Projet PPN: Estimateur de performance de code assembleur par intelligence artificielle - Amélioration de la précision, performance, et parallélisation

BAUMANN Pierre
Master 1 CHPS

DARDILL Hugo
Master 1 CHPS

DELMAY Arthur
Master 1 CHPS

Résumé
Notre projet consiste à construire un réseau de neurones dont le but va être de prédire la performance de noyaux de calcul en nombre de cycles CPU.
Lors du premier semestre, nous avons établi les bases de notre réseau de neurones, avec un ensemble de données réduit.
Pour ce second semestre, nous avons comme but d'entrainer ce réseau sur un ensemble plus complet, d'en améliorer sa précision, d'améliorer sa performance en temps de calcul, et de le paralléliser.

Table des matières

1. Introduction

Dans ce rapport, nous allons dans un premier temps faire quelques rappels brefs sur les réseaux de neurones, et introduire certains termes, afin de bien pouvoir comprendre certains notions abbordées.
Aussi, nous allons parler brièvement de notre jeu de données, ainsi que de son extraction. (Vous pouvez passer ces parties si vous avez déjà lu le rapport du semestre dernier).

Et aussi, nous allons présenter un schéma pour un entrainement afin de vous aidez à interpréter nos résultats.

Ensuite, nous allons parler des différentes approches que nous avons utilisé pour améliorer notre modèle.
Par amélioration, nous pouvons parler de la précision du modèle, de la performance en temps de calcul, ou de la parallélisation des calculs.
Nous allons d'abord nous focaliser sur l'amélioration de la précision, puis sur l'amélioration de la performance et de la parallélisation des calculs.
Comme certaines approches peuvent améliorer plusieurs aspects en même temps, nous avons indiqué pour chaque approche quel(s) aspect(s) elle améliore(nt).

Pour finir, nous allons présenter nos résultats, et en discuter.
Puis nous ferons une conclusion sur ce projet tout au long de l'année.

2. Quelques rappels sur les réseaux de neurones

Si vous avez déjà lu le rapport du semestre dernier, vous pouvez passer cette partie.

2.1. Principe

Un réseau de neurones est un approximateur universel, c'est à dire qu'il peut approximer n'importe quelle fonction, avec une précision arbitraire, à condition d'avoir assez de neurones et de données. Il se calque sur le fonctionnement du cerveau humain : il prend des données en entrée, et il les traite à l'aide de connexions internes, et il renvoie d'autres données en sortie.

On le construit grâce à plusieurs couches de neurones, chaque neurone étant relié à tous les neurones de la couche précédente.

- La couche d'entrée, c'est la toute première couche où les données initiales sont introduites dans le réseau.
- Une ou plusieurs couches cachées, qui traitent et transforment les données grâce à des liens pondérés entre les neurones, afin de pouvoir trouver des relations complexes entre les données.
- La couche de sortie, qui produit le résultat final, et est donc notre prédiction.

On peut modéliser un tel comportement grâce à l'algèbre linéaire et la multiplication matricielle. Notre entrée est un vecteur. Chaque couche est une matrice de poids, qui est multipliée par le vecteur d'entrée, et qui produit un nouveau vecteur. Chaque neurone est un produit scalaire entre le vecteur d'entrée et le vecteur de poids du neurone, plus un biais. Chaque neurone applique une fonction d'activation sur le résultat du produit scalaire, afin de casser la linéarité du réseau de neurones, et de lui permettre d'apprendre des relations complexes entre les données.

Voici notre entrée, qui est un vecteur de réels :

$$e = \begin{pmatrix} e_1 \\ e_2 \\ \dots \\ e_n \end{pmatrix}$$

Pour passer à la couche suivante, on applique une matrice de poids

$$W = \begin{pmatrix} w_{11} & w_{12} & ... & w_1 n \\ w_{21} & w_{22} & ... & w_2 n \\ \vdots & \vdots & \ddots & \vdots \\ w_{\mathrm{m1}} & w_{\mathrm{m2}} & ... & w_{\mathrm{mn}} \end{pmatrix}$$

qui correspond au poids de la relation de chaque neurone de la couche précédente vers chaque neurone de la couche suivante, et un vecteur de biais

$$b = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}$$

Cela nous donne un nouveau vecteur de valeurs

$$c = e \times W + b = \begin{pmatrix} c_1 \\ c_2 \\ ... \\ c_n \end{pmatrix}$$

Ensuite on passe ce vecteur dans une fonction sur chaque valeur du vecteur, afin de transformer la valeur de chaque neurone.

$$\begin{pmatrix} c_1 \\ c_2 \\ ... \\ c_n \end{pmatrix} \rightarrow \begin{pmatrix} \sigma(c_1) \\ \sigma(c_2) \\ ... \\ \sigma(c_n) \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_n \end{pmatrix}$$

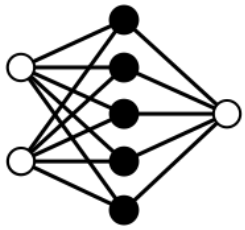
Et on répète ce processus avec l'entrée de la prochaine couche qui est maintenant le vecteur a jusqu'à la couche de sortie.

2.2. Quelques termes

Voici quelques termes que nous allons utiliser tout au long de ce rapport.

- Topologie du réseau de neurones

La topologie d'un réseau de neurones est la suite des tailles de ses différentes couches. Par exemple, le réseau de neurones suivant :



possède une topologie de (2, 5, 1), c'est à dire qu'il a 2 neurones en entrée, une couche cachée de 5 neurones, et 1 neurone en sortie.

- Fonction d'activation

Une fonction d'activation est une fonction mathématique qui est appliquée à chaque neurone d'un réseau de neurones artificiels. Elle permet de décider de la valeur de sortie du neurone. Si cette fonction est non linéaire, elle permet de rendre le réseau de neurones non linéaire, et donc de prédire des patterns plus complexes. On la note souvent σ(x)

3. Le jeu de données

Si vous avez déjà lu le rapport du semestre dernier, vous pouvez passer cette partie.

3.1. Description

Notre jeu de données consiste en plein de petites fonctions C faisant divers calculs, elles sont aussi appelés noyaux.
Par exemple, une fonction de multiplication de 2 matrices, extraction du minimum d'un tableau, calcul de la séquence de Fibonacci, etc…
La plupart des noyaux sont centrés sur du calcul scientifique, avec peu de branchements, mais nous avons essayé de diversifier ce jeu de données avec différents niveaux de parallélisation SIMD, quelques fonctions récursives, etc…

3.2. Benchmark des noyaux

Pour chacun de ces noyaux, nous avons utilisé une bibliothèque de benchmarks appelée nanobench [1], pour savoir combien de cycles CPU cette fonction prennait en moyenne.
Nous avons utilisé des outils de parsing de code C pour essayer d'automatiser l'adaptation du jeu de données tiré de divers projets de benchmark pour pouvoir l'utiliser avec nanobench.

Par exemple, voici les résultats d'un benchmark sur l'addition de 2 vecteurs :

ns/op	op/s	err%	ins/op	cyc/op	IPC	bra/op	miss%	total	benchmark
517.13	1,933,765.68	0.3%	7,177.00	2,075.59	3.458	1,027.00	0.1%	0.01	`vec_add_f32_no_smid`
133.21	7,506,981.41	0.1%	1,801.00	535.60	3.363	259.00	0.4%	0.01	`vec_add_f32_simd128`
70.87	14,110,601.83	1.3%	906.00	284.85	3.181	131.00	0.8%	0.01	`vec_add_f32_simd256`
312.48	3,200,195.08	0.3%	7,177.00	1,253.48	5.726	1,027.00	0.1%	0.01	`vec_add_f64_no_smid`
159.01	6,288,784.20	0.2%	3,593.00	638.74	5.625	515.00	0.2%	0.01	`vec_add_f64_simd128`
83.85	11,926,400.78	0.0%	1,802.00	338.11	5.330	259.00	0.4%	0.01	`vec_add_f64_simd256`

Tableau 1. – Résultats du benchmarking du noyau de calcul d'addition de 2 vecteurs

Les 2 colonnes qui nous intéressent sont cyc/op, qui est le nombre de cycles CPU moyen que prend le noyau de calcul à s'exécuter, et err%, qui est l'erreur relative de la mesure, afin de savoir si la mesure est fiable.
Pour tous nos noyaux, l'erreur maximale était de 1%, qui est assez précis pour nous.

3.3. Extraction des noyaux

Maintenant, il faut extraire les codes assembleurs des noyaux qu'on vient de benchmarké.

Nous utilisons un désassembleur comme objdump voir le code généré, mais reconnaître et extraire les kernels un par un serait trop long.
Nous avons donc automatisé le processus en ajoutant quelques macros afin de mettre des labels et noms de fonctions spécifiques pour pouvoir plus facilement les reconnaître et les isoler.

```
000000000000e0e0 <_Z19vec_add_f64_no_smid_start_PPN_LABELv>:
    e0e0: 48 8d 05 19 62 01 00  lea    0x16219(%rip),%rax        # 24300 <f64a>
    e0e7: 48 8d 15 12 82 01 00  lea    0x18212(%rip),%rdx        # 26300 <f64b>
    e0ee: 48 8d 35 0b a2 01 00  lea    0x1a20b(%rip),%rsi        # 28300 <f64c>
    e0f5: bf 00 04 00 00        mov    $0x400,%edi
    e0fa: 48 c7 c1 00 00 00 00  mov    $0x0,%rcx
    e101: f2 0f 10 04 c8        movsd  (%rax,%rcx,8),%xmm0
    e106: f2 0f 10 0c ca        movsd  (%rdx,%rcx,8),%xmm1
    e10b: f2 0f 58 c1           addsd  %xmm1,%xmm0
    e10f: f2 0f 11 04 ce        movsd  %xmm0,(%rsi,%rcx,8)
    e114: 48 83 c1 01           add    $0x1,%rcx
    e118: 48 39 f9              cmp    %rdi,%rcx
    e11b: 7c e4                 jl     e101 <_Z19vec_add_f64_no_smidv+0x21>

000000000000e11d <vec_add_f64_no_smid_end_PPN_LABEL>:
    e11d: c3                    ret
    e11e: 66 90                 xchg   %ax,%ax
```

Liste 1. – Exemple d'assembleur pour le noyau d'addition de 2 vecteurs

4. Entrainement type

Avant de commencer, voici comment nous avons entrainé et mesuré la précision de notre modèle, c'est-à-dire la qualité de la prédiction du réseau de neurones.
On cherche à la maximiser, avec une valeur proche de 1 signifie que le réseau de neurones arrive à bien prédire la valeur réelle, et 0 pas du tout, voici sa formule:

$$P = \frac{1}{n} * \sum_{i=1}^{n} \left(\frac{\left|\text{cycles}_{\text{prédits}}(\text{entrée}_i) - \text{cycles}_{\text{réels}}(\text{entrée}_i)\right|}{\max\left(\text{cycles}_{\text{réels}}(\text{entrée}_i), \text{cycles}_{\text{prédits}}(\text{entrée}_i)\right)} \right)$$

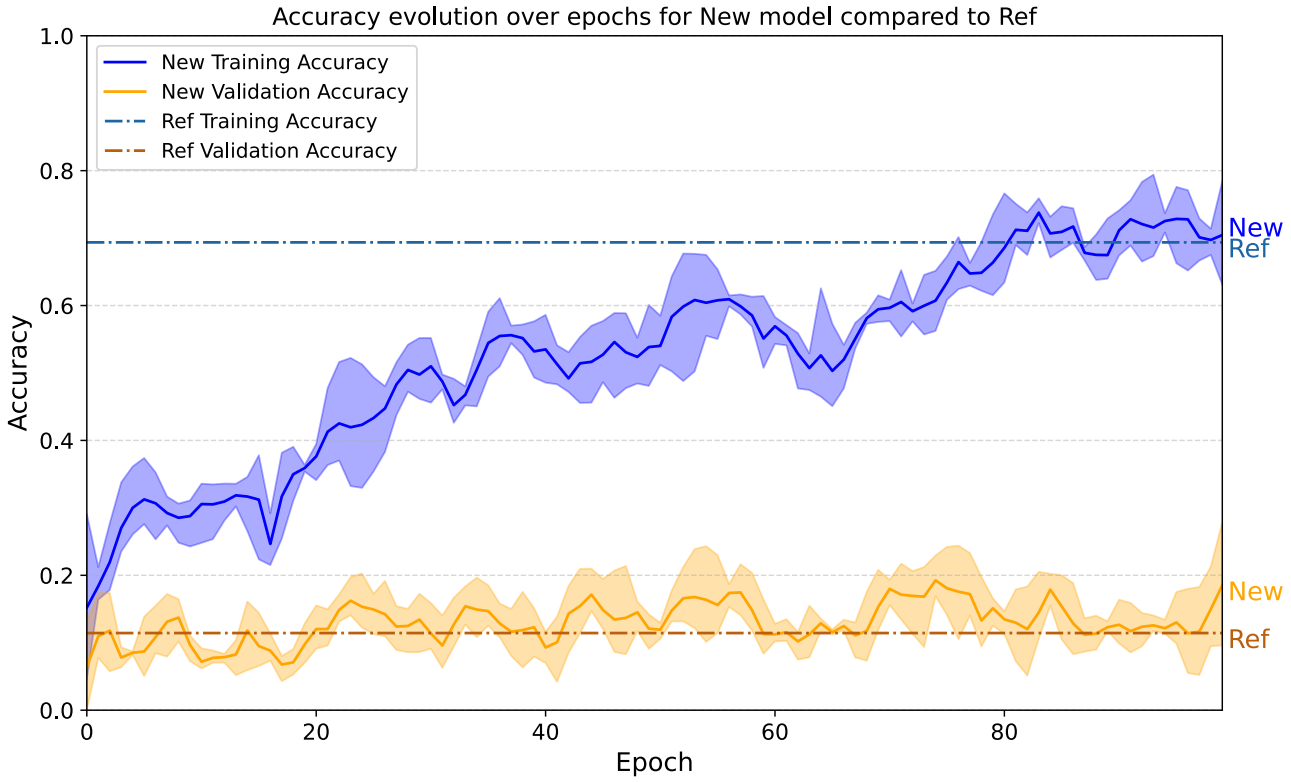
Vous pouvez voir que nous avons utiliser une échelle relative multiplicative plutôt qu'additive, car prédire 100 au lieu de 200 (2 fois moins de cycles) est aussi grave que prédire 400 au lieu de 200 (2 fois plus de cycles), et prédire 1000 au lieu de 100, est aussi grave que prédire 100 au lieu de 10. Par exemple, une précision de 0.9 signifie que notre modèle est en moyenne à ± 10% de la réalité.

Nous avons 2 ensemble de données : un ensemble d'entrainement et un ensemble de validation. L'ensemble d'entrainement est utilisé pour ajuster les poids du réseau de neurones, tandis que l'ensemble de validation est utilisé pour évaluer la précision du modèle sur des données qu'il n'a jamais vu.

Notre but principal est de minimiser la précision de l'ensemble de validation, mais une amélioration sur l'ensemble d'entrainement reste non négligeable.

Dû à la nature stochastique de l'entrainement de réseaux de neurones, nous avons fait tourner chaque modèle 3 fois, et nous présentons à chaque époque, la moyenne des 3 modèles, ainsi que le pire et le meilleur modèle, afin d'avoir une tranche de précisions que nous pouvons attendre à un instant donné.

Voici un exemple d'un résultat:



Summary of Training and Validation results

Metric	New	Ref	Difference
Training time	29.08s ± 0.18s	28.18s ± 0.65s	+0.90s
Training accuracy	0.70	0.69	+0.01
Validation accuracy	0.19	0.11	+0.07

Fig. 1. – Exemple de résultats d'entrainement

Ici, nous comparons le modèle New avec le modèle Ref.

