المملكة المغربية
وزارة التربية الوطنية
والتعليم الأولي والرياضة

المركز الجهوي لمهن التربية والتكوين لجهة الشرق
Centre Régional des Métiers de l'Education et de la formation de l'Oriental

DEPARTEMENT DES MATHEMATIQUES

Cycle : Enseignement Secondaire Qualifiant des Mathématique

(ESQM)

MEMOIRE DE FIN D'FORMATION

Pour l'obtention du Diplome de Qualification des Enseignants Stagiaires délivré par le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental

(CRMEF)

les réseaux de neurones artificiels : Application à la classification d'images

Réalisé par :
ZGAOUA Khalid

Sous l'encadrement de :
Pr. Y. Derfoufi

Soutenu le 2026

Devant le jury composé de :

Pr. X	CRMEF OUJDA
Pr. Y. Derfoufi	CRMEF OUJDA

Année Universitaire 2025 / 2026

le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental de OUJDA
Hay Al Massira, La Annexe Boudir
Tel : 212 (0)5 36 68 35 31 – Fax : 212 (0)5 35 60 82 14
Site web : http ://www.crmeforiental.com

Table des matières

Table des figures

Liste des tableaux

Remerciement

Ce travail est le fruit de la combinaison d'efforts de plusieurs personnes. Je remercie tout d'abord le tout puissant qui, par sa grâce m'apermis d'arriver au bout de mes efforts en me donnant la santé, la force, le courage et en me faisant entourer des mer veilleuses personnes dont je tiens à remercier. Je remercie :

Mes deux directeurs de mémoire Monsieur **Y.Derfoufi**, professeur à le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental de OUJDA et Monsieur **El Boutkhili Fayssal**, professeur au lycée Ibn El Haytame de OUJDA . Grâce à leur savoir, leurs idées, leur disponibilité et leur soutien, j'ai pu réaliser ce travail.

Les membres du jury : Monsieur **X** professeur à le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental de OUJDA et Monsieur **Y**, professeur à le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental de OUJDA, qui m'ont honoré en acceptant d'évaluer et de juger mon travail.

Tous les enseignants de le Centre Régional des Métiers de l'Education et de la Formation de l'Oriental de OUJDA, pour leurs enseignements de qualité et leurs conseils qui nous ont permis de poursuivre notre itinéraire académique jusqu'à présent.

Mes parents qui m'ont toujours encouragés dans la poursuite de mes études, ainsi que pour leur aide, leur compréhension et leur soutien.

Mes Frères et soeurs pour leurs encouragements durant tout mon parcours.

Mes camarades et tous ceux qui de près ou de loin ont contribué à l'accomplissement de ce travail.

Dédicace

À mes parents :

Aucun mot ne pourra exprimer tous mes sentiments d'amour et de gratitude, je vous remercie beaucoup pour vos efforts le long de ces années, pour votre présence rassurante et pour tout l'amour que vous m'avez réservé, vous étiez et vous resterez pour moi les parents idéals qui m'ont toujours conseillé et orienté dans le bon sens, peut-être que j'ai encore des réserves sur des décisions que vous avez préféré de les prendre à ma place, mais soyez sûr que je n'ai jamais perdu ma croyance en votre sagesse que je la considère jusqu'à l'instant indiscutable.

Chers parents, je vous remercie une autre fois pour vos sacrifices, que Dieu vous prête la longue vie et la bonne santé a fin que je puisse vous rendre la pareille à mon tour, je vous aime très fort.

À mes chers frères et soeurs :

Qui ont toujours été là pour moi, avec leur support leurs encouragements.Qu'ils trouvent dans ce travail, toute ma reconnaissance et tout mon grand amour envers eux. Je vous aime beaucoup.

À tout les membres de ma famille :

En témoignage de profonds liens qui nous unissent, veuillez trouver à travers ce travail l'expression de mon grand amour, mon attachement et ma profonde reconnaissance.

Introduction générale

Dans notre monde moderne, monde numérique où la technologie se développe à grande échelle, l'intelligence artificielle a fait un grand pas en se manifestant comme un atout majeur dans la société moderne. Le terme IA (Intelligence Artificielle) désigne toutes les techniques et méthodes qui permettent de doter des systèmes informatiques de capacités proches de celles de l'être humain.

L'apprentissage profond « Deep Learning » est une branche très importante de l'intelligence artificielle qui a connu un développement énorme ces dernières années. Le Deep Learning s'appuie sur un réseau de neurones artificiels s'inspirant du cerveau humain. Ce réseau est capable de décider de manière autonome, en fonction de certaines entrées fournies par le système.

Le réseau de Neurone Convolutif (CNN) est une architecture d'apprentissage en profondeur bien connue inspirée du réseau de neurones artificiels. Ils utilisent des couches de convolution pour extraire automatiquement les caractéristiques pertinentes des images et sont entraînés sur de vastes ensembles de données annotées afin d'apprendre à reconnaître différents motifs et structures présents dans les images. Les CNN ont montré des performances remarquables dans de nombreux domaines et il a été principalement utilisé dans diverses applications, notamment la classification d'images, la détection d'objets et la reconnaissance automatique de chiffres, de lettres, de textes manuscrits, de personnes.

L'objectif de notre travail est de développer un programme pour la classification des images en utilisant des techniques de réseau de neurones covolutionnels (CNN).

Par conséquent, notre mémoire est subdivisé comme suit :

Le premier chapitre est une présentation générale de l'apprentissage automatique,et qui va englober dans un premier temps une introduction aux réseaux de neurones artificiels, et au cours de laquelle ont raitera leur histoire, leurs définitions, ainsi que leurs procédures d'apprentissage, et dans un second nous ferons notre premier contact avec le modèle neuronal le plus célèbre, et qui sera nommé "perceptron multicouches", nous allons définir sa nature, sa topologie, ses variantes, et enfin on va consacrer à l'étude de quelques algorithmes d'apprentissage dans les réseaux profonds, ainsi que les caractéristiques de chaque algorithme. Nous allons aborder aussi les réseaux de neurones convolutifs (CNNs),

et leurs types ainsi que leur avantage par rapport aux réseau de neurones artificiels entièrement connectés dans le cadre de traitement des images.

Les modèles de réseau neuronal (NN) sont bien adaptés aux domaines où de grands ensembles de données étiquetées sont disponibles, car leur capacité peut facilement être augmentée en ajoutant plus de couches ou plus d'unités dans chaque couche. Cependant, les grands réseaux avec des millions ou des milliards de paramètres peuvent facilement sur-ajusté (Overfitting). En conséquence, des méthodes de régularisation des réseaux de neurones ont été développées, et qui feront l'objet du deuxième chapitre.

Finalement, dans le troisième chapitre nous proposons une architecture matérielle pour un module d'un réseau CNN, que nous définissons en détails et nous exposerons la présentation et la discutions des différents résultats obtenus lors de la phase d'expérimentation.

Chapitre 1

Réseaux de neurones profonds

1.1 Introduction

Un réseau de neurones artifsciels, ou réseau neuronal artificiel, est un ensemble d'algorithmes dont la conception est à l'origine très schématiquement inspirée du fonctionnement des neurones biologiques.Il permet de traiter des problèmes de différentes natures que les outils classiques ont du mal à résoudre.
Dans ce chapitre, nous introduisons la modélisation mathématique des réseaux de neurones artificiels. Par la suite nous allons détailler les principes de base de l'apprentissage en profondeur en mettant l'accent sur les réseaux de neurone convolutif (CNNs).

1.2 Historique

De nombreux ouvrades ont permis de documenter l'histoire des recherches en réseaux de neurones. Les recherches menées dans le domaine du connexionnisme ont démarré avec la présentation en 1943 par W.McCulloch et W.Pitts [1] d'un modèle simplifié de neurone biologique communément appelé neurone formel (Artificiels).Ils ont montré théoriquement que les réseaux de neurones formels simples peuvent réaliser des fonctions logiques, arithmétiques et symboliques complexes.

En 1958, F.Rosenblatt [1] développe le modèle du Perceptron. Ce dernier possède deux couches de neurones : une de perception (sert à recueillir les entrées) et une couche de décision. C'est le premier modéle pour lequel un processus d'apprentissage a pu étre défini.

S'inspirant du perceptron, Widrow et Hoff [1], développent dans la méme période, le modèle de l'Adaline (Adaptive Linear Element). Ce dernier sera, par la suite, le modèle de base des réseaux de neurones multicouches.

Les recherches sur les réseaux de neurones ont été pratiquement abandonnées lorsque M.Minsky et S.Papert [1] ont publié leur livre (Perceptrons en 1969) et démontré les limites théoriques du perceptron, en particulier, l'impossibilité de traiter les problèmes non linéaires par ce modèle.

Une révolution survient alors dans le domaine de réseaux de neurones artificiels, une nouvelle génération de réseaux de neurones capable de traiter avec succès des phénomènes

non linéaires : le perceptron multicouche, il ne possède pas les défauts mis en évidence par Minsky. Proposé pour la première fois par Werbos [1]. Le perceptron Multicouche apparait en 1986 introduit par Rumelhart, et simultanément sous une appellation voisine, chez le Cun(1985). Ces systèmes reposent sur la rétro-propagation du gradient de l'erreur dans des systèmes a plusieurs couches, chacune de types Adaline de Bernard Widrow, proche du perceptron de Rumelhart. L'évolution de la théorie des réseaux de neurones formels est inspirée directement du développement des travaux biologique sur le cerveau humain.

1.3 Modèles d'un neurone

Comme les réseaux de neurones mis au point par les informaticiens sont largement inspirés de ce que la biologie nous apprend sur ceux que l'on trouve chez les étres vivants, il convient d'abord de décrire brièvement le modèle biologique.

1.3.1 Neurone Biologique

Le cerveau est l'organe de commande le plus inconnu de la biologie de l'homme ou de l'animal. Les cellules nerveuses, appelées neurones, sont les éléments de base du système nerveux central.

Dans leur organisation générale et leur système biochimique, cependant, des caractéristique particulières qui se distinguent par quatre fonctions spécialisées :

— Recevoir des signaux en provenance des neurones voisines ;
— Intégrer ces signaux ;
— Engendrer un influx nerveux ;
— Conduire et transmettre l'influx nerveux à un neurone capable de recevoir ;

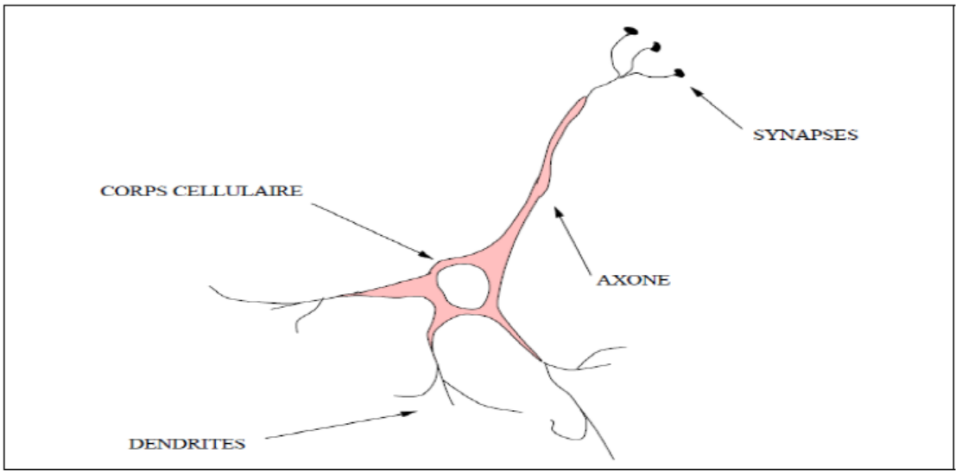

FIGURE 1.1 Un neurone biologique et ses principales composantes

Axone : l'axone transporte l'influx nerveux vers les synapses.

Synapses : transmettent l'influx nerveux provenant de l'axone vers d'autres cellules à partir neurotransmetteurs.

Dendrites : selon leur longueur et leur perméabilité, affectent la quantité d'influx nerveux qui se rend au noyau.

Noyau : est le centre des réactions électrochimiques, si les stimulations externes sont suffisantes, le noyau provoque l'envoi d'un influx nerveux électrique à travers l'axone.

1.3.2 Neurone formel(Artificiel)

Inspiré du système biologique, le neurone formel est un automate caractérisé par un petit nombre de fonctions mathématiques. Il traite un signal recueilli (signal d'entrée) à travers ses connexions entrantes pour fournir un signal de sortie calculé par la fonction de tansfert. En générale, on considère que chaque neurone fournit une information additive aux unités de calcul (neurones) qui lui sont connectées [1].

la valeur de la somme pondérée est, simplement, la somme des sorties des différents neurones en amont avec lesquels il est connecté, La figure 2 résume la structure détaillée du neurone formel (Artificiel).

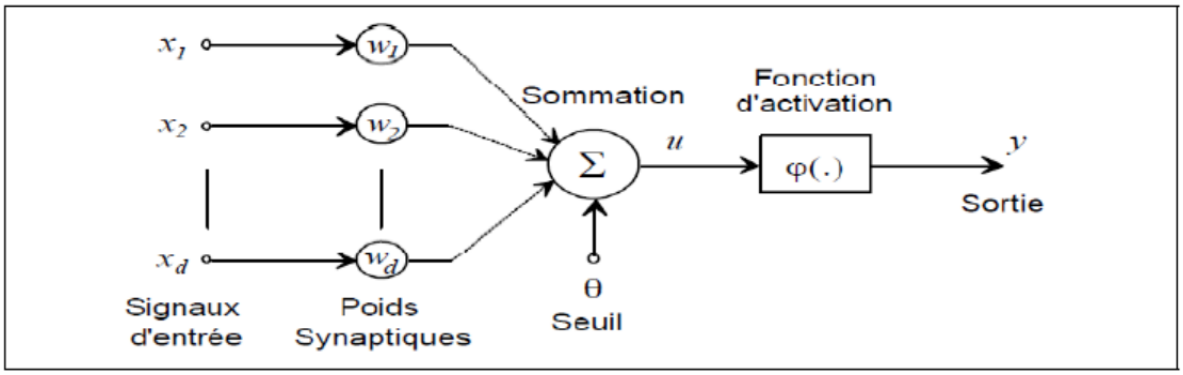

FIGURE 1.2 Modéle générale du neurone formel

X se compose de d entrée des neurones correspondent au vecteur de caractéristique du signal d'entrée.

W : Vecteur des poids du neurone.

θ : Biais interne.

φ : Fonction d'activation.

$y = \varphi(u)$: Sortie du neurone, avec $u = \sum_{i=1}^{d} w_i x_i - \theta$.

1.3.3 Fonction d'activation

La fonction d'activation est une fonction mathématique appliquée à un signale en sortie d'un neurone artificiel. Le terme de "fonction d'activation" vient de l'équivalent biologique "potentiel d'activation", seuil de stimulation qui, une fois atteint entraine une

réponse du neurone. La fonction d'activation est souvent une fonction non linéaire [1].

Les types de fonctions d'activation les plus utilisées sont :

Fonction seuil : est une fonction qui prend des valeurs discrètes en fonction d'un seuil donné, typiquement $\varphi(x) = 0$ si $x < \theta$ et $\varphi(x) = 1$ si $x \geq \theta$.

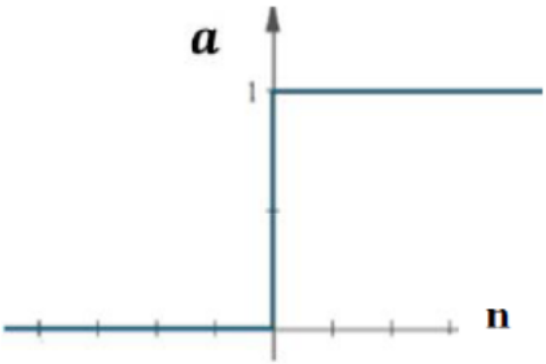

FIGURE 1.3 Fonction seuil

Fonction linéaire : est une fonction de la forme $\varphi(x) = ax$ où a est une constante réelle.

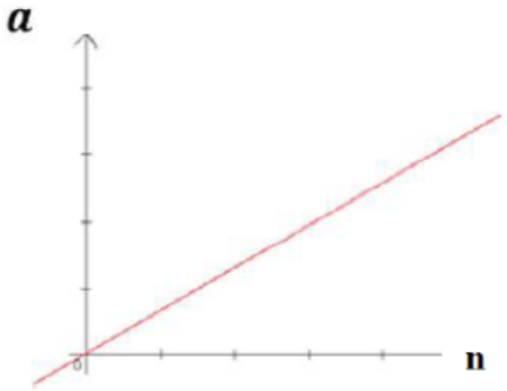

FIGURE 1.4 Fonction linéaire

Fonction tangente hyperbolique : C'est une autre fonction non-linéaire, mais qui donne une convergence plus que la sigmoïde puisque la sortie de la fonction sigmoïde est centrée sur 0, ce qui favorise la fonction tanh pour l'entrainement des modèles profond.

$$\varphi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.3.1}$$

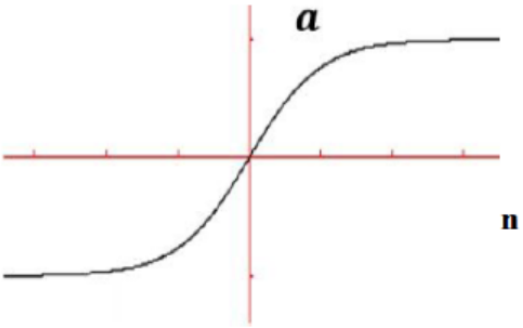

FIGURE 1.5 Fonction tangente hyperbolique

Fonction sigmoide : C'est une fonction d'activation non linéaire simple à calculer et à différencier. Elle se définie en :

$$\varphi(x) = \frac{1}{1 + e^{-x}} \tag{1.3.2}$$

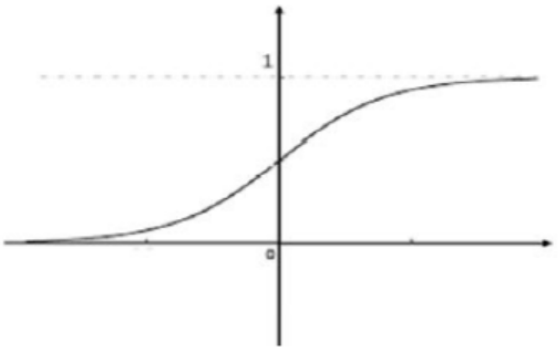

FIGURE 1.6 Fonction sigmoide

Fonction RELU : Abréviation de Unités Rectifié linéaires C'est la fonction la plus populaire et la plus utilisé dans les CNN profond c'est une fonction très simple définit par :

$$\varphi(x) = \max(0, x) \tag{1.3.3}$$

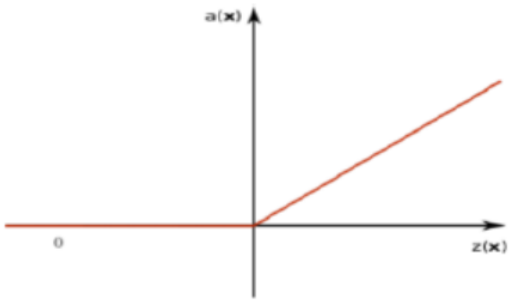

FIGURE 1.7 Fonction ReLU

Les caractéristiques des fonctions d'activation sont les suivantes :

— la monotonie : la fonction d'activation est toujours croissant.

— les euillage : possède une valeur au-dessous de laquelle sa valeur est négligeable.

— la saturation : elle possède une valeur maximale au-dessus de laquelle sa valeur de réponse est essentiellement fixe, ceci permet d'éviter de propager de grandes valeurs dans le réseau.

