République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Abderrahmane Mira - Béjaïa

Faculté de Technologie

Département d'Automatique, Télécommunication et d'Électronique

Projet de Fin d'Étude

Pour l'obtention du diplôme de Master

Filière : Electronique

Spécialité : Instrumentation

Thème

Développement d'outils de vision artificielle embarquée avec Deep Learning : Application à la reconnaissance de panneaux de signalisation routière.

Préparé par :
BELLAGH Jugurtha

Dirigé par :
Dr TIGHZERT Lyes

Soutenu le : 09/11/2021

Examiné par :

Pr MENDIL Boubekeur — Université de Béjaïa

M^me^ MEZZAH Samia — Université de Béjaïa

Année Universitaire : 2020/2021

Remerciement

Je tiens à remercier mon encadreur le **Dr Lyes TIGHZERT** qui a accepté de m'encadrer et de me soutenir le long de ce travail. Je le remercie également pour toute son aide précieuse qui restera à jamais graver sur mes neurones.

J'exprime toute ma reconnaissance et mes remerciements à Monsieur le **Pr Boubekeur MENDIL** de l'Université de Béjaia pour le très grand honneur qu'il me fait en présidant le jury de ce mémoire.

Je tiens à remercier **M^{me} Samia MEZZAH** de l'Université de Béjaia pour avoir accepté d'être jury de ce mémoire.

Je tiens à remercier toute ma famille, et en particulier mes chers parents, pour le soutien psychologique et le matériel qu'ils m'ont toujours offerts.

Je tiens à remercier mes amis Fatah KAHLOUCHE et Abderrahim MECELLEM qui m'ont aidé psychologiquement et matériellement.

Dédicace

A mes très chers parents, à mes frangins Sofiane, Boubekeur et Lyes.

Abstract

This Master's Thesis is focused on development of computer vision's tools intended for embedded systems with using Deep Learning (DL). First, a training of a Convolutional Neural Network (CNN) for road sign recognition is done with Matlab Deep Learning Toolbox by using the Road Sign Detection dataset with four (04) classes containing about one thounsand images. Then, the hardware of a computer vision system is built around a Raspberry Pi 4 Model B and embedded on a 2WD(2 Wheels Driving) mobile robot. Finally, the neural network model obtained on Matlab with training is exported to onnx file format with ONNX library and implemented on the Raspberry Pi 4 with the ONNX Runtime interpreted by Ptyhon 3. The results of training, the image acquisition and inference times, the used memory capacity, the power consumption of the system and the accuracy of the model on the Raspberry Pi are reported in the document.

Keywords: Artificial Intelligence (AI), Deep Learning (DL), Convolutional Neural Network (CNN), Computer Vision (CV), Matlab, Python, ONNX, ONNX Runtime, OpenCV, PyTorch, Raspberry Pi, embedded systems.

Résumé

Ce mémoire de Master est axé sur le développement d'outils de vision artificielle destinée aux systèmes embarqués avec l'usage du Deep Learning (DL). En premier lieu, l'apprentissage d'un réseau de neurones convolutif (CNN) pour la reconnaissance des panneaux de signalisation routière est effectué avec Matlab Deep Learning Toolbox en utilisant la base de données Road Sign Detection avec quatre (04) classes contenant environ un millier d'images. Ensuite, le hardware d'un système de vision par ordinateur est construit autour d'un Raspberry Pi 4 Model B et embarqué sur un robot mobile à deux roues motrices. Enfin, le modèle du réseau de neurones obtenu sur Matlab après apprentissage est exporté en format de fichier onnx avec la librairie ONNX et implémenté sur le Raspberry Pi 4 avec l'environnement d'exécution ONNX Runtime interprété par Python 3. Les résultats de l'apprentissage, les temps d'acquisition et d'inférence des images, la capacité en mémoire utilisée, la consommation énergétique du système et la précision du modèle sur le Raspberry Pi sont rapportés dans le document.

Mots clés : Intelligence Artificielle (IA), Deep Learning (DL), Réseaux de Neurones Convolutifs, Vision par ordinateur, Matlab, Python, ONNX, ONNX Runtime, OpenCV, PyTorch, Raspberry Pi, systèmes embarqués.

Sommaire

Table des figures

Liste des tableaux

Liste des Algorithmes

Listings

Glossaire

2WD	2 Wheels Driving
AI	Artificial Intelligence
BLE	Bluetooth Low Energy
CIA	Central Intelligence Agency
CNN	Convolutional Neural Network
CNNs	Convolutional Neural Networks
CPU	Central Processor Unit
CSI	Camera Serial Interface
CV	Computer Vision
DC	Direct Current
DL	Deep Learning
DSP	Digital Signal Processor
FCN	Fully Convolutional Network
fps	frame per second
GAFAM	Google, Amazon, Facebook, Apple et Microsoft
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GPU	Graphics Processor Unit
GTSRB	German Traffic Sign Road Benchmark
IA	Intelligence Artificielle
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MIPI	Mobile Industry Processor Interface
ML	Machine Learning
MSE	Mean Squared Error

NASA National Qeronautics and Spacial Agency
NLP Natural Language Processing
NSA National Security Agency

ONNX Open Neural Networks eXchange
ORT Open Neural Networks eXchange Runtime
OS Operating System

PSPnet Pyramid Scene Parsing Network
PWM Pulse Width Modulation

RAM Ramdom Access Memory
ReLU Rectified Linear Unit
RGB Red-Green-Blue
RPi Raspberry Pi

SBC Single-Board Computer
SBCs Single-Board Computers
SGD Stochastic Gradient Descent
SGDM Stochastic Gradient Descent with Momentum
SoC System On-Chip

USB Universal Serial Bus

ZIP Zone Improvement Plan

Introduction Générale

Contexte

Les technologies et les sciences contemporaines ne cessent de se développer grâce à l'apparition de nouveaux concepts, moyens, méthodes et approches plus efficaces afin de résoudre un problème quelconque.

Parmi les technologies les plus avancées, on compte l'*intelligence artificielle*. Ce domaine qui semble émerger dans les années cinquante par une question posée par le mathématicien anglais **Alan Mathison Turing** : "La machine peut-elle penser ?", apparue dans son article "Computing Machinery and Intelligence"[1] où il propose l'expérience *"The Imitation Games"* visant à effectuer une classification de deux personnes selon leur sexe par le biais des données des réponses obtenues.

Depuis ses premiers travaux jusqu'aux modèles développés par les géants ***Google, Amazon, Facebook, Apple et Microsoft (GAFAM)***, l'*IA* a connu un parcours d'une considérable expansion et est devenue un domaine multidisciplinaire, actuellement, la puissance de cette science est exploitée dans plusieurs domaines par une multitude d'entreprises comme dans la conduite autonome par ***Tesla***, dans la médecine par ***MaxQ AI***, l'astronomie par la ***NASA*** [2], dans le développement des systèmes autonomes par les ***GAFAM***, dans la cybersécurité par la ***CIA*** et la ***NSA***, et encore dans beaucoup d'autres domaines.

Cette science de prédilection de divers géants de l'industrie moderne est multidisciplinaire et a produit au cours des années un ensemble de branches, de courants de pensées et même de divergences d'opinions au cœur des débats philosophiques et politiques. Le Machine Learning, le Natural Language Processing (NLP) et la planification sont des exemples de disciplines de l'*IA*. Le Deep Learning est un sous-ensemble du Machine Learning qui est consiste essentiellement en l'apprentissage d'un réseau de neurones profond, ce dernier étant composé de trois couches ou plus.

Depuis sa naissance, l'*IA* a pour objectif de résoudre des problèmes posés au quotidien et effectuer des tâches communes à la manière dont un être humaine procède afin de les accomplir. Même si le mode opératoire de *IA* est bien loin de celui dont le cerveau humain fonctionne, elle est tout de même inspirée de ce dernier.

Afin de traiter une information visuelle venant de l'extérieur, une capture d'image et une extraction de l'information utile de l'image capturée doit être faite. Dans le cas de l'être humain, d'une manière assez simple, l'image est perçue par l'œil avant d'être envoyée au cerveau pour le traitement. En *IA*, l'image est acquise par une capteur photographique et l'information est traitée et analysée par une unité de calcul (microprocesseur, microcontrôleur, DSP, etc.).

La vision artificielle vise donc à concevoir un système qui a la capacité d'effectuer cette tâche, ainsi, les réseaux de neurones profonds et les concepts du Deep Learning sont généralement sollicités dans ce cas.

Problématique

La recherche de solutions pour la conduite autonome des véhicules est souvent motivée par le gain de temps et l'amélioration de la sécurité routière. Dans notre cas,

nous allons traiter une partie d'une vision artificielle embarquée sur les véhicules autonomes, donc notre travail sera centré sur le développement d'un outil de reconnaissance de panneaux de signalisation routière.

Présentation de nos approches

L'approche choisie pour résoudre la problématique est l'usage des réseaux de neurones de convolution qui sont un élément incontournable en vision artificielle moderne. Afin d'y parvenir, nous faisons appel aux outils de développement softwares de Matlab pour l'apprentissage du réseau de neurones. Pour une application pratique, nous avons choisi de réaliser un robot mobile avec un hardware Raspberry Pi 4 Model B et une caméra pour une implémentation avec l'écosystème ONNX Runtime.

Organisation du mémoire

Afin d'illustrer cela, notre travail a été porté sur le développement d'outils de vision artificielle embarquée, et a été organisé ainsi :

- le premier chapitre est dédié aux bases et aux rudiments des réseaux de neurones et du Deep Learning.
- le deuxième chapitre est accordé aux réseaux de neurones convolutifs (CNNs).
- le troisième chapitre est consacrée à l'apprentissage d'un modèle CNN pour la reconnaissance de panneaux de signalisation routière.
- le quatrième chapitre aborde l'implémentation du modèle sur l'ordinateur ***Raspberry Pi 4***.

Enfin, les conclusions sur les résultats et le comportement du système implémenté sont tirées en conclusion générale de ce document. En outre, des perspectives d'améliorations des performances du modèle sont ensuite proposées.

Chapitre I

Le B.A.-ba des réseaux de neurones artificiels et du Deep Learning

I.1 Introduction

L'étude des neurones biologiques proposée en 1943 par **Warren McCulloch** et **Walter Pitts** a conduit à l'invention du perceptron en 1958 par **Franck Rosenblatt**. Ainsi, le premier engoûment pour l'Intelligence Artificielle (IA) voit le jour. Le Deep Learning (DL) a été introduit en 1986 dans la communauté du Machine Learning (ML) et les obstacles rencontrés en Deep Learning sont principalement les données d'apprentissage et de validation et la puissance de calcul. Avec le développement d'internet, l'augmentation de la densité d'intégration des circuits intégrés et l'apparition de nouvelles technologies dans l'industrie des CPU et des GPU, le monde a connu une croissance considérable en terme de données et de puissance de calcul dans les années 2000 et encore plus dans les années 2010, ce n'est qu'à ce moment-là que le Deep Learning prend son envol grâce à la compétition ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

I.2 Définitions

I.2.1 Machine Learning

Le Machine Learning (ML) est un sous-domaine de l'Intelligence Artificielle (IA). Il est connu en français sous le nom de "Apprentissage Automatique". C'est une technique de création d'un modèle en se reposant sur des données, les données peuvent être, entre autres, des images, du son ou du texte [3].

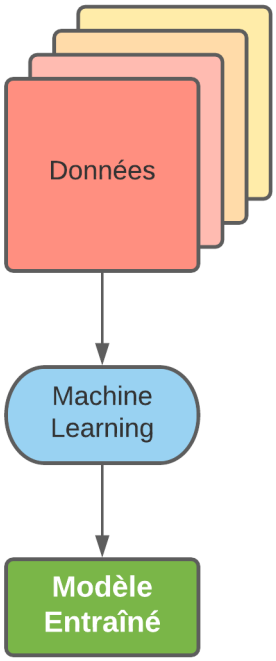


FIGURE I.1 : Processus d'apprentissage en Machine Learning.

Il existe quatre familles de types d'approches utilisées en ML [4] :

- Apprentissage basé sur l'information.
- Apprentissage basé sur la similarité.

- Apprentissage basé sur la probabilité.
- Apprentissage basé sur l'erreur.

I.2.2 Deep Learning

Le Deep Learning (DL) est une discipline qui apparient au Machine Learning (ML), qui à son tour fait partie de l'Intelligence Artificielle (IA). Le DL consiste à modéliser et entrainer un réseau de neurones profond. Un réseau de neurones profond est composé d'une couche d'entrée où les données sont insérées dans le réseau. Une couche de sortie où les prédictions sont faites. Et finalement une ou plusieurs couches intermédiaires entre la couche d'entrée et la couche de sortie, appelées aussi couches cachées (hidden layers).

I.2.3 La vision artificielle

La vision artificielle ou "Computer Vision", est un domaine qui permet de concevoir des systèmes sur des machines qui seront en mesure d'analyser et d'interpréter des images et/ou des vidéos numériques. Par conséquent, certaines tâches peuvent être automatisées [5].

I.3 Le perceptron

Le perceptron ou neurone artificiel est une conception artificielle inspirée du neurone biologique par le biais de moyens électriques ou informatiques. Un neurone biologique est une cellule capable recevoir une information via des terminaisons appelées : "dendrites". Les informations reçues sont alors traitées à l'intérieure de la cellule et ensuite un signal électrique est envoyé via l'axone à d'autres neurones [6]

(la Figure I.2). Le synapse est une structure permettant de transmettre un signal d'un neurone à un autre. L'activité synaptique est mesurée par la force synaptique [7].

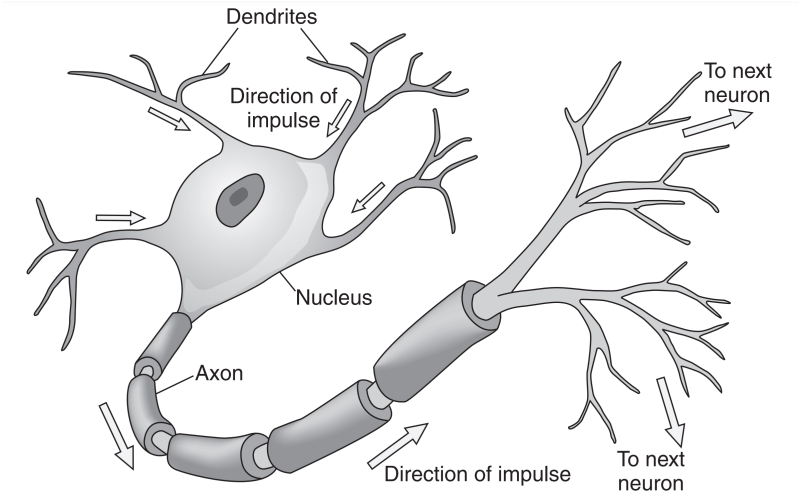


FIGURE I.2 : Schéma basique du neurone biologique [6].

Le neurone artificiel est mathématiquement une fonction. L'analogie entre le neurone biologique et le neurone artificiel peut être décrite comme étant les dendrites sont les entrées x_i, l'axone est la sortie y et les forces synaptiques sont les poids w_i.

La Figure I.3 illustre le modèle simple d'un neurone artificiel. Les entrées x_i sont multipliées par les poids w_i, puis les produits résultants sont additionnés, un biais b ("bias") (qui permet de décaler la fonction verticalement) est ajouté au résultat, le tout est passé dans une fonction d'activation qui produira la sortie y [3].

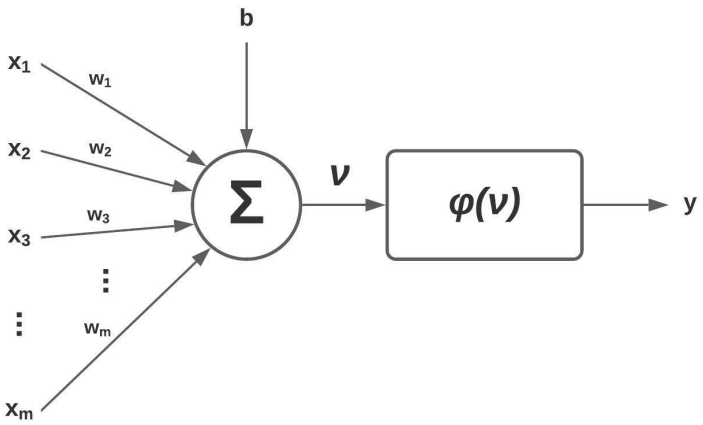


FIGURE I.3 : Diagramme du perceptron.

L'expression de la somme ν est exprimée comme suit :

$$\nu = \left(\sum_{i=1}^{m} w_i \times x_i \right) + b \tag{I.1}$$

La sortie y est alors produite grâce à la fonction d'activation φ :

$$y = \varphi(\nu) \tag{I.2}$$

Les poids w et les entrées x peuvent être exprimés sous forme matricielle, soit :

$$W = \begin{bmatrix} w_1 & w_2 & \dots & w_m \end{bmatrix} \text{ et } X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}.$$

Le résultat ν sera alors exprimé :

$$\nu = \begin{bmatrix} w_1 & w_2 & \dots & w_m \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + b = W \times X + b \tag{I.3}$$

I.4 Les réseaux de neurones

Un réseau de neurones est un ensemble de neurones inter-connectés d'une manière précise. Ils sont organisés sous forme de couches séparant les entrées des sorties du réseau.

I.4.1 Les couches d'un réseau de neurones

Une couche dans un réseau de neurones est une superposition de plusieurs neurones. Les données entrantes dans chaque neurone sont multipliées par un poids (La Figure I.4).

