TP8 : Deep Learning et Réseau de Neurones Convolutionnel

Axel Bröns
axel.brons@edu.ece.fr

Valentin Kocijancic
valentin.kocijancic@edu.ece.fr

Lyon, le 30 avril 2025

Nous attestons que ce travail est original, qu'il est le fruit d'un travail commun au binôme et qu'il a été rédigé de manière autonome.

Motivation

Le but de ce TP est d'implémenter une des variantes de réseau de neurones très utile pour le traitement des images. Cette variante fonctionne grâce à des convolution c'est ainsi qu'on l'appelle CNN (Convolutional Neural Network). Ce TP va nous apprendre à l'implémenter, l'utiliser et l'analyser, mais on va se restreindre sur une carte Arduino DUE qui a une faible puissance par rapport aux ordinateurs.

1 Résolution sur NN Classique

T1. Nombre de datas pour entraînement

Avoir un très grand jeu de données pour un réseau de neurones a des avantages et des inconvénients, en effet, plus on augmente le nombre de datas, plus notre réseau de neurones pourra s'entrainer sur le plus de valeurs possibles et ainsi être plus performant. Par ailleurs, le temps de calcule augmente aussi parce que le réseau de neurones doit effectuer plus de calculs.

T2. Données d'entraînement & données de tests

On sépare en deux dataset les données de l'entraînement et les données de tests parce que si nous effectuons des prédictions sur des données qui nous ont servis d'entrainement, les résultats vont être biaisés. En effet, cela semble plutôt logique que nous voulons évaluer notre modèle sur des vraies valeurs extérieur à notre dataset, alors le plus simple est de simuler ces vrais valeurs en ne donnant qu'une partie de toutes nos données au réseau de neurones et après nous évaluerons notre réseau de neurones sur le reste de nos données non entraîner.

E1. Taille de notre réseau de neurones

Etant donné que chaque image fait 28×28, on a une image de 784 pixels donc il y a 784 neurones sur la première couche. Entre la couche 1 et 2 il y a au total, $784 \times 25 = 19600$ poids. Concernant la couche 2 et 3, il y a $25 \times 7 = 175$ poids. Et pour finir pour la couche 3 et la couche 3 et la couche de sortie il y a $7 \times 1 = 7$ poids. Donc, quand on additionne tous les poids nous nous retrouvons avec : $19600 + 175 + 7 = 19782$ poids.

E2.

On rappelle que la mémoire SRAM de l'Arduino DUE est de 96 Ko. La fonction `flatten2vector` sert à convertir un tableau 2D en un tableau 1D. Dans notre cas, nous avons des images $28 \times 28 = 784$ où chaque valeurs est un `float` (4 octets) donc au total nous avons $784 \times 4 = 3136$ octets. Nous rappelons de même que nous avons 50 images, donc si nous stockons toutes les images en une dimensions nous aurons $50 \times 3136 = 156800$ octets, ce qui dépasse la mémoire SRAM que propose l'Arduino DUE. C'est pour cette raison que nous devons la mettre dans la boucles pour ne traiter qu'une image à la fois et éviter la surcharge de la mémoire SRAM.

E3. Temps de convergence

Avec un seuil MSE de 0.03 nous mettons 165 399 ms pour converger ce qui est très lent mais cohérent avec l'ampleur de nos données. Nous pouvons calculer l'erreur moyenne sur les données prédites de test et nous avons une moyenne de $MSE_{moy} = 0.1468894$. On remarque le modèle peut se tromper et n'est pas parfait.

2 Conception du CNN

T3. Explications

L'implémentation de convolution (dans notre cas 2 fois) est le fait de faire ressortir certains traits de nos images, et extraire les caractéristiques importantes de celle-ci pour que le réseau de neurones soit le plus optimisé possible. L'importance du *padding* en implémentant les convolutions est de garder la même taille de matrice en sortie. Le maxpooling, lui, sert à réduire la taille de la matrice (l'image) sans perdre les informations importantes et de réduire au maximum le bruit de notre image. Voici l'algorigramme du code correspondant :

