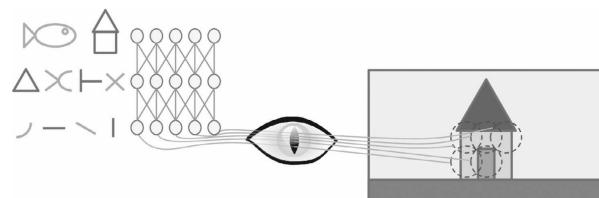




UNIVERSITÉ PARIS-EST CRÉTEIL

MÉMOIRE DE TER

Reconnaissance d'images par réseaux de neurones



Danian SUN
Guillaume PLANQUETTE
Master 1 de Mathématiques et
applications

*Encadrants : M. Jacques PRINTEMPS
Mme Sophie LARUELLE*

Remerciement

Nous sommes reconnaissant à Madame Sophie LARUELLE et à Monsieur Jacques PRINTEMS des aides fournis et de leur patience pour la rédaction de ce mémoire et la réalisation de ce projet de TER.

Table des matières

Introduction	3
1 Les réseaux de neurones et leur fonctionnement	4
1.1 Cousin biologique et cousin artificiel	4
1.1.1 Parlons un peu de la biologie	4
1.1.2 La transition vers l'artificiel	5
1.1.3 La création du perceptron	5
1.1.4 Le multicouche c'est meilleur	6
1.2 Algorithmes et fonctions	7
1.2.1 La rétropagation	7
1.2.2 La classification	8
1.2.3 La fonction d'activation	8
1.2.4 La fonction de coût	9
1.2.5 La descente de gradient	9
2 Les Réseaux de neurones convolutifs	11
2.1 L'inspiration biologique	11
2.2 Couche de convolution	11
2.2.1 Filtres	13
2.2.2 Empilage de cartes de caractéristiques	14
2.2.3 Besoins de RAM	15
2.3 Couche de Pooling	15
2.4 CNN	16
2.5 Exemples	17
2.5.1 LeNet-5	17
2.5.2 AlexNet	18
3 Cas concret	20

Introduction

Neural Network are at the beginning of what makes us intelligent creature. Thanks to them, we are able to visualise, think and realise our craziest ideas. Indeed our dream to fly at any cost, as birds involving in the wind, made us innovate in new ground to create planes capable to transport a huge amount of people through the air. Our intelligence, cleverness, comes from our neural network, and we have been able to create some artificial ones. Here we are on what interest us : These neural network, which made us so smart, allowed us to create what we can now commonly call : "Artificial Neural Network" (ANN or "Réseaux de neurones artificiels" RNA).

However, even if planes have birds as models, they don't flap their wings. The same applies to ANN (RNA) which became over the years quite different from their biological relatives. Some researchers even support the change of the denomination from "Neural " to "Unit" Network to highlight that the limitations met by the Biological Neural Network are not applicable to the ANN. Indeed, ANN are now powerful, adaptable and expandable. They are able to adapt themselves to a variety of extremely complex learning tasks, such as vocal recognition (Siri, Cortana, Ok Google, etc.), audiovisual product recommendation (Youtube, Spotify, etc.), acquire knowledge and learning to beat the Go Game World Champion (DeepMind's AlphaGo), classification of billion images (Google Images) and many more. Hence their crucial role in the deep learning environment.

In our case, we will be focused on the image classification which is a specific situation where ANNs face another major problem : the image recognition. To complete this task, we need to have taken the biological functioning of the human eye as a reference. At purpose, Convolutional Neural Network (CNN ou "Réseaux de neurones convolutifs") were created. They are very complex RNA which allow the system to deeply analysed images to identify every object or pattern we are looking for. A simple example could be radars, which measure vehicles speeds and identify their matriculation, through images taken by the camera, if they overspeed. The use of ANN and CNN in our daily life is already common and have with no doubt huge consequences on it.

Those neural networks freshly created were able to use the latest innovation in this subject. From the computational power increase to the simple access to important training database including multiple optimisation and training tricks, CNN have made possible big progress in young and not so developed field at the time. As mentioned above with the radars example, image and sound recognition, it opens a large innovation path in our society with the images and videos classification, recently with autonomous vehicles or vocal recognition.

In this thesis, we will start by explaining what clearly artificial neural network are and their functioning (how they work). Then we will describe convolutional neural network and finally show a concrete example of convolutional neural network used for images classification on a brand new database.

Chapitre 1

Les réseaux de neurones et leur fonctionnement

1.1 Cousin biologique et cousin artificiel

1.1.1 Parlons un peu de la biologie

Les neurones biologiques sont des cellules d'aspect inhabituelles que nous pouvons retrouver principalement dans les cerveaux d'animaux. elles sont constituées d'un corps cellulaire disposant d'un noyau et des éléments complexes qui composent une cellule classique. En plus de ceux-ci, elles possèdent de nombreux prolongements qu'on appelle des *dendrites*, et d'un long prolongement appelé *axone*. Sa longueur peut varier d'un *axone* à un autre, c'est-à-dire que ce dernier peut être aussi long que les *dendrites* mais peut aussi atteindre un mètre de longueur. Il se décompose en des ramifications qui portent le nom de *téloïdendron* à son extrémité qui quant à eux se terminent par des structures minuscules appelées *synapses terminales*. C'est ces derniers qui permettent les échanges de signaux électriques entre les différents neurones biologiques.

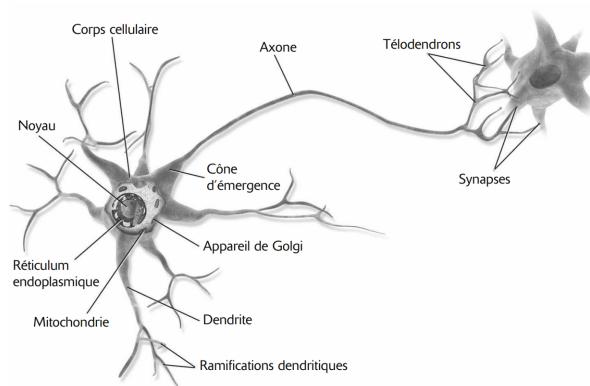


FIGURE 1.1 – Un neurone biologique

Un neurone biologique seul semble posséder des comportements relativement simples mais dans la nature les neurones sont organisés en de vaste réseau de milliards de neurones, chacun étant relié à des milliers d'autres. Ainsi un réseau de neurones relativement simple peut réaliser des calculs extrêmement complexes. Un exemple parlant serait le réseau des fourmis, ces derniers sont capables de s'organiser pour construire une fourmilière complexe voire s'assembler pour faire des figures de grandes envergures à leur échelle comme par exemple les célèbres ponts de fourmis pour traverser une rivière ou atteindre un objet en hauteur. L'architecture des réseaux de neurones fait encore l'objet d'une recherche active, et la cartographie du cerveau a permis de comprendre que les neurones sont souvent organisés en

couches successives notamment dans le cortex cérébral qui est la couche externe de notre cerveau.

1.1.2 La transition vers l'artificiel

Le premier modèle de neurone artificiel a été proposé par McCulloch et Pitts en 1943[1]. C'était un modèle très simpliste de neurone : il présentait une ou plusieurs entrées binaires qui peuvent être actives ou inactives et une sortie binaire. Le principe est que l'activation de la sortie du neurone se base sur un seuil de nombre de d'entrées à activer. Ainsi malgré le côté simpliste de ce modèle, il était possible de construire un réseau de neurones artificiels capable de calculer n'importe quelle proposition logique.

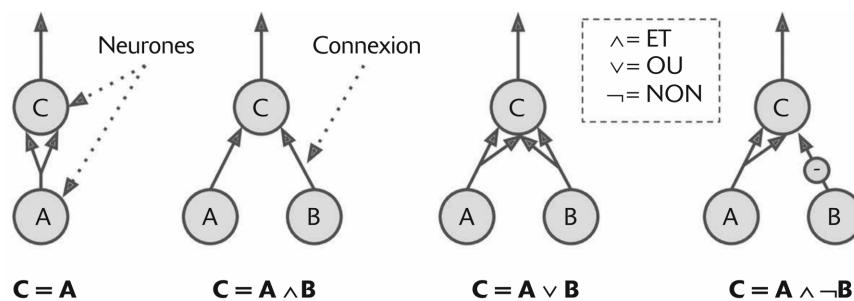


FIGURE 1.2 – Réseaux de neurones artificiels réalisant des calculs logiques élémentaires

Interprétation de la Figure 1.2 :

- $C = A$ désigne la fonction identité. Si le neurone A est activé alors le neurone C aussi. De même si le neurone A est désactivé, le neurone C le sera aussi.
- $C = A \wedge B$ réalise un ET logique. Le neurone C ne peut être activé que si le neurone A et le neurone B le sont.
- $C = A \vee B$ réalise un OU logique. Le neurone C peut être activé si le neurone A l'est ou si le neurone B l'est ou voire même si les deux le sont.
- $C = A \wedge \neg B$ réalise une proposition logique un peu plus complexe que les propositions d'au dessus. Le neurone C ne peut être activé que par l'activation du neurone A couplé à l'inactivité du neurone B, c'est-à-dire que si le neurone B est activé, alors le neurone C sera inactif même si le neurone A est actif.