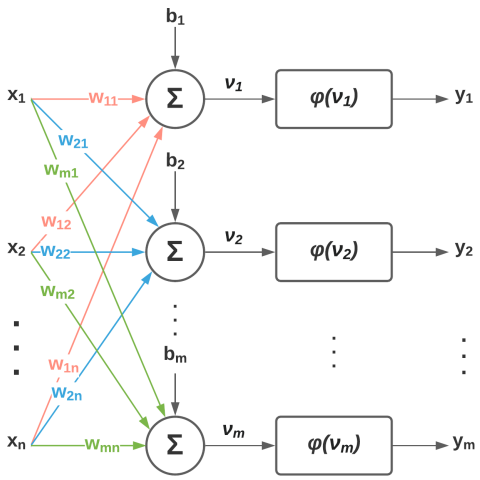


FIGURE I.4 : Une couche d'un réseau de neurones.

Si on reprend l'équation I.1, la somme des ν_i sera :

$$\forall i \in \{1, \ldots, m\}, \quad \nu_i = \left(\sum_{j=1}^{n} w_{ij} \times x_j \right) + b_i \tag{I.4}$$

Et les sorties y_i seront :

$$\forall i \in \{1, \ldots, m\}, \quad y_i = \varphi(\nu_i) \tag{I.5}$$

Soit n la dimension du vecteur X (nombre de données en entrée), et m le nombre de neurones dans une couche donnée, les expressions précédentes peuvent être écrites sous la forme matricielle, si W est la matrice des poids et N est le vecteur de sortie composé d'éléments ν_i :

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \ldots & w_{mn} \end{bmatrix} \tag{I.6}$$

$$N = W \times X + B \tag{I.7}$$

$$\begin{bmatrix} \nu_1 \\ \nu_2 \\ \vdots \\ \nu_m \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \ldots & w_{mn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{I.8}$$

Le vecteur de sortie Y sera exprimé ainsi :

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} \varphi(\nu_1) \\ \varphi(\nu_2) \\ \vdots \\ \varphi(\nu_m) \end{bmatrix} \tag{I.9}$$

I.5 Les fonctions d'activation communément utilisées

Les fonctions d'activation sont des fonctions mathématiques qui permettent de produire une sortie y en fonction de la somme ν_m des entrées x pondérées par des poids w.

L'usage des fonctions non-linéaires est nécessaire dans les réseaux de neurones modernes pour des raisons de différentiabilité, c.-à-d. lors de l'apprentissage d'un réseau de neurones, l'erreur se propage de l'entrée vers la sortie, et les sorties des fonctions d'activation doivent être différentiables [8] afin de pouvoir utiliser des méthodes d'optimisation comme la SGD (Descente de Gradient).

Il existe une multitude de fonctions d'activation. Leur usage diffère en fonction de l'application. Les principales fonctions utilisées sont décrites ci-dessous.

I.5.1 La fonction ReLU

La fonction ReLU ou "Rectified Linear Unit" est une fonction d'activation non-linéaire très utilisée en DL dont l'expression mathématique est décrite comme suit :

$$f_{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{I.10}$$

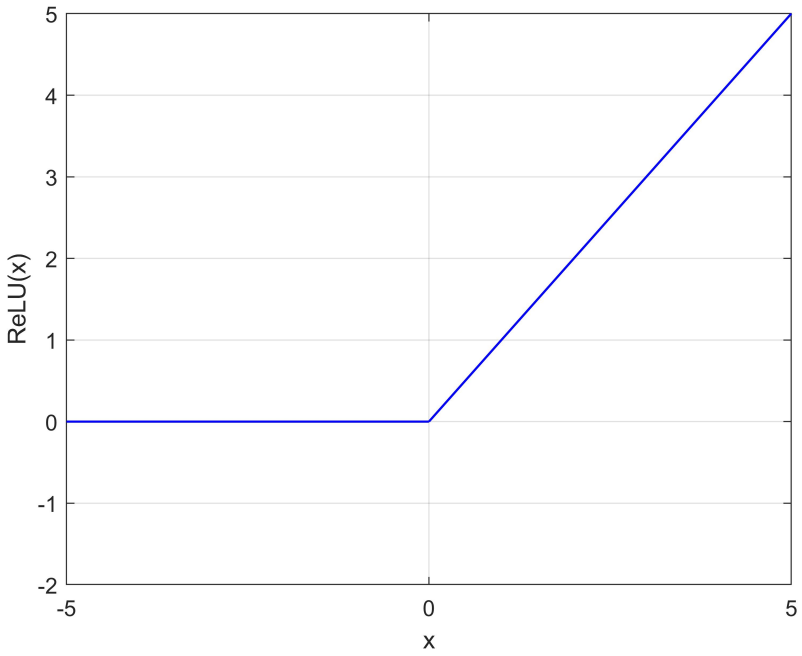


FIGURE I.5 : Courbe de la fonction ReLU.

I.5.2 La fonction Sigmoïde ou fonction Logistique

La sortie de la fonction Sigmoïde est comprise entre 0 et 1. A cet effet, la fonction est utilisée pour prédire une probabilité notamment dans la classification. Son expression mathématique est :

$$f_{sig}(x) = \frac{1}{1 + e^{-x}} \tag{I.11}$$

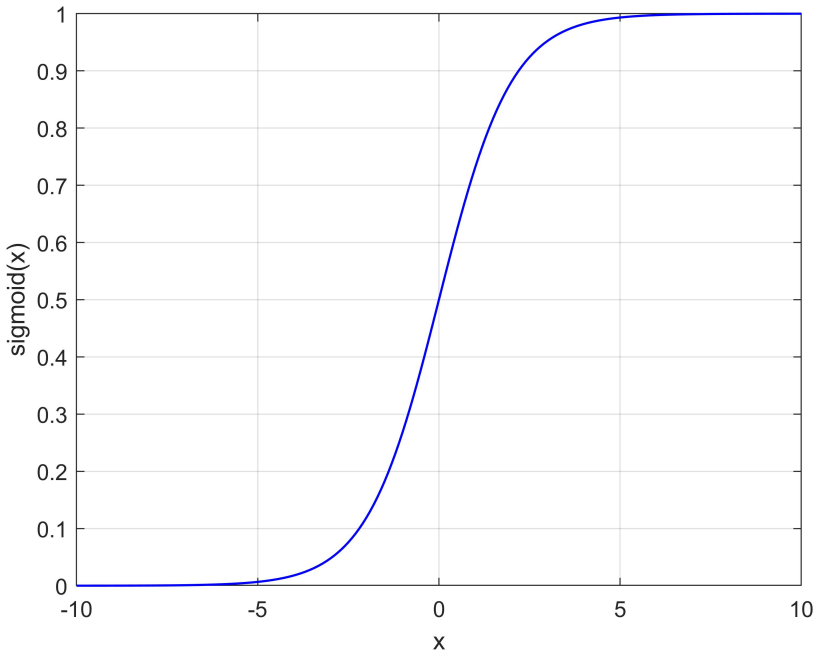


FIGURE I.6 : La courbe de la fonction Sigmoïde.

I.5.3 La fonction tangente hyperbolique

La fonction de la tangente hyperbolique notée tanh est similaires à la fonction Sigmoïde, celle-ci a une sortie comprise entre -1 et 1, la tanh est exprimée par l'équation :

$$f_{tanh}(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{I.12}$$

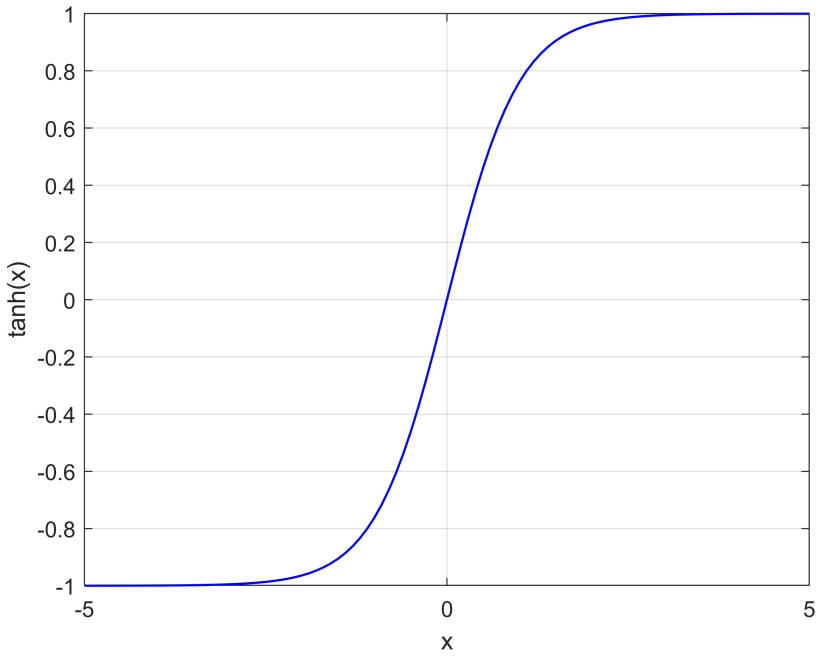


FIGURE I.7 : La courbe de la fonction tangente hyperbolique.

I.5.4 La fonction Softmax

Cette fonction est très utilisée pour la classification et souvent c'est la couche de sortie qui porte cette fonction d'activation. Si n est le nombre de neurones de la couche *Softmax* et $j \in \{1, ..., n\}$, alors son expression est :

$$f_{Softmax}(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \tag{I.13}$$

I.6 Apprentissage d'un réseau de neurones

L'apprentissage d'un réseau de neurones en IA consiste à générer un modèle à travers des algorithmes d'apprentissage en se basant sur des données. On dit que le réseau apprend lui-même de ses expériences [9].

I.6.1 Les types d'apprentissage

Il existe trois principaux types d'apprentissage en réseaux de neurones.

L'apprentissage supervisé

Dans l'apprentissage supervisé d'un réseau de neurones les données d'entrée et les données cibles ("targets") sont préalablement connues.

L'apprentissage non-supervisée

Contrairement à l'apprentissage supervisé, durant l'apprentissage non-supervisé, les données d'entrée sont connues par contre les données cibles sont inconnues.

L'apprentissage par renforcement

L'apprentissage par renforcement est différent des apprentissages cités auparavant. Il consiste pour un agent de prendre des décisions selon des expériences rencontrées. Et selon les décisions prises par l'agent, l'environnement dans lequel il est plongé, lui accorde des récompenses positives ou négatives.

I.6.2 Propagation directe

Durant l'apprentissage, faire propager les données depuis l'entrée d'un réseau de neurones à sa sortie est appelé : "Propagation directe". Les sorties de chaque couche du réseau sont calculées en fonctions de ses poids et des sorties de la couche précédente (Équations I.8 et I.9), et ce jusqu'à la dernière couche de sortie.

I.6.3 Algorithme de rétro-propagation

Durant l'apprentissage d'un réseau de neurones, il est généralement nécessaire d'effectuer une opération appelée : "Backpropagation" ou "Delta Rule", en français "Rétro-propagation". Celle-ci permet de mettre à jour à chaque itération les valeurs des poids connectés entre les nœuds (neurones). La valeur à ajouter ou à soustraire de la valeur existante d'un poids dépend de la valeur de l'erreur calculée lors de la propagation et du critère d'optimisation.

La fonction Coût

La fonction coût est la fonction qui permet de calculer l'erreur commise par le réseau lors de la propagation directe de la donnée. Elle est calculée en fonction de la valeur prédite et la valeur attendue.

La fonction coût est souvent assimilée à une fonction appelée "Loss Function". La fonction Loss est calculée à partir d'une seule propagation (itération), et la fonction coût à partir d'un mini-batch ou d'une base de donnée complète, c'est la moyenne de l'ensemble des fonctions Loss pour l'ensemble de données.

Il existe de multiples fonctions Loss dépendant de l'application. La fonction de Régression est utilisée pour la prédiction de valeurs continues. La fonction Loss de classification binaire pour classifier deux classes et la fonction Loss Multi-classification pour plusieurs classes.

La fonction coût Mean Squared Error (MSE) est est l'une des fonctions les plus utilisées en ML. Elle est calculée ainsi :

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y - \hat{y})^2 \tag{I.14}$$

où m est le nombre de données propagées (taille du mini-batch ou de la base de données), y est la valeur prédite et $\hat{y}$ est la valeur attendue.

En considérant le réseau de neurones multicouche illustré sur la Figure I.8 suivante :

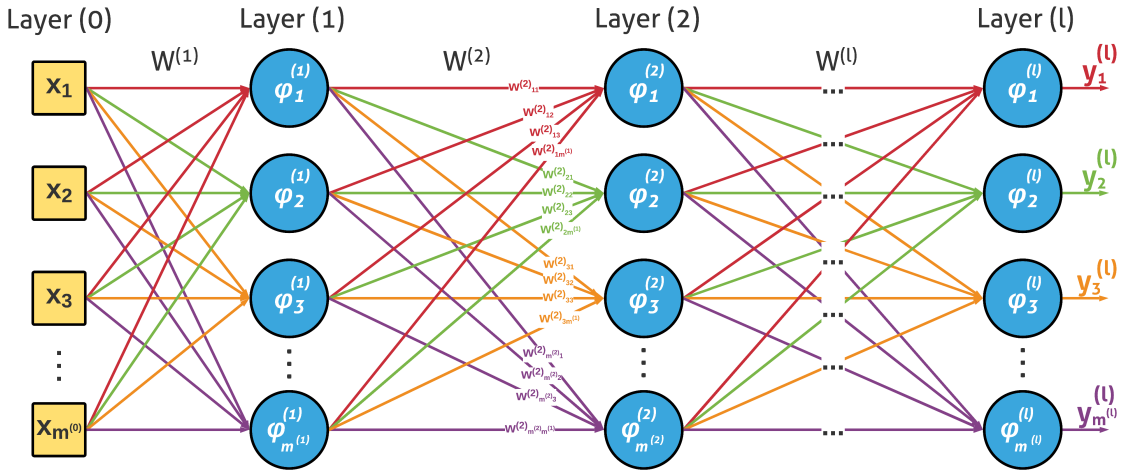


FIGURE I.8 : Inférence des données à travers un réseau de neurones multicouche.

L'algorithme 1 suivant montre l'opération d'apprentissage d'un réseau de neurones multicouche avec la rétro-propagation :

Algorithme 1 : Algorithme d'apprentissage avec rétro-propagation

Input : $learningRate : \alpha, dataInputs : \mathcal{D} \leftarrow \{(\mathbf{X_p}, \mathbf{d_p})\}$,
$numberOfEpochs : epochs$

Output : $W^{(k)}$ `// k is the layer position`

for $k \leftarrow 1$ **to** l **do**
  $initWeights(W^{(k)})$ `// Weights initialization`
end
for $q \leftarrow 1$ **to** $epochs$ **do**
  foreach $(\mathbf{X_p}, \mathbf{d_p})$ ***in*** $\mathcal{D} = \{(\mathbf{X_p}, \mathbf{d_p})\}$ ***as*** (X, d) **do**
    `/* Forward propagation */`
    $Y^{(0)} = X$
    for $k \leftarrow 1$ **to** l **do**
      $Y^{(k)} = \Phi^{(k)}(W^{(k)} \times Y^{(k-1)})$
    end
    `/* Backpropagation */`
    for $i \leftarrow 1$ **to** $m^{(l)}$ **do**
      $e_i^{(l)} = d_i - y_i^{(l)}$
      $\delta_i^{(l)} = \varphi_i'^{(l)} \left(\sum_{j=1}^{m^{(n-1)}} w_{ij}^{(l)} \times y_i^{(l-1)} \right) \times e_i^{(l)}$
    end
    for $k \leftarrow l - 1$ **to** 1 **do**
      for $j \leftarrow 1$ **to** $m^{(k)}$ **do**
        $e_j^{(k)} = \sum_{i=1}^{m^{(k+1)}} w_{ij} \times \delta_i^{(k+1)}$
        $\delta_j^{(k)} = \varphi_j'^{(k)} \left(\sum_{i=1}^{m^{(k-1)}} w_{ij}^{(k)} \times y_i^{(k-1)} \right) \times e_j^{(k)}$
      end
    end
    for $k \leftarrow 1$ **to** l **do**
      for $i \leftarrow 1$ **to** $m^{(k)}$ **do**
        for $j \leftarrow 1$ **to** $m^{(k-1)}$ **do**
          $\Delta w_{ij}^{(k)} = \alpha \times \delta_i^{(k)} \times y_j^{(k-1)}$
          $w_{ij}^{(k)} = w_{ij}^{(k)} + \Delta w_{ij}^{(k)}$ `// Weights update`
        end
      end
    end
  end
end

Où l est le nombre de couches du réseau, k est la position de la couche. La couche (0) (Layer (0)) désigne le vecteur d'entrée. Le m est le nombre de nœuds d'une couche donnée, c.-à-d. $m^{(k)}$ est le nombre de neurones de la couche k (La Figure I.8 illustre la disposition des couches et des neurones). Les majuscules sont des vecteurs ou matrices des variables en minuscules, c.-à-d. si $i \in \{1, ..., m\}$ et pour une couche donnée k :

$$Y^{(k)} = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_m^{(k)} \end{bmatrix} \tag{I.15}$$

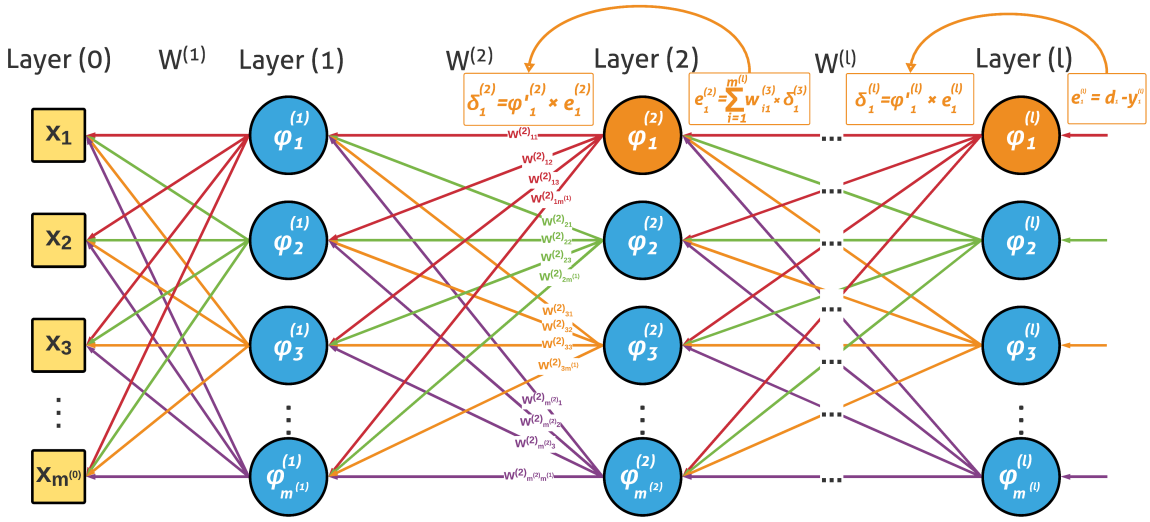


FIGURE I.9 : Calcul des erreurs en rétro-propagation (Exemple de calculs sur les neurones Oranges).

