Université de Versailles-Saint-Quentin-en-Yvelines / Paris-Saclay

Spécialité :
Master 1 Calcul Haute Performance et Simulation

Rapport du Projet de Programmation Numérique

Sujet :
Implémentation et optimisation d'un réseau de neurones pour la reconnaissance de chiffres manuscrits en C

07 Mai 2023

Enseignant Encadrant :
DELVAL Aurélien

Auteurs :
BENNACER Meriem
DEMIGHA Hamza Amine
KEDDAD Youcef
SAKKA Chadha

Remerciements

Tout d'abord, nous tenons à exprimer notre profonde gratitude à notre encadrant, M. DELVAL Aurélien, pour son soutien inestimable, ses conseils précieux et son expertise tout au long de ce projet. Ses orientations et ses encouragements nous ont permis de surmonter la plupart des défis rencontrés et de mener à bien notre travail. Nous apprécions sincèrement sa disponibilité, sa patience et son approche pédagogique, qui ont grandement contribué à notre compréhension des concepts et à l'acquisition des compétences nécessaires à la réalisation de ce projet. Nous tenons à le remercier chaleureusement pour son engagement et son accompagnement, sans lesquels ce rapport n'aurait pas été possible.

Nous tenons également à exprimer notre gratitude à toutes les personnes qui prendront le temps de lire ce rapport. Votre intérêt et vos éventuelles suggestions seront d'une grande valeur pour nous, et nous permettront d'améliorer notre travail et d'enrichir notre compréhension des concepts abordés. Nous espérons que les résultats présentés ici sauront susciter votre curiosité et vous encourager à approfondir les recherches dans le domaine de l'intelligence artificielle et l'optimisation de la précision et des performances des réseaux de neurones. Nous vous remercions sincèrement pour votre attention et sommes impatients de recevoir vos retours et commentaires constructifs.

Table des matières

Introduction générale **1**

Présentation du modèle de base **2**
1.1 Contexte . . . 2
1.2 Objectifs . . . 2
1.3 Planning d'avancement . . . 3

Amélioration de la précision **4**
1. Taux d'apprentissage . . . 4
2. Recherche du nombre de noeuds optimal . . . 4
3. Comparaison des fonctions d'activation . . . 6
3.1 La fonction Sigmoïde . . . 7
3.2 La fonction ReLU . . . 7
3.3 La fonction Softmax . . . 7
3.4 Analyse comparative des résultats . . . 8
4. Implémentation d'une deuxième couche cachée . . . 9
5. Phase d'entraînement . . . 11
5.1 Entraînement du réseau . . . 11
5.2 Propagation avant (feed_forward) . . . 11
5.3 Rétropropagation (BackPropagate) . . . 11
5.4 La descente de gradient . . . 12
6. Étude de la précision dans le temps . . . 14
7. Bonus : Le prétraitement de jeux de données . . . 15

Amélioration de la performance **17**
Analyse et Exploration des Stratégies de Parallélisation . . . 17
1. Architecture x86-64: . . . 17
2. Mesure de performance de la version séquentielle . . . 17
Parallélisation et optimisation des performances . . . 18
1. Parallélisation des opérations élémentaires . . . 18
2. Un format à virgule flottante . . . 21
3. Utilisation des directives OpenMP . . . 22

Conclusion et perspectives **25**

Table des figures

Figure 1 : Architecture de notre réseau de neurones de base 2
Figure 2 : Variation du taux de précision en fonction du nombre de neurones avec Sigmoïde 6
Figure 3 : Comparaison des performances des fonctions d'activation 8
Figure 4 : Nouvelle structure du réseau de neurones avec 2 couches cachées . . 9
Figure 5 : Variation du taux de précision en fonction du nombre des images testées : une couche cachée Vs 2 couches cachées 10
Figure 6 : Variation du taux de précision en fonction du nombre des "epochs" 14
Figure 7 : L'affichage d'une image représentant le digit 0 après le traitement . 15
Figure 8 : Comparaison de la version de base (DOTPROD) et la version optimisée (DGEMM) 19
Figure 9 : Comparaison de la version de base et la version utilisant CBLAS sur toutes les opérations matricielles 20
Figure 10 : Comparaison de la dernière version optimisée avec CBLAS (format double) et la version CBLAS (format float) 21
Figure 11 : Comparaison de la version CBLAS avec le format float et la version OpenMP 22
Figure 12 : Comparaison de toutes les pistes d'optimisation explorées 23

Liste des tableaux

TABLE1 : Tableau d'analyse du code séquentiel avec l'outil GNU profiler : gprof . 18
TABLE2 : Données relatives aux threads, à la taille de l'entraînement, aux nœuds, au temps d'entraînement (s) et à la prédiction du réseau (%). 23

Introduction

Ce rapport présente en détail le travail effectué durant le deuxième semestre, dont l'objectif était d'explorer les différentes méthodes pour améliorer les performances de notre réseau de neurones. Nous avons cherché à améliorer la précision des prédictions ainsi que le temps nécessaire d'exécution pour la phase d'entraînement et la phase de test.

Le domaine des réseaux de neurones étant très vaste, nous avons choisi de nous concentrer sur des pistes de recherche qui pourraient se révéler intéressantes. Nous avons ainsi étudié des modifications de l'architecture du réseau de neurones, des méthodes d'optimisation différentes, ainsi que l'utilisation de différentes bibliothèques optimisées pour exploiter la parallélisation et accélérer le traitement de données en divisant les tâches de calcul entre plusieurs threads.

Ce rapport présente les différentes approches que nous avons étudiées, les avantages et les inconvénients de chaque méthode et leur impact sur la précision des prédictions et sur les temps de calcul. Il constitue ainsi une synthèse des travaux réalisés au cours de ce deuxième semestre pour améliorer les performances de notre réseau de neurones.

Voici le lien vers le Code source du projet.

Présentation du modèle de base

1.1 Contexte

Le réseau de neurones implémenté au premier semestre est un perceptron multicouche. C'est une architecture de réseau de neurones *feedforward* où le flot d'informations circule de la couche d'entrée (784 noeuds) vers la couche de sortie (10 noeuds) passant par une seule couche cachée formée de 100 noeuds. L'unique fonction d'activation testée au premier semestre est la fonction Sigmoïd. Le meilleur taux de prédiction mesuré est de 84 % pour une base d'entraînement de 5000 images et une base de test de 1000 images. Voici un schéma représentatif de l'architecture du réseau de neurone de base :

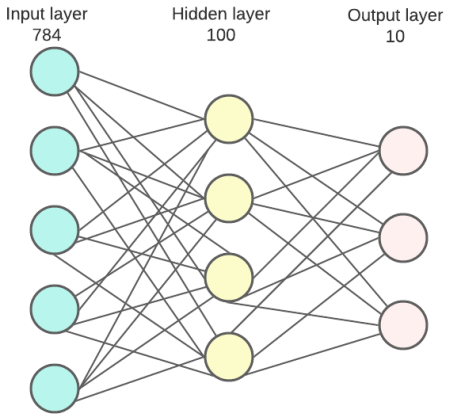

FIGURE 1 – Architecture de notre réseau de neurones de base

1.2 Objectifs

L'objectif du deuxième semestre est d'apporter dans un premier temps, des corrections sur le réseau implémenté au premier semestre notamment l'implémentation de notre propre code qui lit les images et leurs étiquettes à partir du jeu de données MNIST. Ensuite, d'améliorer la précision de la prédiction et le temps d'entraînement en exploitant le parallélisme afin de garantir une meilleure performance.

1.3 Planning d'avancement

Février
— Réunion 1
— Le pré-traitement des images de la Data Set MNIST.
— Correction de la version séquentielle de base implémentée au premier semestre.
— Une compréhension plus approfondie de l'apprentissage profond.
Mars
— Réunion 2 : le 03 mars 2023
— Comparaison de différentes topologies du réseau en jouant sur le nombre de couches cachées et le nombre de noeuds.
— Ajout et test des différentes fonctions d'activation.
— Mesure de performance de la version séquentielle.
Avril
— Réunion 3 : le 07 avril 2023
— L'exploration des différentes pistes d'optimisation de la descente de gradient.
— Exploration des pistes de parallélisation.
— Parallélisation des opérations élémentaires avec CBLAS.
Mai
— Réunion 4 : le 04 mai 2023
— La parallélisation du code avec les directives de OpenMP.
— Itération sur la version parallèle.
— Mesures des performances de la version parallèle.
— Finalisation de la rédaction du rapport.

