Date : 14 mars 2018
Auteurs : 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

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 chéques.

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, représentants 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 :*

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

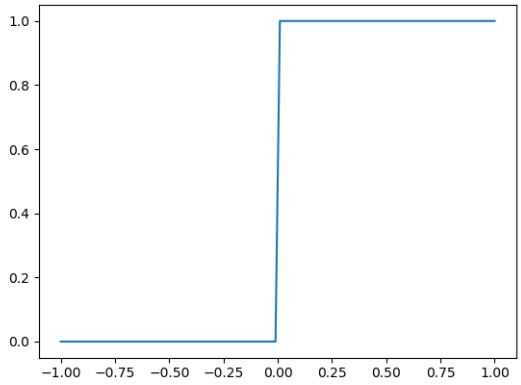

FIGURE 1 – La fonction échelon unité, ou fonction de Heaviside.

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 suffisament importante, renvoie un signal à d'autres neurones au travers de son axone.

En notant $x_1, x_2, \ldots, x_n$ ces stimulations, qui sont donc les entrées du perceptron, et $w_1, w_2, \ldots, w_n$ 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 appelera 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.

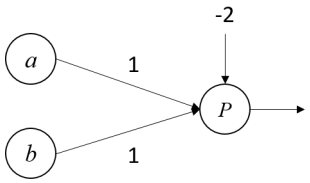

FIGURE 2 – Fonction ET logique.

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.

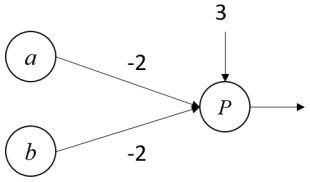

FIGURE 3 – Fonction NAND.

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.

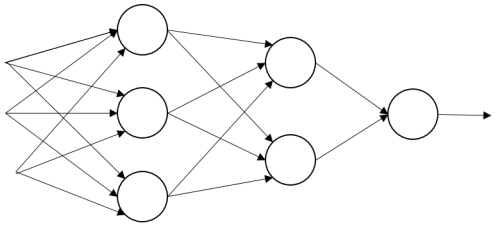

FIGURE 4 – Un réseau de perceptrons.

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 suivres, 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 $y \geq x$.

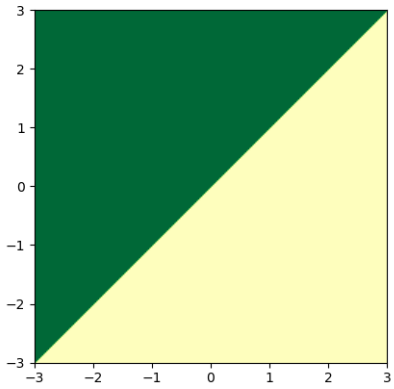

FIGURE 5 – Découpage du plan en deux par un perceptron.

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 récuperer la partie de l'espace correspondant à l'intersection de toutes les parties correspondants aux sorties positives des perceptrons de la couche précédente.

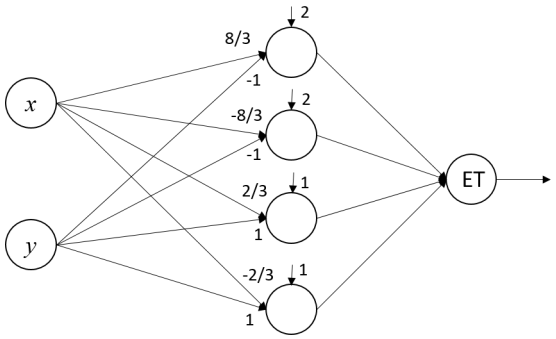

FIGURE 6 – Un premier réseau de perceptrons.

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 reonnaître). 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.

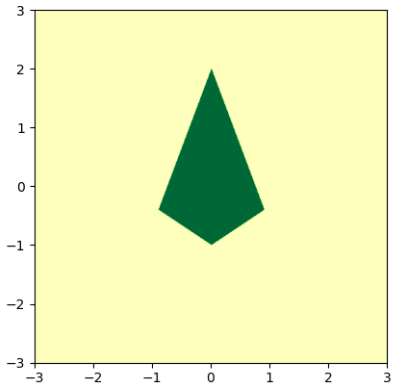

FIGURE 7 – Découpage du plan par un réseau de perceptrons.

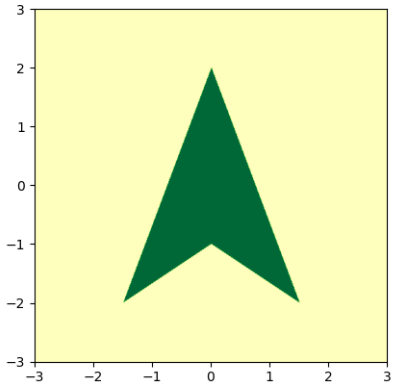

FIGURE 8 – Un découpage non convexe du plan avec un réseau de perceptrons à deux couches.

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

Entre les deux, on à 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 computationnelle : 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 $w^1, w^2, \cdots, w^L$ les matrices de poids de L couches de neurones, et $b^1, b^2, \cdots, b^L$ les vecteurs des biais associés, et en supposant que tous les neurones ont pour fonction d'activation σ, on

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

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 $(w^l)_{l\in[\![1,L]\!]}$ aient les mêmes dimensions. Autrement dit, il peut y avoir un nombre différent de neurones dans chaque couche ; le seule impératif est que les dimensions d'une couche à la suivante soient cohérentes.

