

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

Guillaume Rochette

20 octobre 2017

Table des matières

1	Introduction	3
1.1	Présentation de l'Entreprise	3
1.1.1	Le Groupe Thales	3
1.1.2	Thales Services	3
1.2	Objectifs du Stage	4
2	L'Imagerie Satellite	6
2.1	Fonctionnement d'un Satellite d'Observation	6
2.1.1	Capteurs Optiques	6
2.1.2	L'Orbite des Satellites	9
2.2	Images Satellite	10
2.2.1	Métadonnées	10
2.2.2	Compression et Formats de Fichier	11
2.2.3	Caractéristiques d'une Image Satellite	11
3	L'Apprentissage Machine	14
3.1	Définition d'un Algorithme Apprenant	14
3.1.1	La Tâche	14
3.1.2	La Mesure de la Performance	15
3.1.3	L'Expérience	15
3.2	Contraintes d'Apprentissage et Régularisation	16
3.2.1	Sous-apprentissage, Sur-apprentissage et Capacité d'un Algorithme Apprenant	16
3.2.2	Régularisation	18
3.3	Paramétrisation et Validation d'un Algorithme	19
3.4	Optimisation	19
3.4.1	Méthodes exactes	19
3.4.2	Méthodes itératives	20
4	L'Apprentissage Profond	23
4.1	Réseaux de Neurones Profonds	23
4.1.1	Définition d'un Neurone Artificiel	23
4.1.2	Définition d'un Réseau de Neurones	24
4.1.3	Notations	26
4.1.4	Propagation	27
4.1.5	Rétro-propagation du Gradient	27
4.1.6	Théorème d'Approximation Universelle	31
4.1.7	Techniques de Régularisation	31
4.2	Réseaux Convolutifs	33
4.2.1	L'Opérateur de Convolution	33
4.2.2	Intérêts de la Convolution dans les Réseaux de Neurones	35
4.2.3	La Convolution appliquée aux Réseaux de Neurones	36
4.2.4	La Réduction de la Dimensionnalité	38
4.3	Applications à la Vision par Ordinateur	40

4.3.1	Classification	40
4.3.2	Détection d'Objets	41
4.3.3	Segmentation d'Images	42
5	L'Apprentissage Profond appliqué à l'Imagerie Satellite	44
5.1	Cartographie Automatisée par Segmentation de Classes	44
5.1.1	Présentation des Données	45
5.1.2	Calcul GPU et Framework Deep Learning	48
5.1.3	Expériences	48
5.1.4	Protocole expérimental	58
5.1.5	Résultats	58
5.1.6	Comparaisons entre modèles	58
6	Conclusion	59
6.1	Algorithme de Descente de Gradient : Adam	61

Chapitre 1

Introduction

1.1 Présentation de l'Entreprise

1.1.1 Le Groupe Thales

Le groupe Thales, historiquement fabricant d'électronique, est un acteur français majeur dans les secteurs de l'aéronautique, le spatial, la défense, la sécurité et le transport terrestre.



FIGURE 1.1 – Secteurs d'activités de Thales

Le groupe Thales est implanté dans 56 pays et emploie 64000 salariés, dont 34000 collaborateurs répartis dans 70 sites en France.

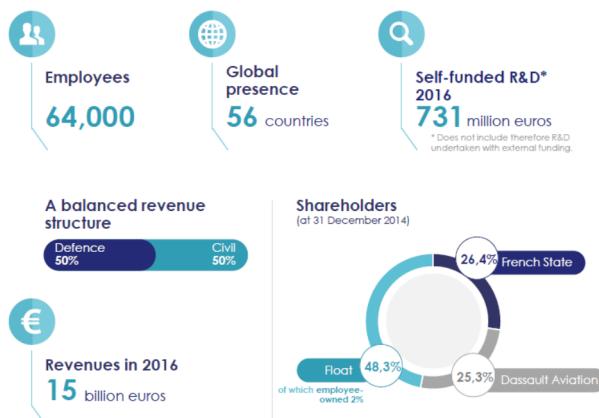


FIGURE 1.2 – Chiffres Clés du Groupe Thales

1.1.2 Thales Services

Thales Services est une entreprise spécialisée dans les activités de conception, développement et maintenance de systèmes informatiques sécurisés.

Elle est implantée sur tout le territoire français, chaque site ayant des spécialisations différentes.



FIGURE 1.3 – Implantation de Thales Services en France

Le groupe Thales, et par extension Thales Services, est organisé de façon *matricielle*, c'est-à-dire par zones géographiques et par domaines d'activités.

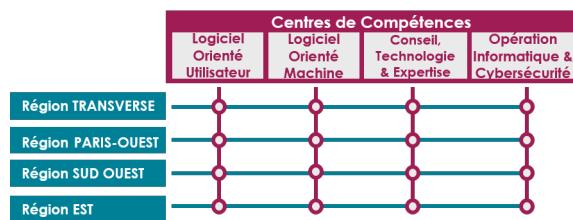


FIGURE 1.4 – Organisation matricielle de Thales Services

Ce stage s'effectue dans le *Centre de Compétences Logiciel Orienté Machine* de la *Région Sud-Ouest*.

1.2 Objectifs du Stage

La place de l'apprentissage profond grandit dans de nombreux domaines de l'industrie, et particulièrement dans les secteurs liés au traitement d'image.

Le potentiel de l'apprentissage profond à réaliser des tâches jugées trop complexes pour des algorithmes *traditionnels* est immense. Pour certaines applications de vision, les machines offrent aujourd'hui des performances supérieures à celles d'un humain. Les avancées les plus spectaculaires à ce jour sont majoritairement réalisées sur des images provenant de la photographie grand-public, avec ses particularités et ses contraintes.

Thales Services, dont l'un des clients est le Centre National d'Études Spatiales (CNES), produit des suites de logiciels utilisés dans la chaîne de traitement d'images pour des satellites tels que Pléiades, SPOT ou encore Sentinel-2.

L'application de méthodes d'apprentissage profond à des images satellites est prometteuse mais encore peu utilisée, justifiant l'intérêt et la volonté de Thales Services de monter en compétence dans ce domaine.

Les objectifs de ce stage sont les suivants :

- Apporter des connaissances relatives à l'apprentissage profond appliquée au traitement de l'image.
- Déterminer des opportunités d'application de vision par ordinateur pour des images satellite, telle que de la segmentation d'image ou de la recherche de similarités.
- Proposer et évaluer des modèles pour les applications proposées.

Dans un premier temps, nous aborderons l'imagerie spatiale, présentant ses enjeux, le fonctionnement général d'un satellite de télédétection, et enfin les caractéristiques et problématiques liées aux images satellite.

Dans un second temps, nous définirons l'apprentissage machine et ses concepts clés.

Dans un troisième temps, nous définirons précisément le domaine de l'apprentissage profond, nous expliquerons en détail le fonctionnement des réseaux de neurones, les motivations de cette science, ainsi qu'une explication détaillée des réseaux de neurones spécialisés dans le traitement des images

Enfin, nous ferons le lien entre apprentissage profond et imagerie spatiale, justifiant l'utilité d'appliquer le premier au second, ainsi que les applications réalisées au cours de ce stage permettant d'extraire des informations des images satellite. Les tâches que nous présenterons sont la segmentation d'images et la recherche de similarités.

Chapitre 2

L'Imagerie Satellite

De tout temps, l'humain a toujours souhaité pouvoir observer le monde, que ce soit à des fins militaires, scientifiques ou à simple but de contemplation.

Les besoins en matière de *télédétection* sont plus anciens que les premiers satellites artificiels. Dès la fin du XIX ème siècle, lors du siège de Paris en 1870, des ballons effectuaient des missions de reconnaissance en utilisant les progrès en matière de photographie.

Jusqu'aux années 1970, la télédétection servait majoritairement à l'espionnage et la cartographie, principalement réalisée par le biais d'appareils photographiques emportés dans des avions. Les premières images de la Terre depuis l'Espace ont été réalisées de cette même manière, des appareils photographiques à microfilm étaient embarqués dans des satellites artificiels. Il est à noter qu'un défi de taille était la réception de ces microfilms alors largués vers la Terre depuis l'Espace.

L'apparition de *capteurs numériques* a révolutionné le domaine de la télédétection, car permettant de s'abstraire du support physique, problématique de par sa quantité limitée à bord, ainsi que sa réception.

Nous aborderons d'abord le fonctionnement d'un *satellite d'observation*, puis nous survolerons les différentes particularités d'une image satellite.

2.1 Fonctionnement d'un Satellite d'Observation

Nous allons d'abord présenter brièvement le fonctionnement des différents types de capteurs numériques existants. Puis nous aborderons les différents mécanismes relativement au contrôle de la trajectoire du satellite.

2.1.1 Capteurs Optiques

Selon leur longueur d'onde, les ondes électromagnétiques portent des noms différents

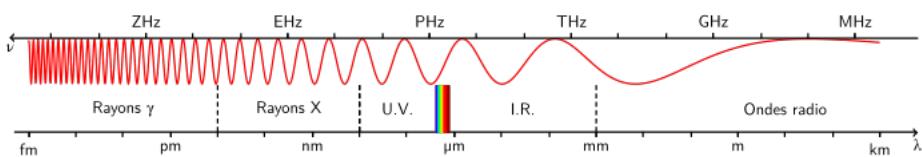


FIGURE 2.1 – Domaines du spectre électromagnétique

Le domaine *optique* est la réunion du domaine du visible et de l'infrarouge (I.R.). Bien que les satellites puissent embarquer aussi d'autres types de capteurs, nous n'en parlerons pas, car cela n'a pas de rapport avec l'Imagerie Satellite.

Par ailleurs, nous nous focaliserons uniquement sur les capteurs passifs, c'est-à-dire des capteurs qui ne font que mesurer le rayonnement d'un objet, contrairement aux capteurs actifs, comme les radars qui émettent un rayonnement et en observent l'écho sur l'objet.

Capteurs de type Whisk Broom

Ce sont les premiers capteurs historiques utilisés pour l'imagerie satellite, utilisés par exemple dès 1970 sur les satellites du programme LANDSAT de la NASA.

Le fonctionnement de ce type de capteur s'apparente au mécanisme des premiers scanners, c'est à dire une unique cellule de détection, qui nécessitait un double balayage selon les axes des lignes et des colonnes pour décrire la scène. En anglais, le terme *whisk broom* désigne des balais à *fouetter*, comme par exemple des plumeaux.

Voici un schéma décrivant le fonctionnement d'un capteur de type *whisk broom* :

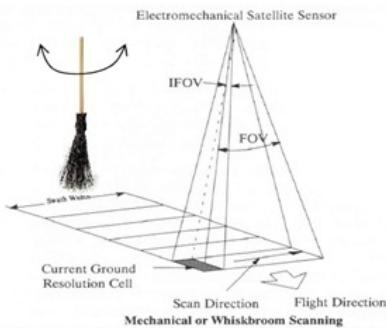


FIGURE 2.2 – Capteur de type *Whisk Broom*

Les avantages d'un capteur *whisk broom* sont les suivants :

- Étant donné qu'un seul capteur est utilisé, l'étalonnage de l'appareil est facile, permettant ainsi un rendu homogène de l'image.
- La fauchée, c'est-à-dire la largeur au sol de l'image, peut être très large.

Ce type de capteur présente aussi des inconvénients :

- Le balayage mécanique est complexe à mettre en œuvre, au niveau des mécanismes à utiliser pour compenser les oscillations du satellite. L'usure de tels mécanismes limite la durée de vie utile du satellite.
- Le temps d'intégration, c'est-à-dire le temps d'acquisition requis pour effectuer une mesure, est le fruit d'un compromis entre une bonne qualité radiométrique et la résolution au sol. En effet, pour obtenir une mesure de qualité, on préfère faire une moyenne sur un grand nombre d'acquisitions, limité par la vitesse orbitale du satellite et l'amplitude des oscillations du capteur, impactant directement la largeur de la fauchée.

Capteurs de type Push Broom

Le fonctionnement de ces capteurs est inspiré des scanners et photocopieurs grand-public apparus dans les années 1980.