Amélioration de la précision

Le perceptron multicouche est une architecture simple et flexible qui convient à différents types de problèmes, y compris la reconnaissance de chiffres manuscrits. Cette flexibilité permet d'ajuster l'architecture en modifiant le nombre de couches, le nombre de neurones par couche et les fonctions d'activation pour améliorer les performances. Les perceptrons multicouches sont efficaces pour résoudre des problèmes de classification et de régression grâce aux algorithmes de rétro-propagation. Leur capacité à travailler avec des fonctions complexes et non linéaires leur permet d'apprendre à partir de données d'entrée et de faire des prédictions précises. En somme, la simplicité et la flexibilité de l'architecture du perceptron multicouche en font une solution puissante pour divers types de problèmes d'apprentissage automatique.

L'ajustement du nombre de couches et de neurones et le choix de la fonction d'activation dans un réseau de neurones est une étape importante pour optimiser la précision du modèle. En effet, elle détermine la capacité du modèle à appréhender des structures complexes et à effectuer des prédictions pertinentes sur de nouvelles données.

1. Taux d'apprentissage

Le taux d'apprentissage (learning rate) est un hyper-paramètre crucial dans les algorithmes d'apprentissage supervisé, en particulier dans l'entraînement des réseaux de neurones. Concrètement, il détermine la taille des pas effectués lors de l'ajustement des poids du modèle pour minimiser la fonction de perte. Le taux d'apprentissage optimal dépend de plusieurs facteurs, tels que la complexité du modèle, les données d'entraînement et l'architecture du réseau de neurones. Il n'y a donc pas de valeur unique pour un "learning rate" optimal.
Nous avons alors testé plusieurs valeurs comme 0.001, 0.01, 0.1 et 0.15. Nous avons noté que les petites valeurs réduisent le risque de "sauts" trop importants lors de la mise à jour des poids. Elles peuvent en revanche rendre le processus d'apprentissage plus lent en provoquent des oscillations ou voire une divergence. Par conséquent, le modèle risque de rester coincé dans un minimum local. Nous avons alors fixé la valeur du taux d'apprentissage de notre réseau à 0.1 avant de procéder à l'optimisation de notre réseau.

2. Recherche du nombre de noeuds optimal

Dans cette partie du projet, on travaille d'abord sur le modèle de base qui contient une unique couche cachée. On cherche à déterminer le nombre de noeuds "optimal" qui assure le

meilleur taux de prédiction. Il n'y a pas de règle absolue pour déterminer le nombre de nœuds optimal, mais il existe différentes techniques telles que la validation croisée, la recherche de grille, ou la recherche aléatoire qui aident à trouver la meilleure configuration pour une tâche spécifique. Dans notre cas, nous avons opté pour la méthode de validation aléatoire suite au non aboutissement de la méthode de validation croisée.

Le nombre de neurones dans chaque couche cachée doit être ajusté en fonction de la complexité du problème et de la quantité de données disponibles. On part d'un nombre modéré de neurones par exemple 100 et on augmente progressivement le nombre de neurones jusqu'à ce que les performances cessent de s'améliorer.

Par ailleurs, il faut noter que choisir un nombre de nœuds trop faible peut conduire à un **sous-apprentissage** (underfitting), car le modèle n'est pas suffisamment complexe pour capturer les relations entre les caractéristiques d'entrée et les étiquettes de sortie. En d'autres termes, le modèle est trop simple ou insuffisamment entraîné pour bien généraliser à de nouvelles données.

D'un autre côté, un nombre de nœuds trop élevé peut conduire à un **surapprentissage** (overfitting), Ce phénomène se produit lorsque le modèle contient plus de paramètres par rapport au nombre d'entrées d'un problème dans notre cas il s'agit des 784 pixels de l'image. Si on fixe un nombre assez élevé de noeuds le réseau serait capable de capter le "bruit" de l'image et le considérer comme une caractéristique pertinente pour la classification. Dans le meilleur des cas, un réseau trop grand n'améliorera simplement pas la précision, mais dans le pire des cas, il pourrait la dégrader.

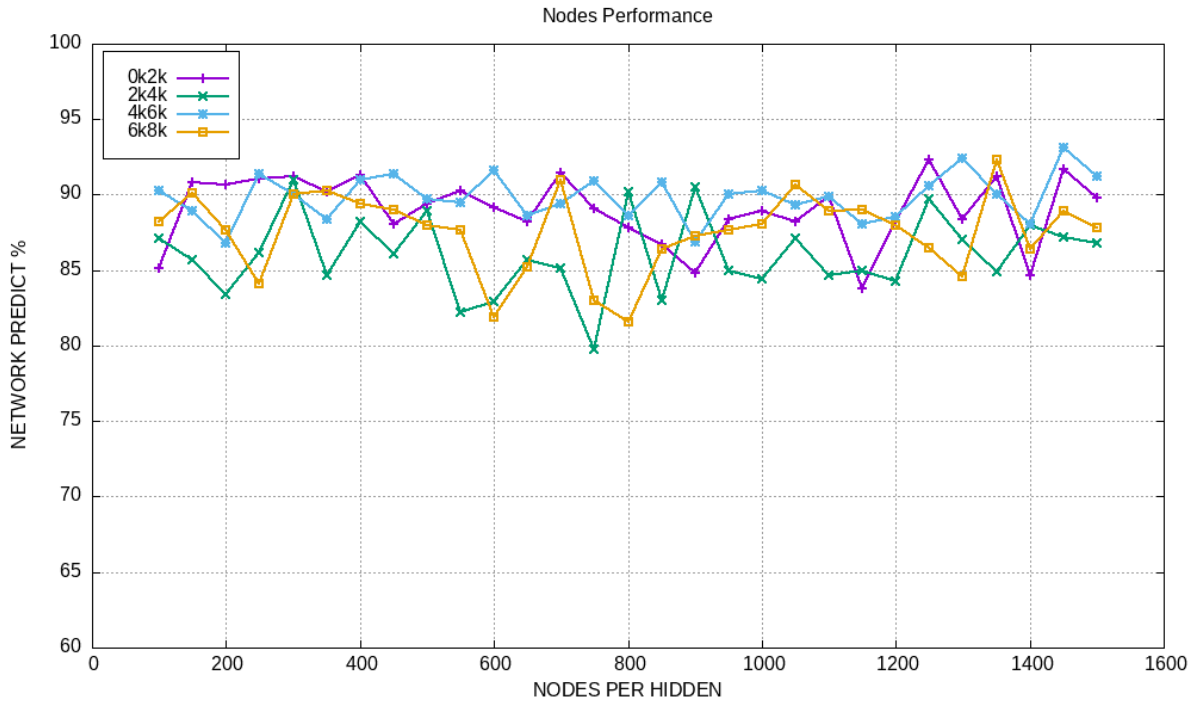

FIGURE 2 – Variation du taux de précision en fonction du nombre de neurones avec Sigmoïde

La figure ci-dessous montre la variation du taux de prédiction de notre réseau en fonction du nombre de noeuds sur la couche cachée pour 8000 images d'entraînement. Chaque courbe correspond à un intervalle de 2000 images sur les 8000 images d'entraînement. Les résultats obtenus nous montre que le modèle est globalement "stable" avec un taux qui varie entre 80 % et 93 %. En revanche, nous remarquons que les 4 courbes sont très proches et forment une zone d'intersection pour un nombre de noeuds égal à 300. Nous en déduisons alors que c'est une valeur optimale pour le nombre de noeuds sur la couche cachée mais peut-être pas l'unique.

3. Comparaison des fonctions d'activation

L'objectif principal des fonctions d'activation est de transformer les signaux d'entrée du neurone (c'est-à-dire les sorties pondérées des neurones précédents) en un signal de sortie. Cette sortie est ensuite transmise aux neurones de la couche suivante dans le réseau tout en maintenant les valeurs des neurones dans une plage contrôlée. Ainsi, on évite les problèmes d'"amplification", où les valeurs augmentent après chaque couche. C'est pourquoi les fonctions d'activation retournent généralement des valeurs dans l'intervalle [0, 1].

3.1 La fonction Sigmoïde

La fonction **sigmoïde** appelée également fonction logistique ou fonction en S est une fonction non linéaire utilisée dans les réseaux de neurones artificiels comme fonction d'activation pour les couches cachées des réseaux de neurones. Dans ce cas, elle introduit une non-linéarité dans le modèle et permet au réseau de modéliser des relations complexes entre les entrées et les sorties en estimant la probabilité d'appartenance à une classe. Elle est définie par l'équation suivante :

$$f_{sig}(x) = \frac{1}{1+e^{-x}} \tag{1}$$

Mise à part le coût élevée de la fonction sigmoïde. Celle-ci transforme les valeurs d'entrée en une plage de sortie comprise entre 0 et 1 et lorsqu'elle est utilisée pour la rétropropagation, la dérivée de la sigmoïde a tendance à être petite, en particulier pour des valeurs d'entrée très positives loin de zéro ou très négatives. Cela peut entraîner un phénomène appelé "vanishing gradient", où les gradients deviennent si petits qu'ils ne sont pratiquement pas mis à jour pendant l'entraînement, ralentissant ainsi la convergence et rendant l'apprentissage difficile pour les couches profondes du réseau.

