Stage de Fin d'Études
Création d'un Moteur de Recherche dans les Images Satellite

Guillaume Rochette

28 septembre 2017

Table des matières

Chapitre 1

Introduction

1.1 Présentation de l'entreprise

1.2 Problématique(s)

1.3 Objectifs du stage

Chapitre 2

L'Imagerie Satellite

2.1 Introduction

2.1.1 Histoire

2.1.2 Applications

2.2 Géométrie des images

2.2.1 Référentiels

2.2.2 Acquisition de l'image

2.2.3 Corrections géométriques

2.3 Radiométrie des images

2.3.1 Principe d'acquisition de l'image

2.3.2 Corrections radiométriques

2.4 Résolution des images

2.4.1 Échantillonnage

2.4.2 Interpolation d'images

2.4.3 Amélioration de la résolution

Chapitre 3

L'Apprentissage Machine

L'apprentissage machine permet d'accomplir des tâches traditionnellement compliquées pour les ordinateurs, mais vues comme simple pour l'humain.

En effet, l'humain apprend par exemple dès le plus jeune âge à discerner les contours sur une image, d'en décrire le contenu et même de visualiser un objet sous différents angles. Des algorithmes classiques ont du mal à produire des bons résultats sur ce genre de problèmes.

À l'inverse, les ordinateurs possèdent une capacité très supérieure à l'humain pour résoudre des calculs complexes, ou des problèmes d'ordre combinatoire.

D'un point de vue scientifique et philosophique, l'apprentissage machine est très intéressant et soulève beaucoup de questions, car l'étude de l'apprentissage appliqué aux machines nous permettrait peut-être d'entrevoir certains principes définissant l'intelligence.

3.1 Définition d'un Algorithme Apprenant

Un algorithme apprenant est un algorithme capable d'utiliser des données pour accomplir des tâches. La définition la plus célèbre, proposée par T.M Mitchell, est la suivante :

> On dit qu'un algorithme apprend grâce à une expérience E, par rapport à une classe de tâches à accomplir T, dont on peut calculer la mesure de performance (ou d'accomplissement) P, si sa capacité d'accomplir la tâche T, mesurée par la performance P, s'améliore avec l'expérience E.

Un tel algorithme basera son apprentissage sur un jeu de données. Un jeu de données est un ensemble d'exemples.

Chaque exemple, noté e, est composé d'une entrée, notée x, à laquelle on peut associer, dans le cas d'un apprentissage supervisé, une sortie attendue, notée $\hat{y}$.

3.1.1 La Tâche

Le processus d'apprentissage ne représente pas la tâche. L'apprentissage symbolise les moyens d'acquérir la possibilité, qui peut-être aussi vue comme la capacité, d'accomplir une tâche en particulier.

Par exemple, si l'on veut apprendre à un robot à lire, la tâche en question sera la capacité à lire.

En apprentissage machine, une tâche, notée T, consiste à faire, pour chaque exemple e, correspondre une entrée x à un résultat de sortie y.

Une entrée x est au sens statistique un individu décrit par un ensemble de variables.

Un individu est représenté par un vecteur $x \in \mathbb{R}^n$, avec n étant le nombre de variables décrivant l'individu et x_i la i-ème variable.

Par exemple, une image est vue comme une matrice $I \in \mathbb{R}^{m \times n}$ de pixels, que nous pouvons projeter sur un vecteur $x \in \mathbb{R}^k$, avec $k = m \times n$.

Voici à présent un aperçu non exhaustif des tâches.

Classification

La classification consiste à déterminer, pour une entrée $x \in \mathbb{R}^n$, en sortie, une ou plusieurs parmi k catégories, ou classes, associées à l'entrée.

Pour résoudre ce type de problème, l'algorithme apprenant doit produire une fonction $f : \mathbb{R}^n \in E$, avec E étant un ensemble de dimension k.

On notera que la structure de E n'est pas fixée.

On peut par exemple avoir :

— $E = \{1, .., k\}$, dans ce cas, la classification est dite simple, car pour tout individu x donné, il ne peut correspondre qu'une seule classe.
— $E = \{0, 1\}^k$, dans ce cas la classification est multiple, car il peut correspondre 0, 1 ou plusieurs classes.
— $E = \{y \in [0, 1]^k / \sum_{i=1}^{k} y_i = 1\}$, dans ce cas, la sortie y représente une distribution de probabilité.

La classification est par exemple utilisée pour la reconnaissance d'objets, l'entrée x étant l'image, et la sortie y, la classe de l'objet dans l'image.

Régression

La régression consiste à prédire une valeur réelle en fonction d'un individu en entrée.

L'algorithme doit par conséquent se comporter comme une fonction $f : \mathbb{R}^n \to \mathbb{R}$, en faisant correspond à un individu donné $x \in \mathbb{R}^n$, une prédiction $y \in \mathbb{R}$.

On l'utilise par exemple pour l'approximation de fonctions modélisant un phénomène ou encore la prédiction des cours des actions.

Autres applications

Les deux applications citées ci-dessus sont les plus connues et les plus largement utilisées, toutefois il en existe un grand nombre telles que :

— La transcription, consistant à traduire des données sans structure particulière en données ayant une structure discrète. Par exemple, extraire le texte d'une image, ou la reconnaissance vocale.
— La traduction, consistant à traduire une suite de caractères d'une langue à une autre.
— Le débruitage, ou *denoising* : Il s'agit de reconstituer à partir d'un exemple bruité $\tilde{x} \in \mathbb{R}^n$, l'original $x \in \mathbb{R}^n$, en prédisant la probabilité $p(x|\tilde{x})$, c'est à dire que x soit l'original de $\tilde{x}$.

3.1.2 La Mesure de la Performance

Afin de quantifier les performances d'un algorithme d'apprentissage machine, il nous faut définir une mesure. Cette mesure, notée P, est généralement spécifique à la tâche T donnée à l'algorithme.

Pour des tâches, telles que la classification, la transcription ou la traduction, on mesure généralement la précision, c'est à dire la proportion d'exemples ou la sortie proposée par le modèle est similaire à la sortie attendue.

Néanmoins, cette mesure n'est pas générale, en effet, par exemple pour une tâche de régression, si l'écart entre la sortie produite et la sortie attendue est faible mais non nul, alors faut-il considérer le modèle comme valide ?

Pour cela, nous définissons une mesure plus générale, une *fonction de perte* associée. Cette fonction de perte n'a pas de forme particulière car elle dépend de la tâche T à réaliser, mais elle décroît à mesure que la sortie y produite par l'algorithme est "bonne".

Afin de réaliser un bon apprentissage, il nous donc minimiser cette *fonction de perte*.

L'apprentissage machine se résume donc à optimiser notre algorithme pour modéliser au mieux une situation.

3.1.3 L'Expérience

On peut distinguer deux grandes classes d'algorithmes d'apprentissage machines, les algorithmes d'apprentissage *non-supervisés* et *supervisés*.

Une expérience E peut être comprise comme le fait d'apprendre sur un *jeu de données*.

Apprentissage non supervisé

Les algorithmes d'apprentissage machine non-supervisés ont pour but d'apprendre sur des jeux de données ne contenant que des entrées x, et par exemple apprendre la distribution de probabilité $p(x)$ du jeu de données, ou bien de répartir les données dans des clusters, par le biais d'une distance donnée.

Apprentissage supervisé

Les algorithmes d'apprentissage machine supervisés ont pour but d'apprendre sur un jeu de données, à associer une entrée x à une sortie y, que l'on veut proche de la sortie attendue $\hat{y}$. Ce qui peut être vu comme l'apprentissage de la probabilité de $p(y|x)$.

3.2 Contraintes d'apprentissage et régularisation

3.2.1 Sous-apprentissage, Sur-apprentissage et Capacité d'un algorithme apprenant

L'un des défi, et intérêt, majeur en apprentissage machine est la possibilité de bien raisonner sur de nouveaux exemples encore inconnus de l'algorithme. Cette capacité de raisonnement s'appelle la généralisation. En effet, un algorithme apprenant s'appuie pour l'entraînement sur un ensemble d'apprentissage, ce qui nous permet de mesurer l'*erreur d'apprentissage*, terme équivalent à la *fonction de perte* sur les données d'apprentissage, et ainsi de la minimiser.

Ce qui différencie un simple problème d'optimisation de l'apprentissage machine, est que nous souhaitons non seulement que notre algorithme ait une *erreur d'apprentissage* faible, mais que sa capacité à généraliser, traduite par l'*erreur de généralisation* ou *erreur de test*, soit aussi petite que possible.

Ainsi, pour juger de l'efficacité d'un algorithme apprenant, nous devons prendre en compte l'*erreur d'apprentissage*, mais aussi que la différence entre l'*erreur d'apprentissage* et l'*erreur de test*.

Sous-apprentissage

Le phénomène de sous-apprentissage survient lorsque pour un problème donné et un modèle choisi, l'*erreur d'apprentissage* n'est pas suffisamment faible pour obtenir de bons résultats.

On peut donc interpréter le sous-apprentissage comme étant l'incapacité d'un modèle à ajuster un phénomène.

Sur-apprentissage

Le phénomène de sur-apprentissage survient lorsque l'écart entre l'*erreur d'apprentissage* et l'*erreur de test* est trop grand. Cela signifie que notre modèle, proposé par l'algorithme d'apprentissage, a été capable d'apprendre suffisamment sur les exemples d'entraînement, mais n'est pas capable de généraliser sur des exemples de test, sur lesquels il n'a pas appris.

Capacité

On peut alors définir la *capacité* d'un modèle comme étant la mesure théorique de ses performances d'apprentissage mais aussi de ses performances en terme de généralisation.

Ainsi le modèle le plus adapté à notre problème aura une capacité suffisante pour apprendre correctement la fonction à produire par rapport aux données, mais ne sera pas trop grande, afin de garder un pouvoir suffisant de généralisation.

Exemple

Nous allons maintenant introduire un exemple nous permettant d'illustrer ces concepts.

La tâche T est une régression polynomiale, c'est-à-dire que nous devons trouver un polynôme $P \in \mathbb{R}^q$ ajustant au mieux nos données.