En bleu, nous avons l'évolution de la précision de l'ensemble d'entrainement, et en orange, l'évolution de la précision de l'ensemble de validation.

De plus, nous avons une barre qui représente ce qu'avait atteint le modèle de référence à sa dernière époque, et on cherche à la dépasser.

Aussi, il y a un tableau récapitulatif des résultats, avec les 2 précisions, le temps d'entrainement, et montre les changements positifs et négatifs par rapport au modèle de référence.

5. Amélioration de la précision

À la fin du 1er semestre, nous avions déjà un code de réseau de neurones, capable de faire des prédictions, et de corriger ses erreurs avec de la descente de gradient [2], avec Eigen [3] en backend d'algèbre linéaire. Mais nous n'avions pas assez de données pour l'entrainer correctement, et donc nous n'avons pas de résultat de référence.

5.1. Formats [Précision, Performance]

Pour rappel, notre réseau de neurones doit prendre en entrée un vecteur de réels entre 0 et 1, et il faut transformer la représentation binaire du code assembleur pour pouvoir la donner en entrée. Pour convertir cette entrée, nous avons 2 solutions :

- Encoder bit par bit chaque instruction : 1 bit = 1 réel

- Encoder par octets : En effet, en général, dans les analyseurs de code binaire, on les regarde en octets.

Ici, chaque pair de nombre héxadécimal correspond à un octet. Donc, chaque octet qui était compris entre 0 et 255, et on peut le normaliser entre 0 et 1 en divisant par 256.

Pour chaque de ces 2 solutions, nous avons aussi ajouté une version avec un marqueur de fin d'instruction. (une valeur unique qui indique la fin de l'instruction) Comme les humains aiment bien travailler avec des instructions claires, et que les réseaux de neurones essayent d'imiter le cerveau humain, nous avons pensé que ça pourrait aider.

Voici un exemple d'instruction, et les 4 encodages possibles :

```
xchg   %ax,%ax
# Code hexadécimal : 66 90
# Code binaire : 01100110 10010000
```

$$\left(\frac{66}{256}, \frac{90}{256}\right)$$

Héxadécimal sans marqueur

$$\left(\frac{66+1}{256}, 0, \frac{90+1}{256}, 0\right)$$

Héxadécimal avec marqueur

$$(0,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0)$$

Binaire sans marqueur

$$(1,0,1,0,0,1,1,0,0.5,1,0,0,1,0,0,1,0,0.5)$$

Binaire avec marqueur

Fig. 2. – Différents encodages possibles d'une instruction assembleur

Les formats en octets ont cependant un avantage non négligeable : ils sont plus compacts.
En effet, la taille de chaque couche du réseau de neurones peut être donc divisée par 8 (un peu moins pour les formats avec marqueurs), ce qui permet de réduire aussi la taille du réseau de neurones, et donc le temps d'entrainement.

Nous avons donc testé ces 4 formats différents, cependant les formats binaires étant moins compact, le temps d'entrainement est plus long, et pour notre ensemble de données, on a estimé le temps d'un entrainement complet à 58h.
Pour quand même pouvoir comparer les résultats, nous avons fait tourner les modèles sur une petite partie des données.

Voici les résultats d'entrainement de ces différents formats:

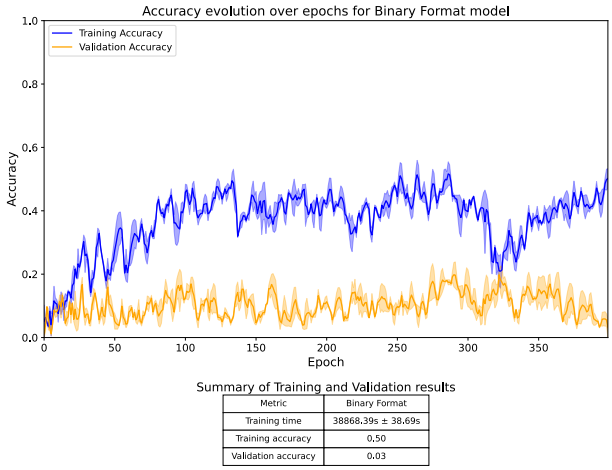


Fig. 3. – Résultats d'entrainement du format binaire

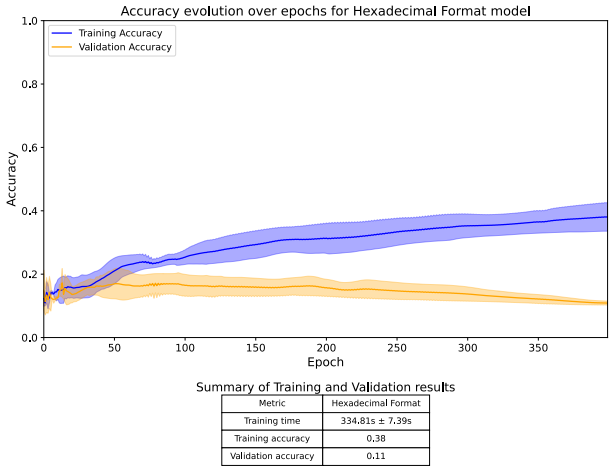


Fig. 4. – Résultats d'entrainement du format héxadécimal

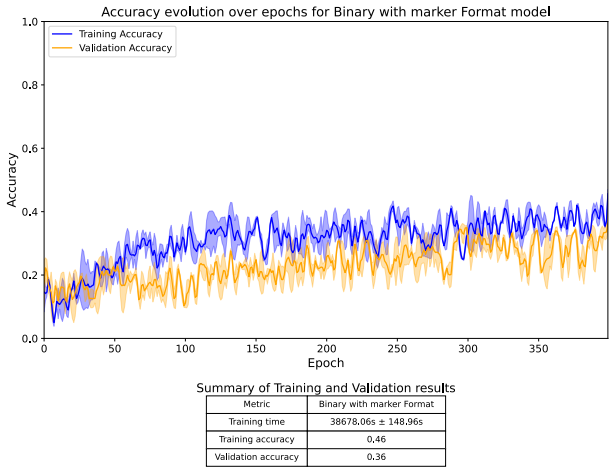


Fig. 5. – Résultats d'entrainement du format binaire avec marqueur

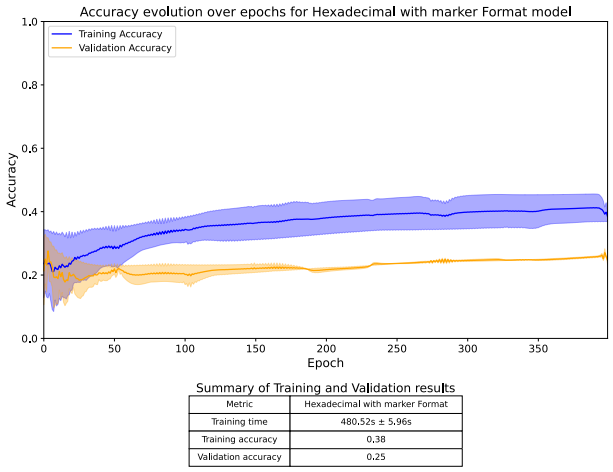


Fig. 6. – Résultats d'entrainement du format héxadécimal avec marqueur

Ici, nous voyons que le format binaire est bien plus chaotique que le format héxadécimal, mais n'a pas d'avantage de précision, donc on va rester sur le format héxadécimal.

Le format avec marqueur pour l'héxadécimal semble être meilleur que la version sans, sur le petit jeu de données, mais pour être sûr, nous allons tester les 2 sur le jeu de données complet.

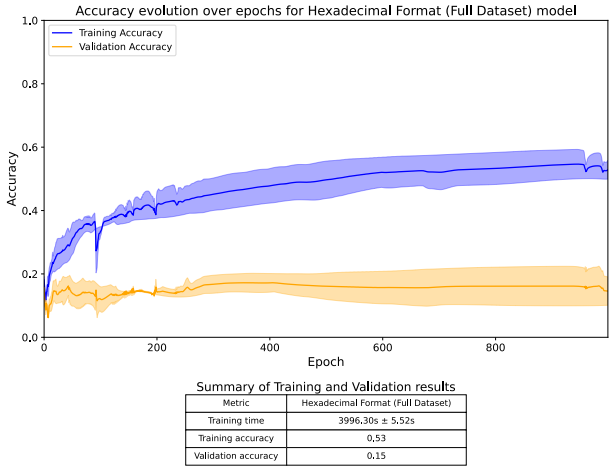


Fig. 7. – Résultats d'entrainement du format héxadécimal sans marqueur

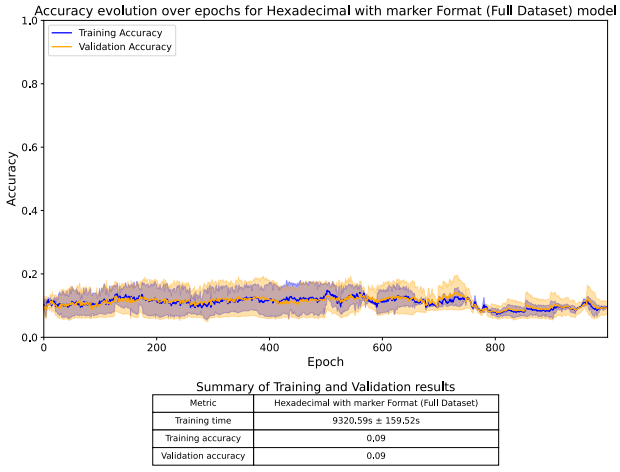


Fig. 8. – Résultats d'entrainement du format héxadécimal avec marqueur

Surprenamment, sur le jeu de données complet, c'est l'inverse, le format sans marqueur est meilleur que le format avec marqueur.
Nous avons donc décidé de garder le format héxadécimal sans marqueur pour la suite.

5.2. Encodage du nombre de cycles [Précision]

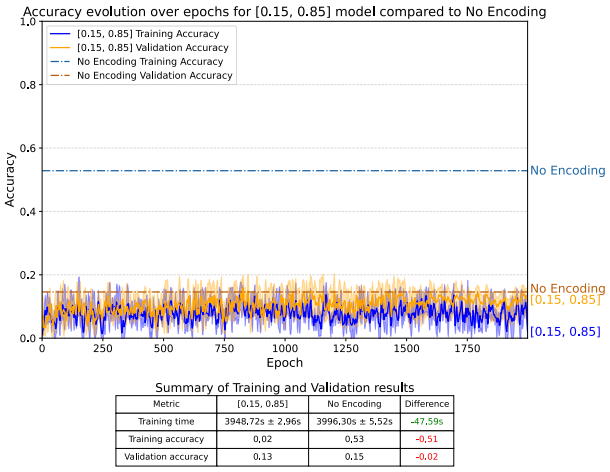
Nous avons décider de limiter le nombre de cycles à 4 milliards (2^{32}), car c'est la taille maximale d'un entier 32 bits, correspond à 1 seconde sur la plupart des processeurs, et est suffisant pour quasiment l'entiereté des noyaux que nous avons.
Il faut cepedant normaliser les entrées de façon à ce qu'elles soient comprises entre 0 et 1, pour que le réseau de neurones puisse les traiter.
Au début, nous avons naïvement normalisé les nombres de cycles en divisant par 4 milliards, mais en analysant les prédictions, nous avons remarqué que le réseau avait du mal pour les noyaux prenant un petit nombre de cycles.
Nous supposons que cela est dû au fait d'avoir donc beaucoup de valeurs à prédire très proches de 0, et le réseau de neurones a eu du mal à apprendre, notamment car:

- Il faut des poids et biais très grands pour compenser ces petites valeurs.
- Certaines fonctions d'activation ont des problèmes avec des valeurs très proches de 0, notamment pour la descente de gradient.
- Les fonctions d'activation peuvent aussi avoir des problèmes de saturation, par exemple, pour sigmoid, obtenir des valeurs de sortie petites requiert des valeurs d'entrée encore plus petite, à cause de la nature exponentielle de la fonction.
- La précision des nombres flottants peut jouer.

Pour résoudre ce problème, nous avons essayé plusieurs solutions.

- Normaliser sur un autre interval, ici sur [0.15, 0.85]
- Découper sur 4 sorties les différents puissances de 1000 cycles (milliards, millions, milliers, unités)
- Utiliser différentes échelles avant de normaliser ($\log_2$, $\log_{10}$, racine carrée)

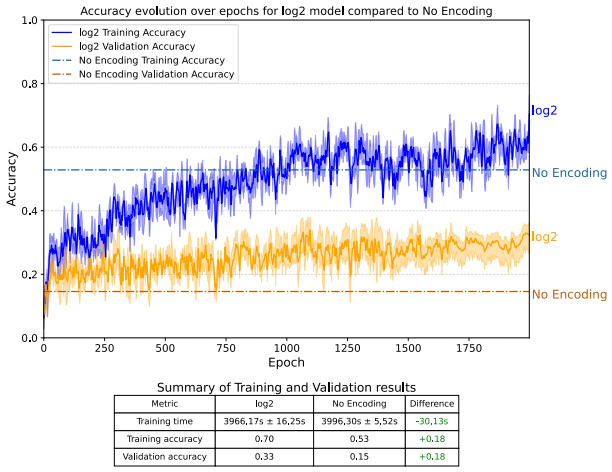
Voici les résultats d'entrainement de ces différentes normalisations :



Summary of Training and Validation results

Metric	[0.15, 0.85]	No Encoding	Difference
Training time	3948.72s ± 2.96s	3996.30s ± 5.52s	-47.59s
Training accuracy	0.02	0.53	-0.51
Validation accuracy	0.13	0.15	-0.02

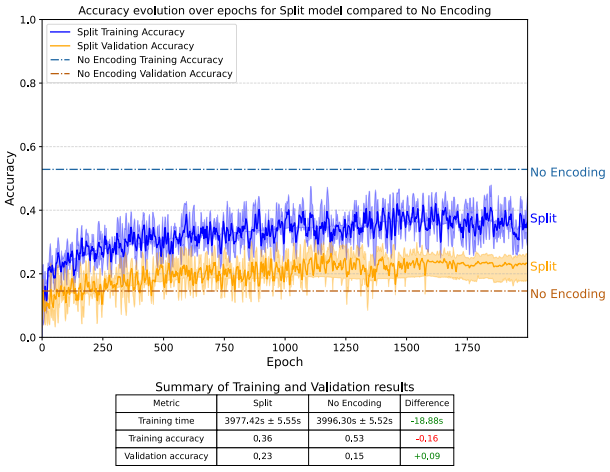
Fig. 9. – Résultats d'entrainement avec normalisation sur [0.15, 0.85]



Summary of Training and Validation results

Metric	log2	No Encoding	Difference
Training time	3966.17s ± 16.25s	3996.30s ± 5.52s	-30.13s
Training accuracy	0.70	0.53	+0.18
Validation accuracy	0.33	0.15	+0.18

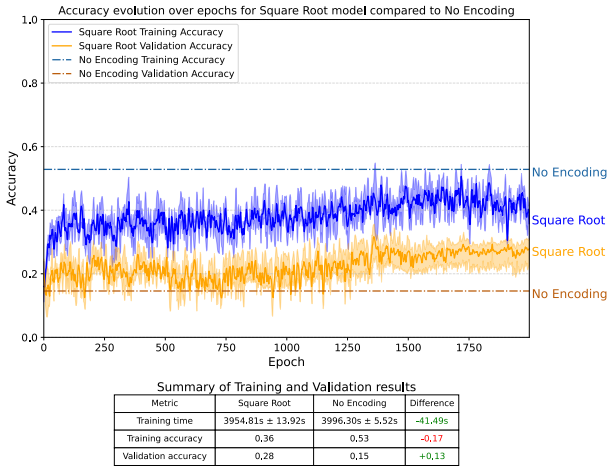
Fig. 10. – Résultats d'entrainement avec normalisation sur log2



Summary of Training and Validation results

Metric	Split	No Encoding	Difference
Training time	3977.42s ± 5.55s	3996.30s ± 5.52s	-18.88s
Training accuracy	0.36	0.53	-0.16
Validation accuracy	0.23	0.15	+0.09

Fig. 11. – Résultats d'entrainement avec normalisation sur 4 sorties



Summary of Training and Validation results

Metric	Square Root	No Encoding	Difference
Training time	3954.81s ± 13.92s	3996.30s ± 5.52s	-41.49s
Training accuracy	0.36	0.53	-0.17
Validation accuracy	0.28	0.15	+0.13

Fig. 12. – Résultats d'entrainement avec normalisation sur racine carrée

Nous observons que normaliser sur un autre interval est pire, et en observant les prédictions, cela est dû à la précision flottante qui convertit 10 et 2000 cycles à la même valeur (0.15)
Diviser les différentes unités a aussi empiré la prédiction.
Pour les différentes échelles, log2 a réussi à améliorer la précision, contrairement à la racine carrée.