En reprenant les notations précédentes, on a que w^l_{jk} 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^l_j 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 $w^1, w^2, \cdots, w^L$ les matrices des poids et $b^1, \cdots, b^L$ 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 :*

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

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 à 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 dans 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 suffisament petit pour que l'on puisse s'assurer de diminuer le coût C et suffisament 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,

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

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^l_{jk} et les b^l_j. 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 :

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

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 ferrons 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 :

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2.2}$$

Erreur au niveau de la couche l :

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

Dérivées partielles par rapport aux biais :

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{2.4}$$

Dérivées partielles par rapport aux poids :

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{2.5}$$

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.

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

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.

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \tag{2.6}$$

$$= \sum_k \frac{\partial C}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} \tag{2.7}$$

$$= \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j}\delta^{l+1}_k \tag{2.8}$$

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

$$z^{l+1}_k = \sum_i w^{l+1}_{ki} a^l_i + b^{l+1}_k = \sum_i w^{l+1}_{ki}\sigma(z^l_i) + b^{l+1}_k$$

Ce qui nous permet d'obtenir

$$\frac{\partial z^{l+1}_k}{\partial z^l_j} = w^{l+1}_{kj}\sigma'(z^l_j)$$

On obtient donc le résultat voulu

$$\delta^l_j = \sum_k w^{l+1}_{kj}\delta^{l+1}_k\sigma'(z^l_j)$$

□

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,

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

□

2.5 Reconnaissance avec un réseau de neurone

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

2.5.1 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.

2.5.2 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-batchs*. Lorsque l'on a utilisé tous les *mini-batchs*, et donc toutes les données, on dit que l'on a effectuer une *epoch*. On peut alors en commencer une autre en effectuant un nouveau découpage aléatoire du jeu d'entraînement.

2.5.3 Résultats

En prenant $p = 30$ neurones sur la couche cachée, et en prenant des *mini-batchs* 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.

3 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*.

3.1 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, treshold = 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] >= treshold 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 considéra qu'un pixel est colorié dès qu'il y aura un peu d'encre dessus, ce qui peut également poser des problèmes.

3.2 Densités de pixels coloriés

On s'intéresse ici au niveau moyen de coloration des pixels. Pour cela, on regroupe les pixels par groupes de 16 (une fenêtre de 4 pixels sur 4 pixels) et on calcule pour chaque groupe la valeur moyenne des pixels. Comme les images ont une taille de 28x28, on obtient au final 49 valeurs que l'on peut utiliser comme *feature*.

La figure 9 montre l'effet de l'application d'un tel filtre sur une image. L'image finale a été agrandie à titre de comparaison mais fait en réalité 49x49 pixels.

On peut généraliser ce procédé en effectuant autre chose qu'une moyenne (en général un maximum) et en autorisant les fenêtres d'application à se chevaucher. C'est ce qui est fait dans les réseaux convolutifs qui seront abordés plus loin. En attendant, on peut observer en figure 10 l'effet de l'utilisation de la fonction `max`

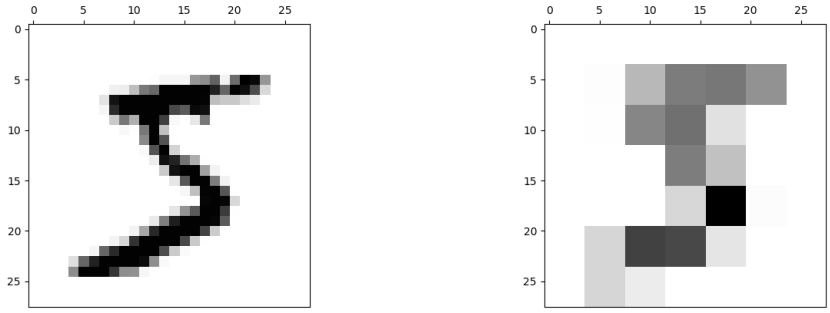

FIGURE 9 – Densité de pixels sur le chiffre 5.

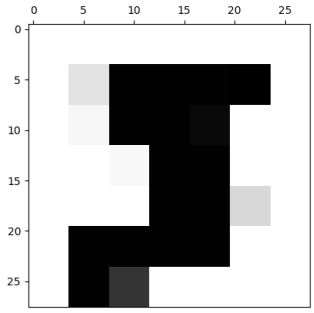

FIGURE 10 – Filtre `max` sur des fenêtres non recouvrantes de taille 4x4.

à la place de `mean`. Comme on peut le voir, il est plus difficile de reconnaître le chiffre. C'est pourquoi nous utiliserons simplement la densité dans nos essais.

3.3 Nombre de croissements (*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.

3.4 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.

3.5 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 :

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

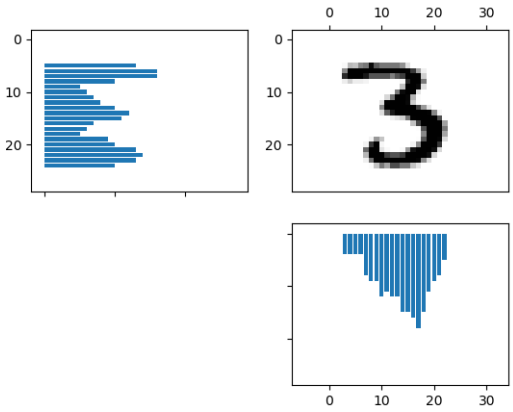

FIGURE 11 – Histogramme des projections du chiffre 3.