I.7 Conclusion

Dans ce chapitre, nous avons abordé les définitions et les connaissances élémentaires des réseaux de neurones artificiels, particulièrement les réseaux de

neurones profonds. En outre, nous avons présenté le fonctionnement de ces réseaux ainsi que les mécanismes régissant leur apprentissage. Les notions acquises dans ce chapitre nous permettront de mieux aborder les réseaux de neurones convolutifs dans le Chapitre II ci-après.

Chapitre II

Les réseaux de neurones convolutifs

II.1 Introduction

Les réseaux de neurones convolutifs ou Convolutional Neural Networks (CNNs) est le type de réseaux de neurones le plus utilisés en vision artificielle. Leur particularité réside dans le fait qu'ils utilisent des filtres de convolution afin d'extraire les caractéristiques d'une image. Les couches de convolution peuvent apparaître à n'importe quelle couche du réseau. Les CNNs sont utilisés pour la reconnaissance et la classification, l'analyse vidéo, recherche de médicaments et aussi la segmentation.

II.2 Définition

Cette architecture est determinée par la dimension de la couche d'entrée, le nombre, l'ordre et la nature des couches composant le réseau de neurones profond. Les CNNs peuvent être implémentés suivant une architecture précise. La dimension des la couche d'entrée, la succession, l'ordre et la nature des couches qui compose le

réseau profond détermineront le type d'architecture. Chaque architecture est plus ou moins spécifique pour une application donnée.

II.3 Les couches d'un réseau de convolution

II.3.1 La couche de convolution

La couche de convolution est une couche qui produit une carte de caractéristiques ("Feature map output") selon un certain nombre de filtres ("Kernels" en anglais). La sortie est une matrice, chacun de ses éléments est calculé à partir de la somme de la somme des multiplications des éléments du filtre par les éléments de la matrice d'entrée, chaque élément du résultat est trouvé en fonction de la position du filtre [10]. Un filtre 2D est mathématiquement une matrice F de dimensions $d_w \times d_h$. L'entrée est une matrice X de dimensions $w \times h$, et pour $i \in [1, k]$, et Y_i étant le résultat de convolution, les dimensions $w_{Conv} \times h_{Conv}$ de Y_i :

$$w_{Conv} = \left\lfloor \frac{w - d_w + s}{s} \right\rfloor \tag{II.1}$$

$$h_{Conv} = \left\lfloor \frac{h - d_h + s}{s} \right\rfloor \tag{II.2}$$

Où s est le "Stride". Ce concept va être présenté après la convolution simple qui utilise un Stride de 1.

Pour $\forall i \in [1, w]$ et $\forall j \in [1, h]$, le résultat de convolution est calculé ainsi :

$$y_{ij} = \sum_{u=1}^{d_h} \sum_{v=1}^{d_w} f_{uv} \times x_{[u+i-1][v+j-1]} \tag{II.3}$$

Où p est appelé "Padding" qui est un nombre de lignes et de colonnes dont la valeur des pixels sont égales à 0 qui sont ajoutées autour de l'image. La Figure II.1 ci-dessous illustre un Padding de 1 appliqué sur une matrice de dimensions 4×4.

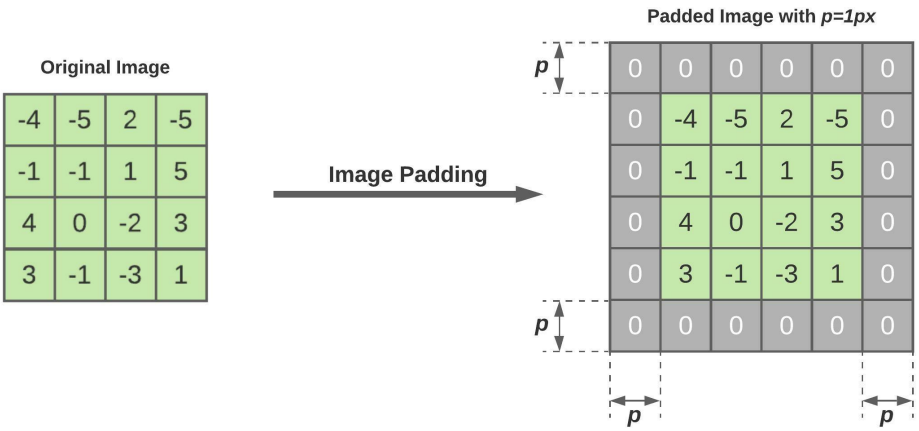


FIGURE II.1 : Le padding de 1 appliqué sur une matrice.

La Figure II.2 suivante illustre l'opération de convolution où l'élément de la matrice de sortie en *bleu* est calculé en sommant les multiplications des éléments du filtre en *vert* par les éléments adjacents de la même région de la matrice d'entrée en *orange*. A chaque étape, le filtre est glissé d'un cran pour calculer l'élément correspondant [10].

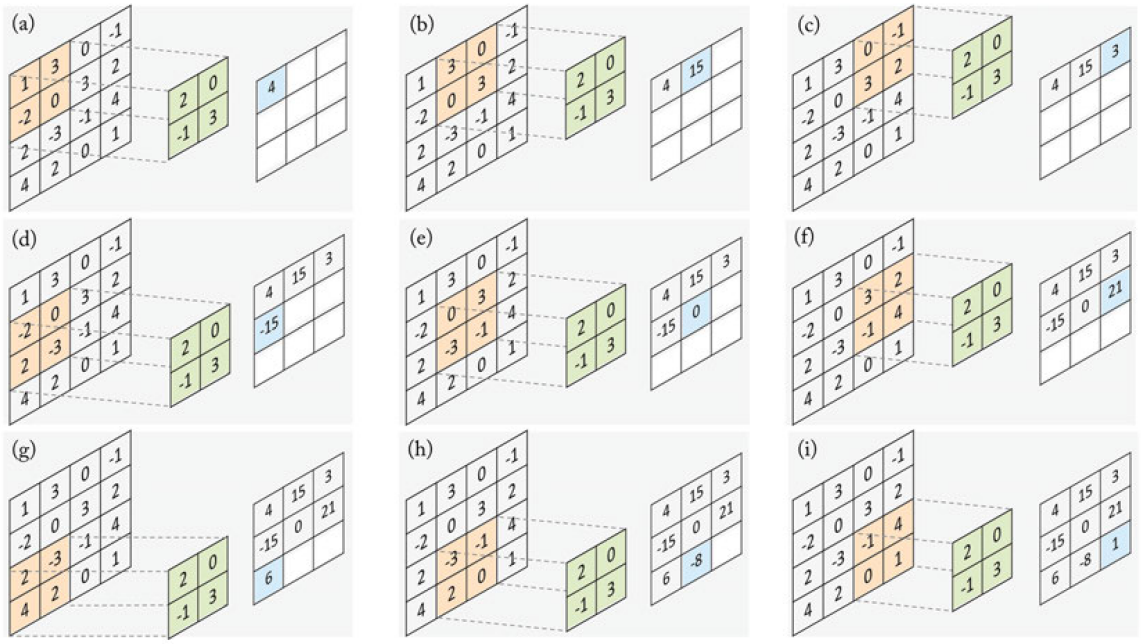


FIGURE II.2 : L'opération de convolution avec un filtre 2×2.

Une opération appelée : sous-échantillonnage ("sub-sampling") peut être appliquée en convolution. Elle consiste à faire glisser à chaque étape le filtre d'un certain nombre de crans, horizontalement et verticalement, ce nombre est appelé : "Stride". L'application de cette opération permet de réduire la dimension de la matrice de sortie en augmentant le Stride [10]. Le Stride de l'opération présentée sur la Figure II.2 est égal à 1 puisque le filtre est glissé d'un seul cran à chaque étape.

Les résultat de la Figure II.2 sont comparés avec ceux de la Figure II.3 où un Stride de 2 est appliqué en gardant la même dimension que celle du filtre précédent.

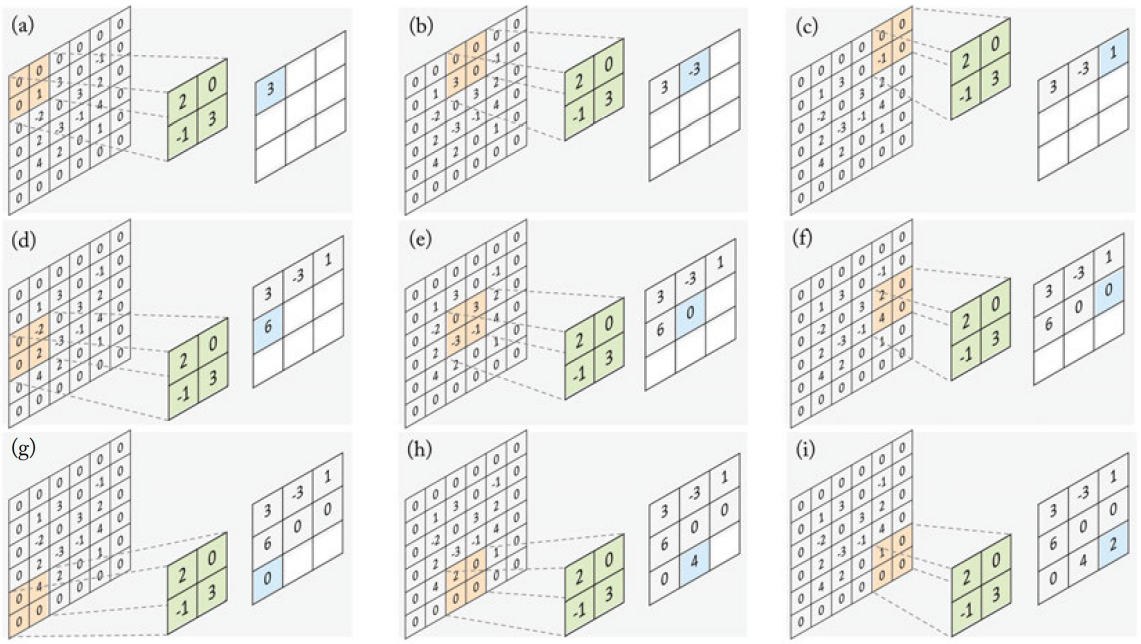


FIGURE II.3 : L'opération de convolution avec un filtre 2×2 et un Stride de 2.

II.3.2 La couche Pooling

La couche Pooling est une technique de sous-échantillonnage qui permet de réduire la taille de l'entrée. Semblablement à l'opération de convolution, le Stride s et la taille t de la région où le Pooling est opéré doivent être spécifiés. La sortie est calculée à partir du maximum ou de la moyenne de la région de Pooling [10].

Si la taille de l'entrée est $w \times h$ de la région de Pooling est $t_w \times t_h$, le Stride est notée s, la taille de la matrice de sortie sera :

$$w_{Pool} = \left\lfloor \frac{w - t_w + s}{s} \right\rfloor \tag{II.4}$$

$$h_{Pool} = \left\lfloor \frac{h - t_h + s}{s} \right\rfloor \tag{II.5}$$

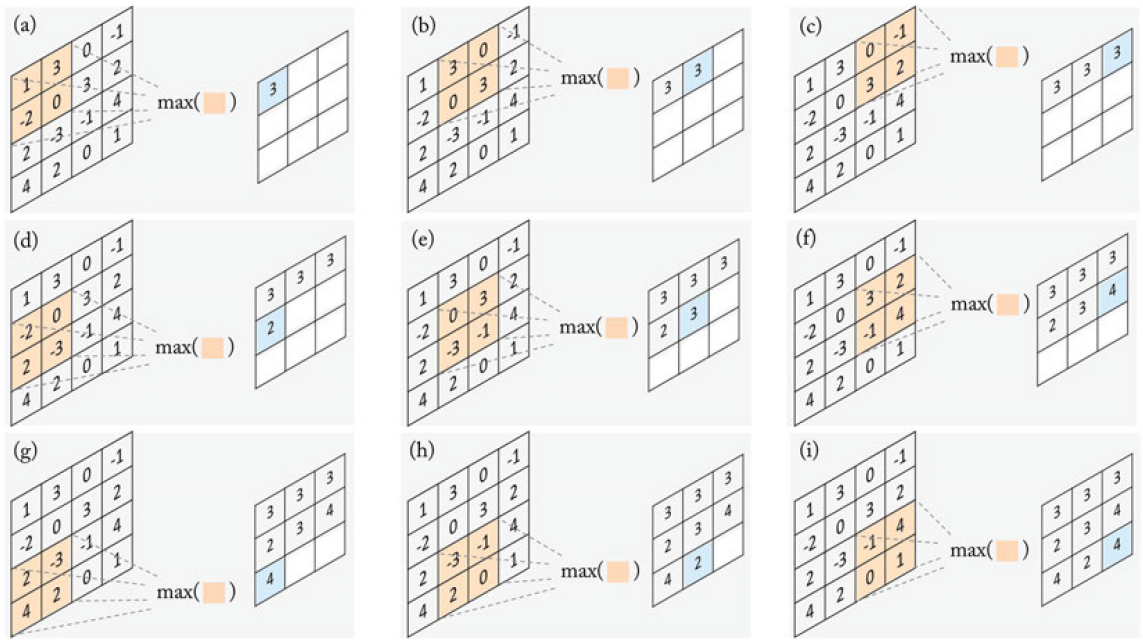


FIGURE II.4 : L'operation de Max Pooling.

Le Max Pooling

Le Max Pooling calcule les éléments de la sortie en retenant uniquement la valeur maximale de la région de Pooling.

Le Mean Pooling

Le Mean Pooling est l'operation qui permet de calculer chaque élément de la sortie par rapport via une région adjacente de l'entrée.

II.3.3 La couche Fully Connected

La couche "Fully Connected" ou entièrement connectée est une couche qui se compose d'un empilement de neurones, chacun est alimenté par des entrées provenant

des sorties de la couche précédente et sa sortie est connectée à tous les neurones de la couche suivante.

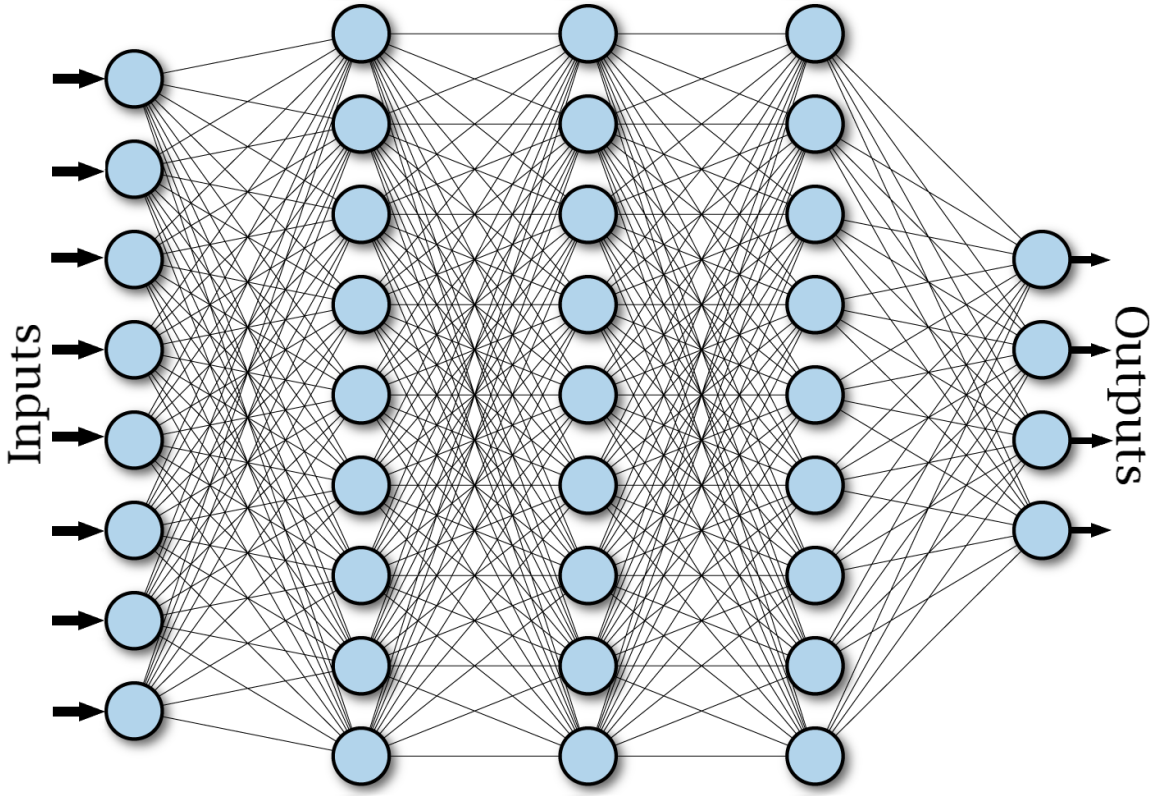


FIGURE II.5 : Réseau profond multicouche avec une couche d'entrée et trois couches cachées Fully Connected et une couche de sortie.