Après, pour les échelles logarithmes, nous avons aussi essayé d'ajouter un terme constant, ou de change de base, mais cela n'a fait que baisser la prédiction de notre modèle, donc nous utiliserons la transformation

$$f(x) = \frac{\log_2(x)}{\log_2(4 * 10^9)}$$

pour la suite.

5.3. Descentes de gradient, mini-batch et optimiseurs [Précision, Performance, Parallélisation]

Jusqu'ici, nous avons utilisé la descente de gradient 1 par 1, avec un pas fixe.
C'est à dire que pour chaque donnée, nous la donnons au réseau de neurones, nous calculons l'erreur, et nous ajustons les poids du réseau de neurones en fonction de cette erreur.

Pour cela, soit on peut le faire donnée par donnée, soit on peut le sur un paquet de données en même temps, c'est ce qu'on appelle le Mini-batch.
Pour implémenter ces batchs, nous avons dû repenser la manière de faire fonctionner le réseau de neurones.
Avant, nos entrées étaient en forme de vecteurs, que nous faisons passer de couche en couche. Nous pouvons les faire passer l'un après l'autre, mais cela n'est pas très efficace.

Nous pouvons prendre avantage des propriétés de l'algèbre linéaire, et faire passer tous les vecteurs d'un coup.
En effet, soit x, y et z 3 vecteurs, et W la matrice de poids pour la couche suivante.

$$x \times W = x'$$

$$y \times W = y'$$

$$z \times W = z'$$

Nous pouvons construire une matrice E contenant les 3 vecteurs:

$$E = \begin{pmatrix} x_{11} & x_{12} & ... & x_1n \\ y_{11} & y_{12} & ... & y_1n \\ z_{11} & z_{12} & ... & z_1n \end{pmatrix}$$

Et nous pouvons faire passer cette matrice de la même manière que les vecteurs:

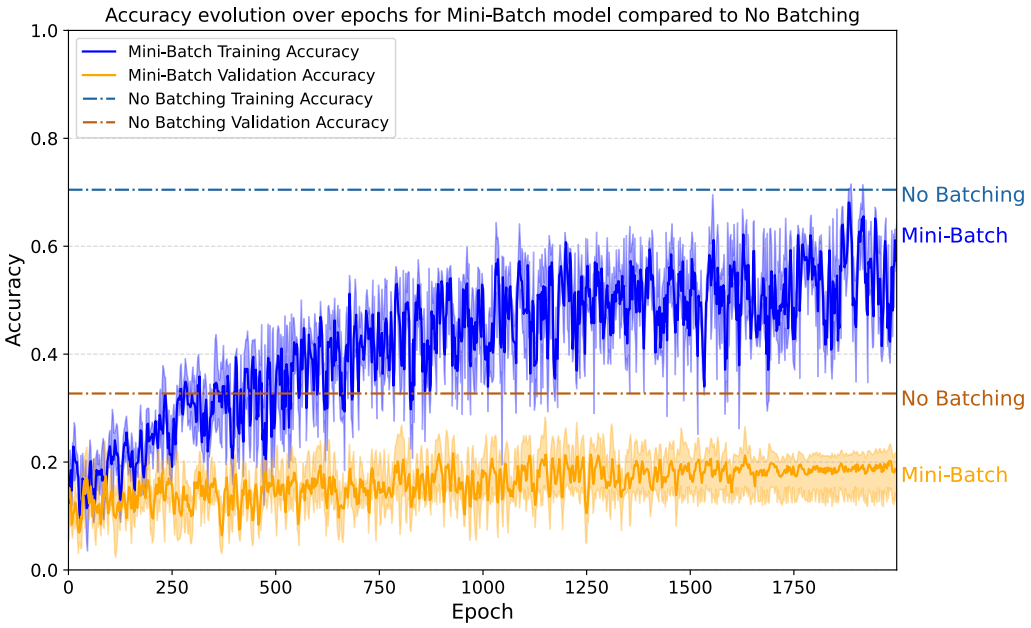
$$E \times W = E'$$

avec

$$E' = \begin{pmatrix} x'_{11} & x'_{12} & ... & x'_1n \\ y'_{11} & y'_{12} & ... & y'_1n \\ z'_{11} & z'_{12} & ... & z'_1n \end{pmatrix}$$

Ce qui nous donne bien l'équivalent de faire passer chaque vecteur un par un, mais en une seule fois.

Maintenant, comparons les résultats d'entrainement du modèle avec mini-batch.
Ici le pas choisi μ est de 0.05, et la taille du batch est de 16. Car ce sont les valeurs qui ont donné les meilleurs résultats.



Summary of Training and Validation results

Metric	Mini-Batch	No Batching	Difference
Training time	2419.05s ± 3.22s	3966.17s ± 16.25s	-1547.12s
Training accuracy	0.61	0.70	-0.10
Validation accuracy	0.19	0.33	-0.14

Fig. 13. – Résultats d'entrainement du modèle avec mini-batch

On peut voir qu'on a perdu en précision, peu pour l'ensemble d'entrainement, mais presque de moitié pour l'ensemble de validation.
Cependant, le temps d'entrainement a été réduit de plus de 40%, ce qui est très appréciable.
Cela s'explique car la descente de gradient est une opération très coûteuse, et le fait d'en faire 1 pour 16 réduit en conséquence le temps d'entrainement.

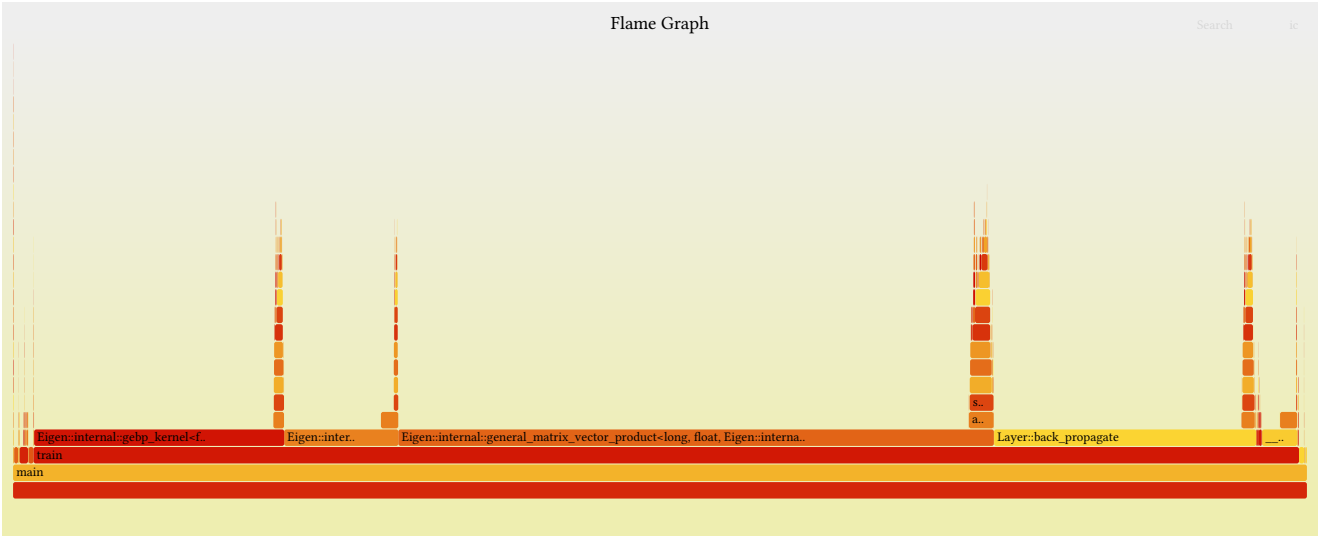


Fig. 14. – Temps de calcul de la descente de gradient

Dans ce flamegraph, on peut voir la proportion de temps passé dans la descente de gradient (backpropagation), alors que le temps passé dans la passe avant est négligeable.

Pour garder cette optimisation par batch, sans pour autant perdre en précision, il existe des outils appelés « optimiseurs » [4].
Les optimiseurs sont des algorithmes qui changent la manière de descendre le long du gradient, afin de trouver le maximum de précision et de converger, plus rapidement ou plus stablement.

En soit, la descente de gradient par batch est un optimiseur (appelé SGD pour Stochastic Gradient Descent), mais il en existe d'autres qui sont plus performants.

Voici ceux que nous avons testé :

- Momentum

Momentum est un optimiseur qui conserve sa direction de descente lors des prochaines époques. C'est à dire qu'il garde en mémoire la direction de la descente de gradient des époques précédentes, et il avance dans cette direction même si la pente change. Cela le rend robuste aux petits changements de direction, mais peut aussi le faire passer à coté d'un bon minimum local.

- RMSProp (Root Mean Square Propagation)

RMSProp ajuste le taux d'apprentissage pour chaque paramètre en fonction de l'ampleur des gradients récents, afin d'éviter de faire de grands pas lorsque les gradients sont grands. Il permet de mieux gérer les pentes trop raides pour les prendre à un pas fixe, et donc de mieux converger.

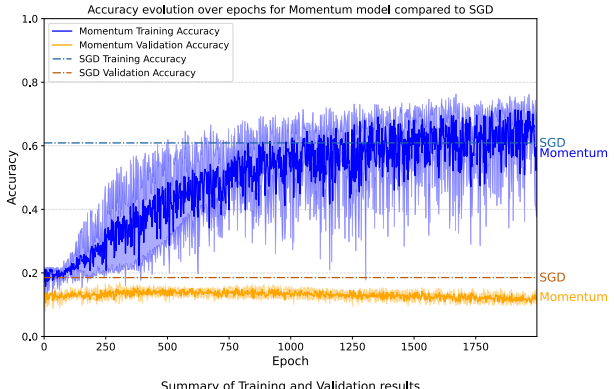
- Adam (Adaptive Moment Estimation)

Adam est un optimiseur qui combine les avantages de Momentum et RMSProp. Il utilise le momentum pour garder la direction de la descente, et il ajuste le taux d'apprentissage pour chaque paramètre en fonction de l'ampleur des gradients récents. Il est souvent considéré comme l'un des meilleurs optimiseurs, car il converge rapidement et est robuste aux petits changements de direction.

- AMSGrad (Adam with Maximum past Square Gradients)

AMSGrad est une variante d'Adam qui impose des bornes sur le taux d'apprentissage pour chaque paramètre. Cela permet d'éviter que le taux d'apprentissage ne devienne trop petit, ou trop grand, et donc de mieux converger. Cependant, il est plus coûteux en calculs, car il doit garder en mémoire les valeurs précédentes des gradients, ainsi que les valeurs des bornes.

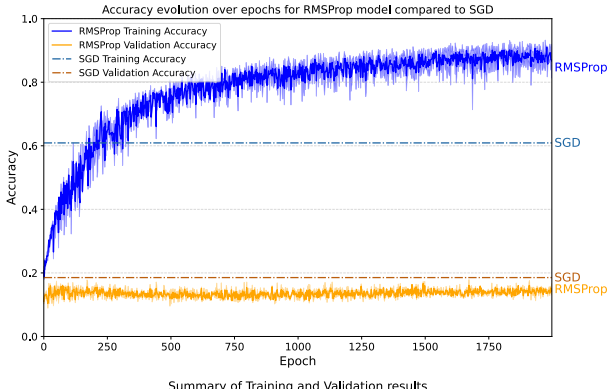
Nous avons testé ces 4 optimiseurs, pour une taille de batch de 16. Voici les meilleurs résultats obtenus :



Summary of Training and Validation results

Metric	Momentum	SGD	Difference
Training time	2376.66s ± 19.38s	2419.05s ± 3.22s	-42.39s
Training accuracy	0.60	0.61	-0.01
Validation accuracy	0.12	0.19	-0.06

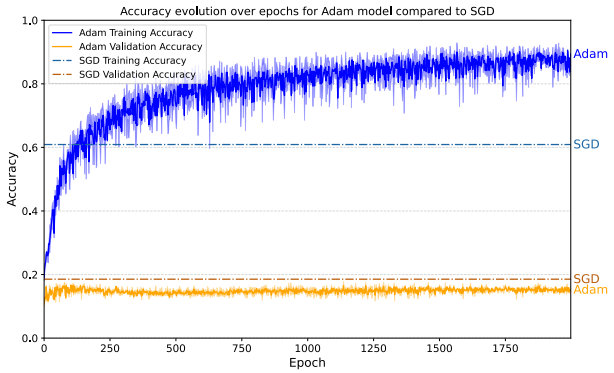
Fig. 15. – Résultats d'entrainement du modèle avec Momentum



Summary of Training and Validation results

Metric	RMSProp	SGD	Difference
Training time	4314.76s ± 7.08s	2419.05s ± 3.22s	+1895.72s
Training accuracy	0.87	0.61	+0.26
Validation accuracy	0.15	0.19	-0.04

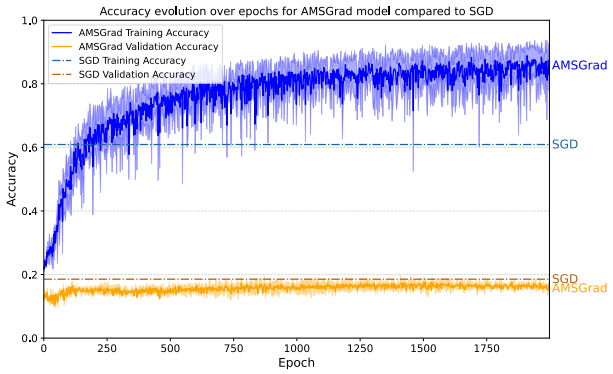
Fig. 16. – Résultats d'entrainement du modèle avec RMSProp



Summary of Training and Validation results

Metric	Adam	SGD	Difference
Training time	3637.20s ± 1.85s	2419.05s ± 3.22s	+1218.15s
Training accuracy	0.89	0.61	+0.28
Validation accuracy	0.15	0.19	-0.03

Fig. 17. – Résultats d'entrainement du modèle avec Adam



Summary of Training and Validation results

Metric	AMSGrad	SGD	Difference
Training time	4659.39s ± 35.95s	2419.05s ± 3.22s	+2240.34s
Training accuracy	0.86	0.61	+0.25
Validation accuracy	0.17	0.19	-0.02

Fig. 18. – Résultats d'entrainement du modèle avec AMSGrad

A part Momentum qui n'a pas réussi à converger, les autres optimiseurs ont tous réussi à augmenter la précision de l'ensemble d'entrainement à plus de 85%. Même s'il n'y a pas de progrès réel sur l'ensemble de validation.
Adam semble être le meilleur optimiseur, avec une précision légèrement supérieure à 90%, avec le temps d'entrainement le plus court parmis les 3. (3637s pour Adam, contre 4059s pour AMSGrad, et 4315s pour RMSProp)
Nous allons donc utiliser Adam pour la suite.