3.2 La fonction ReLU

La fonction **ReLU : Rectified Linear Unit** utilisée également dans les réseaux de neurones artificiels. Elle est définie comme étant la fonction max(0, x), où x est l'entrée de la fonction. Celle-ci a plusieurs avantages par rapport aux à la sigmoïde. D'une part, elle est plus simple à calculer et moins coûteuse car elle ne nécessite pas l'utilisation de fonctions exponentielles ou logarithmiques. Voici l'équation de la fonction ReLU normalisée :

$$ReLU(v_i) = \frac{\max(0,v_i)}{\max_{1\leq i\leq |V|} v_i} \tag{2}$$

Cependant, la ReLu peut présenter certains inconvénients, telle que la "mort de neurones" ou "dying ReLu" qui se produit lorsque l'entrée est négative et que la sortie est donc nulle. Nous mettons simplement à 0 toutes les valeurs inférieures à 0 et les divisons par la valeur maximale de v (pour garantir que $\forall i, v_i \leq 1$). Le redimensionnement des valeurs en fonction de max(v) permet généralement d'éviter le phénomène de saturation. En termes de performance, le calcul de $max(0, v_i)$ est beaucoup moins coûteux que les exponentielles de σ, mais le $max(v_i, |V|i = 1)$ peut être assez coûteux, dans le cas où nous devrions chercher la valeur maximale d'un vecteur assez grand.

3.3 La fonction Softmax

La fonction **Softmax** est une fonction d'activation utilisée principalement dans les problèmes de classification. Elle convertit un vecteur d'entrée de scores réels en un vecteur de probabilités, où la somme des probabilités de toutes les classes est égale à 1. Celle-ci est

appliquée uniquement sur la couche de sortie.

$$S(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{3}$$

3.4 Analyse comparative des résultats

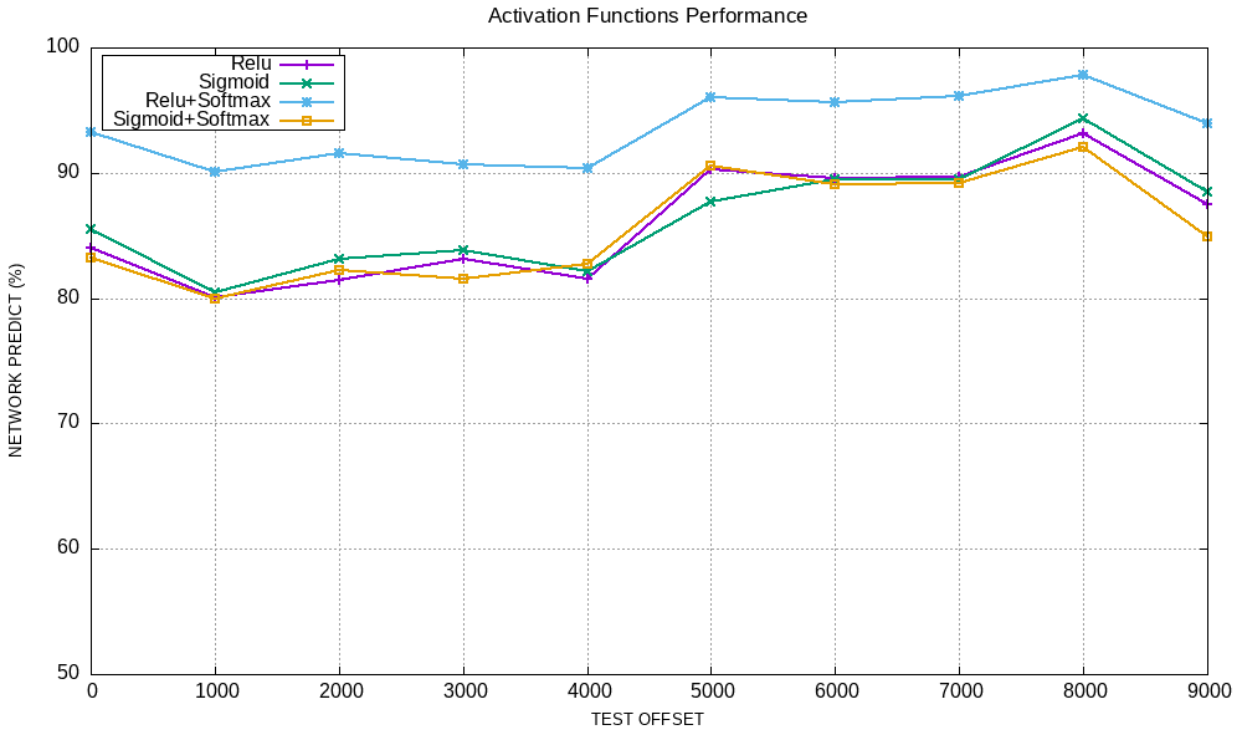

FIGURE 3 – Comparaison des performances des fonctions d'activation

La figure ci-dessus montre la variation du taux de prédiction de notre réseau en fonction des images testées. Nous avons divisé les 10 000 images de test en mini-batchs, chacun contenant 1 000 images. Nous avons utilisé 300 nœuds pour la couche cachée et entraîné le réseau avec 60 000 images. Chaque courbe correspond à une fonction d'activation.

Le graphe nous montre que, malgré la simplicité du jeu de données MNIST, l'application de la ReLu sur les couches cachées et de la fonction softmax sur le couche de sortie donne un taux de prédiction nettement meilleur. Comme mentionné précédemment la fonction ReLU a moins de problèmes de disparition du gradient que la fonction sigmoïde et la fonction Softmax permet de calculer la probabilité de chaque classe de sortie, ce qui facilite la comparaison entre les différentes sorties et permet une classification plus précise.

4. Implémentation d'une deuxième couche cachée

On implémente une couche cachée supplémentaire dans la version améliorée avec 300 noeuds sur la première couche cachée et 50 sur la deuxième dans l'optique de permettre au réseau de neurones d'apprendre des représentations plus complexes des données. Comme le montre la figure suivante :

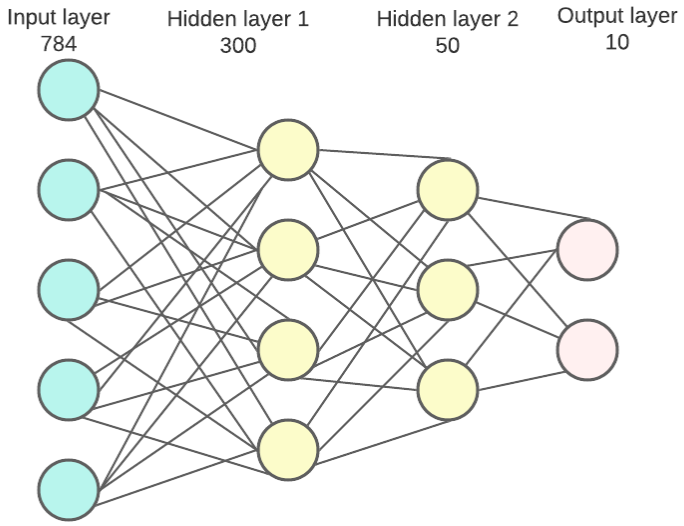

FIGURE 4 – Nouvelle structure du réseau de neurones avec 2 couches cachées

Cette approche peut améliorer les performances dans certains cas, mais peut également rendre le processus d'apprentissage un peu plus lent comme dans notre cas. Par ailleurs, le taux de prédiction du réseau avec deux couches cachées demeure toujours supérieur à celui d'un réseau avec une seule couche.