Le premier satellite équipé d'un tel dispositif était SPOT 1, un satellite conçu par le CNES en 1986. Le principe de fonctionnement est assez simple, au lieu d'utiliser une unique cellule comme pour les capteurs *whisk broom*, on utilise des lignes, aussi appelées barrettes, de cellules de détection. Ainsi l'acquisition est simultanée pour les détecteurs composants une seule et même ligne.

Une image, composée de m lignes et n colonnes, est donc le résultat de n acquisitions simultanées répétées à m intervalles de temps suivant la trajectoire du satellite.

Voici un schéma décrivant le fonctionnement d'un capteur de type *push broom* :

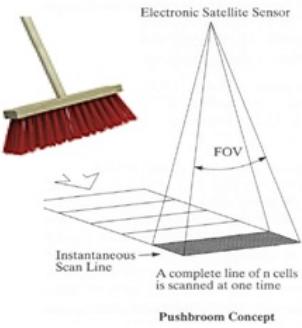


FIGURE 2.3 – Capteur de type *Push Broom*

Les avantages d'un capteur *push broom* sont les suivants :

- Simplicité du système.
- Perturbations géométriques seulement le long des colonnes.
- Larges fauchées grâce à la miniaturisation des cellules.

Ce type de capteur présente aussi des inconvénients :

- Le plan focal, ou l'alignement, de l'ensemble des capteurs est assez difficile à réaliser.
- En raison du nombre important de détecteurs, l'étalonnage radiométrique de l'ensemble est assez complexe.

Capteurs de type Matriciel

Le fonctionnement de ce capteur est similaire à celui d'un appareil photographique disponible au grand public.

Les cellules de détection sont réparties en forme de grille. Ainsi tous les pixels de l'image sont acquis de façon simultanée. Voici un schéma décrivant le fonctionnement d'un capteur de type *matriciel* :

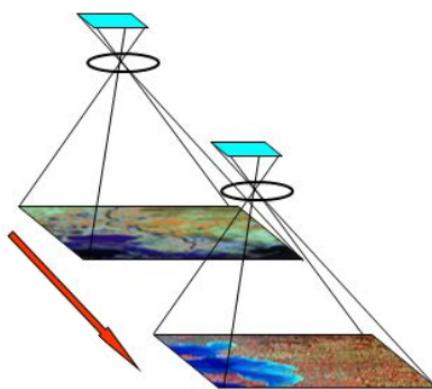


FIGURE 2.4 – Capteur de type *Matriciel*

Les avantages d'un capteur *matriciel* sont les suivants :

- Grâce à l'acquisition simultanée de tous les pixels de l'image, il ne peut y avoir de perturbations géométriques, sauf en cas de flou de bougé.
- Possibilité de photographier une zone selon plusieurs angles (de visée).

Ce type de capteur présente aussi des inconvénients :

- Alignement des capteurs très difficile à réaliser de par le nombre très important de détecteurs et de leur répartition en forme de matrice.
- Étalonnage radiométrique délicat en raison du grand nombre de détecteurs.
- Difficultés d'assembler des prises de vue pour en faire des mosaïques.

2.1.2 L'Orbite des Satellites

Le mouvement d'un satellite artificiel autour de la Terre, appelé *orbite*, est qualifié de képlérien car il respecte les trois lois de Kepler :

- *Loi des orbites* : Le satellite décrit une trajectoire elliptique, l'un des foyer de l'ellipse étant la Terre. Dans le cas présent, l'ellipse est quasi-circulaire.
- *Loi des aires* : L'aire de la surface délimitée par la Terre et un arc \widehat{AB} de la trajectoire du satellite, noté \widehat{ABT} , est égale à l'aire d'une autre surface définie par un autre arc \widehat{CD} de l'ellipse, , noté \widehat{CDT} , si le temps de parcours de l'arc \widehat{AB} , noté $t_{\widehat{AB}}$, est égal à $t_{\widehat{CD}}$.
- *Loi des périodes* : Le carré de la période T d'un satellite est proportionnel au cube du demi-grand axe a de la trajectoire elliptique du satellite : $\frac{T^2}{a^3} = \frac{4\pi^2}{\mu}$

Deux types d'orbites sont principalement utilisées pour les satellites

L'Orbite Géostationnaire

L'orbite *géostationnaire* implique comme son nom l'indique que la position du satellite est fixe par rapport à la Terre. Il faut que l'orbite du satellite soit circulaire autour de la Terre, et que la vitesse *angulaire* du satellite soit égale à la vitesse *angulaire* de la Terre.

Ce type d'orbite est utilisé pour les satellites de télécommunication, ainsi que pour des applications de surveillance de la Terre comme la météorologie, nécessitant un survol constant d'une zone définie.

L'Orbite Héliosynchrone

L'orbite *héliosynchrone* signifie que le satellite est "synchronisé" avec le soleil, ce qui en d'autres termes implique que le satellite doit tourner à la même vitesse autour de la Terre que la Terre autour du Soleil.

On s'intéresse particulièrement aux orbites quasi-polaires héliosynchrones. En effet, le satellite se déplace presque selon une longitude et conserve un angle constant entre le plan d'orbite et la direction Terre-Soleil.

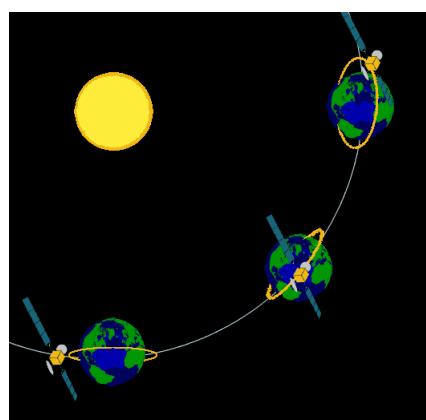


FIGURE 2.5 – Orbite Héliosynchrone

Ce type d'orbite est très intéressant pour l'imagerie spatiale car cela signifie que le satellite repasse à un point à la même heure de la journée, ce qui dans le cas d'un satellite d'observation nous permet de prendre des photographies avec une lumière ayant toujours le même angle d'incidence et donc un ensoleillement comparable.

Trace au Sol et Fauchée d'un Satellite

La trace au sol est la projection de la trajectoire décrite par un satellite à la surface du globe terrestre.

La fauchée d'un satellite est la surface au sol couverte par la prise de vue du capteur optique.

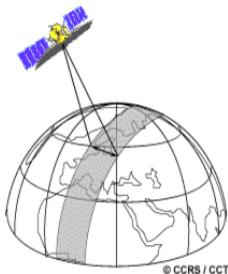


FIGURE 2.6 – Fauchée d'un satellite

Dans le cas d'une orbite quasi-polaire, la trace au sol recouvrira alors presque toute la surface de la Terre (sauf aux pôles), permettant de revisiter les zones de façon périodique et à la même d'heure grâce aux propriétés d'héliosynchronisme.

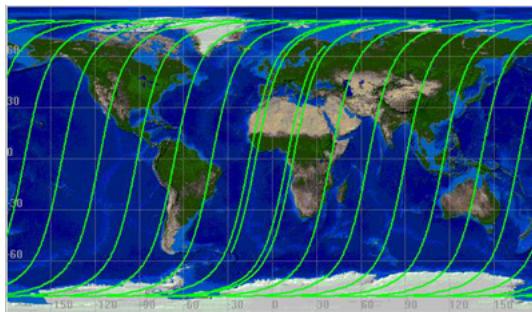


FIGURE 2.7 – Trace au sol des satellites SPOT sur 24h

2.2 Images Satellite

Après avoir détaillé les différents types de capteurs optiques et différentes orbites utilisées pour les satellites, nous allons à présent développer un peu plus en détail le produit, les *images satellite*.

2.2.1 Métadonnées

À chaque image satellite, on associe des métadonnées, c'est-à-dire des données externes, qui ne sont pas des *données images*, mais qui apportent des informations complémentaires à l'image.

Voici une liste non-exhaustive des informations habituellement contenues dans les métadonnées.

Le Système de Coordonnées

Un système de coordonnées est un référentiel permettant d'exprimer la position de points dans un espace, et par conséquent une position sur le globe terrestre. La Terre n'offre pas de surface géométriquement régulière, de ce fait on peut la représenter de différentes manières :

- Une sphère est une approximation grossière de la Terre, venant du fait que la Terre est aplatie aux pôles, ce qui est dû à sa rotation. Mais cette approximation ne suffit pas pour de la cartographie précise.
- Un ellipsoïde, une ellipse en trois dimensions, nous permet de modéliser cet aplatissement. C'est ce modèle qui est retenu pour la cartographie.

Le système géodésique le plus courant est WGS 84 (World Geodetic System 1984), car utilisé par le système GPS. Il définit une représentation de l'ellipsoïde terrestre, ce système est référencé en 2 dimensions, longitude et latitude, et en 3 dimensions en ajoutant l'altitude.

On peut aussi les exprimer en représentation plane, résultant d'une projection, parfois partielle, de l'ellipsoïde (en 3 dimensions) sur un plan (en 2 dimensions). On citera par exemple les projections de Mercator ou de Lambert.

Ces informations, décrivant le système de coordonnées, sont contenues dans le *Well-Known Text*, qui est une chaîne de caractères structurée résumant la définition du repère géographique. En voici un exemple :

```
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433],
    AUTHORITY["EPSG","4326"]]
```

Positionnement et Résolution de l'Image

Il est indispensable d'associer à toute image satellite une position dans l'espace. Une origine, généralement le coin supérieur gauche, est définie dans le système de référencement.

À cette origine, on associe la résolution spatiale de l'image, c'est-à-dire les dimensions d'un pixel dans l'espace projeté. Par exemple,

```
Origin = (32.50338839999999,15.54891119989999)
Pixel Size = (0.00000270000000,-0.00000270000000)
```

La seconde composante de la taille d'un pixel est ici négative, car l'image à pour origine le coin supérieur gauche, le parcours de l'image se fait donc en avançant dans les longitudes (sens Ouest-Est) mais en descendant les latitudes (sens Nord-Sud).

En ajoutant à ces connaissances la dimension de l'image en pixel, interpoler la position des autres coins de l'image. Ce calcul d'interpolation reste une approximation car il faut prendre en compte l'angle de visée du satellite par rapport au sol.

2.2.2 Compression et Formats de Fichier

Le format de fichier, aussi appelé *driver*, spécifie la façon dont les données sont écrites sur le disque de stockage. Il existe de nombreux formats de fichiers offrant des possibilités de compression avec ou sans pertes.

Un format très utilisé dans l'imagerie spatiale est le format GeoTIFF, ou GTiff, qui est un dérivé du format TIFF. Il est très utilisé de par sa flexibilité et le nombre de méthodes de compression existantes.

On citera aussi les formats :

- JPEG est le format le plus courant en photographie numérique, mais étant un format de compression avec pertes il est peu utilisé en imagerie spatiale.
- JPEG2000 est un format propriétaire, successeur annoncé du JPEG, il offre de grandes possibilités de compression utilisant des ondelettes (ratio de 1 : 50), mais la lecture/écriture sont très lentes dues aux opérations de compression/décompression.
- LUM est un format utilisé par le CNES en interne sur la chaîne de traitement des images et pour des simulations, c'est un fichier de format binaire accompagné de métadonnées.

2.2.3 Caractéristiques d'une Image Satellite

Nous allons à présent voir quelques propriétés concernant les images, ainsi que certaines particularités des images satellite.

Angle de visée

Pour les appareils photographiques classiques, l'angle de visée dépend de celui qui tient l'appareil, il en est de même pour les satellites.

L'angle de visée est variable, permettant de re-photographier une zone sans avoir à attendre une période orbitale entière, durant environ 26 jours pour les satellites de la famille SPOT.

Cependant pour pouvoir obtenir de "belles" images, afin de pouvoir les exploiter de façon optimale il faut re-projeter l'image pour se retrouver au *nadir*, le *nadir* étant le vecteur normal à la surface terrestre depuis le satellite. C'est donc le vecteur opposé du zénith.

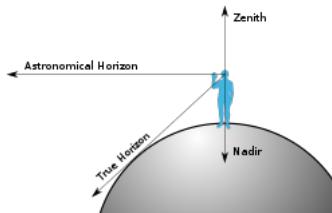


FIGURE 2.8 – Nadir, Zénith et Horizons

Le fait que l'angle de visée re-projeté soit pratiquement toujours le même permet d'unifier les traitements à appliquer sur l'image et simplifier son exploitation ultérieure.