5.4. Améliorer la généralisation [Précision]

5.4.1. Ajout de bruit

Une technique classique pour améliorer la précision d'un modèle est d'ajouter du bruit durant l'entrainement. Cela permet d'améliorer la généralisation du modèle, car il apprend à ne pas se fier à des valeurs précises, mais à des tendances.

Nous avons tester 2 manières d'ajouter du bruit :

Passe avant perturbée: durant la passe avant, on ajoute du bruit sur les valeurs des neurones de la forme $\alpha \times \text{valeur} + \beta$.

Dropout [5]: Durant l'entrainement, on tue des neurones aléatoirement. A chaque époque, on fait un masque de neurones à tuer, c'est à dire qu'on met à 0 la valeur de certains neurones, ce qui empêche le réseau d'avoir des dépendances trop fortes, et d'avoir des neurones plus importants pour la prédiction que d'autres.

Malheuresement, ces techniques sont plus efficaces lors d'un cas de sur-apprentissage, ce qui n'est pas notre cas, et donc le bruit ajouté n'a fait que de perturber le réseau de neurones, et de le faire descendre en précision.

5.4.2. Facteur de régularisation L2

Le facteur de régularisation L2 est une technique assez utilisée dans l'état de l'art [6], qui permet de réduire le sur-apprentissage en ajoutant un terme à la fonction de coût, visant à pénaliser les poids trop grands.
Cela permet d'éviter que le réseau se fie trop à certaines relations entre les neurones, et de le forcer à généraliser.

Comme pour le bruit, augmenter le facteur de régularisation L2 n'a eu que des effets négatifs sur la précision du modèle, et a fait baisser la précision de l'ensemble d'entrainement.

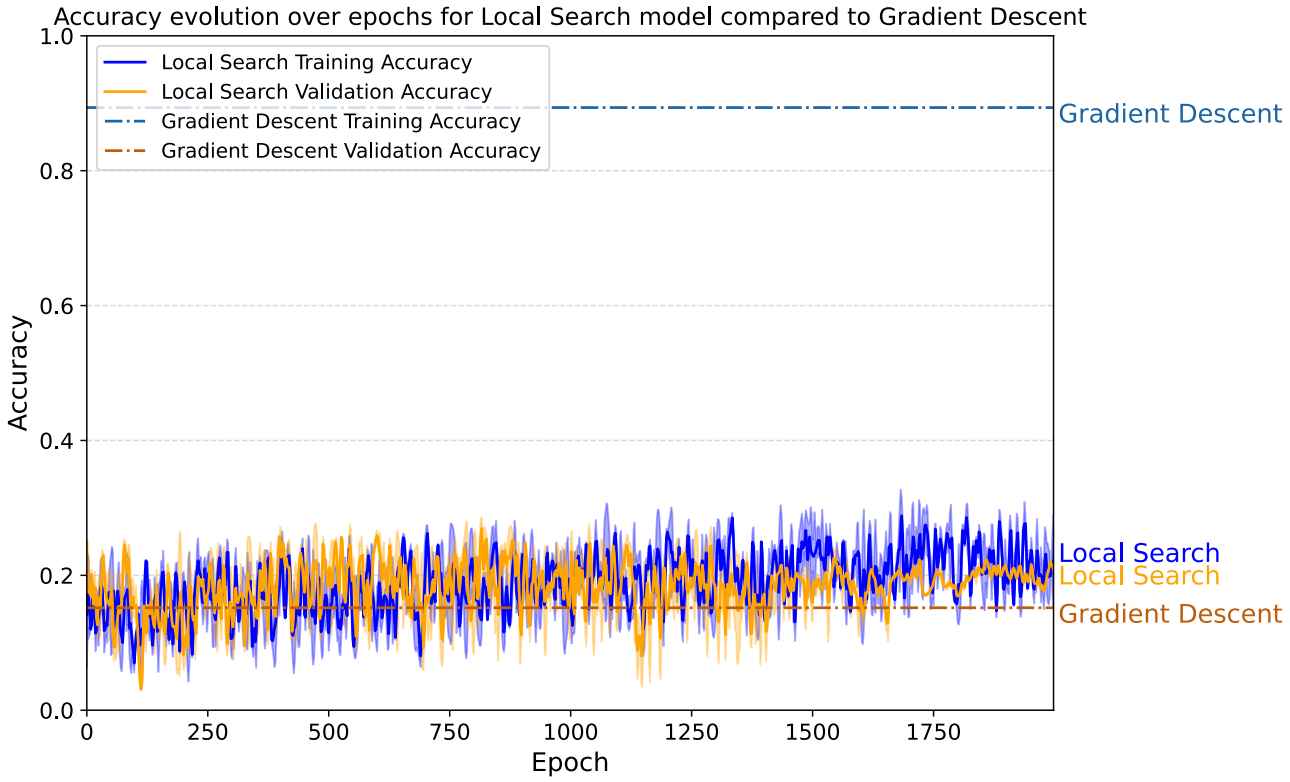
5.5. Exploration [Précision]

Les réseaux de neurones sont en réalité un problème d'optimisation: on explore l'espace des paramètres possibles (poids et biais), et on essaie de trouver ceux qui maximisent la précision du modèle.

Jusqu'ici, nous avons utilisé la descente de gradient pour explorer cet espace, mais nous avons aussi utilisé d'autres algorithmes d'optimisation.

5.5.1. Recherche Locale

La recherche locale consiste à faire des pas dans diverses directions autour d'un point donné, et de garder le meilleur des points voisins trouvés.
Cela a le même effet que de faire les descentes de gradient par dérivée approximée, mais avec beaucoup moins de variations à calculer.



Summary of Training and Validation results

Metric	Local Search	Gradient Descent	Difference
Training time	6704.01s ± 14.13s	3637.20s ± 1.85s	+3066.81s
Training accuracy	0.22	0.89	-0.67
Validation accuracy	0.21	0.15	+0.06

Fig. 19. – Résultats d'entrainement du modèle avec recherche locale

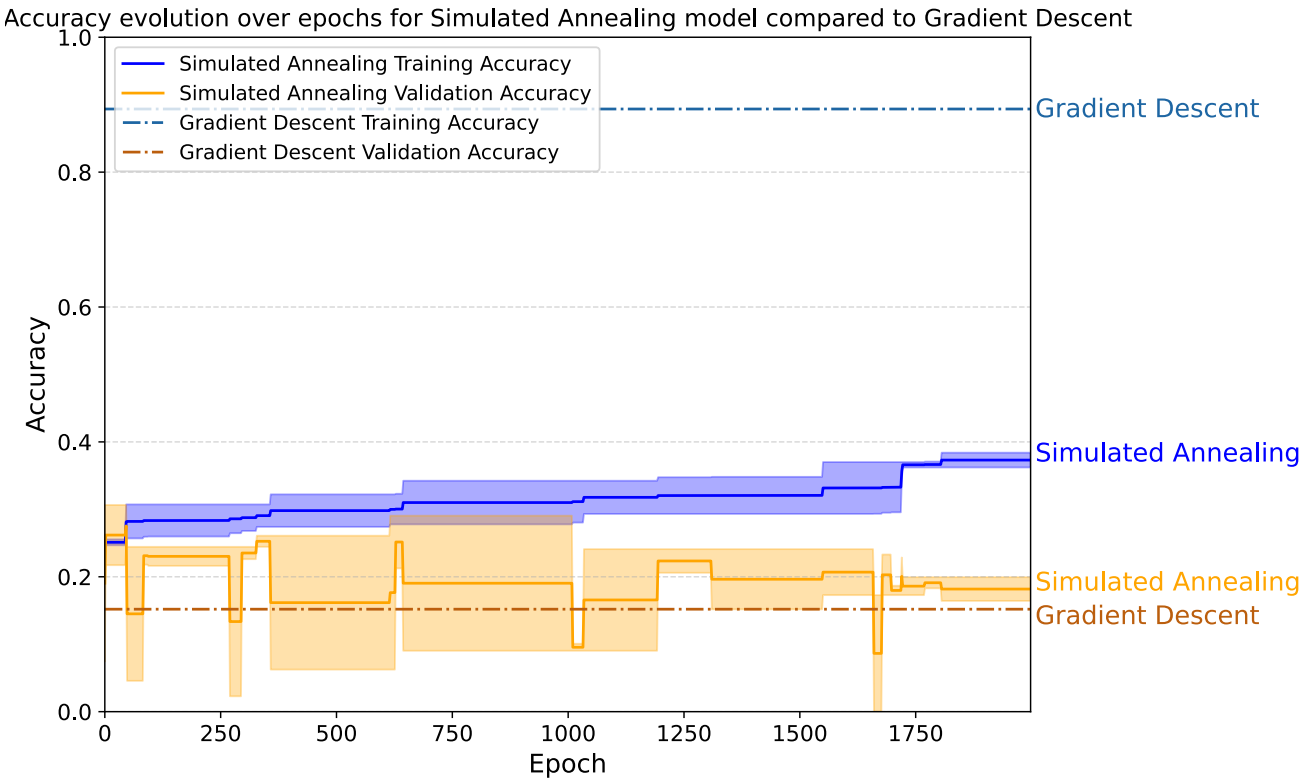
Malheureusement, les résultats sont très médiocres, en plus d'être plus lent, donc on va écarter cette approche.

5.5.2. Simulated Annealing

Cet algorithme est inspiré de la physique, notamment de la métallurgie. Et est utilisé dans beaucoup de domaines, comme la recherche opérationnelle, l'optimisation combinatoire, etc.

Il fonctionne en gardant un ensemble de paramètres candidats, qui sont les meilleurs candidats trouvés jusqu'à présent, et en les modifiant selon un principe de refroidissement.
C'est à dire qu'au début de l'algorithme, nous avons beaucoup de variations autour des candidats, et au fur et à mesure que l'algorithme avance, nous diminuons la température, et donc les variations.
Ceci à pour effet de faire une grande exploration au début, et de se concentrer sur les meilleurs candidats au fur et à mesure.

Après avoir implementé cet algorithme, voici les résultats obtenus:



Summary of Training and Validation results

Metric	Simulated Annealing	Gradient Descent	Difference
Training time	6835.60s ± 11.86s	3637.20s ± 1.85s	+3198.40s
Training accuracy	0.37	0.89	-0.52
Validation accuracy	0.18	0.15	+0.03

Fig. 20. – Résultats d'entrainement du modèle avec Simulated Annealing

Cette fois aussi, les résultats sont médiocres, et le modèle n'arrive pas du tout à apprendre. Nous pensons que cela vient du fait que nous avons trop de paramètres à explorer, et que l'algorithme n'arrive pas à converger. On peut tout de même noter que la précision d'entrainement augmente en escalier, quand le modèle trouve un meilleur candidat par chance.

Le principe de garder un ensemble de bons candidats reste une bonne idée, donc nous avons aussi tenter de fusionner cette méthode à la descente de gradient, mais sans succès non plus.

5.6. Topologie par évolution [Précision, Performance, Parallélisation]

Maintenant que nous avons des algorithmes d'entrainement qui fonctionnent bien, nous allons revenir sur la topologie de notre réseau de neurones.

Cette fois, au lieu de choisir des topologies arbitraires selon ce qui nous semble bien, nous allons utiliser un algorithme génétique pour trouver la meilleure topologie.
Un algorithme génétique est un type spécial d'algorithme pour les problèmes dont la solution optimale n'est pas évidente, mais dont la qualité d'une solution est facile à juger.
Il se base sur le principe de l'évolution et de la sélection naturelle, c'est à dire qu'il va générer un certains nombre de solutions aléatoires, appelées invidus, puis nous allons garder les meilleures, et les faire reproduire en croisant certains gènes, et en ajoutant un peu de mutation.
Au fur et à mesure, les individus vont s'améliorer, et nous allons garder le meilleur individu.

Le plus important est de bien définir quels sont les gènes de nos individus, comment les muter, et comment les évaluer.

Les gènes de nos individus sont la topologie de notre réseau de neurones, donc le nombre de couches cachées, et le nombre de neurones dans chaque couche.

Pour les muter, nous allons ajouter ou enlever une couche, ou ajouter ou enlever des neurones dans une couche.
Il est difficile de « croiser » des topologies, donc nous allons juste faire des mutations.

Pour les évaluer, nous allons les entrainer sur notre ensemble de données, et les noter en fonction de leur précision. Aussi, pour éviter que les réseaux ne deviennent trop gros, nous allons aussi les pénaliser en fonction de leur taille.
La formule de notation sera donc :

$$(1 + (70 \times \text{Précision}_{\text{validation}} + 30 \times \text{Précision}_{\text{entrainement}}))^2 - \sqrt[4]{\sum_{c \in \text{ couches}} \text{taille}_c}$$

Nous avons mit 70% de la note sur la précision de l'ensemble de validation, car nous cherchons le modèle capable le mieux de généraliser, et 30% sur la précision de l'ensemble d'entrainement, car c'est toujours bien d'avoir un modèle qui apprend bien.
Le poids de la précision est 6 fois plus important que la pénalité de la taille, car c'est ce qui nous semblait avoir le meilleur équilibre d'après des tests.

Aussi, cet algorithme possède un grand avantage: il très parallélisable, car chaque individu peut être évalué en parallèle.

```
auto individus = new individus[nb_individus];
// On génère des individus aléatoires
for (int i = 0; i < nb_individus; i++) { // <- Entièrement parallélisable
    individus[i] = new individu_aléatoire();
}
for (int i = 0; i < nb_iters; i++) {
    // On évalue les individus
    float[] notes = new float[nb_individus];
    for (int j = 0; j < nb_individus; j++) { // <- Entièrement parallélisable
        notes[j] = evaluer(individus[j]);
    }

    // On garde les meilleurs
    individus.sort_by(notes);
    individus = individus[10%];

    // On croise et on mute la génération suivante
    individus += reproduction(individus);
    for (int j = 0; j < nb_individus; j++) { // <- Entièrement parallélisable
        mutation(&individus[j]);
    }
}
```

Liste 3. – Pseudo-code d'un algorithme génétique

Aussi, lors des différents entrainements fait lors du projet, nous avons remarqué que les modèles plus mauvais à la fin étaient aussi plus mauvais au début, et que nous pouvions facilement voir quand un modèle n'allait pas aller bien loin.
On va donc vérifier de temps en temps chaque modèle, et si il n'est pas bon, on arrête l'entrainement, pour laisser plus de temps de calcul aux modèles plus prometteurs.
Cette une technique courante appelée **Early Stopping**.

Afin de pouvoir faire plus d'itérations, nous avons entrainé les modèles sur une petite partie des données, mais en général, avoir un meilleur résultat que les autres sur le jeu de donnés réduit implique aussi d'avoir un meilleur résultat sur l'ensemble complet des données.

Voici l'étude en scalabilité forte pour l'algorithme génétique, sur 5 générations, possédant 16 individus.