Ainsi avec un peu d'imagination, il est pleinement possible de réaliser des réseaux neurones pouvant calculer des expressions logiques complexes. De plus, ce modèle étant le précurseur voire même l'origine de tous les modèles de réseau de neurones artificiels qui ont pu exister, il a pris pour nom : "le neurone formel".

1.1.3 La création du perceptron

Mais il y a tout de même des problèmes liés au fait que le neurone formel ne fonctionne qu'en *binaire* et ne possède pas encore la capacité d'apprendre ce qui font que ce dernier ne fonctionne pas complètement comme son cousin biologique. C'est alors que Frank Rosenblatt inventa en 1957[2] le modèle que l'on nomme le "*perceptron*". C'est une des architectures des RNA les plus simples qui se base sur un neurone artificiel légèrement différent du neurone formel. Ce neurone artificiel est appelé *unité logique à seuil* (TLU, Threshold Logic Unit) ou parfois *unité linéaire à seuil* (LTU, Linear Threshold Unit). Les entrées et la sortie sont à présent des nombres (x_1, x_2, \dots, x_n) au lieu des valeurs binaires et chaque connexion en entrée possède un poids (w_1, w_2, \dots, w_n). Ainsi le TLU calcule une somme pondérée des entrées ($z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{x}^T\mathbf{w}$) puis il applique une fonction échelon (step) à cette somme et produit le résultat : $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{x}^T\mathbf{w})$.

Un perceptron étant constitué d'une seule couche de TLU et chaque TLU étant connecté à toutes les entrées, cette couche est appelée une couche intégralement connectée, ou couche dense. Les entrées du *perceptron* sont connectés à des

neurones d'entrée qui se contentent de sortir l'entrée qui leur a été fournie. L'ensemble des neurones d'entrée forment la couche d'entrée. Mais il arrive souvent qu'une caractéristique de biais soit ajoutée, elle a pour nom *neurone de terme constant*, ou *neurone de biais* et sa sortie est toujours constante et égale à 1.

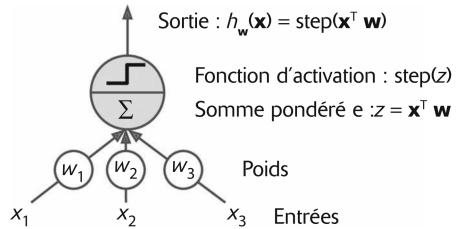


FIGURE 1.3 – Une unité logique à seuil (LTU)

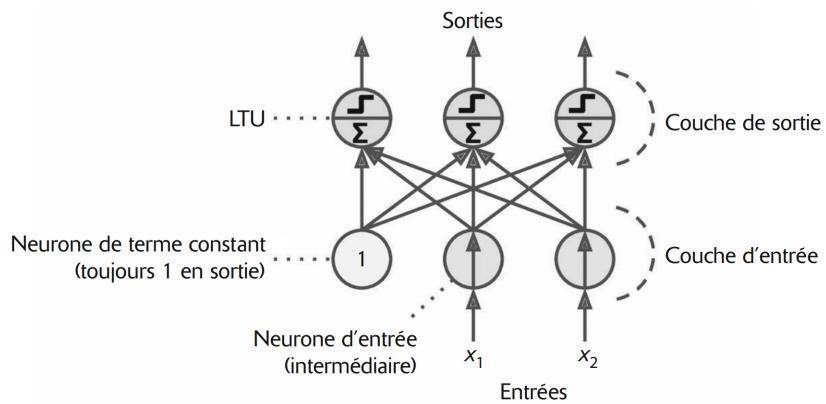


FIGURE 1.4 – Architecture de perceptron avec deux neurones d'entrée, un neurone de terme constant et trois neurones de sortie

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

Dans cette équation :

- \mathbf{X} représente la matrice des caractéristiques d'entrée. Elle comprend une ligne par instance et une colonne par caractéristique.
- \mathbf{W} est la matrice des poids, elle contient tous les poids de connexions excepté ceux des neurones de terme constant. Elle comprend une ligne par neurone d'entrée et une colonne par neurone artificiel de la couche.
- \mathbf{b} est le vecteur de termes constants , il contient tous les poids des connexions entre le neurone de terme constant et les neurones artificiels. Il comprend un terme constant par neurone artificiel.
- ϕ est la fonction d'activation, dans le cas des TLU il s'agit d'une fonction échelon.

La fonction échelon la plus utilisée dans les perceptrons est la fonction heaviside et parfois la fonction signe. Qui nous donne si on suppose que le seuil soit égal à 0 :

$$\text{heaviside}(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{si } z < 0 \\ 0 & \text{si } z = 0 \\ 1 & \text{si } z > 0 \end{cases}$$

1.1.4 Le multicouche c'est meilleur

Mais en 1969[3] Marvin Minsky et Seymour Papert ont révélé que le perceptron avait tout de même d'importantes faiblesses, notamment le fait qu'il soit incapable de résoudre certains problèmes triviaux, comme le problème de classi-

fication du OU exclusif. Cependant il était possible de lever cette limite grâce au *perceptron multicouche* (PMC). Un PMC est avant tout un empilement de plusieurs *perceptrons*. Il est constitué d'une couche d'entrée, d'une ou plusieurs couches de TLU appelées *couches cachées* et d'une dernière couche de TLU appelée *couche de sortie*. Chaque couche, à l'exception de la *couche de sortie*, comprend un neurone de terme constant et elle est intégralement reliée à la suivante.

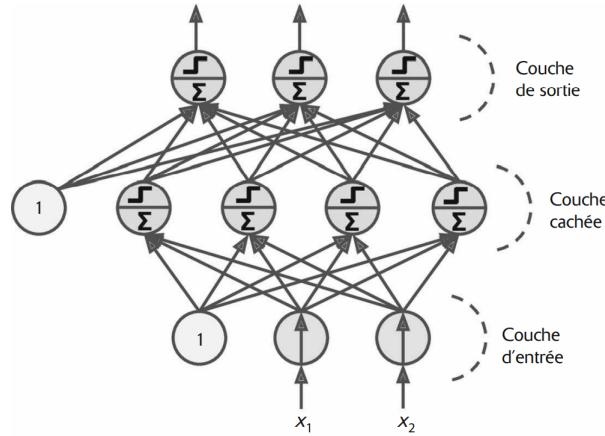


FIGURE 1.5 – Perceptron multicouche avec deux entrées, une couche cachée de quatre neurones et trois neurones de sortie

1.2 Algorithmes et fonctions

1.2.1 La rétropropagation

Pendant plusieurs années, les chercheurs n'arrivaient pas à trouver une manière d'entraîner les PMC, c'était le premier hiver de l'IA. Et il a fallut attendre 1986[4] pour que David Rumelhart, Geoffrey Hinton et Ronald Williams introduisent dans un article révolutionnaire un algorithme d'entraînement à *rétropropagation* qui est toujours très utilisé de nos jours.

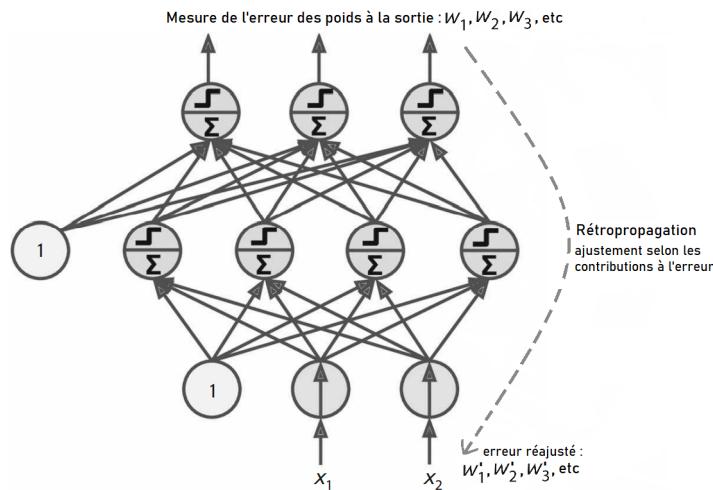


FIGURE 1.6 – Rétropropagation et réajustement des poids des connexions w_1, w_2, w_3 , etc.