Bandes et Fusion d'Images

Une image, et en particulier une image satellite, est constituée de bandes.

Chaque bande contient la réponse impulsionnelle de la scène sur une longueur d'onde donnée.

Voici les caractéristiques radiométriques du satellite Pléiades :

Couleur	Bandes Spectrales	Résolution
Bleu	430-550nm	2m
Vert	490-610nm	2m
Rouge	600-720nm	2m
Proche Infrarouge	750-950nm	2m
Panchromatique	480-830nm	0.5m

La bande panchromatique ne discrimine pas les couleurs, son domaine de sensibilité est similaire à la vision humaine, autour de 400-750nm. Ce sont des images en noir et blanc.

Comme nous pouvons le voir sur le tableau, toutes les bandes produites par le satellite Pléiades ne sont pas à la même résolution, les bandes *multispectrales* sont à 2m, tandis que la bande *panchromatique* est à 0.5m. Pourtant le produit après traitement est composé des 4 bandes couleurs échantillonnées à 0.5m.

Le traitement appliqué s'appelle la fusion d'images et repose sur :

1. La transformée en ondelettes de l'image *panchromatique haute résolution* en une image approximée, c'est-à-dire de plus basse résolution.
2. La substitution de l'image approximée par l'image *multispectrale*.
3. La transformée inverse en ondelettes de l'image *multispectrale* pour obtenir une image *multispectrale haute résolution*.

Prétraitement d'une Image Satellite

Un grand nombre de traitements interviennent dans la correction géométrique des images afin que le produit final puisse être superposable à une carte. Voici un récapitulatif des niveaux de traitements pour une image satellite de la famille SPOT :

- Niveau 1A : Correction des défauts radiométriques issus des écarts de sensibilité entre les détecteurs élémentaires.
- Niveau 1B : Égalisation radiométrique et traitements géométriques relatifs aux variations d'altitude du satellite, à la rotation et à la courbure de la Terre, aux variations de l'angle de visée, etc.

- Niveau 2A : Projection de l'image dans le référentiel WGS 84, la localisation d'un pixel est précise à 300m.
- Niveau 2B : Correction géométriques : utilisation de points d'appuis au sol, de la modélisation de la dynamique de vol, etc.
- Niveau 3 : Produit appelé *orthoimage*, où l'erreur de localisation est encore diminuée grâce à des modèles numériques d'élévation pour ajuster le relief, l'erreur de localisation est de 10m pour des images ayant une résolution de 10m.

Chapitre 3

L'Apprentissage Machine

L'apprentissage machine permet d'accomplir des tâches traditionnellement compliquées pour les ordinateurs, mais considérées simples 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 :

- $E = \{1, \dots, k\}$, 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$, 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 où 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 faut 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 défis majeurs 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 :

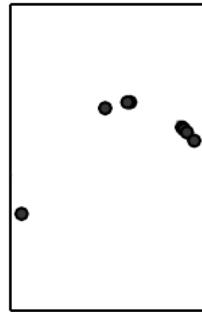


FIGURE 3.1 – Nuage de 10 points à ajuster

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.

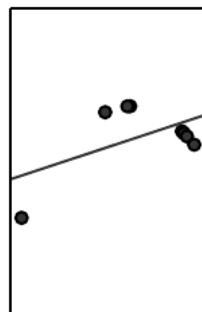


FIGURE 3.2 – Cas de sous-apprentissage

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.

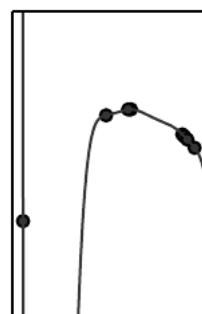


FIGURE 3.3 – Cas de sur-apprentissage

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.

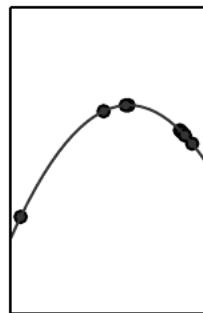


FIGURE 3.4 – Capacité du modèle adaptée

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.

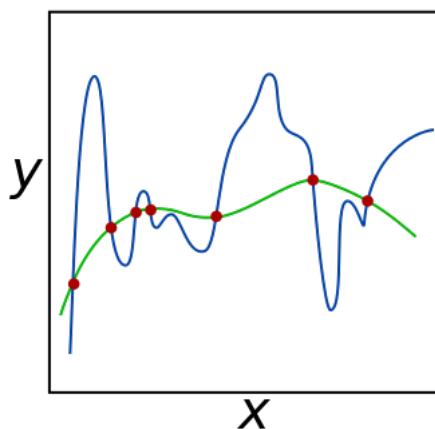


FIGURE 3.5 – Dilemme du choix entre deux modèles

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 $f(\theta) = \sum_{i=1}^q \theta_i^2$ 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 *regularisation* est employé pour toute 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 faut 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 :

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

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

$$P(x) \text{ est le vecteur de polynômes } P(x_i). \text{ Donc } P(x) = \begin{pmatrix} 1 & x_1 & \cdots & x_1^q \\ \vdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ 1 & x_n & \cdots & x_n^q \end{pmatrix} \begin{pmatrix} b \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} = X\theta$$

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 $\frac{\partial x^T A x}{\partial x} = 2Ax$)

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

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 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 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,

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

Algorithme 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.

Dans un second temps, nous verrons 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* 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 $f : \mathbb{R}^p \rightarrow \mathbb{R}$, paramétrée par $\theta = (b, w) = (b, w_1, \dots, w_p)$.

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

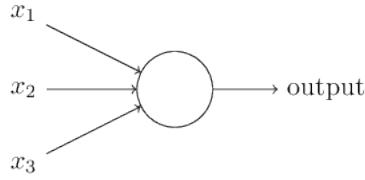


FIGURE 4.1 – Schéma de représentation d'un Neurone

Cette application est la composée de deux fonctions distinctes :

- Une fonction d'*agrégation* $h : \mathbb{R}^p \rightarrow \mathbb{R}$, combinant l'entrée x avec les poids w et le biais b . Dans le cas d'un neurone *totalelement 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* $g : \mathbb{R} \rightarrow \mathbb{R}$, *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 :

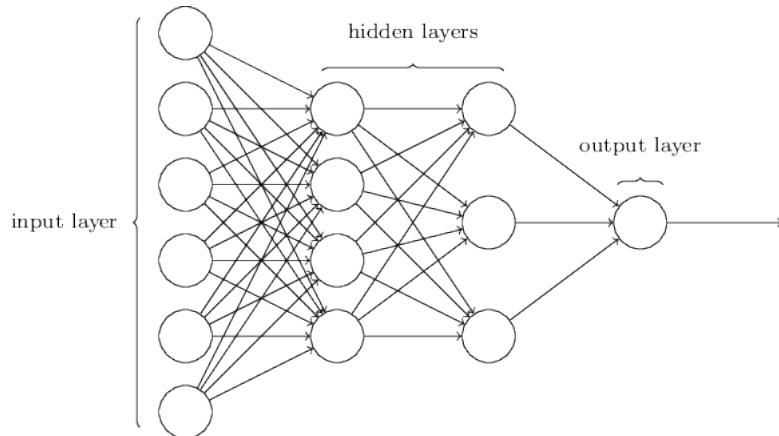


FIGURE 4.2 – Schéma pédagogique de représentation d'un Réseau de Neurones

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 :

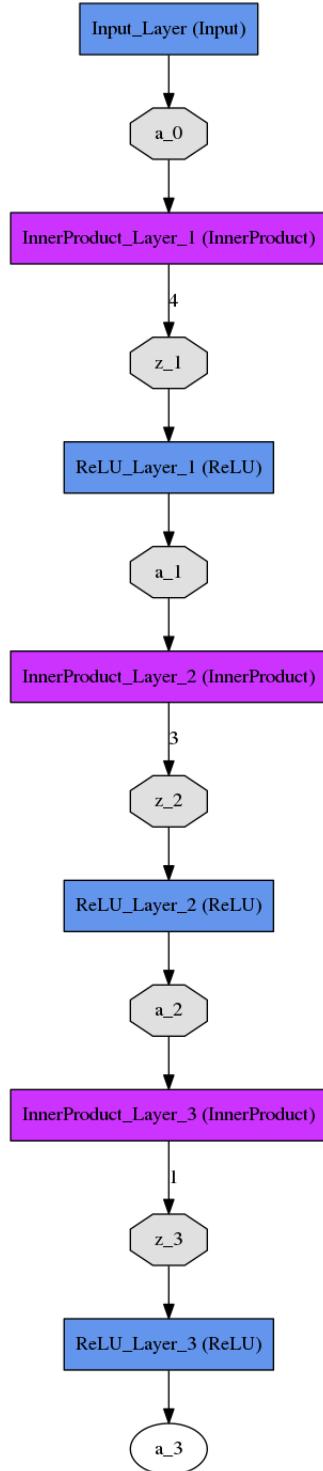


FIGURE 4.3 – Schéma courant de représentation d'un Réseau de Neurones

Ainsi, pour décrire un réseau de neurones, nous considérons 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 $\theta = (b, w) = (b, w_1, \dots, w_p)$ 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 $S = (S_1, S_2, \dots, S_L)$ 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 indiquée 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 un 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 notée ${}^kz^j$ 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 $S = (S_1, S_2, \dots, S_L)$. 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 :

$$\begin{aligned} {}^kz &= ({}^kW)^T({}^{k-1}a) + {}^kb \\ {}^ka &= g({}^kz) \end{aligned}$$

Ainsi,

Algorithm 4 Algorithme matriciel de Propagation

Require: Un réseau de neurones de L couches, défini par $S = (S_1, S_2, \dots, S_L)$. 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 . $\{\alpha_k$ 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}

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 ${}^k W_i^j$ et des biais ${}^k b^j$.

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** Θ .

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,

$$\begin{cases} \frac{\partial C}{\partial {}^L W_i^j} = \frac{\partial C}{\partial {}^L a^j} \times \frac{\partial {}^L a^j}{\partial {}^L z^j} \times \frac{\partial {}^L z^j}{\partial {}^L W_i^j} \\ \frac{\partial C}{\partial {}^L b^j} = \frac{\partial C}{\partial {}^L a^j} \times \frac{\partial {}^L a^j}{\partial {}^L z^j} \times \frac{\partial {}^L z^j}{\partial {}^L b^j} \end{cases}$$

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 $\frac{\partial {}^L z^j}{\partial {}^L b^j} = 1$, 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 rétro-propagation.

Finalement, nous obtenons,

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

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,

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

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 $\frac{\partial {}^{L-1} z^j}{\partial {}^{L-1} b^j} = 1$, 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 :

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

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 ,

$$\begin{cases} \frac{\partial C}{\partial^L W_i^j} = {}^L \delta^j \times \frac{\partial^L z^j}{\partial^L W_i^j} = {}^L \delta^j \times {}^{L-1} a^i \\ \frac{\partial C}{\partial^L b^j} = {}^L \delta^j \times \frac{\partial^L z^j}{\partial^L b^j} = {}^L \delta^j \end{cases}$$

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 $L-1$,

$$\begin{cases} \frac{\partial C}{\partial^{L-1} W_i^j} = {}^{L-1} \delta^j \times \frac{\partial^{L-1} z^j}{\partial^{L-1} W_i^j} = {}^{L-1} \delta^j \times {}^{L-2} a^i \\ \frac{\partial C}{\partial^{L-1} b^j} = {}^{L-1} \delta^j \times \frac{\partial^{L-1} z^j}{\partial^{L-1} b^j} = {}^{L-1} \delta^j \end{cases}$$

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 $\forall k = 1..L$,

$$\begin{cases} \frac{\partial C}{\partial^k W_i^j} = {}^k \delta^j \times \frac{\partial^k z^j}{\partial^k W_i^j} = {}^k \delta^j \times {}^{k-1} a^i \\ \frac{\partial C}{\partial^k b^j} = {}^k \delta^j \times \frac{\partial^k z^j}{\partial^k b^j} = {}^k \delta^j \end{cases}$$

Avec

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

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 $S = (S_1, S_2, \dots, S_L)$. 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
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 à des librairies de BLAS.

