OpenNas : un cadre adaptable de recherche automatique d'architecture neuronale

Léo Pouy

université PARIS-SACLAY

OpenNas : Un cadre adaptable de recherche automatique d'architecture neuronale

OpenNas : An adaptable neural architecture search framework

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée dans les unités de recherche LIST Laboratoire d'intégration de systèmes et de technologies (Université Paris-Saclay, CEA) et ESTACA'Lab, sous la direction de Chokri MRAIDHA, directeur de recherche, la co-direction de Cherif LAROUCI, directeur de recherche, le co-encadrement de Fouad KHENFRI, enseignant-chercheur, et le co-encadrement de Patrick LESERF, enseignant-chercheur

Thèse soutenue à Laval, le 06 décembre 2023, par

Léo Pouy

Composition du jury

Membres du jury avec voix délibérative

Jean-Philippe BABAU Professeur des Universités, Université de Bretagne Occidentale	Président
Carola DOERR Directrice de Recherche, CNRS – Sorbonne Université	Rapporteur & Examinatrice
Frédéric SAUBION Professeur des Universités, Université d'Angers	Rapporteur & Examinateur
Marc SCHOENAUER Directeur de Recherche, INRIA Saclay - Île-de-France	Examinateur
Ahmed SAMET Maitre de conférences, INSA Strasbourg	Examinateur

THESE DE DOCTORAT
NNT : 2023UPASG089

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : OpenNas : Un cadre adaptable de recherche automatique d'architecture neuronale
Mots clés : Auto-ML, Apprentissage Automatique, Espace de Recherche, Intelligence Artificielle, Optimisation multiobjectif, Recherche Automatique d'Architecture Neuronale

Résumé : Lors de la création d'un modèle de réseau de neurones, l'étape dite du « *fine-tuning* » est incontournable. Lors de ce *fine-tuning*, le développeur du réseau de neurones doit ajuster les hyperparamètres et l'architecture du réseau pour que ce-dernier puisse répondre au cahier des charges. Cette étape est longue, fastidieuse, et nécessite de l'expérience de la part du développeur. Ainsi, pour permettre la création plus facile de réseaux de neurones, il existe une discipline, l'« *Automatic Machine Learning* » (Auto-ML), qui cherche à automatiser la création de *Machine Learning*. Cette thèse s'inscrit dans cette démarche d'automatisation et propose une méthode pour créer et optimiser des architectures de réseaux de neurones (*Neural Architecture Search*).

Pour ce faire, un nouvel espace de recherche basé sur l'imbrication de blocs a été formalisé. Cet espace permet de créer un réseau de neurones à partir de blocs élémentaires connectés en série ou en parallèle pour former des blocs composés qui peuvent eux-mêmes être connectés afin de former un réseau encore plus complexe. Cet espace de recherche a l'avantage d'être facilement personnalisable afin de pouvoir influencer la recherche automatique vers des types d'architectures (VGG, Inception, ResNet, etc.) et contrôler le temps d'optimisation. De plus, il n'est pas contraint à un algorithme d'optimisation en particulier.

Dans cette thèse, la formalisation de l'espace de recherche est tout d'abord décrite, ainsi que des techniques dites d'« encodage » afin de représenter un réseau de l'espace de recherche par un entier naturel (ou une liste d'entiers naturels). Puis, des stratégies d'optimisation applicables à cet espace de recherche sont proposées. Enfin, des expérimentations de recherches d'architectures neuronales sur différents jeux de données et avec différents objectifs en utilisant l'outil développé (nommé OpenNas) sont présentées.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : OpenNas : An adaptable neural architecture search framework
Keywords : Auto-ML, Deep Learning, Artificial Intelligence, Multi-objective optimization, Neural Architecture Search, Search Space

Abstract : When creating a neural network, the "fine-tuning" stage is essential. During this fine-tuning, the neural network developer must adjust the hyperparameters and the architecture of the network so that it meets the targets. This is a time-consuming and tedious phase, and requires experience on the part of the developer. So, to make it easier to create neural networks, there is a discipline called Automatic Machine Learning (Auto-ML), which seeks to automate the creation of Machine Learning. This thesis is part of this Auto-ML approach and proposes a method for creating and optimizing neural network architectures (Neural Architecture Search, NAS).

To this end, a new search space based on block imbrication has been formalized. This space makes it possible to create a neural network from elementary blocks connected in series or in parallel to form compound blocks which can themselves be connected to form an even more complex network. The advantage of this search space is that it can be easily customized to influence the NAS for specific architectures (VGG, Inception, ResNet, etc.) and control the optimization time. Moreover, it is not constrained to any particular optimization algorithm.

In this thesis, the formalization of the search space is first described, along with encoding techniques to represent a network from the search space by a natural number (or a list of natural numbers). Optimization strategies applicable to this search space are then proposed. Finally, neural architecture search experiments on different datasets and with different objectives using the developed tool (named OpenNas) are presented.

Remerciements

Tout d'abord, je souhaite remercier mes encadrants sans qui tout ceci n'aurait pu être possible. Chokri Mraidha, directeur de thèse, qui, malgré la distance, a toujours su se montrer disponible, avenant et de bons conseils. Cherif Larouci, directeur de thèse, qui m'a permis de travailler à l'ESTACA dans les meilleures conditions. Patrick Leserf, encadrant, dont les conseils m'ont toujours permis de prendre du recul sur mon travail et de tirer le meilleur de celui-ci. Enfin, je tenais à remercier tout particulièrement Fouad Khenfri, encadrant. Au commencement de cette thèse, il m'avait promis soutien jusqu'au bout de l'aventure, et n'a jamais failli à cette promesse. Je le remercie pour son temps, son investissement et nos débats éclairés dépassant souvent la fin de la journée. Pour moi, cette thèse est aussi la sienne.

Je remercie aussi l'ensemble des membres du jury. Ainsi, j'exprime ma reconnaissance à Carola Doerr et Frédéric Saubion, pour avoir accepté la charge de rapporter ces travaux de thèse. Je remercie également Marc Schoenauer, Jean-Philippe Babau et Ahmed Samet pour leur participation en tant qu'examinateurs.

Je tiens également à remercier mes collègues de l'ESTACA. Sandrine Pioger, pour tout le soutien qu'elle m'a apporté et qu'elle continue à apporter à tous les doctorants. Sébastien Saudrais et Michaël Chauvin qui m'ont supporté successivement en tant qu'élève, stagiaire, puis doctorant. Enfin, je remercie mes voisins de bureau Quentin Ladurée, Alexandre Barbier et Antoine Poirot, ainsi que les techniciens Yoann Seiller et Raphaël Defossez qui plus que des collègues sont devenus des amis.

Je remercie Nils et Minette, mes camarades félins m'ayant accompagnés lors de mes rédactions nocturnes.

Je remercie mon meilleur ami, Vincent Tisserand, pour toutes les aventures virtuelles, imaginaires et réelles que nous partageons depuis dix ans et qui, pendant cette thèse, m'ont permis de changer d'air et de tenir le coup. Je remercie aussi son fils, mon filleul, pour tout le bonheur qu'il irradie déjà à son âge.

Je remercie très chaleureusement mes parents, Adeline et Jean-Claude, pour tout le soutien et l'amour qu'ils me donnent depuis toujours. Je ne serais jamais allé aussi loin sans eux, et leur suis éternellement reconnaissant.

Je remercie aussi mon frère, Étienne, dont le parcours est très différent du mien et qui me remonte toujours le moral lorsque d'aventure nous nous croisons.

Enfin, je remercie Olivia Tonussi, qui partage ma vie depuis plus de dix ans. Tout au long de cette thèse, elle a partagé le fardeau de mes recherches : mes joies comme mes peines, mes victoires comme mes échecs. Elle a su être présente dans les moments difficiles, et je lui donne toute ma gratitude et ma reconnaissance pour tout le soutien qu'elle m'a fourni.

Table des matières

Introduction et motivation **17**

1 Apprentissage profond et Auto-ML **21**
1.1 Apprentissage profond . . . 23
1.1.1 Introduction . . . 23
1.1.2 Perceptron multicouche . . . 24
1.1.3 Réseau de neurones convolutifs . . . 31
1.1.4 Évolution des architectures de CNN . . . 36
1.2 Auto-ML . . . 43
1.2.1 Méthodologie de développement de *Deep Learning* . . . 43
1.2.2 Qu'est-ce qu'un Auto-ML ? . . . 46
1.2.3 Recherche automatique d'architecture . . . 46
1.2.4 Analyse de l'état de l'art et positionnement des travaux de thèses . . . 54
1.3 Positionnement par rapport à l'état de l'art . . . 56
1.4 Conclusion . . . 59

2 Espace de recherche imbriqué **61**
2.1 Introduction . . . 62
2.2 Présentation de la structure de l'espace de recherche et de sa complexité . . . 63
2.2.1 Blocs élémentaires . . . 63
2.2.2 Blocs composés . . . 64
2.2.3 Architecture neuronale à base des blocs . . . 74
2.2.4 Exemples d'imbrications recréant des CNN connus . . . 77
2.3 Représentation d'une architecture par des entiers . . . 78
2.3.1 Encodage et décodage des pipelines . . . 79
2.3.2 Décodage et encodage des rubans . . . 84
2.4 Conclusion . . . 92

3 Recherche d'architecture neuronale avec l'espace de recherche imbriqué **93**
3.1 Introduction . . . 94
3.2 Formulation du problème . . . 95
3.2.1 Espace de recherche . . . 95
3.2.2 Contraintes d'inégalité . . . 97
3.2.3 Fonctions objectif . . . 98
3.3 Optimisation en double boucle . . . 101
3.4 Optimisation en simple boucle . . . 102
3.4.1 Configuration heuristique d'un Pipeline . . . 105
3.4.2 Configuration heuristique d'un Ruban . . . 108
3.5 Optimisation en seule boucle avec un espace de recherche dynamique . . . 111

3.5.1 Identifiant unique pour les architectures . . . 112
3.6 Conclusion . . . 114

4 Évaluations **115**
4.1 Introduction . . . 116
4.2 Description du framework OpenNas . . . 117
4.3 Expérimentations . . . 119
4.3.1 Étude de convergence . . . 119
4.3.2 Étude d'amélioration de performances . . . 125
4.3.3 Comparaison à d'autres Auto-MLs . . . 126
4.4 Conclusion . . . 131

Conclusion et perspectives **133**
Conclusion . . . 133
Perspectives . . . 134

Liste des publications **137**

A1 Générateur de multiensemble **149**

A2 Générateur de décomposition faible **151**

A3 Générateur de compositions **153**

Table des figures

1 La vision ordinateur consiste à capter et comprendre son environnement . . . 18
2 Différents types de traitement d'image, tirées du jeu de donnée PascalVOC [26] . . . 18

1.1 Sous-ensembles de d'intelligence artificielle . . . 24
1.2 Perceptron, un neurone artificiel seul. . . . 24
1.3 Perceptron multicouche . . . 25
1.4 Descente du gradient . . . 27
1.5 Surentraînement . . . 31
1.6 exemple de déplacement du filtre sur une image en utilisant le même pas 1, 2 et 3 dans les deux axes $(s_h = s_w)$. . . . 33
1.7 exemple de dilatation du filtre en utilisant la même dilatation 1, 2 et 3 dans les deux axes $(d_h = d_w)$. . . . 33
1.8 exemple de marge ajouté sur l'image pour gardé la même taille de l'image d'entrée à la sortie. 34
1.9 Convolution 2D avec un noyau de 3x3 appliquée sur une matrice de dimension 5×5 . . . 35
1.10 LeNet-5 [48] . . . 37
1.11 AlexNet [46] . . . 38
1.12 Représentations de VGG-16 . . . 39
1.13 représentation compacte de GoogLeNet. . . . 40
1.14 Resnet-50 [37] . . . 41
1.15 DenseNet-121 [41] . . . 42
1.16 Méthode de développement d'un modèle de DL, appelée *fine tuning*. . . . 44
1.17 Structure en chaîne . . . 47
1.18 Espace de recherche basé sur les cellules . . . 48
1.19 Architecture trouvée par Auto-DeepLab pour Cityscapes [18]. À gauche les changements de dimensions spatiales le long du modèle, avec le chemin de *downsampling* retenu (flèches noires) ; et à droite la cellule trouvée par la recherche. Extrait de [51]. . . . 49
1.20 Espace de recherche hiérarchique . . . 50
1.21 Morphisme . . . 51
1.22 Apprentissage par Renforcement . . . 52
1.23 Algorithme génétique . . . 53

2.1 Bloc élémentaire avec sa configuration. . . . 64
2.2 Exemples de blocs élémentaires dont la partie pour extraire les caractéristiques est développée. e_1 est un bloc simple représentant une unique couche de convolution 3×3, tandis que e_2 représente un bloc identité (utilisé dans ResNet [37]) . . . 64
2.3 Représentation d'un pipeline de $\mathbb{B}_i$ avec n_i blocs internes. Les options (p, r) pour chaque bloc interne b_{ij} sont aussi représentées ainsi que leur influence sur les dimensions du tenseur de sortie du pipeline. . . . 66
2.4 Exemple de pipelines à 3 emplacements pour organiser 2 blocs élémentaires. . . . 66

2.5 Représentation d'un ruban de $\mathbb{B}_i$ avec n_1 blocs internes. L'influence des options sur les dimensions du ruban est représentée en sortie. 70
2.6 Exemple de Rubans à 3 emplacements et 2 blocs élémentaires. 71
2.7 Exemple d'imbrication sur 3 niveaux. 75
2.8 Blocs CONV et RES de ResNet-50 utilisés comme blocs élémentaires. 78
2.9 Temps d'exécutions des algorithmes 2.7 et 2.8 dans le pire des cas pour décoder un ruban de $\mathbb{R}_i$ en fonction du nombre n de couches et du nombre b d'éléments dans $\mathbb{C}_{i-1}$ 89

3.1 Optimisation de $X = (x_a, x_r, x_p)$ en deux boucles d'optimisation imbriquées. 102
3.2 Exemple d'optimisation conjointe de l'architecture imbriquée x_a et des options x_r et x_p en une seule boucle. 103
3.3 Des pipelines dont les options sont configurées heuristiquement. 105
3.4 Exemple d'options d'un $b_1 \in P_L$ dont les options sont réparties parmi les blocs internes selon l'algorithme 3.1. 108
3.5 Exemple de sélection de sous-ensemble $\mathbb{E}' \subset \mathbb{E}$ de cardinalité $n'_0 = 3$. 111
3.6 Exemple d'une façon d'obtenir le même pipeline avec deux x_a différents par la sélection de deux sous-ensembles $\mathbb{E}'_1$ et $\mathbb{E}'_2$ 113
3.7 Exemple d'une façon d'obtenir des pipelines différents avec le même x_a par la sélection de deux sous-ensembles $\mathbb{E}'_1$ et $\mathbb{E}'_2$ 113

4.1 Diagramme de classes d'OpenNas 118
4.2 Exemple de fronts de Pareto. Extrait de [4] 121
4.3 Résultats de l'application de U-NSGA-III sur l'espace de recherche sur 6 itérations 122
4.4 Résultats de l'application de l'optimisation bayésienne sur l'espace de recherche sur 6 itérations 123
4.5 Architecture retenue par l'optimisation bayesienne 126
4.6 Évolution des fronts de Pareto au fur et à mesure des générations. 128
4.7 Évolution des fronts de Pareto au fur et à mesure des générations de LEMONADE. Extrait de [24]. 128
4.8 Mise à l'échelle logarithmique de la figure 4.6 129

Liste des Algorithmes

2.1 Nombre d'architectures possibles 76
2.2 Conversion d'un m-uplet en décimal 80
2.3 Conversion d'un décimal en m-uplet 80
2.4 Encodage d'un pipeline 82
2.5 Décodage d'un pipeline 84
2.6 Décodage par multiensemble 85
2.7 Décodage par composition faible 86
2.8 Décodage des rubans 89
2.9 Encodage de multiensemble ou composition faible 90
2.10 Encodage des rubans 91

3.1 Algorithme de répartition équitable 107

A1.1 Générateur de multiensembles 150

A2.1 Générateur de compositions faibles 151

A3.1 accel_asc 153

Acronymes

ReLU Unité Linéaire Rectifiée / *Rectified Linear Unit*. 37, 38, 125

Auto-ML Apprentissage automatique automatisé / *Automated Machine Learning*. 17, 19–21, 23, 42–44, 54–57, 62, 131, 133

CNN Réseaux neuronaux convolutionnels / *Convolutional Neural Network*. 7, 17–21, 23, 31, 32, 36–38, 42, 44–46, 50–52, 54, 59, 61–63, 77, 92, 93, 112, 117, 119, 120, 127, 129, 133

DL Apprentissage profond / *Deep Learning*. 9, 17, 19, 20, 23, 24, 31, 44, 50, 56, 59, 118, 133, 135

FC Couche complètement connectée / *Fully Connected layer*. 32, 36–38, 50, 120, 125, 127

FLOPs Opérations en Virgule Flottante / *Floating Point Operations*. 26, 27, 35, 42, 43, 45, 97, 99, 108, 135

GPU Processeurs graphiques / *Graphics Processing Unit*. 31, 37, 42, 55, 126, 128, 129

IA Intelligence Artificielle / *Artificial Intelligence*. 17, 23

LiDAR Détection et estimation de la distance par la lumière / *Light Detection And Ranging*. 17

MACs Multiplications-ACcumulations / *Multiply–ACcumulate*. 45

MBSSE Ingénierie système et logicielle basée modèle / *Model-based Systems and Software Engineering*. 19

ML Apprentissage automatique / *Machine Learning*. 17–19, 23, 46

MLP Perceptron multicouches / *MultiLayer Perceptron*. 24–27, 32

NAS Recherche d'architecture neuronale / *Neural Architecture Search*. 19, 20, 46, 48, 52, 54–56, 58, 59, 62, 115, 116, 119, 123, 126–129, 131, 133, 134

NN Réseau neuronal / *Neural Network*. 24, 25, 46, 54

NSGA-III Non-dominated Sorting Genetic Algorithm III. 121

RL Apprentissage par Renforcement / *Reinforcement learning*. 51, 57

RNN Réseau Neuronal Récurrent / *Recurent Neural Network*. 51

U-NSGA-III Unified Non-dominated Sorting Genetic Algorithm III. 10, 115, 119, 121, 122, 124, 125, 127, 131, 134

Notations

Nombres & Tableaux

x	Un scalaire
X	Un vecteur ou un uplet
$\mathbf{X}$	Une matrice
$\mathbf{X}$	Un tenseur
$\|\|X\|\|$	Taille d'un vecteur ou d'un uplet

Ensemble

$\mathbb{E}$	Un ensemble
$\mathbb{R}$	L'ensemble des réels
$\mathbb{N}$	L'ensemble des entiers naturels
$\{a, b, c\}$	L'ensemble contenant les scalaires a, b et c
$[0, n]$	L'ensemble des réels de 0 à n
$[\![0, n]\!]$	L'ensemble des entiers naturels de 0 à n
$\#\mathbb{E}$	Le cardinal de l'ensemble $\mathbb{E}$
$\binom{n}{k}$	Le coefficient binomial, « n parmi k »
$(\mathbb{E}, \mathcal{M})$	Le multiensemble avec $\mathbb{E}$ comme ensemble support et une multiplicité $\mathcal{M}$
$\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$	Le nombre de k-combinaisons avec répétition d'un ensemble à n éléments, « n multi-choisi k »

Indexation

X_i	Élément i du vecteur X
$\mathbf{X}_{i,j}$	Élément (i, j) de la matrice $\mathbf{X}$, avec i la hauteur et j la largeur
$\mathbf{X}_{i,j,k}$	Élément (i, j, k) du tenseur $\mathbf{X}$, avec i la hauteur, j la largeur et k le canal
E_i	Élément i d'un ensemble dénombrable $\mathbb{E}$ composé d'uplets
e_i	Élément i d'un ensemble dénombrable $\mathbb{E}$ composé de scalaires
$\mathbb{I}_{\mathbb{E}}$	Ensemble index d'un ensemble $\mathbb{E}$

Fonctions

$\mathcal{F} : \mathbb{E} \to \mathbb{F}$	Une application $\mathcal{F}$ de $\mathbb{E}$ vers $\mathbb{F}$
$\mathrm{sigmoid}(x)$	Fonction sigmoïde, i.e. $(1 + \exp(-x))^{-1}$
$\mathrm{ReLU}(x)$	Fonction Unité Linéaire Rectifiée (ReLU), i.e. $\max(0, x)$
$\arg\max(X)$	Renvoie l'index correspondant à l'élément le plus élevé de X
$\mathrm{softmax}(X) = Y$	Fonction $\mathrm{softmax}$, i.e. $\forall j \in [\![1, \|\|X\|\|]\!], Y_j = \frac{\exp(X_j)}{\sum_{k=1}^{\|\|X\|\|} \exp(X_k)}$

Introduction et motivation

Le véhicule autonome

Le véhicule autonome représente le futur de l'industrie automobile. Certains prototypes circulent déjà sur les routes et leur niveau d'autonomie est sans cesse croissant. Ces niveaux d'autonomies sont au nombre de six, le plus élevé correspondant au véhicule 100% autonome, piloté par une intelligence artificielle (IA). Concevoir et modéliser de tels systèmes demande de résoudre de nombreux verrous technologiques et scientifiques. L'un d'entre eux consiste à permettre au véhicule de prendre des décisions rapidement et en toutes circonstances, aussi bien ou même mieux qu'un conducteur expérimenté. Cette fonction de prise de décision multicritère ne sera pas possible sans avancées considérables dans le développement de l'IA.

À ce jour, il n'existe pas de méthode rigoureuse ou standard permettant au concepteur d'un système autonome d'intégrer efficacement les fonctions de prise de décision basées sur des techniques d'IA. L'état de l'art montre que parmi ces techniques d'IA, l'apprentissage automatique (*Machine Learning*, ML), et plus particulièrement l'apprentissage profond (*Deep Learning*, DL) avec ses réseaux neuronaux convolutifs (*Convolutional Neural Network*, CNN) est un des outils les plus puissant pour la reconnaissance de l'environnement (Computer Vision), première étape d'un processus de prise de décision. Or, le développement de DL nécessite de l'expérience et est un processus empirique et fastidieux. Ainsi, dans un premier temps, nous cherchons à être capable de concevoir aisément un algorithme de DL. Pour ce faire, les techniques d'*Automatic Machine Learning* (Auto-ML) sont particulièrement intéressantes. En effet, l'Auto-ML permet la création automatisée d'algorithmes de DL, notamment une recherche d'architecture neuronale (nombre de couches, type de convolutions, etc.) et une optimisation des hyper-paramètres d'entraînement (taux d'apprentissage, *dropout*, *weight decay*, etc.).

Vision par Ordinateur

Pour pouvoir fonctionner dans l'espace où il évolue, un système autonome doit pouvoir reconnaître son environnement. On appelle « Vision Ordinateur » (*Computer Vision*) la discipline cherchant à effectuer cette tâche. Pour cela, des capteurs sont utilisés afin de recueillir des données de l'environnement. Dans le cas des véhicules autonomes, il s'agit de la route, la position du véhicule sur celle-ci, la signalisation, les autres usagers, etc. Pour recueillir ces informations, on peut utiliser des caméras [9, 18], des radars [10], ou des LiDARs [10, 31] et il faut par la suite analyser ces données afin de pouvoir les utiliser (figure 1).

On distingue différentes « tâches » pour la Vision Ordinateur : la classification, la détection d'objet et la segmentation (figure 2). La classification est la plus basique des analyses : elle cherche à déterminer le *label* d'une image, c'est-à-dire son contenu. Le jeu de données

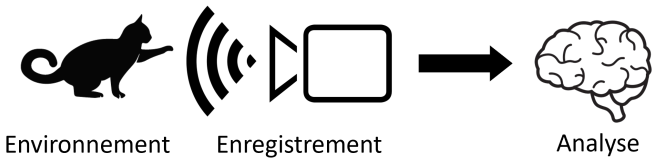


Figure 1 – La vision ordinateur consiste à capter et comprendre son environnement

le plus connu de classification d'image est MNIST [48], qui est constitué de chiffres écrits à la main, le but étant de reconnaître lesdits chiffres. La détection d'objet quant à elle détermine non seulement le contenu d'une image, mais aussi la position des objets (généralement représentée sous la forme de *bounding box*). Enfin, la segmentation est une classification de chaque pixel de l'image. On crée ainsi une partition de l'image en séparant les différents objets présents sur cette dernière.

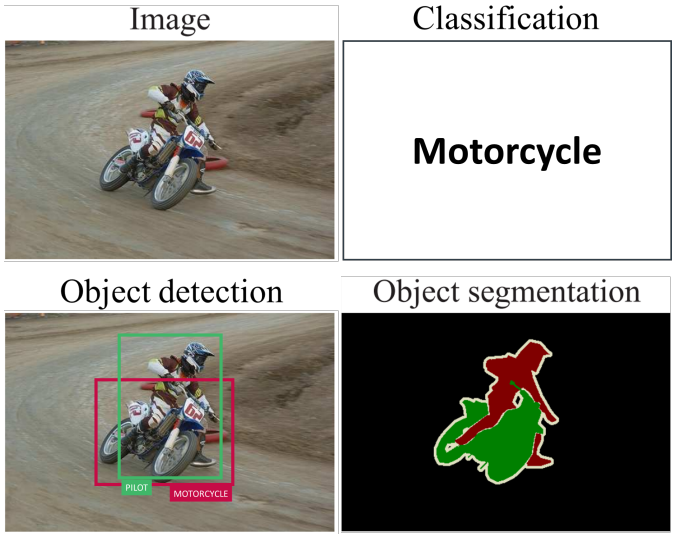


Figure 2 – Différents types de traitement d'image, tirées du jeu de donnée PascalVOC [26]

Pour réaliser ces traitements d'images, deux approches sont possibles : des techniques utilisant des techniques et algorithmes créés par des humains, ou de l'Apprentissage automatique / *Machine Learning* (ML). Depuis l'explosion du ML qui s'opère depuis les années 2010, c'est cette dernière technique qui domine la recherche dans le domaine de la vision par ordinateur [56]. Ainsi, les travaux de cette thèse se portent sur le développement de techniques facilitant le développement d'algorithmes de ML (et plus particulièrement de CNN).

Contribution de la thèse

L'objectif de la thèse est le développement de méthodes permettant de faciliter le développement d'intelligences artificielles, notamment pour les concepteurs de véhicules autonomes. Pour cela, l'Auto-ML est un outil intéressant, qui pourra être adapté à une utilisation dans un contexte d'ingénierie système et logicielle basée sur des modèles (MBSSE). Plus particulièrement, la thèse porte sur l'élaboration d'une méthode de recherche automatique d'architecture neuronale (*Neural Architecture Search*, NAS) simple à mettre en place (afin que même les utilisateurs les plus inexpérimentés puissent développer un modèle de DL aisément), mais aussi adaptable (afin que les utilisateurs plus expérimentés puissent utiliser l'outil de façon plus poussée en apportant leur expertise). Nous avons nommé cette méthode « Open-NAS ».

Un espace de recherche adaptable :

Pour pouvoir effectuer une NAS, la première étape est d'avoir un espace de recherche contenant les architectures possibles.

La première contribution de cette thèse est donc une formalisation d'un modèle d'espace de recherche, dit « imbriqué ». Cet espace de recherche, inspiré des espaces de recherches hiérarchiques [52, 51, 90], vise à être le plus adaptable possible. En effet, l'idée du modèle est de laisser un maximum de liberté dans la définition de l'espace de recherche, tout en gardant une certaine simplicité d'utilisation. Son adaptabilité vient aussi du fait de pouvoir aisément appliquer un algorithme d'optimisation quelconque.

Des stratégies d'optimisation pour algorithmes méta-heuristiques :

La deuxième contribution de cette thèse est un ensemble de propositions de stratégies d'optimisation utilisant des algorithmes d'optimisation méta-heuristiques applicables à cet espace de recherche.

Trois stratégies sont proposées dans la thèse pour résoudre ce problème de synthèse d'architectures neuronales. En effet, chaque nouvelle stratégie réduit les degrés de libertés de l'espace de recherche afin d'augmenter l'efficacité de l'optimisation. C'est-à-dire pouvoir trouver une architecture optimale en moins de temps.

Plan général de la thèse

Le contenu de cette thèse est organisé en quatre chapitres selon le plan donné ci-dessous :

Chapitre 1 :

Ce chapitre présente les notions de DL et d'Auto-ML nécessaires à la compréhension des contributions de la thèse. Dans ce chapitre, la notion de DL, et plus particulièrement de CNN, est expliquée. La thèse portant sur la NAS, un descriptif des architectures de CNN qui ont marqué l'histoire du ML est donné, afin que le lecteur soit à l'aise avec les différentes structures d'architecture de CNN. Cette première partie s'adresse donc principalement aux lecteurs

novices en DL. Ensuite, une description et un état-de-l'art de l'Auto-ML, et notamment des espaces de recherche et des techniques d'optimisation pour la NAS, est fait.

Chapitre 2 :

Ce chapitre est consacré à la formalisation de l'espace de recherche imbriqué développé pour Open-NAS. Il y est tout d'abord présenté le principe de blocs, pouvant s'imbriquer pour former des structures et finalement une architecture de CNN. L'espace de recherche est présenté sous forme d'ensembles mathématiques, notamment de n-uplets et de multiensembles. Il est ensuite présenté les algorithmes permettant l'encodage des blocs en entiers naturels, permettant ultimement de représenter une architecture de CNN entière à l'aide d'un simple entier naturel.

Chapitre 3 :

Dans ce chapitre, les propositions de stratégies d'optimisation sont présentées, après une formulation du problème. Les trois stratégies proposées sont une optimisation directe en double boucle, puis une optimisation avec une seule boucle (à l'aide d'algorithmes heuristiques), et enfin une optimisation avec un espace de recherche dynamique.

Chapitre 4 :

Dans ce chapitre, une description de l'implémentation de OpenNas est tout d'abord faite. Puis différentes expérimentations menées à l'aide de cet outil sont décrites. Une première expérimentation est la comparaison d'une optimisation avec un algorithme génétique ou avec une optimisation bayésienne, dans le cas d'une stratégie de simple boucle d'optimisation. Puis, une expérience améliorant les performances de VGG [76] sur la base de données CIFAR-10 [45] a été menée.

1 - Apprentissage profond et Auto-ML

Dans ce chapitre, les clés de compréhension du sujet et des enjeux de cette thèse sont données. Tout d'abord, la première section décrit le fonctionnement de l'apprentissage profond (*Deep Learning*), puis présente une série d'architectures qui ont marqué l'histoire du *Deep Learning* de par leurs idées ayant chacune grandement contribué à améliorer les performances atteignables avec l'apprentissage profond. Ensuite, un état-de-l'art de l'apprentissage automatique automatisé (Automated Machine Learning, Auto-ML) sera fait, en se concentrant sur les techniques de recherche d'architectures de *Deep Learning* et les algorithmes d'optimisations qui leur sont associés.

1.1 Apprentissage profond . . . 23
1.1.1 Introduction . . . 23
1.1.2 Perceptron multicouche . . . 24
Propagation . . . 26
Descente du gradient . . . 27
Rétropropagation du gradient et entraînement . . . 29
1.1.3 Réseau de neurones convolutifs . . . 31
Couche de convolution . . . 32
Couche de sous-échantillonnage (*pooling*) . . . 35
1.1.4 Évolution des architectures de CNN . . . 36
LeNet-5 [48] . . . 36
AlexNet [46] . . . 37
VGG-16 [76] . . . 38
GoogLeNet [85] . . . 39
ResNet [37] . . . 41
Densenet [41] . . . 41
Récapitulatif concernant les architectures historiques . . . 42
1.2 Auto-ML . . . 43
1.2.1 Méthodologie de développement de *Deep Learning* . . . 43
Critères d'évaluation d'un CNN . . . 45
1.2.2 Qu'est-ce qu'un Auto-ML ? . . . 46
1.2.3 Recherche automatique d'architecture . . . 46
Espaces de recherche . . . 46
Techniques d'optimisation . . . 51

1.2.4 Analyse de l'état de l'art et positionnement des travaux de thèses . . . 54
Espaces de recherches . . . 54
Techniques d'optimisations . . . 55
Challenges de l'Auto-ML . . . 56
1.3 Positionnement par rapport à l'état de l'art . . . 56
1.4 Conclusion . . . 59

1.1 . Apprentissage profond

1.1.1 . Introduction

Le domaine de l'intelligence artificielle est vaste. D'après Le Robert [2], il s'agit de l'« ensemble des théories et des techniques développant des programmes informatiques complexes capables de simuler certains traits de l'intelligence humaine (raisonnement, apprentissage...) ». Ainsi, n'importe quel algorithme peut être considéré comme de l'intelligence artificielle, et on en distingue deux catégories : l'IA symbolique [3] et le connexionnisme [1]. L'IA symbolique regroupe les algorithmes basés sur des manipulations de symboles tandis que le connexionnisme se base sur des unités connectées simulant un réseau de neurones. Dans cette thèse, nous allons nous intéresser à l'IA connexsionniste.

L'Apprentissage automatique / *Machine Learning* (ML) est un sous-domaine de l'intelligence artificielle (figure 1.1) cherchant à permettre aux machines d'apprendre de résoudre un problème donné. C'est-à-dire que la machine, au départ naïve, va au fur et à mesure de son entraînement augmenter la justesse de ses prédictions sur les données d'apprentissage. Parmi les différentes techniques d'apprentissage automatique, on distingue deux catégories : l'apprentissage automatique traditionnel et l'apprentissage profond (*Deep Learning*, DL).

Le ML traditionnel regroupe, entre autres, les techniques suivantes [35] : K plus proches voisins, régression linéaire, régression logistique, machines à vecteurs de support, arbres de décision et forêts aléatoires. Cependant, cette thèse se concentre sur les algorithmes d'apprentissage profond, qui sont les plus utilisés dans le cadre de la vision par ordinateur.

Les algorithmes d'Apprentissage profond / *Deep Learning* (DL) s'inspirent des réseaux de neurones du cerveau humain. Le terme « profond » vient du fait que ces réseaux sont constitués de plusieurs couches de neurones. Plus un réseau possède de couches, plus il est profond.

Dans cette section sont données les bases de ce qu'est l'apprentissage profond, en commençant par le perceptron multicouche (section 1.1.2) pour finir par les Réseaux neuronaux convolutionnels / *Convolutional Neural Network* (CNN) (section 1.1.3). Ensuite, une présentation d'architectures de CNN ayant chacune apporté une innovation marquante est faite (section 1.1.4). Les lecteurs déjà familiarisés avec les concepts de DL et plus particulièrement de CNN pourront donc passer directement à la section 1.2 présentant le principe d'Auto-ML et un état de l'art de ce dernier.

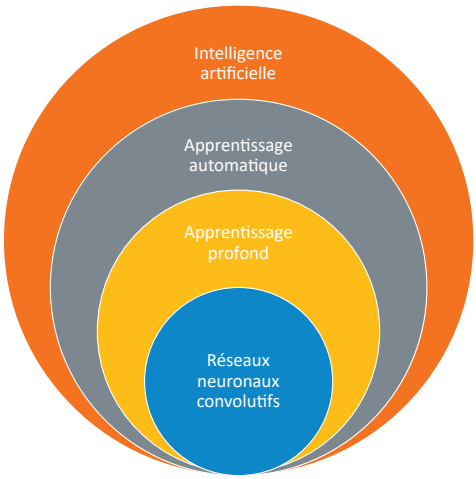


Figure 1.1 – Sous-ensembles de d'intelligence artificielle

1.1.2 . Perceptron multicouche

Afin de comprendre ce qu'est l'apprentissage profond (*Deep Learning*, DL), cette section présente la notion de perceptron (neurone artificiel), puis de perceptron multicouche (réseau de neurones artificiels) ainsi que la notion de propagation et rétropropagation permettant l'apprentissage.

En 1957, Rosenblatt [70] présente la notion de perceptron, que l'on peut considérer comme un neurone artificiel seul ; lorsque l'on parle d'intelligence artificielle, les termes « perceptron » et « neurones » sont équivalents. Un neurone se comporte de la façon suivante (figure 1.2) : il prend une valeur x en entrée à laquelle on applique un poids w et un biais b tel que $z = w \times x + b$ puis on applique une fonction d'activation φ pour obtenir la sortie du neurone (aussi appelée activation du neurone) y tel que $y = \varphi(z)$. Les fonctions d'activation les plus utilisées sont les fonctions tangente hyperbolique ($\tanh(z)$), sigmoïde ($\mathrm{sigmoid}(z) = \frac{1}{1+e^{-z}}$), Unité Linéaire Rectifiée ($\mathrm{ReLU}(z) = \max(0, z)$).

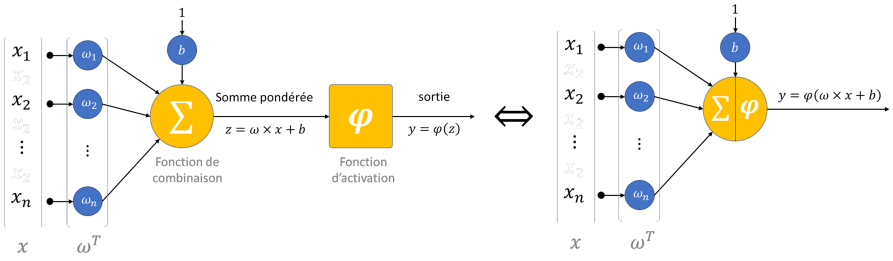


Figure 1.2 – Perceptron, un neurone artificiel seul.

Puis en 1986, Rumelhart [72] propose un perceptron multicouche (*MultiLayer Perceptron*, MLP) et donc le premier réseau de neurones artificiels, ou *Artificial Neural Network* (NN). On

retrouve plusieurs dénominations pour décrire un NN : réseau neuronal, algorithme de *Deep Learning*, ou encore modèle de *Deep Learning*.

L'architecture d'un MLP se compose de plusieurs couches, chacune contenant des neurones interconnectés. La configuration de cette architecture est définie par le nombre de couches, le nombre de neurones dans chaque couche et la fonction d'activation utilisée pour chaque neurone. Dans la figure 1.3 est décrite l'architecture d'un MLP comprenant une couche d'entrée L_0 qui reçoit les entrées du système. Cette couche contient n_0 neurones qui représentent le nombre d'entrées dans le système. Ce MLP a également une couche de sortie qui produit la sortie du système. Cette couche contient n_{l+1} neurones, où l est le nombre de couches cachées entre les deux couches d'entrée et de sortie. Le MLP peut être configuré par l couches cachées $\mathbb{L} = \{L_1, L_2, ..., L_k, ..., L_l\}$, où chaque couche L_k est constituée de n_k neurones $a^{(k)} = [a_1^{(k)}, a_2^{(k)}, ..., a_i^{(k)}, ..., a_{n_k}^{(k)}]^T$. Chaque neurone $a_i^{(k)}$ de la couche L_k est connecté à tous les neurones de la couche précédente L_{k-1}, et chaque connexion est pondérée par un poids w_{ij} qui relie le neurone i de la couche k au neurone j de la couche $k-1$. Par conséquent, la sortie du neurone $a_i^{(k)}$ est l'application de la fonction d'activation φ_k à la somme des entrées de ce neurone plus son biais (pour simplifier la figure 1.3, le biais n'est pas illustré), ce qui donne :

$$\forall k \in [\![1, l+1]\!], \forall i \in [\![1, n_k]\!], a_i^{(k)} = \varphi_k(\sum_{j=1}^{n_{k-1}} (w_{ij} \times a_j^{(k-1)} + b_i^{(k)})) \tag{1.1}$$

Pour la couche d'entrée, l'activation (ou sortie) du neurone est égale à chaque donnée d'entrée x_i : $\forall i \in [\![1, n_0]\!], a_i^{(0)} = x_i$.

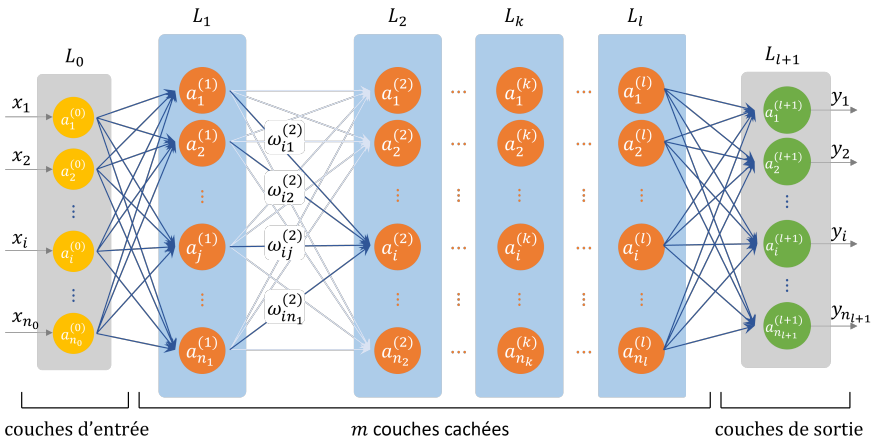


Figure 1.3 – Perceptron multicouche (*MultiLayer Perceptron*, MLP).

Propagation

La propagation est le fait de « propager » une donnée de neurone en neurone, couche par couche, depuis la couche d'entrée jusqu'à la couche de sortie du réseau. Pour calculer la sortie d'une couche L_k en fonction de la précédente, une matrice de poids $\mathbf{W}^{(k)}$ liant les n_k neurones d'une couche L_k aux n_{k-1} neurones d'une couche L_{k-1} ainsi qu'un vecteur des biais $B^{(k)}$ de la couche k sont définis tels que :

$$\mathbf{W}^{(k)} = \begin{bmatrix} w_{11}^{(k)} & \cdots & w_{1n_{k-1}}^{(k)} \\ \vdots & \ddots & \vdots \\ w_{n_k1}^{(k)} & \cdots & w_{n_kn_{k-1}}^{(k)} \end{bmatrix} \; ; \; B^{(k)} = \begin{bmatrix} b_1^{(k)} \\ \vdots \\ b_{n_k}^{(k)} \end{bmatrix}$$

Le vecteur d'activation $a^{(k)}$ de la couche k peut être calculé tel que $a^{(k)} = \varphi_k(\mathbf{W}^{(k)}a^{(k-1)} + b^{(k)})$, où $a^{(k-1)}$ est le vecteur de sortie de la couche L_{k-1} et φ_k est la fonction d'activation de la couche L_k. Ainsi, en calculant successivement la sortie d'une couche en fonction de la précédente, on peut, en partant de la couche d'entrée du réseau, calculer l'activation de sa couche de sortie. Cette opération s'appelle la « propagation », car l'information en entrée se propage le long du réseau jusqu'à sa sortie.

Les poids et les biais d'un neurone sont appelés « paramètres », car ce sont eux qui se mettent à jour lors de l'entraînement et qui permettent au réseau d'apprendre. Chaque neurone d'une couche L_k a donc un nombre de paramètres égal au nombre de neurones n_{k-1} de la couche L_{k-1} précédente, plus un paramètre correspondant à son biais. Ainsi, le nombre de paramètres $p^{(k)}$ d'une couche L_k est donné par l'équation suivante :

$$p^{(k)} = n_k \left(n_{k-1} + 1\right) \tag{1.2}$$

Et le nombre total de paramètres du réseau p correspond à la somme des paramètres de toutes les couches cachées $\mathbb{L}$ et de la couche de sortie L_{l+1} :

$$p = \sum_{k=1}^{l+1} p^{(k)} = \sum_{k=1}^{l+1} n_k \left(n_{k-1} + 1\right) \tag{1.3}$$

Pour calculer le nombre d'Opérations en Virgule Flottante (FLOPs) pour un MLP, il faut prendre en compte les dimensions des couches et les opérations effectuées lors de la propagation en avant à travers les couches. Pour chaque couche, les opérations effectuées sont la multiplication matricielle entre les activations de la couche précédente et la matrice des poids, suivies de l'addition du biais et de l'application d'une fonction d'activation. Pour une couche k entièrement connectée (*dense layer*) avec n_{k-1} neurones d'entrée et n_k neurones de sortie, il y a $n_{k-1} \cdot n_k$ multiplications (de la multiplication matricielle) et $n_{k-1} \cdot n_k + n_k$ additions (de la multiplication matricielle et de l'addition des biais). Donc, le total des opérations pour une seule couche est :

$$\forall h, w, l : FLOP(L_k) = 2 \cdot (n_{k-1} \cdot n_k) + n_k \tag{1.4}$$

Pour calculer le nombre total de FLOPs pour un modèle MLP, nous additionnons les FLOPs de chaque couche :

$$FLOP(MLP) = \sum_{k=1}^{l+1} FLOP(L_k) \tag{1.5}$$

Descente du gradient

Pour entraîner le réseau, on utilise le principe de descente du gradient [34, 71]. Cette technique permet de trouver les paramètres correspondants au minimum d'une fonction grâce à la dérivée de ladite fonction par rapport à chacune de ses variables (ou paramètres). L'algorithme de descente du gradient est comme suit :

1. On prend un point initial $X = (x_1, \ldots, x_n)$
2. Pour chaque paramètre $x_1, \ldots, x_n$, on calcule la dérivée partielle de f au point X : $\frac{\partial f}{\partial x_i}(X)$
3. On se déplace vers un nouveau point (plus proche du minimum) X tel que $\forall i x_i := x_i - \lambda \frac{\partial f}{\partial x_i}(x)$, où λ est le « taux d'apprentissage » (*learning rate*).
4. Les étapes 2 et 3 sont répétées un certain nombre d'itérations.

La figure 1.4 illustre une descente du gradient sur 4 itérations permettant de s'approcher du minimum de la fonction x^2. En partant d'un point $x = 1.33$, on calcule, avec un taux d'apprentissage $\lambda = 0.2$, $x - \lambda \frac{dx^2}{dx} = 1.33 - 0.2 \times 2 \times 1.33 = 0.798$. Le nouveau point est donc $x = 0.798$, ce qui est effectivement plus proche du minimum de x^2, car $0.798^2 = 0.637 < 1.33^2 = 1.77$. On voit sur la figure qu'à chaque itération, on s'approche encore un peu plus du minimum.

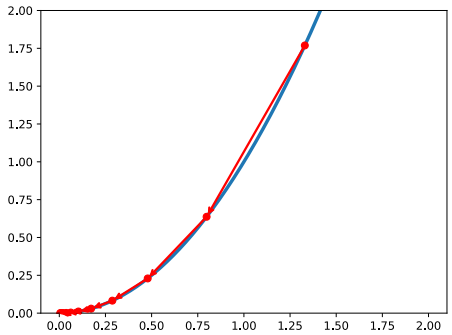


Figure 1.4 – Exemple d'une descente du gradient sur une fonction de coût en x^2

Dans le cadre d'un réseau neuronal, la fonction dont on doit trouver le minimum est la « fonction de coût » (*loss function*) $J : [0,1]^{n_{l+1}} \to [0,1]$ qui dépend de l'erreur de la

prédiction. Cette erreur est la différence entre la sortie attendue Y et la sortie $\hat{Y} = a^{(l+1)}$ prédite par le réseau. En reprenant l'exemple du réseau illustré par la figure 1.3 en supposant qu'il a deux sorties $n_{l+1} = 2$, propageons une donnée dont on sait qu'elle est sensée activer le premier neurone $(Y_1 = 1)$ et pas le second $(Y_2 = 0)$. Mais imaginons que la prédiction donne $\hat{Y}_0 = 0.3$ et $\hat{Y}_1 = 0.6$. Alors, en considérant une fonction de coût quadratique moyenne $(J = \frac{1}{n_{l+1}} \sum_i^{n_{l+1}} (Y_i - \hat{Y}_i)^2)$, la valeur de cette fonction sera de $J = \frac{\left(Y_0 - \hat{Y}_0\right)^2 + \left(Y_1 - \hat{Y}_1\right)^2}{2} = \frac{(1-0,3)^2 - (0-0,6)^2}{2} = 0,425$. Le but de l'entraînement sera donc de réduire la valeur de cette fonction de coût (qui va réduire l'erreur) grâce à la descente du gradient.

