Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

Reconnaissance des Plantes à l'aide d'un Réseau de Neurones Convolutif

Réalisé par: Aicha Maaoui, Abdeljalil Bouchelga, Mohamed Aitmhand,

Farhat Aitaider, Lydia Chouaki

Encadré par: Prof. Hugo Bolloré

Janvier 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

La reconnaissance des plantes avec un réseau de neurones, à partir des images données, n'est pas -toujours- une tâche évidente. Ceci est dû au fait qu':

- Il y a une apparence diversifiée de structures complexes des plantes,
- Le problème de classification est réalisé, dans les majorités des cas, avec un nombre de classes assez élevé.

Le problème de classification des images de plantes est ainsi un problème bien posé, où les classe sont délimitées. Nous avons besoin dans ce cas d'informations sur les l'images pour représenter et identifier l'objet qu'elle contient.

L'objectif de ce projet consiste à la reconnaissance des images de plantes grâce à un réseau de neurones convolutif: On utilise l'algorithme **CNN (Convolutional Neural Network)**.

Initialement, on dispose d'un jeu de données de 3655 images de fleurs, formants 5 classes différentes. Ceci nous permet de faire l'apprentissage d'un classifieur et générer à la fin un model capable de classifier nos images.

Afin de réaliser ce projet, nous avons procédé comme suit:

- Comprendre le fonctionnement des réseaux de neurones en informatique (IA), ainsi que la base de fonctionnement d' un algorithme CNN,
- Travailler sur l'exploration des données afin de déterminer les couches à utiliser dans notre réseau de neurones,
- Etat de l'art des algorithmes de traitement d'image existants dans le domaine de la reconnaissance de plantes,
- Pogrammation en utilisant le langage de programmation $c++$ pour le classifieur, ainsi

que le langage *Python* pour la structuration des jeux de donnes.

Les étapes du projet sont structurés à l'aide de *github* dans le dépot crée avec des *issues*, qui indiquent l'état d'avancement du projet comme illustré dans la figure (1).

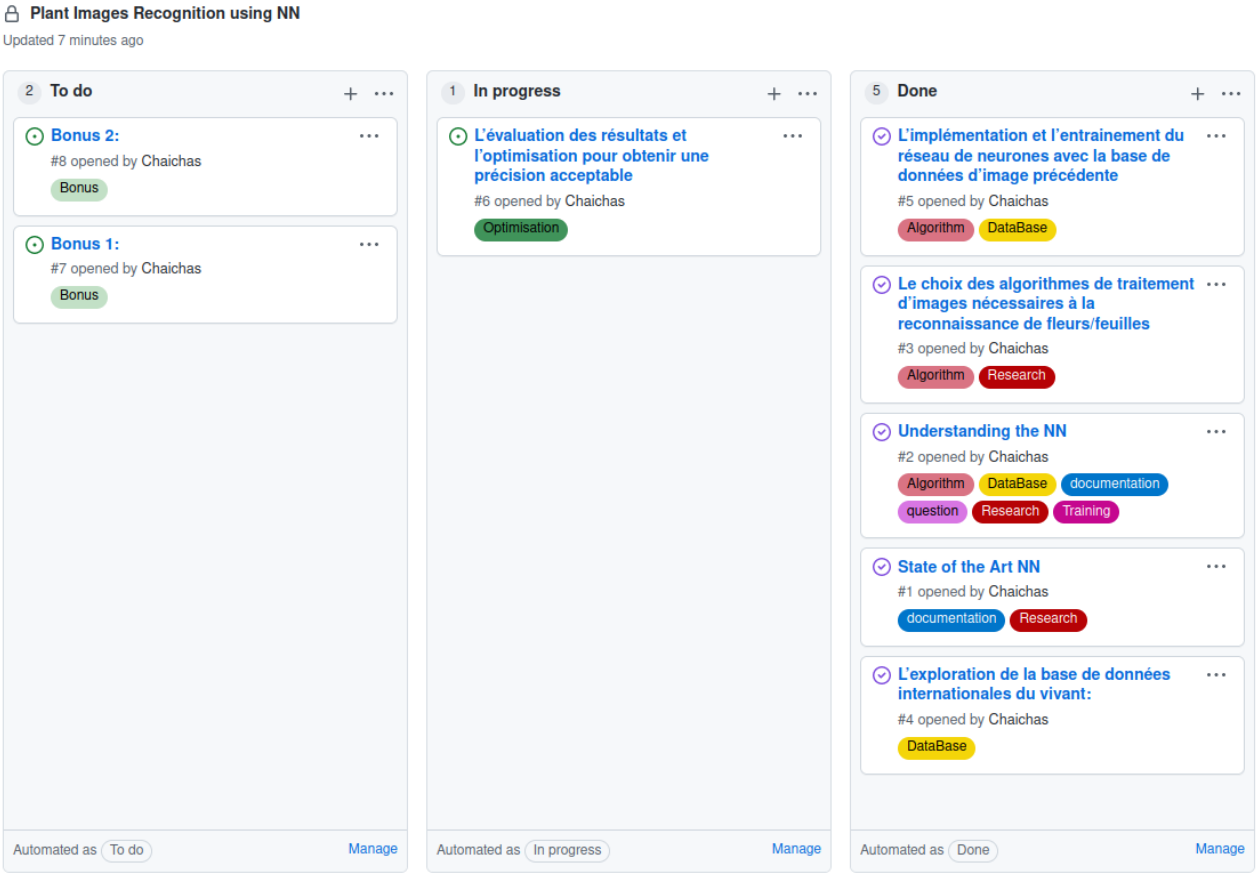

Figure 1: Description de l'état d'avancement du Project CNN.

Contents

List of Tables

Chapter 1

Implémentation du CNN

1.1 Introduction

Un réseau de neurones est une interprétation machine du cerveau humain, contenant des millions de neurones. Ces derniers transmettent des informations sous forme d'impulsions électriques, *"Deep Neural Network: Qu'est-ce qu'un réseau de neurones profond ?"* 2021.

Ils sont utilisés pour résoudre aux problèmes complexes qui nécessitent des calculs analytiques similaires à ceux du cerveau humain, *"Les Réseaux de Neurones artificiels"* 2018.

On peut citer comme applications les plus courantes des réseaux de neurones, *"Les Réseaux de Neurones artificiels"* 2018:

- **La classification:** C'est la distribution des données par paramètres. Par exemple, on dispose d'un ensemble de personnes à l'entrée. Il faut décider à laquelle d'entre elles on accorde un prêt. Ce processus peut être effectué par un réseau de neurones, en analysant des informations telles que l'âge, la solvabilité, Les antécédents de crédit, etc,
- **La prédiction:** C'est la capacité de prédire la prochaine étape. Par exemple, la hausse ou la baisse d'une action en fonction de la situation du marché boursier,
- **Reconnaissance:** Elle représente actuellement l'utilisation la plus répandue des réseaux de neurones. On cite comme titre d'exemple "Google" lorsqu'on cherche une photo ou dans les appareils photo des téléphones lors de la détection de la position du visage, etc.

Fonctionnement d'un réseau de neurones:

Un neurone est une unité de calcul qui reçoit des informations, effectue des calculs simples pour les transmettre plus loin. Ils sont divisés en trois types principaux, comme illustré dans la figure (1.1), *"Réseaux de Neurones"* 2013:

- Entrée(X_i), avec les paramètres W_i (poids) et b_j (bias),
- Caché (F).
- Sortie (Y_i).

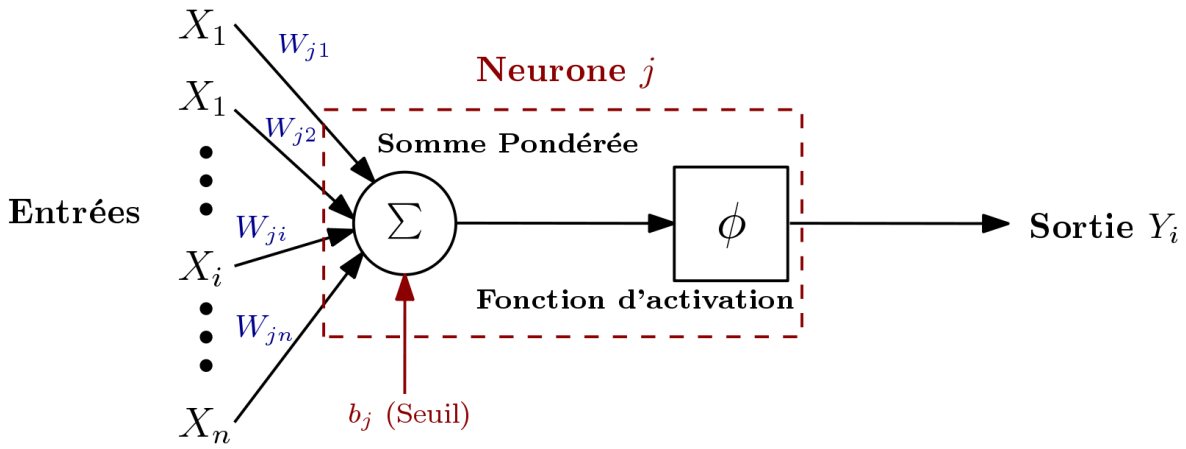

Figure 1.1: Eléments d'un réseau de neurones.

