



**Institut Nationale des Sciences Appliqués et des Technologies**  
**UNIVERSITE DE CARTHAGE**

---

# **STAGE**

**Génie Logiciel**

**RN quantifié pour un contexte deep learning embarqué**

---

**Author:**

**Rami ZOUARI**

**Prof. Meriem JAIDANE**  
Supervisor, ENIT, Tunisia

**Prof. Yosra BEN JEMAA**  
Supervisor, ENIS, Tunisia

**2021/2022**

## **Abstract**

Graph creation is an extremely important part of modern systems. Allowing faster and more flexible data manipulations later on. Choosing the right graph creation strategy will cut operational time to a minimum without exhausting the memory.

This report details our journey from presenting the urge to use different creation strategies to the proof of concept that demonstrate its utility.

### **Acknowledgements**

We would like to express our deep gratitude to our advisors, Lilia SFAXI and Mariem LOUKIL, who gave us the chance to work on this interesting project, and their expertise was extremely valuable in formulating the research questions and methodology. Your insightful comments pushed us to refine our thinking and took our work to the next level. We were able to step out of our comfort zone while working on this original topic, and for that we are very grateful.

# Contents

<b>List of Figures</b>	<b>5</b>
<b>List of Tables</b>	<b>6</b>
<b>Introduction</b>	<b>7</b>
<b>1 Problématique</b>	<b>8</b>
1.1 Introduction . . . . .	8
1.2 Contexte . . . . .	9
1.2.1 Histoire de l'apprentissage approfondie . . . . .	9
1.2.2 Le coût de l'IA . . . . .	10
1.2.3 Les limites des systèmes embarqués . . . . .	11
1.3 Le Binary Neural Network . . . . .	12
1.3.1 Introduction . . . . .	12
1.3.2 La signification de "Binary" . . . . .	12
1.4 BinaryFlow . . . . .	13
1.4.1 Introduction . . . . .	13
1.4.2 Raison d'exister . . . . .	13
1.4.3 Signification du nom . . . . .	13
1.4.4 Solutions offertes par BinaryFlow . . . . .	13
<b>2 Approche Théorique</b>	<b>15</b>
2.1 Introduction . . . . .	15
2.2 Formalisation . . . . .	15
2.2.1 Notations . . . . .	15
2.2.2 Définition d'un BNN . . . . .	17
2.2.3 Objectif . . . . .	18
2.3 Entraînement . . . . .	19
2.3.1 Difficulté du problème discret . . . . .	19
2.3.2 Gradient Nul . . . . .	19
2.4 Optimisations . . . . .	21
2.4.1 Définitions . . . . .	21
2.4.2 Codage sur 1bit . . . . .	21
2.4.3 Utilisation de XNOR . . . . .	21
2.4.4 Utilisation de POPCOUNT . . . . .	23
2.4.5 Gain Temps & Mémoire . . . . .	25

<b>3 Modélisation</b>	<b>26</b>
3.1 Introduction . . . . .	26
3.1.1 Importance de ce chapitre . . . . .	26
3.1.2 Convention Utilisée . . . . .	26
3.2 Histoire . . . . .	27
3.3 Comparaison . . . . .	28
3.3.1 Par quantification . . . . .	28
3.3.2 Par architecture . . . . .	28
3.4 BinaryNet . . . . .	29
3.4.1 Conception . . . . .	29
3.4.2 Topologie . . . . .	29
3.4.3 Utilisation d'Estimateur Direct de gradient . . . . .	29
3.4.4 Optimisations possible . . . . .	31
3.5 XNOR-NET . . . . .	32
3.5.1 Conception . . . . .	32
3.5.2 Topologie . . . . .	32
3.5.3 Réduction de l'erreur de quantification . . . . .	32
3.5.4 Quantification optimale d'un vecteur . . . . .	32
3.5.5 Quantification d'un produit scalaire de deux vecteurs . . . . .	34
3.5.6 Quantification d'une couche dense par produits scalaires . . . . .	36
3.5.7 Quantification d'une couche convolutionnelle par produits scalaires . . . . .	36
3.5.8 Quantification d'une opération bilinéaire quelconque . . . . .	38
3.6 ABCNet . . . . .	41
3.6.1 Conception . . . . .	41
3.6.2 Topologie . . . . .	41
3.6.3 Objectif . . . . .	41
3.6.4 Choix de binarisations de $\mathbf{W}^{(l)}$ . . . . .	42
3.6.5 Choix de binarisations de $a^{(l)}$ . . . . .	43
3.6.6 Quantification par tenseur binaire . . . . .	43
3.6.7 Quantification par produit scalaire . . . . .	43
3.7 BiRealNet . . . . .	45
3.7.1 Conception . . . . .	45
3.7.2 Topologie . . . . .	45
3.7.3 Block BiRealNet . . . . .	45
3.7.4 Liaison résiduelle . . . . .	46
3.7.5 Choix de la quantification . . . . .	46
<b>4 Implémentation</b>	<b>47</b>
4.1 Introduction . . . . .	47
4.2 Paradigmes . . . . .	48
4.2.1 Dans l'implémentation de BinaryFlow . . . . .	48
4.2.2 Dans l'utilisation de BinaryFlow . . . . .	48
4.2.3 Dans l'extension de BinaryFlow . . . . .	49
4.3 Conception . . . . .	49
4.4 Couches . . . . .	50
4.4.1 BinaryNet . . . . .	51
4.4.2 XnorNet . . . . .	53
4.4.3 ABCNet . . . . .	54

4.5	Block	55
4.6	Binarisations	56
4.6.1	Propagation en Avant	56
4.6.2	Propagation en arrière	58
4.6.3	Implémentation	58
4.7	Régularisations	59
4.7.1	Erreur de quantification des noeuds	59
4.7.2	Erreur de quantification des poids	59
4.7.3	Erreur de quantification de l'opération bilinéaire	59
<b>5</b>	<b>Analyse</b>	<b>60</b>
5.1	Introduction	60
5.2	Méthodologie adoptée	61
5.2.1	CRISP-DM	61
5.3	Régression de $f : x \rightarrow 2x^2 + 3x + 2$ sur $[-4, 4]$	62
5.3.1	Importance	62
5.3.2	Jeux de données	62
5.3.3	Modèle	62
5.3.4	Hypothèses	62
5.3.5	Problème Formel	62
5.3.6	Entraînement	62
5.3.7	Performances de prédiction	63
5.4	Classification MNIST	65
5.4.1	Introduction	65
5.4.2	Topologie	65
5.4.3	Modèle	66
5.4.4	Performances de prédictions	66
5.4.5	Performance mémoire	66
5.4.6	Performance temps	67
5.5	Classification Free Spoken Digits	69
5.5.1	Importance	69
5.5.2	Introduction	69
5.5.3	Structure	70
5.5.4	Analyse	70
<b>6</b>	<b>Déploiement</b>	<b>71</b>
6.1	Methodology	71
6.1.1	Selection	72
6.1.2	Preprocessing and feature selection	72
6.1.3	Transformation	75
6.1.4	Data Mining	75
6.1.5	Model Selection	76
6.2	Results	77
6.2.1	Train test split	77
6.2.2	Cross Validation	77
6.3	Final Model	78
6.4	Conclusion	79

<b>Conclusion</b>	<b>80</b>
<b>Bibliography</b>	<b>83</b>

# List of Figures

1.1	Lee Sedol analysant le tablier après sa première victoire du match	10
2.1	Un exemple d'une binarisation et sa dérivée	20
2.2	Table de multiplication en utilisant XNOR	21
2.3	Calcul du produit scalaires	23
3.1	Exemple d'un BinaryConnect à une seule couche cachée	29
3.2	Estimateur direct du gradient	30
3.3	Utilisation de facteurs $\alpha$ et $\beta$ dans une couche dense	36
3.4	Utilisation de facteurs $\alpha$ et $\beta$ dans une couche convolutionnelle	38
3.5	Exemple d'un ABCNet à une seule couche	41
3.6	Exemple d'une liaison résiduelle d'un BiRealNet	45
4.1	Structure de la bibliothèque	49
4.2	Le contenu de <code>layers</code>	50
4.3	BinaryNet	52
4.4	XnorNet	53
4.5	ABCNet	54
4.6	Le contenu de <code>blocs</code> , et de <code>BiRealNet</code>	55
4.7	Traçage des fonction Signe et Heaviside	56
4.8	Traçage de la fonction signe décalée à gauche par $\mu$	57
4.9	Traçage de la fonction signe stochastique	57
4.10	Diagramme de classe global du module <code>binaryflow.quantizers</code>	58
5.1	Méthodologie CRISP-DM	61
5.2	Courbe de chaque modèle	63
5.3	Des images de MNIST	65
5.4	Architecture du modèle MNIST	65
5.5	Des images de MNIST	69
6.1	Knowledge Discovery in Databases Methodology	71
6.2	Selection stage	72
6.3	Data integration	73
6.4	Linear regression on ideapad machine	75
6.5	Linear regression on legion5 machine	75
6.6	Final Model	78

# List of Tables

1.1	Coût des modèles de IA. . . . .	11
2.1	Terminologie des jeux de données . . . . .	16
2.2	Terminologie de la division en Lots . . . . .	16
2.3	Different machines characteristics . . . . .	17
3.1	Le tableau d'avancement des BNNs . . . . .	27
3.2	Approches de Quantification . . . . .	28
3.3	Approches de Quantification . . . . .	28
4.1	Terminologie de la division en Lots . . . . .	51
4.2	Les Les estimations de gradient présentes . . . . .	58
5.1	Comparaison entre les modèles en utilisant Float32 . . . . .	67
5.2	Comparaison entre les modèles en utilisant Float8 . . . . .	67
5.3	Comparaison entre les modèles en utilisant Float8 . . . . .	67
6.1	Using train test split validation on memory models . . . . .	77
6.2	Using train test split validation on time models . . . . .	77
6.3	Using 10-fold cross validation on memory models . . . . .	78
6.4	Using 10-fold cross validation on time models . . . . .	78

# Introduction

With the rise of big data architectures, the need for creating large graphs has increased dramatically these last few years. And to process these large graphs, hardware limits (Memory, Computational power, ...) started to present a very large obstacle. Our personal computers have limited resources so we can only process very limited graphs.

With this paper, we came across this golden rule: how to create large graphs without exhausting the memory? This question could be answered by designing a couple of strategies for graph creation.

This report defines our approach to implement the different graph creation strategies and eventually our machine learning model. It is organized into 5 chapters:

In the first chapter, we will start by giving a brief history on graph theory, state its importance and then we will modelize our problem mathematically and state our hypothesis and different strategies.

In the second chapter, we will first detail the different paradigms and approaches we followed. Then we will present our system design and explain in details its different components.

In the third chapter, we will start by mentioning the different tools and languages used. And then we will describe our progress and the different challenges we have met.

In the fourth chapter, we will analyze our data in order to create an efficient model later on.

In the fifth chapter, we describe machine learning methodology and its different steps. Present our machine learning model in addition to the results.

# Chapter 1

## Problématique

### 1.1 Introduction

Dans ce chapitre, nous allons

## 1.2 Contexte

### 1.2.1 Histoire de l'apprentissage approfondie

Cette section est basée sur l'article de Tim Dettmers [31]

#### Réseaux de Neurones

L'origine des réseaux de neurones, peut être considéré de Ivakhnenko et Lapa [12] qui ont réussi à implémenter un réseau neuronale à fonctions d'activations polynomiales. Mais ils n'ont pas utilisé la propagation en arrière qui n'était pas répandues dans ces années.

#### Propagation en Arrière

La propagation en arrière était dérivée dans les années 1960 mais sous une forme incomplète et inefficace. En outre, sa forme moderne était dérivée par Linnainmaa dans sa thèse de mastère [20] "Taylor expansion of the accumulated rounding error" en 1970. La propagation ne serait répandues que dans 1985 dans lequel les recherches on montré qu'elle donne des représentations intéressantes des paramètres.

#### L'hiver de l'IA

Malgrès les succès de la propagation en arrière, et son incorporation dans les couches convolutionnelles (LeNet [17]) et récurrentes (spécifiquement LSTM [10]), l'intérêt à l'intelligence artificielle était minimale dans les années 1980-1990, et les avancements dans ce domaine étaient minimales. De plus, les machines à vecteurs de supports (SVM [4]) étaient préférés au lieu des réseaux de neurones vu la complexité de ces derniers.

#### Réapparition grâce aux GPUs

Grâce au progrès des ordinateurs, et l'arrivée des cartes graphiques (GPUs), l'utilisation des réseaux de neurones était de plus en plus pratique, et progressivement, la complexité de ces réseaux croît.

#### Explosion de l'apprentissage approfondie

Le moment décisif à l'apprentissage approfondie était le succès de Krizhevsky, Sutskever et Hinton [16] à créer un réseau de neurones profonds avec lequel ils ont importé la compétition de ILSVRC-2012 ImageNet.

Ce moment a marqué l'abandon l'ingénierie des fonctionnalités et le début de l'apprentissage des fonctionnalités. Et puis, Facebook, Google et Microsoft ont rapidement investi dans la recherche de l'apprentissage approfondie.

#### Un niveau surhumain

Dès les années 2010s, beaucoup de modèles de IA ont montré un niveau surhumain dans des tâches que nous avons cru impénétrable par les ordinateurs.

Comme un exemple pertinent, nous allons parler de la suite Alpha développée par l'équipe DeepMing de Google:

1. Le modèle AlphaGo était le premier agent ayant un élo surhumain, ceci est prouvé par son match historique en 2016 contre Lee Sedol<sup>1</sup> dans lequel AlphaGo a gagné 4-1 contre lui<sup>2</sup>.
2. Le modèle AlphaGo Zero est une amélioration de AlphaGo. La majeure différence est dans son entraînement, qui n'est pas basé sur des jeux entre des humains comme le cas de AlphaGo, mais il s'est entraîné en jouant contre lui-même.
3. Le modèle AlphaZero est encore une amélioration de AlphaGo Zero, qui avait un niveau surhumain dans Go, Shogi et l'échec<sup>3</sup>



Figure 1.1: Lee Sedol analysant le tablier après sa première victoire du match

### 1.2.2 Le coût de l'IA

Cette explosion de l'IA a causé une explosion dans les coûts de leurs entraînement, déploiement et fonctionnement.

#### Le coût économique

Les modèles d'intelligences artificielles les plus robustes nécessitent des ressources énormes pour leur entraînement et même fonctionnement.

En effet:

- L'entraînement de AlphaGo Zero a coûté approximativement 25 millions de dollars [7].
- L'entraînement de NAS a coûté approximativement 3 millions de dollars [22].

<sup>1</sup>Lee Sedol est un des joueurs de Go les plus accomplis dans l'histoire du jeu, avec 18 titres internationaux.

<sup>2</sup>Ce match était une des causes pour laquelle Lee Sedol s'est retiré de Go en 2019. Il a cité que la IA est "une entité qui ne pourrait être vaincue."

<sup>3</sup>Dans l'échec, il a même écrasé Stockfish en 2018 avec un résultat de (+155 -6 =839). Avant l'apparition de AlphaZero, Stockfish était le plus puissant moteur d'échec.

### Le coût énergétique

L'utilisation gigantesque des ressources causent une consommation énorme d'énergie.  
Par exemple:

- Dans le match contre Lee Sedol, AlphaGo a utilisé 1202 GPUs et 176 CPUs. Une estimation de la puissance consommé montre un chiffre colossal de 1 mégawatts [22].
- Pour entraîner NAS, l'énergie totale consommée est estimée à 656347 kWh [29].

### Le coût environnemental

Le coût environnemental des modèles IA avancés est souvent très importants.  
En effet, les estimations de CO<sub>2</sub> générés par ces algorithmes peut dépasser l'ordre de milliers de kilogrammes:

•

Modèle / Objet	Hardware	Puissance (Watt)	Temps de fonctionnement (Heures)	CO <sub>2</sub> équivalents (kg)	Budget cloud (dollars américains)
Transformer <sub>base</sub>	P100x8	1415.78	12	11.79	\$41 - \$410
Transformer <sub>big</sub>	P100x8	1515.43	84	87.08	\$289 - \$981
ELMo	P100x3	1415.78	12	11.79	\$433 - \$1472
BERT <sub>base</sub>	V100x64	12041.51	79	652.27	\$3751 - \$12571
NAS	P100x8	1515.43	274120	284019	\$942973 - \$3201722
AlphaGo		≈ 1000000	960	96000	≈ \$250000000
Climatiseur		840			
Refrigérateur		1233			
Avion SF → NY				900	
Personne durant 1 an				5000	
Voiture durant sa vie				57152	
Maison durant son construction [8]				50000 - 80000	

Table 1.1: Coût des modèles de IA.

#### 1.2.3 Les limites des systèmes embarqués

## 1.3 Le Binary Neural Network

### 1.3.1 Introduction

### 1.3.2 La signification de "Binary"

## 1.4 BinaryFlow

### 1.4.1 Introduction

BinaryFlow est une bibliothèque de Deep Learning basée sur Larq et TensorFlow

### 1.4.2 Raison d'exister

### 1.4.3 Signification du nom

Le choix de nom "BinaryFlow" est influencé de nom "TensorFlow" de la fameuse bibliothèque d'apprentissage automatique et de programmation différentielle qui est créée et maintenue par Google.

TensorFlow se traduit en "flux de tenseurs" qui est l'utilisation excessive des tenseurs - qui sont grosso modo des tableaux numériques multidimensionnels - dans les calculs différentiels.

Par analogie, BinaryFlow se traduit en "flux binaire" qui signifie l'utilisation excessive des opérations booléennes sur des tenseurs binaires.