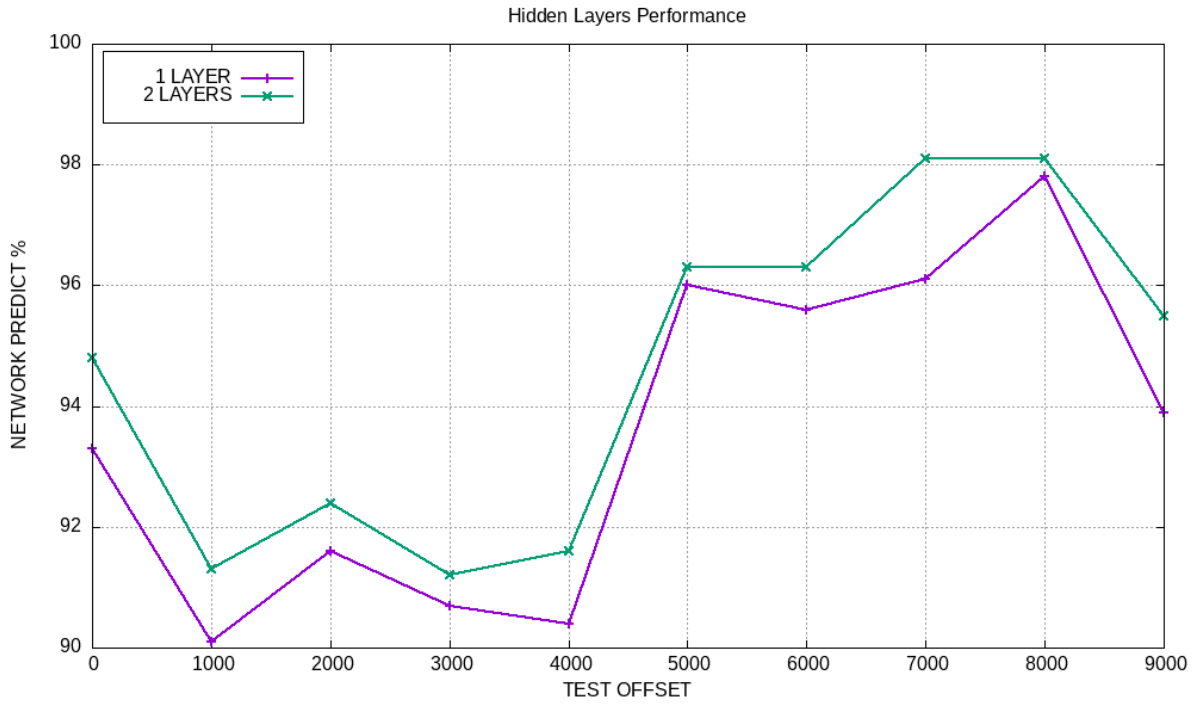

FIGURE 5 – Variation du taux de précision en fonction du nombre des images testées : une couche cachée Vs 2 couches cachées

La figure ci-dessus montre la variation du taux de prédiction en fonction des images testées. Nous avons testé un réseau de 300 nœuds avec une seule couche cachée et un réseau de 300 nœuds dans la première couche cachée et 50 nœuds dans la deuxième, les 2 réseaux sont et entraînés avec 60 000 images.

On remarque bien que celui de 2 couches cachées est le meilleur et on atteint le meilleur taux de prédiction sur l'échantillon [8000, 9000] du jeu de données du test. Ce qui montre que cet intervalle contient des images dont la complexité et les caractéristiques correspondent le mieux à la manière dont le réseau a été entraîné.

5. Phase d'entraînement

5.1 Entraînement du réseau

Algorithme 3 : Phase d'entraînement implémentée dans notre code :

procedure TrainNetwork($net, input\_data, output\_data, learning\_rate, n\_iterations$)
  for $i \leftarrow 1$ **to** $n\_iterations$ **do**
    for $(inputs, outputs) \in zip(input\_data, output\_data)$ **do**
      $feed\_forward(net, inputs)$
      $back\_propagate(net, outputs)$
      $gradient\_descent(net, learning\_rate)$
    end for
  end for
end procedure

5.2 Propagation avant (feed_forward)

Cette étape calcule les sorties pour chaque couche du réseau neuronal en utilisant les entrées, les poids et les biais des neurones, ainsi que les fonctions d'activation appropriées.

5.3 Rétropropagation (BackPropagate)

Cette étape calcule les erreurs et les gradients pour chaque couche du réseau. Pour la couche de sortie, les erreurs sont calculées comme la différence entre les sorties attendues et les sorties actuelles. Pour les autres couches, les erreurs sont calculées en utilisant les erreurs de la couche suivante et les poids des connexions entre les neurones. Les gradients sont ensuite calculés en multipliant les erreurs par les dérivées des fonctions d'activation.

Algorithme 4 : Rétropropagation implémentée dans notre code :

procedure BackPropagate($net, expected\_outputs$)
  for $L \in$ reversed layers of net **do**
    for $N \in L$ **do**
      if L is output layer **then**
        $error \leftarrow (expected\_outputs[N] - N.output)$
      else
        $error \leftarrow \sum(W * E$ for each $(W, E) \in zip(N.outgoing\_weights, next\_layer\_errors))$
      end if
      $N.delta \leftarrow error * activation\_derivative(N.output)$
    end for
  end for
end procedure

5.4 La descente de gradient

La descente du gradient est utilisée dans l'apprentissage des réseaux de neurones pour minimiser la fonction de coût. Celle-ci mesure la différence entre les prédictions du modèle et les valeurs réelles. Notre objectif est de trouver les paramètres : poids, biais et idéalement le taux d'entraînement qui minimiseront cette fonction de coût. L'algorithme fonctionne en calculant les gradients (dérivées partielles) de la fonction de coût par rapport aux paramètres du modèle. Ces gradients indiquent la direction dans laquelle la fonction de coût augmente le plus rapidement. Pour minimiser la fonction de coût, nous devons ajuster les paramètres du modèle dans la direction opposée aux gradients, c'est-à-dire dans la direction de la plus forte diminution. Il existe différentes variantes, telles que la descente du gradient stochastique ou la descente du gradient avec moment, qui permettent d'améliorer la convergence et la vitesse de l'algorithme.

La descente de gradient stochastique (SGD) : C'est l'approche la plus simple et la plus basique de la descente de gradient. À chaque itération, un seul exemple d'apprentissage est utilisé pour mettre à jour les poids du modèle. Cette méthode étant à la fois rapide et simple à mettre en œuvre, mais peut facilement tomber dans le phénomène du "surapprentissage" et être sensible au bruit ou irrégularités de l'image. Dans ce cas, le réseau obtenu risque d'être entraîné à correspondre de manière très proche au jeu de données d'entraînement utilisé, produisant ainsi plus d'erreurs lors de la réalisation de prédictions sur un autre jeu de données différent de celui de la phase d'entraînement. Le gradient de la fonction coût résultante sera probablement très élevé ce qui nous amènera à ajuster les paramètres du réseau dans la mauvaise direction. Par conséquent le réseau risque de prendre plus de temps pour converger vers une solution optimale ce qui dégradera la performance de notre réseau en général.

Algorithme 5 : Descente de gradient Stochastique - Basique :

Données : La fonction f, Paramètres initiaux θ, Jeu d'entraînement T, nombre d'epochs e, taux d'apprentissage η

Résultats : Paramètres optimisés θ

for `i=1 to e do` **do**
/* Itération sur le jeu de données */
  for `j=1 to e do` **do** :
    $\theta \leftarrow \theta - \eta \nabla f(\theta, \mathrm{B}_j)$
  end for
end for

Pour l'implémentation de notre réseau, nous nous sommes inspiré de cet algorithme pour implémenter la descente de gradient qui s'applique lors de la rétropropagation des erreurs

pour ajuster les poids et les biais des couches cachées et de la couche de sortie. On l'utilise lors de la mise à jour des poids et des biais dans la fonction train_network.

Algorithme 6 : La descente de gradient implémentée dans notre code :

procedure GradientDescent($net, learning\_rate$)
  for $L \in$ layers of net except input layer **do**
    for $N \in L$ **do**
      for $(W, B) \in$ (weights and biases connected to N) **do**
        $gradient\_W \leftarrow N.delta * connected\_neuron\_output$
        $gradient\_B \leftarrow N.delta$
        $W \leftarrow W + learning\_rate * gradient\_W$
        $B \leftarrow B + learning\_rate * gradient\_B$
      end for
    end for
  end for
end procedure

Dans cette étape de la phase d'entraînement met à jour les poids et les biais de chaque neurone en utilisant les gradients calculés lors de la rétropropagation et le taux d'apprentissage spécifié. Pour chaque neurone, les poids et les biais sont mis à jour en ajoutant le produit du taux d'apprentissage et du gradient correspondant.
Cependant, il est important de noter que la descente du gradient a également des inconvénients, tels que la sensibilité au choix du taux d'apprentissage, la possibilité de tomber dans des minima locaux et la lenteur de la convergence dans certaines situations. Il existe plusieurs variantes de la descente de gradient, chacune ayant ses propres avantages et inconvénients. Nous avons exploré quelques unes de ces méthodes uniquement en théorie pour l'instant mais qu'on envisage de tester plus tard :

- **Momentum** : La descente de gradient avec moment , une méthode qui ajoute une composante de "moment" aux mises à jour de poids, ce qui permet d'accélérer la convergence et de réduire les oscillations où le moment est un hyperparamètre qui doit être ajusté.