Dans le cas où un réseau de neurones est constitué d'un grand nombre de neurones, on introduit le terme *couche*. Par conséquent, il existe:

- Une couche d'entrée qui reçoit les informations,
- n couches cachées (généralement un max de 3) qui les traitent,
- une couche de sortie qui génère le résultat.

Chacun des neurones a 2 paramètres principaux: (i) les données d'entrée et (ii) les données de sortie.

Lors de l'initialisation du réseau de neurones, les poids W_i sont attribués d'une manière aléatoire.

La fonction d'activation est un moyen de normaliser les données d'entrée, *"Fonction d'activation"* 2021. Il existe beaucoup de fonctions d'activation. Cependant, on considère les fonctions basiques : *Linéaire*, *Softmax* et *Sigmoïde (Logistique)*. La principale différence entre ces derniers consiste à la plage de valeurs fournis à la sortie.

1.2 Description du Réseau de Neurones du Projet

1.2.1 Description de la base de données

Dans cette partie, on se propose de faire l'analyse du travail réalisé sur la partie *Dataset* (Base de données).

Collecte des données

Dans un premier temps, on a commencé la collecte des données depuis *"Plantae"* 2020. Cependant, on a rencontré des difficultés, se représentant en:

- Nombres insuffisants d'image pour chaque classe de plantes,
- Complexité des images (beaucoup de détails).

Ce qui ne permet pas de générer un model parfait de reconnaissance d'image. Pour cela, on a considéré les différents jeux de données partagés par des éditeurs dans *"PlantVillage Dataset"* 2020, plus facile à manipuler.

Exploration des données

Notre jeux de données se compose des images de fleurs de 5 classes, nommées comme suit:

- Daisy,
- Dandelion,
- Rose,
- Sunflower,
- Tulip.

Pour les images de classes choisies pour les entraîner, il existe des informations perturbatrices, i.e., il y a quelques images qui présentes des insectes perturbants l'unicité des fleurs et ces photos ne représentent qu'une partie du jeux donnés, comme illustré dans la figure (1.2).

Figure 1.2: Présence d'un insecte, qui perturbe l'unicité de la fleur.

Sinon, le jeu de données choisit contient des images adéquates pour générer un model de classification d'image.

Modifications apportées

Dans le jeux de données collecté depuis le site Kaggle, on trouve l'existence de classes volumineuses, i.e., 984 le maximum d'images de la classe Tulip, et 733 pour le classe de Sunflower.

De plus, les photos ont des tailles très variés. La majorité ont comme dimensions 200×200.

Il était donc important d'ajuster la Dataset utilisée, afin d'avoir un standard avec lequel on peut travailler et générer un model de meilleur performance.

Dans un premier temps, on a redimensionné toutes les photos des classes à une dimension de 50×50 pixels, car dans ce cas on aura:

- une conservation de la bonne visualisation (qualité) des images, lues sous forme de matrices carrées réduites,
- Réduction du temps total de calcul.

Dans un second temps, on a définit le même nombre d'images associées à chaque classe, pour avoir un jeu de données uniforme.

Pour se faire, on a généré le fichier *resize.py* (en python), qui permet de modifier le jeu de données utilisé pour générer le model.

1.2.2 Description des classes

Class Data

a. OpenCV

OpenCV (Open Computer Vision) est une bibliothèque graphique libre, initialement développée par "Intel", spécialisée dans le traitement d'images en temps réel. *OpenCV C++* est livré avec cet incroyable conteneur d'images Mat qui gère tous, *"Lire et afficher une image dans OpenCV en utilisant C++"* 2022, *"OpenCV"* 2022.

La classe Data a pour but de stocker des images comprenant 50×50 pixels dans un vecteur 1D en transformant l'images RVB de 3 canaux à 1 canal.

Les valeurs de pixels sont souvent des nombres entiers non signés compris entre 0 et 255.

Bien que ces valeurs de pixels puissent être présentées directement avec modèles de réseaux de neurones dans leur format brut, cela peut entraîner des problèmes lors de l'apprentissage qui sera plus lent que prévu du modèle. Donc il est avantageux de normaliser les valeurs de pixels et de changer la plage $[0, 255]$.

Dans notre cas, on a mis à l'echelle les valeurs de pixels dans la plage $[0.5, 0.5]$, *"OpenCV"* 2022.

L'image est une matrice représentée par la classe cv: Mat.

Chaque élément de la matrice représente un pixel. Pour les images ayant des niveaux de gris, l'élément de matrice est représenté par un nombre de 8 bits non signé (0 à 255). Pour une image couleur au format RVB, il existe 3 de ces nombres, un pour chaque composante de couleur. La fonction utilisé pour lire une image dans opencv est: **imread()**.

Cette fonction permet de lire les images et prend les 2 arguments suivants :

- **Nom de fichier:** L'adresse complète de l'image à charger est de type string,
- **flag:** C'est un argument optionnel et détermine le mode dans lequel l'image est lue. Dans notre code, nous avons utilisé le mode "$CV\_LOAD\_IMAGE\_COLOR$" pour télécharger l'image en couleur.

Les images du jeu de données sont des images RVB à 3 canaux. On a transformé les images de 3 canaux à 1 canal pour faciliter le travail. Pour ce faire, on a utilisé $Cv :: vect3b$ dans *opencv* pour stocker les couleurs des pixels.

Module Direction_file

le but du module Direction est de récupérer les fichiers (l'image) et son label. Le label indique à quelle famille il appartient.

Le premier chiffre du nom est le label de l'image. Par exemple: "$0img\_0.jpg$" représente la classe rose.

Il existe 5 types de fleurs, on a les labels suivantes:

- Tulip (label 0),
- Daisy (label 1),
- Dandelion (label 2),
- Sunflower (label 3),
- Rose (label 4).

Les images de ces fleurs (comptant 3500 images) se trouvent dans le répertoire DataSet.

1.2.3 Couche de Comvolution

Le but de cette partie est la description de la classe *Convolution_layer* implémentée dans le code. Généralement, la convolution utilise du ensemble de filtres (formés par des Kernels) pour extraire les caractéristiques (features) d'une image donnée à l'entrée.

Les images en couleur sont representées par une matrice de pixels. Un pixel se dispose de 3 canaux RGB (Rouge-Vert-Bleu), comme illustré dans la figure (1.3). Chaque filtre doit de même avoir 3 canaux, égal à l'image d'entrée, comme illustré dans la figure (1.4).

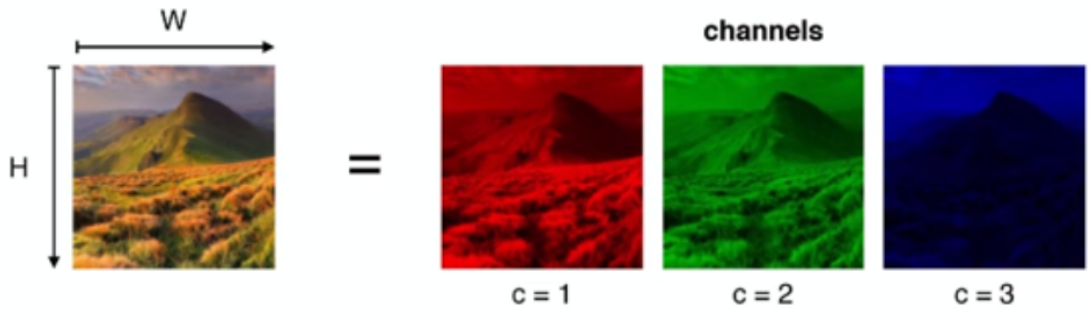

Figure 1.3: Représententation d'une image par 3 canaux RGB.

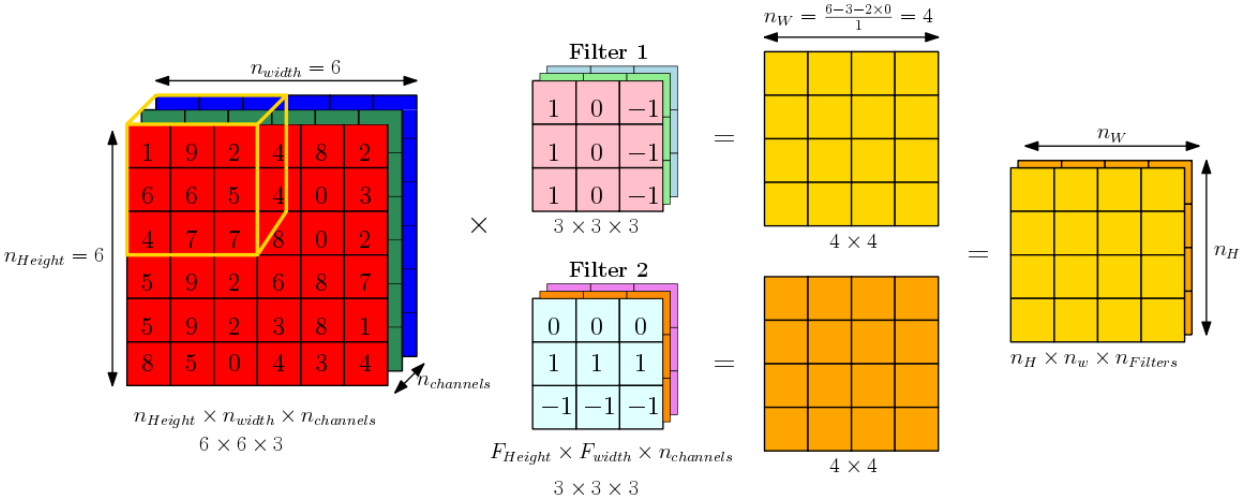

Figure 1.4: Convolution avec deux filtres, stride $S = 1$, padding $p = 0$, nombre de canaux = 3.

