Projet d'Optimisation 2020 - Sujet 1

Apprentissage Profond et l'algorithme du gradient stochastique

Andy
Noah
Soline

2020-06

Table des matières

1. Introduction

Les progrès réalisés ces dernières années dans le domaine de la reconnaissance de l'écriture manuscrite sont dus en grande partie à l'utilisation d'approches statistiques. Parmi celles-ci, deux techniques ont plus particulièrement été mises à contribution. Il s'agit d'une part des approches connexionnistes et d'autre part des modélisations basées sur les modèles de Markov cachés. Les premières sont particulièrement intéressantes par leur fort pouvoir discriminant et leur capacité à construire des frontières de décisions complexes dans des espaces de grande dimension, les secondes modélisent par une approche paramétrique les séquences d'observations générées par des processus stochastiques tels que, par exemple, l'écriture manuscrite. Dans ce projet, nous nous intéressons à étudier l'architecture d'apprentissage profond : le réseau neuronal convolutif, qui a de nombreuses applications dans la reconnaissance d'image et vidéo, les systèmes de recommandation et le traitement du langage naturel. E.g., Étant données des images de chiffres écrits à la main (entre 0 à 9). Nous allons construire un modèle CNN et l'algorithme SGD pour reconnaitre l'écriture manuscrite.

2. Mathématiques

Nous allons utiliser le réseau neuronal convolutif pour classifier et reconnaître les images. Et nous allons utiliser l'algorithme du gradient stochastique pour l'objectif de l'optimisation.

2.1 Modèle de CNN

La convolution est le cœur du réseau neuronal convolutif. A l'origine, une convolution est un outil mathématique utilisé en retouche d'image. En fait, une convolution prend simplement en entrée une image et un filtre, effectue un calcul, puis renvoie une nouvelle image.
Il y a en général cinq types de couches importantes pour un réseau neuronal convolutif : l'entrée, la couche de convolution, la couche de pooling, la couche entièrement connectée et la sortie.

2.1.1 Entrée

*MNIST compresse l'image en noir et blanc contenant 784 pixels, représentée par une matrice carrée de taille 28(largeur)*28(hauteur) et en niveaux de gris (donc 1 seul canal).*
Nous conservons des entiers entre 0 et 255 alors que dans la version améliorée, nous avons converti en float avant de diviser par 255 pour avoir des valeurs entre 0 et 1. Cette conversion et normalisation permet de réduire l'écart entre les valeurs extrêmes et d'éviter les overflow dans les calculs.

2.1.2 Couche de convolution

La couche de convolution est la composante clé des réseaux de neurones convolutifs. Son but est de repérer la présence d'un ensemble de caractéristiques dans les images reçues en entrée. Pour cela, nous réalisons un filtrage par convolution : le principe est de faire "glisser" une fenêtre représentant la caractéristique sur l'image et de calculer le produit de convolution entre la caractéristique et chaque portion de l'image balayée.
La couche de convolution est caractérisée par trois paramètres :
1. La taille du neurone convolutif : dans notre projet, la taille du neurone convolutif est de $f_1 * f_2$ *;*

2. Le pas(noté s) contrôle le chevauchement des champs récepteurs : plus le pas est petit, plus les champs récepteurs se chevauchent et plus le volume de sortie sera grand ;
3. Le padding(p) : c'est de faire un rembourrage autour de l'image pour éviter notre image rétrécir à chaque fois qu'elle est annalysée des bords ou d'autres caractéristiques et éviter de perdre beaucoup d'informations sur les bords de l'image.
Et on appelle profondeur de la couche le nombre des noyaux de convolution.
Lorsque le neurone convolutif fonctionne, les caractéristiques d'entrée sont balayées régulièrement, et calculées par le produit matriciel et l'addition en considérant un biais améliorable dans le champ récepteur :

$$Z_d^r(i,j) = \sum_{c=1}^{C^{r-1}} \sum_{m=1}^{f_1^{r-1}} \sum_{n=1}^{f_2^{r-1}} w_{c,d}^{r-1}(m,n) pZ_c^{r-1}(is_1^{r-1}+m, js_2^{r-1}+n) + b_d^{r-1} \tag{2.1}$$

$$avec\ H^r = \frac{H^{r-1}-f_1^{r-1}+2p_1^{r-1}}{s_1^{r-1}}+1 \quad L^r = \frac{L^{r-1}-f_2^{r-1}+2p_2^{r-1}}{s_2^{r-1}}+1 \quad i \in [1,H^r] \quad j \in [1,L^r]$$

$$Z_d^r(i,j)\ :\ valeur\ de\ la\ position\ (i,j)\ du\ d^{ime}\ canal\ de\ la\ r^{ime}\ couche$$

(f_1^{r-1}, f_2^{r-1}), (s_1^{r-1}, s_2^{r-1}) *et* (p_1^{r-1}, p_2^{r-1}) *: dimension, pas et padding du neurone connectant la* $(r-1)^{ime}$ *et la* r^{ime} *couche de convolution*

$$w^{r-1}\ et\ b^{r-1}\ :\ poids\ et\ biais\ du\ neurone\ connectant\ la\ (r-1)^{ime}\ et\ la\ r^{ime}\ couches\ de\ convolution$$

$$pZ^{r-1}\ :\ vecteur\ bidimensionnel\ après\ le\ padding\ de\ Z^{r-1}$$

$$Z^r\ :\ Résultats(map\ caractéristique)\ de\ la\ r^{ime}\ couches\ de\ convolution$$

$$H^r\ :\ hauteur\ de\ Z^r \quad L^r\ :\ largeur\ de\ Z^r$$

2.1.3 Couche de pooling

La couche de pooling permet de réduire le nombre de paramètres et de calculs dans le réseau. On améliore ainsi l'efficacité du réseau et on évite le sur-apprentissage.