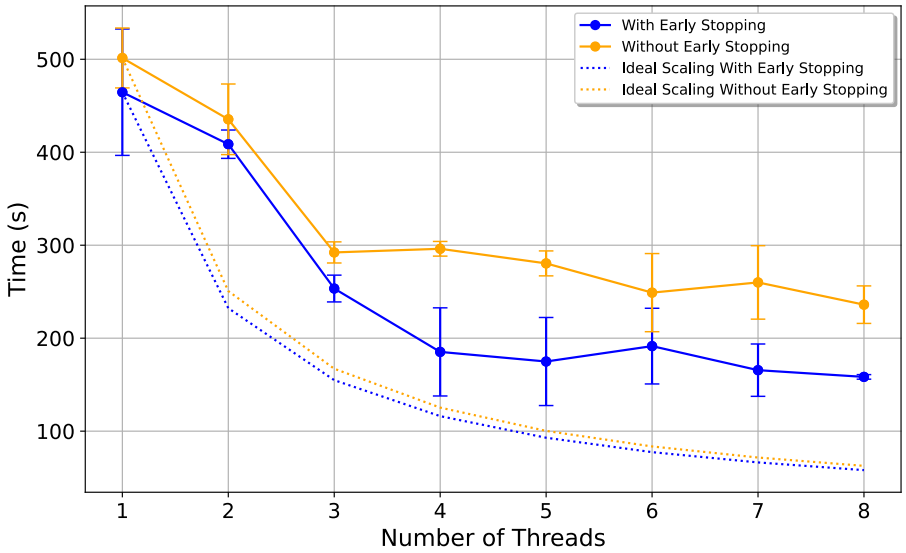


Fig. 21. – Étude de scalabilité forte de l'algorithme génétique

Nous pouvons voir que la version avec Early Stopping arrive à mieux scaler, et est en général plus rapide. Il y a tout de même pas mal de bruit, car l'algorithme génétique est très stochastique, mais les tendances sont visibles.

Maintenant, si on laisse plus de temps à l'algorithme, avec plus d'individus, plus de générations, et plus d'époques par candidat, voici ce que l'on obtient:

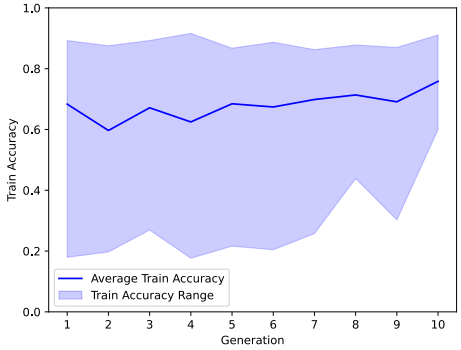


Fig. 22. – Évolution de la précision d'entrainement des candidats

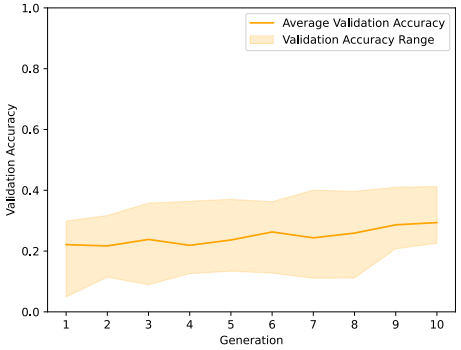


Fig. 23. – Évolution de la précision de validation des candidats

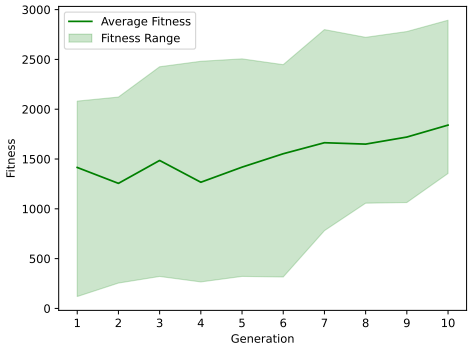


Fig. 24. – Évolution de la note des candidats

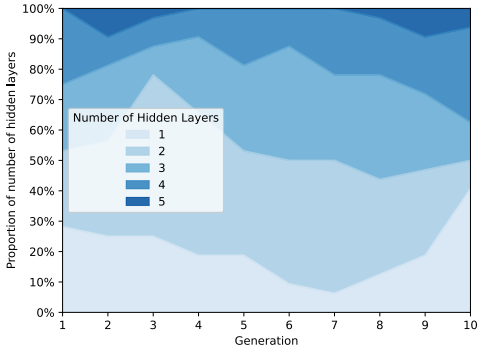


Fig. 25. – Évolution du nombre de couches cachées des candidats

On peut voir une évolution positive de la précision d'entrainement et de validation moyenne, ainsi que de la note des candidats.
Ceci indique que l'algorithme génétique marche et converge vers les meilleurs solutions.

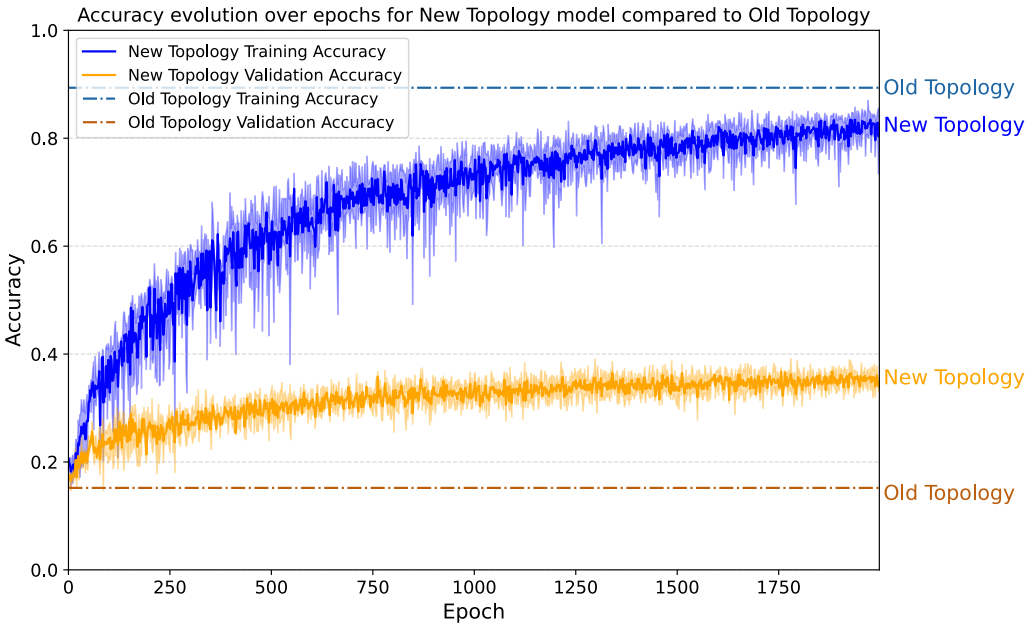
Pour la précision d'entrainement, les meilleurs candidats stagnent à 0.9 sur toutes les générations, et donc semble être la limite.

Pour la précision de validation, les meilleurs candidats commençaient à un peu moins de 0.3, et au fil du temps, ils sont montés à un peu plus de 0.4 vers la fin.
On peut d'ailleurs observer 2 légers bons aux générations 3 et 7, sûrement dû à des candidats qui ont été très bons par hasard, et qui se sont répliqués dans les générations suivants.
On peut aussi voir un plateau à 0.4 à partir de la génération 7, et donc semble aussi s'orienter vers une limite.

Pour la note, on peut observer un phénomène similaire, avec les mêmes bons aux générations 3 et 7, avec une augmentation progressive de la moyenne et de la plage de notes.

Sur le nombre de couches cachées, au début c'est assez chaotique, mais au fur et à mesure, on voit que 2 nombres prennent le dessus, 1 et 4 couches cachées.
Les candidats avec 1 couche cachée ont moins de paramètres à ajuster, et donc plus facile d'entrainer et de trouver une topologie qui convient.
Les candidats avec 4 couches cachées sont sûrement les plus adpatés pour le problème, avec bien plus de paramètres pour prédire des patterns complexes.

Et parmis tous les candidats, toute génération confondue, l'algorithme génétique s'est arrêté sur une topologie de (1375, 988, 236, 1047, 70, 1).
Comparons cela avec la topologie choisie au début, qui était (1375, 1375, 342, 1).



Summary of Training and Validation results

Metric	New Topology	Old Topology	Difference
Training time	2711.84s ± 8.61s	3637.20s ± 1.85s	-925.36s
Training accuracy	0.81	0.89	-0.08
Validation accuracy	0.35	0.15	+0.19

Fig. 26. – Comparaison des résultats d'entrainement de la nouvelle et de l'ancienne topologie

On peut remarquer plusieurs choses.
Déjà, il n'y a pas tous les gains excomptés. Lors de l'évolution, on a utilisé le plus petit jeu de données, qui a réussi à aller au delà de 40% de précision de validation, mais sur le jeu de données complet, la précision de validation est de 36%, mais c'était attendu.

Ensuite, le modèle arrive mieux à généraliser, car la précision de validation a doublé, mais la précision d'entrainement a légèrement baissé.
Enfin, on peut noter un gain de performance de 25% sur le temps d'entrainement. Même s'il y a plus de couches qu'avant (6 contre 4), il y a moins de neurones par couche, donc les matrices de calcul sont plus petites, et comme la multiplication de matrices est en $O(n^3)$, réduire le n est plus important que d'augmenter le nombre de matrices.

5.7. Autres modifications qui n'ont pas eu d'impact

Ici, nous allons lister brièvement les modifications que nous avons essayé, mais qui n'ont pas eu d'impact remarquable, qui ne sont pas très intéressantes, et qui n'ont pas été retenues.

- Différentes fonctions d'activation: Aucune différence notable entre elles.
- Pseudo-Aléatoire et Véritable Aléatoire: Il y a beaucoup d'aléatoire durement l'entrainement, et nous avons essayé de comparer les 2, mais aucune différence notable non plus. Nous sommes donc restés sur le pseudo-aléatoire par soucis de simplicité.

6. Amélioration des performances et de la parallélisation

Maintenant que nous avons un modèle assez bon, nous allons essayer d'améliorer le temps d'entrainement de notre modèle, et de le paralléliser.
Ici, nous allons nous focaliser uniquement sur l'entrainement, car le temps de chargement des données est négligeable.

Pour mesurer les gains de performance, nous allons utiliser le temps pris pour faire 20 époques, et en déduire le nombre d'époques par seconde que notre modèle est capable de faire.
Pour référence, le modèle actuel est capable de faire 20 époques en ~27.26s, soit ~0.73 époques/s.

D'abord, il faut anaylser les goulots d'étranglement de notre code, et voir où nous pouvons gagner du temps.

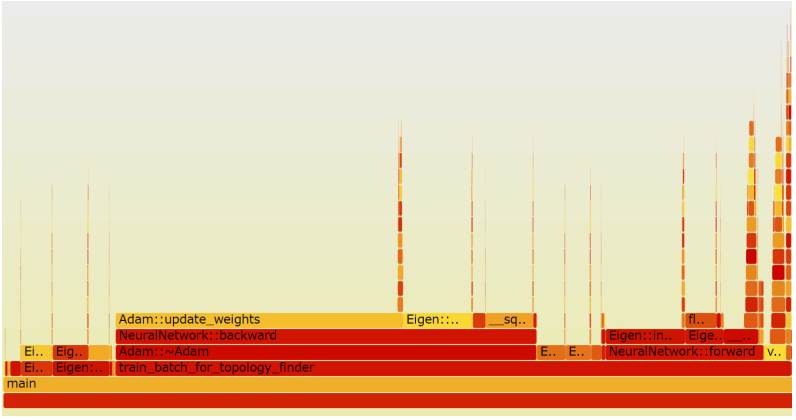


Fig. 27. – Flamegraph d'un entrainement actuel

Dans l'entrainement, on peut voir que la passe arrière (backward) prend beaucoup de temps, notamment à cause de l'optimiseur.
Parmi forward et backward, on peut remarquer qu'il y a beaucoup de temps passé dans des appels à Eigen, ce qui est attendu car les calculs matriciels sont très coûteux.

Clairement, le plus gros des calculs se passent dans les manipulations des matrices et des optimiseurs, et donc ceux-ci vont être nos axes d'optimisations principaux.

6.1. Optimisation des accès mémoire [Performance]

D'abord, nous avons réécrit nos algorithmes pour qu'ils soient plus optimisés pour les accès mémoire.
On a remplacé des boucles imbriquées par des fonctions Eigen, sûrement plus optimisées, comme accéder à une colonne spécifique en appelant .col() au lieu d'itérer avec une boucle classique.
Ensuite, nous avons analysé nos boucles et algorithmes, et nous les avons réécrits pour qu'ils soient plus optimisés pour des matrices en column-major, car c'est la manière dont Eigen stocke les matrices.

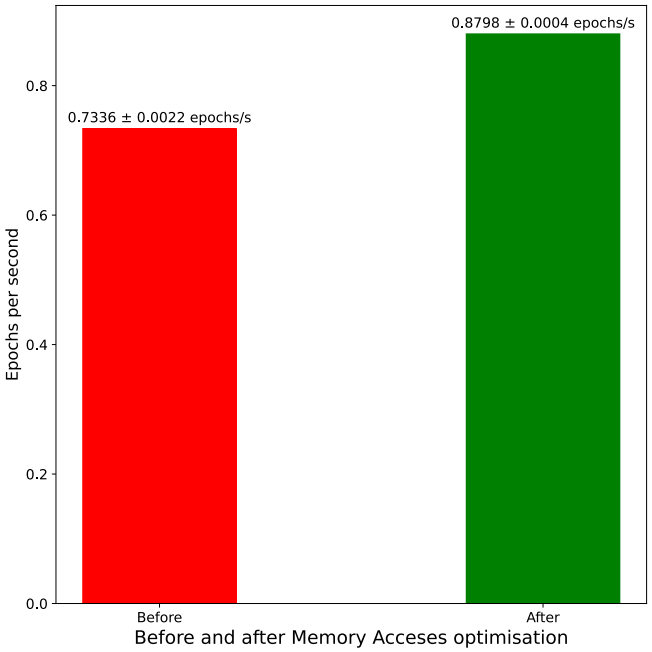


Fig. 28. – Comparaison des performances avant et après optimisation des accès mémoire

Rien que cela nous a permis d'avoir un speedup de ~1.2x, ce qui est déjà un bon début.

6.2. Parallélisation des calculs matriciels [Performance, Parallélisation]

Jusqu'à présent, nous sommes restés sur 1 seul thread, mais Eigen nous permet de paralléliser les calculs matriciels grâce à OpenMP [7]. Juste en rajoutant un appel à `::setNbThreads(x)` au début de notre programme.
De plus, comme nous avions écrit notre algorithme de mini-batch et de descente de gradient par calcul matriciel, au lieu de le faire vecteur par vecteur, nous avons pu bénificier de cette parallélisation sans modifier notre code.

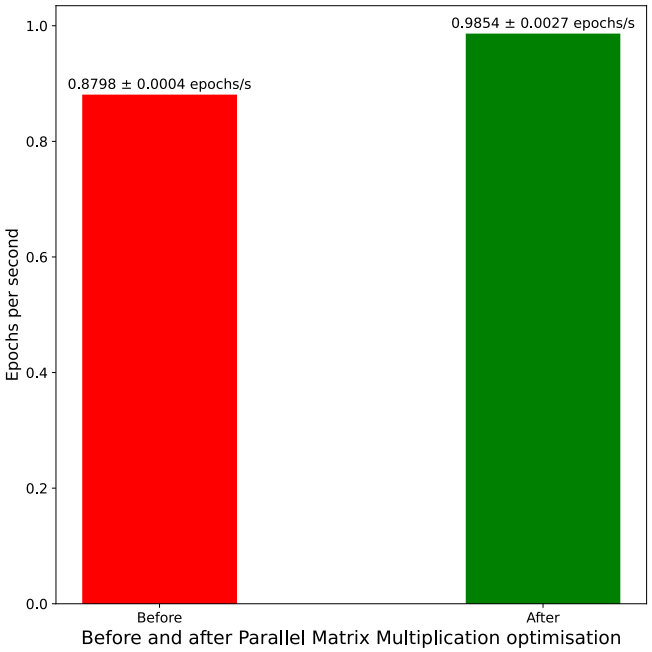


Fig. 29. – Comparaison des performances avant et après parallélisation des calculs matriciels

Cette parallélisation « gratuite » (car nous avions déjà architecturé notre code pour cela) nous a permis d'avoir un speedup de ~1.12x, ce qui est relativement bon, surtout que notre topologie est assez petite, et donc que les matrices ne dépassent pas 1500x1500.

