Implémentation de CNNs avec NumPy

Lenny Malard
Melpy Project

Abstract
L'implémentation de réseaux de neurones artificiels à partir de zéro constitue un véritable défi pédagogique dans l'apprentissage du Deep Learning. Ce processus exige une compréhension approfondie des concepts fondamentaux, nécessaire pour transcrire ces connaissances dans la conception d'algorithmes optimisés. C'est dans cette optique d'apprentissage que j'ai créé Melpy, une bibliothèque de Deep Learning qui, à l'origine, n'implémentait que de simples perceptrons multicouches[2]. C'est donc pourquoi, j'ai entrepris d'implémenter les réseaux de neurones convolutifs afin d'étendre les possibilités offertes par Melpy et d'enrichir mes connaissances.

Étant donné que Melpy repose sur l'utilisation de NumPy, de nouvelles couches spécifiques à ces architectures ont été ajoutées et optimisées en exploitant les fonctionnalités de cette bibliothèque. Les résultats obtenus montrent que Melpy est désormais capable de produire des performances globalement comparables à celles de Keras, dans la classification de jeux de données telles que MNIST[3] et CIFAR-10[5]. Cependant, ces résultats mettent également en évidence les limites de la bibliothèque en termes d'optimisation computationnelle, notamment dû à l'interprétation du code en Python, ce qui constitue un axe d'amélioration pour l'avenir.

Table des matières

Introduction

Les réseaux de neurones convolutifs (CNNs) ont été cités pour la première fois par Yann LeCun en 1998, dans le papier *"Gradient-Based Learning Applied to Document Recognition"*[7]. Il y met en avant l'apprentissage automatique des motifs présents dans les images d'un jeu de données et démontre l'efficacité du modèle dans des tâches de classification. Cependant, ce n'est qu'en 2012 que cette approche gagna en popularité grâce à la victoire d'AlexNet[6]

durant la compétition de détection d'images ImageNet. Depuis, les CNNs sont reconnues comme des architectures performantes dans le domaine de la Vision par Ordinateur, et sont exploités pour de nombreuses tâches telles que la reconnaissance faciale, l'estimation de poses ou encore la reconnaissance d'actions [8]. Il est donc aujourd'hui naturel que les bibliothèques de Deep Learning les proposent, et c'est pourquoi Melpy ne déroge pas à la règle, bien que la bibliothèque ne reste qu'un support académique. Nous verrons dans ce rapport la théorie derrière les CNNs et les choix dans leur implémentation pour des tâches de classification. Nous discuterons également des performances de Melpy par rapport à Keras, un framework de référence pour la réalisation de modèles de Deep Learning de haut niveau.

Théorie

Dans cette section, nous verrons l'aspect théorique derrière les CNNs.

2.1 L'architecture

Les CNNs se composent de deux parties principales : une partie dédiée à l'extraction des caractéristiques des images d'entrée, et une autre à leur classification.

En effet, le réseau va d'abord utiliser une couche de Convolution pour créer des cartes de caractéristiques, puis va ensuite réduire la taille des données grâce à une couche de Pooling. Ce processus est répété et ajusté selon la profondeur nécessaire au réseau.

Ensuite, les cartes de caractéristiques sont transformées en vecteurs aplatis, où chaque pixel devient une caractéristique. Ces vecteurs alimentent un perceptron multicouche (MLP), qui, via des couches entièrement connectées, détermine la classe de chaque image.

Pour imager, nous pouvons prendre comme exemple l'architecture de la Figure 1 :

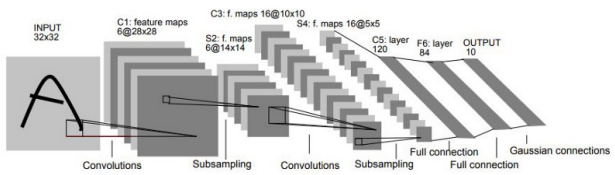

Figure 1: Architecture de LeNet-5 [7]

Nous observons deux couches de Convolution et deux couches de Pooling. On voit également le MLP composé d'une couche d'entrée, d'une couche cachée et d'une couche de sortie.

2.2 Les couches de Convolution

Nous verrons ici les calculs de propagation avant et de propagation arrière nécéssaires à l'entrainement des couches de convolution.

L'opération de convolution est définie par :

$$y[m,n] = x[m,n] * h[m,n] \tag{1}$$

$$= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j]h[m-i,n-j] \tag{2}$$

Avec x l'entrée, y la sortie et h le filtre.

Cependant, pour simplifier les calculs, nous éffectuerons des opérations de cross-corrélation, définis par :

$$y[m,n] = x[m,n] \star h[m,n] \tag{3}$$
$$= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j]h[m+i,n+j] \tag{4}$$
$$= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j]rot_{180°}\big[h[m-i,n-j]\big] \tag{5}$$
$$= x[m,n] * rot_{180°}\big[h[m,n]\big] \tag{6}$$

Pour nos calculs, nous posons les termes suivant :

l l'indice d'une couche
y la sortie
x l'entrée
w le filtre (ou kernel)
σ la fonction d'activation
L la fonction coût
m, n les indices des pixels de la sortie
i, j les indices des pixels du filtre
m', n' les indices des pixels de la dérivée de l'entrée
i', j' les indices des pixels de la dérivée du filtre
H, W la hauteur et la largeur de la sortie
k la taille du filtre

Afin de simplifier les calculs théoriques, nous nous restreignons à une cross-corrélation sans padding et sur qu'un seul canal.

2.2.1 Propagation Avant

Une couche de convolution aura comme opération en propagation avant :

$$y^l = \sigma(y^{l-1}) \star w^l \tag{7}$$
$$= x^l \star w^l \tag{8}$$
$$y^l_{m,n} = \sum_{j=0}^{k-1} \sum_{i=0}^{k-1} \sigma(y^{l-1}_{i,j}) \cdot w^l_{m+i,n+j} \tag{9}$$
$$= \sum_{j=0}^{k-1} \sum_{i=0}^{k-1} \sigma(y^{l-1}_{m+i,n+j}) \cdot w^l_{i,j} \tag{10}$$

Cela génère des cartes de caractéristiques qui mettent en évidence certains motifs d'une image, aidant le MLP à la classifier. Le nombre de cartes correspond au nombre de filtres dans la couche. Par exemple, pour obtenir 6 cartes de caractéristiques, il faut utiliser 6 filtres différents.

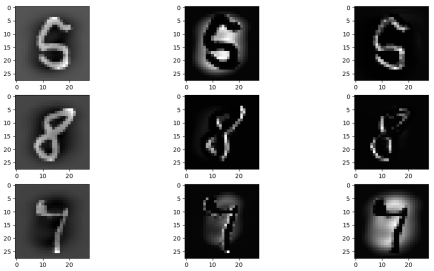

Figure 2: Cartes de caractéristiques générées avec Melpy sur le dataset MNIST. Les entrées se situent dans la première colonne.

Sur la Figure 2, nous observons que certains pixels de l'image d'entrées sont activés et d'autres non. On peut l'expliquer par le fait que les filtres vont être entrainés à activer uniquement certaines caractéristiques telles que les bords verticaux, les bords horizontaux ou des formes plus complexes si nécessaire.