Les couches Fully Connected peuvent être activées par une multitude de fonctions d'activation.

II.4 Les bases de données

L'apprentissage des CNNs nécessite une importante quantité de données, ce sont les "Datasets" (bases de données). Pour des applications en vision artificielle (vision par ordinateur), elles sont généralement organisées sous forme d'images enregistrées

sous un format de fichiers spécifique, et de "Labels" (dans le cas d'un apprentissage supervisé) qui désignent la classe correspondante à l'image. Chaque image est associée à son Label (Classe).

Les images sont généralement divisées en deux catégories, images d'apprentissage et images de test.

Chaque base de données a sa capacité (Nombre d'images contenues) et son nombre de classes (Nombre de Labels).

Parmi les bases de données les plus populaires, on compte MNIST pour la classification des chiffres, CIFAR-10 et CIFAR-100 pour les objets et animaux. Les plus massives sont ImageNet contenant environ quatorze millions d'images, MS-COCO contenant plus de 300000 images avec 80 classes. Ces bases de données peuvent occuper plusieurs centaines de gigaoctets de mémoire sur les supports de stockage de données numériques.

II.5 Architectures des réseaux de neurones convolutifs

Le nombre et la disposition, d'une manière précise, des couches d'un réseau de neurones les unes par rapports aux autres, créent une structure appelée "Architecture Neuronale". Depuis leur création, les CNNs disposent d'une multitude d'architectures. Quelques architectures les plus populaires et leur application sont détaillées ci-dessous.

II.5.1 L'architecture LeNet

Appelé aussi "LeNet-5", cette architecture est utilisée pour la reconnaissance des caractères décimaux manuscrits de 0 jusqu'à 9 (Appliquée sur la base de donnée MNIST). Elle est composée d'une couche d'entrée prenant des données de dimensions

32×32 avec une seule chaine (niveau de gris) [11].

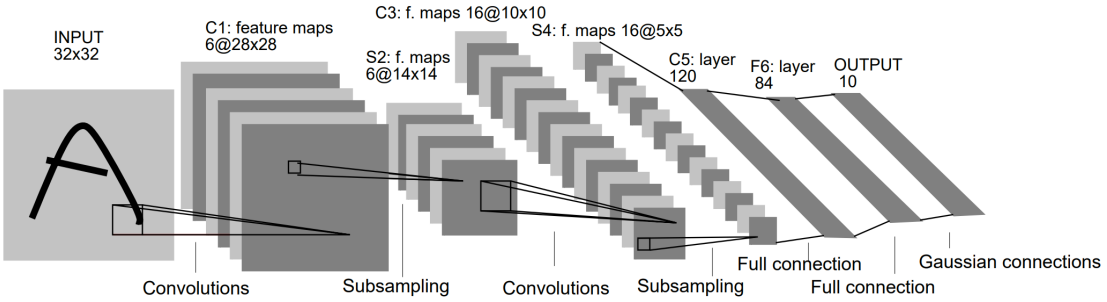


FIGURE II.6 : Architecture de LeNet-5.

II.5.2 L'architecture AlexNet

AlexNet est la première architecture de CNN à grande échelle. Elle a aboutit à un redémarrage des réseaux de neurones profonds, elle a gagné la compétition ILSVRC en 2012 [10]. Elle prend des images de $224 \times 224 \times 3$ (en RGB) et contient huit couche dont les cinq premières sont des couches de convolution et les trois dernières sont des couches Fully Connected. Cette architecture se présente ainsi :

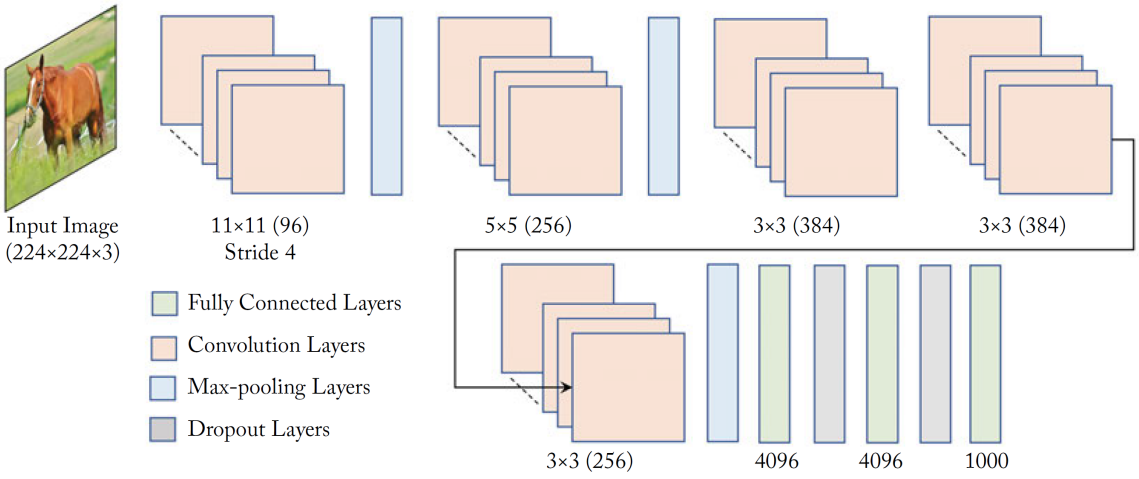


FIGURE II.7 : Architecture d'AlexNet [10].

II.5.3 L'architecture VGGNet

VGGnet est l'une des architectures les plus populaires, elle est proposée en 2014 par K.Simonyan et A.Zisserman [12] avec une utilisation stricte de filtres de convolution de taille 3×3 avec une profondeur de 16 et 19 couches communément appelées : VGGnet-16 et VGGnet-19 respectivement. Ci-après, la Figure II.8 illustre l'architecture VGGnet-16 avec treize couches de convolution et trois couches Fully Connected.

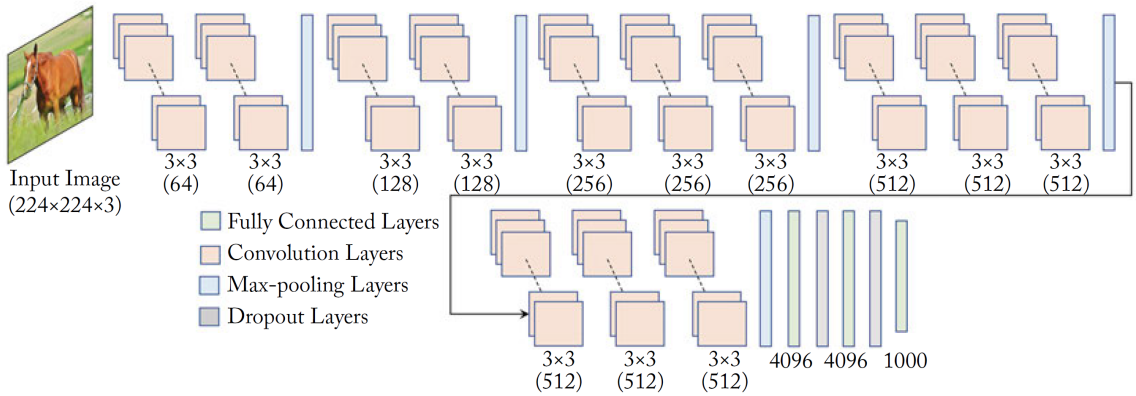


FIGURE II.8 : Architecture VGGnet-16 [10].

II.6 Conclusion

Dans ce chapitre, nous avons décrit d'une manière générale les réseaux de neurones convolutifs (CNNs), les différentes opérations utilisées et finalement, trois des architectures les plus populaires ont été aussi présentées. Dans le processus d'apprentissage d'un réseau de neurones convolutif, il est nécessaire de choisir une architecture et appliquer les opérations nécessaires, ceci sera détaillé dans le prochain chapitre.

Chapitre III

Apprentissage d'un modèle de réseau de neurones convolutif pour une vision artificielle en conduite autonome

III.1 Introduction

L'apprentissage d'un réseau de neurones de convolution pour une vision artificielle consiste a générer un modèle de classification des images. Dans notre cas, pour une application sur la conduite autonome de véhicules, le réseau de neurones sera capable de classifier des panneaux de signalisation routière. Ainsi, les décisions prises par l'ordinateur du véhicule seront en fonction du panneau de signalisation reconnu par le réseau.

Afin d'atteindre cet objectif, il est nécessaire de passer par un certain nombre d'étapes. La première est de choisir un outil de développement et le langage de

programmation. Par la suite, il faut obtenir la base de données adéquate et ensuite l'architecture du CNN. Finalement les paramètres nécessaires pour l'apprentissage et le lancement de l'apprentissage sur une machine.

Dans ce chapitre, ces étapes sont détaillées et les résultats de l'apprentissage sont rapportés.

III.2 Approche choisie pour l'apprentissage

III.2.1 Environnement de développement

En DL, il existe d'innombrables outils développés spécialement pour l'apprentissage automatique : Deep Learning Toolbox - MATLAB développé par Mathworks, PyTorch par Facebook ou Tensorflow par Google font partie de ces outils. Cet environnement peut être choisi en fonction de sa puissance, sa documentation, sa licence, et son langage de programmation.

Le Deep Learning Toolbox - MATLAB et PyTorch seront les deux outils choisis pour cette application. Par conséquent, les langages de programmation seront Matlab et Python. Le Toolbox de MATLAB sera utilisé pour l'apprentissage, et PyTorch pour la mise en œuvre de la vision artificielle sur le Raspberry Pi.

III.2.2 La base de données

Pour la reconnaissance de panneaux de signalisation routière, une base de données contenant des images de panneaux de signalisation est utilisée. La "Road Sign Detection Dataset" est une base de donnees contenant 4 classes : Panneau de Passage pour piétons, Panneau de Stop, Panneau de Limitation de Vitesse et les

Feux Tricolores. C'est une base de 250Mo contenant 877 images avec 1243 panneaux. La base est organisée en deux dossiers : "images" contenant des fichier ".png" et "annotations" fichier ".xml" décrivant la position et les dimensions du panneau sur l'image et sa catégorie (classe). La base de données sera partagée en deux parties, 75% des images pour l'apprentissage (soit 933 images) et 25% des images pour la validation (soit 310 images).

FIGURE III.1 : Sélection aléatoire d'images de la base de données Road Sign Detection.

La base de données est préparée en commençant par une opération d'extraction des panneaux dans les images est faite en se basant sur les données des fichiers ".xml" (position et dimensions du panneaux), puis les images extraites sont organisées

par dossiers portants le nom de leur catégorie (classe) respective. Les images sont transformées en niveau de gris.

La base de données sera augmentée avec la fonction : `imageDataAugmenter()`. L'augmentation de la base de données est effectuée à partir du code suivant :

```
%  loading the dataset (datasetPath: path of the dataset folder)
dataset = imageDatastore(datasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
%  split the dataset (75% for training, 25% for validation)
[dtTrain, dtVal] = splitEachLabel(dataset, 0.75,'randomized');
%  setting a dataset augmenter
agmntr = imageDataAugmenter( ...
    'RandRotation',[-15 15], ...
    'RandXTranslation', [-3 3], ...
    'RandYTranslation', [-3 3]);
%  setting input images size (28 by 28 one channel)
imsz = [28 28 1];
%  generating the augmented image datastore for training
dtTrainAug = augmentedImageDatastore( ...
    imsz, dtTrain, ...
    "DataAugmentation",agmntr);
```

Listing III.1 : Préparation de la base de données (Chargement - Partage - Augmentation)

La fonction d'augmentation crée de nouvelles images à partir de la base de données lors de l'apprentissage en apportant quelques modifications aux originales, dans ce cas, des rotations aléatoires avec un angle compris entre $-15°$ et $+15°$ et des translations aléatoires horizontales et verticales de -3 à 3 pixels.

III.2.3 L'architecture du réseau convolutif

L'architecture choisie est une architecture inspirée de l'architecture LeNet-5. Elle contient une couche d'entrée qui reçoit des données de 28×28 avec une seule chaine de couleur (niveau de gris), trois couche de convolution, deux couche Fully Connected et enfin une couche de sortie Softmax avec 4 sorties (4 classes).

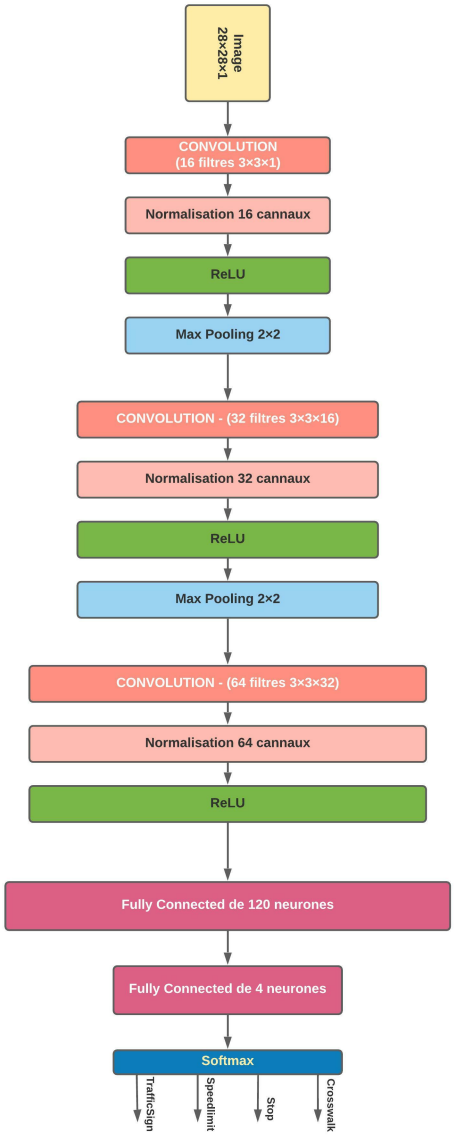


FIGURE III.2 : L'architecture du CNN choisie pour la reconnaissance automatique des panneaux.

Cette architecture est implémentée sous Matlab en définissant les différentes couches comme suit : :

```
function lyrs = setLayers()
lyrs = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,16,'padding',1)
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2, 'Stride',2)

    convolution2dLayer(3,32,'padding',1)
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2, 'Stride',2)

    convolution2dLayer(3,64,'padding',1)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(200)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(4)
    softmaxLayer

    classificationLayer];
end
```

Listing III.2 : Implémentation de l'architecture du CNN sous MATLAB

III.3 Apprentissage du réseau de neurones convolutif

III.3.1 Choix des hyperparamètres

Le choix des hyperparamètres se fait par l'appel de la fonction `trainingOptions` du Toolbox de DL. Cette fonction retourne les hyperparamètres du réseaux sous forme d'une variable.

La descente de gradient stochastique avec moment est choisie comme algorithme de rétropropagation. Le moment a été mis à **0,9** et le taux d'apprentissage à **0,01**.

Les images de validation sont passées à cette fonction en paramètres.

La fonction `trainingOptions` est appelée à travers le code suivant :

```
function options = setOptions(imVal)
options = trainingOptions('sgdm',...
    'Momentum', 0.9,...
    'InitialLearnRate',0.01,...
    'MaxEpochs',10,...
    'Shuffle','every-epoch',...
    'ValidationData',imVal,...
    'ValidationFrequency',5,...
    'Verbose',true,...
    'Plots','training-progress');
end
```

Listing III.3 : Fonction d'hyperparamétrage du réseau

La méthode de la SGDM étant choisie, le paramètre `'sgdm'` est passé à la fonction et le moment est mis à $0,9$ avec la paire de paramètres : `'Momentum',0.9`. Le taux d'apprentissage est mis à $0,01$ avec `'InitialLearnRate',0.01`. Les images de la base de données doivent être passées un certain nombre de fois, ce nombre est appelé "epoch", il est choisi à partir de la paire : `'MaxEpochs',10`. Et

`'Shuffle','every-epoch'` demande à la fonction de mélanger la base de données, c.-à-d. l'ordre de passage des images dans le réseau durant l'apprentissage est différent pour chaque epoch. La fréquence de validation et les données de validation sont définies respectivement par `'ValidationFrequency',5` et `'ValidationData',imVal` (La fréquence de validation est 10, c.-à-d. qu'à chaque 10 itérations, on passe les données de validation au réseau). Les derniers hyperparamètres sont pour visualiser les résultats et l'évolution de l'apprentissage par du texte et graphiquement (Figure III.3).

III.3.2 Lancement de l'apprentissage

Pour lancer l'apprentissage du réseau préparé, on appelle la fonction `trainNetwork`, elle prend en paramètres les données d'apprentissage, l'architecture du réseau établie par `setLayers` et les paramètres définis par `setOptions`. Cette fonction renvoie une donnée de type "SeriesNetwork" contenant l'ensemble des poids du réseau de neurones. Elle permet aussi de suivre l'évolution de l'apprentissage (La précision du modèle et l'erreur commise) par le biais d'un graphique et d'un tableau.