Pour ce faire, nous avons besoin de la dérivée de la fonction de coût par rapport à chaque paramètre, c'est-à-dire les poids et le biais. On va donc calculer, pour tout neurone de la couche de sortie $a_i^{(l+1)}$, $\frac{\partial J}{\partial w_{ij}^{(l+1)}}$ (on rappelle que $w_{ij}^{(l+1)}$ est le poids liant le neurone i de la couche $(l+1)$ au neurone j de la couche l) et $\frac{\partial J}{\partial b_i^{(l+1)}}$. Cela permet ainsi de mettre à jour les poids et les biais de la dernière couche en suivant l'algorithme de descente du gradient :

$$w_{ij}^{(l+1)} := w_{ij}^{(l+1)} - \lambda \frac{\partial J}{\partial w_{ij}^{(l+1)}} \tag{1.6}$$

$$b_i^{(l+1)} := b_i^{(l+1)} - \lambda \frac{\partial J}{\partial b_i^{(l+1)}} \tag{1.7}$$

Pour calculer $\frac{\partial J}{\partial w_{ij}^{(l+1)}}$, on utilise le principe de dérivation en chaîne, permettant d'énoncer l'équation suivante :

$$\frac{\partial J}{\partial w_{ij}^{(l+1)}} = \frac{\partial z_i^{(l+1)}}{\partial w_{ij}^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial J}{\partial a_i^{(l+1)}} \tag{1.8}$$

Or

1. $\frac{\partial z_i^{(l+1)}}{\partial w_{ij}^{(l+1)}} = a_j^{(l)}$, car $z_i^{(l+1)} = \sum_j w_{ij}^{(l+1)} a_j^{(l)} + b_i^{(l+1)}$.
2. $\frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} = \varphi'\left(z_i^{(l+1)}\right)$ la dérivée de la fonction d'activation.
3. $\frac{\partial J}{\partial a_i^{(l+1)}}$ la dérivée partielle de la fonction de coût ne peut pas être simplifiée dans le cas général. Notons cependant, dans le cas déjà énoncé de l'erreur quadratique, une simplification peut être faite de la façon suivante : $\frac{\partial J}{\partial a_i^{(l+1)}} = 2\left(a_i^{(l+1)} - \hat{a}_i^{(l+1)}\right)$.

On peut donc réécrire l'équation 1.8 ainsi :

$$\frac{\partial J}{\partial w_{ij}^{(l+1)}} = a_j^{(l)} \varphi'\left(z_i^{(l+1)}\right) \frac{\partial J}{\partial a_i^{(l+1)}} \tag{1.9}$$

De même, on peut exprimer la dérivée de la fonction coût par rapport au biais ainsi :

$$\frac{\partial J}{\partial b_i^{(l+1)}} = \varphi'\left(z_i^{(l+1)}\right) \frac{\partial J}{\partial a_i^{(l+1)}} \tag{1.10}$$

Cependant il faut aussi mettre à jour les paramètres des couches cachées. Pour cela, on utilise une technique appelée la rétropropagation du gradient, faisant l'objet de la prochaine section.

Rétropropagation du gradient et entraînement

La rétropropagation du gradient [72] (*backpropagation*) permet de mettre à jour les paramètres d'un réseau neuronal en propageant le gradient de l'erreur depuis la couche de sortie vers la première couche cachée ; donc le gradient est propagé dans le sens inverse de la propagation présentée en section 1.1.2 d'où le nom « rétropropagation ».

Similairement à la couche de sortie (équation 1.8), on utilise le principe de dérivation en chaîne pour calculer le gradient de l'erreur par rapport à un poids d'une couche cachée L_k :

$$\frac{\partial J}{\partial w_{ij}^{(k)}} = \frac{\partial z_i^{(k)}}{\partial w_{ij}^{(k)}} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}} \frac{\partial J}{\partial a_i^{(k)}} \tag{1.11}$$

$$\frac{\partial J}{\partial w_{ij}^{(k)}} = a_j^{(k-1)} \varphi' \left(z_i^{(k)} \right) \frac{\partial J}{\partial a_i^{(k)}} \tag{1.12}$$

Et pour le biais :

$$\frac{\partial J}{\partial b_i^{(k)}} = \frac{\partial z_i^{(k)}}{\partial b_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}} \frac{\partial J}{\partial a_i^{(k)}} \tag{1.13}$$

$$\frac{\partial J}{\partial b_i^{(k)}} = \varphi' \left(z_i^{(k)} \right) \frac{\partial J}{\partial a_i^{(k)}} \tag{1.14}$$

Pour calculer $\frac{\partial J}{\partial a_i^{(k)}}$, on utilise une fois de plus la dérivation en chaîne :

$$\frac{\partial J}{\partial a_i^{(k)}} = \sum_{h=1}^{n_{k+1}} \frac{\partial z_h^{(k+1)}}{\partial a_i^{(k)}} \frac{\partial a_h^{(k+1)}}{\partial z_h^{(k+1)}} \frac{\partial J}{\partial a_h^{(k+1)}} \tag{1.15}$$

$$\frac{\partial J}{\partial a_i^{(k)}} = \sum_{h=1}^{n_{k+1}} w_{hi}^{(k+1)} \varphi' \left(z_h^{(k+1)} \right) \frac{\partial J}{\partial a_j^{(k+1)}} \tag{1.16}$$

Résumons les équations de la rétropropagation du gradient permettant de calculer l'in-

fluence des poids et des biais sur l'erreur dans tout le réseau :

$$\sigma_i^{(l+1)} = \frac{\partial J}{\partial a_i^{(l+1)}} \quad \text{pour la couche de sortie}$$

$$\sigma_i^{(k)} = \sum_{h=1}^{n_{k+1}} w_{hi}^{(k+1)} \varphi' \left(z_h^{(k+1)} \right) \sigma_h^{(k+1)} \quad \text{pour les couches caches k} \leq \text{l}$$

$$\frac{\partial J}{\partial w_{ij}^{(k)}} = a_j^{(k-1)} \varphi' \left(z_i^{(k)} \right) \sigma_i^{(k)} \quad \text{pour les poids de toutes les couches k} \leq \text{l} + 1$$

$$\frac{\partial J}{\partial b_i^{(k)}} = \varphi' \left(z_i^{(k)} \right) \sigma_i^{(l)} \quad \text{pour les biais de toutes les couches k} \leq \text{l} + 1$$

Ces équations servent ensuite à mettre à jour les poids et les biais selon la descente du gradient (équations 1.6 et 1.7) :

$$\forall k \leq l+1,\ w_{ij}^{(k)} := w_{ij}^{(k)} - \lambda \frac{\partial J}{\partial w_{ij}^{(k)}} \tag{1.17}$$

$$\forall k \leq l+1,\ b_i^{(k)} := b_i^{(k)} - \lambda \frac{\partial J}{\partial b_i^{(k)}} \tag{1.18}$$

Ainsi, un entraînement de réseau neuronal se déroule de la façon suivante. La rétropropagation du gradient (donc la descente du gradient) est effectuée une fois sur chaque élément du jeu de données d'entraînement. À chaque fois que ce passage sur toutes les données d'entraînement est effectué, on dit que l'on a réalisé une « itération », ou « *epoch* ». Ainsi, on va entraîner le réseau sur un certain nombre d'*epochs*, jusqu'à ce que l'erreur converge vers une erreur minimum. Il faut cependant faire attention à ne pas tomber dans le surentraînement (*overfitting*). C'est-à-dire que, bien que l'on atteigne une erreur très faible sur les données d'entraînement, lors du passage sur les données de test (jamais vues par le réseau pendant l'entraînement), les résultats soient bien moindres, voire divergent. Cela arrive généralement lorsque le nombre d'itérations est trop élevé. À l'inverse, il ne faut pas non plus stopper l'apprentissage trop tôt, sinon le modèle n'a pas le temps de converger vers une erreur minimum et le modèle est donc sous-entraîné. La figure 1.5 illustre un modèle dont l'erreur de validation diminue jusqu'à un minimum avant de tomber dans le surentraînement (la section 1.2.1 donne plus de détail sur la méthode de développement du *Deep Learning*).

Pour accélérer l'entraînement, on utilise la méthode des lots (*batch*). En effet, une itération de descente du gradient consiste à entraîner et mettre à jour les paramètres du réseau par rapport à chaque donnée. La méthode des lots consiste à accélérer une itération en propageant l'ensemble du jeu de données d'un coup puis en mettant à jour les paramètres en une seule fois. Pour ce faire, le jeu de données est regroupé en un seul tenseur (tableau multidimensionnel) ; par exemple, pour un jeu de 1000 images en couleurs (3 canaux) de résolution 32×32, on regroupera ces images dans un seul tenseur de dimension $(32, 32, 3, 1000)$ au

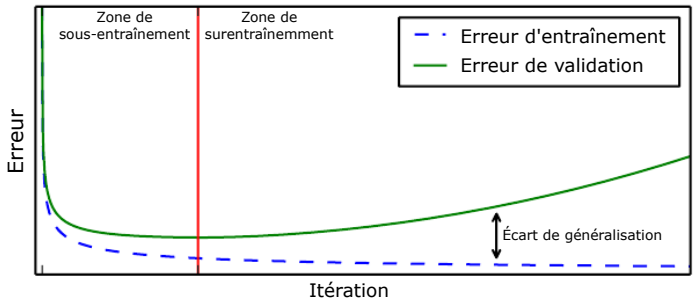


Figure 1.5 – Exemple de modèle dont l'erreur diminue au cours des itérations d'entraînement (zone de sous-apprentissage), puis l'erreur d'entraînement continue de diminuer tandis que l'erreur de validation commence à augmenter (zone de surentraînement). Adapté de [34].

lieu d'avoir 1000 tenseurs de dimension $(32, 32, 3)$. Cependant cette méthode n'est pas viable pour la plupart des jeux de données, car le tenseur (et sa propagation) devient trop lourd à calculer. La méthode utilisée est donc celle des « mini-lots » (*mini-batch*). Cette méthode consiste à regrouper non pas l'entièreté du jeu de données en un seul lot, mais en plusieurs petits lots. Le nombre de données qui sont regroupées en un mini-lot est appelé « taille du lot » (*batch size*). En reprenant l'exemple précédent, les tenseurs à propager deviennent de dimensions $(32, 32, 3, \textit{batch size})$. Il s'agit de la méthode la plus couramment utilisée, en utilisant le *batch size* le plus élevé possible en fonction de la configuration matérielle utilisée pour l'entraînement. Ce qui rend cette méthode efficace est l'utilisation de processeurs graphiques (*Graphics Processing Units*, GPUs), particulièrement performants pour le calcul matriciel. Ainsi, bien qu'un GPU ait une fréquence d'opération plus faible qu'un processeur, il peut effectuer des calculs en utilisant des tenseurs de grandes dimensions (contrairement au processeur qui calcule en série), ce qui in fine rend ce composant plus rapide que le processeur pour traiter l'ensemble des données.

1.1.3 . Réseau de neurones convolutifs

Les réseaux de neurones convolutifs (ou *Convolutional Neural Network*, CNN), présentés au départ par Yann LeCun et al. [48] avec LeNet, ont l'avantage qu'ils peuvent traiter des matrices. Ceux-ci sont donc tout désignés pour le traitement d'image, qui ne sont que des matrices de pixels, et sont dans les faits plus performants [1] qu'un simple perceptron multi-

1. En DL, lorsqu'on parle de « performance », on parle généralement des indicateurs de performance (*metrics*) sur le jeu de données de test. Ces indicateurs sont, par exemple, l'exactitude (*accuracy*), la précision (*precision*) et le score $F1$ (*F1-score*) pour la classification, et l'indice de Jaccard/intersection sur l'union (*IoU*) pour la segmentation. Plus de détails sur les critères d'éva-

couche pour le traitement d'image (voir section 1.1.4). Ainsi un CNN est constitué d'une succession de couches de convolutions (partie 1.1.3). On peut aussi y mêler des couches de sous-échantillonnage (*pooling*, partie 1.1.3) pour finir sur une ou plusieurs couches de perceptrons (partie 1.1.2). Dans ce cas, on appelle ces couches de perceptrons des couches complètement connectées (*Fully Connected layer*, FC).

Couche de convolution

Une couche de convolution, abrégée en conv, prend un tenseur en entrée $\mathbf{X}$ de dimensions $(h_{\mathbf{X}}, w_{\mathbf{X}}, c_{\mathbf{X}})$ et produit un tenseur en sortie $\mathbf{Y}$ de dimensions $(h_{\mathbf{Y}}, w_{\mathbf{Y}}, c_{\mathbf{Y}})$. Cette sortie, appelée « carte de fonction » (*feature map* ou *activation map*), est le résultat de la convolution appliquée entre le tenseur d'entrée $\mathbf{X}$ et le filtre $\mathbf{F}$ de dimensions $(h_{\mathbf{F}}, w_{\mathbf{F}}, c_{\mathbf{X}}, c_{\mathbf{Y}})$ où $c_{\mathbf{X}}$ est le nombre de canaux d'entrée qui définit le nombre de canaux de filtre, et $c_{\mathbf{Y}}$ est le nombre de filtres qui détermine le nombre de canaux du tenseur de sortie $\mathbf{Y}$. Par conséquent, nous pouvons calculer le nombre de paramètres à entraîner pour chaque couche de convolution en utilisant l'équation suivante :

$$\mathcal{P}\left(conv\right) = h_{\mathbf{F}} \cdot w_{\mathbf{F}} \cdot c_{\mathbf{X}} \cdot c_{\mathbf{Y}} + \mathfrak{b} \cdot c_{\mathbf{Y}} \tag{1.19}$$

où $\mathfrak{b}$ désigne l'utilisation ou non de biais dans la couche de convolution appliquée pour chaque filtre, donc $\mathfrak{b} \in \{0, 1\}$. Nous pouvons observer dans l'équation 1.19 que le nombre de paramètres ne dépend pas de la taille (ou plutôt de la hauteur et de la largeur) des tenseurs d'entrée et de sortie, contrairement à une couche FC. Cette dernière nécessite un grand nombre de paramètres pour modéliser des relations complexes entre les entrées et les sorties. En conséquence, le nombre de paramètres d'un modèle basé sur des couches de convolution est généralement plus faible comparé à un modèle MLP offrant des performances similaires. De plus, la convolution permet de retenir des informations spatiales dans l'image, en extrayant ainsi des caractéristiques, comme des formes ou des bords (on nomme ce processus *feature extraction*).

Les paramètres d'une couche de convolution sont les suivants :

1. La taille du noyau (*kernel*, $(h_{\mathbf{F}}, w_{\mathbf{F}})$) : la dimension du filtre qui sera appliqué sur l'image.
2. Le nombre de filtres (*filters*) : le nombre de noyaux qui sera appliqué, définissant ainsi le nombre cartes de fonctions en sortie de la couche de convolution.
3. Le pas (*stride*, (s_h, s_w) ou s si $s_h = s_w$) : le nombre de pixels dont le filtre va se décaler à chaque calcul sur les deux axes de l'image. s_h pour décaler sur la hauteur de l'image, et s_w sur la largeur de l'image. Il est à remarquer qu'un pas supérieur à 1 réduit la dimension des données (par exemple, un pas de 2 réduit la dimension de moitié), il est donc possible d'utiliser une convolution avec un pas supérieur d'une façon similaire à une couche de *pooling* (partie 1.1.3).

luation sont donnés dans la section 1.2.1

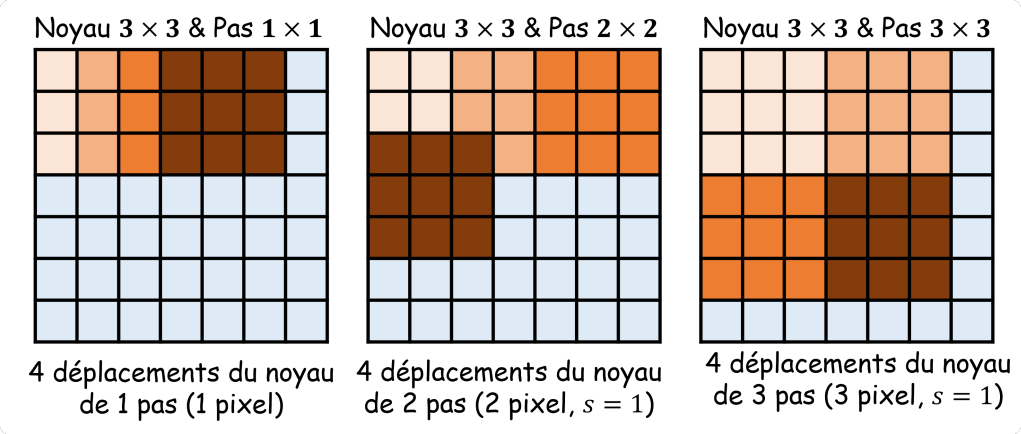


Figure 1.6 – exemple de déplacement du filtre sur une image en utilisant le même pas 1, 2 et 3 dans les deux axes $(s_h = s_w)$.

4. La dilatation (*dilatation rate*, (d_h, d_w) ou d si $d_h = d_w$) : une dilatation non nulle implique que le filtre est éclaté, autrement dit que les pixels traités à chaque pas ne sont pas adjacents, on laisse un espace égal au taux de dilatation entre chaque point du filtre. Une couche de convolution avec une dilatation est aussi appelée *atrous convolution*.

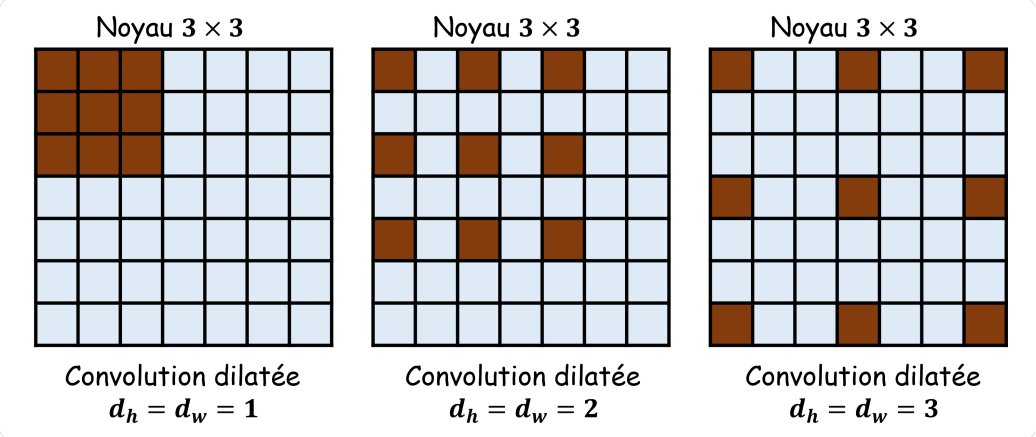


Figure 1.7 – exemple de dilatation du filtre en utilisant la même dilatation 1, 2 et 3 dans les deux axes $(d_h = d_w)$.

5. La marge (*padding*, $\mathfrak{p}$) : il s'agit d'un enrobage qui peut être appliqué autour de l'image afin de conserver les dimensions de l'image avant et après la couche. Dans les faits, il s'agit d'ajouter des 0 autour de la matrice, d'où le nom de *zero-padding*. La marge se calcule automatiquement via les paramètres de dilatation et la taille du filtre. Par conséquent, la marge est appliquée ou non avec $\mathfrak{p} \in \{0, 1\}$.

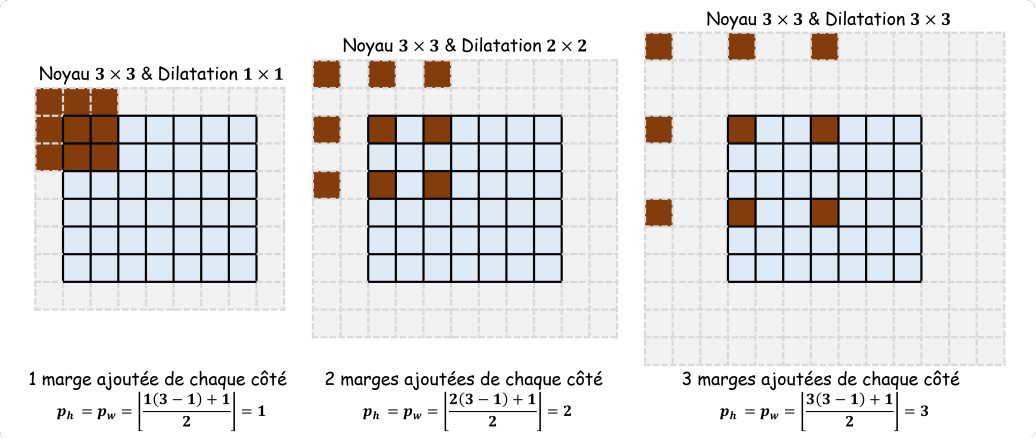


Figure 1.8 – exemple de marge ajouté sur l'image pour gardé la même taille de l'image d'entrée à la sortie.

En fonction des paramètres de la couche de convolution et de la taille de la matrice d'entrée $(h_{\mathbf{X}}, w_{\mathbf{X}})$, la taille de la matrice de sortie $(h_{\mathbf{Y}}, w_{\mathbf{Y}})$ est donnée par les équations suivantes :

$$h_{\mathbf{Y}} = \left\lfloor \frac{h_{\mathbf{X}} - (d_h(h_{\mathbf{F}} - 1) + 1) + 2\mathfrak{p}p_h}{s_h} \right\rfloor + 1 \tag{1.20}$$

$$w_Y = \left\lfloor \frac{w_X - (d_w(w_F - 1) + 1) + 2\mathfrak{p}p_w}{s_w} \right\rfloor + 1 \tag{1.21}$$

avec $p_h = \left\lfloor \frac{d_h(h_F-1)+1}{2} \right\rfloor$ et $p_w = \left\lfloor \frac{d_w(w_F-1)+1}{2} \right\rfloor$. Les pas s_h et s_w ne peuvent pas être plus de 1 si une de deux dilatations d_h ou d_w est supérieure à 1.

Pour chaque opération de convolution, nous calculons l'élément du tenseur de sortie $\mathbf{Y}$ en utilisant l'équation suivante :

$$\forall h, w, l : \mathbf{Y}_{h,w,l} = b_l + \sum_{k=0}^{c_X-1} \sum_{j=0}^{w_F-1} \sum_{i=0}^{h_F-1} \mathbf{X}_{x,y,k} \times \mathbf{F}_{i,j,k,l} \tag{1.22}$$

avec $x = s_x \cdot h + d_x \cdot i$ et $y = s_y \cdot w + d_y \cdot j$ où $x < h_X$ et $y < w_X$.

Pour expliquer plus en détails le fonctionnement d'une couche de convolution, prenons l'exemple illustré dans la figure 1.9, où une convolution 2D est effectuée sur une matrice $\mathbf{X}$ de dimension 5×5 à l'aide d'un seul filtre $\mathbf{F}$ avec un noyau de taille 3×3. Le filtre $\mathbf{F}$ est appliqué sur l'intégralité de la matrice $\mathbf{X}$ en avançant d'un pas $s = 1$, c'est-à-dire que le filtre ne se décale que d'un pixel après chaque application sur la matrice. À chaque application du filtre, le produit matriciel de Hadamard (produit terme à terme $\odot$) est effectué entre le filtre $\mathbf{F}$ et la partie de la matrice $\mathbf{X}$ que le filtre recouvre, puis les éléments de la matrice résultante de ce produit sont sommés, et le biais est ajouté pour obtenir une seule sortie. En déplaçant

le filtre sur l'image d'entrée, on aura la matrice de sortie de la couche de convolution $\mathbf{Y}$ de dimension 3×3.

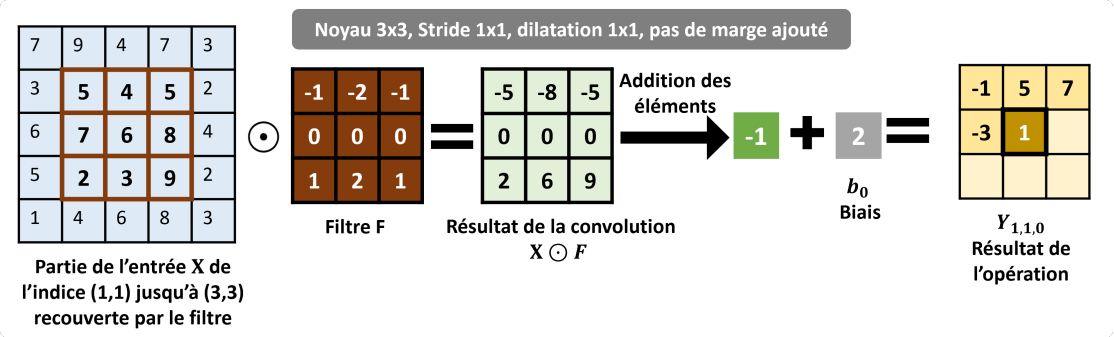


Figure 1.9 – Convolution 2D avec un noyau de 3x3 appliquée sur une matrice de dimension 5×5

L'équation 1.22 présente une seule opération de convolution pour déterminer un seul élément du tenseur de sortie $\mathbf{Y}$. Chaque opération de convolution effectue $(h_F \cdot w_F \cdot c_X)$ multiplications et $((h_F \cdot w_F \cdot c_X) + 1)$ additions. Donc, le total des opérations pour une seule convolution est :

$$\forall h, w, l : FLOP(\mathbf{Y}_{h,w,l}) = 2 \cdot (h_F \cdot w_F \cdot c_X) + 1 \tag{1.23}$$

Pour calculer le nombre total de FLOPs pour la couche de convolution qui produit la sortie $\mathbf{Y}$, nous devons multiplier le nombre de FLOPs, obtenu par l'équation 1.23, par le nombre total de convolutions effectuées $h_Y \cdot w_Y \cdot c_Y$. Cela donne :

$$FLOP(\mathbf{Y}) = (2 \cdot (h_F \cdot w_F \cdot c_X) + 1) \cdot h_Y \cdot w_Y \cdot c_Y \tag{1.24}$$

Couche de sous-échantillonnage (*pooling*)

Les couches de sous-échantillonnage (ou *pooling*) permettent de réduire les dimensions de la matrice, réduisant par la même le besoin en puissance de calcul. Une autre utilité du *pooling* est de faire une agrégation des caractéristiques extraites par les couches de convolution, rendant le modèle plus robuste (meilleure généralisation et moins prompt au surapprentissage). Comme une couche de convolution, une couche de *pooling* utilise un filtre qui passe sur toute la matrice d'entrée. Cependant, l'opération effectuée par le filtre est différente et ne possède pas de poids. Les caractéristiques d'une couche de sous-échantillonage sont les suivantes :

1. Le type de sous-échantillonage :
   — Pour le sous-échantillonnage par valeur moyenne (*average-pooling*), les valeurs présentes dans le filtre sont moyennées.

— Pour le sous-échantillonnage par valeur maximum (*max-pooling*), la sortie est égale à l'élément le plus grand du filtre.

2. La taille d'échantillonage (*pooling size*, $(h_{\mathbf{F}}, w_{\mathbf{F}})$) : comme le noyau d'une couche de convolution, il s'agit de la taille de la fenêtre sur laquelle sera fait le sous-échantillonage.
3. Le pas (*stride*, , (s_h, s_w) ou s si $s_h = s_w$) : le nombre de pixels dont la fenêtre va se décaler (similaire au pas des couches de convolution). Généralement, $(s_h, s_w) = (h_{\mathbf{F}}, w_{\mathbf{F}})$, ce qui fait qu'à chaque pas les fenêtres ne se superposent pas.

1.1.4 . Évolution des architectures de CNN

Lorsqu'on parle d'architecture pour un CNN, il s'agit de décrire combien de couches sont présentes dans le réseau, et, pour chaque couche, de décrire son type (convolution, *pooling*, ou FC), ses paramètres (taille de noyau, nombre de filtres, pas, marge et dilatation pour les couches de convolutions et de *pooling* ; et nombre de neurones pour une FC), et sa fonction d'activation. La section suivante présente une revue chronologique d'architectures notables.

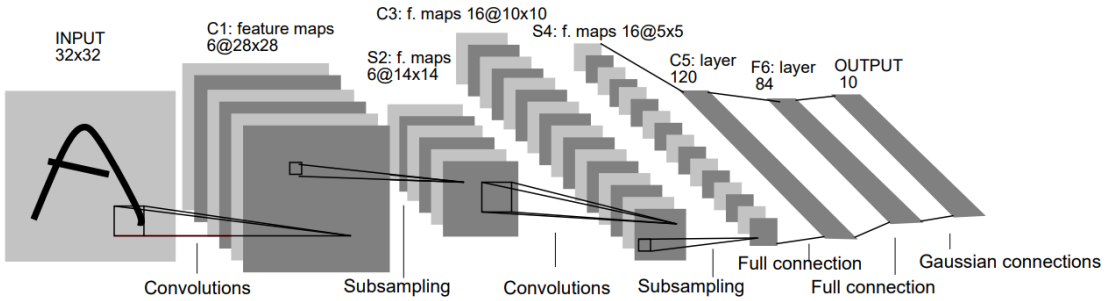
L'architecture d'un réseau neuronal (avec la qualité du jeu de données et les hyperparamètres), est un des points déterminant de sa performance[1]. C'est aussi principalement elle qui va déterminer la vitesse d'exécution et le poids en mémoire, autrement dit le coût en puissance informatique (*computational cost*). Dans cette section, nous décrirons donc chronologiquement des réseaux neuronaux qui ont marqué l'histoire de l'apprentissage automatique par les innovations dont elles sont à l'origine (ou ont fortement popularisé). Ceci permet d'appréhender les choix de structure du chapitre 2.

LeNet-5 [48]

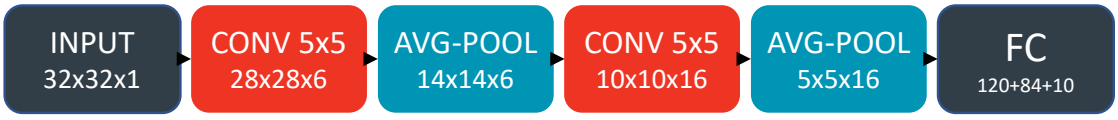
En 1998, Lecun et al. présente LeNet-5 [48], un CNN développé spécifiquement pour la classification d'images de chiffres manuscrits (cette publication présente aussi le jeu de données MNIST utilisé pour l'entraînement et validation de ce modèle). Il s'agit d'une architecture simple, qui a posé les bases des CNN : alterner couches de convolutions avec une fonction d'activation et des couches de *pooling*.

L'architecture de LeNet-5 est la suivante (figure 1.10) :

1. L'architecture prend en entrée (*input*) une image de 32×32 pixels et un seul canal (noir et blanc).
2. Une couche de convolution avec un noyau de 5×5 et 6 filtres avec comme fonction d'activation une tangente hyperbolique ($\tanh$) suivie d'une couche de sous-échantillonnage par valeur moyenne (*average pooling*), divisant les dimensions par deux.
3. Une couche de convolution avec un noyau de 5×5 et 16 filtres avec comme fonction d'activation $\tanh$ suivi d'une couche d'*average pooling*, divisant les dimensions par deux.
4. Trois FC de 120, 84 et 10 neurones avec respectivement une fonction d'activation $\tanh$, $\tanh$ et $\mathrm{sigmoid}$.



(a) LeNet-5 tel que présenté par LeCun et al. Extrait de [48]



(b) Représentation de LeNet-5 [48] sous forme de blocs.

Figure 1.10 – LeNet-5 [48]

Cette première architecture de CNN a atteint une exactitude de 98% sur MNIST et a posé les bases des CNN, bien qu'à sa publication, ce modèle n'eût pas un grand succès du fait des limitations matérielles de son temps (notamment l'accès à des GPUs).

AlexNet [46]

Vint ensuite AlexNet [46] (sur le dataset ImageNet [21]) dont l'architecture est la suivante (figure 1.11) :

1. En entrée, une image de dimensions $227 \times 227 \times 3$ (avec trois canaux donc en RGB).
2. Une couche de convolution avec un noyau de 11×11, 96 filtres, un pas de 4 et avec comme fonction d'activation une Unité Linéaire Rectifiée (*Rectified Linear Unit*, $\mathbf{ReLU}$). Le pas de 4 fait passer les dimensions de 227×227 à 55×55 (voir équation 1.20). Ensuite, une couche de sous-échantillonnage par valeur maximum (*max-pooling*) divise encore les dimensions par deux.
3. Une couche de convolution avec un noyau de 5×5, 256 filtres et une activation $\mathbf{ReLU}$, suivie d'une couche de sous-échantillonnage par valeur maximum (*max-pooling*) divisant les dimensions par deux.
4. Trois couches de convolution avec un noyau de 3×3, respectivement 384, 384 et 265 filtres et une activation $\mathbf{ReLU}$.
5. Une couche de sous-échantillonnage par valeur maximum (*max-pooling*) divisant les dimensions par deux.
6. Trois FC de 2048, 2048 et 1000 neurones avec respectivement une fontion d'activation ReLU, ReLU et $\mathrm{softmax}$.

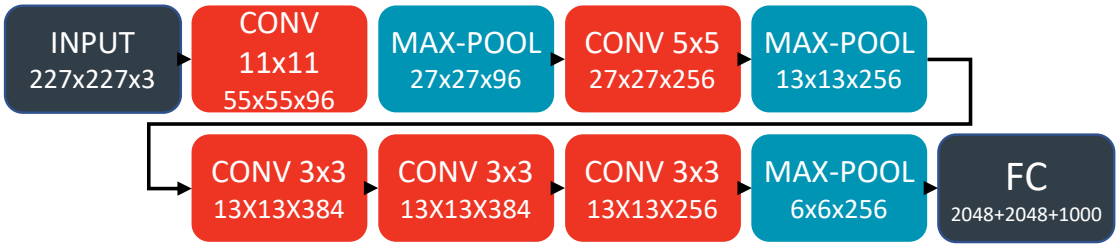


Figure 1.11 – AlexNet [46]

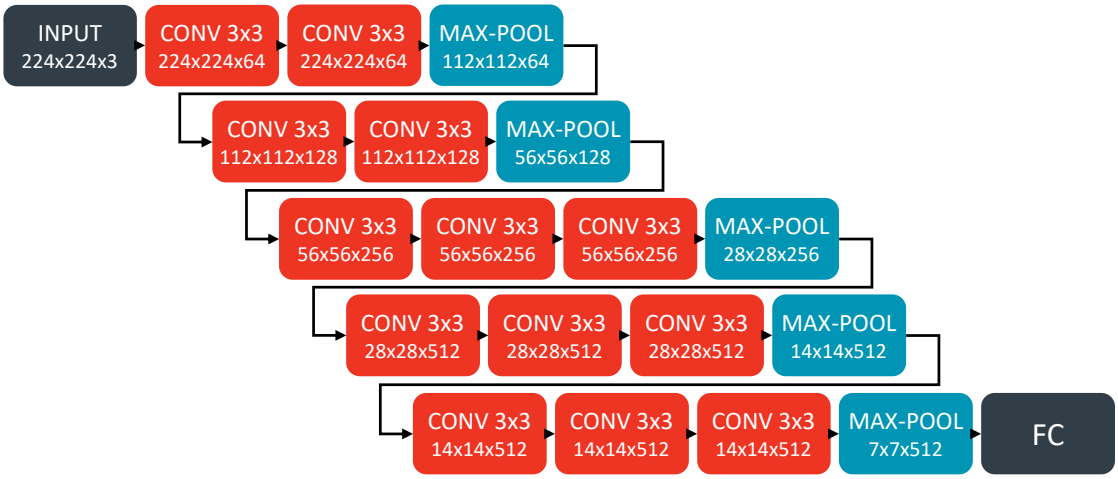
Les points intéressants de cette architecture sont tout d'abord l'implémentation de la fonction d'activation $\mathbf{ReLU}$ qui est largement utilisée de nos jours dans les couches cachées des CNN modernes. Mais aussi l'utilisation de dropout [79] comme régularisation dans les FC pour limiter le surentraînement (*overfitting*), bien que cela augmente le nombre de calculs nécessaires. AlexNet a atteint une exactitude de 84.7% sur ImageNet (qui est un jeu de données plus complexe que MNIST, il est donc plus difficile d'obtenir une bonne exactitude).

VGG-16 [76]

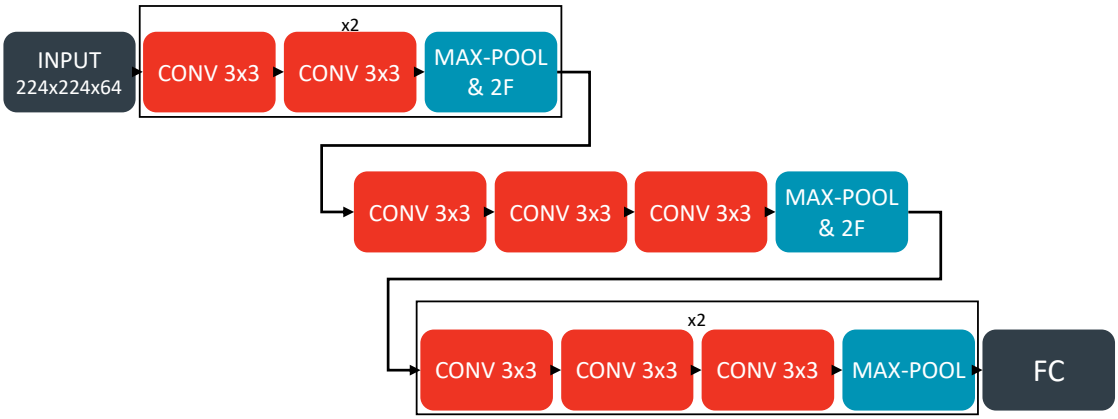
Une architecture notable suivante est VGG-16 [76] avec 13 couches de convolutions et 3 FC (ainsi que 5 couches de *max-pooling*). L'architecture détaillée est décrite par la figure 1.12a. Un point notable est que l'architecture se répète : d'abord un bloc constitué de deux couches de convolutions 3×3 suivi de *max-pooling* se répète deux fois (il s'agit des premières lignes de la figure), puis un bloc constitué de trois couches de convolutions 3×3 suivi de *max-pooling* se répète trois fois (les trois lignes suivantes), et enfin on arrive sur les FC.

Bien que les types d'opérations se répètent, ce n'est pas tout à fait le cas du nombre de filtres. On peut cependant y trouver une logique simple qui est que lorsqu'une couche de maxpooling est effectuée, le nombre de filtres pour la convolution suivante est multiplié par deux. D'où la figure 1.12b, où l'on définit une opération de réduction "max-pool & 2F" qui va donc diviser les dimensions par deux mais multiplier le nombre de filtres par deux. On définit aussi que, bien que l'entrée réelle soit de dimensions $224 \times 224 \times 3$, on indique qu'elle est de $244 \times 224 \times 64$ pour signifier que la première convolution a 64 filtres. De ce fait, on a pu rendre la représentation de VGG-16 plus compacte en s'affranchissant d'indiquer le nombre de filtres sous chaque opération et en rassemblant les opérations en groupes se répétant.

VGG-16 a une architecture plus profonde que ces prédécesseurs, mais utilise de plus petits noyaux, ce qui permet de ne pas trop augmenter le coût en puissance de calcul. Ainsi, en atteignant une exactitude de 92.7% sur ImageNet, il montre qu'augmenter la profondeur d'un réseau (plus que la taille de ses noyaux) augmente ces performances.



(a) Représentation complète



(b) Représentation compacte

Figure 1.12 – Représentations de VGG-16

GoogLeNet [85]

GoogLeNet [85] est une architecture avec cinq millions de paramètres, inspirée de *Network In Network* [50]. Le tableau 1.1 récapitule l'architecture exacte de GoogLeNet. Il s'agit d'une répétition de groupes de couches de convolutions appelées « *inception modules* » (en référence au film *Inception* [61]). Ces *Inception modules* ont pour principe d'avoir des couches de convolutions en parallèles, ensuite fusionnées par une concaténation. L'idée est que chacun de ces canaux parallèles de convolutions extrait des caractéristiques différentes, car leurs noyaux sont différents (1×1, 3×3 et 5×5). Contrairement à VGG-16, il n'y a pas de relation logique au nombre de filtres au travers du réseau. Ainsi, si l'on veut compacter la représentation de GoogLeNet, il faut faire abstraction du nombre de filtres dans les couches de convolution.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1.1 – Tableau récapitulant l'architecture de GoogleNet. Extrait de [85]

Ainsi, la figure 1.13 illustre l'architecture de GoogLeNet de façon compacte en représentant le module *inception* à la façon d'une couche. Dans cette représentation, le nombre de filtres dans les couches de *pooling* est quand même précisé.

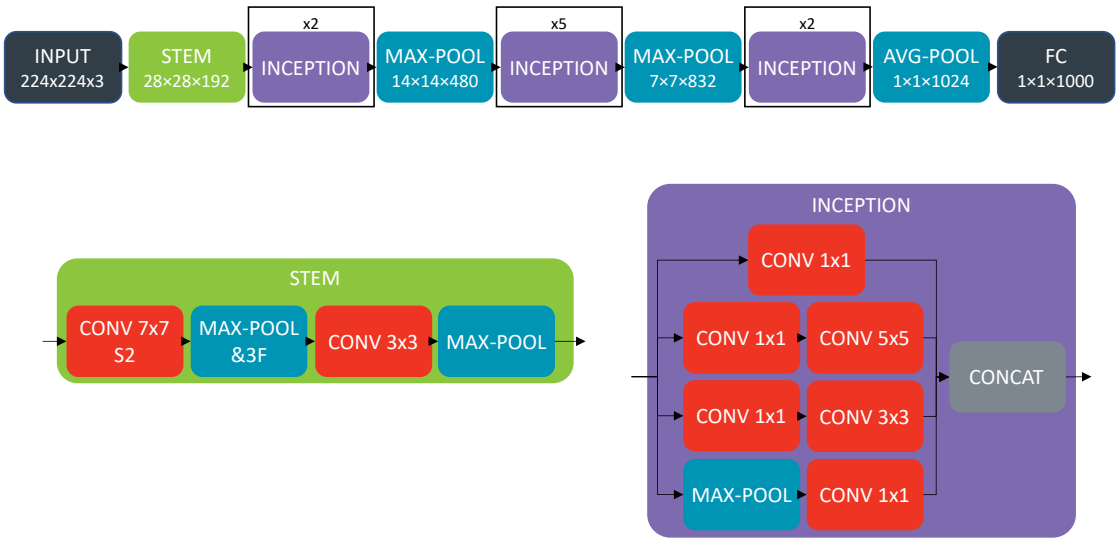


Figure 1.13 – représentation compacte de GoogLeNet.

Cette architecture est donc, en plus d'être plus profonde que VGG-16, est aussi plus « large », avec ces couches en parallèles. Grâce à cette idée, GoogLeNet atteint une exactitude de $93,33\%$ sur ImageNet.

ResNet [37]

Un réseau neuronal résiduel [37], ou « ResNet » pour *Residual Network*, a pour particularité de posséder des connexions dites « SKIP », ou « identité » (*residual*). Pour faire une connexion « SKIP », on ajoute à un tenseur un autre tenseur provenant d'une couche moins profonde dans le réseau. Dans [37], cela se traduit par les blocs « identité » (notés » RES » dans la figure 1.14). Il y a aussi des blocs « convolution » (notés « CONV » dans la figure 1.14), qui introduisent une couche de convolution avec un noyau 1×1 servant à effectuer une projection linéaire lorsque les dimensions du tenseur en entrée ne sont pas les mêmes que celui en sortie. Dans les faits, les blocs RES préservent les dimensions de l'entrée, tandis que les blocs CONV divisent les dimensions spatiales par deux et doublent le nombre de filtres.

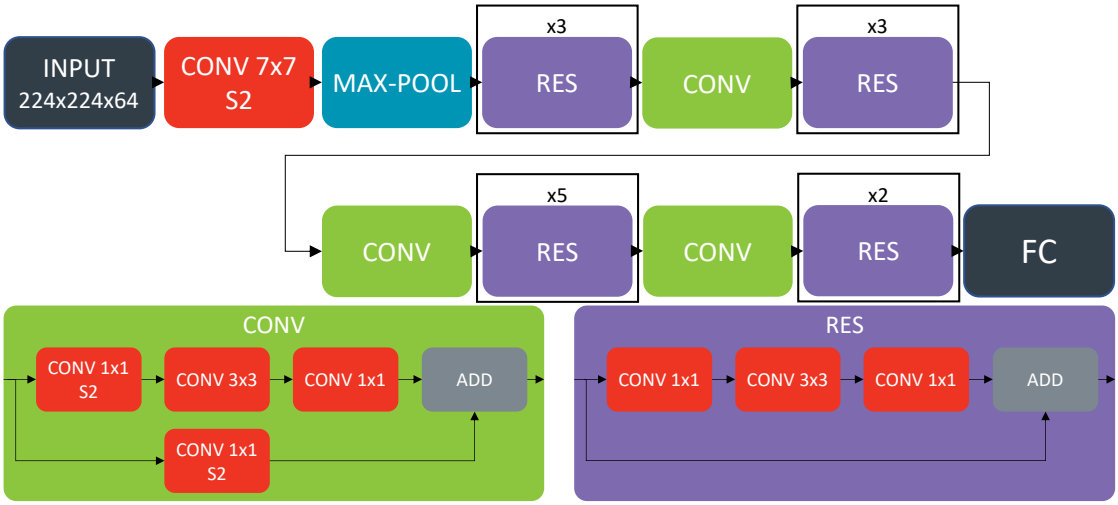


Figure 1.14 – Resnet-50 [37]

L'intérêt des connexions SKIP est de retenir les caractéristiques extraites précédemment. Cela permet de construire des réseaux plus profonds sans perdre en généralisation (réduction du phénomène d'*overfitting*). ResNet a une exactitude de $96,43\%$ sur ImageNet.

Densenet [41]

L'architecture DenseNet [41] incrémente sur l'idée du ResNet de la section précédente. Dans cette architecture, toutes les cartes de caractéristiques sont liées. En effet, dans un bloc « *dense* », toutes les cartes de caractéristiques précédentes sont concaténées avant les couches de convolutions (une CONV1×1 suivi d'une CONV3×3) comme illustré par le bloc en bas à droite de la figure 1.15. Cette concaténation n'est possible que dans les blocs *dense*, car les dimensions ne changent pas dans ces blocs. Les dimensions sont donc réduites uniquement entre les blocs *dense* grâce à un bloc *transition* (avec une CONV1×1 et un *average pooling*). Ce bloc est noté « TRANSI » dans la figure 1.15.