2.2.2 Propagation Arrière

En propagation arrière, nous aurons besoin des dérivées de la fonction coût L par rapport à l'entrée x^l et par rapport au filtre w^l.

Commençons par calculer $\frac{\partial L}{\partial w^l_{i',j'}}$:

$$\frac{\partial L}{\partial w^l_{i',j'}} = \sum_{n=0}^{W-1}\sum_{m=0}^{H-1} \frac{\partial L}{\partial w^l_{i',j'}} \cdot \frac{\partial x^{l+1}_{m,n}}{\partial x^{l+1}_{m,n}} \tag{11}$$

$$= \sum_{n=0}^{W-1}\sum_{m=0}^{H-1} \frac{\partial L}{\partial x^{l+1}_{m,n}} \cdot \frac{\partial x^{l+1}_{m,n}}{\partial w^l_{i',j'}} \tag{12}$$

On pose :
$\frac{\partial L}{\partial x^{l+1}_{m,n}} = \delta^{l+1}_{m,n}$

$$\implies \frac{\partial L}{\partial w^l_{i',j'}} = \sum_{n=0}^{W-1}\sum_{m=0}^{H-1} \delta^{l+1}_{m,n} \cdot \frac{\partial x^{l+1}_{m,n}}{\partial w^l_{i',j'}} \tag{13}$$

$$\frac{\partial x^{l+1}_{m,n}}{\partial w^l_{i',j'}} = \frac{\partial}{\partial w^l_{i',j'}} \left(\sum_{j=0}^{W-1}\sum_{i=0}^{H-1} \sigma(x^l_{m+i,n+j}) \cdot w^l_{i,j} \right) \tag{14}$$

$$= \frac{\partial}{\partial w^l_{i',j'}} (\sigma(x^l_{m+i',n+j'}) \cdot w^l_{i',j'}) \tag{15}$$

$$= \sigma(x^l_{m+i',n+j'}) \cdot \frac{\partial}{\partial w^l_{i',j'}} (w^l_{i',j'}) \tag{16}$$

$$= \sigma(x^l_{m+i',n+j'}) \tag{17}$$

$$\implies \frac{\partial L}{\partial w^l_{i',j'}} = \sum_{n=0}^{W-1}\sum_{m=0}^{H-1} \delta^{l+1}_{m,n} \cdot \sigma(x^l_{m+i',n+j'}) \tag{18}$$

$$= \delta^{l+1}_{i',j'} \star \sigma(x^l_{i',j'}) \tag{19}$$

Calculons ensuite $\frac{\partial L}{\partial x^l_{m',n'}}$:

$$\frac{\partial L}{\partial x^l_{m',n'}} = \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \frac{\partial L}{\partial x^l_{m',n'}} \cdot \frac{\partial x^{l+1}_{m'-i,n'-j}}{\partial x^{l+1}_{m'-i,n'-j}} \tag{20}$$

$$= \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \frac{\partial L}{\partial x^{l+1}_{m'-i,n'-j}} \cdot \frac{\partial x^{l+1}_{m'-i,n'-j}}{\partial x^l_{m',n'}} \tag{21}$$

$$= \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \delta^{l+1}_{m'-i,n'-j} \cdot \frac{\partial x^{l+1}_{m'-i,n'-j}}{\partial x^l_{m',n'}} \tag{22}$$

$$\frac{\partial x^{l+1}_{m'-i,n'-j}}{\partial x^l_{m',n'}} = \frac{\partial}{\partial x^l_{m',n'}} \left(\sum_{j'=0}^{k-1}\sum_{i'=0}^{k-1} \sigma(x^l_{m'-i+i',n'-j+j'}) \cdot w^l_{i',j'} \right) \tag{23}$$

$$= \frac{\partial}{\partial x^l_{m',n'}} (\sigma(x^l_{m',n'}) \cdot w^l_{i,j}) \tag{24}$$

$$= w^l_{i,j} \cdot \frac{\partial}{\partial x^l_{m',n'}} (\sigma(x^l_{m',n'})) \tag{25}$$

$$= w^l_{i,j} \cdot \sigma'(x^l_{m',n'}) \tag{26}$$

$$\implies \frac{\partial L}{\partial x^l_{m',n'}} = \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \delta^{l+1}_{m'-i,n'-j} \cdot w^l_{i,j} \cdot \sigma'(x^l_{m',n'}) \tag{27}$$

$$= \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \delta^{l+1}_{m'-i,n'-j} \cdot w^l_{i,j} \tag{28}$$

$$= rot_{180^\circ} \left\{ \sum_{j=0}^{k-1}\sum_{i=0}^{k-1} \delta^{l+1}_{m'+i,n'+j} \cdot w^l_{i,j} \right\} \tag{29}$$

$$= \delta^{l+1}_{m',n'} \star rot_{180^\circ}\{w^l_{m',n'}\} \tag{30}$$

Lors du calcul de $\frac{\partial L}{\partial x^l_{m',n'}}$, nous avons éffectué une convolution. Cela vient du fait que les éléments de la dérivée doivent être positionnés à l'emplacement des pixels ayant contribué aux erreurs correspondantes.

Les dérivées calculées sont ensuite utilisées dans un algorithme de Descente de Gradient[10], afin de mettre à jour le reste des paramètres du réseau.

2.3 Les couches de Pooling

Une couche de Pooling résume les informations d'une image en réduisant les pixels d'une fenêtre à une seule valeur représentative. Cette méthode, déjà présente en 1998 dans l'architecture de LeNet-5, réduit également le coût des calculs en temps et en mémoire. Nous étudierons ici l'une de ses variantes les plus courantes : le Max Pooling.

2.3.1 Propagation Avant

Le Pooling est une opération algorithmique donc nous n'aborderons pas la formulation mathématique.

Deux paramètres influencent le Max Pooling : le *stride* et la taille de la fenêtre (*poolsize*). Le *stride* détermine le nombre de pixels à sauter entre chaque position de la fenêtre.
Cette dernière de taille *poolsize* parcourt les pixels en lignes et en colonnes, avec un pas de valeur *stride*, pour produire une nouvelle image de taille $\lfloor \frac{Taille\ d'entrée - Poolsize}{Stride} + 1 \rfloor$. À chaque déplacement, elle sélectionne la valeur maximale de la fenêtre analysée, constituant un nouveau pixel de la sortie.

On peut visualiser l'opération de la manière suivante :

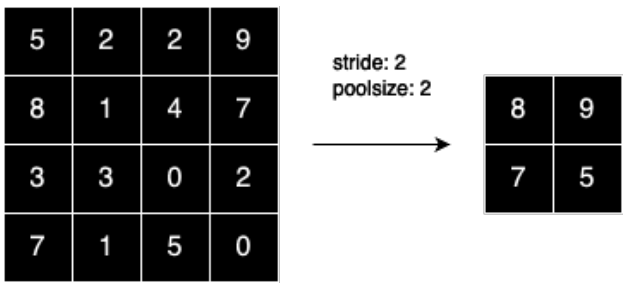

Figure 3: Exemple de la propagation avant d'un Max Pooling

Dans la Figure 3, nous voyons qu'un Max Pooling de *poolsize* 2 et de *stride* 2 a eu pour effet de diviser la taille de l'image par 2.