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

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

avec

$$\overline{x} = \frac{m_{10}}{m_{00}} \qquad \overline{y} = \frac{m_{01}}{m_{00}}$$

3.6 Transformée de Fourier du contour

On s'intéresse ici au contour des chiffres. On peut voir le contour comme une fonction périofique de $\mathbb{R}$ dans $\mathbb{R}^2$. On peut donc lui appliquer une transformée de Fourier pour en récupérer un spectre de fréquence. En conservant les plus faibles fréquences, on s'intéressera à la forme générale du contour sans considérer les détails.

En pratique, on travaille avec un contour discret. Dans notre cas, ce contour est stocké en utilisant le codage de chaînes de Freeman. Il s'agit simplement d'un codage indiquant les directions à suivre pour tracer le contour (e.g. nord, sub, nord-est, etc...). Si l'on ajoute à cela un point de départ, on est en mesure de reconstruire le contour à partir de la chaîne.

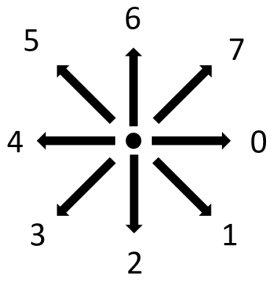

FIGURE 12 – Codage de Freeman d'une chaîne.

On code donc un contour comme un K-uplet $a_1, \cdots a_K$ où les a_i sont des entiers dans $[\![0, 7]\!]$. On notera $V = a_1 a_2 \cdots a_K$ la chaîne représentant le contour.

Cette chaîne définie un ensemble de points $(p_i)_i$ dont les coordonnées sont tous des entiers et vérifiant :

$$\|p_i - p_{i-1}\| = \begin{cases} 1 \text{ si } a_i \text{ est paire} ; \\ \sqrt{2} \text{ sinon} ; \end{cases}$$

avec $p_0 = (0, 0)$.

On associe également, pour tout couple de points consécutifs un temps de traversé Δt qui est égal à la distance séparant les deux points. Cette quantité vérifie donc :

$$\Delta t_i = 1 + \left(\frac{\sqrt{(2)} - 1}{2} \right) (1 - (-1)^{a_i})$$

Le temps mis pour traverser les p premiers liens de la chaîne vaut donc

$$t_p = \sum_{i=1}^{p} \Delta t_i$$

et la période de la chaîne est $T = t_K$.

Nous avons implémenté une fonction permettant de calculer le contour d'une image. Voici en quelques mots son principe de fonctionnement. On commence par partir du pixel situé en haut à droite de l'image et on parcours les pixels de gauche à droite et de haut an bas jusqu'à trouver un pixel colorié. On prend comme point de départ le dernier pixel non colorié sur lequel on est passé. On simule ensuite le comportement d'une coccinelle qui serait posée sur sur ce pixel et qui serait orientée direction nord-ouest. Le but de la coccinelle est d'avancer en maintenant toujours à sa droite les pixels coloriés. Pour avancer, elle regarde, à partir de sa direction actuelle, quelle est la case la plus à droite qu'elle peut atteindre sans faire demi-tour. Si la première case à droite qu'elle peut atteindre correspond à un demi-tour, elle regarde la première case à sa gauche qu'elle peut atteindre. [Note : il y a quelques autres subtilités non décrites ici pour simplifier, le code est disponible dans le fichier `features.py`] L'algorithme se termine quand la coccinelle revient à son point de départ.

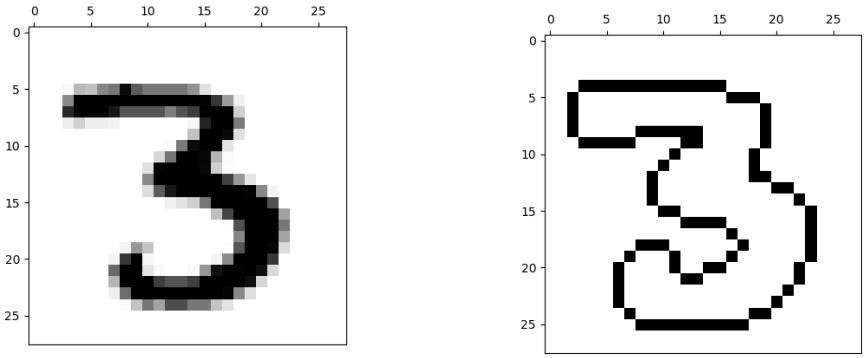

FIGURE 13 – Contour calculé du chiffre 3.

On peut ensuite appliquer une transformée de Fourier à ce contour discret. Considérons que notre contour est une fonction périodique de période T de $\mathbb{R}$ dans $\mathbb{R}^2$ dont les composantes sont $x(t)$ et $y(t)$. Alors sa décomposition en séries de Fourier s'écrit

$$x(t) = A_0 + \sum_{n=1}^{+\infty} a_n \cos\left(\frac{2n\pi t}{T}\right) + b_n \sin\left(\frac{2n\pi t}{T}\right)$$

où

$$A_0 = \frac{1}{T}\int_0^T x(t)\mathrm{d}t \qquad a_n = \frac{2}{T}\int_0^T x(t)\cos\left(\frac{2n\pi t}{T}\right)\mathrm{d}t \qquad b_n = \frac{2}{T}\int_0^T x(t)\sin\left(\frac{2n\pi t}{T}\right)\mathrm{d}t$$

Nous allons calculer ces coefficients de manière explicite en écrivant la dérivée $x'(t)$ de deux manières. On rappelle que x est dérivable en tout point $t \in]0,T[$ n'appartenant pas à $\{t_1, \cdots, t_K\}$.

