Projet CLANU 2021-2022: Optimisation ADAM des MLP

O. Bernard, E. Bretin, T. Grenier, J-F. Mogniotte, L-V. Phung, C. Reichert

28 mars 2022

Résumé

Voici l'énoncé du projet CLANU partie informatique. Ce projet a pour objectif principal de vous faire concevoir un logiciel en langage C++ illustrant une méthode mathématique spécifique : l'optimisation ADAM pour l'apprentissage automatique par réseaux de neurones de type Perceptron Multi Couches (MLP). La partie informatique se focalise sur le développement C/C++ d'un MLP de classification en 9 classes d'images IRM.

Pour le module IF2, les compétences visées par ce projet CLANU sont :

— le développement et la mise en œuvre d'algorithmes d'optimisation en C/C++
— la production de code efficace en mémoire et en temps
— le respect de l'hygiène de codage
— l'analyse de résultats
— l'agrégation et la pérennisation des acquis (IF1, IF2 et préalables)
— l'autonomie en programmation et compréhension de code avancé

Concernant le déroulement et l'évaluation du projet CLANU **pour IF2**, voici les éléments importants :

— Le projet peut être conduit seul, en binôme ou en groupe,
— La seule évaluation de projet sera individuelle et portera sur les compétences visées par le projet. Elle sera réalisée sur moodle et un test 'blanc' sera fait au préalable.
— Les développements doivent être faits avec qmake/QtCreator. En cas d'incapacité à faire fonctionner ces outils sur son système personnel, les étudiants peuvent utiliser les ordinateurs de GE en physique ou via le VPN, ou encore par connexion au bureau à distante ou le bureau virtuel.
— Pour des raisons de performances et de stabilité de l'environnement de développement, linux est conseillé.

1 Présentation

Dans ce projet, vous allez travailler sur le problème de classification d'images IRM à l'aide d'un réseau de type MLP.

1.1 La base de données

La base de données IRM2D contient plus de 30000 images en niveau de gris, de taille 64×64 et au format png. Il s'agit des mêmes données que pour la partie MA du projet. Ces images sont des coupes 2D extraites de volumes 3D. Dans ce projet, il s'agit d'identifier d'une part la séquence IRM parmi : l'imagerie pondérée en T1, pondérée en T2 ou en densité de proton (PD) ; puis d'autre part, le plan de coupe anatomique parmi le plan axial (ou transverse), coronal (ou frontal) et sagittal. Un exemple est donné sur la figure 1. Les images sont classées en 9 classes numérotées de 0 à 8.

1.2 Le réseau de neurones

Le réseau de neurones utilisé est un MLP (*Multi layer Perceptron*).

Ce réseau a $L+1$ couches où la couche 0 est la couche d'entrée (valeur de l'entrée) et la couche L est la couche de sortie. Compte tenu des données d'entrée, la couche d'entrée aura $n_0 = 4096$ neurones. Concernant la couche de sortie, elle aura $n_L = 9$ neurones. En effet la classification, ici en 9 classes, nécessite l'utilisation du même nombre de neurones en sortie que de classe.

D'un point de vue informatique,

— n_0 sera `nInputUnit`
— n_L sera `nOutputUnit`
— n_j pour $j = 1 \dots L-1$ sera représenté par un tableau `nHiddenUnit`