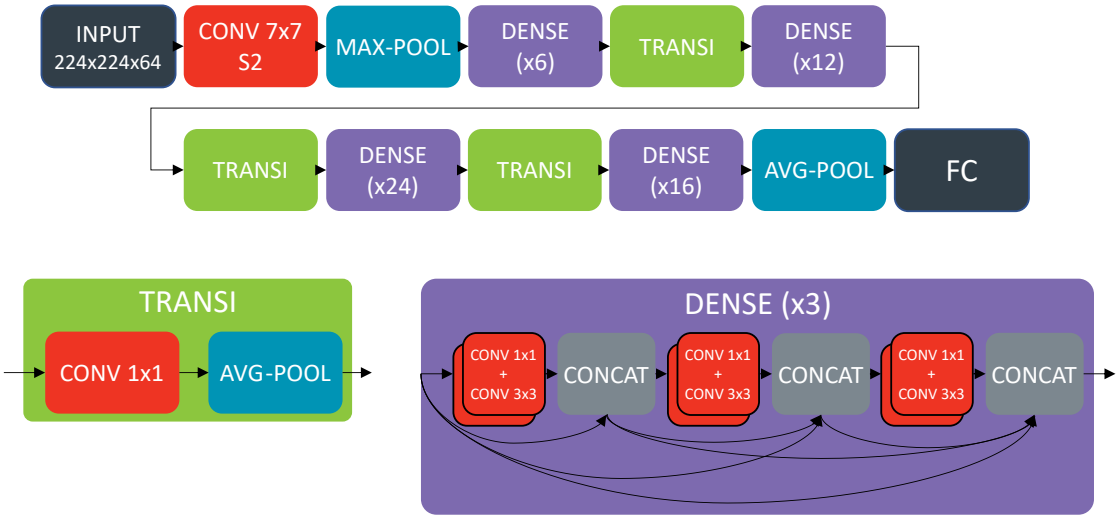


Figure 1.15 – DenseNet-121 [41]

Récapitulatif concernant les architectures historiques

Au fur et à mesure de l'histoire de l'apprentissage profond, les architectures sont devenues de plus en plus performantes, mais aussi de plus en plus complexes. Cependant, cette complexité ne se manifeste pas de la même façon dans toutes ces architectures. Par exemple, comme le montre le tableau 1.2, GoogleNet possède moins de paramètres d'entraînement et a besoin de moins de mémoire que AlexNet, mais il demande plus de puissance de calcul au GPU (FLOPs). Ceci est dû au fait que les tenseurs utilisés dans GoogleNet ont des dimensions plus réduites et les convolutions des noyaux plus petits, mais ce réseau est beaucoup plus profond que AlexNet.

Les améliorations peuvent aussi venir, plus que de la profondeur et des réglages de tailles de tenseurs et noyaux, de techniques innovantes comme le ResNet et le DenseNet qui récupèrent des tenseurs dans les couches moins profondes du réseau afin de mieux propager les caractéristiques. De telles innovations servent d'ailleurs désormais à créer des réseaux très légers mais toujours performants. C'est le cas pour SimpleNet [36] qui utilise de la régularisation à outrance pour entraîner un CNN simple (avec peu de paramètres, FLOPs et mémoire) longuement tout en évitant le surentraînement.

Ainsi, le choix de l'architecture d'un CNN est un élément crucial de son design, car elle détermine les ressources dont il aura besoin mais aussi influe sur ces performances. Dans la section 1.2 suivante, nous verrons la notion d'Auto-ML permettant, entre autres, à définir automatiquement une architecture de CNN.

Modèle	AlexNet [46]	GoogleNet [85]	ResNet152 [37]	VGG16 [76]	NIN [50]	SimpleNet [36]
Année	2012	2014	2015	2014	2014	2016
Paramètres	60 M	7 M	60 M	138 M	7,6 M	5,4 M
FLOPs	7,27 G	16,04 G	11,3 G	154,7 G	11,06 G	652 M
Mémoire	217 MB	40 MB	230 MB	512,24 MB	29 MB	20 MB
Exactitude	63,3 %	69,8 %	78,57 %	74,4 %	91,2 %*	72,03 %

*Sur CIFAR-10 [45]

Table 1.2 – Nombre de paramètres, de FLOPs, de taille mémoire et d'exactitude (sur ImageNet [21]) des modèles présentés. Adapté de [36]

1.2 . Auto-ML

Dans cette section est présenté le principe d'Apprentissage automatique automatisé / *Automated Machine Learning* (Auto-ML). Tout d'abord, nous verrons comment un modèle de *Deep Learning* est développé par un humain, puis comment l'Auto-ML tente d'automatiser ce processus. La partie suivante présente l'état de l'art des différents espaces de recherches et techniques d'optimisation pour la recherche automatique d'architectures.

1.2.1 . Méthodologie de développement de *Deep Learning*

Lorsque l'on développe un modèle de *Deep Learning*, la méthodologie est généralement la suivante [32] :

1. Préparation des données. Cela peut inclure un nettoyage (en supprimant les données inutiles, incomplètes, redondantes ou incorrectes) ou une augmentation (en créant artificiellement plus de données grâce à des déformations, rotation, rognage, etc.).
2. Définition d'une première architecture. Le choix de cette architecture se base sur l'intuition du développeur et ses connaissances en *Deep Learning* et sur le jeu de données. Cette première étape doit donc se conclure par le développement d'un modèle non optimisé mais fonctionnel sur le jeu de données. Par exemple, l'entrée du modèle doit pouvoir accueillir le jeu de données, le nombre de neurones de sortie doit correspondre au nombre de labels (pour un problème de classification), et le modèle ne doit pas être trop gourmand en mémoire (selon la configuration matérielle pour l'entraînement).
3. Définition des hyperparamètres d'entraînement. Ces hyperparamètres peuvent être le taux d'apprentissage, le nombre d'itérations, le *batch size*, ou encore des régularisations comme le *dropout* [79] et le *weight decay* [47].
4. Entraînement.
5. Analyse des résultats de l'entraînement (aussi appelé « validation ») :
   - Quelle est sa performance finale (erreur sur le jeu de données de validation) ?
   - Quel est le temps de propagation ?

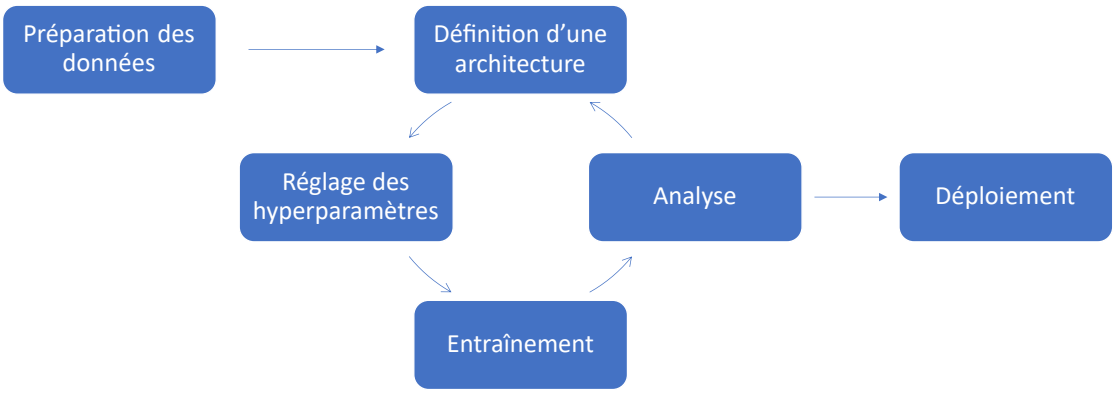


Figure 1.16 – Méthode de développement d'un modèle de DL, appelée *fine tuning*.

— Comment l'erreur a-t-elle convergé vers un minimum sur le jeu de données d'entraînement ? De validation ? (Figure 1.5)

6. À la suite de cette analyse, on ajuste l'architecture et/ou les hyperparamètres. C'est ce qu'on appelle le « *fine tuning* ».

7. On répète les étapes 4, 5 et 6 jusqu'à atteindre les objectifs. Ces objectifs sont généralement d'abord sur la performance du modèle (le modèle a atteint l'exactitude minimum souhaitée sur les données de validation), mais peuvent aussi être le temps d'inférence (le modèle met au plus un temps donné pour faire une prédiction), ou encore la consommation énergétique lors de l'exécution (le modèle consomme au plus une quantité d'énergie donnée lors de son exécution).

8. Déploiement du modèle.

Cette méthode requiert beaucoup de connaissances et d'expérience dans le développement de *Deep Learning*, notamment pour l'analyse des résultats après entraînement et pour affiner les paramétrages (*fine tune*) en fonction. Elle nécessite aussi beaucoup de temps, car la boucle d'optimisation doit être répétée de nombreuses fois et chaque entraînement est chronophage. Cependant, cette méthode étant itérative, il est possible de l'automatiser afin de pouvoir créer des modèles de DL (sans intervention humaine durant l'optimisation) et rendre accessible cette technologie à des personnes non spécialistes. On appelle Apprentissage automatique automatisé / *Automated Machine Learning* (Auto-ML) un algorithme tentant une telle automatisation (qu'il s'agisse de DL ou de *Machine Learning* traditionnel). Avant de continuer sur la notion d'Auto-ML, la sous-section suivante décrit les critères d'évaluation d'un CNN pertinents dans le cadre de cette thèse et pouvant servir d'objectifs d'optimisation pour les Auto-ML.

Critères d'évaluation d'un CNN

Lors de l'évaluation du modèle, plusieurs critères (*metrics*) sont à prendre en compte [84] :

1. Tout d'abord le critère de la performance [58] sur les données de validation. Cette performance peut se traduire par :
   - L'exactitude top-1 (*top-1 accuracy*). C'est à dire le pourcentage de données dont la prédiction du modèle correspond au label de la donnée :

$$exactitude = \frac{prédictions\_correcte}{nombre\_de\_préditions}$$

   - L'exactitude top-5. Il s'agit d'une exactitude plus laxiste où, pour que la prédiction soit considérée comme correcte, le label de la donnée doit correspondre à l'une des cinq premières prédictions (les cinq prédictions avec la probabilité la plus haute).
   - La précision. Ce critère concerne une classe en particulier et détermine le nombre de données pour lesquelles la classe a correctement été prédite (vrai positif) et pénalise les données ne faisant pas partie de cette classe mais qui ont quand même été prédites comme telles (faux positif) :

$$précision = \frac{vrai\_positif}{vrai\_positif + faux\_positif}$$

     .
   - Le rappel. Ce critère concerne aussi une classe en particulier et détermine le nombre de données pour lesquelles la classe a correctement été prédite (vrai positif) mais pénalise les données qui n'ont pas été prédites comme faisant partie de la classe bien qu'elles soient bien labélisées comme telles (faux négatif) :

$$rappel = \frac{vrai\_positif}{vrai\_positif + faux\_négatif}$$

   - Le score $F1$. Ce score combine la précision et le rappel en une moyenne harmonique :

$$F1 = 2 \times \frac{précision \times rappel}{précision + rappel}$$

2. Un autre critère à prendre en compte est le nombre de poids du modèle. En effet, ce critère peut refléter la taille de la mémoire que prend le modèle.
3. Le nombre d'opérations multiplieur-accumulateur (MACs) ou d'opérations FLOPs. Ce critère peut permettre d'évaluer la puissance de calcul nécessaire pour faire tourner le modèle.

1.2.2 . Qu'est-ce qu'un Auto-ML ?

Un Auto-ML a pour vocation d'automatiser au maximum le processus de développement d'un algorithme de *Machine Learning*, autrement appelé « modèle » de *Machine Learning*. En se limitant au *Deep Learning*, d'après la méthode décrite en section 1.2.1, on peut séparer les étapes de développement en trois parties : la préparation de données, la recherche d'architecture et l'optimisation des hyperparamètres. Ces méthodes sont décrites de manière concise par Xin He et al. dans [38], en plus du *Feature Engineering* propre au ML traditionnel.

Pour cette thèse, le choix a été fait de se concentrer sur la partie « recherche d'architecture ». En effet, cette partie est critique lors d'une approche multi-objectifs (notamment dans le cadre des systèmes embarqués) du développement de *Deep Learning*, car le choix de l'architecture a une grande influence sur des paramètres tels que le temps d'exécution ou la taille mémoire. Les deux autres points, préparation de données et optimisation des hyperparamètres, sont surtout déterminants du point de vue des performances du modèle (erreur).

La section suivante traite donc plus en détail de la recherche d'architecture automatique, ou *Neural Architecture Search* (NAS).

1.2.3 . Recherche automatique d'architecture

La Recherche d'architecture neuronale / *Neural Architecture Search* (NAS) est la discipline cherchant à automatiser la recherche d'une architecture de *Deep Learning* la plus performante par rapport à un jeu de données. Il s'agit donc de la partie « Définition d'une architecture » dans la figure 1.16. Si l'on fait de la NAS seule, le « Réglage des hyperparamètres » est fixe. Pour effectuer une NAS, il y a tout d'abord besoin d'un espace de recherche. Cet espace détermine quelles architectures sont explorables et comment elles sont construites. Ensuite, pour explorer cet espace, il faut utiliser une technique d'optimisation cherchant à converger vers l'architecture optimale.

Les deux sous-sections suivantes présentent différents espaces de recherche et techniques d'optimisations de l'état de l'art [38, 25]. Puis celles-ci seront analysées par rapport à la problématique de la thèse dans la section 1.2.4

Espaces de recherche

Structure en chaîne

Une façon assez directe et simple de définir un NN est de le représenter sous forme d'une liste dont chaque élément représente une couche du NN. C'est finalement ainsi que sont représentés les CNN dans les figures de la section 1.1.4. Ce type d'espace de recherche peut être trouvé sous l'appellation « structure en chaîne » (*chain-structured*) [25] ou « entièrement structuré » (*entire-structured*) [38].

La figure 1.17 montre un exemple simple (à gauche) d'un CNN représentable par une liste $L =$[CONV3×3, MAX-POOL, CONV3×3, CONV3×3, MAX-POOL]. Tandis que le CNN de droite est plus complexe à représenter sous forme de chaîne, car il comporte des *SKIP*. Dans ce

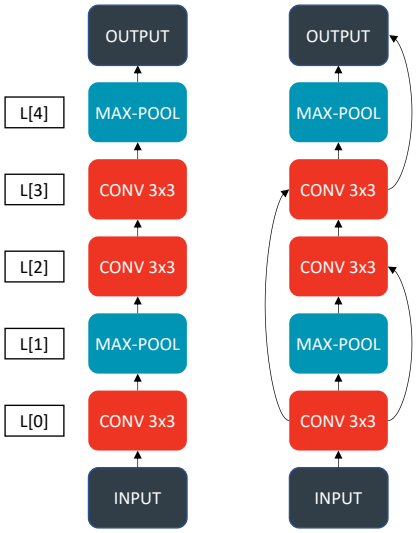


Figure 1.17 – Structure en chaîne
Deux réseaux de neurones présenté sous la forme d'une liste L.

cas, on peut par exemple noter les couches comme des fonctions dont les arguments sont les couches entrantes. Ce qui donnerait pour le réseau de droite la liste L =[CONV3×3(INPUT), MAX-POOL(L[0]), CONV3×3(L[1], L[0]), CONV3×3(L[2], L[0]), MAX-POOL(L[3])].

La structure en chaîne, bien que facile à implémenter, a le désavantage d'avoir un nombre de degrés de liberté augmentant rapidement avec la profondeur du réseau et l'implémentation de connexions parallèles (comme des *SKIP*). En effet, pour un réseau de N couches, avec O opérations (de convolutions, *pooling*, etc.), le nombre de réseaux possibles à représenter avec structure en chaîne, autrement dit la taille de l'espace de recherche, est de O^N. C'est pourquoi les espaces de recherche suivants cherchent à réduire le nombre de degrés de liberté, même sur des réseaux très profonds.

Espace basé sur des cellules

En partant de l'observation que la répétition de modules de sous-structure de réseaux neuronaux pour former un réseau plus grand performe bien ([85, 37, 41], respectivement présentés dans les sections 1.1.4, 1.1.4 et 1.1.4), Zoph et al. [95] et Zhong et al. [92] proposent l'idée de l'espace de recherche basé sur les cellules.

Le principe de l'espace de recherche basé sur des cellules est de rechercher le contenu d'une cellule (que l'on peut aussi trouver sous le nom de motif, ou bloc) qui sera répétée un certain nombre de fois pour créer le réseau, plutôt que de rechercher le réseau en entier. Ainsi, on peut augmenter la profondeur du réseau sans changer le nombre de degrés de liberté de l'espace de recherche.

La figure 1.18 présente un exemple d'espace de recherche basé sur les cellules, tel que

défini par Zoph et al. [95]. Le réseau est constitué d'une succession de cellules normales (que l'on peut aussi appeler « cellules de convolution »), et de cellules de réduction. La partie gauche de la figure montre l'intérieur d'une cellule. Une cellule contient des blocs, ici deux. À l'intérieur de chaque bloc se trouvent des couches de convolutions, des connexions SKIP dont les résultats sont ensuite ajoutés pour former la sortie du bloc. S'il s'agit d'une cellule de réduction, il peut aussi y avoir des couches de *pooling* pour effectuer la réduction de dimensions. Enfin, les sorties des blocs sont concaténées pour former la sortie de la cellule.

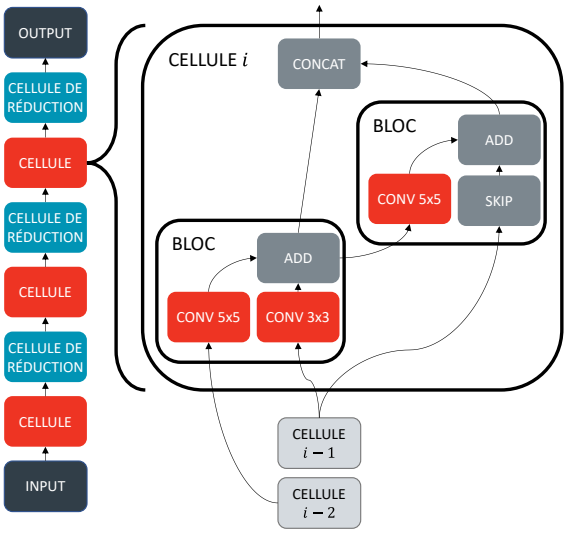


Figure 1.18 – Espace de recherche basé sur les cellules
Exemple d'espace de recherche basé sur les cellules.

Lorsque l'on utilise cet espace de recherche, l'optimisation automatique se fait sur la micro-architecture (la cellule), il y a ensuite toujours besoin d'optimiser manuellement la macro-architecture : le nombre de cellules et comment elles sont connectées. Pour cela, Cai et al. [11] utilisent la micro-architecture trouvée par leur NAS dans une macro-architecture connue, telle que *DenseNet* [41].

Pour accélérer la recherche d'architecture et la rendre moins gourmande en mémoire, une solution est d'entraîner le modèle avec un nombre moindre de cellules et ainsi passer moins de temps par entraînement, qui est la partie la plus longue de la boucle d'optimisation. Puis le modèle est validé avec un plus grand nombre de cellules, augmentant théoriquement ses performances. C'est la méthode utilisée par DARTS [53], qui entraîne ses modèles avec 8 cellules puis effectue sa phase de validation avec 20 cellules.

Cependant, une bonne cellule pour un modèle peu profond ne sera pas forcément idéale pour un modèle plus profond. Dans l'idéal, il faudrait donc pouvoir optimiser la micro-architecture et la macro-architecture de concert. Une première idée, utilisée par P-DARTS [14], consiste à augmenter graduellement le nombre de cellules durant la recherche et, pour

contrôler le surplus en coût de calcul, réduire le nombre d'opérations (convolutions, *pooling*, etc.) candidates. Une autre tentative est l'espace de recherche hiérarchique présenté dans la section suivante.

Espace de recherche hiérarchique

Un espace de recherche hiérarchique [51, 52, 57, 86, 90] créé en plusieurs étapes, un réseau à plusieurs niveaux hiérarchiques. En réalité, beaucoup d'espaces de recherches basés sur des cellules [95, 62, 63, 6, 91, 69, 68], tels que celui présenté dans la figure 1.18, sont des espaces hiérarchiques avec deux niveaux de hiérarchie. En effet, la recherche du contenu des cellules (micro-architecture) constitue le premier niveau, tandis que le second est la macro-architecture. Cependant, l'optimisation de la macro-architecture n'est pas automatisée.

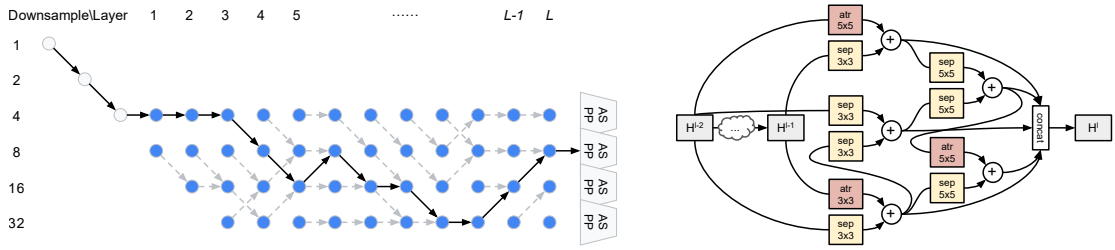


Figure 1.19 – Architecture trouvée par Auto-DeepLab pour Cityscapes [18]. À gauche les changements de dimensions spatiales le long du modèle, avec le chemin de *downsampling* retenu (flèches noires); et à droite la cellule trouvée par la recherche. Extrait de [51].

Ainsi, Liu et al. proposent *Auto-DeepLab* [51] un espace de recherche hiérarchique pour la segmentation d'image (notamment sur le jeu de données *Cityscapes* [18]). Leur espace de recherche est séparé en deux : un niveau cellule (*Cell Level Search Space*) et un niveau réseau (*Network Level Search Space*). Au niveau de la recherche de cellule, la méthode est similaire à Zoph et al. [95]. En revanche, pour le niveau réseau, ils mettent en place une recherche automatique du contrôle des dimensions spatiales le long du réseau. Ainsi, comme illustré sur la gauche de la figure 1.19, le long des L couches (*Layers*) du réseau, le modèle peut descendre ou monter d'un niveau de « sous-échantillonnage » (*downsample*). C'est-à-dire que passer d'un niveau de *downsample* de 2 à un niveau 4 double le nombre de filtres des couches de convolution et divise par deux les dimensions des tenseurs (hauteur et largeur). De ce fait, chaque nœud bleu de la figure 1.19 correspond à une cellule (telle que présentée dans la partie droite de la figure) et les flèches noires représentent le chemin de *downsampling* retenu par la recherche automatique : lorsque la flèche descend, les dimensions sont réduites ; lorsque la flèche monte, les dimensions augmentent ; et lorsque la flèche est horizontale, les dimensions sont inchangées. Cette méthode de recherche hiérarchique est particulièrement adaptée aux jeux de données de segmentation, étant donné que les dimensions ne doivent pas être trop faibles pour recréer une image ; contrairement à la classification où les dimensions

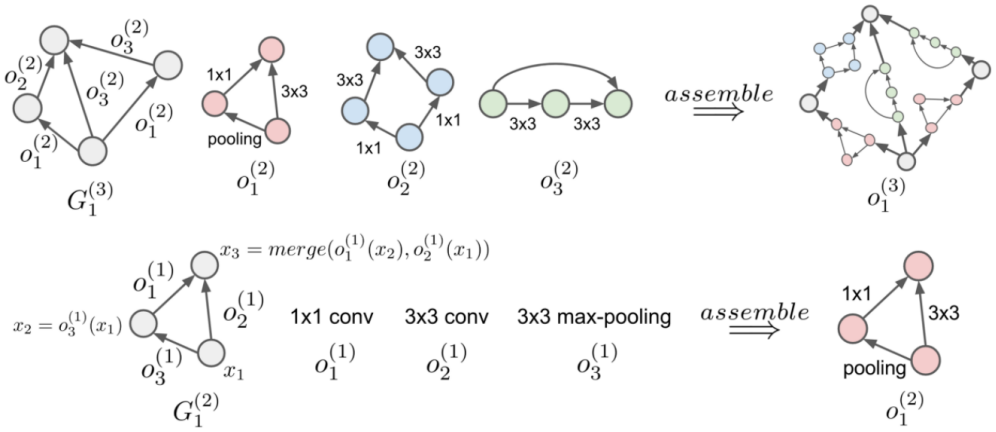


Figure 1.20 – Espace de recherche hiérarchique de *HierNAS*. La ligne du bas présente le premier niveau de hiérarchie assemblé pour former le deuxième, et la ligne du haut le deuxième niveau de hiérarchie assemblé pour former le troisième. Extrait de [52].

peuvent être réduites au maximum (donc où n'avoir que des cellules de réduction n'est pas un problème).

Cependant, le nombre de blocs à l'intérieur d'une cellule est toujours défini manuellement. Pour automatiser encore plus, Liu et al. [52] proposent, avec *HierNAS*, une méthode avec plus de niveaux de hiérarchies (figure 1.20). Le premier niveau de hiérarchie consiste en couches simples (convolutions et *pooling*) qui sont reliées pour former les structures du second niveau de hiérarchie. Ensuite, ces structures sont de nouveau reliées afin de former de nouvelles structures au niveau trois de hiérarchie. Ce dernier niveau de hiérarchie correspond au réseau neuronal entier.

Morphisme

Le morphisme (*morphism*) est un type d'espace de recherche qui modifie un modèle de DL préexistant pour tenter d'améliorer ses performances. En 2016, Chen et al. [13] proposent *Net2Net*, un espace de recherche par morphisme qui prend un CNN initial et l'approfondit ou l'élargit pour améliorer ses performances, voir figure 1.21.

Cette technique s'inspire de la notion d'apprentissage par transfert (*transfer learning*) [89], qui consiste à prendre un modèle de CNN déjà existant et entraîné sur un jeu de données similaire et à le réentraîner. Généralement, cela se fait en ne mettant à jour que les poids du FC en fin de modèle et en ne mettant pas à jour les poids de la partie extraction de caractéristiques, ainsi le modèle garde son ancienne capacité d'extraction de caractéristiques.

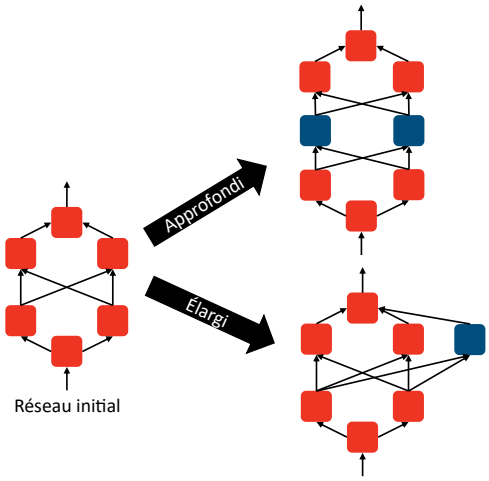


Figure 1.21 – Exemple de morphisme utilisé par Net2Net [13]. Les neurones bleus sont ajoutés au réseau de neurones rouges.

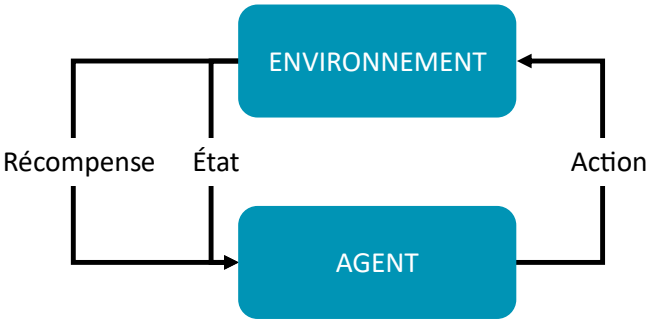
Techniques d'optimisation

Les techniques d'optimisation présentées dans cette section permettent de naviguer dans les espaces de recherche de la section précédente, afin de tenter de trouver l'architecture la plus optimale.

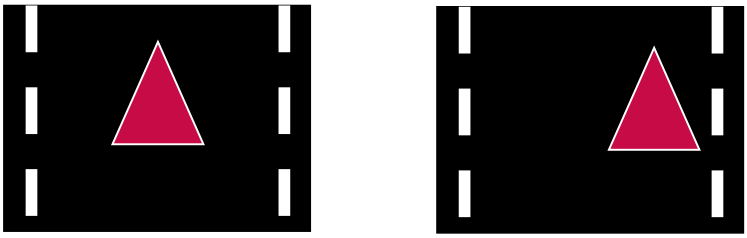
Apprentissage par renforcement

L'apprentissage par renforcement (ou *Reinforcement Learning*, RL) [83] est une technique de machine learning qui consiste à entraîner un « agent » pouvant observer et influencer son environnement. Comme illustré dans la figure 1.22a, l'agent peut effectuer des actions sur l'environnement ; en conséquence, l'environnement renvoie un état à l'agent. Cet état peut être, par exemple dans le cas du contrôle d'un véhicule, la position de l'agent sur la route, sa vitesse, son parallélisme aux lignes de signalisations. En fonction de l'état de l'agent dans l'environnement, une récompense va lui être attribuée si l'état correspond à celui attendu. En reprenant l'exemple du contrôle d'un véhicule, si on souhaite que ce dernier reste au centre d'une voie de circulation, on lui accordera une grande récompense s'il se trouve au milieu de la voie (partie gauche de la figure 1.22b), tandis que la récompense sera de plus en plus faible s'il s'approche du bord de la voie (partie droite de la figure 1.22b).

Zoph et al. [94] proposent en 2017 un NAS se servant d'un réseau neuronal récurrent (*Recurent Neural Network*, RNN) [59, 74] comme agent pour le RL. Dans ce cas, l'action de l'agent est la génération d'un CNN et sa récompense est l'exactitude du modèle (l'inverse de son erreur sur le jeu de données de validation).



(a) Relation entre l'agent et son environnement



(b) Détermination de la récompense pour un véhicule devant rester au centre de sa voie de circulation

Figure 1.22 – Apprentissage par Renforcement

Algorithme génétique

L'algorithme génétique [7, 20] est une méthode d'optimisation qui s'inspire de la théorie de l'évolution de Darwin [19] et du principe de « survie du plus apte » [78]. L'algorithme génétique se présente comme suit (figure 1.23) :

1. L'algorithme crée une « population » initiale de candidats. Chaque « individu » de la population est défini par ses « gènes ». Par exemple, dans le cas d'une structure en chaînes (1.2.3), les gènes seraient les éléments de la liste définissant l'architecture. Ainsi, dans la figure 1.17, $L[0]$ est le premier gène, $L[1]$ le deuxième, etc.
2. Puis chaque individu de la population est « évalué ». Dans le cas d'une NAS, cette évaluation consiste à entraîner les architectures de la population et extraire leurs erreurs respectives.
3. Des suites de l'évaluation, les meilleurs individus sont « sélectionnés ». Donc, pour des architectures de CNN, les individus avec la plus faible erreur sont retenus.
4. Les individus ainsi sélectionnés sont alors reproduits par « croisement » et/ou « mutation ». Le croisement consiste à mélanger les gènes de deux individus pour en créer un nouveau. Tandis que la mutation consiste à changer spontanément la valeur d'un ou plusieurs gènes d'un individu.

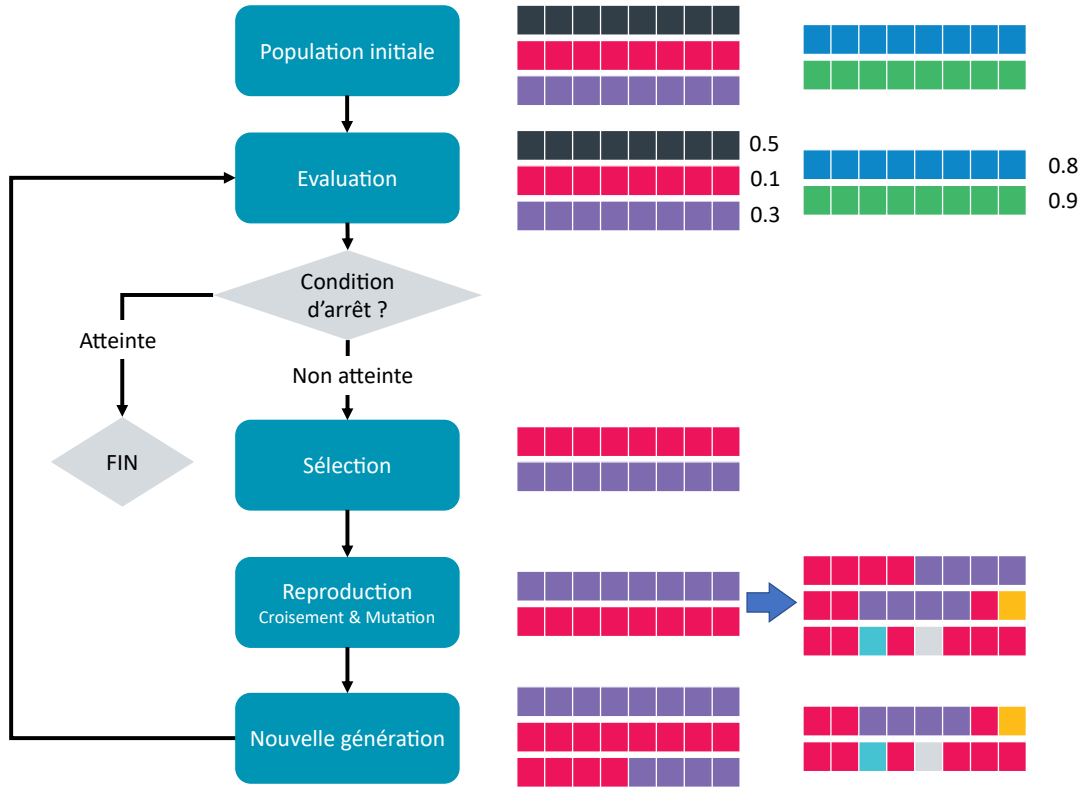


Figure 1.23 – Organigramme de l'algorithme génétique (à gauche). Et un exemple d'une génération (à droite), où une ligne correspond à un individu composé de gènes de couleurs

5. La nouvelle population créée lors de l'étape précédente forme donc la nouvelle « génération » de population et les étapes 2 et 3 seront répétées avec cette dernière.

Cet algorithme est répété pendant un certain nombre de générations, où chaque nouvelle génération est théoriquement plus performante que la précédente. L'algorithme s'arrête après une évaluation (deuxième point) si la condition d'arrêt est atteinte. Cette condition peut être un nombre précis de générations, une convergence de l'algorithme (par exemple si la plus faible erreur obtenue lors des évaluations ne change pas pendant un certain nombre de générations).

L'algorithme génétique peut aussi être utilisé pour l'entraînement du réseau de neurones. On parle alors de neuro-évolution. Il est ainsi possible, contrairement à la descente du gradient généralement utilisée pour l'entraînement, d'à la fois optimiser les paramètres du réseau et son architecture [82, 81, 80].

Descente du gradient

La descente du gradient, déjà présentée dans la section 1.1.2 pour optimiser les paramètres des NN, peut aussi être appliquée pour la NAS. Cependant, contrairement aux poids d'un NN, le choix des opérations o (contenues dans l'ensemble des opérations O) constitutifs d'une architecture de CNN n'est pas un problème continu. De ce fait, DARTS [53] propose une « relaxation continue » pour rendre l'espace de recherche continue et dérivable et ainsi permettre l'optimisation par descente du gradient.

Cette relaxation est faite par une fonction $\mathrm{softmax}$ $\overline{o}^{(i,j)}$ telle que suit :

$$\overline{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o' \in O} \exp\left(\alpha_{o'}^{(i,j)}\right)} o(x) \tag{1.25}$$

On peut voir cette méthode comme un double réseau de neurones. En effet, $\alpha_o^{(i,j)}$ représente le poids d'une opération o entre les nœuds i et j, et l'opération avec le poids $\alpha^{(i,j)}$ le plus élevé est retenue pour faire partie de l'architecture. Donc les $\overline{o}^{(i,j)}$ continues (appelées « opérations mixées » dans [53]) sont remplacées par l'opération $o^{(i,j)} = \arg\max_{o \in O} \alpha_o^{(i,j)}$.

Il y a ainsi deux types de poids : les poids α pour l'architecture et les poids ω à l'intérieur des convolutions. Dans DARTS [53], ces deux poids sont optimisés conjointement. Les poids ω est optimisés avec le jeu de données d'entraînement, avec les poids α fixés. De même, les poids α sont optimisés avec le jeu de données de validation, avec les poids ω fixés.

Recherche aléatoire

Une dernière manière de rechercher une architecture est tout simplement de créer des architectures aléatoirement, les entraîner et garder la meilleure. Même si cela peut paraître peu optimal, Li et Talwalkar [49] ont montrés que la recherche aléatoire est en fait compétitive avec l'état-de-l'art des NAS et atteignait des résultats comparables à ENAS [63].

Lorsque l'on présente un algorithme d'optimisation de NAS, il convient donc de vérifier que cette méthode performe mieux qu'une simple recherche aléatoire.

1.2.4 . Analyse de l'état de l'art et positionnement des travaux de thèses

Dans cette section, les différents espaces de recherche et techniques d'optimisation présentés précédemment sont comparés entre eux. Ensuite, une énumération des différents challenges de l'Auto-ML est faite.

Espaces de recherches

Concernant les espaces de recherche, la structure en chaîne (1.2.3) est certes une méthode simple pour des CNN basiques, mais pour des réseaux plus élaborés notamment avec des couches en parallèle (comme pour des ResNet ou GoogLeNet) cette représentation devient moins adaptée. En effet, cette méthode a pour désavantage d'avoir un nombre de degrés de liberté augmentant beaucoup avec le nombre de couches (chaque couche est un nouveau

degré). Quant au morphisme (1.2.3), il a pour désavantage de devoir commencer sa recherche à partir d'une architecture déjà existante. Cela limite donc les possibilités de rechercher des architectures différentes.

En revanche, les espaces de recherche basés sur les cellules (1.2.3), contrairement à la structure en chaîne, permettent d'augmenter la profondeur du réseau sans augmenter les degrés de liberté (il suffit d'augmenter le nombre de cellules déjà trouvées). Avec cette méthode, on perd en revanche en souplesse, car on ne peut que trouver des architectures avec un motif se répétant. Et un nouveau paramètre d'optimisation apparaît : le nombre de cellules. Ce paramètre doit donc être géré manuellement, ou il faut se tourner vers l'espace de recherche hiérarchique (1.2.3) qui optimise à la fois les cellules et la macro-architecture. Mais cela rajoute des paramètres à l'optimisation.

Espace de recherche	Ne nécessite pas de modèle initial	Gère les dimensions	Degrés de libertés controlâbles
En chaîne	X		
Basé sur les cellules	X		X
Hiérarchique	X	X	X
Morphisme		X	X

Table 1.3 – Comparaison entre espaces de recherche

D'après le tableau récapitulatif 1.3, l'espace de recherche hiérarchique est le plus versatile, ne nécessitant pas de modèle initial, pouvant gérer les dimensions des tenseurs et permettant de contrôler aisément le nombre de degrés de libertés. Ainsi, ce type d'espace de recherche paraît le plus prometteur pour un NAS adaptable (voir section 1.2.4). Notre proposition (chapitre 2) s'inspire principalement de ce type d'espace.

Techniques d'optimisations

Concernant le temps d'optimisation nécessaire à chaque technique d'optimisation, comme illustré par la table 1.4, l'algorithme génétique et l'apprentissage par renforcement présentent une grande variance dans ce domaine. En effet, certains Auto-ML utilisant l'algorithme génétique prennent de 2000 à 3000 jours de GPU [2] tandis que d'autres prennent moins d'un jour de GPU. De même, avec l'apprentissage par renforcement, les temps d'optimisation vont de 22 400 jours à moins d'une demi-journée. En revanche, la descente du gradient est la technique paraissant la plus rapide avec la plupart de ces temps ne dépassant pas un jour de GPU ou deux.

Ainsi, la descente du gradient semble efficace, et surtout plus rapide que ces homologues, mais est limitée à de l'optimisation mono-objectif et nécessite l'application d'une relaxation

2. Un jour de GPU est une unité de mesure du temps d'optimisation d'un Auto-ML. Il s'agit du nombre de jours d'optimisations multiplié par le nombre de GPUs utilisés lors de l'entraînement. Cela permet de faire une meilleure comparaison du temps d'optimisation en réduisant l'influence de la puissance de calcul disponible.

continue. Avec l'apprentissage par renforcement [40] ou l'algorithme génétique [55] en revanche, la mise en pratique de l'algorithme est plus directe qu'avec la descente du gradient et il est possible de faire de la recherche d'architecture multi-objectifs.

Challenges de l'Auto-ML

Il existe plusieurs challenges dans le domaine de l'Auto-ML, parmi lesquels :

1. Faire en sorte que les **performances** (*accuracy, f-score, precision and recall, Intersection over Union*, etc.) des modèles trouvés par la recherche soient toujours plus élevés. Il s'agit ici de l'objectif incontournable de la recherche en Intelligence Artificielle. Même en s'attaquant à d'autres problématiques, celui-ci ne peut être ignoré.
2. Réduire le **temps d'optimisation**. L'automatisation apportée par l'Auto-ML permet de réduire le temps passé par un humain à effectivement développer le modèle. Cependant, cela se fait au prix d'un temps d'optimisation de l'Auto-ML souvent élevé (voir table 1.4). Ainsi, chercher à réduire ce temps d'optimisation est un axe de recherche important.
3. Être capable d'optimiser **un maximum d'étapes** du pipeline de développement de *Deep Learning* (voir figure 1.16). En effet, le but ultime de l'Auto-ML serait d'avoir une automatisation de bout en bout de la création d'un modèle d'apprentissage automatique. Tandis qu'actuellement, les Auto-ML se concentrent principalement sur la recherche d'architecture et/ou l'optimisation des hyperparamètres.
4. Être **simple d'utilisation**. C'est-à-dire faire en sorte que l'utilisateur ait le moins de saisies à faire pour trouver un modèle qui lui convient, toujours dans l'esprit d'une automatisation la plus complète possible.
5. Être **adaptable**. C'est-à-dire avoir un seul Auto-ML capable de s'adapter pour travailler sur différents jeux de données et tâches (classification, détection d'objet, segmentation). Ainsi, un seul Auto-ML pourrait être capable de développer un modèle pour n'importe quelle tâche, réduisant encore la nécessité d'entrées manuelles.

Dans le cadre de cette thèse, nous avons décidé de nous focaliser sur les deux derniers points. En effet, le but étant de faciliter un maximum le développement de DL, que ce soit pour des utilisateurs expérimentés ou non. Le point de réduire le temps d'optimisation est cependant tout de même pris en compte, car il s'agit d'un paramètre important des Auto-ML.

1.3 . Positionnement par rapport à l'état de l'art

Il ressort de la recherche bibliographique que, bien que de nombreuses publications proposent des solutions de NAS [92, 95, 51, 52, 57, 90, 93] pouvant aussi fournir une optimisation multi-objectifs [40, 55, 88, 15], ces outils sont fermés et peu adaptables. En effet, ils sont élaborés pour une technique d'optimisation spécifique et des jeux de données particuliers (un algorithme de NAS développé pour de la classification ne pourra pas faire de segmentation).

Référence	Publié dans	#Params (Millions)	Top-1 Acc(%)	Jours de GPU	#GPUs	Algorithme
ResNet-110 [2]	ECCV16	1.7	93.57	-	-	Créé manuellement
PyramidNet [207]	CVPR17	26	96.69	-	-	
DenseNet [127]	CVPR17	25.6	96.54	-	-	
GeNet#2 (G-50) [30]	ICCV17	-	92.9	17	-	Algorithme génétique
Large-scale ensemble [25]	ICML17	40.4	95.6	2,500	250	
Hierarchical-EAS [19]	ICLR18	15.7	96.25	300	200	
CGP-ResSet [28]	IJCAI18	6.4	94.02	27.4	2	
AmoebaNet-B (N=6, F=128)+c/o [26]	AAAI19	34.9	97.87	3,150	450 K40	
AmoebaNet-B (N=6, F=36)+c/o [26]	AAAI19	2.8	97.45	3,150	450 K40	
Lemonade [27]	ICLR19	3.4	97.6	56	8 Titan	
EENA [149]	ICCV19	8.47	97.44	0.65	1 Titan Xp	
EENA (more channels)[149]	ICCV19	54.14	97.79	0.65	1 Titan Xp	
EA NASv3[12]	ICLR17	7.1	95.53	22,400	800 K40	Apprentissage par renforcement
NASv3+more filters [12]	ICLR17	37.4	96.35	22,400	800 K40	
MetaQNN [23]	ICLR17	-	93.08	100	10	
NASNet-A (7 @ 2304)+c/o [15]	CVPR18	87.6	97.60	2,000	500 P100	
NASNet-A (6 @ 768)+c/o [15]	CVPR18	3.3	97.35	2,000	500 P100	
Block-QNN-Connection more filter [16]	CVPR18	33.3	97.65	96	32 1080Ti	
Block-QNN-Depthwise, N=3 [16]	CVPR18	3.3	97.42	96	32 1080Ti	
ENAS+macro [13]	ICML18	38.0	96.13	0.32	1	
ENAS+micro+c/o [13]	ICML18	4.6	97.11	0.45	1	
Path-level EAS [139]	ICML18	5.7	97.01	200	-	
Path-level EAS+c/o [139]	ICML18	5.7	97.51	200	-	
ProxylessNAS-RL+c/o[132]	ICLR19	5.8	97.70	-	-	
FPNAS[208]	ICCV19	5.76	96.99	-	-	
RL DARTS(first order)+c/o[17]	ICLR19	3.3	97.00	1.5	4 1080Ti	Descente du gradient
DARTS(second order)+c/o[17]	ICLR19	3.3	97.23	4	4 1080Ti	
sharpDARTS [178]	ArXiv19	3.6	98.07	0.8	1 2080Ti	
P-DARTS+c/o[128]	ICCV19	3.4	97.50	0.3	-	
P-DARTS(large)+c/o[128]	ICCV19	10.5	97.75	0.3	-	
SETN[209]	ICCV19	4.6	97.31	1.8	-	
GDAS+c/o [154]	CVPR19	2.5	97.18	0.17	1	
SNAS+moderate constraint+c/o [155]	ICLR19	2.8	97.15	1.5	1	
BayesNAS[210]	ICML19	3.4	97.59	0.1	1	
ProxylessNAS-GD+c/o[132]	ICLR19	5.7	97.92	-	-	
PC-DARTS+c/o [211]	CVPR20	3.6	97.43	0.1	1 1080Ti	
MiLeNAS[153]	CVPR20	3.87	97.66	0.3	-	
SGAS[212]	CVPR20	3.8	97.61	0.25	1 1080Ti	
GDAS-NSAS[213]	CVPR20	3.54	97.27	0.4	-	
GD NASBOT[160]	NeurIPS18	-	91.31	1.7	-	SMBO
PNAS [18]	ECCV18	3.2	96.59	225	-	
EPNAS[166]	BMVC18	6.6	96.29	1.8	1	
GHN[214]	ICLR19	5.7	97.16	0.84	-	
SMBO NAO+random+c/o[169]	NeurIPS18	10.6	97.52	200	200 V100	Recherche aléatoire
SMASH [14]	ICLR18	16	95.97	1.5	-	
Hierarchical-random [19]	ICLR18	15.7	96.09	8	200	
RandomNAS [180]	UAI19	4.3	97.15	2.7	-	
DARTS - random+c/o [17]	ICLR19	3.2	96.71	4	1	
RandomNAS-NSAS[213]	CVPR20	3.08	97.36	0.7	-	
RS NAO+weight sharing+c/o [169]	NeurIPS18	2.5	97.07	0.3	1 V100	GD+SMBO
RENASNet+c/o[42]	CVPR19	3.5	91.12	1.5	4	EA+RL
EA CARS[40]	CVPR20	3.6	97.38	0.4	-	EA+GD

Table 1.4 – Nombre de paramètres et performance atteinte par des modèles trouvé par différents Auto-MLs. Un tiret (-) signifie que l'information n'est pas donnée par la publication originale. Les Auto-MLs sont regroupés par algorithme d'optimisation. RL, EA, GD et SMBO signifient respectivement Apprentissage par Renforcement / *Reinforcement learning*, *Evolutionary Algorithm*/Algorithme génétique, *gradient descent*/Descente du gradient, et *Surrogate Model-Based Optimization*. Adapté de [38].