1.4 Apprentissage et optimisation dans les réseaux profonds

1.4.1 Perceptron mulicouche (PMC)

Le perceptron multicouche est un type des réseaux de neurones les plus utilisés pour des problèmes d'approximation, de classification et de prédiction. Il est habituellement constitué d'au moins trois couches totalement connectés.

La couche d'entrée : ne joue généralement que le rôle d'interface avec l'extérieur, c'est à dire qu'elle n'effectue pas de traitement sur l'information.

Une ou plusieurs couches cachées : appelées aussi couches de traitement ou couches intermédiaires, leurs intérêt est d'augmenter le nombre de connexions, ce qui accroît la capacité d'un réseau à extraire l'information des données fournies en entrée.

La couche de sortie : traite également l'information qu'elle reçoit de la couche précédente mais elle ne communique pas son résultat aux autres neurones. L'information fournie par chacune de ses unités constitue le vecteur de sortie du réseau. Celui-ci est donc le résultat du traitement d'un vecteur d'entrée par l'ensemble du réseau.

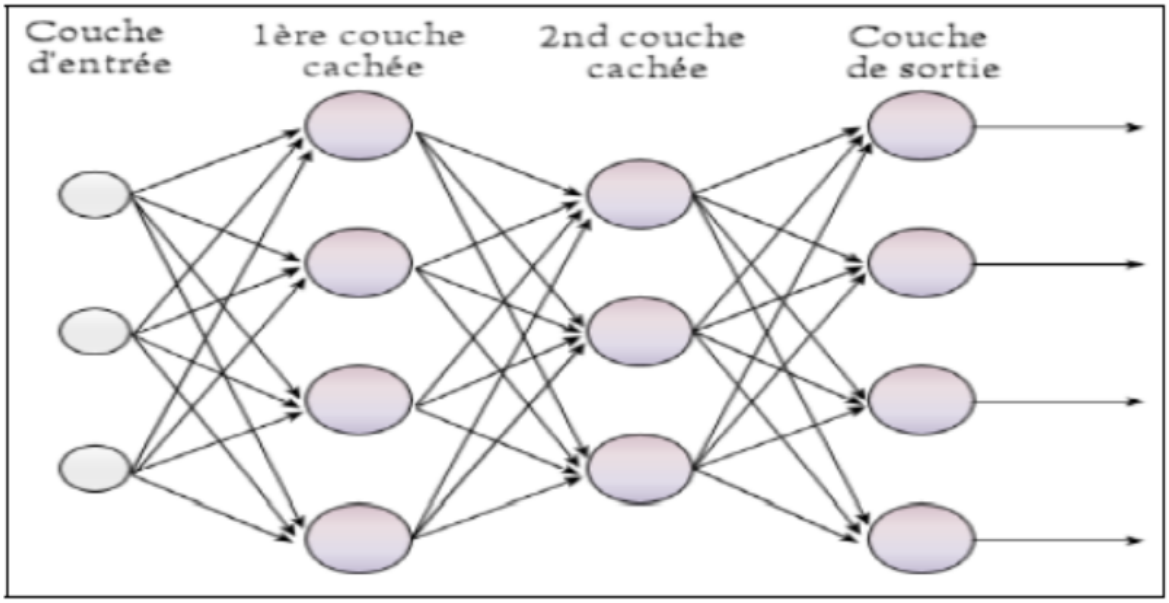

FIGURE 1.8 Un perceptron multicouche avec deux couches cachées

Étant donnée une base d'apprentissage

$$X = \{(x_1, d_1), (x_2, d_2), ..., (x_n, d_n)\}$$

avec :
x_i est l'entrée i.
d_i est la sortie attendue (désirée) associe à x_i.
On veut minimiser l'erreur quadratique (SE) :

$$E(w) = \frac{1}{2}\sum_{i=1}^{n}(d_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{n} E_i \tag{1.4.1}$$

avec :
$E_i = (d_i - y_i)^2$ est l'erreur pour l'entrée x_i(appelé aussi l'erreur local).
y_i est la sortie produite par le réseau pour l'exemple (la donnée) i.
W est la matrice des poids synaptiques du réseau.

Remarque 1.4.1. *Ici on a choisi l'erreur quadratique. Il existe différentes formules pour calculer l'erreur entre la sortie attendue d_i et la sortie produite par le réseau $y_i = F(x_i)$. Par exemple, l'erreur absolue moyenne donnée par :*

$$E(w) = \frac{1}{n}\sum_{i=1}^{n}|d_i - y_i| \tag{1.4.2}$$

Le choix de la fonction erreur dépend du problème, par exemples il a sortie produite est une probabilité $\tilde{y}_i$ avec $0 \leq \tilde{y}_i \leq 1$, Alors la fonction erreur adaptée est l'entropie croisée binaire :

$$E(w) = -\frac{1}{n}\sum_{i=1}^{n}(y_i ln(\tilde{y}_i) + (1 - y_i) ln(1 - \tilde{y}_i)) \tag{1.4.3}$$

1.4.2 Descente de gradient

L'objectif de la méthode de descente de gradient est de trouver un minimum d'une fonction de plusieur svariables le plus rapidement possible. L'idée est très simple, on sait que le vecteur opposé au gradient indique une direction vers des plus petites valeurs de la fonction, il suffit donc de suivre d'un pas cette direction et de recommencer. Cependant, afin d'être encore plus rapide, il est possible d'ajouter plusieurs paramètres qui demandent pas mal d'ingénierie pour être bien choisis.
Imaginons une goutte d'eau en haut d'une colline. La goutte d'eau descend en suivant la ligne de plus grande pente et elle s'arrête lorsqu'elle atteigne un point bas. C'est exactement ce que fait la descente de gradient : Partant d'un point sur une surface, on cherche la pente la plus grande en calculant le gradient et on descend d'un petit pas, on recommence à partir du nouveau point jusqu'à atteindre un minimum local.

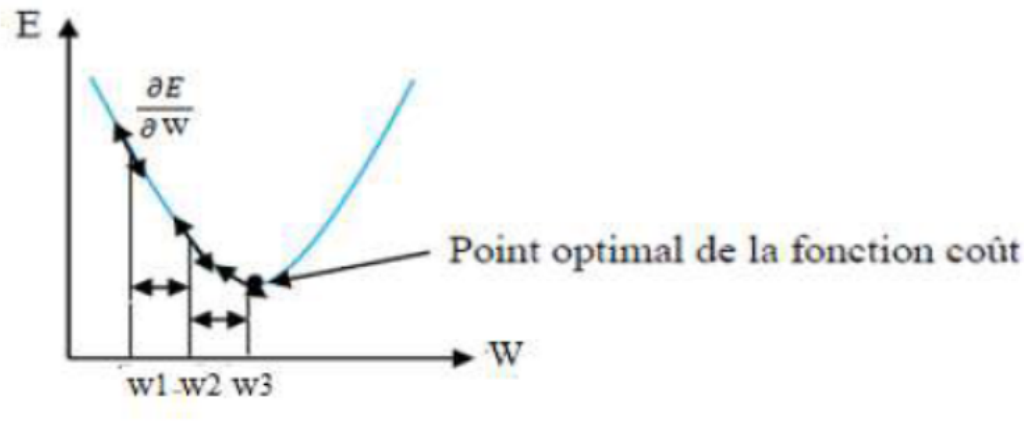

FIGURE 1.9 Optimisation de la fonction coût par descente de gradient.

Descente du gradient classique(cgd)

On suppose que la fonction erreur E est différentiable.

Algorithm 1 Descente de gradient (version itérative)

<Entrées :> Un point initial w_0.
<Entrées :> Un niveau d'erreur $\epsilon > 0$.
<Itération :> On calcule une suite de points $w_1, w_2, ..., w_k, ...$ par récurrence de la façon suivante. Supposons que l'on ait déjà obtenu le point w_k :
(a) On calcule $\nabla E(w_k)$.
(b) On choisit un pas δ_k et on calcule :

$$w_{k+1} \longleftarrow w_k - \delta_k \nabla E(w_k)$$

<Arrêt :> On s'arrête lorsque $||\nabla E(w_k)|| < \epsilon$.

Remarque 1.4.2. *On a*

1. *Évidemment, plus on choisit le point initial w_0 proche d'un minimum local, plus l'algorithme va aboutir rapidement. Cependant, comme on ne sait pas où est ce minimum local (c'est ce que l'on cherche), le plus simple est de choisir un w_0 au hasard.*
2. *Le choix du pas δ_k est crucial. On sait quel'on peut choisir δ_k assez petit de façon à avoir $E(w_{k+1}) < E(w_k)$ car dans la direction de $-\nabla E(w_k)$ la fonction E décroît. On peut fixer à l'avance un pas δ commun à toutes les itérations, par exemple $\delta = 0.01$. On pourrai tégalement tester à chaque itération plusieurs valeurs de δ par balayage ($\delta = 0.001$, puis $\delta = 0.002$...) et choisir pour δ_k celui en lequel E prend la plus petite valeur.*

Descente du gradient stochastique(sgd)

La descente de gradient stochastique est une façon d'optimiser les calculs de la descente de gradient pour une fonction d'erreur associée à une grande série de données. Au lieu de calculer un gradient compliqué et un nouveau point pour l'ensemble des données, on calcule un gradient simple et un nouveau point par donnée. SGD sélectionne au hasard

un point de données dans l'ensemble de données à chaque itération.
Pour minimiser l'erreur et déterminer les meilleurs paramètres, on peut appliquer la méhode du gradient classique. Pour appliquer cette formule, il faut calculer des gradients $\nabla E(w_k)$, or

$$\nabla E(w_k) = \sum_{i=1}^{n} \nabla E_i(w_k) \tag{1.4.4}$$

Il faut donc calculer une somme de n termes à chaque itération, ce qui pose des problèmes d'efficacité pour de grandes valeurs de n.
Pour diminuer la quantité de calculs, l'idée est de considérer à chaque itération un seul gradient E_i à la place de E. C'est-à-dire :

$$w_{k+1} \longleftarrow w_k - \delta_k \nabla E_i(w_k) \tag{1.4.5}$$

pour une seule erreur E_i (correspondant à la donnée numéro i). L'itération suivante se basera sur l'erreur E_{i+1}.
Quel est l'intérêt de cette méthode ?
Dans la méthode de gradient classique, on calcule à chaque itération un «gros» gradient (associe à la totalité des n données) qui nous rapproche d'un grand pas vers le minimum. Ici on calcule n «petits» gradients qui nous rapprochent du minimum.

Descente par lot (mini-batch)

Il existe une méthode intermédiaire entre la descente de gradient classique (qui tient compte de toutes les données à chaque itération) et la descente de gradient stochastique (qui n'utilise qu'une seule donnée à chaque itération). La descente de gradient par lots (mini-lots, mini-batch) est une méthode intermédiaire : on divise les données par paquets detaille m. Pour chaque paquet (appelé «lot(batch)»), on calcule un gradient et on effectue une itération.
Au bout de n/m itérations, on a parcouru tout le jeu de données : cela s'appelle une époque(epoche).
La formule est donc :

$$w_{k+1} \longleftarrow w_k - \delta_k \nabla (E_{j0+1} + E_{j0+2} + ... + E_{j0+m})(w_k) \tag{1.4.6}$$

pour w_{k+2}, on part de w_{k+1} et on utilise le gradient de la fonction

$$E_{j0+m+1} + E_{j0+m+2} + ... + E_{j0+2m} \tag{1.4.7}$$

Remarque 1.4.3. *On a*

1. *Pour $m = 1$, c'est exactement la descente de gradient stochastique.Pour $m = n$, c'est la descente de gradient classique.*
2. *Cette méthode combine le meilleur des deux mondes : la taille des données utilisées à chaque itération peut être adaptée à la mémoire et le fait de travailler par lots évite les pas erratiques de la descente stochastique pure.*

1.4.3 Accélération(Moment, Nesterov)

Le choix du pas δ n'est pas la seule amélioration possible de la méthode du gradient, nous allons voir comment la modifier à l'aide du «moment». Commençons par revenir

à l'analogie de la descente du gradient classique qui correspond à une goutte d'eau qui descend une montagne : la goutte emprunte le chemin qui suit la courbe de plus grande pente, quitte à serpenter et osciller lors de la descente. Imaginons que l'on lance maintenant une balle assez lourde du haut de la même montagne. Cette balle va suivre, comme la goutte d'eau,le chemin de la plus forte pente, mais une fois lancée elle va acquérir de l'inertie, appelée moment, qui va atténuer ses changements de direction. Ainsi la balle ne s'embarrasse pas des petits aléas du terrain et dévale la pente plus rapidement que la goutte d'eau.
Nos petits aléas de terrain nous viennent du fait que l'on ne calcule pas exactement le gradient de la fonction d'erreur en utilisant tout le jeu de données à chaque fois, mais seulement un échantillon. Cela peut conduire à certains gradients mal orientés. L'inertie de la balle est en quelque sorte la mémoire de la trajectoire passée qui corrige les mouvements erratiques.

Moment(Momentum SGD)

L'objectif principal de la méthode dite de Moment [2] est d'accélérer le processus de descente de gradient, et ceci en rajoutant un vecteur de vélocité.
La formule de la descente de gradient avec moment est :

$$v_{k+1} \longleftarrow \mu v_k - \delta \nabla E(w_k)$$

$$w_{k+1} \longleftarrow w_k - v_{k+1}$$

où $\mu, \delta \in \mathbb{R}$. Le vecteur v_{k+1} est calculé au début de chaque itération et représente la mise à jour de la vélocité d'une "balle dévalant une pente".

Nesterov

Dans la méthode précédente, le moment et le gradient sont calculés au même point w_k. La méthode de Nesterov[3] est une variante de cette méthode. Elle consiste à appliquer-d'abord le moment, pour obtenir un point w'_k , puis de calculer le gradient en ce point(et non en w_k). La formule est donc :

$$w'_k \longleftarrow w_k + \mu v_k$$

$$v_{k+1} \longleftarrow \mu v_k - \delta \nabla E(w'_k)$$

$$w_{k+1} \longleftarrow w_k + v_{k+1}$$

C'est un petit avantage par rapport à la méthode du moment puisqu'on calcule le gradient au point w'_k qui est censé être plus près de la solution w_{min} que w_k.

1.4.4 Extensions de la descente de gradient

Il existe plusieurs variantes de l'algorithme de descente de gradient. Dans la suite, nous présentons trois méthodes qui sont Adagrad, RMSProp, Adam.

Adagrad

Le principe de la méthode Adagrad est de faire que le taux d'apprentissage s'adapte aux paramètres, faisant de sorte qu'il s'ajuste automatiquement, en fonction de "l'éparsité" des paramètres. Adagrad abaisse progressivement le taux d'apprentissage mais pas de la même manière pour tous les paramètres : les dimensions à pente plus prononcée voient leur taux abaissé plus rapidement que celles à pente douce. Plus formellement, le pas est décrit par :

$$\forall i, (w_{k+1})_i \longleftarrow (w_k)_i - \alpha \frac{(\nabla E(w_k))_i}{\sqrt{\sum_{j=1}^{k} (\nabla E(w_j))_i^2}}, \alpha > 0$$

RMSProp

Cet algorithme ajuste automatiquement le taux d'apprentissage à chaque paramètre, commeA dagrad. Cependant, il ne cumule que les gradients issus des itérations récentes. Pour cela, il utilise une moyenne glissante :

$$\forall i, (\nabla_{k+1})_i \longleftarrow \delta(\nabla_k)_i - (1-\delta)(\nabla E(w_k))_i^2$$

$$\forall i, (w_{k+1})_i \longleftarrow (w_k)_i - \alpha \frac{(\nabla E(w_k))_i}{\sqrt{(\nabla_{k+1})_i}}, \alpha > 0$$

Adagrad modifie le taux d'apprentissage à chaque itération k pour chaque paramètre w_i. Ici, $\delta(\nabla_k)_i$ est la moyenne quadratique glissante du gradient. La division du gradient de la fonction objective par la racine de l amoyenne quadratique glissante (c'est à dire l'amplitude) améliore la convergence.

Adam

Adam[4] est l'un des algorithmes les plus récents et les plus efficaces pour l'optimisation par descente de gradient. Le principe est le même que pour Adagrad et RMSProp : il adapte automatiquement le taux d'apprentissage pour chaque paramètre. Sa particularité est de calculer (m_k, v_k) des "estimations adaptatives des moments".Il peut donc être vu comme une généralisation de l'algorithme Adagrad :

$$\forall i, (m_{k+1})_i \longleftarrow \beta_1(m_k)_i + (1-\beta_1)(\nabla E(w_k))_i^2$$

$$\forall i, (v_{k+1})_i \longleftarrow \beta_2(v_k)_i + (1-\beta_2)(\nabla E(w_k))_i^2$$

$$\forall i, (w_{k+1})_i \longleftarrow (w_k)_i - \alpha \frac{\sqrt{(1-\beta_2)}}{1-\beta_1} \frac{(m_k)_i}{\sqrt{(v_k)_i + \epsilon}}$$

$$\alpha, \epsilon > 0, \beta_1, \beta_2 \in]0, 1[$$

Ici m_k est le premier moment du gradient (la moyenne) et v_k est son second moment (variance non-centrée). ϵ est un paramètre de précision. Les paramètres β_1 et β_2 sont utilisés pour réaliser des moyennes d'exécution sur les moments m_k et v_k respectivement.

1.4.5 Algorithme de Rétro-propagation du gradient

L'algorithme d'apprentissage par rétro-propagation du gradient de l'erreur est un algorithme itératif qui a pour objectif de trouver les poids des connexions minimisant l'erreur quadratique moyenne commise par le réseau sur l'ensemble d'apprentissage [5] .Cette minimisation par une méthode du gradient conduit à l'algorithme d'apprentissage de rétro-propagation. Avant d'énoncer l'algorithmede rétro-propagation, nous citons les différentes notations utilisées dans celui-ci, qui sont :

Le problème de l'apprentissage pour les perceptrons multicouches consiste à reconnaitre la contribution de chaque poids sur l'erreur globale du réseau, et l'algorithme de rétro-propagation du gradient permet de faire cela [9] :

1. La propagation de l'information de l'entrée jusqu'à la sortie.
2. Le calcul de l'erreur en sortie.
3. La rétro-propagation de l'erreur de la sortie jusqu'aux entrées.

Nous décrirons dans la suite les différentes équations qui permettent de faire la mise à jour des pondérations de notre réseau.