Initialement, l'image d'entrée est considérée comme un volume de dimension ($input\_number \times Matrix\_height \times Matrix\_width \times Channels\_number$). Après la réduction du nombre de canaux de 3 à 1, chaque image d'entrée sera considérée comme une matrice $2D$ de dimensions ($input\_number \times Matrix\_height \times Matrix\_width$).

Chaque filtre appliqué aux images d'entrée est un volune de dimension initiale ($Filter\_height \times Filter\_width \times Channels\_number$). Par conséquent, les filtres disposent d'un volume total de ($Filter\_height \times Filter\_width \times Channels\_number \times Filters\_number$). Après réduction de nombre de canaux de 3 à 1, les filtres totaux appliqués a l'image auront comme dimensions ($Filter\_height \times Filter\_width \times Filters\_number$).

La couche de convolution (ou *convolution_layer*) est la couche de base d'un réseau de neurones convolutif. Elle représente la matrice de valeurs obtenues en sommant les produits entre chaque pixel de l'image et le pixel du filtre. Soit l'exemple illustré dans la figure (1.5).

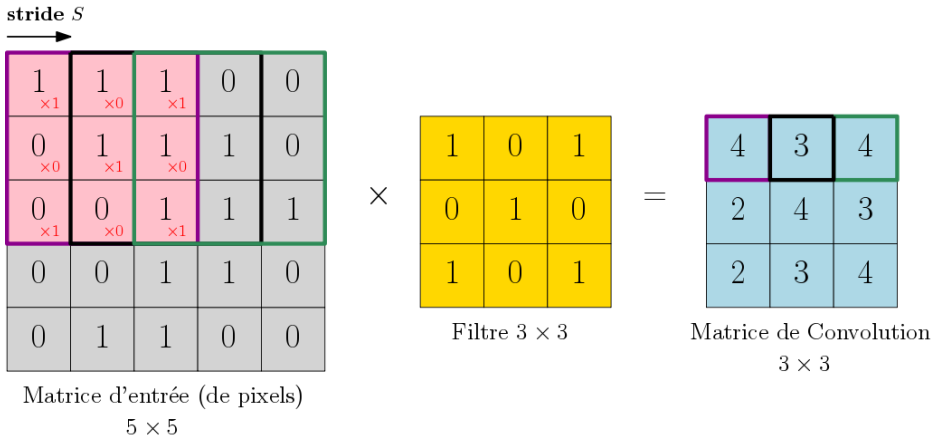

Figure 1.5: Matrice de convolution, stride $S = 1$, padding $p = 0$.

On explique dans ce qui suit les étapes d'obtention de la matrice de convolution, *"Réseau neuronal convolutif"* 2021, *"Convolutional neural network"* 2021.

Paramètres de dimensionnement du volume de la couche de convolution (volume de sortie):

- ***1. Filter (or Kernels) size (dimensions du filtre):*** indique le nombre de pixels du filtre. A titre d'exemple, le filtre presenté dans la figure (1.5) est de taille 3×3,
- ***2. Stride (or offset)*** S ***(pas):*** indique le déplacement (en nombre de pixels) à chaque itération.

  Dans la figure (1.5), on a un stride $S = 1$, ainsi chaque filtre est décalé de 1 pixel par rapport au bloc de la matrice d'entrée, auquel il est superposé. Cette procédure est représentée dans la figure (1.5) par les cadres en mauve, noir et vert dans la matrice d'entrée.
- ***3. Padding*** p ***(Marge à zéro):*** Parfois, dans le but de contrôler la dimension spatiale du volume de sortie, on met des zéros à la frontière du volume de l'image d'entrée.

  En revenant à notre exemple illustré dans la figure (1.5), on a $n = 5$ pixels à l'entrée, le filtre a comme dimension $f = 3$. Alors, la matrice de convolution aura comme dimension $n_1 = n - f + 1 = 3$ (on risque de perdre quelques informations).

  Maintenant, si on souhaite obtenir une dimension $n_1 = 5$ de la matrice de convolution obtenue comme sortie (égale à la dimension d'entrée), et on a une taille fixe du filtre $f = 3$, alors on doit chercher n tel que $n_1 = 5$. Dans ce cas, on trouvera $n = 7$.

  Par conséquence, on ajoute un bloc enveloppant la matrice d'entrée formée par les pixels, comme illustrée dans la figure (1.6). Le but de cette procédure est d'éviter la perte des données en pixels de la matrice d'entrée.

  En raison de simplicité de la couche de convolution, on ne considère pas de padding dans notre projet ($p = 0$). Il s'agit d'une "comvolution valide".

  La dimension du volume de sortie est décrite dans l'équation (1.1).

$$Largeur_{matrice\ sortie} = \frac{Largeur_{matrice\ d'entrée} - Largeur_{filtre} + 2 \times Padding}{Stride} + 1 \quad (1.1)$$

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

Matrice d'entrée (de pixels)
7×7

Figure 1.6: Matrice d'entrée en pixels (avec bloc de 0: Zero padding).

On applique l'équation (1.1) dans notre projet (où la taille de matrice d'entrée est 50×50 pixels et le filtre est 3×3). on aura comme taille de matrice de convolution à la sortie:

$$Largeur_{matrice\ de\ sortie} = \frac{50-3+2\times 0}{1} + 1 = 50 - 2 = 48 = Largeur_{matrice\ d'entrée} - 2$$

Soit la figure (1.7), illustration du procédure de convolution de ce projet.

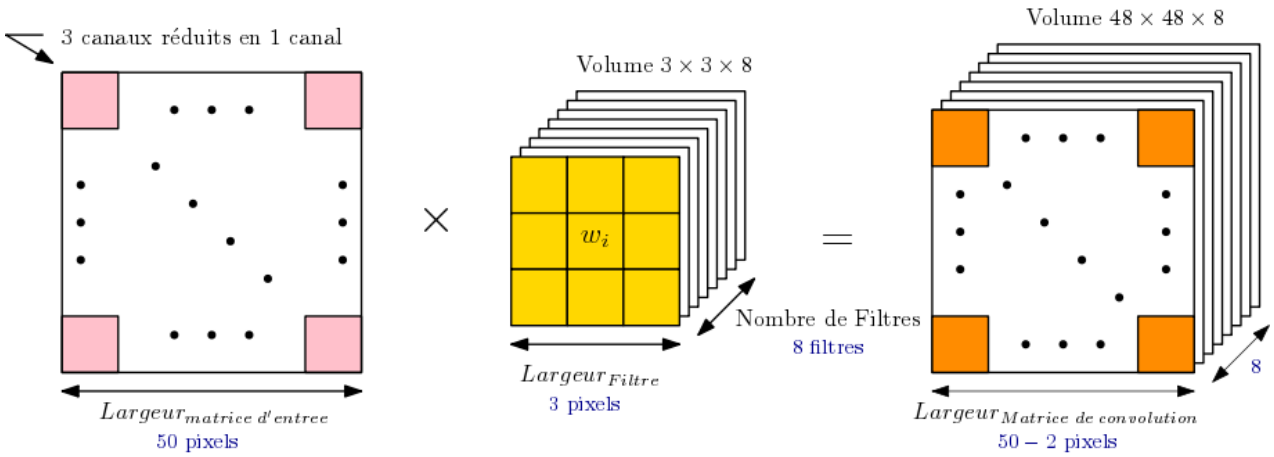

Figure 1.7: Couche de convolution du Projet.

Etapes de Convolution:

Pour produire la matrice de convolution à la sortie de la couche, on utilise une image d'entrée et un filtre. Les étapes de convolution de ce filtre avec l'image d'entrée sont:

- Superposition du filte (initialisé avec des valeurs de poids aléatoires) avec l'image d'entrée,

- Multiplication membre-à-membre de pixels dans le filtre et ceux correspondants dans l'image d'entrée,
- Sommation de valeurs obtenus pour un emplacement de filtre donné,
- Répétition de la procédure de convolution pour tous les emplacements du filtre.

Les paramètres utilisés dans le project de reconnaissance des images de plantes sont regroupés dans le tableau (1.1).

Dimensions de l'image d'entrée	50×50
Nombre de canaux de l'image d'entrée	3 (RGB)
Nombre de canaux à l'entrée de *Convolution_layer*	1
Dimensions du filtre	3×3
Nombre de filtres utilisés	8
Stride S	1
Padding p	0

Table 1.1: Paramètres de la couche de convolution du projet.

Le choix du filtre dans le cardre du projet est justifié par le fait qu'un filtre de dimensions impaires et minimales est le plus utilisé, *"Deciding optimal kernel size for CNN"* 2018.

Le nombre total des poids dans les 8 filtres est égal à: $3 \times 3 \times 8 = 72$ poids.

Le volume de sortie de la matrice de convolution est de dimension $(48 \times 48 \times 8)$.

Classe Random pour initialiser les poids du filtre:

Le but de cette partie est la description de la classe *Random_weights* pour initialiser les poids des filtres appliqués à une image donnée a l'entrée.

Generalités:

La première étape de l'utilisation de la méthode de **Gradient Descent** est la génération de nombres aléatoires pour les différentes valeurs de filtres utilisés.