C'est pourquoi, les travaux de cette thèse cherchent à fournir un outil de NAS ouvert dont l'espace de recherche large peut être facilement personnalisable : l'utilisateur est libre de choisir les opérations de convolution et de pooling présentes dans l'espace voire de déterminer des blocs prédéfinis, par exemple des blocs RES (section 1.1.4) ou INCEPTION (section 1.1.4). De plus, cet espace de recherche se veut « naïf », c'est-à-dire que n'importe quel algorithme d'optimisation peut lui être appliqué.

Dans les sections suivantes, nous présenterons donc une proposition d'espace de recherche dit « imbriqué » dont les objectifs sont les suivants :

1. **Adaptabilité de l'optimisation** : Afin de proposer une plus grande liberté d'utilisation, l'espace de recherche doit être capable de s'adapter à n'importe quel algorithme d'optimisation selon les besoins de l'utilisateur.
2. **Adaptabilité de la recherche** : L'espace doit être capable de rechercher des architectures variées. C'est-à-dire que l'utilisateur peut orienter la recherche vers des architectures spécifiques (comme des ResNet ou des DenseNet).
3. Permettre le **contrôle des degrés de liberté** : Afin de maîtriser le temps d'optimisation, l'espace de recherche doit pouvoir être configuré de façon à avoir la main sur le nombre de degrés de liberté.

1.4 . Conclusion

Les CNN sont un type de DL particulièrement efficace pour le traitement d'image. Cependant, leur développement (et leur optimisation) nécessite du temps, des ressources matérielles informatique et surtout de l'expérience. Ainsi, la NAS est une discipline recherchant automatiquement l'architecture optimale par rapport à un jeu de données. Les NAS actuelles sont cependant peu ouvertes et il faudra utiliser un *framework* de NAS différent selon chaque tâche (classification, détection d'objets ou segmentation). Le sujet de cette thèse est donc le développement d'un outil de NAS, « Open-NAS », adaptable. C'est-à-dire qu'il peut être utilisé avec n'importe quel algorithme d'optimisation, ces degrés de liberté sont contrôlables et il est applicable à différents jeux de données.

La première contribution de cette thèse est la formalisation d'un espace de recherche reposant sur un principe d'imbrication. C'est-à-dire que des blocs élémentaires sont utilisés pour former des blocs composés, qui eux-mêmes peuvent à nouveau former des blocs composés de plus haut niveau, jusqu'à former une architecture de réseau de neurones. Cela nous permet in fine de représenter des architectures de CNN complexes avec un nombre de paramètres plus ou moins réduit selon le niveau d'imbrication que l'on utilise. Cette formalisation est décrite dans le chapitre 2.

Le chapitre 3 présente trois propositions de stratégies d'optimisation applicables à l'espace de recherche présenté dans le chapitre 2, non sans avoir au préalable présenté le problème d'optimisation multi-objectifs sous contrainte.

Enfin, le chapitre 4 présente tout d'abord l'implémentation de OpenNas, puis, diverses expériences menées avec cet outil afin d'en évaluer les capacités.

2 - Espace de recherche imbriqué

Dans le but d'avoir un modèle d'espace de recherche adaptable, permettant à la fois de contrôler les degrés de liberté et les architectures possibles, ce chapitre formalise une manière de présenter des architectures de CNN sous forme d'une imbrication de blocs. Cette forme permet in fine de représenter un CNN avec un simple entier naturel, facilitant l'application d'un algorithme d'optimisation quelconque, et se veut paramétrable pour laisser un maximum de liberté et de contrôle lors de la définition de l'espace de recherche pour une optimisation.

Pour cela, une présentation de l'espace de recherche imbriqué à l'aide de la théorie des ensemble est faite. Puis les algorithmes d'encodage et de décodage qui permettent de convertir un uplet ou un multiensemble d'entiers naturels en un unique entier naturel sont expliqués. Il est aussi montré des variantes de ces algorithmes permettant de générer des uplets et multiensembles de façon dynamique à partir d'un entier (c'est à dire avec un nombre variable d'éléments).

2.1 Introduction . . . 62
2.2 Présentation de la structure de l'espace de recherche et de sa complexité . . . 63
2.2.1 Blocs élémentaires . . . 63
2.2.2 Blocs composés . . . 64
Composition série pour créer des Pipelines . . . 65
Composition parallèle pour créer des Rubans . . . 70
2.2.3 Architecture neuronale à base des blocs . . . 74
2.2.4 Exemples d'imbrications recréant des CNN connus . . . 77
VGG . . . 77
GoogLeNet . . . 77
ResNet-50 . . . 78
2.3 Représentation d'une architecture par des entiers . . . 78
2.3.1 Encodage et décodage des pipelines . . . 79
Encodage des pipelines . . . 81
Décodage des pipelines . . . 82
2.3.2 Décodage et encodage des rubans . . . 84
Décodage des rubans . . . 85
Encodage des rubans . . . 90
2.4 Conclusion . . . 92

2.1 . Introduction

Dans le chapitre précédent, nous avons présenté la Recherche d'architecture neuronale / *Neural Architecture Search* (NAS), la discipline du Apprentissage automatique automatisé / *Automated Machine Learning* (Auto-ML), qui se concentre sur la recherche d'une architecture de CNN optimale sur un jeu de données. Un NAS, pour fonctionner, a besoin d'un espace de recherche définissant les architectures trouvables et d'un algorithme d'optimisation pour trouver l'architecture optimale le plus rapidement possible. Dans ce chapitre nous nous concentrerons sur la formalisation d'un espace de recherche.

En effet, les espaces de recherche présentés dans l'état de l'art sont restreints : il sont optimisés pour un algorithme d'optimisation particulier, pour une tâche particulière (classification, détection d'objets, segmentation). C'est pourquoi dans ce chapitre, nous présentons un espace de recherche « imbriqué » qui se veut adaptable. C'est-à-dire pouvant accepter tous types d'algorithmes d'optimisation et pouvant être appliqué sur tous types de tâches. De plus, cet espace est largement contrôlable du point de vue de ces degrés de liberté et de ses possibilités d'architectures.

Ainsi, dans la première partie 2.2, nous présenterons la formalisation de l'espace de recherche imbriqué à l'aide de la théorie des ensembles. En effet, l'espace est basé sur un ensemble de blocs élémentaires (section 2.2.1) qui contiennent les couches du réseau (convolution, sous-échantillonage, activation, etc.). Un sous-ensemble de ces blocs élémentaires est arrangé pour former des blocs composés (section 2.2.2). L'ensemble de ses arrangements de sous-ensemble de blocs élémentaires forme ainsi l'ensemble des blocs composés de niveau 1. Ensuite, ces blocs composés de niveau 1 servent à leur tour de base pour former l'ensemble des blocs composés de niveau 2, etc. Nous présentons aussi les deux types de blocs composés : les pipelines (section 2.2.2) où les blocs qui le composent sont placés en série (il s'agit donc d'un arrangement avec répétition), et les rubans (section 2.2.2) où les blocs qui le composent sont placés en parallèle (il s'agit donc d'une combinaison avec répétition).

Dans une deuxième partie 2.3, nous présenterons comment les pipelines et les rubans peuvent être encodés afin d'être représentés par un entier naturel. Cela permet d'avoir un seul paramètre d'optimisation, ensuite applicable dans une fonction objectif. L'encodage des pipelines et des rubans est différent, en effet les pipelines étant des arrangements et les rubans des combinaisons, leur conversion en entier ne peut se faire par le même algorithme. Ainsi, la section 2.3.1 présente les algorithmes d'encodage et de décodage des pipelines, ayant une complexité logarithmique. Cependant, le codage des rubans, ne peut se faire qu'avec un algorithme ayant une complexité exponentielle. Dans la section 2.3.2, nous présenterons donc tout d'abord des algorithmes de la littérature permettant le décodage des rubans. Puis nous présentons un algorithme développé lors de cette thèse. Celui-ci toujours de complexité exponentielle mais évitant certaines boucles par rapport aux algorithmes précédents, réduisant ainsi tout de même le temps d'exécution.

2.2 . Présentation de la structure de l'espace de recherche et de sa complexité

Pour créer l'architecture de CNN, on utilise un principe d'assemblage de blocs. Les blocs sont assemblés afin de créer une structure, et cette structure peut elle-même servir de bloc pour créer une nouvelle structure.

On appelle donc « bloc » un élément constitutif de notre espace de recherche. Un bloc a une seule entrée et une seule sortie. On distingue deux types de blocs : les blocs « élémentaires » et les blocs « composés » :

1. **Blocs élémentaires** (niveau 0) : ces blocs sont la base de l'espace l'espace de recherche, et permettent de créer des blocs supérieurs. ils constituent le niveau 0 de l'architecture.
2. **Blocs composés** (niveau ≥ 1) : ces blocs sont composés d'autres blocs. Les blocs composés de blocs élémentaires sont de niveau 1, les blocs composés de blocs niveau 1 sont de niveau 2, etc. On distingue deux types de compositions :
   (a) **Composition série (Pipeline)** : les blocs sont assemblés en série.
   (b) **Composition parallèle (Ruban)** : les blocs sont regroupés en parallèle.

Les sections suivantes présentent tout d'abord les trois types de blocs en détails : tout d'abord les blocs élémentaires, puis les deux types de blocs composés (pipeline et rubans). Il est ensuite expliqué comment les blocs sont imbriqués pour former des structures complexes.

2.2.1 . Blocs élémentaires

Les blocs élémentaires forment la base de l'espace de recherche et sont définis manuellement. Ces blocs peuvent avoir une ou plusieurs couches de convolution (section 1.1.3) pour extraire les cartes de caractéristiques du tenseur d'entrée $\mathbf{X}$ (de dimensions $(h_{\mathbf{X}}, w_{\mathbf{X}}, c_{\mathbf{X}})$) en gardant la même taille de ce dernier. Ils peuvent aussi avoir une couche de sous-échantillonnage (pooling voir section 1.1.3) pour réduire la taille du tenseur de sortie $\mathbf{Y}$ (figure 2.1). Les blocs élémentaires sont donc des applications e telles que $e(\mathbf{X}) = \mathbf{Y}$.

Les dimensions du tenseur de sortie dépendent des dimensions du tenseur d'entrée et des paramètres de configuration de ses couches. Nous appelons aussi ces paramètres de configuration « options ». Dans cette thèse, nous n'utilisons que deux paramètres d'options p et $r \in \mathbb{N}$ pour configurer un bloc e, noté par $e(r, p)$. On précise que la notation $e(r, p)$ correspond à la configuration d'un bloc e avec des paramètres r et p. Le premier paramètre r définit le nombre de cartes de fonction (*feature maps*) dans le tenseur de sortie en configurant le nombre de filtres de la dernière couche de convolution à $2^r c_{\mathbf{X}}$. Le deuxième paramètre p définit si la taille du tenseur de sortie est réduite en configurant le pas (*stride*) de la couche de sous-échantillonnage et la taille de sa fenêtre d'échantillonage sur la même valeur 2^p (voir section 1.1.3). On obtient une taille de tenseur de sortie qui est égale à $\left(\lfloor \frac{h_{\mathbf{X}}}{2^p} \rfloor, \lfloor \frac{w_{\mathbf{X}}}{2^p} \rfloor\right)$.

Dans la figure 2.1, le bloc e_1, à gauche, présente un bloc simple uniquement constitué d'une couche de convolution avec un noyau 3×3. L'utilisation de bloc simple tel que dans le

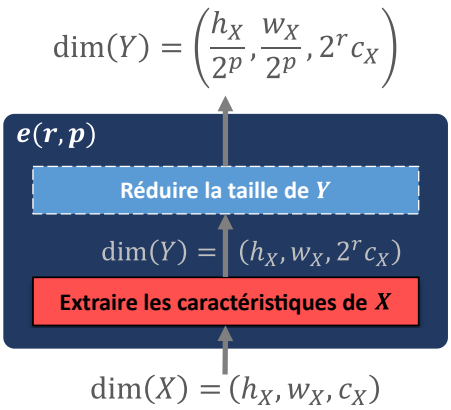


Figure 2.1 – Bloc élémentaire avec sa configuration.

bloc e_1 permet de rendre la recherche d'architecture plus libre. À l'inverse, on peut orienter la recherche vers des architectures plus spécifiques en utilisant des blocs élémentaires plus élaborés. Dans la figure 2.2, le bloc e_2, à droite, est un bloc utilisé dans l'architecture ResNet [37]. Cela permet donc d'inciter la recherche à trouver des architectures ressemblant à un ResNet.

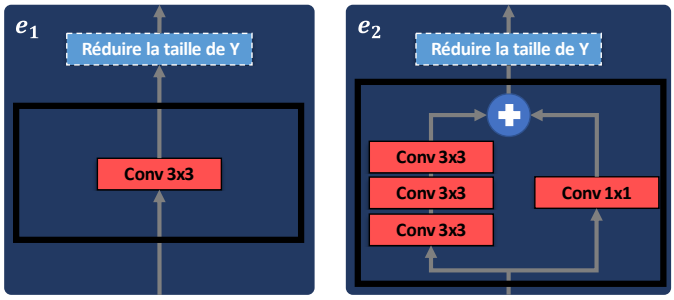


Figure 2.2 – Exemples de blocs élémentaires dont la partie pour extraire les caractéristiques est développée. e_1 est un bloc simple représentant une unique couche de convolution 3×3, tandis que e_2 représente un bloc identité (utilisé dans ResNet [37])

L'ensemble des blocs élémentaires est nommé $\mathbb{E}$ tel que $\mathbb{E} = \{e_1, e_2, \ldots, e_{n_0}\}$, avec n_0 le nombre de blocs définis dans $\mathbb{E}$. Dans la section suivante, nous allons voir comment créer des nouveaux blocs à partir des blocs élémentaires de $\mathbb{E}$.

2.2.2 . Blocs composés

Un bloc est dit « bloc composé » car il est créé à partir d'autres blocs. On nomme $\mathbb{B}_i$ l'ensemble des blocs composés créés à partir d'un autre ensemble de blocs composés $\mathbb{B}_{i-1}$. L'indice $i \in \mathbb{N}^*$ présente le niveau de la composition et $\mathbb{B}_0 = \mathbb{E}$ est l'ensemble des blocs élémentaires. Le principe de niveaux est expliqué plus en détails dans la section 2.2.3.

Définition 2.2.1 *Soit $\mathbb{B}_{i-1}$ un ensemble de cardinal n_{i-1}. On définit l'ensemble $\mathbb{B}_i$ comme l'ensemble des blocs composés, tel que $\mathbb{B}_i = \{b_{i1}, ..., b_{ij}, ..., b_{in_i}\}$. Chaque bloc composé b_{ij} est construit en utilisant des éléments d'un multiensemble $\pi_{b_{ij}}$, dont l'ensemble support est $\mathbb{B}_{i-1}$. Le multiensemble $\pi_{b_{ij}}$ doit avoir une cardinalité inférieure ou égale à $m_i \in \mathbb{N}^*$, où m_i représente le nombre d'emplacements maximum possible pour positionner les éléments du multiensemble $\pi_{b_{ij}}$ dans le bloc composé b_{ij}. On dit que l'ensemble des blocs composés $\mathbb{B}_i$ existe si les conditions suivantes sont vérifiées :*

1. *Aucun des blocs composés de $\mathbb{B}_i$ n'est vide :*

$$\forall b_{ij} \in \mathbb{B}_i, \pi_{b_{ij}} \neq \varnothing$$

2. *L'union de tous les multiensembles des blocs composés de $\mathbb{B}_i$ est égale à $\mathbb{B}_{i-1}$:*

$$\bigcup_{j=1}^{n_i} \pi_{b_{ij}} = \mathbb{B}_{i-1}$$

3. *Tous les blocs composés de $\mathbb{B}_i$ sont distincts deux à deux :*

$$\forall j, k \in [\![1, n_i]\!], j \neq k \Rightarrow b_{ij} \neq b_{ik}$$

4. *Le cardinal de chaque multiensemble de bloc composé de $\mathbb{B}_i$ est inférieur ou égal à m_i :*

$$\forall j \in [\![1, n_i]\!], |\pi_{b_{ij}}| \leq m_i$$

Dans les deux sections suivantes, nous allons voir comment utiliser des applications de composition série et de composition parallèle pour créer un ensemble des blocs composés en respectant la définition 2.2.1.

Composition série pour créer des Pipelines

Un pipeline est un bloc composé dont les blocs internes sont placés en série (figure 2.3. De ce fait, l'ordre des blocs internes dans le bloc composé de type pipeline est important.

Définition 2.2.2 *On considère un ensemble $\mathbb{B}_{i-1}$ de cardinal n_{i-1}. On peut former un nouvel ensemble $\mathbb{B}_i$ qui contient tous les pipelines possibles, chacun étant un arrangement avec répétitions de m_i éléments parmi $\mathbb{B}_{i-1}$. Autrement dit, chaque élément de $\mathbb{B}_i$ une séquence de m_i d'éléments appartenant à $\mathbb{B}_{i-1}$, qui sont reliés en série pour former un pipeline. Cette notion peut être représentée comme la puissance cartésienne $m_i^{ième}$ de l'ensemble $\mathbb{B}_{i-1}$, notée $\mathcal{P}(\mathbb{B}_{i-1}, m_i) = \mathbb{B}_{i-1}^{m_i}$. On peut également voir cet ensemble comme le produit cartésien de m_i copies de $\mathbb{B}_{i-1}$. Ainsi, chaque élément de $\mathbb{B}_i$ est un pipeline de m_i-uplet $(x_1, x_2, ..., x_{m_i})$, où chaque $x_j \in \mathbb{B}_{i-1}$:*

$$\mathcal{P}\left(\mathbb{B}_{i-1}, m_i\right) = \mathbb{B}_{i-1}^{m_i} = \prod_{k=1}^{m_i} \mathbb{B}_{i-1} = \{(x_1, x_2, ..., x_{m_i}) \,|\forall j, x_j \in \mathbb{B}_{i-1}\} \tag{2.1}$$

Ainsi, le cardinal n_i de l'ensemble $\mathbb{B}_i$ est

$$n_i = n_{i-1}^{m_i} \tag{2.2}$$

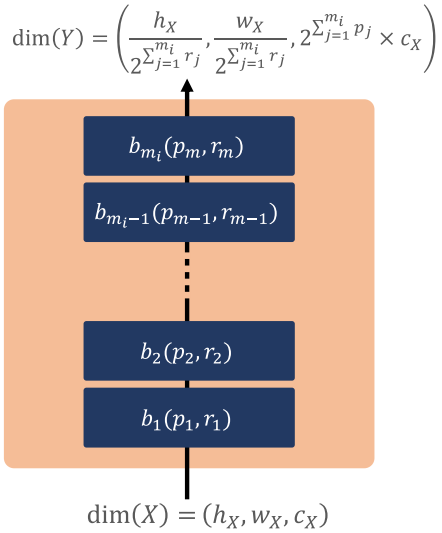


Figure 2.3 – Représentation d'un pipeline de $\mathbb{B}_i$ avec n_i blocs internes. Les options (p, r) pour chaque bloc interne b_{ij} sont aussi représentées ainsi que leur influence sur les dimensions du tenseur de sortie du pipeline.

Exemple 2.2.1 *La figure 2.4 illustre $8 = 2^3$ pipelines composés de 2 blocs élémentaires : $\{e_1, e_2\}$ où chaque élément est distinct de l'autre $e_1 \neq e_2$. Dans chaque pipeline, il y a 3 places (3-uplet) pour ordonner les blocs élémentaires.*

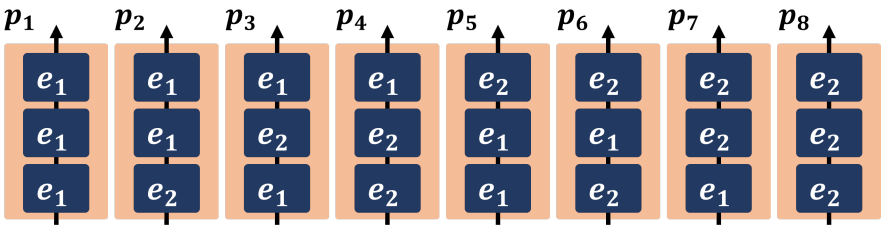


Figure 2.4 – Exemple de pipelines à 3 emplacements pour organiser 2 blocs élémentaires.

Le tableau présenté ci-dessous représente le même exemple que celui illustré par la figure précédente. Pour les exemples suivants, nous utiliserons des tableaux plutôt que des figures afin d'optimiser l'espace dans ce chapitre.

Table 2.1 – Exemple de pipelines avec 3 emplacements pour ordonner 2 blocs élémentaires en utilisant la définition 2.2.2.

l_i \ p_i	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
l_1	e_1	e_2	e_1	e_2	e_1	e_2	e_1	e_2
l_2	e_1	e_1	e_2	e_2	e_1	e_1	e_2	e_2
l_3	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2

La définition 2.2.2 satisfait les quatre conditions de la définition 2.2.1. Toutefois, elle peut être étendue, car tous les pipelines créés sont toujours construits à partir de m_i blocs de l'ensemble $\mathbb{B}_{i-1}$, ce qui signifie que les cardinalités des pipelines sont toujours fixes. Donc $\forall j \in [\![1, n_i]\!], |\pi_{b_{ij}}| = m_i$. Nous cherchons donc à créer des blocs avec des cardinalités pouvant être inférieures ou égales m_i (de taille dynamique), afin de mieux correspondre à la condition 4 de la définition 2.2.1. Dans la littérature, un bloc SKIP est utilisé pour créer des blocs avec des cardinalités dynamiques. Ce bloc ne contient pas un traitement, il se contente de transférer l'entrée vers la sortie. Ce bloc est déjà utilisé dans les travaux [27, 51, 90] pour construire des blocs avec des cardinalités dynamiques. Cependant, l'utilisation de ce type de bloc avec les pipelines ne respecte pas les conditions 1 et 3 de la définition 2.2.1. Le bloc SKIP peut créer un bloc composé contenant un ensemble vide, ce qui rend la première condition non respectée. De plus, le bloc SKIP peut créer plusieurs similitudes de blocs composés, ce qui ne respecte pas la troisième condition. Il est claire que la cardinalité de l'ensemble $\mathbb{B}_i$ en ajoutant le bloc SKIP est

$$n_i = (n_{i-1} + 1)^{m_i} \tag{2.3}$$

Exemple 2.2.2 *Le tableau 2.2 illustre* $27 = (2+1)^3$ *pipelines composés de* 2 *blocs élémentaires* $\{e_1, e_2\}$ *et un bloc SKIP* $\{e_0\}$*. Dans chaque pipeline, il y a 3 places (3-uplet) pour ordonner les blocs élémentaires avec le SKIP. Les blocs SKIP* e_0 *sont notés* $-$ *dans le tableau.*

Table 2.2 – Exemple de pipelines avec 3 emplacements pour ordonner 2 blocs élémentaires et 1 bloc SKIP.

l_i \ p_i	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}
l_1	$-$	e_1	e_2	$-$	e_1	e_2	$-$	e_1	e_2	$-$	e_1	e_2	$-$	e_1
l_2	$-$	$-$	$-$	e_1	e_1	e_1	e_2	e_2	e_2	$-$	$-$	$-$	e_1	e_1
l_3	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	e_1	e_1	e_1	e_1	e_1
l_i \ p_i	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}	p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	
l_1	e_2	$-$	e_1	e_2	$-$	e_1	e_2	$-$	e_1	e_2	$-$	e_1	e_2	
l_2	e_1	e_2	e_2	e_2	$-$	$-$	$-$	e_1	e_1	e_1	e_2	e_2	e_2	
l_3	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2	

Dans le tableau 2.2, on peut clairement observer que le premier pipeline p_1 est composé de trois blocs SKIP connectés en série. Cela signifie que ce pipeline n'effectue aucun traitement sur son entrée, ce qui contredit la première condition définie dans 2.2.1. De plus, les sous-ensembles de pipelines $\{p_2, p_4, p_{10}\}$, $\{p_3, p_7, p_{19}\}$, $\{p_5, p_{11}, p_{13}\}$, $\{p_6, p_{12}, p_{16}\}$, $\{p_8, p_{20}, p_{22}\}$ et $\{p_9, p_{21}, p_{25}\}$ correspondent à des pipelines qui effectuent exactement les mêmes traitements. Cela ne respecte pas la troisième condition définie dans 2.2.1.

Afin de respecter toutes les conditions de la définition 2.2.1, nous utilisons la définition 2.2.3 pour construire des pipelines avec des cardinalités dynamiques qui prennent des valeurs entre 1 et m_i.

Définition 2.2.3 *Soit $\mathbb{B}_{i-1}$ un ensemble de cardinalité n_{i-1}. On peut former un nouvel ensemble $\mathbb{B}_i$ qui contient tous les pipelines possibles avec des cardinalités dynamiques qui varient entre 1 et m_i en utilisant l'équation 2.1 de la définition 2.2.2 :*

$$\mathcal{P}\left(\mathbb{B}_{i-1}, 1 : m_i\right) = \bigcup_{k=1}^{m_i} \mathcal{P}\left(\mathbb{B}_{i-1}, k\right) = \bigcup_{k=1}^{m_i} \mathbb{B}_{i-1}^{k} \tag{2.4}$$

Ainsi, le cardinal n_i de l'ensemble $\mathbb{B}_i$ devient

$$n_i = \sum_{k=1}^{m_i} n_{i-1}^{k} = \begin{cases} \frac{n_{i-1}\left(n_{i-1}^{m_i}-1\right)}{n_{i-1}-1} & \text{si } n_{i-1} > 1, \\ m_i & \text{si } n_{i-1} = 1. \end{cases} \tag{2.5}$$

Exemple 2.2.3 *Le tableau 2.3 illustre $14 = 2^1 + 2^2 + 2^3$ pipelines composés de 2 blocs élémentaires $\{e_1, e_2\}$ en utilisant la définition 2.2.3. Dans chaque pipeline, il y a 3 places (3-uplet) pour ordonner les blocs élémentaires.*

Table 2.3 – Exemple de pipelines avec 3 emplacements pour ordonner 2 blocs élémentaires en utilisant la définition 2.2.3.

l_i \ p_i	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}
l_1	e_1	e_2	e_1	e_2	e_1	e_2	e_1	e_2	e_1	e_2	e_1	e_2	e_1	e_2
l_2	–	–	e_1	e_1	e_2	e_2	e_1	e_1	e_2	e_2	e_1	e_1	e_2	e_2
l_3	–	–	–	–	–	–	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2

Dans le tableau 2.3, on peut clairement observer que l'ensemble de les pipelines créés par la définition 2.2.3 respecte parfaitement toutes les conditions de la définition 2.2.1. De plus, La cardinalité donnée par la définition 2.2.3 équation 2.5 est très inférieure de celle donnée par l'équation 2.3 dans le cas on veut utiliser le bloc SKIP. La différence entre les deux cardinalités nous donne le nombre des pipelines dupliqués plus le pipeline SKIP qui ne contient que le bloc SKIP. Par exemple, en prenant la cardinalité donnée dans l'exemple 2.2.2 qui est de 27 et celle de l'exemple 2.2.3 qui est de 14 ; la différence est de 13 (12 dupliqués (2*6 sous-ensembles) et 1 avec des SKIP).

La théorème ci-dessous montre que la cardinalité donnée par l'équation 2.5 est toujours inférieure de celle donnée par l'équation 2.3 pour tout n et m. ou m présente le nombre d'emplacements dans un pipeline (ou m-uplet) et n est le nombre de blocs utilisé pour créer les nouveaux blocs composés.

Théorème 2.2.1 *Pour tout m et $n \in \mathbb{N}^*$, $(n+1)^m$ est strictement supérieur à $n + n^2 + \cdots + n^m$:*

$$(n+1)^m > \sum_{k=1}^{m} n^k = \begin{cases} \frac{n(n^m-1)}{n-1} & \text{si } n > 1, \\ m & \text{si } n = 1. \end{cases} \tag{2.6}$$

Preuve 2.2.1 *On considère deux cas $n = 1$ et $n > 1$.*
***Cas** $n = 1$: l'inégalité 2.6 devient $2^m > m$. Comme le terme de droite (2^m) et strictement supérieure au terme de gauche (m), on dit que l'inégalité 2.6 est bien vérifiée dans le cas où $n = 1$ pour tout $m \in \mathbb{N}^*$.*

***Cas** $n > 1$: Dans ce cas, nous démontrons que la suite u_m est strictement positive pour tout entier naturel m et n. La suite u_m est donnée par la différence entre le terme de droite et le terme de gauche dans l'inégalité 2.6 :*

$$u_m = (n+1)^m - \frac{n\left(n^m - 1\right)}{n-1} > 0 \tag{2.7}$$

dans le cas où $m = 1$, l'inégalité 2.7 donne :

$$u_1 = (n+1) - n = 1 > 0$$

Puisque $u_1 > 0$, pour démontrer que l'inégalité 2.6 pour tout $m \in \mathbb{N}^*$, montrons que u_m est strictement croissante :

$$\forall m \in \mathbb{N}^*, u_{m+1} > u_m \quad \Leftrightarrow \quad (n+1)^{m+1} - \frac{n\left(n^{m+1} - 1\right)}{n-1} > (n+1)^m - \frac{n\left(n^m - 1\right)}{n-1} \tag{2.8}$$

$$\Leftrightarrow \quad (n+1)^{m+1} - (n+1)^m > \frac{n\left(n^{m+1} - 1\right)}{n-1} - \frac{n\left(n^m - 1\right)}{n-1} \tag{2.9}$$

$$\Leftrightarrow \quad (n+1)^m (n+1-1) > \frac{n\left(n^{m+1} - n^m\right)}{n-1} \tag{2.10}$$

$$\Leftrightarrow \quad n(n+1)^m > n^{m+1} \frac{n-1}{n-1} \tag{2.11}$$

$$\Leftrightarrow \quad (n+1)^m > n^m \tag{2.12}$$

$$\Leftrightarrow \quad n+1 > n \tag{2.13}$$

$$\Leftrightarrow \quad 1 > 0 \tag{2.14}$$

Puisque $1 > 0$, *l'inégalité* $u_{m+1} > u_m$ *est vraie, et donc la suite* u_m *est strictement croissante. Nous pouvons conclure que* $(n+1)^m$ *est strictement supérieur à* $n + n^2 + ... + n^m$ *pour tout* m *et* n *appartenant à* $\mathbb{N}^*$.

Composition parallèle pour créer des Rubans

Dans un bloc composé de type ruban, les blocs internes sont disposés en parallèle, contrairement à un pipeline où ils sont disposés en série. C'est-à-dire que tous les blocs qui composent un ruban ont pour entrée celle du ruban, puis les sorties de tous les blocs sont concaténées ou additionnées afin de former la sortie du ruban. Dans un ruban, l'ordre des blocs internes n'a pas d'importance contrairement au pipeline.

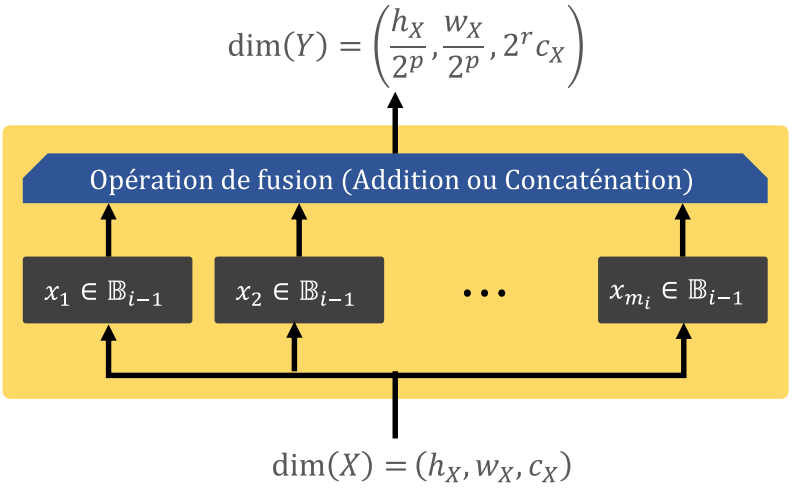


Figure 2.5 – Représentation d'un ruban de $\mathbb{B}_i$ avec n_1 blocs internes. L'influence des options sur les dimensions du ruban est représentée en sortie.

Définition 2.2.4 *On considère un ensemble* $\mathbb{B}_{i-1}$ *de cardinal* n_{i-1}. *On peut former un nouvel ensemble* $\mathbb{B}_i$ *qui contient tous les rubans possibles, chacun étant une* m_i*-combinaison avec répétition de l'ensemble* $\mathbb{B}_{i-1}$. *L'ordre n'étant pas pris en compte, deux séquences définissent le même bloc composé si l'on peut obtenir l'un à partir de l'autre en permutant ces éléments. Autrement dit, un ruban peut être représenté comme un multiensemble.*

Donc, l'ensemble de $\mathbb{B}_i$ *est l'ensemble de tout les multiensembles distincts de cardinal* m_i *sur l'ensemble* $\mathbb{B}_{i-1}$ *de cardinal* n_{i-1}, *notée* $\mathcal{R}(\mathbb{B}_{i-1}, m_i)$:

$$\mathcal{R}(\mathbb{B}_{i-1}, m_i) = \left\{ \bigcup_{k=1}^{n_{i-1}} (b_{ik}, x_k) \mid \forall k, b_{ik} \in \mathbb{B}_{i-1} \text{ et } x_k \in [\![0, m_i]\!] ; \sum_{k=1}^{n_{i-1}} x_k = m_i \right\} \quad (2.15)$$

avec (b_{ik}, x_k) *représentant un bloc* b_{ik} *avec une multiplicité de* x_k*. C'est-à-dire que* $(b_{ik}, 0) = \varnothing,\ (b_{ik}, 1) = \{b_{ik}\},\ (b_{ik}, 2) = \{b_{ik}, b_{ik}\},\ (b_{ik}, 3) = \{b_{ik}, b_{ik}, b_{ik}\}$*, etc.*

Ainsi, le cardinal n_i *de* $\mathbb{B}_i$ *est le nombre de solutions de l'équation* $\sum_{k=1}^{n_{i-1}} x_k = m_i$ *avec* $x_k \in [\![0, m_i]\!]$ *ce qui donne :*

$$n_i = |\mathbb{B}_i| = \left(\!\!\binom{n_{i-1}}{m_i}\!\!\right) = \binom{n_{i-1} + m_i - 1}{m_i} = \frac{(n_{i-1} + m_i - 1)!}{m_i!\,(n_{i-1} - 1)!} \tag{2.16}$$

Le symbole $\left(\!\!\binom{n}{k}\!\!\right)$ *est dit « n multichoisit k » (par analogie avec l'anglais «* multichoose *»)*

Remarque 2.2.1 *Le cardinal* n_i *de* $\mathbb{B}_i$ *est le nombre de solutions de l'équation* $\sum_{k=1}^{n_{i-1}} x_k = m_i$ *avec* $x_k \in [\![0, m_i]\!]$*. Or, les solutions de l'équation* $\sum_{k=1}^{n_{i-1}} x_k = m_i$ *sont aussi les* n_i*-compositions faibles de* m_i*. On peut donc aussi représenter les rubans de* $\mathbb{B}_i$ *par des* n_{i-1}*-compositions faibles de* m_i*, où chaque élément de la composition faible correspond à un* x_k*.*

Par exemple : soit un ensemble $\mathbb{B}_{i-1} = e_1, e_2, e_3$ *(donc* $n_{i-1} = 3$*) et* $m_i = 12$*. Soit* $b_{ij} \in \mathcal{R}(\mathbb{B}_{i-1}, m_i)$ *tel que* $b_{ij} = \{e_0, e_0, e_1, e_2, e_2\}$*. On peut aussi représenter* b_{ij} *par la* n_{i-1}*-composition faible de* m_i *suivante :* $b_{ij} = (2, 1, 2)$*. En effet, il y a* 2 *occurrences de* e_0*,* 1 *occurrence de* e_1 *et* 2 *occurrences de* e_2*.*

Exemple 2.2.4 *La figure 2.6 illustre* $\left(\!\!\binom{2}{3}\!\!\right) = 4$ *Rubans composés de* 2 *blocs élémentaires* $\{e_1, e_2\}$ *où chaque élément est distinct à l'autre* $e_1 \neq e_2$*. Dans chaque Ruban, il y a 3 places pour combiner les blocs élémentaires en parallèle.*

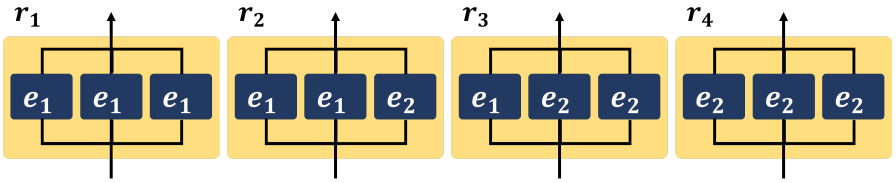


Figure 2.6 – Exemple de Rubans à 3 emplacements et 2 blocs élémentaires.

La définition 2.2.4 satisfait les quatre conditions de la définition 2.2.1. Toutefois, tous les rubans créés ont des cardinalités m_i fixes. Contrairement au pipeline, l'utilisation de bloc SKIP est possible pour créer des rubans avec des cardinalités inférieure ou égale de m_i en respectant toutes les conditions de la définition 2.2.1 sauf la première condition où l'utilisation de ce bloc génère un ruban avec un ensemble vide, noté $r_0 = \{SKIP\}$. Afin de satisfaire la première condition, nous pouvons simplement supprimer ce ruban de l'ensemble des rubans générés avec l'application $\mathcal{R}$ donnée par la définition 2.2.4. Ainsi, la définition ci-dessous donne l'ensemble des rubans avec des cardinalités dynamiques avec l'utilisation de bloc SKIP :

Définition 2.2.5 *Soit $\mathbb{B}_{i-1}$ un ensemble de cardinalité n_{i-1}. On peut former un nouvel ensemble $\mathbb{B}_i$ qui contient tous les rubans possibles avec des cardinalités dynamiques qui varient entre 1 et m_i en ajoutant un bloc SKIP dans l'ensemble $\mathbb{B}_{i-1}$. Par conséquent, l'application $\mathcal{R}$ de la définition 2.2.4 devient :*

$$\mathbb{B}_i = \mathcal{R}\left(\mathbb{B}_{i-1} \cup \{SKIP\}, m_i\right) \setminus \{r_0\} \tag{2.17}$$

Avec $r_0 = \{SKIP\}$

Ainsi, le cardinal $n_i = |\mathbb{B}_i|$ de l'ensemble $\mathbb{B}_i$ devient

$$n_i = \left(\!\!\binom{n_{i-1}+1}{m_i}\!\!\right) - 1 = \binom{n_{i-1}+m_i}{m_i} - 1 = \frac{(n_{i-1}+m_i)!}{m_i! n_{i-1}!} - 1 \tag{2.18}$$

Exemple 2.2.5 *Le tableau 2.4 illustre $10 = \left(\!\!\binom{3}{3}\!\!\right)$ rubans composés de 2 blocs élémentaires $\{e_1, e_2\}$ et un bloc SKIP $\{e_0\}$. Dans chaque ruban, il y a 3 places max pour allouer les blocs élémentaires avec le SKIP en utilisant la définition 2.2.5. Par conséquent, l'ensemble $\mathbb{B}_i$ est l'ensemble de tous les rubans créés sauf le ruban r_0 qui ne contient que les blocs SKIP (qui a aucun traitement sur l'entrée du ruban). Ainsi, $\mathbb{B}_i = \{r_1, r_2, \cdots, r_9\}$.*

Table 2.4 – Exemple de rubans avec 3 emplacements pour ordonner 2 blocs élémentaires et 1 bloc SKIP.

l_i \ r_i	r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9
l_1	–	e_1	e_2	e_1	e_1	e_2	e_1	e_1	e_1	e_2
l_2	–	–	–	e_1	e_2	e_2	e_1	e_1	e_2	e_2
l_3	–	–	–	–	–	–	e_1	e_2	e_2	e_2

Dans le tableau 2.4, on peut clairement observer que l'ensemble $\mathbb{B}_i$ est constitué aussi de 2 rubans $\{r_1, r_2\}$ de cardinalité 1 et de 3 rubans $\{r_3, r_4, r_5\}$ de cardinalité 2 et le dernier de 4 rubans $\{r_6, r_7, r_8, r_9\}$ de cardinalité 3. Ainsi, l'utilisation de l'application $\mathcal{R}$ de la définition 2.2.4 est possible également pour construire des rubans avec des cardinalités dynamiques en utilisant le deuxième argument de l'application $\mathcal{R}$. La définition suivante donne l'ensemble des rubans avec des cardinalités dynamiques sans l'utilisation de bloc SKIP :

Définition 2.2.6 *Soit $\mathbb{B}_{i-1}$ un ensemble de cardinalité n_{i-1} et $\mathcal{R}$ une application donnée par la définition 2.2.4, il est possible de former un nouvel ensemble $\mathbb{B}_i$ qui contient tous les rubans possibles avec des cardinalités dynamiques qui varient entre 1 et m_i suivant l'équation suivante :*

$$\mathbb{B}_i = \mathcal{R}\left(\mathbb{B}_{i-1}, 1 : m_i\right) = \bigcup_{k=1}^{m_i} \mathcal{R}\left(\mathbb{B}_{i-1}, k\right) \tag{2.19}$$

Ainsi, le cardinal $n_i = |\mathbb{B}_i|$ de l'ensemble $\mathbb{B}_i$ devient

$$n_i = \sum_{k=1}^{m_i} \left(\!\!\binom{n_{i-1}}{k}\!\!\right) \tag{2.20}$$

Exemple 2.2.6 *Le tableau 2.5 illustre $9 = \left(\!\!\binom{2}{1}\!\!\right) + \left(\!\!\binom{2}{2}\!\!\right) + \left(\!\!\binom{2}{3}\!\!\right) = 2+3+4$ rubans composés de 2 blocs élémentaires $\{e_1, e_2\}$. Dans chaque ruban, il y a 3 places max pour allouer les blocs élémentaires en utilisant la définition 2.2.6.*

Table 2.5 – Exemple de rubans avec 3 emplacements pour ordonner 2 blocs élémentaires en utilisant la définition 2.2.6.

l_i \ r_i	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9
l_1	e_1	e_2	e_1	e_1	e_2	e_1	e_1	e_1	e_2
l_2	$-$	$-$	e_1	e_2	e_2	e_1	e_1	e_2	e_2
l_3	$-$	$-$	$-$	$-$	$-$	e_1	e_2	e_2	e_2

Dans le tableau 2.5, on peut clairement observer que l'ensemble des rubans créés par la définition 2.2.6 respecte parfaitement toutes les conditions de la définition 2.2.1. De plus, la cardinalité donnée par la définition 2.2.6 équation 2.20 est égale à celle donnée par l'équation 2.18 de la définition 2.2.5. La théorème ci-dessous montre cette égalité pour tout n et m. ou n est le nombre de blocs utilisé pour créer les nouveaux blocs composés et m représente le nombre d'emplacements dans un ruban.

Théorème 2.2.2 *Pour tout $m \in \mathbb{N}$ et $n \in \mathbb{N}^*$*

$$\sum_{k=0}^{m} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m}\!\!\right) \tag{2.21}$$

Preuve 2.2.2 *Pour démontrer l'identité (eq. 2.21), nous pouvons utiliser une preuve par récurrence sur m. Soit $P(m)$ la proposition*

$$\sum_{k=0}^{m} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m}\!\!\right)$$

Cas de base : *Lorsque $m = 0$, la somme à gauche est simplement $\left(\!\!\binom{n}{0}\!\!\right) = 1$ (car il y a une seule façon de choisir 0 éléments parmi n, même avec répétition). De l'autre côté, $\left(\!\!\binom{n+1}{0}\!\!\right) = 1$, donc $P(0)$ est vrai.*

Hypothèse de récurrence : *Supposons que $P(m)$ est vrai pour un certain m non négatif, c'est-à-dire que*

$$\sum_{k=0}^{m} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m}\!\!\right) \tag{2.22}$$

Étape de récurrence : *Nous devons montrer que $P(m+1)$ est vrai, c'est-à-dire*

$$\sum_{k=0}^{m+1} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m+1}\!\!\right) \tag{2.23}$$

Nous remarquons que l'équation (2.23)peut se reformuler comme suit :

$$\sum_{k=0}^{m+1} \left(\!\!\binom{n}{k}\!\!\right) = \sum_{k=0}^{m} \left(\!\!\binom{n}{k}\!\!\right) + \left(\!\!\binom{n}{m+1}\!\!\right) \tag{2.24}$$

Par l'hypothèse de récurrence (2.22), l'équation (2.24) devient :

$$\sum_{k=0}^{m+1} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m}\!\!\right) + \left(\!\!\binom{n}{m+1}\!\!\right)$$

Par la propriété des coefficients binomiaux (formule de Pascal) :

$$\left(\!\!\binom{n+1}{m+1}\!\!\right) = \binom{n+m+1}{m+1} = \binom{n+m}{m} + \binom{n+m}{m+1}$$

En remplaçant les multichoisit par les coefficients binomiaux, nous obtenons

$$\begin{aligned}\left(\!\!\binom{n+1}{m}\!\!\right) + \left(\!\!\binom{n}{m+1}\!\!\right) &= \binom{n+m}{m} + \binom{n+m}{m+1} \\ &= \binom{n+m+1}{m+1} \\ &= \left(\!\!\binom{n+1}{m+1}\!\!\right)\end{aligned}$$

Donc, la proposition $P(m+1)$ est vraie.

Par récurrence, nous pouvons donc conclure que l'identité $\sum_{k=0}^{m} \left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n+1}{m}\!\!\right)$ est vraie pour tout entier non négatif m.

Ainsi, les blocs composés sont formés à partir de blocs élémentaires en « pipelines » (en série) ou en « rubans » (en parallèle). Pour former un réseau de neurones, il est possible de continuer ce principe de composition en formant un nouveau niveau de blocs composés à partir d'autres blocs composés. La section suivante présente cette composition sur plusieurs niveaux.