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 $S = (S_1, S_2, \dots, S_L)$. 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.}

$${}^L\delta = \nabla_{L-1}a C$$

$$\nabla_{Lb}C = {}^L\delta$$

$$\nabla_{LW}C = {}^L\delta \times {}^{L-1}a^T$$

for $k = L - 1$ **to** 1 **do**

{Calcul du gradient sur la k -ème couche.}

$${}^k\delta = ((\nabla_k a^{k+1} z)^T \times {}^{k+1}\delta) \odot \nabla_k z^k a$$

$$\nabla_{kb}C = {}^k\delta$$

$$\nabla_{kW}C = {}^k\delta \times {}^{k-1}a^T$$

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 aux dépends 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 Prématué

Pour un modèle avec une capacité suffisante pour sur-apprendre, on observe généralement que tandis 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.

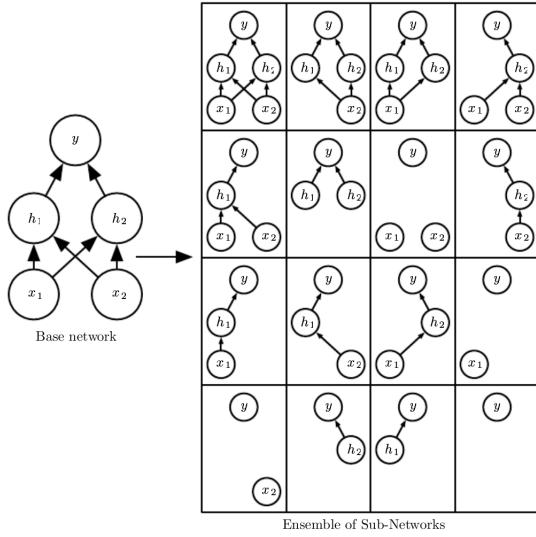


FIGURE 4.4 – Exemple d’application de la méthode de *Dropout*

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 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 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 vues 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 :

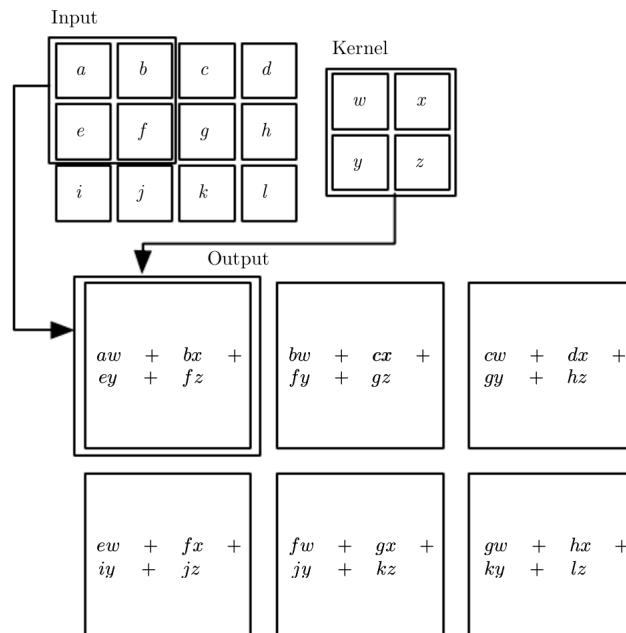


FIGURE 4.5 – Exemple de convolution entre une image I , (8, 3) et un noyau K , (2, 2)

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

Nous parlerons ici de façon plus générale de convolution entre 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 par la multiplication matricielle du vecteur d'entrée ${}^{k-1}a$ de dimension m , par la matrice kW de dimension $m \times n$, la sortie du neurone ${}^{k-1}a$ sera donc de dimension n .

Pour rappel :

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

On voit donc qu'ici l'interaction, ou connectivité, est totale puisque toutes les entrées ${}^{k-1}a^j$ sont connectées par les poids ${}^kW_i^j$ 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 kW , 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 :

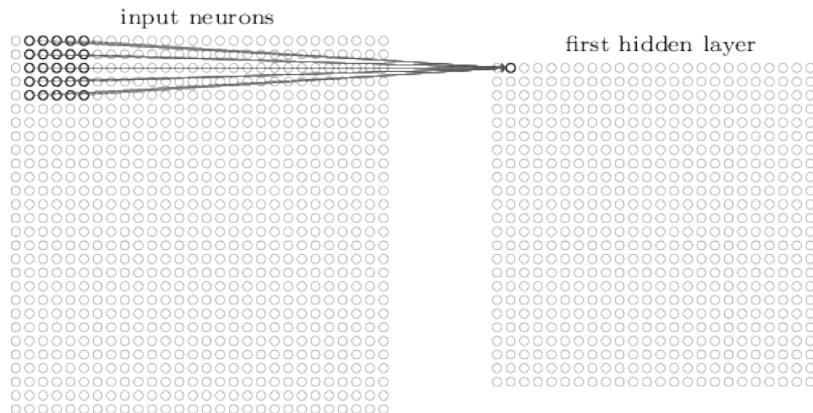


FIGURE 4.6 – Représentation de l'interaction partielle

Le Partage des Paramètres et Invariance

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 qu'au lieu d'apprendre pour chaque élément du vecteur d'entrée un paramètre associé, un filtre unique, *composé d'un petit nombre de paramètre*, sera appris en rapport au vecteur d'entrée.

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 souhaité :

1. 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 :

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

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

2. 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 .

3. 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 sommes suivants,

- $l \in \llbracket 0, C_K^{in} \rrbracket$
- $m \in \llbracket 0, H_K \rrbracket$ et $(m + i) \in \llbracket 0, H_I \rrbracket$
- $n \in \llbracket 0, W_K \rrbracket$ et $(n + j) \in \llbracket 0, W_I \rrbracket$

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) ,

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,

— $l \in \llbracket 0, C_K^{in} \rrbracket$

— $m \in \llbracket 0, H_K \rrbracket$ et $(m + i \times s_H) \in \llbracket 0, H_{I'} \rrbracket$

— $n \in \llbracket 0, W_K \rrbracket$ et $(n + j \times s_W) \in \llbracket 0, W_{I'} \rrbracket$

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 $(3 \times 256 \times 256) = (196, 608)$.

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. Chaque variable est donc dépendante de ses voisins. La deuxième est plus beaucoup 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.

Pooling d'Information

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 de pooling 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.

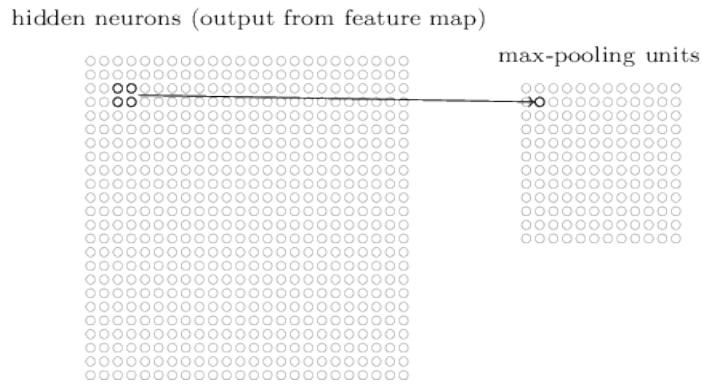


FIGURE 4.8 – *Max-pooling* sur une fenêtre (2, 2)

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 “malléable”, 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

Les applications basées sur les réseaux de neurones occupent une place de plus en plus importante dans la société, le champ d’application est très large, on les retrouve notamment dans des outils de reconnaissance vocale, par exemple : Siri, Cortana ; dans des outils de vision par ordinateur, utilisés par exemple dans les voitures autonomes ; ou bien encore dans la création d’intelligences artificielles, liés à des jeux de réflexion, comme Deep Mind au jeu de Go, ou encore des agents conversationnels. Pour ces applications, les performances des réseaux de neurones sont comparables voire supérieures à celles d’un humain.

Nous allons dans cette partie présenter quelques tâches liées à la vision par ordinateur, domaine dans lequel l’influence des réseaux de neurones est grande.

4.3.1 Classification

La classification est sans doute le problème le plus simple en matière de vision, il s’agit de prédire l’appartenance d’une image à une classe, généralement liée aux objets liés dans l’image.

Datasets

Un des premiers *datasets* d’images pour la classification est le MNIST, composé de 70000 images noir et blanc de 28×28 pixels, chacune contenant un chiffre manuscrit.



FIGURE 4.9 – Données issues de MNIST

Même si l'intérêt à reconnaître les chiffres est limité, ce *dataset* a longtemps servi au niveau académique de *benchmark* pour comparer les différents modèles.

On citera aussi les *datasets* CIFAR-10, CIFAR-100, et ImageNet. CIFAR-10 et CIFAR-100 sont composés chacun de 60000 images RGB de 32×32 pixels, réparties en 10 classes pour le premier et 100 classes pour le second.



FIGURE 4.10 – Données issues de CIFAR-10

ImageNet contient une base de classification d'images contenant plus de 10 millions d'URLs d'images de taille variable réparties en 1000 classes. Les images sont majoritairement issues de sites comme Flickr. Ce *dataset*, créé en 2009, fait l'objet d'une compétition depuis 2010, qui a permis de réaliser des progrès impressionnantes en traitement d'images.

Architectures de Modèles

En 2011, l'erreur de classification était de 25%. En 2012, le réseau de neurones convolutifs AlexNet, composé de 8 couches, atteignait alors 16% d'erreur. En 2014, GoogLeNet (22 couches), suivi de près par VGG-16 (16 couches), remportait la compétition avec une erreur de 6.7% (respectivement 7.3%). En 2015, ResNet, un réseau de 152 couches, atteignait alors 3.6%. En moyenne, un humain non-entraîné réalise environ 10% d'erreur et un humain entraîné 5%.

4.3.2 Détection d'Objets

La détection d'objets consiste à prédire la classe et l'emplacement d'un ou plusieurs objets dans une image. L'emplacement d'un objet est généralement défini par une *bounding box*, ou région d'intérêt, représentée par un rectangle entourant l'objet.

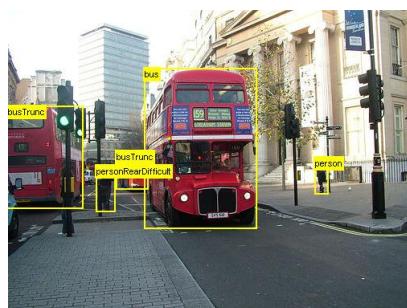


FIGURE 4.11 – Image et *bounding boxes* issues de Pascal VOC

Datasets

Le *dataset* Pascal VOC est composé de 11530 images contenant au total 27450 régions d'intérêt décrivant 20 classes.

ImageNet possède aussi une base de détection d'objets contenant environ 450000 images contenant 475000 objets référencés répartis en 200 classes.

Architectures de Modèles

Deux architectures remarquables sont les réseaux de type R-CNN et YOLO, l'une par ses performances exceptionnelles, l'autre par le compromis réalisé entre performances et vitesse d'exécution.

Les architectures de type R-CNN se caractérisent par un schéma de fonctionnement divisé en deux parties, la première étant un réseau de neurones assurant la localisation potentielle d'objets, la seconde étant un autre réseau de neurones dédié à la classification de chacune des zones potentielles.

Les architectures de type YOLO ont à l'inverse la particularité d'effectuer en parallèle la localisation et la classification des objets, proposant ainsi une grande vitesse d'exécution.

4.3.3 Segmentation d'Images

Comme pour la détection d'objets, la segmentation vise à identifier les objets appartenant à une classe et à les localiser sur l'image. La différence apparaît sur la sortie produite, dans le cas de la segmentation, cela consiste à prédire, pour chaque pixel de l'image, sa classe. La segmentation répond à des besoins de précision plus élevés que la détection d'objets, du fait de la prédiction d'un masque de la taille de l'image d'entrée.



FIGURE 4.12 – Image et masque de classes issus de Pascal VOC

Un des avantages de la segmentation d'images par rapport à la détection d'objets, outre le gain potentiel de précision, est qu'il est possible à partir d'un masque de segmentation de produire des *bounding boxes* en utilisant la polygonisation des masques. L'opération inverse n'est pas vraiment intéressante puisque les polygones sont des rectangles.