- **Adagrad** : La descente de gradient adaptative qui ajuste dynamiquement le taux d'apprentissage pour chaque paramètre du modèle en fonction de l'historique des gradients. Cela permet d'accélérer la convergence et de réduire la sensibilité aux hyperparamètres.

6. Étude de la précision dans le temps

Lorsque nous entraînons un réseau de neurones, nous ajustons les poids et les biais du réseau en utilisant la rétropropagation de l'erreur pour minimiser la fonction de coût. Cependant, l'ajustement des poids et des biais ne garantit pas que la précision du réseau augmente constamment au fil du temps et qu'elle se stabilise au bout d'un moment.

Ainsi, il est utile de suivre l'évolution de la précision du réseau pendant la phase d'apprentissage en effectuant périodiquement des phases de prédiction avec un ensemble de données de test. Pour éviter de tester toutes les données à chaque itération, nous pouvons diviser les données de test en mini-batch de 100 images. Après chaque mini-batch, nous enregistrons la précision du réseau et traçons les valeurs pour visualiser la convergence de la précision au fil du temps.

Cela nous permet de savoir si notre modèle s'améliore au fil de l'apprentissage et d'observer la vitesse à laquelle la précision du réseau converge comme le montre la courbe ci-dessous :

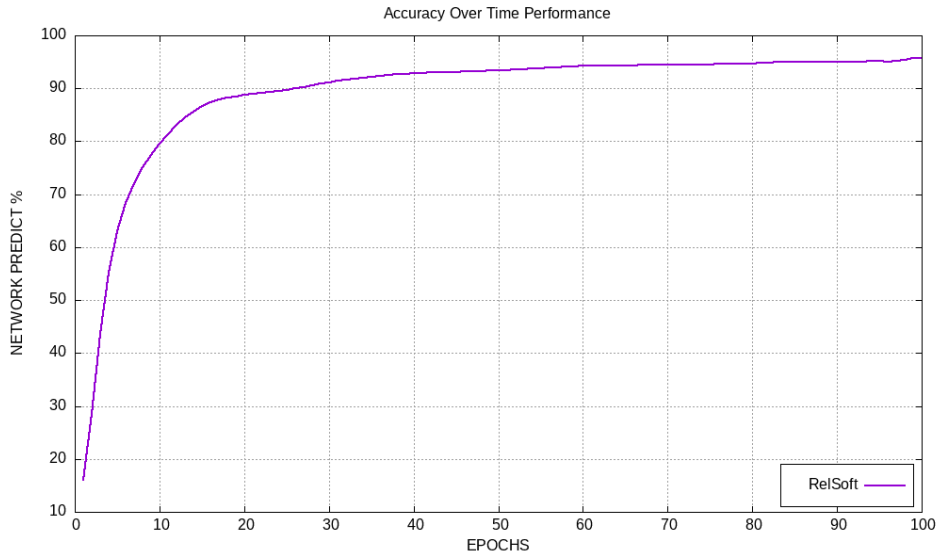

FIGURE 6 – Variation du taux de précision en fonction du nombre des "epochs"

Le graphe indique que la précision du réseau s'améliore rapidement au début de l'entraînement et se stabilise après environ 40 époques d'apprentissage. Passé ce point, on observe peu de progrès dans la précision du réseau, ce qui suggère que le modèle a atteint un niveau de performance optimal. Cette courbe nous donne une idée de la vitesse à laquelle notre modèle apprend et converge vers une solution optimale pour la reconnaissance de chiffres manuscrits sur le MNIST dataset.
Dans le cas présent, puisque la précision du réseau se stabilise après environ 40 époques d'apprentissage, on pourrait décider d'arrêter l'entraînement à ce stade pour éviter le sur-

apprentissage. Le "early stopping" nécessite généralement la validation croisée pour évaluer la performance du modèle sur un ensemble de données de validation distinct de l'ensemble d'entraînement, afin de déterminer le moment optimal pour arrêter l'apprentissage.

7. Bonus : Le prétraitement du jeu de données

- **Le redimensionnement :** Les images sont déjà de taille 28x28, on a donc pas eu besoin de les redimensionner.

- **La centralisation et la mise en échelle :** La centralisation consiste à soustraire la moyenne de tous les pixels de l'image de chaque pixel de l'image. Cela permet d'obtenir une image centrée sur zéro, ce qui facilite l'apprentissage, La mise à l'échelle consiste à diviser chaque pixel de l'image par l'écart type des pixels de l'image. Cela permet aux modèles de classification de mieux prendre en compte les différences de luminosité et de contraste dans les images.

- **L'application des filtres :** Celle-ci permet d'améliorer la qualité des images, extraire des caractéristiques importantes ou réduire le bruit. Nous avons testé le filtre passe-bas sur l'ensemble des images,en revanche nous n'avons pas constaté une grande amélioration en terme de temps et de taux de précision. Contrairement à la centralisation et la mise en échelle, qui ont apporté une amélioration comme sur la figure ci-dessous :

FIGURE 7 – L'affichage d'une image représentant le digit 0 après le traitement

- **La normalisation :** Les valeurs de chaque Pixel doivent généralement être normalisées pour avoir une moyenne de 0 et un écart-type de 1. Cela permet de réduire les effets des différences d'intensité d'image et de faciliter la convergence des algorithmes d'apprentissage.

Nous avons des fichiers binaires MNIST pour charger les données dans des tableaux d'images et de labels, chaque image a son étiquette. Voir la figure suivante qui représente l'affichage de l'une des image obtenue après l'avoir extrait depuis les fichiers binaires. Nous avons par la suite calculé la moyenne et l'écart type des pixels dans l'ensemble des images d'entraînement et de test, afin de centrer et mettre en échelle les images dans l'ensemble des images d'entraînement et de test Enfin, les données sont maintenant prêts pour la prochaine étape qui consiste à les entraîner avec un réseau de neurones et enfin les tester.

Amélioration de la performance

L'objectif de cette partie du rapport est d'examiner et de discuter les différentes approches et techniques mises en œuvre pour améliorer les performances de notre modèle de réseau de neurones. Nous nous concentrerons sur les méthodes de parallélisation et les différentes techniques pour optimiser l'apprentissage et la vitesse d'exécution de notre modèle. Nous aborderons l'utilisation de bibliothèques optimisées, telles que CBLAS et OpenMP, pour tirer parti des capacités de parallélisation et de calcul des systèmes modernes.

Analyse et Exploration des Stratégies de Parallélisation

1. Architecture x86-64 :

Model	Cores	Max Freq (GHz)	L1 (KiB)	L2 (KiB)	L3 (MiB)
AMD Ryzen 5 5500U	6	4.0	32	512	4

2. Mesure de performance de la version séquentielle

Avant d'envisager d'améliorer les performances, il est important d'analyser les performances et la complexité du code actuel en terme de temps d'exécution et de précision, d'identifier les parties du code qui prennent le plus de temps à s'exécuter (boucles, fonctions ou calculs). Pour choisir on choisira la meilleure approche de parallélisation adaptée à notre environnement, il existe plusieurs outils de profilage tels que :

- L'environnement d'analyse et d'optimisation de performance **MAQAO** (Modular Assembly Quality Analyzer and Optimizer) : un outil de performance et d'optimisation développé pour les applications basées sur les architectures x86-64 et ARM. Il se concentre principalement sur l'analyse et l'optimisation du code assembleur généré par les compilateurs. MAQAO fournit des informations détaillées sur les performances du code et les problèmes potentiels, tels que les goulots d'étranglement, les dépendances de données et les optimisations manquantes.

- **gprof** : Un outil de profilage basé sur GNU pour les programmes en C, C++ et Fortran. Il fournit des informations détaillées sur le temps d'exécution de chaque fonction

et l'appel des fonctions.

- **Valgrind** : Un outil d'analyse dynamique qui inclut un profileur de cache et un "profiler" d'exécution (appelé Callgrind) qui nous permet d'analyser en profondeur l'exécution de votre programme, y compris les performances de cache et les relations d'appel entre les fonctions.

En raison de contraintes de temps et d'une maîtrise insuffisante de l'outil MAQAO, nous avons choisi de nous concentrer sur le profilage de notre code en utilisant gprof pour le moment.