### 1.4.4 Solutions offertes par BinaryFlow

#### Couches

BinaryFlow supporte toutes les couches offertes par TensorFlow et Larq. Mais il aussi supporte:

- BinaryNet: Dense, ConvND, TransposedConvND, SeparableConvND.
- XnorNet: Dense, ConvND.
- XnorNet++: Dense, ConvND.
- ABCNet: Dense, Conv1D, Conv2D, Conv3D
- BiRealNet: Dense, Conv1D, Conv2D, Conv3D
- MeliusNet: Dense, Conv1D, Conv2D, Conv3D

#### Binarisations

BinaryFlow supporte toutes les binarisations standard offertes par Larq, et les étend en ajoutant:

- Binarisation décalée: qui est une métá-binarisation, qui prend une binarisation  $\Psi$  et donne  $\Psi_\mu = x \rightarrow \Psi(x + \mu)$  avec  $\mu \in \mathbb{R}$  entraînable
- Binarisation stochastique: qui est une métá-binaristation qui prend une distribution de probabilité réelle  $\mathcal{D}$  en paramètre, et une binarisation  $\Psi$ , et donne la fonction:  $x \rightarrow \Psi(x - z)$  avec  $z \sim U$
- Binarisation stochastique décalée

## Optimiseurs

BinaryFlow offre les optimiseurs suivants

- SGD
- Adam[15]
- Bob[9]

## Déploiement

BinaryFlow conserve tous les optimisations offerte par Larq

## Chapter 2

# Approche Théorique

### 2.1 Introduction

In this chapter we will start by presenting the used paradigms and approaches, the global domain Diagram used in the analysis phase. And then we will explain the different components and their behavior.

### 2.2 Formalisation

[14] Le BNN, bien que son objectif est clair, elle n'a pas d'approche triviale pour arriver à un réseau "binaire". Pour cela, on va tout d'abord formaliser notre objectif, est après ça, on va essayer de définir un BNN

#### 2.2.1 Notations

Pour simplifier, nous allons commencer par la terminologie des réseaux à perceptrons multicouches.

## Observations

$r$	nombre d'exemplaires du jeux de données
$\mathbf{X}^{\{k\}}$	l'entrée du $k^{\text{ème}}$ exemplaire
$\mathbf{X} = \left( \mathbf{X}^{\{1\}}, \dots, \mathbf{X}^{\{r\}} \right)$	le tenseur des entrées
$\mathbf{Y} = \left( \mathbf{Y}^{\{1\}}, \dots, \mathbf{Y}^{\{r\}} \right)$	le tenseur des résultats
$\mathbf{U} = \left( \mathbf{U}^{\{1\}}, \dots, \mathbf{U}^{\{r\}} \right)$	le tenseur des observations
$\hat{\mathbf{Y}}$	Une estimation de $\mathbf{Y}$
$\mathcal{L}$	Une fonction objective
$\mathcal{L}(\hat{\mathbf{Y}}, \mathbf{Y})$	L'erreur d'estimation de $Y$

Table 2.1: Terminologie des jeux de données

## Division en Lots

$m$	nombre de lots
$s_k$	la taille du $k^{\text{ème}}$ lot
$\mathbf{X}^{[k]}$	le $k^{\text{ème}}$ lot d'entrées
$\mathbf{Y}^{[k]}$	le $k^{\text{ème}}$ lot de sorties
$\mathbf{U}^{[k]}$	le $k^{\text{ème}}$ lot des observation
$\mathcal{L}(\hat{\mathbf{Y}}^{[k]}, \mathbf{Y}^{[k]})$	l'erreur d'estimation de $k^{\text{ème}}$ lot

Table 2.2: Terminologie de la division en Lots

**Remarque 1** *Par défaut, on va diviser les lots en une taille fixe  $s$ .*

*Comme une exception, le dernier lot va avoir le reste de cette division*

## MLP et CNN

$\mathcal{N}$	Un réseau de neurones
$\mathcal{N}_{\mathbf{W}}$	Un réseau de neurones défini par le tenseur $\mathbf{W}$
$\mathbf{W}^{(l)}$	La matrice de liaison de la $l^{\text{ème}}$ couche
$\sigma^{(l)}$	la fonction d'activation de la $l^{\text{ème}}$ couche
$z^{(l)} = \mathbf{W}^{(l)} \star a^{(l-1)}$	le contenu de la couche $l$ avant l'activation
$a^{(l)} = \sigma^{(l)}(z^{(l)}) = \sigma^{(l)}(\mathbf{W}^{(l)} \star a^{(l-1)})$	le contenu de la couche $l$ après l'activation
$W_{i,j}^{(l)}$	le poids de neurone $a_j^{(l-1)}$ dans la $i^{\text{ème}}$ neurones avant l'activation
$x = a^{(0)}$	L'entrée du réseau de neurones
$\hat{y} = a^{(l)} = \mathcal{N}_{\mathbf{W}}(x)$	La sortie du réseau de neurones
$n_l$	le nombre de neurones de la $l^{\text{ème}}$ couche

Table 2.3: Different machines characteristics

**Remarque 2** Le terme  $\mathbf{W}^{(l)} \star a^{(l-1)}$  est interprété comme:

1. une multiplication matricielle  $\mathbf{W}^{(l)} \cdot a^{(l-1)}$  dans le cas d'une couche dense
2. une opération de convolution  $\mathbf{W}^{(l)} * a^{(l-1)}$  dans le cas d'une couche convolutionnelle

### 2.2.2 Définition d'un BNN

Avant de définir un BNN, nous allons introduire le concept des binarisations et des quantifications:

**Définition 1** Une binarisation est une fonction  $\Psi : \mathbb{R} \rightarrow \mathcal{B}$  avec  $\mathcal{B}$  un ensemble à deux éléments.

Dans le cas d'un tenseur, la binarisation est appliquée élément par élément.

On va distinguer essentiellement deux valeurs particulières de  $\mathcal{B}$  :

- $\mathcal{B} = \{\pm 1\}$
- $\mathcal{B} = \{0, 1\}$

**Définition 2** Un scalaire  $u$  est dit binarisé s'il varie dans un ensemble  $\mathcal{B}$  à deux éléments.

Un tenseur de rang  $r$  et de dimension  $n_1 \times \dots \times n_r$  est dit binarisé s'il varie dans un ensemble  $\mathcal{B}^{n_1 \times \dots \times n_r}$  avec  $\mathcal{B}$  à deux éléments.

**Définition 3** Une quantification est un opérateur  $\Psi$  appliqué à un tenseur juste avant l'opération bilinéaire en conservant ses dimensions.

**Remarque 3** Dans les réseaux de neurones classiques, il n'y a pas de quantifications. En fait, l'absense d'une quantification est équivalente à l'utilisation de la quantification `id` :  $x \rightarrow x$

### Dans la littérature

Le BNN n'admet pas d'une définition unique. En contre partie, dans la littérature[34], la plupart des réseaux sont nommées binaires puisque :

1. Pour chaque paire de noeuds connexe  $(u, v)$ . Le poids de la liaison est  $\omega(u, v) = \pm 1$
2. Pour chaque couche cachée, la fonction d'activation est la fonction signe sign

## Notre définition

Nous avons proposé cette définition pour essaier d'unifier les approches des BNNs:

**Définition 4** *Un BNN est tout réseau de neurones admettant au moins une couche dont les quantifications sont des binarisations*

*Un BNN idéal est un réseau de neurones dont toutes les quantifications sont des binarisations*

**Remarque 4** *Dans ce stage, nous allons concentrer sur les binarisations dont  $\mathcal{B} = \{\pm 1\}$ , et surtout la binarisation sign*

**Remarque 5** *Bien que la définition est un peu permissive, nous n'allons considérer que les BNNs qui ont un nombre "important"<sup>1</sup> de binarisations.*

### 2.2.3 Objectif

#### Notations

- Soit  $\mathcal{D} = (\mathbf{X}, \mathbf{Y})$  le jeu de données
- Soit  $\mathcal{N}_{\mathbf{W}}$  un réseau de neurones admettant le tenseur de paramètres  $\mathbf{W}$ .

#### Problème d'optimisation

L'objectif "idéal" est la résolution du problème d'optimisation suivant:

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} \mathcal{L}(\mathcal{N}_{\mathbf{W}}(\mathbf{X}), \mathbf{Y}) \quad (2.1)$$

L'objectif pour un BNN est aussi la minimisation de (2.1), mais la seule différence est que le réseau admet des binarisations comme quantifications.

---

<sup>1</sup>Grosso modo, le nombre de binarisations doit être aussi important pour justifier les apports des binarisations.

## 2.3 Entraînement

Notre définition de BNN pose beaucoup de problème dans l'entraînement, dont la résolution exige des approches sophistiquées.

### 2.3.1 Difficulté du problème discret

Soit  $\mathcal{N}$  un réseau de neurones binaire idéal ayant une topologie fixe, et soit  $\eta$  le nombre total de ses paramètres binarisés.

Dans le plus mauvais cas, la complexité de trouver les valeurs optimales des paramètres pour ce réseau est  $\mathcal{O}(2^\eta)$  qui n'est pas du tout pratique.

Deux solutions se présentent:

1. Approcher le problème discret avec des algorithmes d'approximations et des heuristiques.
2. Retourner à l'optimisation continue de la fonction objective,

### 2.3.2 Gradient Nul

Nous allons opter pour la deuxième solution dans ce stage. Mais, malheureusement, on ne peut pas appliquer directement les méthodes d'optimisations de premier ordre, ou même d'ordre supérieur. Le problème majeur provient de la lemme suivante:

**Lemme 1** *Toute binarisation continue presque partout<sup>2</sup> admet une dérivée nulle presque partout*

**Preuve 1** *Soit  $\Phi : \mathbb{R} \rightarrow \{a_0, a_1\}$  une binarisation continue partout sauf un ensemble  $\mathcal{D}$  de mesure nulle, avec  $a_0, a_1 \in \mathbb{R}$  et  $a_0 \neq a_1$ .*

*Soit  $x_0 \in \mathbb{R}$  tels que  $\Phi$  est continue en  $x_0$ , et soit  $0 < \epsilon < |a_1 - a_0|$  et  $\delta \in \mathbb{R}_+^*$  tel que*

$$x \in \mathcal{B}(x_0, \delta) \implies |\Phi(x) - \Phi(x_0)| < \epsilon$$

*On a nécessairement,  $\Phi(\mathcal{B}(x_0, \epsilon)) = \{a_i\}$  avec  $i \in \{0, 1\}$ , et donc  $\Phi$  est constante sur  $\mathcal{B}(x_0, \delta)$ , et ainsi dérivable sur cet interval et en particulier en  $x_0$ , et on a:*

$$\Phi'(x_0) = 0$$

*Ainsi:*

$$\forall x \in \mathbb{R}, \quad \Phi \text{ continue en } x \implies \Phi \text{ dérivable en } x \text{ et } \Phi'(x) = 0$$

*Finalement, on a  $\Phi$  est continue sur  $\mathcal{S} = \mathbb{R} \setminus \mathcal{D}$ , avec  $\mathcal{D}$  est de mesure nulle, et donc  $\Phi$  est dérivable sur  $\mathcal{S}$  et par conséquent  $\Phi' = 0$  sur  $\mathcal{S}$  ■*

Cette lemme pose un problème puisque la mise à jour des paramètres sera impossible avec des gradient nuls presque partout.

Pour résoudre ce problème, nous pouvons:

1. Faire une approximation de la binarisation par une fonction plus régulière, surtout dans la propagation en arrière.

---

<sup>2</sup>Le mot "presque partout" est interprété au sens de la théorie des mesures, et précisément la mesure naturelle dans  $\mathbb{R}$  (mesure de Leibniz). Par exemple, dans un sens, la "dérivée" de la fonction sign est la fonction de Dirac  $\delta$

2. Faire une relaxation concernant la notion de gradient à l'aide d'un opérateur  $\mathcal{G}$  adéquat "avec lequel on peut appliquer les méthodes de premier ordre.

Dans le chapitre suivant, nous allons voir chacune des approches 1 et 2.

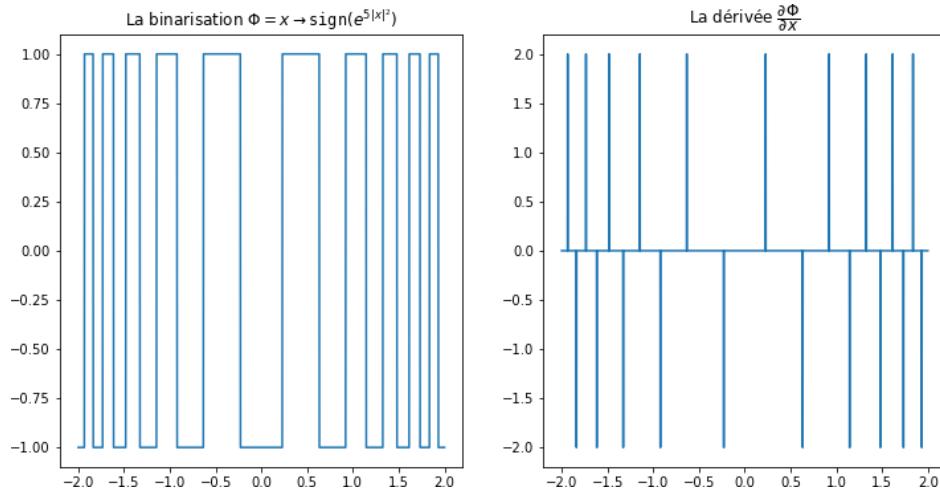


Figure 2.1: Un exemple d'une binarisation et sa dérivée

## 2.4 Optimisations

### 2.4.1 Définitions

On note par:

- $(\mathbb{B}, +, \cdot, \bar{\cdot}, 0, 1)$  une algèbre de boole.
- l'opérateur **XOR** par  $\oplus$  défini par:  

$$a \oplus b = \text{XOR}(a, b) = a \cdot \bar{b} + \bar{a} \cdot b = \bar{a} \oplus \bar{b}$$
- l'opérateur **XNOR** par  $\odot$  défini par:  

$$a \odot b = \text{XNOR}(a, b) = \overline{\text{XOR}(a, b)} = (a + \bar{b})(\bar{a} + b) = a \cdot b + \bar{a} \cdot \bar{b} = \bar{a} \odot \bar{b}$$

### 2.4.2 Codage sur 1bit

Puisque les poids et noeuds des couches cachées sont tous binarisés, on peut coder chaque valeur sur un seul bit. Ainsi on peut introduire le codage:

$$\Psi : \begin{cases} -1 & \rightarrow 0 \\ 1 & \rightarrow 1 \end{cases}$$

En fait, on regroupe chaque 8 variables dans la même case mémoire.

### 2.4.3 Utilisation de XNOR

#### Transformation de $\times$ en XNOR

Dans le cas d'un BNN, les noeuds et les poids sont binarisés. Ainsi toute multiplication est entre deux éléments de  $\{-1, 1\}$ .

Or, par la transformation bijective:

$$\Psi : \begin{cases} 1 & \rightarrow 1 \\ -1 & \rightarrow 0 \end{cases}$$

On a l'égalité suivante:

$$\forall a, b \in \{-1, 1\}, \quad \Psi(a \times b) = \Psi(a) \odot \Psi(b)$$

Encoding (Value)	XNOR (Multiply)
0 (-1)	0 (-1)
0 (-1)	1 (+1)
1 (+1)	0 (-1)
1 (+1)	1 (+1)

Figure 2.2: Table de multiplication en utilisant XNOR

**Preuve 2** On a le groupe  $(\mathbb{C}_2, \times)$  avec  $\mathbb{C}_2 = \{z \in \mathbb{C} / z^2 = 1\} = \{-1, 1\}$  est cyclique et d'ordre 2. Ainsi, il est isomorphe au groupe  $(\mathbb{F}_2, \oplus)$  où  $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$ , et  $\oplus$  est l'addition modulo 2. L'isomorphisme entre eux est:

$$\Psi_1 = \begin{cases} -1 & \rightarrow 1 \\ 1 & \rightarrow 0 \end{cases}$$

De plus, le corps  $(\mathbb{F}_2, \oplus, \times)$  induit l'algèbre de boole  $(\mathbb{F}_2, +, \times, \bar{\phantom{x}}, 0, 1)$  avec:

$$\begin{aligned} \bar{a} &= a \oplus 1 \\ a + b &= \overline{\bar{a} \times \bar{b}} \end{aligned}$$

Or, par dualité de l'algèbre booléenne, l'opérateur de négation  $\Psi_2 : a \rightarrow \bar{a}$  est un isomorphisme entre  $(\mathbb{F}_2, +, \times, \bar{\phantom{x}}, 0, 1)$  et  $(\mathbb{F}_2, \times, +, \bar{\phantom{x}}, 1, 0)$

Ainsi on a:

$$\begin{aligned} \forall a, b \in \{-1, 1\}, \quad \Psi_2 \circ \Psi_1(a \times b) &= \Psi_2(\Psi_1(a \times b)) \\ &= \Psi_2(\Psi_1(a) \oplus \Psi_1(b)) \text{ isomorphisme 1} \\ &= \overline{\Psi_1(a) \oplus \Psi_1(b)} \\ &= \Psi_1(a) \odot \Psi_1(b) \text{ isomorphisme 2} \\ &= \overline{\Psi_1(a)} \odot \overline{\Psi_1(b)} \\ &= \Psi_2 \circ \Psi_1(a) \odot \Psi_2 \circ \Psi_1(b) \end{aligned}$$

Maintenant, il ne reste qu'à vérifier que  $\Psi = \Psi_2 \circ \Psi_1$  :

$$\begin{aligned} \Psi_2 \circ \Psi_1(-1) &= \Psi_2(\Psi_1(-1)) \\ &= \Psi_2(1) \\ &= 0 \\ \Psi_2 \circ \Psi_1(1) &= \Psi_2(\Psi_1(1)) \\ &= \Psi_2(0) \\ &= 1 \quad \blacksquare \end{aligned}$$

## Avantages

En fait, dans les circuits intégrés, l'opération XNOR est plus rapide que  $\times$ . En fait, elle:

- consomme moins d'énergie
- peut être faite 32 fois dans un seul cycle. Par comparaison, une seule opération  $\times$  peut dépasser 1 cycle

#### 2.4.4 Utilisation de POPCOUNT

##### Définition de POPCOUNT

POPCOUNT est une instruction qui permet de compter les bits qui sont mis à 1 dans un vecteur de bits  $v$ .