La première écriture s'obtient en appliquant une transformée de Fourier à x' qui est une fonction périodique de période T.

$$x'(t) = \sum_{n=1}^{+\infty} \alpha_n \cos\left(\frac{2n\pi t}{T}\right) + \beta_n \sin\left(\frac{2n\pi t}{T}\right)$$

où

$$\alpha_n = \frac{2}{T}\int_0^T x'(t)\cos\left(\frac{2n\pi t}{T}\right)\mathrm{d}t \qquad \beta_n = \frac{2}{T}\int_0^T x'(t)\sin\left(\frac{2n\pi t}{T}\right)\mathrm{d}t$$

Il n'y a évidemment pas de terme constant dans cette écriture car l'intégrale de x' sur une période vaut 0 puisque $x(0) = x(T)$.

Ces quantités sont faciles à calculer car x' est constante par morceaux.

$$x'(t) = \frac{\Delta x_p}{\Delta t_p} \text{ pour } t_{p-1} < t < t_p$$

On obtient donc

$$\begin{aligned}\alpha_n &= \frac{2}{T}\sum_{p=1}^{K}\frac{\Delta x_p}{\Delta t_p}\int_{t_{p-1}}^{t_p}\cos\left(\frac{2n\pi t}{T}\right)\mathrm{d}t \\ &= \frac{1}{n\pi}\sum_{p=1}^{K}\frac{\Delta x_p}{\Delta t_p}\left(\sin\left(\frac{2n\pi t_p}{T}\right) - \sin\left(\frac{2n\pi t_{p-1}}{T}\right)\right)\end{aligned}$$

De même,

$$\beta_n = -\frac{1}{n\pi}\sum_{p=1}^{K}\frac{\Delta x_p}{\Delta t_p}\left(\cos\left(\frac{2n\pi t_p}{T}\right) - \cos\left(\frac{2n\pi t_{p-1}}{T}\right)\right)$$

La seconde expression de $x'(t)$ est obtenue en dérivant directement son expression sous forme de série.

$$x'(t) = \frac{2n\pi}{T}\sum_{n=1}^{+\infty} -a_n \sin\left(\frac{2n\pi t}{T}\right) + b_n \cos\left(\frac{2n\pi t}{T}\right)$$

On écrit ensuite l'égalité des coefficients devant les sinus et cosinus d'une période donnée.

$$\begin{cases}\alpha_n = \frac{2n\pi}{T} b_n \\ \beta_n = \frac{2n\pi}{T} a_n\end{cases}$$

Au final,

$$a_n = \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\frac{\Delta x_p}{\Delta t_p}\left[\cos\left(\frac{2n\pi t_p}{T}\right) - \cos\left(\frac{2n\pi t_{p-1}}{T}\right)\right]$$

$$b_n = \frac{T}{2n^2\pi^2}\sum_{p=1}^{K}\frac{\Delta x_p}{\Delta t_p}\left[\sin\left(\frac{2n\pi t_p}{T}\right) - \sin\left(\frac{2n\pi t_{p-1}}{T}\right)\right]$$

On peut appliquer le même raisonnement pour $y(t)$ et on obtiendra des coefficients C_0, c_n et d_n vérifiant les mêmes équations.

On pourra utiliser les coefficients a_n, b_n, c_n, d_n comme *features*.

La figure 14 montre la reconstruction du contour de l'exemple précédent en ajoutant à chaque étape des composantes (on commence à l'ordre 1 pour finir à l'ordre 12).

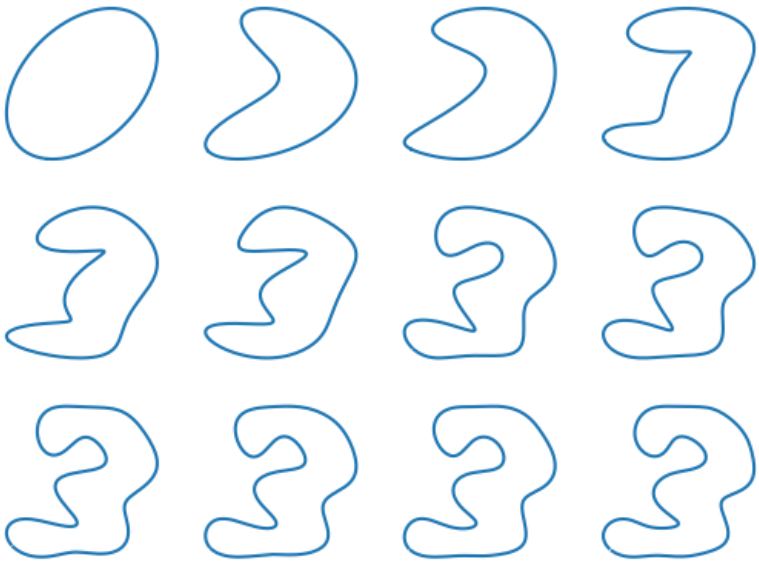

FIGURE 14 – Reconstruction d'un contour.

Cette figure permet d'estimer un intervalle d'ordres que l'on pourrait utiliser comme *feature*.