Sur une image réelle, l'opération donnerait le résultat suivant :

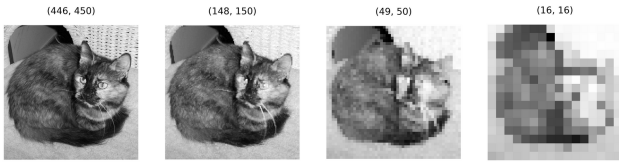

Figure 4: Exemple d'une opération de Max Pooling sur une image[1]

On observe ici une réduction de la taille de l'image tout en préservant ses informations essentielles.

2.3.2 Propagation Arrière

En propagation arrière, on redistribue les erreurs de la couche précédente, aux positions des maxima utilisé en propagation avant. Les autres pixels sont quant à eux désactivés.

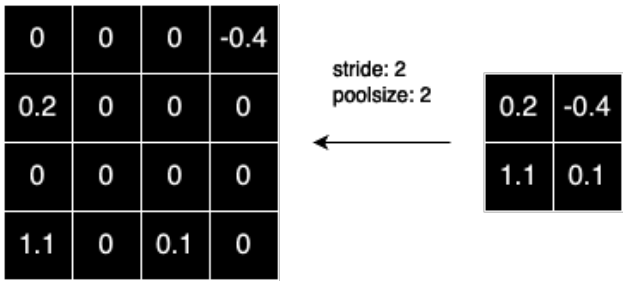

Figure 5: Exemple de la propagation arrière d'un Max Pooling

Implémentation

Dans cette section, nous verrons l'implémentation des couches de Pooling et de Convolution à l'aide de NumPy.

Il existe diverses façons de réaliser un Pooling et une Convolution en Python. La méthode la plus intuitive consiste à parcourir chaque pixel d'une image avec des boucles, enregistrer les informations dans des listes, puis effectuer les calculs élément par élément. Cependant, cette approche s'avère très lente pour des opérations complexes sur plusieurs dimensions.

En effet, Python est un langage interprété, ce qui signifie que chaque ligne de code est traduite en instructions machine à l'exécution. Cela engendre une surcharge importante, surtout lorsque des boucles répétitives traitent un grand volume de données. Ce manque d'efficacité devient rapidement problématique avec des images de grande taille ou des ensembles volumineux.

Voici ci-dessous les résultats de calculs montrant la différence de temps entre une méthode naïve et l'utilisation de NumPy pour effectuer un produit matriciel.

Dims de A	**Dims de B**	**Rapport de temps** t_1/t_2
$(10^2, 10^1)$	$(10^1, 10^2)$	$\frac{0.029}{0.00046} = 63.04$
$(10^3, 10^2)$	$(10^2, 10^3)$	$\frac{10.65}{0.07} = 152.14$
$(10^3, 10^3)$	$(10^3, 10^3)$	$\frac{124.64}{0.6} = 207.73$

Table 1: Comparaison des temps de calcul pour le produit matriciel entre les matrices A et B

$t1$ le temps de calcul avec des boucles en secondes

$t2$ le temps de calcul avec NumPy en secondes

On vois dans Table 1 à la ligne 3, qu'il a fallu 2 minutes à la méthode naïve pour calculer le produit matriciel entre A et B contre 0,6 secondes pour NumPy.

L'utilisation de NumPy est donc à privilégier pour accélérer les calculs tensoriels. En effet, la bibliothèque est implémentée en C, ce qui permet d'effectuer des opérations directement en langage machine. Le papier *"The NumPy array: a structure for efficient numerical computation"*[11] explique bien son fonctionnement et le secret de cette rapidité.

3.1 La méthode Im2Col

Im2Col (image to column) est une méthode de vectorisation introduite en 2006 par trois chercheurs de l'Inria[4]. Cette technique convertit les tenseurs d'images en matrices tout en les décomposant en fenêtres, ce qui simplifie les calculs de convolution et de pooling tout en s'intégrant naturellement avec NumPy.

L'opération se présente de la manière suivante :

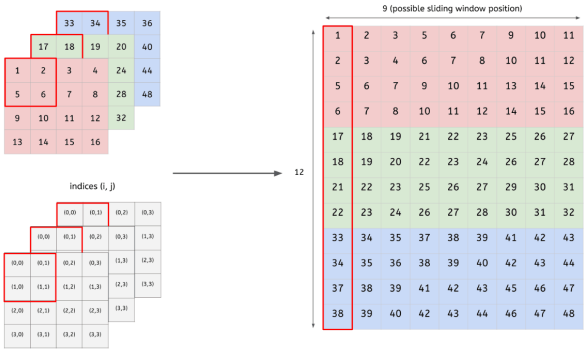

Figure 6: Transformation en matrice d'une image RGB[9]

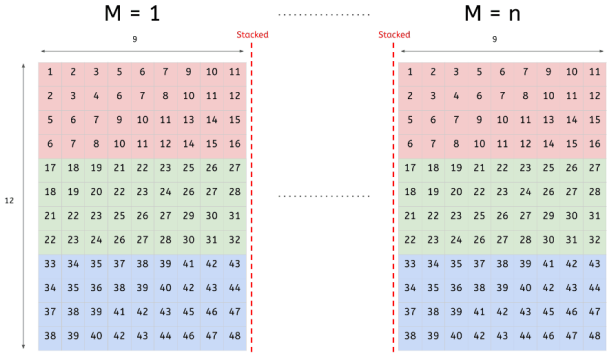

Figure 7: Transformation en matrice de n images RGB[9]

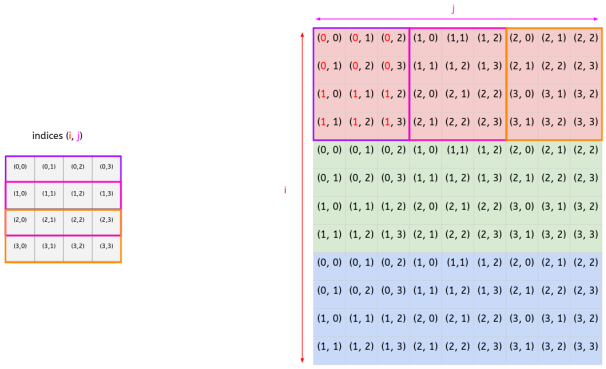

Figure 8: Valeurs de la matrice remplacées par leurs indices dans l'image[9]

Dans cet exemple, nous utilisons un *stride* de 1 et une taille de fenêtre de valeur 2, sur des images de taille (4,4) en RGB. Comme illustré dans les Figures 6, 7 et 8, les canaux d'une image sont concaténés verticalement (de haut en bas), tandis que les fenêtres sont disposées horizontalement (de gauche à droite). De même, chaque image est concaténée horizontalement.