**Remarque 6**  $v$  peut être la représentation binaire d'un nombre  $n$

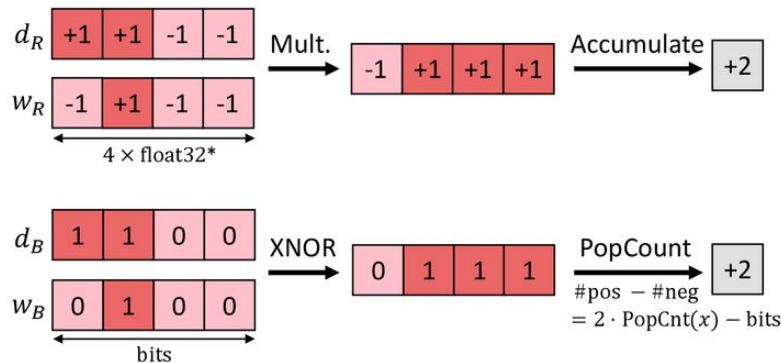
##### Produit scalaire de deux vecteurs binarisés

Soit  $u, v \in \{-1, 1\}^n$ . On a:

$$\begin{aligned}
 \langle u, v \rangle &= \sum_{i=1}^n u_i v_i \\
 &= \sum_{i=1}^n \Psi^{-1}(\Psi(u_i) \odot \Psi(v_i)) \\
 &= \text{POPCOUNT}(\Psi(u) \odot \Psi(v)) \cdot \Psi^{-1}(1) + (n - \text{POPCOUNT}(\Psi(u) \odot \Psi(v))) \cdot \Psi^{-1}(0) \\
 &= \text{POPCOUNT}(\Psi(u) \odot \Psi(v)) - (n - \text{POPCOUNT}(\Psi(u) \odot \Psi(v))) \\
 &= 2 \cdot \text{POPCOUNT}(\Psi(u) \odot \Psi(v)) - n \\
 &= (\text{POPCOUNT}(\Psi(u) \odot \Psi(v)) \ll 1) - n
 \end{aligned}$$

## Convolution with bitwise operations

Multiplication and addition are replaced by bitwise XNOR and PopCount.



\* To enable usage of fast floating-point GPU kernels in PyTorch/Tensorflow during BNN training.

Figure 2.3: Calcul du produit scalaires

### Estimation de coût

On suppose pour  $m \in \mathbb{N}^*$ , que notre processeur supporte dans une seule instruction, un **XNOR** bit-wise des deux vecteurs bits  $u, v \in \{0, 1\}^m$

On suppose aussi pour cette même valeur de  $m$  que notre processeur supporte dans une seule instruction, un **POPCOUNT** bit-wise d'un vecteur bits  $u \in \{0, 1\}^m$

Soit  $n = mr$ ,  $r \in \mathbb{N}^*$ , et soit  $u, v \in \{-1, 1\}^n$

Pour  $p \in \{1, r\}$ , soit:

$$\begin{cases} u^{[p]} = (u_{(p-1)m+1}, \dots, u_{pm}) \\ v^{[p]} = (v_{(p-1)m+1}, \dots, v_{pm}) \end{cases}$$

On a:

$$\begin{aligned} \langle u, v \rangle &= \sum_{i=1}^n u_i v_i \\ &= \sum_{p=1}^r \langle u^{[p]}, v^{[p]} \rangle \\ &= \sum_{p=1}^r 2 \cdot \text{POPCOUNT} \left( \Psi(u^{[p]}) \odot \Psi(v^{[p]}) \right) - m \\ &= 2 \sum_{p=1}^r \text{POPCOUNT} \left( \Psi(u^{[p]}) \odot \Psi(v^{[p]}) \right) - \sum_{p=1}^r m \\ &= \left( \sum_{p=1}^r \text{POPCOUNT} \left( \Psi(u^{[p]}) \odot \Psi(v^{[p]}) \right) \ll 1 \right) - n \end{aligned}$$

Ainsi, on a optimisé  $n$  multiplications et  $n - 1$  additions en:

- $r - 1 = \frac{n}{m} - 1$  additions.
- $r$  instructions **XNOR**
- 1 décalage à gauche.
- 1 soustraction.

### 2.4.5 Gain Temps & Mémoire

#### Gain Mémoire

Les implémentations classiques des RN utilisent des paramètres en flottantes à précision simple. c'est à dire chaque paramètre prends 32 bits.

Avec le codage considéré, on peut avoir un gain de mémoire  $\alpha_{\text{mémoire}} = 32$

#### Gain Temps

Le gain de l'opération de produit scalaire est:

$$\begin{aligned}\alpha_{\langle \cdot, \cdot \rangle} &= \frac{n + n - 1}{\frac{n}{m} - 1 + \frac{n}{m} + 1 + 1} \\ &= \frac{2n - 1}{2\frac{n}{m} + 1} \\ &= \frac{2 - \frac{1}{n}}{2\frac{1}{m} + \frac{1}{n}} \\ &= \frac{2m - \frac{m}{n}}{2 + \frac{m}{n}} \\ &= \frac{m - \frac{m}{2n}}{1 + \frac{m}{2n}} \\ &= m - \frac{m(m + 1)}{2n} + o\left(\frac{m^3}{n^2}\right)\end{aligned}$$

On a:

- Pour un PC personnel, ou même téléphone:  $m = 64$ .
- Pour les réseaux de neurones standards,  $32 \leq n \leq 1024$

Ainsi une estimation de  $\alpha_{\text{temps}}$  est :

$$31.5 \leq \alpha_{\text{temps}} \leq \frac{2047}{32} \approx 62.03$$

# Chapter 3

## Modélisation

### 3.1 Introduction

Dans ce chapitre, on va étudier les approches faites dans la conception et implémentation des réseaux de neurones binaires, dériver les formules nécessaires dans les cas des couches dense et convolutionnelles, et dans le cas générique.

#### 3.1.1 Importance de ce chapitre

Dans les articles de BNNs considérés, les formules données ne sont généralement applicable qu'aux couches convolutionnelles à 2 dimensions.

Nous chercherons ici à trouver une approche générique avec laquelle on peut dériver pour chaque modèle considéré les équations des couches dense, convolutionnelles, récurrentes, etc...

Notre approche admet principalement 2 caractéristiques:

#### Rigueur

Pour établir les équations nécessaires, nous avons suivi les étapes suivantes:

1. Définir l'utilité potentielle du modèle
2. Traduire l'objectif en un problème d'optimisation mathématique
3. Définir les hypothèses nécessaires
4. Etablir les formules

#### Flexibilité

Grâce au niveau de rigueur utilisé, on peut facilement modifier et même améliorer les équations. Dans notre implémentation de chaque modèle, nous allons se baser sur les formules de ce chapitre en offrant la possibilité de faire quelques modifications.

#### 3.1.2 Convention Utilisée

Dans les démonstrations, nous allons suivre les conventions mathématiques de l'algèbre linéaire et de l'optimisation mathématique. Mais dans les implémentations, nous allons suivre les conventions de TensorFlow.

### 3.2 Histoire

Modèle	Année	Idée(s)
BinaryConnect[6]	2015	<ul style="list-style-type: none"> <li>• Binarisation des poids en <math>\pm 1</math></li> <li>• Optimisation des instructions MAC<sup>1</sup> en des sommes.</li> <li>• Estimateur direct du gradient pour résoudre le problème d'annulation de gradient.</li> </ul>
BinaryNet[5]	2015	<ul style="list-style-type: none"> <li>• Binarisation des paramètres et noeuds en <math>\pm 1</math></li> <li>• Optimisation des instructions MAC<sup>2</sup> en des XNOR et POPCOUNT.</li> </ul>
XNOR-NET[25]	2016	Introduction de deux facteurs $\alpha$ et $\beta$ à chaque produit scalaire pour minimiser l'erreur de l'opérateur bilinéaire $\star$
ABCNet[19]	2017	<ul style="list-style-type: none"> <li>• Utilisation de <math>n \in \mathbb{N}^*</math> binarisations pour les paramètres: <math>\widetilde{\mathbf{W}}^{(l),1}, \dots, \widetilde{\mathbf{W}}^{(l),n}</math></li> <li>• Utilisation de <math>m \in \mathbb{N}^*</math> binarisations pour les noeuds: <math>\tilde{a}^{(l),1}, \dots, \tilde{a}^{(l),m}</math></li> <li>• Décomposition de l'opération bilinéaire <math>\mathbf{W}^{(l)} \star a^{(l-1)}</math> en une combinaison linéaire des <math>\widetilde{\mathbf{W}}^{(l),i} \star \tilde{a}^{(l-1),j}</math> pour <math>i \in \{1, \dots, n\}</math> et <math>j \in \{1, \dots, m\}</math></li> </ul>
BiRealNet[21]	2019	<ul style="list-style-type: none"> <li>• Ajout d'une architecture résiduelle</li> <li>• Approximation régulière plus fine de la fonction sign</li> </ul>
MeliusNet	2020	<ul style="list-style-type: none"> <li>• Utilisation des blocs denses pour augmenter la capacité des caractéristiques</li> <li>• Utilisation des blocs d'améliorations pour augmenter la qualité des caractéristiques</li> </ul>

Table 3.1: Le tableau d'avancement des BNNs

### 3.3 Comparaison

#### 3.3.1 Par quantification

Modèle	Avantages	Inconvénients(s)
BinaryNet[5]	<ul style="list-style-type: none"> <li>• Le plus léger</li> <li>• Interférence rapide</li> </ul>	<ul style="list-style-type: none"> <li>• Induit des modèles très simple.</li> <li>• Admet des mauvaises performances de prédictions pour les tâches difficiles<sup>3</sup>.</li> <li>• Instable.</li> </ul>
XNOR-NET[25]	<ul style="list-style-type: none"> <li>• Le premier BNN performant.</li> <li>• Interférence rapide.</li> </ul>	<ul style="list-style-type: none"> <li>• Réintroduit des multiplication et divisions.</li> <li>• Performances modestes.</li> </ul>
ABCNet[19]	<ul style="list-style-type: none"> <li>• Très performant.</li> <li>• Il est stable par rapport aux autres.</li> <li>• Admet une justification théorique grâce à l'algèbre linéaire.</li> </ul>	<ul style="list-style-type: none"> <li>• Le temps d'interférence est en <math>\mathcal{O}(nm)</math>.</li> <li>• La mémoire est en <math>\mathcal{O}(n + m)</math></li> </ul>

Table 3.2: Approches de Quantification

#### 3.3.2 Par architecture

Modèle	Avantages	Inconvénients(s)
BiRealNet[21]	<ul style="list-style-type: none"> <li>• BNN performant</li> <li>• Conserve la rapidité d'interférence.</li> <li>• Modèle plus stable</li> </ul>	<ul style="list-style-type: none"> <li>• Conçu spécifiquement pour les architectures convolutionnelles</li> <li>• Instable.</li> </ul>
MeliusNet	<ul style="list-style-type: none"> <li>• Plus performant que BiRealNet</li> <li>• Conserve la rapidité d'interférence.</li> <li>• Modèle plus stable</li> </ul>	<ul style="list-style-type: none"> <li>• Plus lourds que BiRealNet</li> <li>• Conçu spécifiquement pour les architectures convolutionnelles</li> </ul>

Table 3.3: Approches de Quantification

## 3.4 BinaryNet

### 3.4.1 Conception

BinaryNet[5] est le premier réseau de neurones totalement binaires dans ces couches cachées, c'est à dire tout passage d'une couche cachée à une autre se fait par des accumulations de nombres binarisés ( $\pm 1$ ).

Autrement dit, c'est le premier réseau de neurones binaires conforme à notre définition.

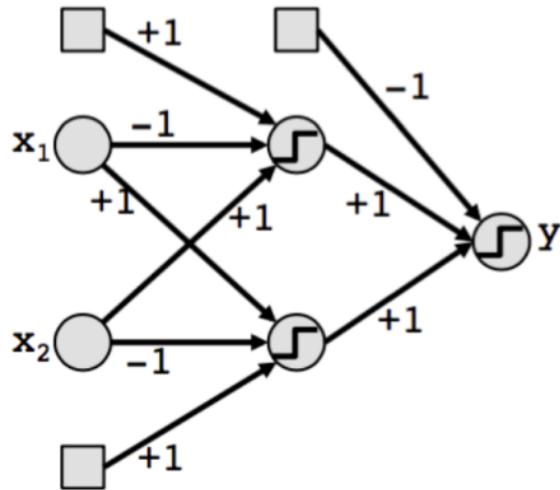


Figure 3.1: Exemple d'un BinaryConnect à une seule couche cachée

### 3.4.2 Topologie

BinaryNet est indépendant de l'architecture choisie: Elle peut être dense, convolutionnelle, récurrente, etc...

Mais en pratique, il n'est testé que dans le cas dense et convolutionnel.

### 3.4.3 Utilisation d'Estimateur Direct de gradient

#### Problématique

Puisque la fonction signe est constante par morceau, son dérivé est donc nulle presque partout (dans le sens de la théorie des mesures).

Ainsi, tout algorithme basé sur le gradient doit mettre en considération

#### Principe

L'entraînement d'un BNN est similaire à un RN classique:

1. Une propagation en avant pour calculer la valeur de la fonction objective
2. Une propagation en arrière pour calculer le gradient des paramètres
3. La mise à jour des paramètres

4. Refaire l'étape 1, jusqu'à satisfaire un critère bien déterminé (convergence, dépassement de la limite des époches, taux d'entraînement négligeable, etc...)

L'estimateur direct de gradient[3] sert à corriger le problème du gradient nul en faisant une estimation plus régulière de la fonction signe dans la propagation arrière (la fonction signe reste toujours utilisée dans la propagation avant)

L'estimation utilisée de la fonction signe est la fonction hardtanh définie par:

hardtanh :  $\mathbb{R} \rightarrow \mathbb{R}$

$$x \rightarrow \begin{cases} -1 & \text{si } x < -1 \\ x & \text{si } |x| \leq 1 \\ 1 & \text{si } x > 1 \end{cases}$$

En fait son dérivé est:

$$\begin{aligned} \forall x \in \mathbb{R}, \quad \frac{\partial \text{hardtanh}}{\partial x}(x) &= \mathbb{1}_{|x| \leq 1}(x) \\ &= \begin{cases} 1 & \text{si } |x| \leq 1 \\ 0 & \text{sinon} \end{cases} \end{aligned}$$

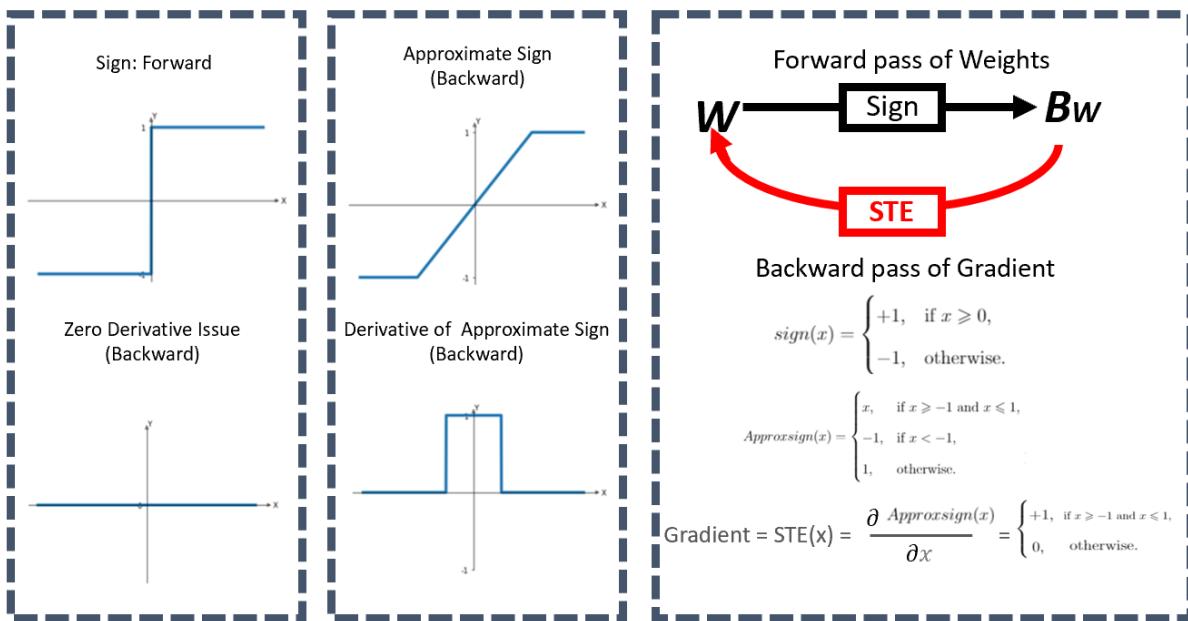


Figure 3.2: Estimateur direct du gradient

### Estimation du gradient

Soit  $f$  une fonction. Soit  $\mathbf{x}, \mathbf{y}$  deux tenseurs tels que  $f(\mathbf{x}) = y$ . On note par  $\mathcal{G}_{\mathbf{x}}$  l'estimation de gradient  $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$  (par la méthode STE) et on la définit par:

$$\mathcal{G}_{\mathbf{x}} = \begin{cases} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \odot \mathbb{1}_{|\mathbf{x}| \leq 1} & \text{si } f = \text{sign} \\ \mathcal{G}_y \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} & \text{sinon} \end{cases}$$

**Remarque 7** *Le calcul de  $\mathbb{1}_{|\mathbf{x}| \leq 1}$  se fait élément par élément.*

**Remarque 8** *L'opérateur  $\odot$  dénote la multiplication élément par élément*

La mise à jour des paramètres se fait en remplaçant chaque gradient par son estimation.