L'apprentissage est lancé sous *MATLAB 2021b Academic* roulant sous *Windows 11 Home* sur une machine avec un processeur *Intel Core i7-10510U* et une mémoire vive (RAM) de *16GB DDR4* sans carte graphique (GPU intégré).

Le lancement de l'apprentissage se fait en appelant la fonction `trainNetwork` par la ligne de code : `net = trainNetwork(imTrainAug, layers, options)`, le paramètre `imTrainAug` est l'ensemble des images d'apprentissage, `layers` pour les couches qui définissent l'architecture du réseau et `options` sont les hyperparamètres. Lorsque l'apprentissage s'achève, on obtient la variable `net` contenant le modèle. Le graphique suivant est tracé durant l'apprentissage :

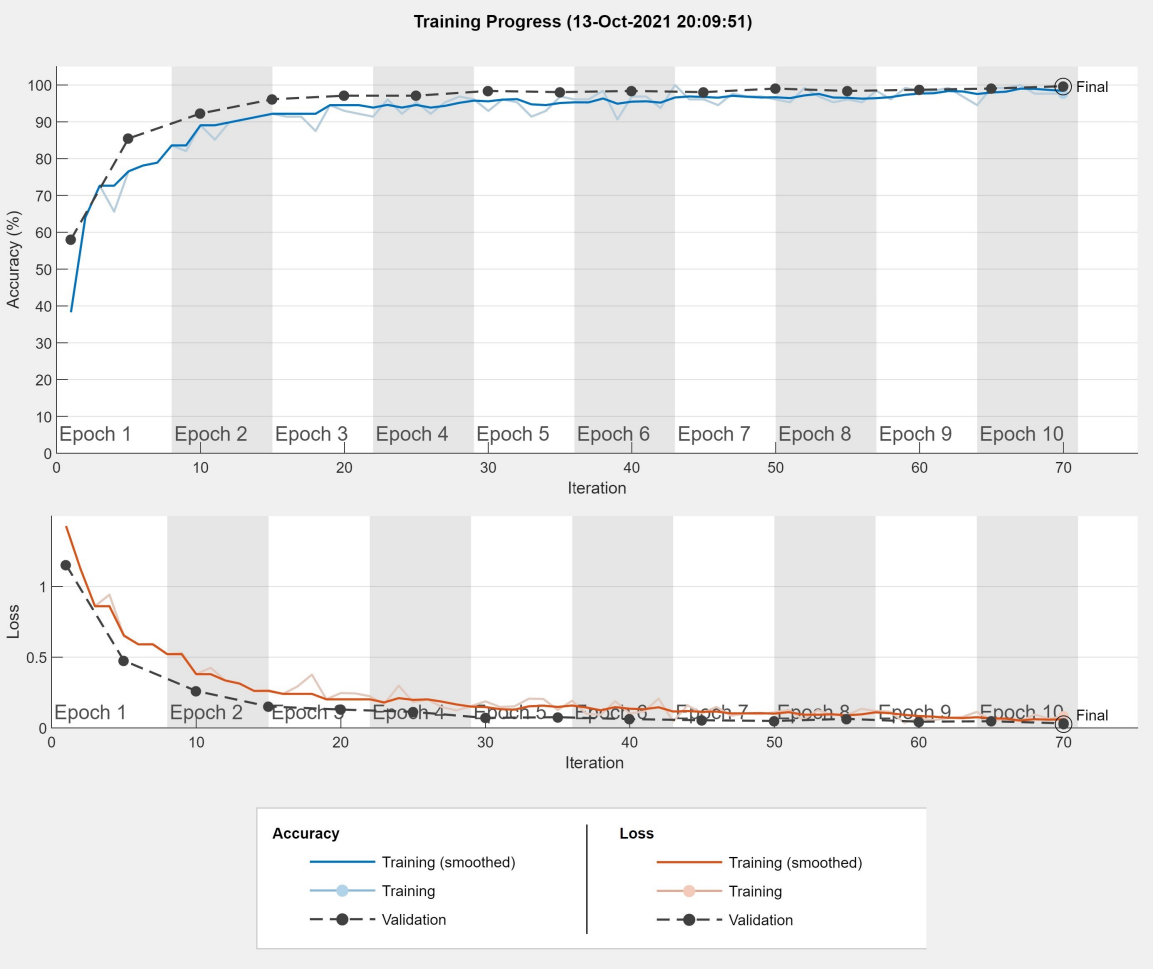
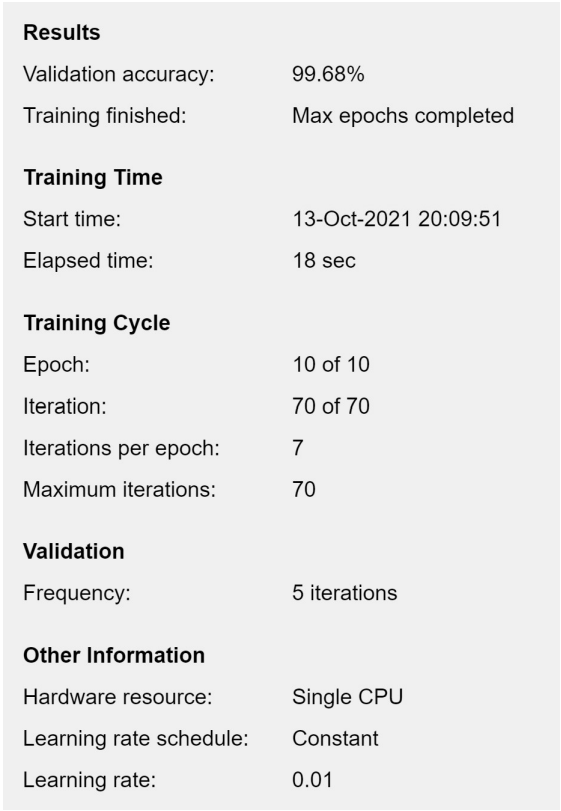


FIGURE III.3 : Courbe d'évolution de la précision et de l'erreur du modèle durant l'apprentissage.

La Figure III.3 ci-dessus, montre l'évolution de l'apprentissage du modèle. D'une part, la première courbe (en bleu) est tracée selon la progression de la precision du modèle qui augmente au fur et à mesure de l'apprentissage. D'autre part, la courbe (en rouge), montre que l'erreur commise par le modèle diminue à travers les epochs d'apprentissage.



Results	
Validation accuracy:	99.68%
Training finished:	Max epochs completed
Training Time	
Start time:	13-Oct-2021 20:09:51
Elapsed time:	18 sec
Training Cycle	
Epoch:	10 of 10
Iteration:	70 of 70
Iterations per epoch:	7
Maximum iterations:	70
Validation	
Frequency:	5 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.01

FIGURE III.4 : Détails de l'apprentissage du modèle.

L'apprentissage a pris un temps total de 18 secondes et le modèle a atteint une précision de $99,68\%$ sur les images de validation. La précision du modèle augmente progressivement avec les epochs et l'erreur diminue.

Le Tableau III.1 suivant rapporte en détail l'évolution du modèle entrainé.

TABLEAU III.1 : Tableau récapitulatif des résultats d'apprentissage

Epoch	**Itération**	**Temps écoulé (s)**	**Précision du Mini-batch (%)**	**Précision de Validation (%)**	**Erreur de Mini-batch**	**Erreur de Validation**
1	1	3s	38,28%	58,06%	1,4270	1,1527
1	5	4s	76,56%	85,48%	0,6499	0,4752
2	10	5s	89,06%	92,26%	0,3800	0,2613
3	15	6s	92,19%	96,13%	0,2661	0,1518
3	20	7s	92,97%	97,10%	0,2468	0,1298
4	25	8s	95,31%	97,10%	0,1823	0,1138
5	30	9s	92,97%	98,39%	0,1875	0,0708
5	35	10s	96,88%	98,06%	0,1273	0,0760
6	40	11s	96,88%	98,39%	0,1013	0,0624
7	45	12s	96,09%	98,06%	0,1041	0,0544
8	50	13s	96,09%	99,03%	0,1095	0,0492
8	55	14s	96,09%	98,39%	0,0892	0,0641
9	60	15s	98,44%	98,71%	0,0686	0,0441
10	65	17s	99,22%	99,03%	0,0533	0,0472
10	70	**18s**	97,66%	**99,68%**	0,0829	**0,0326**

III.3.3 Vérification du réseau

Pour vérifier le modèle généré stocké dans la variable `net`, on utilise la fonction `perdict` ou `classify`. Elle prend en paramètres : le modèle et les images à prédire. Ces images passées au réseau sont des images qui n'ont jamais été utilisées durant

l'apprentissage.

Avant de passer les images au réseau, un traitement au préalable s'impose. Elle doivent être redimensionnées en 28×28, transformées en niveau de gris en double precision.

Une fois les images traitées, pour les prédire, des inférences sur le CNN peuvent être effectuées avec la fonction : `[yp, scores] = classify(net, imTest)`, où `yp` est un vecteur de taille égale au nombre d'observations (nombre d'images prédites) et `scores` est une matrice des scores ou des probabilités de prédiction, pour chaque observation, elle contient la plausibilité exprimée en pourcentage de chaque classe. La Figure III.5 rapporte les résultats :

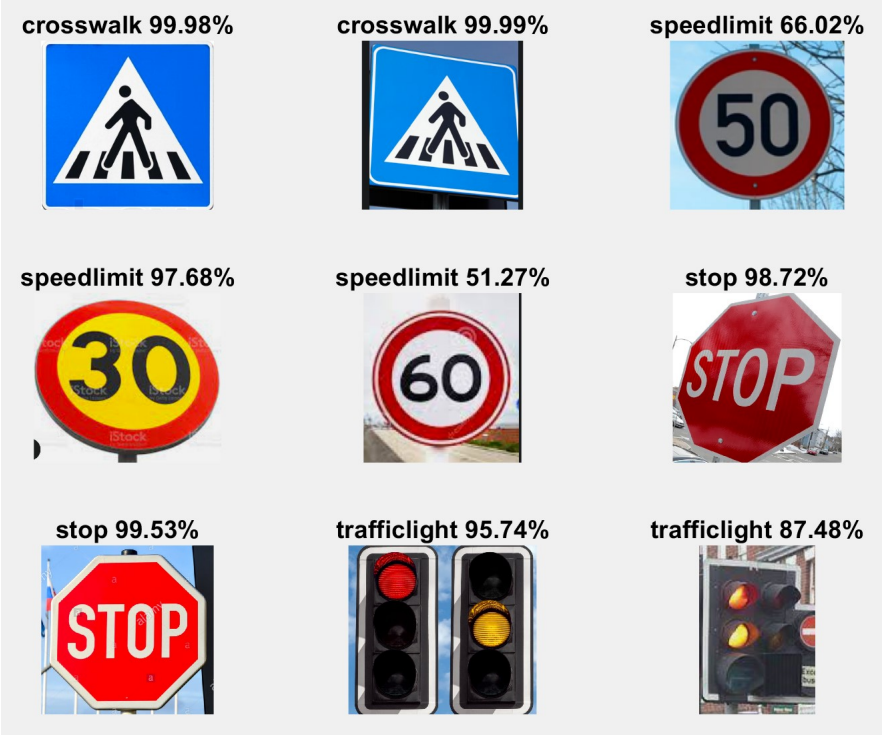


FIGURE III.5 : Prédiction de panneaux de signalisation par le modèle.

On peut remarquer sur la Figure III.5 que notre modèle prédit le panneau avec un pourcentage élevé si le panneau est pris de face sur l'image, le cas du panneau

Stop : le premier est pris de côté et il est prédit à $98,72\%$ par contre le deuxième qui est pris de face est prédit à $99,53\%$. L'exposition à la lumière peut aussi influencer sur la prédiction, c'est le cas du panneau de Limitation de Vitesse (speedlimit 50 et 60) sont prédits à $66,02\%$ et $51,27\%$ contrairement au deuxième (le 30) qui est prédit à $97,68\%$.

Ce modèle peut être amélioré par différents moyens. Un élargissement de la base de données, c.-à-d. l'utilisation d'une base de données proposant un large nombre de classes. Ainsi, le modèle pourrais prédire d'autres panneaux de signalisation existants. La base de données German Traffic Sign Road Benchmark (GTSRB) peut être utilisée, elle propose jusqu'à 40 classes avec 50000 images [13]. Ce modèle est limité à la reconnaissance, donc il faut lui passer uniquement des images contenant exclusivement des panneaux de signalisation. Afin d'extraire le panneaux de signalisation dans le paysage, la segmentation des panneaux est nécessaire, des algorithmes de ML comme Viola-Jones [14] peuvent être utilisés. En DL, la segmentation peut être faite grâce aux réseaux de convolution comme le Vanilla Fully Convolutional Network (FCN) [15] ou Pyramid Scene Parsing Network (PSPnet) [16] en utilisant des bases de données dédiées comme MS-COCO ou Cityscapes.

III.4 Analyse du réseau de neurones convolutif entraînée

Pour analyser un réseau de neurones sur Matlab, on utilise la fonction `analyzeNetwork(net)` où `net` est le modèle du réseau à analyser. La fonction dresse un graphique et un tableau de description des caractéristiques du réseau, on y trouve les noms des couches, leur enchainement et leurs paramètres s'ils existent.

Les Tableau III.2 récapitule ce que donne la fonction d'analyse du réseau.

TABLEAU III.2 : Tableau descriptif de l'architecture de notre CNN

Couche	**Dimension**	**Paramètres**	**Nbr. de Param./couche**
Convolution	$28 \times 28 \times 16$	Poids $3 \times 3 \times 1 \times 16$ Biais $1 \times 1 \times 16$	160
Batch Normalization	$28 \times 28 \times 16$	Offset $1 \times 1 \times 16$ Echelle $1 \times 1 \times 16$	32
ReLU	$28 \times 28 \times 16$	-	0
Average Pooling	$14 \times 14 \times 16$	-	0
Convolution	$14 \times 14 \times 32$	Poids $3 \times 3 \times 16 \times 32$ Biais $1 \times 1 \times 32$	4640
Batch Normalization	$14 \times 14 \times 32$	Offset $1 \times 1 \times 32$ Echelle $1 \times 1 \times 32$	64
ReLU	$14 \times 14 \times 32$	-	0
Average Pooling	$7 \times 7 \times 32$	-	0
Convolution	$7 \times 7 \times 64$	Poids $3 \times 3 \times 32 \times 64$ Biais $1 \times 1 \times 64$	18496
Batch Normalization	$7 \times 7 \times 64$	Offset $1 \times 1 \times 64$ Echelle $1 \times 1 \times 64$	128
ReLU	$7 \times 7 \times 64$	-	0
Fully Connected	$1 \times 1 \times 200$	Poids $3 \times 200 \times 3136$ Biais 200×1	627400
Batch Normalization	$1 \times 1 \times 200$	Offset $1 \times 1 \times 200$ Echelle $1 \times 1 \times 200$	400
ReLU	$1 \times 1 \times 200$	-	0
Fully Connected	$1 \times 1 \times 4$	Poids $4 \times 200 \times 3136$ Biais 4×1	804
Softmax	$1 \times 1 \times 4$	-	0
Total de couches	**16**	**Total de paramètres**	**652124**

Il est possible de visualiser les poids de différentes couches d'un réseau de neurones grâce à la fonctions : `deepDreamImage(net, name, channels)`, `net` est le réseau de neurones concerné, `name` est le nom de la couche concernées et `channels` est un vecteur indiquant les chaines (ou filtres) à visualiser.

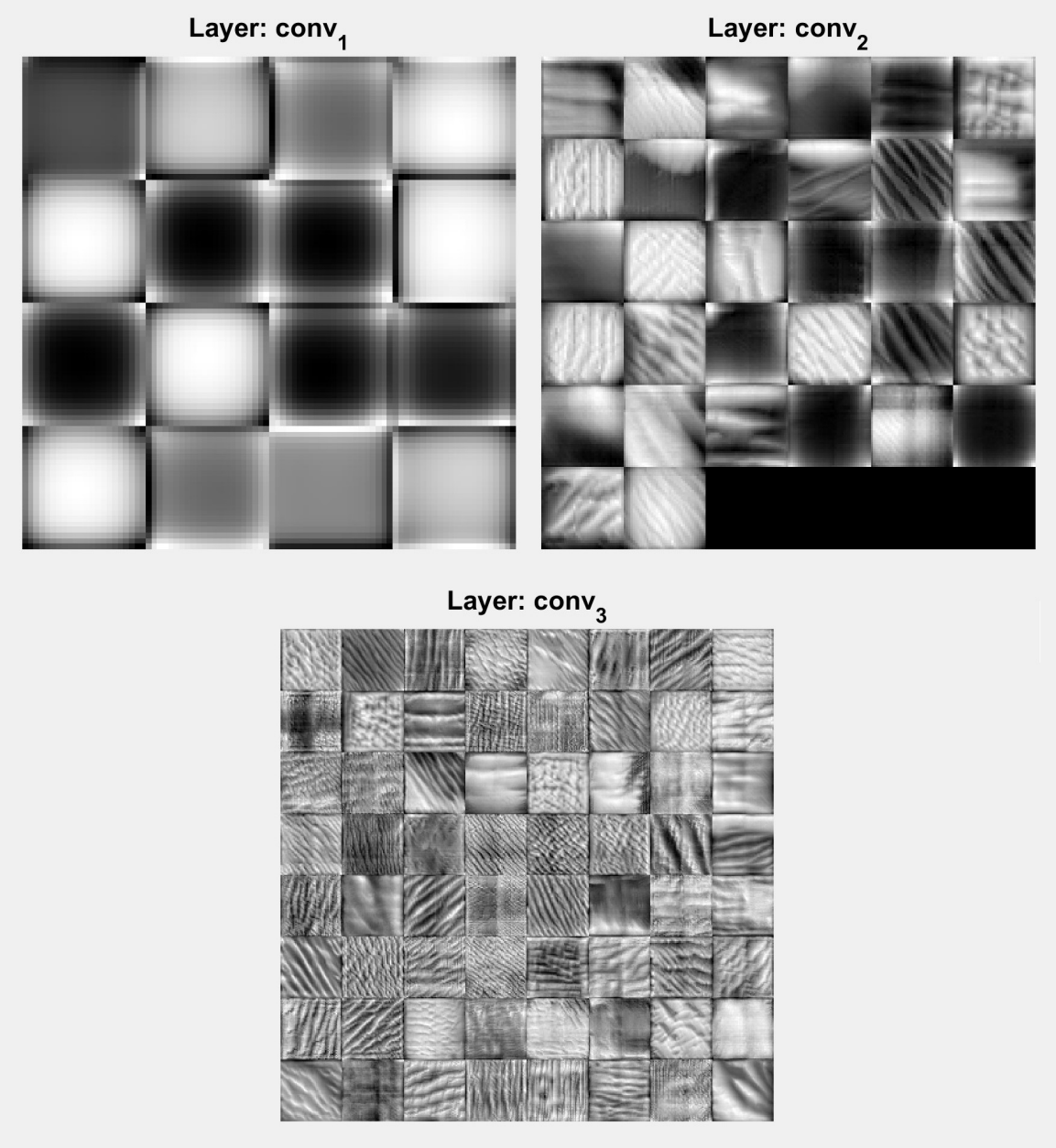


FIGURE III.6 : L'apparence des filtres des trois couches de convolution du modèle entrainé.