6.3. Parallélisation des optimiseurs [Performance, Parallélisation]

Les optimiseurs travaillant sur des mises-à-jour terme par terme des poids et des biais, sans dépendances entre les itérations, nous avons pu paralléliser ces calculs aussi.

Comme ces bouts de code ne font peu, voire pas d'appels à Eigen, nous avons dû les paralléliser nous même, à l'aide de OpenMP.

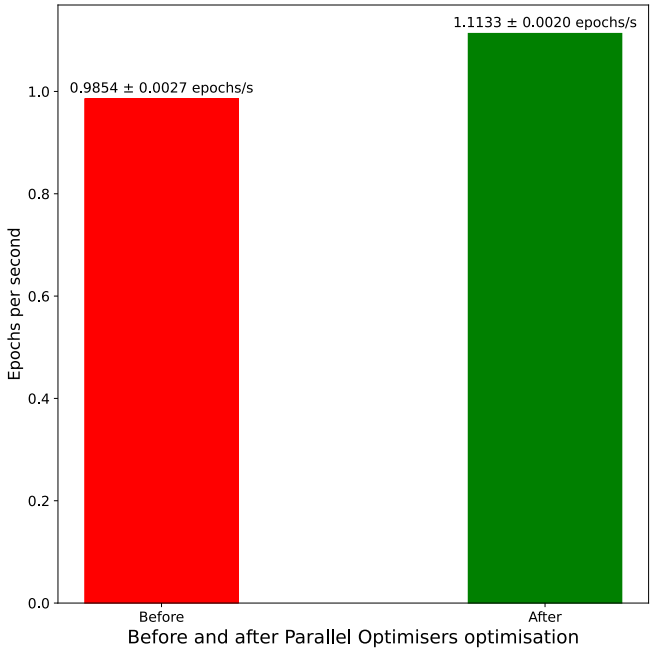


Fig. 30. – Comparaison des performances avant et après parallélisation des optimiseurs

Cette parallélisation nous a permis d'avoir un speedup de ~1.13x, ce qui est relativement bon.
Cela nous a aussi permis d'avoir plus de code parallélisé, et potentiellement plus de gains sur des topologies plus grandes, et sur des ordinateurs plus puissants, d'après la loi d'Amdahl.

6.4. Calculs creux [Performance]

Pour rappel, nos noyaux à analyser sont de tailles très différentes en instructions, certains peuvent tenir en quelques dizaines d'octets tandis que d'autres peuvent en faire jusqu'à plus de 1000.
Nous avons dû mettre en place un système de tokens max et de remplissage pour pouvoir tous les traiter avec le même réseau de neurones. Comme nous avons rempli le vide avec des zéros, nous avons des matrices creuses, et nous nous sommes demandé si nous pouvions en tirer parti.

6.4.1. Analyse des creux

Déjà, voyons à quel point nos matrices sont creuses.

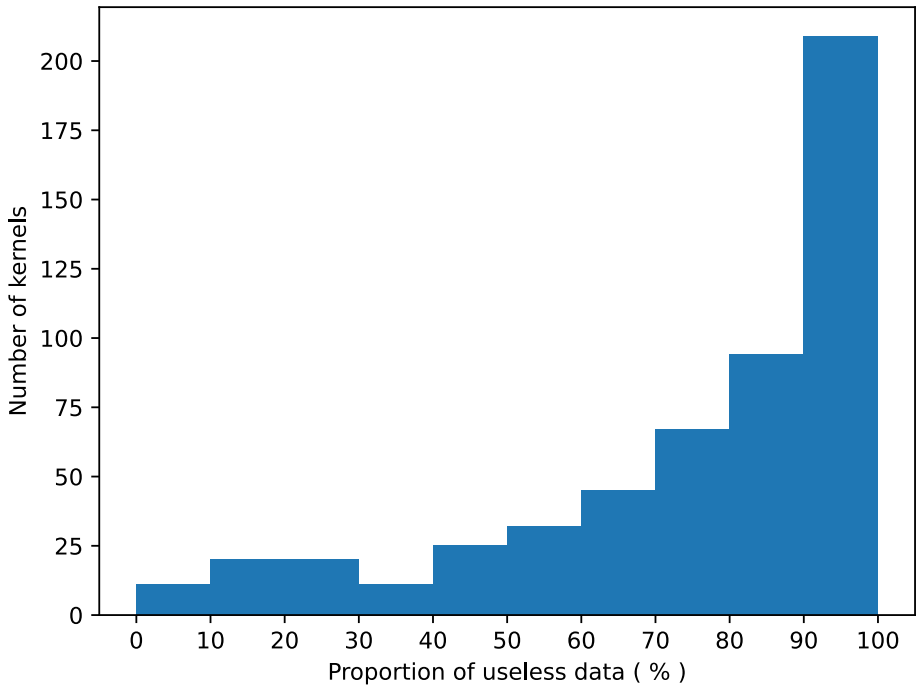


Fig. 31. – Étude de proportions de vide dans les noyaux

Nous pouvons voir qu'il y a une très grande proportion de vide dans les noyaux, notamment plus de 200 d'entre eux ont entre 90% et 100% de vide, et 100 d'entre eux ont entre 80% et 90% de vide. Seule une poignée de noyaux sont plus ou moins pleins.
Adapter notre réseau de neurones pour pouvoir analyser les plus gros noyaux a pénalisé les plus petits noyaux, et donc nous avons décidé de faire de l'algèbre creuse pour optimiser nos calculs.

Avant cela, il faut analyser la structure de nos matrices creuses.
Comme nous avons rempli le vide avec des zéros, pour nos vecteurs d'entrée, tous les zéros sont à la fin.
Et quand on les fusionne en une matrice d'entrées, pour chaque ligne, il existe un seuil au dessus duquel il n'y a plus que des zéros sur la droite.
Exemple:

$$E = \begin{pmatrix} x_{11} & x_{12} & 0 & 0 & 0 \\ x_{21} & 0 & 0 & 0 & 0 \\ x_{31} & x_{32} & x_{33} & 0 & 0 \end{pmatrix}$$

En plus, le premier élément de chaque ligne est la valeur qui encode la taille du noyau, et donc nous pouvons l'utiliser pour savoir où se trouve le seuil directement.

Si on reprend la formule de la multiplication de matrices :

$$c_{ij} = \sum_{k=1}^{n} (a_{ik} * b_{kj})$$

En prenant en compte le seuil de zéros, nous pouvons le réécrire comme :

$$\begin{aligned} c_{ij} &= \sum_{k=1}^{seuil_i} (a_{ik} * b_{kj}) + \sum_{k=seuil_i+1}^{n} (0 * b_{kj}) \\ &= \sum_{k=1}^{seuil_i} (a_{ik} * b_{kj}) \end{aligned}$$

Cette forme de matrice creuse n'étant pas courante, elle n'a pas de nom défini, ni d'implémentation dans les bibliothèques d'algèbre linéaire, notamment Eigen.

6.4.2. Notre implémentation

Comme il n'y a pas d'implémentation de cette forme de matrice creuse dans Eigen, nous avons dû l'implémenter nous même.

Pour mesurer les gains potentiels, nous avons comparé la version naïve (boucle ijk classique, sans algèbre creuse) et notre version creuse (boucle ijk classique, avec algèbre creuse), puis en rajoutant des techniques d'optimisation vu en TP de Techniques d'Optimisations Parallèles comme le cache blocking.
Nous l'avons aussi comparé avec la multiplication de matrices creuses d'Eigen, mais qui est généralisée pour n'importe quel type de matrice creuse, avec un format inspiré des formats Compressed Row/Column, et donc pas optimisée pour notre cas.
Puis nous l'avons comparé avec la multiplication de matrices denses d'Eigen.

Pour les comparer, nous avons multiplier toute la matrice d'entrée $(534 * 1375)$ par une matrice de taille concordante $(1375 * 1375)$, et nous avons mesuré le temps moyen pris pour faire cette multiplication.

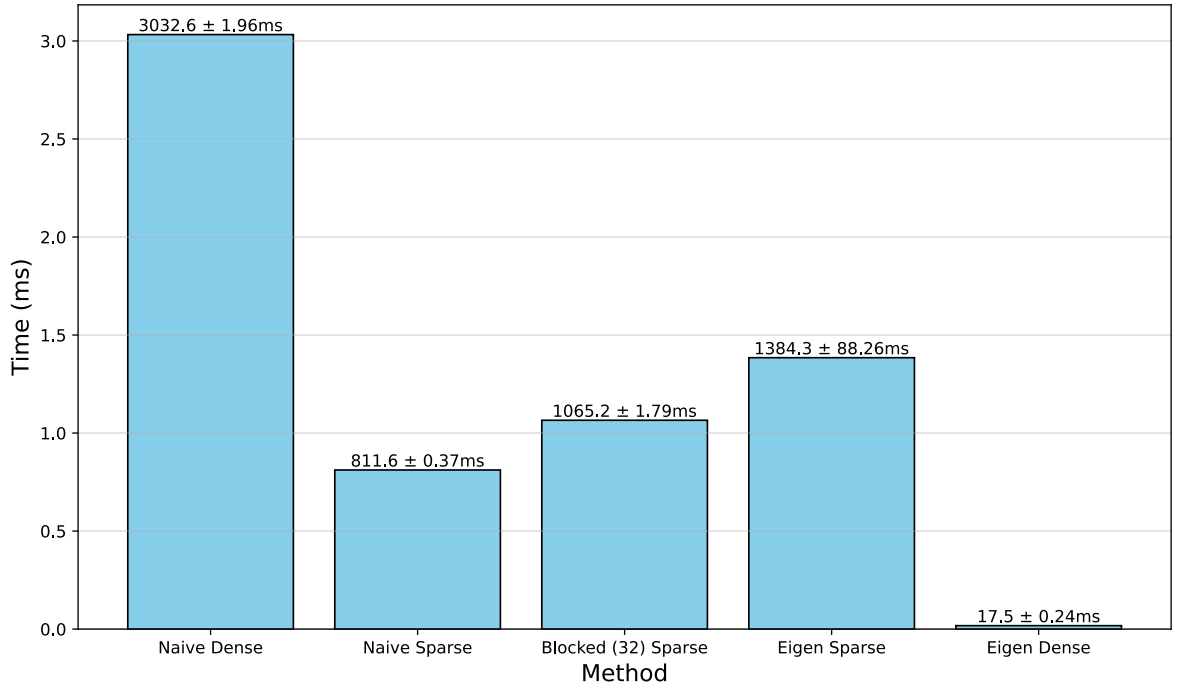


Fig. 32. – Comparaison des performances de la multiplication de matrices creuses

Déjà, nous pouvons voir qu'entre les 2 versions naïves, la version creuse est plus de 3x plus rapide que la version dense, ce qui prouve l'intérêt de l'algèbre creuse.
Ensuite, nous pouvons voir qu'ajouter du cache blocking à la version creuse n'a pas permis d'améliorer les performances, sûrement car la taille de nos matrices est trop petite pour que cela ait un impact.

Pour Eigen, nous pouvons voir que la version creuse est plus lente que notre version creuse, sûrement dû à la construction de la matrice creuse qui prend trop de temps pour être amortie par la multiplication.
Enfin, la version dense d'Eigen est de loin la plus rapide, car cela fait longtemps que les algorithmes de multiplication de matrices denses sont optimisés, et que même avec des calculs en plus à faire, elle est juste plus rapide.

6.4.3. Les sous-matrices d'Eigen

Eigen permet de faire des calculs sur des sous-matrices, avec une fonctionnalité qu'ils appellent « blocks ».
Nous nous sommes dit qu'il serait sûrement intéressant de l'utiliser pour faire nos calculs creux, tout en gardant la performance de la version dense.

Si on multiplie une matrice creuse par une matrice dense,

$$\begin{pmatrix} c_{11} & c_{12} & 0 & 0 & 0 \\ c_{21} & 0 & 0 & 0 & 0 \\ c_{31} & c_{32} & c_{33} & 0 & 0 \end{pmatrix} \times \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \\ d_{41} & d_{42} & d_{43} \\ d_{51} & d_{52} & d_{53} \end{pmatrix}$$

On peut définir 2 sous-matrices, une pour la matrice creuse, où nous pouvons ignorer toutes les colonnes dépassant le maximum de chaque seuil, et une pour la matrice dense, où nous pouvons ignorer toutes les lignes qui vont faire un produit scalaire avec les colonnes vides de la matrice creuse.

$$\begin{pmatrix} c_{11} & c_{12} & 0 \\ c_{21} & 0 & 0 \\ c_{31} & c_{32} & c_{33} \end{pmatrix} \times \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix}$$

Avec le reste de la matrice de résultat contenant des zéros.

Même si nous faisons du mini-batch, comme il y a beaucoup de noyaux assez vides, il est probable que les seuils maximaux soient tout de même relativement bas pour la plupart des batchs.
Nous avons donc mesuré les performances de la multiplication creuse et dense sur des batchs de 16 noyaux aléatoires sur notre ensemble de données.

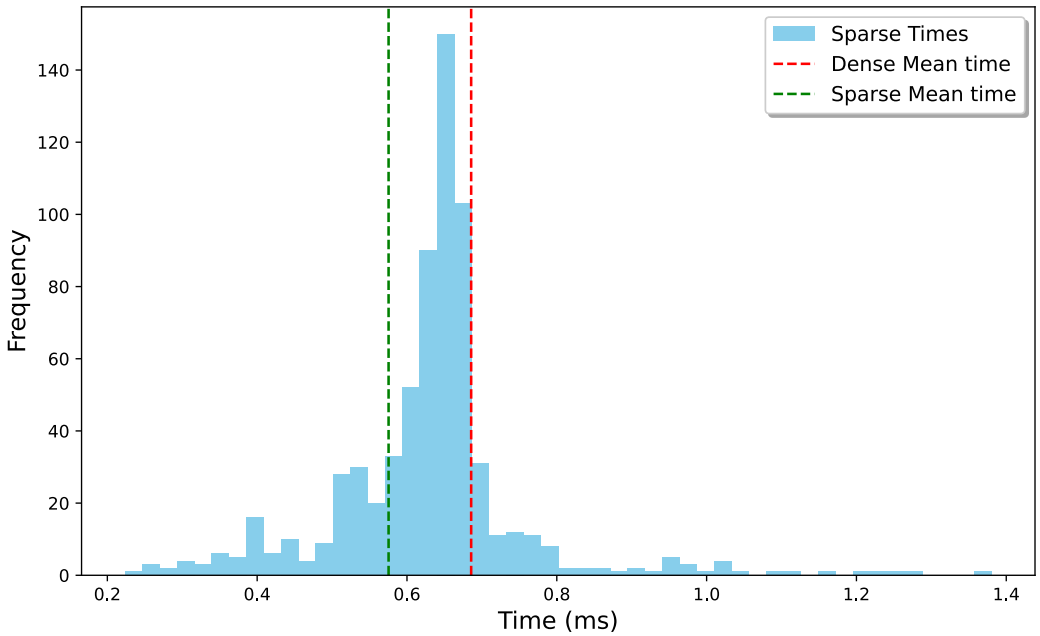


Fig. 33. – Comparaison des performances de la multiplication de matrices creuses et denses sur des batchs de 16 noyaux

Nous pouvons voir une assez grande variabilité entre les batchs pour la version creuse, allant de 0.2ms à 1.4ms.
Mais en général, la version creuse est plus rapide que la version dense, avec une moyenne de ~0.57ms pour la version creuse, et de ~0.69ms pour la version dense.