Pour la couche de sortie :

Prenons un neurone j de la couche de sortie,lorsque le n^{me} exemple est lui présenté, il produit une sortie $y_j(n)$ alors que, c'est la valeur $d_j(n)$ qui est attendue,Notons :

$$E_j(n) = d_j(n) - y_j(n) \tag{1.4.8}$$

Et de ce fait, l'erreur globale du réseau pour l'exemple n est :

$$E(n) = \frac{1}{2}\sum_{j=1}^{m} E_j^2(n) = \frac{1}{2}||d(n) - y(n)||_2^2 \tag{1.4.9}$$

où m est le nombre de neurones dans la couche de sortie. Le but de l'apprentissage est de minimiser l'erreur moyenne correspondante aux N exemples d'apprentissage, c'est à dire minimiser :

$$E_{moy} = \frac{1}{N}\sum_{n=1}^{N} E(n) \tag{1.4.10}$$

On note :

$$u_i(n) = (\sum_i w_{ij}(n) \times y_i(n)) - b_j(n) \tag{1.4.11}$$

L'entrée totale d'un neurone dont la fonction d'activation est f_j, sa sortie s'écrit alors sous la forme de

$$y_j(n) = f_j(u_i(n))$$

Le mécanisme de rétro-propagation est basé sur l'affectation d'une correction $\Delta w_{ji}(n)$ et $\Delta b_j(n)$ aux poids et aux biais, on utilisera la règle dite du delta (delta rule), c'est à dire que la modification sera proportionnelle au gradient suivant :

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial E_j(n)} \times \frac{\partial E_j(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial u_j(n)} \times \frac{\partial u_j(n)}{\partial w_{ji}(n)}$$

Qui détermine la direction dans laquelle on recherche les valeurs de w_{ij}, nous appellerons taux d'apprentissage le facteur de proportionnalité et nous le noterons η par la suite. d'après l'équation 1.4.6, on peut écrire :

$$\frac{\partial E(n)}{\partial E_j(n)} = E_j(n)$$

De plus :

$$\frac{\partial y_j(n)}{\partial u_j(n)} = f'_j(u_j(n))$$

Finalement et à partir de 1.4.8 :

$$\frac{\partial u_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

Soit $\delta_j(n)$ le gradient local défini par :

$$\delta_j(n) = -\frac{\partial E(n)}{\partial E_j(n)} \times \frac{\partial E_j(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial u_j(n)} \tag{1.4.12}$$

D'après ce qui précède ceci est égal à :

$$\delta_j(n) = E_j(n) \times f'_j(u_j(n)) \tag{1.4.13}$$

La correction appliquée au poids $w_{ji}(n)$ sera donc la suivante (règle du delta) :

$$\Delta w_{ji}(n) = -\eta \times \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{1.4.14}$$

Soit encore :

$$\Delta w_{ji}(n) = \eta \times \delta_j(n) y_i(n) \tag{1.4.15}$$

Du même, la correction apportée au biais s'écrira sous la forme :

$$\Delta b_j(n) = \eta \times \frac{\partial E(n)}{\partial b_j(n)} \tag{1.4.16}$$

En suivant un raisonnement analogue à ce qui précède on obtient :

$$\frac{\partial E(n)}{\partial b_j(n)} = \frac{\partial E(n)}{\partial E_j(n)} \times \frac{\partial E_j(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial u_j(n)} \times \frac{\partial u_j(n)}{\partial b_j(n)}$$

$$\frac{\partial E(n)}{\partial b_j(n)} = -\delta_j(n) \times \frac{\partial u_j(n)}{\partial b_j(n)}$$

$$\frac{\partial E(n)}{\partial b_j(n)} = \delta_j(n)$$

D'où :

$$\Delta b_j(n) = -\eta.\delta_j(n)$$

Ces deux équations de mise à jour ne sont valables que pour la couche de sortie, car on a utilisé l'erreur de sortie $E_j(n) = d_j(n) - y_j(n)$

Pour les couches cachées :

En ce qui concerne les autres couches, on ne peut plus utiliser cette formule pour l'erreur, alors nous serons forcement obligés de déterminer les nouvelles équations de mise à jour.

Considérons un neurone j sur la dernière couche cachée, (celle qui se place juste avant celle de sortie.), nous pouvons et en utilisant une formule analogue à 1.4.10 définir le gradient local par :

$$\delta_j(n) = \frac{\partial E(n)}{\partial y_j(n)}) \times \frac{\partial y_j(n)}{\partial u_j(n)}$$

$$\delta_j(n) = \frac{\partial E(n)}{\partial y_j(n)}) \times f'_j(u_j(n))$$

Et d'après 1.4.6 on peut écrire :

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k E_k(n) \times \frac{\partial E_k(n)}{\partial y_j(n)}$$

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k E_k(n) \times \frac{\partial E_k(n)}{\partial u_k(n)} \times \frac{\partial u_k(n)}{\partial y_j(n)}$$

Or, le neurone k est sur la couche de sortie, d'où :

$$\frac{\partial E_k(n)}{\partial u_k(n)} = \frac{\partial (d_k(n) - y_k(n))}{\partial u_k(n)}$$

$$\frac{\partial E_k(n)}{\partial u_k(n)} = \frac{\partial (d_k(n) - f_k(u_k(n)))}{\partial u_k(n)}$$

$$\frac{\partial E_k(n)}{\partial u_k(n)} = f'_k(u_k(n))$$

En outre, et d'après 1.4.8 :

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

Donc nous obtenons :

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k E_k(n) \times f'_k(u_k(n)) \times w_{kj}(n)$$

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k \delta_k(n) \times w_{kj}(n)$$

Le gradient local pour un neurone de la dernière couche cachée est alors donné par :

$$\delta_j(n) = (\sum_k \delta_k(n) \times w_{kj}(n)) \times f'_j(u_j(n))$$

Ceci peut se généraliser et sans aucun souci aux autres couches internes, nous obtenons ainsi les règles de mise à jour des poids et des biais avec la méthode de rétro-propagation du gradient.

Pour cette raison, les poids et les biais doivent être actualisés en utilisant :

$$w_{ji} \longleftarrow w_{ji} + \eta \times \delta_j(n) \times y_i(n)$$

$$b_j \longleftarrow b_j + \eta \times \delta_j(n)$$

Avec :

$$\delta_j(n) = E_j(n) \times f'_j(u_j(n))$$

Et :

$$E_j(n) = \begin{cases} d_j(n) - y_j(n) & \text{si j est un neurone de sortie.} \\ \sum_k \delta_k(n) w_{kj}(n) & \text{si j est un neurone interne.} \end{cases}$$

Ainsi, l'erreur de sortie se propage vers l'entrée (c'est à dire dans le sens inverse de ce lui qui est utilisé pour propager un signal à travers le réseau) a fin de corriger de couche en couche les valeurs des poids et des biais.

La mise en oeuvre de l'algorithme

L'algorithme de rétro-propagation du gradient et comme la suite :

Algorithm 2 La méthode de rétro-propagation du gradient (Back-propagation)

Initialisation : Initialiser tous les poids et les biais du réseau de neurones.

Présentation d'un exemple en entrée : Présenter un exemple en entrée $x(n) = (x_1(n), x_2(n), ..., x_p(n))$ ainsi que la sortie désirée correspondante $d(n) = (d_1(n), d_2(n), ..., d_m(n))$.

Calculer les sorties : calcules successivement les sorties des différentes couches pour l'entrée x.

La mise à jour des poids et des biais : modifser récursivement les poids et les biais du réseau suivant les règles détaillées avant :

$$w_{ji} \longleftarrow w_{ji} + \eta \times \delta_j(n) \times y_i(n)$$

$$b_j \longleftarrow b_j + \eta \times \delta_j(n)$$

Avec :

$$\delta_j(n) = E_j(n) \times f'_j(u_j(n))$$

Et :

$$E_j(n) = \begin{cases} d_j(n) - y_j(n) & \text{si j est un neurone de sortie.} \\ \sum_k \delta_k(n) w_{kj}(n) & \text{si j est un neurone interne.} \end{cases}$$

Critère d'arrêt : Répéter les étapes 2, 3 et 4 jusqu'à un nombre maximum d'itérations ou jusqu'à ce que l'erreur quadratique moyenne soit inférieure à un certain seuil.

1.5 Apprentissage des réseaux de neurone artificiels

L'apprentissage des réseaux de neurone sartificiels est une phase qui permet de déterminer ou de modifier les paramètres du réseau, afin d'adopter un comportement désiré. Cela est réalisé en adaptant, grâce à certaines règles, les vecteurs de poids. Plusieurs algorithmes d'apprentissage ont été développés depuis la première règle d'apprentissage de Hebb en 1949 [4]. Il existe essentiellement deux sortes d'apprentissages (selon les exemples sont étiquetés ou non) :

— L'apprentissage supervisé

— L'apprentissage non supervisé

1.5.1 Apprentissage supervisé

L'apprentissage dit supervisé est caractérisé par la présence d'un ensemble de N exemples étiquetés$\{(x_1, d_1), (x_2, d_2), ..., (x_N, d_N)\}$.
où x_i désigne un stimulus (entrée) et d_i la cible pour ce stimulus, c'est-à-dire la sortie désirée du réseau.Chaque couple (x_i, d_i) correspond donc à un cas d'espèce de ce que le réseau devrait produire (lacible) pour un stimulus donné. Pour cette raison, l'apprentissage supervisé est aussi qualifié d'apprentissage par des exemples.
On peut distinguer deux grands types d'apprentissage supervisé : la classification et la régression.

Classification

Lorqu'on fait de la classification, l'entrée est l'instance d'une classe et l'étiquette est la classe correspondante. En reconnaissance de caractères, par exemple, l'entrée serait une suite de pixels représentant une lettre et la classe serait la lettre représentée (ou son index).
La classification consiste donc à apprendre une fonction f_{class} de $\mathbb{R}^d$ dans $\mathbb{N}$ qui associe à un vecteur sa classe. Dans certains cas, on pourra vouloir que la fonction f_{class} soit à valeurs dans $[0, 1]^k$ telle que chaque élément du vecteur de sortie représente la probabilité d'appartenance à une classe (la somme des éléments sera donc 1).

$$f_{class} : \quad \mathbb{R}^d \longrightarrow \mathbb{N}$$

$$entree \longmapsto f_{class}(entree) = classe$$

Régression

Dans le cas de la régression, l'entrée n'est pas associée à une classe mais à une ou plusieurs quantités continues. Ainsi, l'entrée pourrait être les caractéristiques d'une personne (son âge, son sexe,son niveau d'études) et l'étiquette son revenu.
La régression consiste donc à apprendre une fonction f_{regr} de $\mathbb{R}^d$ dans $\mathbb{R}^k$ qui associe à un vecteur sa valeur associée.

$$f_{regr} : \quad \mathbb{R}^d \longrightarrow \mathbb{R}^k$$

$$entree \longmapsto f_{regr}(entree) = valeur$$

1.5.2 Apprentissage non supervisé

L'apprentissage non supervisé des réseaux de neurones consiste, comme dans le cas d'apprentissage supervisé, à modifier les poids des connections des neurones. Dans ce cas, les exemples de base d'apprentissage sont des données seules : il n'est pas possible de modifier les poids du réseau en fonction d'une erreur sur les réponses souhaitées, puisqu'aucune réponse n'est connue a priori.
La figure 1.10 illustre les deux types d'apprentissage : supervisé et non supervisé.

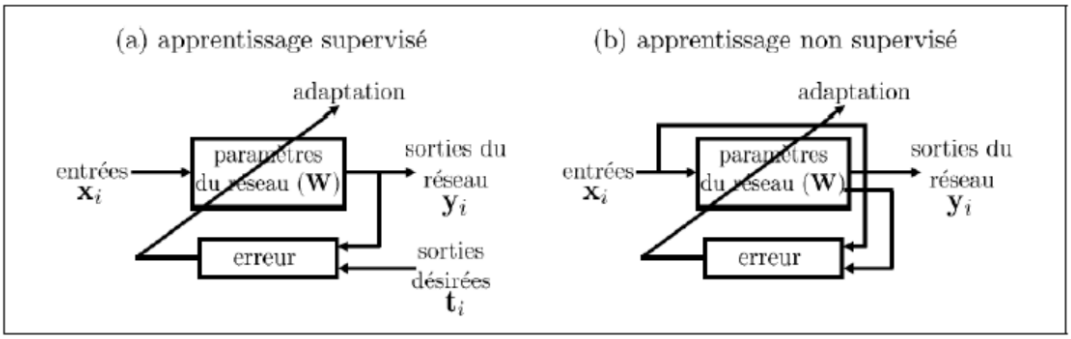

FIGURE 1.10 Synoptique de l'apprentissage supervisé et non supervisé d'un réseau de neurones artificiels

1.6 Réseaux de neurones convolutifs

L'architecture des ANN discutée précédemment est appelée réseaux de neurones FC (FCNN). La raison est que chaque neurone d'une couche i est connecté à tous les neurones des couches $i-1$ et $i+1$. Chaque connexion entre deux neurones a deux paramètres : le poids et le biais. Ajouter plus de couches et de neurones augmente le nombre de paramètres. Par conséquent, l'apprentissage de tels réseaux prend beaucoup de temps, même sur des appareils dotés de plusieurs unités de traitement graphique (GPU) et de plusieurs unités centrales de traitement (CPU). Il devient impossible de former de tels réseaux sur des PC avec des capacités de traitement et de mémoire limités.
Dans l'analyse de données multidimensionnelles telles que les images, lesCNN (également connus sous le nom de ConvNets) sont plus efficaces entermes de temps et de mémoire que les réseaux FC.
ANN est la base du CNN, avec quelques modifications ajoutées pour le rendre adapté à l'analyse de grandes quantités de données. La connexion de tous les neurones augmente le nombre de paramètres même lors de l'analyse de très petites images (par exemple, une de 150×150 pixels). La couche d'entrée dans ce cas aura 22500 neurones. La connecter à un autre couche cachée avec 500 neurones, le nombre de paramètres requis est de $22500 \times 500 = 11250000$. Les applications du monde réel peuvent fonctionner avec des images à haute dimension où la plus petite dimension peut avoir 1000 pixels et plus.
Pour une image d'entrée detaille 1000×1000 et une couche cachée de 2000 neurones, le nombre de paramètres est égal à 2 milliards. Notez que l'image d'entrée est grise.

1.6.1 Algèbre de convolution

Cette section est consacrée à une étude théorique dans laquelle nous présenterons quelques définitions et propriétés mathématiques qui forment la base des réseaux de neurones convolutifs.

Théorème 1.6.1. *Soit $f, g \in L^1(\mathbb{R}^d)$, pour presque tout $x \in \mathbb{R}^d$, $y \longmapsto f(y)g(x-y)$ est intégrable et la fonction :*

$$(f \star g)(x) := \int_{\mathbb{R}^d} f(y)g(x-y)dy \tag{1.6.1}$$

est elle même intégrable sur $\mathbb{R}^d$, de plus :

$$||f \star g||_1 < ||f||_1 ||g||_1$$

Preuve 1. *Par le théorème de Tonelli [9], $f \otimes g : (x,y) \longmapsto f(y)g(x)$ est intégrable sur $\mathbb{R}^d \otimes \mathbb{R}^d$ et $||f \otimes g||_{L^1(\mathbb{R}^d \otimes \mathbb{R}^d)} = ||f||_{L^1(\mathbb{R}^d)} ||g||_{L^1(\mathbb{R}^d)}$.*
L'application :

$$\phi : (x,y) \longmapsto (x-y, y)$$

est un $\mathcal{C}^1-$difféomorphisme de $\mathbb{R}^d \times \mathbb{R}^d$ de jacobien 1, donc par le théorème de changement de variable :

$$||f \otimes g||_{L^1(\mathbb{R}^d \times \mathbb{R}^d)} = ||(f \otimes g) \circ \phi||_{L^1(\mathbb{R}^d \times \mathbb{R}^d)} \tag{1.6.2}$$

$$||f \otimes g||_{L^1(\mathbb{R}^d \times \mathbb{R}^d)} = \int_{\mathbb{R}^d \times \mathbb{R}^d} |f(y)g(x-y)|dxdy \tag{1.6.3}$$

Donc, la fonction $y \longmapsto f(y)g(x-y)$ est Lebesque intégrable sur $\mathbb{R}^d \times \mathbb{R}^d$ et la conclusion est une conséquence directe du théorème de Fubini.

Définition 1.6.1. *Le produit de convolution de deux fonctions réelles ou complexes f et g, est une autre fonction, qui se note $f \star g$ et qui est définie par :*

$$(f \star g)(x) := \int_{\mathbb{R}^d} f(y)g(x-y)dy \tag{1.6.4}$$

Proposition 1.6.1. *le produit de convolution est associatif et commutatif.*

Preuve 2. *soit $f, g \in L^1(\mathbb{R}^d)$*

- commutativité : *soit N un négligeable tel que :*

$$y \longmapsto f(y)g(x-y) \quad et \quad y \longmapsto g(y)f(x-y)$$

soient intégrables pour tout $x \in \mathbb{R}^d \setminus N$.
L'application

$$\phi_x : y \longmapsto x - y$$

est un $\mathcal{C}^1-$difféomorphisme sur $\mathbb{R}^d$ de Jacobien $(-1)^d$, donc pour tout $x \in \mathbb{R}^d \setminus N$ on a :

$$(f \star g)(x) = \int_{\mathbb{R}^d} f(y)g(x-y)dy$$

$$(f \star g)(x) = \int_{\mathbb{R}^d} f(\phi_x(y))g(x-\phi_x(y))dy$$

$$(f \star g)(x) = \int_{\mathbb{R}^d} f(x-y)g(x)dy$$

$$(f \star g)(x) = (g \star f)(x)$$

- associativité : *pour presque tout $x \in \mathbb{R}^d$, on a :*

$$(f \star g) \star h(x) = \int_{\mathbb{R}^d} (f \star g)(y) h(x-y) dy$$

$$(f \star g) \star h(x) = \int_{\mathbb{R}^d} (\int_{\mathbb{R}^d} f(z) g(y-z) dz) h(x-y) dy \quad (1)$$

ou $\int_{\mathbb{R}^d} f(z)g(y-z)dz$ est définie pour presque tout $x \in \mathbb{R}^d$, de même :

$$f \star (g \star h)(x) = \int_{\mathbb{R}^d} f(z)(g \star h(x-z)) dz$$

$$f \star (g \star h)(x) = \int_{\mathbb{R}^d} f(z)(\int_{\mathbb{R}^d} g(y) h(x-z-y) dy) dz \quad (2)$$

on passe de (1) à (2) par changement de variable $y \longmapsto y - z$ et théorème de Fubini. Par le théorème de Tonelli :

$$(x, y, z) \longmapsto (f \otimes g \otimes h) \circ \phi(x, y, z) = f(x)g(y)h(z)$$

est intégrable sur $\mathbb{R}^{3d}$.
En considérant le difféomorphisme :

$$\phi(x, y, z) = (x - y - z, y, z)$$

qui vérifie $|det(D\phi)| = 1, (f \otimes g \otimes h \circ \phi)$ est également intégrable. Par le théorème de Fubini cette fonction est intégrable par rapport à (y,z) et pour presque tout $x \in \mathbb{R}^d$

$$f \star (g \star h)(x) = \int_{\mathbb{R}^d \times \mathbb{R}^d} (h \otimes g \otimes f) \circ \phi(x, y, z) dy dz$$

par composition avec le difféomorphisme $\Theta : (y, z) \longmapsto (y - z, z)$, de Jacobien 1, on a pour presque tout x :

$$f \star (g \star h)(x) = \int_{\mathbb{R}^d \times \mathbb{R}^d} (h \otimes g \otimes f) \circ \phi \circ \Theta(x, y, z) dy dz$$

$$f \star (g \star h)(x) = \int_{\mathbb{R}^d} (\int_{\mathbb{R}^d} (h \otimes g \otimes f) \circ \phi \circ \Theta(x, y, z) dy) dz \qquad (Fubini)$$

$$f \star (g \star h)(x) = (f \star g) \star h(x)$$

Définition 1.6.2. *(Cas discret)*
Soient $(f(n))_{n\in\mathbb{Z}}$ et $(g(n))_{n\in\mathbb{Z}}$ deux suites de nombres réels. Le produit de convolution est la suite $(h(n))_{n\in\mathbb{Z}}$ dont le terme général est défini par :

$$(f \star g)(x) = \sum_{k=-\infty}^{+\infty} f(n-k) g(k) \quad (1.6.5)$$

Remarque 1.6.1. *On a*

1. *Lorsque l'on suppose que les termes de g sont nuls en dehors des indices appartenant à $[-K, +K]$:*

$$(f \star g)(x) = \sum_{k=-K}^{+K} f(n-k) g(k) \quad (1.6.6)$$

   c'est le cas le plus utilisé, en effet la suite $(g(n))_{n\in\mathbb{Z}}$ représente le motif (Kernel).
2. *Le produit de convolution dans le cas discret est commutatif et associatif.*

1.6.2 Convolution pour l'imagerie

Une image est un tableau de nombre, une image gris est représentée par un tableau à deux dimension et une image couleur (RGB) est représentée par un tableau à trois dimensions.