Nous avons, dans la Figure 8, remplacé les valeurs des pixels par leur position dans l'image, révélant ainsi des motifs récurrents. Ces motifs nous aideront à générer les indices i, j et k nécessaires pour construire la matrice à partir des positions des pixels. Examinons ces motifs en détail et modélisons-les mathématiquement.

Ici, le motif de l'indice i au niveau 1 (zone violette dans la Figure 8) est $(0, 0, 1, 1)$. On remarque ensuite que les éléments sont additionnés par 1 au niveau 2 (zone rose dans la Figure 8) puis additionnés une nouvelle fois au niveau 3 (zone orange dans la Figure 8). Nous pouvons donc généraliser avec la formule suivante :

$i_m = \{0_0 + (m-1) \times stride, 0_1 + (m-1) \times stride, ..., 0_{k-1} + (m-1) \times stride, 1_0 + (m-1) \times stride, 1_1 + (m-1) \times stride, ..., 1_{k-1} + (m-1) \times stride, (k-1)_0 + (m-1) \times stride, (k-1)_1 + (m-1) \times stride, ..., (k-1)_{k-1} + (m-1) \times stride\}$, où m correspond au niveau, à partir de $m = 1$ et k la taille de la fenêtre.

Voici le code Python nécessaire pour générer les indices i :

```
1 import numpy as np
2
3 level1 = np.repeat(np.arange(
    window_shape), window_shape)
4 level1 = np.tile(level1, image_shape
    [1])
5
6 increment = stride * np.repeat(np.
    arange(output_height),
    output_width)
7
8 i = level1.reshape(-1, 1) +
    increment.reshape(1, -1)
```

Dans la ligne 3, `np.arrange(window_shape)` génère une séquence d'entiers allant de 0 à `window_shape-1` et `np.repeat(..., window_shape)` répète chaque élément de cette séquence un nombre de fois égal à `window_shape`.

Dans la ligne 4, `np.tile(level1, image_shape[1])` répète tout le tableau `level1` horizontalement un nombre de fois égal au nombre de canaux.

Dans la ligne 6, `np.arange(output_height)` génère une séquence d'entiers allant de 0 à `output_height-1` et `np.repeat(..., output_width)` répète chaque entier de cette séquence un nombre de fois égal à `output_width`. Ensuite, le résultat est multiplié par le *stride*.

Dans la ligne 8, `level1.reshape(-1, 1)` transforme le tableau level1 en un vecteur colonne. De même, `increment.reshape(1, -1)` transforme le tableau `increment` en un vecteur ligne. L'addition entre ces deux matrices suit les règles de broadcasting de NumPy[11], combinant chaque élément de `level1` avec chaque élément de increment pour produire une matrice contenant l'ensemble des indices i.

Concernant l'indice j, les motifs observés sont : $j_{1,2,3} = \{(0, 1, 0, 1), (1, 2, 1, 2), (2, 3, 2, 3)\}$. On voit qu'à chaque déplacement de la fenêtre, les indices j sont additionnés par 1 à partir de j_1. On peut donc généraliser de la manière suivante :

$j_n = \{0_0 + (n-1) \times stride, 1_0 + (n-1) \times stride, ..., (k-1)_0 + (n-1) \times stride, ..., 0_{k-1} + (n-1) \times stride, 1_{k-1} + (n-1) \times stride, ..., (k-1)_{k-1} + (n-1) \times stride\}$
où n correspond au glissement, à partir de $n = 1$ et k la taille de la fenêtre.

Voici le code Python nécessaire pour générer les indices j :

```
1 import numpy as np
2
3 slide1 = np.tile(np.arange(
    window_shape), window_shape *
    image_shape[1])
4 increment = stride * np.tile(np.
    arange(output_width),
    output_height)
5
6 j = slide1.reshape(-1, 1) +
    increment.reshape(1, -1)
```

Dans la ligne 3, `np.arange(window_shape)` génère une séquence d'entiers allant de

0 à `window_shape-1` et `np.repeat(..., window_shape)` répète chaque élément de cette séquence un nombre de fois égal à `window_shape`.

Dans la ligne 4, `np.tile(level1, image_shape[1])` répète tout le tableau level1 horizontalement un nombre de fois égal au nombre de canaux.

Dans la ligne 6, `np.arange(output_height)` génère une séquence d'entiers allant de 0 à `output_height-1` et `np.repeat(..., output_width)` répète chaque entier de cette séquence un nombre de fois égal à `output_width`. Ensuite, le résultat est multiplié par le *stride*.

Dans la ligne 8, `slide1.reshape(-1, 1)` transforme le tableau `slide1` en un vecteur colonne. De même, `increment.reshape(1, -1)` transforme le tableau `increment` en un vecteur ligne. L'addition entre ces deux matrices suit, encore une fois, les règles de broadcasting de NumPy[11], combinant chaque élément de slide1 avec chaque élément de increment pour produire une matrice contenant l'ensemble des indices j.

Concernant l'indice des canaux k, il suffit de répéter les indices des canaux pour chaque pixel dans une fenêtre.

Voici le code Python nécessaire pour générer les indices k :

```
1 import numpy as np
2
3 k = np.repeat(np.arange(image_shape
    [1]), window_shape * window_shape
    ).reshape(-1, 1)
```

Dans cette ligne, `np.arange(image_shape[1])` génère une séquence d'entiers allant de 0 au nombre de canaux de l'image. Ensuite, `np.repeat(..., window_shape * window_shape)` répète chaque entier de cette séquence un nombre de fois égal à `window_shape * window_shape`, c'est-à-dire le nombre de pixels dans une fenêtre. Enfin, `.reshape(-1, 1)` transforme le tableau résultant en un vecteur colonne, contenant l'ensemble des indices k.

Maintenant que nous avons les indices, il reste à générer la matrice correspondant au jeu de données :

```
1 import numpy as np
2
3 k, i, j = get_indices(images.shape,
    window_shape, stride)
4 columns = np.concatenate(images[:, k
    , i, j], axis=-1)
```

Dans ce code, chaque image est vectorisée puis concaténée avec les précédentes. À la fin, nous obtenons des colonnes de matrices correspondant à chacune d'entre elles.

L'opération inverse (Col2Im) est également possible en replaçant chaque pixel à sa position initiale :

```
1 import numpy as np
2
3 images = np.zeros(image_shape)
4 k, i, j = get_indices(image_shape,
    window_shape, stride)
5 cols_reshaped = np.array(np.hsplit(
    columns, image_shape[0]))
6 np.add.at(images, (slice(None), k, i
    , j), cols_reshaped)
```

Dans le cas où le *stride* crée des fenêtres qui se superposent, les pixels sont additionnés aux positions concernées (voir Figure 9). Cela n'aura pas d'effets négatifs dans nos calculs étant donné que cela réspecte leur contribution.