### Justification Théorique

#### 3.4.4 Optimisations possible

Etant un réseau totalement binarisé dans les couches cachées, BinaryConnect peut avoir plusieurs optimisations, y compris:

- Utilisation du Codage binaire
- Utilisation des instructions XNOR et POPCOUNT pour accélérer la propagation avant
- Utilisation de ADA-Max pour accélérer la mise à jour des paramètres
- Utilisation de "Shifted Batch Normalisation" pour accélérer le calcul de la normalisation par lots.

## 3.5 XNOR-NET

### 3.5.1 Conception

XNOR-NET[25] est le premier réseau de neurones binaires admettant des performances acceptables. Il est basé sur BinaryNet, et il met l'accent sur l'utilisation de XNOR et POPCOUNT [25].

De plus, Il introduit une nouvelle approche pour réduire l'erreur de quantification

### 3.5.2 Topologie

Comme BinaryConnect, XNOR-NET[25] est indépendant de l'architecture et la topologie utilisée.

En autre partie, il n'est pratiquement implémenté que dans les architectures denses et convolutionnelles.

### 3.5.3 Réduction de l'erreur de quantification

Pour  $\mathbf{W}^{(l)}$  et  $a^{(l-1)}$ , on veut trouver deux facteurs  $\alpha, \beta \in \mathbb{R}_+$ , et deux binarisations  $\tilde{\mathbf{W}}^{(l)}$  et  $\tilde{a}^{(l-1)}$  respectivement de  $\mathbf{W}^{(l)}$  et  $a^{(l-1)}$  tels que:

$$\mathbf{W}^{(l)} \approx \alpha \tilde{\mathbf{W}}^{(l)} \quad (3.1)$$

$$\tilde{a}^{(l-1)} \approx \beta \tilde{a}^{(l-1)} \quad (3.2)$$

$$\mathbf{W}^{(l)} \star a^{(l-1)} \approx \alpha \beta \tilde{\mathbf{W}}^{(l)} \star \tilde{a}^{(l-1)} \quad (3.3)$$

Dans l'objectif décrit, les binarisations ne sont pas nécessairement la fonction sign, mais on montrera que c'est toujours un bon choix.

### 3.5.4 Quantification optimale d'un vecteur

Dans cette section, nous allons trouvé les paramètres  $\alpha$  et  $\beta$  optimaux des équation respectifs (3.1) et (3.2) sans tenir compte de l'équation (3.3).

C'est à dire, nous allons dériver les quantification optimales de  $\mathbf{W}^{(l)}$  et  $a^{(l-1)}$  sans tenir compte de la qualité de la quantification (3.3) de  $z^{(l)} = \mathbf{W}^{(l)} \star a^{(l-1)}$ .

En effet, nous allons raisonner d'une manière générale. Et puis les deux résultats vont être immédiatement déduits.

### Notations

On va dénoter par:

- $n \in \mathbb{N}^*$  la dimension de notre espace vectoriel.
- $w$  un vecteur de  $\mathbb{R}^n$ .
- $\tilde{w} \in \{-1, 1\}^n$  une binarisation de  $w$ .
- $\gamma \in \mathbb{R}_+$  un facteur d'échelle.

## Objectif

L'objectif est de trouver  $\tilde{w}$  et  $\gamma$  qui minimisent  $\|w - \gamma\tilde{w}\|_2^2$  pour un  $u$  donné:

$$(\gamma, w) = \underset{\gamma, \tilde{w}}{\operatorname{argmin}} \|w - \alpha\tilde{w}\|_2^2 \quad (3.4)$$

## Résolution du problème

On a:

$$\begin{aligned} \forall \gamma \in \mathbb{R}, \forall \tilde{w} \in \{-1, 1\}^n, \quad \|w - \gamma\tilde{w}\|_2^2 &= \langle w, w \rangle - 2\gamma\langle w, \tilde{w} \rangle + \gamma^2\langle \tilde{w}, \tilde{w} \rangle \\ &= \langle w, w \rangle - 2\gamma\langle w, \tilde{w} \rangle + n\gamma^2 \\ \text{car } \tilde{w} \in \{-1, 1\}^n, \quad \|\tilde{w}\|_2^2 &= \sum_{i=1}^n \tilde{w}_i^2 = n \end{aligned}$$

Maintenant, on a  $\langle w, \tilde{w} \rangle$  est maximisé pour  $\tilde{w} = \operatorname{sign} w$ , et par conséquent puisque  $\gamma \geq 0$   $\langle w, w \rangle - 2\gamma\langle w, \tilde{w} \rangle + n\gamma^2$  est minimisé pour  $\tilde{w} = \operatorname{sign} w \quad \forall \gamma \in \mathbb{R}_+$ .

Ainsi le problème est réduit à la minimisation de la fonction quadratique suivante:

$$H(\gamma) = \langle w, w \rangle - 2\gamma\langle w, \operatorname{sign} w \rangle + n\gamma^2 \quad (3.5)$$

En effet,  $H$  étant une fonction convexe sur  $\mathbb{R}$ , admet un unique minimum local en  $x^* = \frac{\langle w, \operatorname{sign} w \rangle}{n}$ .  
Or, on a:

$$\begin{aligned} x^* &= \frac{\langle w, \tilde{w} \rangle}{n} = \frac{1}{n} \sum_{i=1}^n w_i \operatorname{sign} w_i \\ &= \frac{1}{n} \sum_{i=1}^n |w_i| = \frac{\|w\|_1}{n} \geq 0 \end{aligned}$$

Ainsi,  $\gamma = x^*$ , et donc l'expression de  $\gamma$  est:

$$\gamma = \frac{\|w\|_1}{n} \quad (3.6)$$

## Quantification de $\mathbf{W}^{(l)}$

On a  $\mathbf{W}^{(l)} \in E$  où  $E \cong \mathbb{R}^{n_1}$  avec  $n_1 = \dim \mathbf{W}^{(l)}$ .

Ainsi, on a  $\|\mathbf{W}^{(l)} - \alpha\tilde{\mathbf{W}}^{(l)}\|_2^2$  est minimisé pour:

$$\begin{cases} \alpha &= \frac{\|\mathbf{W}^{(l)}\|}{n_1} \\ \tilde{\mathbf{W}}^{(l)} &= \operatorname{sign} \mathbf{W}^{(l)} \end{cases} \quad (3.7)$$

### Quantification de $a^{(l)}$

On a  $a^{(l)} \in F$  où  $F \cong \mathbb{R}^{n_2}$  avec  $n_2 = \dim a^{(l)}$ .

Ainsi, on a  $\|a^{(l)} - \beta \tilde{a}^{(l)}\|_2^2$  est minimisé pour:

$$\begin{cases} \beta &= \frac{\|a^{(l)}\|}{n_2} \\ \tilde{a}^{(l)} &= \text{sign } a^{(l)} \end{cases} \quad (3.8)$$

### 3.5.5 Quantification d'un produit scalaire de deux vecteurs

Dans cette partie, nous allons tenir compte de la quantification de  $z^{(l)}$ , c'est à dire nous allons minimiser l'erreur de l'équation (3.3).

La résolution de cette équation n'est pas aussi triviale que (3.1) et (3.2), et donc nous allons majorer l'erreur de quantification  $\|\langle u, v \rangle - \langle \alpha \tilde{u}, \beta \tilde{v} \rangle\|_2^2$  par une fonction plus facile à optimiser, et nous allons retrouver les expressions (3.7) et (3.8) de la section précédente.

#### Objectif

- Soit  $n \in \mathbb{N}$  la dimension de l'espace euclidien.
- Soit  $u, v \in \mathbb{R}^n$

Notre objectif est de trouver deux facteurs  $\alpha, \beta \in \mathbb{R}_+$  et deux vecteurs  $u, v \in \{-1, 1\}^n$  qui réduisent  $\|\langle u, v \rangle - \langle \alpha \tilde{u}, \beta \tilde{v} \rangle\|_2^2$

#### Une Borne supérieure de l'erreur

Une formule exacte pour le problème originale est un peu difficile. Pour cela, on pose  $w = u \odot v$ , et on cherche une borne supérieure de  $\|\langle u, v \rangle - \langle \alpha \tilde{u}, \beta \tilde{v} \rangle\|_2^2$  en fonction de  $\|w - \alpha \beta \tilde{u} \odot \tilde{v}\|_2^2$  [25]:

$$\begin{aligned} \|\langle u, v \rangle - \langle \alpha \tilde{u}, \beta \tilde{v} \rangle\|_2^2 &= (\langle u, v \rangle - \langle \alpha \tilde{u}, \beta \tilde{v} \rangle)^2 \\ &= \left( \sum_{i=1}^n u_i v_i - \alpha \beta \tilde{u}_i \tilde{v}_i \right)^2 \\ &= \left| \sum_{i=1}^n \sum_{j=1}^n (u_i v_i - \alpha \beta \tilde{u}_i \tilde{v}_i)(u_j v_j - \alpha \beta \tilde{u}_j \tilde{v}_j) \right| \\ &\leq \sum_{i=1}^n \sum_{j=1}^n |u_i v_i - \alpha \beta \tilde{u}_i \tilde{v}_i| |u_j v_j - \alpha \beta \tilde{u}_j \tilde{v}_j| \\ &\leq \sum_{i=1}^n \sum_{j=1}^n \|u \odot v - (\alpha \tilde{u}) \odot (\beta \tilde{v})\|_\infty^2 \\ &\leq n^2 \|u \odot v - (\alpha \tilde{u}) \odot (\beta \tilde{v})\|_\infty^2 \\ &\leq n^2 \|u \odot v - (\alpha \tilde{u}) \odot (\beta \tilde{v})\|_2^2 \\ &\leq n^2 \|w - \alpha \beta \tilde{u} \odot \tilde{v}\|_2^2 \end{aligned}$$

Avec ce résultat, on est sûr qu'en minimisant  $\|w - \alpha \beta \tilde{u} \odot \tilde{v}\|_2^2$ , on minimise la borne supérieure de l'erreur de quantification.

### Minimisation de $\|w - \alpha\beta\tilde{u} \odot \tilde{v}\|_2^2$

En effet, Puisque  $\tilde{u} \odot \tilde{v}$  génère tout les vecteurs binaires, et  $\alpha\beta$  génère tous les réels positifs, on peut trouver  $\underset{\alpha, \beta, \tilde{u}, \tilde{v}}{\operatorname{argmin}} \|w - \alpha\beta\tilde{u} \odot \tilde{v}\|_2^2$  à partir de  $\underset{\gamma, \tilde{w}}{\operatorname{argmin}} \|w - \gamma\tilde{w}\|_2^2$ .

Or, on a déjà trouvé ce dernier. Ainsi on a:

$$\begin{cases} \tilde{u} \odot \tilde{v} = \tilde{w} = \operatorname{sign} w = \operatorname{sign} u \odot \operatorname{sign} v \\ \alpha\beta = \gamma = \frac{\|w\|_1}{n} \end{cases} \quad (3.9)$$

### Valeur de $\alpha, \beta, \tilde{u}$ et $\tilde{v}$

Pour trouver chacune de  $\alpha, \beta$  on fait les hypothèses suivantes:

- $u_1, \dots, u_n$  et  $\mathbf{x}$  suivent une distribution  $\mathcal{U}$  à moment absolu de premier ordre fini.
- $v_1, \dots, v_n$  et  $\mathbf{y}$  suivent une distribution  $\mathcal{V}$  à moment absolu de premier ordre fini.
- $\forall i, j \in \{1, \dots, n\}$ ,  $u_i$  et  $v_j$  sont indépendents
- $\alpha$  et  $\tilde{u}$  ne dépendent que de  $u$
- $\beta$  et  $\tilde{v}$  ne dépendent que de  $v$

En exploitant les hypothèses proposées, on peut facilement trouver  $\tilde{u}$  et  $\tilde{v}$ :

$$\begin{cases} \tilde{u} = \operatorname{sign} u \\ \tilde{v} = \operatorname{sign} v \end{cases} \quad (3.10)$$

Pour  $\alpha$  et  $\beta$ , on donne la démonstration suivante:

$$\begin{aligned} \mathbb{E}[\gamma] &= \mathbb{E}\left[\frac{\|w\|_1}{n}\right] = \frac{1}{n}\mathbb{E}[\|w\|_1] \\ &= \frac{1}{n}\sum_{i=1}^n \mathbb{E}[|u_i v_j|] = \frac{1}{n}\sum_{i=1}^n \mathbb{E}[|u_i|]\mathbb{E}[|v_j|] \\ &= \frac{1}{n}\sum_{i=1}^n \mathbb{E}[|\mathbf{x}|]\mathbb{E}[|\mathbf{y}|] = \mathbb{E}[|\mathbf{x}|]\mathbb{E}[|\mathbf{y}|] \\ &= \mathbb{E}\left[\frac{\|u\|_1}{n}\right] \times \mathbb{E}\left[\frac{\|v\|_1}{n}\right] \\ &= \mathbb{E}[\alpha]\mathbb{E}[\beta] \end{aligned}$$

Ainsi avec les hypothèses proposées on trouve  $\mathbb{E}[\alpha] = \mathbb{E}[|\mathbf{u}|]$  et  $\mathbb{E}[\beta] = \mathbb{E}[|\mathbf{v}|]$ , et donc:

$$\begin{cases} \alpha \approx \frac{\|u\|_1}{n} \\ \beta \approx \frac{\|v\|_1}{n} \end{cases} \quad (3.11)$$

En conclusion, on trouve[25]:

$$\begin{cases} u \approx \frac{\|u\|_1}{n} \operatorname{sign} u \\ v \approx \frac{\|v\|_1}{n} \operatorname{sign} v \end{cases} \quad (3.12)$$

### 3.5.6 Quantification d'une couche dense par produits scalaires

Dans ce cas, on a:

$$\mathbf{W}^{(l)} a^{(l-1)} = \begin{pmatrix} \langle \mathbf{W}_1^{(l)}, a^{(l-1)} \rangle \\ \vdots \\ \langle \mathbf{W}_{n_l}^{(l)}, a^{(l-1)} \rangle \end{pmatrix}$$

On applique la réduction d'erreur de quantification élément par élément:

$$\begin{cases} a^{(l-1)} \approx \beta \operatorname{sign} a^{(l-1)} & \beta = \frac{\|a^{(l-1)}\|_1}{n_{l-1}} \\ \forall i \in \{1, \dots, n_l\}, \quad \mathbf{W}_i^{(l)} \approx \alpha_i \operatorname{sign} \mathbf{W}_i^{(l)} & \alpha_i = \frac{\|\mathbf{W}_i^{(l)}\|_1}{n_{l-1}} \end{cases} \quad (3.13)$$

Ainsi

$$\begin{aligned} \mathbf{W}^{(l)} a^{(l-1)} &\approx \begin{pmatrix} \beta \alpha_1 \langle \operatorname{sign} \mathbf{W}_1^{(l)}, \operatorname{sign} a^{(l-1)} \rangle \\ \vdots \\ \beta \alpha_{n_l} \langle \operatorname{sign} \mathbf{W}_{n_l}^{(l)}, \operatorname{sign} a^{(l-1)} \rangle \end{pmatrix} \\ &\approx \beta \left( \operatorname{sign} \mathbf{W}^{(l)} \operatorname{sign} a^{(l-1)} \right) \odot \boldsymbol{\alpha} \end{aligned} \quad (3.14)$$

Full-precision		Binary		Binary with scaling	
Input	Weight	Input	Weight	Input Scaling	Weight Scaling
0.1	0.5 -0.5 -0.1	1	1 -1 -1	0.4	0.33 0.45 0.4
-0.7	-0.1 0.5 0.5	-1	-1 1 1		
0.5	-0.4 -0.7 0.3	1	-1 -1 1		
0.3	0.3 -0.1 -0.7	1	1 -1 -1		
Result	0.01 -0.78 -0.42	Result	2 -4 -2	Result	0.26 -0.72 -0.32
		Error	1.99 3.22 1.58	Error	0.25 0.06 0.1
Normalized	1.01 -0.95 -0.06	Normalized	1.25 -1.0 -0.25	Normalized	1.2 -1.06 -0.14
		Error	0.24 0.05 0.19	Error	0.19 0.11 0.08

Figure 3.3: Utilisation de facteurs  $\alpha$  et  $\beta$  dans une couche dense

### 3.5.7 Quantification d'une couche convolutionnelle par produits scalaires

**Remarque 9** Nous allons raisonner sur les convolutions à 2 dimensions. Mais en effet, les résultats peuvent être trivialement généralisés.