P est donc de la forme :

$$P(z) = b + \sum_{j=1}^{q} \theta_j z^j$$

L'expérience E est un ensemble de n points (x_i, y_i), que l'on peut résumer en deux vecteurs :

$$x, y \in \mathbb{R}^n$$

Et la mesure de performance P, le critère des moindres carrés, c'est-à-dire :

$$J_0(x, y) = \frac{1}{2} \sum_{i=1}^{n} (P(x_i) - y_i)^2$$

Si l'on considère que $P(x) = (P(x_i))_{i=1..q}$, alors on peut réécrire $J_0(x, y)$ tel que :

$$J_0(x, y) = \frac{1}{2}(P(x) - y)^T (P(x) - y)$$

Voici le nuage de 10 points à ajuster :

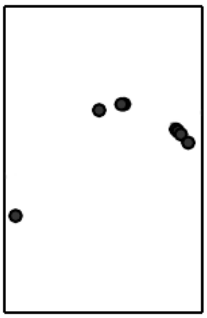

FIGURE 3.1 – Nuage de 10 points à ajuster

Si nous choisissons un polynôme $P \in \mathbb{R}^1$, c'est à dire de la forme $P(z) = b + \theta z$, en minimisant le *problème des moindres carrés*, nous obtenons la courbe suivante.

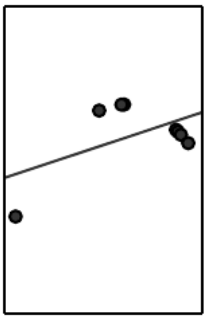

FIGURE 3.2 – Cas de sous-apprentissage

Sans même faire d'études statistiques, on peut voir que le modèle ajuste mal les données.

Si nous choisissons par contre un polynôme $P \in \mathbb{R}^9$, en minimisant le *problème des moindres carrés*, nous obtenons la courbe suivante.

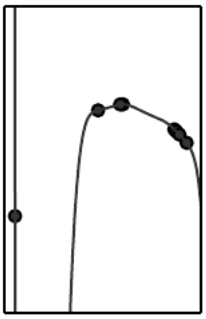

FIGURE 3.3 – Cas de sur-apprentissage

On peut s'apercevoir que même si la droite passe exactement par nos 10 points, le modèle ne semblera pas ajuster d'éventuelles nouvelles données.

Enfin, si l'on choisit un polynôme $P \in \mathbb{R}^2$, nous ajustons tous les points du jeu de données et le modèle semble suffisamment simple pour généraliser de nouvelles données.

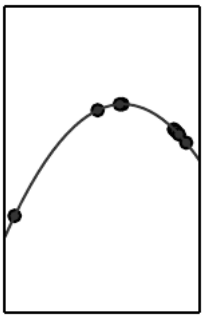

FIGURE 3.4 – Capacité du modèle adaptée

3.2.2 Régularisation

Dans l'exemple précédent, nous avons vu que le choix du modèle influait sur sa capacité, c'est à dire de tenter de produire une fonction suffisamment générale par rapport au phénomène représenté par les données.

Afin d'améliorer les capacités de généralisation des algorithmes d'apprentissage machine, un principe, dont les premières formulations sont attribuées à Ptolémée, II^e siècle, connu dans la littérature, sous le nom du Rasoir d'Occam (ou Ockham) est le suivant :

> Les hypothèses suffisantes les plus simples sont les plus vraisemblables.

Ce principe de parcimonie, ou de simplicité, consiste à ne pas utiliser d'hypothèses spécifiques à un problème, si il existe des hypothèses générales plus simples répondant à ce même problème.

Ainsi pour résoudre notre problème, notre modèle doit être suffisamment performant sur un problème spécifique, mais doit être assez général pour produire des résultats corrects si l'on change quelques hypothèses.

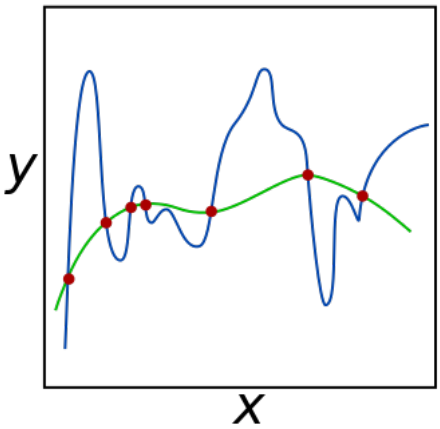

FIGURE 3.5 – Dilemme du choix entre deux modèles

Dans cet exemple, nous choisirons bien entendu la courbe verte, car le modèle vert est beaucoup plus simple et nous semble plus cohérent par rapport à nos données.

Cette recherche d'un modèle optimal mais suffisamment simple peut se faire en ajoutant une composante à la mesure de performance.

En reprenant l'exemple précédent, nous pouvons ajouter à $J_0(x, y)$ une fonction f arbitraire, qui aurait tendance à pénaliser l'effet de sur-apprentissage.

Cette fonction f peut dépendre par exemple des coefficients θ_i du polynôme.

Ce qui nous donne une fonction :

$$J(x, y, \theta) = J_0(x, y) + \lambda f(\theta)$$

On peut choisir par exemple $f(\theta) = \sum_{i=1}^{q} \theta_i^i$ et ainsi pénaliser les coefficients d'ordre élevé, ainsi notre algorithme aura tendance à proposer des polynômes ayant des coefficients d'ordre élevé de petite taille, et ainsi réduire artificiellement l'ordre du polynôme choisi.

Le terme de *régularisation* est employé pour tout modification apportée à l'algorithme apprenant visant à réduire l'*erreur de généralisation*, mais pas l'*erreur d'apprentissage*.

3.3 Paramétrisation et Validation d'un Algorithme

La plupart des algorithmes d'apprentissage machine sont paramétrés afin de contrôler plus précisément leur apprentissage.

Ces paramètres s'appellent des *hyper-paramètres*.

Par exemple dans l'exemple de la régression polynomiale, nous avons utilisé deux *hyper-paramètres* :

— q : le degré du polynôme P utilisé.

— λ : le coefficient de décroissance appliquée à la fonction $f(\theta)$.

En effet, ces deux paramètres doivent être manuellement choisis par rapport à notre problème, en fonction de notre besoin de généralisation (λ) ou de performance sur l'apprentissage (q).

Afin de vérifier que notre algorithme soit correctement paramétré, il nous faut construire un troisième jeu de données, appelé ensemble de validation.

Voici les propriétés de ces jeux de données.

— Le premier jeu de données est l'ensemble d'apprentissage, notre algorithme apprendra donc uniquement sur cet ensemble.

— Le second jeu de données est l'ensemble de test, il est indépendant du premier. Nous pourrons vérifier grâce à ce jeu de données si l'algorithme est apte à généraliser.

— Le troisième jeu de données est l'ensemble de validation. En règle générale, on soustrait au jeu d'apprentissage une petite fraction de ses données. Grâce à lui, nous pouvons vérifier l'hyper-paramétrisation de notre algorithme.

Ainsi la procédure d'expérimentation est la suivante :

1. Choix d'un modèle.
2. Choix des hyper-paramètres.
3. Apprentissage sur l'ensemble d'apprentissage.
4. Estimation de la capacité de généralisation du modèle sur l'ensemble de validation.
5. Si la capacité à généraliser est trop faible, c'est à dire que l'*erreur de validation* est trop élevée, on modifie les hyper-paramètres, puis retour à l'étape d'apprentissage, sinon on passe à l'étape suivante.
6. Évaluation finale des performances du modèle sur l'ensemble de test.
7. Si l'*erreur de test* est trop élevée, alors il retourner à la première étape et choisir un nouveau modèle, sinon on valide le modèle.

3.4 Optimisation

Dans cette section, nous aborderons les grands principes utilisés pour réaliser l'apprentissage à proprement parler.

Nous rappelons qu'en effet le but d'un algorithme d'apprentissage machine est s'améliorer à réaliser une tâche T grâce à une expérience E, cette amélioration étant mesurée par P.

Nous voulons donc ici, minimiser l'*erreur d'apprentissage* afin d'améliorer nos performances sur la tâche à réaliser.

3.4.1 Méthodes exactes

Les méthodes exactes sont très pratiques, car elles permettent de trouver la solution de façon immédiate.

Pour résoudre le problème $min_\theta J(x, y)$, il nous faut donc annuler le gradient de la fonction de perte, soit

$$\nabla_\theta J(x, y) = 0$$

Cela nécessite un certain travail en amont, dépendant entièrement du type d'algorithme utilisé. De plus il n'existe **pas toujours de solution exacte unique**.

Exemple

Prenons le cas de la régression polynomiale, le problème est le suivant :

$$min_{\theta,b} J(x, y) = \frac{1}{2}(P(x) - y)^T(P(x) - y)$$
$$\leftrightarrow \text{Trouver } \theta \in \mathbb{R}^q \text{ et } b \in \mathbb{R}, \text{ tels que la fonction de perte } J(x, y) \text{ soit minimale.}$$

Pour rappel,

$$P(z) = b + \sum_{j=1}^{q} \theta_j z^j,$$
un polynôme de degré q,
et $P(x) = (P(x_i))_{i=1..n}$,
le polynôme d'un vecteur $P(x)$ est le vecteur de polynômes $P(x_i)$.

$$\text{Donc } P(x) = \begin{pmatrix} 1 & x_1 & \cdots & x_1^q \\ \vdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ 1 & x_n & \cdots & x_n^q \end{pmatrix} \begin{pmatrix} b \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} = X\theta$$

On peut alors réécrire $J(x, y)$ de la façon suivante :

$$J(x,y) = \frac{1}{2}(P(x) - y)^T(P(x) - y) = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$
$$\rightarrow J(x,y) = \frac{1}{2}(\theta^T X^T X\theta - 2\theta^T X^T y + y^T y)$$

Il nous faut maintenant trouver le minimum de $J(x, y)$, ce qui revient à résoudre $\nabla_\theta J(x, y) = 0$

Or, grâce aux dérivées vectorielles ($\frac{\partial x^T a}{\partial x} = a$, et $\frac{\partial x^T Ax}{\partial x} = 2Ax$)

$$\nabla_\theta J(x,y) = \frac{1}{2}(2X^T X\theta - 2X^T y) = X^T X\theta - X^T y$$

Donc,

$$X^T X\theta - X^T y = 0$$
$$\rightarrow X^T X\theta = X^T y$$
$$\rightarrow \theta = (X^T X)^{-1} X^T y$$

Néanmoins une méthode exacte présente des inconvénients. Par exemple, pour calculer θ, la complexité algorithmique est de l'ordre de $O(n^3)$, avec n le nombre de données, à cause de l'inversion de matrice.

Cette complexité cubique pose donc des problèmes quand n est très grand.

De plus cette méthode présente des problèmes de stabilité numérique, si le conditionnement de $X^T X$ est grand.

3.4.2 Méthodes itératives

Nous avons vu dans un premier temps, que les méthodes exactes permettaient parfois avec, un peu de calcul formel, d'avoir une solution optimale à notre problème, et ce immédiatement.

Une méthode itérative, comme son nom l'indique, converge vers le résultat après plusieurs itérations.

Elles peuvent remplacer les méthodes exactes, quand celles-ci sont soit inapplicables, coûteuses ou inconnues. De plus lorsqu'un problème est mal conditionné, elles limitent la propagation des erreurs.