Datasets

Pascal VOC possède aussi une base de segmentation d'images contenant 2913 images décrivant 20 classes. Microsoft COCO (Common Objects in Context) contient 328000 images décrivant 91 classes.

Ces deux *datasets* possèdent la particularité de segmenter les images d'une part selon la classe mais aussi en fonction de l'instance, permettant ainsi de différencier deux objets de même nature au sein d'une image.

Architectures de Modèles

Deux architectures remarquables sont les réseaux de type FCN et DeconvNet, elles utilisent toutes les deux le principe de décomposer l'image en *features* par convolutions successives, puis de reconstituer l'image en image segmentée en appliquant plusieurs couches de déconvolutions.

Les architectures de type FCN reconstituent l'image en combinant les *features* dé-convolutées avec les *features* après *pooling*.

Les architectures de type DeconvNet reconstituent l'image en utilisant des couches d'*unpooling*, inversant l'effet des couches de *pooling*. Nous reviendront en détail leur fonctionnement dans la suite.

Chapitre 5

L'Apprentissage Profond appliqué à l'Imagerie Satellite

5.1 Cartographie Automatisée par Segmentation de Classes

La cartographie de la Terre a toujours été une des priorités de l'humain. Les cartes ont surtout servi dans un premier temps à la navigation maritime et à une utilisation militaire. Elles ont tout le temps fait l'objet de convoitise, à tel point qu'au Portugal au XVI ème siècle le transfert de cartes à un étranger était passible de la peine de mort. L'aéronautique a ensuite permis d'affiner les cartes, mais sur des volumes toujours restreints. L'arrivée des satellites de télédétection permet alors d'acquérir quasiment en temps réel des images de la Terre.

Cette masse de données à traiter conduit à la formation de la géomatique, discipline consistant en trois activités principales la collecte, le traitement et la diffusion des données géographiques.

Il existe plusieurs types de cartographie, parmi lesquelles on retrouve :

- La cartographie physique, ou topographie, répertoriant les reliefs du terrain.
- La cartographie humaine, mesurant des indices sociologiques.
- La cartographie administrative, répertoriant les frontières entre les pays.
- Les cartographies de type historique, biologique, etc.

Pour manipuler ces données, d'échelle parfois mondiale, on utilise des Systèmes d'Information Géographique, permettant de collecter, traiter, analyser, et enfin de présenter ces données.

Dans ce rapport, nous nous intéresserons principalement à la cartographie administrative et plus particulièrement à la cartographie d'infrastructures, telles que les routes et les bâtiments.

Aujourd'hui la majorité des données concernant les infrastructures ont été collectées "à la main". Dans le cas d'OpenStreetMap, la base de données géographiques est remplie par les utilisateurs, soit en relevant des positions GPS en allant directement sur le terrain, soit en cartographiant depuis des images spatiales. Ce processus est très long et ne permet pas encore une cartographie globale de toutes les infrastructures sur Terre. Toutefois, certaines administrations publiques ont partagé des bases de données, notamment l'intégralité du cadastre français mis à disposition par la Direction générale des Finances publiques ou encore la base de données TIGER, concernant les routes, divisions administratives et points d'eau des États-Unis. Ces initiatives ont permis de réaliser de grandes avancées dans le projet, mais ne permettent pas de cartographier toute la surface du globe.

Avec les progrès récents de l'apprentissage profond, permettant d'obtenir d'excellentes performances en vision par ordinateur, et le besoin immense d'automatiser la cartographie de la Terre, il semble intéressant d'utiliser des méthodes telle que la segmentation d'images pour identifier précisément des infrastructures.

Cela permettrait d'établir de façon automatisée des cartes détaillées de la surface du globe, permettant une utilisation civile, militaire ou encore humanitaire, permettant ainsi de comparer les cartes avant et après une catastrophe naturelle.

5.1.1 Présentation des Données

SpaceNet est une base de données d'images satellite très haute résolution pour la détection et segmentation de bâtiments.

Elle est issue de la collaboration entre CosmiQ Works, DigitalGlobe et NVIDIA. C'est une des premières bases de données d'image très haute résolution disponible au grand public. Les données sont hébergées par Amazon, sur les serveurs AWS S3 (Amazon Web Services Simple Storage Service).

La base de données SpaceNet actuelle fut construite en deux temps, lors des deux éditions du SpaceNet Challenge, elle couvre 5 zones géographiques :

- Rio de Janeiro, Brésil.
- Las Vegas, États-Unis.
- Paris, France.
- Shanghai, Chine.
- Khartoum, Soudan.

La première édition du SpaceNet Challenge ne portait que sur Rio de Janeiro, les images provenant du satellite WorldView-2.

La seconde édition du SpaceNet Challenge portait sur Las Vegas, Paris, Shanghai et Khartoum, les images provenant du satellite WorldView-3.

Images

Les deux satellites proviennent de la famille WorldView, dont les images sont distribuées par DigitalGlobe.

WorldView-2 fut envoyé en orbite le 8 octobre 2009, tandis que WorldView-3 fut lancé le 13 août 2014. Les deux satellites ont des caractéristiques similaires, mais se différencient sur la résolution au sol des capteurs optiques :

Résolution	WorldView-2	WorldView-3
Bande Panchromatique	0.46m	0.31m
Bandes Multispectrales	1.84m	1.24m

Les satellites WorldView-2 et WorldView-3 possèdent un capteur panchromatique et 8 capteurs multispectraux, qui décrivent les couleurs suivantes :

Panchromatique	Côtier	Bleu	Vert	Jaune
450-800nm	400-450nm	450-510nm	510-580nm	585-625nm
Rouge	Rouge Frontière	Proche Infrarouge 1	Proche Infrarouge 2	
630-690nm	705-745nm	775-895nm	860-1040nm	

Nous n'utiliserons que les données issues de la deuxième édition du SpaceNet Challenge, car les images multispectrales de la première édition n'ont pas été fusionnées avec la bande panchromatique laissant la résolution à 1.84m.

Les images multispectrales de la seconde édition font 650×650 pixels et ont une résolution au sol de 0.31m et sont encodées sur 16 bits.



FIGURE 5.1 – Images (RGB, 8-bits, Correction [2 – 98]%) extraites des zones Vegas, Paris, Shanghai et Khartoum

Voici la distribution des images en fonction des zones géographiques :

Villes	Las Vegas	Paris	Shanghai	Khartoum	Total
Images	3851	1148	4582	1012	10593
Taille	26.1Go	7.8Go	31.0Go	6.8Go	71.7Go

Labels

Labels issus de SpaceNet Le SpaceNet Challenge consiste à détecter les bâtiments visibles depuis les images satellites.

La labellisation des données a été réalisée par l'Amazon Mechanical Turk, une plateforme de crowdsourcing spécialisée dans la constitution de *datasets* pour des problèmes d'intelligence artificielle.

À chaque image du *dataset* correspond un fichier GeoJSON contenant l'empreinte au sol des différents bâtiments, sous forme de polygones. Ces polygones sont représentés par des ensembles de coordonnées géodésiques, c'est-à-dire sous forme d'un couple latitude et longitude.

Voici un extrait de fichier GeoJSON issu de SpaceNet :

```
{
  "type": "FeatureCollection",
  "crs": {
    "type": "name",
    "properties": {
      "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
    }
  },
  "features": [
    {
      "type": "Feature",
      "properties": {
        "id": "00000000-0000-0000-0000-000000000000"
      }
    }
  ]
}
```

```

    "OBJECTID": 0,
    "FID_VEGAS_": 0,
    "Id": 0,
    "FID_Vegas": 0,
    "Name": "None",
    "AREA": 0.000000,
    "Shape_Leng": 0.000000,
    "Shape_Le_1": 0.000000,
    "SISL": 0.000000,
    "OBJECTID_1": 0,
    "Shape_Le_2": 0.000000,
    "Shape_Le_3": 0.000811,
    "Shape_Area": 0.000000,
    "partialBuilding": 1.000000,
    "partialDec": 0.556769
  },
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          [-115.302252599989529, 36.208218543364502, 0.0],
          [-115.302260806999982, 36.208218603000034, 0.0],
          [-115.302261041999941, 36.208197527000038, 0.0]
          [...]
        ]
      ]
    ]
  }
}
[...]
}

```

Le *dataset* SpaceNet ne contient que des polygones. Il existe, pour le format GeoJSON, plusieurs types d'objets géométriques simples :

- *Point*, représentant un point.
- *LineString*, un ensemble ouvert ou fermé de points.
- *Polygon*, une surface délimitée par un ensemble fermé de points. Par exemple, un triangle ABC sera représenté par ABCA.

Labels issus d'OpenStreetMap Les labels de SpaceNet, produit par l'Amazon Mechanical Turk, ne localisent que les bâtiments dans les images. Cependant d'autres informations directement observables depuis les images existent, comme par exemple les routes, ou encore les cours d'eau. C'est pourquoi il a été jugé intéressant de compléter les labels originels de SpaceNet par des labels issus d'OpenStreetMap. Ces labels sont disponibles au format GeoJSON et il est facile de les récupérer depuis l'API d'OpenStreetMap. Cependant ces labels ne sont pas de même qualité que ceux provenant du *dataset* SpaceNet, il est important de garder à l'esprit que ces labels sont issus d'un projet communautaire, ils peuvent donc faire l'objet d'imprécisions, par exemple concernant la localisation des éléments géographique, potentiellement due aux imprécisions de GPS amateurs, ou parfois tout simplement manquants, car personne n'a encore répertorié la zone géographique en question.

Rasterisation des Labels Les labels issus de SpaceNet ou d'OpenStreetMap sont proposés sous forme vectorielle et donc par conséquent utilisables pour de la détection d'objets. Les labels vectoriels, au même titre que les images vectorielles, possèdent l'avantage d'être quasiment invariants à la résolution de l'image. Mais pour réaliser de la segmentation d'images, il nous faut pouvoir produire un label au format matriciel de même dimensions que l'image d'entrée. L'opération permettant de produire une

image matricielle à partir d'une image vectorielle s'appelle rasterisation. Bien que l'opération ne soit pas bijective, car irréversible à cause de la résolution finie de l'image matricielle. Il existe une opération effectuant la transformation inverse, on parlera alors de polygonisation. L'opération de rasterisation consiste à produire, à partir des données vectorielles, ainsi que des dimensions et la résolution de l'image souhaitée, une image matricielle où chaque pixel prend une valeur selon si il est contenu ou non dans un polygone.

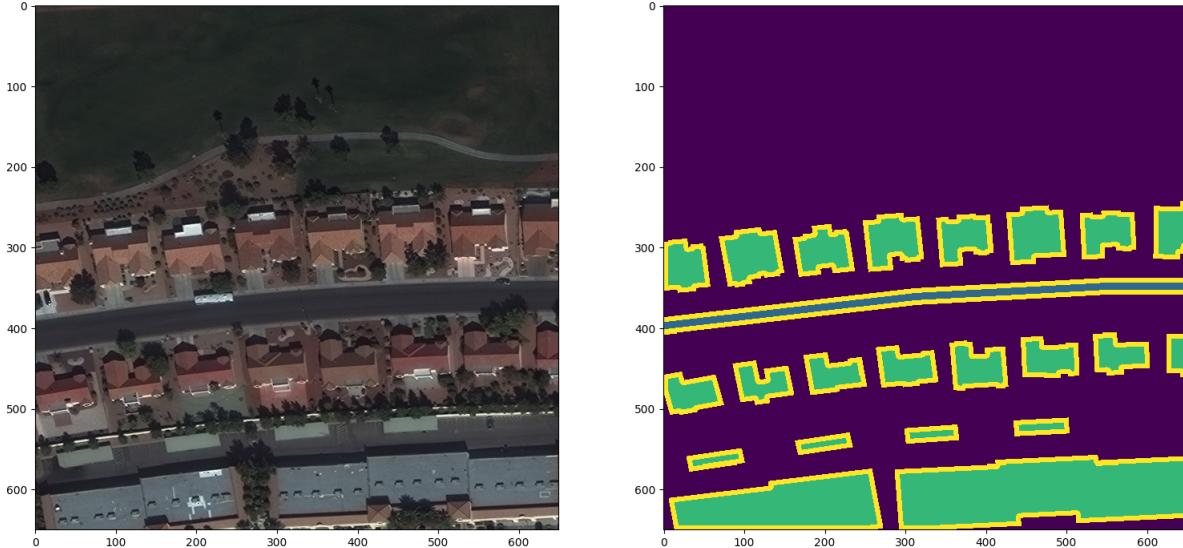


FIGURE 5.2 – Image et Masque de Segmentation produit par couplage des données SpaceNet et OpenStreetMap (Las Vegas)

Les labels d'OpenStreetMap sont très libres au niveau des informations qu'ils contiennent, le champ *properties*, donnant des informations sur l'objet décrit, n'est pas systématiquement présent ou partiellement renseigné. Il est donc difficile de catégoriser chaque donnée.