**Remarque 10** Pour trouver des expressions de  $\alpha$  et  $\beta$ , nous allons raisonner sur une convolution sans strides ni dilation, ni groupements. On va proposer ensuite un truc qui permet de recouvrir les formules de  $\alpha$  et  $\beta$  dans ces cas.

Soit  $\mathbf{W}^{(l)}$  une couche convolutionnelle qui prends  $C_{\text{in}}$  canaux et donne  $C_{\text{out}}$  canaux et dont les noyaux sont de dimensions  $(w, h)$ .

La formule de  $z^{(l)}$  est:

$$\forall c \in \{1, \dots, C_{\text{out}}\}, \quad z_c^{(l)} = \sum_{c'=1}^{C_{\text{in}}} \mathbf{W}_{c,c'}^{(l)} * a_{c'}^{(l-1)} \quad (3.15)$$

**Remarque 11** On peut aussi écrire (3.15) sous la forme compacte:

$$z^{(l)} = \mathbf{W}^{(l)} * a^{(l-1)} \quad (3.16)$$

où  $\mathbf{W}^{(l)}$  est de dimension  $C_{\text{out}} \times C_{\text{in}} \times w \times h$ , et  $a^{(l-1)}$  est de dimension  $C_{\text{in}} \times w \times h$

Pour simplifier, nous allons raisonner sur le cas  $C_{\text{out}} = 1$ , puis nous allons généraliser:

**Cas d'un seul canal de sortie:**  $C_{\text{out}} = 1$

On commence par le cas  $C_{\text{out}} = 1$  : On a:

$$\alpha = \frac{\|\mathbf{W}^{(l)}\|_1}{w \cdot h \cdot C_{\text{in}}} \quad (3.17)$$

$$\forall i, j \quad \beta_{i,j} = \frac{\|a_{[i,w],[j,h]}^{(l-1)}\|_1}{w \cdot h \cdot C_{\text{in}}} \quad (3.18)$$

avec  $a_{[i,w],[j,h]}^{(l-1)}$  est le sous-tenseur de dimensions  $C_{\text{in}} \times w \times h$  qui commence à la position  $i$  dans son deuxième axe, et  $j$  dans son troisième axe.

En faite,  $(\beta_{i,j})_{i,j}$  constitue un tenseur  $\beta^4$  de dimension  $w \times h$ , et la formule de  $z^{(l)}$  sera:

$$z^{(l)} = \left( \text{sign } \mathbf{W}^{(l)} * \text{sign } a^{(l-1)} \right) \odot \beta \alpha \quad (3.19)$$

**Cas général**

Dans le cas où  $C_{\text{out}} > 1$ , on décompose cette couche convolutionnelle en plusieurs convolutions. et on dérive la formule de chacune:

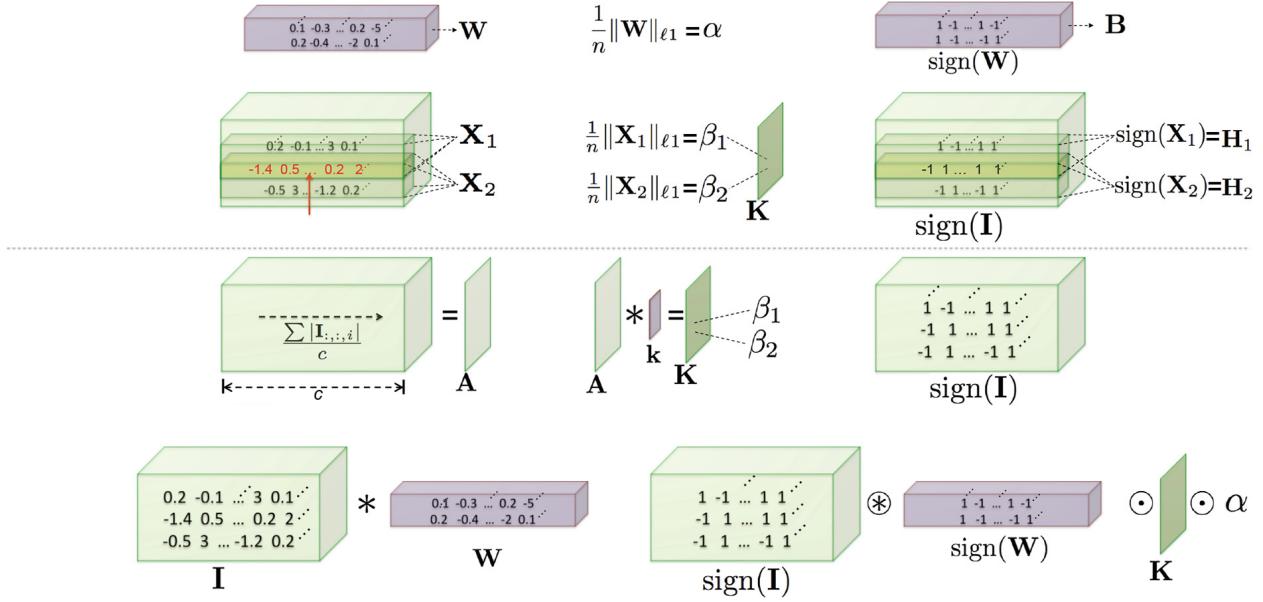
$$\forall c \in \{1, \dots, C_{\text{out}}\}, \quad z_c^{(l)} = \left( \text{sign } \mathbf{W}_c^{(l)} * \text{sign } a^{(l-1)} \right) \odot \beta \alpha_c \quad (3.20)$$

Finalement, on peut l'écrire dans la forme compacte:

$$z^{(l)} = \left( \text{sign } \mathbf{W}^{(l)} * \text{sign } a^{(l-1)} \right) \odot (\boldsymbol{\alpha} \otimes \boldsymbol{\beta}) \quad (3.21)$$

où  $\otimes$  est le produit extérieur entre deux tenseurs.

<sup>4</sup>Le temps de calcul de  $\boldsymbol{\beta}$  peut être optimisé grâce aux techniques de la programmation dynamique


 Figure 3.4: Utilisation de facteurs  $\alpha$  et  $\beta$  dans une couche convolutionnelle

### 3.5.8 Quantification d'une opération bilinéaire quelconque

#### Hypothèses

Les hypothèses suivantes ne posent aucune contrainte supplémentaires, en effet on les pose pour formaliser notre démonstration:

- On suppose que  $\mathbf{W}^{(l)}$  et  $a^{(l-1)}$  sont deux membres des espaces vectoriels réels respectifs  $E_l$  et  $F_{l-1}$ .
- On suppose que  $\star : E_l \times F_{l-1} \rightarrow H_l$  est bilinéaire avec  $H_l$  est un espace vectoriel réel de dimension  $\dim H_l = s$ .
- On suppose que  $\mathcal{B}_1$  et  $\mathcal{B}_2$  sont les bases respectifs de  $E_l$  et  $F_{l-1}$ .

#### Décomposition de $\star$ en formes bilinéaires

On a  $\star$  peut être décomposé élément par élément en  $s$  forme bilinéaire  $\star_1, \dots, \star_s : E_l \times F_{l-1} \rightarrow \mathbb{R}$ . Soit  $M_i = \text{mat}(\star_i, \mathcal{B}_1, \mathcal{B}_2)$ . On a donc:

$$\mathbf{W}^{(l)} \star a^{(l-1)} = \begin{pmatrix} \mathbf{W}^{(l)} \star_1 a^{(l-1)} \\ \vdots \\ \mathbf{W}^{(l)} \star_s a^{(l-1)} \end{pmatrix} = \begin{pmatrix} \mathbf{W}^{(l)T} M_1 a^{(l-1)} \\ \vdots \\ \mathbf{W}^{(l)T} M_s a^{(l-1)} \end{pmatrix}$$

## Décomposition en valeurs singulières

Dans cette section, nous allons quantifier chaque forme bilinéaire indépendamment de l'autre. Soit  $i \in \{1, \dots, s\}$ . On a d'après la décomposition en valeurs singulières:

$$\exists V_i \in \mathcal{O}(\mathbb{R}, m), \exists U_i \in \mathcal{O}(\mathbb{R}, n), \exists \Sigma_i \in \mathcal{M}(\mathbb{R}, n, m) \text{ diagonal} / \quad \begin{cases} M_i = U_i^T \Sigma_i V_i \\ \Sigma_{i,p,p} = \sigma_{i,p} \geq 0 \quad \forall p \in \{1, \dots, \min(n, m)\} \\ \Sigma_{i,p,p} = \sigma_{i,p} = 0 \quad \forall p > \min(n, m) \end{cases}$$

On peut encore factoriser  $\Sigma_i$  en deux matrices diagonales  $A_i \in \mathcal{M}(\mathbb{R}, \min(n, m), n)$  et  $B_i \in \mathcal{M}(\mathbb{R}, \min(n, m), m)$  tels que:

$$\begin{cases} A_{i,p,p} = B_{i,p,p} = \sqrt{\sigma_{i,p}} \quad \forall p \in \{1, \dots, \min(n, m)\} \\ \Sigma_i = A_i^T B_i \end{cases} \quad (3.22)$$

On a donc:

$$\begin{aligned} \mathbf{W}^{(l)} \star_i a^{(l-1)} &= \mathbf{W}^{(l)T} M_i a^{(l-1)} \\ &= \mathbf{W}^{(l)T} U_i^T A_i^T B_i V_i a^{(l-1)} \\ &= (A_i U_i \mathbf{W}^{(l)})^T (B_i V_i a^{(l-1)}) \\ &= \langle A_i U_i \mathbf{W}^{(l)}, B_i V_i a^{(l-1)} \rangle \end{aligned} \quad (3.23)$$

Pour quantifier ce produit scalaire, on doit d'abord généraliser l'équation (3.4) en:

$$(\gamma, \tilde{w}) = \underset{\gamma, \tilde{w}}{\operatorname{argmin}} \|AUw - \gamma AU\tilde{w}\|_2^2 \quad (3.24)$$

Avec

- $A \in \mathcal{M}(\mathbb{R}, m, n)$  une matrice diagonale dont les coefficient diagonaux  $A_{p,p} = \sqrt{\sigma_p}$  sont positifs
- $U \in \mathcal{O}(\mathbb{R}, n)$  une matrice orthogonale.

### Quantification optimale de $AUw$

Avant de résoudre ce problème, nous allons utiliser la propriété suivante:

$$U^T A^T A U = A^T A = D^2 \quad (3.25)$$

Avec  $D \in \mathcal{M}(\mathbb{R}, m)$  la matrice diagonale tels que  $D_{p,p} = \sqrt{\sigma_p}$

En effet, en exploitant (3.25) on a:

$$\begin{aligned} \|AUw - \gamma AU\tilde{w}\|_2^2 &= \langle AUw, AUw \rangle - 2\gamma \langle AUw, AU\tilde{w} \rangle + \gamma^2 \langle AU\tilde{w}, AU\tilde{w} \rangle \\ &= \langle AUw, AUw \rangle - 2\gamma \langle AUw, AU\tilde{w} \rangle + \gamma^2 \langle AU\tilde{w}, AU\tilde{w} \rangle \\ &= \langle w, U^T A^T A U w \rangle - 2\gamma \langle w, U^T A^T A U \tilde{w} \rangle + \gamma^2 \langle \tilde{w}, U^T A^T A U \tilde{w} \rangle \\ &= \langle w, D^2 w \rangle - 2\gamma \langle w, D^2 \tilde{w} \rangle + \gamma^2 \langle \tilde{w}, D^2 \tilde{w} \rangle \\ &= \langle Dw, Dw \rangle - 2\gamma \langle Dw, D\tilde{w} \rangle + \gamma^2 \langle D\tilde{w}, D\tilde{w} \rangle \end{aligned}$$

Or on a:

$$\langle D\tilde{w}, D\tilde{w} \rangle = \sum_{i=1}^n \sigma_i \tilde{w}_i^2 = \sum_{i=1}^n \sigma_i$$

Et donc on a:

$$\|\Sigma V w - \gamma \Sigma V \tilde{w}\|_2^2 = \langle Dw, Dw \rangle - 2\gamma \langle Dw, D\tilde{w} \rangle + \gamma^2 \sum_{i=1}^n \sigma_i$$

Maintenant, d'une façon analogue à la démonstration de 3.5.4, on peut montrer que  $\langle Dw, D\tilde{w} \rangle$  est maximisé pour  $\tilde{w} = \text{sign } w$ , et donc puisque  $\gamma \geq 0$ , on a  $\|AUw - \gamma AU\tilde{w}\|_2^2$  est minimisé pour  $\tilde{w} = \text{sign } w$  indépendamment de  $\gamma$ .

En s'inspirant de 3.5.4, on trouve:

$$\begin{cases} \gamma &= \frac{\sum_{i=1}^n \sigma_i |w_i|}{\sum_{i=1}^n \sigma_i} \\ \tilde{w} &= \text{sign } w \end{cases} \quad (3.26)$$

### Quantification optimale de $A_i U_i \mathbf{W}^{(l)}$

D'après la résolution du problème (3.24), et en exploitant le résultat (3.26):

$$\forall i \in \{1, \dots, r\} \begin{cases} \alpha_i &= \frac{\sum_{j=1}^n \sigma_{i,j} |\mathbf{W}_j^{(l)}|}{\sum_{j=1}^n \sigma_{i,j}} \\ \tilde{\mathbf{W}}^{(l)} &= \text{sign } \mathbf{W}^{(l)} \end{cases} \quad (3.27)$$

### Quantification optimale de $B_i V_i a^{(l-1)}$

De même, on trouve que:

$$\forall i \in \{1, \dots, s\} \begin{cases} \beta_i &= \frac{\sum_{j=1}^n \sigma_{i,j} |a_j^{(l-1)}|}{\sum_{j=1}^n \sigma_{i,j}} \\ \tilde{a}^{(l-1)} &= \text{sign } a^{(l-1)} \end{cases} \quad (3.28)$$

### Quantification de l'opérateur bilinéaire

Le résultat final est peut être écrit sous cette forme compacte:

$$\mathbf{W}^{(l)} \star a^{(l-1)} \approx \left( \text{sign } \mathbf{W}^{(l)} \star \text{sign } a^{(l-1)} \right) \odot \boldsymbol{\alpha} \odot \boldsymbol{\beta} \quad (3.29)$$

## 3.6 ABCNet

### 3.6.1 Conception

ABCNet[19] est un réseau de neurones binaires basé sur XNOR-Net, son nom est une abréviation de "Accurate Binary Convolutional Network", et donc comme son nom suggeste, il est très performant (comparable à un CNN classique).

Il peut être considéré comme une amélioration directe de XnorNet, puisqu'il réduit l'erreur de quantification d'une variable en faisant une somme pondérée de binarisations prédefinie.

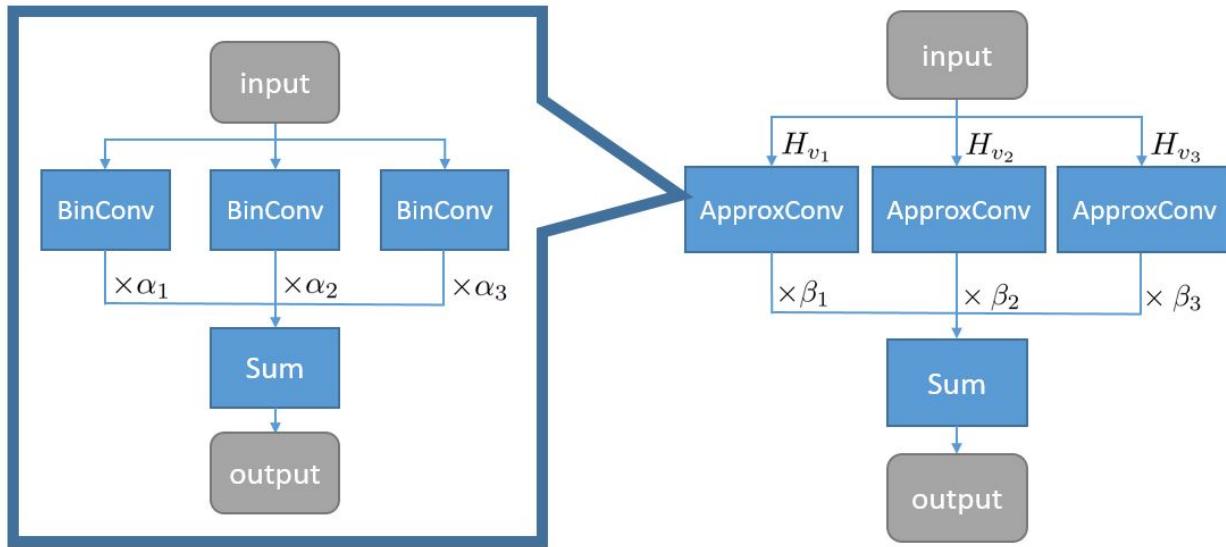


Figure 3.5: Exemple d'un ABCNet à une seule couche

### 3.6.2 Topologie

Le ABCNet[19] est conçu explicitement pour les réseaux de neurones binaires convolutionnelles.

Mais comme le XNOR-NET, il peut être généralisé à n'importe quel architecture (dense, récurrente, etc...)

### 3.6.3 Objectif

#### Notations

On dénote par

1.  $\Psi_1, \dots, \Psi_n$  des binarisations de  $\mathbf{W}^{(l)}$
2.  $\Phi_1, \dots, \Phi_m$  des binarisations de  $a^{(l-1)}$

#### Approximation par combinaisons linéaires

L'objectif d'un ABCNet est d'approximer  $\mathbf{W}^{(l)}$  en une combinaison linéaires de  $n$  binarisations, et  $a^{(l-1)}$  en une combinaison linéaires de  $m$  binarisations.