Il existe un grand nombre de méthodes itératives, mais nous n'en aborderons qu'une seule catégorie, car utile pour la suite, les méthodes de descente du gradient. Les méthodes de descente de gradient, sont très utilisées en apprentissage machine, de par leur facilité d'implémentation, ainsi que de leur convergence plutôt rapide.

Algorithme général de Descente de Gradient

Voici à présent l'algorithme général du gradient, appliqué à une fonction $J(x, y, \theta)$.

Algorithm 1 Algorithme général de Descente de Gradient

Require: L'ensemble des entrées $x = (x_i)_{i=1}^n$ et l'ensemble des sorties $y = (y_i)_{i=1}^n$
x_i et y_i sont des vecteurs réels de dimensions quelconques,
θ_0, les paramètres initiaux du modèle,
et $\epsilon > 0$, seuil de tolérance.
repeat
Calcul de $\nabla_\theta J(x, y, \theta_k)$.
Calcul de α_k. $\{\alpha_k$ peut être soit une constante, soit calculé en fonction de gradients $\nabla_\theta J(x, y, \theta)\}$
$\theta_{k+1} = \theta_k - \alpha_k \nabla_\theta J(x, y, \theta_k)$.
until $\nabla_\theta J(x, y, \theta_k) \leq \epsilon$
return θ_{k+1}

Dans notre cas, nous souhaitons minimiser la fonction de perte de associée à notre modèle par rapport aux données d'apprentissage.

Soit x et $y \in \mathbb{R}^n$, les entrées et les sorties.

On pose ici $J(x, y, \theta)$ l'erreur moyenne du modèle par rapport aux données d'apprentissage.

$$J(x, y, \theta) = \frac{1}{n} \sum_{i=1}^{n} J(x_i, y_i, \theta)$$

Donc,

$$\nabla_\theta J(x, y, \theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta J(x_i, y_i, \theta)$$

Méthode de Descente de Gradient Stochastique

La réflexion derrière la descente de gradient stochastique est que le gradient de l'ensemble est la moyenne des gradients des couples (x, y) .

D'après la Loi des Grands Nombres,

$$\mathbb{E}(X) \approx \frac{1}{n} \sum_{i=1}^{n} x_i$$

Avec X une distribution, dont les données x_i sont issues.

Par ailleurs, si l'on échantillonne de façon uniforme m individus parmi les n données d'apprentissage, on aura :

$$\mathbb{E}(X) \approx \frac{1}{m} \sum_{i=1}^{m} x_i$$

Ainsi,

$$\frac{1}{n} \sum_{i=1}^{n} x_i \approx \frac{1}{m} \sum_{i=1}^{m} x_i$$

L'ensemble des m variables extraites de x s'appelle *mini-lot* ou *mini-batch*

Algorithm 2 Algorithme de Descente de Gradient Stochastique

Require: L'ensemble des entrées $x = (x_i)_{i=1}^n$ et l'ensemble des sorties $y = (y_i)_{i=1}^n$
x_i et y_i sont des vecteurs réels de dimensions quelconques,
$m < n$,
θ_0, les paramètres initiaux du modèle, initialisés aléatoirement,
$\alpha > 0$, le pas de descente
et $\epsilon > 0$, seuil de tolérance.
repeat
  $\hat{x}, \hat{y} = sample(x, y, m)$
  Calcul de $\nabla_\theta J(\hat{x}, \hat{y}, \theta_k)$.
  Calcul de α_k. $\{\alpha_k$ peut être soit une constante, soit calculé en fonction de gradients $\nabla_\theta J(x, y, \theta)\}$
  $\theta_{k+1} = \theta_k - \alpha_k \nabla_\theta J(\hat{x}, \hat{y}, \theta_k)$.
until $\nabla_\theta J(\hat{x}, \hat{y}, \theta_k) \leq \epsilon$
return θ_{k+1}

L'intérêt de cette méthode repose principalement sur le fait que le gradient moyen du *mini-batch* est une très bonne approximation du gradient moyen calculé sur l'ensemble des données, et nous permet ainsi d'ajuster les paramètres du modèle plus souvent, permettant généralement une convergence plus rapide.

Chapitre 4

L'Apprentissage Profond

L'apprentissage profond se différencie de l'apprentissage machine classique particulièrement sur deux points :

— Il existe des dizaines, voire des centaines de classes d'algorithmes d'apprentissage pour résoudre des tâches variées, et par extension des problèmes très variés, cependant dans le cas de l'apprentissage profond, même si il existe des différences, le principe utilisé reste globalement le même.
De ce fait leur capacité de généralisation est beaucoup plus forte, permettant ainsi de résoudre plus efficacement certains problèmes jugés difficiles à résoudre par des algorithmes d'apprentissage classiques.

— Si le principe de l'algorithme reste globalement le même, la structure du modèle est quant à elle très variable, et il est même très important de la faire varier d'un problème à un autre, car elle est intrinsèquement dépendante du problème et de ses données.

Nous présenterons dans un premier temps le fonctionnement des réseaux totalement connectés, car les plus simples. Ainsi que différentes méthodes employées pour améliorer les performances des modèles.

Puis nous verrons ensuite une variante, les réseaux convolutifs.

Enfin nous aborderons succinctement différentes applications existantes appliquées à la vision.

4.1 Réseaux de Neurones Profonds

Les réseaux de neurones profonds, aussi appelés *Deep Feedforward Networks* définissent un *mapping* entre un vecteur d'entrée x et sa sortie associée y de sorte que $y = f(x, \theta)$ et apprennent les paramètres θ de sorte à produire la meilleure estimation.

Ils sont dits "*feedforward*", ou en français à "propagation unique", car l'information se déplace à travers la fonction, depuis l'entrée x, puis par des étapes intermédiaires, pour arriver à la sortie y, sans aucun retour en arrière.

Les réseaux de neurones permettant un *feedback* ou *retour d'expérience* sont appelés font partie de la classe des réseaux récurrents.

Les réseaux de types "*feedforward*" sont de loin les réseaux de neurones les plus utilisés.

On utilise le mot *réseau* dans l'appellation réseaux de neurones, car ils sont composés d'une multitude de fonctions *simples*, organisées en réseau.

On pourrait par exemple imaginer notre fonction $f(x, \theta)$ comme étant la composée de plusieurs fonctions, telle que $f(x, \theta) = u_{\theta_1} \circ v_{\theta_2} \circ w_{\theta_3}(x)$ avec $\theta = (\theta_1, \theta_2, \theta_3))$, un ensemble de paramètres.

C'est par ailleurs cette architecture distribuée, qui grâce à une puissance de calcul grandissante, par le biais du calcul GPU, permet de résoudre des problèmes de plus en plus complexes.

4.1.1 Définition d'un Neurone Artificiel

Ce que l'on appelle *neurone* est effectivement inspiré du fonctionnement d'un neurone biologique, bien que différent à bien des égards.

Un neurone artificiel est une application $f : \mathbb{R}^p \to \mathbb{R}$, paramétrée par $\theta = (b, w) = (b, w_1, ..., w_p)$.

Le réel b s'appelle le biais, et le vecteur $w = (w_1, ..., w_p)$ s'appelle vecteur de poids.

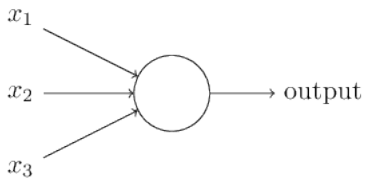

FIGURE 4.1 – Schéma de représentation d'un Neurone

Cette application est la composée de deux fonctions distinctes :

— Une fonction d'*agrégation* $h : \mathbb{R}^p \to \mathbb{R}$, combinant l'entrée x avec les poids w et le biais b.
Dans le cas d'un neurone *totalement connecté* (le plus basique), elle à l'allure suivante :

$$h_\theta(x) = h_{b,w}(x) = w^T x + b = \sum_{i=1}^{p} w_i x_i + b$$

— Une fonction d'*activation* $g : \mathbb{R} \to \mathbb{R}$, *non-linéaire*.
Des exemples :
  — *ReLU*, pour *Rectified Linear Unit* : $g(z) = max(0, z)$
  — *Sigmoïde* : $g(z) = \frac{1}{1+e^{-z}}$
  — *Tangente hyperbolique* : $g(z) = \tanh(z)$

Il est important de garder à l'esprit que ces deux opérations successives sont distinctes, mais qu'elles sont regroupées ici pour montrer les similarités avec les neurones biologiques.

Ainsi notre neurone peut être résumé par une fonction f telle que :

$$f_\theta(x) = g \circ h_\theta(x) = g(w^T x + b)$$

La sortie de notre neurone, par la suite notée a, est l'*image* de la fonction f par l'entrée x :

$$a = f_\theta(x) = g \circ h_\theta(x) = g(w^T x + b)$$

4.1.2 Définition d'un Réseau de Neurones

Avec la définition précédente d'un neurone, nous pouvons alors schématiser un réseau de neurones de la façon suivante :

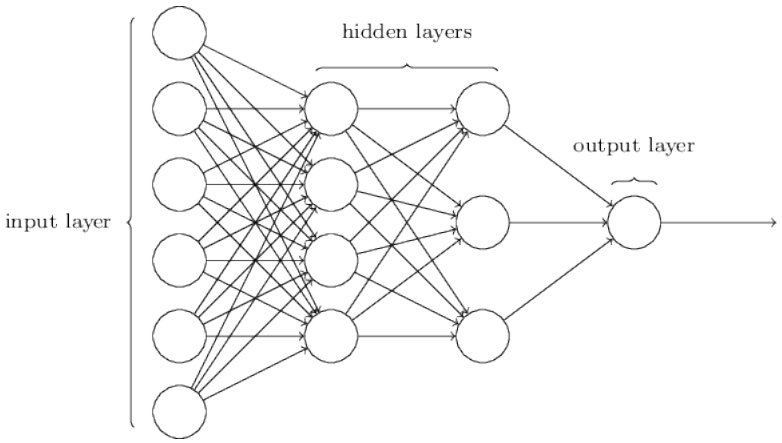

FIGURE 4.2 – Schéma pédagogique de représentation d'un Réseau de Neurones

Ici, chaque cercle représente un neurone, équivalent à ceux définis précédemment.

Toutefois, sur la première couche, appelée *couche d'entrée*, les neurones produisent en sortie, le *vecteur d'entrée* x, ici $x \in \mathbb{R}^6$.

On peut noter aussi que la dernière couche, appelée *couche de sortie*, les neurones produisent le vecteur y, ici $y \in \mathbb{R}^1 = \mathbb{R}$.