Time %	cumulative(sec)	self(sec)	calls	self(ms/call)	Total(ms/call)	name
100.00	0.72	0.72	3000	0.24	0.24	dotprod
0.00	0.72	0.00	4000	0.00	0.00	copy_mat
0.00	0.72	0.00	3000	0.00	0.00	add
0.00	0.72	0.00	3000	0.00	0.00	check_dimensions
0.00	0.72	0.00	2002	0.00	0.00	free_mat
0.00	0.72	0.00	2000	0.00	0.00	apply
0.00	0.72	0.00	2000	0.00	0.00	mat_max
0.00	0.72	0.00	2000	0.00	0.00	scale
0.00	0.72	0.00	1000	0.00	0.00	mat_argmax
0.00	0.72	0.00	1000	0.00	0.72	predict_network
0.00	0.72	0.00	1000	0.00	0.00	softmax

TABLE 1 – Tableau d'analyse du code séquentiel avec l'outil GNU profiler : gprof

Parallélisation et optimisation des performances

1. Optimisation des opérations élémentaires

Une manière simple de commencer à paralléliser notre code sans modifier l'algorithme global consiste à effectuer des optimisations au niveau des opérations élémentaires, par exemple sur la partie opération sur les matrices. Pour ce faire, nous commençons par exploiter les bibliothèques optimisées.

BLAS : Basic Linear Algebra Subprograms Une collection de routines de bas niveau pour effectuer des opérations d'algèbre linéaire courantes, telles que l'arithmétique vectorielle et matricielle, les produits scalaires et la multiplication matricielle. CBLAS fournit donc un ensemble de fonctions C qui appellent les routines sous-jacentes de BLAS. Ces routines fournissent des directives de compilation qui permettent de qualifier des sections de code à exécuter en parallèle.

DGEMM : "Double precision General Matrix-Matrix Multiplication" C'est une

fonction de la bibliothèque CBLAS qui implémente en langage C des routines de calcul d'algèbre linéaire basique. Cette fonction effectue la multiplication de deux matrices en double précision, avec une syntaxe simplifiée et optimisée pour les calculs matriciels.

Comme notre code utilise fréquemment la fonction DOTPROD pour calculer le produit matriciel, cette dernière est responsable d'une grande partie du temps d'exécution. Afin d'améliorer les performances, nous allons remplacer la fonction dotprod par la fonction DGEMM fournie par CBLAS, qui est optimisée pour les calculs de produits matriciels. Nous allons ensuite comparer les temps d'exécution avant et après cette modification pour évaluer l'impact de cette optimisation sur les performances de notre code.

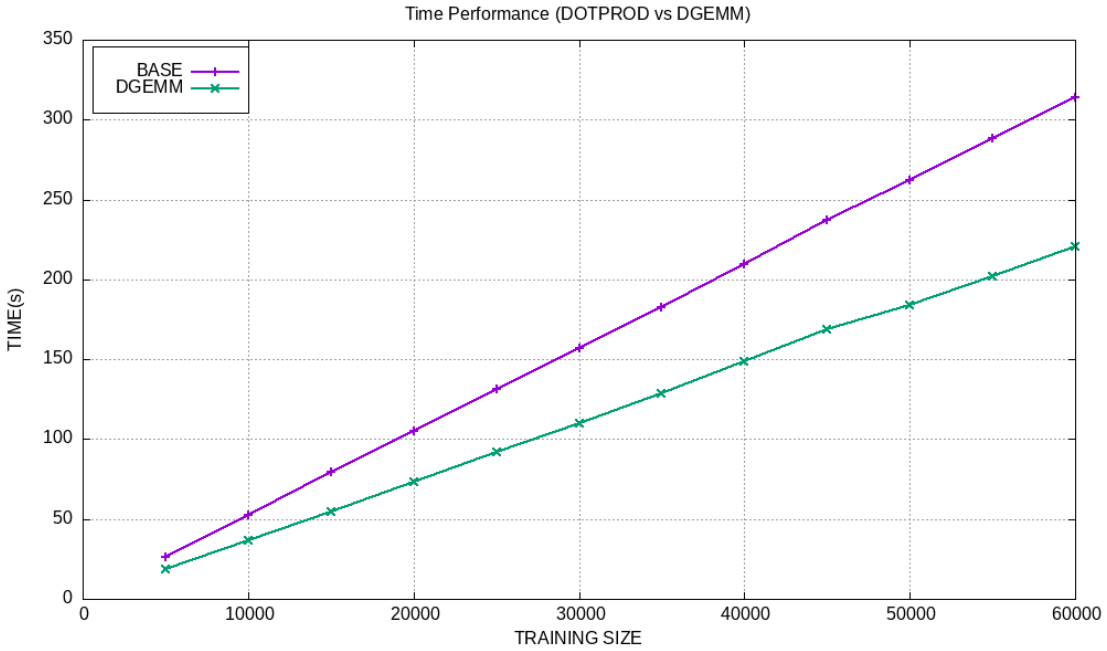

FIGURE 8 – Comparaison de la version de base (DOTPROD) et la version optimisée (DGEMM)

Le graphique ci-dessus illustre la variation du temps d'exécution de la phase d'entraînement du réseau de neurones en fonction du nombre d'images traitées. Deux courbes sont représentées sur le graphique : la première présente la variation du temps d'exécution en fonction du nombre d'images entraînées en utilisant la fonction DOTPROD et la seconde en utilisant la fonction DGEMM. Nous remarquons une augmentation de 30% de temps d'exécution dans la phase d'entraînement de notre réseau de neurones. Nous en déduisons alors que l'utilisation de fonctions d'algèbre linéaire optimisées telles que DGEMM améliore dans notre cas considérablement le temps d'exécution des réseaux de neurones.

Ensuite, nous avons remplacé toutes nos fonctions qui effectuent des opérations matricielles par CBLAS afin de voir une nouvelle fois la différence de performance. La figure

ci-dessous montre les résultats obtenues.

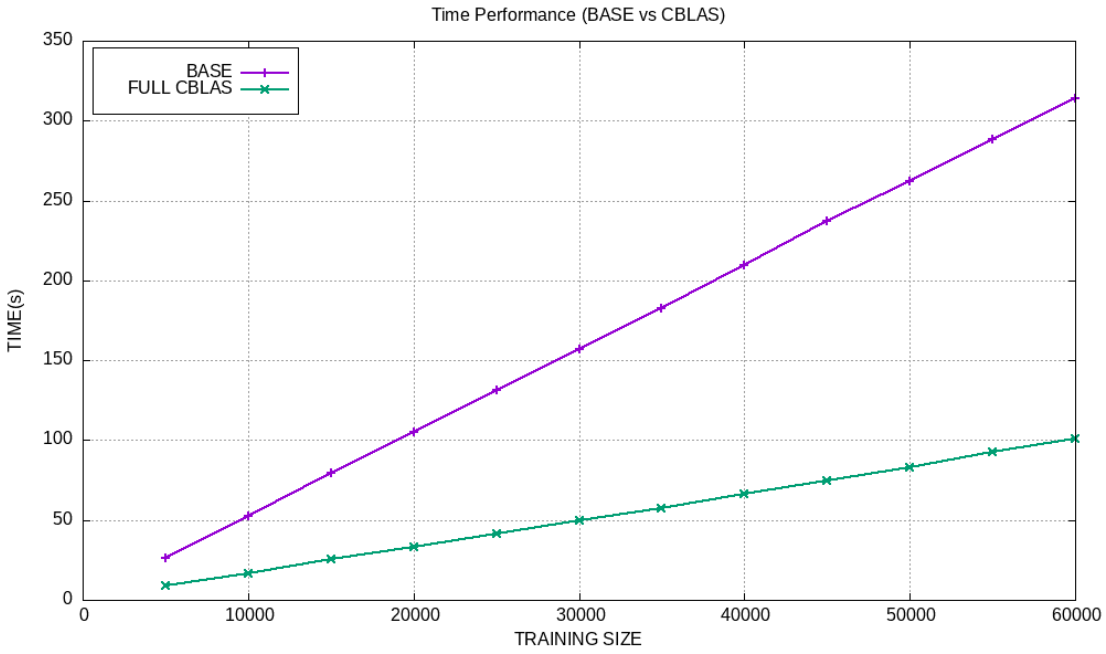

FIGURE 9 – Comparaison de la version de base et la version utilisant CBLAS sur toutes les opérations matricielles

En observant le deuxième graphique ci-dessus, on constate que la temps d'éxecution du code en utilisant CBLAS sur toutes les opérations matricielles dans notre code, est nettement plus réduit que celui de la version de base. Ceci signifie que l'utilisation de CBLAS a permis de réduire le temps d'exécution de l'entraînement de notre réseau de neurones. En effet, la bibliothèque CBLAS est conçue pour tirer parti des caractéristiques spécifiques de l'architecture matérielle sur laquelle elle est exécutée. Cela inclut l'utilisation d'instructions SIMD (Single Instruction, Multiple Data) pour effectuer des calculs parallèles sur de multiples données simultanément, ainsi que l'exploitation des hiérarchies de mémoire cache pour réduire les coûts d'accès à la mémoire. Cette exploitation de l'architecture matérielle conduit à une accélération significative des opérations d'algèbre linéaire.