Le principe de cet algorithme de *rétropropagation* est le suivant : pour chaque instance d'entraînement, l'algorithme de *rétropropagation* commence par effectuer une prédiction (*passe vers l'avant*), mesure l'erreur en comparant la sortie souhaitée et la sortie réelle, traverse chaque couche en arrière pour mesurer la contribution à l'erreur de chaque connexion (*passe vers l'arrière*) et termine en ajustant légèrement les poids des connexions de manière à réduire l'erreur avec l'étape de descente de gradient (que nous verrons plus tard) avec la contribution à l'erreur calculée juste avant.

1.2.2 La classification

Dans notre cas, notre système se base sur un algorithme d'apprentissage communément appelé *la classification*. Cette dernière permet à notre système d'apprendre à partir d'une base de données étiquetées. Un exemple simple et commun serait le filtre anti-spam que nous pouvons retrouver dans nos boîtes mails. Son apprentissage se base sur l'identification des e-mails frauduleux et des e-mails normaux à partir d'une base de données connue de mails signalés par des utilisateurs. Les données étant étiquetées, nous connaissons donc les réponses d'avance. Ainsi en entraînant le système sur une telle base de données, cela nous permet par exemple : d'établir une mesure de performance en pourcentage de réussite sur l'exactitude du filtre après l'entraînement. Ce genre d'apprentissage est appelé entre-autre un *apprentissage supervisé*.

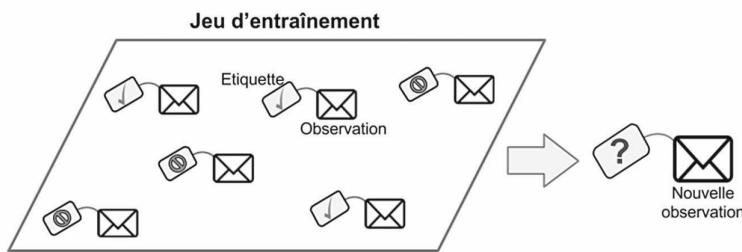


FIGURE 1.7 – Jeu d'entraînement étiqueté pour une tâche de classification

1.2.3 La fonction d'activation

Pour que l'algorithme de *rétropropagation* puisse fonctionner correctement, il a fallut aussi apporter une modification essentielle à l'architecture du PMC. La *fonction échelon* a dû être remplacée par la *fonction logistique* :

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

Ce changement est fondamental, car la *fonction échelon* qui n'est composée que de sections plats ne pouvait avoir de gradient. Tandis que la fonction logistique possède une dérivée non nulle en tout point ce qui permet de faire la *rétropropagation* à l'aide du gradient. Il y a aussi d'autres *fonctions d'activations* possibles à la place de la *fonction logistique*, comme :

- La fonction tangente hyperbolique :

$$\tanh(z) = 2\sigma(2z) - 1 = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

Cette fonction d'activation a une forme de "S" et est continue et dérivable, mais ses valeurs de sorties se trouvent sur] - 1, 1[à la place de]0, 1[pour la fonction logistique. Ce qui permet de centrer en zéro la sortie de chaque couche au début de l'entraînement.

- La fonction ReLU (Rectified Linear Unit) :

$$\text{ReLU}(z) = \max(0, z)$$

Cette fonction est continue mais non dérivable en $z = 0$ et sa dérivée pour $z < 0$ est 0. Cependant, elle fonctionne très bien et a l'avantage d'être rapide à calculer. Ce qui fait d'elle la fonction par défaut que l'on utilise.

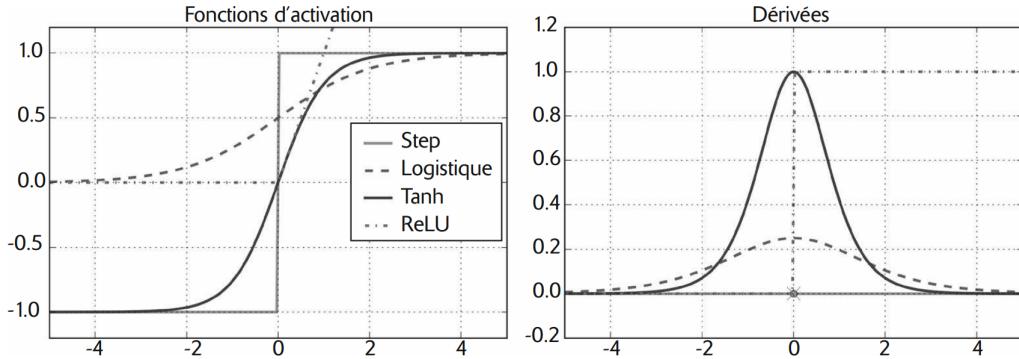


FIGURE 1.8 – Fonctions d'activation et leur dérivée

1.2.4 La fonction de coût

La mesure de performance établit s'appelle *la fonction de coût* ou *la fonction de perte*, et comme son nom l'indique c'est une fonction qui nous donne le coût du système sur un jeu de données. Dans la classification, la fonction de coût la plus commune est la fonction "*binary cross-entropy*" qui permet de mesurer la différence entre deux probabilités de distribution pour une variable prise aléatoirement. Mais dans le cas où il y a plus de deux probabilités de distribution, il faudra utiliser la fonction "*categorical cross-entropy*". Voici comment sont ces deux types de fonctions :

$$\begin{aligned} \text{LossBinary} &= -\frac{1}{\text{taille de sortie}} \sum_{i=1}^{\text{taille de sortie}} \theta_i \cdot \log(\hat{\theta}_i) + (1 - \theta_i) \cdot \log(1 - \hat{\theta}_i). \\ \text{LossCategorical} &= -\sum_{i=1}^{\text{taille de sortie}} \theta_i \cdot \log(\hat{\theta}_i) \end{aligned}$$

Dans les deux fonctions :

- $\hat{\theta}_i$ représente la ième valeur scalaire du vecteur donnant les valeurs de sortie prédites par le système
 - θ_i représente la ième valeur scalaire du vecteur donnant les valeurs de sortie souhaitées
- Il existe bien sûr d'autres sortes de fonctions de coût qui sont adaptées à des situations encore plus délicates.

1.2.5 La descente de gradient

Dans l'optimisation de la fonction de coût, il y a aussi la descente de gradient qui est énormément utilisée. Cette dernière consiste à corriger petit à petit les paramètres pour minimiser la fonction de coût, mais pour cela, la fonction de coût doit être convexe. Elle calcule donc le gradient de la fonction de coût à un point donné Θ , puis progresse en direction du gradient descendant. Et lorsque le gradient est nul, on aura atteint le minimum.

Pour ce faire, il faut d'abord remplir Θ avec des valeurs aléatoires, on appelle cette étape *l'initialisation aléatoire*. Puis on l'améliore progressivement en tentant de faire décroître la fonction de coût jusqu'à ce que l'algorithme converge vers un *minimum*. C'est ici qu'intervient l'hyperparamètre le *taux d'apprentissage* : η (learning rate), c'est un paramètre de l'algorithme d'apprentissage on ne cherchera donc pas à l'optimiser pendant l'apprentissage. Mais nous pouvons tout

de même le choisir, voire le tester, manuellement pour trouver le η le plus adapté au modèle. Son rôle est de déterminer la dimension des pas que va prendre notre algorithme pour faire la descente de gradient. Si η est trop petit l'algorithme devra effectuer énormément d'itération pour converger, et cela peut prendre beaucoup de temps. Tandis que si η est trop élevé, on risque de dépasser le point le plus bas et finir de l'autre côté, voire peut-être plus haut qu'avant, ce qui pourrait aussi faire diverger l'algorithme.