Attention, la description faite ici, n'est qu'à but pédagogique. Dans le cas courant, on ne considère pas un réseau en individualisant chaque neurone, mais plutôt couche par couche :

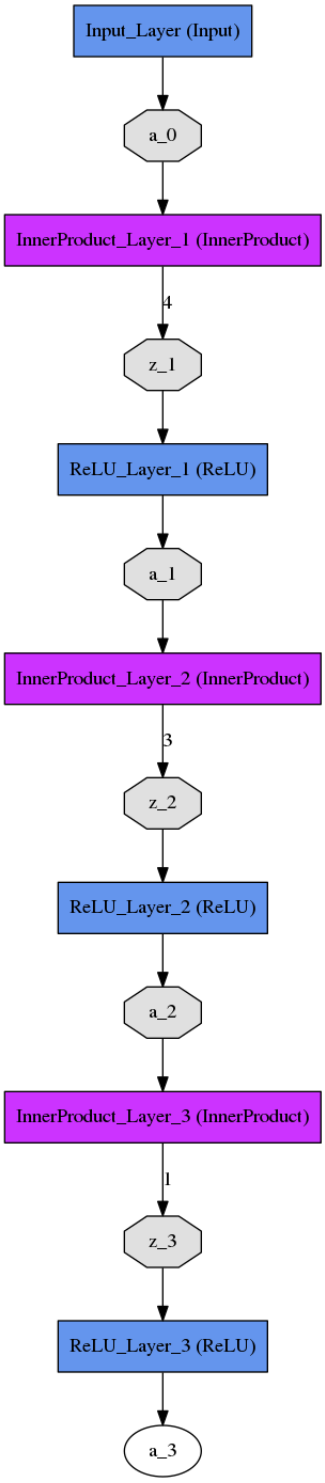

FIGURE 4.3 – Schéma courant de représentation d'un Réseau de Neurones

Ainsi, pour décrire un réseau de neurones, nous mesurerons deux caractéristiques :

Profondeur

Le nombre de couches d'un réseau de neurones est appelé *profondeur*, plus ce nombre est élevé, plus la capacité théorique de généralisation est forte.

Même si il n'existe pas de seuil officiel différenciant les réseaux de neurones, *peu profonds*, dits *superficiels*, des réseaux de neurones *profonds*, on parle d'*apprentissage profond* lorsque le nombre de couches cachées est ≥ 2.

On parlera d'apprentissage très profond, lorsque le nombre de couches cachées est ≥ 10.

Largeur

Le nombre de neurones composant une couche est appelé *largeur*.

Nous pouvons par ailleurs noter que le nombre de paramètres $\theta = (b, w) = (b, w_1, .., w_p)$ d'un neurone n'est pas arbitraire.

En effet, la *largeur* de chaque couche est égale au nombre de paramètres θ des neurones de la couche suivante.

Par ailleurs, la *largeur* de la première couche est égale à la taille du vecteur d'entrée x.

Expression générale

On explicitera généralement l'architecture d'un réseau de neurones par un vecteur $s \in \mathbb{N}^L$.

Ainsi $S = (S_1, S_2, ..., S_L)$ représente un réseau de neurones de *profondeur* L (L couches), chacune possédant s_i neurones ayant s_{i-1} poids et un biais par neurone.

4.1.3 Notations

Afin de ne pas se perdre dans les calculs qui vont survenir, nous allons définir une notation nous permettant de repérer sans ambiguïté la position des différents éléments d'un réseau de neurones.

Important !

Chaque élément du réseau de neurones peut être défini par sa position dans le réseau.

Pour les *poids* :

— Un poids *réel* noté ${}^kW_i^j$ sera le i-ème poids du j-ème neurone, situé dans la k-ème couche.

— Par conséquent ${}^kW^j$ sera quant à lui le vecteur de poids du j-ème neurone, situé dans la k-ème couche.

— Enfin ${}^kW$ sera la matrice des poids de la k-ème couche.

Pour les *biais* :

— Un biais *réel* noté ${}^kb^j$ sera le biais du j-ème neurone, situé dans la k-ème couche.

— Par conséquent ${}^kb$ sera quant à lui le vecteur de biais de la k-ème couche.

Nous noterons par ailleurs avec la lettre a, qui sera indicée par la suite, la sortie d'une couche du réseau.

Pour les *sorties* :

— Une sortie d'un neurone noté ${}^ka^j$ sera la sortie du j-ème neurone, situé dans la k-ème couche.

— Par conséquent ${}^ka$ représente le vecteur de sortie de la k-ème couche, mais aussi le vecteur d'entrée de la $(k+1)$-ème couche.

— Le vecteur d'entrée x, sera noté ${}^0a$.

— Le vecteur de sortie y, sera noté ${}^La$ dans réseau de L couches.

Enfin, les résultats de la fonction de combinaison sont de même dimensions que les sorties de couche.

Pour les *combinaisons* :

— Une combinaison noté ${}^kz^j$ sera la combinaison du j-ème neurone, situé dans la k-ème couche.

— Par conséquent ${}^kz$ représente le vecteur des combinaisons de la k-ème couche.

4.1.4 Propagation

On parle de *propagation à sens unique* dans un réseau de neurones, car l'entrée x, affectée à la première couche, est transmise à la suivante, qui produira alors un vecteur de sortie.

Ce vecteur de sortie sera alors considéré comme un vecteur d'entrée pour la couche suivante, qui produira à l'aide de cette entrée un autre vecteur de sortie.

Et ainsi de suite, jusqu'à la dernière couche, qui produira finalement la sortie du réseau : y.

Nous voilà à présent correctement armés pour voir la propagation dans un réseau de neurones.

Voici l'algorithme de propagation au sein d'un réseau de neurones :

Algorithm 3 Algorithme de Propagation

Require: Un réseau de neurones de L couches, défini par $S = (S_1, S_2, ..., S_L)$.
Un vecteur d'entrée $x \in \mathbb{R}^n = \mathbb{R}^{S_1}$,
${}^0a = x$
for $k = 1$ **to** L **do**
  for $j = 1$ **to** S_k **do**
    ${}^kz^j = ({}^kW^j)^T({}^{k-1}a) + {}^kb^j = \sum_{i=1}^{S_{k-1}} {}^kW_i^j \times {}^{k-1}a^i + {}^kb^j$
    ${}^ka^j = g({}^kz^j)$
    {${}^ka^j$ est, comme vu précédemment, la sortie du j-ème neurone de la k-ème couche,
    ${}^kz^j$ est le résultat de la fonction d'agrégation, résultat très utile par la suite. ${}^kW^j$ est le vecteur de poids du j-ème neurone de la k-ème couche,
    et ${}^kb^j$ est le biais du j-ème neurone de la k-ème couche.}
  end for
end for
return ${}^La$ {${}^La$ est la sortie du réseau de neurones, que nous comparerons ensuite à y.}

Néanmoins, il existe aujourd'hui un grand nombre de bibliothèques permettant de faire du calcul matriciel de façon optimisée, appelées Basic Linear Algebra Subprograms (BLAS), telles que ATLAS, GotoBLAS, OpenBLAS.

Pour rappel, ${}^kW$ représente la matrice des poids de la k-ème couche, et ${}^{k-1}a$ le vecteur de sortie de la $(k-1)$-ème couche.

Par conséquent, on peut calculer le vecteur de sortie de la k-ème couche de la façon suivante :

$$ {}^kz = ({}^kW)^T({}^{k-1}a) + {}^kb $$
$$ {}^ka = g({}^kz) $$

Ce qui nous permet alors d'écrire l'algorithme de propagation de façon matricielle (et laisser aux bibliothèques de calcul matriciel le soin d'optimiser les calculs) :

Algorithm 4 Algorithme matriciel de Propagation

Require: Un réseau de neurones de L couches, défini par $S = (S_1, S_2, ..., S_L)$.
Un vecteur d'entrée $x \in \mathbb{R}^n = \mathbb{R}^{S_1}$,
${}^0a = x$
for $k = 1$ **to** L **do**
  ${}^kz = ({}^kW)^T({}^{k-1}a) + {}^kb$
  ${}^ka = g({}^kz)$
end for
return ${}^La$

4.1.5 Rétro-propagation du Gradient

L'algorithme de rétro-propagation est souvent à tort considéré comme étant la méthode d'optimisation utilisée pour minimiser la fonction de perte.

C'est en effet le rôle d'une méthode de descente de gradient.

Voici pour rappel une méthode générale de descente de gradient, vue dans la première partie :

Algorithm 5 Algorithme général de Descente de Gradient

Require: x et $y \in \mathbb{R}^n$,
θ_0, les paramètres initiaux du modèle,
et $\epsilon > 0$, seuil de tolérance.
repeat
  Calcul de $\nabla_\theta J(x, y, \theta_k)$.
  Calcul de α_k. {α_k peut être soit une constante, soit calculé en fonction de gradients $\nabla_\theta J(x, y, \theta)$}
  $\theta_{k+1} = \theta_k - \alpha_k \nabla_\theta J(x, y, \theta_k)$.
until $\nabla_\theta J(x, y, \theta_k) \leq \epsilon$
return θ_{k+1}

L'algorithme de rétro-propagation du gradient a pour simple but de calculer, de façon élégante, le gradient $\nabla_\theta J(x, y, \theta)$, avec θ représentant ici tous les paramètres du modèle, c'est à dire l'ensemble des poids ${}^kW_i^j$ et des biais ${}^kb^j$.

Calcul formel du Gradient

Il nous faut donc calculer $\frac{\partial J(x,y,\theta)}{\partial {}^{k}W_i^j}$, traduisant l'influence du poids ${}^{k}W_i^j$ sur la fonction de perte $J(x,y,\theta)$.

Et $\frac{\partial J(x,y,\theta)}{\partial {}^{k}b^j}$, qui traduit l'influence du biais ${}^{k}b^j$ sur la fonction de perte $J(x,y,\theta)$.

Toute dérivée partielle $\frac{\partial u}{\partial v}$ représente l'influence qu'a l'élément v sur l'élément u.

Nous utiliserons le Théorème de dérivation des fonctions composées, aussi appelée *chain rule*.

Nous rappelons que $J(x,y,\theta)$ est une fonction de perte de la forme :

$$J(x,y,\theta) = C({}^{L}a(x), y)$$

Afin de mieux comprendre la preuve, on peut s'aider d'un ***dessin*** ☺.

Calcul sur la Couche de Sortie Commençons par calculer le gradient de l'erreur pour les poids de la couche de sortie, $\frac{\partial C}{\partial {}^{L}W_i^j}$.

Ainsi que le gradient pour les biais de la couche de sortie, $\frac{\partial C}{\partial {}^{L}b^j}$.