2.2.3 . Architecture neuronale à base des blocs

Il est possible de construire une architecture neuronale en combinant plusieurs blocs imbriqués les uns après les autres. Ces blocs imbriqués peuvent être représentés par des niveaux (figure 2.7). Le premier niveau est composé de blocs créés à partir de blocs élémentaires (section 2.2.1). Dans le deuxième niveau, les blocs sont créés à partir des blocs composés

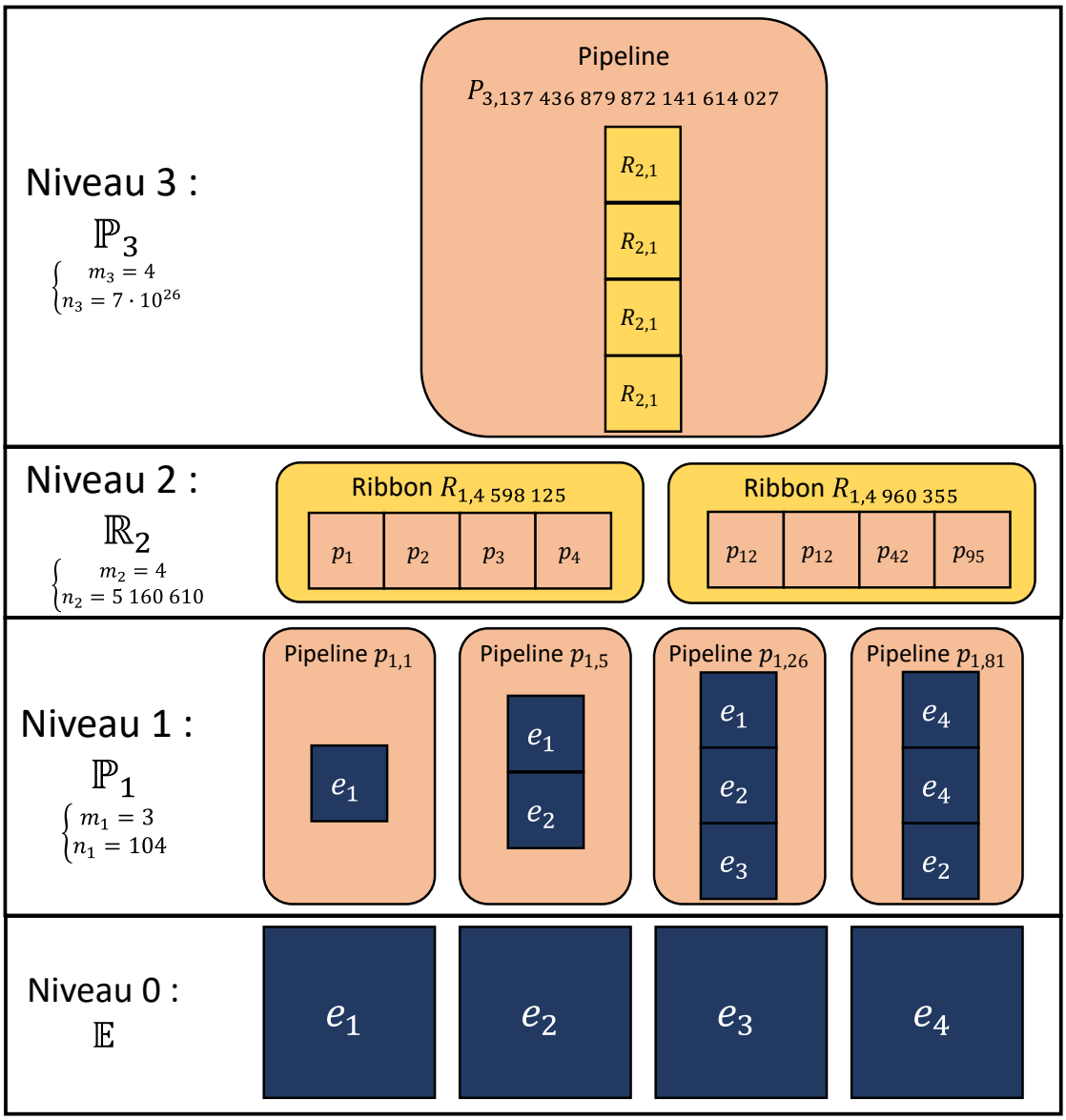


Figure 2.7 – Exemple d'imbrication sur 3 niveaux.

du premier niveau. Ainsi, chaque niveau utilise les blocs du niveau inférieur. De plus, pour chaque niveau, nous avons le choix entre une composition en série avec des pipelines (section 2.2.2) ou une composition en parallèle avec des rubans (section 2.2.2).

En attribuant le type de composition q_i à chaque niveau i, où $q_i = 1$ représente une composition en série avec des pipelines et $q_i = 0$ représente une composition en parallèle avec des rubans, nous pouvons définir l'ensemble $\mathbb{Q} = \{q_1, q_2, \cdots, q_L\}$ qui représente le type de compositions de l'architecture pour L niveaux, où le dernier niveau est un pipeline ($q_L = 1$). Ainsi, pour chaque niveau i, nous avons un ensemble de blocs composés $\mathbb{B}_i$ avec une cardinalité n_i calculée soit en utilisant la cardinalité du pipeline (voir équation 2.5) ou celle du ruban (voir équation 2.20). On peut donc calculer n_i avec la formule suivante :

$$n_i = q_i \left(\sum_{k=1}^{m_i} n_{i-1}^k \right) + (1 - q_i) \left(\sum_{k=1}^{m_i} \left(\!\!\binom{n_{i-1}}{k}\!\!\right) \right) \tag{2.25}$$

avec n_{i-1} le cardinal de l'ensemble de niveau inférieur $\mathbb{B}_{i-1}$. m_i représente le nombre d'emplacements d'un bloc composé de $\mathbb{B}_i$ dans lesquels on peut allouer des blocs de $\mathbb{B}_{i-1}$, avec $\mathbb{B}_0 = \mathbb{E}$ (section 2.2.1).

Afin de déterminer la cardinalité du dernier niveau, c'est-à-dire le nombre d'architectures possibles, en fonction du nombre de blocs élémentaires, de la liste des types de composition $\mathbb{Q}$ et de la liste des nombres d'emplacements possibles $\mathbb{M} = \{m_1, m_2, \cdots, m_L\}$, nous utilisons de manière récursive l'équation 2.25. L'algorithme 2.1 illustre comment cette équation est appliquée pour calculer le nombre d'architectures possibles qui représente notre espace de recherche pour trouver les meilleures architectures neuronales.

Algorithme 2.1 : Nombre d'architectures possibles

Données : n_0, L, $\mathbb{M}$ et $\mathbb{Q}$ respectivement sont nombre de blocs élémentaires, nombre de niveau, ensemble d'emplacement et ensemble de type de composition

Résultat : n_L nombre d'architectures possibles

pour $i = 1, \cdots, L$ **faire**
  $n_i \leftarrow$ utiliser l'équation 2.25 avec $m_i \in \mathbb{M}$ et $q_i \in \mathbb{Q}$
fin
retourner n_L

Dans un ensemble d'architectures $\mathbb{A}$, dont la taille est déterminée par l'algorithme 2.1, nous pouvons calculer les dimensions du tenseur de sortie $(h_{\mathbf{Y}}, w_{\mathbf{Y}}, c_{\mathbf{Y}})$ pour chaque architec-

ture $a_k \in \mathbb{A}$ en utilisant les équations suivantes :

$$\begin{aligned} h_{\mathbf{Y}} &= \frac{h_{\mathbf{X}}}{2^{p_{L,k}}} \\ w_{\mathbf{Y}} &= \frac{w_{\mathbf{X}}}{2^{p_{L,k}}} \\ c_{\mathbf{Y}} &= 2^{r_{L,k}} c_{\mathbf{X}} \end{aligned} \tag{2.26}$$

avec

$$\begin{aligned} \forall i \in \{L, \ldots, 2, 1\} : p_{i,k} &= q_i \sum_{j \in \mathbb{M}_{i,k}} p_{i-1,j} + (1 - q_i) \max_{j \in \mathbb{M}_{i,k}} (p_{i-1,j}) \\ \forall i \in \{L, \ldots, 2, 1\} : r_{i,k} &= q_i \sum_{j \in \mathbb{M}_{i,k}} r_{i-1,j} + (1 - q_i) \max_{j \in \mathbb{M}_{i,k}} (r_{i-1,j}) \end{aligned} \tag{2.27}$$

L'ensemble $\mathbb{M}i, k$ représente les blocs internes du niveau inférieur $i - 1$ qui sont utilisés pour construire le k-ième bloc du niveau supérieur i. Le tenseur d'entrée est représenté sous la forme d'un triplet $(h_{\mathbf{X}}, w_{\mathbf{X}}, c_{\mathbf{X}})$. Le paramètre $p_{i,k}$ indique le nombre de cartes de fonction créées par le k-ième bloc du niveau i, tandis que le paramètre $r_{i,k}$ donne la taille du tenseur réduite par le k-ième bloc du niveau i.

2.2.4 . Exemples d'imbrications recréant des CNN connus

Pour montrer la versatilité du modèle d'espace de recherche présenté dans les sections précédentes, nous présentons ici des exemples d'espaces de recherches capable de trouver les CNN de la section 1.1.4. Tout d'abords le cas simple de VGG, puis celui plus complexe de GoogLeNet, tout deux en utilisant des blocs élémentaires simples. Enfin, nous présenterons ResNet avec des blocs élémentaires plus élaboré.

VGG

L'architecture de VGG est plutôt simple (voir section 1.1.4), ne nécessitant que des pipelines. De ce fait, avec uniquement un bloc élémentaire constitué d'une convolution avec un noyau de 3×3, et un *max-pooling* en options, il est possible de trouver l'architecture VGG. Pour cela, il faut que l'ensemble $\mathbb{P}_1$ des pipelines de niveau 1 est une multiplicité $m_1 \geq 3$. Ensuite l'ensemble $\mathbb{P}_2$ des pipelines de niveau 2 doit avoir une multiplicité $m_2 \geq 5$.

GoogLeNet

L'architecture GoogLeNet 1.1.4 est plus complexe, avec des liaisons en parallèle et nécessite donc des rubans. Pour pouvoir trouver cette architecture, les blocs élémentaires sont les suivant :

— e_1 =conv3×3 [1]

— e_2 =conv5×5

1. conv$W \times H$ est opération de convolution dont le noyau est de W par H.

— e_3 =conv7 $\times$ 7

— e_4 =max-pool

Pour pouvoir trouver un bloc correspondant à l'*Inception* (voir figure 1.13), les pipelines de $\mathbb{P}_1$ doivent avoir une multiplicité de $m_1 \geq 2$ et les rubans de $\mathbb{R}_2$ doivent avoir une multiplicité de $m_2 \geq 4$ avec une concaténation comme opération de fusion. Ensuite pour obtenir la bonne profondeur d'architecture, il faut que les pipelines appartenant à $\mathbb{P}_3$ aient une multiplicité de $m_3 \geq 11$.

Il est à noter qu'avec notre formalisation il n'est pas possible d'obtenir les mêmes nombre de filtres sur toutes les opérations de convolution. En effet, ces nombres ne suivant pas de logique (voir table 1.1), il n'est pas possible de configurer adéquatement les blocs élémentaires pour multiplier le nombre filtres de façon telle qu'il corresponde exactement à GoogLeNet.

ResNet-50

Pour montrer un exemple d'une orientation de l'espace de recherche vers une architecture particulière, nous présentons ici une espace de recherche non seulement capable de trouver ResNet-50 mais qui ne recherche que des architectures similaire. L'intérêt de restreindre ainsi la recherche est de réduire le temps d'optimisation (en réduisant les degrés de liberté).

Pour cela, nous utilisons les blocs « CONV » et « RES » présentés dans la figure 1.14 et 2.8 comme blocs élémentaires. En utilisant ces blocs élémentaires, il existe un pipeline de $\mathbb{P}_1$ avec $m_1 \geq 17$ correspondant à ResNet-50.

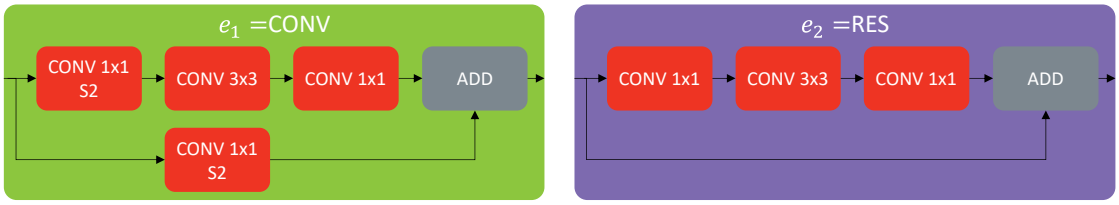


Figure 2.8 – Blocs CONV et RES de ResNet-50 utilisés comme blocs élémentaires.

2.3 . Représentation d'une architecture par des entiers

Avec la section précédente, nous avons vu le principe d'imbrication permettant de créer des architectures à partir de blocs élémentaires. Dans cette section, nous utilisons la notion de familles indexées pour encoder et décoder un bloc composé par des entiers. Afin de comprendre les deux mécanismes d'encodage et de décodage, nous allons d'abord donner la définition de famille indexée pour un ensemble de blocs.

Définition 2.3.1 *Soit un ensemble de blocs composés* $\mathbb{B}_i = \{b_{i0}, b_{i1}, \cdots, b_{ij}, \cdots, b_{in_i-1}\}$ *avec une cardinalité* n_i*. On dit que l'ensemble* $\mathbb{B}_i$ *est une « famille indexée » (ou tout simplement « famille ») lorsqu'il existe un ensemble index* $\mathbb{I}_i = \{0, 1, \cdots, n_i - 1\}$ *correspondant*

à cette famille tel que $\forall j \in \mathbb{I}_i, (b_{ij})_{j\in\mathbb{I}_i} = \mathbb{B}_i$. *Autrement dit, il existe une application* $\mathcal{A}$ *permettant de retrouver un élément* $b_{ij} \in \mathbb{B}_i$ *à partir de son index* $j \in \mathbb{I}_i$ *:*

$$\begin{aligned}\mathcal{A} : \mathbb{I}_i &\to \mathbb{B}_i \\ j &\to b_{i,j} = \mathcal{A}(j)\end{aligned}$$

et son application inverse $\mathcal{A}^{-1}$ *permet de retrouver l'index de l'élément* $b_{ij} \in \mathbb{B}_i$ *:*

$$\begin{aligned}\mathcal{A}^{-1} : \mathbb{B}_i &\to \mathbb{I}_i \\ b_{i,j} &\to j = \mathcal{A}^{-1}(b_{i,j})\end{aligned}$$

En utilisant la définition de famille indexées 2.3.1, la définition ci-dessous donne une compréhension de la notion d'encodage et de décodage qui sera utilisée plus tard pour créer un blocs composé (ou une architecture) par des entiers.

Définition 2.3.2 *Soit un bloc composé* $b_{ij} \in \mathbb{B}_i$*, on dit que l'application* $\mathcal{F}$ *peut encoder le bloc composé* $b_{i,j}$ *par son index* $j = \mathcal{A}^{-1}(b_{i,j})$ *en utilisant les indices de blocs appartenant à* $\mathbb{B}_{ij}$ *:*

$$\begin{aligned}\mathcal{F} : \mathbb{I}_{i-1} &\to \mathbb{I}_i \\ \mathbb{I}_{b_{ij}} &\to j = \mathcal{F}(b_{ij})\end{aligned}$$

avec $\mathbb{I}_{b_{ij}} = \mathcal{A}^{-1}(b_{ij})$*. L'application décodage est l'inverse de l'application encodage nommée* $\mathcal{F}^{-1}$ *:*

$$\begin{aligned}\mathcal{F}^{-1} : \mathbb{I}_i &\to \mathbb{I}_{i-1} \\ j &\to \mathbb{I}_{b_{ij}} = \mathcal{F}^{-1}(j)\end{aligned}$$

2.3.1 . Encodage et décodage des pipelines

Dans la section 2.2.2, nous avons vu que la création d'un ensemble de pipelines à partir d'un ensemble de blocs donnés nécessite l'utilisation du produit cartésien. En utilisant ce dernier avec l'ensemble d'index, nous pouvons encoder et décoder un pipeline en utilisant la méthode de conversion de bases. Dans le cas de l'encodage, on cherche à passer de la base $b \in \mathbb{N}$ (quelconque) à la base 10 (décimal). La formule suivante est utilisée pour réaliser la conversion de b à 10 :

$$(x_1, x_2, \cdots, x_m)_b = \left(\sum_{i=1}^{m} x_i * b^{m-i}\right)_{10} \tag{2.28}$$

On note que le terme de gauche représente un nombre en base b écrit sous la forme d'un m-uplet dans lequel les chiffres x_j peuvent prendre des valeurs de l'ensemble $\{0, 1, \cdots, b-1\}$.

Algorithme 2.2 : Conversion d'un m-uplet en décimal

Données : X est m-uplet et b est le base de X
Résultat : x nombre de base 10

```
m ← longeur(X)
x ← 0
pour i = 1, ··· , m faire
    x ← x + X[i] * b^(m−i)
fin
```

Le terme de droite représente un nombre décimal appartenant à l'ensemble $\{0, 1, \cdots, b^m - 1\}$. L'algorithme ci-dessous implémente l'équation (2.28).

Dans le cas du décodage, nous effectuons une série de divisions par la base b à m reprises. À chaque division, les chiffres du m-uplet sont déterminés en utilisant le reste de la division, comme illustré dans l'équation suivante pour une conversion de 10 à b :

$$(x)_{10} = \Big(\frac{x}{b^{m-1}} \bmod b, \frac{x}{b^{m-2}} \bmod b, \cdots, \frac{x}{b^0} \bmod b\Big)_b \tag{2.29}$$

avec mod représentant le Modulo qui donne le reste de la division. L'algorithme ci-dessous implémente l'équation (2.29).

Algorithme 2.3 : Conversion d'un décimal en m-uplet

Données : x est un nombre décimal, b est la base dans laquelle convertir x, et m est la taille de l'uplet en sortie.
Résultat : X, un m-uplet de base b.

```
X ← [0]
pour i = 1, ··· , m faire
    X[i] ← x mod b
    x ← ⌊x/b⌋
fin
```

Exemple 2.3.1 *Prenons par exemple, un ensemble de blocs élémentaires $\mathbb{B}_0 = \{e_1, e_2\}$ et son ensemble index $\mathbb{I}_0 = \{0, 1\}$ avec un cardinal $n_0 = 2$. En utilisant la définition 2.2.2 avec $m_1 = 3$ places (3-uplet) pour ordonner les deux blocs élémentaires dans un pipeline, on aura un ensemble de pipelines $\mathbb{B}_1 = \{p_1, p_2, \cdots, p_{n_1}\}$ avec une cardinalité $n_1 = n_0^{m_1} = 2^3 = 8$ et son ensemble d'index $\mathbb{I}_i = \{0, 1, \cdots, n_1 - 1\}$.*

Le tableau 2.6 illustre le codage du pipeline de la gauche vers la droite et son décodage de la droite vers la gauche en utilisant respectivement l'équation (2.28) et (2.29).

L'exemple 2.3.1 illustre l'utilisation de la conversion de bases lorsque l'on applique la définition 2.2.2 pour créer des pipelines avec une cardinalité fixe. Dans les sections suivantes,

Table 2.6 – Exemple de conversion de bases pour un pipeline à 3 places pour ordonner 2 blocs.

(3-uplet) de $\mathbb{B}_0$	(3-uplet) de $\mathbb{I}_0$ de base 2	nombre décimal de $\mathbb{I}_1$	pipeline p_j de $\mathbb{B}_1$
(e_1, e_1, e_1)	$(0, 0, 0)$	0	p_1
(e_1, e_1, e_2)	$(0, 0, 1)$	1	p_2
(e_1, e_2, e_1)	$(0, 1, 0)$	2	p_3
(e_1, e_2, e_2)	$(0, 1, 1)$	3	p_4
(e_2, e_1, e_1)	$(1, 0, 0)$	4	p_5
(e_2, e_1, e_2)	$(1, 0, 1)$	5	p_6
(e_2, e_2, e_1)	$(1, 1, 0)$	6	p_7
(e_2, e_2, e_2)	$(1, 1, 1)$	7	p_8

nous explorerons comment utiliser la même convention de bases pour coder et décoder des pipelines ayant des cardinalités dynamiques, telles que définies dans la définition 2.2.3.

Encodage des pipelines

Définition 2.2.3 (Rappel) *Soit $\mathbb{B}_{i-1}$ un ensemble de cardinalité n_{i-1}. On peut former un nouvel ensemble $\mathbb{B}_i$ qui contient tous les pipelines possibles avec des cardinalités dynamiques qui varient entre 1 et m_i en utilisant l'équation 2.1 de la définition 2.2.2 :*

$$\mathcal{P}\left(\mathbb{B}_{i-1}, 1 : m_i\right) = \bigcup_{k=1}^{m_i} \mathcal{P}\left(\mathbb{B}_{i-1}, k\right) = \bigcup_{k=1}^{m_i} \mathbb{B}_{i-1}^{k} \tag{2.30}$$

Ainsi, le cardinal n_i de l'ensemble $\mathbb{B}_i$ devient

$$n_i = \sum_{k=1}^{m_i} n_{i-1}^{k} = \begin{cases} \frac{n_{i-1}\left(n_{i-1}^{m_i}-1\right)}{n_{i-1}-1} & \textit{si } n_{i-1} > 1, \\ m_i & \textit{si } n_{i-1} = 1. \end{cases} \tag{2.31}$$

Selon la définition 2.2.3, un ensemble de pipelines $\mathbb{B}_i$ avec des cardinalités dynamiques $\{1, 2, \ldots, m_i\}$ peut être créé à partir d'un ensemble de blocs $\mathbb{B}_{i-1}$ composé de n_{i-1} éléments. Cela peut être réalisé en utilisant l'équation (2.4), rappelée ci-dessous :

$$\mathbb{B}_i = \bigcup_{k=1}^{m_i} \mathbb{B}_{i-1}^{k} \tag{2.32}$$

En se référant à la définition 2.3.1 de l'ensemble d'index, l'équation précédente peut être réécrite de la manière suivante :

$$\mathbb{I}_i = \left\{0, 1, \cdots, \sum_{k=1}^{m_i} n_{i-1}^k - 1\right\} = \bigcup_{k=1}^{m_i} \left(\left\{0, 1, \cdots, n_{i-1}^k - 1\right\} \oplus \left\{\sum_{l=1}^{k} n_{i-1}^l - n_{i-1}^k\right\}\right) \tag{2.33}$$

avec $\oplus$ représentant l'addition élément par élément.

À partir de l'équation (2.33), nous pouvons déduire la relation suivante :

$$\forall y \in \mathbb{I}_i, \exists x \in \left\{0, \cdots, n_{i-1}^k - 1\right\} \text{ et } k \in \{1, \cdots, m_i\} : y = x + \sum_{l=1}^{k} n_{i-1}^l - n_{i-1}^k \tag{2.34}$$

Dans l'équation (2.34), la valeur de x peut être calculée en convertissant un k-uplet de base n_{i-1} en décimal. Par conséquent, l'algorithme 2.4 illustre l'encodage d'un pipeline en entier à partir de leur représentation en k-uplets, où k varie de 1 à m_i et n_{i-1} est la base de ce k-uplet.

Algorithme 2.4 : Encodage d'un pipeline

Données : X est k-uplet et b est le base de X.
Résultat : y nombre de base 10.

1 $k \leftarrow$ longueur(X)
2 $x \leftarrow$ résultat de la conversion de X en décimal en utilisant l'algorithme 2.2
3 **si** *b>1* **alors**
4 $\quad y \leftarrow x + \frac{b^k - b}{b-1}$
5 **fin**
6 **sinon**
7 $\quad y \leftarrow x + k - 1$
8 **fin**

On note que les lignes 4 et 7 dans l'algorithme sont déduites à partir de l'équation (2.5). Dans la section suivante, nous aborderons l'opération inverse qui consiste à décoder un pipeline, c'est-à-dire à identifier les blocs qui composent ce pipeline.

Décodage des pipelines

Dans la section précédente, nous avons utilisé l'équation (2.34) pour encoder un pipeline en utilisant ses blocs constitutifs. À présent, nous allons utiliser la même équation pour décoder un pipeline, c'est-à-dire retrouver ses blocs à partir d'un entier. Ainsi, l'équation (2.34) devient :

$$\forall x \in \left\{0, \cdots, n_{i-1}^k - 1\right\} \text{ et } k \in \{1, \cdots, m_i\}, \exists y \in \mathbb{I}_i : x = y - \sum_{l=1}^{k} n_{i-1}^l - n_{i-1}^k \tag{2.35}$$

Dans l'équation 2.35, la valeur x peut être convertie en un k-uplet en utilisant l'algorithme 2.3. y représente le pipeline à décoder. Afin de trouver k, la taille de l'uplet (ou le nombre de blocs) utilisés pour coder le pipeline, nous pouvons nous baser sur le théorème suivant.

Théorème 2.3.1 *Pour un nombre décimal y appartenant à un ensemble d'index $\{0, 1, \cdots, \sum_{i=1}^{m} b^i\}$ avec m et $b \in \mathbb{N}^*$, on peut dire que y représente un k-uplet de base b, où*

$$k = \begin{cases} \lfloor log_b\,(b + y(b-1)) \rfloor & \textit{si } b > 1, \\ y + 1 & \textit{si } b = 1. \end{cases} \tag{2.36}$$

Preuve 2.3.1 *Pour $y \in \{0, 1, \cdots, \sum_{i=1}^{m} b^i\}$ et m et $b \in \mathbb{N}^*$. Il existe une valeur entière $k \in \{1, 2, \cdots, m\}$ pouvant borner la valeur y suivant l'équation suivante :*

$$\sum_{i=1}^{k} b^i - b^k \leq y < \sum_{i=1}^{k} b^i \tag{2.37}$$

En considérant deux cas $b = 1$ et $b > 1$, on aura la démonstration suivante :

***Cas où** $b = 1$: l'équation (2.37) devient*

$$k - 1 \leq y < k$$

Et par conséquent,

$$k = y + 1 \tag{2.38}$$

***Cas où** $b > 1$: nous pouvons réécrire l'équation (2.37) en utilisant la formule d'une suite géométrique, ce qui nous donne :*

$$\begin{aligned} eq.(2.37) \Leftrightarrow \quad & \frac{b^{k+1} - b}{b-1} - b^k \leq y & < \frac{b^{k+1} - b}{b-1} \\ \Leftrightarrow \quad & \frac{b^k - b}{b-1} \leq y & < \frac{b^{k+1} - b}{b-1} \\ \Leftrightarrow \quad & b^k - b \leq y(b-1) & < b^{k+1} - b \\ \Leftrightarrow \quad & b^k \leq b + y(b-1) & < b^{k+1} \end{aligned}$$

On applique un logarithme de base b sur les trois termes, on aura

$$k \leq log_b\,(b + y(b-1)) < k + 1$$

Par conséquent,

$$k = \lfloor log_b\,(b + y(b-1)) \rfloor \tag{2.39}$$

Suivant l'équation (2.35), le théorème 2.3.1 et l'algorithme 2.3, le décodage d'un pipeline se fait par l'algorithme suivant :

Algorithme 2.5 : Décodage d'un pipeline

Données : y est un nombre décimal, b est la base dans laquelle convertir y, et m est la taille de l'uplet max en sortie.

Résultat : X, un k-uplet de base b où $1 \leq k \leq m$.

```
2.2
si b>1 alors
    k ← ⌊log_b(b + y(b − 1))⌋
    x ← y − (b^k − b)/(b − 1)
fin
sinon
    k ← y + 1
    x ← y − k + 1 = 0
fin
k ←résultat de la conversion de x en k-uplet en utilisant l'algorithme 2.3
```

2.3.2 . Décodage et encodage des rubans

Dans la section 2.2.2, nous avons vu qu'un ruban peut être créé à partir de m blocs provenant d'un autre ensemble de blocs dont le cardinal est n. Un ensemble de rubans peut être représenté de deux façon différentes ; soit par des multiensembles (définition 2.2.6), soit par des n-compositions faibles de m (remarque 2.2.1). Par exemple, un ruban r_j créé par 3 blocs de l'ensemble $\{e_1, e_2, e_3, e_4\}$ peut être représenté par un multiensemble e_1, e_2, e_2 ou par une composition faible $(1, 2, 0, 0)$. En utilisant les deux représentations, il n'existe pas de formule, comme c'est le cas pour le changement de base des pipelines, permettant d'encoder ou de décoder directement un ruban. Ainsi il faut passer par un algorithme générant les rubans un à un jusqu'à atteindre le ruban à chercher. Cependant un tel algorithme peut atteindre une complexité élevée.

Dans cette section, contrairement à la précédente, nous commencerons par décrire les algorithmes de décodage. Nous présenterons deux algorithmes de génération respectivement de multiensembles et de n-compositions faibles d'un nombre m tous deux provenant de la littérature, puis un algorithme (algorithme 2.8) développé dans cette thèse accédant plus rapidement à la solution. Par la suite, nous présentons la méthode de création de toutes les solutions en amont et le stockage dans un tableau afin d'y accéder instantanément, mais au prix d'un besoin accru en mémoire. Nous comparerons cette dernière méthode avec l'algorithme 2.8. Enfin, nous décrirons comment faire l'encodage d'un ruban, à partir des algorithmes utilisés pour le décodage.

Décodage des rubans

A. Décodage par génération de multiensembles

La première solution envisagée pour faire correspondre un entier x avec un multiensemble X fut de générer un par un les multiensembles (tels que définis dans l'équation 2.2.4) x fois. Pour ce faire, l'algorithme utilisé est tiré de l'*algorithm 7* développé par Ehrlich dans [23] et implémenté par Garrison [29]. L'implémentation de [29] est retranscrite dans l'algorithme A1.1, donnée en annexe A1. Il s'agit d'un générateur. Ce qui signifie que lorsqu'on appelle ce générateur, l'algorithme ira jusqu'au « **céder** » à la ligne 9, et renverra la valeur de X. Puis lorsque l'on appellera à nouveau le générateur, l'algorithme reprendra jusqu'à rencontrer à nouveau un « **céder** » (ou un « **retourner** » qui mettra fin au générateur).

Ainsi, à l'aide de ce générateur, l'algorithme 2.6 permet de générer un décodage de x sous la forme d'un multiensemble. Pour ce faire, on créé le générateur en ligne 2 avec les paramètres adéquats. Puis, en ligne 4, on appelle le générateur (la fonction « `suivant` » génère la prochaine itération du générateur) dans une boucle x fois. A la fin de la boucle, on a donc dans X le multiensemble correspondant au décodage de x.

Algorithme 2.6 : Décodage par multiensemble

Données : Un nombre x à décoder, un nombre n indiquant la taille des multiensembles, un nombre b représentant la taille de l'ensemble support

Résultat : Un multiensemble X

```
1 objects ← [0, 1, ..., b]
2 gen ← Générateur de multiensembles(n, objects)
3 x_compteur ← 0
4 tant que x_compteur ≤ x faire
5 |   X ← suivant(gen)
6 |   x_compteur ← x_compteur + 1
7 fin
8 retourner X
```

B. Génération de compositions faibles

Cette seconde solution est similaire à la précédente, mais génère des b-compositions faibles de n_1 (donc des multiensembles tels que définis dans la remarque 2.2.1) au lieu de multiensembles définis par leurs éléments (équation 2.2.4). Tout comme l'algorithme A1.1, un générateur est utilisé pour pouvoir générer itérativement des compositions faibles. Pour ce faire, l'algorithme « weakcomps » de Putman [66] est utilisé. Cet algorithme est retranscrit dans l'algorithme A2.1 en annexe A2, sous le nom de « Générateur de compositions faibles »

(à noter que cet algorithme utilise la fonction « `combinations` »[2]).

À noter l'algorithme génère les compositions faibles de façon ascendante. Ce qui veut dire que l'algorithme 2.7 est inversé par rapport à l'algorithme 2.6. Autrement dit le ruban correspondant à `Décodage par multiensemble`(x, n, b) est le même ruban que celui correspondant à `Décodage par composition faible` $\left(\left(\left(\!\binom{b}{n}\!\right)\right) - x, n, b\right)$. Ceci est illustré dans la table 2.7, où à chaque entier en base 10 est associé à son décodage par composition faible ou par multiensemble.

Algorithme 2.7 : Décodage par composition faible

Données : Un nombre x à décoder, un nombre n indiquant la taille des multiensembles, un nombre b représentant la taille de l'ensemble support

Résultat : Un multiensemble X

```
gen ← Générateur de compositions faibles(b, n)
x_compteur ← 0
tant que x_compteur ≤ x faire
    X ← suivant(gen)
    x_compteur ← x_compteur + 1
fin
retourner X
```

C. Génération accélérée de compositions faibles

Les méthodes de décomposition vues précédemment génèrent tous les rubans jusqu'à arriver au bon. De ce fait, plus l'entier à décoder est grand, plus l'algorithme mettra du temps à arriver à la solution. C'est pourquoi dans cette section, une méthode pour accélérer le décodage des rubans est envisagée.

L'idée est toujours de générer des rubans jusqu'à arriver à la solution, cependant des étapes peuvent être sautées pour arriver plus rapidement au résultat. En effet, pour générer toutes les b-compositions faibles de n_1, plutôt que d'utiliser l'algorithme A2.1, il est possible de procéder de la manière suivante : on génère une à une les b-compositions[3] de n_1 et, pour chacune de ces compositions, on génère toutes les permutations distinctes[4] possibles. Ainsi, vu qu'il est possible de connaître le nombre de permutations distinctes (grâce au théorème 2.3.2), on peut déterminer avant de générer ces permutations si la solution se trouve dans ses permutations. Puis, si la solution fait partie de ses permutations, on peut commencer à générer les permutations ; sinon, on passe à la composition suivante.

2. La fonction `combinations` est une fonction du module python *itertools*.
3. Il s'agit ici de compositions normales et non pas « faibles ».
4. Une permutation distincte est en quelques sortes un anagramme, mais dans notre cas avec des nombres plutôt que des lettres.

$\mathbb{I}$ en base 10	$\mathbb{B}_i$ sous forme de composition faible	$\mathbb{B}_i$ sous forme de multiensemble
0	$(0,0,4)$	$(0,0,0,0)$
1	$(0,1,3)$	$(0,0,0,1)$
2	$(0,2,2)$	$(0,0,0,2)$
3	$(0,3,1)$	$(0,0,1,1)$
4	$(0,4,0)$	$(0,0,1,2)$
5	$(1,0,3)$	$(0,0,2,2)$
6	$(1,1,2)$	$(0,1,1,1)$
7	$(1,2,1)$	$(0,1,1,2)$
8	$(1,3,0)$	$(0,1,2,2)$
9	$(2,0,2)$	$(0,2,2,2)$
10	$(2,1,1)$	$(1,1,1,1)$
11	$(2,2,0)$	$(1,1,1,2)$
12	$(3,0,1)$	$(1,1,2,2)$
13	$(3,1,0)$	$(1,2,2,2)$

Table 2.7 – Exemple de correspondance entre des index (en base 10) et des rubans de $\mathcal{R}(\mathbb{B}_i, m_i)$ (avec $m_i = 4$ blocs et $n_{i-1} = 3$ blocs dans $\mathbb{B}_{i-1}$) respectivement sous la forme de composition faible ou de multiensemble. Respectivement tels que donnés par l'algorithme 2.7 et 2.6.

Théorème 2.3.2 *(Tiré du théorème 2.8 de [39]) Soit un b-uplet pouvant être constitué de $k+1$ nombres différents, tels qu'il y ait m_0 occurrences du nombre 0, m_1 occurrences du nombre 1, etc. Alors le nombre de permutations distinctes possible (autrement dit : d'anagrammes possible) de ce b-uplet est le coefficient multinomial :*

$$\binom{b}{m_0, m_1, \ldots, m_k} = \frac{b!}{m_0! m_1! \ \ldots \ m_k!}$$

Exemple 2.3.2 *Le sextuplet (A, B, B, C, C, C) possède **6** éléments, dont **1** A, **2** Bs et **3** Cs. Donc le nombre de répétitions distinctes de ce sextuplet est :*

$$\frac{6!}{1!2!3!} = 60$$

Cette méthode est décrite par l'algorithme 2.8. Cet algorithme utilise l'algorithme `accel_asc` de Kelleher (l'algorithme 5.5 dans [43], donné en annexe A3) pour créer des compositions dans l'ordre ascendant (ligne 2). Puis est calculé si x peut être atteint avec des permutations de la composition actuelle. Si tel est le cas (ligne 5), alors un générateur de permutations est créé à partir de la composition actuelle (ligne 6), et, jusqu'à arriver à x, les permutations sont générée (lignes 7 à 9).

Il est à noter que le décodage des rubans rubans accéléré (en utilisant l'algorithme 2.8), ne génère pas les solutions dans l'ordre ascendant (comme le décodage par composition faible, algorithme 2.7), ni descendant. Cependant cet algorithme est bien déterministe. C'est-à-dire qu'une solution correspondra toujours au même x, et vice-versa.

D. Comparaison des différentes méthodes de décodage des rubans

Comparons tout d'abord la méthode de décodage par composition faible (algorithme 2.7) avec la méthode de décodage accélérée (algorithme 2.8). La figure 2.9 montre la différence de vitesse d'exécution entre notre méthode accélérée et la génération de toutes les compositions faibles. En effet, La figure 2.9b étant à l'échelle logarithmique, les deux algorithmes semblent de complexité plus ou moins exponentielle. Cependant, l'algorithme de décodage accélérée montre une rapidité d'exécution plus rapide d'un facteur d'environ 10.

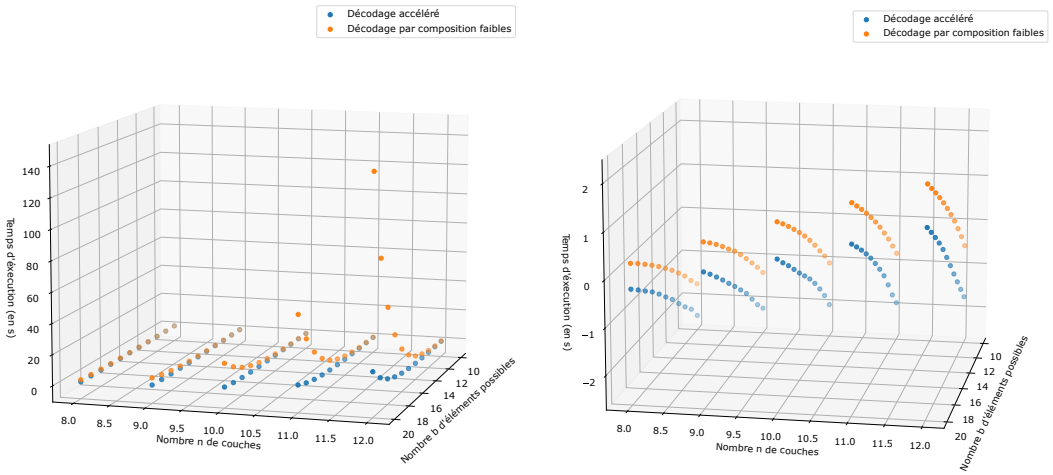
Ainsi il vaut mieux préconiser l'algorithme 2.8 pour faire le décodage direct des rubans. Cependant, pour des nombres élevés, l'algorithme peut commencer à être long. En effet, pour un décodage d'un ruban de $\mathbb{R}_i$ de 12 éléments pris dans un ensemble $\mathbb{C}_{i-1}$ de 30 blocs différents, l'algorithme a eu besoin de presque 2000 secondes (en python avec un processeur i7-8565U). De ce fait, il peut être intéressant de générer à l'avance toutes les possibilités de décodage dans une base de données, et ainsi d'accéder directement aux décodages par le biais de cette dernière. Cela a l'avantage d'être une méthode quasi-instantanée (au temps d'accès à une base de donnée près), mais l'inconvénient de demander beaucoup de mémoire pour stocker cette base de données. Par exemple, pour stocker (en python à l'aide de Pickle) tous les rubans de $\mathbb{R}_i$ de 12 éléments pris dans un ensemble $\mathbb{C}_{i-1}$ de 15 blocs différents il faut déjà 257 Mo.

Algorithme 2.8 : Décodage des rubans

Données : Un nombre x à décoder, un nombre n indiquant la taille des multiensembles, un nombre b représentant la taille de l'ensemble support

Résultat : Un multiensemble X

```
1  x_i ← 0
2  asc ← accel_asc(n)
3  tant que x_i ≤ x faire
4      comp ← suivant(asc)
5      si x_i + n_permutations(comp) ≥ x alors
6          permutation ← distinct_permutation(comp)
7          pour i de 0 à (x − x_i + 1) faire
8              X ← suivant(permutation)
9          fin
10         retourner X
11     fin
12 fin
```



(a) Linéaire

(b) Logarithmique

Figure 2.9 – Temps d'exécutions des algorithmes 2.7 et 2.8 dans le pire des cas pour décoder un ruban de $\mathbb{R}_i$ en fonction du nombre n de couches et du nombre b d'éléments dans $\mathbb{C}_{i-1}$

Encodage des rubans

Pour encoder les rubans, il est possible de modifier légèrement les algorithmes de décodage afin qu'ils effectuent l'encodage. Tout d'abord l'algorithme 2.9 présente comment les algorithmes 2.6 et 2.7 peuvent être adaptés pour faire l'encodage. Pour ce faire, on initialise un générateur de multiensemble ou de compositions faibles selon la nature du X à encoder, et on initialise un x à 0. Puis, on génère des X_{gen} jusqu'à obtenir un X_{gen} égal à X, tout en incrémentant x afin de compter le nombre de générations. Lorsque $X_{gen} = X$, le x correspondant peut être retourner.

Algorithme 2.9 : Encodage de multiensemble ou composition faible

Données : Un multiensemble ou une composition faible X à encoder, un nombre n indiquant la taille des multiensembles, un nombre b représentant la taille de l'ensemble support

Résultat : Un nombre x

```
1  gen ← Générateur de multiensembles(n, [0, 1, ..., b]) ou
     Générateur de compositions faibles(b, n)
2  x ← 0
3  tant que Vrai faire
4  |  X_gen ← suivant(gen)
5  |  si X_gen = X alors
6  |  |  retourner X
7  |  fin
8  |  x ← x + 1
9  fin
10 retourner X
```

Tout comme pour le décodage, il est possible d'utiliser l'algorithme 2.8, plus rapide que les algorithmes précédents. Pour ce faire, comme décrit dans l'algorithme 2.10, on initialise un x à 0 et un générateur de compositions. Pour chaque composition générée, on vérifie si cette-dernière est une permutation de X et, si tel est le cas, on génère des permutations de la composition jusqu'à trouver X et on retourne le x (qui à été incrémenté à chaque génération de permutation). Sinon, x est augmenté du nombre de permutations ignorées ne contenant pas X.

Algorithme 2.10 : Encodage des rubans

Données : Un composition faible X à encoder, un nombre n indiquant la taille des multiensembles, un nombre b représentant la taille de l'ensemble support

Résultat : Un nombre x

```
1  x ← 0
2  asc ← accel_asc(n)
3  tant que Vrai faire
4      comp ← suivant(asc)
5      si comp est une permutation de X alors
6          permutation ← distinct_permutation(comp)
7          tant que Vrai faire
8              si X = suivant(permutation) alors
9                  retourner x
10             fin
11             x ← x + 1
12         fin
13     fin
14     sinon
15         x ← x + n_permutations(comp)
16     fin
17 fin
```

2.4 . Conclusion

Dans ce chapitre, nous avons présenté un espace de recherche basé sur des blocs/cellules [95, 92, 11, 62] pouvant s'imbriquer pour former un vaste panel d'architectures de CNN mais laissant aussi la possibilité d'orienter la recherche vers des architectures spécifiques. De plus cet espace de recherche, grâce à la possibilité transformer des entiers en architecture, et vice-versa, peut facilement contrôler le nombre de variables à optimiser.

Dans la section 2.2, la formalisation de cet espace de recherche est détaillée. La base de l'imbrication (niveau 0) est un ensemble de blocs élémentaires $\mathbb{E}$. Ces blocs élémentaires, définis par l'utilisateur, contiennent les opérations de convolutions. En arrangeant un sous-ensemble de $\mathbb{E}$, on peut former des blocs composés. L'ensemble des arrangements de l'ensemble des sous-ensembles de $\mathbb{E}$ forment l'ensemble des blocs composé de niveau 1 : $\mathbb{B}_1$. Ces blocs composés peuvent eux-mêmes être arrangés pour former l'ensemble $\mathbb{B}_2$ de niveau 2, et ainsi de suite. Il y a deux types d'arrangements : les pipelines (en série) et le rubans (en parallèle).

Tandis que la section 2.3 montre comment encoder une architecture en un entier naturel. Pour cela, plusieurs algorithmes sont nécessaires. En effet, l'encodage d'un pipeline diffère de celui d'un ruban. De plus, une difficulté supplémentaire apparaît pour l'encodage de pipelines dynamiques (définition 2.2.3). Nous avons donc développé une formule permettant l'encodage dynamique (équation 2.5) et une pour le décodage dynamique (équation 2.35). Enfin, il n'existe pas, à notre connaissance, de formule permettant d'encoder et décoder les rubans. La solution envisagée est donc de générer un à un les rubans et de déterminer son encodage en comptant les itérations. Pour accélérer ce processus, nous avons développé l'algorithme 2.8 qui génère d'abord des compostions, puis, lorsque la composition pouvant correspondre au ruban est trouvée, effectue des permutations jusqu'à obtenir le bon ruban.

Dans le chapitre suivant, nous proposons des stratégies d'optimisations qui peuvent être appliquées à cet espace de recherche. Ce chapitre rentre aussi plus en détails concernant l'importance des paramètres de configurations (les options) et comment les appliquer dans l'imbrication. Dans le dernier chapitre, nous appliquerons ces stratégie avec différents algorithmes d'optimisation.

3 - Recherche d'architecture neuronale avec l'espace de recherche imbriqué

Ce chapitre présente différentes stratégies d'optimisations permettant d'utiliser l'espace de recherche imbriqué présenté au chapitre 2 pour rechercher automatiquement une architecture de CNN.

Chaque nouvelle stratégie présentée offre plus de contrôle sur les degrés de liberté. Améliorant le temps d'optimisation au détriment du nombre d'architectures trouvables. La première stratégie est purement méta-heuristique, mais nécessite deux boucles d'optimisation, tandis que la deuxième introduit des algorithmes heuristiques pour n'avoir qu'une seule boucle. Enfin, la dernière stratégie rend l'espace de recherche dynamique pour grandement réduire le nombre de degrés de liberté.