Notations 1. *on désigne par :*
I : une image.
A : la matrice associé a l'image I.
M : le motifs (kernel, filtre, masque).

Définition 1.6.3. *Convolution (deux dimensions)*
La convolution en deux dimensions est une opération qui :

1. *à partir d'une matrice d'entrée A*
2. *et d'une matrice d'un motif M*

associe une matrice de sortie $A \star M$ donnée par :

$$(A \star M)(i,j) = \sum_n \sum_m A(i-n, j-m) M(n,m) \tag{1.6.7}$$

Tout d'abord, il faut retourner la matrice M :

FIGURE 1.11 Motif retourner

Le calcul de $A \star M$ s'effectue coefficient par coefficient :

1. On centre le motif retourné sur la position du coefficient à calculer.
2. On multiplie chaque coefficient de A par le coefficient du motif retourné en face.
3. La somme de ces produits donne un coefficient de $A \star M$

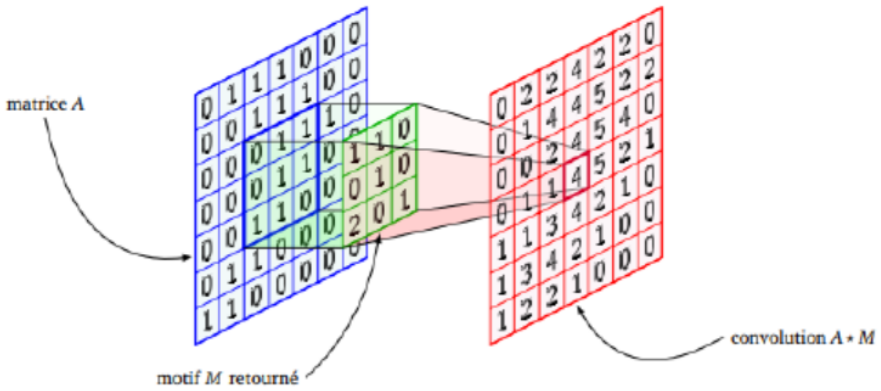

FIGURE 1.12 Calcul d'un coefficient de $A \star M$

Exemple 1.6.1. *Voici un exemple de calcul*

Translation : *Une convolution par la matrice*

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

correspond à une translation des coefficients vers le bas. Par exemple :

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \star \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

De même pour une translation des coefficients vers la droite.

Moyenne : *On effectue une moyenne locale des coefficients à l'aide du motif :*

$$\frac{1}{9}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

On obtient une image floue par application du motif moyenne :

FIGURE 1.13 Application du filtre moyenne

Remarque 1.6.2. *Parfois, il peut être intéressant de conserver une certaine dimension dans les tailles des images en sortie des convolutions. Le padding consiste simplement à ajouter des zéro tout autour d'une matrice (image) pour en augmenter la taille.*
Par exemple,

Convolution de même taille : *La matrice $B = A \star M$ est de même taille que la matrice A. Pour les calculs, on peut être amené à rajouter des zéro svirtuels sur les bords de la matrice A.*

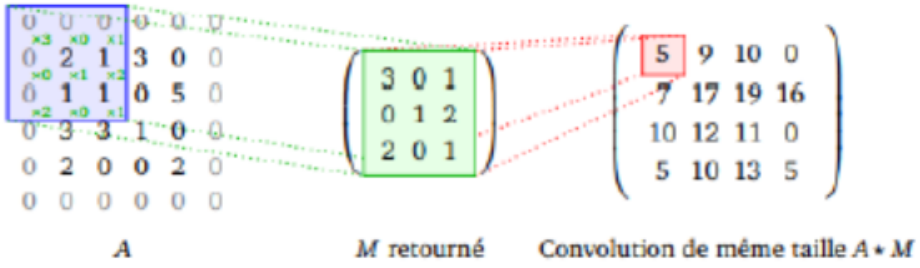

FIGURE 1.14 Convolution de même taille

Convolution étendue : *On rajoute deux rangées de zéros virtuels autour de la matrice A. Si A est de taille $n \times p$ et M est de taille 3×3 alors pour cette convolution la matrice B est de taille $(n+2) \times (p+2)$*

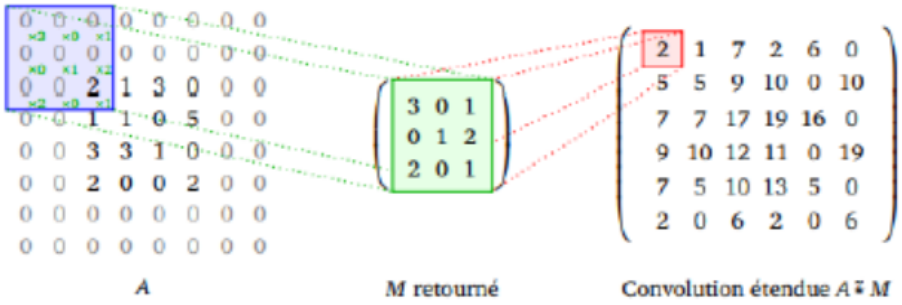

FIGURE 1.15 Convolution étendue

Convolution restreinte : *On ne s'autorise pas à rajouter des zéros virtuels. Pour cette convolution la matrice B est donc de taille $(n-2) \times (p-2)$*

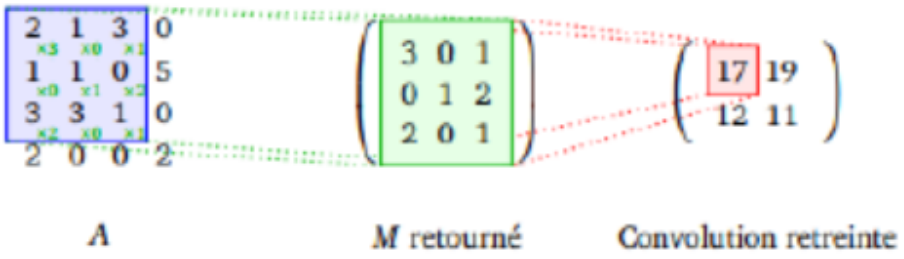

FIGURE 1.16 Convolution restreinte

1.6.3 Sous-échantillonnage (Pooling)

Le pooling (regroupement de termes) consiste à transformer une matrice en une matrice plus petite tout enssayant d'en garder les caractéristiques principales.
Un pooling de taille k transforme une matrice de taille $n \times p$ en une matrice de taille k fois plus petite, c'est-à-dire de taille $n/k \times p/k$. Une sous-matrice de taille $k \times k$ de la matrice de départ produit un seul coefficient de la matrice d'arrivée.

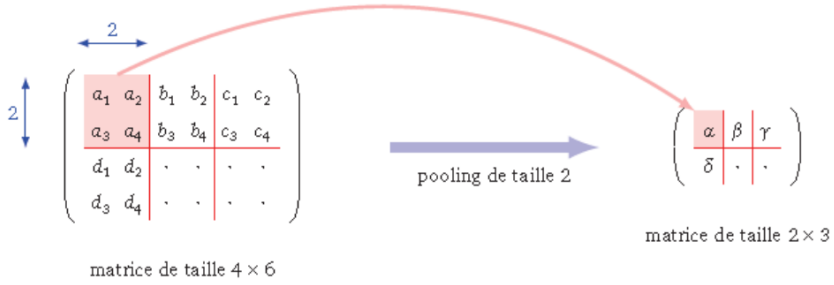

FIGURE 1.17 Pooling de taille 2×2

On distingue deux types de pooling :

1. Le max-pooling de taille k consiste à retenir le maximum de chaque sous-matrice de taille $k \times k$
2. Le pooling en moyenne de taille k (average pooling) consiste à retenir la moyenne des termes de chaque sous-matrice de taille $k \times k$

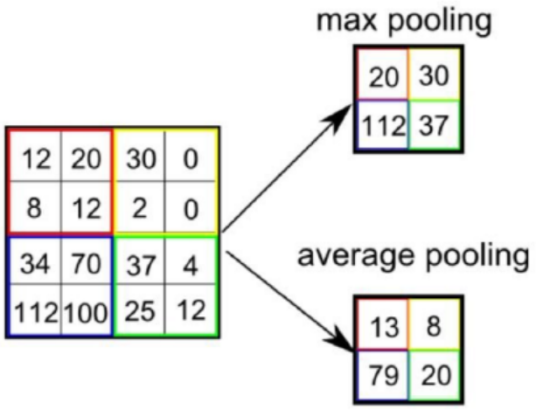

FIGURE 1.18 Max-pooling et average-pooling de taille 2×2

Le max-pooling, qui ne retient que la valeur la plus élevée par sous-matrice, permet de détecter la présence d'une caractéristique (par exemple un pixel blanc dans une image noire).Tandis que le pooling en moyenne prend en compte tous les termes de chaque sous-matrice (par exemple avec 4 pixels d'une image de ciel, on retient la couleur moyenne).

1.6.4 Neurone et couche de convolution

Neurone de convolution

À l'aide de la convolution, nous allons définir un nouveau type de couche de neurones : une couche de convolution. Tout d'abord un neurone de convolution de taille 3×3 est un neurone classique ayant 9 entrées.

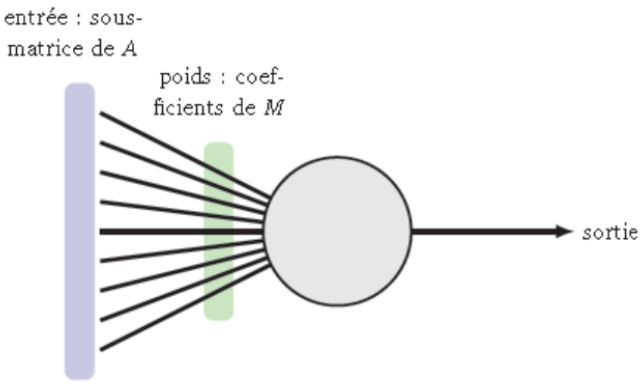

FIGURE 1.19 Neurone de convolution 1

Les poids du neurone correspondent aux coefficients d'une matrice de convolution (le motif) :

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \tag{1.6.8}$$

Imaginons que l'entrée soit une image ou un tableau de dimension 2, pour nous ce sera une matrice A. Alors un neurone de convolution est relié à une sous-matrice 3×3 de A.

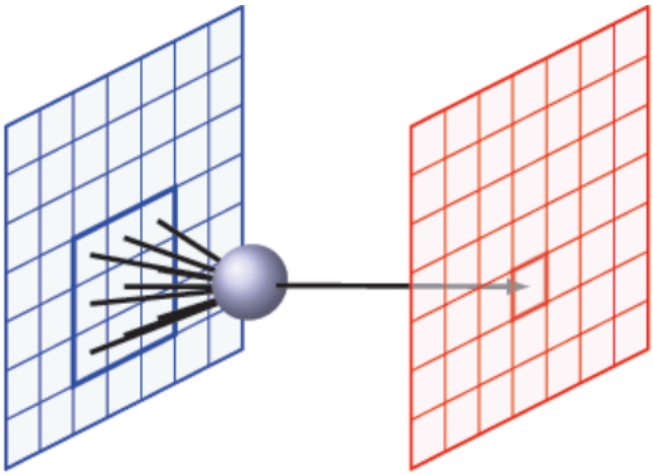

FIGURE 1.20 Neurone de convolution 2

Ce neurone agit comme un élément de convolution. Par exemple si la sous-matrice de A est notée

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{1.6.9}$$

alors le neurone produit la sortie :

$$s = a_{11}m_{33} + a_{12}m_{32} + a_{13}m_{31} + a_{21}m_{23} + a_{22}m_{22} + a_{23}m_{21} + a_{31}m_{13} + a_{32}m_{12} + a_{33}m_{11}$$

N'oublier pas que dans le produit de convolution, on commence par retourner la matrice M.

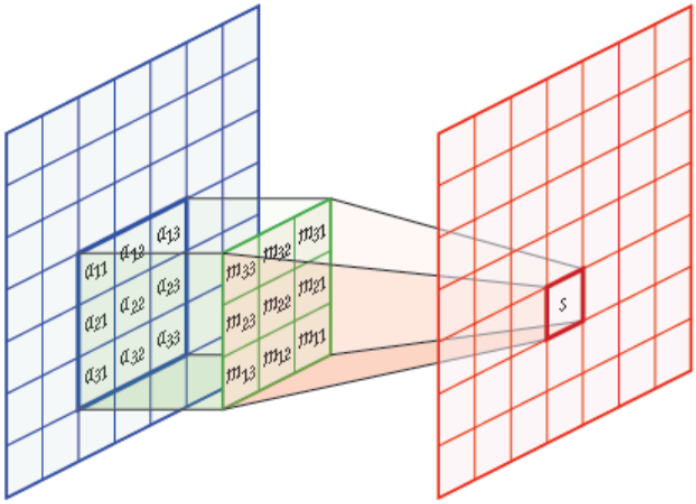

FIGURE 1.21 Neurone de convolution 3

On pourrait aussi composer par une fonction d'activation au niveau du neurone, mais on le fera plus tard à l'aide d'une « couche d'activation ». Plus généralement un neurone de convolution de taille $p \times q$ possède pq arêtes et donc pq poids à définir ou à calculer.

Couche de convolution

Considérons une entrée A représentée par une matrice de taille $n \times p$.
Une couche de convolution (pour un seul motif) est la donnée d'une matrice M appelé motif (par exemple de taille 3×3) et qui renvoie en sortie les coefficients de la matrice $A \star M$.

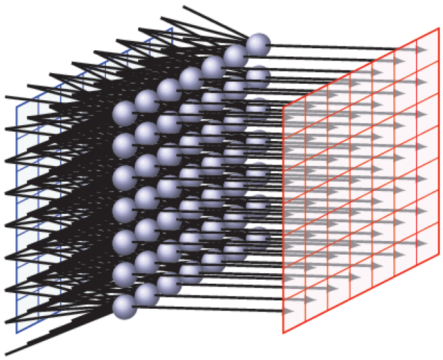

FIGURE 1.22 Couche de convolution

Pour une entrée de taille $n \times p$, il y a donc np neurones de convolutions, chacun ayant 9 arêtes (car le motif est de taille 3×3). Il est très important de comprendre que les poids sont communs à tous les neurones (ce sont les coefficients de M). Ainsi, pour une couche de convolution, il y a seulement 9 poids à déterminer pour définir la couche de convolution (bien que le nombre de neurones puisse être très grand).

Remarque 1.6.3. *On a*

1. *Terminologie : le motif M s'appelle aussi le noyau (kernel), le filtre ou encore le masque.*
2. *Le motif peut être d'une taille différente de la taille* 3×3 *considérée dans les exemples qui est cependant la plus courante.*
3. *Il existe également des variantes avec biais ou fonction d'activation.*
4. *Combinatoire : dans le cas d'une couche complètement connectée, le nombre de poids à calculer serait énorme. En effet, une couche de np neurones complètement connectée à une entrée de taille* $n \times p$ *amènerait à calculer* $(np)^2$ *poids. Pour* $n = p = 100$*, cela fait 10000 neurones et 100000000 poids. Pour rappel, notre couche de convolution est définie par 9 poids (quels que soient n et p).*
5. *D'un point de vue mathématique, une couche de convolution associée au motif M est l'application :*

$$F : \mathcal{M}_{n,p} \longmapsto \mathcal{M}_{n,p}$$

$$A \longmapsto A \star M$$

où $\mathcal{M}_{n,p}$ *est l'ensemble des matrices de taille* $n \times p$*. Si on transforme une matrice en un (grand) vecteur, alors on obtient une application* $F : \mathbb{R}^{np} \longmapsto \mathbb{R}^{np}$

Différentes couches d'un réseau de neurones

Plusieurs filtres : dans la pratique qu'une couche de convolution est définie avec plusieurs motifs, c'est-à-dire un nombre ℓ de matrices $M_1, M_2, ..., M_\ell$. Ainsi pour une entrée A de taille $n \times p$, une couche de convolution à ℓ motifs renvoie une sortie de taille $n \times p \times \ell$, correspondant à $(A \star M_1, A \star M_2, ..., A \star M_\ell)$.

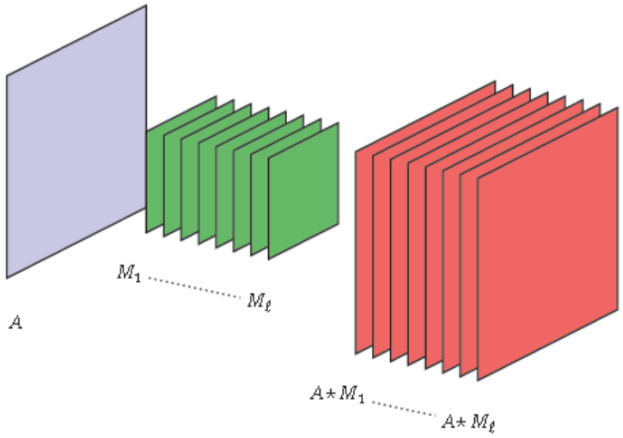

FIGURE 1.23 Plusieurs filtres.

Convolution à plusieurs filtres à partir de plusieurs canaux : C'est le cas général dans la pratique. Une entrée donnée par plusieurs canaux $\mathcal{A} = (A_1, A_2, ..., A_k)$, associée à des motifs $M_1, M_2, ..., M_\ell$ (qui sont donc chacun des 3-tenseurs de taille (3,3, k)) produit une sortie de taille $n \times p \times \ell$, correspondant à $(A \star M_1, A \star M_2, ..., A \star M_\ell)$. Si l'entrée $\mathcal{A}$ est de taille (n, p, k) alors la sortie est de taille $n \times p \times \ell$.

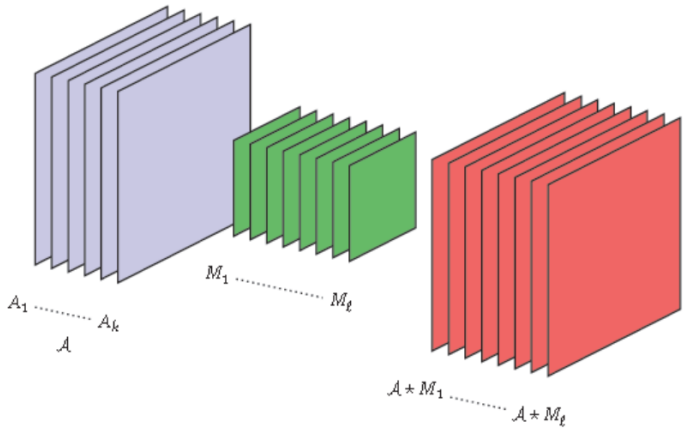

FIGURE 1.24 Convolution à plusieurs filtres à partir de plusieurs canaux.

Noter que sur cette figure chaque motif M_i est représenté par carré 3×3, alors qu'en fait chacun devrait être une boîte en 3 dimensions de taille $3 \times 3 \times$ k.