Sur la dernière couche, nous avons

$${}^{L}z^j = ({}^{L}W^j)^T({}^{L-1}a) + {}^{L}b^j$$

Et

$${}^{L}a^j = g({}^{L}z^j)$$

Utilisons à présent la règle de chaînage sur $\frac{\partial C}{\partial {}^{L}W_i^j}$, car ${}^{L}a^j$ est une fonction composée.

Nous obtenons donc,

$$\left\{ \frac{\partial C}{\partial {}^{L}W_i^j} = \frac{\partial C}{\partial {}^{L}a^j} \times \frac{\partial {}^{L}a^j}{\partial {}^{L}z^j} \times \frac{\partial {}^{L}z^j}{\partial {}^{L}W_i^j} \frac{\partial C}{\partial {}^{L}b^j} = \frac{\partial C}{\partial {}^{L}a^j} \times \frac{\partial {}^{L}a^j}{\partial {}^{L}z^j} \times \frac{\partial {}^{L}z^j}{\partial {}^{L}b^j} \right.$$

On s'aperçoit ici que le terme $\frac{\partial C}{\partial {}^{L}a^j}$, exprime la variation de $C({}^{L}a(x), y)$ en fonction de la sortie du neurone ${}^{L}a^j$.

On voit aussi que le terme $\frac{\partial {}^{L}a^j}{\partial {}^{L}z^j}$, traduit la variation de la sortie du neurone ${}^{L}a^j$ en fonction de l'agrégat ${}^{L}z^j$, dépend de la *fonction d'activation* choisie.

Nous reviendrons sur les expressions de $\frac{\partial C}{\partial {}^{L}a^j}$ et $\frac{\partial {}^{L}a^j}{\partial {}^{L}z^j}$, car elles nous permettront de choisir de façon éclairée la fonction d'activation et la fonction coût,car des simplifications peuvent s'opérer.

Enfin $\frac{\partial {}^{L}z^j}{\partial {}^{L}W_i^j} = {}^{L-1}a^i$ et $\frac{\partial {}^{L}z^j}{\partial {}^{L}b_i^j} = 1$, ce qui est évident à vérifier, car

$${}^{L}z^j = \sum_{i=1}^{S_{L-1}} {}^{L}W_i^j \times {}^{L-1}a^i + {}^{L}b^j$$

Ces deux expressions traduisent l'influence des poids ${}^{L}W_i^j$ et du biais ${}^{L}b^j$ sur l'agrégat, ou résultat de la fonction d'agrégation, ${}^{L}z^j$

Pour la suite, nous allons poser ${}^{L}\delta^j = \frac{\partial C}{\partial {}^{L}a^j} \times \frac{\partial {}^{L}a^j}{\partial {}^{L}z^j}$, cela interviendra quand nous montrerons le caractère récursif de la rétropropagation.

Finalement, nous obtenons,

$$\left\{ \frac{\partial C}{\partial {}^{L}W_i^j} = {}^{L}\delta^j \times \frac{\partial {}^{L}z^j}{\partial {}^{L}W_i^j} = {}^{L}\delta^j \times {}^{L-1}a^i \frac{\partial C}{\partial {}^{L}b^j} = {}^{L}\delta^j \times \frac{\partial {}^{L}z^j}{\partial {}^{L}b^j} = {}^{L}\delta^j \right.$$

Avec,

$${}^{L}\delta^j = \frac{\partial C}{\partial {}^{L}a^j} \times \frac{\partial {}^{L}a^j}{\partial {}^{L}z^j}$$

Calcul sur l'avant dernière couche Après avoir explicité les calculs des $\frac{\partial C}{\partial {}^{L}W_i^j}$ et $\frac{\partial C}{\partial {}^{L}b^j}$, passons à présent au calcul sur l'avant dernière couche, soit aux $\frac{\partial C}{\partial {}^{L-1}W_i^j}$ et $\frac{\partial C}{\partial {}^{L-1}b^j}$.

Grâce à la *chain rule*, nous obtenons de la même façon que précédemment,

$$\left\{ \frac{\partial C}{\partial {}^{L-1}W_i^j} = \frac{\partial C}{\partial {}^{L-1}a^j} \times \frac{\partial {}^{L-1}a^j}{\partial {}^{L-1}z^j} \times \frac{\partial {}^{L-1}z^j}{\partial {}^{L-1}W_i^j} \frac{\partial C}{\partial {}^{L-1}b^j} = \frac{\partial C}{\partial {}^{L-1}a^j} \times \frac{\partial {}^{L-1}a^j}{\partial {}^{L-1}z^j} \times \frac{\partial {}^{L-1}z^j}{\partial {}^{L-1}b^j} \right.$$

On peut remarquer que l'utilisation de la *Chain rule* permet d'exprimer facilement $\frac{\partial^{L-1}a^j}{\partial^{L-1}z^j}$, $\frac{\partial^{L-1}z^j}{\partial^{L-1}W_i^j}$ et $\frac{\partial^{L-1}z^j}{\partial^{L-1}b^j}$.

En effet si la fonction d'activation reste la même pour tout le réseau de neurones, alors le terme $\frac{\partial^{L-1}a^j}{\partial^{L-1}z^j}$ est la même que $\frac{\partial^{L}a^j}{\partial^{L}z^j}$ calculée plus haut.

De même que précédemment $\frac{\partial^{L-1}z^j}{\partial^{L-1}W_i^j} = {}^{L-2}a^i$ et $\frac{\partial^{L-1}z^j}{\partial^{L-1}b_i^j} = 1$, car

$$^{L-1}z^j = \sum_{i=1}^{S_{L-2}} {}^{L-1}W_i^j \times {}^{L-2}a^i + {}^{L-1}b^j$$

La difficulté réside ici, dans le calcul de $\frac{\partial C}{\partial^{L-1}a^j}$.

Si on trace un réseau de neurones quelconque.

Prenons le j-ème neurone de la couche $L-1$, on peut voir qu'il transmet le résultat de son activation ${}^{L-1}a^j$ à tous les neurones de la couche L.

Cette diffusion d'informations a aussi un impact sur le coût, puisque si erreur il y a alors, les neurones de la couche L en feront les "frais".

Un neurone ${}^{L-1}a^i$ propage l'erreur à un neurone ${}^{L}a^j$ proportionnellement au poids ${}^{L}W_i^j$.

Nous devons donc décomposer $\frac{\partial C}{\partial^{L-1}a^j}$, grâce à la *chain rule*.

Le résultat est le suivant :

$$\frac{\partial C}{\partial^{L-1}a^j} = \sum_{l=1}^{L-1} \frac{\partial C}{\partial^{L}a^l} \times \frac{\partial^{L}a^l}{\partial^{L}z^l} \times \frac{\partial^{L}z^l}{\partial^{L-1}a^l}$$

En effet, le neurone ${}^{L-1}a^i$ transfère son erreur à tous les neurones ${}^{L}a^j$ de la couche suivante par le biais des poids, car $\frac{\partial^{L}z^l}{\partial^{L-1}a^j} = {}^{L-1}W_j^l$.

On remarque par ailleurs que

$$\frac{\partial C}{\partial^{L-1}a^j} = \sum_{l=1}^{L-1} {}^{L}\delta^l \times \frac{\partial^{L}z^l}{\partial^{L-1}a^j}$$

Car

$${}^{L}\delta^j = \frac{\partial C}{\partial^{L}a^j} \times \frac{\partial^{L}a^j}{\partial^{L}z^j}$$

Donc,

$${}^{L-1}\delta^j = \frac{\partial C}{\partial^{L-1}a^j} \times \frac{\partial^{L-1}a^j}{\partial^{L-1}z^j} = \sum_{l=1}^{L-1} {}^{L}\delta^l \times \frac{\partial^{L}z^l}{\partial^{L-1}a^j} \times \frac{\partial^{L-1}a^j}{\partial^{L-1}z^j}$$

Récapitulatif Récapitulons, afin de voir se dessiner un schéma de récurrence.

Sur la couche L,

$$\left\{ \frac{\partial C}{\partial^{L}W_i^j} = {}^{L}\delta^j \times \frac{\partial^{L}z^j}{\partial^{L}W_i^j} = {}^{L}\delta^j \times {}^{L-1}a^i \frac{\partial C}{\partial^{L}b^j} = {}^{L}\delta^j \times \frac{\partial^{L}z^j}{\partial^{L}b^j} = {}^{L}\delta^j \right.$$

Avec,

$${}^{L}\delta^j = \frac{\partial C}{\partial^{L}a^j} \times \frac{\partial^{L}a^j}{\partial^{L-1}z^j}$$

Sur la couche $L-1$,

$$\left\{ \frac{\partial C}{\partial^{L-1}W_i^j} = {}^{L-1}\delta^j \times \frac{\partial^{L-1}z^j}{\partial^{L-1}W_i^j} = {}^{L-1}\delta^j \times {}^{L-2}a^i \frac{\partial C}{\partial^{L-1}b^j} = {}^{L-1}\delta^j \times \frac{\partial^{L-1}z^j}{\partial^{L-1}b^j} = {}^{L-1}\delta^j \right.$$

Avec,

$${}^{L-1}\delta^j = \sum_{l=1}^{L-1} {}^{L}\delta^l \times \frac{\partial^{L}z^l}{\partial^{L-1}a^j} \times \frac{\partial^{L-1}a^j}{\partial^{L-1}z^j}$$

Calcul généralisé Essayons à présent d'exprimer une forme récurrente applicable à chaque couche.

On a $\forall k = 1..L$,

$$\left\{ \frac{\partial C}{\partial^{k}W_i^j} = {}^{k}\delta^j \times \frac{\partial^{k}z^j}{\partial^{k}W_i^j} = {}^{k}\delta^j \times {}^{k-1}a^i \frac{\partial C}{\partial^{k}b^j} = {}^{k}\delta^j \times \frac{\partial^{k}z^j}{\partial^{k}b^j} = {}^{k}\delta^j \right.$$

Avec

$${}^{k}\delta^j = \left\{ \frac{\partial C}{\partial^{L-1}a^j},\ si\ k = L \sum_{l=1}^{k-1} {}^{k+1}\delta^l \times \frac{\partial^{k+1}z^l}{\partial^{k}a^j} \times \frac{\partial^{k}a^j}{\partial^{k}z^j},\ si\ k \in \{1..L-1\} \right.$$

Ce qui nous permet d'exprimer l'algorithme de rétro-propagation de la façon suivante :

Algorithm 6 Algorithme de Rétro-propagation

Require: Un réseau de neurones de L couches, défini par $S = (S_1, S_2, ..., S_L)$.
Un vecteur d'entrée $x \in \mathbb{R}^n = \mathbb{R}^{S_1}$,
Un vecteur de sortie $y \in \mathbb{R}^m = \mathbb{R}^{S_L}$,
{Calcul du gradient sur la couche de sortie.}
for $j = 1$ **to** $L - 1$ **do**
  ${}^L\delta^j = \frac{\partial C}{\partial^{L-1} a^j}$
  $\frac{\partial C}{\partial^L b^j} = {}^L\delta^j$
  for $i = 1$ **to** L **do**
    $\frac{\partial C}{\partial^L W_i^j} = {}^L\delta^j \times {}^{L-1}a^i$
  end for
end for
for $k = L - 1$ **to** 1 **do**
  {Calcul du gradient sur la k-ème couche.}
  for $j = 1$ **to** $k - 1$ **do**
    ${}^k\delta^j = \sum_{l=1}^{k-1} {}^{k+1}\delta^l \times \frac{\partial^{k+1} z^l}{\partial^k a^j} \times \frac{\partial^k a^j}{\partial^k z^j}$
    $\frac{\partial C}{\partial^k b^j} = {}^k\delta^j$
    for $i = 1$ **to** k **do**
      $\frac{\partial C}{\partial^k W_i^j} = {}^k\delta^j \times {}^{k-1}a^i$
    end for
  end for
end for

Cet algorithme nous permet donc de calculer de façon très efficace tous les $\frac{\partial C}{\partial^k W_i^j}$ et $\frac{\partial C}{\partial^k b^j}$.