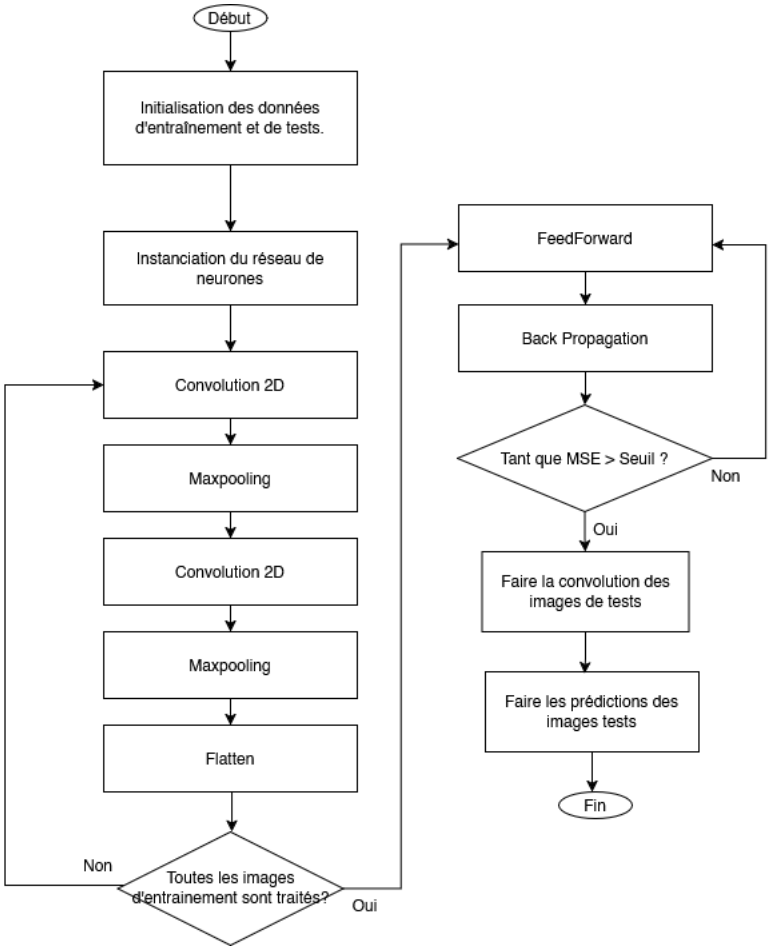

Figure 1: Algorigramme du réseau de neurones avec convolution

E4. Inversion du kernel

Sachant que notre kernel est celui-ci :

$$K = \begin{pmatrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & 1 & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{pmatrix}$$

On remarque que le kernel est symétrique, ainsi pas besoin de le retourner.

E5. Choix du kernel

D'après le cours, le choix du kernel est choisi pour accentuer les contours des images et ainsi être plus visible pour le réseau de neurones.

3 Analyse des Résultats

E6. Analyse & précision

Voici l'output qu'on obtient avec les MSE calculé pour chaque données tests :

```
=-[OUTPUTS]-=
1 | Estimate : 0.3200690 MSE : 0.1024441
1 | Estimate : 0.2432173 MSE : 0.0591547
1 | Estimate : 0.8881119 MSE : 0.7887427
1 | Estimate : 0.5251659 MSE : 0.2757992
1 | Estimate : 0.1651099 MSE : 0.0272613
1 | Estimate : 0.2097154 MSE : 0.0439805
1 | Estimate : 0.9167983 MSE : 0.8405191
1 | Estimate : 0.2213512 MSE : 0.0489964
1 | Estimate : 0.2475452 MSE : 0.0612786
1 | Estimate : 0.1859252 MSE : 0.0345682
1 | Estimate : 0.2224244 MSE : 0.0494726
1 | Estimate : 0.2223763 MSE : 0.0494512
1 | Estimate : 0.1436676 MSE : 0.0206404
1 | Estimate : 0.1796506 MSE : 0.0322744
1 | Estimate : 0.2087749 MSE : 0.0435870
1 | Estimate : 0.9512050 MSE : 0.9047909
8 | Estimate : 0.9527730 MSE : 0.0022304
8 | Estimate : 0.9535794 MSE : 0.0021549
8 | Estimate : 0.9616414 MSE : 0.0014714
8 | Estimate : 0.9612359 MSE : 0.0015027
8 | Estimate : 0.9532008 MSE : 0.0021902
8 | Estimate : 0.9595895 MSE : 0.0016330
8 | Estimate : 0.9021899 MSE : 0.0095668
8 | Estimate : 0.9381650 MSE : 0.0038236
8 | Estimate : 0.9602408 MSE : 0.0015808
8 | Estimate : 0.9428021 MSE : 0.0032716
8 | Estimate : 0.9490938 MSE : 0.0025914
8 | Estimate : 0.9590751 MSE : 0.0016748
8 | Estimate : 0.9473379 MSE : 0.0027733
8 | Estimate : 0.5234108 MSE : 0.2271373
```