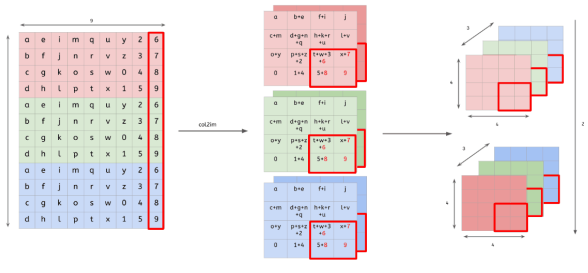

Figure 9: Opération Col2Im[9]

L'ensemble du code de la méthode Im2col se trouve dans le repo de Melpy, à l'adresse suivante : https://github.com/lennymalard/melpy-project/blob/main/melpy/im2col.py

3.2 Les couches de Convolution

Les couches *Convolution2D* héritent de la superclasse *Layer* de Melpy, ce qui leur impose l'utilisation des méthodes **forward()** et **backward()** pour effectuer respectivement les propagations avant et arrière.

Voici, dans un premier temps, le code de la propagation avant :

```
def forward(self):
    self.input_padded = self.explicit_padding()

    self.input_cols = im2col(self.input_padded, self.kernel_size, self.stride)
    self.filter_cols = self.weights.reshape(self.out_channels, -1)

    output_height, output_width = self.get_output_size(self.inputs.shape[2], self.inputs.shape[3])

    self.output_cols = self.filter_cols @ self.input_cols

    self.outputs = np.array(np.hsplit(self.output_cols, self.inputs.shape[0])).reshape(
        (self.input_padded.shape[0], self.out_channels, output_height, output_width)
    )

    if self.biases is not None:
        self.outputs += self.biases

    return self.outputs
```

Les images d'entrée sont d'abord paddées via `self.explicit_padding()` pour ajuster leurs dimensions, puis converties en colonnes avec `im2col()`. Les poids de la couche sont également aplatis en une matrice pour pouvoir effectuer un produit matriciel avec les colonnes d'images. Les dimensions de sortie sont calculées avec `self.get_output_size()`, et la convolution est réalisée en multipliant les colonnes des images par les filtres aplatis. Les résultats sont réassemblés en images, ajustés si des biais sont présents, puis retournés comme résultats finaux de la couche pour être propagés en avant.

Voici maintenant le code de la propagation arrière :

```
def backward(self, dX):
    self.dY = dX

    flipped_filters = self.weights[:, :, ::-1, ::-1]
    flipped_filters_cols = flipped_filters.reshape(self.out_channels, -1)

    self.dY_reshaped = self.dY.reshape(self.dY.shape[0] * self.dY.shape[1], self.dY.shape[2] * self.dY.shape[3])
    self.dY_reshaped = np.array(np.vsplit(self.dY_reshaped, self.inputs.shape[0]))
    self.dY_reshaped = np.concatenate(self.dY_reshaped, axis=-1)

    self.dX_cols = flipped_filters_cols.T @ self.dY_reshaped
    self.dW_cols = self.dY_reshaped @ self.input_cols.T

    self.dX_padded = col2im(self.dX_cols, self.input_padded.shape, self.kernel_size, self.stride)

    if self.padding == "same":
        (pad_top, pad_bottom, pad_left, pad_right) = self.calculate_padding()
        self.dX = self.dX_padded[:, :, pad_top:-pad_bottom, pad_left:-pad_right]
    else:
```

```
        self.dX = self.dX_padded

    self.dW = self.dW_cols.reshape((
    self.dW_cols.shape[0], self.
    in_channels, self.kernel_size,
    self.kernel_size))

    if self.biases is not None:
        self.dB = np.sum(self.dY,
    axis=(0, 2, 3), keepdims=True)

    return self.dX
```

Les gradients de sortie `dX` sont assignés à `self.dY`, et les filtres sont retournés de 180° puis aplatis en colonnes pour les calculs. Les gradients `self.dY` sont convertis en colonnes, et les gradients des entrées `self.dX` sont calculés via un produit matriciel entre les colonnes des filtres retournés et dY. Les gradients des poids `self.dW` sont quant à eux obtenus par multiplication entre dY et les colonnes d'entrée transposées. Les gradients des biais `self.dB`, si présents, sont calculés par une somme. Les gradients d'entrée sont reconstitués avec `col2im()` et ajustés pour retirer le padding si nécessaire. Finalement, les gradients `self.dX` sont retournés pour poursuivre la propagation arrière.

L'ensemble du code lié aux couches de Convolution se trouvent dans le repo de Melpy, à l'adresse suivante : https://github.com/lennymalard/melpy-project/blob/main/melpy/layers.py

3.3 Les couches de Pooling

Les couches *Pooling2D* héritent également de la superclasse *Layer* de Melpy, leur imposant les mêmes contraintes que *Convolution2D*.

Voici donc le code de la propagation avant :

```
def forward(self):
    output_height = int((self.inputs
    .shape[2] - self.pool_size + self
    .stride) // self.stride)
    output_width = int((self.inputs.
    shape[3] - self.pool_size + self.
    stride) // self.stride)

    output_shape = (self.inputs.
    shape[0], self.inputs.shape[1],
    output_height, output_width)

    self.input_cols = im2col(self.
    inputs, self.pool_size, self.
    stride)
    self.input_cols_reshaped = np.
    array(np.hsplit(np.array(np.
    hsplit(self.input_cols, self.
    inputs.shape[0])), self.inputs.
    shape[1]))

    self.maxima = np.max(self.
    input_cols_reshaped, axis=2)
    self.maxima_reshaped = self.
    maxima.reshape(self.inputs.shape
    [1], -1)

    self.outputs = col2im(self.
    maxima_reshaped, output_shape, 1,
    1)

    return self.outputs
```

Les dimensions de sortie sont d'abord calculées, en fonction des dimensions d'entrée, de la taille de la fenêtre de pooling, et du *stride*. Les entrées sont ensuite transformées en colonnes à l'aide de `im2col()`. Les colonnes sont restructurées pour correspondre aux canaux et échantillons, puis l'opération `np.max()` extrait les maxima de chaque fenêtre. Ces valeurs maximales sont réorganisées et reconstruites en utilisant `col2im()` pour former les sorties finales. Ces sorties sont finalement retournées pour poursuivre la propagation avant.

Voici maintenant le code de la propagation arrière :

```
def backward(self, dX):
    self.dY = dX
    self.dX = np.zeros_like(self.
    inputs)

    self.dY_cols = im2col(self.dY,
```

```
    1, 1)
    self.dY_cols_reshaped = np.array
    (np.hsplit(np.array(np.hsplit(
    self.dY_cols, self.dY.shape[0])),
     self.dY.shape[1])).transpose(0,
    1, 3, 2)

    self.input_cols = im2col(self.
    inputs, self.pool_size, self.
    stride)
    self.input_cols_reshaped = np.
    array(np.hsplit(np.array(np.
    hsplit(self.input_cols, self.
    inputs.shape[0])), self.inputs.
    shape[1])).transpose(0, 1, 3, 2)

    self.output_cols = im2col(self.
    outputs, 1, 1)
    self.output_cols_reshaped = np.
    array(np.hsplit(np.array(np.
    hsplit(self.output_cols, self.
    inputs.shape[0])), self.inputs.
    shape[1])).transpose(0, 1, 3, 2)

    self.mask = np.array(self.
    input_cols_reshaped == self.
    output_cols_reshaped, dtype=np.
    uint64)

    self.dX_cols = np.concatenate(np
    .concatenate(np.array(self.mask *
     self.dY_cols_reshaped).transpose
    (0, 1, 3, 2), axis=1), axis=1)
    self.dX = col2im(self.dX_cols,
    self.inputs.shape, self.pool_size
    , self.stride)

    return self.dX
```

Les gradients reçus `dX` sont sauvegardés dans `self.dY`, puis transformés en colonnes avec `im2col()`. Les entrées originales et les sorties de la propagation avant sont également converties en colonnes et restructurées. Un masque binaire identifiant les maxima des fenêtres de pooling est créé pour ne transmettre que les gradients associés à ces maxima. Les gradients modifiés sont réassemblés en une forme plate, puis reconstruits avec `col2im()` pour correspondre aux dimensions des entrées. Enfin, les gradients des entrées reconstitués `self.dX` sont retournés pour continuer la propagation arrière.