La Figure III.6 précédente montre les filtres des trois couches de convolution. La première contient 16 filtres, la deuxième 32 et la troisième 64. Les Figures III.7 et III.8 suivantes montrent les activations sur quelques couches du réseau.

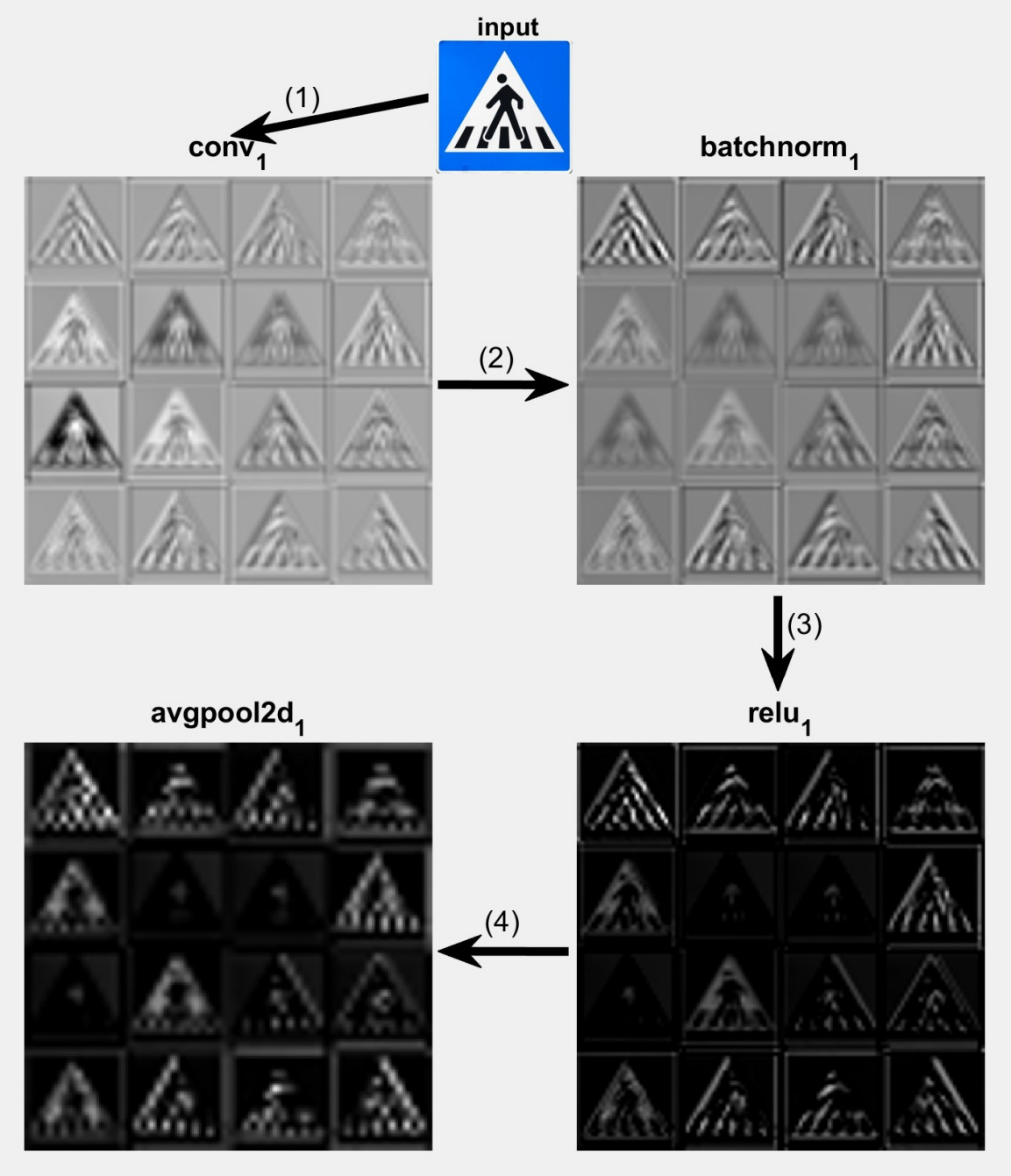


FIGURE III.7 : Les sorties des quatre premières couches du réseau pour un panneau de passage pour piétons (Convolution - Normalisation - ReLU - AvgPooling).

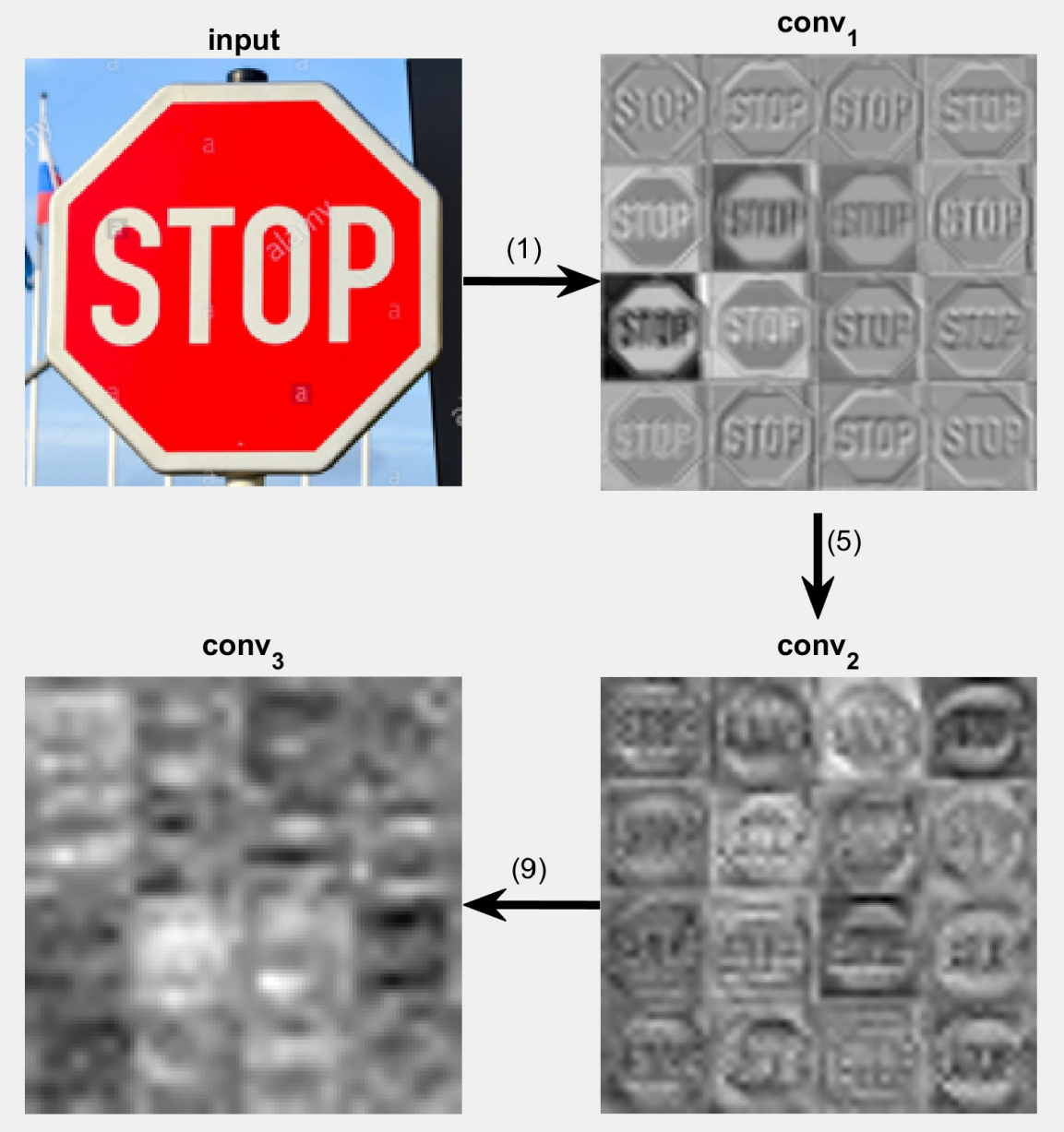


FIGURE III.8 : Les seize premières activations de chacune des trois couches de convolution du modèle pour un panneau de stop.

III.5 Conclusion

Dans ce chapitre, nous avons pu exécuter un apprentissage sur un réseau de neurones convolutifs d'architecture inspirée de LeNet-5 à l'aide d'une base de données de panneaux de signalisation routière à quatre classes. Le modèle généré par cette opération a atteint les 99% de précision mais il peut tout de même être amélioré. Dans le chapitre suivant, nous aurons besoin de l'ensemble des paramètres de notre modèle, soit le fichier `.mat`, pour implémenter le CNN sur un Raspberry Pi.

Chapitre IV

Implémentation du modèle entrainé sur un ordinateur Raspberry Pi

IV.1 Introduction

Ce dernier chapitre, sera consacré à la réalisation du système de vision artificielle à base de l'ordinateur Raspberry Pi 4. Le modèle du réseau de neurones convolutif généré dans le chapitre III sera implémenté au moyen de l'écosystème ONNX sous Python. Une méthode d'implémentation au moyen du framework PyTorch sera également présentée. Le système de vision artificielle sera embarqué sur un robot mobile 2WD (robot à 2 roues motrices).

IV.2 Description du système de vision artificielle

Le système de vision artificielle développé est construit autour d'un SBC Raspberry Pi 4. Les images sont acquises à partir d'une caméra et sont transférées au

Raspberry Pi via l'interface Camera Serial Interface (CSI), les décisions prises par le système agissent, par l'intermédiaire d'un pont en H, sur les actionneurs du robot qui sont les deux moteurs à courant continu attachés aux deux roues motrices. La Figure IV.1 suivante décrit le système.

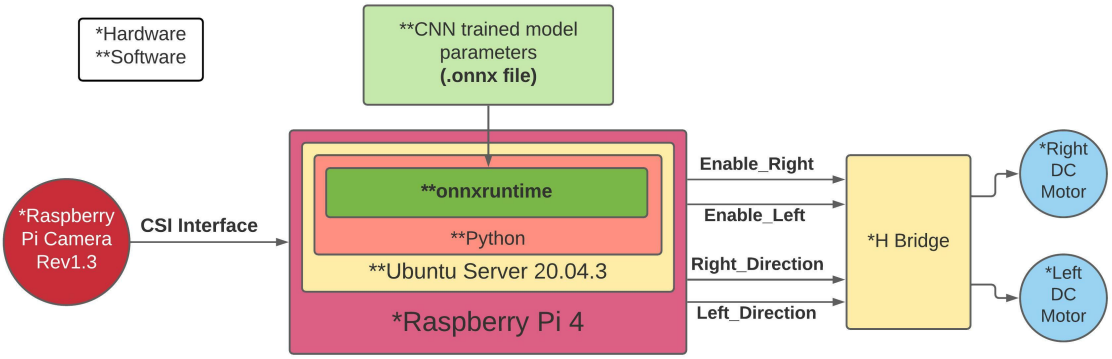


FIGURE IV.1 : Le schéma détaillé du système de vision artificielle.

IV.3 La composition du système

Après avoir décrit le système précédemment, ses différentes composantes sont détaillées ci-dessous.

IV.3.1 L'unité centrale

Hardware

Le système est développé autour de l'ordinateur à carte unique SBC *Raspberry Pi 4 Model B*. Cette ordinateur de la taille d'une carte bancaire est principalement basé sur :

- Un processeur SoC BCM2711 Quad-Core Cortex A72 (ARM v8) cadencé à une

fréquence de 1,5GHz.

- Quatre (04) GB de RAM LPDDR4-3200.
- Connectivité sans fil : Bluetooth 5.0, BLE et WiFi IEEE 802.11b/g/n/ac 2,4GHz et 5,0GHz.
- Interface Ethernet.
- Deux (02) ports USB 2.0 et deux (02) ports USB 3.0
- Deux sorties micro-HDMI avec des résolutions allant jusqu'à la 4K et décodage hardware vidéo allant jusqu'à 4Kp60..
- Quarante (40) broches General Purpose Input/Output (GPIO).
- Une fente de carte de mémoire micro-SD pour le stockage de l'Operating System (OS) et le stockage permanent de données.

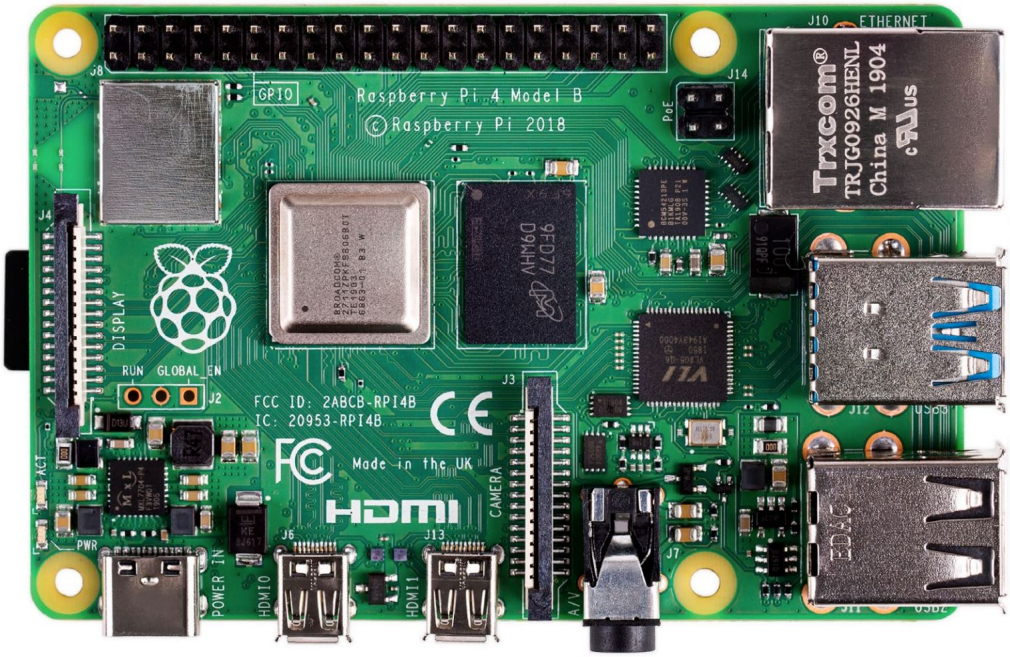


FIGURE IV.2 : La carte Raspberry Pi 4 Model B.

Software

Les SBCs de la série Raspberry Pi sont conçus pour exécuter un système d'exploitation (OS). Le système d'exploitation fourni par la Raspberry Pi Foundation est appelé : *Raspberry Pi OS* (autrefois appelé : *Raspbian*) qui est une distribution Linux basée sur la distribution Debian (32-bits). D'autres OS sont également compatibles avec les cartes Raspberry Pi. Dans notre cas, la distribution *Ubuntu Server 20.04.3 LTS 64-bits* (Nom de code : *Focal Fossa*) a été installée sur la carte. Cette distribution est uniquement opérationnelle en ligne de commande. Elle a été choisie pour des raisons de performance et de compatibilité logicielle. Son noyau Linux est de version *5.4.0-1047-raspi*. Afin que la vision artificielle puisse être opérationnelle, un certain nombre de programmes doivent être préalablement installés sur l'OS. Le Tableau IV.1 suivant présente les programmes nécessaires en détail :

TABLEAU IV.1 : Les programmes nécessaires au fonctionnement de la vision artificielle

Programme	**Version**	**Description**
Ubuntu Server	20.04.3 LTS 64-bits (ARM64)	Système d'exploitation
Python 3	3.8.10	Interpréteur Python du programme principal de la vision artificielle
OpenCV 2	4.5.5	Acquisition et pré-traitement de l'image
NumPy	1.21.5	Calculs mathématiques et manipulation des données
onnxruntime	1.10.0	Inférence sur réseaux de neurones et optimisation sur les modèles `.onnx`
PyTorch	1.8.1	Inférence sur les modèles réseaux de neurones `.pth`

IV.3.2 Le capteur

Afin d'acquérir les images, nous utilisons un module caméra "Raspberry Pi Camera Rev1.3" qui est une caméra de 5MP (5 Méga-Pixels), soit une capture d'image fixe d'une résolution maximale de 2592×1944 pixels. Elle est capable aussi de faire une capture vidéo à 30fps avec une résolution de $1080p$ ou à 60fps avec une résolution de $720p$. Le module est compatible MIPI CSI-2. L'image est alors communiquée à l'ordinateur via cette interface. Dans notre cas, la caméra est utilisée avec une résolution de 64×64 pixels à un taux de capture de 60fps.

