction d'activation, mis à part pour la dernière couche) obtenait alors une précision de 96,3%. Pour juger l'efficacité de σ, utilisons exactement le même modèle sur la même phase d'entraînement, mais uniquement avec des σ. On obtient

alors une précision légèrement inférieure de 94,3% (282 succès pour 18 erreurs). Sur cet exemple, ReLU semble plus efficace, mais l'avantage reste encore une fois léger.

De la même manière, considérons maintenant l'exemple de CIFAR de la partie 5.1.2 et les 49.3% de précision obtenus. Si on utilise σ non seulement sur les couches fortement connectées, mais aussi les couches de convolution, la précision du réseau chute à 12,3% (37 succès pour 263 erreurs), ce qui est à peine supérieur aux performances baseline de 10%. Cela peut s'expliquer par le fait que la fonction σ est très peu adaptée aux convolutions multicouches, car le fait de sommer les résultats pour chaque couche d'entrée va causer une saturation rapide de la fonction.

Pour conclure, si la fonction σ a des performances presque similaires à ReLU (autant en terme de temps de calcul qu'en précision obtenue) sur des applications simples, elle trouve ses limites sur des problèmes et structures de réseaux plus complexes tels que les CNN par exemple. On comprend donc pourquoi des fonctions de rectification telles que ReLU sont aujourd'hui considérées comme le choix standard.

Remarque : Le code complet de ce benchmark se trouve dans le fichier `perf/activation_functions.cpp` de la bibliothèque.

5.2.3 Optimiseur SGD vs Adam

De la même manière que nous avons comparé l'efficacité de σ et ReLU, essayons de juger le gain de précision apporté par l'algorithme Adam sur la SGD classique. On va encore une fois utiliser les résultats sur MNIST de la partie 5.1.1 comme référence. On va donc reprendre exactement la même application, mais en utilisant l'algorithme SGD avec un taux d'apprentissage $\eta = 0,1$. Cette valeur semble en effet donner de bons résultats en pratique, et les paramètres d'Adam ayant été laissés par défaut, simplement choisir une valeur pour η avec le bon ordre de grandeur devrait permettre une comparaison "équitable". On obtient alors une précision de 91,7% (275 succès pour 25 erreurs), ce qui reste assez proche des résultats obtenus avec Adam.

Enfin, réalisons la même expérience sur l'exemple de CIFAR de la partie 5.1.2. On gardera $\eta = 0,1$. Cela donne une précision de 44,7% (134 succès pour 166 échecs). Encore une fois, les résultats sont légèrement en deça de ceux de Adam mais restent assez comparables.

Evidemment, il est possible d'ajuster très précisément le coefficient d'apprentissage de SGD, ce qui nous permettrait de nous rapprocher encore plus des résultats de Adam. De la même manière, cela est aussi un signe que si les paramètres par défaut de Adam donnent des résultats satisfaisants, il serait probablement nécessaire de les ajuster pour tirer entièrement parti de l'algorithme. Cela reste cependant plus complexe, car les paramètres sur lesquels il est possible de jouer sont plus nombreux, mais aussi car cela nécessite une bonne compréhension de l'effet de chacun sur l'évolution du taux d'apprentissage effectif dans le cas de notre modèle.

On concluera que l'algorithme SGD, quand il est bien paramétré, permet sur les applications relativement simples que nous avons vu d'obtenir des résultats comparables à Adam. Concernant ce dernier, nous n'avons probablement pas exploité son plein potentiel en gardant les paramètres par défaut, et il serait intéressant de voir comment leur modification permettrait d'améliorer la précision.

5.2.4 Convolution naïve vs im2col

Nous avons défini en partie 3.3.2 l'opération de convolution multicanal, puis présenté en partie 3.3.3 la méthode im2col qui permet d'optimiser le calcul de cette opération, et qui a été utilisée pour l'implémentation des CNN. On se propose dans cette partie de comparer le calcul d'une convolution multicanal "naïve" (c'est à dire une combinaison de différentes convolutions matricielles sur plusieurs matrices), avec une convolution im2col, composée à la suite d'une opération im2col, du produit de matrices correspondant, et d'une opération col2im pour redimensionner le résultat. On choisira de faire le test sur une convolution comportant de relativement nombreux canaux, car ce sont celles-ci qui sont les plus coûteuses en terme de calcul, mais aussi qui permettent le meilleur speedup avec la méthode im2col. On choisira donc 16 channels d'entrée et 32 channels de sortie, avec des kernels de taille 3×3. On fera ainsi varier la taille des matrices de base, qui resteront carrées. Voici les temps d'exécution moyen des 2 versions pour différentes tailles :