On veut trouver  $(\Psi_1, \dots, \Psi_n), (\Phi_1, \dots, \Phi_m), (\alpha_1, \dots, \alpha_n) \in \mathbb{R}_+^n, (\beta_1, \dots, \beta_m) \in \mathbb{R}_+^m$  tel que:

$$\mathbf{W}^{(l)} \approx \sum_{i=1}^n \alpha_i \Psi_i(\mathbf{W}^{(l)}) \quad (3.30)$$

$$a^{(l-1)} \approx \sum_{i=1}^m \beta_i \Phi_i(a^{(l-1)}) \quad (3.31)$$

$$\mathbf{W}^{(l)} \star a^{(l-1)} \approx \sum_{i=1}^n \sum_{j=1}^m \alpha_i \beta_j \Psi_i(\mathbf{W}^{(l)}) \star \Phi_j(a^{(l-1)}) \quad (3.32)$$

### 3.6.4 Choix de binarisations de $\mathbf{W}^{(l)}$

Il y'a plusieurs méthodes pour choisir les binarisations de  $\mathbf{W}^{(l)}$

#### Base orthogonale de Hadamard

Cette technique crée des binarisations qui ne dépendent pas de  $\mathbf{W}^{(l)}$ . En interprétant  $\mathbf{W}^{(l)}$  comme un vecteur d'un espace euclidien  $E$  de dimension  $r = 2^s \geq n$ . On cherche la matrice de Hadamard  $H_s$ , puis on choisit  $n$  colonnes (ou lignes)  $\Psi_1, \dots, \Psi_n$ .

Dans ce cas, la famille  $(\Psi_1, \dots, \Psi_n)$  induit un sous-espace  $F$  de  $E$  de dimension  $n$ .

Ainsi, le problème est réduit à une projection orthogonale de  $E$  vers  $F$

En notant  $K = \text{mat}(\Psi_1, \dots, \Psi_n)$ , la valeur optimale de  $\alpha$  qui minimise l'erreur quadratique de quantification est:

$$\alpha = K^T \mathbf{W}^{(l)} \quad (3.33)$$

#### Famille de fonctions sign décalées

Cette technique crée des binarisations de la forme:

$$\Psi_i(\mathbf{W}^{(l)}) = \text{sign}(\mathbf{W}^{(l)} + \mu_i) \quad (3.34)$$

Avec  $\mu_1, \dots, \mu_n \in \mathbb{R}$  un paramètre de décalage. Il peut être entraînable ou non.

#### Approximation par distribution

Cette technique se base sur l'observation que la distribution des poids (non-binarisés) est dense[19], et elle est proche d'une distribution normale. L'expression de  $\Psi_i$  est:

$$\Psi_i(\mathbf{W}^{(l)}) = \text{sign}\left(\mathbf{W}^{(l)} - \mathbb{E}[\mathbf{W}^{(l)}] + \mu_i \sqrt{\mathbb{V}[\mathbf{W}^{(l)}]}\right) \quad (3.35)$$

Où

- $\mathbb{E}[\mathbf{W}^{(l)}]$  est un estimateur de l'espérance de la distribution des paramètres: c'est la valeur moyenne de tous ces paramètres.
- $\mathbb{V}[\mathbf{W}^{(l)}]$  est une estimation de leur variance.
- $\mu_i$  est fixe ou entraînable.

Quelque soit le cas, on peut initialiser  $\mu_i$ [19] par:

$$\mu_i = -1 + 2 \cdot \frac{i-1}{n-1} \quad (3.36)$$

### 3.6.5 Choix de binarisations de $a^{(l)}$

Puisque  $a^{(l)}$  dépend toujours de l'entrée  $x$ . Pour cela  $a^{(l)}$  va généralement varier d'une inférence à une autre.

Pour cela, on doit créer des binarisations qui dépendent de  $a^{(l)}$ . Par exemple, on peut choisir:

#### Famille de fonctions sign décalées

Cette technique crée des binarisations de la forme:

$$\Psi_j(a^{(l)}) = \text{sign}(a^{(l)} + \kappa_j) \quad (3.37)$$

Avec  $\kappa_1, \dots, \kappa_m \in \mathbb{R}$  un paramètre de décalage. Il peut être entraînable ou non.

#### Approximation par distribution

$$\Phi_j(a^{(l)}) = \text{sign}\left(a^{(l)} - \mathbb{E}[a^{(l)}] + \kappa_j \sqrt{\mathbb{V}[a^{(l)}]}\right) \quad (3.38)$$

Où

- $\mathbb{E}[a^{(l)}]$  est une estimation de l'espérance de  $a^{(l)}$  dans le lot courant.
- $\mathbb{V}[a^{(l)}]$  est une estimation de la variance de  $a^{(l)}$  dans le lot courant
- $\kappa_j$  est fixe ou entraînable.

Dans la phase d'inférence, pour éviter le coût de calcul des paramètres statistiques, on peut remplacer:

- $\mathbb{E}[a^{(l)}]$  du lot courant par  $K_1 = \mathbb{E}_B[\mathbb{E}[a^{(l)}]]$ , qui est la valeur moyenne de tous les lots du jeu de données.
- $\mathbb{V}[a^{(l)}]$  du lot courant par  $K_2 = \frac{m}{m-1} \mathbb{E}_B[\mathbb{V}[a^{(l)}]]$ , qui est une estimation sans biais de la variance de  $a^{(l)}$  dans tous les lots du jeu de données, avec  $m$  la taille de lot.

### 3.6.6 Quantification par tenseur binaire

C'est la méthode où on fait les quantifications de  $\mathbf{W}^{(l)}$  et  $a^{(l-1)}$  en suivant les équations respectifs (3.30) et (3.31)

La quantification de  $z^{(l)}$  va respecter l'équation (3.32).

### 3.6.7 Quantification par produit scalaire

La quantification par produit scalaire est une variante dans laquelle les variables  $\alpha_1, \dots, \alpha_n$  de (3.30) et  $\beta_1, \dots, \beta_m$  de (3.31) sont calculées pour chaque produit scalaire.

C'est à dire, L'équation (3.32) est décomposée terme par terme en des produits scalaires, dans chacune on cherche les  $(\alpha_i)_{i \in \{1, \dots, j\}}$  et  $(\beta_j)_{j \in \{1, \dots, m\}}$ .

### Couche Dense

C'est une généralisation de l'équation (3.14) de XnorNet.

On a:

$$\begin{aligned}
 \mathbf{W}^{(l)} \cdot a^{(l-1)} &\approx \begin{pmatrix} \left\langle \sum_{i=1}^n \alpha_{i,1} \Psi_i \left( \mathbf{W}_1^{(l)} \right), \sum_{j=1}^m \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \\ \vdots \\ \left\langle \sum_{i=1}^n \alpha_{i,n_l} \Psi_i \left( \mathbf{W}_{n_l}^{(l)} \right), \sum_{j=1}^m \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \end{pmatrix} \\
 &\approx \begin{pmatrix} \sum_{i=1}^n \sum_{j=1}^m \left\langle \alpha_{i,1} \Psi_i \left( \mathbf{W}_1^{(l)} \right), \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \\ \vdots \\ \sum_{i=1}^n \sum_{j=1}^m \left\langle \alpha_{i,n_l} \Psi_i \left( \mathbf{W}_{n_l}^{(l)} \right), \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \end{pmatrix} \\
 &\approx \sum_{i=1}^n \sum_{j=1}^m \begin{pmatrix} \left\langle \alpha_{i,1} \Psi_i \left( \mathbf{W}_1^{(l)} \right), \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \\ \vdots \\ \left\langle \alpha_{i,n_l} \Psi_i \left( \mathbf{W}_{n_l}^{(l)} \right), \beta_j \Phi_j \left( a^{(l-1)} \right) \right\rangle \end{pmatrix} \\
 &\approx \sum_{i=1}^n \sum_{j=1}^m \beta_j \left( \Psi_i \left( \mathbf{W}^{(l)} \right) \cdot \Phi_j \left( \text{sign } a^{(l-1)} \right) \right) \odot \boldsymbol{\alpha}_i
 \end{aligned} \tag{3.39}$$

### Couche convolutionnelle

C'est une généralisation de l'équation (3.19) de XnorNet[25].

On a:

$$\mathbf{W}^{(l)} a^{(l-1)} \approx \sum_{i=1}^n \sum_{j=1}^m \left( \Psi_i \left( \mathbf{W}^{(l)} \right) * \Phi_j \left( a^{(l-1)} \right) \right) \odot (\boldsymbol{\alpha}_i \otimes \boldsymbol{\beta}_j) \tag{3.40}$$

### Opérateur bilinéaire quelconque

C'est une généralisation de 3.5.8, l'équation (3.29) va être généralisé en:

$$\mathbf{W}^{(l)} \star a^{(l-1)} \approx \sum_{i=1}^n \sum_{j=1}^m \left( \Psi_i \left( \mathbf{W}^{(l)} \right) \star \Phi_j \left( a^{(l-1)} \right) \right) \odot \boldsymbol{\alpha}_i \odot \boldsymbol{\beta}_j \tag{3.41}$$

## 3.7 BiRealNet

### 3.7.1 Conception

BiRealNet[21] est un réseau de neurones binarisés basés sur BinaryNet.

Son nom signifie le fait que s'il est binaire, il a aussi le comportement continu présent dans les réseaux de neurones classiques à valeurs réelles.

Il n'est pas basé sur la quantification comme les autres BNNs, mais sur l'architecture du réseau. En effet il réduit la perte d'information causée par la quantification par l'ajout des connexions résiduelles. Dans BiRealNet les connexions résiduelles sont sous la forme d'une somme simple.

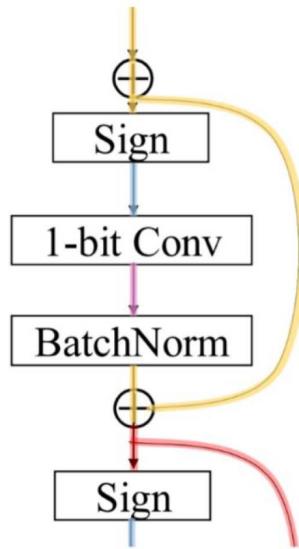


Figure 3.6: Exemple d'une liaison résiduelle d'un BiRealNet

### 3.7.2 Topologie

BiRealNet[21] est pratiquement indépendant de la topologie elle-même. Mais la présence des liaisons résiduelles y induit quelques contraintes.

La principale contrainte est que la couche résiduelle et la couche actuelle doivent avoir les mêmes dimensions.

### 3.7.3 Block BiRealNet

Pour bien augmenter l'apport d'informations avec BiRealNet, la connexion résiduelle est faite entre deux blocs connexes et qui vérifient la contrainte sur les dimensions.

De plus, le résidu doit avoir la possibilité de varier sur un ensemble réel. C'est à dire varier sur un grands ensemble de valeurs.

Un exemple concis est la figure 3.6 qui montre une connexion résiduelle pour chaque bloc de:

- Quantification par signe
- Convolution binarisée
- Normalisation par Lots

En effet, selon cette figure, la connexion va être faite entre le résultat du bloc actuel et le bloc précédent.

### 3.7.4 Liaison résiduale

#### Notation

Soit  $\mathcal{R}(l)$  l'ensemble de couches qui ont une liaison résiduale avec la  $l^{\text{ème}}$  couche.

#### Contraintes sur $\mathcal{R}(l)$

Dans BiRealNet, la seule contrainte présente est que le réseau de neurone doit rester acyclique:

$$\forall l, \forall l' \in \mathcal{R}(l), \quad l' < l \quad (3.42)$$

#### Equation

La liaison résiduale induit un changement de la formule de  $z^{(l)}$  trouvée dans le tableau 2.3 situé dans la section 2.2.1. La nouvelle expression de  $z^{(l)}$  est:

$$z^{(l)} = \mathbf{W}^{(l)} \tilde{\star} a^{(l-1)} + \sum_{l' \in \mathcal{R}(l)} z^{(l')} \quad (3.43)$$

Dans cette équation,  $\tilde{\star}$  signifie que l'opérateur  $\star$  est quantifié avec une quantification quelconque. Dans la plus part des cas,  $\mathcal{R}(l)$  est un singleton qui contient seulement la couche du dernier bloc, ce qui est le cas par exemple pour la figure 3.6

### 3.7.5 Choix de la quantification

Dans l'équation (3.43), nous n'avons explicité aucune quantification, et c'est l'une des avantages de BiRealNet: il est orthogonal au type de binarisation<sup>5</sup>.

Ainsi, l'opérateur quantifié  $\tilde{\star}$  peut être basé sur BinaryNet, XnorNet, ABCNet, etc...

---

<sup>5</sup>En effet, BiRealNet est un modèle basé sur l'architecture, et ces modèles sont généralement orthogonaux aux modèles basées sur la quantification, dans le sens où on peut créer un modèle en exploitant en même temps une quantification d'un et une architecture de l'autre.

# Chapter 4

## Implémentation

### 4.1 Introduction

Dans ce chapitre, on va implémenter les modèles qu'on a décri dans le chapitre précédent tout en suivant les paradigme que Keras et Larq respectent.

**Définition des types** Bien que Python est dynamiquement typé, nous allons inférer dans ce chapitre et dans notre bibliothèque un typage statique pour faciliter la compréhension.

Nous allons dénoter pour un ensemble de types  $T, T_1, \dots, T_n, Q, Q_1, \dots, Q_m$ :

- $\text{List}[T]$  pour n’importe quel itérable sur  $T$
- $\text{Tuple}[T_1, \dots, T_n]$  pour un tuple contenant des éléments de  $T_1, \dots, T_n$  dans l’ordre indiqué.
- $\mathcal{T}[T] = \text{Tensor}[T]$  pour un type compatible avec `tensorflow.Tensor` avec `dtype=T`
- $\mathcal{T}[T, n] = \text{Tensor}[T, n]$  pour un type compatible avec `tensorflow.Tensor` avec un rang égal à  $n$  et avec `dtype=T`
- $\mathcal{V}[T] = \text{Vector}[T]$  pour un type compatible avec `tensorflow.Tensor` avec un rang égal à 1 et avec `dtype=T`
- $\mathcal{M}[T] = \text{Matrix}[T]$  pour un type compatible avec `tensorflow.Tensor` avec un rang égal à 2 et avec `dtype=T`
- $\mathcal{R}[T] = \text{Random}[T]$  pour une fonction aléatoire sans arguments.
- $\mathcal{F}[T_1, \dots, T_n \rightarrow Q_1, \dots, Q_m] = \text{Function}[T_1, \dots, T_n \rightarrow Q_1, \dots, Q_m]$  pour une fonction dont les arguments sont de types respectifs  $T_1, \dots, T_n$  et dont les valeurs sonts de types respectifs  $Q_1, \dots, Q_m$
- $\text{End}[T] = \text{Function}[T_1, \dots, T_n \rightarrow T_1, \dots, T_n]$  pour une fonction dont l’argument et les valeurs sont de types respectifs  $T_1, \dots, T_n$ .
- $\mathcal{P}[T_1, \dots, T_n] = \text{Predicate}[T_1, \dots, T_n]$  pour un prédictat (fonction booléenne) acceptant  $n$  arguments de types respectifs  $T_1, \dots, T_n$

## 4.2 Paradigmes

### 4.2.1 Dans l'implémentation de BinaryFlow

Dans l'implémentations de la bibliothèque BinaryFlow, nous avons adopté principalement:

- La programmation orientée objet
- La programmation générique

#### Programmation orientée objet

#### Programmation générique

### 4.2.2 Dans l'utilisation de BinaryFlow

Etant une extension de Keras et Larq, nous recommanderions l'utilisateur à suivre les bonnes pratiques de ces bibliothèques.

En effet nous encouragerions à suivre une de ces interfaces:

- L'interface Séquentielle
- L'interface fonctionnelle
- L'interface orientée objet

#### Interface séquentielle

C'est l'interface la plus facile à exploiter, mais naturellement elle ne supporte pas les couches résiduelles.

L'utilisation de cette interface sert à créer une objet de type `tensorflow.keras.Sequential` contenant la séquence des couches dont le type est `List[tensorflow.keras.layers.Layer]`

#### Interface fonctionnelle

Cette interface met en valeur le fait qu'un modèle se traduit en des fonctions évaluer dans un certian ordre à un tenseur initial  $X$ .

Pour cela, cette interface introduit un tenseur symbolique  $X$  qui décrit l'entrée du réseau de neurones. Puis les fonctions sont appliqués au tenseur  $X$  pour établir le modèle et fixer les dimensions de:

- Ces paramètres
- L'entrée de chaque fonction
- La sortie de chaque fonction

Une fois le modèle est construit, on peut l'entraîner, et puis le déployer.

## Interface orientée objet

Cette interface met en valeur l'aspect orienté objet de TensorFlow (et aussi BinaryFlow), il se base sur le fait que qu'un utilisateur peut définir son modèle en héritant de la classe `tensorflow.keras.Model`. En utilisant cette interface, l'utilisateur peut surcharger les méthodes de `tensorflow.keras.Model`, ce qui lui permet de personnaliser:

- L'entraînement du modèle
- L'inférence du modèle
- L'évaluation du modèle
- La fonction coût du modèle
- ...

## Interface bas niveaux

Cette interface utilise directement les fonctionnalités de TensorFlow sans exploiter Keras. Elle est aussi supportée par BinaryFlow, mais on ne la recommande que pour les utilisateurs avancés.

