

Date : 4 février 2018
Auteur : Vincent CHAMBRIN
Fabien Rouillon

Projet Mathématiques et Applications

Deep networks for character recognition

Sommaire

Le but de ce projet est d'étudier l'utilisation de réseaux de neurones profonds pour la reconnaissance de caractères manuscrits. Différentes méthodes de classifications seront utilisées et comparées à un réseau de neurones simple et à des algorithmes n'utilisant pas de réseaux de neurones. Le langage Python sera utilisé pour l'écriture des programmes, ce dernier fournissant bon nombre de bibliothèques de *machine learning*. Pour mieux comprendre les algorithmes et idées mis en jeux, une brève introduction aux principes des réseaux de neurones sera présentée.

Mots-clés : réseaux de neurones, *machine learning*, *deep networks*, reconnaissance de caractères, Python.

Table des matières

Sommaire	ii
Table des matières	iii
1 Introduction	1
2 Les réseaux de neurones	2
2.1 Le modèle du perceptron	2
2.2 Structure des réseaux de neurones	6
2.3 Apprentissage par la descente de gradient	8
2.4 L'algorithme de rétro-propagation	8
3 Reconnaissance via un réseau de neurones simple	12
3.1 Construction du réseau de neurones	12
3.2 Apprentissage	12
3.3 Résultats	12
4 Reconnaissance basée sur l'extraction de <i>features</i>	13
4.1 Binarisation de l'image	13
4.2 Densités de pixels coloriés	13
4.3 Nombre de croisements (<i>crossings</i>)	13
4.4 Histogramme des projections	13
4.5 Moments	14
4.6 Transformée de Fourier du contour	14
4.7 Transformée de Fourier de l'image	14
5 Reconnaissance basée sur les réseaux convolutionnels	15
6 Comparaison des résultats	16

1 Introduction

La reconnaissance de caractères, ou plus généralement de textes manuscrits, est un domaine en pleine expansion. De nombreux géants de l'informatique comme Google, Apple ou Microsoft se battent pour offrir à leurs utilisateurs (et probablement à d'autres fins) le meilleur système de reconnaissance de texte. Les utilisations sont nombreuses :

- application de prises de notes ;
- saisie de formulaires ;
- saisie d'une adresse pour le système GPS de certaines voitures ;
- reconnaissance d'adresse postale sur les enveloppes ;
- reconnaissance d'un montant sur les chèques.

Dans le cadre de ce projet, on se contentera de tenter de reconnaître des chiffres. On utilisera pour cela le jeu de données MNIST, qui est un jeu de données contenant 60000 images d'apprentissage et 10000 images de test. Ces images, représentant des chiffres écrits à la main, sont en niveaux de gris et font 28 pixels par 28 pixels. Elles sont issues de deux jeux de données collectés par le NIST (United States' National Institute of Standards and Technology). A chaque image est associé un label (i.e. le chiffre représenté). Ce jeu de données est pratiquement devenu un standard dans le monde de la reconnaissance de caractères.

Ce rapport est découpé en plusieurs parties. Dans un premier temps, une introduction sur les réseaux de neurones permettra de se familiariser avec ces derniers. On sera alors en mesure d'écrire nous-même les algorithmes pour les réseaux les plus simples et nous pourrons construire un premier réseau permettant de répondre au problème de classification que l'on se pose.

Ensuite, nous pourrons mettre en oeuvre des techniques de '*machine learning*' plus poussés et plus adaptés à notre domaine. On commencera pour cela par étudier certaines techniques propres au problème de reconnaissance de caractères (l'extraction de *features*). Enfin, nous nous pencherons sur l'utilisation de réseaux de neurones convolutionnels, ces derniers étant particulièrement adaptés aux problèmes de vision par ordinateur et de traitement d'image.

2 Les réseaux de neurones

2.1 Le modèle du perceptron

Définition 1 (Fonction de Heaviside). *On appelle fonction de Heaviside la fonction H définie de \mathbb{R} dans $\{0, 1\}$ par :*

$$H(x) = \begin{cases} 1, & \text{si } x \geq 0 \\ 0, & \text{sinon.} \end{cases}$$

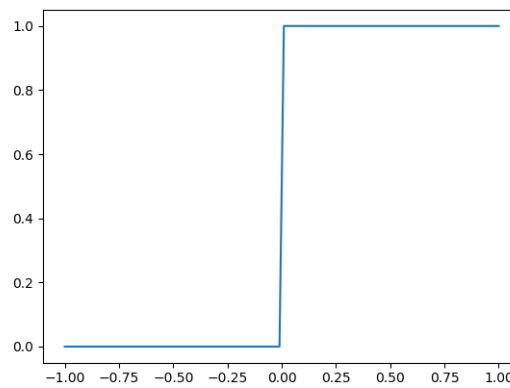


FIGURE 1 – La fonction échelon unité, ou fonction de Heaviside.