Il existe plusieurs types de pooling :
«Max pooling» : qui revient à prendre la valeur maximale de la sélection.
«Average pooling», soit la moyenne des pixels de la sélection : on calcule la somme de toutes les valeurs et on divise par le nombre de valeurs.
«Sum pooling» : c'est la moyenne des pixels sans avoir divisé par le nombre de valeurs
De manière générale, il est recommandé d'utiliser max-pooling car en pratique, il se distingue

de average pooling sur les cas extrêmes et est quasiment équivalent à average pooling dans les autres cas.

*Dans notre projet, nous utilisons le max pooling de taile 2*2 et de pas 1.*

2.1.4 Couche entièrement connectée

La couche entièrement connectée reçoit un vecteur en entrée et produit un nouveau vecteur en sortie. Pour cela, elle applique une combinaison linéaire puis éventuellement une fonction d'activation aux valeurs reçues en entrée. La couche entièrement connectée permet de classifier l'image en entrée du réseau : elle renvoie un vecteur de taille N, où N est le nombre de classes dans notre problème de classification d'images.

2.1.5 Sortie

L'amont de la couche de sortie est généralement une couche entièrement connectée. Chaque élément du vecteur indique la probabilité pour l'image en entrée d'appartenir à une classe. Chaque valeur du tableau en entrée vote en faveur d'une classe. Mais les votes n'ont pas tous la même importance : la couche leur accorde des poids qui dépendent de l'élément du tableau et de la classe. Pour calculer les probabilités, la couche de sortie multiplie donc chaque élément en entrée par un poids, fait la somme, puis applique une fonction d'activation(logistique si N=2, softmax si N>2).

2.2 Opération et Fonctionnement

2.2.1 Propagation

La première étape est l'étape de propagation vers l'avant. On prélève un échantillon et le rentre dans le réseau et calcule le resultat correspondant à la sortie. À ce stade, les informations de la couche d'entrée sont transférées à la couche de sortie étape par étape et ce processus est également le processus que le réseau effectue après la formation.

Nous notons n le nombre de couches de notre réseau neuronal convolutif, le output de la r^{ime} *couche est noté* a^r *;* w^r *: Poids de la* r^{ime} *couche ;* b^r *: Biais de la* r^{ime} *couche ;* σ *: Fonction d'activation, c'est généralement ReLU(rectification linéaire).*

Pour $r \in [2, n-1]$ *:*

1. If la r^{ime} couche est la couche de convolution, alors

$$Z^r = ReLU(Z^{r-1} \times w^r + b^r) \tag{2.2}$$

Soit explicit,

$$Z_d^r(i,j) = \sum_{c=1}^{C^{r-1}} \sum_{m=1}^{f_1^{r-1}} \sum_{n=1}^{f_2^{r-1}} w_{c,d}^{r-1}(m,n) pZ_c^{r-1}(is_1^{r-1} + m, js_2^{r-1} + n) + b_d^{r-1} + b_d^{r-1} \tag{2.3}$$

2. If la r^{ime} couche est la couche de pooling, alors

$$Z^r = pool(Z^{r-1}) \tag{2.4}$$

«pool» est le processus de réduction du tenseur d'entrée en fonction de la taille de la zone de pooling et des normes de pooling.
Soit explicit, pour «Max pooling»,

$$Z_c^r(i,j) = \max_{is_1^{r-1} \leq m < is_1^{r-1} + f_1^{r-1}; js_2^{r-1} \leq n < js_2^{r-1} + f_2^{r-1}} (pZ_c^{r-1}(i,j)) \tag{2.5}$$

3. If la r^{ime} couche est la couche entièrement connectée, alors

$$Z^r = \sigma(Z^{r-1} \times w^r + b^r) \tag{2.6}$$

Pour r=n(sortie) :

$$Z^n = softmax(Z^{n-1} \times w^n + b^n)) \tag{2.7}$$

2.2.2 Rétropropagation de la couche entièrement connectée et de la fonction de loss

Nous notons F : la fonction de coût ;
δ^i : la dérivée partielle de F par rapport à Z^i du neurone de la i^{ime} couche

$$\delta^i = \frac{\partial F}{\partial Z^i} \tag{2.8}$$

Ensuite, nous calculons par les calculs des dérivées partielles et par la récurrence et en utilisant la règle de la chaîne :

$$\frac{\partial F}{\partial w_{i,j}^{r-1}} = \frac{\partial F}{\partial Z_j^r} * \frac{\partial Z_j^r}{\partial w_{i,j}^{r-1}} = \delta_j^r * \frac{\partial(\sum_{k=1}^{l_{|r-1|}} Z_k^{r-1} w_{k,j}^{r-1} + b_j^{r-1})}{\partial w_{i,j}^{r-1}} = \delta_j^r * Z_i^{r-1} \tag{2.9}$$

avec Z_j^r : l'élément associé à la $j^{(ime)}$ colonne du vecteur ligne Z^r ; $w_{i,j}^{r-1}$: l'élément associé au position(i,j) de la matrice w^{r-1} qui signifie le poids de la $(r-1)^{ime}$ couche ; ($\mid l_{r-1} \mid$, $\mid l_r \mid$) : la dimension de w^{r-1}.
Donc, nous pouvons obtenir :

$$\frac{\partial F}{\partial w^{r-1}} = \delta^r \times (Z^{r-1})^T \tag{2.10}$$

De la même manière, nous pouvons obtenir :

$$\frac{\partial F}{\partial b^{r-1}} = \delta^r \tag{2.11}$$

Nous obtenons de ce processus les dérivées partielles de la fonction de coût F par rapport au poids et au biais pour la r^{ime} couche($r \in [2, n]$).
Ensuite, nous pouvons calculer :

$$\begin{aligned}
\delta_i^r &= \frac{\partial F}{\partial Z_i^r} \\
&= \frac{\partial F}{\partial Z^{r+1}} * \frac{\partial Z^{r+1}}{\partial Z_i^r} \\
&= \frac{\partial F}{\partial Z^{r+1}} * \frac{\partial Z^r \times w^r + b^r}{\partial Z_i^r} \\
&= \sum_{j=1}^{|l_{r+1}|} \frac{\partial F}{\partial Z_j^{r+1}} * \frac{\partial (\sum_{k=1}^{|l_r|} Z_k^r \times w_{k,j}^r + b_j^r)}{\partial Z_i^r} \\
&= \sum_{j=1}^{|l_{r+1}|} \delta_j^{r+1} \times w_{i,j}^r \\
&= \delta^{r+1} \times ((w^r)^T)_i
\end{aligned} \tag{2.12}$$

avec (1, $\mid l_r \mid$) : la dimension de b^{r-1}

Plus généralement,

$$\delta^r = \frac{\partial F}{\partial Z^r} = \delta^{r+1} \times (w^r)^T \tag{2.13}$$