L'ensemble du code lié aux couches de Pooling se trouvent dans le repo de Melpy, à l'adresse suivante : https://github.com/lennymalard/melpy-project/blob/main/melpy/layers.py

Expérimentations

Dans cette section, nous verrons les expérimentations éffectuées et discuterons de leurs résulats.

Pour évaluer la justesse et l'efficacité de l'implémentation, il a fallu entraîner des architectures, ajuster leurs hyperparamètres, puis comparer les résultats obtenus avec ceux de Keras sur les mêmes architectures.

Le jeu de données MNIST a été choisi pour le premier test. Composé de 60 000 images de chiffres manuscrits (de 0 à 9), il est l'un des jeux de données les plus connus pour les tâches de classification d'images. Chaque image mesure 28x28 pixels et est en niveaux de gris. Ce jeu de données est particulièrement populaire car il est simple à traiter et offre une grande qualité et quantité d'exemples, ce qui en fait une référence pour les premières expérimentations en apprentissage profond.

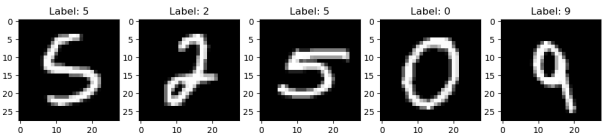

Figure 10: Exemples d'images provenant du jeu de données MNIST

Cependant, afin d'augmenter la complexité des expérimentations, le jeu de données CIFAR-10 a également été utilisé. Il contient 60 000 images réparties en 10 classes (avions, voitures, oiseaux, chats, cerfs, chiens, grenouilles, chevaux, bateaux et camions). Les images sont des photographies sous échantillonnées, qui mesurent 32x32 pixels et qui sont en couleur (format RGB), ce qui les rend plus complexes à traiter que celles de MNIST. La diversité des classes et le manque de corrélation entre elles ajoutent également de la complexité à ce jeu de données. En ce qui concerne la qualité de ces dernières, elle est comparable à celle de MNIST.

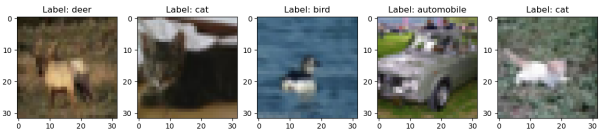

Figure 11: Exemples d'images provenant du jeu de données CIFAR-10

4.1 Méthode

Le protocole expérimental est le suivant :

Pré-traitement des données Les données sont pré-traitées pour être compatibles avec les modèles.

Création et évaluation d'architectures avec Melpy Des architectures sont créées et entraînées avec différents hyperparamètres pour identifier celle qui offre la meilleure généralisation.

Création et entraînement des architectures avec Keras L'architecture sélectionnée est ensuite entraînée avec Keras de manière similaire.

Comparaison des métriques finales On compare les métriques finales d'une bibliothèque avec celles de l'autres.

L'entraînement a été effectué sur un processeur Intel i5-12600K avec 32 Go de RAM. Il est important de noter que les temps de calcul ont pu être influencés par des processus externes indésirables. Concernant la métrique utilisée, seul le coût de l'architecture au fil des époques a été présenté, car il reflète parfaitement les performances d'un modèle.

4.2 Résultats et discussion

4.2.1 Pré-traitement des données

Afin d'éviter des problèmes tels que *Dead ReLU*, l'explosion des gradients ou encore la disparition des gradients, il est nécessaire de mettre les images à l'échelle.

Pour ce faire, nous commençons par normaliser les données, en les mettant sur une échelle allant de 0 à 1, puis nous les standardisons.

Enfin, les étiquettes doivent être encodées. Pour cela, nous utilisons l'encodeur One-Hot.

4.2.2 Sélection du modèle

MNIST

L'architecture sélectionnée pour MNIST est la suivante :

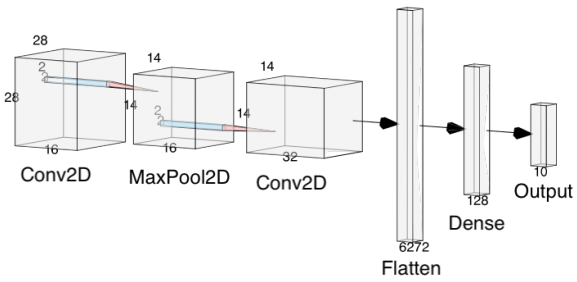

Figure 12: Réseau de neurones sélectionné pour MNIST

Comme illustré dans la Figure 10, le réseau de neurones commence par une couche de convolution qui génère 16 cartes de caractéristiques, suivie d'une couche de pooling qui réduit leur taille de moitié. Ensuite, une seconde couche de convolution produit 32 cartes de caractéristiques, ensuite aplaties pour alimenter un MLP ayant une couche cachée composée de 128 neurones.

La couche cachée utilise la fonction d'activation Leaky ReLU, tandis que la sortie applique une fonction Softmax. Pour optimiser l'architecture, l'algorithme Adaptive Momentum (Adam) a été utilisé pour minimiser la fonction de coût d'entropie croisée catégorielle (Categorical Cross-Entropy).

Le choix a été fait sur les 5 entrainements suivants :

	P	Δ	e	t/e	Lot	γ	t
1	15 840	$1C_{w,b} + 2D_{w,b} = 3$	15	32sec	256	1^{-4}	08min 01sec
2	201 226	$1C_{w,b} + 2D_{w,b} = 3$	15	02min 23sec	256	1^{-4}	35min 58sec
3	806 394	$2C_{w,b} + 2D_{w,b} = 4$	15	01min 53sec	256	1^{-4}	28min 22sec
4	1 624 842	$2C_{w,b} + 4D_{w,b} = 6$	10	04min 42sec	256	1^{-4}	47min 03sec
5	1 615 466	$2C_{w,b} + 2D_{w,b} = 4$	10	04min 40sec	256	1^{-4}	46min 49sec

Table 2: Entrainements éffectués sur MNIST avec Melpy

P étant le nombre de paramètres dans l'architecture

Δ étant la prodondeur de l'architecture avec C une couche de convolution, D une couche entièrement connectée, w l'utilisation de poids et b l'utilisation de biais

e étant le nombre d'époques passés pour entrainer l'architecture

Lot étant le nombre d'entrées entrainées sur une passe avant et arrière

γ étant le taux d'apprentissage utilisé dans l'algorithme d'optimisation

t étant le temps qu'il a fallu pour entrainer l'architecture

t/e étant le temps passé en moyenne sur une époque

La taille des lots et le taux d'apprentissage restent ici inchangés, car ils se sont révélés significativement plus stables que d'autres configurations testées, n'apportant pas d'informations pertinentes sur les entrainements.