Le but du perceptron est de modéliser le comportement du neurone biologique. Ce dernier est stimulé par des signaux qui lui parviennent par ses dendrites et, si la stimulation est suffisamment importante, renvoie un signal à d'autres neurones au travers de son axone.

En notant x_1, x_2, \dots, x_n ces stimulations, qui sont donc les entrées du perceptron, et w_1, w_2, \dots, w_n les pondérations associées à ces entrées, on peut simplement exprimer le comportement du perceptron par :

$$\text{sortie} = \begin{cases} 1 & \text{si } w \cdot x \geq \text{seuil} \\ 0 & \text{si } w \cdot x < \text{seuil.} \end{cases}$$

Par la suite, on préférera, plutôt que de considérer un seuil, considérer une quantité b que l'on appellera biais définie par $b = -\text{seuil}$, et on écrira :

$$\text{sortie} = H(w \cdot x + b)$$

Les perceptrons peuvent être utilisés pour coder des fonctions logiques. Par exemple, en prenant $w = (1, 1)$ et $b = -2$, notre perceptron code un ET logique (avec la convention 0 pour FAUX et 1 pour VRAI).

De même, en prenant $w = (-2, -2)$ et $b = 3$, on code la fonction NAND (négation du ET logique). Pour s'en convaincre, on peut dresser le tableau des sorties en fonctions de toutes les entrées possibles comme cela est fait dans la table 1.

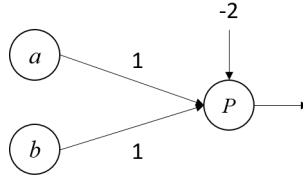


FIGURE 2 – Fonction ET logique.

x_1	x_2	$wx + b$	Sortie
0	0	3	1
0	1	1	1
1	0	1	1
0	0	-1	0

TABLE 1 – Table de vérité du perceptron.

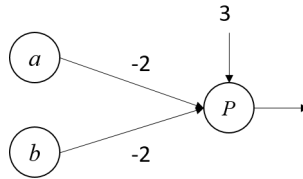


FIGURE 3 – Fonction NAND.

Comme la fonction NAND permet de coder n'importe quelle fonction logique, on en déduit qu'en utilisant la sortie de perceptrons comme entrées d'autres perceptrons, on peut construire n'importe quelle fonction logique.

Exercice : Coder la fonction OU logique pour n entrées.

Cette idée d'empiler des couches de perceptrons pour coder une fonction logique conduit naturellement à l'idée que l'on pourrait se faire d'un réseau de neurones.

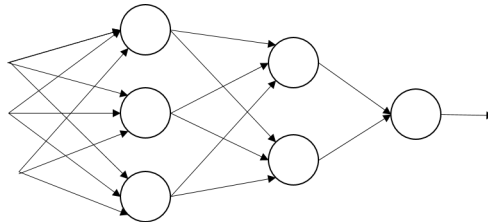


FIGURE 4 – Un réseau de perceptrons.

Jusqu'à présent, nous nous sommes restreint à des entrées binaires (0 ou 1) codant des valeurs logiques. Ceci n'est pas nécessaire : les entrées d'un perceptron peuvent très bien être des réels. Plaçons nous, pour les exemples qui vont suivre, dans \mathbb{R}^2 . Pour chaque point (x, y) du plan, on donnera x comme première entrée d'un perceptron et y comme seconde entrée. Le perceptron ayant pour paramètre $w = (\alpha_x, \alpha_y)$ et b va alors séparer l'espace en deux plans par la droite d'équation $\alpha_y y + \alpha_x x + b = 0$. Les points tels que $\alpha_y y + \alpha_x x + b \geq 0$ auront pour sortie 1, et les autres auront pour sortie 0.

Ceci est illustré par la figure 5 dans laquelle la partie verte correspond à une sortie positive du perceptron

dès lors que $y \geq x$.