Il est important de noter que la réduction du temps d'exécution peut varier en fonction de la complexité du réseau de neurones et de la taille des données d'entraînement. Cependant, dans la plupart des cas et dans notre cas concrètement, l'utilisation de bibliothèques d'algèbre linéaire optimisées comme CBLAS améliore considérablement le temps d'exécution de la phase d'entraînement des réseaux de neurones.

2. Un format à virgule flottante

Un format à virgule flottante est un système de représentation de nombres réels sur un ordinateur, conçu pour stocker des nombres comprenant à la fois une partie entière et une partie fractionnaire. Ce qui s'avère particulièrement utile pour l'optimisation des algorithmes numériques, car il permet de représenter des nombres réels avec une précision adéquate tout en étant stocké sous une forme compacte.

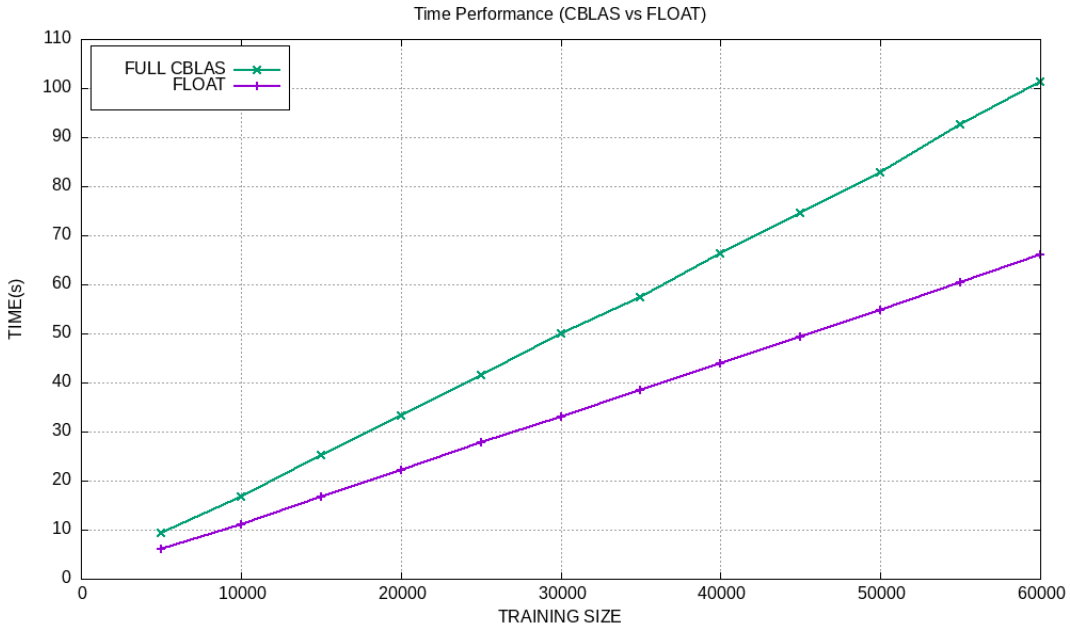

FIGURE 10 – Comparaison de la dernière version optimisée avec CBLAS (format double) et la version CBLAS (format float)

La figure ci-dessus montre que le temps d'exécution de la phase d'entraînement de notre réseau de neurones en version CBLAS avec un format de point flottant (sur 32 bits) au lieu d'un format double (sur 64 bits) est nettement réduit. Ceci implique l'utilisation de SGEMM au lieu de DGEMM et ainsi de suite pour le reste des fonctions appelées depuis CBLAS. En effet, ce format réduit l'utilisation accrue de la mémoire et assure une consommation d'énergie moins importante. Ce qui assure une exécution plus rapide des opérations mathématiques nécessaires pour l'entraînement de notre réseau de neurones

Le choix d'un format à virgule flottante approprié permet alors de trouver un équilibre entre la précision des calculs et l'efficacité de la mémoire et du temps de calcul. Ce compromis est essentiel pour assurer des performances optimales des algorithmes et des applications qui en dépendent.

3. Utilisation des directives OpenMP

On utilise les directives de OpenMP pour la parallélisation du code. Cette bibliothèque garantie le partage des tâches de calcul sur plusieurs processus légers afin d'accélérer l'exécution du code.

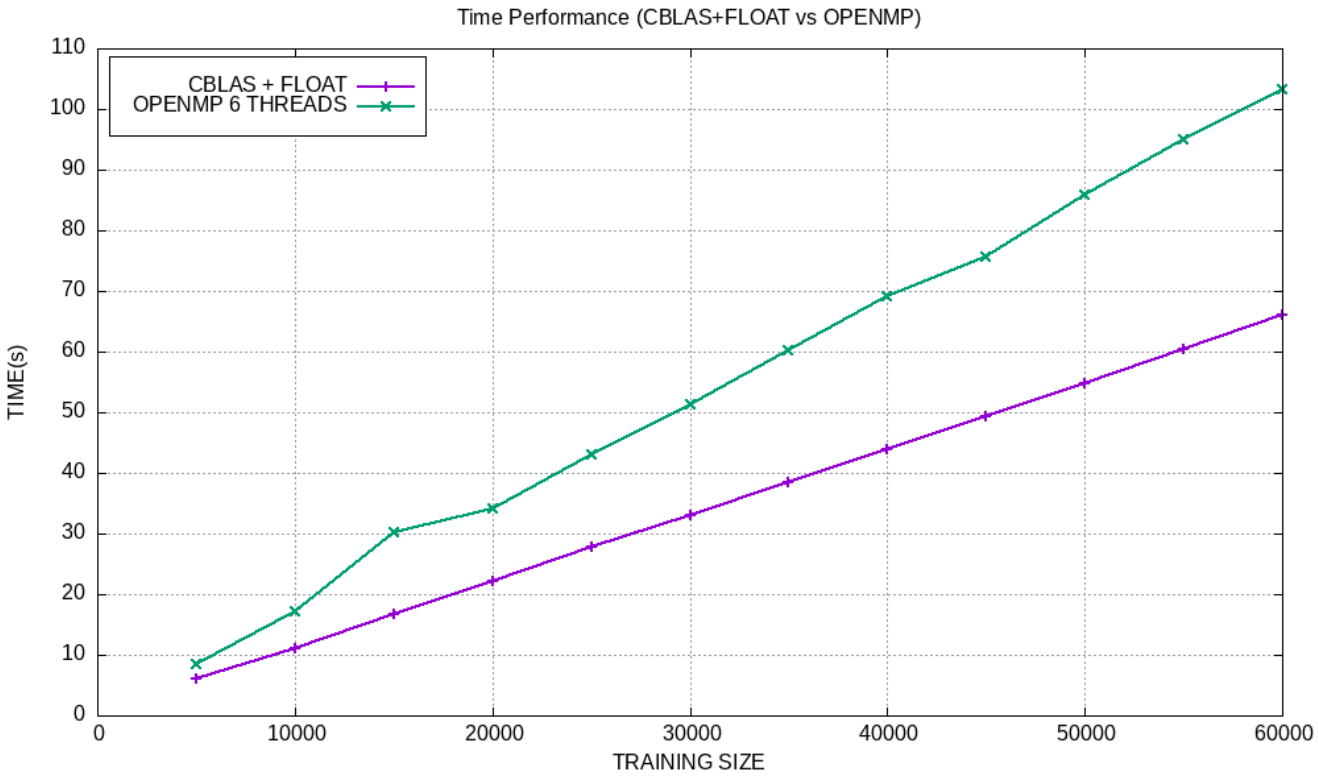

FIGURE 11 – Comparaison de la version CBLAS avec le format float et la version OpenMP

Nous avons mesuré le temps d'exécution du code d'entraînement de notre réseau de neurones en utilisant le format à point flottant et la bibliothèque OpenMP, nous avons constaté que l'utilisation format à virgule flottante est légèrement meilleure que celle avec OpenMP, cela peut indiquer que les gains de performance apportés par la parallélisation avec OpenMP sont moins importants que ceux obtenus avec le "floating point format".
En effet, la parallélisation ne garantit pas toujours une amélioration significative des performances, en particulier dans les situations où la communication entre les threads ou les tâches parallèles est un goulot d'étranglement ou lorsque la charge de travail n'est pas uniformément répartie entre les threads. Nous pouvons dire aussi que, vu la petite taille de notre réseau de neurones, la parallélisation n'était pas vraiment efficace. Par ailleurs, il est souvent inutile de mettre plus de threads OpenMP que ce que nous en avons sur notre machine. Cela entraînera le processeur à effectuer du scheduling entre les threads et à changer de contexte à chaque fois, augmentant ainsi la surcharge liée au parallélisme. Il serait donc préférable de réaliser

des tests plus approfondis et d'analyser les résultats en détail.