Taille de la matrice	Convolution naïve (ns)	Convolution im2col (ns)
10	11,479	1
50	218,730	59,723
100	901,228	183,469
250	6427,840	1095,189
500	31054,247	5209,825

TABLE 5.3 – Temps de calcul relatifs de convolutions multicanal

Comme on pouvait l'espérer, l'approche im2col permet donc un speedup considérable, supérieur à 10 pour des matrices de petites tailles, et de l'ordre de 6 sur une matrice de taille 500. Il faudrait cependant noter que ce résultat serait moins impressionnant si on avait diminué le nombre de channels dans notre benchmark. Le speedup, sur 3 couches d'entrées pour une image RGB par exemple, serait probablement plus faible et ce benchmark se rapproche donc plus du calcul d'une couche de convolution intermédiaire.

A ce titre, il pourrait être intéressant d'étudier plus en détail le gain de performances de im2col en faisant varier d'autres paramètres que la taille des matrices d'entrée. Ce benchmark permet cependant de confirmer l'intérêt et la validité de l'approche im2col, d'ailleurs déjà observés dans la pratique.

Remarque : Le code complet de ce benchmark se trouve dans le fichier `perf/convolutions.cpp` de la bibliothèque.

5.2.5 Comparaison avec PyTorch

Pour terminer sur l'évaluation des performances de notre bibliothèque, il peut être intéressant de la comparer rapidement à une autre bibliothèque de machine learning proche de l'état de l'art. Pour cette expérience, on choisira PyTorch. Il s'agit d'une bibliothèque développée par Facebook et dont la première version date de 2016. Comme notre bibliothèque, elle est écrite en C++, même si elle propose une interface en Python. Pour cette raison, il s'agit d'un bon choix pour effectuer une comparaison de performances. Le site officiel de PyTorch (`https://pytorch.org/`) propose plus d'informations à son sujet.

Le test que l'on choisira d'effectuer sera celui du MLP de l'exemple 5.1.1. On reproduira donc exactement les mêmes conditions de test dans les deux bibliothèques : on utilisera le même réseau avec les mêmes fonctions de

coût et d'activation, et procéderons à une phase d'entraînement avec l'algorithme SGD. Cette dernière se fera sur 60 000 entrées réparties en batches de taille 5, ce qui correspond à l'intégralité du dataset MNIST. Voici les temps d'exécution moyens obtenus sur TensorSlow (notre bibliothèque) et PyTorch :

Bibliothèque	Temps d'exécution (s)
TensorSlow (1 thread)	218
TensorSlow (4 threads)	226
PyTorch (1 thread)	94
PyTorch (4 threads)	63

TABLE 5.4 – Temps d'entraînement d'un MLP sur MNIST (s)

Comme cela était attendu, les performances de PyTorch sont supérieures. Cela pourrait s'expliquer entre autres par une meilleure parallélisation de PyTorch (qui semble plus performant avec 4 coeurs), ou par l'utilisation de BLAS optimisés. Si il est possible de les choisir avec Eigen, ils ne sont pas utilisés par défaut, et n'ont pas été activés dans notre bibliothèque pour des soucis de portabilité. Malgré tout, être à 43% (pour 1 thread) ou 28% (pour 4 threads) des performances d'une bibliothèque telle que PyTorch reste relativement satisfaisant : si quelques efforts ont été faits pour optimiser les performances de notre bibliothèque, nous avons vu qu'il reste plusieurs pistes d'amélioration qui mériteraient d'être explorées pour se rapprocher des librairies de référence.

Chapitre 6

Conclusion

Pour conclure sur ce projet, nous organiserons notre analyse en deux points.

Tout d'abord, abordons le développement de la bibliothèque. D'une part, les problématiques que nous avons eu à résoudre nous ont permis d'exploiter de nombreuses fonctionnalités du C++. Mais au delà de ça, nous avons aussi eu l'occasion d'utiliser des bibliothèques intéressantes telles que Eigen. Il a été possible d'en tirer pleinement parti tout au long du développement : celle-ci a non seulement facilité le travail, mais a également permis d'offrir des performances bien supérieures à ce que l'on aurait pu viser si il avait fallu implémenter nous-mêmes toutes les opérations sur les matrices. Une autre bibliothèque importante et qui n'a pas été abordée jusque là est Google Test (`https://github.com/google/googletest`). Au fur et à mesure du processus de développement, on a mis en place tout un ensemble de tests pour s'assurer que chaque nouveau composant était fonctionnel. De plus, le code étant hébergé sur Github depuis le début du développement, il a été possible de combiner ces tests avec les Github Workflow. Ces derniers permettant ainsi à chaque modification sur la branche `master` de lancer l'ensemble des tests qui avaient été développés et de vérifier leur succès. Même si cela était assez basique, cela a permis la mise en place d'une forme d'intégration continue dans notre mode de travail.