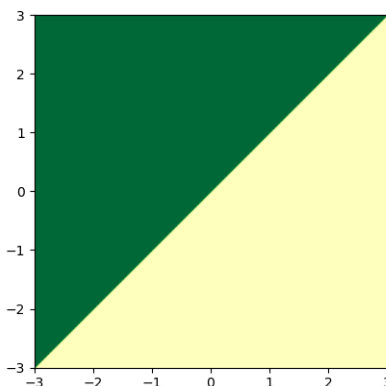


FIGURE 5 – Découpage du plan en deux par un perceptron.

En utilisant plusieurs perceptron, on va pouvoir effectuer plusieurs sous-découpage de l'espace, et en utilisant un ET logique dans une dernière couche on va pouvoir récupérer la partie de l'espace correspondant à l'intersection de toutes les parties correspondant aux sorties positives des perceptrons de la couche précédente.

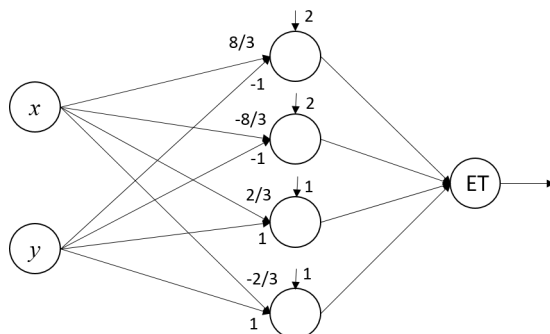


FIGURE 6 – Un premier réseau de perceptrons.

Cette approche a ses limites car elle ne permet d'obtenir qu'une partie convexe du plan. Pour obtenir des parties plus complexes (e.g. non convexes, ou non fortement connexes), on peut utiliser d'autres fonctions logiques, comme le OU.

Il n'est pas difficile d'imaginer que l'on puisse étendre ce que l'on vient de faire en 2 dimensions à un nombre plus grand de dimensions ; et avec un non plus une seule sortie mais plusieurs (correspondant par exemple chacune à une classe, à un chiffre que l'on souhaite reconnaître). On pourrait donc en théorie construire un réseau de neurones avec des perceptrons qui serait capable de reconnaître des chiffres. L'un des problèmes de cette approche est qu'il faudrait déjà connaître les coefficients (ou paramètres) de l'ensemble des perceptrons du réseau. On pourrait alors avoir envie d'écrire un algorithme permettant de construire un tel réseau. On peut imaginer plusieurs principes pour cet algorithme :

1. tester tous les coefficients possibles pour les perceptrons ;
2. partir d'une configuration aléatoire et modifier petit à petit les coefficients jusqu'à obtenir un résultat satisfaisant.

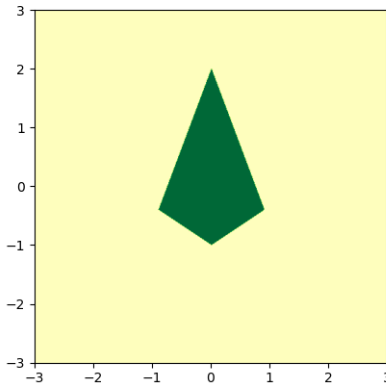


FIGURE 7 – Découpage du plan par un réseau de perceptrons.

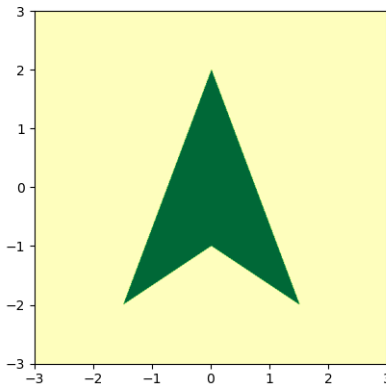


FIGURE 8 – Un découpage non convexe du plan avec un réseau de perceptrons à deux couches.

La première idée est tout simplement irréalisable car il y a une infinité de valeurs possibles pour chaque coefficient du réseau. On pourrait penser à prendre une approche probabiliste en testant un certains nombres de réseau et en espérant tomber sur un bon ; ce qui paraît néanmoins très improbable ! La deuxième idée n'est également pas envisageable car elle souffre d'un problème difficile à gérer. La discontinuité de la fonction de Heaviside fait qu'un petit changement de coefficient peut faire passer la sortie d'un neurone de 0 à 1 ce qui peut correspondre à un changement significatif de l'entrée des neurones de la couche suivante. Il est donc compliqué d'évaluer l'impact d'un changement à une couche donnée sur les couches qui suivent.