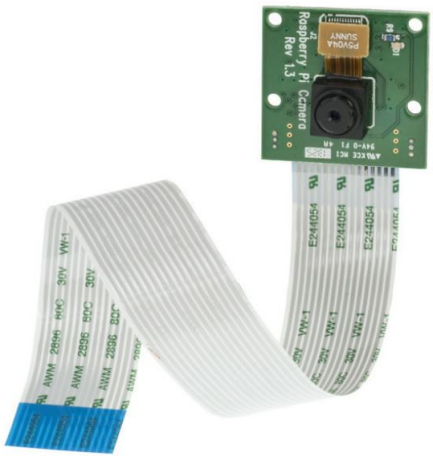


FIGURE IV.3 : Le module Raspberry Pi Camera Rev1.3.

IV.3.3 L'actionneur

La partie puissance

Les actionneurs du système sont deux (02) moteurs à courant continu de 12V raccordés aux deux roues motrices du robot, un pour chaque côté. Les moteurs tournent en même temps à la même vitesse pour faire avancer le robot. Ils tournent tous les deux dans le sens inverse pour faire reculer le robot. Pour faire tourner le

robot dans un direction donnée, il faut ralentir l'un des deux moteurs par rapport à l'autre avec la PWM.

La partie commande

La mise en marche, l'arrêt et le sens de rotation des moteurs est commandé par le Raspberry Pi avec des signaux de sorties du périphérique GPIO. Ces signaux sont injectés dans les deux ponts en H. Le circuit intégré L298N est un circuit driver de moteurs à courant continu et de moteurs pas-à-pas. Il comporte 2 ponts en H, c.-à-d. qu'il peut piloter 2 moteurs DC.

IV.3.4 Présentation du robot développé

Le châssis du robot est composé d'une plaque en verre d'epoxy, où tous les modules et composants sont y attachés. Deux moteurs DC sont montés sous la plaque. les deux roues sont directement raccordées aux arbres tournants des moteurs. Une troisième roue non-motrice est libre pour pivoter lors du changement de direction du robot, elle est montée sous la plaque. L'ordinateur RPi est montée sur la plaque d'epoxy et est connecté à la caméra et au driver des moteurs DC.

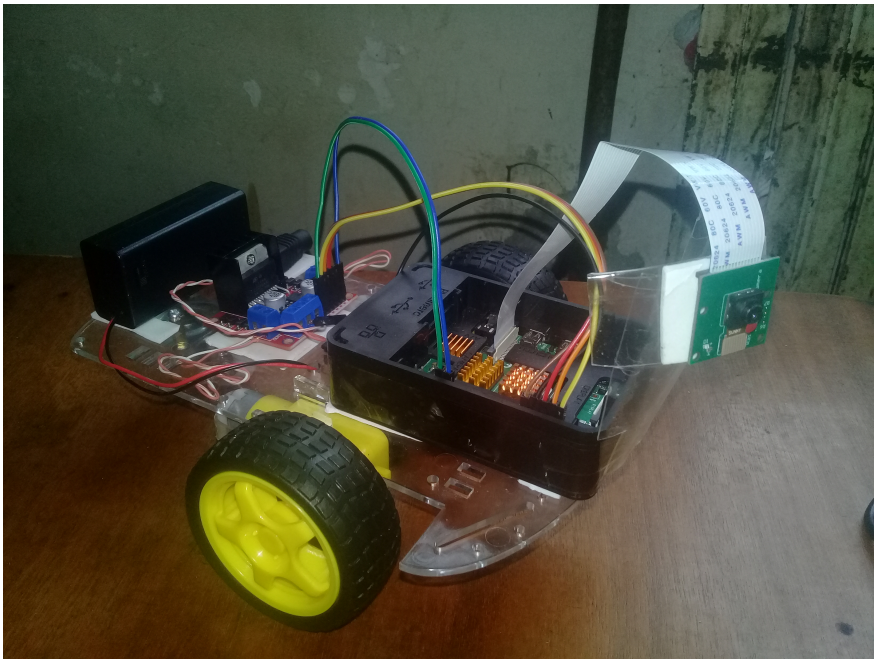


FIGURE IV.4 : Le robot mobile avec son système de vision artificielle.

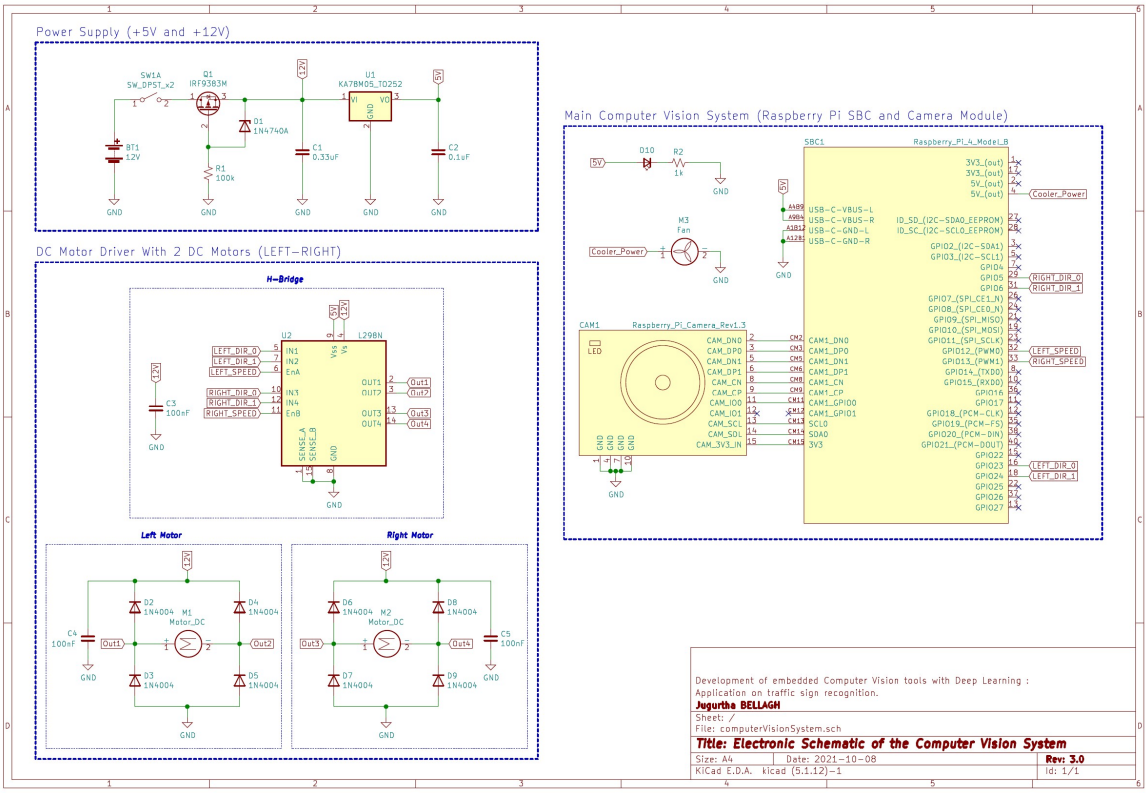


FIGURE IV.5 : Schéma électronique général du hardware du système.

IV.4 Implémentation du système de vision artificielle sur le Raspberry Pi

IV.4.1 Acquisition d'images sur le Raspberry Pi

L'acquisition des images sur le RPi se fait à partir du module "Raspberry Pi Camera" via l'interface CSI. L'activation de l'interface CSI se fait en ajoutant les lignes `start_x=1` et `gpu_mem=512` en fin du fichier `config.txt` situé dans le répertoire `/boot/firmware/`. L'activation de l'interface CSI est nécessaire pour initier la communication entre le Raspberry Pi et le module caméra. La bibliothèque Python qui permet l'acquisition des images est *OpenCV 2*. Le code Python suivant permet d'y aboutir, l'image acquise dans est enregistrée dans un fichier `.jpg` :

```
import cv2                      #OpenCV 2 library import

cap = cv2.VideoCapture(0)       #Init image acquisition
if not cap.isOpened():          #Verifying camera opening
    print('Error during camera opening.')
    exit()
cap.set(3,64)                   #Set low resolution capture
cap.set(4,64)                   #64 by 64 pixels
cap.set(cv2.CAP_PROP_FPS, 60) #set frame rate at: 60fps

retval, im = cap.read()         #Read the image
if not retval:                  #Verifying frame lecture
    print('Error reading the frame.')
    exit()
cv2.imwrite('testcv2.jpg', im)#Save the image
cap.release()                   #Release the capture
cv2.destroyAllWindows()         #Close windows if opened
```

Listing IV.1 : Programme d'acquisition des images.

IV.4.2 Implémentation du modèle sous Python

Python est un langage de programmation interprété, il est utilisé pour le développement dans plusieurs domaines notamment en IA. L'implémentation software d'un réseau de neurone artificiel dépend des outils choisis au départ pour l'apprentissage et du hardware sur lequel le réseau doit être opérationnel. Deux méthodes d'implémentation sous Python seront présentées ci-dessous. La première étant la méthode que nous avons utilisée durant le développement de notre système. La deuxième est la méthode alternative à utiliser dans le cas d'un apprentissage d'un modèle sur une plateforme de ML Python, comme exemple PyTorch. La methode PyTorch est proposée dans ce travail uniquement à titre de comparaison avec la méthode ONNX et comme solution alternative si nécessaire.

Implémentation sous Python avec le package onnxruntime

L'ONNX est un écosystème open source développé initialement par Facebook, il permet l'interopérabilité des modèles d'intelligence artificielle : de DL et de ML entre différents frameworks (Matlab, PyTorch, Keras, TensorFlow, ...)(Figure IV.6). L'ONNX fournit des outils de conversion de formats de fichier du modèle en fichier `.onnx`.

L'Open Neural Networks eXchange Runtime (ORT) est un projet open source développé par Microsoft. Comme les autres Runtimes, il a pour but d'effectuer des inférences sur des modèles d'intelligence artificielle en format `.onnx`, assurer une meilleure optimisation du modèle et accélérer l'inférence. Ces fichiers `.onnx` peuvent être issus de conversions à partir de différentes plateformes de DL.

Afin d'implémenter sous Python notre modèle généré précédemment à l'aide de l'outil Deep Learning Toolbox de Matlab, il est nécessaire de convertir le format de fichier du modèle sauvegardé sous l'extension `.mat` en format `.onnx` pour pouvoir le

charger dans Python. Pour cela, une fonction dans le Toolbox est dédiée à cette conversion. Cette fonction est contenue dans le module Open Neural Networks eXchange (ONNX) de Matlab, sa syntaxe est : `exportONNXNetwork(net, file)` où `net` est le réseau de neurones enregistré sous Matlab (le fichier `.mat`), et `file` est le nom du fichier `.onnx` de destination. Le fichier `.onnx` porte l'architecture du réseau et les paramètres associés à chaque couches.

Cette figure illustre le processus :

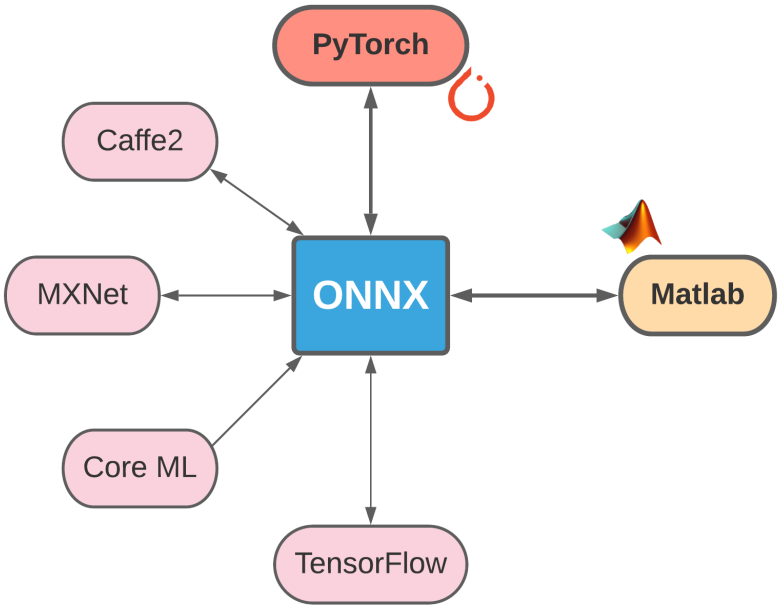


FIGURE IV.6 : Processus d'échange de fichiers ONNX entre différentes plateformes de ML.

Une fois le fichier `.onnx` obtenu, on l'importe sur Python après avoir installé ONNX (le package `onnx`) et le Runtime ORT (le package `onnxruntime`). Le package `onnx` permet de visualiser l'architecture du modèle ainsi que le nom et les détails de chaque couche du réseau. L'inférence s'effectue avec le package `onnxruntime`.

Le code Python principal du système de vision artificielle (Acquisition de l'image/Traitement de l'image/Inférence/Calcul du résultat) est le suivant :

```
import onnxruntime
import numpy as np
import cv2
from time import sleep

classes = ('crosswalk', 'speedlimit', 'stop', 'trafficlight')
#loading the model pretrained
modelpath = './onnxmodel.onnx'
rts = onnxruntime.InferenceSession(modelpath) #Runtime Session
#image acquisition
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    exit()
cap.set(3,64)                  #Set low resolution capture
cap.set(4,64)                  #64 by 64 pixels
cap.set(cv2.CAP_PROP_FPS, 60) #set frame rate at: 60fps
#classification
while True:
    retval, im = cap.read()    #Read the image
    if not retval:             #Verifying frame lecture
        print('Error reading the frame.')
        break
    im = cv2.rotate(im,cv2.ROTATE_180) #Rotate if required
    im = cv2.resize(im, (28,28))       #Resize the image
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) #Convert to gray
    #Reshape and change image number type
    im = np.array(im).reshape(1,1,28,28).astype(np.float32)
    out = rts.run(None, {'imageinput': im}) #inference
    yp=np.argmax(out[0])       #Output calculation
    print(classes[yp])         #Display Output
#close
cap.release()                  #Release the capture
cv2.destroyAllWindows()         #Close windows if opened
```

Listing IV.2 : Programme de classification d'images avec ONNX Runtime.

La précision du modèle sous Python avec le fichier `.onnx` sur les mêmes images de validation est de $99,35\%$, elle a diminué avec $0,33\%$.

Les résultats des temps d'acquisition et de traitement des images, le temps d'inférence, la température du CPU, l'occupation en mémoire de stockage, et en mémoire vive, l'occupation du CPU par le programme exécuté et la consommation énergétique du système sont indiquées dans le Tableau IV.2 suivant :

TABLEAU IV.2 : Détails sur notre implémentation avec ONNX Runtime

	Mesure	**Valeur**	**Unité**	**Total**
Temps	Acquisition d'images	4,1563	ms	**6,4401 ms**
	Pré-traitement d'images	0,1457	ms	
	Inférence	**2,0677**	ms	
	Post-traitement	0,0702	ms	
Hardware	Utilisation du CPU	**114,9**	%	1,14/4 coeurs
	Température du CPU (système stable)	**26**	°C	/
	Température du CPU (2min. d'exéc.)	**32**	°C	/
	RAM	2,0	%	/
		0,06877472	Go	/
	Taille du fichier **.onnx**	**2613791**	Octets	/
	Taille du fichier programme	**697**	Octets	
Consom.	Au démarrage (système stable)	0,55	A	/
		2,9095	W	/
	Durant l'interprétation du programme	0,78	A	/
		4,1028	W	/

Les mesures renseignées dans le Tableau IV.2 ci-dessus ont été prises dans un environnement à $14°C$ avec 11000 échantillons d'images. Les temps sont calculés à partir de la moyenne des temps des 11000 images. Ces données peuvent varier en fonction de l'OS, de sa version, de la version des programmes et packages, des appareils de mesure et de la température. L'appareil de mesure des courants et des tensions pour le calcul des puissances sont mesurés avec un multimètre *Digital Multimeters DT-9205A*. Le système de refroidissement (ventilateur) de la carte consomme seul $2,06W$, c.-à-d. la SBC consomme au démarrage $0,8464W$. Pour éviter les surchauffes

de la carte, le ventilateur est enclenché dès le démarrage du Raspberry Pi, donc les valeurs inscrites sur le tableau dans le champ consommation énergétique intitulé "Consom." sont la somme des consommations du Raspberry Pi et du refroidisseur. La température du CPU est sondée lorsque le système est stable et elle est de $26°C$, elle sondée après 02 minutes d'inférence en boucle (sans pause). Le pourcentage d'utilisation du CPU est par rapport à ses 4 coeurs qui totalisent 400%, et celui de la mémoire RAM est par rapport à la quantité de mémoire utilisable qui est de : $3,44Go$ ($3,438736Go$). La mémoire de stockage disponible OS non-inclus est de $50Go$.

Implémentation sous Python avec PyTorch

Parmi les frameworks les plus utilisés sont PyTorch Tensorflow et Keras. Sous certaines conditions, l'utilisation d'un framework de DL est obligatoire. Nous présentons dans cette partie une méthode d'implémentation en Python avec le framework PyTorch.