On se propose d'utiliser dans cette partie des classes issues de la **bibliothèque standard (STL)** de $C++$ (Standard Template Library, dont le préfixe est **std::**), *"Programmation Generique, Bibliotheque Standard (STL)"* 2018.

La bibliothèque **Random** de STL permet de générer des variables aléatoires de loi donnée. On se propose d'utiliser la loi de densité de STL, **std::normal_distribution**.

Description de **std::normal_distribution** $(C++11)$:

Il s'agit de la génération de nombres aléatoires selon la distribution de nombres aléatoires normale (ou gaussienne), *"std::normal_distribution"* 2021, comme défini dans l'équation (1.2).

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-1}{2} \frac{x-\mu}{\sigma}^2 \tag{1.2}$$

avec μ est la moyenne de distribution (mean) et σ est l'écart-type (standard deviation (stddev).

L'utilisation de la distribution normale s'explique par le fait que c'est la génération d'un grand nombre de variables aléatoires indépendantes, suivants une même distribution.

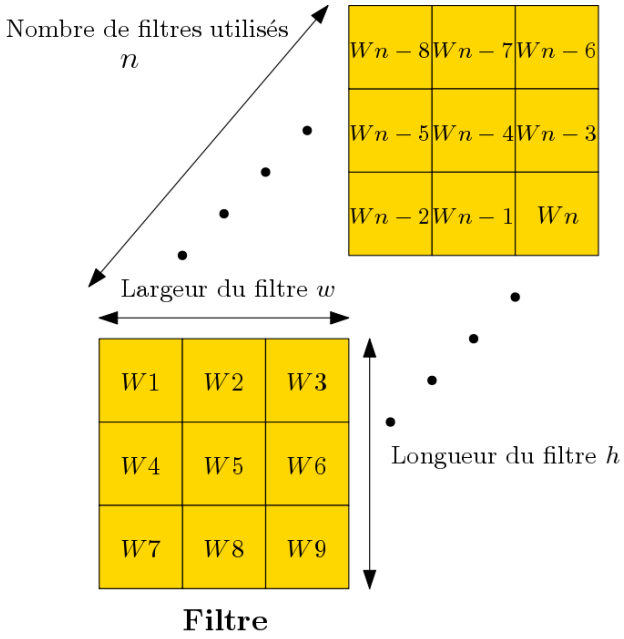

Figure 1.8: Filtres de CNN initialisés par des nombre aléatoires.

D'après la figure (1.8), il est clair que la classe Random_weights sera responsable de générer $(w \times h) \times n$ valeurs aléatoires.

Description de l'architecture de la classe Random_weights:

Dans un premier temps, on utilise (*"std::normal_distribution::(constructor)"* 2021) pour la construction de nombre aléatoires à partir d'une graine (seed) basée sur le temps.

```
#include <iostream>
#include <chrono>
#include <random>

void Random_weights(double nb_filters, double nb_weights, std::vector<std::vector
    <double>>& nb_tot)
{
  // construct a random generator engine from a time-based seed: Ref{14}
  unsigned seed = std::chrono::system_clock::now().time_since_epoch().count(); //
    time; system real time
  std::default_random_engine generator (seed); //random generator engine

  //( result_type mean = 0.0, result_type stddev = 1.0 )
  std::normal_distribution<double> distribution (0.0,1.0);
  }
```

Dans un second temps, on remplit les matrices de filtres (array de type vector). On utilise la ligne de code "*number = distribution(generator)*" dans l'exemple donné dans (*"std::normal_distribution"* 2021) pour remplir un poid d'un filtre considéré. Ainsi, on ajoute deux boucles à la classe Random_weights definies ci-dessous:

```
for(int ii = 0; ii < nb_filters; ii++) //loop on the total number of filters
{
    std::vector<double> one_filter; //array initialisation with no defined size
    for (int jj = 0; jj < nb_weights; jj++) //nb_weights = filter height * filter
     width
    {
      double number = (distribution(generator)); //random number from a random
    generator engine, Ref[13]
      one_filter.push_back(number); // Filling of 1 filter with random values
    }
    nb_tot.push_back(temp); //Filling of all filters with random values
  }
```

On associe à cette classe le code source **Random_weights.cpp** et l'en-tête **Random_weights.h**.

1.2.4 Classe Pooling_layer

Le but de cette partie est la description de la classe *Pooling_layer*, introduite après la classe de convolution et qui vise a réduire les données à partir de la matrice de convolution passée en entrée.

Generalités et choix:

Il existe differentes types de Pooling. On peut citer le "Max-Pooling", ainsi que l' "Average-Pooling", *"Réseau neuronal convolutif"* 2021.

Le "Max-Pooling" est plus efficace car il permet de maximiser le poids des activations fortes, *"Classification des Images Médicales"* n.d., ce qui justifie le choix du **"Max-Pooling"** dans le projet.

De plus, ce type de pooling offre les avantages suivantes, *"Réseau neuronal convolutif"* 2021, *"Classification des Images Médicales"* n.d.:

- Réduire le sur-apprentissage,
- Gagner le temps de calcul en sous-échantillonnage de l'image (downsampling),
- Conserver les caractéristiques les plus importantes de l'image.

Le "Max-Pooling" est appliquée à la sortie de la couche de convolution, comme un filtre de dimension 2×2 et se déplace avec un stride (pas) $S = 2$, *"Classification des Images Médicales"* n.d., comme illustré dans la figure (1.9).

La matrice obtenue à la sortie de la couche de Max-Pooling est obtenue en cherchant la valeur maximale du pixel dans chaque bloc ($Filter\_Hight \times Filter\_Width$) de la matrice d'entrée. Dans l'exemple illustré dans la figure (1.9), on considère des blocs 2×2 de la matrice d'entrée. La procédure de Max-Pooling est donnée dans la figure (1.10).

A titre d'exemple, on considère le bloc de la matrice d'entrée 4×4. La valeur maximale correspond à 9. Elle sera écrite comme premier élément de la matrice de Max-Pooling. On répète cette procédure pour les autres blocs de la matrice d'entrée.

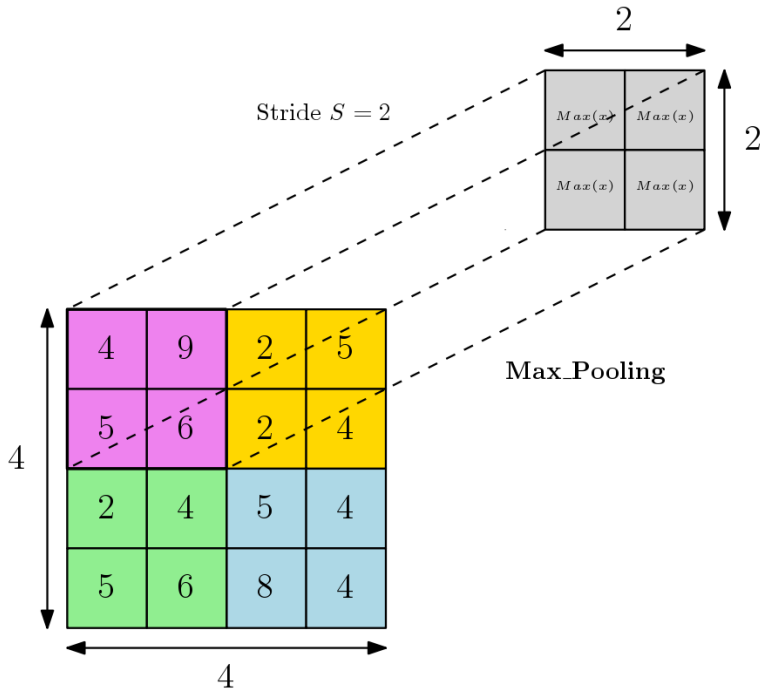

Figure 1.9: Max-Pooling d'une matrice d'entrée 4×4 avec un noyau de Pooling 2×2, stride $S = 2$.

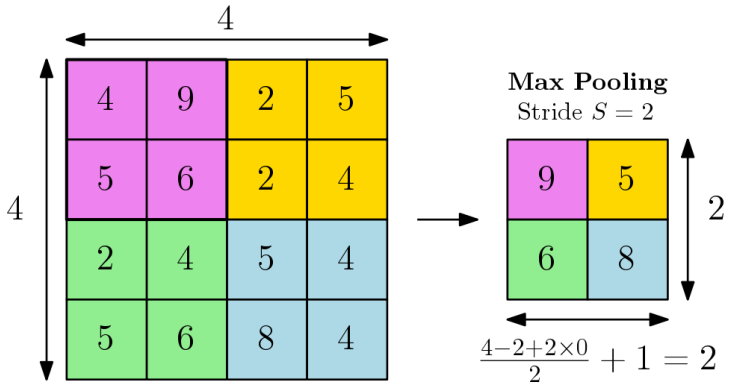

Figure 1.10: Matrice de Max-Pooling à partir d'une matrice d'entrée 4×4, stride $S = 2$.

La matrice de Max-Pooling obtenue en sortie de la couche de pooling est une matrice 2×2, qui:

- compresse la longueur de la matrice d'entrée par un facteur de 2,
- compresse la largeur de la matrice d'entrée par un facteur de 2.