1.6.5 Structure des réseaux de neurones convolutifs

Un réseau neuronal convolutif CNN est un algorithme d'apprentissage enprofondeur qui peut prendre en compte une image d'entrée, attribuer une importance (poids et biais apprenables) à divers aspects/objets de l'image et être capable de les différencier les uns des autres. Le prétraitement requis dans un CNN est beaucoup plus faible par rapport aux autres algorithmes de classification. Alors que dans les méthodes primitives, les filtres sont conçus à la main, avec une formation suffisante, les CNN sont la capacité d'apprendre ces filtres/caractéristiques.
L'architecture d'un CNN est analogue à celle du modèle de connectivité des neurones dans le cerveau humainet s'inspire de l'organisation du cortex visuel. Les neurones individuels ne répondent aux stimuli que dans une région restreinte du champ visuel appelée champ récepteur. Une collection de ces champs se chevauchent pour couvrir toute la zone visuelle. Nous utilisons trois principaux types de couches pour créer des architectures CNNs : la couche de convolution, la couche de mise en commun(pooling) et la couche entièrement connectée.

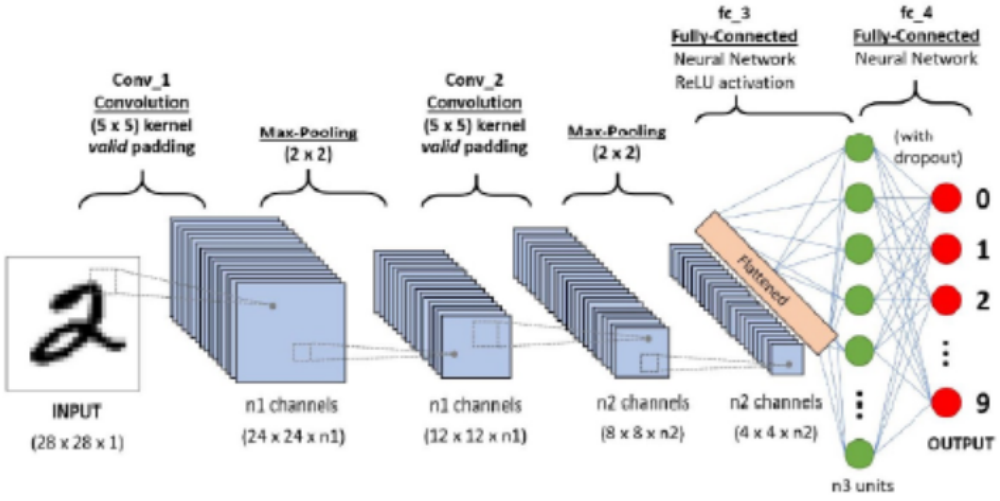

FIGURE 1.25 Architecture d'un réseau de neurones convolutif

Apprentissage et inférence de réseau CNN

L'implémentation d'un réseau CNN est réalisée en deux phases. La première phase est l'apprentissage, où le modèle apprend à faire la classification en utilisant l'algorithme de rétropropagation. La deuxième phase est l'inférence, où le modèle prédit la sortie de nouveaux échantillons de données, en utilisant le modèle appris. Nous pouvons voir les différents processus dans la Figure I.12. Habituellement, l'apprentissage est effectué une seule fois alors que l'inférence est exécutée chaque fois que le réseau CNN doit traiter une nouvelle entrée. Ainsi, la plupart des efforts doivent être concentrés sur l'accélération de la phase d'inférence.

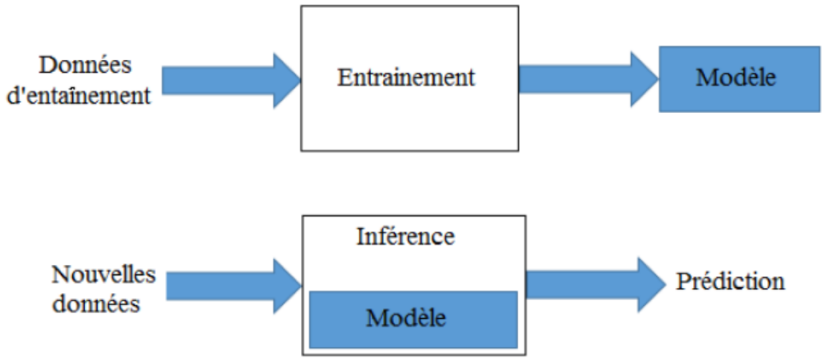

FIGURE 1.26 Les deux phases d'implémentation d'un réseau CNN.

Durant la phase d'apprentissage l'algorithme détermine les valeurs des filtres et des biais à utiliser pour les différentes couches de convolution et les couches entièrement connectées. Durant cette phase, l'algorithme utilise les paramètres suivants :

Epochs : un epoch correspond à un seul passage en avant et en arrière de la totalité des données d'apprentissage à travers le réseau. Puisque on utilise un ensemble de données limité et le Gradient Descent qui est un processus itératif, la mise à jour des poids avec un seul passage ou une seule epoch ne suffit pas. Donc il faut plusieurs epochs pour arriver aux valeurs optimales des poids.

Batch : pour les bases de données de grande taille, on ne pout pas transmettre l'intégralité de l'ensemble de données au réseau neuronal en une seule fois. Ainsi, on divise l'ensemble de données en nombre de lots ou parties appelé batch.

Iteration : une itération correspond au passage d'un seul batch. Donc, le nombre total d'itération est égale au nombre de batch par epoch multiplier par le nombre total d'epochs.

Exemples d'architectures de réseaux CNN

Différentes améliorations de l'architecture CNN ont été apportées depuis 1998 à ce jour. Cette section détaille certains des réseaux CNN bien connus. Ce sont des exemples de la façon de construire un CNN efficace.

LeNet-5 (1998) : LeNet-5, un réseau convolutionnel à 7 niveaux pionnier de LeCun et al en 1998, qui classe les chiffres, a été appliqué par plusieurs banques pour reconnaître les nombres manuscrits sur les chèques (chèques) numérisés en images d'entrée en niveaux de gris de 32 x 32 pixels. La capacité de traiter des images à plus haute résolution nécessite des couches plus grandes et plus convolutives, de sorte que cette technique est limitée par la disponibilité des ressources informatiques.

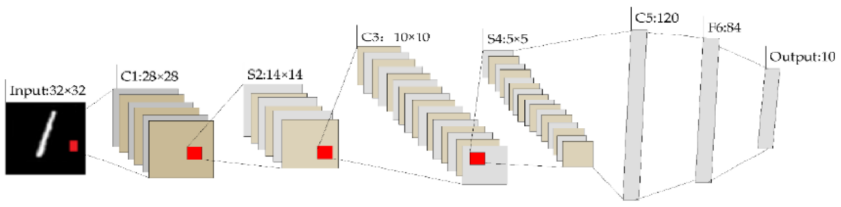

FIGURE 1.27 Architecture LeNet 5.

AlexNet (2012) : AlexNet est considérée comme la première architecture CNN profonde, qui a montré des résultats révolutionnaires pour les tâches de classification et de reconnaissance d'images. AlexNet a été proposé par Krizhevsky et al. Ce qui a amélioré la capacité d'apprentissage du CNN en l'approfondissant et en appliquant plusieurs stratégies d'optimisation des paramètres. La conception architecturale de base d'AlexNet est illustrée à la Figure 1.28.

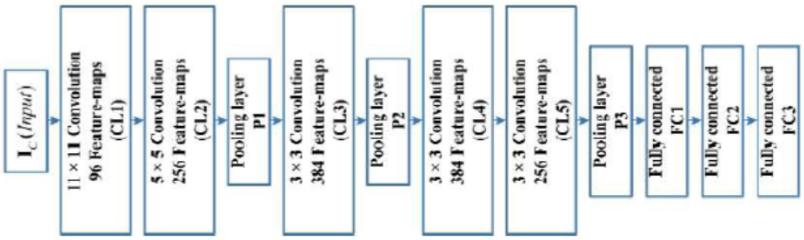

FIGURE 1.28 Architecture AlexNet.

VGGNet (2014) : Après que la technique CNN a été déterminée efficace dans le domaine de la reconnaissance d'images, un principe de conception simple et efficace pour CNN a été proposé par Simonyan et Zisserman. Cette conception innovante s'appelait Visual Geometry Group (VGG). L'architecture VGG utilise des filtres 3 × 3 plutôt que les filtres 5 × 5 et 11 × 11. Cela a montré expérimentalement que l'utilisation de ces filtres de petite taille pouvait produire la même influence que les filtres de grande taille. En général, VGG a obtenu des résultats significatifs pour les problèmes de localisation et de classification d'images. Cependant, le coût de calcul de VGG était excessif en raison de son utilisation d'environ 140 millions de paramètres, ce qui représentait son principal défaut.

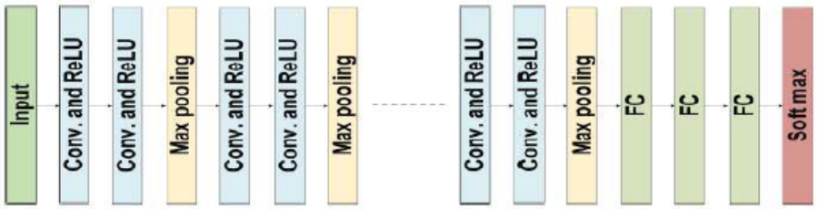

FIGURE 1.29 Architecture de VGG.

GoogleNet(2014) : GoogleNet a été le gagnant du concours 2014-ILSVRC et également connu sous le nom d'Inception-V1. L'objectif principal de l'architecture GoogleNet était d'atteindre une grande précision avec un coût de calcul réduit (Szegedy et al. 2015). Il a introduit le nouveau concept de bloc de création dans CNN, par lequel il incorpore des transformations convolutives à plusieurs échelles en utilisant des idées de division, de transformation et de fusion . La Figure 1.30 illustre l'architecture du bloc de base de l'architecture de GoogLeNet.

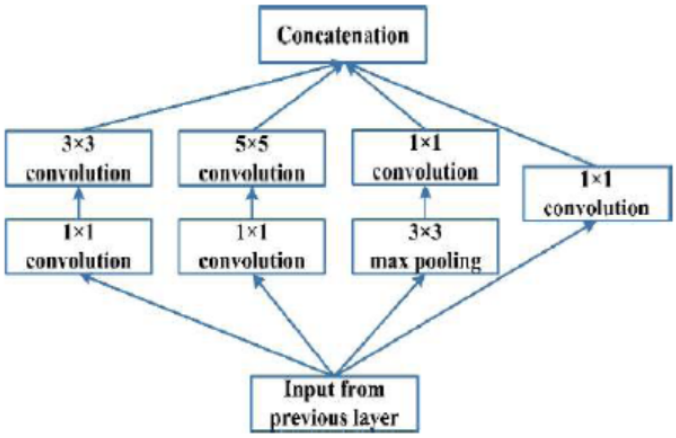

FIGURE 1.30 Structure de base d'un bloc du réseau GoogleNet.

ResNet (2015) : He et al ont développé le réseau ResNet (Residual Network), qui a remporté le prix ILSVRC 2015. Leur objectif était de concevoir un réseau ultra-profond par rapport aux réseaux précédents. Plusieurs types de ResNet ont été développés en fonction du nombre de couches (à partir de 34 couches et jusqu'à 1202 couches). Le type le plus courant était ResNet 50, qui comprenait 49 couches convolutifs plus une seule couche FC. Le nombre total de poids du réseau ResNet 50 est de 25,5 M, tandis que le nombre total de MAC est de 3,9 M. La nouvelle idée de ResNet est son utilisation du concept de voie de contournement, comme le montre la Figure 1.31 pour résoudre le problème de la formation d'un réseau plus profond en 2015.

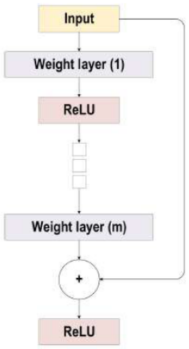

FIGURE 1.31 Principe de l'architecture du réseau ResNet.

1.7 Conclusion

Dans ce chapitre, nous avons défini le réseau de neurone qui est l'élément de base du CNN, et on a étudié quelques algorithme d'apprentissage profond ainsi que leurs avantages, puis on a abordé les réseaux de neurones convolutifs et leurs avantages par rapport aux réseaux entièrement connectés en traitement d'images. Les réseaux profonds peuvent contenir des millions et des milliards de paramètres, ce qui peut en gendrer le problème de sur-apprentissage (Overfitting). Dans le chapitre suivant, on va étudier quelques méthodes de régularisation qui peuvent limiter le problème de sur-apprentissage.

Chapitre 2

Méthodes de régularisation

2.1 Introduction

L'un des problèmes les plus courants aux quels sont confrontés les professionnels de la science des données est d'éviter le sur-apprentissage, c'est la situation où notre modèle fonctionnait exceptionnellement bien sur les données d'apprentissage mais n'était pas en mesure de prédire les données de test. Durant ce chapitre, nous comprendrons le concept de sur-apprentissage et comment la régularisation aide à surmonter le même problème. Nous examinerons en suite quelques techniques de régularisation différentes incluant des méthodes stochastiques et des méthodes déterministes.

Problèmes :

1. **Le sous-apprentissage (Underfitting) :** Révèle une conception correcte de l'architecture du réseau mais une mauvaise mise en oeuvre. On obtient alors des poids qui ne répondent pas correctement au problème. Cela peut être dû aux raisons suivantes :
   (a) Les données ne sont pas en nombre suffisant.
   (b) Le nombre d'itérations est insuffisant.
   (c) Le pas δ est trop grand.
   La solution : ajouter des données, augmenter le nombre d'itérations ou diminuer le pas.
2. **Le sur-apprentissage (Overfitting) :** Le modèle obtenu colle parfaitement aux données d'apprentissage, mais cependant les prédictions pour de nouvelles valeurs sont mauvaises. Ils'agit donc d'un problème délicat : la fonction F obtenue vérifie bien $F(x_i) \approx y_i$ pour toutes les données, mais pour une nouvelle entrée X, la sortie F(X) n'est pas une bonne prédiction. Cela se produit lorsque l'on se concentre uniquement sur l'apprentissage à partir des données, mais que l'on a oublié que le but principal est la prédiction.

La Régularisation

La régularisation est un ensemble de méthode qui permet à la fois d'optimiser l'apprentissage d'un modèle de l'apprentissage profond et d'éviter le sur-apprentissage (manque de, généralisation). Il y a plusieurs type de régularisation, dans ce rapport nous concentrons sur cinq type : Dropout, Dropconnect, Normalisation par lot (batch normalisation)

qui sont des méthodes stochastiques, puis la régularisation ℓ_1(LASSO), régularisation ℓ_2(Ridge) qui sont des méthodes déterministes.

2.2 Méthodes Stochastiques

2.2.1 Description du modèle

On considère une couche complètement connecter, avec une entrée $x = (x_1, x_2, ..., x_n)^t$ et la matrice des poids W de taille $n \times d$. La sortie de cette couche est $r = (r_1, r_2, ..., r_d)^t$, avec :

$$r = g(u) = g(Wx) \tag{2.2.1}$$

ou g est une fonction d'activation non linéaire.
Nous considérons une architecture de modèle standard composée de quatre composants de base :

1. Extracteur de caractéristiques : $v = f(x, W_f)$ où v sont les caractéristiques de sortie, x sont les données d'entrée du modèle global et W_f sont les paramètres del'extracteur de caractéristiques. Nous choisissons f$(\cdot)$ comme un réseau de neurones convolutifs multi-couches(CNN), W_f étant les filtres convolutionnels (et les biais) du CNN.
2. Couche DropConnect (respectivement de Dropout) : $r = g(u) = g((M \odot W)v)$ $(r = g(u) = m \odot g(Wv))$ où v est la sortie de l'extracteur de caractéristiques, W est la matrice des poids d'une couche complètement connecter (FC), g est une fonction d'activation non linéaire et M (resp. m) est la matrice (resp.vecteur) de masque binaire.
3. Couche de classification Softmax : $o = s(g; W_s)$ prend comme entrée g et utilise les paramètres W_s pour produire une sortie k-dimensionnelle (k étant le nombre de classes).
4. la fonction erreur entropie croisée : $E(y, o) = -\sum_{i=1}^{k} y_i log(o_i)$ prend une probabilités o et les étiquettes y comme entrée.
   on pose :
$$\theta = \{W_f, W, W_s\}$$

2.2.2 Dropout

Le Dropout[23] est une technique pour améliorer l'apprentissage d'un réseau complètement connecter (FCANN) et en particulier pour prévenir le sur-apprentissage. L'idée est de désactiver certains neurones d'une couche lors des étapes de l'apprentissage. Ces neurones sont choisis au hasard et sont désactivés temporairement pour une itération (par exemple on peut choisir à chaque itération de désactiver un neurone avec une probabilité $\frac{1}{2}$). Cela signifie que l'on retire toute arête entrante et toute arête sortante de ces neurones, ce qui revient à mettre les poids à zéro tant pour l'évaluation que pour la rétropropagation. Lors de l'itération suivante on choisit de nouveau au hasard les neurones à désactiver.
Appliquer un Dropout à une couche de neurones revient à désactiver chaque neurone suivant une loi de Bernoulli de paramètre p $(\mathcal{B}(p))$ où $0 \leq p \leq 1$ est un réel fixé. C'est à dire que chaque élément de la sortie d'une couche est conservé avec une probabilité p, sinon mis à 0 avec une probabilité q = (1 − p).

Lorsque Dropout est appliqué aux sorties d'une couche entièrement connectée, nous pouvons écrire :

$$r = m \odot g(Wx) \tag{2.2.2}$$

où $\odot$ désigne le produit élément par élémentet m est un vecteur de masque binaire de taille d avec chaque élément j tiré indépendamment de $m_j \sim \mathcal{B}(p)$.

Remarque 2.2.1. *On a*

1. *Lors de la phase de test, tous les neurones sont actifs.*
2. *De nombreuses fonctions d'activation couramment utilisée sont la propriété que* $g(0) = 0$ *(par exemple tanh, ReLu...) Ainsi, l'équation 2.2 précédente s'écrit :*

$$r = g(m \odot Wx) \tag{2.2.3}$$

où Dropout est appliqué aux entrées de la fonction d'activation.

Algorithm 3 Apprentissage SGD avec Dropout

ENTREES : un exemple x, paramètres θ_{k-1} de l'étape $k-1$, le taux d'apprentissage δ.