3.7 Transformée de Fourier de l'image

On utilise la fonction `rfft` de la bibliothèque `scipy.fftpack` sur l'image en niveaux de gris pour passer dans le domaine de Fourier. La procédure, qui consiste en une double appel à `rfft`, renvoie une image de même taille que l'entrée.

L'image dans le domaine de Fourier est la décomposition de l'image dans le domaine fréquentiel. Il est possible de reconstruire l'image en utilisant la fonction `irfft` de la même bibliothèque. Cette fonction permet également de visualiser la base utilisée dans le domaine fréquentiel (c.f. figure 15).

La figure 16 montre la reconstruction d'une image en ajoutant progressivement des fréquences. La figure se lit de gauche à droite et de haut en bas. L'image numéro i est reconstruite à partir de tous les coefficients d'ordre $p, q < i$ soit i^2 coefficients.

Dans ce qui suit, on se propose d'expliquer les calculs effectués par la fonction `rfft`.

Soit $x = (x_0, \cdots, x_{2n-1})$ un vecteur réel de taille $2n$.

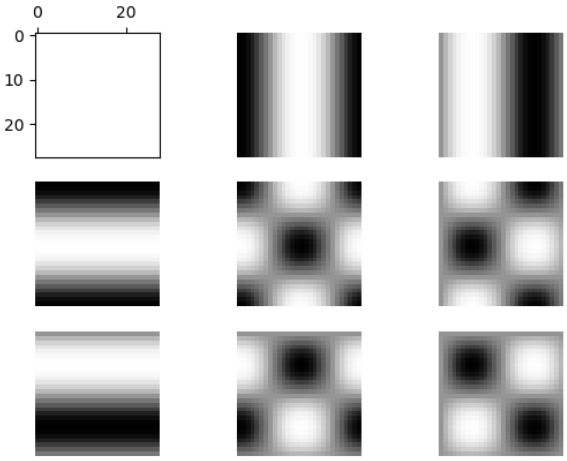

FIGURE 15 – Premiers éléments de la base de Fourier.

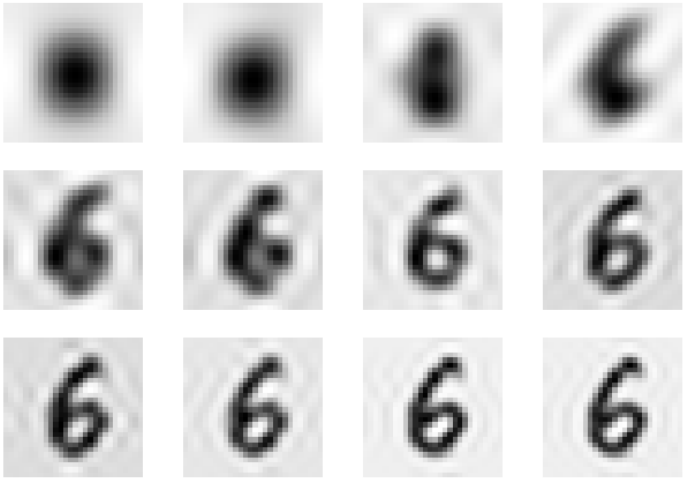

FIGURE 16 – Reconstruction d'un 6.

On définie la transformée de Fourier discrète de x par un vecteur y de même taille dont les composantes sont :

$$y_j = \sum_{k=0}^{2n-1} x_k \exp\left(-\frac{2ijk\pi}{2n}\right)$$

On remarque que $y_j = \mathrm{Conj}\, y_{2n-j}$. Ainsi, seules les valeurs y_j pour $j \leq n$ sont utiles. De plus, pour $j = n$, on obtient $y_n = \mathrm{Conj}\, y_n$, donc la partie imaginaire de y_n est nulle. Enfin, y_0 est un réel. Ainsi, on peut encoder toute l'information contenue dans les y_j pour j de 0 à $2n-1$ dans de vecteur réel $f(x)$ de taille $2n$ définit

ci-après.

$$f(x) = \begin{pmatrix} y_0 \\ \mathrm{Re}(y_1) \\ \mathrm{Im}(y_1) \\ \vdots \\ \mathrm{Re}(y_{n-1}) \\ \mathrm{Im}(y_{n-1}) \\ \mathrm{Re}(y_n) \end{pmatrix}$$

Il est possible de reconstruire x à partir de sa transformée.

$$\begin{aligned} x_j^* &= \frac{1}{2n} \sum_{k=0}^{2n-1} y_k \exp\left(\frac{2ijk\pi}{2n}\right) \\ &= \frac{1}{2n} \sum_{k=0}^{2n-1} \sum_{l=0}^{2n-1} x_l \exp\left(-\frac{2ikl\pi}{2n}\right) \exp\left(\frac{2ijk\pi}{2n}\right) \\ &= \frac{1}{2n} \sum_{l=0}^{2n-1} \sum_{k=0}^{2n-1} x_l \exp\left(\frac{2ik(j-l)\pi}{2n}\right) \end{aligned}$$

Pour $l = j$, les termes de la seconde somme valent tous x_l et il y en a $2n$, on se retrouve donc avec $2nx_l$. Pour $l \neq j$, on a la somme suivante :

$$x_l \sum_{k=0}^{2n-1} \exp\left(\frac{2i(j-l)\pi}{2n}\right)^k$$

qui est une somme géométrique valant

$$x_l \times \frac{1 - \exp\left(\frac{2i(j-l)\pi}{2n}\right)^{2n}}{1 - \exp\left(\frac{2i(j-l)\pi}{2n}\right)} = 0$$

(que l'on peut également voir comme une somme des racines $2n$-ème de l'unité)

Au final, $x_j^* = x_j$.