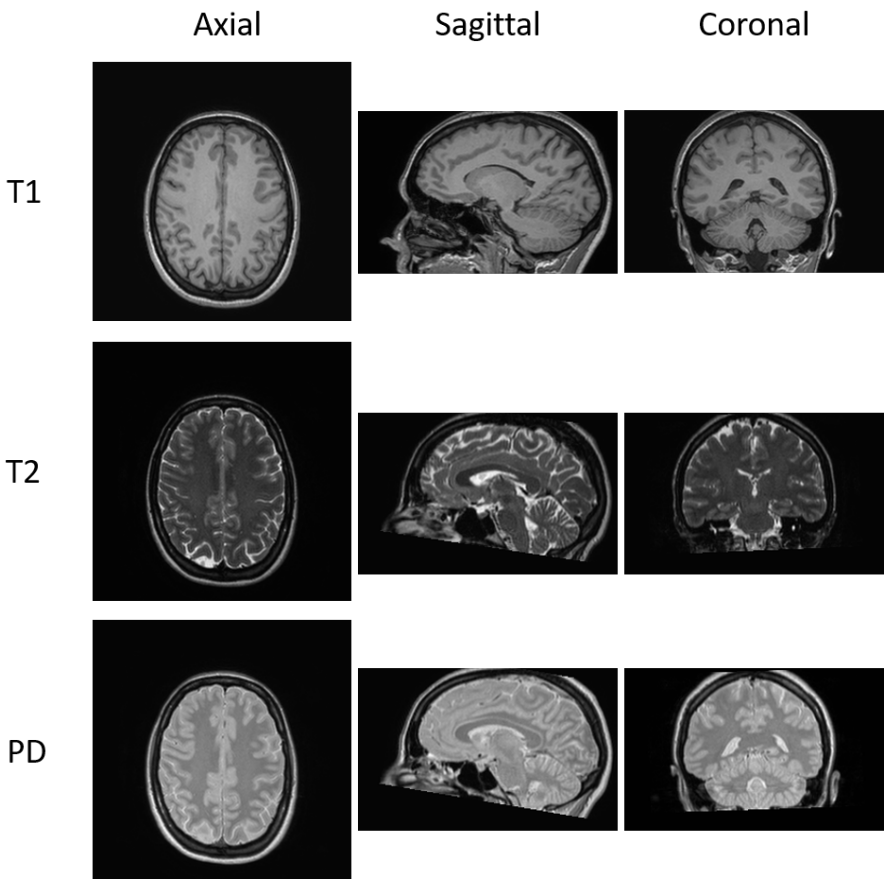

FIGURE 1 – Illustration des 9 classes du problème.

— Le nombre de couches cachées $L-1$ est `nHiddenLayer` (entre la couche d'entrée et de sortie)

Chaque neurone est constitué de ses poids W_j et d'un biais b. On notera W_{ij} les poids et b_i les biais des différents neurones d'une couche donnée.

Pour créer un réseau de neurones, il faut créer une variable (ou objet) de type `MLP_Network_SGD` puis allouer le réseau en fonction de son architecture.

```
1    MLP_Network_SGD mlp('L');
2    mlp.Allocate(nInputUnit,nHiddenUnit,nOutputUnit, nTrainingSet);
```

Le 'L' permet de désigner la fonction d'activation des couches cachées ('L' pour Leaky ReLU). La fonction d'activation de la couche de sortie est *softmax* (les 9 valeurs de sortie seront dans $[0;1]$).

1.3 La fonction de coût

Pour cette classification, nous avons opté pour l'entropie croisée entre la valeur à prédire y et la prédiction $\hat{y}$ pour une entrée x :

$$L(x,y;W,b) = -\sum_{i=1}^{i=n_L} y.log(h_{W,b}(x)) = -\sum_{i=1}^{i=n_L} y.log(\hat{y}) \tag{1}$$

avec x l'entrée du réseau $h_{W,b}$. La prédiction, qui est un vecteur de taille n_L, sera notée $\hat{y} = h_{W,b}(x)$ et la vérité $y \in \{0;1\}^{n_L}$ est encodée sous la forme *one-hot* ou encodage un parmi n. Dans cet encodage, la classe 0 est représentée par le vecteur $[1,0,0,0,0,0,0,0,0]$ et la classe 7 par $[0,0,0,0,0,0,0,1,0]$. Il faut bien noter que la prédiction $\hat{y}$ est la sortie du réseau après la fonction *softmax*, c'est à dire un vecteur de 9 valeurs : $\hat{y} \in [0;1]^{n_L}$.

Il ne faut pas confondre fonction de coût et mesure de performance.

— Le coût (ou *cost*, *loss function*) : la fonction de coût évaluée sur l'ensemble d'apprentissage (training loss) et sur l'ensemble de validation (validation loss). C'est sur le coût du jeu d'entrainement qu'est calculé le gradient et les évolutions des poids du réseau.