Il existe une version privilégiant le calcul matriciel, permettant un calcul plus rapide, grâce aux bibliothèques de calcul matriciel.

Nous ne redémontrerons pas l'algorithme, mais nous en donneront toutefois la formulation.

Algorithm 7 Algorithme de Rétro-propagation Matriciel

Require: Un réseau de neurones de L couches, défini par $S = (S_1, S_2, ..., S_L)$.
Un vecteur d'entrée $x \in \mathbb{R}^n = \mathbb{R}^{S_1}$,
Un vecteur de sortie $y \in \mathbb{R}^m = \mathbb{R}^{S_L}$,
{Calcul du gradient sur la couche de sortie.}
${}^L\delta = \nabla_{{}^{L-1}a} C$
$\nabla_{{}^L b} C = {}^L\delta$
$\nabla_{{}^L W} C = {}^L\delta \times {}^{L-1}a^T$
for $k = L - 1$ **to** 1 **do**
  {Calcul du gradient sur la k-ème couche.}
  ${}^k\delta = ((\nabla_{{}^k a} {}^{k+1}z)^T \times {}^{k+1}\delta) \odot \nabla_{{}^k z} {}^k a$
  $\nabla_{{}^k b} C = {}^k\delta$
  $\nabla_{{}^k W} C = {}^k\delta \times {}^{k-1}a^T$
end for

Nous implémenterons alors cette méthode, car plus rapide en machine.

Nous disposons à présent des éléments suivants :

— Une méthode de descente de gradient.
— L'algorithme de rétro-propagation, permettant le calcul du gradient de la fonction de perte de manière efficace.

Ainsi, nous disposons alors des éléments permettant de créer un réseau de neurones, ainsi que d'effectuer son apprentissage.

Nous allons maintenant quitter cette partie plutôt calculatoire, pour s'intéresser aux différents moyens existants pour améliorer les capacités de généralisation de nos futurs réseaux de neurones.

4.1.6 Théorème d'Approximation Universelle

Les réseaux de neurones permettent de représenter universellement n'importe quelle fonction, dans le sens où il existe un réseau de neurones, ayant une architecture donnée, des poids et biais donnés, pouvant approximer une fonction en particulier.

Le *Théorème d'Approximation Universelle* nous dit qu'il existe un réseau suffisamment "grand" pour avoir la précision souhaitée sur les résultats.

Toutefois, ce théorème ne dit pas quelle sera la taille de ce réseau, mais en affirme seulement l'existence.

4.1.7 Techniques de Régularisation

Un défi majeur en apprentissage machine est de choisir un algorithme qui aura de bons résultats,non seulement à l'apprentissage, mais aussi sur de nouvelles entrées.

Les stratégies utilisées en apprentissage machine, permettant de réduire l'*erreur de test*, parfois au dépend de l'*erreur d'apprentissage*, sont appelées techniques de régularisation.

Nous avons, dans le premier chapitre, vu les concepts généraux de la *généralisation*, du *sous-apprentissage* et du *sur-apprentissage*.

Nous verrons à présent des méthodes concrètes appliquées aux réseaux de neurones profonds.

Certaines méthodes ajouteront des contraintes d'optimisation sur la fonction de perte, d'autres des restrictions sur les paramètres.

Si les méthodes sont choisies avec soin, elles peuvent améliorer grandement la performance des réseaux de neurones profonds sur l'ensemble des données de test.

Pénalisation des paramètres

La pénalisation des paramètres est une des premières formes de régularisation, utilisée depuis les débuts de l'apprentissage machine.

Cette régularisation limite la capacité de modèles, tels que les réseaux de neurones, les régressions linéaires, polynomiales et logistiques, en ajoutant une composante de pénalisation des paramètres $\Omega(\theta)$ à la fonction de perte $J(x, y, \theta)$, tel que

$$J(x, y, \theta) = J_0(x, y, \theta) + \lambda\Omega(\theta)$$

Avec $\lambda \geq 0$, un hyper-paramètre pondérant la contribution de la composante de pénalisation.

Si $\lambda = 0$, il n'y a pas de régularisation.

Nous ajouterons par ailleurs que cette pénalisation n'affecte que les poids ${}^kW_i^j$ et non les biais ${}^kb^j$, car les biais requièrent moins de données pour ajuster précisément les données.

Régularisation L^2 Soit w, l'ensemble des poids du modèle,

On pose,

$$\Omega(\theta) = \Omega(w) = \frac{1}{2}||w||_2^2 = \frac{1}{2}w^T w$$

La fonction de perte s'écrit donc,

$$\tilde{J}(x, y, \theta) = J(x, y, \theta) + \frac{\lambda}{2}w^T w$$

Par conséquent le gradient s'écrit de la façon suivante,

$$\nabla_\theta \tilde{J}(x, y, \theta) = \nabla_\theta J(x, y, \theta) + \lambda w$$

Ainsi le poids ${}^kW_i^j$ aura tendance à diminuer par rapport à une fraction de lui même.

Augmentation du jeu de données

Une des meilleures façons d'améliorer la capacité de généralisation d'un modèle, est de l'entraîner sur plus de données.

Malheureusement, la taille des ensembles de données est généralement limitée.

Une façon de contourner le problème est de créer artificiellement des données et de les ajouter à l'ensemble d'apprentissage.

Cette méthode est principalement utilisée pour un domaine : la reconnaissance d'objets. En effet, il est très facile de transformer les données existantes par des transformations, telles que des rotations, des translations ou des homothéties, paramétrées aléatoirement. On peut aussi le faire sur d'autres types de données en ajoutant du *bruit blanc* sur une partie des entrées.

Arrêt prématuré

Lors de qu'un modèle avec une capacité suffisante pour sur-apprendre, on observe généralement que tandis ce que l'*erreur d'apprentissage* continue de diminuer, l'*erreur d'entraînement* recommence à augmenter.

Cela signifie que nous avons atteint un *minimum local* sur les données de test, il convient alors d'arrêter l'apprentissage avant que l'*erreur de généralisation* empire.

Dropout

La méthode *Dropout* est une méthode assez particulière de régularisation, qui consiste à entraîner tour à tour des sous-réseaux de notre réseaux de neurones.

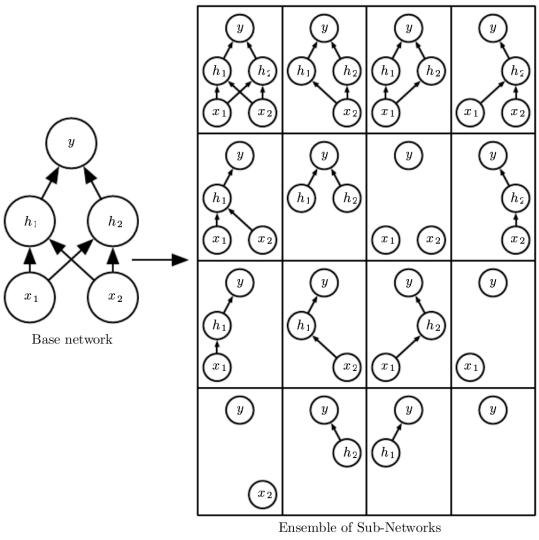

FIGURE 4.4 – Exemple d'application de la méthode de *Dropout*

Ainsi les différents sous-modèles se partagent les paramètres θ du modèle original.

Il en résulte un modèle beaucoup plus robuste, car les différents sous-modèles entraînent une recherche de paramètres pour atteindre des minima locaux différents.

Cela incite donc à une convergence vers un minimum local partagé par les différents sous-modèles.

Ce minimum local est de façon générale un bien meilleur minimum que ceux propres aux sous-modèles.

Cette méthode présente toutefois un inconvénient, elle est difficile à implémenter, mais est présente dans la plupart des librairies d'apprentissage profond.

4.2 Réseaux Convolutifs

Les réseaux de neurones convolutifs sont une famille particulière de réseaux de neurones, qui excelle en terme de performance dans le domaine de la vision par ordinateur.

Ils sont en effet spécialisés dans le traitement de données ayant une structure en forme de grille, c'est à dire que les données sont liées spatialement ou temporellement.

Voici des exemples de données ayant ce genre de structure :

— Les séries temporelles pouvant être vue comme une grille de dimension 1, car assimilables à des signaux, ayant pour unité un intervalle de temps fixe.

— Les images, pouvant être considérées comme étant des grilles de dimensions 2 de pixels.

Dans un premier temps, nous définirons rapidement l'opérateur de convolution.

Dans un second temps, nous discuterons de l'intérêt des convolutions dans les réseaux de neurones, ainsi que des différences existantes entre la convolution au sens mathématique et celle employée pour l'apprentissage profond.

Enfin nous parlerons d'une opération très souvent complémentaire à celle de *convolution*, appelée *pooling*.

4.2.1 L'Opérateur de Convolution

L'opérateur de convolution est un opérateur bilinéaire, c'est-à-dire linéaire par rapport à chacune des opérandes, et commutatif, généralement noté $*$.

En traitement du signal, cet opérateur représente l'interaction entre un filtre et un signal.
Elle peut être vue comme une moyenne mobile d'un signal pondérée par une fonction appelée *noyau*.

Cas continu

Le produit de convolution de deux fonctions réelles ou complexes f et g est défini tel que,

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(x-t)g(t)dt = \int_{-\infty}^{+\infty} f(t)g(x-t)dt$$

Cas discret

Le produit de convolution de deux suites f et g est défini tel que,

$$(f * g)(n) = \sum_{m=-\infty}^{+\infty} f(n-m)g(m) = \sum_{m=-\infty}^{+\infty} f(m)g(n-m)$$

Application aux images

On rappelle qu'une image I de M lignes et N colonnes est décrite par une fonction discrète $I(i,j)$ à support fini $S \subset \mathbb{Z}^2$ et à valeurs dans $\mathbb{R}$ ou $\mathbb{N}$.