3.1 Introduction . . . 94
3.2 Formulation du problème . . . 95
3.2.1 Espace de recherche . . . 95
Paramètre de l'architecture . . . 95
Paramètre de la configuration de l'architecture . . . 96
3.2.2 Contraintes d'inégalité . . . 97
Contraintes liées à la complexité de l'architecture . . . 97
Contraintes liées au tenseur de sortie . . . 98
3.2.3 Fonctions objectif . . . 98
Fonctions objectif liées à la complexité de l'architecture . . . 99
Fonctions objectif liées à l'entraînement du modèle . . . 100
3.3 Optimisation en double boucle . . . 101
3.4 Optimisation en simple boucle . . . 102
3.4.1 Configuration heuristique d'un Pipeline . . . 105
Distribution séquentielle des options . . . 106
Distribution équitable des options . . . 107
3.4.2 Configuration heuristique d'un Ruban . . . 108
Cas d'une fusion par addition . . . 109
Cas d'une fusion par concaténation . . . 110
3.5 Optimisation en seule boucle avec un espace de recherche dynamique . . . 111
3.5.1 Identifiant unique pour les architectures . . . 112
3.6 Conclusion . . . 114

3.1 . Introduction

Dans le chapitre précédent, nous avons présenté une méthode hiérarchique pour la conception d'architectures neuronales. Chaque architecture est définie par un simple entier naturel, servant d'identifiant pour l'architecture neuronale. Cet identifiant facilite la construction de l'architecture, qui est sous la forme d'un bloc composé, assemblé à partir d'autres blocs composés provenant du niveau inférieur. Ce processus de construction des blocs composés se poursuit en descendant à travers les niveaux jusqu'à atteindre les blocs composés du premier niveau, qui sont créés à l'aide de blocs élémentaires. Pour transformer l'architecture en un modèle prédictif, capable de générer des sorties à partir de tenseurs d'entrée, tous les blocs élémentaires sélectionnés pour former les blocs composés au premier niveau de l'architecture doivent être configurés avec des options contrôlant les dimensions des tenseurs. En conséquence, le processus d'obtention du modèle neural optimal implique deux étapes : d'abord, choisir l'architecture la plus adaptée, puis déterminer la configuration optimale pour les blocs élémentaires constituant cette architecture.

Pour trouver un modèle optimal, comprenant à la fois l'architecture et sa configuration, nous devons explorer un nombre exponentiel d'architectures. Par exemple, prenons le cas de 3 niveaux de type pipeline et 4 blocs élémentaires, ce qui donne environ 2.1583×10^{17} architectures possibles. De plus, pour chaque architecture contenant k blocs élémentaires au niveau 0, un nombre exponentiel de configurations doit être examiné, correspondant à $\prod_{i=1}^{m} p_i^k$, où p_i représente la valeur maximale d'une option et m est le nombre d'options. À titre d'exemple illustratif, considérons $m = 2$, $p_1 = p_2 = 2$, et $k = 20$, ce qui donne 1.0995×10^{12} configurations à évaluer. Ainsi, ces deux problèmes sont considérés comme NP-difficiles, ce qui signifie qu'on ne peut pas espérer trouver la solution optimale dans un laps de temps raisonnable. De plus, il n'existe aucune méthode directe pour déterminer si une architecture avec une configuration donnée est optimale pour un jeu de données spécifique. La seule approche est de recourir aux techniques d'optimisation pour évaluer et choisir le meilleur modèle. Ces techniques incluent des méthodes exactes, des méthodes métaheuristiques et des méthodes heuristiques. Les méthodes exactes garantissent, contrairement aux méthodes heuristiques ou métaheuristiques, de trouver la solution optimale à un problème d'optimisation. Malheureusement, ces méthodes peuvent être coûteuses en temps de calcul, en particulier pour des problèmes de grande taille ou des problèmes NP-difficiles. Dans de tels cas, les approches métaheuristiques ou heuristiques peuvent être préférées, même si elles ne garantissent pas toujours une solution optimale.

Dans ce chapitre, nous proposons trois stratégies d'optimisation. La première stratégie est une optimisation en double boucle, qui est la solution native de notre problème, en se basant sur l'utilisation des algorithmes métaheuristiques. La deuxième stratégie améliore la première en utilisant une simple boucle d'optimisation en intégrant des algorithmes heuristiques pour la configuration. Quant à la troisième, elle améliore la seconde en s'adaptant dynamiquement à notre espace de recherche, qui reste fixe pour les deux premières stratégies.

3.2 . Formulation du problème

Dans cette section, nous verrons tout d'abord la recherche d'architecture automatique d'un point de vue global. En effet, nous cherchons à optimiser l'architecture imbriquée, donc à trouver l'agencement de blocs élémentaires et la configuration des options qui minimisent une ou plusieurs fonctions objectif données (par exemple avoir la plus faible erreur possible sur les données de validation d'un jeu de données). Par conséquent, le problème de recherche d'architecture automatique se formalise comme un problème d'optimisation multi-objectif sous contraintes, décrit par l'équation suivante :

$$\min_{X_{\min} \leq X \leq X_{\max}} \mathcal{F}(X) = \begin{cases} f_1(X) \\ f_2(X) \\ \vdots \\ f_n(X) \end{cases} \tag{3.1}$$
$$\text{sujet aux contraintes}$$
$$g_i(X) \leq 0, i = 1, 2, \ldots, m$$

où $X = (x_1, x_2, \ldots, x_p)$ représente les p variables à optimiser. $X_{\min}$ et $X_{\max}$ désignent les frontières de l'espace de recherche avec $X_{\min} \leq X \leq X_{\max}$. $\mathcal{F}$ est un ensemble de n fonctions objectif f_i à minimiser. Chaque fonction objectif $f_i(X)$ renvoie un critère d'évaluation (voir section 1.2.1) par rapport au paramètre X. Les contraintes d'inégalité $g_i(X) \leq 0$ sont des conditions à satisfaire. Dans la suite, nous allons détailler l'espace de recherche avec ses frontières, les contraintes d'inégalité, ainsi que les fonctions objectif utilisées.

3.2.1 . Espace de recherche

Comme indiqué dans la section 1.2.3 sur les espaces de recherches hiérarchiques, il est aussi intéressant de pouvoir optimiser les réductions de dimensions le long du modèle. Dans notre espace de recherche, ce contrôle des dimensions passe par la configuration des options des blocs élémentaires utilisés pour créer l'architecture. Il y a donc trois paramètres à optimiser que nous définissons tels que le paramètre d'optimisation de l'architecture imbriquée x_a et les paramètres d'optimisation de la configuration des options x_r et x_p (respectivement le nombre de cartes de fonction et la taille des tenseurs). Par conséquent, l'espace de recherche sera le vecteur $X = (x_a, x_r, x_p)$.

Paramètre de l'architecture

Dans la section 2.3, nous avons exploré la méthode de création d'une architecture à l'aide du décodage d'un nombre entier. Ce nombre entier correspond au paramètre d'optimisation de l'architecture imbriquée, qui détermine l'agencement des blocs constituant l'architecture. Nous notons ce paramètre comme étant x_a, et nous pouvons obtenir l'architecture a correspondante en utilisant un algorithme de décodage $\mathcal{A}$, soit $a = \mathcal{A}(x_a)$.

Le paramètre x_a est un nombre entier positif inférieur à une valeur maximum $x_{a_{\max}}$ tel que $0 \leq x_a < x_{a_{\max}}$. Cette valeur maximale $x_{a_{\max}}$ représente le nombre total d'architectures possibles. Pour calculer $x_{a_{\max}}$, nous pouvons utiliser l'algorithme 2.1, en prenant en compte le nombre de blocs élémentaires, le nombre de niveaux, les emplacements disponibles pour les blocs internes à chaque niveau, ainsi que les types de blocs composés utilisés à chaque niveau (pipelines ou rubans). Cette valeur maximale permet de définir la taille de l'espace de recherche des architectures potentielles.

Paramètre de la configuration de l'architecture

Pour chaque architecture créée, nous avons un nombre variable de blocs élémentaires à configurer. Supposons que pour chaque architecture $a_j \in \mathbb{A}$, nous disposons d'un ensemble de blocs élémentaires $\mathcal{E}(a_j) = \left\{b_1, b_2, \ldots, b_i, \ldots, b_{n_j} | \forall i, b_i \in \mathbb{E}\right\}$; avec $\mathcal{E}$ est une application qui retourne la liste des blocs élémentaires utilisée pour créer l'architecture a_j. Chaque bloc élémentaire nécessite deux paramètres de configuration à optimiser : r et p (voir section 2.2.1).

Le paramètre r est utilisé pour multiplier le nombre de cartes de fonction d'entrée $c_{\mathbf{X}}$ par 2^r, ce qui donne le nombre de cartes de fonction de sortie $c_{\mathbf{Y}} = 2^r c_{\mathbf{X}}$.

Le paramètre p est utilisé pour diviser la taille du tenseur d'entrée $(h_{\mathbf{X}}, w_{\mathbf{X}})$ par 2^p, ce qui donne la taille du tenseur de sortie $(h_{\mathbf{Y}}, w_{\mathbf{Y}}) = \left(\lfloor \frac{h_{\mathbf{X}}}{2^p} \rfloor, \lfloor \frac{w_{\mathbf{X}}}{2^p} \rfloor\right)$.

Afin d'optimiser ces paramètres de configuration pour chaque bloc élémentaire, nous utilisons l'encodage et le décodage des pipelines (voir section 2.3.1). Ainsi, l'encodage des paramètres de configuration est donné par les formules suivantes :

$$\begin{aligned} \left(r_1, r_2, \cdots, r_{n_j}\right)_{r_{\max}} &= \left(\sum_{i=1}^{n_j} r_i \times r_{\max}^{n_j - i}\right)_{10} \\ \left(p_1, p_2, \cdots, p_{n_j}\right)_{p_{\max}} &= \left(\sum_{i=1}^{n_j} p_i \times p_{\max}^{n_j - i}\right)_{10} \end{aligned} \tag{3.2}$$

Dans ces équations, le terme de gauche représente un nombre en base $r_{\max}$ (respectivement $p_{\max}$) écrit sous forme d'un n_j-uplet où les options $r_i \in [\![0, r_{\max}[\![$ (respectivement $p_i \in [\![0, p_{\max}[\![$). Le terme de droite représente un nombre décimal $x_r \in [\![0, r_{\max}^{n_j}[\![$ (respectivement $x_p \in [\![0, p_{\max}^{n_j}[\![$).

Pour le décodage, nous utilisons les formules suivantes :

$$\begin{aligned} (x_r)_{10} &= \left(\frac{x_r}{r_{\max}^{n_j-1}} \bmod r_{\max}, \frac{x_r}{r_{\max}^{n_j-2}} \bmod r_{\max}, \ldots, \frac{x_r}{r_{\max}^{0}} \bmod r_{\max}\right)_{r_{\max}} \\ (x_p)_{10} &= \left(\frac{x_p}{p_{\max}^{n_j-1}} \bmod p_{\max}, \frac{x_p}{p_{\max}^{n_j-2}} \bmod p_{\max}, \ldots, \frac{x_p}{p_{\max}^{0}} \bmod p_{\max}\right)_{p_{\max}} \end{aligned} \tag{3.3}$$

Dans ces équations, x_r et x_p représentent l'encodage des paramètres de configuration à optimiser tels que $0 \leq x_r < r_{\max}^{n_j}$ (respectivement $0 \leq x_p < p_{\max}^{n_j}$).

Remarque 3.2.1 *On remarque clairement que les limites de paramètres de configuration sont déterminées par le nombre de blocs élémentaires utilisés (n_j) pour créer une architecture donnée. Par conséquent, l'optimisation des paramètres x_r et x_p est étroitement liée au choix de l'architecture.*

3.2.2 . Contraintes d'inégalité

Toutes les architectures doivent se conformer aux contraintes d'inégalité $g_i(X) \leq 0$, qui sont réparties en deux catégories distinctes. La première catégorie regroupe les contraintes liées à la complexité de l'architecture, tandis que la deuxième catégorie comprend les contraintes liées au tenseur de sortie.

Contraintes liées à la complexité de l'architecture

Dans un système embarqué, les contraintes mémoire et le temps de calcul sont très importants. Afin d'assurer l'exigence demandée par le concepteur pour l'occupation mémoire et le temps de calcul, nous proposons ci-dessous les deux contraintes sur la complexité de l'architecture.

Contrainte sur les paramètres de l'architecture

Il est essentiel que chaque architecture a vérifie que le volume total des paramètres à entraîner $\mathcal{P}(a)$ soit inférieur ou égal au nombre maximal de paramètres $\mathcal{P}_{\max}$ défini par le concepteur. L'application $\mathcal{P}$ retourne le volume total des paramètres à entraîner pour une architecture donnée a. Ainsi, la contrainte d'égalité sur les paramètres à entraîner est donnée par :

$$g_1(X) = \frac{\mathcal{P}(a)}{\mathcal{P}_{\max}} - 1 \leq 0 \tag{3.4}$$

Cette contrainte permet de garantir que l'architecture respecte les contraintes fixées en termes de capacité de stockage des paramètres. En imposant cette contrainte, nous nous assurons que l'architecture ne dépasse pas les limites spécifiées, ce qui peut être crucial pour les systèmes avec des ressources limitées ou pour des applications nécessitant une empreinte mémoire réduite.

Contrainte sur le nombre de FLOPs

De manière similaire, nous pouvons appliquer la contrainte ci-dessous afin de limiter la recherche que sur les architectures qui possèdent un nombre d'opérations (FLOPs) $\mathcal{O}(a)$ inférieur ou égal à celui prédéfini par le concepteur $\mathcal{O}_{\max}$ avec $\mathcal{O}$ est une application qui retourne le nombre d'opérations (FLOPs) requises pour prédire une sortie par une architecture a :

$$g_2(X) = \frac{\mathcal{O}(a)}{\mathcal{O}_{\max}} - 1 \leq 0 \tag{3.5}$$

Cette contrainte permet de contrôler la quantité de calcul nécessaire pour exécuter le modèle, en garantissant une utilisation optimale des ressources de calcul disponibles. En limitant le nombre d'opérations, nous favorisons des architectures plus efficientes en termes de coût de calcul, ce qui peut être crucial dans des environnements à ressources limitées ou pour des applications nécessitant une faible consommation énergétique.

Contraintes liées au tenseur de sortie

Le concepteur peut limiter le tenseur de sortie $\mathbf{Y}$ du modèle entre deux valeurs

$$(h_{\min}, w_{\min}, c_{\min}) \leq (h_{\mathbf{Y}}, w_{\mathbf{Y}}, c_{\mathbf{Y}}) \leq (h_{\max}, w_{\max}, c_{\max}) \tag{3.6}$$

avec la contrainte que

$$(1, 1, 1) \leq (h_{\min}, w_{\min}, c_{\min})$$

Afin de formuler cette inégalité (3.6), nous la décomposons en six contraintes d'inégalité distinctes :

$$\begin{aligned}
g_3(X) &= \frac{h_{\min}}{h_{\mathbf{Y}}} - 1 \leq 0 \\
g_4(X) &= \frac{h_{\mathbf{Y}}}{h_{\max}} - 1 \leq 0 \\
g_5(X) &= \frac{w_{\min}}{w_{\mathbf{Y}}} - 1 \leq 0 \\
g_6(X) &= \frac{w_{\mathbf{Y}}}{w_{\max}} - 1 \leq 0 \\
g_7(X) &= \frac{c_{\min}}{c_{\mathbf{Y}}} - 1 \leq 0 \\
g_8(X) &= \frac{c_{\mathbf{Y}}}{c_{\max}} - 1 \leq 0
\end{aligned} \tag{3.7}$$

Ces six contraintes d'inégalité spécifient les relations entre les dimensions minimales et maximales du tenseur de sortie $\mathbf{Y}$, assurant ainsi que les valeurs sont comprises dans les limites définies par le concepteur. On note ici que lorsque les limites minimale et maximale sont égales, ces contraintes d'inégalité peuvent se transformer en contraintes d'égalité. Cela signifie que les dimensions du tenseur de sortie doivent être exactement égales aux valeurs spécifiées par les limites, sans possibilité de variation.

3.2.3 . Fonctions objectif

Dans cette section, nous proposons un ensemble de fonctions objectif pour mesurer la qualité de l'architecture neuronale (ou modèle). Nous allons classifier les fonctions objectifs en deux catégories "fonctions objectif liées à la création du modèle" et "fonctions objectif liées

à l'entraînement du modèle". Il est à noter que le temps de l'entraînement est plus important que le temps de la création du modèle.

Fonctions objectif liées à la complexité de l'architecture

Après chaque création d'une architecture, nous avons la possibilité de déterminer à la fois le volume total des paramètres à entraîner et le nombre d'opérations (FLOPs) nécessaires pour prédire une sortie. Cette procédure implique l'addition des paramètres (resp. des FLOPs) associés à chaque couche du modèle. Cette évaluation nous offre une vue d'ensemble de la complexité de l'architecture, ce qui nous permet d'évaluer les ressources nécessaires pour la déployer dans un système embarqué.

Dans ce contexte, l'utilisation de fonctions objectif devient pertinente pour minimiser la complexité du modèle. En minimisant le nombre de paramètres et les opérations requises, nous sommes en mesure de réduire la taille du modèle et d'optimiser ses performances sur des dispositifs embarqués où les ressources peuvent être limitées. Cela permet d'assurer une meilleure efficacité énergétique, une exécution plus rapide et une adaptation plus facile du modèle aux contraintes matérielles spécifiques.

Minimisation du volume total des paramètres à entraîner

Nous proposons d'utiliser la fonction objectif suivante pour minimiser le volume total des paramètres à entraîner :

$$\min_{X} f_1(X) = \min_{a=\mathcal{A}(X)} \frac{\mathcal{P}(a)}{\mathcal{P}_{\max}} \tag{3.8}$$

En minimisant cette fonction, nous réduisons le nombre de paramètres d'entraînement du modèle, ce qui permet de réduire son empreinte mémoire. Le terme $\mathcal{P}_{\max}$ représente le volume maximal de paramètres à ne pas dépasser, défini par l'utilisateur en fonction des contraintes matérielles et $\mathcal{P}(a)$ est une application qui retourne le volume total des paramètres à entraîner pour une architecture a.

Minimisation du nombre de FLOPs

De la même manière, nous pouvons utiliser la fonction objectif suivante pour minimiser le nombre d'opérations (FLOPs) requises :

$$\min_{X} f_2(X) = \min_{a=\mathcal{A}(X)} \frac{\mathcal{O}(a)}{\mathcal{O}_{\max}} \tag{3.9}$$

En minimisant cette fonction, nous réduisons le nombre d'opérations nécessaires pour calculer les sorties, ce qui se traduit par un temps d'exécution plus court du modèle. Le terme $\mathcal{O}_{\max}$ représente le nombre maximal d'opérations à ne pas dépasser, également défini par l'utilisateur en fonction des contraintes matérielles. L'application $\mathcal{O}(a)$ retourne le nombre d'opérations (FLOPs) requises pour prédire une sortie par une architecture a.

L'utilisation de ces fonctions objectif permet de guider le processus d'optimisation du modèle afin de trouver des configurations qui répondent aux contraintes matérielles spécifiques, en minimisant à la fois le volume des paramètres et le nombre d'opérations requises.

Fonctions objectif liées à l'entraînement du modèle

Pour évaluer les performances d'une architecture a donnée, il est essentiel de procéder à son entraînement après sa création $a = \mathcal{A}(X)$. Afin d'étudier les performances du modèle, nous utilisons des données de validation $\mathbb{D}_{val}$ qui ne sont pas utilisées lors de l'entraînement du modèle. Dans ce contexte, la fonction de perte $\mathcal{L}_{val}$ sur les données de validation peut être utilisée comme fonction objectif. En minimisant cette fonction, nous améliorons la capacité du modèle à effectuer des prédictions précises.

Minimisation de la fonction de perte de validation (*validation loss*)

$$\begin{gathered} \min_X f_3(X) = \min_{a=\mathcal{A}(X)} \mathcal{L}_{val}\left(\mathbb{D}_{val}, w^*(a), a\right) \\ \text{sujet aux contraintes} \\ w^*(a) = \arg\min_w \left(\mathbb{D}_{train}, w, a\right) \end{gathered} \tag{3.10}$$

Cependant, l'utilisation exclusive de cette fonction de perte comme fonction objectif ne nous permet pas de comparer de manière efficace les meilleurs modèles disponibles. Il est donc crucial de prendre en compte d'autres métriques d'évaluation, telles que l'exactitude, la précision, le rappel, le F1-score, etc., afin de comparer et de sélectionner les meilleurs modèles. Ces métriques complémentaires fournissent une évaluation plus complète des performances, en tenant compte de différents aspects du modèle. Par conséquent, l'utilisation d'une combinaison appropriée de la fonction de perte sur les données de validation et des métriques d'évaluation pertinentes permet une évaluation précise et éclairée des performances des modèles.

Minimisation de la métrique d'évaluation

Dans ce cas-là, la fonction objectif vise à maximiser la métrique d'évaluation $\mathcal{M}$, telles que l'exactitude, la précision, le rappel, ou le F1-score, du modèle sur les données de validation. L'objectif est d'obtenir un modèle qui réalise des prédictions correctes. La fonction objectif peut être formulée comme suit :

$$\begin{gathered} \min_X f_4(X) = \min_{a=\mathcal{A}(X)} 1 - \mathcal{M}_{val}\left(\mathbb{D}_{val}, w^*(a), a\right) \\ \text{sujet aux contraintes} \\ w^*(a) = \arg\min_w \left(\mathbb{D}_{train}, w, a\right) \end{gathered} \tag{3.11}$$

3.3 . Optimisation en double boucle

Le problème d'optimisation défini dans la section précédente possède trois paramètres x_a, x_r et x_p à optimiser. Le premier paramètre définit l'architecture $a = \mathcal{A}(x_a)$ et les deux autres configurent cette architecture, définissant ainsi le modèle à entraîner $m = \mathcal{M}(a, x_r, x_p)$. Selon la remarque 3.2.1, l'optimisation en double boucle est la seule stratégie possible pour résoudre le problème d'optimisation défini par l'équation (3.1). La figure 3.1 illustre les deux boucles d'optimisations. La boucle externe optimise l'architecture via le paramètre x_a (« Recherche d'architecture »). Cette boucle donne à chaque fois une architecture fixe pour que la deuxième boucle interne optimise le modèle en configurant l'architecture avec les paramètres d'optimisation x_r et x_p (« Recherche de modèle ») :

1. Après les initialisations, la boucle d'optimisation « Recherche d'architecture » propose une première architecture a via le paramètre x_a.
2. Pour cette architecture $a = \mathcal{A}(x_a)$, une configuration x_r et x_p des options est proposée, créant un premier modèle $m = \mathcal{M}(a, x_r, x_p)$.
3. Ce modèle m est entraîné puis validé (c'est-à-dire que l'on en évalue ses performances).
4. Une première optimisation « Recherche de modèle » est faite uniquement sur x_r et x_p jusqu'à atteindre la condition d'arrêt (par exemple un nombre d'itération ou une condition sur la convergence des performances).
5. Une fois x_r et x_p optimisées pour ce x_a donné, la boucle « Recherche d'architecture » reprend et un nouveau x_a est proposé, puis x_r et x_p sont optimisées par rapport à ce nouveau x_a.
6. L'étape précédente est répétée jusqu'à atteindre la condition d'arrêt pour l'optimisation de x_a et les meilleurs paramètres (x_a, x_r, x_p) sont retournées.

Les inconvénients liés à l'utilisation de cette stratégie d'optimisation sont principalement la durée d'optimisation et la convergence. En effet, en utilisant un algorithme d'optimisation pour chaque boucle, le processus d'optimisation requiert généralement un plus grand nombre d'itérations par rapport à une approche en boucle simple. Cette multiplication des itérations peut entraîner une convergence plus lente vers une solution optimale et engendrer des temps de calcul très élevés. D'un autre côté, une optimisation en boucle double peut offrir une exploration plus étendue de l'espace de recherche, ce qui est bénéfique pour explorer différentes solutions. Toutefois, cela se fait au prix d'un temps de calcul plus élevé, ce qui rend cette stratégie inefficace pour notre problème qui est très complexe.

Dans la section suivante, notre objectif est de trouver une stratégie d'optimisation en boucle simple qui permette de réduire significativement le temps de calcul tout en assurant une convergence rapide vers une solution optimale.

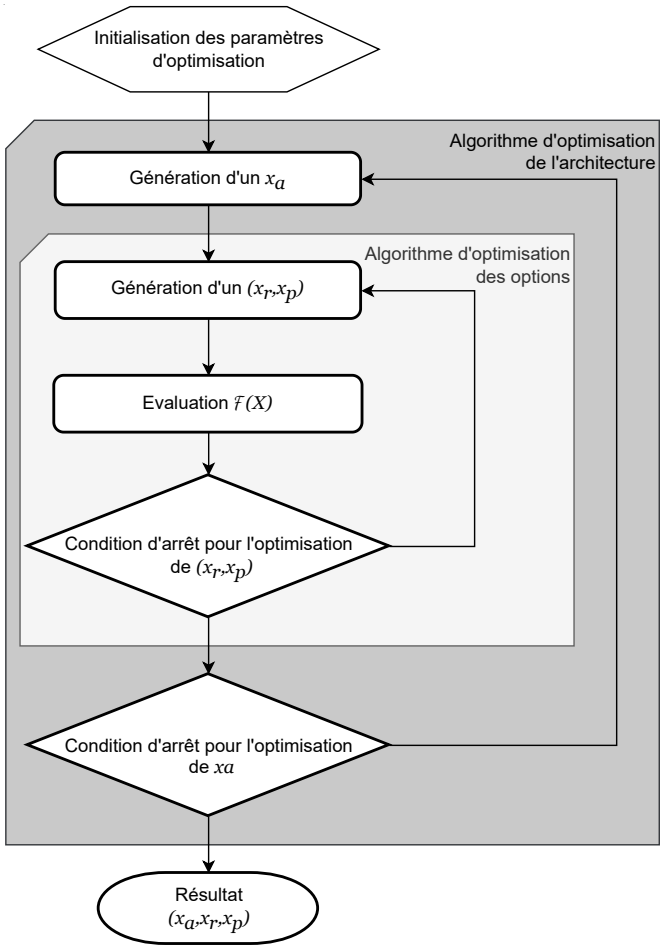


Figure 3.1 – Optimisation de $X = (x_a, x_r, x_p)$ en deux boucles d'optimisation imbriquées.

3.4 . Optimisation en simple boucle

Pour pouvoir optimiser l'architecture et sa configuration conjointement, et pouvoir avoir une seule boucle d'optimisation (telle que présentée dans la figure 3.2), celles-ci doivent être indépendantes. Plus précisément, il doit être possible de définir la configuration x_r et x_p sans connaître l'architecture x_a. Dans cette section, nous présenterons dans quel cas x_r et x_p sont dépendant de x_a, puis nous redéfinirons les deux paramètres x_r et x_p pour qu'ils soient indépendants de x_a à l'aide d'algorithmes heuristiques.

Les paramètres x_r et x_p sont dépendants de x_a par le fait que, selon la valeur de x_a, les valeurs maximum de x_r et x_p varient. En effet, modifier la valeur de x_a peut signifier que le nombre de bloc dans l'architecture augmente, donc le nombre d'options à configurer augmente aussi. Cela est le cas si les options r et p sont contraintes par une valeur maximum,

respectivement $r_{\max}$ et $p_{\max}$.

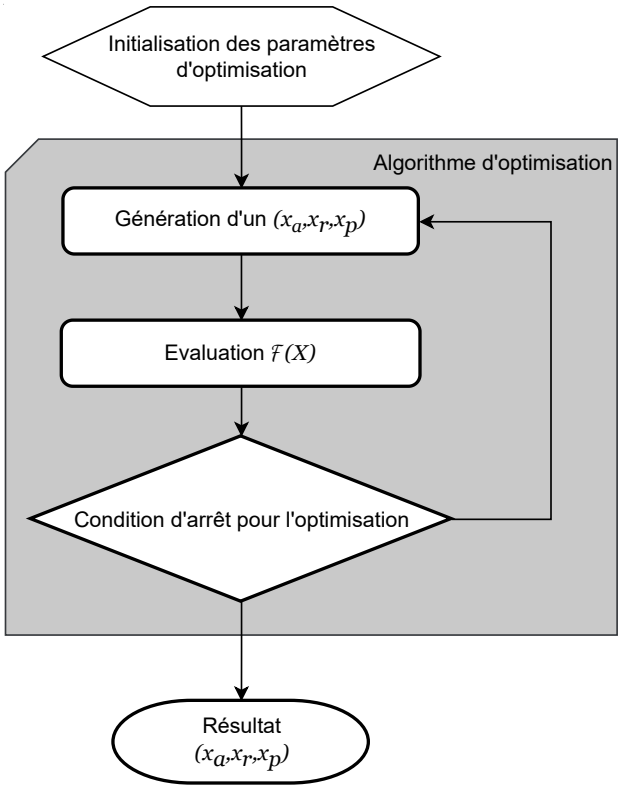


Figure 3.2 – Exemple d'optimisation conjointe de l'architecture imbriquée x_a et des options x_r et x_p en une seule boucle.

Exemple 3.4.1 *Prenons un espace de recherche avec un seul bloc élémentaire* $\mathbb{E} = \{e_1(p,r)\}$, *avec* $p, r \in \{0,1\}$ *et n'ayant qu'un seul niveau de pipelines* $\mathbb{P}_1$. *Si* $x_a = 3$, *l'architecture correspond à* $p_{1,3} = (e_1(p,r), e_1(p,r), e_1(p,r))$ *et donc* $x_{r_{\max}} = 2^3 = 8$ *(respectivement* $x_{p_{\max}} = 2^3 = 8$*). Donc pour l'architecture définie par le pipeline* $p_{1,3}$, *il y a* $64 = 8 \times 8$ *possibilités de configuration. Tandis que si* $x_a = 2$, *l'architecture correspond à* $p_{1,2} = (e_1(p,r), e_1(p,r))$ *et donc* $x_{r_{\max}} = 2^2 = 4$ *(respectivement* $x_{p_{\max}} = 2^2 = 4$*). Ainsi, il y a seulement* 16 *possibilités de configurer l'architecture* $p_{1,2}$.

Pour que x_r, x_p et x_a ne soient pas dépendants, il faut donc redéfinir x_r et x_p et leurs limites. En utilisant les contraintes liées au tenseur de sortie définie dans la section 3.2.2 équation (3.6) et l'équation (2.26) dans la section 2.2.3, nous trouvons les inégalités

suivantes :

$$h_{\min} \leq h_{\mathbf{Y}} = \frac{h_{\mathbf{X}}}{2^{p_{L,k}}} \leq h_{\max} \tag{3.12}$$

$$w_{\min} \leq w_{\mathbf{Y}} = \frac{w_{\mathbf{X}}}{2^{p_{L,k}}} \leq w_{\max} \tag{3.13}$$

$$c_{\min} \leq c_{\mathbf{Y}} = 2^{r_{L,k}} c_{\mathbf{X}} \leq c_{\max} \tag{3.14}$$

En appliquant un logarithme de base 2 sur les équations 3.12, 3.13 et 3.14, on aura :

$$\texttt{log}_2\left(\frac{h_{\mathbf{X}}}{h_{\max}}\right) \leq p_{L,k} \leq \texttt{log}_2\left(\frac{h_{\mathbf{X}}}{h_{\min}}\right) \tag{3.15}$$

$$\texttt{log}_2\left(\frac{w_{\mathbf{X}}}{w_{\max}}\right) \leq p_{L,k} \leq \texttt{log}_2\left(\frac{w_{\mathbf{X}}}{w_{\min}}\right) \tag{3.16}$$

$$\texttt{log}_2\left(\frac{c_{\mathbf{X}}}{c_{\max}}\right) \leq r_{L,k} \leq \texttt{log}_2\left(\frac{c_{\mathbf{X}}}{c_{\min}}\right) \tag{3.17}$$

Le paramètre $p_{L,k}$ indique le nombre de cartes de fonction appliqué par le k-ième bloc de niveau L (niveau le plus haut), tandis que le paramètre $r_{L,k}$ donne la taille du tenseur réduite par la même architecture.

Pour déterminer les paramètres de configuration au niveau des blocs élémentaires utilisés pour configurer l'architecture k, nous devons résoudre récursivement l'équation (2.27) présentée dans la section 2.2.3. Pour éviter l'utilisation d'une deuxième boucle d'optimisation, nous proposons des algorithmes heuristiques permettant d'obtenir une configuration unique et efficace. Ainsi, les paramètres d'optimisation x_r et x_p correspondent aux paramètres des blocs du dernier niveau, à savoir $r_{L,k}$ et $p_{L,k}$ respectivement. Les limites appliquées sur ces paramètres deviennent :

$$\max\left(\texttt{log}_2\left(\frac{h_{\mathbf{X}}}{h_{\max}}\right), \texttt{log}_2\left(\frac{w_{\mathbf{X}}}{w_{\max}}\right)\right) \leq x_p \leq \min\left(\texttt{log}_2\left(\frac{h_{\mathbf{X}}}{h_{\min}}\right), \texttt{log}_2\left(\frac{w_{\mathbf{X}}}{w_{\min}}\right)\right) \tag{3.18}$$

$$\texttt{log}_2\left(\frac{c_{\mathbf{X}}}{c_{\max}}\right) \leq x_r \leq \texttt{log}_2\left(\frac{c_{\mathbf{X}}}{c_{\min}}\right) \tag{3.19}$$

Remarque 3.4.1 *Les paramètres d'optimisation x_r et x_p peuvent être des constantes si $h_{\min} = h_{\max}$, $w_{\min} = w_{\max}$ et $c_{\min} = c_{\max}$.*

Les contraintes liées au tenseur de sortie, définies dans la section 3.2.2, équation (3.6), seront remplacées par la contrainte d'inégalité suivante :

$$\forall x_i \in \mathcal{E}(a_k) \text{ et } p_i, r_i \in \mathbb{N} : \begin{cases} p_i - p_{\max} \leq 0 \\ r_i - r_{\max} \leq 0 \end{cases} \tag{3.20}$$

L'application $\mathcal{E}$ retourne la liste des blocs élémentaires utilisée pour créer l'architecture a_k. Les paramètres de configuration p_i et r_i déterminent la configuration du bloc élémentaire

x_i. Les valeurs $p_{\max}$ et $r_{\max}$ représentent les limites appliquées aux blocs élémentaires. Par exemple, pour un bloc avec une opération de *pooling*, si la limite $p_{\max} = 3$, le paramètre p_i peut prendre les valeurs $\{0, 1, 2, 3\}$ correspondant $\{2^0, 2^1, 2^2, 2^3\}$, pour configurer différentes tailles de *pooling*.

Dans les sections suivantes, nous allons présenter des propositions de tels algorithmes heuristiques et leurs effets sur la taille de l'espace de recherche.

3.4.1 . Configuration heuristique d'un Pipeline

Dans la structure du pipeline illustrée dans la figure 3.3, les blocs internes sont alignés en série. Notre objectif dans cette section est de déterminer la configuration de ces éléments internes uniquement à partir de la configuration globale du pipeline et des paramètres maximaux de chaque bloc.

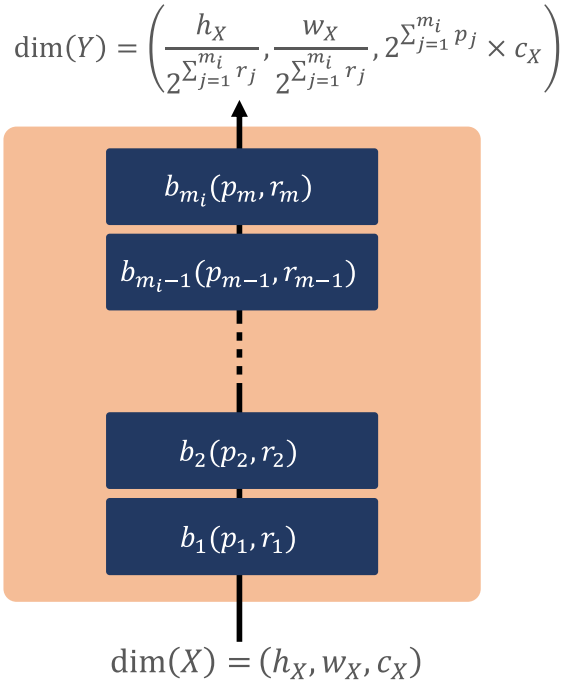


Figure 3.3 – Des pipelines dont les options sont configurées heuristiquement.

Prenons en compte le problème où un pipeline est constitué de m blocs internes, représentés par $\mathbb{B} = \{b_1, b_2, \ldots, b_m\}$. Imaginons que chaque bloc possède une limite supérieure quant à la diminution de la dimension spatiale du tenseur entrant, notée $\bar{p}_i$, et une autre concernant l'augmentation de la profondeur du tenseur entrant, $\bar{r}_i$. Ces limites peuvent aussi être représentées par une fonction qui fournit la limite supérieure d'un bloc b_i. Par exemple, $\mathcal{C}_p : \mathbb{B} \to \mathbb{N}$ où $\mathcal{C}_p(b_i) = \bar{p}_i$ (et de même, $\mathcal{C}_r : \mathbb{B} \to \mathbb{N}$ pour $\mathcal{C}_r(b_i) = \bar{r}_i$). Le défi est de définir la configuration de chaque bloc avec le couple (p_i, r_i) tout en prenant en compte la dimension du tenseur de sortie du pipeline $\left(\frac{h_{\mathbf{X}}}{2^p}, \frac{w_{\mathbf{X}}}{2^p}, 2^r c_{\mathbf{X}}\right)$. Ici, p et r sont des paramètres

fixés. Le tenseur d'entrée est représenté par $(h_{\mathbf{X}}, w_{\mathbf{X}}, c_{\mathbf{X}})$. Ainsi, pour résoudre le problème, nous devons résoudre ces égalités :

$$\begin{gathered}
\frac{h_{\mathbf{X}}}{2^p} = \frac{h_{\mathbf{X}}}{2^{\sum_{i=1}^{m} p_i}} \\
\frac{w_{\mathbf{X}}}{2^p} = \frac{w_{\mathbf{X}}}{2^{\sum_{i=1}^{m} p_i}} \\
c_{\mathbf{X}} \times 2^r = c_{\mathbf{X}} \times 2^{\sum_{i=1}^{m} r_i} \\
\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{gathered} \tag{3.21}$$

Ces égalités peuvent être simplifiées par les égalités suivantes :

$$\begin{gathered}
p = \sum_{i=1}^{m} p_i \\
r = \sum_{i=1}^{m} r_i \\
\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{gathered} \tag{3.22}$$

Afin de résoudre ce problème défini par l'équation (3.22), nous proposons deux approches heuristiques. La première repose sur la répartition séquentielle des options tandis que la seconde privilégie une distribution équilibrée des options.

Distribution séquentielle des options

Dans cette méthode, l'objectif est de configurer les blocs internes de manière séquentielle. Dès le premier bloc interne du pipeline, le nombre de canaux (ou filtres) est augmenté en exploitant constamment la limite maximale du bloc. En ce qui concerne la dimension spatiale (hauteur $h_{\mathbf{X}}$ et largeur $w_{\mathbf{X}}$), nous adoptons une stratégie similaire à celle des canaux, mais en procédant du bloc de sortie vers le bloc d'entrée. Ainsi, cette approche peut se programmer en utilisant les équations suivantes :

$$\forall i = 1, \ldots, m-1 : p_{i+1} = \min\left(\bar{p}_{i+1}, \max\left(p - \sum_{j=1}^{i} p_j, 0\right)\right) \tag{3.23}$$

$$\forall i = m, \ldots, 2 : r_{i-1} = \min\left(\bar{r}_{i-1}, \max\left(r - \sum_{j=i}^{m} r_j, 0\right)\right) \tag{3.24}$$

avec les tests de faisabilité suivants :

$$p \leq \sum_{i=1}^{m} \bar{p}_i \tag{3.25}$$

$$r \leq \sum_{i=1}^{m} \bar{r}_i \tag{3.26}$$

$$\tag{3.27}$$

Distribution équitable des options

Une deuxième approche peut être de répartir équitablement les options d'un pipeline parmi tous ses blocs internes. Pour cela, un algorithme simple de distribution équitable peut être employé. Cet algorithme est détaillé dans 3.1. Il vise à allouer un nombre d'options x dans une liste X de la manière la plus uniforme possible, tout en tenant compte des contraintes imposées par la liste $\bar{X}$. Pour commencer, la valeur la plus basse entre une limite maximale donnée et le quotient issu de la division euclidienne de x par la taille de la liste n est attribuée à chaque élément de la liste, assurant ainsi une première allocation parfaitement uniforme. Le reste est ensuite distribué séquentiellement à partir du début de la liste.

Algorithme 3.1 : Algorithme de répartition équitable

Données : Un nombre x à répartir dans une liste X de dimension n. $\bar{X}$ liste contient les limites maximales

Résultat : Une liste X contenant une répartition équitable

```
part_equitable ← min(⌊x/n⌋, min(X̄))
reste ← x − n ∗ part_equitable
pour i de 0 à n − 1 faire
    X[i] ← part_equitable
fin
i = 0
tant que reste > 0 faire
    si X[i] < X̄[i] alors
        X[i] ← X[i] + 1
        reste ← reste − 1
    fin
    i ← i + 1
    i ← i mod n
fin
retourner X
```

Dans le cadre de la répartition des options (p, r) d'un pipeline parmi ces blocs internes,

l'algorithme 3.1 peut être utilisé successivement sur p puis r avec une liste représentant les blocs internes. Ainsi, tel qu'illustré dans la figure 3.4, en prenant l'exemple d'un bloc $b_2\,(10,6)$ avec $m_L = 3$ blocs internes, la répartition des options $p = 10$ et $r = 6$ se fera de telle manière que ces blocs internes seront : $b_1\,(4,2)$, $b_2\,(3,2)$ et $b_3\,(3,2)$.

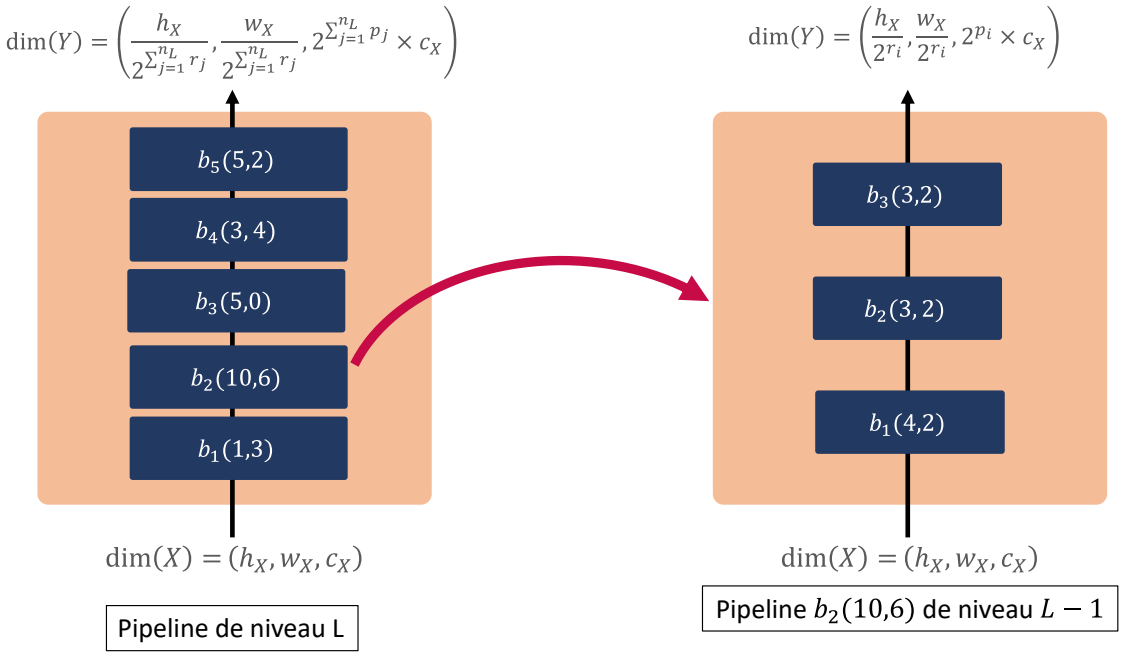


Figure 3.4 – Exemple d'options d'un $b_1 \in P_L$ dont les options sont réparties parmi les blocs internes selon l'algorithme 3.1.

3.4.2 . Configuration heuristique d'un Ruban

Dans un ruban, les blocs internes sont disposés en parallèle, à l'inverse d'un agencement en pipeline où ils se succèdent séquentiellement. Dans le contexte du ruban, tous les blocs partagent l'entrée du ruban, et leurs sorties sont fusionnées pour constituer la sortie du ruban, par concaténation ou addition. Dans les deux méthodes de fusion, les dimensions spatiales (largeur et hauteur du tenseur) des blocs doivent être identiques pour que la fusion soit réalisable. En ce qui concerne la profondeur du tenseur, lors d'une fusion par addition, les profondeurs des tenseurs des blocs internes doivent être égales. Ce n'est pas le cas pour la fusion par concaténation où la profondeur du tenseur de sortie est la somme des profondeurs des tenseurs des blocs internes.

Dans cette section, nous cherchons comment configurer les blocs internes du ruban en ne connaissant que la dimension du tenseur de sortie du ruban et les paramètres de configuration maximale de chaque bloc interne. Notre but est de développer des algorithmes heuristiques visant à minimiser à la fois le nombre de paramètres à entraîner et le nombre d'opérations à virgule flottante (FLOPs) nécessaires pour prédire la sortie du ruban.

Considérons le problème suivant, un ruban contient m blocs internes $\mathbb{B} = \{b_1, b_2, \ldots, b_m\}$ où leurs sorties sont fusionnées par une opération d'addition ou de concaténation. Supposons qu'il y ait une valeur maximale définie pour chaque bloc en ce qui concerne la réduction de la dimension spatiale du tenseur d'entrée du ruban, notée $\bar{p}_i$, ainsi qu'une valeur maximale pour l'augmentation de la profondeur du tenseur d'entrée du ruban, notée $\bar{r}_i$. Ces deux valeurs maximales peuvent également être exprimées sous la forme d'une fonction (ou application) qui retourne la valeur maximale d'un bloc b_i, désignée comme $\mathcal{C}_p : \mathbb{B} \to \mathbf{N}$, telle que $\mathcal{C}_p(b_i) = \bar{p}_i$ (et respectivement, $\mathcal{C}_r : \mathbb{B} \to \mathbf{N}$, telle que $\mathcal{C}_r(b_i) = \bar{r}_i$). Le problème se posant ici, est de configurer chaque bloc par le tuple (p_i, r_i) en respectant la dimension du tenseur de sortie de notre ruban $\left(\frac{h_{\mathbf{X}}}{2^p}, \frac{w_{\mathbf{X}}}{2^p}, 2^r c_{\mathbf{X}}\right)$. les paramètres de configuration p et r sont des paramètres donnés. Le tenseur d'entrée du ruban est donné par $(h_{\mathbf{X}}, w_{\mathbf{X}}, c_{\mathbf{X}})$.