Nous avons choisi dans le cas où les labels de SpaceNet sont combinés avec ceux provenant d'OpenStreetMap, que seuls les *Polygons* et les *LineStrings* non fermées seront rasterisés.

Cela dégrade quelque peu la qualité des données car nous considérons que tous les *Polygons* représentent des bâtiments et que toutes les *LineStrings* représentent des routes. Même si les *Polygons* représentent quasiment tout le temps des bâtiments, les *LineStrings* représentent parfois des cours d'eau, ou encore des délimitations administratives.

De plus, les *LineStrings* forment des courbes, elles ont donc une aire nulle, or les routes sont des surfaces, il a donc fallu choisir arbitrairement la largeur d'une route à rasteriser. Nous avons choisi 3 mètres afin d'être sûr de ne pas empiéter sur des bâtiments alentours.

C'est pourquoi il faudra prendre en compte ces erreurs contenues dans le *dataset* au moment d'analyser les résultats des différents modèles.

Le contour des objets rasterisés, ici en jaune, n'appartient pas à la même classe que son objet, ceci est fait pour pallier aux erreurs de localisation des objets. En effet un pixel faisant parti du contour d'un objet ne sera pas évalué lors du calcul de la perte et de la précision, cela permet aux modèles de ne pas se focaliser sur les détails mais plus sur le cœur de l'objet.

5.1.2 Calcul GPU et Framework Deep Learning

A FAIRE

5.1.3 Expériences

Nous souhaitons présenter deux expériences ayant produit les résultats les plus significatifs.

La première expérience couvre les quatre zones géographiques, c'est-à-dire Las Vegas, Paris, Shanghai et Khartoum, et utilise uniquement les labels provenant de SpaceNet. La segmentation réalisée est binaire, car cela revient à prédire pour chaque pixel de l'image son appartenance à un bâtiment ou non.

La seconde expérience exploite uniquement les données de la région de Las Vegas, et utilise des labels provenant de SpaceNet et d'OpenStreetMap. La segmentation sera multi-classes, il faudra en effet prédire pour chaque pixel de l'image son appartenance à un bâtiment, ou une route, ou bien d'aucune de ces deux classes.

Avant de présenter chacune des expériences et de comparer les modèles utilisés, nous allons d'abord présenter l'architecture ayant inspiré ces modèles, nommée DeconvNet.

DeconvNet

DeconvNet est un réseau de neurones créé par H.Noh, en 2015, de l'université POSTECH, Corée du Sud.

Nous allons d'abord décrire son architecture, puis discuter des avantages et inconvénients de celle-ci.

DeconvNet est un réseau convolutif introduisant un nouveau type de couche, l'*unpooling*, faisant la particularité de ce modèle.

DeconvNet est un réseau de neurones composé de deux parties, la première est composée de couches de convolution et de *pooling*, décomposant l'image en une *map de features*; et la seconde est composée de couches de déconvolution et d'*unpooling*, construisant à partir de la *map de features* l'image segmentée.

La première partie n'est autre que le réseau de neurones VGG-16, classé second lors de l'ILSVRC 2014. La seconde partie peut être vu comme le symétrique de la première reconstruisant l'image segmentée.

Unpooling Le principe d'une couche de *max pooling* est de filtrer le bruit tout en réduisant la dimension des données. Cette réduction de la dimension est souvent de nature spatiale, créant une perte d'informations spatiales. Pour pallier à cela, H.Noh a créé le concept de couche d'*unpooling*, produisant l'effet quasi-inverse. L'idée est la suivante, si lors de l'opération de *pooling*, nous retenons les indices des activations maximales, alors nous pouvons ensuite reconstruire une *carte à trous* des activations dans la dimension originelle. Voici un schéma décrivant l'opération d'*unpooling* :

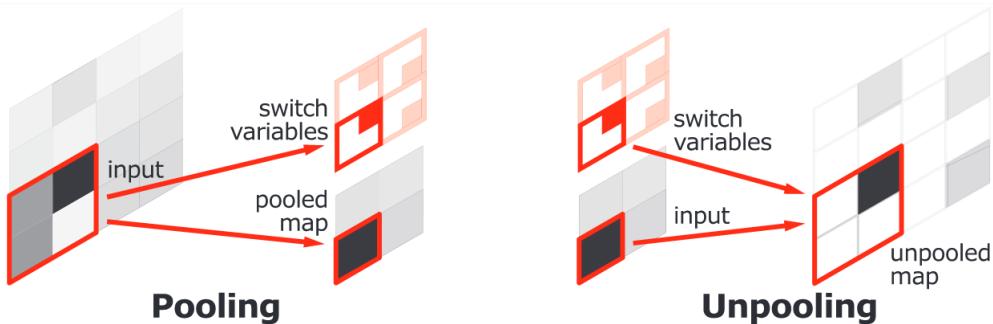


FIGURE 5.3 – Couches de *pooling* et d'*unpooling*

Comme on peut le voir dans la couche de *pooling*, l'activation la plus élevée, ici la plus sombre, est transmise dans la *map de features*, ainsi que sa position dans la *map des indices*. Alors lors de l'*unpooling*, nous réutiliserons la *map des indices* pour recréer la *map de features* à ses dimensions précédentes.

Déconvolution Une couche de convolution effectue un produit matriciel entre un noyau et une fenêtre d'une *map de features*, qui grossièrement dit, revient calculer une moyenne pondérée sur un ensemble de valeurs spatialement proches. Elle permet ainsi de calculer de nouvelles *features* à partir d'une *map de features* existante. La déconvolution effectue l'opération inverse, c'est-à-dire de partir

d'une *feature* pour produire une *map de features*. Voici un schéma décrivant la relation entre une convolution et une déconvolution :

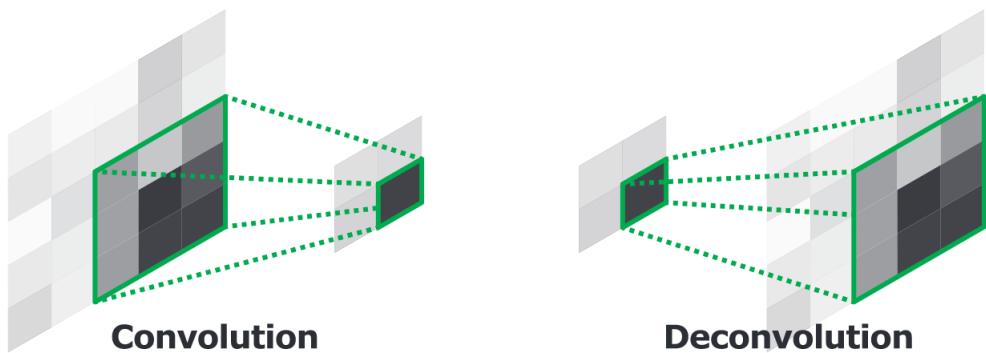


FIGURE 5.4 – Couches de convolution et de déconvolution

Comme on peut le voir, la couche de convolution agrège des entrées voisines par une moyenne pondérée qui sera transmise à la couche suivante. La couche de déconvolution réalise l'opération inverse, à partir d'une seule entrée elle reconstitue le voisinage d'entrées.

Architecture Après avoir expliqué le fonctionnement des couches de déconvolution et d'*unpooling*, nous allons présenter l'architecture de DeconvNet et expliquer certaines particularités.

Voici le détail de l'architecture de DeconvNet, en omettant les couches de *Batch Normalization* et d'activation ReLU présentes après chaque couche de convolution :

Couche	Noyau	Features	Stride	Pad	Dimensions en sortie
Input	-	-	-	-	(4, 224, 224)
conv1_1	(3, 3)	64	1	1	(64, 224, 224)
conv1_2	(3, 3)	64	1	1	(64, 224, 224)
pool1	(2, 2)	-	2	0	(64, 112, 112)
conv2_1	(3, 3)	128	1	1	(128, 112, 112)
conv2_2	(3, 3)	128	1	1	(128, 112, 112)
pool2	(2, 2)	-	2	0	(128, 56, 56)
conv3_1	(3, 3)	256	1	1	(256, 56, 56)
conv3_2	(3, 3)	256	1	1	(256, 56, 56)
conv3_3	(3, 3)	256	1	1	(256, 56, 56)
pool3	(2, 2)	-	2	0	(256, 28, 28)
conv4_1	(3, 3)	512	1	1	(512, 28, 28)
conv4_2	(3, 3)	512	1	1	(512, 28, 28)
conv4_3	(3, 3)	512	1	1	(512, 28, 28)
pool4	(2, 2)	-	2	0	(512, 14, 14)
conv5_1	(3, 3)	512	1	1	(512, 14, 14)
conv5_2	(3, 3)	512	1	1	(512, 14, 14)
conv5_3	(3, 3)	512	1	1	(512, 14, 14)
pool5	(2, 2)	-	2	0	(512, 7, 7)
fc6	(7, 7)	4096	1	0	(4096, 1, 1)
fc7	(1, 1)	4096	1	0	(4096, 1, 1)
deconv-fc6	(7, 7)	512	1	0	(512, 7, 7)
unpool5	(2, 2)	-	2	0	(512, 14, 14)
deconv5_1	(3, 3)	512	1	1	(512, 14, 14)
deconv5_2	(3, 3)	512	1	1	(512, 14, 14)
deconv5_3	(3, 3)	512	1	1	(512, 14, 14)
unpool4	(2, 2)	-	2	0	(512, 28, 28)
deconv4_1	(3, 3)	512	1	1	(512, 28, 28)
deconv4_2	(3, 3)	512	1	1	(512, 28, 28)
deconv4_3	(3, 3)	256	1	1	(256, 28, 28)
unpool3	(2, 2)	-	2	0	(256, 56, 56)
deconv3_1	(3, 3)	256	1	1	(256, 56, 56)
deconv3_2	(3, 3)	256	1	1	(256, 56, 56)
deconv3_3	(3, 3)	128	1	1	(128, 56, 56)
unpool2	(2, 2)	-	2	0	(128, 112, 112)
deconv2_1	(3, 3)	128	1	1	(128, 112, 112)
deconv2_2	(3, 3)	64	1	1	(64, 112, 112)
unpool1	(2, 2)	-	2	0	(64, 224, 224)
deconv1_1	(3, 3)	64	1	1	(64, 224, 224)
deconv1_2	(3, 3)	64	1	1	(64, 224, 224)
Output	(1, 1)	X	1	0	(X, 224, 224)

On peut tout de suite remarquer que le réseau est symétrique, la couche *fc7* étant le point central de cette symétrie.

Comme évoqué précédemment, l'idée est dans un premier temps de décomposer l'image en *map of features* par convolutions successives (et de *pooling*), puis dans un second temps de reconstituer une version segmentée de l'image par déconvolution (et d'*unpooling*).

La première partie du réseau, de *conv1_1* à *fc7*, n'est autre que le réseau VGG-16, créé par K.Simonyan, classé second lors de l'ILSVRC 2014.

Ce réseau est l'un des plus utilisés (ou réutilisés, comme ici) en vision par ordinateur, car il est d'une extrême simplicité. Il n'utilise que des convolutions de noyau (3, 3), qui sont jugées robustes pour l'apprentissage.