### 4.2.3 Dans l'extension de BinaryFlow

Pour étendre notre bibliothèques, nous recommanderions de suivre les paradigmes utilisées dans l'implémentation qui sont décrite dans la section 4.2.1.

## 4.3 Conception

BinaryFlow est implémentés sous formes des modules suivants:

- Module des couches
- Module des blocs
- Module des quantificateurs
- Module des coûts

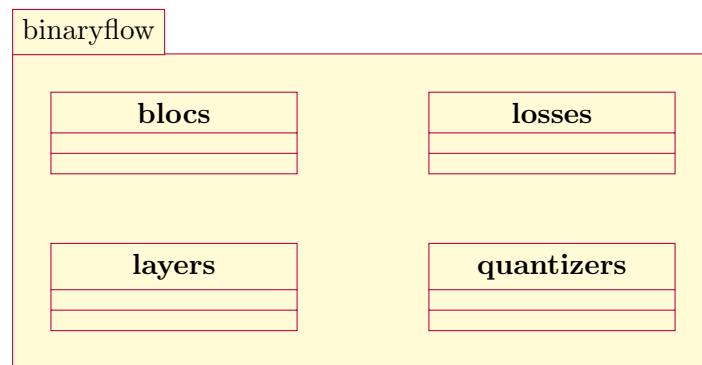


Figure 4.1: Structure de la bibliothèque

Dans les sections suivantes nous allons parler de chaque module.

## 4.4 Couches

Ce module est l'implémentation des couches des modèles suivant:

- BinaryNet
- XnorNet
- ABCNet

Dans ce module, chaque couche admet un nom de la forme `NomModèle.TypeCouche` avec:

- `NomModèle`  $\in \{\text{BinaryNet}, \text{XnorNet}, \text{ABCNet}\}$
- `TypeCouche`  $\in \{\text{Dense}, \text{Conv1D}, \text{Conv2D}, \text{Conv3D}\}$

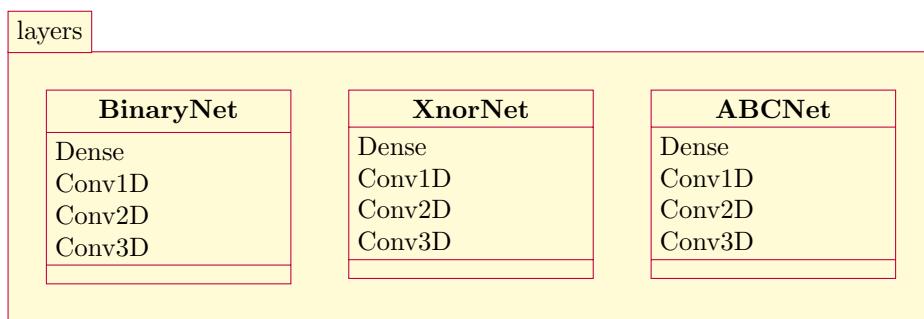


Figure 4.2: Le contenu de `layers`

#### 4.4.1 BinaryNet

Le module `BinaryNet` contient 4 classes qui sont: `Dense`, `Conv1D`, `Conv2D` et `Conv3D`. Nous avons définir ces classes comme des synonymes respectivement de `Larq.layers.QuantDense`, `Larq.layers.QuantConv1D`, et `Larq.layers.QuantConv3D`

##### API commun

Ces 4 couches admettent des attributs similaires, qui sont:

- `kernel_quantizer` qui est la quantification de poids.
- `input_quantizer` qui est la quantification des noeuds.
- `kernel_constraint` qui est la contrainte sur les poids.

Pour conformer à la formulation originale de `BinaryNet` [5] et `BinaryConnect` [6] ces valeurs doivent être initialisés à

Attribut	Valeur
<code>kernel_quantizer</code>	<ul style="list-style-type: none"><li>• <code>Larq.quantizers.SteSign</code></li><li>• <code>BinaryFlow.quantizers.StochasticSteSign</code></li></ul>
<code>input_quantizer</code>	<ul style="list-style-type: none"><li>• <code>Larq.quantizers.SteSign</code></li><li>• <code>BinaryFlow.quantizers.StochasticSteSign</code></li></ul>
<code>kernel_constraint</code>	<ul style="list-style-type: none"><li>• <code>weight_clip</code></li></ul>

Table 4.1: Terminologie de la division en Lots

##### Dense

Les attributs définissant cette couches sont:

- `units` qui est la dimension de sortie
- `kernel` qui est la matrice de paramètres

##### ConvND

Les couches convolutionnelles admettent des attributs similaires.

Pour une couche convolutionnelle à  $N \in \{1, 2, 3\}$  dimensions, on a:

- `filters` est le nombre de canaux de sorties.
- `kernel_size` est la taille du noyau de convolution

- **padding** est le padding utilisée dans la convolution.
- **strides** est la taille de pas entre deux convolutions successives

## BinaryNet

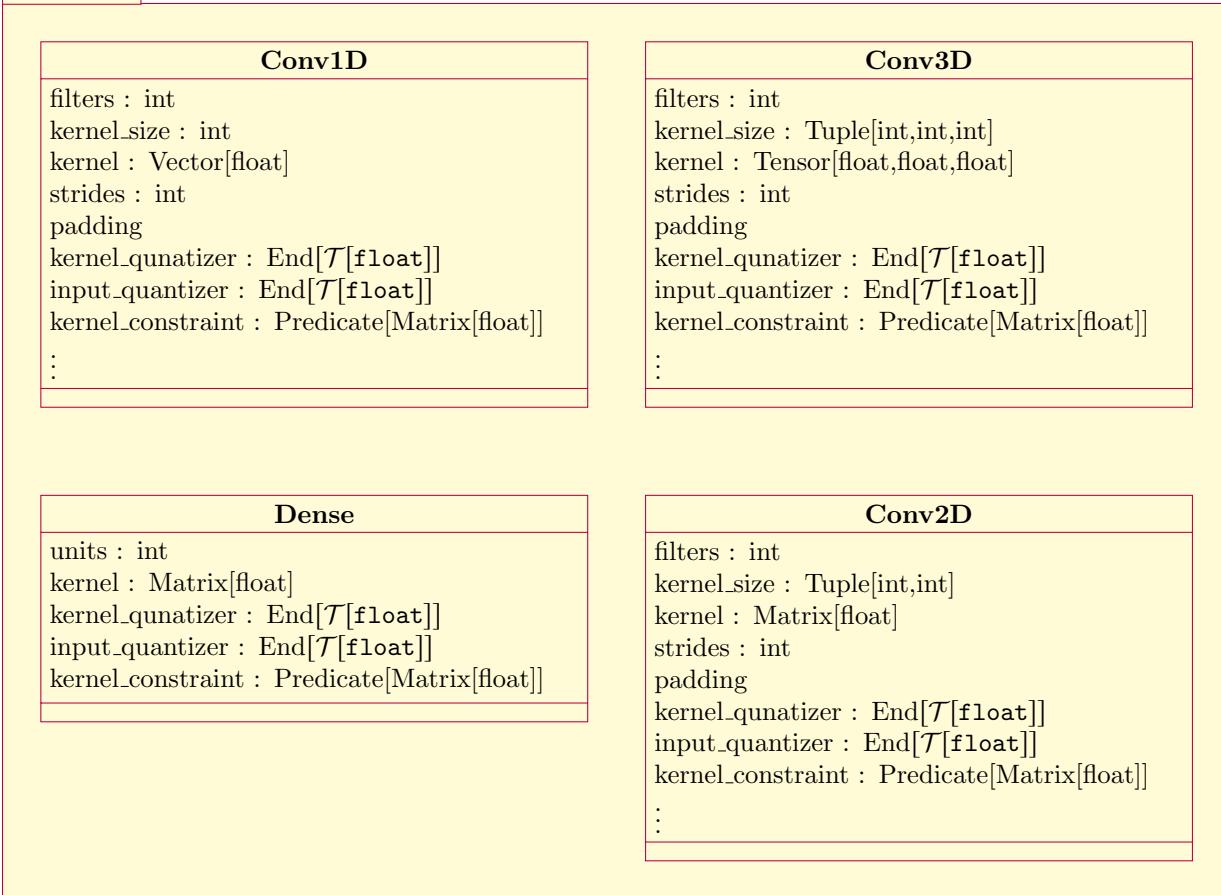


Figure 4.3: BinaryNet

#### 4.4.2 XnorNet

Le module `XnorNet` contient 4 classes qui sont: `Dense`, `Conv1D`, `Conv2D` et `Conv3D`.

Chacune de ces classes est une classe fille de son contrepart dans le module `BinaryNet` présenté dans 4.4.1.

#### Attributs Ajouté

Le seul attribut ajouté est `alpha_trainable` qui demande si  $\alpha$  doit être entraîné ou non. Par défaut il est égal à `False`.

#### Initialisaiton de $\alpha$

Indépendemment de `alpha_trainable`,  $\alpha$  est toujours initialisé à 3.14 dans le cas d'une couche dense, et à 3.14 dans le cas d'une couche convolutionnelle.

XnorNet											
<table border="1"> <thead> <tr> <th>Conv1D</th></tr> </thead> <tbody> <tr> <td>filters : int</td></tr> <tr> <td>kernel_size : int</td></tr> <tr> <td>kernel : Vector[float]</td></tr> <tr> <td>strides : int</td></tr> <tr> <td>padding</td></tr> <tr> <td>kernel_quanatizer : Random[float]</td></tr> <tr> <td>input_quantizer : Random[float]</td></tr> <tr> <td>kernel_constraint : Predicate[Matrix[float]]</td></tr> <tr> <td>alpha_trainable : Boolean</td></tr> <tr> <td>:</td></tr> </tbody> </table>	Conv1D	filters : int	kernel_size : int	kernel : Vector[float]	strides : int	padding	kernel_quanatizer : Random[float]	input_quantizer : Random[float]	kernel_constraint : Predicate[Matrix[float]]	alpha_trainable : Boolean	:
Conv1D											
filters : int											
kernel_size : int											
kernel : Vector[float]											
strides : int											
padding											
kernel_quanatizer : Random[float]											
input_quantizer : Random[float]											
kernel_constraint : Predicate[Matrix[float]]											
alpha_trainable : Boolean											
:											
<table border="1"> <thead> <tr> <th>Conv3D</th></tr> </thead> <tbody> <tr> <td>filters : int</td></tr> <tr> <td>kernel_size : Tuple[int,int,int]</td></tr> <tr> <td>kernel : Tensor[float,float,float]</td></tr> <tr> <td>strides : int</td></tr> <tr> <td>padding</td></tr> <tr> <td>kernel_quanatizer : Random[float]</td></tr> <tr> <td>input_quantizer : Random[float]</td></tr> <tr> <td>kernel_constraint : Predicate[Matrix[float]]</td></tr> <tr> <td>alpha_trainable : Boolean</td></tr> <tr> <td>:</td></tr> </tbody> </table>	Conv3D	filters : int	kernel_size : Tuple[int,int,int]	kernel : Tensor[float,float,float]	strides : int	padding	kernel_quanatizer : Random[float]	input_quantizer : Random[float]	kernel_constraint : Predicate[Matrix[float]]	alpha_trainable : Boolean	:
Conv3D											
filters : int											
kernel_size : Tuple[int,int,int]											
kernel : Tensor[float,float,float]											
strides : int											
padding											
kernel_quanatizer : Random[float]											
input_quantizer : Random[float]											
kernel_constraint : Predicate[Matrix[float]]											
alpha_trainable : Boolean											
:											
<table border="1"> <thead> <tr> <th>Dense</th></tr> </thead> <tbody> <tr> <td>filters : int</td></tr> <tr> <td>kernel : Matrix[float]</td></tr> <tr> <td>kernel_quanatizer : Random[float]</td></tr> <tr> <td>input_quantizer : Random[float]</td></tr> <tr> <td>kernel_constraint : Predicate[Matrix[float]]</td></tr> <tr> <td>alpha_trainable : Boolean</td></tr> </tbody> </table>	Dense	filters : int	kernel : Matrix[float]	kernel_quanatizer : Random[float]	input_quantizer : Random[float]	kernel_constraint : Predicate[Matrix[float]]	alpha_trainable : Boolean				
Dense											
filters : int											
kernel : Matrix[float]											
kernel_quanatizer : Random[float]											
input_quantizer : Random[float]											
kernel_constraint : Predicate[Matrix[float]]											
alpha_trainable : Boolean											
<table border="1"> <thead> <tr> <th>Conv2D</th></tr> </thead> <tbody> <tr> <td>filters : int</td></tr> <tr> <td>kernel_size : Tuple[int,int]</td></tr> <tr> <td>kernel : Matrix[float]</td></tr> <tr> <td>strides : int</td></tr> <tr> <td>padding</td></tr> <tr> <td>kernel_quanatizer : Random[float]</td></tr> <tr> <td>input_quantizer : Random[float]</td></tr> <tr> <td>kernel_constraint : Predicate[Matrix[float]]</td></tr> <tr> <td>alpha_trainable : Boolean</td></tr> <tr> <td>:</td></tr> </tbody> </table>	Conv2D	filters : int	kernel_size : Tuple[int,int]	kernel : Matrix[float]	strides : int	padding	kernel_quanatizer : Random[float]	input_quantizer : Random[float]	kernel_constraint : Predicate[Matrix[float]]	alpha_trainable : Boolean	:
Conv2D											
filters : int											
kernel_size : Tuple[int,int]											
kernel : Matrix[float]											
strides : int											
padding											
kernel_quanatizer : Random[float]											
input_quantizer : Random[float]											
kernel_constraint : Predicate[Matrix[float]]											
alpha_trainable : Boolean											
:											

Figure 4.4: XnorNet

#### 4.4.3 ABCNet

Le module `ABCNet` contient 4 classes qui sont: `Dense`, `Conv1D`, `Conv2D` et `Conv3D`. Chacune de ces classes est une classe fille de `tensorflow.keras.Model`.

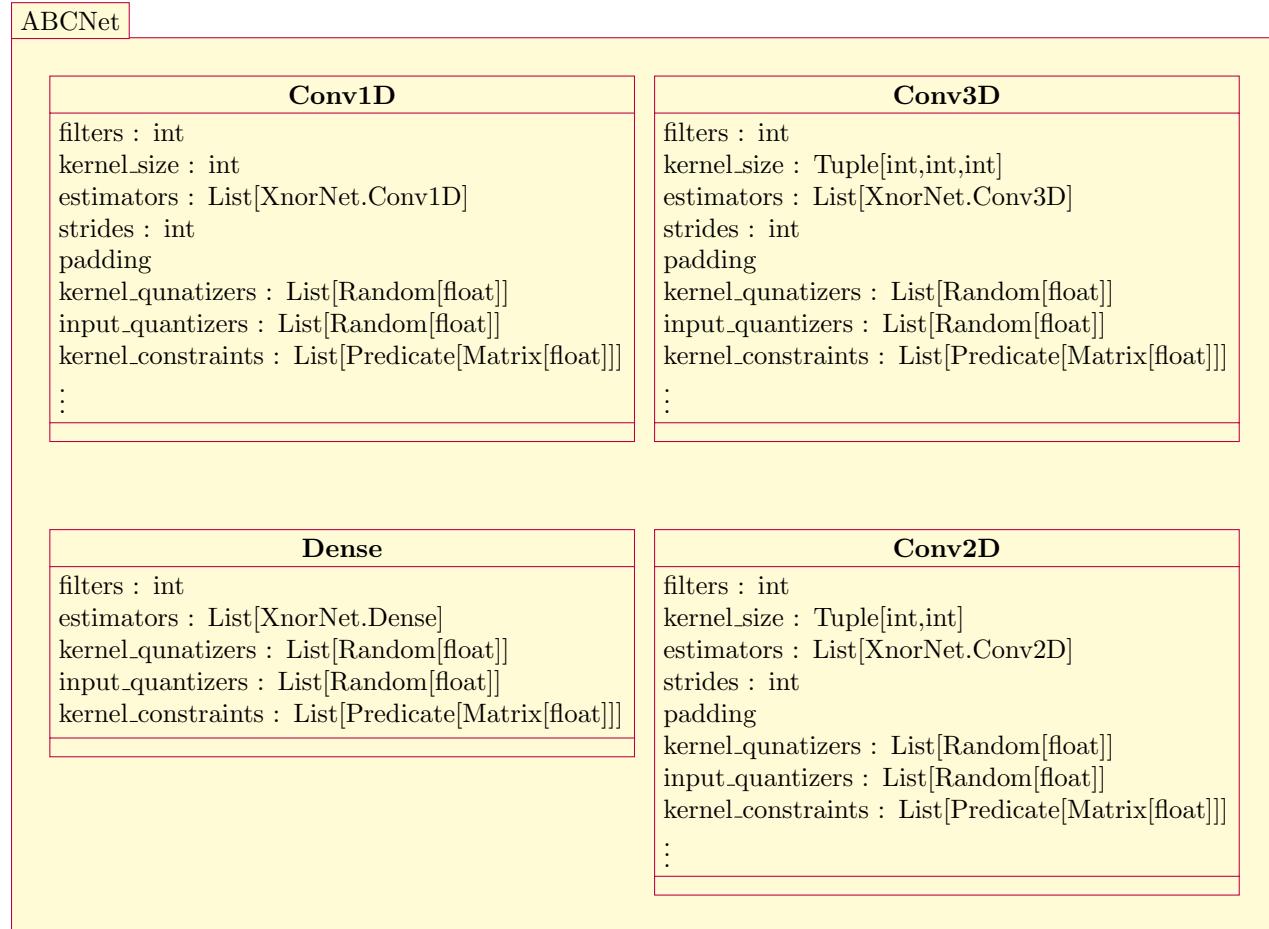


Figure 4.5: ABCNet

## 4.5 Block

Ce module est l'implémentation des "couches" des modèles suivant:

- BiRealNet
- MeliusNet

Vue leur nature résiduelle, une "couche" de l'un de ces modèles constitue effectivement un bloc.