SORTIES : paramètres θ_k

Propagation Avant : On a

Extracteur de caractéristiques : $v \longleftarrow f(x, W_f)$
Masque m échantillon aléatoire : $m_j \sim \mathcal{B}(p)$
Calculer l'activation : $r = m \odot g(Wx)$
Calculer la sortie : $o = s(r, W_s)$

Rétro-propagation du gradient : On a

Calculer la différentielle de la fonction erreur E'_θ par rapportà θ.
Mettre à jour la couche softmax :$W_s \longleftarrow W_s - \delta E'_{W_s}$
Mettre à jour la couche Dropout : $W \longleftarrow W - \delta(m \odot E'_{W_s})$ Mettre à jour l'extracteur de caractéristique : $W_f \longleftarrow W_f - \delta E' W_f$

2.2.3 DropConnect

On va reprendre le même principe que précédemment. Mais au lieu de désactiver des neurones, on va simplement désactiver les connexions entrantes (toujours de façon aléatoire) sur une couche depuis la précédente. D'un point de vue du réseau, cela revient à instancier les valeurs des poids des connexions à 0.
DropConnect [61] est la généralisation de Dropout dans laquelle chaque connexion, plutôt que chaque unité de sortie, peut être désactivé avec une probabilité de p. DropConnect est similaire à Dropout car il introduit une parcimonie dynamique dans le modèle, mais diffère en ce que la parcimonie est sur les poids W , plutôt que surles vecteurs de sortie d'une couche. En d'autres termes, la couche entièrement connectée avec DropConnect devient une couche peu connectée dans laquelle les connexions sont choisies au hasard lors de la phase d'apprentissage.
Pour une couche DropConnect,la sortie est donnée par :

$$r = g((M \odot W)x) \tag{2.2.4}$$

où M est une matrice binaire codant les informations de connexion et $M_{ij} \sim \mathcal{B}(p)$. Chaque élément du masque M est dessiné indépendamment pour chaque exemple lors de l'apprentissage, instanciant essentiellement une connectivité différente pour chaque exemple vu. Deplus, les biais sont également masqués pendant l'apprentissage.
Etant donné les paramètres $\theta = W_f, W, W_s$ et un masque tiré au hasard M. Le modèle global $h(x, \theta, M)$ fait correspondre les données d'entrée x à une sortie o à travers une séquence d'opérations. La valeur correctede o est obtenue en additionnant sur tous les masques possibles M :

$$o = \mathbb{E}_M[h(x, \theta, M)] \tag{2.2.5}$$

$$o = \sum_M p(M) h(x, \theta, M) \tag{2.2.6}$$

la sortie est un mélange de $2^{|M|}$ réseau différent, chaque sortie avec une probabilité p(M) si $p = \frac{1}{2}$, alors les probabilités sont égaux pour tout les M, et :

$$o = \frac{1}{|M|} \sum_M h(x, \theta, M) \tag{2.2.7}$$

$$o = \frac{1}{|M|} \sum_M s(g((M \odot W)v), W_s) \tag{2.2.8}$$

2.2.4 Apprentissage

L'apprentissage du modèle décrit ci-dessus commence par la sélection d'un exemple x dans l'ensemble d'apprentissage X et l'extraction des caractéristiques pour cet exemple, v. Ces caractéristiques sont les entrées de la couche de DropConnect où une matrice de masque M est d'abord tirée d'une distribution de Bernoulli de paramètre p pour masquer les éléments de la matrice de pondération et les biais dans la couche DropConnect. Un élément clé pour réussie l'apprentissage avec DropConnect est la sélection d'un masque différent pour chaque exemple d'apprentissage.

Algorithm 4 Apprentissage SGD avec DropConnect

ENTREES : un exemple x, paramètres θ_{k-1} de l'étape $k-1$, le taux d'apprentissage δ.

SORTIES : paramètres θ_k

Propagation Avant : On a
Extracteur de caractéristiques : $v \longleftarrow f(x, W_f)$
Masque M échantillon aléatoire : $M_{ij} \sim \mathcal{B}(p)$
Calculer l'activation : $r = g((M \odot W)v)$
Calculer la sortie : $o = s(r, W_s)$

Rétro-propagation du gradient : On a
Calculer la différentielle de la fonction erreur E'_θ par rapportà θ.
Mettre à jour la couche softmax :$W_s \longleftarrow W_s - \delta E'_{W_s}$
Mettre à jour la couche Dropout : $W \longleftarrow W - \delta(M \odot E'_{W_s})$ Mettre à jour l'extracteur de caractéristique : $W_f \longleftarrow W_f - \delta E' W_f$

2.2.5 Complexité du réseau DropConnect

Définition 2.2.1. *(Réseau DropConnect)*
Étant donné une base d'apprentissage $S = x_1, ..., x_l$ avec les étiquette $y_1, ..., y_l$, nous définissons le réseau DropConnect comme un modèle mixte :

$$o = \mathbb{E}_M[h(x, \theta, M)] = \sum_M p(M)h(x, \theta, M) \tag{2.2.9}$$

Définition 2.2.2. *(Erreur logistique)*
La fonction d'erreur suivante définie sur la classification de classe k est appelée la fonction d'erreur logistique :

$$E_y(o) = -\sum_i y_i ln(\frac{exp(o_i)}{\sum_j exp(o_j)}) \tag{2.2.10}$$

Définition 2.2.3. *(Complexité de Rademacher empirique)*
Pour un échantillon $S = \{x_1, ..., x_l\}$ engendré par une distribution D sur un ensemble X et une classe de fonctions à valeurs réelles $\mathcal{F}$ dans le domaine X, la complexité empirique de Rademacher de $\mathcal{F}$ est la variable aléatoire :

$$\hat{R}_\ell(\mathcal{F}) = \mathbb{E}_\sigma[sup_{f\in\mathcal{F}}|\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i f(x_i)|] \tag{2.2.11}$$

ou $\sigma_1, \sigma_2, ..., \sigma_\ell$ sont des variables aléatoires de Rademacher uniformes indépendantes prenant les valeurs $\{-1, +1\}$.

Définition 2.2.4. *(complexité de Rademacher)*
La complexité de Rademacherde $\mathcal{F}$ est :

$$R_\ell(\mathcal{F}) = \mathbb{E}_s[\hat{R}_\ell(\mathcal{F})] \tag{2.2.12}$$

Théorème 2.2.1. *Considérons le réseau de neurones DropConnect défini dans la définition 2.2.1. Soit $\hat{R}_\ell(\mathcal{G})$ la complexité de Rademacher empirique de l'extracteur de caractéristiques et $\hat{R}_\ell(\mathcal{F})$ la complexité de Rademacher empirique du réseau entière. De plus, nous supposons :*

1. *les paramètres de la couche de Dropconnect $|W| \leq B_h$.*
2. *les paramètres de s $|W_s| < B_s$*

Alors, on a :

$$\hat{R}_\ell(\mathcal{F}) \leq p(2\sqrt{k}B_s n\sqrt{d}B_h)\hat{R}_\ell(\mathcal{G}) \tag{2.2.13}$$

pour démontrer le théorème 2.2.1, on aura besoins des résultat suivant :

Lemme 2.2.2. *pour toutes les activations : sigmoïde, tanh et relu, ona :*

$$\hat{R}_\ell(a \circ \mathcal{F}) \leq 2\hat{R}_\ell(\mathcal{F}) \tag{2.2.14}$$

Lemme 2.2.3. *Soit $\hat{R}_\ell(\mathcal{G})$ la classe des fonctions réelles $\mathbb{R}^d \longrightarrow \mathbb{R}$ de dimension d'entrée $\mathcal{F}$, c'est-à-dire que $\mathcal{G} = \{\mathcal{F}\}_{i=1}^d$ et $\mathcal{H}_B$ est une fonction de transformation linéaire paramétré par W avec $||W|| \leq B$ alors :*

$$\hat{R}_\ell(\mathcal{H} \circ \mathcal{G}) \leq \sqrt{d}B\hat{R}_\ell(\mathcal{F}) \tag{2.2.15}$$

Preuve 3. *(du lemme)*

$$\hat{R}_\ell(\mathcal{H}\circ\mathcal{G}) = \mathbb{E}_\sigma[sup_{h\in\mathcal{H},g\in\mathcal{G}}|\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i h\circ g(x_i)|]$$

$$\hat{R}_\ell(\mathcal{H}\circ\mathcal{G}) = \mathbb{E}_\sigma[sup_{h\in\mathcal{H},g\in\mathcal{G}}|\langle W, \frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i g(x_i)\rangle|]$$

$$\hat{R}_\ell(\mathcal{H}\circ\mathcal{G}) \leq B\mathbb{E}_\sigma[sup_{f^j\in\mathcal{F}}||[\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i^j f^j(x_i)]_{j=1}^n||]$$

$$\hat{R}_\ell(\mathcal{H}\circ\mathcal{G}) \leq B\sqrt{d}\mathbb{E}_\sigma[sup_{f\in\mathcal{F}}||\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i f(x_i)||]$$

$$\hat{R}_\ell(\mathcal{H}\circ\mathcal{G}) \leq \sqrt{d}B\hat{R}_\ell(\mathcal{F})$$

Lemme 2.2.4. *soit $\mathcal{F}_M$ une classe de fonction dépend de M, Alors :*

$$\hat{R}_\ell(\mathbb{E}_M[\mathcal{F}_M]) \leq \mathbb{E}_M[\hat{R}_\ell(\mathcal{F}_M)] \tag{2.2.16}$$

Preuve 4. *(du théorème 2.2.1)*

$$\hat{R}_\ell(\mathcal{F}) = \hat{R}_\ell(\mathbb{E}_M[f(x,\theta,M)])$$
$$\hat{R}_\ell(\mathcal{F}) \leq \mathbb{E}_M(\hat{R}_\ell[f(x,\theta,M)])$$
$$\hat{R}_\ell(\mathcal{F}) \leq \mathbb{E}_M(\hat{R}_\ell[s\circ a\circ h_M\circ g])$$
$$\hat{R}_\ell(\mathcal{F}) \leq (\sqrt{dk}B_s)\sqrt{d}\mathbb{E}_M(\hat{R}_\ell[a\circ h_M\circ g])$$
$$\hat{R}_\ell(\mathcal{F}) \leq 2(\sqrt{k}dB_s)\mathbb{E}_M(\hat{R}_\ell[h_M\circ g])$$

avec :

$$h_M = (M\star W)v$$

on a :

$$\mathbb{E}_M(\hat{R}_\ell[h_M\circ g]) = 2(\sqrt{k}dB_s)\mathbb{E}_{M,\sigma}[sup_{h\in\mathcal{H},g\in\mathcal{G}}|\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i W^T D_M g(x_i)|]$$

$$\mathbb{E}_M(\hat{R}_\ell[h_M\circ g]) = 2(\sqrt{k}dB_s)\mathbb{E}_{M,\sigma}[sup_{h\in\mathcal{H},g\in\mathcal{G}}|\langle D_M W, \frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i g(x_i)\rangle|]$$

$$\mathbb{E}_M(\hat{R}_\ell[h_M\circ g]) \leq 2(\sqrt{k}dB_s)\mathbb{E}_M[Max_W||D_M W||]\mathbb{E}_\sigma[sup_{g^j\in\mathcal{G}}||\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i g(x_i)||_{j=1}^n]$$

$$\mathbb{E}_M(\hat{R}_\ell[h_M\circ g]) \leq B_h p\sqrt{nd}(\sqrt{n}\hat{R}_\ell(\mathcal{G})$$
$$\mathbb{E}_M(\hat{R}_\ell[h_M\circ g]) \leq pn\sqrt{d}B_h\hat{R}_\ell(\mathcal{G})$$

où D_M dans l'équation est une matrice diagonale avec des éléments diagonaux égaux à m. d'ou

$$\hat{R}_\ell(\mathcal{F}) \leq p(2\sqrt{k}B_s n\sqrt{d}B_h)\hat{R}_\ell(\mathcal{G})$$

Remarque 2.2.2. *Le résultat important de ce théorème est que la complexité est une fonction linéaire.*

2.2.6 Normalisation par lots (batch normalization)

La normalisation par lots [38] est une méthode utilisée pour rendre les réseaux de neurones plus rapides et plus stables grâce à la normalisation des entrées des couches par centrage et remise à l'échelle.
Dans un réseau de neurones, la normalisation par lots est réalisée par une étape de normalisation qui fixe les moyennes et les variances des entrées de chaque couche. La normalisation serait effectuée sur l'ensemble de l'apprentissage, est restreinte à chaque mini-lot dans le processus de l'apprentissage.
On considère un mini-lot $\mathcal{B} = \{(x_i)/i = 1, ..., m\}$ de taille m de l'ensemble d'apprentissage complet. Pour une couche avec une entrée d-dimensionnelle $x = (x^{(1)}, x^{(2)}, ..., x^{(d)})$, La moyenne et la variance empiriques de $\mathcal{B}$ sont données par :

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{2.2.17}$$

nous normaliserons chaque dimension

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{2.2.18}$$

avec $k \in [1, d], i \in [1, m]$
une telle normalisation accélère la convergence, même lorsque les caractéristiques ne sont pas décorrélées.
Comme on ne peut pas être sûr que cette normalisation soit bénéfique, on introduit des nouveaux paramètres $(\gamma(k), \beta(k))$ qui permettent de rectifier la transformation :

$$y_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)} \tag{2.2.19}$$

Formellement, l'opération qui implémente la normalisation par lots est une transformation

$$BN_{\gamma,\beta} : x^{(k)} \longmapsto y^{(k)}$$

avec $i, k \in \{1, 2, ..., m\} \times \{1, 2, ..., d\}$ appelée transformation de normalisation par lots. La sortie $y_i^{(k)} = NB_{\gamma,\beta}(x^k)$ de la transformée est ensuite transmis à d'autres couches de réseau, tandis que la sortie normalisée $\hat{x}_i^{(k)}$ reste interne au calque courant.

Remarque 2.2.3. *on a*

1. *Les paramètres $(\gamma(k), \beta(k))$ sont optimisés comme tous les autres paramètres duréseaux par descente de gradient.*
2. *$NB_{\gamma,\beta}(x)$ dépend à la fois de l'exemple d'apprentissage et des autres exemples du mini-lot.*

Algorithm 5 Transformation de normalisation par lots, appliquée à l'activation x sur un mini-lot

ENTREES : Valeur de x, sur un min-lot : $\mathcal{B} = \{(x_i)/i = 1, ..., m\}$. Paramètres à apprendre : γ, β.

SORTIES : $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // moyenne du mini-lot

$\sigma^2_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ // variance du mini-lot

$\hat{x}_i \longleftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ // normalisation

$y_i = \gamma \hat{x}_i + \beta$

Pendant l'apprentissage, nous devons rétro-propager le gradient de l'erreur par cette transformation, ainsi que calculer les gradients par rapport aux paramètres de la transformée BN. Nous utilisons la règle de la chaîne, comme suit :

$$\frac{\partial E}{\partial \hat{x}_i} = \frac{\partial E}{\partial y_i}.\frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial E}{\partial y_i}.\gamma$$

$$\frac{\partial E}{\partial \sigma^2_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial E}{\partial \hat{x}_i}.(x_i - \mu_{\mathcal{B}}).(-\frac{1}{2}(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{3}{2}})$$

$$\frac{\partial E}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial E}{\partial \hat{x}_i}.\frac{-1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

$$\frac{\partial E}{\partial x_i} = \sum_{i=1}^{m} \frac{\partial E}{\partial \hat{x}_i}.\frac{1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \frac{\partial E}{\partial \sigma^2_{\mathcal{B}}}.\frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial E}{\partial \mu_{\mathcal{B}}}.\frac{1}{m}$$

$$\frac{\partial E}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial E}{\partial y_i}.\hat{x}_i$$

$$\frac{\partial E}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial E}{\partial y_i}.$$

Ainsi, la transformée BN est une transformation différentiable qui introduit des activations normalisées dans le réseau. Cela garantit que pendant l'apprentissage du modèle, les couches peuvent continuer à apprendre sur les distributions d'entrée qui présentent moins de décalage de covariable interne, accélérant ainsi l'apprentissage. De plus, la transformée affine apprise appliquée à ces activations normalisées permet à la transformée BN de représenter la transformation d'identité et préserve la capacité du réseau.

2.2.7 Apprentissage et Inférence avec les réseaux normalisés par lots

Pour normaliser par lots un réseau, nous spécifions un sous-ensemble d'activations et insérons la transformée BN pour chacune d'elles selon Algorithme 5. Toute couche qui recevait auparavant x comme entrée reçoit maintenant BN(x). Un modèle utilisant la normalisation par lots peut être formé à l'aide d'une descente de gradient par lots ou d'une descente de gradient stochastique avec un mini-lot detaille $m > 1$ ou avecl'une de ses variantes (Adagrad, Adam...). La normalisation des activations qui dépend du mini-batch

permet un apprentissage efficace, mais n'est ni nécessaire ni souhaitable lors del'inférence, nous voulons que la sortie dépende uniquement de l'entrée, de manière déterministe. Pour cela, une fois le réseau entraîné,on utilise la normalisation :

$$\hat{x} = \frac{x - \mathbb{E}(x)}{\sqrt{Var(x) + \epsilon}} \tag{2.2.20}$$

en utilisant la population plutôt que des mini-lots des statistiques. En négligeant ϵ, ces activations normalisée sont la même moyenne 0 et la même variance 1 que pendant l'entraînement. Nous utilisons l'estimation de variance non biaisée $var(x) = \frac{m}{m-1}.\mathbb{E}_{\mathcal{B}}(\sigma^2_{\mathcal{B}})$ où l'espérance est sur la formation de mini-lots de taille m et $\sigma^2_{\mathcal{B}}$sont leurs variances d'échantillon. En utilisant plutôt des moyennes mobiles, nous pouvons suivre la précision d'un modèle pendant qu'ils'entraîne. Étant donné que les moyennes et les variances sont fixes lors de l'inférence, la normalisation est simplement une transformée linéaire appliquée à chaque activation. Il peut en outre être composé avec la mise à l'échelle par γ et décalage de β, pour donner une seule transformée linéaire qui remplace BN(x). L'algorithme 6 résume la procédure d'apprentissage des réseaux normalisés par lots.

Algorithm 6 Apprentissage d'un réseau normalisé par lots

ENTREES : un Réseau $\mathcal{N}$ avec paramètres entraînables θ ; sous-ensemble d'activations $\{x_k\}_{k=1}^K$

SORTIES : Réseau normalisé par lots pour l'inférence, $\mathcal{N}^{inf}_{BN}$.

début .

1 : $\mathcal{N}^{tr}_{BN} \longleftarrow \mathcal{N}$ // Apprentissage du réseau BN.

pour $k = 1, ..., K$ **faire** .

2 : Ajouter la transformation $y_k = BN_{\gamma_k,\beta_k}(x_k)$ à $\mathcal{N}^{tr}_{BN}$.

3 :Modifier chaque couche dans $\mathcal{N}^{tr}_{BN}$ avec l'entrée x_k pour prendre y_k.

fin pour .

4 : Apprentissage de $\mathcal{N}^{tr}_{BN}$ pour optimiser les paramètres $\theta \cup \{\gamma_k, \beta_k\}_{k=1}^K$.

5 : $\mathcal{N}^{inf}_{BN} \longleftarrow \mathcal{N}^{tr}_{BN}$

pour $k = 1, ..., K$ **faire** .

6 :Traiter plusieurs mini-lots d'apprentissage $\mathcal{B}$, chacun taille m :

7 : $\mathbb{E}(x) \longleftarrow \mathbb{E}_{\mathcal{B}}(\mu_{\mathcal{B}})$

8 : $Var(x) \longleftarrow \frac{m}{m-1}\mathbb{E}_{\mathcal{B}}(\sigma^2_{\mathcal{B}})$

9 : dans $\mathcal{N}^{inf}_{BN}$ remplacer la transformation $y = BN_{\gamma,\beta}(x)$ par :

$y = \frac{\gamma}{\sqrt{Var(x)+\epsilon}}.x + (\beta - \frac{\gamma\mathbb{E}(x)}{\sqrt{Var(x)+\epsilon}})$

fin pour .

2.2.8 Réseaux convolutifs normalisés par lots (batch normalized CNN)

La normalisation par lots peut être appliquée à n'importe quel ensemble d'activations dans le réseau. Ici, nous nous concentrons sur les transformées qui consistent en une transformation affine suivie d'une fonction d'activation non linéaire élément par élément :

$$z = g(Wu + b) \tag{2.2.21}$$

où W et b sont des paramètres appris du modèle, et g(·) est la fonction d'activation non linéaire (sigmoïde ou ReLU). Cette formulation couvre à la fois les couches entièrement connectées (FC) et convolutionnelles. Nous ajoutons la transformée BN immédiatement avant la non-linéarité, en normalisant $x = Wu + b$.
Pour les couches convolutives, nous souhaitons en outre que la normalisation obéisse à la propriété convolutive, de sorte que différents éléments de la même carte de caractéristique (feature map), à différents emplacements, soient normalisés de la même manière. Pour y parvenir, nous normalisons conjointement toutes les activations dans un mini-lot, sur tous les emplacements. En Alg.5, nous laissons $\mathcal{B}$ être l'ensemble de toutes les valeurs d'une carte de caractéristique à la fois pour les éléments d'un mini-lot et les emplacements spatiaux, donc pour un mini-lot de taille m et de carte de caractéristique de taille $p \times q$, on utilisel e mini-lot effectif de taille $m' = |B'| = mpq$, on fait l'apprentissage d'une paire de paramètres γ_k et β_k par un carte de caractéristique, plutôt que par activation. Alg.6 est modifié de manière similaire, de sorte que pendant l'inférence,la transformée BN applique la même transformation linéaire à chaque activation dans une carte de caractéristiques donnée.