On peut remarquer que les convolutions sont regroupées ici en 5 blocs, chaque bloc étant séparé par une couche de *pooling*. Si l'on fait abstraction des couches d'activation ReLU, chaque bloc de convolution se résume comme une composée de convolutions. Or on peut remarquer que deux convolutions $(3, 3)$ successives équivalent à une convolution $(5, 5)$, venant de la propriété de majoration du support :

Soit f et $g \in L^2(\mathbb{R}^n)$ alors,

$$supp(f * g) \subset \overline{supp(f) + supp(g)} = \{x + y | x \in supp(f), y \in supp(g)\}$$

Ainsi l'architecture VGG-16 peut être alors approximée simplement par un réseau comprenant 5 couches de convolutions entrecoupées de couches d'activation et de *pooling*, les deux premières étant $(5, 5)$ et les trois suivantes étant $(7, 7)$. Ce modèle serait bien moins stable à l'apprentissage, mais aussi moins performant, car l'insertion d'activations entre les convolutions permet d'encore mieux séparer les plans de solutions.

La seconde partie du réseau, propre à DeconvNet, utilise les couches de déconvolution et d'*unpooling* détaillées précédemment. On remarquera aussi l'utilisation de déconvolutions successives de noyau $(3, 3)$, pour les mêmes raisons que les convolutions de la première partie.

Enfin la couche de sortie effectue le mapping entre les *features* produite et le nombre de classes souhaitées en sortie, par une convolution de noyau $(1, 1)$.

La philosophie de l'architecture DeconvNet pourrait se résumer à dans un premier temps décomposer l'image en *features* très profondes à travers 13 couches de convolution entrecoupées de *pooling*, puis de réduire l'image en un vecteur de 4096 *features*, avant de reconstituer une image segmentée à travers 13 couches de déconvolution entrecoupées d'*unpooling*.

Avantages et Inconvénients Nous allons à présent lister les différents avantages et inconvénients de l'architecture DeconvNet. Les avantages sont les suivants :

- Simplicité et élégance de conception.
- Réutilisation de VGG-16, modèle très performant en vision par ordinateur.
- Grande finesse de segmentation des détails des objets.
- 69% de précision sur PASCAL VOC 2012, offrant les meilleures performances en 2015.

Les inconvénients sont les suivants :

- Très grand nombre de *FLOPs*, nécessitant des GPUs très puissantes.
- Apprentissage complexe, divisé en deux phases, chacune ayant un ensemble d'apprentissage distinct.

Première Expérience

N'ayant pas de cartes graphiques suffisamment puissantes en terme de capacité de mémoire, seulement 4 Go, il n'a pas été possible de reprendre l'architecture de DeconvNet et l'entraîner à la segmentation d'images avec les labels provenant uniquement de SpaceNet, ce qui en fait de la segmentation binaire.

Des architectures, réutilisant les notions de convolution, *pooling*, déconvolution et *unpooling*, ont été esquissées. Ces architectures peuvent être vues comme des versions allégées de DeconvNet.

Architectures Afin de produire et entraîner une version allégée de DeconvNet, nous avons décidé, pour la première expérience, d'enlever des couches proche du *centre de symétrie* du réseau.

En effet, les couches les plus gourmandes en matières de nombre de paramètres d'apprentissage sont les couches *fc6* et *deconv-fc6*, car ce sont des convolutions ayant des noyaux de dimensions $(4096, 512, 7, 7)$ et $(512, 4096, 7, 7)$, soit $102760448 \approx 1.02 \times 10^8$ paramètres à ajuster pour chacune de ces deux couches, ce qui représente environ 822 Mo par couche.

Nous avons donc défini plusieurs modèles allégés de DeconvNet dans l'optique de les comparer, la logique de leur conception est simple, on pouvait remarquer que DeconvNet était constitué de 11 blocs de couches, dont 5 blocs de convolution, 5 de déconvolution et 1 bloc central. Nous avons donc décider de créer des modèles contenant 5 et 7 blocs, en voici leur définitions :

Couche	Noyau	Features	Stride	Pad	Dimensions en sortie	SegNet5	SegNet7
Input	-	-	-	-	(4, 224, 224)	✓	✓
conv1_1	(3, 3)	64	1	1	(64, 224, 224)	✓	✓
conv1_2	(3, 3)	64	1	1	(64, 224, 224)	✓	✓
pool1	(2, 2)	-	2	0	(64, 112, 112)	✓	✓
conv2_1	(3, 3)	128	1	1	(128, 112, 112)	✓	✓
conv2_2	(3, 3)	128	1	1	(128, 112, 112)	✓	✓
pool2	(2, 2)	-	2	0	(128, 56, 56)	✓	✓
conv3_1	(3, 3)	256	1	1	(256, 56, 56)	✓	✓
conv3_2	(3, 3)	256	1	1	(256, 56, 56)	✓	✓
conv3_3	(3, 3)	256	1	1	(256, 56, 56)	-	✓
pool3	(2, 2)	-	2	0	(256, 28, 28)	-	✓
conv4_1	(3, 3)	512	1	1	(512, 28, 28)	-	✓
conv4_2	(3, 3)	512	1	1	(512, 28, 28)	-	✓
deconv4_3	(3, 3)	256	1	1	(256, 28, 28)	-	✓
unpool3	(2, 2)	-	2	0	(256, 56, 56)	-	✓
deconv3_1	(3, 3)	256	1	1	(256, 56, 56)	-	✓
deconv3_2	(3, 3)	256	1	1	(256, 56, 56)	-	✓
deconv3_3	(3, 3)	128	1	1	(128, 56, 56)	✓	✓
unpool2	(2, 2)	-	2	0	(128, 112, 112)	✓	✓
deconv2_1	(3, 3)	128	1	1	(128, 112, 112)	✓	✓
deconv2_2	(3, 3)	64	1	1	(64, 112, 112)	✓	✓
unpool1	(2, 2)	-	2	0	(64, 224, 224)	✓	✓
deconv1_1	(3, 3)	64	1	1	(64, 224, 224)	✓	✓
deconv1_2	(3, 3)	64	1	1	(64, 224, 224)	✓	✓
Output	(1, 1)	2	1	0	(2, 224, 224)	✓	✓

Comme nous pouvons le voir, les deux modèles possèdent un bloc central différent de DeconvNet. Dans DeconvNet, l'entrée était de dimension (512, 7, 7), puis ramené à un vecteur de 4096 *features* dans la couche *fc6*, ce qui dé-spatialisait entièrement l'information. Ici dans le bloc central l'information reste spatialisée, car on ne passe pas de (256, 56, 56) (ou (512, 28, 28)) à un vecteur de *n features*.

Hyper-paramètres Nous allons à présent lister les hyper-paramètres liés aux apprentissages effectués.

Algorithme de Descente de Gradient Nous avons choisi d'utiliser, comme algorithme de descente la variante *Adam*, signifiant *Adaptative Moment Estimation*, qui utilise le moment des gradients précédents et qui moyenne l'amplitude des gradients calculés. Nous avons utilisé les paramètres préconisés par les auteurs, à savoir $\beta = 0.9$, $\gamma = 0.999$ et $\alpha = 0.1$. Pour plus de détails, voir en annexe.

Régularisation L^2 Nous avons choisi d'utiliser, comme taux de régularisation L^2 , $\lambda = 0.0005$, qui est une valeur couramment choisie en vision par ordinateur.

Proportions de données d'apprentissage Il est courant dans le cas de la segmentation d'images de prendre 90% des données pour l'ensemble d'apprentissage et 10% pour l'ensemble de validation.

Taille d'un *mini-batch* Nous avons fixé la taille d'un mini-batch à 64, valeur communément utilisée dans la littérature.

Nombre d'itérations Nous avons fixé à 50000 le nombre d'itérations lors de l'apprentissage, valeur jugée suffisante, au delà de laquelle si un modèle n'a toujours pas suffisamment appris, alors sa capacité sera remise en question.

Résultats

Graphes des Pertes Voici le graphe des pertes du modèle SegNet5 :

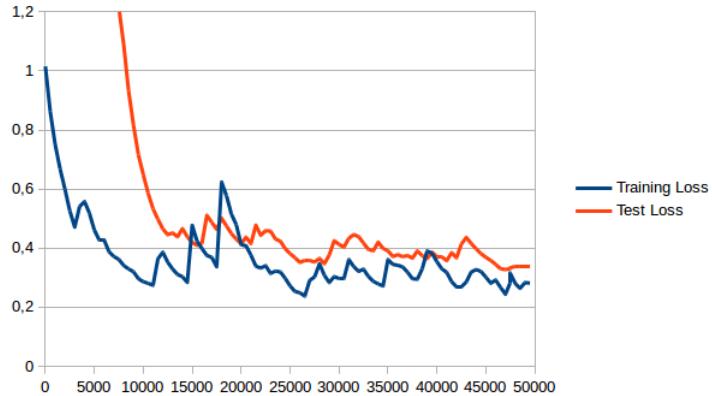


FIGURE 5.5 – Graphe de la fonction de perte pour le modèle SegNet5

L'écart entre l'erreur d'apprentissage et l'erreur de test est faible, on peut donc conclure qu'il n'y a peu ou pas de sur-apprentissage. De plus l'erreur ne varie pas de façon particulièrement erratique, ce qui est signe d'un bon apprentissage.

Voici le graphe des pertes du modèle SegNet7 :

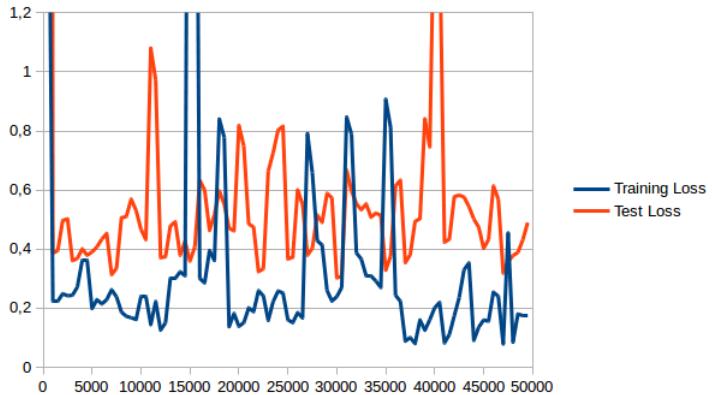


FIGURE 5.6 – Graphe de la fonction de perte pour le modèle SegNet7

Ici l'écart entre l'erreur d'apprentissage et l'erreur de test est grande, ce qui laisse supposer du sur-apprentissage. De plus l'erreur varie de façon erratique, signe d'un mauvais apprentissage.

Analyse des Résultats Voyons à présent les meilleures performances des modèles.

La précision est calculée de la façon suivante :

$$Accuracy(X) = \frac{1}{|X|} \sum_{x \in X} \frac{1}{|I| \times |J|} \sum_{i \in I, j \in J} \mathbb{1}_{\{x_{i,j} = l_{i,j}\}}$$

Avec $x_{i,j}$ l'*argmax* de couche *output* du modèle aux coordonnées (i, j) , et $l_{i,j}$ la valeur du label aux coordonnées (i, j) . Ce qui revient à calculer la moyenne sur toutes les données de l'ensemble, de la moyenne du nombre de pixels correctement classifiés sur le nombre total de pixels.

Pour SegNet5, l'itération 46500 semble être la meilleure.

La précision globale est de 0.892154, ce qui signifie qu'environ $\frac{9}{10}$ des pixels sont correctement classifiés.

La matrice de confusion est la suivante :

Réel/Prédit	Bâtiment	Absence
Bâtiment	0.747	0.253
Absence	0.114	0.886

Pour SegNet7, l'itération 47000 semble être la meilleure.

La précision globale est de 0.911284.

La matrice de confusion est la suivante :

Réel/Prédit	Bâtiment	Absence
Bâtiment	0.672245	0.327755
Absence	0.080381	0.919619

On remarque ici que le modèle a tendance à mieux classifier l'absence de bâtiments que la présence de bâtiments.

On constate aussi qu'il n'y a pas d'améliorations par rapport à SegNet5.

Exemples de Résultats

Deuxième Expérience

Une analyse empirique de la première expérience laisse suggérer que les réseaux SegNet5 et SegNet7 souffre de *tunnel vision*. En effet, comme on peut le voir, un grand bâtiment sur l'image n'est pas détecté comme un bloc unique de pixels, mais plutôt comme la réunion de plusieurs petits blocs.