Par récurrence immédiate, nous obtenons :

$$\delta^r = \delta^{r+1} \times (w^r)^T = \delta^{r+2} \times (w^{r+1})^T \times (w^r)^T = \delta^n \times (w^{n-1})^T \times (w^{n-2})^T \times ... \times (w^r)^T \tag{2.14}$$

2.2.3 Rétropropagation de la couche de convolution (avec padding) et de la fonction de coût

$$\begin{aligned}\frac{\partial F}{\partial w_{c,d}^{r-1}(m,n)} &= \sum_{i=1}^{H^r}\sum_{j=1}^{L^r}\frac{\partial F}{\partial Z_d^r(i,j)} * \frac{\partial Z_d^r(i,j)}{\partial w_{c,d}^{r-1}(m,n)} \\ &= \sum_{i=1}^{H^r}\sum_{j=1}^{L^r}\delta_d^r(i,j) * \frac{\partial(\sum_{c=1}^{C^{r-1}}\sum_{m=1}^{f_1^{r-1}}\sum_{n=1}^{f_2^{r-1}} w_{c,d}^{r-1}(m,n) pZ_c^{r-1}(is_1^{r-1}+m, js_2^{r-1}+n)+b_d^{r-1})}{\partial w_{c,d}^{r-1}(m,n)} \\ &= \sum_{i=1}^{H^r}\sum_{j=1}^{L^r}\delta_d^r(i,j) * pZ_c^{r-1}(is_1^{r-1}+m, js_2^{r-1}+n)\end{aligned} \tag{2.15}$$

De la même manière, nous obtenons :

$$\frac{\partial F}{\partial b_d^{r-1}} = \sum_{i=1}^{H^r}\sum_{j=1}^{L^r}\delta_d^r(i,j) \tag{2.16}$$

Toujours par les calculs comme le processus explcité dans (2.3.2) et avec des opérations matricielles plus complexes, nous obtenons à la fin :

$$\frac{\partial F}{\partial pZ_c^{r-1}(i,j)} = \sum_{c=1}^{C^r}\sum_{m=1}^{f_1^{r-1}}\sum_{n=1}^{f_2^{r-1}} rot_\pi w_{c,d}^{r-1}(m,n) p\delta_d^{r_{padding}}(i+m, j+n) \tag{2.17}$$

avec $p\delta_d^{r_{padding}}(i,j)$ après le padding((f_1^{r-1}, f_2^{r-1}) de 0) autour de $\delta^{r_{padding}}$ ($\delta^{r_{padding}}$: après le padding de $(s_1^{r-1}-1, s_2^{r-1}-1)$ de 0 dans la matrice de δ^r).

$$\begin{aligned}\delta_c^{r-1} &= (\frac{\partial F}{\partial pZ_c^{r-1}(i,j)})_{(p_1^{r-1}\le i<H^{r-1}+p_1^{r-1}, p_2^{r-1}\le j<L^{r-1}+p_2^{r-1})} \\ &= (\sum_{c=1}^{C^r}\sum_{m=1}^{f_1^{r-1}}\sum_{n=1}^{f_2^{r-1}} rot_\pi w_{c,d}^{r-1}(m,n) p\delta_d^{r_{padding}}(i+m, j+n))_{(p_1^{r-1}\le i<H^{r-1}+p_1^{r-1}, p_2^{r-1}\le j<L^{r-1}+p_2^{r-1})}\end{aligned} \tag{2.18}$$

2.2.4 Rétropropagation du pooling

Nous utilisons «Max pooling» dans notre projet.
On note I(a,b,c)=$(i,j) \mid arg\max_{m,n}(pZ_c^{r-1}(i,j))_{(is_1^{r-1}\le m<is_1^{r-1}+f_1^{r-1}, js_2^{r-1}\le n<js_2^{r-1}+f_2^{r-1})}$: l'ensemble

des position(i,j) dans la r^{ime} couche tel que la valeur maximale du cananl c dans la $(r-1)^{ime}$ couche dans le processus de «Max pooling».

$$\frac{\partial F}{\partial pZ_c^{r-1}(a,b)} = \sum_{(i,j)\in I(c,a,b)} \frac{\partial F}{\partial Z_c^r(i,j)} * \frac{\partial Z_c^r(i,j)}{\partial pZ_c^{r-1}(a,b)} = \sum_{(i,j)\in I(c,a,b)} \delta_c^r(i,j) \tag{2.19}$$

Et donc, la matrice

$$\begin{aligned} \delta_c^{r-1} &= \left(\frac{\partial F}{\partial pZ_c^{r-1}(a,b)}\right)_{(p_1^{r-1}\leq a<H^{r-1}+p_1^{r-1}, p_2^{r-1}\leq b<L^{r-1}+p_2^{r-1})} \\ &= \left(\sum_{(i,j)\in I(c,a,b)} \delta_c^r(i,j)\right)_{(p_1^{r-1}\leq a<H^{r-1}+p_1^{r-1}, p_2^{r-1}\leq b<L^{r-1}+p_2^{r-1})} \end{aligned} \tag{2.20}$$

2.2.5 Fonction d'activation

Si aucune fonction d'activation n'est utilisée, quel que soit le nombre de couches du réseau neuronal, la sortie est une combinaison linéaire d'entrées. La fonction d'activation introduit des facteurs non linéaires dans le neurone de sorte que le réseau neuronal peut se rapprocher de toute fonction non linéaire pour être appliqué à de nombreux modèles non linéaires.
Nous utilisons deux types de fonctions d'activation dans notre projet.