— La précision (accuracy) : ici le pourcentage d'exemples bien classés pour l'ensemble d'apprentissage (training accuracy) et pour l'ensemble de validation (validation accuracy) qui permet de savoir si le réseau répond bien au problème qui lui est posé.

Le réseau doit voir ses poids W et biais b évoluer de manière à faire décroitre L. Cette optimisation est faite par descente de gradient qui nécessite d'abord le calcul du gradient de L.

1.4 Le gradient

Vous avez vu en mathématiques que le gradient de L, l'ensemble des dérivées de L par rapport aux paramètres W_{ij} et b_i du réseau, est calculé avec l'algorithme de rétropropagation.

1. La passe avant de cet algorithme est réalisée par la fonction `mlp.ForwardPropagateNetwork( inputTraining[k])`. L'entrée appliquée au réseau `mlp` est `inputTraining[k]` c'est à dire la k-ème image d'entrée x_k. Le réseau prédit un résultat sur sa dernière couche.
2. La passe arrière est réalisée par la fonction `mlp.BackwardPropagateNetwork(desiredOutputTraining [k])` où `desiredOutputTraining[k]` est la valeur y_k souhaitée pour x_k. Tous les gradients des W et b sont calculés par cette fonction.

1.5 Descente de gradient stochastique (SGD)

La mise à jour des W et b du réseau de fait par l'appel de `mlp.UpdateWeight(learningRate)`. Cette fonction ne doit être appelée qu'après la passe arrière. Elle consiste à calculer les nouvelles valeurs $W_{ij}^{[t+1]}$ et $b_i^{[t+1]}$:

1. $W_{ij}^{[t+1]} \leftarrow W_{ij}^{[t]} - \eta \frac{\partial L(W_{ij}^{[t]})}{\partial W_{ij}^{[t]}}$
2. $b_i^{[t+1]} \leftarrow b_i^{[t]} - \eta \frac{\partial L(b_i^{[t]})}{\partial b_i^{[t]}}$

avec η le taux d'apprentissage.

Si on fait un appel à ces trois fonctions (Forward, Backward et Update) pour chaque image d'entrée x_k, on parle de descente de gradient stochastique (une mise à jour des poids du réseau est faite pour chaque image). Cette approche est assez rapide mais l'évolution de l'entrainement est souvent bruité.

L'approche de Descente de Gradient classique consisterait à faire une passe avant et arrière pour toutes les images d'entrée disponibles (le gradient des variables W et b se cumulent) puis ensuite d'appliquer une seule mise à jour des poids et biais. Cette approche peut être extrêmement longue pour des grandes bases de données. L'évolution de la fonction de coût est cependant très monotone.

Une approche intermédiaire consiste à prendre un petit groupe (ou **lot**) d'images tirées au hasard (sans remise) dans la base de données. Pour chaque image d'un lot, une passe avant et arrière est faite. Puis quand toutes les images du lot sont traitées, une mise à jour des poids et biais du réseau est faite puis on passe au lot suivant.

On défini le terme **d'époque** qui représente un nombre de répétitions de ce traitement par lot (ou *batch* ou *minibatch*). Ce nombre de lots permet de couvrir tout ou partie de la base de donnée. Quand on change d'époque, toutes les images sont remises et on procède à de nouveaux tirages de lots.

Cette approche par lot est disponible dans le code fourni. En utilisant la variable `nMiniBatch` pour fixer l'effectif des lots, on peut facilement passer de l'une à l'autre de ces 3 approches (SGD, cGD, par lot).

1.6 ADAM

Les approches SGD ne prennent pas en compte le passé de l'évolution du gradient. Plusieurs chercheurs ont montré qu'une telle information, permettant de créer une inertie pour la mise à jour des poids, accélère considérablement la minimisation de la fonction de coût (Nesterov, Duchi (AdaGrad), Hinton (RMSProp), Kingma (ADAM)...).