Après réflexion, nous sommes arrivés à la conclusion que cela était du au fait que le réseau ne zoomait pas assez.

Dans le cas de SegNet5, on ne s'intéresse ici qu'à la résolution de l'image et des *map de features*. Les couches *pool1* et *pool2*, divisent chacune la résolution de l'image par 2, soit par $2 \times 2 = 4$ après *pool2*. De ce fait, les *map de features* après *pool2*, ont pour résolution spatiale (56, 56), un pixel de ces couches représente seulement une zone de (4, 4) pixels de l'image originale.

Passons à présent aux couches *conv3_1* et *conv3_2*, ce sont toutes les deux des convolutions (3, 3), ce qui équivaut grossièrement à une convolution (5, 5).

On rappelle qu'une convolution s'apparente à une moyenne pondérée sur un voisinage. On comprend donc que le réseau prend en compte les *features* d'un pixel et de son voisinage (5, 5), et porte un "avis" quand à la nature des objets contenus dans ce voisinage.

Ici une fenêtre (5, 5) de pixels, représente une zone de (20, 20) pixels de l'image originale. Et c'est à peu près sur cette quantité d'information que le réseau va pouvoir juger si oui ou non la fenêtre (5, 5) représente un bâtiment.

Or les données SpaceNet ont une résolution de 0.31cm, une fenêtre de pixels de dimensions (20, 20) au sol représente, une zone de $6.2 \times 6.2 \text{ m}^2$, ce qui est évidemment beaucoup plus petit que la plupart des bâtiments ou maisons.

Dans cette deuxième expérience, nous avons recréé deux versions miniatures de DeconvNet.

Architectures Au lieu d'enlever des couches proche du *centre de symétrie* du réseau, comme réalisé dans la première expérience, nous avons plutôt décider de compacter l'architecture DeconvNet.

Ainsi les modèles conçus réduisent plus rapidement la dimension de l'image en employant des couches de *pooling* ayant un plus grand champ-récepteur.

Nous avons aussi décidé de diviser par 2 le nombre de *features* calculées à chaque convolution, dans le but d'accélérer les calculs.

Nous avons donc décidé de créer deux modèles se différenciant sur le bloc central, en voici leur définitions :

Couche	Noyau	Features	Pad	Stride	Dimensions en sortie
Input	-	-	-	-	(4, 224, 224)
conv1_1	(3, 3)	32	1	1	(32, 224, 224)
conv1_2	(3, 3)	32	1	1	(32, 224, 224)
pool1	(4, 4)	-	4	0	(32, 56, 56)
conv2_1	(3, 3)	64	1	1	(64, 56, 56)
conv2_2	(3, 3)	64	1	1	(64, 56, 56)
pool2	(4, 4)	-	4	0	(64, 14, 14)
conv3_1	(3, 3)	128	1	1	(128, 14, 14)
conv3_2	(3, 3)	128	1	1	(128, 14, 14)
conv3_3	(3, 3)	128	1	1	(128, 14, 14)
pool3	(2, 2)	-	2	0	(128, 7, 7)
conv4_1	(3, 3) (7, 7)	256	1 0	1 1	(256, 7, 7) (256, 1, 1)
conv4_2	(3, 3) (1, 1)	256	1 0	1 1	(256, 7, 7) (256, 1, 1)
deconv4_3	(3, 3) (7, 7)	128	1 0	1 1	(128, 7, 7) (128, 7, 7)
unpool3	(2, 2)	-	2	0	(128, 14, 14)
deconv3_1	(3, 3)	128	1	1	(128, 14, 14)
deconv3_2	(3, 3)	128	1	1	(128, 14, 14)
deconv3_3	(3, 3)	64	1	1	(64, 14, 14)
unpool2	(4, 4)	-	4	0	(64, 56, 56)
deconv2_1	(3, 3)	64	1	1	(64, 56, 56)
deconv2_2	(3, 3)	32	1	1	(32, 56, 56)
unpool1	(4, 4)	-	4	0	(32, 224, 224)
deconv1_1	(3, 3)	32	1	1	(32, 224, 224)
deconv1_2	(3, 3)	32	1	1	(32, 224, 224)
Output	(1, 1)	3	1	0	(3, 224, 224)

Les différences par rapport à la première expérience sont l'utilisation de couche de *pooling* ayant un noyau (4, 4) et un pas de déplacement de 4, permettant de réduire rapidement les dimensions des *map of features*.

Les différences entre les deux modèles se font au niveau du bloc central, dans l'un, l'information reste spatialisée, nous opérons seulement deux convolutions (3, 3) comme dans l'expérience précédente et dans l'autre, nous reprenons un bloc similaire à celui de DeconvNet.

Hyper-paramètres Les hyper-paramètres restent les même par rapport à l'expérience précédente.

Résultats

Graphes des Pertes Voici le graphe des pertes du modèle ZNet :

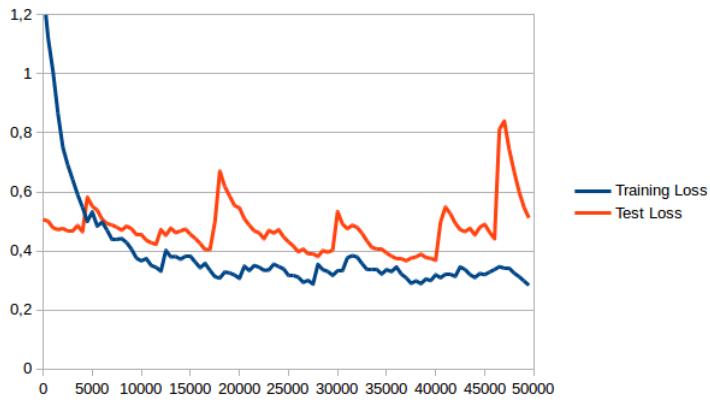


FIGURE 5.7 – Graphe de la fonction de perte pour le modèle ZNet

L'écart entre l'erreur d'apprentissage et l'erreur de test reste faible avant l'itération 40000, on peut donc conclure qu'il n'y a peu ou pas de sur-apprentissage avant 40000 itérations. De plus l'erreur varie de façon peu erratique, ce qui est signe d'un bon apprentissage.

Voici le graphe des pertes du modèle ZNet2 :

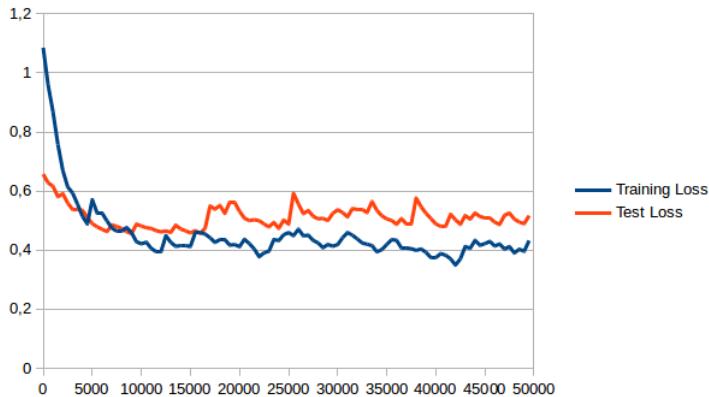


FIGURE 5.8 – Graphe de la fonction de perte pour le modèle ZNet2

L'écart entre l'erreur d'apprentissage et l'erreur de test est faible, on peut donc conclure qu'il n'y a peu ou pas de sur-apprentissage. De plus l'erreur ne varie pas de façon particulièrement erratique, ce qui est signe d'un bon apprentissage.

Analyse des Résultats Voyons à présent les meilleures performances des modèles.

La précision est calculée de la même façon que précédemment.

Pour ZNet, l'itération 37000 semble être la meilleure.

La précision globale est de 0.83479.

La matrice de confusion est la suivante :

Réel/Prédit	Bâtiment	Route	Absence
Bâtiment	0.825		0.175
Route	0.262*	0.738	0.262*
Absence	0.176		0.824

La matrice de confusion n'est pas complète, car Caffe ne produit pas l'ensemble des informations lors de l'évaluation du modèle.

L'étoile, *, signifie que la valeur est partagée entre les deux champs.

Pour ZNet2, l'itération 34000 semble être la meilleure.

La précision globale est de 0.777292.

La matrice de confusion est la suivante :

Réel/Prédit	Bâtiment	Route	Absence
Bâtiment	0.868	0.132	
Route	0.278*	0.722	0.278*
Absence	0.25		0.75

De ces deux modèles, le premier semble plus performant que le second, ce qui indique peut être que garder l'information spatialisée sur le bloc central est important.

Exemples de Résultats

5.1.4 Protocole expérimental

5.1.5 Résultats

5.1.6 Comparaisons entre modèles

Chapitre 6

Conclusion

Bibliographie

- Deep Learning : <http://deeplearning.net/>
- Deep Learning Book : <http://www.deeplearningbook.org/>
- Neural Networks and Deep Learning : <http://neuralnetworksanddeeplearning.com/>

Annexes

6.1 Algorithme de Descente de Gradient : Adam

La méthode *Adam*, signifiant *Adaptive Moment Estimation*, est une amélioration de la méthode *RMSProp*.

Cette méthode utilise à la fois le concept de moyenne mobile ainsi que le concept d'utilisation des moments, c'est à dire d'utiliser aussi les gradients précédents :

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta J(x, y, \theta_k) - \beta \nabla_\theta J(x, y, \theta_{k-1})$$

On définit alors la matrice G_k de la même façon que dans *RMSProp* :

$$G_k = \gamma G_{k-1} + (1 - \gamma)(\nabla_\theta J(x, y, \theta_k) \odot \nabla_\theta J(x, y, \theta_k)) \times I_n$$

On définit par ailleurs m_k , le moment du gradient $\nabla_\theta J(x, y, \theta)$, tel que :

$$m_k = \beta m_{k-1} + (1 - \beta)\nabla_\theta J(x, y, \theta_k)$$

On peut remarquer que m_k est lui-même une moyenne mobile des gradients. Nous utiliserons alors les estimateurs non-biaisés de m_k et G_k :

$$\hat{m}_k = \frac{m_k}{1 - \beta^k}$$

et

$$\hat{G}_k = \frac{G_k}{1 - \gamma^k}$$

Ainsi à chaque itération, nous appliquerons à θ , la formule suivante :

$$\theta_{k+1} = \theta_k - \alpha(\hat{G}_k)^{-\frac{1}{2}} \hat{m}_k$$

Nous avons alors l'algorithme suivant :

Algorithm 8 Algorithme de Descente de Gradient Stochastique utilisant *Adam*

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

$m < n$,

θ_0 , les paramètres initiaux du modèle, initialisés aléatoirement,

$\beta = 0.9$, $\gamma = 0.999$, $\alpha = 0.001$.

et $\epsilon > 0$, seuil de tolérance.

repeat

$\hat{x}, \hat{y} = sample(x, y, m)$

Calcul de $\nabla_\theta J(\hat{x}, \hat{y}, \theta_k)$.

$m_k = \beta m_{k-1} + (1 - \beta)\nabla_\theta J(x, y, \theta_k)$

$\hat{m}_k = \frac{m_k}{1 - \beta^k}$

$G_k = \gamma G_{k-1} + (1 - \gamma)\nabla_\theta J(x, y, \theta_k) \odot \nabla_\theta J(x, y, \theta_k) \times I_n$

$\hat{G}_k = \frac{G_k}{1 - \gamma^k}$

$\theta_{k+1} = \theta_k - \alpha(\hat{G}_k)^{-\frac{1}{2}} \times \hat{m}_k$

until $\nabla_\theta J(\hat{x}, \hat{y}, \theta_k) \leq \epsilon$

return θ_{k+1}

Nous retiendrons cet algorithme pour résoudre les problèmes d'optimisation pour deux raisons :

- Il ne nécessite pas de pas de paramétrisation, si ce n'est le choix de la taille des mini-lots m , ne nécessitant peu de tests.
- Il offre une convergence très rapide, grâce à la méthode des moments, et stable, grâce à la normalisation non biaisée du gradient $\nabla_{\theta}J(x, y, \theta)$.