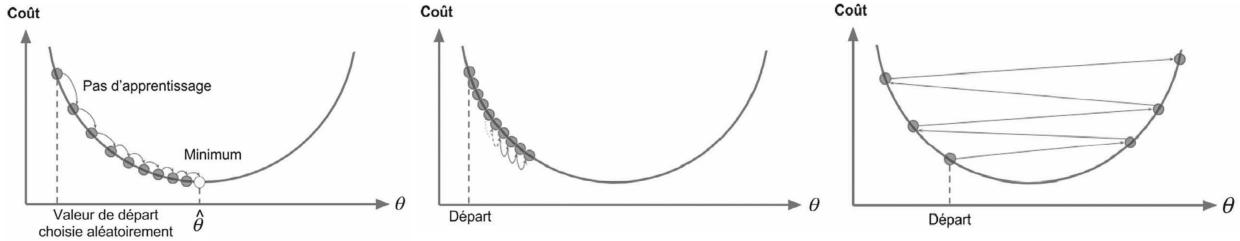


FIGURE 1.9 – De gauche à droite : des descentes de gradient avec un taux d'apprentissage η correct, trop petit et trop élevé

Pour implémenter une descente de gradient, il suffit de calculer le gradient de la fonction de coût par rapport à chaque paramètre θ_j du modèle, donc sa dérivée partielle. Ce qui nous donne pour une fonction de coût f :

$$\nabla_{\Theta} f(\Theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} f(\Theta) \\ \frac{\partial}{\partial \theta_1} f(\Theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} f(\Theta) \end{pmatrix}$$

Une fois qu'on a le vecteur gradient, il suffira d'aller dans la direction opposée pour faire la descente, ce qui revient à soustraire $\nabla_{\Theta} f(\Theta)$ de Θ . C'est ici qu'apparaît le *taux d'apprentissage* η soit :

$$\Theta^{(\text{étape suivante})} = \Theta - \eta \nabla_{\Theta} f(\Theta)$$

Il existe plusieurs méthodes pour permettre à la descente de gradient d'atteindre le minimum plus rapidement. Il y a la *normalisation* qui permet d'ajuster la taille des variables à une échelle identique pour que la descente de gradient soit plus efficace, ou la *descente de gradient stochastique* qui, contrairement à la descente de gradient classique qui utilise l'ensemble du jeu d'entraînement pour calculer les gradients à chaque étape, n'utilise qu'une observation prise au hasard à chaque étape dans l'ensemble d'entraînement et calcule les gradients en ne se basant que sur cette seule observation à chaque fois, ou encore la *descente de gradient par mini-lots* qui quand à elle, comme son nom l'indique, prend de petits sous-ensembles d'observations sélectionnées aléatoirement pour calculer les gradients à chaque étape.

Chapitre 2

Les Réseaux de neurones convolutifs

2.1 L'inspiration biologique

À l'image des neurones, la structure du cortex visuel est tout d'abord étudiée sur des animaux, plus précisément des chats, dans les années 1958[5] par David H.Hubel et en 1959[6] par Torsten Wiesel. Est alors observé que certains neurones du cortex visuel ont un champ récepteur local qui restreint la région du champ visuel et ne traite que les stimulus visuel dans cette région. Les différentes régions peuvent se chevaucher et recouvrant ainsi l'ensemble du champ visuel.

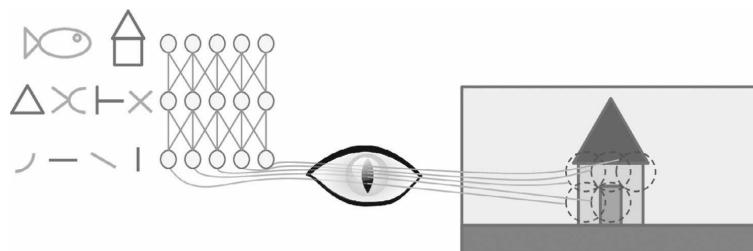


FIGURE 2.1 – À chaque neurone du cortex visuel est attribué un motif spécifique dans sa région ; les motifs étant de plus en plus complexes au fil du réseau de neurone

L'architecture de ces réseaux de neurones semble complexe mais cela ne la rend que plus puissante. Comme représenté dans la *figure 2.1*, chaque neurone n'est connecté qu'à certains neurones en amont. Même s'ils peuvent avoir un champ récepteur commun, ils n'étudient pas les même motifs. D'autant que pour les neurones de plus haut niveau, certains s'acquittent de motifs plus complexes en utilisant des combinaisons de motifs étudiés à plus bas niveau. De ces architectures est apparu le *neocognitron*[7] en 1980 qui donnera bien plus tard les *réseaux de neurones convolutifs*. Parmis les grandes étapes du développement de ces réseaux de neurones, l'intégration de *couches de convolution* et de *couches de pooling* a permis la création d'architecture célèbre encore utilisée aujourd'hui comme *LeNet-5* présenté en 1998[9].

2.2 Couche de convolution

Au même titre que notre cortex visuel, on utilise des champs récepteurs locaux afin de décortiquer et réduire les images à traiter. Ici ce sont des couches de convolution empilées qui font ce travail.

À la différence des RNA, les neurones de la première couche ne sont connectés qu'aux pixels situés à l'intérieur de leurs champs récepteurs respectifs. Ainsi avec cette même logique, les neurones suivant ne sont connectés qu'avec ceux dont le centre du champ récepteur est à l'intérieur d'un petit rectangle de la couche précédente.

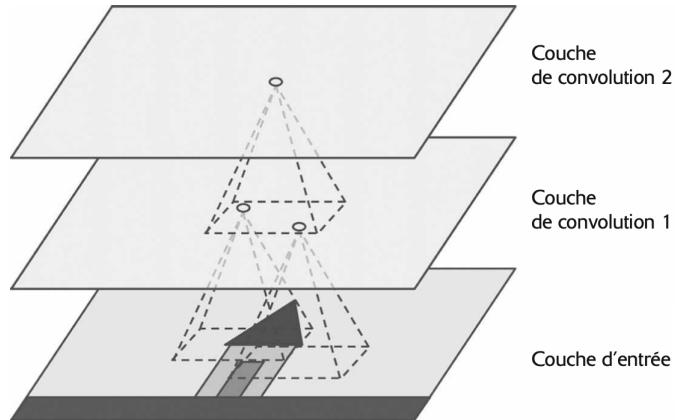


FIGURE 2.2 – Couches d'un CNN avec des champs récepteurs locaux rectangulaires

Cette logique permet donc de décortiquer n'importe quels motifs complexes en plus petit motifs étudié par des neurones de plus bas niveaux.

Pour se représenter les couches de neurones plus facilement, on peut les aplatisir en surface quadrillée de deux dimensions. Chaque neurone représentant un carré d'une couche de taille $n \times m$.

Pour représenter le passage d'une couche à une autre, prenons un neurone situé en ligne $i \leq n$ et colonne $j \leq m$ d'un couche donnée. Ce neurone est connecté aux sorties des neurones de la couche précédente situés dans un rectangle de taille $f_h \times f_w$. C'est à dire, aux neurones aux lignes i à $i + f_h - 1$ et colonnes j à $j + f_w - 1$.

Pour que les deux couches aient les mêmes hauteur et largeur, on rajoute un cadre de zéro autour de la couche inférieure, comme une marge. Cela se nomme un *remplissage par zéros* (*zero padding*) ou bien plus simplement un *ajout d'une marge de zéros*.

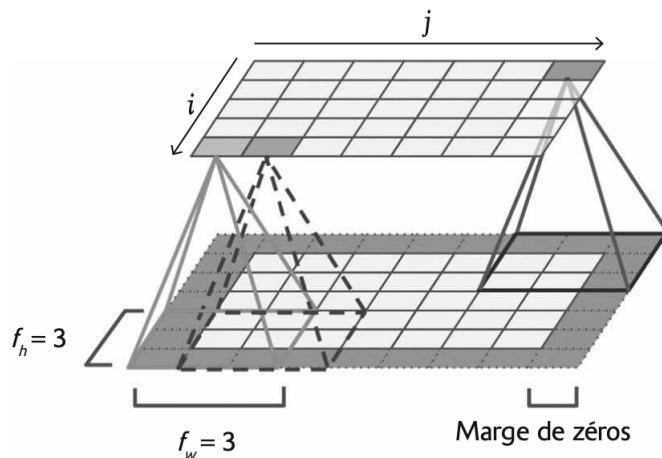


FIGURE 2.3 – Connexions entre deux couches avec l'ajout d'une marge de zéros

Lors de l'écriture de l'algorithme, il est également possible d'attribuer à la variable "*padding*" la valeur "*VALID*" qui n'ajoute pas de marge de zéros et peut ignorer certaines lignes et colonnes à la place. L'autre valeur attribuable pour avoir une marge de zéros étant "*SAME*".

Dans certains cas, à cause d'une image de trop haute définition par exemple et pour diminuer la quantité de calcul, il est nécessaire de connecter une couche d'entrée à une couche plus petite. Pour cela, on peut espacer les champs récepteurs à l'aide d'un pas, vertical s_h et horizontal s_w .