On propose dans ce projet CLANU d'implémenter la méthode ADAM (*Adaptive Momentum Estimation*). D'un point de vu informatique, voici le détail algorithmique d'ADAM pour les poids W_{ij} d'une couche du réseau :

1. $m_{W_{ij}}^{[t+1]} \leftarrow \beta_1 m_{W_{ij}}^{[t]} + (1-\beta_1) \frac{\partial L(W_{ij}^{[t]})}{\partial W_{ij}^{[t]}}$
2. $s_{W_{ij}}^{[t+1]} \leftarrow \beta_2 s_{W_{ij}}^{[t]} + (1-\beta_2) \left(\frac{\partial L(W_{ij}^{[t]})}{\partial W_{ij}^{[t]}} \right)^2$
3. $m_{W_{ij}}^{[t+1]} \leftarrow m_{W_{ij}}^{[t+1]} / (1-\beta_1^t)$

4. $s_{W_{ij}}^{[t+1]} \leftarrow s_{W_{ij}}^{[t+1]}/(1-\beta_2^t)$
5. $W_{ij}^{[t+1]} \leftarrow W_{ij}^{[t]} - \alpha_t m_{W_{ij}}^{[t+1]} / \left(\sqrt{s_{W_{ij}}^{[t+1]}} + \epsilon\right)$

avec β_1 et β_2 les taux de décroissances des moyennes mobiles exponentielles (usuellement autour de 0.9 et 0.99, respectivement), $m_{W_{ij}}^{[t+1]}$ et $s_{W_{ij}}^{[t+1]}$ les moyennes mobiles des moments d'ordre un et deux du gradient $\frac{\partial L(W_{ij}^{[t]})}{\partial W_{ij}^{[t]}}$ de $W_{ij}^{[t]}$, t est le numéro de l'itération, et $\alpha_t = \eta \frac{\sqrt{1-\beta_2^t}}{(1-\beta_1^1)}$, et η le taux d'apprentissage.

Les lignes 3 et 4 servent à accélérer le démarrage de l'algorithme car $m_{W_{ij}}^{[0]}$ et $s_{W_{ij}}^{[0]}$ sont initialisées à 0.

Cet algorithme est donné pour les W_{ij}, il est le même pour les biais b_i de chaque neurone.

2 Questions IF2

Une grande partie du travail consiste à analyser le code fourni et son fonctionnement. Ce code est long et complexe. Il prend en compte les éléments de programmation de toute la formation en informatique GE (IF1, IF2, IF3 et IF4).

Il est écrit en C++ et non en C (`printf` → `cout`, `scanf` → `cin`, `malloc` → `new`, ...).

Un des objectifs de formation est de vous faire adopter une méthode vous permettant de ne pas vous égarer dans les détails.

Il vous sera demandé quelques développements et des consignations dans un "rapport". Même si il n'y a pas de rapport à rendre, il est demandé de rédiger vos réponses et vos éléments de compréhension en vue de l'évaluation. Votre rapport et votre code seront les seuls éléments autorisés lors de l'évaluation individuelle.

2.1 Projet QtCreator

Commencer par récupérer le code du projet sur moodle. **Ouvrez avec QtCreator le projet** `MLP_ADAM.pro`**.** Ce projet contient 2 bibliothèques et le code à modifier.

2.2 Prise en main du projet

Dans cette partie, la performance d'un réseau de neurones pré-entrainé sur 5000 images va être mesurée sur le jeu de données test. A partir de la source `irm_test` disponible dans le projet `MLP_ADAM` :

1. Ajuster la variable `SRC_PATH` au début du fichier `irm_mlp_test.cpp`. Elle doit correspondre au répertoire de votre projet.
2. Exécuter le programme `irm_mlp_test`. Ce programme a besoin d'un argument : il s'agira de lire le fichier de réseau `models/best/irm_adam.bin`. L'argument `best/irm_adam.bin` est à passer en ligne de commande (dans QtCreator : *Projets* → *Run* → *"Command Line Argument"*).
3. Dans votre rapport vous donnerez un exemple d'exécution. La précision de ce réseau sur 1000 images de test est elle bonne par rapport à vos expérimentations en MA précédentes ? Qu'est ce que la "précision" exactement ? Comment est elle calculée et en quoi est elle plus appropriée que la fonction de coût pour apprécier la performance du réseau de neurones ?
4. Modifier le code pour que la précision soit calculée sur la totalité des images de test. La précision a t'lle évolué ? que conclure sur le réseau et les images ?
5. Passer le projet en mode *"Debug"* pour pouvoir mettre des *breakpoints* dans le code. A l'aide du débogage, déterminer l'architecture du réseau pré-entrainé `irm_ADAM.bin` (nombre de couches et nombre de neurones par couche) ainsi que les fonctions d'activation utilisées.