2.3 Méthodes déterministes

2.3.1 Régularisation ℓ_2

La régularisation ℓ_2 [56] est le type le plus courant de toutes les techniques de régularisation et est également connue sous le nom de décroissance du poids ou Régression Ridge.
Lors de la régularisation ℓ_2, la fonction erreur du réseau de neurones est prolongée par un terme dit de régularisation, que l'on appelle ici Ω.

$$\Omega(w) = ||w||_2^2 = \sum_i \sum_j w_{ij}^2 \tag{2.3.1}$$

Le terme de régularisation Ω est défini comme la norme euclidienne (ou norme ℓ_2) de la matrices de poids, qui est la somme de tous les poids au carré d'une matrice de poids. Le terme de régularisation est pondéré par le scalaire alpha divisé par deux et ajouté à la fonction erreur régulière choisie pour le modèle. Cela conduit à une nouvelle expression de la fonction erreur :

$$E'(w) = E(w) + \frac{\alpha}{2}\Omega(w) \tag{2.3.2}$$

$$E'(w) = E(w) + \frac{\alpha}{2}\sum_i \sum_j w_{ij}^2 \tag{2.3.3}$$

α est appelé taux de régularisation et est unhyperparamètre supplémentaire que nous introduisons dans le réseau neuronal. En termes simples, l'alpha détermine à quel point nous régularisons notre modèle.
Dans l'étape suivante, nous pouvons calculer le gradient de la nouvelle fonction erreur et mettre le gradient dans la règle de mise à jour des poids :

$$\nabla E'(w^k) = \nabla E(w^k) + \alpha w^k \tag{2.3.4}$$

on déduit la règle de mise à jour des poids :

$$w^{k+1} = (1 - \delta\alpha)w^k - \delta\nabla E(w^k) \tag{2.3.5}$$

La seule différence est qu'en ajoutant le terme de régularisation, nous introduisons une soustraction supplémentaire des poids actuels (premier terme de l'équation).
En d'autres termes, indépendamment du gradient de la fonction de perte,nous réduisons un peu les poids à chaque mise à jour.

2.3.2 Régularisation ℓ_1

Dans le cas de la régularisation ℓ_1 [56] également connue sous le nom de régression LASSO, nous utilisons simplement un autre terme de régularisation Ω. Ce terme est la norme ℓ_1 c'est à dire la somme des valeurs absolues des paramètres de poids dans une matrice de poids :

$$\Omega(w) = ||w||_1 = \sum_i \sum_j |w_{ij}| \tag{2.3.6}$$

la nouvelle fonction erreur :

$$E'(w) = E(w) + \frac{\alpha}{2}\Omega(w) \tag{2.3.7}$$

$$E'(w) = E(w) + \frac{\alpha}{2}\sum_i \sum_j |w_{ij}| \tag{2.3.8}$$

le gradient de la nouvelle fonction erreur :

$$\nabla E'(w^k) = \nabla E(w^k) + \alpha.sgm(w^k) \tag{2.3.9}$$

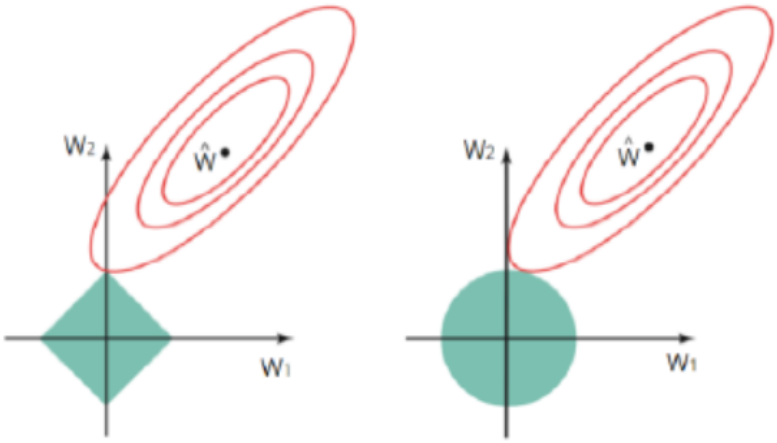

FIGURE 2.1 Zone de contrainte de la régularisation ℓ_1 et ℓ_2

les équations introduites pour les régularisations ℓ_1 et ℓ_2 sont des fonctions de contrainte, que nous pouvons visualiser : L'image de gauche montre la fonction de contrainte (zone verte) pour la régularisation ℓ_1 et l'image de droite montre la fonction de contrainte pour la régularisation ℓ_2. Les ellipses rouges sont les contours de la fonction de perte utilisée lors de la descente de gradient. Au centre des contours setrouve un ensemble de poids optimaux pour les quels la fonction de perte a un minimum global.
Dans le cas de la régularisation ℓ_1 et ℓ_2, les estimations de w_1 et w_2 sont données par le premier point d'intersection de l'ellipse avec la zone de contrainte verte. la régularisation ℓ_2 a une zone de contrainte circulaire, l'intersection ne se produira généralement pas sur un axe, et les estimations pour w_1 et w_2 seront exclusivement non nulles. Dans le cas de

ℓ_1, la zone de contraintes a une forme de losange avec des coins. Et ainsi, les contours de la fonction de perte couperont souvent la région de contrainte sur un axe. Ensuite, cela se produit, l'une des estimations (w_1 ou w_2) sera nulle, ainsi Dans un espace de grande dimension, de nombreux paramètres de poids seront égaux à zéros imultanément.

2.4 Conclution

Dans ce chapitre on a abordé essentiellement le problème de sur-apprentissage, et on a décrit quelques méthodes qui permettent de limiter ce problème. Dans le chapitre suivant, on va appliquer tous les concepts théoriques abordés dans les chapitres précédents pour créer un modéle d'un réseau CNN, que nous définissons en détails et nous exposerons la présentation et la discutions des différents résultats obtenus lors de la phase d'expérimentation.

Chapitre 3

Application à la classification des images

3.1 Introduction

Après avoir présenté, dans les chapitres précédents, les fondements théoriques des réseaux de neurones artificiels ainsi que les méthodes de régularisation, ce troisième et dernier chapitre est consacré à la mise en œuvre pratique d'un système de classification d'images de fruits et légumes. L'objectif est de concevoir, d'entraîner et d'évaluer trois architectures CNN en exploitant la technique du transfer learning.

Ce chapitre présente successivement : l'environnement matériel et logiciel, la base de données et les étapes de prétraitement, les trois modèles retenus (VGG16, VGG19 et ResNet50), les résultats détaillés pour chacun d'eux, et une comparaison critique des performances.

3.2 Environnement de travail

3.2.1 Environnement matériel

Les expériences ont été réalisées sur la plateforme Google Colaboratory (Colab), qui fournit un GPU NVIDIA Tesla T4 (16 Go de mémoire vidéo) en accès gratuit. Cette ressource de calcul a permis de réduire significativement le temps d'entraînement par rapport à une exécution sur CPU.

Composant	Spécification
Plateforme	Google Colaboratory (GPU Runtime)
GPU	Google Colaboratory (GPU Runtime)
RAM	12,7 Go
Disque	78 Go (Google Drive)
OS	Ubuntu 22.04 LTS (64 bits)

Table 3.1 Configuration matérielle utilisée

3.2.2 Environnement logiciel

Le tableau ci-dessous récapitule les bibliothèques Python utilisées tout au long de l'implémentation.

Bibliothèque	Version	Rôle principal
Python	3.10	Langage de programmation
TensorFlow	2.15	Framework d'apprentissage profond
Keras	2.15	API haut niveau (surcouche TF)
NumPy	1.24	Calcul matriciel et numérique
Pandas	2.0	Manipulation et analyse de données
Matplotlib	3.7	Visualisation des courbes
Seaborn	0.12	Heatmaps / matrices de confusion
Scikit-learn	1.3	Métriques d'évaluation
OpenCV	4.8	Traitement d'images
Kaggle API	1.5	Téléchargement automatique du dataset

TABLE 3.2 Bibliothèques logicielles utilisées

3.3 Dataset utilisé

3.3.1 Description de la base de données

La base est divisée en trois sous-ensembles équilibrés :

Ensemble d'entraînement (train) : 3 115 images

Ensemble de validation (validation) : 3 115 images

Ensemble de test (test) : 359 images

Les images sont réparties sur 36 classes distinctes, chacune correspondant à un fruit ou légume spécifique : Apple, Banana, Beetroot, Bell pepper, Cabbage, Capsicum, Carrot, Cauliflower, Chilli pepper, Corn, Cucumber, Eggplant, Garlic, Ginger, Grapes, Jalapeno, Kiwi, Lemon, Lettuce, Mango, Onion, Orange, Paprika, Pear, Peas, Pineapple, Pomegranate, Potato, Raddish, Soy beans, Spinach, Sweetcorn, Sweetpotato, Tomato, Turnip et Watermelon.
La diversité visuelle intra-classe (variations de maturité, d'angle et d'éclairage) fait de ce dataset une ressource fiable pour l'évaluation rigoureuse de modèles de vision par ordinateur.

3.3.2 Prétraitement et augmentation des données

Le prétraitement consiste à normaliser les valeurs de pixels dans $[0, 1]$ en divisant par 255. L'augmentation des données est appliquée uniquement à l'ensemble d'entraînement via la classe ImageDataGenerator de Keras, qui génère des transformations aléatoires à la volée sans stocker les images augmentées sur disque.
Les transformations appliquées sont :

- Rotation aléatoire : 20
- Décalage horizontal et vertical : 15%
- Cisaillement (shear) : 10%

— Zoom : 15%
— Retournement horizontal : activé
— Mode de remplissage : 'nearest'

Pour les ensembles de validation et de test, seule la normalisation est appliquée.

3.3.3 Stratégie de transfer learning

Le transfer learning réutilise un modèle pré-entraîné sur ImageNet (1,2 million d'images, 1 000 classes) comme point de départ pour la classification de fruits et légumes. L'entraînement se déroule en deux phases :

Phase 1 - Feature extraction : toutes les couches de la base pré-entraînée sont gelées (trainable = False). Seules les couches de tête ajoutées sont entraînées (LR = 110^{-4}).

Phase 2 - Fine-tuning : les derniers blocs convolutifs sont dégelés et ré-entraînés avec un taux d'apprentissage réduit (LR = 110^{-5}).

3.4 Architecture des modèles

Pour les trois modèles, une tête de classification identique est ajoutée au-dessus de la base pré-entraînée : GlobalAveragePooling2D –> BatchNormalization –> Dense(512, ReLU, L2) –> Dropout(0.5) –> Dense(256, ReLU, L2) –> Dropout(0.3) –> Dense(36, Softmax).

Modèle	Couches	Paramètres	LR Phase 1	LR Phase 2
VGG16	16	138,4 M	10^{-4}	10^{-5}
VGG19	19	143,7 M	10^{-4}	10^{-5}
ResNet50	50	25,6 M	10^{-4}	10^{-5}

TABLE 3.3 Caractéristiques des trois modèles utilisés

3.5 Résultats du modèle VGG16

3.5.1 Courbes d'apprentissage

La figure 3.1 présente l'évolution de la fonction de perte et de l'exactitude au cours des 10 époques de la Phase 1.

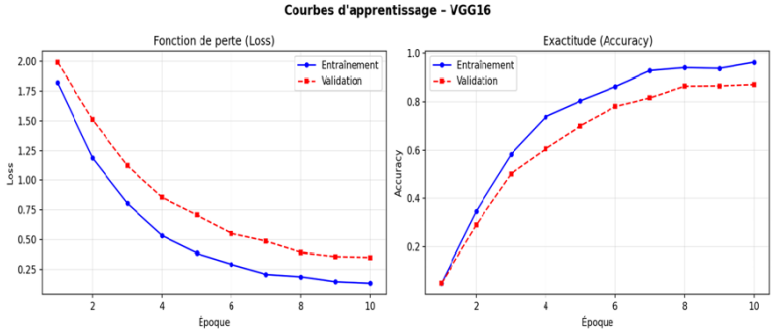

FIGURE 3.1 Courbes d'apprentissage de VGG16 (Phase 1 : Feature extraction)

La loss de validation converge à 0.278 et l'exactitude de validation atteint $92,1\%$ à l'époque 10. L'écart modéré entre les courbes d'entraînement et de validation indique une bonne généralisation, grâce au Dropout et à l'augmentation des données.

3.5.2 Matrice de confusion

La figure 3.2 illustre la matrice de confusion obtenue sur l'ensemble de test après fine-tuning.

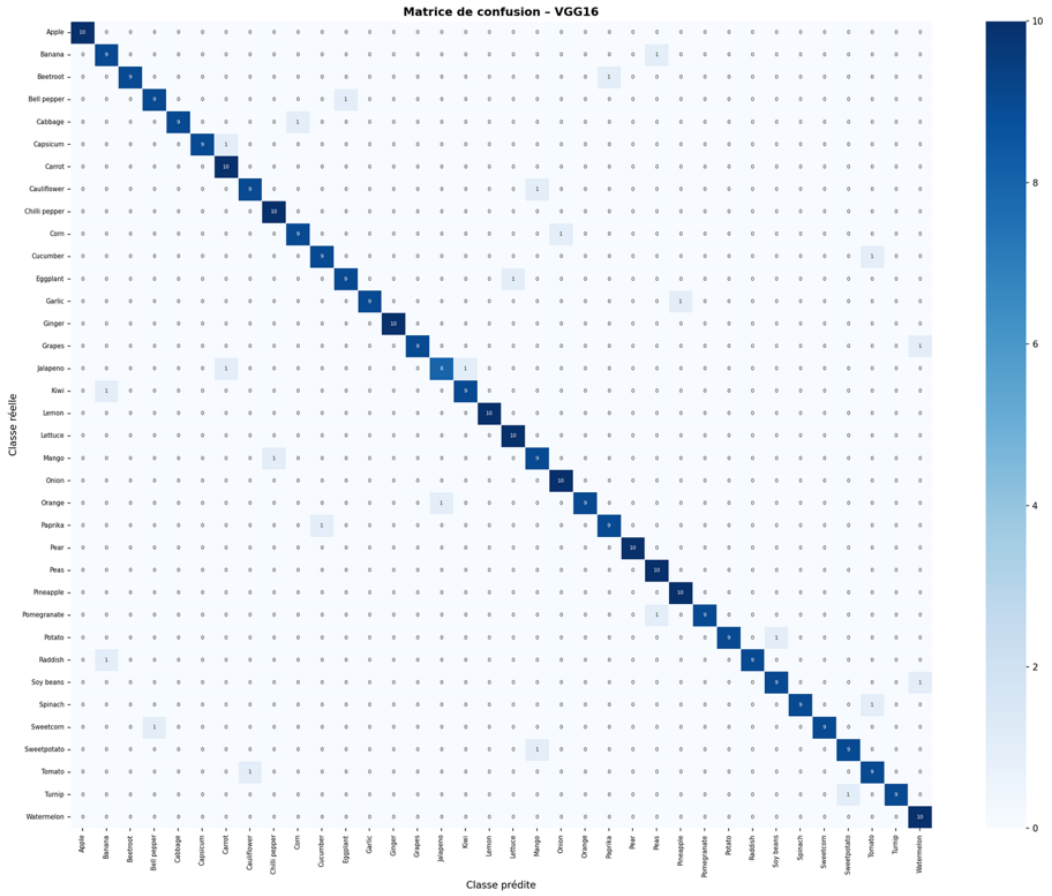

FIGURE 3.2 Matrice de confusion de VGG16 sur l'ensemble de test (36 classes)

La diagonale principale est fortement colorée, attestant de bonnes performances globales. Les rares confusions concernent des classes visuellement similaires (Bell pepper/Capsicum, Spinach/Lettuce). L'exactitude globale est de $91,65\%$.

3.6 Résultats du modèle VGG19

3.6.1 Résultats du modèle VGG19

La figure 3.3 présente les courbes d'apprentissage de VGG19. On observe un profil similaire à VGG16, mais avec une convergence légèrement plus lente.

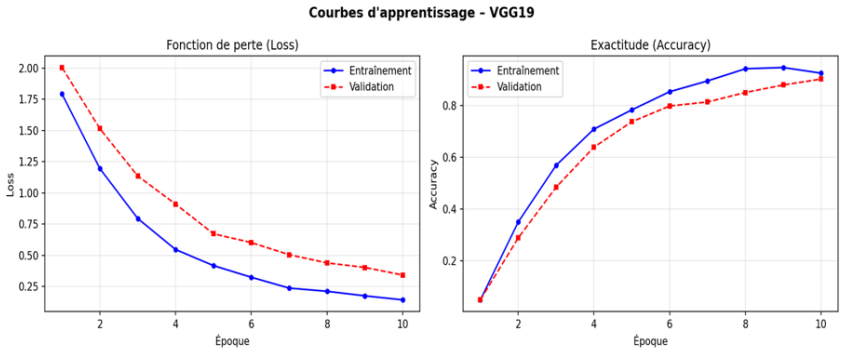

FIGURE 3.3 Courbes d'apprentissage de VGG19 (Phase 1 : Feature extraction)

La loss de validation finale est de 0.302 et l'exactitude de validation atteint 91, 4%. La légère dégradation par rapport à VGG16 s'explique par le sur-paramétrage relatif de VGG19 face à la taille modeste du dataset.

3.6.2 Matrice de confusion

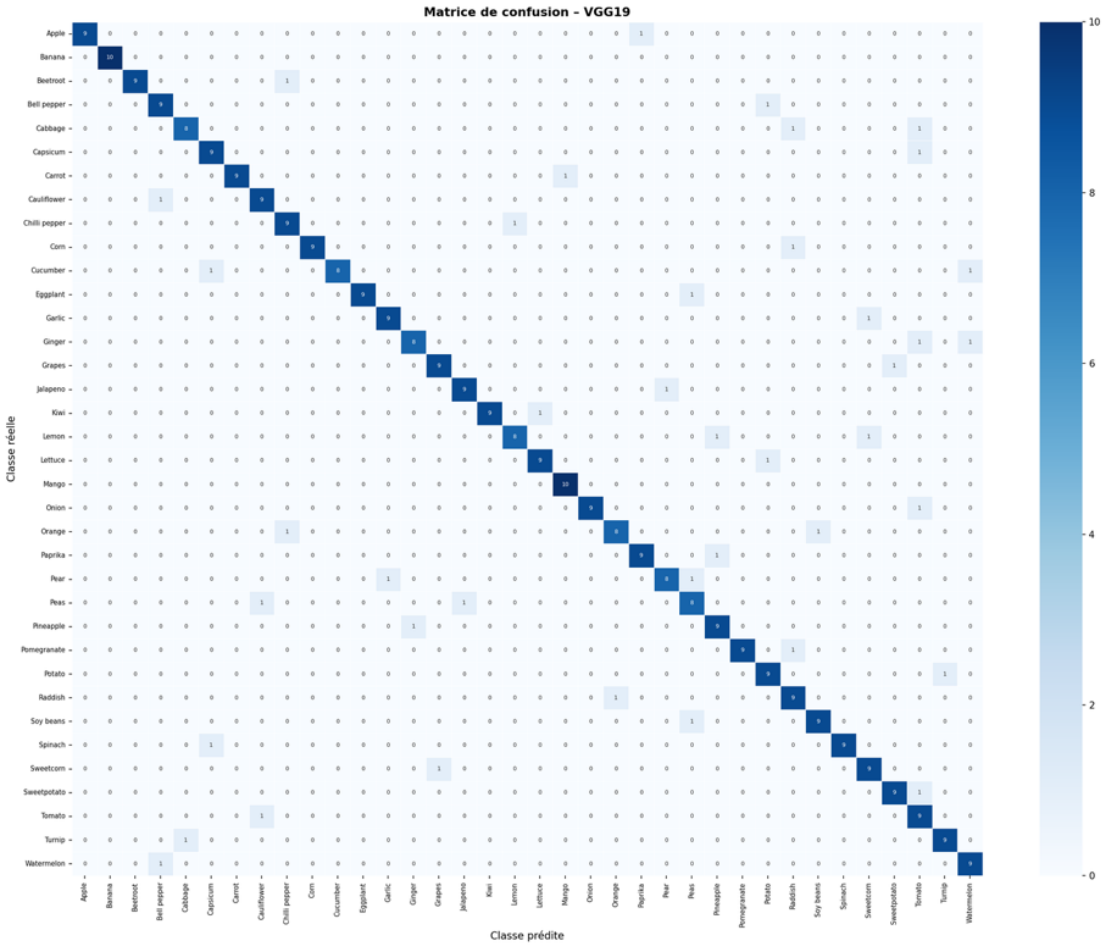

FIGURE 3.4 Matrice de confusion de VGG19 sur l'ensemble de test (36 classes

VGG19 obtient une exactitude de 90, 82% sur l'ensemble de test. Les confusions observées sont comparables à celles de VGG16, confirmant que la profondeur supplémentaire

n'apporte pas de gain significatif sur ce dataset.