Dans la figure 2.5, les deux pas sont égaux, ce qui n'est pas obligatoire, et en partant d'une couche d'entrée de taille

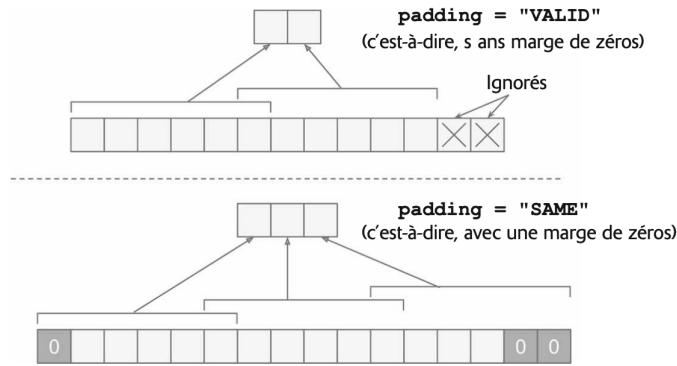


FIGURE 2.4 – Options de padding "SAME" et "VALID" avec une largeur d'entrée de 13, une largeur de filtre de 6 et un pas de 5

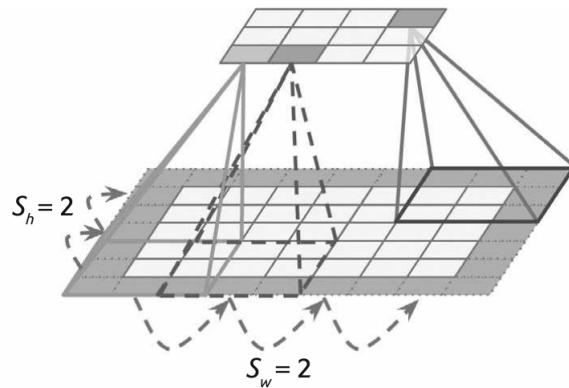


FIGURE 2.5 – Passage à une couche plus petite à l'aide d'un ajout de pas de 2

5×7 , on arrive à une couche de taille 3×4 en utilisant des champs récepteurs 3×3 .

Par rapport à notre premier exemple, le même neurone, en ligne i et colonne j , est connecté aux neurones aux lignes $i \times s_h$ à $(i \times s_h) + f_h - 1$ et aux colonnes $j \times s_w$ à $(j \times s_w) + f_w - 1$.

2.2.1 Filtres

À chaque neurone est associé une matrice de poids, de la taille du champ récepteur, aussi appelée filtre (ou *noyau de convolution*). Cela peut être un filtre vertical (comme une fente sur le champ récepteur) défini par une matrice remplie de 0 à l'exception de la colonne centrale pleine de 1. Ou bien un filtre horizontal avec cette fois ci uniquement une ligne horizontale centrale de 1 (voir Figure 2.6).



FIGURE 2.6 – Deux filtres donnant des cartes de caractéristiques différentes

Si l'on applique à une image, une couche de neurone utilisant le même filtre on obtient une carte de caractéristiques (*feature map*) spécifique. Avec l'exemple du filtre vertical, l'image est altérée, les lignes blanches verticales sont mises en valeur et le reste devient flou.

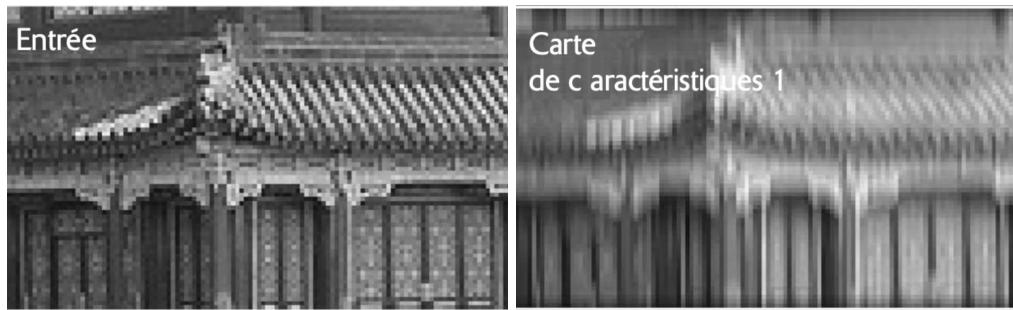


FIGURE 2.7 – Application du filtre vertical à une image

Ces filtres sont automatiquement sélectionnés par la couche de convolution selon leur utilité pour accomplir la tâche donnée. Les couches supérieures apprennent alors à combiner ces filtres en motifs plus complexes.

2.2.2 Empilage de cartes de caractéristiques

Dans le but de faire des architectures plus puissantes, les couches de convolutions peuvent être composées de plusieurs filtres, chacun associé à une carte de caractéristiques, qui s'appliquent simultanément à toutes leurs entrées. Chacune des cartes de caractéristiques possède un neurone par pixel et l'ensemble des neurones d'une carte a la même paramétrisation (poids et terme constant). Le champ récepteur d'un neurone s'étend cependant sur toutes les cartes de caractéristiques.

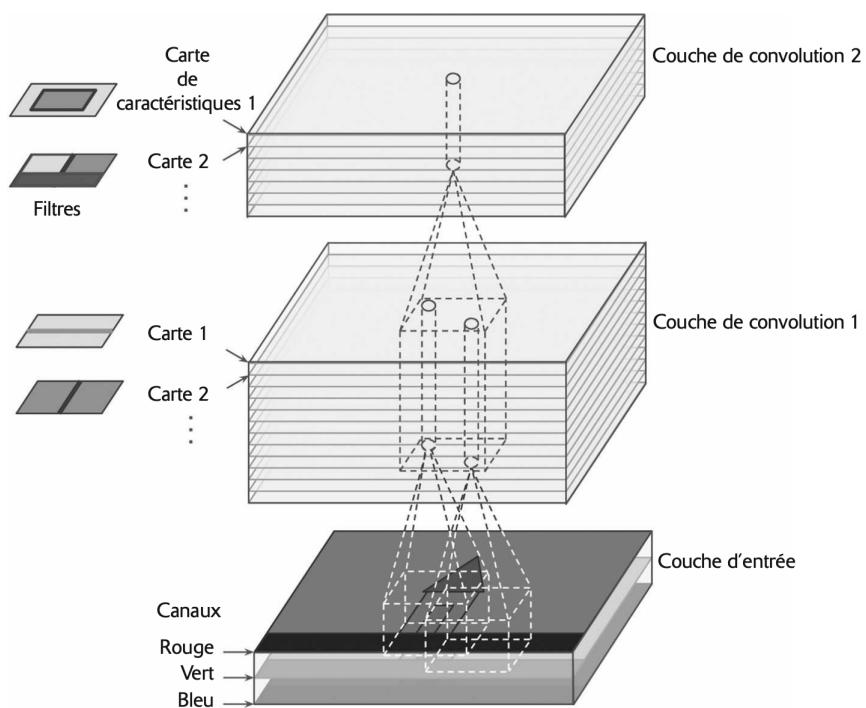


FIGURE 2.8 – Couches de convolution avec plusieurs cartes de caractéristiques appliquées à une image en RVB.

Suivant l'image en entrée, il peut y avoir plusieurs couches à traiter (trois en RVB : Rouge-Vert-Bleu, une en nuance de gris, ou plus avec par exemple les fréquences lumineuses).

Ainsi, la même logique utilisée avec une carte de caractéristiques peut être appliquée avec plusieurs cartes dans une même couche de convolution. Tous les neurones situés de la ligne $i \times s_h$ à $(i \times s_h) + f_h - 1$ et de la colonne $j \times s_w$ à $(j \times s_w) + f_w - 1$, de toutes les cartes de caractéristiques de la couche l , sont connectés au neurone en ligne i et en colonne j de la carte de caractéristiques k dans la couche de convolution $l + 1$. Il en est de même pour tous les neurones en i et j dans les autres cartes de caractéristiques de cette couche.

On peut calculer la sortie $z_{i,j,k}$ de ce neurone donné dans une couche de convolution $l + 1$ avec cette équation :

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{w=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} \mathbf{x}_{i',j',k'} \cdot \mathbf{w}_{u,v,k',k} \quad \text{avec } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

où :

- $\mathbf{x}_{i',j',k'}$ est la sortie du neurone dans la couche l , ligne i' , colonne j' et carte de caractéristiques k' (ou canal k' si la couche précédente est la couche d'entrée).
- b_k est le terme constant de la carte de caractéristiques k dans la couche l , qui agit comme un réglage de la luminosité locale par exemple.
- $\mathbf{w}_{u,v,k',k}$ le poids de la connexion entre tout neurone de la carte de caractéristiques k de la couche $l + 1$ et l'entrée située ligne u et colonne v dans la carte de caractéristiques k' .