Cas d'une fusion par addition

Dans le cas d'un ruban avec un opérateur de fusion par addition, nous devons résoudre ces égalités :

$$\begin{aligned}
\frac{h_{\mathbf{X}}}{2^p} &= \frac{h_{\mathbf{X}}}{2^{p_1+\alpha_1}} = \cdots = \frac{h_{\mathbf{X}}}{2^{p_m+\alpha_m}} \\
\frac{w_{\mathbf{X}}}{2^p} &= \frac{w_{\mathbf{X}}}{2^{p_1+\alpha_1}} = \cdots = \frac{w_{\mathbf{X}}}{2^{p_m+\alpha_m}} \\
c_{\mathbf{X}} \times 2^r &= c_{\mathbf{X}} \times 2^{r_1+\beta_1} = \cdots = c_{\mathbf{X}} \times 2^{r_m+\beta_m} \\
&\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : &\left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{aligned} \tag{3.28}$$

Ces égalités peuvent être simplifiées par les égalités suivantes :

$$\begin{aligned}
p &= p_1 + \alpha_1 = \cdots = p_m + \alpha_m \\
r &= r_1 + \beta_1 = \cdots = r_m + \beta_m \\
&\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : &\left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{aligned} \tag{3.29}$$

Il y a une infinité de solutions de ces égalités (3.29), pour notre problème, nous utilisons la solution suivante :

$$\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i = \min\left(\bar{p}_i, p\right) \text{ avec } \alpha_i = p - p_i \\ r_i = \min\left(\bar{r}_i, r\right) \text{ avec } \beta_i = r - r_i \end{array} \right. \tag{3.30}$$

avec les tests de faisabilité suivants :

$$p \leq \max\left(\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_m\right) \tag{3.31}$$

$$r \leq \max\left(\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_m\right) \tag{3.32}$$

Les tests de faisabilité ci-dessus nous permettent de vérifier si les blocs internes du ruban sont capables de générer un tenseur qui a la même dimension du tenseur de sortie du ruban. Le paramètre α_i représente le nombre *pooling* 2×2 à appliquer après un bloc interne, ou un *pooling* avec une taille d'échantillonnage et un pas égaux à α_i). Le paramètre β_i permet d'ajuster le nombre de canaux (dimension en profondeur) tout en préservant la dimension spatiale en utilisant la convolution 1×1 avec un nombre de filtres égal à $2^{p_i+\beta_i} c_{\mathbf{X}}$.

Cas d'une fusion par concaténation

Dans le cas d'un ruban avec un opérateur de fusion par concaténation, nous aurons les égalités suivantes :

$$\begin{gathered}
\frac{h_{\mathbf{X}}}{2^p} = \frac{h_{\mathbf{X}}}{2^{p_1+\alpha_1}} = \cdots = \frac{h_{\mathbf{X}}}{2^{p_m+\alpha_m}} \\
\frac{w_{\mathbf{X}}}{2^p} = \frac{w_{\mathbf{X}}}{2^{p_1+\alpha_1}} = \cdots = \frac{w_{\mathbf{X}}}{2^{p_m+\alpha_m}} \\
c_{\mathbf{X}} \times 2^r = c_{\mathbf{X}} \times 2^{r_1+\beta_1} + \cdots + c_{\mathbf{X}} \times 2^{r_m+\beta_m} \\
\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{gathered} \tag{3.33}$$

Ces égalités peuvent aussi être simplifiées par les égalités suivantes :

$$\begin{gathered}
p = p_1 + \alpha_1 = \cdots = p_m + \alpha_m \\
2^r = 2^{r_1+\beta_1} + \cdots + 2^{r_m+\beta_m} \\
\text{sujet aux contraintes} \\
\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i \leq \mathcal{C}_p(b_i) = \bar{p}_i \\ r_i \leq \mathcal{C}_r(b_i) = \bar{r}_i \end{array} \right.
\end{gathered} \tag{3.34}$$

On remarque que la différence entre les égalités de l'équation (3.29) et (3.34) se trouve dans la dimension en profondeur. Lors de l'opération de concaténation, tous les canaux des tenseurs sont regroupés en un unique tenseur. Par conséquent, le nombre de canaux de ce tenseur résultant est égal à la somme des canaux des tenseurs concaténés. En utilisant une décomposition en puissances de 2 (e.g. $2^5 = 2^4 + 2^3 + 2^2 + 2^2$), nous pouvons proposer la solution suivante :

$$\forall b_i \in \mathbb{B} : \left\{ \begin{array}{l} p_i = \min\left(\bar{p}_i, p\right) \text{ avec } \alpha_i = p - p_i \\ r_i = \min\left(\bar{r}_i, \hat{r}_i\right) \text{ avec } \beta_i = \hat{r}_i - r_i \text{ et } \hat{r}_i = \max\left(r - i, r - m - 1\right) \end{array} \right. \tag{3.35}$$

avec les tests de faisabilité suivants :

$$p \leq \max\left(\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_m\right) \tag{3.36}$$

$$2^r \leq 2^{\bar{r}_1} + \cdots + 2^{\bar{r}_m} \tag{3.37}$$

$$m \leq r + 1 \tag{3.38}$$

3.5 . Optimisation en seule boucle avec un espace de recherche dynamique

Dans les sections précédentes, nous avons vu comment définir une optimisation méta-heuristique de l'architecture x_a et des options x_p et x_r avec une seule boucle d'optimisation. Cependant, en l'état, un problème de l'espace de recherche imbriqué est l'augmentation rapide de la taille de ce dernier avec l'augmentation du nombre n_0 de blocs élémentaires et du nombre de niveaux d'imbrications L (voir section 2.2.3). De plus, un nombre de blocs n_{i-1} élevé dans un niveau $i-1$ implique que la décomposition d'un ruban de niveau i prendra beaucoup plus de temps (voir section 2.3.2.B). Pour limiter le nombre de blocs élémentaires disponible pendant l'optimisation, nous introduisons le principe d'optimisation à base de sélection.

Cette optimisation se base sur le fait de prendre un sous-ensemble des ensembles de blocs pour former les architectures imbriquées. Ces sous-ensembles sont nommés :

1. $\mathbb{E}' \subset \mathbb{E}$ pour les blocs élémentaires.
2. $\mathbb{P}' \subset \mathbb{P}$ pour les pipelines.
3. $\mathbb{R}' \subset \mathbb{R}$ pour les rubans.

Par exemple, comme illustré dans la figure 3.5, la sélection des $n'_0 = 3$ premiers blocs élémentaires de $\mathbb{E} = \{e_1, e_2, \ldots, e_{42}\}$ pour former l'espace $\mathbb{E}'$ permet de réduire le nombre de pipelines dans $\mathbb{P}'_1$ à 39 (contre $75\ 893$ pour $\mathbb{P}_1$).

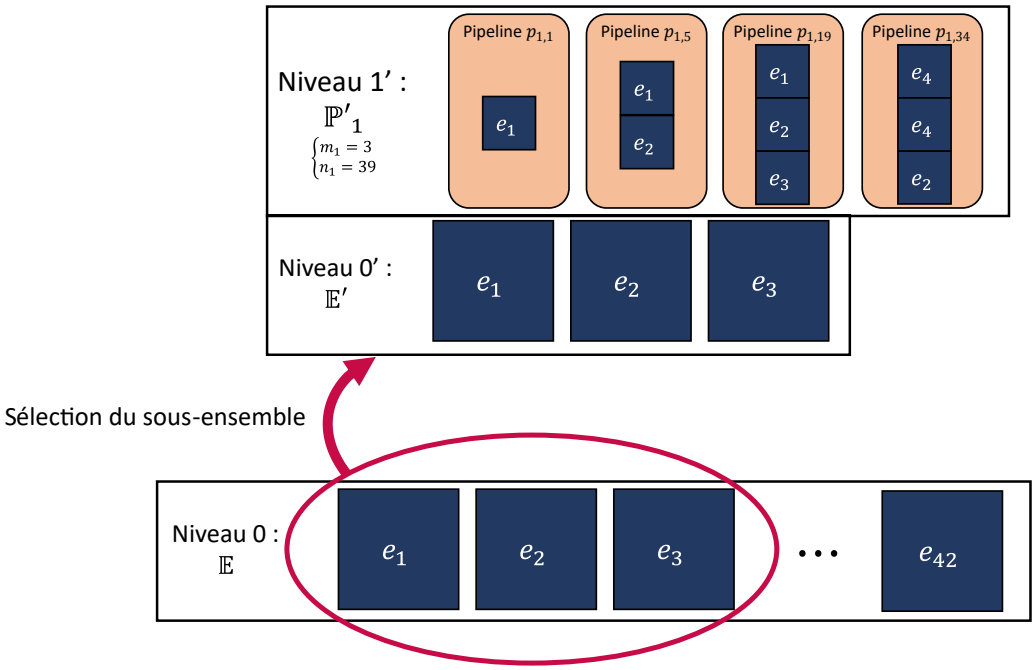


Figure 3.5 – Exemple de sélection de sous-ensemble $\mathbb{E}' \subset \mathbb{E}$ de cardinalité $n'_0 = 3$.

Ainsi, on peut définir une nouvelle stratégie d'optimisation utilisant la sélection de sous-ensembles avec la fonction objectif suivante :

$$\mathcal{F}\left(\left[x_1^{(\mathbb{E})}, x_2^{(\mathbb{E})}, \ldots, x_{n'_0}^{(\mathbb{E})}\right], \left[x_1^{(\mathbb{B}_1)}, x_2^{(\mathbb{B}_1)}, \ldots, x_{n'_1}^{(\mathbb{B}_1)}\right], \left[x_1^{(\mathbb{B}_2)}, x_2^{(\mathbb{B}_2)}, \ldots, x_{n'_2}^{(\mathbb{B}_2)}\right], \ldots, \left[x_1^{(\mathbb{B}_{L-1})}, x_2^{(\mathbb{B}_{L-1})}, \ldots, x_{L-1}^{(\mathbb{B}_{L-1})}\right], x_a, x_r, x_p\right) \quad (3.39)$$

où les $x_k^{(\mathbb{E})}$ sont les blocs élémentaires sélectionnés parmi $\mathbb{E}$ pour former $\mathbb{E}'$, $x_k^{(\mathbb{B}_i)}$ sont les blocs composés sélectionnés parmi $\mathbb{B}_i$ pour former $\mathbb{B}'_i$ et x_a, x_r et x_p sont les paramètres d'optimisations tels que présentés dans la précédente fonction objectif 3.1.

Cependant, l'optimisation par sélection de sous-ensemble fait apparaître un problème de redondance dans les architectures de l'espace de recherche. En effet, deux couples de x_a différents peuvent représenter la même architecture en sélectionnant des sous-ensembles adéquats. La section suivante discute de ce problème de redondance.

3.5.1 . Identifiant unique pour les architectures

Avec la sélection de sous-ensembles pour créer les architectures, des problèmes de redondances peuvent apparaître. En effet, deux x_a différents peuvent représenter le même bloc. C'est-à-dire que, pour un espace de recherche à L niveaux :

$$\exists \left(\mathbb{E}'_1, \mathbb{E}'_2\right) \subset \mathbb{E}^2, \exists x_{a1} \neq x_{a2}, b_{L,x_{a1}} = b_{L,x_{a2}} \quad (3.40)$$

où $b_{L,x_{a1}}$ et $b_{L,x_{a2}}$ sont respectivement les x_{a1}ième et x_{a2}ième blocs de niveau L construits à base respectivement de $\mathbb{E}'_1$ et $\mathbb{E}'_2$. Dans la figure 3.6, il est montré deux pipelines de niveau 1 identiques mais avec des x_a différents.

Le problème inverse existe aussi. C'est-à-dire qu'un même x_a peut définir des architectures différentes :

$$\exists \left(\mathbb{E}'_1, \mathbb{E}'_2\right) \subset \mathbb{E}^2, \exists x_{a1} = x_{a2}, b_{L,x_{a1}} \neq b_{L,x_{a2}} \quad (3.41)$$

où $b_{L,x_{a1}}$ et $b_{L,x_{a2}}$ sont respectivement les x_{a1}ième et x_{a2}ième blocs de niveau L construits à base respectivement de $\mathbb{E}'_1$ et $\mathbb{E}'_2$. Dans la figure 3.7, il est montré deux pipelines de niveau 1 différents mais avec des x_a identiques.

Cette redondance pose le problème que, lors de l'optimisation, la même architecture peut être proposée plusieurs fois. L'algorithme va donc être amené à réévaluer une architecture plusieurs fois. En considérant que l'évaluation implique l'entraînement d'un CNN, cela implique une perte de temps importante. Pour résoudre ce problème, nous proposons l'utilisation d'un identifiant global pour reconnaître une architecture déjà entraînée.

En effet, le problème de redondance apparaît avec le fait que les ensembles $\mathbb{E}'$ et $\mathbb{B}'_i$ utilisé pour construire les architectures diffèrent. Donc, pour avoir un identifiant global pour tout $\mathbb{E}' \subset \mathbb{E}$ et $\mathbb{B}'_i \subset \mathbb{B}_i$, il faut considérer le x_a directement par rapport à $\mathbb{E}$ et $\mathbb{B}_i$. Ainsi, à chaque nouvelle proposition d'architecture x_a, l'identifiant global est déterminé, et, si une architecture avec le même identifiant a déjà été entraînée, l'algorithme peut directement récupérer l'évaluation de cette dernière, évitant de perdre du temps à recalculer la fonction objectif avec ces paramètres.

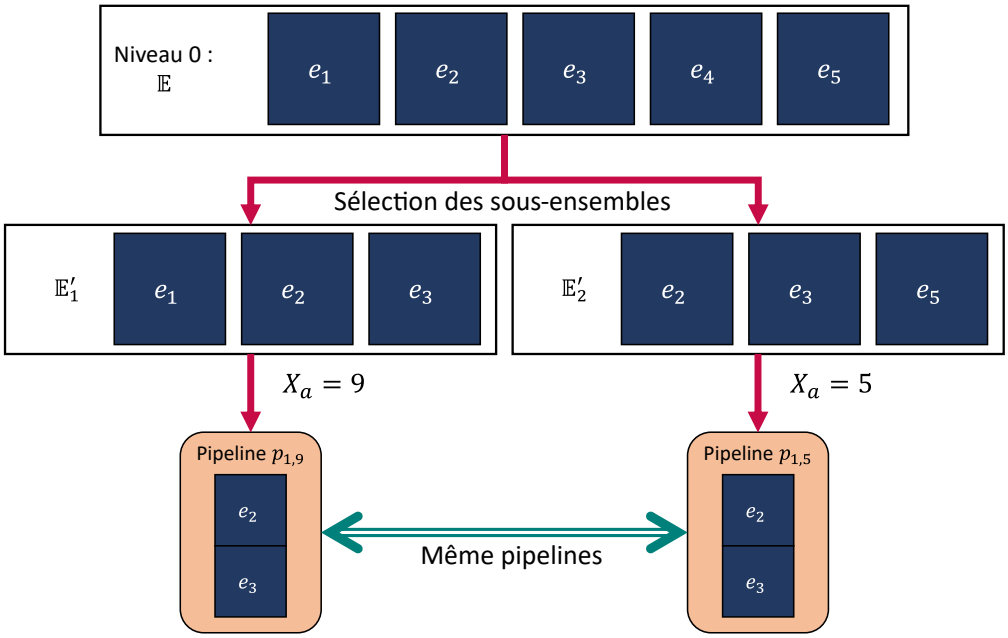


Figure 3.6 – Exemple d'une façon d'obtenir le même pipeline avec deux x_a différents par la sélection de deux sous-ensembles $\mathbb{E}'_1$ et $\mathbb{E}'_2$

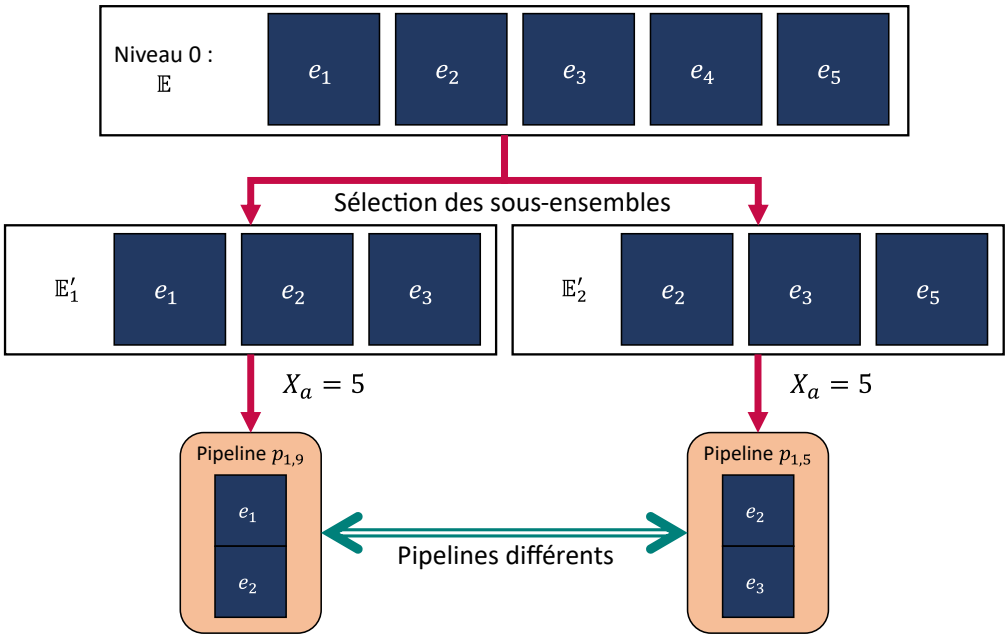


Figure 3.7 – Exemple d'une façon d'obtenir des pipelines différents avec le même x_a par la sélection de deux sous-ensembles $\mathbb{E}'_1$ et $\mathbb{E}'_2$

3.6 . Conclusion

Dans ce chapitre, nous avons présenté différentes stratégies d'optimisation pour minimiser la fonction objectif définie par l'équation 3.1. L'optimisation se fait selon deux paramètres : x_a qui contrôle l'architecture neuronale à base de blocs (section 2.3), et le couple (x_r, x_p) qui contrôle la configuration des options. Au fur et à mesure de la présentation de ces méthodes, le contrôle sur les degrés de liberté (et donc notamment sur le temps d'optimisation) est de plus en plus fort.

Ces stratégies sont basées sur des méthodes méta-heuristiques et nous avons présenté dans un premier temps comment faire l'optimisation en deux boucles : l'une optimisant x_a et l'autre optimisant (x_r, x_p) à chaque x_a (section 3.3). Puis, nous avons montré comment (x_r, x_p) peut être défini indépendamment de x_a, permettant ainsi de n'avoir qu'une boucle d'optimisation (section 3.4) à l'aide d'algorithmes heuristiques. Enfin, nous présentons une optimisation dynamique se basant sur la sélection de sous-ensembles de $\mathbb{E}$ et $\mathbb{B}_i$ (section 3.5). Cette dernière permet de réduire $x_{a_{\max}}$, évitant d'avoir des décodages trop importants à calculer.

Parmi ces stratégies, l'optimisation conjointe de x_a et (x_r, x_p) semble la plus prometteuse comparée à la double boucle d'optimisation qui nécessite entre autres plus d'optimisations de (x_r, x_p). Cela est discuté plus en détails au chapitre suivant (section 4.3.1). En revanche, pour des espaces de recherches de grandes dimensions (nombre élevé de niveaux et de blocs élémentaires), l'utilisation de l'optimisation par sélection de sous-ensemble permet de gagner du temps sur les calculs d'encodage et de décodage (section 2.3), tout en réduisant le nombre de degrés de liberté.

Le chapitre suivant porte sur différentes expériences menées grâce aux stratégies d'optimisation présentées dans le présent chapitre. Tout d'abord, une présentation de l'implémentation de l'espace de recherche imbriqué (présenté au chapitre 2) est faite, puis les expériences sont décrites et discutées.

4 - Évaluations

Ce chapitre présente tout d'abord l'implémentation de l'espace de recherche (2), nommée « OpenNas ». Puis trois expériences utilisant OpenNas sont présentées.

La première est une optimisation bi-objectifs sur MNIST étudiant la convergence de U-NSGA-III et l'optimisation bayésienne appliqués à OpenNas. La deuxième présente une recherche d'architecture uniquement de type VGG aboutissant à de meilleurs résultats que le VGG-16 original. La dernière expérience compare OpenNas à d'autres méthodes de NAS [24, 63, 67, 12, 22].

4.1 Introduction . . . 116
4.2 Description du framework OpenNas . . . 117
4.3 Expérimentations . . . 119
4.3.1 Étude de convergence . . . 119
Choix de l'optimisation en simple boucle . . . 119
Paramètres de l'espace de recherche imbriqué . . . 120
Front de pareto avec UNSGA-III . . . 121
Somme pondérée avec optimisation bayésienne . . . 123
Conclusion . . . 124
4.3.2 Étude d'amélioration de performances . . . 125
Résultats . . . 126
4.3.3 Comparaison à d'autres Auto-MLs . . . 126
Résultats . . . 127
4.4 Conclusion . . . 131

4.1 . Introduction

Dans les chapitres précédents, nous avons tout d'abord vu une méthode pour représenter des architectures neuronales par des entiers naturels. L'architecture est construite à partir de blocs de bases agencés en série (pipeline) et/ou en parallèle (ruban) pour former une structure imbriquée, puis cette structure peut être encodée pour être représentée par un simple entier naturel. Cette représentation permet de pouvoir facilement appliquer un algorithme d'optimisation afin de rechercher une architecture neuronale optimale. Dans un second temps, nous avons présenté des stratégies d'optimisation pour utiliser cette méthode le plus efficacement possible. Notamment grâce à des algorithmes heuristiques permettant de rendre indépendants l'architecture et le contrôle des dimensions des tenseurs, ce qui donne la possibilité d'optimiser ces paramètres en une seule boucle.

Dans ce chapitre, nous présentons des expériences utilisant ces méthodes. Dans un premier temps, l'implémentation de ces méthodes est décrite, sous la forme d'un *framework* que nous avons nommé « OpenNas ». Puis nous présentons trois expériences menées à l'aide de ce framework.

La première expérience étudie la convergence vers une solution lors de l'application d'un algorithme génétique et d'une optimisation bayésienne à OpenNas, et compare ces deux algorithmes d'optimisation. La deuxième montre comment utiliser le framework pour optimiser une architecture déjà connue, dans ce cas VGG [76]. Enfin, la dernière expérience est une optimisation multi-objectif sur CIFAR-10 pour comparer OpenNas à d'autres techniques de Recherche d'architecture neuronale / *Neural Architecture Search* (NAS).

4.2 . Description du framework OpenNas

Dans cette section est présentée brièvement l'implémentation de l'espace de recherche présenté au chapitre 2. Nous avons nommé ce framework « OpenNas ».

Cette présentation est faite par l'intermédiaire du diagramme de classe de OpenNas (figure 4.1. Les classes sont les suivantes :

— La classe « *Block* » : Cette classe abstraite [1] décrit les blocs. Tous les types de blocs sont hérités de cette classe. Les blocs peuvent avoir un nom (name) et ont une fonction run qui sert pour la construction du modèle (voir classe Neural Network).

— La classe « Elementary block » : Cette classe hérite de la classe *Block* et correspond aux blocs élémentaires définis en section 2.2.1. Elle possède un identifiant (id) qui permet de connaître par quel entier ils sont représentés (voir section 2.3). Par exemple, on peut définir des blocs élémentaires e_0, e_1 et e_2 dont les identifiants respectifs seraient 0, 1, et 2. Chaque bloc élémentaire a une fonction run différente. En effet, la méthode run($\mathbf{X}$) d'une instance de cette classe représentant un bloc élémentaire e correspond à l'application $e\,(\mathbf{X})$ (voir section 2.2.1).

— La classe « *Composed block* » : Cette classe abstraite correspond aux blocs composés de la définition 2.2.1 et hérite de la classe *Block*. Un *Composed block* est composé de *Block*s (appelés inner_blocks : blocs internes). Le nombre maximum d'inner_blocks est donné par l'attribut n_layers. En reprenant les termes de la formalisation des blocs composés (définition 2.2.1), une instance de la classe *Composed block* représente un élément b_{ij} de l'ensemble $\mathbb{B}_i$ avec inner_blocks$\in \mathbb{B}_{i-1}$ et n_layers$= m_i$. Toutes les méthodes de cette classe sont abstraites et définies dans les classes enfants (Pipeline et Ribbon).

— La classe « Pipeline » : Cette classe, hérite de *Composed block* et correspond aux pipelines de la définition 2.2.3. Les méthodes **encode** et **decode** permettent respectivement l'encodage (algorithme 2.4) et le décodage (algorithme 2.5) de pipelines. Quant à la méthode run, elle appelle la méthode run de chaque bloc interne contenu dans inner_blocks en série.

— La classe « Ribbon » : Cette classe, hérite aussi de *Composed block* et correspond aux rubans de la définition 2.2.6. Les méthodes **encode** et **decode** permettent respectivement l'encodage (algorithme 2.10) et le décodage (algorithme 2.8) de rubans. Comme pour les pipelines, la méthode run appelle la méthode run des blocs internes. Cependant les méthodes run de chaque bloc interne contenu dans inner_blocks sont appelées en parallèle puis sont fusionnées par la méthode merge.

— La classe « Neural Network » : Cette classe permet de créer le CNN à partir de *Block*s. La méthode build_network crée un CNN en créant tout d'abord des blocs

1. Une classe abstraite est une classe dont on ne peut pas directement créer d'instance. Elle doit être instanciée par une de ces classes enfant.

sur n_levels niveaux en appelant la méthode **decode** de chaque *Block*. Puis la méthode run de chaque *Block* est appelée créant effectivement le réseau.

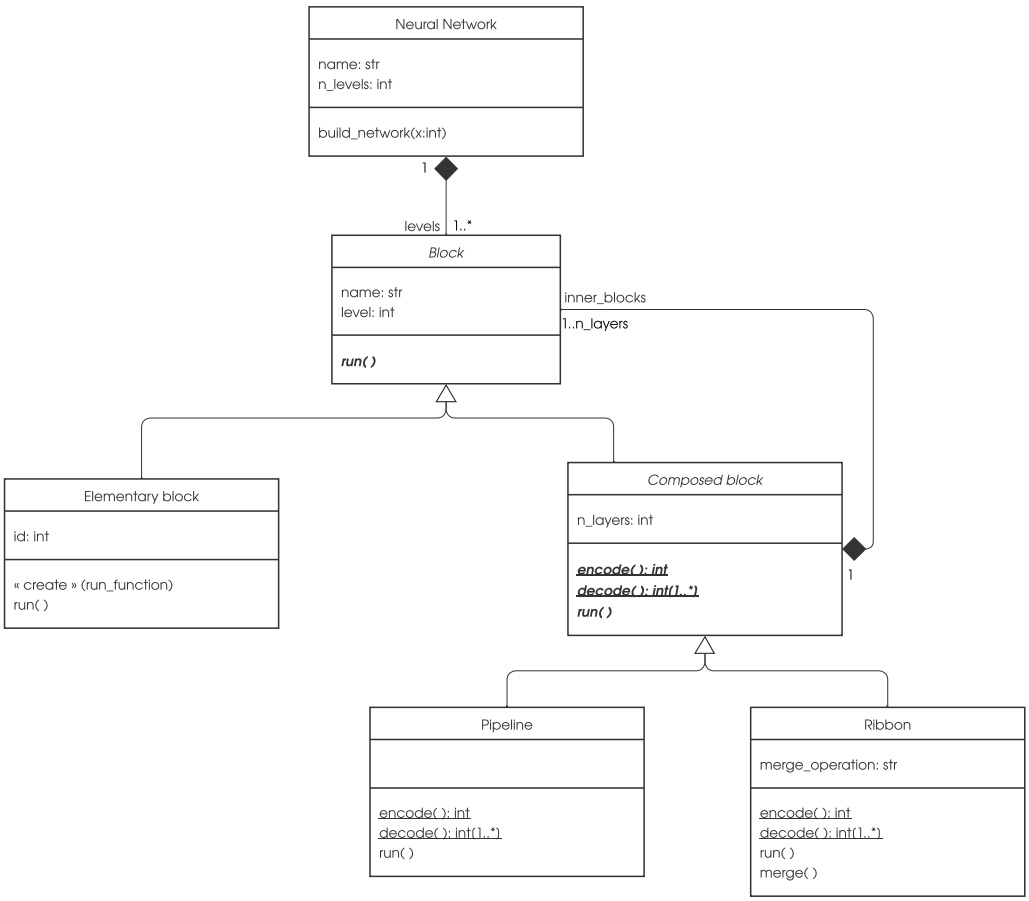


Figure 4.1 – Diagramme de classes d'OpenNas

En terme de « patron de conception » [75], la structure d'OpenNas correspond à un composite, où la classe *Block* est le composant, la classe *Elementary block* est la feuille et la classe *Composed block* est le composite.

Ces classes ne présentent que l'espace de recherche (chapitre 2). Pour faire une optimisation, il faut donc y ajouter un algorithme d'optimisation et une méthode d'évaluation des architectures. En utilisant le framework Python de DL « Keras » [17], la classe Neural Network s'interface de la façon suivante :

— L'algorithme d'optimisation propose une architecture sous la forme d'un entier x.

— La méthode build_network de la classe Neural Network est lancée avec le paramètre x. Cela permet de créer les blocs correspondant à l'architecture x.

— La méthode run du bloc de plus haut niveau est lancée pour créer un CNN à l'aide de Keras de manière fonctionnelle [16].

La section suivante présente des expérimentations menées à l'aide de cette implémentation.

4.3 . Expérimentations

Dans cette section sont décrites les différentes expérimentations menées pendant cette thèse. Le but de ces expérimentations fut de prouver la viabilité de l'espace de recherche présenté au chapitre 2 et les stratégies d'optimisation présentées au chapitre 3. Pour ce faire, trois études ont été menées.

La première (section 4.3.1) est une étude sur MNIST [48] avec un espace de recherche simple, pour vérifier si la méthode converge bien vers une solution. La deuxième tente, sur plusieurs jeux de données, de trouver une architecture similaire à VGG-16 mais qui aurait de meilleures performances. Enfin, la dernière étude compare les résultats de notre méthode à d'autres méthodes de NAS, dont des résultats sur Cifar-10 [45] sont disponibles publiquement.

4.3.1 . Étude de convergence

Une première évaluation a été menée à l'aide du jeu de données MNIST [48] qui est l'un des premiers et plus simples jeux de données pour la classification d'images (voir section 1.1.4). Sa simplicité fait que MNIST ne nécessite pas d'architecture trop profonde. Cela permet une recherche sur des architectures rapides à entraîner, donc un temps expérimental plus court.

Les deux algorithmes d'optimisation utilisés pour cette expérimentation sont l'algorithme génétique U-NSGA-III et l'optimisation bayésienne. Le premier a été choisi pour sa facilité de mise en place lors d'une optimisation multi-objectifs et car l'algorithme génétique est déjà largement utilisé dans le domaine de la recherche d'architecture (voir section 1.2.3). L'optimisation bayésienne a quant à elle été testée car elle est un algorithme populaire dans l'optimisation des hyperparamètres afin de voir sa viabilité pour la recherche d'architecture.

Choix de l'optimisation en simple boucle

Lors de l'optimisation en double boucle (section 3.3) avec un algorithme génétique, la boucle interne qui optimise la configuration renvoie une population. Cela implique que le nombre d'objectifs de la boucle externe est égal au nombre d'individus d'une population de la boucle interne. Pour éviter cela, on peut sélectionner un individu de la boucle interne comme candidat idéal à renvoyer à la boucle externe. Pour ce faire, une méthode est l'*Augmented Scalarization Function* [87], en choisissant arbitrairement une importance à l'exactitude et au nombre de paramètres, similairement à la somme pondérée pour l'optimisation bayésienne (section 4.3.1). De ce fait, l'optimisation en double boucle paraît peu adaptée avec un algorithme génétique.

Ce problème ne se pose pas avec une optimisation mono-objectif telle que l'optimisation bayésienne, puisque l'évaluation ne renvoie qu'un seul individu (et non pas une population). Cependant, pour que les expériences soient comparables, nous devons utiliser la même stratégie d'optimisation. Nous décidons donc, dans les deux cas, d'utiliser la stratégie d'optimisation en simple boucle (section 3.4).

Paramètres de l'espace de recherche imbriqué

Les expériences ont été menées avec des blocs élémentaires simples, contraignant peu la recherche. C'est-à-dire que l'espace de recherche contient beaucoup d'architectures possibles. Le but de ces expériences est de vérifier si l'application d'algorithmes d'optimisation sur l'espace de recherche converge bien vers une solution, dans le cadre d'une optimisation bi-objectifs selon l'exactitude (soit la fonction objectif définie par l'équation 3.11) et le nombre de paramètres d'entraînement (soit la fonction objectif définie par l'équation 3.8). Pour le nombre de paramètres, nous ajoutons la contrainte d'égalité 3.4 avec $\mathcal{P}_{max} = 10\ 000\ 000$. Les CNN proposés ne peuvent donc pas dépasser les dix millions de paramètres.

Pour cela, deux études sont envisagées. Tout d'abord, une étude des fronts de Pareto [33], puis une somme pondérée des deux objectifs permettant de les fusionner. Les caractéristiques de l'espace de recherche imbriqué sont les suivantes :

1. Les blocs élémentaires correspondent respectivement à une des opérations suivantes :
   - $e_1 =$ Une convolution 2D avec un noyau de 1×1.
   - $e_2 =$ Une convolution 2D avec un noyau de 3×3.
   - $e_3 =$ Une convolution 2D avec un noyau de 5×5.
2. Les options sont distribuées équitablement dans les pipelines, par l'algorithme 3.1, et peuvent prendre des valeurs appartenant à $\{0, 1, 2\}$.
3. Le niveau supérieur $\mathbb{P}_1$ est constitué de pipelines ayant au maximum 2 blocs internes.
4. Le niveau suivant $\mathbb{R}_2$ est constitué de rubans ayant au maximum 2 blocs internes.
5. Enfin, le niveau $\mathbb{P}_3$ forme le dernier pipeline représentant le réseau et peut avoir un maximum de 3 blocs internes.

Ainsi, l'espace de recherche comporte 609 180 agencements de blocs, soit 1 827 540 possibilités d'architecture (en prenant en compte les options).

À la fin de l'architecture définie par l'espace de recherche imbriqué, un module de classification est ajouté. Ce module est constitué d'une couche FC de 10 neurones (un pour chaque classe de MNIST) avec une activation $\mathrm{softmax}$. Ce choix de n'avoir qu'une seule couche FC, contrairement à LeNet [48] (voir section 1.1.4), se justifie par le fait que les couches de FC possèdent beaucoup de paramètres d'entraînement. Avoir moins de FC permet donc déjà de réduire le nombre de paramètres, qui est un des objectifs de l'optimisation.

Concernant les hyperparamètres d'entraînement, l'optimisation des poids est faite par l'algorithme « Adam » [44], avec une taille de lot (*batch size*) de 512, sur 3 itérations.

L'entraînement est fait sur si peu d'itérations pour réduire le temps d'expérimentation, les itérations d'entraînement étant la partie la plus chronophage de la recherche d'architecture. Cependant, MNIST étant un jeu de données simple, 3 itérations permettent déjà d'avoir une convergence.

Front de pareto avec UNSGA-III

Dans le cas de plusieurs fonctions objectifs, le front de Pareto [33] permet de visualiser les solutions les plus optimales en prenant en compte ces deux fonctions. Dans la figure 4.2 les fronts de Pareto « dominent » les solutions sur le graphique et les solutions sur le front de Pareto rouge (ronds) sont les solutions optimales (non dominées). Pour optimiser les fronts de Pareto, autrement dit, réduire l'aire sous la courbe, l'algorithme utilisé est le Unified Non-dominated Sorting Genetic Algorithm III (U-NSGA-III) [73]. U-NSGA-III est une amélioration de NSGA-III ayant de meilleures performances pour le mono-objectif et le bi-objectif, contrairement à NSGA-III qui n'est efficace qu'à partir de trois objectifs. Cet algorithme est un algorithme génétique (section 1.2.3) appliqué aux fronts de Pareto. L'implémentation de cet algorithme à été faite avec la librairie Python « Pymoo » [8].

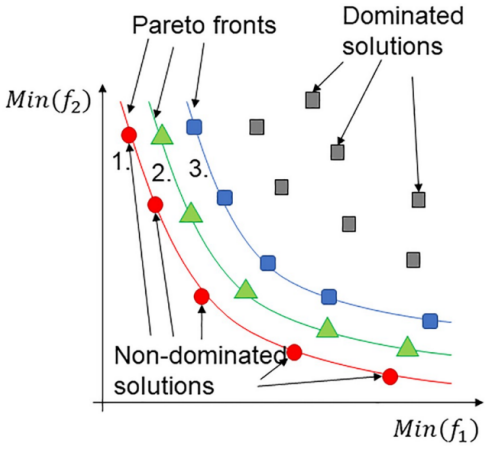


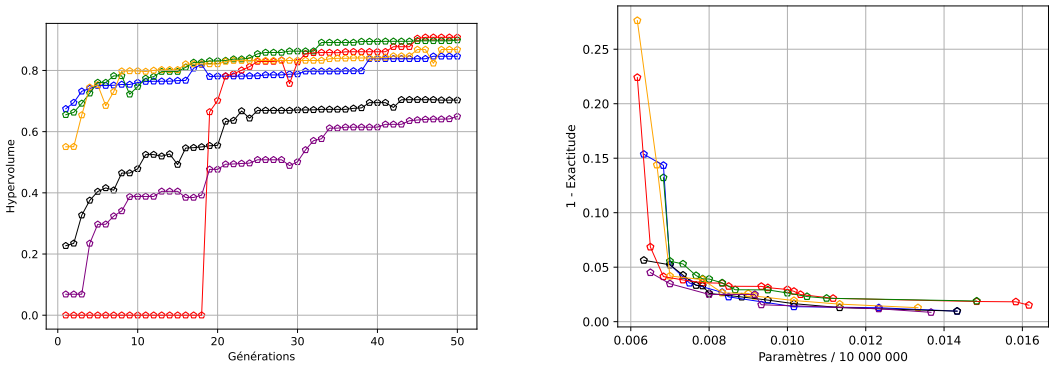
Figure 4.2 – Exemple de fronts de Pareto. Extrait de [4]

Pour cette expérience, les différents paramètres d'optimisation sont les suivants :

1. Nombre d'individus par population : 40
2. Nombre de générations : 50
3. Probabilité de croisement [2] : 0,5
4. Probabilité de mutation [2] : 0,9

2. Paramètres par défaut de Pymoo.

L'expérience a été répétée 6 fois avec les mêmes paramètres (hormis la graine aléatoire). La figure 4.3 présente les résultats de ces expériences. Le critère d'évaluation utilisé est l'hypervolume. Il s'agit, avec deux objectifs, de l'aire au dessus du front de Pareto, par rapport à un point de référence. Dans notre cas, ce point de référence est $(1, 1)$, ce point étant le plus élevé atteignable lors d'une évaluation avec une exactitude de 0 et un nombre de paramètres d'entraînement supérieur ou égal à $\mathcal{P}_{max} = 10\ 000\ 000$.



(a) Hypervolume en fonction des générations

(b) Fronts de Pareto

Figure 4.3 – Résultats de l'application de U-NSGA-III sur l'espace de recherche sur 6 itérations

Les résultats des six expériences sont montrés dans la figure 4.3. La première figure 4.3a, montre l'hypervolume de chaque expérience au fur et à mesure des générations. Nous pouvons constater que, à chaque itération d'expérience, l'optimisation fonctionne, car l'hypervolume augmente. En revanche, elles ne convergent pas au même hypervolume. De même, la figure 4.3b, illustrant les fronts de Pareto finaux de chaque itération, montre que, bien que proches, chaque illustration n'offre pas exactement la même solution.

On peut aussi remarquer que ces expériences n'ont sûrement pas fini de converger. C'est-à-dire qu'en laissant plus de temps (plus de générations) à la recherche, celle-ci devrait converger vers des hypervolumes plus élevés. Ceci n'a pas été plus développé dans cette thèse pour laisser du temps à d'autres expérimentations.

Ainsi, ces expériences permettent de valider que l'application d'un algorithme génétique sur l'espace de recherche permet d'obtenir un front de Pareto avec un hypervolume plus élevé. Cependant, pour converger vers un optimal, la recherche a besoin de plus de générations.

Dans la section suivante, nous faisons la même expérience, mais avec un algorithme d'optimisation bayésienne.

Somme pondérée avec optimisation bayésienne

Une autre manière d'optimiser plusieurs fonctions objectifs, est de transformer le problème multi-objectif en un problème mono-objectif. Pour ce faire, on peut faire une somme pondérée des fonctions objectifs :

$$\mathcal{F}(X) = \sum_i \alpha_i f_i(X), \text{ avec } \sum_i \alpha_i = 1 \tag{4.1}$$

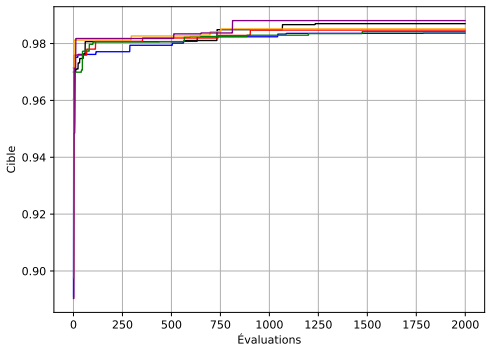
On a donc une seule fonction objectif $\mathcal{F}$ et les coefficients α_i permettent de définir une importance à chaque fonction objectif. Plus α_i est élevé, plus la fonction f_i est optimisée en priorité. On appelle cette fonction « cible » (*target*).

Ainsi, avec une unique fonction objectif, il est possible d'appliquer des algorithmes d'optimisation mono-objectif. Dans notre cas, nous utilisons un algorithme d'optimisation bayésienne [28], à l'aide de la librairie Python « bayesian-optimization » [60]. L'optimisation bayésienne est une des méthodes les plus utilisées pour l'optimisation des hyperparamètres, mais peut aussi être utilisée pour le NAS [77, 42].

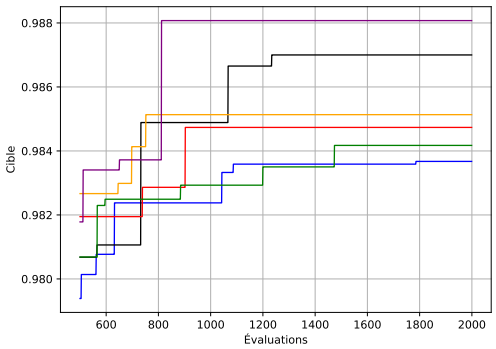
Pour cette expérience, les paramètres d'optimisation sont les suivants :

1. Nombre d'évaluations aléatoires initiales : 500
2. Nombre d'évaluations pendant l'optimisation : 1500

Ainsi, le nombre d'évaluations totales est de 2000. Il y a donc autant d'évaluation dans cette expérience que dans la précédente (40 individus par population $\times$ 50 générations).



(a) Cible maximum atteinte par évaluation



(b) Cible maximum atteinte par évaluation pendant la partie optimisation (après les 500 évaluations aléatoires)

Figure 4.4 – Résultats de l'application de l'optimisation bayésienne sur l'espace de recherche sur 6 itérations

La figure 4.4 montre la cible (équation 4.1) maximum trouvée en fonction du nombre d'évaluations effectuées. Nous pouvons voir, dans la figure 4.4a, que la cible dépasse rapi-

dement 0,98 pendant la recherche aléatoire. Ainsi l'optimisation arrive bel et bien à trouver une solution (pendant la recherche aléatoire), puis trouve des solutions avec une meilleure cible pendant l'optimisation bayésienne. La figure 4.4b se concentre sur la partie optimisation bayésienne (après 500 évaluations), et montre que l'optimisation bayésienne peut converger rapidement vers une solution. En effet, en observant les résultats décrits dans la table 4.1, on remarque que quatre itérations sur six ont fini de converger avant d'atteindre 1250 évaluations. Cependant, les deux itérations suivantes non seulement arrêtent de converger plus tard (respectivement 1473 et 1785 évaluations), mais les cibles atteintes sont les plus faibles de toutes les itérations.

Évaluation	Cible maximum	Écart au maximum
751	0.985134	-2.94×10^{-3}
811	0.988074	0
902	0.984735	-3.34×10^{-3}
1233	0.987000	-1.07×10^{-3}
1473	0.984173	-3.90×10^{-3}
1785	0.983673	-4.40×10^{-3}

Table 4.1 – Évaluation à laquelle la cible maximum de chaque itération est atteinte et comparaison de cette cible maximum entre les différentes itérations par rapport à l'itération ayant la plus haute cible maximum.

Conclusion

Ces deux expériences permettent de voir que l'application d'algorithmes d'optimisations sur notre espace de recherche, avec la stratégie présentée en section 3.4.1, converge bien vers une solution. Par là même, cela montre qu'il est possible d'utiliser différents algorithmes d'optimisation.

La table 4.2 présente la cible maximum atteinte en moyenne pour les six expériences (et sa variance), ainsi que la cible maximum atteinte lors de la meilleure des itérations (ligne « maximum ») et celle atteinte lors de la pire des itérations (ligne « minimum »). En étudiant ces résultats, nous remarquons que les résultats obtenus par U-NSGA-III sont légèrement supérieurs à l'optimisation bayésienne. Et ce, qu'il s'agisse de la moyenne, du maximum, ou du minimum ; et la variance plus faible pour U-NSGA-III prouve aussi sa plus grande stabilité. Cependant cette différence n'est pas significative (inférieure à 0,2%).

Un avantage de U-NSGA-III est qu'il optimise directement les fronts de Pareto. Le résultat final est donc un front de Pareto, ce qui permet de laisser un plus grand choix de solutions. À contrario, l'optimisation bayésienne nécessite de choisir à l'avance l'importance de chaque objectif et ne fournit in fine qu'une seule solution.

Dans la prochaine section, nous utilisons l'algorithme bayésien en mono-objectif pour améliorer l'architecture VGG (section 1.1.4) sur CIFAR-10.

	U-NSGA-III	Bayésien	Comparaison
Moyenne	$0,98666$	$0,98546$	$+0,121\%$
Maximum	$0,98888$	$0,98807$	$+0,082\%$
Minimum	$0,98374$	$0,98367$	$+0,007\%$
Variance	$3,74890 \times 10^{-6}$	$3,86707 \times 10^{-6}$	$-3,152\%$

Table 4.2 – Comparaison entre l'application de U-NSGA-III et de l'optimisation bayésienne à notre espace de recherche.

4.3.2 . Étude d'amélioration de performances

Cette seconde évaluation a pour but de tester la capacité de notre espace de recherche à améliorer des architectures spécifiques, dans ce cas VGG [76] (voir section 1.1.4). Pour ce faire, les blocs élémentaires utilisés ont été définis de la même manière que des blocs de VGG-16 (figure 1.12b). On a donc comme paramètres pour l'espace de recherche :

1. Blocs élémentaires $\mathbb{E}$:
   - $e_1(x_r, x_p)$: conv3×3 $\rightarrow$conv3×3
   - $e_2(x_r, x_p)$: conv3×3 $\rightarrow$conv3×3 $\rightarrow$conv3×3

   où $(x_r, x_p) \in \{0,1\}^2$.
2. Les options sont ici intégré intégré aux blocs élémentaires pour créer le niveau de blocs composé $\mathbb{P}_1$ à partir de l'ensemble de blocs $\mathbb{E}' = \{e_1(0,0), e_1(0,1), e_1(1,0), e_1(1,1), e_2(0,0), e_2(0,1), e_2(1,0), e_2(1,1)\}$, avec une multiplicité de $m_1 = 5$.

Ainsi, l'espace de recherche comporte 1364 possibilités d'architecture.

À la fin du modèle, le même module de classification que dans VGG-16 est ajouté. C'est-à-dire deux FC de 4096 neurones suivis d'une $\mathbf{ReLU}$ puis une FC avec $N_{classes} = 10$ neurones suivit d'un $\mathrm{softmax}$.