La dimension de sortie de la matrice Max-Pooling est calculée par la même formule que celle utilisée dans la couche de convolution:

$$Largeur_{matrice\ sortie} = \frac{Largeur_{matrice\ d'entrée} - Largeur_{filtre} + 2 \times Padding}{Stride} + 1 \qquad (1.3)$$

Dans notre cas, on obtiendra une matrice Max-Pooling de dimensions 24×24 à partir d'une

matrice de convolution (entrée de la couche de pooling) 48×48, comme illustré dans la figure (1.11). Soit un facteur total de compression de $2 \times 2 = 4$.

Le volume de sortie sera dans ce cas $24 \times 24 \times 8$, où 8 représente le nombre des filtres utilisés dans la couche de convolution.

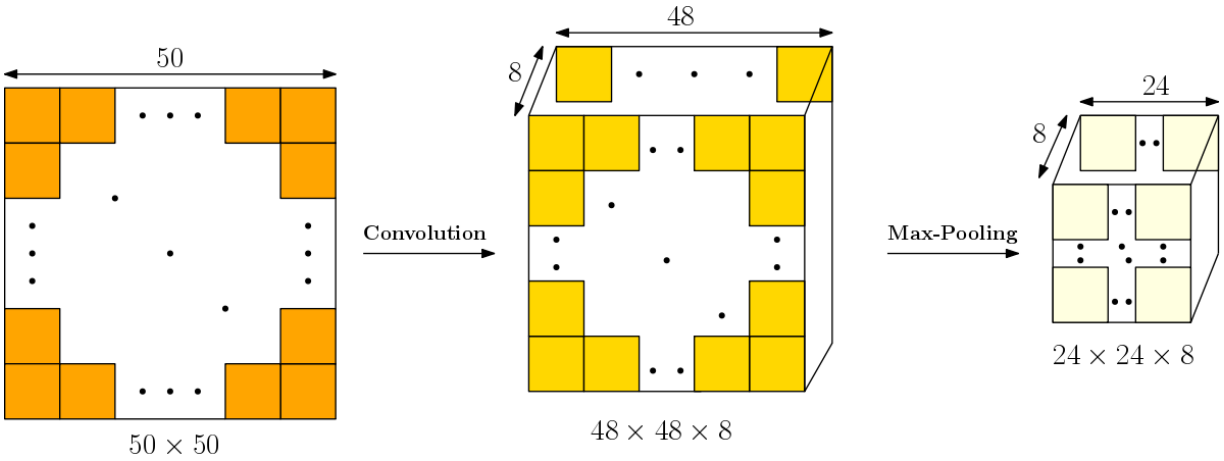

Figure 1.11: Image après passage à la couche de convolution et à la couche de pooling.

1.2.5 Fonction Softmax

Pour compléter le CNN, on doit lui donner la capacité de faire des prédictions.

La couche Softmax est une couche entièrement connectée (dense) qui utilise la fonction Softmax (fonction exponentielle normalisée) comme fonction d'activation.

Il s'agit d'une généralisation de la fonction logistique, *"Fonction softmax"* 2021, qui prend comme entrée un vecteur $z = (z_1, ..., z_K)$ (k réel). La sortie sera un vecteur $\sigma(z)$ de K nombres réels strictement positifs et de somme 1. Ainsi, la fonction de Softmax est donnée par l'équation suivante:

$$\sigma(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)} \qquad \forall j \in \{1, ..., K\} \tag{1.4}$$

Comme on a 5 classes, la couche softmax aura 5 noeuds. Chaque noeud d'entre eux représente une classe de plante et sera connecté à chaque entrée, comme illustré dans la figure (1.12).

Après passage par la couche de softmax, le chiffre représenté par le nœud avec la probabilité la plus élevée sera la sortie du CNN.

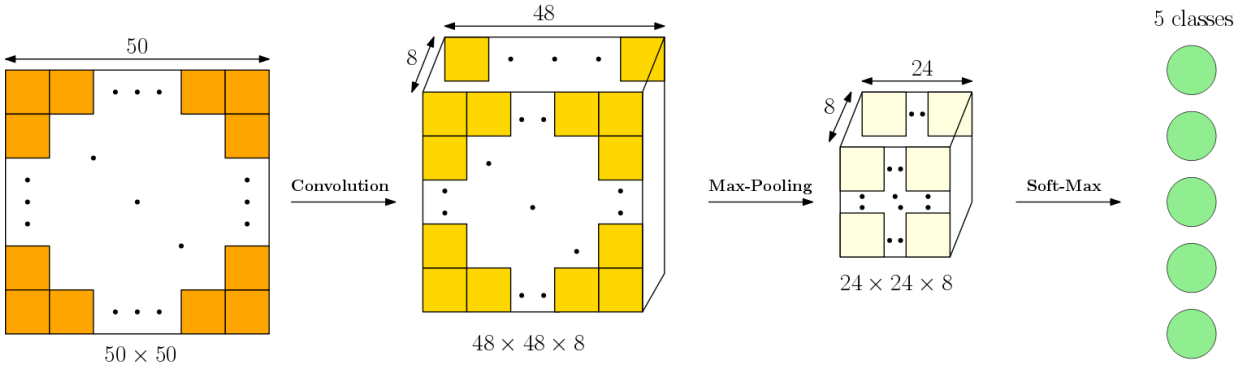

Figure 1.12: Forward-/ Back-propagation du réseau de neurones utilisé.

1.2.6 Backpropagation (Rétropropagation)

Le but de cette partie est la description des étapes de rétropropagation (Backpropagation) utilisée dans les classes *Convolution_layer*, *Pooling_layer* et *Softmax_layer* du project Réseau de Neurones Convolutifs.

a. Définition de la Rétropropagaion (Backpropagation):

La rétropropagation du gradient est utilisée pour entraîner un réseau de neurones. Elle met à jour les poids de chaque neurone, en allant de la dernière couche vers la première, *"Rétropropagation du gradient"* 2021, comme illustré dans la figure (1.13).

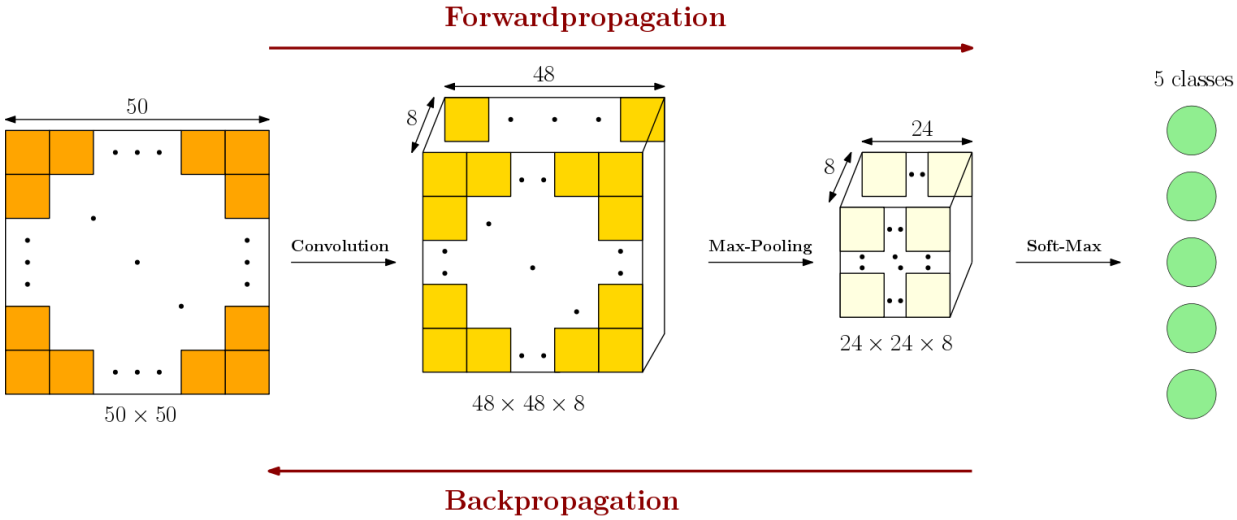

Figure 1.13: Forward-/ Back-propagation du réseau de neurones utilisé.

b. Rétropropagaion pour la couche Softmax:

La couche de Softmax permet de quantifier les prédictions d'appartenance d'une image donnée en entrée à une classe c parmi les 5 classes de sortie. On introduit ainsi la fonction de perte

"**Cross-Entropy Loss**", caracterisée par l'évquation dans (*"BA Friendly Introduction To Cross-Entropy For Machine learning"* 2021):

$$L_{cross-entropy}(\hat{y}, y) = -\sum_{i=1}^{n_{classes}} y_i \times log(\hat{y}_i) \tag{1.5}$$

avec:

$$\begin{cases} n_{classes}: & \text{le nombre de classes,} \\ y_i & \text{la probabilité de choisir une classe (0 ou 1),} \\ \hat{y}_i & \text{la probabilité prédite pour la classe i (par exemple 0.6).} \end{cases}$$

On se place dans le cas où $i = c$, la classe correcte qui doit être choisie. Alors $y_i = 0$ pour tous, sauf la classe correcte. Soit dans ce cas la fonction de perte réduite suivante pour $i = c$:

$$L_{cross-entropy}(\hat{y}_c) = -log(\hat{y}_c) \tag{1.6}$$

où c est la classe correcte et $\hat{y}_c$ est la probabilité prédite pour la classe c.

Le cas idéal sera pour la probabilité $\hat{y}_c = 1$, la fonction de perte $L_{cross-entropy}(\hat{y}_c)$ sera alors nulle.