PyTorch est un framework open source développé par Meta (anciennement appelée Facebook) en langage Python, C++ et Java pour le développement en Deep Learning (DL) et en Machine Learning (ML). Il est l'un des puissants frameworks de ML et il est principalement destiné aux chercheurs.

Afin d'utiliser PyTorch, il est nécessaire d'avoir au préalable installé Python sur la machine. En ce qui concerne PyTorch, il faut l'installer manuellement avec le gestionnaire `conda` ou `pip`. Une fois installé, pour l'utiliser, on charge le package avec la ligne de code `import torch`.

Les modèles pré-entraînés sur PyTorch sont d'extension : `.pth`, ils contiennent la valeurs des paramètres du modèle. Pour exploiter le fichier en question, il faut d'abord définir l'architecture du modèle à l'aide d'une classe Python qui doit correspondre aux paramètres dans le fichier puis charger le fichier, c.-à-d. la même architecture avec

laquelle le modèle a été entraîné.

```
import torch
import torch.nn as nn
import torch.nn.functional as func
import cv2
import numpy as np
import time

class Rsdnet(nn.Module):
    def __init__(self):
        super(Rsdnet, self).__init__()
        #nn.Conv2d(in_channels,out_channels,\
        #          kernel_size,stride,padding)
        self.conv1 = nn.Conv2d(1,16,3,1,1)
        self.conv2 = nn.Conv2d(16,32,3,1,1)
        self.conv3 = nn.Conv2d(32,64,3,1,1)

        self.avg = nn.AvgPool2d(kernel_size=2, stride=2)

        self.batchNorm1 = nn.BatchNorm2d(16)
        self.batchNorm2 = nn.BatchNorm2d(32)
        self.batchNorm3 = nn.BatchNorm2d(64)
        self.batchNorm4 = nn.BatchNorm1d(200)

        self.fcl1 = nn.Linear(in_features=3136,out_features=200)
        self.fcl2 = nn.Linear(in_features=200,out_features=4)

    def forward(self, x):
        #Python layer           #matlab layer equivalent
        x = self.conv1(x)       #convolution2dLayer(3,16,'padding',1)
        x = self.batchNorm1(x) #batchNormalizationLayer
        x = func.relu(x)        #reluLayer

        x = self.avg(x)         #averagePooling2dLayer(2, 'Stride',2)
```

```
34
35          x = self.conv2(x)       #convolution2dLayer(3,32,'padding',1)
36          x = self.batchNorm2(x) #batchNormalizationLayer
37          x = func.relu(x)        #reluLayer
38
39          x = self.avg(x)         #averagePooling2dLayer(2, 'Stride',2)
40
41          x = self.conv3(x)       #convolution2dLayer(3,64,'padding',1)
42          x = self.batchNorm3(x) #batchNormalizationLayer
43          x = func.relu(x)        #reluLayer
44
45          x = torch.flatten(x, 1)#(reshaping data)
46
47          x = self.fcl1(x)        #fullyConnectedLayer(200)
48          x = self.batchNorm4(x) #batchNormalizationLayer
49          x = func.relu(x)        #reluLayer
50
51          x = self.fcl2(x)        #fullyConnectedLayer(4)
52          y = func.log_softmax(x, dim=1)  #softmaxLayer
53
54          return y
55
56 net = Rsdnet()
57 classes = ('crosswalk', 'speedlimit', 'stop', 'trafficlight')
58 # .pth model file loading
59 net.load_state_dict(torch.load('./torchmodel.pth'))
60 net.eval()
61
62 cap = cv2.VideoCapture(0)
63 if not cap.isOpened():
64     exit()
65
66 cap.set(3,64)                  #Set low resolution capture
67 cap.set(4,64)                  #64 by 64 pixels
68 cap.set(cv2.CAP_PROP_FPS, 60) #set frame rate at: 60fps
```

```
69
70 with torch.no_grad():
71     while True:
72         #  image acquisition
73         retval, im = cap.read()
74         if not retval:
75             break;
76         #  image pre-processing
77         im = cv2.resize(im, (28,28))
78         im = cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)
79         im = cv2.rotate(im, cv2.ROTATE_180)
80         im = np.array(im).reshape(1,1,28,28).astype(np.float32)
81         im = torch.from_numpy(im)
82         # torch inference
83         out = net(im)
84         # class calculation
85         yp=np.argmax(out[0])
86         print("Class: %s" % classes[yp])
87
88 cap.release()
89 cv2.destroyAllWindows()
90 print("Closed All.")
```

Listing IV.3 : Programme d'inférence avec PyTorch.

Afin de comparer les performances de l'inférence sur PyTorch par rapport à celles de l'écosystème ONNX Runtime, le Tableau IV.3 est l'équivalent du Tableau IV.2 indiquant les résultats de tests sur une implémentation sur PyTorch. L'architecture définie sur PyTorch (Listing IV.3) sur laquelle ces tests sont réalisés est la même que celle de notre modèle ONNX et elle a le même nombre de paramètres.

TABLEAU IV.3 : Détails sur une implémentation avec le framework PyTorch

	Mesure	Valeur	Unité	Total
Temps	Acquisition d'images	0,6389	ms	**18,3634 ms**
	Pré-traitement d'images	0,2028	ms	
	Inférence	**17,1681**	ms	
	Post-traitement	0,3534	ms	
Hardware	CPU	**391,1**	%	3,91/4 coeurs
	Température du CPU (système stable)	**26**	°C	/
	Température du CPU (2min d'exéc.)	**39**	°C	/
	RAM	4,2	%	/
		0,144426912	Go	/
	Taille du fichier **.pth**	**2619019**	Mo	/
	Taille du fichier programme	**1835**	Mo	
Consom.	Au demarrage	0,55	A	/
		2,9095	W	/
	Durant l'interprétation du programme	0,98	A	/
		5,1548	W	/

On remarque que les performances des inférences sur ONNX Runtime sont beaucoup plus supérieures à celles de PyTorch d'où l'utilité de l'usage d'un écosystème comme ONNX Runtime pour des applications en systèmes embarqués. Sur PyTorch, l'inférence prends en plus de celle d'ONNX Runtime $12ms$ de temps, 276% d'utilisation du CPU et le double de RAM, elle consomme $1W$ de plus. La diminution du temps d'acquisition d'images avec PyTorch est due au temps prolongé de l'inférence, les images sont prêtes au moment du retour de la boucle à l'instruction d'acquisition d'images.

IV.4.3 Tests sur le robot

Vue que notre travail est centrée sur la reconnaissances des panneaux de signalisation, les tests sur le robot se font en s'approchant du panneau et le lancement manuel du programme. Pour automatiser le déclenchement de la classification du

panneau, une segmentation doit être faite au préalable.

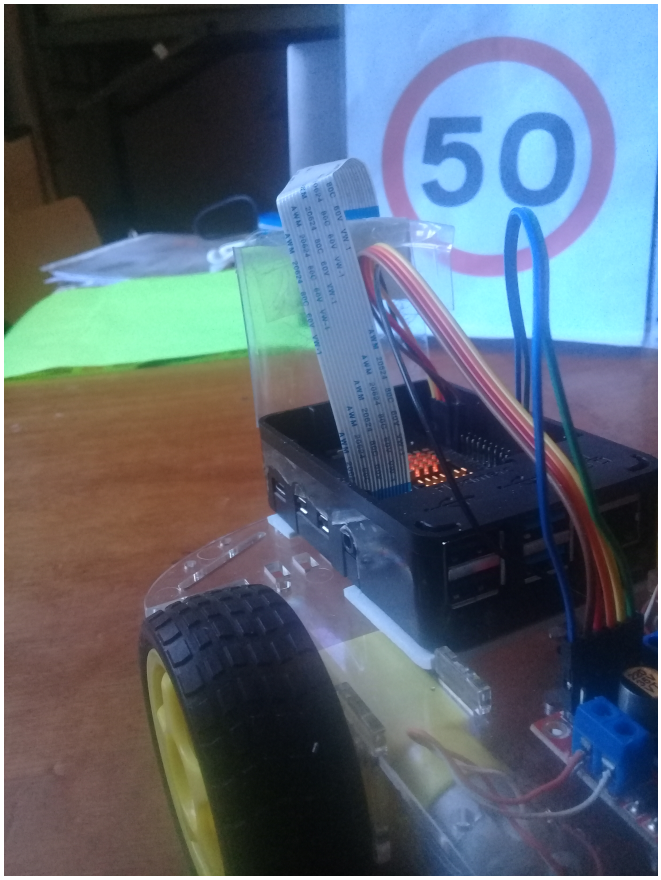


FIGURE IV.7 : Le robot face à un panneau de limitation de vitesse.

L'approche de la caméra de la cible (panneau) devient un inconvénient, elle est utilisée uniquement à des fins de tests du système. Le robot est censé rouler sur une chaussée où tout est visible pour la caméra, les véhicules, les piétons, les infrastructures et aussi les panneaux. Le rôle de la segmentation est justement d'extraire toutes les entités de la scène captée par la camera, ainsi les panneaux de signalisation peuvent être classifiés. (ce que voit le robot durant sa navigation)(Voir Figure IV.8).

(a) L'image de la scène (b) La prédiction

FIGURE IV.8 : Segmentation sémantique d'une image par un réseau de neurones PSPnet [16].

IV.5 Conclusion

Dans ce chapitre, nous sommes parvenus à utiliser un ordinateur à carte unique (SBC) Raspberry Pi 4 Model B à des fins de conception d'un système de vision artificielle destinée à la conduite autonome. En effet, nous avons pu implémenter notre système avec ONNX Runtime sous Python et l'élément principal de ce système est un réseau de neurones de convolution entrainé pour la reconnaissance de panneaux de signalisation routière. Plusieurs mesures on été effectuées notamment le temps d'inférence, d'acquisition et de traitement des images et la consommation énergétique du hardware en fonction de l'exécution des programmes.

Conclusion Générale

Ce mémoire est le fruit d'un projet de fin de cycle de Master réalisé à l'Université de Béjaïa. Il est consacré à la réalisation d'un système de vision artificielle embarquée basé sur le Deep Learning et destiné à la conduite autonome des véhicules. Ce projet est principalement axé sur la reconnaissance des panneaux de signalisation routière. Il nous a permis de de nous confronter à plusieurs problèmes interdisciplinaires comme la robotique et l'Intelligence Artificielle toujours en relation avec l'électronique.

Le premier chapitre est consacré à la présentation des bases et généralités des réseaux de neurones et du Deep Learning. Il nous a permis de présenter et d'approfondir nos connaissances en ce qui concerne les mécanismes de fonctionnement et d'apprentissage des réseaux de neurones artificiels profonds.

Le Chapitre II est consacré aux réseaux de neurones convolutifs (CNNs), les opérations communément utilisées et la présentation de quelques exemples d'architectures populaires de CNNs. Ceci nous a permis de saisir la base et l'élément incontournable en vision artificielle basée sur DL.

Le Chapitre III est consacré à l'apprentissage du CNN sur le logiciel MATLAB à l'aide du Toolbox de Deep Learning. Nous avons réussi à obtenir un modèle de classification de trois panneaux de signalisation routière et d'un feu tricolore.

Le quatrième et dernier chapitre est consacrée à l'implémentation hardware

du modèle obtenu sur un ordinateur compact Raspberry Pi 4 par le biais de la programmation en Python et l'usage de l'écosystème ONNX Runtime. Les temps d'inférence, d'acquisition et de traitement d'images, la température et l'occupation du CPU, l'occupation en mémoire vive et en mémoire de stockage et la consommation énergétique du système sont aussi rapportés.

Les résultats obtenus dans ce document sont satisfaisants, par conséquent, l'objectif principal fixé dans la problématique est atteint : l'apprentissage du réseaux de neurones convolutif est accompli avec 99% de précision du modèle, la conversion du modèle avec ONNX à partir de Matlab pour utilisation sur ONNX Runtime, la réalisation pratique d'un robot mobile et l'implémentation hardware et software de la vision artificielle sur son ordinateur embarqué Raspberry Pi 4 (acquisition des images et leur inférence à travers le réseau de neurones). Une comparaison des performances en terme de temps d'inférences et de consommation énergétique du système avec ONNX Runtime et avec PyTorch est aussi faite.

En perspectives, notre système peut être amélioré par plusieurs moyen à savoir :

- Utilisation d'une base de données plus riche en quantité et en qualité.
- Ajout d'un algorithme ou d'un réseau de neurones de segmentation pour détecter la position et la dimension du panneau sur l'image de la scène.
- Dans le cas où le réseau entre les panneaux de la même famille (comme les panneaux de limitation de vitesse avec un chiffre différent pour chaque panneau, il est possible de faire une segmentation et classifier les chiffres segmentés pour prédire le nombre sur le panneau).
- Amélioration de la navigation autonome avec l'analyse complète de la scène, ajout de capteurs de température et d'humidité, de capteurs de pluie pour identifier le climat et l'état de la chaussée. Aussi, un module d'estimation de la profondeur des objets sur la scène peut être ajoutée [17].

- Intégration d'une carte géographique avec des données GPS.

Bibliographie

[1] A. Turing, D. Ince et A. Turing, *Mechanical intelligence*, sér. Collected works of A.M. Turing. Amsterdam ; New York : New York, NY, U.S.A : North-Holland ; Distributors for the U.S. et Canada, Elsevier Science Pub. Co, 1992, isbn : 978-0-444-88058-1.

[2] R. Francis, T. Estlin, G. Doran et al., "AEGIS autonomous targeting for ChemCam on Mars Science Laboratory : Deployment and results of initial science team use", *Science Robotics*, t. 2, n° 7, eaan4582, 2017. doi : 10.1126/scirobotics.aan4582. eprint : https://www.science.org/doi/pdf/10.1126/scirobotics.aan4582. adresse : https://www.science.org/doi/abs/10.1126/scirobotics.aan4582.

[3] P. Kim, *MATLAB deep learning : with machine learning, neural networks and artificial intelligence*, eng, sér. For professionals by professionals. New York : Springer, 2017, isbn : 978-1-4842-2845-6 978-1-4842-2844-9.

[4] F. Ansari, S. Erol et W. Sihn, "Rethinking Human-Machine Learning in Industry 4.0 : How Does the Paradigm Shift Treat the Role of Human Learning ?", *Procedia Manufacturing*, t. 23C, p. 117-122, avr. 2018. doi : 10.1016/j.promfg.2018.04.003.

[5] R. Szeliski, *Computer Vision : Algorithms and Applications*, sér. Texts in Computer Science. Springer International Publishing, 2022, isbn :

9783030343712. adresse : https://books.google.dz/books?id=A34ZygEACAAJ.

[6] K. Warwick, *Artificial intelligence : the basics*, sér. The basics. New York : Routledge, 2012, ISBN : 978-0-415-56482-3 978-0-415-56483-0 978-0-203-80287-8.

[7] V. N. Murthy, "Synaptic plasticity : Step-wise strengthening", en, *Current Biology*, t. 8, n° 18, R650-R653, sept. 1998, ISSN : 09609822. DOI : 10.1016/S0960-9822(07)00414-9. adresse : https://linkinghub.elsevier.com/retrieve/pii/S0960982207004149 (visité le 22/08/2021).

[8] L. E. Sucar, *Probabilistic graphical models : principles and applications*, eng, sér. Advances in computer vision and pattern recognition. London Heidelberg New York Dordrecht : Springer, 2015, ISBN : 978-1-4471-7054-9 978-1-4471-6698-6.

[9] S. Chakraverty, *Applied artificial neural network methods for engineers and scientists, solving algebraic equations*. New Jersey : World Scientific, 2021, ISBN : 9789811230202.

[10] S. Khan, H. Rahmani, S. A. Ali Shah et M. Bennamoun, *Guide to Convolutional Neural Networks for Computer Vision*, English. Morgan & Claypool Publishers, 2018, OCLC : 1035816520, ISBN : 978-1-68173-021-9. adresse : https://doi.org/10.2200/S00822ED1V01Y201712COV015 (visité le 07/10/2021).

[11] Y. Lecun, L. Bottou, Y. Bengio et P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, t. 86, n° 11, p. 2278-2324, 1998. DOI : 10.1109/5.726791.

[12] K. Simonyan et A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", *arXiv :1409.1556 [cs]*, avr. 2015, arXiv : 1409.1556. adresse : http://arxiv.org/abs/1409.1556 (visité le 26/10/2021).

[13] J. Stallkamp, M. Schlipsing, J. Salmen et C. Igel, "Man vs. computer : Benchmarking machine learning algorithms for traffic sign recognition", *Neural Networks*, n° 0, p. -, 2012, issn : 0893-6080. doi : 10.1016/j.neunet.2012.02.016. adresse : http://www.sciencedirect.com/science/article/pii/S0893608012000457.

[14] P. Viola et M. Jones, "Rapid object detection using a boosted cascade of simple features", in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, t. 1, 2001, p. I-I. doi : 10.1109/CVPR.2001.990517.

[15] E. Shelhamer, J. Long et T. Darrell, "Fully Convolutional Networks for Semantic Segmentation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, t. 39, n° 4, p. 640-651, 2017. doi : 10.1109/TPAMI.2016.2572683.

[16] H. Zhao, J. Shi, X. Qi, X. Wang et J. Jia, "Pyramid Scene Parsing Network", *CoRR*, t. abs/1612.01105, 2016. arXiv : 1612.01105. adresse : http://arxiv.org/abs/1612.01105.

[17] M. Schön, M. Buchholz et K. Dietmayer, "MGNet : Monocular Geometric Scene Understanding for Autonomous Driving", in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, oct. 2021, p. 15 804-15 815.