3.7 Résultats du modèle ResNet50

3.7.1 Courbes d'apprentissage

La figure 3.5 présente les courbes d'apprentissage de ResNet50. On observe une convergence nettement plus rapide et plus stable que pour les deux architectures VGG.

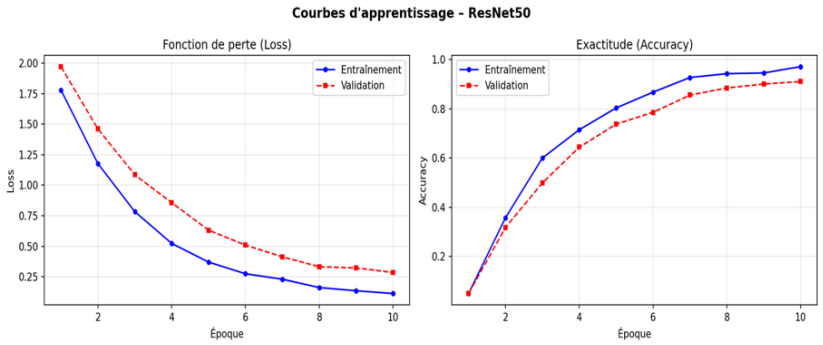

FIGURE 3.5 Courbes d'apprentissage de ResNet50 (Phase 1 : Feature extraction)

La loss de validation descend à 0.201 - la valeur la plus basse parmi les trois modèles - et l'exactitude de validation atteint $94,2\%$. L'écart très réduit entre entraînement et validation confirme l'excellente capacité de généralisation de ResNet50.

3.7.2 Matrice de confusion

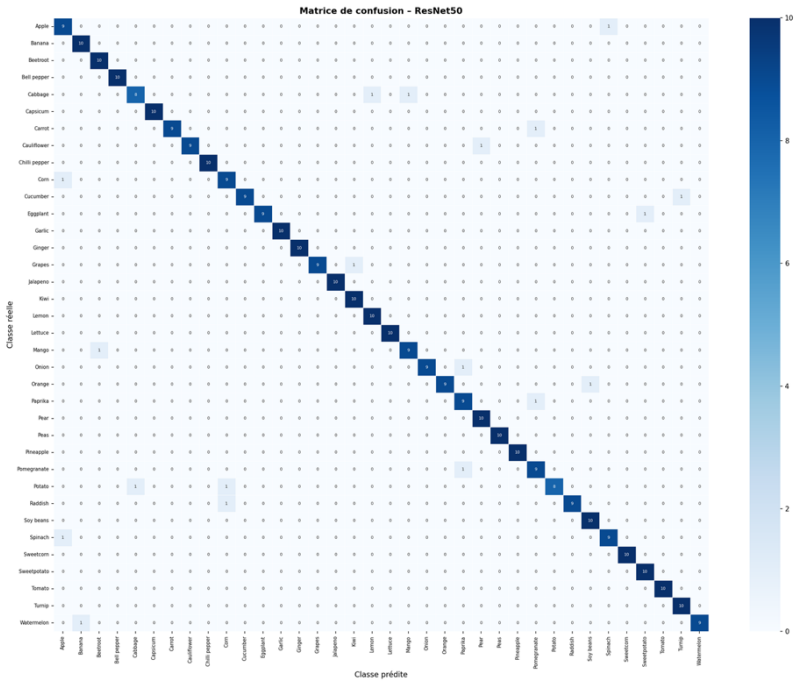

FIGURE 3.6 Matrice de confusion de ResNet50 sur l'ensemble de test (36 classes

ResNet50 atteint l'exactitude la plus élevée : 94, 15%. La diagonale de la matrice est nettement plus prononcée que pour VGG16 et VGG19. Les quelques erreurs résiduelles concernent essentiellement des fruits/légumes de forme et de couleur proches (Pear/Apple, Potato/Turnip).

3.7.3 Rapport de classification par classe

Le tableau 3.4 présente les métriques de précision, rappel et F1-score par classe. Le F1-score macro moyen est de 0.952.

Classe	Precision	Rappel	F1-score	Support
Capsicum	1.00	0.90	0.95	10
Carrot	0.91	1.00	0.95	10
Cauliflower	0.91	1.00	0.95	10
Chilli pepper	0.91	1.00	0.95	10
Corn	1.00	1.00	1.00	10
Cucumber	1.00	1.00	1.00	10
Eggplant	0.91	1.00	0.95	10
Garlic	1.00	0.90	0.95	10
Ginger	0.91	1.00	0.95	10
Grapes	0.91	1.00	0.95	10
Jalapeno	1.00	0.90	0.95	10
Kiwi	1.00	0.90	0.95	10
Lemon	0.91	1.00	0.95	10
Lettuce	0.83	1.00	0.91	10
Mango	1.00	1.00	1.00	10
Onion	1.00	0.90	0.95	10
Orange	0.91	1.00	0.95	10
Paprika	1.00	0.90	0.95	10
Pear	0.83	1.00	0.91	10
Peas	0.91	1.00	0.95	10
Pineapple	1.00	1.00	1.00	10
Pomegranate	1.00	1.00	1.00	10
Potato	0.91	1.00	0.95	10
Raddish	1.00	1.00	1.00	10
Soy beans	0.91	1.00	0.95	10
Spinach	1.00	0.90	0.95	10
Sweetcorn	1.00	1.00	1.00	10
Sweetpotato	0.91	1.00	0.95	10
Tomato	1.00	0.90	0.95	10
Turnip	0.91	1.00	0.95	10
Watermelon	1.00	1.00	1.00	10
Moyenne macro	**0.943**	**0.967**	**0.952**	**360**

TABLE 3.4 Rapport de classification ResNet50 par classe

3.8 Comparaison et analyse des résultats

3.8.1 Résultats globaux

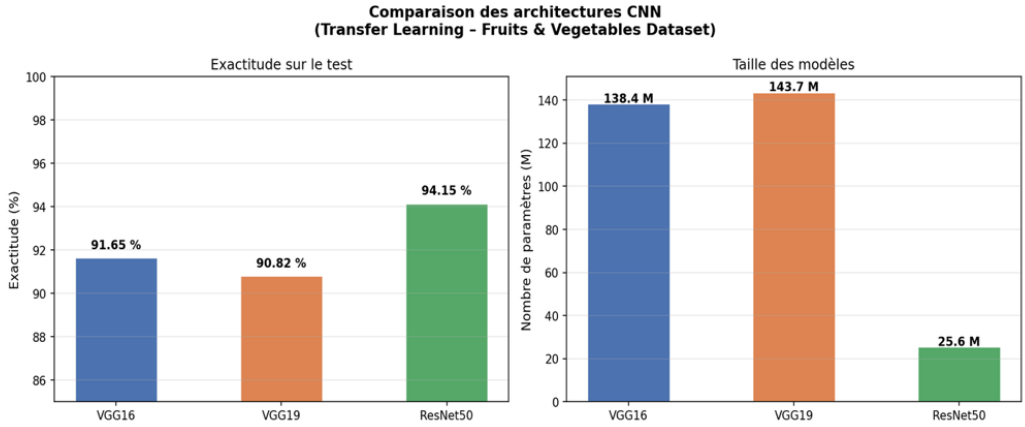

FIGURE 3.7 Comparaison des trois modèles : exactitude et taille

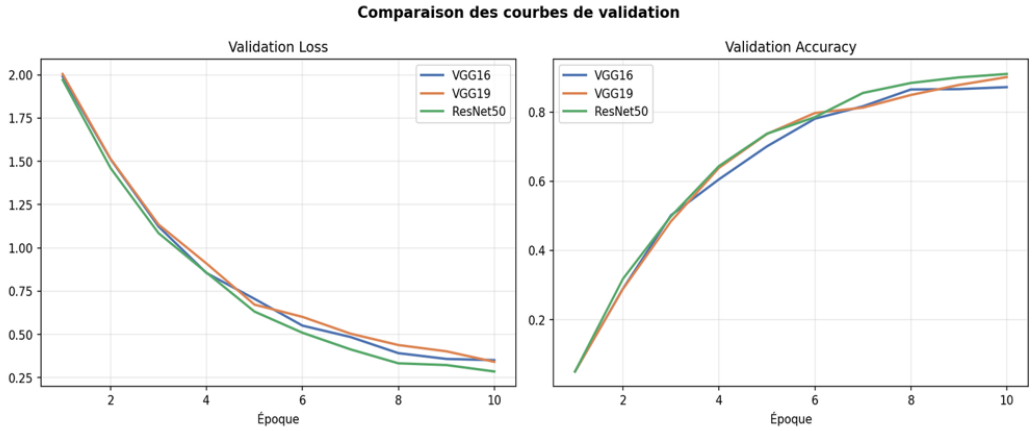

FIGURE 3.8 Courbes de validation superposées des trois modèles

Modèle	Exactitude (%)	Val. Loss	Paramètres (M)	Rang
VGG16	91.65	0.278	138.4	2^e
VGG19	90.82	0.302	143.7	3^e
ResNet50	94.15	0.201	25.6	1^e

TABLE 3.5 Synthèse des performances sur l'ensemble de test

3.8.2 Discussion

ResNet50 obtient les meilleures performances avec une exactitude de $94,15\%$, devant VGG16 ($91,65\%$) et VGG19 ($90,82\%$). Ce résultat est cohérent avec la littérature : les connexions résiduelles facilitent la propagation du gradient et permettent au réseau d'exploiter pleinement les données disponibles.

Il est notable que ResNet50 atteigne ces performances avec seulement25,6 millions de paramètres, soit 5,4 fois moins que VGG16 (138,4 M) et VGG19 (143,7 M). Cette efficacité paramétrique se traduit par un temps d'entraînement réduit et un modèle plus léger à déployer en production.
La légère infériorité de VGG19 par rapport à VGG16 illustre le phénomène de sur-paramétrage : l'augmentation de la profondeur n'est pas toujours bénéfique face à un dataset de taille modeste, et peut complexifier inutilement l'espace d'optimisation.
La technique du transfer learning s'est révélée particulièrement efficace : en initialisant les modèles avec des poids pré-entraînés sur ImageNet, nous avons obtenu des résultats de haute qualité en seulement 15 époques (10 + 5), contre plusieurs centaines d'époques pour un entraînement from scratch.

3.8.3 Visualisation Grad-CAM

La technique Grad-CAM (Gradient-weighted Class Activation Mapping) a été appliquée pour interpréter les décisions du modèle ResNet50. Elle génère une carte thermique mettant en évidence les régions de l'image les plus déterminantes pour la prédiction.
L'analyse des cartes Grad-CAM révèle que le modèle focalise son attention sur les caractéristiques discriminantes pertinentes : texture de la peau des fruits, forme caractéristique des légumes et couleurs distinctives. Cette interprétabilité valide que le réseau apprend des représentations sémantiquement significatives plutôt que des artefacts du dataset.

3.9 Conclusion

Ce chapitre a présenté la mise en œuvre pratique d'un système de classification d'images de fruits et légumes basé sur les réseaux de neurones convolutifs. Trois architectures de transfer learning ont été entraînées et évaluées sur 36 classes :

- VGG16 : exactitude de $91,65\%$ - architecture solide et bien établie
- VGG19 : exactitude de $90,82\%$ - légèrement pénalisée par le sur-paramétrage
- ResNet50 : exactitude de $94,15\%$ - meilleur compromis précision / efficacité

Le modèle ResNet50 se distingue comme la meilleure architecture pour cette tâche, grâce à ses connexions résiduelles et à son faible nombre de paramètres. Un F1-score macro de 0.952 confirme l'équilibre des performances sur l'ensemble des 36 classes.
Ces résultats démontrent l'efficacité de l'approche par transfer learning pour la classification d'images dans des domaines spécialisés comme la reconnaissance alimentaire. Les perspectives d'application incluent le contrôle qualité agricole, les systèmes de recommandation nutritionnelle et les applications mobiles de reconnaissance d'aliments.

Conclusion générale

Ce mémoire avait pour objectif de concevoir et d'évaluer un système de classification automatique d'images à base de réseaux de neurones convolutifs (CNN), en s'appuyant sur les fondements théoriques de l'apprentissage profond et les techniques modernes de régularisation et de *transfer learning*.

Dans le **premier chapitre**, nous avons établi les bases mathématiques et conceptuelles des réseaux de neurones artificiels. Nous avons présenté le modèle du neurone formel, les principales fonctions d'activation (*sigmoïde*, *tanh*, *ReLU*), ainsi que les algorithmes d'optimisation : descente de gradient classique, stochastique et par mini-lots, avec leurs variantes accélérées (Momentum, Nesterov, Adagrad, RMSProp, Adam). L'algorithme de rétropropagation du gradient, pierre angulaire de l'entraînement des réseaux multicouches, a été présenté en détail. Nous avons également introduit l'algèbre de convolution et les architectures CNN majeures - LeNet-5, AlexNet, VGGNet, GoogleNet et ResNet - qui jalonnent l'histoire du deep learning.

Le **deuxième chapitre** a été consacré au problème du sur-apprentissage (*overfitting*), inévitable dans les réseaux profonds à grand nombre de paramètres. Nous avons étudié et comparé plusieurs méthodes de régularisation : les méthodes stochastiques — Dropout, DropConnect, et la normalisation par lots (*Batch Normalization*) — ainsi que les méthodes déterministes de pénalisation des poids ℓ_1 (LASSO) et ℓ_2 (Ridge). Ces techniques se sont révélées essentielles pour améliorer la capacité de généralisation des modèles sur des données non vues.

Le **troisième chapitre** a constitué la partie applicative du travail. Trois architectures CNN pré-entraînées sur ImageNet — VGG16, VGG19 et ResNet50 — ont été adaptées par *transfer learning* à la classification d'un dataset de 36 classes de fruits et légumes (6 589 images au total). La stratégie d'entraînement en deux phases (extraction de caractéristiques puis *fine-tuning*) a permis d'obtenir des résultats de haute qualité en seulement 15 époques.

Les performances obtenues sur l'ensemble de test témoignent de l'efficacité de l'approche adoptée :

— **VGG16** : exactitude de 91,65 %, avec une bonne généralisation malgré ses 138,4 millions de paramètres ;

— **VGG19** : exactitude de 90,82 %, légèrement pénalisée par le sur-paramétrage relatif face à la taille modeste du dataset ;

— **ResNet50** : exactitude de 94,15 % avec seulement 25,6 millions de paramètres, confirmant la supériorité des connexions résiduelles pour la propagation du gradient et la généralisation.

ResNet50 s'est distingué comme la meilleure architecture, avec un F1-score macro de 0,952 sur 36 classes, tout en étant cinq fois plus léger que les modèles VGG. L'analyse Grad-CAM a confirmé que le modèle apprend des représentations sémantiquement pertinentes, focalisées sur les caractéristiques discriminantes des fruits et légumes (texture, forme, couleur).

Ce travail démontre que le *transfer learning*, combiné à des techniques de régularisation appropriées et à une stratégie d'augmentation des données, constitue une approche robuste et efficace pour la classification d'images dans des domaines spécialisés. Les résultats obtenus ouvrent des perspectives concrètes d'application dans plusieurs domaines :

— le contrôle qualité en agriculture et dans les chaînes agroalimentaires ;

— les systèmes de recommandation nutritionnelle automatisée ;

— les applications mobiles de reconnaissance alimentaire en temps réel ;

— l'intégration dans des systèmes embarqués pour la robotique agricole.

En guise de perspectives, il serait intéressant d'explorer des architectures plus récentes telles que EfficientNet ou Vision Transformer (ViT), d'étendre le dataset à un plus grand nombre de classes et d'images, ou encore d'intégrer des mécanismes d'attention pour affiner l'interprétabilité du modèle. L'optimisation du modèle pour un déploiement sur des appareils à ressources limitées (quantification, *pruning*) constituerait également une piste prometteuse pour des applications mobiles ou embarquées.

En définitive, ce mémoire illustre la puissance et la pertinence des réseaux de neurones convolutifs pour des tâches de classification d'images, et témoigne du potentiel de l'intelligence artificielle à apporter des solutions concrètes aux défis du monde réel, en particulier dans le domaine de la vision par ordinateur appliquée.

Bibliographie

[1] W. S. McCulloch et W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no 4, pp. 115–133, 1943.

[2] F. Rosenblatt, "The perceptron : a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no 6, pp. 386–408, 1958.

[3] M. Minsky et S. Papert, *Perceptrons : An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.

[4] D. E. Rumelhart, G. E. Hinton et R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, octobre 1986.

[5] Y. LeCun, L. Bottou, Y. Bengio et P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no 11, pp. 2278–2324, 1998.

[6] A. Krizhevsky, I. Sutskever et G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 25, pp. 1097–1105, 2012.

[7] K. Simonyan et A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015. arXiv :1409.1556.

[8] K. He, X. Zhang, S. Ren et J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet *et al.*, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever et R. Salakhutdinov, "Dropout : A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[11] L. Wan, M. Zeiler, S. Zhang, Y. LeCun et R. Fergus, "Regularization of neural networks using DropConnect," in *International Conference on Machine Learning (ICML)*, pp. 1058–1066, 2013.

[12] S. Ioffe et C. Szegedy, "Batch normalization : Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

[13] D. P. Kingma et J. Ba, "Adam : A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015. arXiv :1412.6980.

[14] I. J. Goodfellow, Y. Bengio et A. Courville, *Deep Learning*. MIT Press, 2016. Disponible en ligne : `http://www.deeplearningbook.org`.

[15] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, no 1, pp. 267–288, 1996.

[16] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh et D. Batra, "Grad-CAM : Visual explanations from deep networks via gradient-based localization," in *International Conference on Computer Vision (ICCV)*, pp. 618–626, 2017.

[17] F. Chollet, *Deep Learning with Python*. Manning Publications, 2017.

[18] O. Russakovsky, J. Deng, H. Su *et al.*, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no 3, pp. 211–252, 2015.

[19] S. J. Pan et Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no 10, pp. 1345–1359, 2010.