La phase d'apprentissage d'un réseau de neurones se compose de deux phases:

- Phase "Forward-propagation", où l'entrée passe par les couches du réseau,
- Phase "Back-propagation", où les gradients sont rétropropagés pour mettre à jour les poids.

En se passant d'une couche à une autre dans la phase de "Forward-propagation", chaque couche cache ses données, qui seront utilisés dans la phase "Back-propagation". Ainsi, cette phase doit obligatoirement être précédée par une phase "Forward-propagation".

Par conséquent, au cours de la phase "Back-propagation", on aura:

- Chaque couche reçoit le gradient de perte par rapport à ses sorties $\frac{\partial L}{\partial out}$,
- Chaque couche renvoie le gradient de perte par rapport à ses entrées $\frac{\partial L}{\partial in}$.

On dérive l'équation (1.7) par rapport aux sorties de la couche softmax *out*, qui seront ses entrées dans la phase "Back-propagation". La couche softmax représente un vecteur $1D$ regroupant les 5 classes de fleurs.

Soit:

$$\frac{\partial L}{\partial out(i)} = \begin{cases} 0 & \text{si } i \neq c \\ \frac{-1}{y_i} & \text{si } i = c \end{cases} \tag{1.7}$$

Couches cachées:

On se propose d'utiliser 3 couches cachées, comme illustré dans la figure (1.14):

- L'entrée avant d'être aplatie (before flattening), qui représente dans notre cas un volume de dimension $(24 \times 24 \times 8)$,
- L'entrée après être aplatie (after flattening), qui sera dans ce cas un vecteur $1D$ de dimension (4608×1),
- Les valeurs passées à la fonction d'activation softmax.

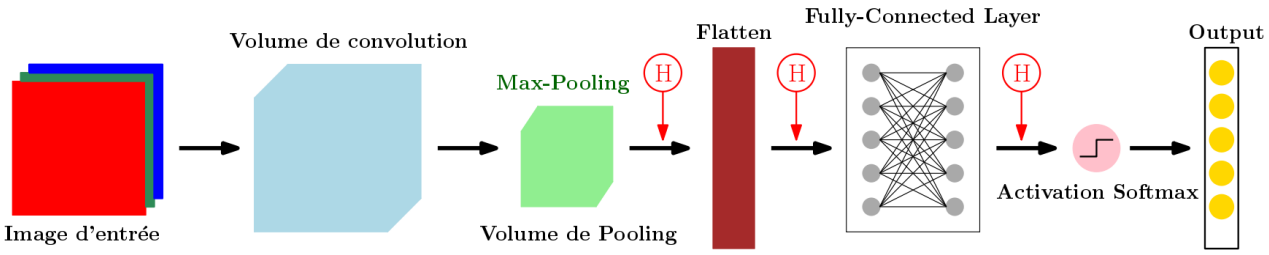

Figure 1.14: Représentation de couches cachées.

On pose t_i le total des classes dans le réseau de neurones. Cherchons maintenant le gradient $\frac{\partial out(c)}{\partial t(i)}$, où c est la classe correcte choisie.

Comme la sortie correspondent à celle de la fonction softmax, alors on a d'après *"Convolutional Neural Networks (CNN): Softmax & Cross-Entropy"* 2018:

$$out(c) = \frac{\exp(t_c)}{\sum_i \exp(t_i)} \tag{1.8}$$

Cette équation indique la probabilité de choisir la classe c parmi la totalité des classes t_i.

Pour simplifier, on pose dans la suite:

$$Sum = \sum_i \exp(t_i) \tag{1.9}$$

Alors, l'équation (1.8) devient:

$$out(c) = \frac{\exp(t_c)}{Sum} \tag{1.10}$$

On distingue deux cas:

- Cas où $i \neq c$: Comme $out(c)$ dépend de Sum, alors en utilisant la règle de la chaîne (Appendix), on aura:

$$\frac{\partial out(c)}{\partial t_i} = \frac{\partial out(c)}{\partial Sum} \times \frac{\partial Sum}{\partial t_i} = \frac{-\exp(t_c)}{Sum^2} \times \exp(t_i) \tag{1.11}$$

- Cas où $i = c$:

  En utilisant la règle de quotient, on aura:

$$\frac{\partial out(c)}{\partial t_c} = \frac{\partial}{\partial t_c}(\frac{\exp(t_c)}{Sum}) = \frac{\exp(t_c) \times Sum - \exp(t_c) \times \frac{\partial Sum}{\partial t_c}}{Sum^2} \tag{1.12}$$

  Notant que $\frac{\partial Sum}{\partial t_c}$ est nulle, sauf dans le cas où $t_i = t_c$, l'équation (1.12) est réduite à:

$$\frac{\partial out(c)}{\partial t_c} = \frac{\exp(t_c) \times (Sum - \exp(t_c))}{Sum^2} \tag{1.13}$$

Maintenant, on cherche à calculer les 3 gradients de perte:

- Le gradient de poids (Weight Gradient) $\frac{\partial L}{\partial W}$ pour mettre à jour les poids,
- Le gradient de Biases (Biases gradient) $\frac{\partial L}{\partial b}$ pour mettre à jour les Biases,
- Le gadient par rapport aux entrées (Input gradient) $\frac{\partial L}{\partial input}$ pour la méthode de rétropropagation, qui va être utilisée dans la couche suivante.

On introduit l'équation depuis (*"Weights and Biases"* 2021), également illustré dans la figure (1.14):

$$T(sortie) = W \times input + b \tag{1.14}$$

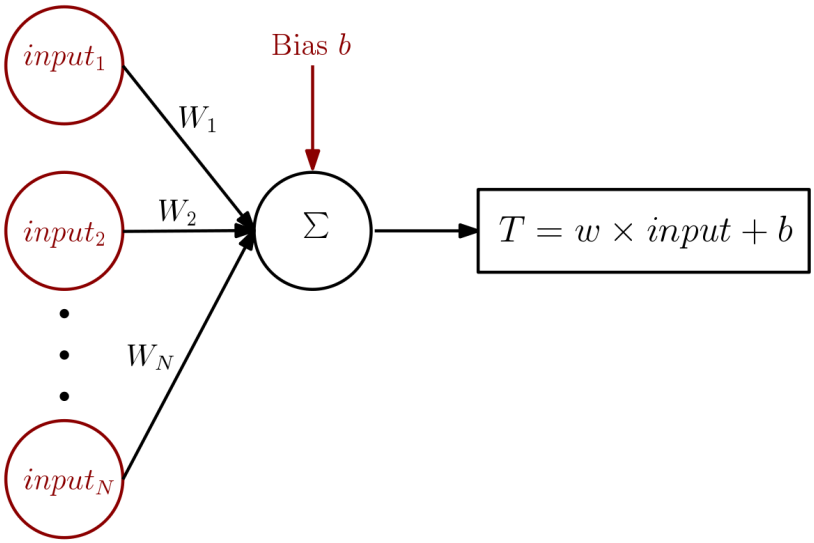

Figure 1.15: Equation reliant les Poids, les Entrées et les Biases.

On dérive l'équation (1.14) par rapport à W, $input$ et b. On obtient:

$$\begin{cases} \frac{\partial T}{\partial W} & = input \\ \frac{\partial T}{\partial b} & = 1 \\ \frac{\partial T}{\partial input} & = W \end{cases} \tag{1.15}$$

En utilisant la règle de la chaîne (Appendix), on aura:

$$\begin{cases} \frac{\partial L}{\partial W} & = \frac{\partial L}{\partial out} \times \frac{\partial out}{\partial T} \times \frac{\partial T}{\partial W} \\ \frac{\partial L}{\partial b} & = \frac{\partial L}{\partial out} \times \frac{\partial out}{\partial T} \times \frac{\partial T}{\partial b} \\ \frac{\partial L}{\partial input} & = \frac{\partial L}{\partial out} \times \frac{\partial out}{\partial T} \times \frac{\partial T}{\partial input} \end{cases} \tag{1.16}$$

c. Rétropropagaion pour la couche Max-Pooling:

Comme la couche de Max-Pooling n'a pas de poids, on ne peut pas faire l'apprentissage. La phase de "Backpropagation" permet de calculer les gradients.

La couche de Max-Pooling permet de réduire le volume obtenue depuis la couche de convolution à

la moitié (comme expliqué précédemment). Il s'agit dans ce cas de la phase "Forwardpropagation".

Contrairement au "Forwardpropagation", la phase de "Backpropagation" permet de doubler la longueur et largeur du gradient de perte.

On considère les illustrations suivantes pour mieux comprendre la phase de "Backpropagation" dans la couche de Max-Pooling.

Soit la matrice 4×4 et la matrice de Max-Pooling illustrées dans la figure (1.16) dans le cas de la phase de "Forwardpropagation".

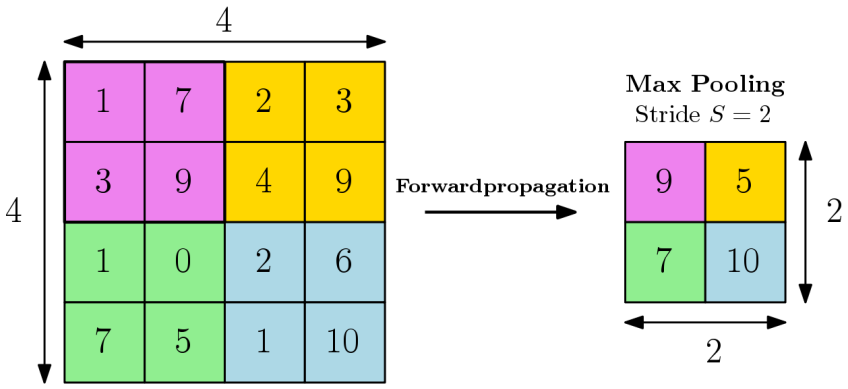

Figure 1.16: Matrice de Max-Pooling dans la phase de "Forwardpropagation".