L'idée qui va être exploré dans les paragraphes suivants est d'utiliser une autre fonction, appelée fonction d'activation, que la fonction échelon d'Heaviside ; et de faire en sorte que cette fonction d'activation soit dérivable (et donc continue) pour pouvoir utiliser des résultats d'analyse et pouvoir ainsi estimer l'impact du changement d'un coefficient. On sera alors en mesure de modifier un réseau généré aléatoirement pour le faire résoudre notre problème de classification (on parlera d'apprentissage).

2.2 Structure des réseaux de neurones

Définition 2 (Fonction sigmoïde). *On appelle fonction sigmoïde la fonction définie de \mathbb{R} dans $[0, 1]$ par :*

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

On notera que $\lim_{x \rightarrow +\infty} \sigma(x) = 1$ et $\lim_{x \rightarrow -\infty} \sigma(x) = 0$. Ainsi, asymptotiquement, la fonction sigmoïde a le même comportement que la fonction de Heaviside.

Entre les deux, on a un comportement qui diffère fortement du perceptron avec notamment la non discontinuité en 0 et $\sigma(0) = \frac{1}{2}$. Dans le cadre de notre utilisation pour la classification on aura tendance à considérer que la sortie du neurone est à VRAI dès qu'elle sera supérieure à une certaine valeur (typiquement $\frac{1}{2}$).

On a également quelques propriétés intéressantes de symétrie. En effet,

$$\sigma'(x) = \frac{e^{-x}}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x))$$

$$\sigma'(x) - \sigma'(-x) = \frac{e^{-x}(e^{2x} + 2e^x + 1) - e^x(e^{-2x} + 2e^{-x} + 1)}{(1 + e^{-x})^2(1 + e^x)^2} = 0$$

ce qui signifie que la sigmoïde admet pour centre de symétrie le point $(0, \frac{1}{2})$.

Cette fonction a de plus l'avantage d'être dérivable en tout point (et de dérivée non nulle), ce qui nous servira plus tard dans le cadre de l'apprentissage.

Définition 3 (Modélisation d'un neurone). *En notant b le biais du neurone, w son vecteur des poids et σ sa fonction d'activation, on peut écrire la sortie a d'un neurone en fonction de son entrée x de la manière suivante :*

$$a = \sigma(w \cdot x + b)$$

Définition 4 (Vectorisation d'une fonction). *Soit f une fonction de $\mathbb{R} \mapsto \mathbb{R}$. On pose pour tout vecteur x de \mathbb{R}^n :*

$$f(x) = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}$$

On dit que l'on vectorise la fonction f .

Achtung! Par abus de notation, on utilise encore le nom de f pour désigner cette fonction, mais il s'agit bien d'une fonction différente puisqu'elle est à valeurs de \mathbb{R}^n dans \mathbb{R}^n . Cet abus nous est néanmoins utile pour définir simplement les couches d'un réseau de neurones.

On considère dans les définitions suivantes une couche d'un réseau de neurones constituée de n neurones. On note w_i le vecteur des poids associé à chaque neurone. On a alors la définition suivante :

Définition 5 (Matrice des poids). *En notant w_i les vecteurs des poids de n neurones, on définit par blocs la matrice w des poids :*

$$w = \begin{pmatrix} w_1^T \\ w_2^T \\ \vdots \\ w_n^T \end{pmatrix}$$

On définit de manière analogue un vecteur des biais.

Définition 6 (Vecteur des biais). *En notant b_i les biais de n neurones, on définit le vecteur des biais d'une couche d'un réseau de neurones par :*

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

On a alors la proposition suivante :

Proposition 2.2.1 (Ecriture matricielle d'une couche). *Soit une couche d'un réseau constituée de n neurones et ayant pour matrice des poids w et pour vecteur des biais b . Si les neurones ont tous la même fonction d'activation σ , on peut écrire les sorties des neurones sous la forme d'un vecteur a vérifiant :*

$$a = \sigma(w \cdot x + b)$$

où les a_i correspondent à la sortie du neurone i .