L'étude a été faite sur CIFAR-10 [45], avec les paramètres d'entraînement suivants :

1. Taille de lot (*batch size*) : 64
2. Nombre d'itérations (*epochs*) : 10
3. Algorithme de descente du gradient : Adam [44]

Concernant l'algorithme d'optimisation bayesienne, de même que pour la section 4.3.1, la librairie « bayesian-optmization » a été utilisée. Les paramètres de l'optimisation sont les suivants : Pour cette expérience, les paramètres d'optimisation sont les suivants :

1. Nombre d'évaluations aléatoires initiales : 50
2. Nombre d'évaluations pendant l'optimisation : 300

Résultats

La recherche d'architecture a duré environ 15 heures, avec un seul GPU. L'architecture retenue par l'optimisation est donnée dans la figure 4.5. Les blocs élémentaires sont illustrés à gauche, puis (au milieu) l'architecture est d'abord illustrée du point de vue du pipeline de $\mathbb{P}_1$ formé par l'agencement des blocs élémentaires e_1 et e_2, ainsi que les options les configurant. Enfin (à droite), l'architecture retenue est décrite explicitement par les opérations qui la composent.

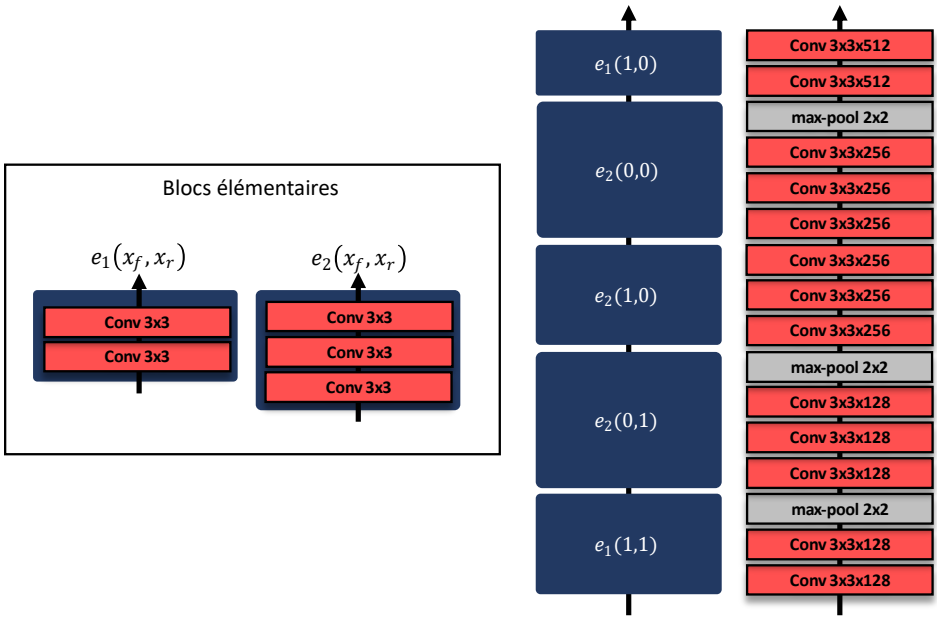


Figure 4.5 – Architecture retenue par l'optimisation bayesienne

Cette architecture a ensuite été entraînée pendant 100 itérations sur le jeu de données afin d'en améliorer l'exactitude. À la suite de cet entraînement, le modèle a atteint une exactitude de 94,62%. Dans la table 4.3, ce résultat est comparé à des architectures développées manuellement. Ces architectures sont des adaptations de VGG, optimisées pour CIFAR-10. Nous pouvons voir que l'architecture trouvée par notre optimisation est plus performante que ces dernières.

4.3.3 . Comparaison à d'autres Auto-MLs

Dans cette section, nous recherchons une architecture adaptée à CIFAR-10 et comparons les résultats obtenus avec ceux d'autres techniques de NAS [24, 63, 67, 12, 22]. Pour cette recherche, les paramètres de l'espace de recherche sont les suivants :

1. L'ensemble des blocs élémentaires $\mathbb{E}$ est constitué des blocs suivants :
   — e_1 = Une convolution 2D avec un noyau de 1×1.

Model de CNN	Exactitude
OpenNas	$94,62\%$
VGG-13 [5]	$93,65\%$
SGR [30]	$93,54\%$
CIFAR-VGG [54]	$91,55\%$

Table 4.3 – Comparaison de l'exactitude atteinte avec l'architecture proposée par OpenNas avec des modèles développés manuellement.

- $e_2 =$ Une convolution 2D avec un noyau de 3×3.
- $e_3 =$ Une convolution 2D avec un noyau de 5×5.

2. Les options sont distribuées équitablement dans les pipelines, par l'algorithme 3.1, et peuvent prendre des valeurs appartenant à $\{0, 1, 2, 3\}$.
3. Le niveau supérieur $\mathbb{P}_1$ est constitué de pipelines ayant au maximum 2 blocs internes.
4. Le niveau suivant $\mathbb{R}_2$ est constitué de rubans ayant au maximum 3 blocs internes.
5. Enfin, le niveau $\mathbb{P}_3$ forment le dernier pipeline représentant le réseau et peut avoir un maximum de 3 blocs internes.

Ainsi, l'espace de recherche comporte 48 361 404 agencements de blocs, soit 435 252 636 possibilités d'architectures (en prenant en compte les options).

À la fin du modèle, un module de classification constitué d'une couche de FC avec $N_{classes} = 10$ neurones est ajouté.

Les paramètres d'entraînement sont les suivants :

1. Taille de lot (*batch size*) : 512
2. Nombre d'itérations (*epochs*) : 20, avec un arrêt précoce (*early stopping*) si l'exactitude de validation n'augmente plus pendant deux itérations.
3. Algorithme de descente du gradient : Adam [44]

L'algorithme d'optimisation utilisé est U-NSGA-III, car cette optimisation est multi-objectif (sur l'exactitude équation 3.11 et sur le nombre de paramètres équation 3.8). De plus, LEMONADE [24], une des méthodes de NAS avec lequel nous nous comparons, utilise aussi un algorithme génétique. L'optimisation a été faite avec une population de 32 individus sur 150 générations.

Résultats

L'optimisation a duré pendant 4 jours avec une RTX 6000. Dans la figure 4.6, on voit l'évolution du front de Pareto au fur et à mesure des générations. Plus le nombre de générations avance et plus le nombre d'architectures sur le front de Pareto augmente. De plus, les architectures moins optimisées sont remplacé par de nouvelles, avec un meilleur rapport exactitude/paramètres.

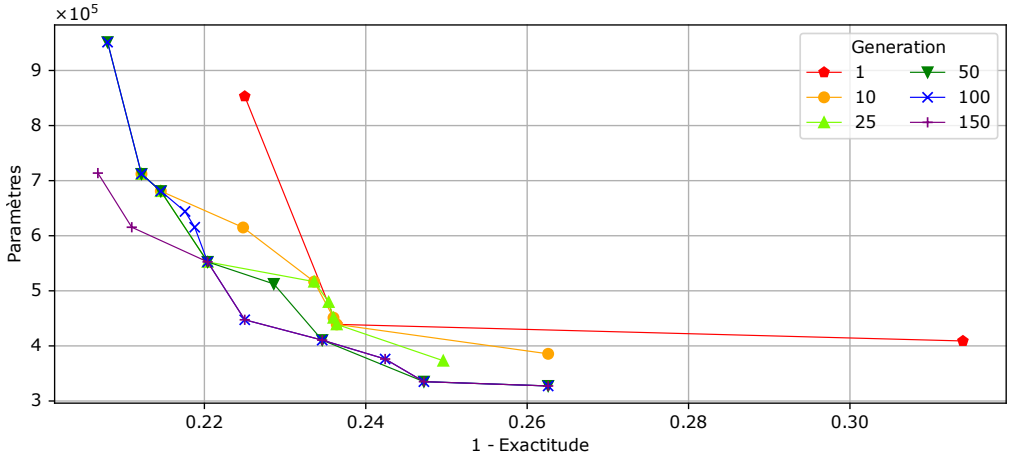


Figure 4.6 – Évolution des fronts de Pareto au fur et à mesure des générations.

En comparant la figure 4.8 (qui est la figure 4.6 mise à l'échelle logarithmique) avec la figure 4.7 (qui est extraite de la publication sur LEMONADE [24]), on remarque que notre méthode compétitive en terme de nombre de paramètres, mais donne de moins bons résultats concernant l'exactitude. De plus, le temps d'optimisation est très réduit pour notre méthode comparé à LEMONADE : 4 jours GPU comparé à 80 jours GPU pour LEMONADE.

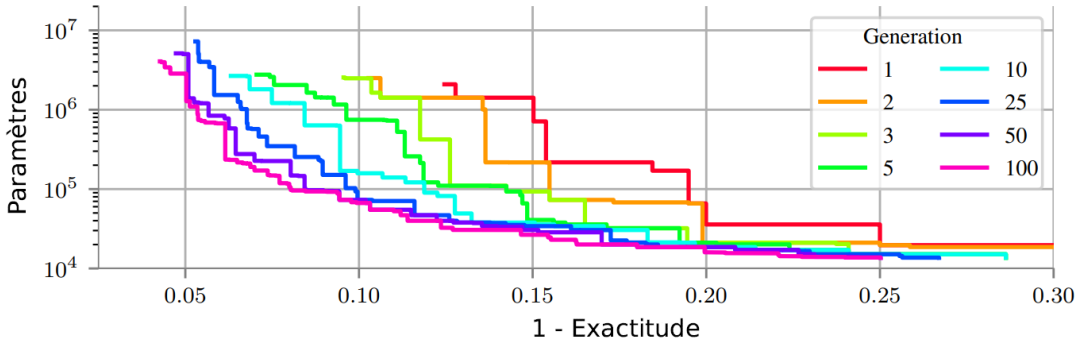


Figure 4.7 – Évolution des fronts de Pareto au fur et à mesure des générations de LEMONADE. Extrait de [24].

Des suites de cette optimisation, chaque architecture du front de Pareto final est ré-entraînée avec 100 itérations au lieu de 20. Ce nouvel entraînement est fait avec de l'augmentation de données et l'ajout de régularisation (*batch-normalization*) afin d'obtenir de meilleurs résultats (ces techniques sont aussi utilisées dans les techniques de NAS comparées).

La table 4.4 présente les résultats obtenus par quatre méthodes de NAS : DPP-Net [22], ENAS [63], PLNT [12] et LEMONADE [24]. La table présente différents résultats pour chacune de ces méthodes où une erreur ($1-$exactitude de test) correspond à un nombre de

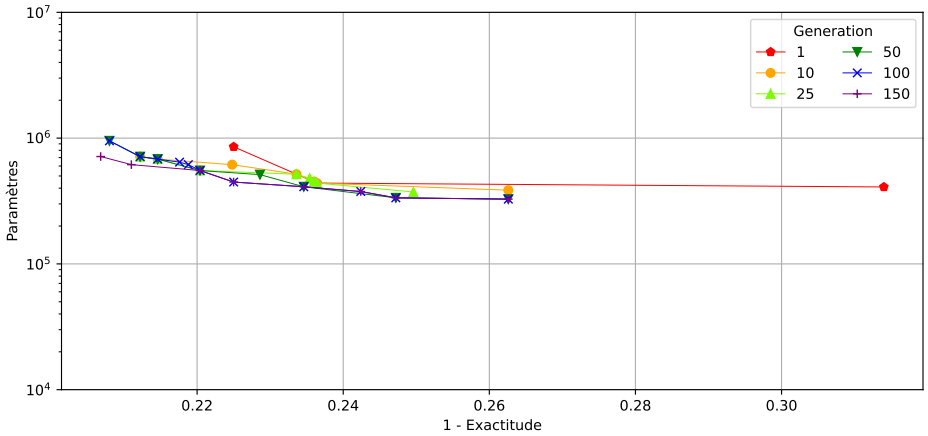


Figure 4.8 – Mise à l'échelle logarithmique de la figure 4.6

paramètres du modèle de CNN.

On peut voir que l'erreur obtenue par les architectures trouvées par OpenNas est largement supérieure à celle trouvée par les autres NAS de la littérature. Cependant il y a une meilleure optimisation du point de vue du nombre de paramètres : 400 000 pour OpenNas contre 500 000 au mieux pour DPP-NET et LEMONADE. De plus, OpenNAs n'a eu besoin que de 4 jours de GPU, contre 8 pour DPP-Net, 24 pour LEMONADE et 2000 pour NasNet. Cela est cependant plus que les 16 heures de GPU nécessaires à ENAS.

NAS	Paramètres	Erreur (%)
OpenNas	$0,4$M	$16,91$
DPP-Net	$0,5$M	$4,62$
LEMONADE	$0,5$M	$4,57$
DPP-Net	$1,0$M	$4,78$
LEMONADE	$1,1$M	$3,69$
NASNet	$3,3$M	$2,65$
ENAS	$4,6$M	$2,89$
PLNT	$5,7$M	$2,49$
LEMONADE	$4,7$M	$3,05$
DPP-Net	$11,4$M	$4,36$
PLNT	$14,3$M	$2,30$
LEMONADE	$13,1$M	$2,58$

Table 4.4 – Comparaison de OpenNas avec d'autres méthodes de NAS sur le jeu de données CIFAR-10.

Un *fine-tuning* plus poussé des hyperparamètres ainsi que l'ajout de plus de régularisation, permettant plus d'itérations d'entraînement sans surentraînement, permettrait sûrement d'améliorer l'exactitude sans changer le modèle. Une autre piste est d'augmenter l'espace de recherche avec plus de couches, par exemple en augmentant la multiplicité des pipelines de $\mathbb{P}_3$.

4.4 . Conclusion

Dans ce chapitre, nous avons présenté trois expériences utilisant le *framework* OpenNas montrant la viabilité et les avantages d'OpenNas en pratique. Le premier avantage est qu'il permet facilement d'appliquer un algorithme d'optimisation quelconque, grâce à l'encodage des architectures. De plus, la taille de l'espace de recherche est facilement contrôlable, en jouant sur le nombre de niveaux d'imbrication et la multiplicité des blocs composés. Enfin, il permet d'orienter la recherche vers des architectures particulières en définissant des blocs élémentaires spécifiques.

La première expérience (section 4.3.1) a permis de valider que l'utilisation du framework est bien capable de générer un modèle de réseaux de neurones fonctionnel. En effet, même lors de la recherche aléatoire de l'optimisation bayésienne (section 4.3.1), l'algorithme bayésien proposait déjà des solutions avec une cible à plus de 98%. De plus, nous avons vu que les différentes optimisations méta-heuristiques (U-NSGA-III et l'optimisation bayésienne) permettaient bel et bien d'augmenter respectivement l'hypervolume des fronts de Pareto et la cible, tout en minimisant le nombre de paramètres.

La deuxième expérience (section 4.3.2) est une optimisation de VGG sur le jeu de données CIFAR-10. Lors de cette expérience, nous avons pu montrer que OpenNas peut être utilisé pour améliorer une architecture connue. En effet, en définissant les blocs élémentaires tels que les blocs composant VGG-16 (figure 1.12b), nous avons pu trouver une architecture semblable à VGG, mais ayant une exactitude d'environ 1% plus élevée que VGG-13 [5].

La dernière expérience (section 1.2) effectue une recherche d'architecture bi-objectifs (exactitude maximisée et nombre de paramètres minimisé) et compare les résultats à d'autres méthodes de NAS. Cette expérience a montré que, bien que OpenNas pouvait plus être rapide et trouver des architectures avec un nombre de paramètres d'entraînement plus faible par rapport aux autres Auto-ML, les architectures trouvées ont une exactitude plus faible que les autres méthodes de NAS.

Conclusion et perspectives

Conclusion

Le domaine de l'intelligence artificielle, plus précisément de l'Apprentissage profond / *Deep Learning* (DL), a connu un renouveau dans les années 2010 et cet essor semble même s'intensifier dans cette décennie de 2020. De ce fait, le DL touche des corps de métier de plus en plus différents et des personnes ne maîtrisant pas encore cette technologie peuvent être amenées à devoir développer des modèles de DL. Pour répondre à cela, la discipline du Apprentissage automatique automatisé / *Automated Machine Learning* (Auto-ML) vise à automatiser la création de modèles de DL.

Lors de cette thèse, nous nous sommes intéressés à la partie Recherche d'architecture neuronale / *Neural Architecture Search* (NAS) de l'Auto-ML, qui consiste à trouver une architecture de réseau de neurones optimale pour un jeu de données. Notre but est de fournir un cadre outillé de NAS adaptable. C'est-à-dire laissant un maximum de libertés quant à la définition de l'espace recherche (orienter la recherche vers un type d'architecture ou au contraire laisser la recherche plus libre), mais aussi pouvant être appliquée avec différents algorithmes d'optimisation. Nous avons nommé ce cadre outillé « OpenNas ».

La première contribution de cette thèse, présentée dans le chapitre 2, est donc une formalisation d'un espace de recherche permettant cet adaptabilité. Le principe de cet espace de recherche se base sur un principe d'imbrications. Des blocs élémentaires définis par l'utilisateur (par exemple des opérations de convolution) permettent de former des blocs composés en les assemblant en série (créant un « pipeline ») ou en parallèle (créant un « ruban »). Ces blocs composés peuvent ensuite à leur tour être assemblés pour former d'autres blocs composés de plus haut niveau d'imbrication, et ainsi de suite ; formant in fine un réseau de neurones. Ainsi, en définissant le nombre et les types (pipeline ou ruban) de niveaux d'imbrication, de même que le nombre de blocs utilisés pour former chaque bloc composé, il est aisément possible de contrôler le nombre de degrés de liberté de l'espace de recherche.

Un autre avantage de la formalisation proposée est de pouvoir encoder les blocs. Cela signifie que les blocs composés sont désignés par un entier naturel, puis les niveaux supérieurs de blocs composés peuvent à leur tour être désignés par une liste de ces entiers. Ensuite, un encodage de cette liste peut être fait pour la convertir en un simple entier naturel. Ainsi, une architecture de CNN complète peut être définie par un entier naturel.

Grâce à l'encodage, l'espace de recherche est facilement adaptable à une technique d'optimisation (algorithme génétique, optimisation bayésienne, apprentissage par renforcement, etc.). En effet, cela permet de n'avoir qu'un seul paramètre à optimiser.

Une seconde contribution de cette thèse, présentée dans le chapitre 3, est la proposi-

tion de trois stratégies d'optimisation à appliquer sur l'espace de recherche précédemment formalisé. Chaque nouvelle stratégie réduit les degrés de libertés de l'espace de recherche afin d'augmenter l'efficacité de l'optimisation. C'est-à-dire pouvoir trouver une architecture optimale en moins de temps.

La première est une optimisation en double boucle, car deux paramètres sont présents dans l'espace de recherche : l'architecture en soi et les « options » servant à contrôler les dimensions des tenseurs le long du modèle. Il s'agit de la stratégie la plus directe, sans choix de la part de l'utilisateur.

La deuxième proposition est une optimisation avec une seule boucle, en rendant l'architecture et les options indépendantes. Pour cela, des algorithmes heuristiques sont appliqués pour distribuer les options.

Enfin, une proposition d'optimisation avec un espace de recherche dynamique est faite. Celle-ci permet de réduire les degrés de liberté en sélectionnant des sous-ensembles de blocs élémentaires et en construisant des architectures à partir de ces derniers.

Pour finir, au chapitre 4, nous faisons une évaluation de OpenNas avec différentes expérimentations. Toutes les expériences ont été menées avec la seconde stratégie d'optimisation présentée précédemment.

Nous avons tout d'abord comparé l'algorithme d'optimisation génétique U-NSGA-III et un algorithme d'optimisation bayésienne dans le cadre d'une optimisation multi-objectif. Les deux objectifs considérés étaient l'exactitude sur les données de validation et le nombre de paramètres d'entraînement du modèle. Avec cette expérience, nous avons pu voir que ces deux algorithmes étaient similaires en terme de solutions trouvées. Cependant, l'optimisation bayésienne semble converger plus rapidement tandis que U-NSGA-III permet une solution sous forme de front de Pareto.

La deuxième expérience montre comment améliorer une architecture déjà existante. En effet, à l'aide de blocs élémentaires tirés de VGG-16, nous avons pu trouver une architecture « VGG-esque » qui surpasse les architectures VGG-esques définies manuellement.

La dernière expérience est une comparaison de « OpenNas » par rapport à d'autres NAS sur le jeu de données CIFAR-10. Nous avons trouvé que l'application de la deuxième stratégie d'optimisation avec U-NSGA-III a permis de trouver des solutions dont l'exactitude et le nombre de paramètres sont similaires aux autres NAS, mais dans un temps en moyenne plus restreint.

Ainsi, les bases d'OpenNas ont pu être posées avec cette thèse : la formalisation de l'espace de recherche avec son encodage, et différentes stratégies d'optimisation.

Perspectives

Les perspectives envisageables sont les suivantes :

— Dans un premier temps, la distribution d'OpenNas sous la forme d'un module Python

open-source. Ce module s'accompagnera de configurations par défaut (blocs élémentaires et stratégies d'optimisation) pour différents types de jeux de données afin de faciliter le développement par des utilisateurs non-initiés en DL. Il sera aussi fourni avec des outils de surveillance pour les utilisateurs plus expérimentés. Par exemple, des retours statistiques sur les blocs élémentaires et composés utilisés lors de la recherche, permettant de déterminer quels types de blocs ont le plus d'impact sur les performances.

— Les évaluations ont été faites avec le *framework* de DL Keras. Cependant, l'espace de recherche en soit ne dépend pas d'un *framework* de DL particulier. Ainsi, des API pour utiliser OpenNas avec d'autres *frameworks*. Le premier *framework* envisagé est Pytorch. En effet, il s'agit, avec Tensorflow (dont Keras dépend), d'un des *frameworks* les plus utilisés, notamment pour la recherche.

— Une perspective à plus long terme est la mise en place d'un recueil d'architectures. En effet, lors de la troisième proposition de stratégie d'optimisation, nous avons introduit le principe d'identifiant (section 3.5.1) pour les architectures. Cet identifiant unique ne l'est que dans un espace de recherche donné. Cependant, il serait intéressant de rendre cet identifiant unique peu importe l'espace de recherche. De ce fait, il serait possible de créer un recueil d'architectures avec leurs caractéristiques (paramètres, exactitude sur des jeux de données, FLOPs). Cela permettrait, en y accédant pendant la recherche, d'accélérer le temps d'optimisation en omettant l'entraînement d'architectures déjà présentes dans le recueil.

— Lors de cette thèse, seules des évaluations sur des jeux de données de classification ont été menées. Des évaluations sur d'autres jeux de données, notamment la détection d'objets et la segmentation d'image, peuvent être menées.

— Enfin, le problème d'encodage des rubans (section 2.3.2) reste ouvert. En effet, les seules solutions pour encoder un ruban, c'est-à-dire de convertir un multiensemble en un entier naturel, utilisent un algorithme générant les multiensembles jusqu'à obtenir celui correspondant au ruban et, en comptant le nombre de générations, permet d'obtenir l'entier naturel correspondant. Une perspective est donc la recherche d'un algorithme permettant d'encoder un ruban de manière directe (comme cela a été possible pour les pipelines), sans avoir à générer un grand nombre de rubans.

Liste des publications

Les publications scientifiques produites pendant cette thèse sont les suivantes :

[1] Pouy, L., F. Khenfri, P. Leserf, C. Mhraida C. Larouci. 2022, An approach for efficient neural architecture search space definition, *36th AAAI Conference on Artificial Intelligence*, AAAI-22 Workshop: Learning Network Architecture During Training

[2] Pouy, L., F. Khenfri, P. Leserf, C. Mhraida C. Larouci. 2023, Open-nas: A customizable search space for neural architecture search, *Proceedings of the 2023 8th International Conference on Machine Learning Technologies*, ICMLT '23, Association for Computing Machinery, New York, NY, USA, ISBN 9781450398329, 102–107, doi: 10.1145/3589883.3589898. URL https://doi.org/10.1145/3589883.3589898

Bibliographie

[1] «Connexionnisme», dans *DataFranca*. URL https://datafranca.org/wiki/Connexionnisme.

[2] «Intelligence», dans *Le Robert Dico en ligne*. URL https://dictionnaire.lerobert.com/definition/intelligence.

[3] «Intelligence artificielle symbolique», dans *DataFranca*. URL https://datafranca.org/wiki/Intelligence_artificielle_symbolique.

[4] Abonyi, J., Ipkovich, G. Dörgő et K. Héberger. 2023, «Matrix factorization-based multi-objective ranking–what makes a good university ?», *PLOS ONE*, vol. 18, n° 4, doi: 10.1371/journal.pone.0284078, p. 1–30. URL https://doi.org/10.1371/journal.pone.0284078.

[5] Ayinde, B. O., T. Inanc et J. M. Zurada. 2019, «On correlation of features extracted by deep neural networks», *CoRR*, vol. abs/1901.10900. URL http://arxiv.org/abs/1901.10900.

[6] Baker, B., O. Gupta, N. Naik et R. Raskar. 2017, «Designing neural network architectures using reinforcement learning», dans *International Conference on Learning Representations*. URL https://openreview.net/forum?id=S1c2cvqee.

[7] Barnier, N. et P. Brisset. 1999, «Optimisation par algorithme génétique sous contraintes», *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, vol. 18, n° 1, p. pp 1–29. URL https://hal-enac.archives-ouvertes.fr/hal-00934534.

[8] Blank, J. et K. Deb. 2020, «pymoo : Multi-objective optimization in python», *IEEE Access*, vol. 8, p. 89 497–89 509.

[9] Brostow, G. J., J. Fauqueur et R. Cipolla. 2008, «Semantic object classes in video : A high-definition ground truth database», *Pattern Recognition Letters*.

[10] Caesar, H., V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan et O. Beijbom. 2020, «nuscenes : A multimodal dataset for autonomous driving», dans *CVPR*.

[11] Cai, H., J. Yang, W. Zhang, S. Han et Y. Yu. 2018, «Path-level network transformation for efficient architecture search», *CoRR*, vol. abs/1806.02639. URL http://arxiv.org/abs/1806.02639.

[12] Cai, H., J. Yang, W. Zhang, S. Han et Y. Yu. 2018, «Path-level network transformation for efficient architecture search», .

[13] Chen, T., I. J. Goodfellow et J. Shlens. 2016, «Net2net : Accelerating learning via knowledge transfer», dans *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, édité par Y. Bengio et Y. LeCun. URL http://arxiv.org/abs/1511.05641.

[14] Chen, X., L. Xie, J. Wu et Q. Tian. 2019, «Progressive differentiable architecture search : Bridging the depth gap between search and evaluation», dans *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, p. 1294–1303, doi: 10.1109/ICCV.2019.00138.

[15] Chen, Z., F. Zhou, G. Trimponias et Z. Li. 2020, «Multi-objective neural architecture search via non-stationary policy gradient», *CoRR*, vol. abs/2001.08437. URL https://arxiv.org/abs/2001.08437.

[16] Chollet, F. 2020, «The Functional API», URL https://keras.io/guides/functional_api/.

[17] Chollet, F. et collab.. 2015, «Keras», https://keras.io.

[18] Cordts, M., M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth et B. Schiele. 2016, «The cityscapes dataset for semantic urban scene understanding», dans *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Darwin, C. 1859, *On the Origin of Species by Means of Natural Selection*, Murray, London. Or the Preservation of Favored Races in the Struggle for Life.

[20] Deb, K., A. Pratap, S. Agarwal et T. Meyarivan. 2002, «A fast and elitist multiobjective genetic algorithm : Nsga-ii», *IEEE Transactions on Evolutionary Computation*, vol. 6, n° 2, doi: 10.1109/4235.996017, p. 182–197.

[21] Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li et L. Fei-Fei. 2009, «Imagenet : A large-scale hierarchical image database», dans *2009 IEEE conference on computer vision and pattern recognition*, Ieee, p. 248–255.

[22] Dong, J.-D., A.-C. Cheng, D.-C. Juan, W. Wei et M. Sun. 2018, «Dpp-net : Device-aware progressive search for pareto-optimal neural architectures», .

[23] Ehrlich, G. 1973, «Loopless algorithms for generating permutations, combinations, and other combinatorial configurations», *J. ACM*, vol. 20, n° 3, doi: 10.1145/321765.321781, p. 500–513, ISSN 0004-5411. URL https://doi.org/10.1145/321765.321781.

[24] Elsken, T., J. H. Metzen et F. Hutter. 2019, «Efficient multi-objective neural architecture search via lamarckian evolution», dans *International Conference on Learning Representations*. URL https://openreview.net/forum?id=ByME42AqK7.

[25] Elsken, T., J. H. Metzen et F. Hutter. 2019, «Neural architecture search : A survey», *Journal of Machine Learning Research*, vol. 20, n° 55, p. 1–21. URL http://jmlr.org/papers/v20/18-598.html.

[26] Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn et A. Zisserman. 2010, «The pascal visual object classes (voc) challenge», *International Journal of Computer Vision*, vol. 88, n° 2, p. 303–338.

[27] Fang, J., Y. Sun, Q. Zhang, Y. Li, W. Liu et X. Wang. 2020, «Densely connected search space for more flexible neural architecture search», dans *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[28] Frazier, P. I. 2018, «A tutorial on Bayesian optimization», .

[29] Garrison, E. 2010, «multichoose», URL https://github.com/ekg/multichoose/blob/master/multichoose.py.

[30] Geifman, Y. et R. El-Yaniv. 2017, «Selective classification for deep neural networks», *CoRR*, vol. abs/1705.08500. URL http://arxiv.org/abs/1705.08500.

[31] Geiger, A., P. Lenz et R. Urtasun. 2012, «Are we ready for autonomous driving ? the kitti vision benchmark suite», dans *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Godbole, V., G. E. Dahl, J. Gilmer, C. J. Shallue et Z. Nado. 2023, «Deep learning tuning playbook», URL http://github.com/google/tuning_playbook, version 1.0.

[33] Goodarzi, E., M. Ziaei et E. Z. Hosseinipour. 2014, *Introduction to Optimization Analysis in Hydrosystem Engineering*, Springer International Publishing, doi: 10.1007/978-3-319-04400-2. URL https://doi.org/10.1007%2F978-3-319-04400-2.

[34] Goodfellow, I., Y. Bengio et A. Courville. 2016, *Deep Learning*, MIT Press. http://www.deeplearningbook.org.

[35] Géron, A. 2019, *Machine Learning avec Scikit-Learn - 2e édition*, ISBN 9782100797820.

[36] Hasanpour, S. H., M. Rouhani, M. Fayyaz et M. Sabokrou. 2016, «Lets keep it simple : using simple architectures to outperform deeper and more complex architectures», .

[37] He, K., X. Zhang, S. Ren et J. Sun. 2016, «Deep residual learning for image recognition», dans *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 770–778, doi: 10.1109/CVPR.2016.90.

[38] He, X., K. Zhao et X. Chu. 2021, «AutoML : A survey of the state-of-the-art», *Knowledge-Based Systems*, vol. 212, doi: https://doi.org/10.1016/j.knosys.2020.106622, p. 106 622, ISSN 0950-7051. URL https://www.sciencedirect.com/science/article/pii/S0950705120307516.

[39] Horváth, G. et S. Tengely. 2013, «Lecture notes for college discrete mathematics», .

[40] Hsu, C., S. Chang, D. Juan, J. Pan, Y. Chen, W. Wei et S. Chang. 2018, «MONAS : multi-objective neural architecture search using reinforcement learning», *CoRR*, vol. abs/1806.10332. URL http://arxiv.org/abs/1806.10332.

[41] Huang, G., Z. Liu, L. Van Der Maaten et K. Q. Weinberger. 2017, «Densely connected convolutional networks», dans *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 2261–2269, doi: 10.1109/CVPR.2017.243.

[42] Kandasamy, K., W. Neiswanger, J. Schneider, B. Poczos et E. P. Xing. 2018, «Neural architecture search with Bayesian optimisation and optimal transport», dans *Advances in Neural Information Processing Systems*, vol. 31, édité par S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi et R. Garnett, Curran Associates, Inc. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf.

[43] Kelleher, J. 2005, *Encoding Partitions As Ascending Compositions*, thèse de doctorat, University College Cork. URL http://jeromekelleher.net/downloads/k06.pdf.

[44] Kingma, D. P. et J. Ba. 2017, «Adam : A method for stochastic optimization», .

[45] Krizhevsky, A. 2009, «Learning multiple layers of features from tiny images», .

[46] Krizhevsky, A., I. Sutskever et G. E. Hinton. 2012, «Imagenet classification with deep convolutional neural networks», dans *Advances in Neural Information Processing Systems*, vol. 25, édité par F. Pereira, C. Burges, L. Bottou et K. Weinberger, Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[47] Krogh, A. et J. Hertz. 1991, «A simple weight decay can improve generalization», dans *Advances in Neural Information Processing Systems*, vol. 4, édité par J. Moody, S. Hanson et R. Lippmann, Morgan-Kaufmann. URL https://proceedings.neurips.cc/paper/1991/file/8eefcfdf5990e441f0fb6f3fad709e21-Paper.pdf.

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

[49] Li, L. et A. Talwalkar. 2020, «Random search and reproducibility for neural architecture search», dans *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, *Proceedings of Machine Learning Research*, vol. 115, édité par R. P. Adams et V. Gogate, PMLR, p. 367–377. URL https://proceedings.mlr.press/v115/li20c.html.

[50] Lin, M., Q. Chen et S. Yan. 2014, «Network in network», dans *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, édité par Y. Bengio et Y. LeCun. URL http://arxiv.org/abs/1312.4400.

[51] Liu, C., L. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille et L. Fei-Fei. 2019, «Auto-deeplab : Hierarchical neural architecture search for semantic image segmentation», *CoRR*, vol. abs/1901.02985. URL http://arxiv.org/abs/1901.02985.

[52] Liu, H., K. Simonyan, O. Vinyals, C. Fernando et K. Kavukcuoglu. 2018, «Hierarchical representations for efficient architecture search», dans *International Conference on Learning Representations*. URL https://openreview.net/forum?id=BJQRKzbA-.

[53] Liu, H., K. Simonyan et Y. Yang. 2019, «DARTS : Differentiable architecture search», dans *International Conference on Learning Representations*. URL https://openreview.net/forum?id=S1eYHoC5FX.

[54] Liu, S. et W. Deng. 2015, «Very deep convolutional neural network based image classification using small training sample size», dans *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, p. 730–734, doi: 10.1109/ACPR.2015.7486599.

[55] Lu, Z., I. Whalen, V. Boddeti, Y. D. Dhebar, K. Deb, E. D. Goodman et W. Banzhaf. 2018, «NSGA-NET : A multi-objective genetic algorithm for neural architecture search», *CoRR*, vol. abs/1810.03522. URL http://arxiv.org/abs/1810.03522.

[56] Mahony, N. O., S. Campbell, A. Carvalho, S. Harapanahalli, G. A. Velasco-Hernández, L. Krpalkova, D. Riordan et J. Walsh. 2019, «Deep learning vs. traditional computer vision», *CoRR*, vol. abs/1910.13796. URL http://arxiv.org/abs/1910.13796.

[57] Miikkulainen, R., J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy et B. Hodjat. 2019, «Chapter 15 - evolving deep neural networks», dans *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, édité par R. Kozma, C. Alippi, Y. Choe et F. C. Morabito, Academic Press, ISBN 978-0-12-815480-9, p. 293–312, doi: https://doi.org/10.1016/B978-0-12-815480-9.00015-3. URL https://www.sciencedirect.com/science/article/pii/B9780128154809000153.

[58] Minaee, S. 2019, «20 popular machine learning metrics. Part 1 : Classification regression evaluation metrics», *Towards Data Science*. URL https://towardsdatascience.com/20-popular-machine-learning-metrics-part-1-classification-regression-evaluation-me

[59] Mittal, A. 2021, «Understanding RNN and LSTM», URL https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e.

[60] Nogueira, F. 2014–, «Bayesian Optimization : Open source constrained global optimization tool for Python», URL https://github.com/fmfn/BayesianOptimization.

[61] Nolan, C. 2010, *Inception*, Warner Bros.

[62] Pham, H., M. Guan, B. Zoph, Q. Le et J. Dean. 2018, «Efficient neural architecture search via parameters sharing», dans *Proceedings of the 35th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 80, édité par J. Dy et A. Krause, PMLR, p. 4095–4104. URL https://proceedings.mlr.press/v80/pham18a.html.

[63] Pham, H., M. Y. Guan, B. Zoph, Q. V. Le et J. Dean. 2018, «Efficient neural architecture search via parameter sharing», *CoRR*, vol. abs/1802.03268. URL http://arxiv.org/abs/1802.03268.

[64] Pouy, L., F. Khenfri, P. Leserf, C. Mhraida et C. Larouci. 2022, «An approach for efficient neural architecture search space definition», dans *36th AAAI Conference on Artificial Intelligence*, AAAI-22 Workshop : Learning Network Architecture During Training.

[65] Pouy, L., F. Khenfri, P. Leserf, C. Mhraida et C. Larouci. 2023, «Open-nas : A customizable search space for neural architecture search», dans *Proceedings of the 2023 8th International Conference on Machine Learning Technologies*, ICMLT '23, Association for Computing Machinery, New York, NY, USA, ISBN 9781450398329, p. 102–107, doi: 10.1145/3589883.3589898. URL https://doi.org/10.1145/3589883.3589898.

[66] Putman, J. 2020, «weakcomp», URL https://gist.github.com/Katsutoshii/8c02ed8eb6977c25a3c7acd4952e4c8a.

[67] Radford, A., J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger et I. Sutskever. 2021, «Learning transferable visual models from natural language supervision», *CoRR*, vol. abs/2103.00020. URL https://arxiv.org/abs/2103.00020.

[68] Real, E., A. Aggarwal, Y. Huang et Q. V. Le. 2019, «Regularized evolution for image classifier architecture search», dans *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence*

Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, AAAI'19/IAAI'19/EAAI'19, AAAI Press, ISBN 978-1-57735-809-1, doi: 10.1609/aaai.v33i01.33014780. URL https://doi.org/10.1609/aaai.v33i01.33014780.

[69] Real, E., S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le et A. Kurakin. 2017, «Large-scale evolution of image classifiers», dans *Proceedings of the 34th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 70, édité par D. Precup et Y. W. Teh, PMLR, p. 2902–2911. URL https://proceedings.mlr.press/v70/real17a.html.

[70] Rosenblatt, F. 1957, «The perceptron - a perceiving and recognizing automaton», cahier de recherche 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York.

[71] Ruder, S. 2016, «An overview of gradient descent optimization algorithms», *arXiv preprint arXiv :1609.04747*.

[72] Rumelhart, D. E., G. E. Hinton et R. J. Williams. 1986, «Learning representations by back-propagating errors», *Nature*, vol. 323, n° 6088, doi: 10.1038/323533a0, p. 533–536, ISSN 1476-4687.

[73] Seada, H. et K. Deb. 2015, «U-nsga-iii : A unified evolutionary optimization procedure for single, multiple, and many objectives : Proof-of-principle results», ISBN 978-3-319-15891-4, p. 34–49, doi: 10.1007/978-3-319-15892-1_3.

[74] Sherstinsky, A. 2020, «Fundamentals of recurrent neural network (RNN) and Long Short-Term Memory (LSTM) network», *Physica D : Nonlinear Phenomena*, vol. 404, doi: https://doi.org/10.1016/j.physd.2019.132306, p. 132 306, ISSN 0167-2789. URL https://www.sciencedirect.com/science/article/pii/S0167278919305974.

[75] Shvets, A. 2018, «Plongée au cœur des patrons de conception», *Refactoring. Guru*.

[76] Simonyan, K. et A. Zisserman. 2015, «Very deep convolutional networks for large-scale image recognition», dans *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, édité par Y. Bengio et Y. LeCun. URL http://arxiv.org/abs/1409.1556.

[77] Snoek, J., H. Larochelle et R. P. Adams. 2012, «Practical Bayesian optimization of machine learning algorithms», .

[78] Spencer, H. 1864, *The principles of biology*, n° vol. 1 dans Spencer, Herbert : A system of synthetic philosophy, Williams and Norgate, p. 444. URL https://books.google.fr/books?id=Ye4GAAAAcAAJ.

[79] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever et R. Salakhutdinov. 2014, «Dropout : A simple way to prevent neural networks from overfitting», *Journal of Machine Learning Research*, vol. 15, n° 56, p. 1929–1958. URL http://jmlr.org/papers/v15/srivastava14a.html.

[80] Stanley, K., J. Clune, J. Lehman et R. Miikkulainen. 2019, «Designing neural networks through neuroevolution», *Nature Machine Intelligence*, vol. 1, doi: 10.1038/s42256-018-0006-z.

[81] Stanley, K. O., D. B. D'Ambrosio et J. Gauci. 2009, «A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks», *Artificial Life*, vol. 15, n° 2, doi: 10.1162/artl.2009.15.2.15202, p. 185–212, ISSN 1064-5462. URL https://doi.org/10.1162/artl.2009.15.2.15202.

[82] Stanley, K. O. et R. Miikkulainen. 2002, «Evolving neural networks through augmenting topologies», *Evolutionary Computation*, vol. 10, n° 2, p. 99–127. URL http://nn.cs.utexas.edu/?stanley:ec02.

[83] Sutton, R. S. et A. G. Barto. 2018, *Reinforcement Learning : An Introduction*, 2e éd., The MIT Press. URL http://incompleteideas.net/book/the-book-2nd.html.

[84] Sze, V., Y.-H. Chen, T.-J. Yang et J. S. Emer. 2017, «Efficient processing of deep neural networks : A tutorial and survey», *Proceedings of the IEEE*, vol. 105, n° 12, doi: 10.1109/JPROC.2017.2761740, p. 2295–2329.

[85] Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke et A. Rabinovich. 2014, «Going deeper with convolutions», *CoRR*, vol. abs/1409.4842. URL http://arxiv.org/abs/1409.4842.

[86] Tan, M., B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard et Q. V. Le. 2019, «Mnasnet : Platform-aware neural architecture search for mobile», dans *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[87] Wu, M., K. Li, S. Kwong et Q. Zhang. 2020, «Evolutionary many-objective optimization based on adversarial decomposition», *IEEE Transactions on Cybernetics*, vol. 50, n° 2, doi: 10.1109/TCYB.2018.2872803, p. 753–764.

[88] Xie, L. et A. L. Yuille. 2017, «Genetic CNN», *CoRR*, vol. abs/1703.01513. URL http://arxiv.org/abs/1703.01513.

[89] Yosinski, J., J. Clune, Y. Bengio et H. Lipson. 2014, «How transferable are features in deep neural networks ?», dans *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, MIT Press, Cambridge, MA, USA, p. 3320–3328.

[90] Zhang, X., H. Xu, H. Mo, J. Tan, C. Yang, L. Wang et W. Ren. 2021, «Dcnas : Densely connected neural architecture search for semantic image segmentation», .

[91] Zhong, Z., J. Yan et C. Liu. 2017, «Practical network blocks design with q-learning», *CoRR*, vol. abs/1708.05552. URL http://arxiv.org/abs/1708.05552.

[92] Zhong, Z., J. Yan, W. Wu, J. Shao et C.-L. Liu. 2018, «Practical block-wise neural network architecture generation», dans *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, p. 2423–2432, doi: 10.1109/CVPR.2018.00257.

[93] Zimmer, L., M. Lindauer et F. Hutter. 2021, «Auto-pytorch tabular : Multi-fidelity metalearning for efficient and robust autodl», .

[94] Zoph, B. et Q. Le. 2017, «Neural architecture search with reinforcement learning», dans *International Conference on Learning Representations*. URL https://openreview.net/forum?id=r1Ue8Hcxg.

[95] Zoph, B., V. Vasudevan, J. Shlens et Q. V. Le. 2018, «Learning transferable architectures for scalable image recognition», dans *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, p. 8697–8710, doi: 10.1109/CVPR.2018.00907.

A1 - Générateur de multiensemble

Cet algorithme est une implémentation de l'*algorithm 7* développé par Ehrlich dans [23]. L'implémentation à été réalisée par Garrison [29]. Cet algorithme est utilisé dans l'algorithme 2.6.

Remarque : Dans le module python *itertools*, la fonction `combinations_with_replacement` est équivalente à l'algorithme A1.1.

Algorithme A1.1 : Générateur de multiensembles

Données : Un nombre k indiquant la taille des multiensembles, un uplet $objects$ représentant l'ensemble support
Résultat : Un multiensemble X

```
1  j, j_1, q ← k, k, k
2  r ←longueur(objects) − 1
3  A ← [0] ∗ k  # Indexs initiaux des multiensembles
4  X ← [0] ∗ k  # Intitialisation de X
5  tant que Vrai faire
6  |  pour i de 0 à k − 1 faire
7  |  |  X[i] ← objects[A[i]]
8  |  fin
9  |  céder X  #Renvoie le résultat (yield en python)
10 |  j ← k − 1
11 |  tant que (j ≥ 0) et (A[j] = r) faire
12 |  |  j ← j − 1
13 |  fin
14 |  si j < 0 alors
15 |  |  sortir # Sort de la boucle tant que
16 |  fin
17 |  j_1 ← j
18 |  tant que j_1 ≤ k − 1 faire
19 |  |  A[j_1] ← A[j_1] + 1
20 |  |  q ← j_1
21 |  |  tant que q < k − 1 faire
22 |  |  |  A[q + 1] ← A[q]
23 |  |  |  q ← q + 1
24 |  |  fin
25 |  |  q ← q + 1
26 |  |  j_1 ← q
27 |  fin
28 |  retourner A
29 fin
```

A2 - Générateur de décomposition faible

Cet algorithme est tiré de l'algorithme « weakcomps » de Putman [66]. Il est utilisé dans l'algorithme 2.7.

Algorithme A2.1 : Générateur de compositions faibles

Données : Un nombre k représentant la taille des compositions faibles de n, le nombre n

Résultat : Une composition faible C

```
1  m ← n + k − 1
2  pour T dans combinations([0, 1, ..., m], k − 1) faire
3  |  U ← T + (m, )
4  |  V ← (−1, ) + T
5  |  C ← [ ]
6  |  pour i de 0 à k − 1 faire
7  |  |  ajouter U[i] − v[i] − 1 à C
8  |  fin
9  |  céder C
10 fin
```

A3 - Générateur de compositions

Cet algorithme est tiré de l'algorithme 5.5 de [43]). Il est disponible sur le site internet suivant : https ://jeromekelleher.net/category/combinatorics.html (consulté le 06/06/2023). Cet algorithme génère des combinaisons dans l'ordre ascendant. Il est utilisé dans l'algorithme 2.8.

Algorithme A3.1 : accel_asc

Données : Un nombre n
Résultat : Une composition de n dans un tableau A

```
1  k ← 2
2  A_1 ← 0
3  y ← n − 1
4  tant que k ≠ 1 faire
5  |  k ← k − 1
6  |  x ← A_k + 1
7  |  tant que 2x ≤ y faire
8  |  |  A_k ← x
9  |  |  y ← y − x
10 |  |  k ← k + 1
11 |  fin
12 |  l ← k + 1
13 |  tant que x ≤ y faire
14 |  |  A_k ← x
15 |  |  A_l ← y
16 |  |  céder [A_1, A_2, ..., A_l]
17 |  |  x ← x + 1
18 |  |  y ← y − 1
19 |  fin
20 |  y ← y + x − 1
21 |  A_k ← y + 1
22 |  céder [A_1, A_2, ..., A_k]
23 fin
```