Supposant maintenant qu'on dispose de la matrice 2×2 contenant les dérivées de L par rapport aux dérivées de *inputs*. Dans la phase de "Backpropagation", on a besoin de créer une matrice 4×4. La position de la dérivée reste la même que celle où on a trouvé la valeur maximale car on a $\frac{\partial L}{\partial input} = 1$, donc $\frac{\partial L}{\partial input} = \frac{\partial L}{\partial output}$. Les autres pixels auront zéro comme valeur, car $\frac{\partial L}{\partial input} = 0$ pour les valeurs non maximales.

En conclusion, on copie le gradient de $\frac{\partial L}{\partial output}$ à $\frac{\partial L}{\partial input}$ dans le cas d'une valeur maximale.

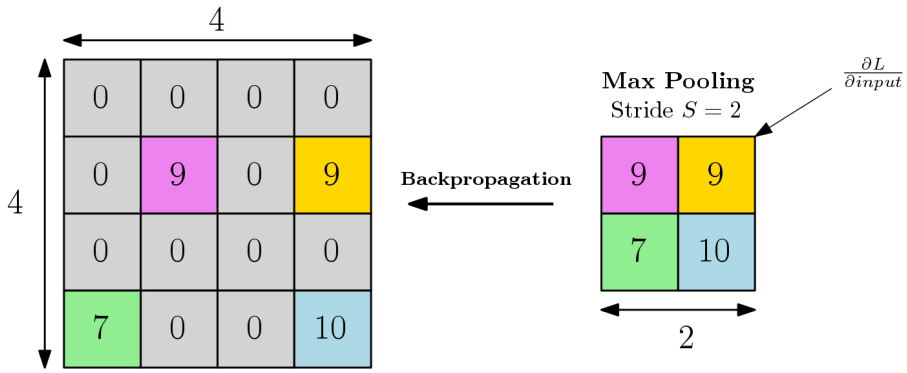

Figure 1.17: Matrice de Convolution obtenue dans la phase de "Backpropagation".

d. Rétropropagaion pour la couche de Convolution:

La rétropropagation est une étape cruciale dans la couche de convolution, car elle permet de mettre à jour les poids des filtres initialisés au debut par des valeurs aléatoires.

On cherche dans cette étape à trouver la derivée $\frac{\partial L}{\partial W}$, qu'on peut la décomposer en utilisant la règle de chaîne (Appendix) de la manière suivante:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial out} \times \frac{\partial out}{\partial W} \tag{1.17}$$

Comme on connait déjà la valeur de la derivée $\frac{\partial L}{\partial out}$, il nous reste que déterminer $\frac{\partial out}{\partial W}$. A ce stade, on se propse la question suivante: Comment le changement des valeurs de poids peut affecter la perte?

Pour y répondre, on se propose de deux cas:

- Cas trivial: On prend comme exemple une image d'entrée 3×3 et un filtre de dimension 3×3 avec des poids tous nulles. On aura dans ce cas une sortie de dimension 1×1 nulle de la couche de convolution (cas de figure (1.18)),

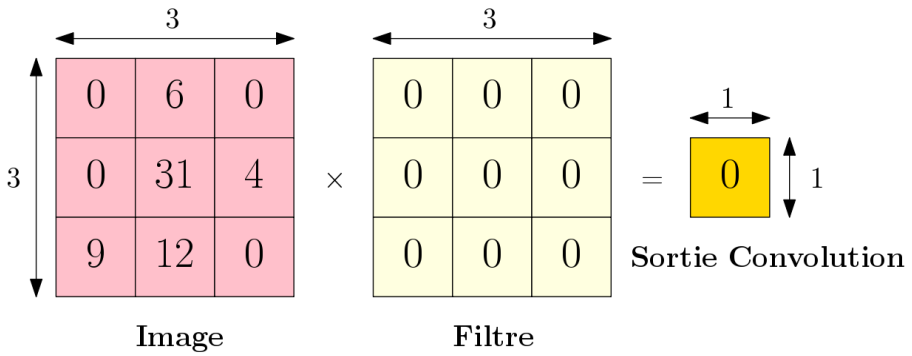

Figure 1.18: Convolution pour des poids de filtre nulles.

- Cas où le poids placé au centre du filtre est non nul: On reprend le même exemple, et on change la valeur du poids placée au centre du filtre à 1. La valeur centrale de la matrice de convolution sera égale à 31 comme illustré dans la figure (1.19). Par conséquent, la derivée $\frac{\partial L}{\partial W}$ obtiendra l'image d'entrée, i.e., $output\_image(i,j) = convolution(Image, Filter)$

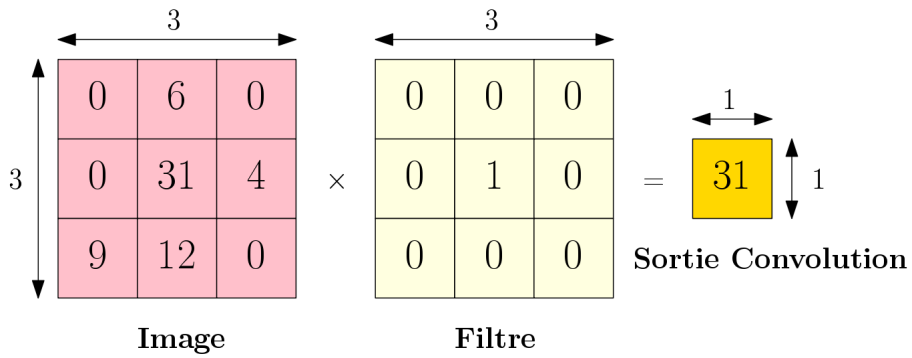

Figure 1.19: Convolution pour un poid au centre du filtre non nul.

D'où l'équation suivante:

$$out(i,j) = convolution(Image, Filtre) = \sum_{x=0}^{n_1=3} \sum_{y=0}^{n_2=3} Image(i+x, j+y) \times Filtre(x,y) \quad (1.18)$$

Finalement, on dérive l'équation (1.18) par rapport aux poids du filtre, on obtient:

$$\frac{\partial out}{\partial W} = Image(i+x, j+y) \quad (1.19)$$

En remplaçant l'équation (1.19) dans l'équation (1.17), on obtient:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Filtre(x,y)} = \sum_i \sum_j \frac{\partial L}{\partial out(i,j)} \times \frac{\partial out(i,j)}{\partial Filtre(x,y)} \quad (1.20)$$

1.2.7 Résultats: Entraînement et test du jeu de données

Train (entrainement): Contient les données d'apprentissage, c'est à dire les données pour lesquels le model sera entrainé.

Test (évaluation): Contient les données permettants de réaliser des tests finaux.

Le jeu de données a été divisé en deux parties:

- Phase d'apprentissage (training), qui se compose de 2965 images,
- Phase de test, qui se compose de 700 images.

Suite à l'exécution du code, on obtient les résultats suivantes:

- Si on teste avec les mêmes jeux de données, les résultats sont bons,
- A la venue d'un nouveau jeu de données, les résultats peuvent avoir une grand différence.

Pour cela, il est important que les données expérimentales soient totalement différentes des données d'apprentissage, afin d'avoir un modèle robuste en sortie.

Après le lancement de l'entraînement et le test pour un nombre d' $epoch = 200$ et un $learning\_rate = 0.005$, on a obtenu une précision de 38.5714% pour un temps d'exécution de 24 heures, comme illustré dans la figure (1.20).

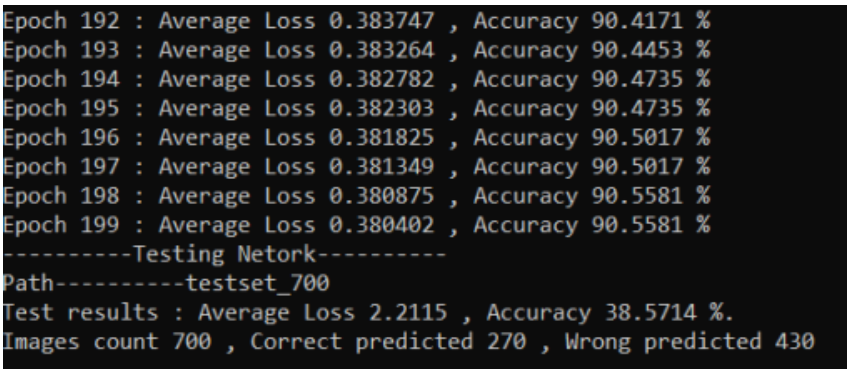

Figure 1.20: Résultats du model pour $epoch = 200$ et $learning\_rate = 0.005$.

On peut avoir plus de précision si on augmente le nombre d' *epochs*, soit $epoch = 600$ ou 700.

On a également essayé de faire les tests pour le cas où $epoch = 600$. Mais, vu que nos machines ne sont pas assez performants, elles se sont plantées. L'exécution nécessite bien de l'espace mémoire que du temps de calcul.