On notera que dans cette proposition, on utilise la version vectorisée de σ . Une écriture équivalente est :

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sigma(w_1 \cdot x + b_1) \\ \sigma(w_2 \cdot x + b_2) \\ \vdots \\ \sigma(w_n \cdot x + b_n) \end{pmatrix}$$

En fait, cette écriture est identique à celle d'un unique neurone ! Elle est également intéressante d'un point de vue computationnel : au lieu de calculer une à une les sorties des neurones, il est plus efficace de faire un calcul matriciel (les bibliothèques étant en générales optimisées pour cela). Cela permet également d'avoir un code plus simple (avec notamment moins de boucles).

Cette écriture permet également d'écrire de manière très condensée un réseau de neurones à plusieurs couches. En notant w^1, w^2, \dots, w^L les matrices de poids de L couches de neurones, et b^1, b^2, \dots, b^L les vecteurs des biais associés, et en supposant que tous les neurones ont pour fonction d'activation σ , on

$$a = \sigma(b^L + w^L \cdot \sigma(b^{L-1} + w^{L-1} \cdot \sigma(\dots))) \quad (2.1)$$

Pour avoir une écriture plus lisible, on introduit la notion suivante qui permet de parler de la sortie pour toute couche :

Définition 7 (Activation d'un neurone). *On appellera activation du neurone j de la couche l la quantité a_j^l définie par :*

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$$

Il s'agit simplement de la sortie de ce neurone.

On peut alors réécrire l'équation 2.1 de la manière suivante, la sortie du réseau étant le vecteur a^L .

$$\begin{cases} a^l = \sigma(w^l \cdot a^{l-1} + b^l) \\ a^0 = x \end{cases}$$

On constate donc que l'on peut très facilement calculer la sortie d'un réseau de neurones par un algorithme itératif.

Autre remarque importante : il n'est pas nécessaire que les $(w^l)_{l \in \llbracket 1, L \rrbracket}$ aient les mêmes dimensions. Autrement dit, il peut y avoir un nombre différent de neurones dans chaque couche ; le seul impératif est que les dimensions d'une couche à la suivante soient cohérentes.

En reprenant les notations précédentes, on a que w_{jk}^l est le poids de la connexion reliant le $k^{\text{ème}}$ neurone de la couche $(l-1)$ au $j^{\text{ème}}$ neurone de la couche l . De la même manière, b_j^l est le biais du $j^{\text{ème}}$ neurone de la couche l .

2.3 Apprentissage par la descente de gradient

On considère dans cette partie un réseau de neurones à L couches, et on note encore w^1, w^2, \dots, w^L les matrices des poids et b^1, \dots, b^L les vecteurs des biais. On cherche dans cette partie à améliorer de manière itérative la valeurs des coefficients du réseau pour améliorer les prédictions. On introduit pour cela une fonction de coûts, croissante de l'erreur, et on se ramène à un problème d'optimisation.

Définition 8 (Fonction de coût quadratique). *Soit X un ensemble d'apprentissage de cardinal n . On note $a^L(x)$ la sortie du réseau de neurones pour l'entrée $x \in X$ et y_x la sortie désirée. On définit la fonction de coût par :*

$$C(w, b) = \frac{1}{2n} \sum_x \|y_x - a^L(x)\|^2$$

On parle aussi de MSE pour *mean square error*.

Par la suite, et pour alléger les notations, on écrira souvent a^L et y au lieu de $a^L(x)$ et y_x .

Cette fonction a le bon goût d'être convexe. Ainsi, on peut la minimiser en annulant sa dérivée. Cependant cette fonction dépend de tous les w^l et b^l ainsi que de la fonction σ , le calcul de la dérivée et de son point d'annulation n'est pas facile. C'est pourquoi on va opter pour un algorithme itératif qui, partant d'un point quelconque (i.e. un couple (w, b) donné), va progressivement s'approcher de la solution.

L'un des algorithmes les plus simples est celui de la descente de gradient. Partant d'un point $v = (w, b)$, on souhaite trouver Δv telle que $C(v + \Delta v) \leq C(v)$. Cet algorithme propose de prendre $\Delta v = -\eta \nabla C(v)$ avec η suffisamment petit pour que l'on puisse s'assurer de diminuer le coût C et suffisamment grand pour que les effets soient significatifs.