1. « softmax »

Nous utilisons un « softmax » qui est une étape purement mathématique qui normalise le vecteur de sortie pour qu'il représente des probabilités (ça ne change en rien l'ordre des prédictions, les valeurs de sortie sont simplement ramenées entre 0 et 1 de sorte à ce que leur somme fasse 1). L'expression de softmax :

$$a_i = \frac{e^{y_i}}{\sum_k e^{y_k}} \tag{2.21}$$

avec a_i : le i^{me} output de softmax ; y_i : le i^{me} output du neurone.

2. « ReLU »

La fonction d'activation ReLU (rectification linéaire) est une fonction dite « rectifier » très utilisée en Deep Learning. Dans les réseau de neurones convolutionnels, elle est appliquée très souvent en sortie d'une couche de convolutions. Une convolution va réaliser des opérations d'additions et de multiplications : les valeurs en sorties sont donc linéaires par rapport à celles en entrée. Mais dans une image, la linéarité n'est pas très présente ni importante (par exemple, les variations entre valeurs de pixels peuvent être importantes dans une région, l'image a des

coins...). ReLU, par sa définition, est une fonction qui vient briser (une partie de) la linéarité toutes les valeurs négatives. En supprimant une partie des données, ReLU permet également d'accélérer les calculs.

$$ReLU(Z) = \begin{cases} 0, & si\ Z \leq\ 0 \\ Z, & si\ Z > 0 \end{cases}$$

Nous notons le output de la r_{ime} couche Z_r, et puis le output devient Z_{r+1} après la fonction d'activation.

$$\delta^r = \frac{\partial F}{\partial Z^{r+1}} * \frac{\partial Z^{r+1}}{\partial Z^r} = \delta^{r+1} \frac{\partial ReLU(Z^r)}{\partial Z^r} = \delta^{r+1} \begin{cases} 0, & si\ Z \leq\ 0 \\ 1, & si\ Z > 0 \end{cases} = \begin{cases} 0, & si\ Z^r \leq\ 0 \\ \delta^{r+1}, & si\ Z^r > 0 \end{cases} \quad (2.22)$$

2.2.6 Flattening (mise à plat)

Le flattening consiste simplement à mettre bout à bout toutes les images (matrices) que nous avons pour en faire un (long) vecteur. Nous plaçons le flattening entre la couche de convolution et la couche entièrement connectée. Le output de la couche de convolution est un tenseur bidimensionnel. Après la couche de convolution, nous obtenons des images caractéristique et nous devons en convertir sous forme de vecteurs pour les lier à la couche entièrement connectée.

2.3 SGD(la descente stochastique du gradient)

2.3.1 Algorithme de descente stochastique en gradient

La méthode de descente en gradient consiste à prendre une valeur de «a» aléatoirement et la faire varier plus ou moins fortement par rapport à la pente de la fonction objective. Au lieu de tester toutes les valeurs de «a», nous faisons varier sa valeur avec des pas variables qui deviennent de plus en plus petits au fur et à mesure que l'on se rapproche du minimum.
L'algorithme de descente de gradient met à jour les paramètres de la fonction de coût F comme suit :

$$w_{k+1} = w_k - \eta_k \nabla_w F(a, w) \quad (2.23)$$

avec η_k le taux d'apprentissage

On regarde maintenant F(w). L'objectif, c'est de minimiser la fonction de coût donc la conditon nécessaire est que $\frac{dF}{dw}$ soit égale à 0. On fait tourner l'algorithme plusieurs fois, et de garder le plus petit des minima.

2.3.2 Théorème de la convergence

Notre but est de minimiser la fonction du coût F. On considère d'abord les fonctions du coût particulières avec la méthode de la descente du gradient générale (GD).

Supposons que F est convexe et que son gradient est Lipschitzien-continu avec la constante L. C'est-à-dire :

$$\|\nabla F(x) - \nabla F(y)\| \leq L \|x - y\| \tag{2.24}$$

Alors la fonction F possède la propriété suivante :

$$F(y) \leq F(x) + \nabla F(x)^T (y - x) + \frac{L}{2} \|y - x\|^2 \tag{2.25}$$

On suppose que y est la valeur renouvelée à partir de la valeur x. D'après notre algorithme

$$y = x - \eta \nabla F(x) \tag{2.26}$$

On met la relation entre x et y dans la propriété introduite avant, et on obtient :

$$F(y) \leq F(x) - (1 - \frac{L\eta}{2})\eta \|\nabla F(x)\|^2 \tag{2.27}$$

On prend le taux d'apprentissage η tel que $0 < \eta < \frac{1}{L}$, alors on obtient :

$$F(y) \leq F(x) - \frac{\eta}{2} \|\nabla F(x)\|^2 \tag{2.28}$$

On remarque bien dans cette contion une diminuation du coût. Maintenant on démontre la convergence de GD.

Notons x^* la valeur optimale du modèle. D'après la convexité de F, on a :

$$F(x) \leq F(x^*) + \nabla F(x)^T (x - x^*) \tag{2.29}$$

On met cette relation dans l'équation (2.28) et on a :

$$
\begin{aligned}
F(y) - F(x^*) &\leq \nabla F(x)^T (x - x^*) - \frac{\eta}{2} \left\| \nabla F(x) \right\|^2 \\
&= \frac{1}{2\eta} (2\eta \nabla F(x)^T (x - x^*) - \eta^2 \left\| \nabla F(x) \right\|^2) \\
&= \frac{1}{2\eta} (\left\| x - x^* \right\|^2 + 2\eta \nabla F(x)^T (x - x^*) - \eta^2 \left\| \nabla F(x) \right\|^2 - \left\| x - x^* \right\|^2) \\
&= \frac{1}{2\eta} (\left\| x - x^* \right\|^2 - \left\| x - \eta \nabla F(x) - x^* \right\|^2)
\end{aligned}
\tag{2.30}
$$