Ensuite, au niveau du fond, nous avons globalement atteint le but que l'on s'était fixé dans ce projet. En partant pratiquement de zéro (on a juste utilisé une bibliothèque d'algèbre linéaire), nous avons passé en revue la base de la théorie englobant le machine learning et les réseaux de neurones, et avons graduellement implémenté les concepts abordés, pour finir par des

mises en pratiques qui ont été globalement satisfaisantes. Le résultat n'est évidemment pas complet, ni parfait : notamment, il pourrait être intéressant de considérer l'utilisation des `Eigen::Tensor`. Ceux-ci nous permettraient en effet d'utiliser des variables avec des dimensions dupérieures à 2. Si ils n'ont pas été retenus car leur support est encore expérimental, ils offrent tout de même une perspective d'amélioration intéressante. De plus, cela aurait auss été une bone idée de paralléliser nos algorithmes à un plus haut niveau, voire de les porter sur GPU pour s'approcher des performances de l'état de l'art (comme observé en 5.2.5).

Cependant, le choix du sujet permettait à ce projet de s'étendre d'une manière virtuellement infinie : le développement d'une bibliothèque comme PyTorch ou TensorFlow par exemple est une tâche très vaste et complexe, nécessitant plusieurs années de travail à des équipes importantes. Il était donc évident qu'il serait impossible de traiter ce sujet d'une manière absolument exhaustive dans le cadre de ce projet. Pour autant, notre travail a posé les bases du fonctionnement de telles bibliothèques, et en donnant une vue malgré tout assez complète de ce domaine, il fournit tous les outils nécessaires pour continuer de s'y intéresser dans des travaux futurs.

Table des figures

Liste des tableaux

Bibliographie

[1] Wikipedia. *Machine learning*. URL : https://en.wikipedia.org/wiki/Machine_learning.

[2] Matthew Scarpino. *TensorFlow for dummies*. John Wiley et Sons, Inc, 2018.

[3] Christopher J.C. Burges Yann LeCun Corinna Cortes. *THE MNIST DATABASE of handwritten digits*. URL : http://yann.lecun.com/exdb/mnist/.

[4] Tom B. Brown et al. "Language Models are Few-Shot Learners". In : (2020).

[5] Grant Sanderson. *Neural Networks*. 2018. URL : https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[6] Wikipedia. *Softmax function*. URL : https://en.wikipedia.org/wiki/Softmax_function.

[7] Baydin et al. "Automatic Differentiation in Machine Learning : A Survey". In : *Journal of Machine Learning Research* (2018).

[8] Wikipedia. *Chain rule*. URL : https://www.wikiwand.com/en/Chain_rule.

[9] Phil Ruffwind. *Reverse-mode automatic differentiation : a tutorial*. 2016. URL : https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation.

[10] Wikipedia. *Tensor derivative (continuum mechanics)*. URL : https://en.wikipedia.org/wiki/Tensor_derivative_(continuum_mechanics).

[11] M. B. Giles. "Collected matrix derivatives results for forward and reverse mode AD". In : (2008).

[12] Wikipedia. *Overfitting*. URL : https://en.wikipedia.org/wiki/Overfitting.

[13] Wikipedia. *Convolution*. URL : https://en.wikipedia.org/wiki/Convolution.

[14] Amir Ali. *Convolutional Neural Network(CNN) with Practical Implementation*. 2019. URL : https://medium.com/machine-learning-researcher/convlutional-neural-network-cnn-2fc4faa7bb63.

[15] Alex Thevenpt. *Conv2d : Finally Understand What Happens in the Forward Pass*. 2020. URL : https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148.

[16] Leonardo Araujo dos Santos. *Making faster - Artificial Intelligence*. URL : https://leonardoaraujosantos.gitbook.io/artificial-inteligence/.

[17] Sharan Chetlur et al. "cuDNN : Efficient Primitives for Deep Learning". In : (2014).

[18] Jimmy Lei Ba Diederik P. Kingma. "Adam : a method for stochastic optimization". In : *ICLR 2015* (2015).

[19] Papers with Code. *Image Classification on MNIST*. URL : https://paperswithcode.com/sota/image-classification-on-mnist.

[20] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009. URL : https://www.cs.toronto.edu/~kriz/cifar.html.

[21] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO - Object Detection Algorithms*. 2018. URL : https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[22] Martin A. Zinkevich et al. "Parallelized Stochastic Gradient Descent". In : (2010).

[23] Christian Versloot. *He-Xavier initialization and activation functions : choose wisely*. URL : https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-functions-choose-wisely/.