Une autre écriture de x_j^*, que l'on utilisera en pratique, est donnée par :

$$x_j^* = \frac{1}{2n} \left[\sum_{k=1}^{n-1} \left[y_k \exp\left(\frac{2ijk\pi}{2n}\right) + \overline{y_k} \exp\left(-\frac{2ijk\pi}{2n}\right) \right] + y_0 + (-1)^j y_n \right]$$

Considérons maintenant une matrice X carrée d'ordre $2n$. De la même manière que l'on avait vectorisé les fonctions d'activation dans la partie concernant les réseaux de neurones, on peut vectoriser f et l'appliquer à la matrice X. On définit donc $f(X)$ comme la matrice dont les colonnes sont les images des colonnes de X par f.

Le passage d'une image X de dimensions paires dans le domaine de Fourier se fait en appliquant la transformée une fois sur les colonnes puis une fois sur les lignes. La fonction `rfft` de `scipy` calcule f.

$$Y = f(f(X)^T)^T$$

Une implémentation de cette fonction est donnée en Python dans le fichier `features.py` sous le nom `rfft_vect` ; cette fonction n'est pas utilisée en pratique pour des raisons de performance.

4 Reconnaissance basée sur les réseaux convolutifs

L'une des méthodes les plus efficaces pour la reconnaissance d'image est basée sur des réseaux convolutifs. Il s'agit de réseaux de neurones profonds où les neurones d'une couche à la suivante ne sont pas forcément tous connectés, notamment dans les premières couches. Sur ces couches, l'image d'entrée est découpée en zones que l'on appelle tuile ou *patch*, qui peuvent ou non se chevaucher. Ces *patch* sont traités séparement par des neurones qui y appliquent des fonctions particulières au traitement d'image (on parle de traitement convolutif). Ce n'est que dans des couches plus profondes que les résultats sont "agglomérés" pour obtenir une prédiction.

On décompose le problème en plusieurs étapes :

1. création du modèle de réseau convolutifs, on définit le nombre de couches et les caractéristiques de chaque couche ;
2. entraînement, on utilise les données d'entraînement pour modifier les poids du réseau ;
3. évaluation de l'*accuracy* du réseau, on utilise le jeu de test pour évaluer les performances de notre réseau ;
4. prédiction, on utilise notre réseau entraîné pour faire des prédictions.

En plus de cela, il est possible d'évaluer la qualité de notre modèle (défini à l'étape 1) par validation croisée. On découpe le jeu d'entraînement en k groupes et on utilise $k-1$ groupes pour effectuer l'entraînement et le dernier groupe pour évaluer le réseau (taux de reconnaissance). On répète l'opération en utilisant chacun des groupes une fois pour l'évaluation. En effectuant plusieurs fois l'opération de découpage en k groupes, on est en mesure d'obtenir une valeur moyenne pour la précision ainsi qu'une variance.

4.1 Construction du réseau et description des différentes couches

Nous avons utiliser la bibliothèque Python `keras` pour construire le réseau convolutif. Le code correspondant à la création du modèle est très simple avec cette bibliothèque.

```
def build_classifier():
    classifier = Sequential()
    classifier.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1), activation='relu'))
    classifier.add(MaxPooling2D(pool_size=(2, 2)))
    classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
    classifier.add(MaxPooling2D(pool_size=(2, 2)))
    classifier.add(Flatten())
    classifier.add(Dropout(0.1))
    classifier.add(Dense(units = 128, activation='relu'))
    classifier.add(Dropout(0.1)) # Overfitting reduction - Dropout
    classifier.add(Dense(units = 10, activation='softmax'))
    classifier.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return classifier
```

Nous allons faire une description ligne à ligne du code précédent.

```
classifier = Sequential()
```

Cette ligne indique que l'on souhaite utiliser un modèle séquentiel, c'est à dire un modèle où les couches du réseau sont parcourues linéairement, les unes après les autres.

```
classifier.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1), activation='relu'))
```

On ajoute à notre réseau une première couche dite de convolution. Comme il s'agit de la première couche, on doit préciser la dimension de l'*input*. Ici, nos images seront des tableaux de 28×28 pixels. L'opération de convolution consiste à appliquer des filtres à nos images. Ce type d'opération est notamment utilisé en traitement d'image pour appliquer une fonction particulière à une image. Par exemple, le filtre moyenne remplace chaque pixel par la moyenne des pixels adjacents et du pixel central. Ce filtre peut etre utilisé par exemple pour réduire le bruit. De la même manière il existe des filtres de détection de contours ou encore des filtres mettent en avant les changements brutaux de luminosité.

FIGURE 17 – Filtre moyenne.

Mathématiquement, appliquer un filtre à une matrice de pixels consiste à effectuer le produit scalaire d'une portion de l'image par une matrice de petite taille (ici 3×3). Le principe de cette opération est que les portions de l'image sur lesquelles sont appliquées ces produits scalaires se chevauchent. Ainsi l'information locale est préservée et on garde la cohérence de l'image.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

FIGURE 18 – Matrice à filtrer.

1	0	1
0	1	0
1	0	1

FIGURE 19 – Exemple de filtre.