Seuil de MSE à 0.03

```
=-[OUTPUTS]-=
1 | Estimate : -0.0356016 MSE : 0.0012675
1 | Estimate : -0.0374651 MSE : 0.0014036
1 | Estimate : 0.7861811 MSE : 0.6180807
1 | Estimate : 0.0106936 MSE : 0.0001144
1 | Estimate : -0.0332324 MSE : 0.0011044
1 | Estimate : -0.0299497 MSE : 0.0008970
1 | Estimate : 0.9491572 MSE : 0.9008995
1 | Estimate : -0.0235190 MSE : 0.0005531
1 | Estimate : -0.0386573 MSE : 0.0014944
1 | Estimate : -0.0330369 MSE : 0.0010914
1 | Estimate : -0.0227859 MSE : 0.0005192
1 | Estimate : -0.0319143 MSE : 0.0010185
1 | Estimate : -0.0366966 MSE : 0.0013466
1 | Estimate : -0.0336689 MSE : 0.0011336
1 | Estimate : -0.0332406 MSE : 0.0011049
1 | Estimate : 0.9291692 MSE : 0.8633553
8 | Estimate : 0.4165776 MSE : 0.3403817
8 | Estimate : 0.9672745 MSE : 0.0010710
8 | Estimate : 0.9583261 MSE : 0.0017367
8 | Estimate : 0.9724156 MSE : 0.0007609
8 | Estimate : 0.0642721 MSE : 0.8755868
8 | Estimate : 0.9719801 MSE : 0.0007851
8 | Estimate : 0.8111776 MSE : 0.0356539
8 | Estimate : 0.9636638 MSE : 0.0013203
8 | Estimate : 0.9721338 MSE : 0.0007765
8 | Estimate : 0.9680957 MSE : 0.0010179
8 | Estimate : 0.9700589 MSE : 0.0008965
8 | Estimate : 0.9719730 MSE : 0.0007855
8 | Estimate : 0.9617530 MSE : 0.0014628
8 | Estimate : 0.2365521 MSE : 0.5828527
```

Seuil de MSE à 0.001

```
=-[OUTPUTS]-=
1 | Estimate : -0.0075517 MSE : 0.0000570
1 | Estimate : -0.0077760 MSE : 0.0000605
1 | Estimate : 0.4293289 MSE : 0.1843233
1 | Estimate : 0.0005355 MSE : 0.0000003
1 | Estimate : -0.0070640 MSE : 0.0000499
1 | Estimate : -0.0069788 MSE : 0.0000487
1 | Estimate : 0.9928402 MSE : 0.9857316
1 | Estimate : -0.0057471 MSE : 0.0000330
1 | Estimate : -0.0081626 MSE : 0.0000666
1 | Estimate : -0.0071662 MSE : 0.0000514
1 | Estimate : -0.0055948 MSE : 0.0000313
1 | Estimate : -0.0067761 MSE : 0.0000459
1 | Estimate : -0.0070769 MSE : 0.0000501
1 | Estimate : -0.0072593 MSE : 0.0000527
1 | Estimate : -0.0074619 MSE : 0.0000557
1 | Estimate : 0.9845943 MSE : 0.9694259
8 | Estimate : 0.2804321 MSE : 0.5177780
8 | Estimate : 0.9951973 MSE : 0.0000231
8 | Estimate : 0.9877431 MSE : 0.0001502
8 | Estimate : 0.9961176 MSE : 0.0000151
8 | Estimate : 0.0206385 MSE : 0.9591490
8 | Estimate : 0.9959991 MSE : 0.0000160
8 | Estimate : 0.8148856 MSE : 0.0342673
8 | Estimate : 0.9941165 MSE : 0.0000346
8 | Estimate : 0.9960884 MSE : 0.0000153
8 | Estimate : 0.9952826 MSE : 0.0000223
8 | Estimate : 0.9958097 MSE : 0.0000176
8 | Estimate : 0.9960670 MSE : 0.0000155
8 | Estimate : 0.9942601 MSE : 0.0000329
8 | Estimate : 0.0760929 MSE : 0.8536043
```

Seuil de MSE à 0.00002

Figure 2: Différent output suivant la valeur du seuil MSE

Concernant les temps de convergence nous pouvons apercevoir que plus on diminue le seuil MSE, logiquement plus notre temps de convergence mettra longtemps. Nous avons aussi calculé les MSE Totaux et nous pouvons les comparer dans un tableau :