Après avoir confirmé que la version creuse était plus rapide, nous l'avons implémentée dans notre code, mais malheureusement, elle n'est utilisable que lors de la transformation de la couche d'entrée, car après, ces zéros sont mélangés avec les autres valeurs, et donc il n'y a plus de zéros faciles à ignorer.
Mais voici les gains que cela a apporté,

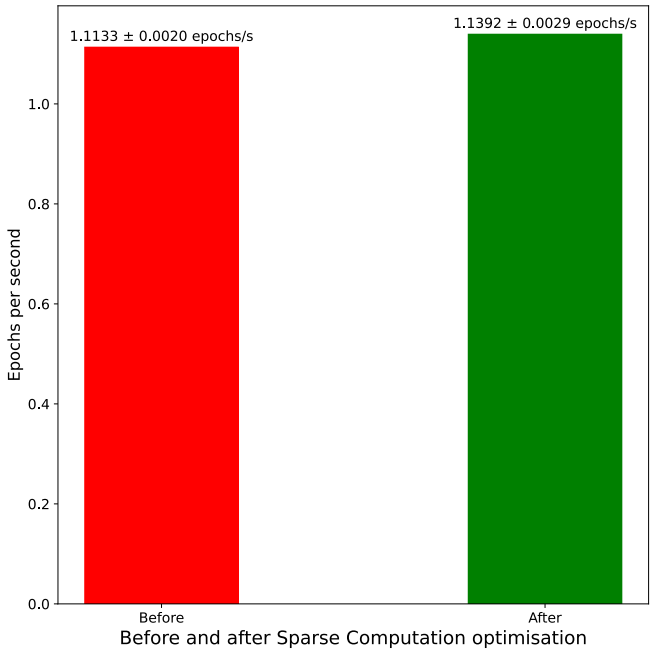


Fig. 34. – Comparaison des performances avant et après ajout d'algèbre creuses

Nous avons quand même un léger speedup de ~1.03x. Ce qui est correct sachant qu'on ne peut l'appliquer qu'une seule fois parmis toute la passe avant.
Il y a un potentiel d'amélioration, car il y a toujours des zéros calculés, mais nous n'avons ni le temps ni le niveau pour pouvoir rivaliser avec Eigen, BLAS, ou LAPACK.

6.5. Approximations mathématiques [Performance]

Dans le milieu de l'IA, il est courant, voire même systématique, de sacrifier un peu de précision arithmétique pour gagner en performance, les réseaux de neurones étant une approximation de la réalité, et les calculs étant très nombreux lors de l'entrainement, cela semble assez naturel de le faire.

Par exemple, il y a eu une émergence récente des flottants avec précision réduite, comme les flottants 16 bits, voire même 8 bits ou 4 bits, car plus rapides à traiter. Ceci est juste pour illustrer notre propos, nous n'avons pas testé ces formats, car notre hardware ne le supporte pas.

Nous avons d'abord rajouté les optimisations de compilation, comme `-ffast-math` et `-funsafe-math-optimizations`, qui donnent le droit au compilateur de réarranger les calculs en arithmétique flottante, de ne pas traiter les cas comme l'infini ou NaN, et de ne pas respecter les normes IEEE 754.

Ensuite, nous avons utilisé des algorithmes approximatifs pour les fonctions mathématiques, notamment dans les fonctions d'activation et optimiseurs qui peuvent utiliser des fonctions assez coûteuses comme l'exponentielle, la racine carrée, etc.

Par exemple, dans l'optimiseur Adam, il y a une division par une racine carrée. Nous avons remplacé ce calcul par le célèbre algorithme de la racine carrée inverse rapide, découvert par John Carmack, pour optimiser des calculs de collisions dans Quake III. Cependant, les processeurs x86 modernes possédant une instruction qui fait exactement cela, utiliser cet algorithme ne nous a pas apporté de gain de performance.

Il reste cependant la fonction exponentielle, utilisée dans certaines fonctions d'activation, comme Sigmoid, qui est très coûteuse à calculer. Nous avons donc exploré plusieurs approximations de cette fonction, que nous allons présenter ici, avant de comparer les performances et précisions de ces algorithmes.

- Approximation polynomiale

D'après le théorème de Taylor-Young, il est possible d'approximer n'importe quelle fonction par une série de Taylor, qui est un polynôme de degré n. Pour l'exponentielle: $e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + ... + \frac{x^n}{n!}$ La suite infinie donne la fonction exacte, mais on peut s'arrêter à un certain degré arbitraire pour obtenir une approximation de l'exponentielle.

- Approximations par produit tronqué

L'exponentielle peut être formulée comme $e^x = \lim_{n \to \inf} \left(1 + \frac{x}{n}\right)^n$

Comme pour l'approximation polynomiale, on peut s'arrêter à un certain n pour obtenir une approximation de l'exponentielle. De plus, si on prend un n puissance de 2, on peut utiliser l'algorithme de la mise à la puissance rapide pour obtenir encore plus de performance. Par exemple, si on prend $n = 2^k$, on peut écrire $x^n = x^{2^k} = \left(x^{2^{k-1}}\right)^2 = \left(\left(x^{2^{k-2}}\right)^2\right)^2 = ... = \left(\left(\left(x^2\right)^2\right)^2 ...\right)^2$. Ce qui nous permet de réduire le nombre d'opérations à $\log_2(n)$ multiplications.

- Algorithme de Schraudolph [8]

Cette algorithme utilise avec ingéniosité le format IEEE-754 et manipule les bits de la représentation binaire des flottants pour approximer e^x.

Déjà, on va réééécrire l'exponentielle comme $e^x = 2^{\frac{x}{\ln(2)}}$.

$\ln(2)$ étant une constante, on peut la pré-calculer, de plus, elle est présente dans la librairie math de C.

Ensuite, on va utiliser le fait que les nombres flottants sont décomposés en 3 parties : le signe, l'exposant et la mantisse. Pour les nombres flottants sans partie décimale, décaler les bits de la mantisse vers les bits d'exposant transforme x en exactement 2^x.

Ensuite, on va faire une interpolation linéaire entre les 2 entiers les plus proches de $\frac{x}{\ln(2)}$, pour pouvoir utiliser la transformation en 2^x. Cette interpolation linéaire est très rapide, car elle ne nécessite que 2 multiplications et 1 addition, en plus de rester une très bonne approximation de la valeur réelle.

Maintenant, voici ces différentes approximations sur l'intervalle $[-10, 10]$, avec un pas de 0.001

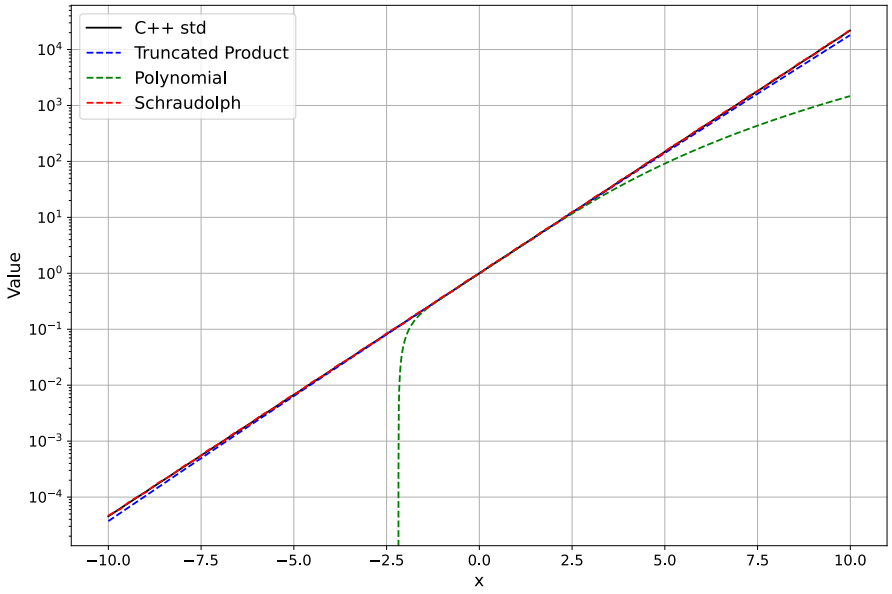


Fig. 35. – Comparaison des différentes approximations de l'exponentielle à l'implémentation standard C++

Nous pouvons remarquer immédiatement une approximation qui sort du lot: l'approximation polynomiale (faite en degré 5).
Elle est très proche de la fonction réelle pour des valeur entre −2 et 2.5, mais elle diverge rapidement pour les autres valeurs. Cela est dû à la nature de la série de Taylor, qui est une approximation au voisinage d'un point, et qui diverge rapidement si on s'en éloigne trop.
Les autres approximations sont toutes très proches de la fonction réelle, mais celle de Schraudolph semble être la plus précise.

Analysons les plus en détails, et comparons leurs performances.
Pour la précision, nous avons calculé les valeurs sur la fonction Sigmoid et sa dérivée, sur l'intervalle $[-100, 100]$, avec un pas de 0.001, et nous avons calculé l'écart type des différences absolues entre l'approximation et l'implémentation standard de C++ pour chaque valeur.
Pour la performance, nous avons mesuré le temps d'exécution de Sigmoid et de sa dérivée, total pour chaque valeur de l'intervalle, ce qui fait 200′000 values à calculer, répété 5 fois.
Le but va être d'avoir une fonction qui soit la plus précise possible, mais aussi la plus rapide possible.

Method	σ error	σ time (ms)	σ' error	σ' time (ms)
C++ std	-	0.932	-	1.765
Polynomial	5.29e-01	0.665	1.48e+01	1.446
Limit	3.21e-05	0.815	2.71e-05	1.605
Schraudolph	8.99e-05	0.744	4.50e-05	1.259

Fig. 36. – Comparaison des différentes approximations de l'exponentielle à l'implémentation standard C++, avec meilleurs candidats mis en avant

Ici, nous pouvons voir que l'approximation polynomiale est très rapide, mais elle est très imprécise, avec un écart type entre 0.5 et 1.5, ce qui est très mauvais, mais attendu au vu de la courbe présentée avant.
L'approximation par limite tronquée est très précise, de l'ordre de $3 * 10^{-5}$, et est plus rapide que l'implémentation standard de C++.
Cependant, l'algorithme de Schraudolph réussit à être assez bon en performance et précision, battant

l'approximation par limite tronquée en performance, sans perdre tant de précision (ordre de $9 * 10^{-5}$). Nous avons donc choisi de l'utiliser.

Et sur un entrainement, voici le gain de performance que nous avons eu:

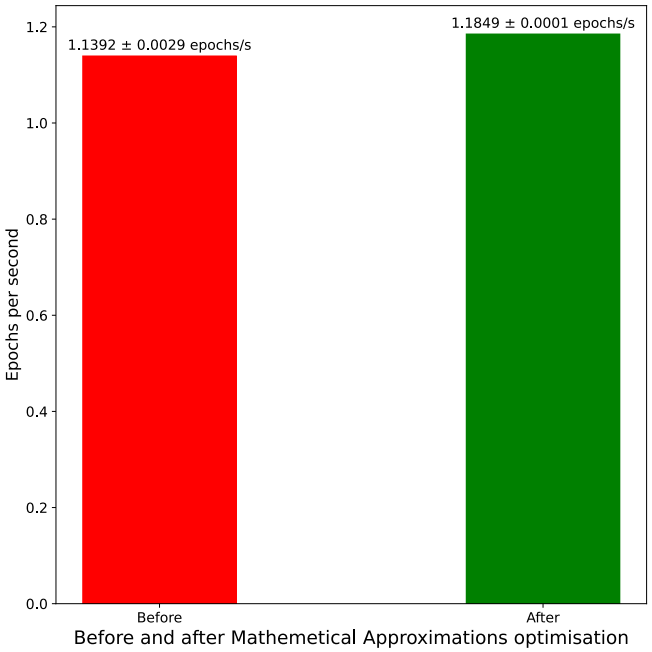


Fig. 37. – Comparaison des performances avant et après l'optimisation des calculs mathématiques

Grâce à l'algorithme de Schraudolph, et des optimisations de compilation, nous sommes passés de ~1.140 époques/s à ~1.185 époques/s, soit un speedup de ~1.04x. Cela n'est pas énorme, car l'implémentation standard de C++ est déjà très optimisée, et de plus, les fonctions d'activations sont moins sollicitées ($O(n^2)$) que les calculs matriciels ($O(n^3)$), mais c'est toujours bon à prendre.

On peut tout de même se poser sur l'impact de la précision du réseau, mais nous verrons lors de la présentation des résultats finaux que cela n'a pas eu d'impact significatif.

6.6. Portage GPU [Performance, Parallélisation]

Les GPU sont de plus en plus utilisés pour l'entrainement de réseaux de neurones, car ils sont très performants pour les calculs matriciels.
On peut même dire qu'ils étaient conçus pour cela à l'origine, car les rendus 3D sont fait avec des calculs matriciels.
Nous allons donc explorer cette voie afin de gagner encore plus de performance.

Pour cela, nous avons implémenté un wrapper minimal de Vulkan pour pouvoir envoyer des données au GPU, et exécuter les kernels dessus.
Nous avons choisi Vulkan car c'est une API portable, et non propriétaire comme CUDA ou HIP, reste très performante, et que nous avions déjà quelques connaissances dessus.
Ensuite, on porte le code de multiplication de matrices sur le GPU, en utilisant les compute shaders de Vulkan.

Maintenant, comparons les performances de la version CPU et GPU, pour des matrices de tailles entre 100x100 et 1500x1500, car c'est la plage de tailles que nous avons pour notre topologie.

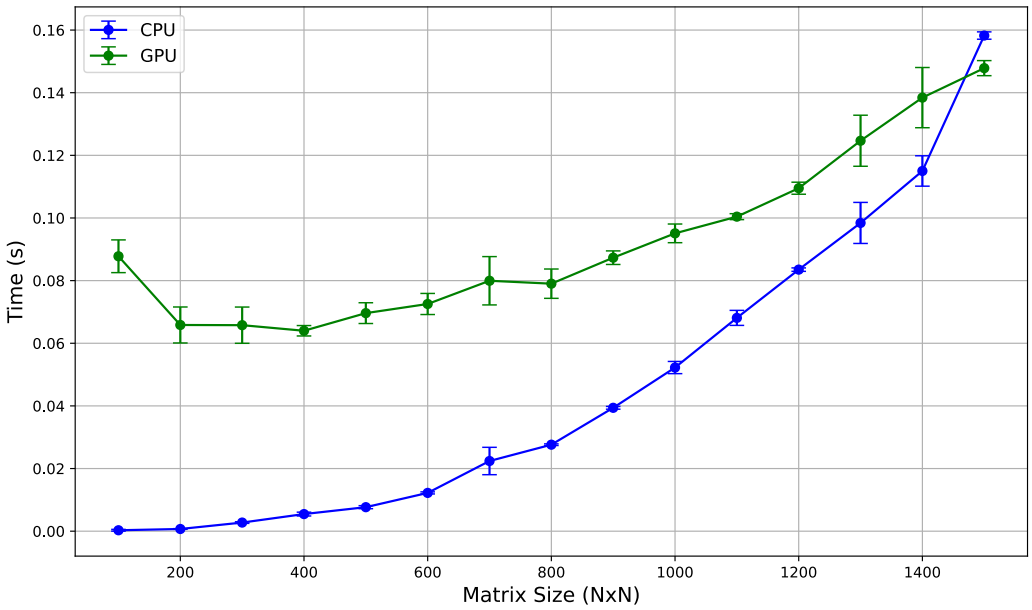


Fig. 38. – Comparaison des performances entre la version CPU et GPU

Pour la version CPU, nous obtenons une courbe polynomiale, ce qui est normal, car la multiplication de matrices est un algorithme de complexité $O(n^3)$.