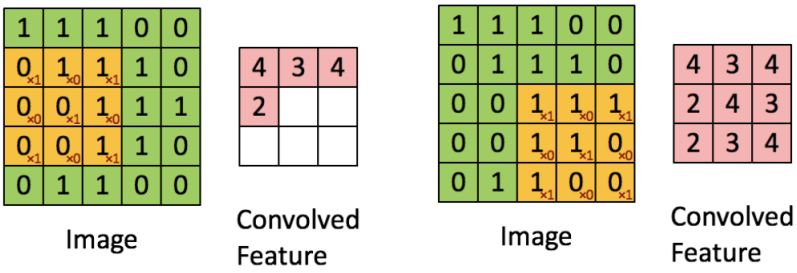

FIGURE 20 – Filtrage de l'image.

On précise également une fonction d'activation qui est ici `relu`. Les fonctions d'activation permettent de transformer le signal entrant en signal de sortie. On peut citer notamment les fonctions tangante hyperbolique, sigmoïde, exponentielle. La fonction d'activation `relu` est la plus populaire dans le cadre des réseaux de neurones à convolution. Les deux premiers arguments sont les plus intéressants. On indique que l'on souhaite applique 32 filtres à des `patch` de taille 3×3.

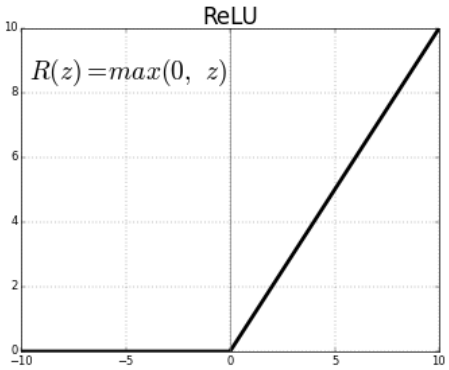

FIGURE 21 – fonction d'activation reLu.

Comme l'on ne précise rien d'autre, les `patch` se chevauchent par défaut avec un décalage de 1. On obtient donc en sortie de cette couche une image de taille 26×26 pour chacun des 32 filtres soit $26 \times 26 \times 32$ valeurs.

```
classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

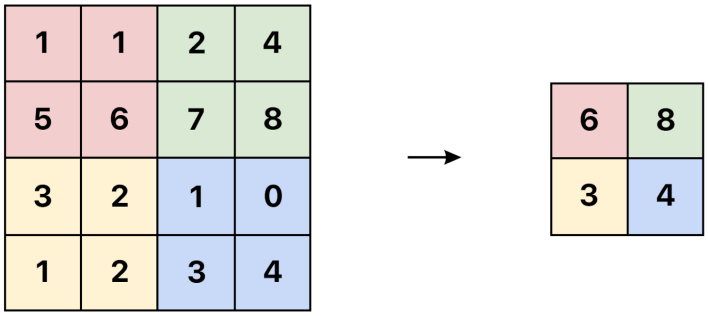

FIGURE 22 – Max pooling operation.

On applique dans cette couche une fonction à des `patch` de taille 2×2 qui ne se chevauchent pas. Le Pooling est une méthode permettant de réduire la taille d'une image tout en préservant les informations essentielles. Cette opération permet aussi de réduire le sur-apprentissage en généralisant l'information localement. Comme le suggère le nom, la fonction appliquée ici est `max`. Nos images de taille 26×26 voient donc leur taille réduite de moitié. On garde cependant une image pour chaque filtre de la première convolution soit un espace de sortie de taille $13 \times 13 \times 32$.

Notons que l'on a plus besoin de préciser la taille de l'entrée dans cette couche (elle est déduite automatiquement de la couche précédente).

```
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
```

On réapplique ici une seconde couche de convolution soit 32 filtres en utilisant des *patch* de taille 3×3. Les images de sortie auront donc une taille de 11×11 et on obtient donc une sortie de taille $11 \times 11 \times 32$ (chacun des 32 filtres n'est ici appliqué qu'à une seule image).

```
classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

On refait un `MaxPooling2D` ce qui réduit la taille de la sortie à $5 \times 5 \times 32$. En effet lorsqu'une zone n'est pas recouverte entièrement par un patch, celle-ci n'est pas prise en compte (cf. zone grisée de la figure suivante).

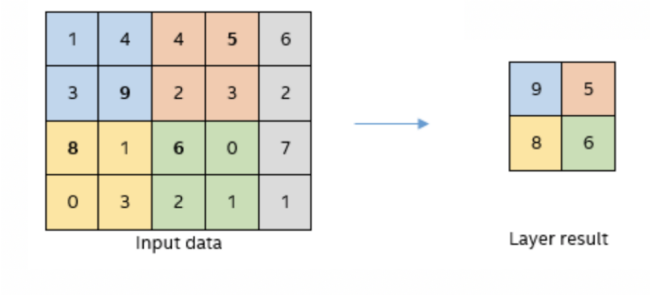

FIGURE 23 – Max pooling avec zone non prise en compte.

```
classifier.add(Flatten())
```

On ajoute alors une couche qui aplatit la sortie de la couche précédente : cette étape est nommée flattening. On obtient donc un vecteur de taille 800.

```
classifier.add(Dropout(0.1))
```

Cette ligne de code supplémentaire permet de mettre aléatoirement à zéro une partie des inputs de notre réseau pendant l'entraînement pour limiter le sur-apprentissage. Ici 0,1des entrées sont mises à zéro.

```
classifier.add(Dense(units = 128, activation='relu'))
```