THREADSNUM	TRAINING SIZE	NODES	TRAINING TIME	NETWORK PREDICT
2	40000	300	107.02	93.5
4	40000	300	103.12	92.5
6	40000	300	102.82	94.5
8	40000	300	107.80	93.7
10	40000	300	113.67	93.5
12	40000	300	118.89	93.5

TABLE 2 – Données relatives aux threads, à la taille de l'entraînement, aux nœuds, au temps d'entraînement (s) et à la prédiction du réseau (%).

Comparaison de temps d'éxecution de toutes les versions optimisées avec la version de base :

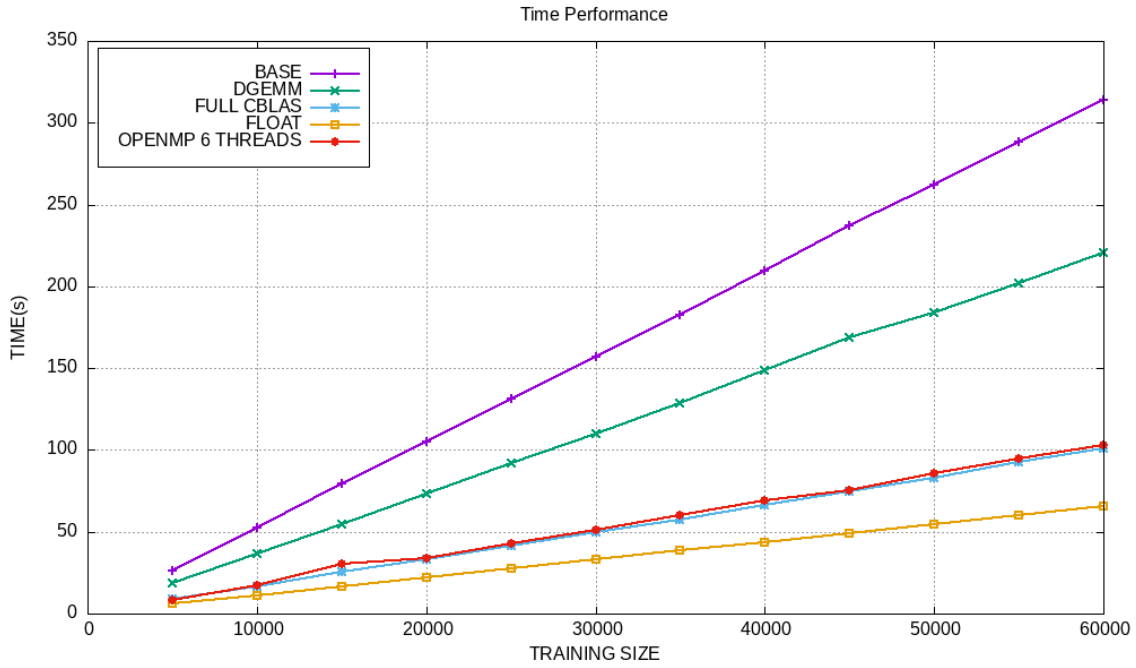

FIGURE 12 – Comparaison de toutes les pistes d'optimisation explorées

Ce graphique montre une nette amélioration des performances grâce aux différentes optimisations apportées au code. La combinaison CBLAS et le format de virgule flottante (float) offre le meilleure temps d'exécution parmi les techniques testées (courbe jaune) indiquant une gestion optimale de la mémoire et témoignant de l'efficacité des fonctions de calcul matriciel optimisées. Ensuite, l'implémentation avec l'utilisation des directives de OpenMP se situe en deuxième position en termes de rapidité d'exécution presque confondue avec la courbe de CBLAS.

Algorithme 7 : Parallélisation du code d'entraînement du réseau de neurones :

```
procedure ENTRAINEMENT(net, images, labels, nbr_img, Relu)
    #pragma omp parallel for shared(net, images, labels, size, Relu)
    for i = 0 à nb_img do
        création_matrice() ;
        #pragma omp parallel for shared(matrice, images)
        for j = i * 784 à j < (i + 1) * 784 do
            matrice[] = images[j] ;
        end for
        index = labels ;
      creation_matrice_output(10, 1) ;
      output(index)=1.0
        #pragma omp critical
            train_network(net, matrice, output, Relu)
    end for
end procedure
```

Algorithme 8 : Parallélisation du code qui test le réseau de neurones :

```
function TEST( net, images, labels, nbr_img, Relu, img_correct = 0 )
    #pragma omp parallel for shared(net, images, labels, size, Relu)
    for i = 0 à nb_img do
        création_matrice() ;
        #pragma omp parallel for shared(matrice, images)
        for j = i * 784 à j < (i + 1) * 784 do
            matrice[] = images[j] ;
        end for
        index = labels ;
      creation_matrice_Prediction ;
        #pragma omp critical
            matrice_prediction = Predict_netwok(net, matrice) ;
        if max(matrice_prediction)==index then
            img_correct + + ;
        end if
    end for
return((1.0 * img_correct / nb_img) * 100 ;
end function
```

Conclusion et perspectives

Conclusion

Les améliorations de performance jouent un rôle crucial dans la mise en œuvre et le déploiement de modèles de réseaux de neurones dans des environnements réels et des applications de production. Un modèle plus rapide et plus efficace permet de traiter de plus grandes quantités de données, d'obtenir des résultats plus rapidement et de réduire les coûts de calcul. Cela peut avoir un impact significatif sur la compétitivité et la pertinence d'un modèle ou d'une solution basée sur l'intelligence artificielle.

Perspectives

Adaptation du taux d'apprentissage Une amélioration supplémentaire dans la partie de l'optimisation de la descente de gradient serait d'adapter le taux d'apprentissage (learning rate) au fil des itérations. Un taux d'apprentissage constant peut poser problème car il est probablement trop petit au début de la phase d'apprentissage et trop grand à la fin. On pourra envisager d'introduire une décroissance du learning rate pour le rendre plus petit après chaque application de la rétropropagation.
Un taux d'apprentissage adaptatif permet d'appliquer des mises à jour plus importantes au début de la phase d'apprentissage, puis de passer progressivement à des corrections plus petites et plus précises à mesure que les paramètres du réseau se rapprochent de leur valeur finale.

Une descente de gradient par mini-lots : Cette variante utilise un sous-ensemble d'exemples d'apprentissage (un mini-lot) pour mettre à jour les poids du modèle à chaque itération. Cela permet de réduire la variance des mises à jour et d'accélérer la convergence. La taille du mini-lot est un hyperparamètre qui doit être ajusté.

Réaliser des mesures de performances plus approfondies sur des machines plus performantes et plus stables.

Références

Goodfellow, I., Bengio, Y., Courville, A. (2016).
Deep Learning. MIT Press. http://www.deeplearningbook.org

LeCun, Y., Bengio, Y., Hinton, G. (2015).
Deep learning. Nature, 521(7553), 436-444.

Kingma, D. P., Ba, J. (2014).
Adam : A method for stochastic optimization. arXiv preprint arXiv :1412.6980.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., ... Ng, A. Y. (2012).
Large scale distributed deep networks. In Advances in neural information processing systems (pp. 1223-1231).

OpenMP Architecture Review Board. (2013).
OpenMP Application Program Interface Version 4.0.
http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986).
Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

Nwankpa, C., Ijomah, W., Gachagan, A., Marshall, S. (2018).
Activation functions : Comparison of trends in practice and research for deep learning. arXiv preprint arXiv :1811.03378.

LeCun, Y., Cortes, C., Burges, C. J. (2010).
MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/

Gonzalez, R. C., Woods, R. E. (2017). Digital image processing.
Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv :1609.04747.

Blackford, L. S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., ... Walker, D. (2002).
An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathe-

matical Software (TOMS), 28(2), 135-151.

BLAS Documentation. https://netlib.org/blas/

GNU profiler. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html$_node/gprof_1.html$