2.3 Premiers développements : comprendre la représentation des données et faire des prédictions

On se focalise sur la représentation des données (en C/C++) et la manipulation des fonctions pour effectuer une prédiction (ou inférence, ou test).

1. Dans le fichier `libread/IRM2D.cpp` répondre aux commentaires de la fonction d'affichage `void PrintImage(float *image, int r, int c)`. Cette fonction permet d'afficher les valeurs d'une matrice 64x64 en console. Dans ce projet, comment est représentée une matrice en mémoire ? Comment accède t'on à la valeur de l'élément (l,m) d'une matrice ? Dans le rapport, donner un exemple.
2. Dans le fichier `irm_mlp_test.cpp`, en vous inspirant de ce qui est fait dans le boucle `while`, ajouter dans le main une boucle permettant d'afficher les indexes de toutes les images qui n'ont pas été correctement classées.
3. Dans le rapport, vous donnerez les lignes de code permettant de faire une prédiction avec le réseau de neurones et notamment comment on déduit la classe à partir d'un *encodage one-hot* et à partir de la sortie du réseau.
4. Modifier le code de `irm_mlp_test.cpp` pour ne pas faire un affichage console mais enregistrer l'image avec la fonction `SaveImage` (voir fichiers `libread/IRM2D.h` et `libread/IRM2D.cpp`. Cette fonction sera appelée par `irm2D.SaveImage(...)` dans le main.

2.4 Seconds développements : entrainement, sauvegarde et ressources

On s'intéresse maintenant à l'apprentissage du réseau par une simple descente de gradient dont `irm_train_sgd.cpp` est le fichier principal.

1. Ajuster les variables `SRC_PATH` et `GNUPLOT_PATH` au début du fichier. Elles doivent correspondre au répertoire du projet et à l'exécutable *gnuplot*. Si nécessaire, installer *gnuplot* sur votre système (gnuplot existe pour linux, iOS et Windows).
2. Ce programme `irm_train_sgd` a besoin d'un argument passé en ligne de commande lors de son exécution : le nom du fichier dans lequel enregistrer les poids du réseau.
3. Exécuter le programme et comparer les exécutions en mode *Debug* et mode *Release* en termes de : temps de calcul et de valeurs obtenues. En plus de ces observations, consigner dans votre rapport comment sont réalisés en C/C++ :
   — les affichages dans la console
   — les affichages avec gnuplot
   — la création des fichiers de poids
4. Alléger la fonction *main* avec une fonction `ACCURACYandLOSS` qui retournera les valeurs d'*accuracy* et de *loss* pour un jeu de données et un réseau. Attention, les objets (variables) "mlp" **doivent impérativement être passés par adresse**. Cette fonction sera appelée 4 fois dans le main.
5. Pour un même réseau, comparer le temps moyen d'entrainement pour une époque sur -au moins- 2 PC différents. Dans le rapport, vous ferez un tableau donnant : les compilateurs utilisés, les systèmes d'exploitation, la référence du processeur, la quantité de mémoire RAM des ordinateurs utilisés, et enfin le temps pour l'apprentissage.
6. Vous pouvez tester différentes architectures du réseau pour améliorer les résultats, mais ne perdez pas trop de temps...(ce n'est pas l'objectif du projet).

2.5 Troisième développement : optimisation ADAM

Il s'agit d'implémenter l'optimisation ADAM.