On considère la somme pour T étapes :

$$
\begin{aligned}
\sum_{t=1}^{T} F(x_t) - F(x^*) &\leq \sum_{t=1}^{T} \frac{1}{2\eta} (\left\| x_{t-1} - x^* \right\|^2 - \left\| x_t - x^* \right\|^2) \\
&\leq \frac{1}{2\eta} (\left\| x_0 - x^* \right\|^2 - \left\| x_T - x^* \right\|^2) \\
&\leq \frac{1}{2\eta} \left\| x_0 - x^* \right\|^2
\end{aligned}
\tag{2.31}
$$

Pour chaque étape, le coût diminue. Donc on obtient :

$$
\begin{aligned}
F(x_T) - F(x^*) &\leq \frac{1}{T} \sum_{t=1}^{T} F(x_t) - F(x^*) \\
&\leq \frac{1}{2\eta T} \left\| x_0 - x^* \right\|^2
\end{aligned}
\tag{2.32}
$$

Dans cette condition, on trouve bien la convergence de GD tant que le nombre d'étapes T soit suffisamment grand.

Pour la méthode de SGD, ou bien SGD sous forme de mini-batch, l'idée est similaire. Comme on prend au hasard un échantillon ou un mini-batch dans l'ensemble total des échantillons, on n'obtient pas le gradient de tout ensemble de données. Cependant, en faisant un touillage de données, on peut considérer que la distribution des échantillons soit aléatoire. Si on note $l_i(x)$ le coût pour le i^{ime} échantillon ou mini-batch, D'après la loi forte de grand nombre, on peut obtenir :

$$
\mathbb{E}(\nabla l_i(x)) = \mathbb{E}(\nabla F(x)) \tag{2.33}
$$

Donc dans ce cas, on obtient une convergence en espérance (convergence au sens L^1).

2.3.3 Complexité

En considérant le résultat obtenu dans la section précédente, on trouve un taux de convergence $O(1/T)$. Si on voudrais avoir une précision $F(x_T) - F(x^*) \leq \varepsilon$, il faut donc une complexité $O(1/\varepsilon)$ qui représente le nombre d'itérations qu'on a besoin. C'est le cas où le taux d'apprentissage est constant au cours de l'optimisation.

On remarque que la complexité dépend aussi du taux d'apprentissage η. Si on prend un taux d'apprentissage différent dans chaque étape, par exemple, un taux qui diminue au cours du temps vérifiant $\eta = O(1/t)$, alors dans ce cas, le taux de convergence deviendra $O(1/\sqrt{T})$. On trouve bien que la complexité devient plus grande, ce qui est raisonable car on «apprend plus lentement» avec un taux d'apprentissage plus petit.

2.3.4 Choix des paramètres

On a discuté dans la section du théorème de la convergence la méthode de SGD dans le cas où la fonction du coût est convexe avec un gradient Lipschitzien-continu. Cependant, en réalité, à cause de l'existence de plusieurs couches et des couches de convolution, la fonction du coût n'est jamais convexe. Le plus souvent, ce que l'on obtient par le modèle n'est pas un résultat optimal global, mais un résultat local, ou seulement un point selle. Par conséquent, pour les méthodes d'optimisation, surtout pour SGD qui est la plus simple, afin d'avoir un résultat de précision satisfaisante, il faut bien choisir les hyper-paramètres tels que la structure du réseau neuronal, les fonction d'activation, le taux d'apprentissage, etc. Souvent, on doit tâtonner et essayer plusieurs valeurs avant de le trouver.

Pour SGD, afin d'éviter que le modèle ne tombe dans une solution optimale locale, on augment souvent le taux l'apprentissage. Cepentant, pour que le modèle s'approche du point optimal, on peut faire décroître le taux l'apprentissage. De plus, le touillage des échantillons est aussi utile pour que l'on ne reste pas dans une solution optimale locale parce que dans ce cas, la fonction du coût optimal change au cours du temps, ce qui change aussi la distribution des points optimaux locaux.

2.3.5 Comparaison avec d'autres algorithmes

1. AdaGrad

AdaGrad permet au taux d'apprentissage de s'adapter en fonction des paramètres. Il ajuste le taux d'apprentissage sur chaque dimension en fonction du gradient de la variable, évitant

ainsi le problème que le même taux d'apprentissage est difficile à adapter à toutes les dimensions. L'algorithme est :

$$s_k = s_{k-1} + \nabla_w^2 \tag{2.34}$$

avec ϵ une constante ajoutée pour maintenir la stabilité numérique (éviter la division par zéro), ex : ϵ=10^{-6}

$$w_{k+1} = w_k - \frac{\eta_k}{\sqrt{s_k + \epsilon}} \nabla_w \tag{2.35}$$

Il effectue des mises à jour plus importantes pour les paramètres peu fréquents et des mises à jour plus petites pour ceux qui sont fréquents. Pour cette raison, il est bien adapté aux données éparses (NLP ou reconnaissance d'image). Un autre avantage est qu'il élimine fondamentalement la nécessité d'ajuster le taux d'apprentissage.
Chaque paramètre a son propre taux d'apprentissage et en raison des particularités de l'algorithme le taux d'apprentissage est monotone décroissant. Cela provoque le plus gros problème : point de temps le taux d'apprentissage est si faible que le système cesse d'apprendre.

2. RMSProp

Lorsque le taux d'apprentissage diminue plus rapidement dans les premières itérations et que la solution actuelle est toujours médiocre, Adagrad peut trouver difficile de trouver une solution utile car le taux d'apprentissage est trop faible dans les itérations ultérieures. Face à ce problème, l'algorithme RMSProp apporte une petite modification à Adagrad. Avec 0<γ<1,

$$s_k = \gamma s_{k-1} + (1 - \gamma)\nabla_w^2 \tag{2.36}$$

$$w_{k+1} = w_k - \frac{\eta_k}{\sqrt{s_k + \epsilon}} \nabla_w \tag{2.37}$$

3. Retour à SGD

- Avantage :

L'algorithme de gradient de descente stochastique est un algorithme simple mais constitue une approche efficace pour l'apprentissage de classifieurs linéaires sous des conditions de coûts convexes. Au lieu de tester toutes les valeurs, nous faisons varier sa valeur avec des pas variables qui deviennent de plus en plus petits au fur et à mesure que l'on se rapproche du minimum. Ça va évitea certains exemples redondants et augmenter l'efficacité.