Seuil	Temps (ms)	MSE Totale
0.03	1 610	≈ 0.058
0.001	13 909	≈ 0.015
0.00002	45 568	≈ 0.076

Si nous comparons avec le réseau de neurones sans convolution, on remarque bien que l'implémentation des convolution a améliorer la précisions mais aussi la rapidité du réseau de neurones. On rappelle que pour un simple seuil de MSE à 0.03 le réseau de neurones sans convolution a mis 165 secondes alors qu'avec convolution la convergence a pris un peu plus de 1 seconde, ce qui est 100 fois plus rapide. Par ailleurs, pour les MSE totale on voit que les prédictions avec le seuil à 0.00002 obtient une MSE Total élevé, ce qui pourrait être la cause d'un sur-entraînement. On peut conclure que le seuil à 0.001 serait le plus optimale.

E7. Ajout d'images

Nous avons 96 Ko de mémoire SRAM pour l'Arduino Due, nous laissons volontairement 15 Ko pour les variables, le buffer etc... le reste nous le réservons pour stocker les images à entrainer. Sachant que chaque image MNIST fait $28 \times 28 = 784$ et $784 \times 4 \approx 3136$ octets. Or de base nous avons 80 images en tout (entrainement + test), ainsi nous avons besoin de $3136 \times 80 \approx 250$ Ko. Nous dépassons déjà la mémoire SRAM. Pour améliorer ou contourner le problème nous passons par la mémoire flash qui est de 512 Ko grâce à l'implémentation de PROGMEM :

```
const float inputs[50][DATA_SIZE][DATA_SIZE] PROGMEM = {...};
```

Ainsi grâce à cette méthode nous pourrons rajouter à peu près 70 images dans notre réseau de neurones ($70 \times 3136 \approx 220$ Ko, donc $220 + 250 = 470$ Ko ce qui nous laisse de la mémoire en plus). L'avantage de cette telle solution sera l'augmentation de précision par rapport à nos résultats obtenus précédemment parce qu'on entraîne plus notre réseaux. L'inconvénient serait sûrement le temps d'exécution de notre programme.

E8. Tests avec différentes couches

Nous mettons désormais les couches {49, 4, 1} et nous prenons un seuil MSE à 0.001 et nous faisons le test. Nous remarquons que notre temps de convergence est bien inférieur par rapport aux couche {49, 3, 3, 1}. Mais nous devons faire attention à la précisions de ce réseau de neurones, il se peut que nous avons sous-entraîné notre modèle et qu'il ne sera jamais performant parce qu'il manque d'entraînement.

De même pour les couches {49, 30, 20, 1} et nous ne changeons pas le seuil MSE et nous n'arrivons jamais à converger, le réseau de neurones reste bloqué dans la boucle d'apprentissage avec une MSE de 0.499997. Ce phénomène peut-être plusieurs chose comme par exemple le "*vanishing gradient*" qui est causé lorsque gradients deviennent très petit pendant l'apprentissage et ainsi les poids ne sont plus mis à jour efficacement. Le deuxième phénomène possible est le sur-entraînement, cela apparait lorsque notre réseau de neurones est trop complexe par rapport aux données qu'on a. Du coup il finit par apprendre le bruit ce qui n'est pas optimal.

E9. Changement des poids

Nous récupérons les poids d'un modèle déjà entrainé sur une machine plus puissante et qui ne nécessite pas de contrainte de mémoire, et nous pouvons désormais enlever tous les tableaux d'entrainement puisque nous ne faisons plus de `BackPropagation` sur notre modèle, ainsi théoriquement, notre prédiction devrais aller plus vite. Nous mettons en place cette technique et nous testons et nous obtenons des prédictions qui se font beaucoup plus rapidement (à peu près 15 ms pour seulement la prédictions donc seulement appelé la fonction `FeedForward`).

T4. Entraînement avec Python

Oui, nous pouvons le faire facilement. Le but ici serait d'entrainer en amont notre réseau de neurones grâce à une machine beaucoup plus puissante que notre carte Arduino. Ainsi la solution serait d'entrainer sur Python avec la bibliothèque TensorFlow par exemple notre réseau de neurones avec toutes les données MNIST. Ensuite nous pourrons prendre les poids et les mettre directement sur notre code de l'Arduino DUE. Donc pour finir, Arduino devra seulement faire les prédictions avec les poids déjà entrainé ce qui nous fait gagner de la performance (en entrainant avec TensorFlow ainsi que toutes ses fonctions dédiées) et aussi de la rapidité.