On ajoute enfin une couche de neurone *fully-connected*. Cela correspond à une couche classique d'un réseau de neurone telle que présentée dans la première partie. On précise la dimension de l'espace de sortie, 128 ; et la fonction d'activation, `relu`.

```
classifier.add(Dropout(0.1))
```

Une autre couche de *dropout* !

```
classifier.add(Dense(units = 10, activation='softmax'))
```

La dernière couche du réseau, ayant 10 sorties correspondant à nos 10 classes. La fonction d'activation utilisée ici est `softmax`. Cette fonction permet de transformer un vecteur de nombre positifs en un vecteur définissant une probabilité (les composantes peuvent être vues comme la probabilité d'appartenance à la classe associée).

```
classifier.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

On compile notre modèle, on peut désormais entraîner notre réseau. Il reste cependant un certain nombre de paramètre à régler pour optimiser notre modèle.

4.2 Optimisation du modèle

4.2.1 Paramètres du modèle

Les principaux paramètres du modèle sont :

— Batch size : les images ne sont pas propagées dans le réseau une par une mais par paquet ou batch. Ce paramètre détermine donc la taille de ces paquets. Les poids du réseau sont mis à jour après le passage de chacun de ces paquets. Typiquement plus le batch size est petit meilleur sera l'apprentissage. En revanche le temps de calcul est alors d'autant plus long. Un bon compromis consiste souvent à prendre ce paramètre égal à 32.
— Nombre d'epochs : un epoch correspond à un passage au travers du réseau de l'ensemble des données d'entrainement. Pour continuer d'enrichir le modèle, ce processus est répété plusieurs fois. La précision du modèle augmente donc avec le nombre d'epochs jusqu'à un certain point de stagnation.
— Optimizer : il s'agit de l'algorithme mettant à jour les poids par rétro propagation. Le optimizer les plus populaires sont le classique stochastic gradient descent, adam et rmsprop. Certains algorithmes sont plus adaptés à certain types de réseau.
— Le nombre de couche, leur agencement, ainsi que les paramètres internes à chaque couche tels que la le nombre de filtres et leut taille pour la convolution et la taille des opérateur de pooling.

Cette dernière partie est la plus complexe à optimiser car contrairement aux premiers paramètres cités, les formes de réseau possibles sont infinis. En effet, s'il est possible de déterminer les paramètres optimaux avec une méthode du type GridSearchCV, il n'est en revanche pas possible d'établir une méthode pour déterminer le nombre de couches idéales, leur ordonnancement et les paramètres associés. C'est ici avant tout l'expérience qui va déterminer la configuration à adopter même s'il est possible detester un certain nombre de configuration en cross validation. Les data scientist mettent souvent en avant l'importance de l'intuition dans l'établissement d'un réseau de neurones, notamment pour cette raison.

4.2.2 Réduction du sur-apprentissage

Nous avons déjà vu une méthode essentielle à la réduction du sur-apprentissage avec l'utilisation de drop out. Tout comme pour les paramètres des différentes couches, le pourcentage de noeuds à aléatoirement mettre à zéro de même que le positionnement de ces drop out au sein du réseau sont des paramètres qui relèvent de l'expérience et de l'expérimentation.

Cependant, il y a un autre point important sur lequel jouer pour limiter le phénomène de sur-apprentissage. Il s'agit de détériorer volontairement le jeu de données d'entrainement par une série de transformation aléatoires telles que des translations, des zooms ou encore des rotations. Ces légères modifications permettent de rendre le jeu de données plus hétérogènes et limitent donc certains biais liés à la méthode de création du jeu de données. Dans le cadre d'enregistrement de nombre manuscrits comme réalisé avec le jeu de données MNIST, malgré un effort pour avoir des données les plus variées possible, en faisant appel à de nombreuses personnes pour la rédaction des caractères, certains biais subsistent. Par exemple la méthode même de numérisation des données, la qualité globales des images, le contraste, l'épaisseur du trait, le centrage des caractère sont autant de facteurs qui influencent l'apprentissage d'un modèle prédictif. Détériorer intelligemment le jeu de donnée initiale peut donc apporter une amélioration significative au modèle en limitant ces biais mais aussi en enrichissant le modèle.

Sous Python `keras`, la méthode couramment utilisée pour réaliser cette opération est ImageDataGenerator dont les principaux paramètres sont :

— rotation_range
— width_shift_range, height_shift_range
— zoom_range

La figure suivante illustre l'utilisation de cette méthode dans le problème classique de classification de photos de chats et de chiens.

Dans le cadre de la prédiction de caractère manuscrits, seuls les zooms et déformation verticales et horizontales sont intéressant ainsi que les rotations dans une certaine mesure (pour imiter l'écriture légèrement penchée de certaines personnes par exemple). Les renversement d'image sont ici à proscrire. Ces modifications permettent à la fois de faire varier l'épaisseur du trait, le centrage de l'image et apporte de la diversité en déformant

FIGURE 24 – Exemple d'utilisation de ImageDataGenerator

certains caractères. Il peut aussi s'avérer judicieux de faire varier les niveaux de gris. Ceci afin de corriger un éventuel biais lié à la méthode d'enregistrement mais aussi pour enrichir une fois de plus le jeu de données en faisant varier ce que l'on pourrait comparer à l'intensité du trait. Dans le cas de l'écriture manuscrite avec un crayon par exemple, plus l'utilisateur appuie fort lorsqu'il écrit, plus les niveaux de gris seront élevées après numérisation. De même le type de stylo utilisé peut être un facteur limitant pour un modèle entrainé avec des données très homogènes.

5 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.