2.2.3 Besoins de RAM

Les couches de convolution ont besoin d'une grande quantité de RAM et surtout lors de l'entraînement. Pendant la rétropropagation, toutes les valeurs intermédiaires calculées pendant la passe avant sont utilisées.

C'est à dire que pour chaque couche de convolution avec des filtres 5×5 , on a 200 cartes de caractéristiques de taille 150×100 , avec un pas de 1 et un padding "same". Ainsi, avec une image RVB, avec donc trois canaux, de 150×100 , on obtient 15200 paramètres. En prenant en compte les 150×100 neurones des 200 cartes de caractéristiques, on arrive à près de 225 millions de calculs à faire par la RAM pour une seule instance (image) d'entraînement.

Afin de résoudre ce problème, l'une des solutions est d'ajouter des couches de pooling afin de *sous-échantillonner* (*ie.* rétrécir) l'image d'entrée, ce qui réduit le nombre de calcul et donc l'utilisation de la mémoire. Cela réduit également le nombre de paramètre, ce qui limite le risque de surajustement.

2.3 Couche de Pooling

La connectivité des neurones entre les couches est similaire dans les couches de convolution que dans les couches de pooling, chaque neurone n'est connecté qu'à quelques neurones de la couche inférieure situés dans un petit champ récepteur rectangulaire. De même, on définit sa taille, le pas et le type de padding. Cependant les neurones d'une couche de pooling n'ont pas de poids et traitent les entrées à l'aide d'une fonction d'agrégation telle que la moyenne ou la valeur maximale (qui est la plus répandue).

Dans le cas de la figure 2.9 (max pooling), seule la valeur d'entrée maximale dans chaque champ récepteur permet le passage à la couche suivante. L'image de sortie a également une hauteur et une largeur deux fois plus petite que celle d'entrée à cause du pas de 2.

En plus des avantages cités plus haut, une couche de pooling ajoute un degré d'*invariance* envers les petites translations comme représenté dans la figure 2.10. Ici on suppose que les pixels clairs ont une valeur inférieure aux pixels sombres et on prend trois images quasi identiques à une translation près. Avec une couche de pooling maximum, seule la sortie de l'image C est différente malgré qu'il reste 75% d'invariance.

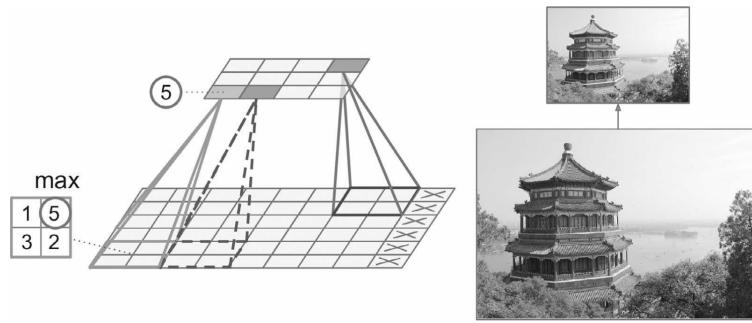


FIGURE 2.9 – Couche de pooling maximum (max pooling) avec un noyau 2×2 , un pas de 2 et aucune marge de zéro"

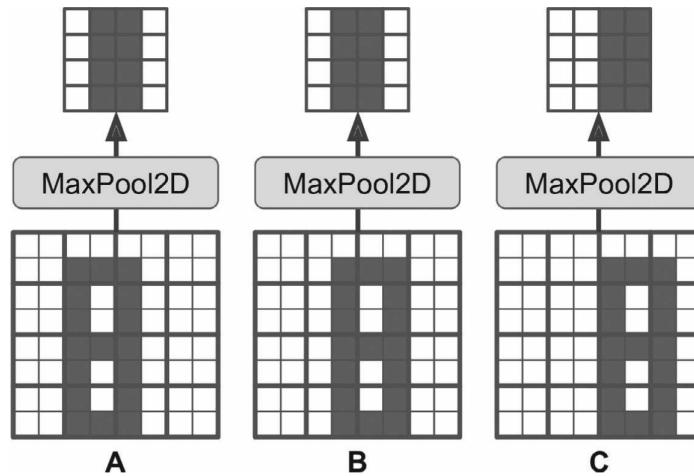


FIGURE 2.10 – Couche de pooling maximum (max pooling) avec un noyau 2×2 , un pas de 2 et aucune marge de zéro"

Ainsi, en ajoutant des couches de pooling entre deux couches de convolution, on peut s'assurer un degré d'invariance de translation à toutes les étapes. Par ailleurs, une petite quantité d'invariance de rotation et d'échelle sont également apportées par une couche de pooling ce qui se révèle être des plus utile.

Cependant, le principe du maximum est très destructeur. Comme on peut le voir sur la figure 2.9, la sortie a une surface quatre fois plus petite, ce qui supprime 75% des valeurs d'entrées. De plus, l'invariance est différente de l'*équivariance* qui est nécessaire dans certains cas où le décalage de l'image de sortie d'un pixel n'est pas souhaitable. Il existe donc une autre possibilité comme couche de pooling, la *couche de pooling moyen* (*average pooling*).

À l'inverse, on peut retrouver dans certains algorithmes des *couches de pooling moyen global* qui sont extrêmement destructrices. Ces couches se limitent à calculer la moyenne sur l'intégralité de chaque carte de caractéristiques et ne produit donc qu'une seule valeur par carte.

2.4 CNN

L'architecture des CNN est donc bien plus compliquée que les RNA vu dans le premier chapitre puisqu'elle intègre plusieurs couches de convolution et de pooling. Ces couches sont souvent empilées avec en plus une couche de ReLU à la suite de chaque couche de convolution. À mesure que l'image traverse le réseau elle rétrécit mais devient également de plus en plus profonde grâce aux couches de convolution. On ajoute souvent au sommet un réseau de neurones non bouclés classique avec quelques couches entièrement connectées avec des couches ReLU. La couche finale produit une prédiction telle que par exemple une couche softmax qui génère des probabilités de classe et peut donc être utile dans

notre cas avec des algorithmes de classification.

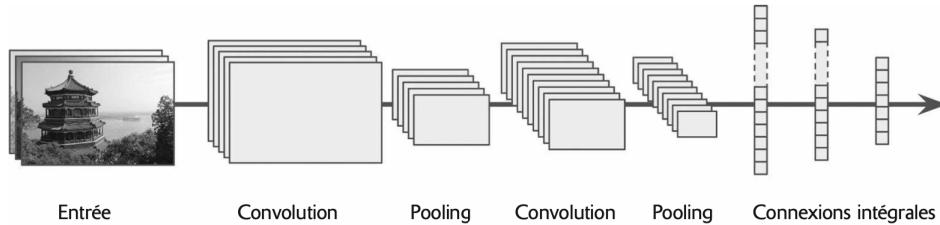


FIGURE 2.11 – Architecture de CNN classique

Contrairement aux autres couches, il est préférable d'utiliser une couche de convolution avec un noyau plus grand pour la première couche ce qui permet de diminuer la dimension spatiale de l'image sans perdre trop d'informations et surcharger les calculs.

Plus l'image est grande, plus on peut multiplier l'enchaînement de quelques couches de convolution et d'une couche de pooling. Le nombre de filtre par couche de convolution augmente (souvent multiplié par deux à chaque couche) plus l'on traverse le réseau afin de travailler sur des motifs plus compliqués.

En plus de ces différentes couches, on peut également ajouter des couches de *dropout* qui est une technique de régularisation très répandue qui réduit également le besoin en ressources matériels (RAM). Cette technique consiste à "éteindre" (ignorer) des neurones avec une probabilité p , appelée taux d'extinction, pendant une étape de l'entraînement. Une autre technique de régularisation est la normalisation qui stabilise la distribution des données au cours de l'apprentissage. Elles permettent ainsi toutes les deux de redimensionner l'image ou de réguler les données.

Une technique de régulation a également été mise au point pour diminuer le surajustement : l'augmentation des données. Utilisée notamment dans l'algorithme **AlexNet**[10], cette technique consiste à augmenter artificiellement la taille du jeu d'entraînement en variant de façon réaliste quelques paramètres des images d'entraînement ou encore en modifiant l'orientation et le cadre de ces images. D'autres paramètres tels que le contraste ou la luminosité peuvent également être modifiés pour au final augmenter considérablement le jeu d'entraînement.