- Désavantage

Les propriétés de convergence des paramètres de réseaux de neurones profonds restent mal comprises car les fonctions de coût ne sont pas convexes. L'optimisation de ces réseaux garde donc une forte composante expérimentale. Pour que l'optimisation donne de bons résultats, il faut qu'il y ait une faible densité de minima locaux qui ont un coût bien plus important que les minima globaux. De plus, en raison du bruit, les étapes d'apprentissage ont des oscillations parasites donc il est possible que la fonction de coût semble qu'elle ne converge jamais.

3. Simulation

Dans cette partie, nous allons représenter et discuter les résultats de la reconnaissance de l'écriture manuscrite obtenus par nos modèles de CNN.

3.1 Modèle général

Dans cette similation, on réalise tous les algorithmes en utilisant Python et Numpy, par exemple, les fonctions de propation pour les couches entièrement connectés, les couches de convolution, de pooling, de flatten, de ReLU, ainsi que les fonctions du coût et les algorithmes d'opimisation. Il est normal que nos modèles marchent plus lentement que ceux réalisés par TensorFlow ou PyTorch, en considérant que Numpy fonctionne avec un seul CPU, sans que les modèles soient accélérés par le GPU. Par conséquent, on a choisi une structure de CNN plus simple, comme montré dans la figure au dessous.

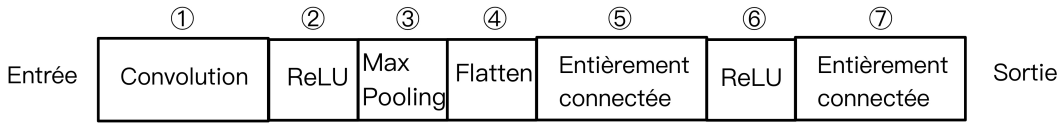

FIGURE 3.1 – Structure de CNN utilisé dans la similation

Dans ce CNN, il existe en total 7 couches contenant une seule couche de convolution. On met 16 filtre dans cette couche. La taille de chaque filtre est $3*3$*, et le pas de ces filtre est* 1 *au cours du fonctionnement du modèle. Comme montré au dessus, on a choisi ReLU comme la fonction d'activation afin d'éviter le problème de disparition du gradient. Pour la couche de Max pooling, on met un filtre de taille* $2*2$ *et de pas* 2*. La première couche entièrement connectée, c'est-à-dire la couche 5, est composée de 64 neurones, et la deuxième, c'est-à-dire la couche 7, en est composée 10 afin d'avoir le résultat qui est composé de 10 probabilités prevues associées à chaque nombre écrit.*

3.2 Analyse des résultats

3.2.1 Modèle 1 - SGD

Dans ce modèle, on utilise SGD comme l'algorithme d'optimisation. Pour la réalisation de SGD, il faut prendre comme argument un dictionnaire contenant tous les paramètres dans le CNN, ainsi qu'un dictionnaire contenant tous les gradients de ces paramètres. Dans cet algorithme, il faut traverser les paramètres en les renouvelant. La relation entre les nouveaux paramètres et leur anciennes valeurs est donnée dans la partie 2.3.1.

Dans ce projet, ce modèle est réalisé dans le document « 1_Model_sgd_984.ipynb » où l'algorithme SGD est réalisé comme une fonction « sgd ».

On fait fonctionner le modèle. On remarque que dans ce modèle, en raison de la pratique, on n'utilise pas le SGD classique mais celui sous forme de mini-batch. À chaque étape, on prend un mini-batch, on calcul les gradients moyens à partir de ce mini-batch comme les gradients des paramètres dans le modèle, et on renouvelle ces paramètres grâce à la fonction « sgd ». Tant que l'on a traversé tout l'ensemble de données, on appelle que l'on a fini une époque.

On prend le résultat après 8 époques. Dans ce contexte, la précision sur l'ensemble de validation est 0.984*. Grâce aux paramètres sauvegardés dans le dossier « 200612094306 », on peut tester le modèle sur l'ensemble de test, et on obtient la même précision* 0.984*.*

On remarque que c'est déjà un résultat satisfaisant pour notre modèle qui est assez simple. En fait, l'ensemble de données MNIST est facile pour entraîner un modèle NN avec SGD. D'après notre test préliminaire, même si on utilise seulement un NN avec les couches entièrement connectées, c'est-à-dire un DNN, on peut déjà avoir une précision environ 0.95*. De plus, on a réalisé un modèle CNN avant où il y avait seulmement 1 filtre dans la couche de convolution. Dans ce cas, on obtient une précision environ* 0.97*, qui est plus grande que DNN grâce à la convolution, en même temps, on trouve bien que le CNN deviendra plus fort avec plus de filtres.*

Pour savoir plus de détails de ce modèle ainsi que les résultats précis obtenus, vous pouvez consulter le document « 1_Model_sgd_984.ipynb » où ce modèle est réalisé avec le dossier « 200612094306 » où les paramètres bien entraînés sont conservés.

(Remarque : En considérant la normativité des codes, tous les commentaires dans les .ipynb documents ainsi que le document « README » sont écrits en angais.)

3.2.2 Modèle 2 - SGD avec moment et défaillance du taux d'apprentissage

Dans ce modèle, on utilise le principe similaire que SGD (mini-batch), mais cette fois on ajoute un moment au SGD et on fait défaillir le taux d'apprentissage au cours d'entraînement du modèle. C'est-à-dire, on choisit d'abord un taux de défaillance $d \ll 1$. *À chaque étape, le taux d'apprentissage diminue de manière suivante :*

$$\eta_t = \frac{\eta_{t-1}}{1 + t \times d} \tag{3.1}$$

Soit un paramètre w *de gradient* ∇w_{t-1} *à l'instant* t. *On définit :*

$$v_t = \gamma v_{t-1} + \eta_t \nabla w_{t-1} \tag{3.2}$$

où on prend un moment γ *qui est souvent* 0.9, *et* $v_0 = 0$.