1. Ajouter un exécutable `irm_train_adam` au projet. Pour cela, il est recommandé de **dupliquer** le répertoire `irm_train_sgd` dans un nouveau répertoire `irm_train_adam`. Ainsi le code sera dans un premier temps une copie de `irm_train_sgd.cpp`. Il vous faudra ensuite renommer les différents fichiers et éditer les fichiers .pro (celui dans `irm_train_adam` et `MLP_ADAM.pro` à la racine du projet) et au besoin... vous documenter sur le sujet !
2. Utiliser les entêtes `#include "MLP_Network_ADAM.h"` et `#include "MLP_Layer_ADAM.h"` à la place de ceux en "SGD" dans le fichier `irm_train_adam.cpp`. Faire de même avec le type des variables `mlp` et `mlp_best`.
3. Compléter la méthode `UpdateWeight` du fichier `MLP_Layer_ADAM.cpp` et valider votre solution. Lors de vos développements, il est demandé :
   — de vous inspirer du code de la fonction `UpdateWeight` de `MLP_Layer_SGD.cpp`
   — d'utiliser les variables `MW` et `MW_next` pour m_{Wij} et sa mise à jour, `SW` et `SW_next` pour s_{Wij} et sa mise à jour, puis de même pour les b_i. Noter que ces variables sont allouées et initialisées. De plus, α_t est déjà calculée (variable `alpha_T`) et les variables `Beta_1` et `Beta_2` vous permettent d'accéder aux valeurs de β_1 et β_2.
   — de désactiver la création des courbes avec *gnuplot* (commenter les 2 lignes du main commençant par `std::thread gnuplot_thread_ ...`)
   — de diminuer le nombre d'images pour l'entrainement et le test (ceci accélère l'exécution, très pratique pendant le développement et la recherche de bugs, mais diminue les performances de classification)
   — de faire régulièrement des compilations de tout le projet via "Tout recompiler" (*Compiler* → *Tout recompiler*) pour prendre en compte les modifications de `MLP_Layer_ADAM.cpp`.
4. Pour un même réseau (utiliser par exemple les valeurs ci-après), comparer les temps par époques, les évolutions des valeurs de coût et d'efficacité pour l'optimisation SGD et ADAM. Consigner vos observations et vos conclusions.

```
1    int nInputUnit    = 64*64;
2    int nOutputUnit   = 9;
3    vector<int> nHiddenUnit({32, 16});
```

```
4     int nHiddenLayer  = 2;
5
6     float learningRate = 0.0001;
7     int maxEpoch      = 250;
8     int nMiniBatch    = 50;
```

5. Ajuster les paramètres du réseau et ceux d'entrainement afin d'obtenir une bonne précision. Pensez à sauvegarder vos réseaux entrainés avec des noms différents ! Dans le rapport, donner les paramètres du réseau et les hyper-paramètres retenus ainsi que les performances obtenues et puis comparer avec celles de `models/best/irm_ADAM.bin`.

3 Évaluations et rendus

Ce projet rentre dans le cadre des modules IF2 et MA2. Ici, n'est détaillée que la partie IF2.

3.1 Organisation

Pour la partie IF2, le projet peut s'effectuer seul, en binôme ou en groupe mais il sera évalué individuellement.

Aucun rendu n'est demandé à l'issu de ce projet. Néanmoins, lors de l'évaluation individuelle, une production de code en lien avec le projet sera demandée ainsi que des éléments de réponses proches de ceux demandés dans le cadre du projet.

Votre enseignant de TD de C en IF1 pourra répondre à vos questions.

3.2 Evaluation individuelle d'Informatique

L'évaluation individuelle de la partie informatique du projet CLANU aura lieu en fin de semestre (proche semaine des DS avec aléas organisationnel). Cette évaluation CLANU de 60 minutes s'effectuera sur *moodle* et prendra la forme d'un TP noté sous QtCreator. Chaque étudiant doit s'assurer du bon fonctionnement de ses systèmes avant le jour du DS.

Un examen blanc sera organisé préalablement à l'examen final. Cet examen blanc est facultatif et sera disponible sur une grande plage temporelle. Afin de cibler les attentes pédagogiques, ce test reproduira les types de questions qui seront posées lors du DS et s'appuiera sur les mêmes outils.

Les questions du DS porteront sur les éléments étudiés au cours de la partie informatique du projet et en lien avec la partie mathématique. Cette évaluation a pour but d'évaluer les capacités en analyse, mathématique et programmation acquises lors de l'apprentissage par projet en s'appuyant sur les éléments mis en oeuvre lors du projet.

Les documents autorisés pour le DS seront les éléments de réponses aux questions du projet ainsi que le code source développé.