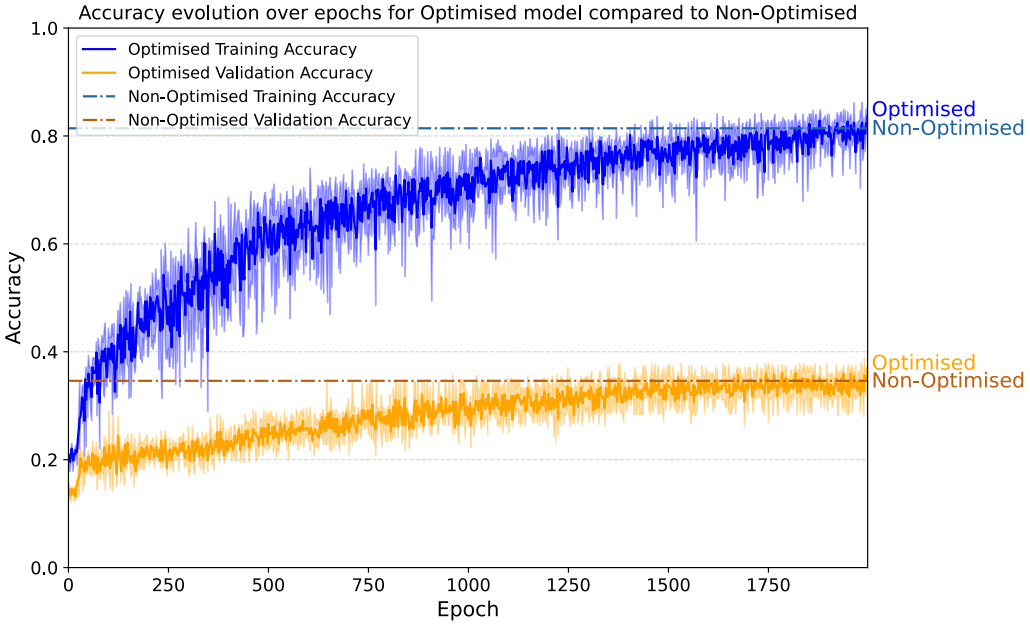
Pour la version GPU, nous avons 2 phases.
Entre **100x100** et **800x800**, nous avons un plateau, dû à la latence d'envoi des données au GPU, qui est plus importante que le temps de calcul.
Entre **800x800** et **1500x1500**, nous avons aussi une courbe polynomiale, mais avec un meilleur scaling que la version CPU, car on peut faire plus de calculs en parallèle sur le GPU. D'ailleurs l'implémentation GPU réussit à battre la version CPU pour 1500x1500.

Le portage GPU pourrait donc apporter des gains, mais nos matrices sont trop petites pour que l'overhead d'envoi des données soit amorti par le temps de calcul.

Peut être que si nous avions porter le reste du code, et avoir toutes les données en local sur la mémoire du GPU, nous aurions eu des gains.
Mais c'est très compliqué à faire, et nous ne connaissons pas assez les architectures GPU pour le faire de manière efficace.

6.7. Gains finaux

Il est maintnenant le temps de voir les résultats finaux de notre modèle, en le comparant avec la version avant les optimisations.



Summary of Training and Validation results

Metric	Optimised	Non-Optimised	Difference
Training time	1633.11s ± 21.40s	2711.84s ± 8.61s	-1078.73s
Training accuracy	0.84	0.81	+0.02
Validation accuracy	0.37	0.35	+0.02

Fig. 39. – Comparaison des résultats d'entrainement de la version optimisée et non optimisée

On peut observer un speedup de ~1.66x sur le temps d'entrainement, avec aucune perte de précision. Même, la précision a légèrement augmenté, mais cela est sûrement au hasard que de réelles améliorations.

7. Récapitulatif de l'évolution du projet

Passons maintenant en revue les différentes étapes de notre projet, et les améliorations que nous avons apportées.

7.1. Améliorations de la précision

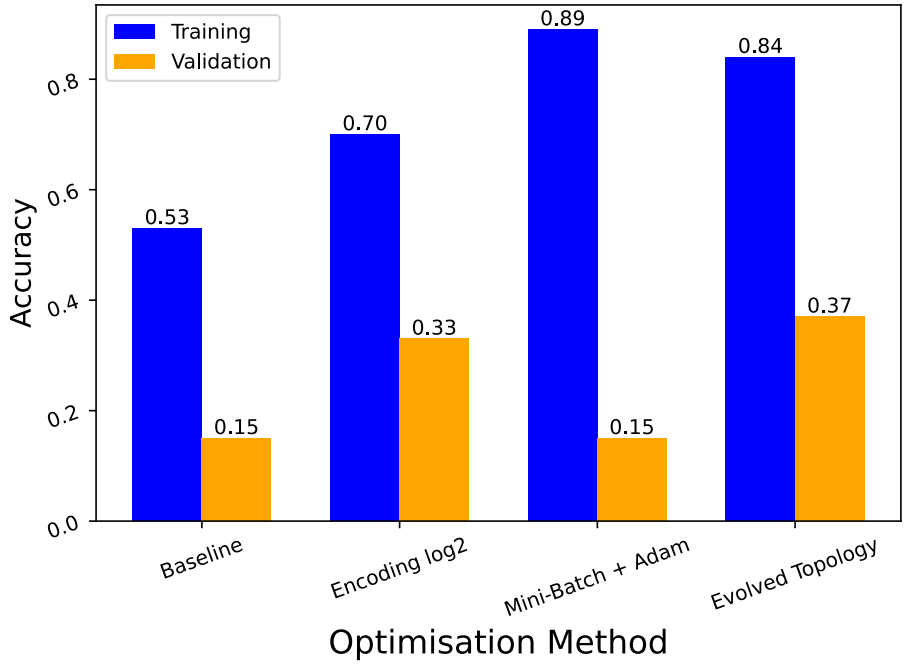


Fig. 40. – Récapitulatif des améliorations de la précision faites

Nous avons réussi à améliorer la précision d'entrainement de 53% à 84%, et la précision de validation de 15% à 37%. Les impacts les plus gros fûrent les optimiseurs ainsi que l'algorithme génétique.

Même si cela ne semble pas énorme, il faut garder en tête que nous avons un jeu de données très petit.

7.2. Améliorations de la performance

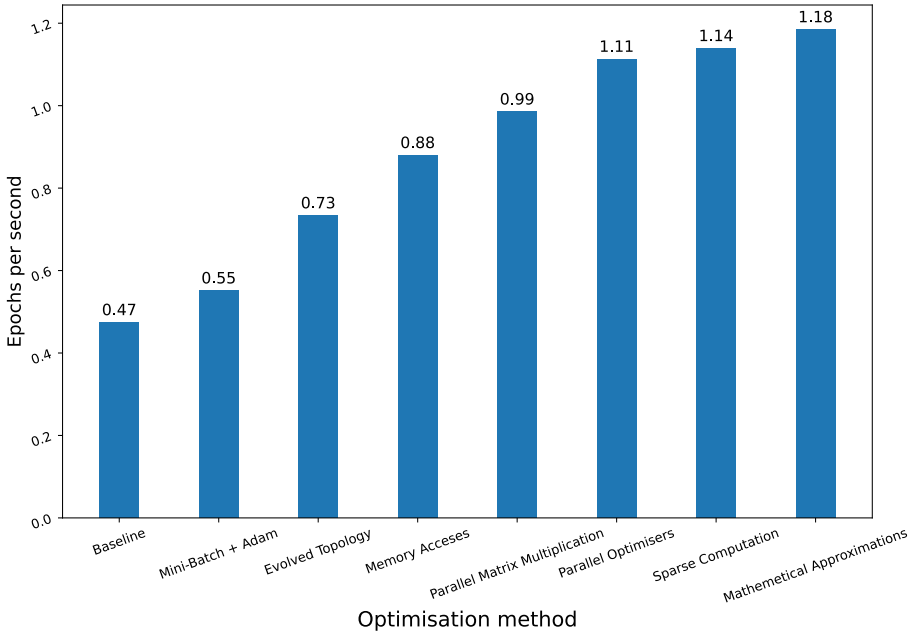


Fig. 41. – Récapitulatif des optimisations faites

Comparé à notre toute première version, nous avons réussi à avoir un speedup de ~2.51x, en considérant que la plupart des calculs sont de l'algèbre linéaire, qui a déjà été optimisée depuis fort longtemps, nous sommes assez satisfaits de ce résultat.

8. Critiques, perspectives, et conclusion

Nous devons cependant rester critique sur nos résultats, car il y a plusieurs biais possibles :

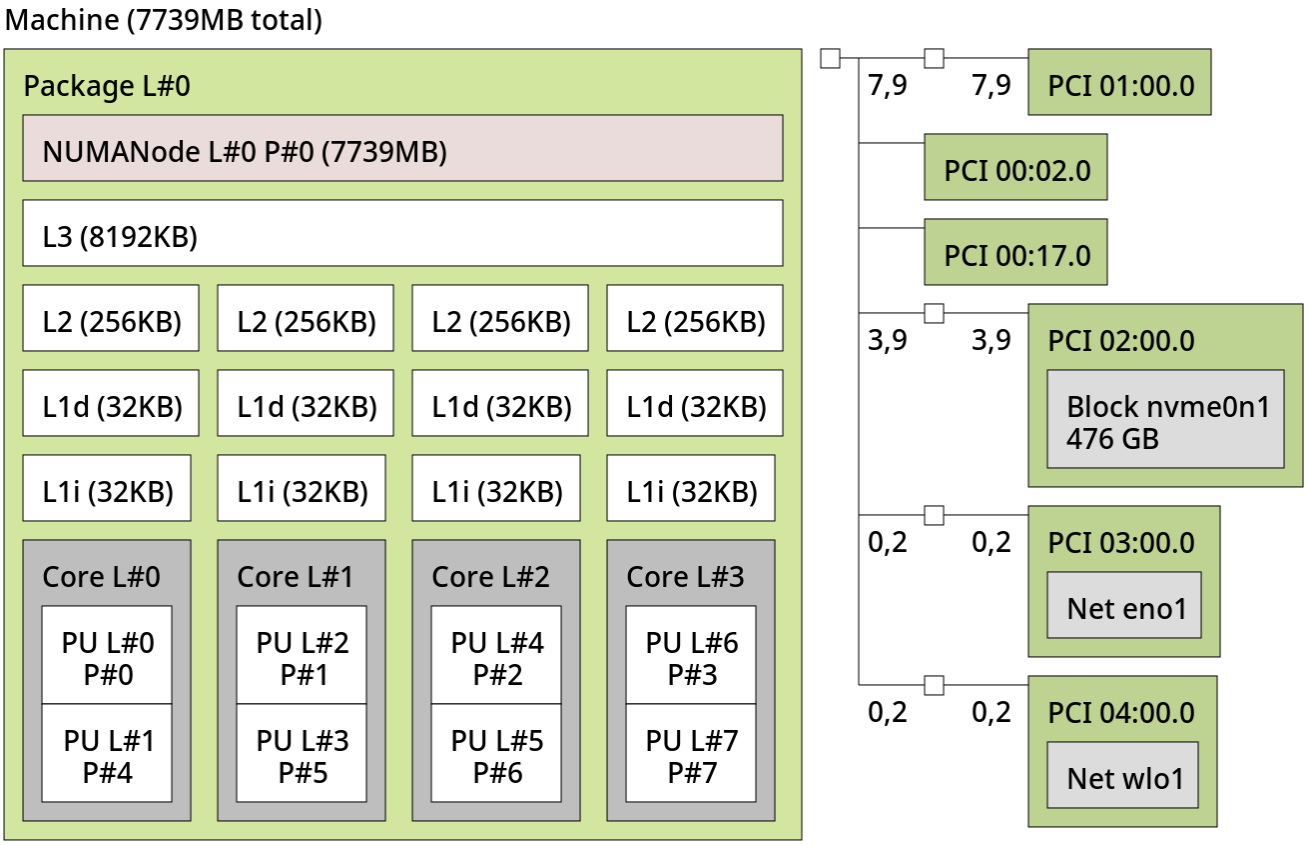
- **Notre jeu de données** : Il est possible que notre jeu de données ne soit pas représentatif de la réalité.
  Nous avons été fourni principalement des noyaux d'algèbre linéaire, qui sont souvent très similaires entre eux, et sans branchements très complexes. Nous avons tenté de diversifier un peu les données avec des algorithmes de tri, des fonctions récursives etc.. mais il est possible que cela ne soit pas suffisant.
  De plus, nous avons arbitrairement choisi une taille maximale de kernel et de nombre de cycles pris en compte, ce qui rend notre modèle moins généralisable.
- **Puissance de calcul** : Nous avons fait tourner nos modèles sur un ordinateur personnel, avec une carte graphique moyenne, et un processeur moyen. Il est possible que nous aurions pu obtenir de meilleurs résultats car entraîner un réseau de neurones est très gourmand en ressources.

Il y a aussi la question de la scalabilité générale du code: est-ce que nous pourrions passer à l'échelle pour un plus gros jeu de données, ou un plus gros réseau de neurones?

Comme la plupart du temps est passé dans des calculs matriciels, il serait intéressant de comparer différentes bibliothèques d'algèbre linéaire, comme BLAS ou LAPACK.
Il reste aussi la piste de faire passer entièrement notre code sur le GPU, si la communication est le problème, mais cela aurait nécessité de prendre le temps de tout réécrire, et nous n'avons pas énormément de connaissance en architectures GPU.

Malgré cela, nous avons réussi à obtenir un modèle qui est relativement précis, surtout pour un jeu de données aussi petit.
Notre code est un framework de réseau de neurones relativement complet et modulaire, avec des encodeurs, des fonctions d'activations, des optimiseurs, du mini-batch, du calcul sparse, etc…
De plus, il propose des fonctionnalités que nous n'avons vu nulle part ailleurs, comme les différentes manières d'explorer l'espace des paramètres, ou un algorithme génétique pour trouver la meilleure topologie.

A Machine utilisée durant les entrainements



Ordinateur portable, processeur Intel Core i5 9ème génération à 4 cœurs, avec 8 threads, 8Go de RAM, et une carte graphique intégrée NVIDIA GeForce GTX 1050.
Compilateur: gcc 13.2.1
OpenMP: 4.5
OS: Linux, Fedora 39

Bibliographie

[1] Martin Leitner-Ankerl, « nanobench ». [En ligne]. Disponible sur: https://nanobench.ankerl.com/

[2] H. B. Curry, « The method of steepest descent for non-linear minimization problems », *Quarterly of Applied Mathematics*, vol. 2, p. 258-261, 1944, [En ligne]. Disponible sur: https://api.semanticscholar.org/CorpusID:125304075

[3] Eigen, « Eigen ». [En ligne]. Disponible sur: http://eigen.tuxfamily.org/index.php?title=Main_Page

[4] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, George E. Dahl, « On Empirical Comparisons of Optimizers for Deep Learning ». [En ligne]. Disponible sur: https://arxiv.org/pdf/1910.05446

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, et R. Salakhutdinov, « Dropout: A Simple Way to Prevent Neural Networks from Overfitting », *Journal of Machine Learning Research*, 2014, [En ligne]. Disponible sur: https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[6] L. Bottou, O. Chapados, J. DeCoste, et J. Weston, « L2 Regularization for Learning Kernels », 2012, [En ligne]. Disponible sur: https://arxiv.org/abs/1205.2653

[7] OpenMP Architecture Review Board, « OpenMP ». [En ligne]. Disponible sur: https://www.openmp.org/

[8] Nicol Schraudolph, « A Fast, Compact Approximation of the Exponential Function ». [En ligne]. Disponible sur: https://nic.schraudolph.org/pubs/Schraudolph99.pdf