À chaque ètape, le paramètre w *est renouvelé de manière suivante :*

$$w_t = w_{t-1} - v_t \tag{3.3}$$

Pour cet algorithme, la défaillance du taux d'apprentissage sert à faire le modèle rappocher de la solution optimale. De plus, on remarque que la modification des paramètres a tenu compte du gradient à l'instant précédent. Donc dans ce cas, le changement de ces paramètres devient plus « lisse », ce qui est différent au SGD général où il existe plus d'oscillations des paramètres au cours de l'entraînement du modèle. Par conséquent, la convergence de ce modèle devient plus rapide.

Pour la réalisation de cet algorithme, il faut ajouter un dictionnaire supplémentaire comme argument qui contient les valeurs de v *mentionnées au dessus. Pout un étape* t, *on renouvelle le taux d'apprentissage* η_t *qui diminue, puis on calcule la valeur* v_t *et on renouvelle les paramètres. Dans ce projet, ce modèle est réalisé dans le document « 3_Model_sgdmd_955.ipynb » où l'algorithme SGD avec le moment et la défaillance du taux d'apprentissage est réalisé comme une fonction « sgd_md ».*

On fait fonctionner ce modèle. Le modèle atteint sa precision optimale pendant 2 époques, avec la précision 0.954 *sur l'ensemble de validation. En testant sur l'ensemble de test, on obtient une précision* 0.955. *On retrouve bien que ce modèle converge plus rapidement que le modèle précédent avec l'algorithme SGD général. Cependant, la précision obtenue par ce modèle n'est pas assez satisfaisante.*

On fait une modification de ce modèle en touillant les données d'entraînement pour chaque époque. Cette nouvelle version est réalisée dans le document « 4_Model_sgdmd_shuffle_956.ipynb ». Sur l'ensemble de test, on obtient une precision 0.956. *On n'observe pas une amélioration évidente. En fait, ce modèle converge très rapidement avec seulement 2 époques, donc le touillage de données n'ont pas d'influence important pour le résultat.*

Par conséquent, même si ce modèle converge plus rapidement, il est probable que ce modèle atteint une solution optimale locale du problème parce que l'on a pas pris le gradient actuel comme un pas mais on a limité la direction du renouvellement des paramètres. Donc on constate que cet algorithme peut être plus adapté au cas où il y a plus de paramètres dans le NN. Dans ce cas, il faut plus d'époques pour entraîner le modèle et la vitesse de convergence peut être important. En faisant le touillage à chaque époque, il est possible que l'on obtienne un meilleur résultat.

Pour savoir plus de détails de ces deux modèles ainsi que les résultats précis obtenus, vous pouvez consulter les documents « 3_Model_sgdmd_955.ipynb » et « 4_Model_sgdmd_shuffle_956.ipynb » où ces deux modèles sont réalisés avec les dossiers « 200613183131 » et « 200614145056 » où les paramètres bien entraînés sont conservés.

3.2.3 Modèle 3 - AdaGrad

Dans ce modèle, on utilise AdaGrad comme l'algorithme d'optimisation. Pour la réalisation de AdaGrad, il faut prendre comme argument un dictionnaire contenant tous les paramètres dans le CNN, ainsi qu'un dictionnaire contenant tous les gradients de ces paramètres. Dans cet algorithme, il faut traverser les paramètres en les renouvelant. La relation entre les nouveaux paramètres et leur anciennes valeurs est donnée dans la partie 2.3.5.1. Il faut définir ϵ *(on utilise e pour le représenter) en avance.*

Dans ce projet, ce modèle est réalisé dans le document « 5_Model_adagrad_971.ipynb » où l'algorithme AdaGrad est réalisé comme une fonction « AdaGrad ».

On prend le résultat après 11 époques. Dans ce contexte, la précision sur l'ensemble de validation est 0.968. *Grâce aux paramètres sauvegardés dans le dossier « 200613003817 », on peut tester le modèle sur l'ensemble de test, et on obtient la précision* 0.971 *qui est un peu meilleur que cell sur l'ensemble de validation.*

Comme le taux d'apprentissage s'adapter en fonction des paramètres, il diminue après chaque propagation arrière. On peut voir que ce modèle converge plus rapidement que le modèle de SGD. Mais il semble que ce modèle atteint une solution optimale locale du problème et on

obtient une précision moindre que le modèle de SGD.

Pour savoir plus de détails de ce modèle ainsi que les résultats précis obtenus, vous pouvez consulter le document « 5_Model_adagrad_971.ipynb » où ce modèle est réalisé avec le dossier « 200613003817 » où les paramètres bien entraînés sont conservés.

3.2.4 Modèle 4 - RMSProp

Dans ce modèle, on utilise RMSProp comme l'algorithme d'optimisation. Pour la réalisation de RMSProp, il faut prendre comme argument un dictionnaire contenant tous les paramètres dans le CNN, ainsi qu'un dictionnaire contenant tous les gradients de ces paramètres. Dans cet algorithme, il faut traverser les paramètres en les renouvelant. La relation entre les nouveaux paramètres et leur anciennes valeurs est donnée dans la partie 2.3.5.2. Il faut définir γ (on utilise r pour le représenter) et ϵ (on utilise e pour le représenter) en avance.

Dans ce projet, ce modèle est réalisé dans le document « 6_Model_rmsprop_977.ipynb » où l'algorithme RMSProp est réalisé comme une fonction « RMSProp ».

On prend le résultat après 10 époques. Dans ce contexte, la meilleure précision sur l'ensemble de validation est 0.978 *à la cinquième époque. Grâce aux paramètres sauvegardés dans le dossier « 200614141913 », on peut tester le modèle sur l'ensemble de test, et on obtient la presque même précision* 0.977*.*

Comme le taux d'apprentissage s'adapter en fonction des paramètres, il diminue après chaque propagation arrière. On peut voir que ce modèle converge plus rapidement que le modèle de SGD et il semble que ce modèle atteint une solution optimale locale du problème. On obtient une précision moindre que le modèle de SGD mais meilleure que le modèle de AdaGrad grâce à la introduction de γ.