Puisque l'image, décrite par la fonction I est nulle en dehors du support, on peut réécrire le produit de convolution entre des bornes finies.

Le produit de convolution en un point (i,j) de deux images I et K, noté $I * K$ est défini tel que,

On choisira d'imiter la notation matricielle pour plus de clarté.

$$(I * K)_{i,j} = \sum_m \sum_n I_{m,n} K_{i-m,j-n} = (K * I)_{i,j} = \sum_m \sum_n K_{m,n} I_{i-m,j-n}$$

L'image K est appelée *noyau*, ou *kernel*, c'est l'équivalent d'un filtre pour les images. Voici un exemple d'application de la convolution sur une image :

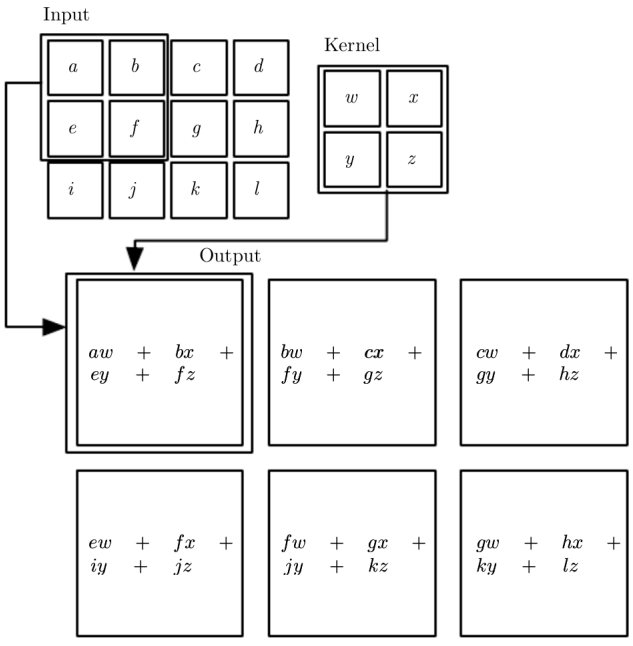

FIGURE 4.5 – Exemple de convolution entre une image I, $(8,3)$ et un noyau K, $(2,2)$

Dans cet exemple de convolution entre une image I, $(8,3)$ et un noyau K, $(2,2)$, on peut remarquer tout d'abord, que la image issue de la convolution est de forme $(6,2)$.

En effet la convolution sans *padding* réduit la taille de la fenêtre.

Si l'image I est de taille (m,n) et le noyau de taille (k,l) alors l'image convolutée sera de taille $(m-k+1, n-l+1)$.

Convolution et Corrélation Croisée

Nous avons défini précédemment la convolution entre deux images, ici une *map de features* I et un noyau K, de la façon suivante : Nous parlerons ici de façon plus générale de tenseurs.

Un *tenseur* est un *tableau multi-dimensionnel*.
L'*ordre* d'un tenseur est égal à la dimension du tableau.

Par exemple, une image en couleurs (RGB) de 256×256 pixels peut être décrite par un tenseur d'ordre 3.

Nous écrirons alors que la dimension de l'image est $3 \times 256 \times 256$, ou bien encore $256 \times 256 \times 3$.

Toutefois, d'un point de vue machine, nous indexons celle variant le plus lentement en premier et celle variant le plus rapidement en dernier. Nous retiendrons ici l'écriture $3 \times 256 \times 256$.

Nous avons défini précédemment la convolution entre deux tenseurs d'ordre 2, ici une *map de features* I et un noyau K, de la façon suivante :

$$(K * I)_{i,j} = \sum_m \sum_n K_{m,n} I_{i-m,j-n}$$

qui est équivalente à

$$(I * K)_{i,j} = \sum_m \sum_n I_{m,n} K_{i-m,j-n}$$

Mais dans la pratique, le noyau K étant beaucoup plus "*petit*" que I, la première formule est préférable, au niveau du calcul des bornes des indices m et n.

Par conséquent, même si la propriété de commutativité existe, une seule expression de la convolution nous intéresse. La corrélation croisée, très proche de la convolution, s'écrit de la façon suivante :

$$(K * I)_{i,j} = \sum_m \sum_n K_{m,n} I_{i+m,j+n}$$

La différence étant par rapport au sens de l'indexation. Cette opération n'est pas commutative contrairement à la convolution, mais étant donnée que seule l'opération $K * I$ nous intéresse, cette propriété ne nous apporte rien.

Il est à noter que dans beaucoup de librairies d'Apprentissage Machine, l'opération de convolution est implémentée en tant que corrélation croisée. Dans la suite, nous appellerons convolution les deux opérations sans distinction.

4.2.2 Intérêts de la Convolution dans les Réseaux de Neurones

La convolution s'appuie sur trois idées importantes permettant l'amélioration des performances, l'*interaction partielle*, le *partage des paramètres* et l'*invariance par translation*.

L'interaction partielle

Dans les réseaux de neurones traditionnels, le passage de l'information d'une couche à la suivante se fait en autre par la multiplication matricielle du vecteur d'entrée ${}^{k-1}a$ de dimension m, par la matrice ${}^{k}W$ de dimension $m \times n$, la sortie du neurone ${}^{k-1}a$ sera donc de dimension n.

Pour rappel :

$${}^{k}z = ({}^{k}W)^T({}^{k-1}a) + {}^{k}b$$

On voit donc qu'ici l'interaction, ou connectivité, est totale puisque toutes les entrées ${}^{k-1}a^j$ sont connectées par les poids ${}^{k}W_i^j$ aux sorties ${}^{k-1}a^i$.

Dans le cas où l'on travaille avec des images, on peut se demander si pour un pixel donné l'information apportée par un pixel situé à l'extrême opposé de l'image est pertinente.

Ainsi, envisager une connectivité partielle entre le vecteur d'entrée ${}^{k-1}a$ et le vecteur de sortie ${}^{k-1}a$ peut sembler intéressante si l'information apportée s'apparente à du bruit.

La convolution nous permet en effet cette interaction partielle, en effet la dimension du noyau ${}^{k}W$, de la forme $k \times n$, et si $k < m$ alors l'interaction sera en effet partielle.

Voici à présent la représentation de l'interaction partielle dans un cas matriciel, plus illustratif qu'un simple cas vectoriel :

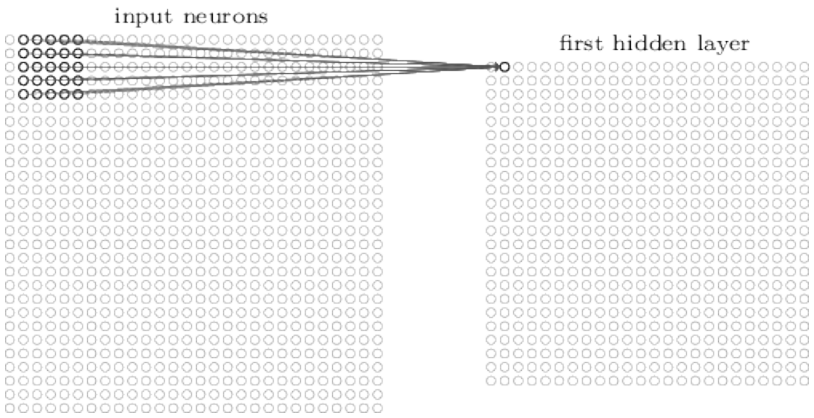

FIGURE 4.6 – Représentation de l'interaction partielle

Le partage des paramètres et invariance par translation

Dans un réseau convolutif, chaque paramètre du noyau est utilisé à toutes les positions du vecteur d'entrée, sauf peut-être pour les premiers et derniers éléments du vecteur d'entrée.

Ainsi le partage de paramètres pour la convolution signifie plutôt que d'apprendre plusieurs ensembles de paramètres pour chaque sous-région du vecteur d'entrée, un seul ensemble sera appris.

Dans le cas de la figure précédente, où la fenêtre est de dimension 5×5, un seul noyau de dimension 5×5 sera appris pour cette couche de convolution.

Ce partage de paramètres permet alors à notre couche de convolution d'être invariante par translation.

En effet supposons que le noyau de notre couche soit entraîné à détecter un motif, ou *feature* en particulier sur une image. Le noyau unique, et donc partagé à toutes les sous-régions de l'entrée, pourra détecter le motif peu importe sa position dans l'image.

Cette propriété d'invariance par translation rend les couches de convolutions très intéressantes, car elles sont plus robustes au bruit que des couches entièrement connectées.

4.2.3 La Convolution appliquée aux Réseaux de Neurones

Le terme *convolution* employé dans le contexte de l'apprentissage profond, diffère quelque peu de la convolution mathématique définie comme ci-dessus.

Tout d'abord, quand on parle d'une convolution dans le contexte des réseaux de neurones, il s'agit en fait de plusieurs convolutions menées en parallèle.

Énumérons à présent les différents noyaux K de convolutions, en fonction de I le tenseur d'entrée et de S le tenseur de sortie :

— Soient I de dimensions (H_I, W_I) et S de dimensions (H_S, W_S),
Et si $H_I > H_S + 1$ et $W_I > W_S + 1$,
Alors K sera de dimensions :

$$(H_K, H_K) = (H_I - H_S + 1, W_I - W_S + 1)$$

Cette convolution nous permet d'extraire seulement une seule *feature* depuis chaque sous-région de I.

— Soient I de dimensions (H_I, W_I) et S de dimensions (C_S, H_S, W_S),
Et si $H_I > H_S + 1$ et $W_I > W_S + 1$,
Alors K sera de dimensions :

$$(C_K^{out}, H_K, W_K) = (C_S, H_I - H_S + 1, W_I - W_S + 1)$$

Cette convolution nous permet d'extraire C_S *features* depuis chaque sous-région de I.

— Soient I de dimensions (C_I, H_I, W_I) et S de dimensions (C_S, H_S, W_S),
Et si $H_I > H_S + 1$ et $W_I > W_S + 1$,
Alors K sera de dimensions :

$$(C_K^{out}, C_K^{in}, H_K, W_K) = (C_S, C_I, H_I - H_S + 1, W_I - W_S + 1)$$

Cette convolution nous permet d'extraire C_S *features* croisées entre les C_I *canaux* de chaque sous-région de I.