L'algorithme peut donc se résumer sous la forme suivante :

1. étant donné $v = (w, b)$, calculer $\nabla C(v)$;
2. calculer $v' = v - \eta \nabla C(v)$;
3. recommencer l'étape 1 avec v' .

On itère jusqu'à ce que le coût soit proche de son minimum.

En pratique on utilisera une variante stochastique de l'algorithme. En effet,

$$C = \frac{1}{n} \sum_x C_x \quad \text{avec} \quad C_x = \frac{1}{2} \|y - a^L\|^2$$

sera d'autant plus long à calculer que n (la taille de l'échantillon d'apprentissage) est grand. C'est pourquoi on tirera $m < n$ entrées au hasard et on se contentera de calculer le coût et son gradient sur ce sous-échantillon.

2.4 L'algorithme de rétro-propagation

L'algorithme de descente de gradient évoqué précédemment est en principe très simple. Cependant, le calcul de $\nabla C(v)$ est loin d'être simple. Cette quantité dépend en effet de tous les w_{jk}^l et les b_j^l . On présente dans cette partie une manière simple de calculer cette quantité via l'algorithme de rétro-propagation. Il est cependant nécessaire d'introduire d'abord quelques quantités utiles.

Définition 9 (Entrées pondérées d'un neurone). *On notera z_j^l les entrées pondérées du neurone j de la couche l .*

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

De manière évidente, on a $a_j^l = \sigma(z_j^l)$.

Comme on la vu dans la partie précédente, le coût C s'écrit comme somme de coûts unitaire :

$$C = \frac{1}{n} \sum_x C_x$$

Dans cette partie, on s'intéressa seulement au calcul de la quantité C_x pour un x donné. La vrai coût C et ∇C se calculant simplement en faisant la somme. On se fixe donc un x et, pour simplifier les notations, on notera $y = y_x$ la sortie voulu et $C = C_x$.

L'idée de l'algorithme de rétro-propagation est de partir d'une quantité que l'on peut facilement calculer à partir de la sortie du réseau (que l'on obtient en faisant un *feedforward*, c'est à dire en parcourant le réseau depuis la première couche jusqu'à la dernière) et de revenir en arrière (*retropopagation*) pour attribuer à chaque coefficient du réseau une part de l'erreur.

Notons a^L la sortie du réseau, on a :

$$C = \frac{1}{2} \|y - a^L\|^2 = \frac{1}{2} \sum_j (y_j - a_j^L)^2$$

On obtient donc très facilement le résultat suivant.

$$\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$$

Que l'on écrira sous forme vectorielle.

$$\nabla_a C = a^L - y$$

La principe de l'algorithme de rétro-propagation va être de partir de cette quantité et de parcourir les couches du réseau à l'envers pour calculer les dérivées partielles $\frac{\partial C}{\partial w_{jk}^l}$ et $\frac{\partial C}{\partial b_j^l}$.

Définition 10 (Erreur d'un neurone). *On définit l'erreur au neurone j de la couche l par :*

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Cette définition est posée de manière un peu arbitraire et permet simplement de simplifier l'écriture des calculs que nous ferons plus tard. Une autre définition possible aurait été de prendre l'erreur comme étant égale à $\frac{\partial C}{\partial a_j^l}$. Cela ne change pas grand chose car on a la relation suivante.

$$\frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \frac{\partial \sigma(z_j^l)}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l)$$

Définition 11 (Produit de Hadamard). *Soit S et T deux matrices (ou vecteurs) de même dimension, on appelle produit de Hadamard de S et T , noté $S \odot T$ la matrice de même dimension que T et donc les composantes sont les produits éléments par éléments des coefficients de S et T , i.e.*

$$(S \odot T)_{ij} = S_{ij} \times T_{ij}$$

Proposition 2.4.1 (Equations de la rétro-propagation). *On a les relations suivantes :
Erreur au niveau de la dernière couche :*

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (2.2)$$

Erreur au niveau de la couche l :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (2.3)$$

Dérivées partielles par rapport aux biais :

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (2.4)$$

Dérivées partielles par rapport aux poids :

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (2.5)$$

Ces équations nous donnent une méthode simple pour calculer les dérivées partielles par rapport aux coefficients du réseau et permettent ainsi d'appliquer la méthode de la descente de gradient.