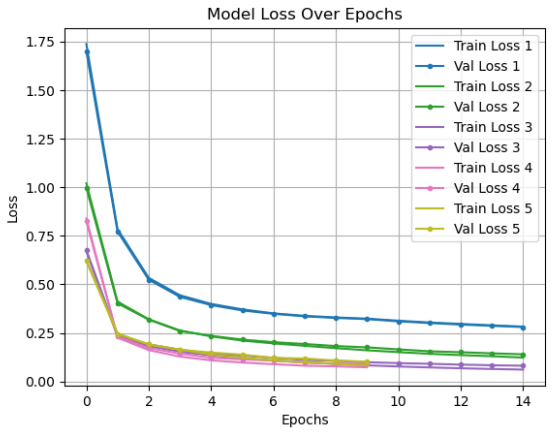

Figure 13: Le coût de l'architecture au fil des époques

Comme l'illustre la Figure 13, les modèles 3, 4 et 5 présentent des performances similaires. Cependant, le modèle 3 est sélectionné en raison de sa rapidité d'exécution sur une époque (voir Table 2).

En dehors de la sélection d'un modèle à comparer avec Keras, nous pouvons relever les informations suivantes concernant les différents entrainements :

- **Nombre de paramètres** : Le nombre de paramètres influence significativement le temps nécessaire pour réaliser une époque.
- **Complexité du modèle** : Les modèles plus complexes tendent à mieux détecter les motifs. Cependant, cette complexité devient moins efficace lorsque le nombre de neurones ou la profondeur de l'architecture devient excessif.

CIFAR-10

L'architecture sélectionnée pour CIFAR-10 est la suivante :

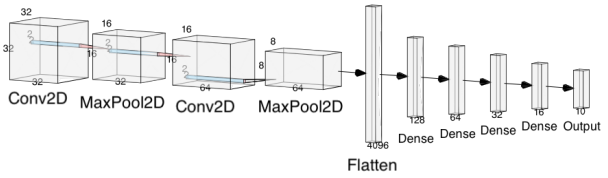

Figure 14: Réseau de neurones sélectionné pour CIFAR-10

Comme illustré dans la Figure 14, le réseau de neurones débute par une couche de convolution générant 32 cartes de caractéristiques, suivie d'une couche de pooling réduisant leur taille de moitié. Cette opération est répétée une seconde fois, avec comme seule différence la couche de convolution produisant cette fois 64 cartes de caractéristiques. Celles-ci sont ensuite aplaties pour alimenter un MLP constitué de 4 couches cachées contenant respectivement 128, 64, 32 et 16 neurones chacune.

Les couches cachées utilisent la fonction d'activation Leaky ReLU, tandis que la sortie applique une fonction Softmax. Pour optimiser l'architecture, l'algorithme Adam a été utilisé pour minimiser la fonction de coût d'entropie croisée catégorielle.

Le choix a été fait sur les 6 entrainements suivants :

	P	Δ	e	t/e	Lot	γ	t
1	1 056 376	$4C_w + 2D_{w,b} = 6$	20	09min 50sec	128	5^{-5}	03h 16min 59sec
2	2 107 146	$2C_w + 2D_{w,b} = 4$	20	17min 24sec	128	1^{-5}	05h 48min 14sec
3	2 107 146	$2C_w + 2D_{w,b} = 4$	35	15min 38sec	128	5^{-5}	09h 07min 21sec
4	2 107 146	$2C_w + 2D_{w,b} = 4$	50	08min 12sec	256	5^{-5}	06h 50min 36sec
5	2 117 866	$2C_w + 2D_{w,b} = 4$	100	10min 27sec	128	5^{-5}	17h 26min 00sec
6	8 398 698	$2C_{w,b} + 2D_{w,b} = 4$	50	08min 34sec	64	5^{-6}	07h 08min 29sec
7	544 122	$2C_{w,b} + 5D_{w,b} = 7$	50	05min 53sec	64	5^{-5}	04h 54min 52sec
8	8 408 442	$2C_{w,b} + 5D_{w,b} = 7$	50	10min 26sec	64	5^{-5}	08h 42min 17sec

Table 3: Entrainements éffectués sur CIFAR-10 avec Melpy

P étant le nombre de paramètres dans l'architecture

Δ étant la prodondeur de l'architecture avec C une couche de convolution, D une couche entièrement connectée, w l'utilisation de poids et b l'utilisation de biais

e étant le nombre d'époques passés pour entrainer l'architecture

Lot étant le nombre d'entrées entrainées sur une passe avant et arrière

γ étant le taux d'apprentissage utilisé dans l'algorithme d'optimisation

t étant le temps qu'il a fallu pour entrainer l'architecture

t/e étant le temps passé en moyenne sur une époque

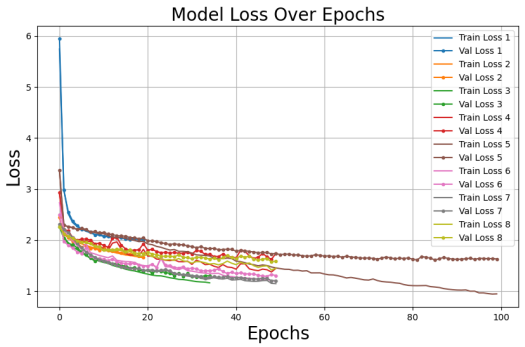

Figure 15: Le coût de l'architecture au fil des époques

Comme le montre la Figure 15, les modèles 3, 7 et 8 se démarquent comme les plus performants. Parmi eux, c'est bien le modèle 7 qui a été retenu. Ce choix repose sur un compromis entre la qualité de sa généralisation, son temps d'entraînement réduit et son efficacité dès le début de ce dernier.

À l'instar des entrainements sur MNIST, nous pouvons relever les informations suivantes à partir de l'ensemble des entrainements sélectionnés pour CIFAR-10 :

- **Taille des lots** : Elle influence de manière significative le temps de calcul par époque ainsi que le nombre d'itérations nécessaires pour atteindre la convergence.
- **Capacité de généralisation** : La taille des lots influence également la capacité du modèle à bien généraliser sur des données non vues, à condition que le modèle ne soit pas trop complexe.
- **Couches de pooling** : L'intégration de couches de pooling à des positions

intéressantes dans l'architecture peut compenser le temps de calcul, bien que cela se fasse au détriment de la quantité d'informations utilisé pour l'apprentissage.