Pour savoir plus de détails de ce modèle ainsi que les résultats précis obtenus, vous pouvez consulter le document « 6_Model_rmsprop_977.ipynb » où ce modèle est réalisé avec le dossier « 200614141913 » où les paramètres bien entraînés sont conservés.

3.2.5 Modèle 5 - SGD avec 2 couches de convolution

On retourne au modèle avec l'algorithme SGD. Cette fois on ajoute une couche de convolution au Modèle 1. C'est-à-dire, on répète les couches 1, 2, 3 entre la couche 3 et la couche 4 montrées dans la Figure 3.1. Le reste de ce modèle est le même que le Modèle 1. On fait fonctionner

le modèle, la vitesse du fonctionnement de ce modèle est raisonnablement plus lente que celle du Modèle 1.

Après 8 époques, on trouve la précision sur l'ensemble de validation 0.985 *et celle sur l'ensemble de test* 0.986. *Dans le meilleur cas, la précision peut atteindre* 0.988. *Ce résultat est assez satisfaisant, ce qui est raisonnable car on a plus de paramètres à entraîner, et donc le modèle peut apprendre plus de caractéristiques des données. Limité par le fonctionnement du Numpy et par la performance de nos ordinateurs, la plupart de nos modèle possèdent selement 1 couche de convolution. Si on construit notre modèle en utilisant TensorFlow ou PyTorch, on peut penser à augmenter le nombre de couches de convolution et le nombre de filtres afin d'avoir un modèle plus fort.*

Pour savoir plus de détails de ce modèle ainsi que les résultats précis obtenus, vous pouvez consulter le document « 2_Model_sgd_2conv_986.ipynb » où ce modèle est réalisé avec le dossier « 200612115604 » où les paramètres bien entraînés sont conservés.

4. Conclusion

Dans ce projet, on a analysé mathématiquement tous les principes de CNN et on a construit des modèles de CNN à partir de Numpy sans l'aide de paquets Python supplémentaires. On a démontré par notre similation la faisabilité de nos modèles. En utilisant de différents algorithmes d'optimisation, on a aussi discuté et commenté les résultats de ces modèles ainsi que leur performances. En général, on trouve que le modèle CNN et l'algorithme SGD atteint bien notre but de la reconnaissance des écritures manuscrites, qui donne une précision plus de 0.98*.*

Théoriquement, la convergence de la méthode de GD existe bien dans les conditions restreintes. Mais en réalité, à cause de l'existence de couches de convolution, la convexité de la fonction du coût ne peut pas être vérifiée. De plus, le fonctionnement de SGD montre une convergence en espérance. Par conséquent, il est peu probable que notre modèle trouve exactement la solution optimale globale basée sur l'ensenble de données, ce qui explique le fait que l'on a jamais obtenu une précision 100%*.*

Pour l'aspect informatif, comme on a construit tout nos modèles à partir de Numpy, il est normal que ces modèles fonctionnent très lentement. Par conséquent, on a choisi principalement un modèle simple qui contient une seule couche de onvolution. On est content du fait que la méthode de SGD fonctionnent bien pour ce modèle en donnant une précsion satisfaisante. Cependant, pour les autres algorithmes d'optimisation, les résultats représentés sont moins bons. Il est possible que ces algorithmes soient plus forts pour les problèmes compliqués, par exemple, un CNN contenant plus de couches et plus de paramètres.

Finalement, on a bien réalisé dans ce projet le fonctionnement pour reconnaître les écritures manuscrites en construisant notre modèle de CNN et en essayant et comparant les différents algorithmes d'optimisation. Pour les autres problèmes de la reconnaissance d'images ainsi que les autres modèles ou les autre algorithmes d'optimisation, il nous reste à étudier dans le futur.

5. Bibliographie

Benlahmar, (2018), Les réseaux de neurones convolutifs, https://datasciencetoday.net/index.php/en-us/deep-learning/173-les-reseaux-de-neurones-convolutifs.

R. Lambert, (2019), Le Réseau de Neurones Convolutifs, http://penseeartificielle.fr/focus-reseau-neurones-convolutifs/.

J. Benesty, Algorithme du gradientstochastique (least-mean-square –LMS), https://studylibfr.com/doc/2676479/algorithme-du-gradient-stochastique--least-mean.

S. Ruder, (2016), An overview of gradient descent optimization algorithms, https://ruder.io/optimizing-gradient-descent/.

O. Milman, (2018), Gradient de descente vs Adagrad vs Momentum dans le TensorFlow, https://webdevdesigner.com/q/gradient-descent-vs-adagrad-vs-momentum-in-tensorflow-895

Jefkine, (2016), Backpropagation in Convolutional Neural Networks, https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

L. Bottou, FE. Curtis and J. Nocedal, (2018), Optimization Methods for Large-Scale Machine Learning.

6. Contributions des membres du groupe

Andy :
Réalisation avec Python les fonctions fondamentales utilisées dans le modèle CNN (les fonctions de propagation et rétropropagation pour les différentes couches, les fonctions d'activation, les fonctions du coût, etc.). Fonctionnement des modèles et analyses des résultats de simulation. Analyse de la convergence, la complexité et le choix des paramètres pour SGD.

Noah :
Réalisation avec Python les parties "Optimizer" (les fonctions sgd, AdaGrad, et RMSProp), "Load data", " Preprocess data", "Construct model", "Method of evaluation", "Save model and load model", "Run model" et "Test model". Fonctionnement des modèles et analyses des résultats de simulation.

Soline :
Apprentissage de la connaissance de CNN. Construction des fonctions de propagation et rétropropagation pour les différentes couches, les fonctions d'activation, les fonctions du coût, comparaison entre l'algorithme SGD avec AdaGrad, RMSProp etc. Rédaction du projet.