References

"BA Friendly Introduction To Cross-Entropy For Machine learning" (2021). Pianalytix. URL: https://pianalytix.com/a-friendly-introduction-to-cross-entropy-for-machine-learning/.

"Backpropagation in Fully Convolutional Networks" (2021). Towards Data Science. URL: https://towardsdatascience.com/backpropagation-in-fully-convolutional-networks-fcns-1a13b75fb56a.

"Classification des Images Médicales" (n.d.). IMAIOS. URL: https://www.imaios.com/fr/Societe/blog/Classification-des-images-medicales-comprendre-le-reseau-de-neurones-convolutifs-CNN.

"Convolutional neural network" (2021). Wikipedia. URL: https://en.wikipedia.org/wiki/Convolutional%5C_neural_network.

"Convolutional Neural Networks (CNN): Softmax & Cross-Entropy" (2018). Super Data Science Team. URL: https://www.andreaperlato.com/aipost/cnn-and-softmax/.

"Deciding optimal kernel size for CNN" (2018). Towards Data Science. URL: https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363.

"Deep Neural Network: Qu'est-ce qu'un réseau de neurones profond ?" (2021). Data Scientest. URL: https://datascientest.com/deep-neural-network.

"Fonction d'activation" (2021). Inside Machine Learning. URL: https://inside-machinelearning.com/fonction-dactivation-comment-ca-marche-une-explication-simple/.

"Fonction softmax" (2021). Wikipedia. URL: https://fr.wikipedia.org/wiki/Fonction_softmax.

"Les Réseaux de Neurones artificiels" (2018). Juri' Predis. URL: https://www.juripredis.com/fr/blog/id-19-demystifier-le-machine-learning-partie-2-les-reseaux-de-neurones-artificiels.

"Lire et afficher une image dans OpenCV en utilisant C++" (2022). Acervo Lima. URL: https://fr.acervolima.com/lire-et-afficher-une-image-dans-opencv-en-utilisant-c/.

"OpenCV" (2022). Acervo Lima. URL: https://fr.wikipedia.org/wiki/OpenCV.

"Plantae" (2020). gbif. URL: https://www.gbif.org/species/6.

"PlantVillage Dataset" (2020). Kaggle. URL: https://www.kaggle.com/abdallahalidev/plantvillage-dataset.

"Programmation Generique, Bibliotheque Standard (STL)" (2018). Université de Sorbonne. URL: https://www.lpsm.paris/pageperso/roux/enseignements/1819/ifma/chap05.pdf.

"Réseau neuronal convolutif" (2021). Wikipedia. URL: https://fr.wikipedia.org/wiki/R%5C%C3%5C%A9seau%5C_neuronal%5C_convolutif.

"Réseau neuronal convolutif" (2021). Wikipedia. URL: https://fr.wikipedia.org/wiki/R%5C%C3%5C%A9seau%5C_neuronal%5C_convolutif.

"Réseaux de Neurones" (2013). Statistic. URL: http://www.statsoft.fr/concepts-statistiques/reseaux-de-neurones-automatises/reseaux-de-neurones-automatises.htm#.YdV2udso9H.

"Rétropropagation du gradient" (2021). Wikipedia. URL: https://fr.wikipedia.org/wiki/R%5C%C3%5C%A9tropropagation_du_gradient.

"std::normal_distribution::(constructor)" (2021). cplusplus. URL: https://www.cplusplus.com/reference/random/normal%5C_distribution/normal%5C_distirbution/.

"std::normal_distribution" (2021). cppreference. URL: https://en.cppreference.com/w/cpp/numeric/random/normal%5C_distribution.

"std::normal_distribution" (2021). cplusplus. URL: http://www.cplusplus.com/reference/random/normal%5C_distribution/.

"Weights and Biases" (2021). AI Wiki. URL: https://docs.paperspace.com/machine-learning/wiki/weights-and-biases.

Appendix A

Appendix Chapter

On considère un noeud z d'un réseau de neurones, *"Backpropagation in Fully Convolutional Networks"* 2021.

Règle de la chaîne (Chain Rule):

Il s'agit d'une règle de dérivation, qui permet de calculer les dérivées des fonctions composées.

- On pose dans le cas où z ne dépend que de a:

$$\begin{cases} a(t) = \text{fonction } (t) & \text{différentiable en } t \\ z(a(t)) = \text{fonction } (a(t)) & \text{différentiable en } a \end{cases} \tag{A.1}$$

Alors la derivée de z par rapport à t dans le cas d'une dépendence unique de Z en t est:

$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial a} \times \frac{\partial a}{\partial t} \tag{A.2}$$

Cette procédure est illustrée dans la figure (A.1).

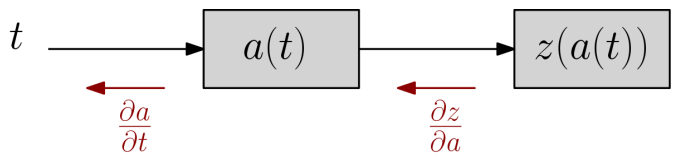

Figure A.1: Règle de la chaîne appliquée à une fonction composée de dépendence simple.

- On pose dans le cas où z dépend que de a_1 et a_2:

$$\begin{cases} a_1(t), a_2(t) = \text{fonction } (t) & \text{différentiable en } t \\ z(a_1(t), a_2(t)) = \text{fonction } (a_1(t), a_2(t)) & \text{différentiable en } a \end{cases} \tag{A.3}$$

Alors la derivée de z par rapport à t dans le cas d'une dépendence unique de Z en t est:

$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial a_1} \times \frac{\partial a_1}{\partial t} + \frac{\partial z}{\partial a_2} \times \frac{\partial a_2}{\partial t} \tag{A.4}$$

Cette procédure est illustrée dans la figure (A.2).

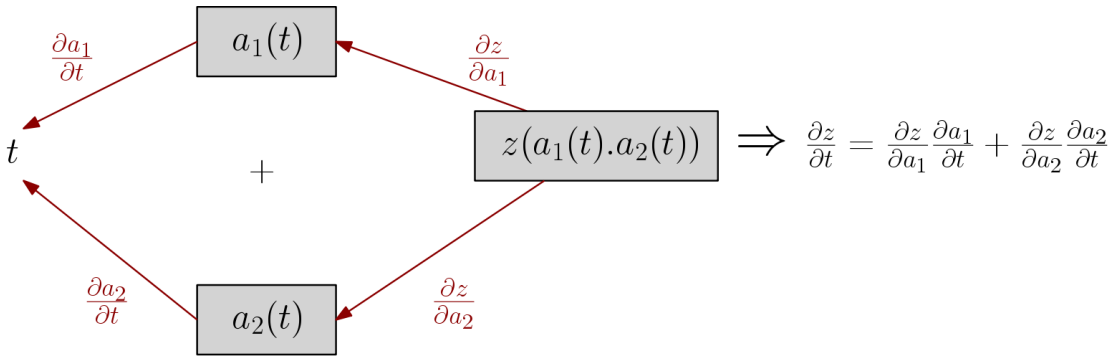

Figure A.2: Règle de la chaîne appliquée à une fonction composée de dépendence multiple.

Gradient:

D'après *"Backpropagation in Fully Convolutional Networks"* 2021, on définit les gradients suivants, également illustrés dans la figure (A.3):

- **Gradient Amont (Upstream gradient):** C'est le gradient que le noeud z reçoit lors de la rétropropagation (Backpropagation),
- **Gradients locaux (Local gradients):** C'est les gradients calculés par rapport aux entrées de z,
- **Gradients Aval (Downstream gradients):** C'est le produit entre le Gradient Amont et Gradients Locaux .

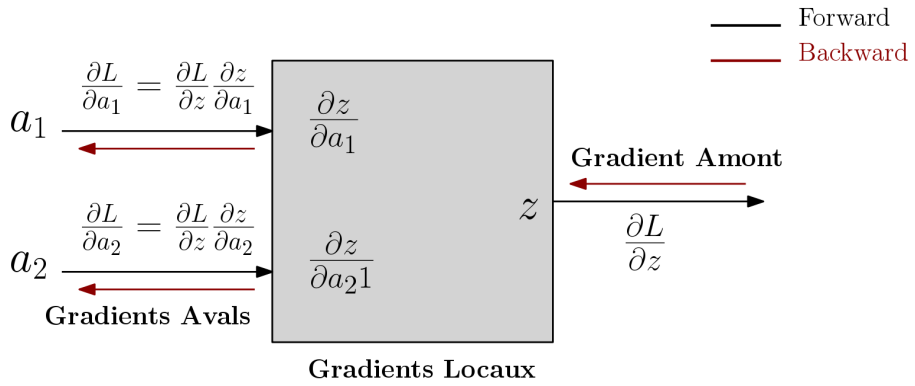

Figure A.3: Formes de Gradients.

Règle de Quotient (Quotient Rule):

Soit deux fonctions f et g dépendantes de t. Alors, on a la dérivée du quotient suivant:

$$\frac{\partial}{\partial t}(\frac{f(t)}{g(t)}) = \frac{f'(t) \times g(t) - f(t) \times g'(t)}{g(t)^2} \tag{A.5}$$

Appendix B

Appendix Chapter

Le Projet est déposé dans le dépot git **"Reconnaissance-de-Plantes-avec-NN"**. Le code SSH de ce dépot est le suivant:

git@github.com:Chaichas/Reconnaissance-de-Plantes-avec-NN.git