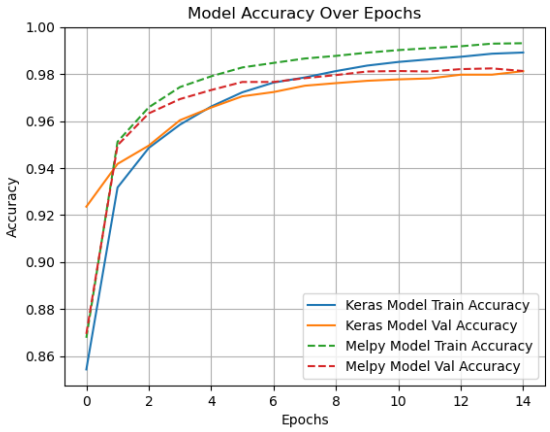

Figure 17: Comparaison de la précision entre Melpy et Keras sur MNIST

4.2.3 Comparaison des résultats

Afin d'évaluer les performances de Melpy dans l'utilisation des CNNs, nous comparons cette bibliothèque avec Keras, une autre bibliothèque de Deep Learning, reconnue pour la simplicité de son utilisation.

Pour ce faire, ce sont les architectures sélectionnées plus tôt qui serviront de référence.

Observons les différences entre les métriques calculées durant l'entraînement pour chaque jeu de données :

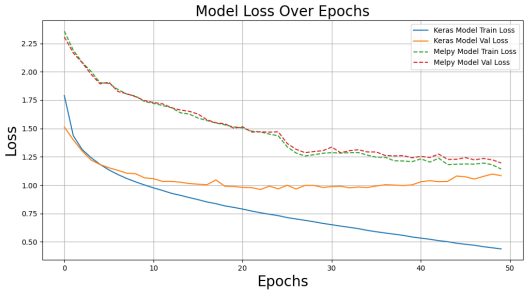

Figure 18: Comparaison du coût entre Melpy et Keras sur CIFAR-10

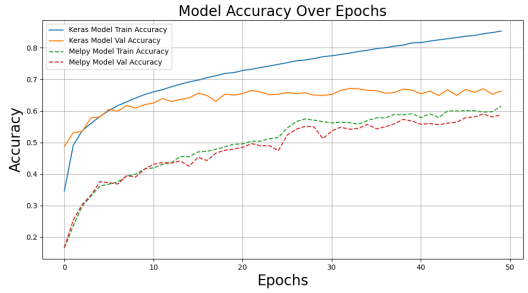

Figure 19: Comparaison de la précision entre Melpy et Keras sur CIFAR-10

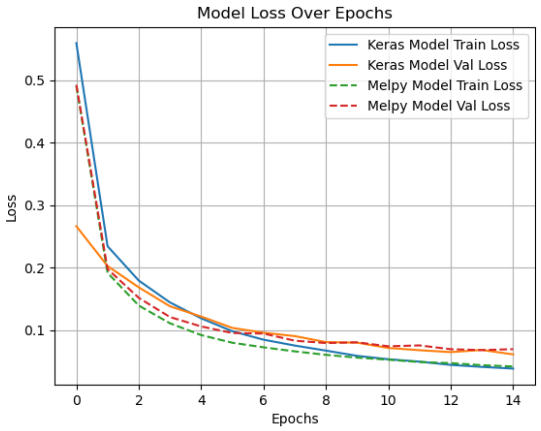

Figure 16: Comparaison du coût entre Melpy et Keras sur MNIST

Comme le montrent les Figures allant de 16 à 19, les résultats varient considérablement d'un jeu de données à un autre. Cette différence peut s'expliquer par une légère divergence dans le calcul du gradient au niveau des couches de convolution. En effet, des

tests comparant les résultats des opérations mathématiques implémentées dans Melpy à celles de TensorFlow, la bibliothèque utilisée par Keras pour ses calculs, ont révélé une différence moyenne allant de 0,4 à 0,6 entre les gradients pour chaque valeur d'entrée.
Cette différence est significative en raison du grand nombre de valeurs impliquées. Par exemple, pour un gradient de taille (1, 3, 32, 32), la somme des erreurs peut atteindre 1800, alors que les autres opérations, testées de manière équivalente, présentent des écarts proches de zéro. Bien que cette divergence justifie une analyse plus approfondie, elle n'empêche pas un apprentissage globalement correct.
En effet, sur le jeu de données MNIST, l'architecture entraînée avec Keras a obtenu des performances équivalentes à celles de Melpy, atteignant une précision proche de 98% sur le jeu de données de validation. En revanche, sur le jeu de données CIFAR-10, bien que l'apprentissage ait été beaucoup plus lent avec Melpy, l'architecture entraînée avec cette dernière a démontré une meilleure capacité de généralisation comparée à celle entraînée avec Keras. Les deux architectures ont convergé vers une précision d'environ 65% sur le jeu de données de validation.

On peut également relevé le temps d'execution qu'il a fallu à Keras sur les mêmes configurations que Melpy :

	Melpy	Keras
MNIST	28min 22sec	01min 15sec
CIFAR-10	04h 54min 52sec	10min 43sec

Table 4: Durées des entrainements éffectués sur CIFAR-10 et MNIST

La différence importante dans les temps d'exécution (voir Table 4) s'explique par les optimisations intégrées à Keras. La bibliothèque compile les modèles en code machine via TensorFlow, ce qui accélère l'exécution des opérations. De plus, Keras exploite plus efficacement le parallélisme sur les CPU, ce qui améliore de surcroit les performances des calculs.
En revanche, Melpy, dans son état actuel, n'intègre pas encore de telles optimisations. Ses opérations restent dépendantes de l'interprétation directe de Python, ce qui allonge les temps d'exécution.

Conclusion

En conclusion, ce projet peut être considéré comme un succès, ayant atteint son objectif initial : implémenter, à partir de zéro et en s'appuyant de NumPy, des outils permettant la création et l'entraînement de réseaux de neurones convolutifs. Cette réalisation a également rempli son rôle pédagogique en m'amenant à apprendre en profondeur les mécanismes fondamentaux des CNNs.

Cependant, des différences ont été observées dans les métriques calculées au cours de l'apprentissage, entre l'implémentation et Keras. Ces écarts nécessiteront un travail plus poussé afin de les réduire. Toutefois, le principal axe d'amélioration reste l'optimisation des calculs. À cet effet, l'intégration de bibliothèques telles que Numba, JAX, ou Cython pourrait permettre de compiler les modèles lors de l'entraînement, améliorant ainsi la rapidité des calculs.

Références

[1] *digitalocean.com*. Image d'exemple de Max Pooling sur une image.

[2] Valentina Emilia Balas and Nikos E Mastorakis. Multilayer perceptron and neural networks. 2009.

[3] Léon Bottou and Chhavi Yadav. Cold case: the lost mnist digits. 2019.

[4] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 2006.

[5] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. 2009.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. 2012.

[7] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradientbased learning applied to document recognition. 1998.

[8] Sinisa Mihajlovic, Dragan Vojo Ivetic, and Ivana Berković. Applications of convolutional neural networks. 2020.

[9] Ferdinand Mom. *hackmd.io*. Images d'illustrations pour la méthode Im2Col.

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2017.

[11] Stéfan van der Walt, Gael Varoquaux, and S. Chris Colbert. The numpy array: a structure for efficient numerical computation. 2011.