2.5 Exemples

Dans cette partie, nous allons vous présenter quelque exemple architecture bien connu utilisée dans la classification d'image notamment pendant des compétitions comme le défi ILSVRC ImageNet[8]. Cette compétition a vu le taux d'erreur pour la classification d'images des meilleurs algorithmes passer de plus de 26% à un peu moins de 2.3% en six ans. Les images utilisées lors de cette compétitions sont haute de 256 pixels et répartie dans plus de 1000 classes. Nous resterons sur des algorithmes relativement peu profond et complexe et qui donnent de très bon résultat malgré tout. Il existe bien d'autre plus complexe tel que le GoogLeNet[11] qui bénéficie de l'intégration d'un module *Inception* ainsi que des couches de normalisation et une fonction d'activation ReLu suivant les couches convolution. Mais également le ResNet[12] qui est un réseau résiduel (*résiduel network* ou ResNet), un CNN de 152 couches très profond utilisant des connexions de saut qui composent des unités résiduels pour faciliter l'entraînement.

Parmis les plus classiques, nous verrons l'architecture LeNet-5 classique (1998) et l'un des premiers gagnants du défi ILSVRC : AlexNet (2012)

2.5.1 LeNet-5

Cette architecture est la première et la plus connue qui alterne couche de convolution et couche de pooling. Elle est largement utilisée pour la reconnaissance des chiffres écrits à la main (MNIST[14]). Avant même l'entrée dans le réseau, les images sont redimensionnées avec une marge de zéros pour atteindre une taille de 32×32 pixels.

Couche	Type	Cartes	Taille	Taille de noyau	Pas	Activation
Out	Intégralement connectée	–	10	–	–	RBF
F6	Intégralement connectée	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Pooling moyen	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Pooling moyen	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Entrée	1	32×32	–	–	–

FIGURE 2.12 – Architecture LeNet-5

Pour l’architecture même du réseau, plusieurs couches de convolution et de pooling sont placées les unes après les autres. Chaque couche de pooling étant légèrement spécifique dans le calcul de ses sorties. Et de même pour la couche de sortie qui dans son calcul de sortie, pénalise davantage les mauvaises prédictions et produit des gradients plus importants et donc qui converge plus rapidement.

2.5.2 AlexNet

L’architecture de l’AlexNet a marqué ses esprits après avoir remporté le défi ILSVRC ImageNet largement en 2012. C’est le premier algorithme à empiler des couches de convolution ainsi qu’à utiliser une augmentation de données et deux Dropout. Les deux dernières techniques permettent de réguler le système et d’éviter le surajustement, en particulier l’augmentation de données. Le Dropout utilisé, uniquement à l’entraînement, aux sorties des deux dernières couches, F9 et F10, a un taux d’extinction assez élevé de 50%.

Nom	Type	Cartes	Taille	Taille de noyau	Pas	Remplissage	Activation
Out	Intégralement connectée	–	1 000	–	–	–	Softmax
F10	Intégralement connectée	–	4 096	–	–	–	ReLU
F9	Intégralement connectée	–	4 096	–	–	–	ReLU
S8	Pooling maximum	256	6×6	3×3	2	valid	–
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Pooling maximum	256	13×13	3×3	2	valid	–
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Pooling maximum	96	27×27	3×3	2	valid	–
C1	Convolution	96	55×55	11×11	4	same	ReLU
In	Entrée	3 (RVB)	227×227	–	–	–	–

FIGURE 2.13 – Architecture AlexNet

Une autre particularité de cette architecture est la *normalisation de réponse local* (LRN ; *Local Response Normalization*) utilisée après l'étape ReLU des couches $C1$ et $C3$. On peut représenter la LRN avec cette équation :

$$b_i = a_i \left(k + \alpha \sum_{j=j_{bas}}^{j_{haut}} a_j^2 \right)^{-\beta} \quad \text{avec} \begin{cases} j_{haut} = \min(i + \frac{r}{2}, f_n - 1) \\ j_{bas} = \max(0, i - \frac{r}{2}) \end{cases}$$

où :

- \mathbf{b}_i est la sortie du neurone dans la carte de caractéristiques i .
- \mathbf{a}_i est l'activation de ce neurone avant la normalisation et après l'étape ReLu.
- \mathbf{k} , α , β et \mathbf{r} sont des hyperparamètres. k étant le *terme constant* et r le *rayon de profondeur*. Dans AlexNet ils valent : $k = 1$, $\alpha = 0.00002$, $\beta = 0.75$ et $r = 2$.
- \mathbf{f}_n le nombre de cartes de caractéristiques.

Cette normalisation permet d'appliquer la rivalité d'activation, observé avec les neurones biologique, en gardant uniquement le résultat d'un seul neurone, s'il s'active fortement, sans prendre en compte les neurones situés au même endroit dans les autres cartes de caractéristiques voisines. Cela crée une spécialisation des cartes de caractéristiques et améliore la généralisation.

Chapitre 3

Cas concret

Nous allons utiliser notre programme en annexe pour montrer ce que peut faire un CCN dans un cas concret. Ici la base de données d'images sera une de Keras qui s'appelle CIFAR-10 dataset : il s'agit d'images en couleurs de taille 32 par 32 pixels réparties en 10 classes avec 6000 images par classe. Les 10 classes sont : "airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship". Chaque image est représentée par un tableau numpy de taille (32,32,3) (pixels + couleurs) et le label est entre 0 et 9. Voici un exemple que donne les premières images du jeu de données :



Nom	Type	Cartes	Taille	Taille de noyau	Pas	Rémpissage	Activation
Out	Intégralement connectée	—	10	—	—	—	Softmax
D15	Dropout 40%	—	128	—	—	—	—
F14	Intégralement connectée	—	128	—	—	—	ReLU
F13	Flatten	—	2048	—	—	—	—
D12	Dropout 30%	—	4 × 4	—	—	—	—
S11	Pooling Maximum	128	4 × 4	2 × 2	2	valid	—
C10	Convolution	128	8 × 8	3 × 3	1	same	ReLU
C9	Convolution	128	8 × 8	3 × 3	1	same	ReLU
D8	Dropout 30%	—	8 × 8	—	—	—	—
S7	Pooling Maximum	64	8 × 8	2 × 2	2	valid	—
C6	Convolution	64	16 × 16	3 × 3	1	same	ReLU
C5	Convolution	64	16 × 16	3 × 3	1	same	ReLU
D4	Dropout 40%	—	16 × 16	—	—	—	—
S3	Pooling Maximum	32	16 × 16	2 × 2	2	valid	—
C2	Convolution	32	32 × 32	3 × 3	1	same	ReLU
C1	Convolution	32	32 × 32	3 × 3	1	same	ReLU
In	Entrée	1	32 × 32	—	—	—	—

L'idée que nous avons eu a été de d'abord faire quelques couches cachées simples de convolution suivis de couches de pooling maximum pour commencer, mais le système finissait souvent par surajuster et ça a été un gros problème

pour nous. Par la suite nous avons du ajouter des couches de Dropout de 20% un peu partout pour justement enlever ces surajustements mais ce n'était pas suffisant et nous avons du éléver les taux de Dropout à 30% voire même 40% et c'est ce qui nous a permis d'avoir le modèle d'entraînement ci-dessus.

Comme il y a plusieurs classes de données et un label à attribuer à chaque fois, nous utiliserons la fonction de coût : *Categorical Cross-entropy* et la méthode de descente de gradient stochastique. Et après maintes tests manuels avec différents taux d'apprentissage, nous avons fini par choisir $\eta = 0.03$.

Nous obtenons donc les courbes d'apprentissages suivantes :

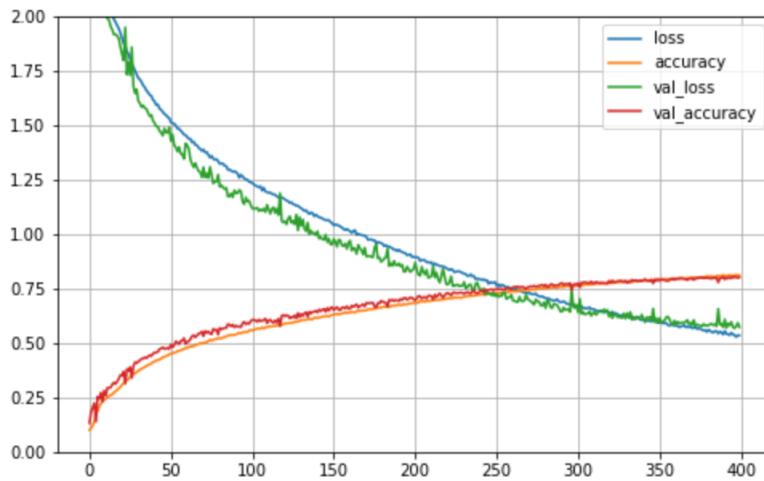


FIGURE 3.1 – Courbes obtenues après l'entraînement du modèle

Nous pouvons voir qu'après 400 entraînements et réajustements, nous avons obtenu un modèle avec à peu près 80% de précision sur le jeu de donnée `x_test` (représenté par `val_accuracy`). Ce qui veut dire que si on lui montre une image du même format que le jeu de donnée mais inconnu à notre système, il aura 80% de chance de trouver l'objet que représente l'image.