Preuve de l'équation 2.2. Il suffit de reprendre ce que l'on a fait précédemment.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

Soit sous forme vectorielle,

$$\delta^L = \frac{\partial C}{\partial a^L} \odot \sigma'(z^L)$$

quantité que l'on écrit $\nabla_a C \odot \sigma'(z^L)$. □

Le résultat 2.3 est certainement le plus intéressant car c'est lui qui permet de remonter le réseau pour attribuer les erreurs aux différentes couches. Il dit grossièrement que pour retrouver l'erreur à la couche l , il suffit d'appliquer la transposée de la matrice de poids de la couche $l+1$ à l'erreur sur cette même couche et de multiplier par la dérivée de la sigmoïde.

Preuve de l'équation 2.3. On applique un raisonnement similaire à la preuve précédente.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \quad (2.6)$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \quad (2.7)$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \quad (2.8)$$

On se remémore ensuite l'expression de z_k^{l+1} :

$$z_k^{l+1} = \sum_i w_{ki}^{l+1} a_i^l + b_k^{l+1} = \sum_i w_{ki}^{l+1} \sigma(z_i^l) + b_k^{l+1}$$

Ce qui nous permet d'obtenir

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

On obtient donc le résultat voulu

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

□

Les deux derniers résultats servent à relier les quantités calculées aux quantités voulues.

Preuve de l'équation 2.4.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l}$$

Or on sait que $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ donc la dérivée deuxième vaut 1. Au final,

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l$$

□

Preuve de l'équation 2.5. $z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j^l$ donc $\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$. Ainsi,

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

□

3 Reconnaissance via un réseau de neurones simple

Dans ce chapitre, on se propose d'utiliser ce que l'on a vu précédemment pour construire un réseau de neurones permettant de répondre à notre problème de classification.

3.1 Construction du réseau de neurones

On se propose de construire un réseau de neurones ayant une seule couche cachée contenant p neurones. Les couches d'entrées et de sorties contiennent respectivement $28 \times 28 = 784$ et 10 neurones.

Chaque neurone de la couche de sortie correspond à une classe (de 0 à 9) et on dira que le réseau classe une entrée dans la classe C_i si la sortie i est la plus élevée.

On rappelle que puisqu'on utilise comme fonction d'activation la sigmoïde, les sorties sont toutes positives.

3.2 Apprentissage

Les images du jeu d'entraînement de MNIST sont des tableaux de 28 par 28 contenant des entiers de 0 à 255. On commence par redimensionner le tableau en un vecteur de taille 784 que l'on divise ensuite par 255 pour obtenir des composantes entre 0 et 1. Cela n'est pas nécessaire dans l'absolue mais donne de meilleurs résultats en pratique. En effet, la sigmoïde faisant intervenir une exponentielle, on aura du mal à calculer numériquement e^{-255} .

Pour les sorties, on converti les différentes étiquettes en des vecteurs de taille 10 dont toutes les composantes sont nulles sauf la composantes correspondant à la classe qui vaut 1.

On initialise les poids et biais de notre réseau par des tirages aléatoires suivant une $\mathcal{N}(0, 1)$.

On découpe la phase d'apprentissage en *epochs*. Comme nous l'avons dit précédemment, on ne cherchera pas à calculer la valeur exacte de la dérivée du coût pour changer nos coefficients mais seulement une approximation de cette dérivée. On va donc découper de manière aléatoire notre jeu d'entraînement en *mini-batch* (i.e. en sous-échantillon) dont l'union fait le jeu complet. Il s'agit donc en quelque sorte d'une partition du jeu de départ créée aléatoirement. On effectue la descente de gradient avec ces *mini-batches*. Lorsque l'on a utilisé tous les *mini-batches*, et donc toutes les données, on dit que l'on a effectué une *epoch*. On peut alors en commencer une autre en effectuant un nouveau découpage aléatoire du jeu d'entraînement.

3.3 Résultats

En prenant $p = 30$ neurones sur la couche cachée, et en prenant des *mini-batches* de taille 10, on arrive en une trentaine d'*epochs* et avec un taux d'apprentissage $\eta = 3,0$ à un taux de reconnaissance d'environ 95% sur le jeu de test.

4 Reconnaissance basée sur l'extraction de *features*

Si l'utilisation des données brutes comme entrée des réseaux de neurones permet déjà d'obtenir des résultats encourageant, il est possible de faire mieux en effectuant un pré-traitement des données. Il s'agit d'extraire des données brutes (i.e. les pixels de l'image) des caractéristiques (ou *features* en anglais) qui permettent de décrire l'image dans un format plus adapté pour une machine.