Cette dernière expression est l'expression générale de la convolution. On peut remarquer pour le premier cas, que $(H_I, W_I) = (1, H_I, W_I)$, $(H_S, W_S) = (1, H_S, W_S)$ et $(H_I, W_I) = (1, 1, H_I, W_I)$

Convolution tensorielle

Soient I de dimensions (C_I, H_I, W_I) et S de dimensions (C_S, H_S, W_S),
Et si $H_I > H_S + 1$ et $W_I > W_S + 1$,
Alors K sera de dimensions :

$$(C_K^{out}, C_K^{in}, H_K, W_K) = (C_S, C_I, H_I - H_S + 1, W_I - W_S + 1)$$

Donc $\forall k \in [0, C_S[,\ \forall i \in [0, H_S[,\ \forall j \in [0, W_S[$,

$$S(k, i, j) = \sum_l \sum_m \sum_n K_{k,l,m,n} \times I_{l,m+i,n+j}$$

Avec les indices de sommations suivants,
- $l \in [0, C_K^{in}[$
- $m \in [0, H_K[$ et $(m + i) \in [0, H_I[$
- $n \in [0, W_K[$ et $(n + j) \in [0, W_I[$

Padding et Strides

Padding Nous avons pu observer précédemment que la convolution ne préservait pas les dimensions spatiales, la hauteur et la largeur, entre le tenseur d'entrée I et le tenseur de sortie S. Soit I un tenseur de dimensions (H_I, W_I) et K un noyau de dimensions (H_K, W_K),
Alors S aura les dimensions suivantes :

$$(H_S, W_S) = (H_I - H_K + 1, W_I - W_K + 1)$$

Toutefois il peut être intéressant dans certains cas de préserver ces dimensions. Le *padding*, signifiant *rembourrage*, permet de remédier à ce problème. Cette méthode consiste à créer un tenseur I', en ajoutant à I des lignes et colonnes de zéros sur ses *bords*.
Ainsi,

$$(H_{I'}, W_{I'}) = (H_I + p_H, W_I + p_W)$$

Donc si on applique la convolution à I',

$$(H_S, W_S) = (H_I + p_H - H_K + 1, W_I + p_W - W_K + 1)$$

Donc,

$$(H_S, W_S) = (H_I, W_I) \leftrightarrow (p_H, p_W) = (H_K - 1, W_K - 1)$$

Dans le cas où les dimensions du noyau sont impaires, le *padding* se fera de façon symétrique autour de I.

Dans le cas où les dimensions du noyau sont paires, le *padding* est par convention pour les bords *haut* et *gauche* de I.

Exemple Dans le cas d'une convolution de noyau K de dimensions $(3, 3)$ appliquée à un tenseur I de dimensions $(3, 3)$, si on veut garder S de même dimensions que I.
Soit

$$I = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Alors $(p_H, p_W) = (2, 2)$
Donc

$$I' = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & c & 0 \\ 0 & d & e & f & 0 \\ 0 & g & h & i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Strides Le terme anglais *stride* peut être traduit en français par le *pas de déplacement*.
Augmenter le pas de déplacement dans une convolution à pour effet de diminuer la dimension du tenseur de sortie S.

Nous n'allons pas ici expliciter l'impact de ce changement sur les effets de l'apprentissage, un lien sera fait dans la partie dédiée au *Pooling*, mais simplement décrire les modifications sur la sortie.

$$(H_S, W_S) = ((H_I - H_K)/s_H + 1, (W_I - W_K)/s_W + 1)$$

Les divisions sont ici *entières* car les dimensions de sorties sont forcément entières.

Convolution généralisée

Soient I de dimensions (C_I, H_I, W_I),
S de dimensions (C_S, H_S, W_S),
et K de dimensions $(C_K^{out}, C_K^{in}, H_K, W_K)$, avec $C_K^{out} = C_S$ et $C_K^{in} = C_I$
Il nous faut alors trouver (p_H, p_W) et (s_H, s_W), tels que :

$$(C_S, H_S, W_S) = (C_S, (H_I + p_H - H_K)/s_H + 1, (W_I + p_W - W_K)/s_W + 1)$$

On pose alors I' le tenseur "*rembourré*" de I, avec

$$(C_{I'}, H_{I'}, W_{I'}) = (C_I, H_I + p_H, W_I + p_W)$$

Donc,

$$(C_S, H_S, W_S) = ((H_{I'} - H_K)/s_H + 1, (W_{I'} - W_K)/s_W + 1)$$

Donc $\forall k \in [0, C_S[,\ \forall i \in [0, H_S[,\ \forall j \in [0, W_S[$,

$$S(k, i, j) = \sum_l \sum_m \sum_n K_{k,l,m,n} \times {I'}_{l,m+i\times s_H, n+j\times s_W}$$

Avec,
- $l \in [0, C_K^{in}[$
- $m \in [0, H_K[$ et $(m + i \times s_H) \in [0, H_I[$
- $n \in [0, W_K[$ et $(n + j \times s_W) \in [0, W_I[$

4.2.4 La Réduction de la Dimensionnalité

La *réduction de la dimensionnalité* est un défi majeur en Statistiques et en Apprentissage Machine. Prenons un exemple pour illustrer le problème. Imaginons que nous souhaitons classifier des images issues d'Internet de dimensions $(3, 256, 256)$, en 1000 classes distinctes. Ce problème revient à devoir trouver un *mapping* entre un tenseur d'entrée de dimensions $(3, 256, 256)$ et un tenseur de sortie de dimension (1000). On remarque qu'on peut transformer le tenseur d'entrée en un tenseur de dimension $(3 \times 256 \times 256) = (196608)$. On peut alors se poser plusieurs questions,
- Quelles sont les variables significatives parmi les 196608 variables disponibles ?
- Comment *transformer*, ou *modeler*, cette quantité d'information pour en extraire l'essentiel, c'est à dire sa classe ?

Tout d'abord la première question n'a pas vraiment de sens dans de nombreux domaines, et particulièrement en *vision par ordinateur*, car les pixels d'une image sont liés spatialement. Ils sont donc dépendants de leurs voisins. La deuxième est plus beaucoup plus intéressante.

Une approche plutôt naïve pourrait suggérer de transformer l'image de dimensions $(3, 256, 256)$ en niveaux de gris de dimensions $(1, 256, 256)$ puis de la *redimensionner* en $(1, 32, 32)$, ce qui nous permet d'obtenir un tenseur d'entrée de dimension (1024).

FIGURE 4.7 – Pré-traitement d'une image

Ici les dimensions d'entrées et de sorties sont comparables, nous pouvons donc "choisir" plus facilement les combinaisons de variables à effectuer pour classifier les images. Toutefois la perte d'information, due aux pré-traitements appliqués aux données, est grande, ce qui aura un impact potentiel sur la performance de notre algorithme apprenant.

Une approche un peu plus réfléchie pourrait suggérer d'une part de ne pas réduire la dimension des données non pas pendant le pré-traitement, mais surtout d'autre part, d'éviter de la réduire d'un seul coup, mais plutôt petit à petit. Nous allons lister deux transformations très efficaces appliquées aux réseaux de neurones, en particulier convolutifs.

Pooling d'informations

Le *pooling*, signifiant *mise en commun*, consiste à résumer l'information apportée par les valeurs de sorties proches en une seule valeur. Il existe deux déclinaisons du *pooling*,

— Le *max pooling*, qui calcule le maximum des valeurs données en entrée.

— L'*average pooling*, qui calcule la moyenne des valeurs données en entrée.

— Les convolutions avec une ou plusieurs *strides* > 1

Nous allons maintenant voir un exemple de calcul d'un *pooling* appliqué sur les composantes spatiales.
Soit I un tenseur de dimensions (C_I, H_I, W_I), et une fonction de *pooling* définie par une fenêtre de dimension (H_P, W_P), de *padding* (p_H, p_W) et de *strides* (s_H, s_W),
Alors S sera de dimensions :

$$(C_S, H_S, W_S) = (C_I, (H_I + p_H - H_K)/s_H + 1, (W_I + p_W - W_K)/s_W + 1))$$

Ces deux déclinaisons sont utilisées dans des contextes assez différents.

Max-Pooling Le *max pooling* est principalement utilisé pour réduire la dimension des données de façon spatiale, c'est une méthode robuste aux bruits ambients, car elle ne transmet que l'information ayant le plus d'intensité.

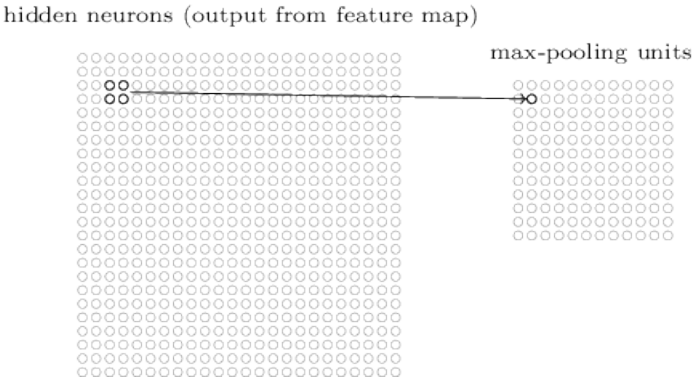

FIGURE 4.8 – *Max-pooling* sur une fenêtre $(2, 2)$

Average Pooling L'*average pooling* est principalement utilisé pour réduire la dimension des données au niveau des canaux, car elle permet de prendre en compte les valeurs des différentes *features* et de les agréger.

Pooling par Convolution Les convolutions à strides > 1 sont à l'heure actuelle *à la mode*. Elles font le travail à la fois de couches de convolution, mais aussi de couches de *pooling*, permettant ainsi d'une part de simplifier le modèle, ce qui réduit le nombre de *floating-point operations*, mais aussi d'avoir une opération de *pooling* plus "maléable", car la fonction calculée n'est pas statique, mais sera amenée à évoluer lors de la rétro-propagation. De plus, elles sont souvent suivies de fonctions d'activation, rendant la sortie plus robuste aux bruits stochastiques. Elle est utilisée dans plusieurs modèles très performants, justifiant ainsi sa popularité montante.

4.3 Applications à la vision par ordinateur

4.3.1 Classification

4.3.2 Détection d'objets

4.3.3 Segmentation d'images

Segmentation de classes

Segmentation d'instances

Chapitre 5

L'Apprentissage Profond appliqué à l'Imagerie Satellite

5.1 Présentation des données

5.1.1 Données SpaceNet

5.1.2 Données Pléiades

5.2 Cartographie Automatisée par Segmentation de Classes

5.2.1 Protocole expérimental

5.2.2 Résultats

5.2.3 Comparaisons entre modèles

5.3 Recherche de Similarités entre Images

5.3.1 Protocole expérimental

5.3.2 Résultats

5.3.3 Comparaisons entre modèles

Chapitre 6

Conclusion