De plus, nous avons enregistré le modèle entraîné qui a la meilleure précision sur le jeu de données de test. Elle a une perte d'environ 57% et une précision de 80% ce qui est parfaitement cohérent avec le résultat de notre entraînement.

$$[0.5674042701721191, 0.8016999959945679]$$

Bibliographie

- [1] A Logical Calculus of the Ideas Immanent in Nervous Activity, *The Bulletin of Mathematical Biology*, 1943, par Warren S.McCulloch et Walter Pitts
- [2] The Perceptron : a Probabilistic Model for Information Storage and Organization in The Brain, *Psychological Review*, 1958, par Frank Rosenblatt
- [3] Perceptrons, *M.I.T. Press*, 1969, par Marvin Minsky et Seymour Papert
- [4] Learning Representations by Back-Propagating Errors, *Nature*, 1986, par David E.Rumelhart, Geoffrey E.Hinton et Ronald J.Williams
- [5] Single Unit Activity in Striate Cortex of Unrestrained Cats, *The Journal of Physiologie*, 1959, par David H.Hubel et Torsten N.Wiesel
- [6] Receptive Fields of Single Neurons in the Cat's Striate Cortex, *The Journal of Physiologie*, 1959, par David H.Hubel et Torsten N.Wiesel
- [7] Neocognitron : A Self-Organizing Neural network Model for a Mechanism of Pattern Recognition Unaffected by Shift in position, *Biological Cybernetics*, 1980, Kunihiko Fukushima
- [8] <http://image-net.org/>, Compétition d'algorithme de classification d'image.
- [9] Gradient-Based Learning Applied to Document Recognition, *Proceedings of the IEEE*, 1998, Yann LeCun *et al.*
- [10] ImageNet Classification with Deep Convolutional Neural Networks, *Proceedings of the 25th International Conference on Neural Information Processing Systems*, 2012, Alex Krizhevsky *et al.*
- [11] Going Deeper with Convolutions, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, Christian Szegedy *et al.*
- [12] Deep Residual Learning for Image Recognition, 2015, Kaiming He *et al.*
- [13] Deep Learning avec Keras et TensorFlow 2e édition, par Aurélien Géron, 2020.
- [14] <http://yann.lecun.com/exdb/mnist/>, base de donnée de chiffre écrits à la main, Yann LeCun *et al.*

Annexe

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow import keras
import tensorflow as tf

#importation des données à entrainer et des données pour tester le système : x représentent les photos et y le correspondant à
#chaque photo de x
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

#permet de vérifier si les dimensions des entrées importées sont bonnes ou pas
assert x_train.shape == (50000, 32, 32, 3)
assert x_test.shape == (10000, 32, 32, 3)
assert y_train.shape == (50000, 1)
assert y_test.shape == (10000, 1)

#normalisation des données photos dans x_train et x_test
x_train = x_train/255.0
x_test = x_test/255.0

#création de noms pour chaque classes
class_names = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
```

```
#affichage des 10 premières images que contienne x_train et les noms correspondant dans y_train
plt.figure(figsize=(15,15))
for i in range(10) :
    ax = plt.subplot(1, 10, i + 1)
    plt.imshow(x_train[i])
    plt.title(class_names[y_train[i][0]])
    plt.axis("off")
```



```
#création de l'entraînement du système
model = keras.models.Sequential([
    keras.layers.Conv2D(32, 3, activation="relu", padding="same", input_shape=(32, 32, 3)),
    keras.layers.Conv2D(32, 3, activation="relu", padding="same",),
    keras.layers.MaxPooling2D(2, padding="valid"),
    keras.layers.Dropout(0.4),

    keras.layers.Conv2D(64, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(64, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2, padding="valid"),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2, padding="valid"),

    keras.layers.Dropout(0.3),
    keras.layers.Flatten(),

    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.4),
    keras.layers.Dense(10, activation="softmax")
])
```

```
#affichage des paramètres liés à chaque couche de l'entraînement défini  
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 32, 32, 32)	896
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
)		
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1290
=====		
Total params:	550,570	
Trainable params:	550,570	
Non-trainable params:	0	

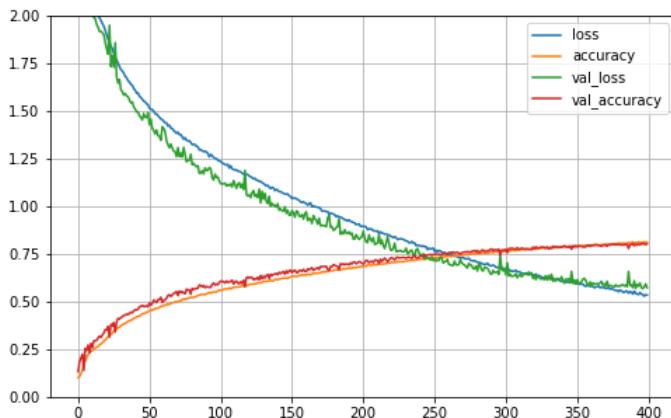
```
#compilation de l'entraînement avec la fonction de coût (Loss), le choix de la descente de gradient : sgd (optimizer) qui est  
#la descente de gradient stochastique et la mesure souhaitée (metrics) qui est la précision  
model.compile(loss="sparse_categorical_crossentropy", optimizer=keras.optimizers.SGD(learning_rate = 0.03), metrics=["accuracy"])
```

```

#callbacks nous permet de sauvegarder le meilleur modèle lors de l'entraînement
callbacks = [
    keras.callbacks.ModelCheckpoint("best_model2.keras",
                                    save_best_only = True),
]
#lancement de l'entraînement sur x_train et y_train 400 fois (epochs) donc 300 modèles par mini-lots de 1024 images (batch-size)
#puis vérification de la précision de chaque modèle avec x_test et y_test. verbose montre juste une animation lors de
#l'entraînement
history = model.fit(x_train, y_train, epochs=400, batch_size=1024, validation_data=(x_test,y_test), verbose=1, callbacks = callbacks)
Epoch 395/400
49/49 [=====] - 2s 44ms/step - loss: 0.5473 - accuracy: 0.8082 - val_loss: 0.5967 - val_accuracy: 0.7967
Epoch 396/400
49/49 [=====] - 2s 44ms/step - loss: 0.5383 - accuracy: 0.8111 - val_loss: 0.5732 - val_accuracy: 0.8036
Epoch 397/400
49/49 [=====] - 2s 44ms/step - loss: 0.5371 - accuracy: 0.8097 - val_loss: 0.5674 - val_accuracy: 0.8017
Epoch 398/400
49/49 [=====] - 2s 44ms/step - loss: 0.5277 - accuracy: 0.8135 - val_loss: 0.5766 - val_accuracy: 0.8029
Epoch 399/400
49/49 [=====] - 2s 44ms/step - loss: 0.5348 - accuracy: 0.8107 - val_loss: 0.5925 - val_accuracy: 0.7985
Epoch 400/400
49/49 [=====] - 2s 44ms/step - loss: 0.5344 - accuracy: 0.8116 - val_loss: 0.5726 - val_accuracy: 0.8021

#on affiche les différentes courbes obtenue avec loss et accuracy qui correspondent à la perte et la précision de chaque modèle
#(les 400 en abscisses) sur le jeu d'entraînement et val_loss et val_accuracy, ceux correspondant au jeu de données x_test et
#y_test
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 2)
plt.show()

```



```

#ici on regarde la perte et la précision du meilleur modèle obtenu sur le jeu de donnée x_test et y_test
best_model = keras.models.load_model("best_model2.keras")
best_model.evaluate(x_test, y_test)

```

```

313/313 [=====] - 1s 3ms/step - loss: 0.5674 - accuracy: 0.8017
[0.5674042701721191, 0.8016999959945679]

```