Ces *features* peuvent être regroupés en deux catégories :

1. les *features* statistiques, qui vont s'intéresser à des densité de pixels, des extremums et autres transformées mathématiques ;
2. les *features* structurelles, qui s'intéressent aux traits (strokes), aux courbes, aux nombres de bifurcations, etc. . . , ces dernières sont plus intuitives pour l'humain.

Nous allons dans ce chapitre présenter quelques *features* qui peuvent être utilisées dans le cadre de la reconnaissance de caractères. Puis nous les utiliserons comme entrées de différents algorithmes de *machine learning*.

4.1 Binarisation de l'image

Les images du jeu de données sont en niveaux de gris. C'est à dire que chaque pixel est représenté par un entier entre 0 et 255 (du blanc au noir). Certains algorithmes ne prennent pas en compte le niveau de coloration du pixel et s'intéresse juste au fait que le pixel soit noir ou blanc. Il est donc utile, dans ce cas de binariser l'image (i.e. de passer à la convention 0 pour un pixel ne contenant pas d'encre et 1 pour un pixel en contenant). On choisit donc arbitrairement un seuil à partir duquel on considère que le pixel est colorié.

```
def binarize(img, threshold = 200):
    w = len(img)
    h = len(img[0])
    ret = img.copy()
    for x in range(w):
        for y in range(h):
            ret[y][x] = 1 if img[y][x] >= threshold else 0
    return ret
```

Bien évidemment, le choix du seuil peut avoir un impact sur le calcul des *features* si l'on choisit un seuil très élevé, beaucoup de pixels seront considérés comme vide. A contrario, si le seuil est très faible, on considérera qu'un pixel est colorié dès qu'il y aura un peu d'encre dessus, ce qui peut également poser des problèmes.

4.2 Densités de pixels coloriés

4.3 Nombre de croisements (*crossings*)

On prend deux points à l'extrémité de l'image et on compte le nombre d'alternance entre les groupes de pixels vides et les groupes de pixels contenant de l'encre. Cette méthode n'est à priori pas sensible à l'épaisseur du trait.

4.4 Histogramme des projections

On compte pour chaque ligne (resp. chaque colonne) le nombre de pixels allumés sur la ligne (resp. colonne). En faisant ça sur l'ensemble de l'image, on obtient deux histogrammes. Cette technique peut aussi être utilisé

pour segmenter des lignes et caractères isolés. Cette technique peut être sensible à l'épaisseur du trait. Pour palier ce problème, on peut renormaliser chaque histogramme en divisant chaque valeur par le total.

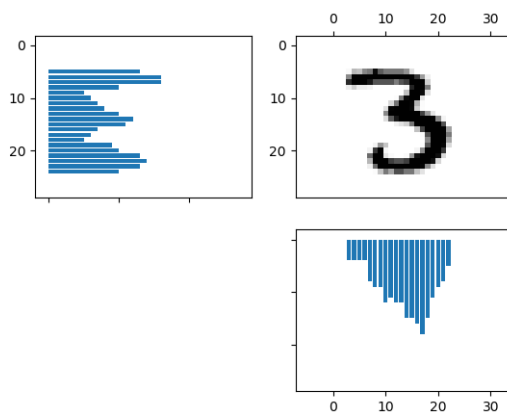


FIGURE 9 – Histogramme des projections du chiffre 3.

4.5 Moments

Dans tout ce qui suit, on note p_{xy} la valeur du pixel (x, y) .

On définit le moment d'ordre $(p + q)$ par :

$$m_{pq} = \sum_x \sum_y x^p y^q p_{xy}$$

En pratique, on préférera utiliser des moments centrés (car invariant par translation de l'image).

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q p_{xy}$$

avec

$$\bar{x} = \frac{m_{10}}{m_{00}} \quad \bar{y} = \frac{m_{01}}{m_{00}}$$

4.6 Transformée de Fourier du contour

4.7 Transformée de Fourier de l'image

5 Reconnaissance basée sur les réseaux convolutionnels

6 Comparaison des résultats

TODO : on pourra dans cette partie comparer les résultats avec des méthodes qui ne sont pas basés sur des réseaux de neurones.