**Remarque 12** *Ce module n'est nécessaire que dans l'utilisation de l'interface séquentielle.*

Dans ce module, chaque bloc admet un nom de la forme `NomBloc.TypeCouche` avec:

- `NomBloc`  $\in \{\text{Sequential}, \text{BiRealNet}, \text{MeliusNet}\}$
- `TypeCouche`  $\in \{\text{Dense}, \text{Conv1D}, \text{Conv2D}, \text{Conv3D}\}$

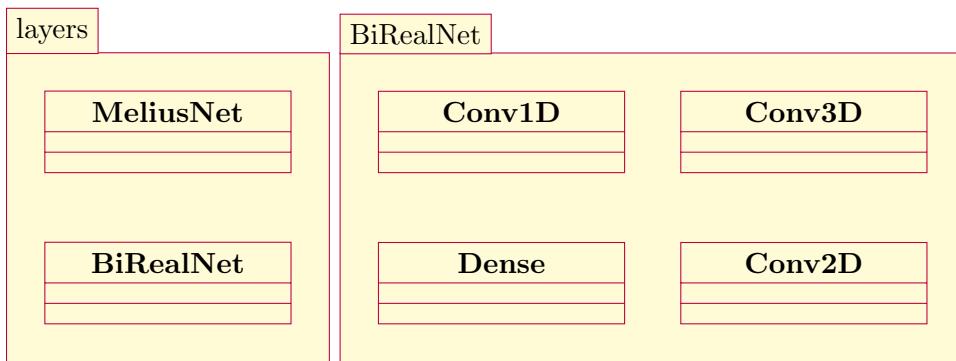


Figure 4.6: Le contenu de `blocks`, et de `BiRealNet`

## 4.6 Binarisations

Dans ce qui précède, nous n'avons parlé que de la fonction signe avec STE dans la propagation en arrière. Mais en effet BinaryFlow, supporte plusieurs binarisations, et ces binarisations peuvent être utilisées dans n'importe quel couche grâce à l'aspect modulaire.

Pour chaque binarisation, on peut aussi personnaliser l'estimation de son gradient, ce qui donne une flexibilité dans le choix de la binarisation.

Pour cela, nous allons étudier les binarisations que nous avons implémentées, et les approximations de leurs gradients

### 4.6.1 Propagation en Avant

**Définition 5** Une méta-binarisation est un opérateur  $\mathcal{K}$  qui prend une binarisation  $\Psi$  et donne une autre binarisation  $\mathcal{K}(\Psi)$

Nous avons les binarisations suivantes

#### Fonction signe

L'expression de cette fonction est:

$$x \rightarrow \text{sign}(x) \quad (4.1)$$

#### Fonction heavyside

L'expression de cette fonction est:

$$x \rightarrow H(x) \quad (4.2)$$

Cette binarisation doit être utilisé avec attention, car son utilisation rend plusieurs formules analytiques de XnorNet et ABCNet invalides.

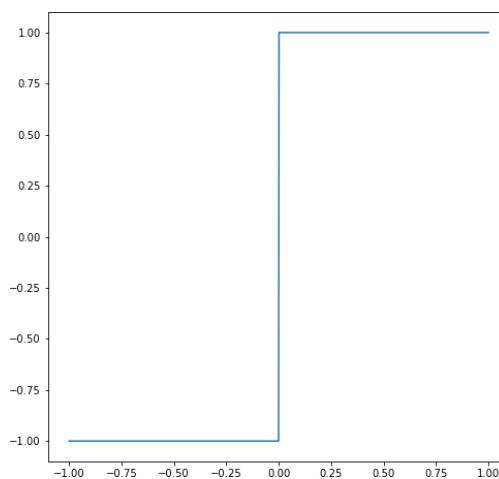


Figure 4.7: Traçage des fonction Signe et Heaviside

## Binarisation décalée

C'est une meta-binarisation qui prend une binarisation  $\Psi$  et donne  $\Psi$  décalé par un paramètre  $\mu$  :

$$x \rightarrow \Psi(x + \mu) \quad (4.3)$$

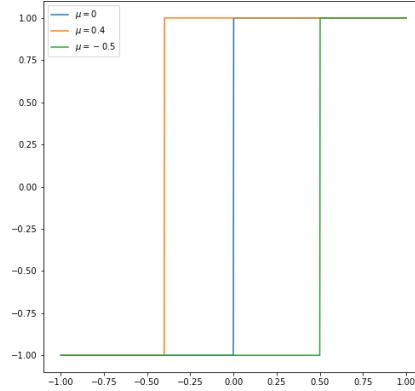


Figure 4.8: Traçage de la fonction signe décalée à gauche par  $\mu$

Le paramètre  $\mu$  peut être fixe ou entraîné.

## Binarisation stochastique

C'est aussi une meta-binarisation. Elle prend une binarisation  $\Psi$  et donne  $\Psi$  décalé par une variable aléatoire  $z$

$$x \rightarrow \Psi(x + z) \quad (4.4)$$

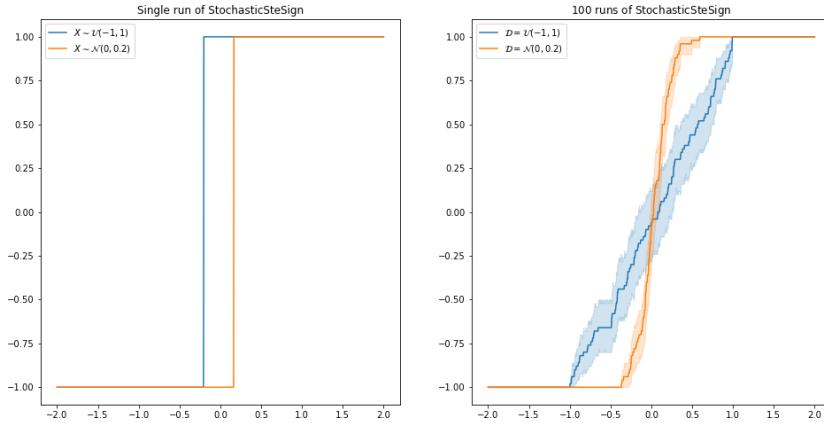


Figure 4.9: Traçage de la fonction signe stochastique

Dans cette figure,  $\mathcal{N}(0, 0.2)$  dénote la distribution normale centrée avec une deviation standard  $\sigma = 0.2$ , et  $\mathcal{U}(-1, 1)$  dénote la distribution uniforme dans l'intervalle  $[-1, 1]$

- La variable aléatoire  $z$  suit une distribution  $\mathcal{D}$  qui peut être fixe, ou même entraînée.
- Cette binarisation peut être utilisée comme une forme de régularisation pour le modèle.

#### 4.6.2 Propagation en arrière

On peut résumer les approximations du gradient par:

Estimation	Expression	Binarisation correspondante
STE	$1_{ x } \leq 1$	Signe, Heavyside.
ApproxSign	$1_{ x } \leq 1 \cdot (2 -  x )$	Sign
SwishSigne	$\frac{\beta(2 - \beta x \tanh(\frac{\beta x}{2}))}{1 + \cosh(\beta x)}$	Signe.

Table 4.2: Les estimations de gradient présentes

#### 4.6.3 Implémentation

Dans `binaryflow`, les binarisations sont implémentées dans le module `quantizers`

Toute binarisation implémentée hérite de la classe `larq.quantizers.Quantizer`, et elle admet aussi un attribut de classe `precision` qui est égal à 1.

Cet attribut permet à Larq de faire les optimisations adéquates dans la phase de déploiement.

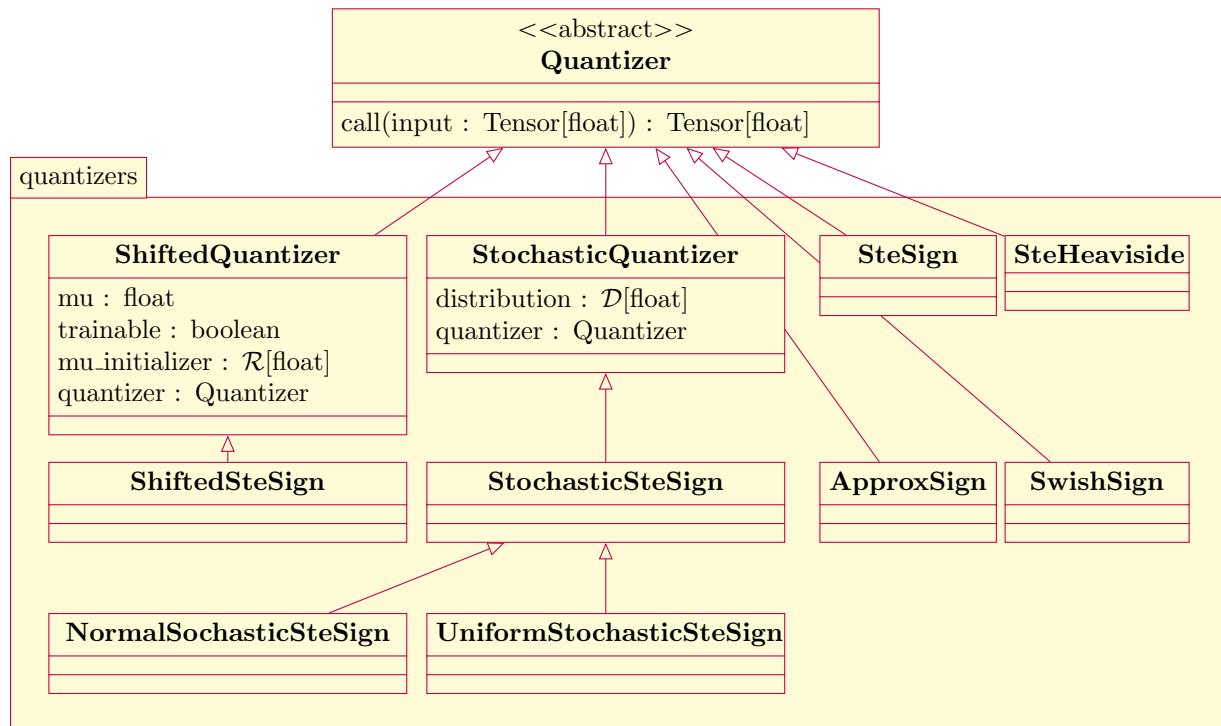


Figure 4.10: Diagramme de classe global du module `binaryflow.quantizers`

## 4.7 Régularisations

Les BNNs supportent les régularisations offertes par les réseaux de neurones classiques. De plus, ils supportent une autre forme de régularisation appelée régularisation de quantification, qui est un terme  $\mathcal{L}_{\text{quantification}}$  ajouté à la fonction objective pour réduire l'erreur de quantification:

$$\mathcal{L} = \mathcal{L}_{\text{model}} + \mathcal{L}_{\text{quantification}} \quad (4.5)$$

Le terme  $\mathcal{L}_{\text{quantification}}$  comporte une somme pondérée de:

### 4.7.1 Erreur de quantification des noeuds

C'est égal à

$$\mathcal{L}_{\text{weight}} = \sum_{a^{(l)} \text{quantified}} \|\tilde{a}^{(l)} - a\|_p^p \quad (4.6)$$

### 4.7.2 Erreur de quantification des poids

C'est égal à

$$\mathcal{L}_{\text{input}} = \sum_{\mathbf{W}^{(l)} \text{quantified}} \|\tilde{\mathbf{W}}^{(l)} - \mathbf{W}^{(l)}\|_p^p \quad (4.7)$$

### 4.7.3 Erreur de quantification de l'opération bilinéaire

C'est aussi l'erreur de quantification de  $z^{(l)}$ .

Cette erreur est l'erreur la plus importante car elle agit directement sur les performances du modèle. Elle est égale à:

$$\mathcal{L}_{\text{output}} = \sum_{\mathbf{z}^{(l)} \text{quantified}} \|\tilde{\mathbf{z}}^{(l)} - \mathbf{z}^{(l)}\|_p^p = \sum \|\tilde{\mathbf{W}}^{(l)} \star \tilde{a}^{(l-1)} - \mathbf{W}^{(l)} \star a^{(l-1)}\|_p^p \quad (4.8)$$

# Chapter 5

## Analyse

### 5.1 Introduction

Dans ce chapitre, on va résoudre 3 problèmes en utilisant les approches classiques d'apprentissages profonds, et aussi avec les réseaux de neurones binarisés. On va essentiellement résoudre:

1. Régression de  $f : x \rightarrow 2x^2 + 3x + 2$  sur  $[-4, 4]$
2. Classification des chiffres en exploitant le jeu de données MNIST
3. Classification des chiffres en exploitant le jeu de données Free Spoken Digits

## 5.2 Méthodologie adoptée

Dans les problèmes, qui se suivent, nous avons suivi la méthodologie CRSIP-DM[11].

### 5.2.1 CRISP-DM

CRISP-DM signifie Cross Industry Standard Process for Data Mining (CRISP-DM) est une méthode mise à l'épreuve sur le terrain permettant d'orienter les travaux d'exploration de données.

#### Cycle CRISP-DM

1. Connaissance du métier.
2. Connaissance des données.
3. Préparation des données.
4. Modélisation des données.
5. Évaluation.
6. Déploiement.

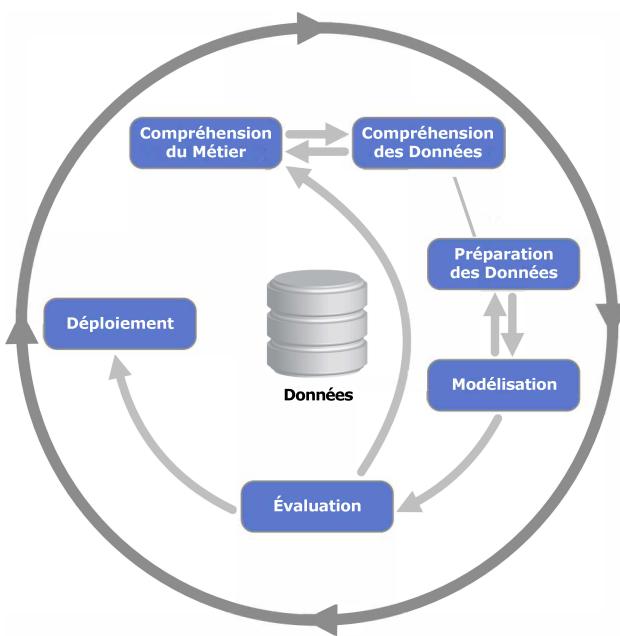


Figure 5.1: Méthodologie CRISP-DM

### 5.3 Régression de $f : x \rightarrow 2x^2 + 3x + 2$ sur $[-4, 4]$

#### 5.3.1 Importance

Ce problème est fait artificiellement étudier la robustesse les BNNs implémentées.

#### 5.3.2 Jeux de données

Dans cette problématique, le jeu de données est généré avec  $n = 20000$  exemplaires:

- $x_1, \dots, x_n \sim \mathcal{U}(-4, 4)$
- $\epsilon_1, \dots, \epsilon_n \sim \mathcal{N}(0, 0.1)$
- $y_1, \dots, y_n$  avec  $y_i = f(x_i) + \epsilon_i \quad \forall i \in \{1, \dots, n\}$ .

#### 5.3.3 Modèle

Notre modèle  $\mathcal{M}_\theta$  est un estimateur de  $f$  paramétrisé par  $\theta \in S$ . L'ensemble  $S$  va dépendre de type du modèle.

Dans notre cas, l'architecture du modèle est décrite par la figure ci-dessous.

Pour la binarisation, nous avons utilisé la fonction sign dans la propagation avant, et nous avons utilisé la méthode STE bornée sur  $[-1, 1]$  pour la propagation en arrière.

#### 5.3.4 Hypothèses

On a fait les hypothèses suivantes:

1. H1:  $x_1, \dots, x_n$  sont mutuellement indépendents
2. H2: Les bruits sont mutuellement indépendentes.
3. H3: Le model  $\mathcal{M}$  ne connaît sur la fonction  $f$  que  $f(x_i) \approx y_i$ , et il ne tient pas compte de l'erreur  $\epsilon_i$ .

#### 5.3.5 Problème Formel

On va chercher  $\theta^*$  tel que:

$$\theta^* = \underset{\theta \in S}{\operatorname{argmin}} \|\mathbf{y} - \mathcal{M}_\theta(\mathbf{x})\|_2^2 = \underset{\theta \in S}{\operatorname{argmin}} \sum_{i=1}^n (y_i - \mathcal{M}_\theta(x_i))^2 \quad (5.1)$$

#### 5.3.6 Entraînement

On a entraîné 3 types de modèles:

- BinaryNet
- XnorNet
- ABCNet

Pour chacun, on a utilisé:

- L'optimiseur Adam, avec `learning_rate` =  $10^{-2}$  et `decay` =  $10^{-4}$
- La fonction objective MSE comme décri dans l'équation (5.1)
- Taille de lot égale à 128
- Une limite de 30 époches. .

### 5.3.7 Performances de prédiction

On a tracé la courbe de chaque modèle en le comparant avec  $f$ :

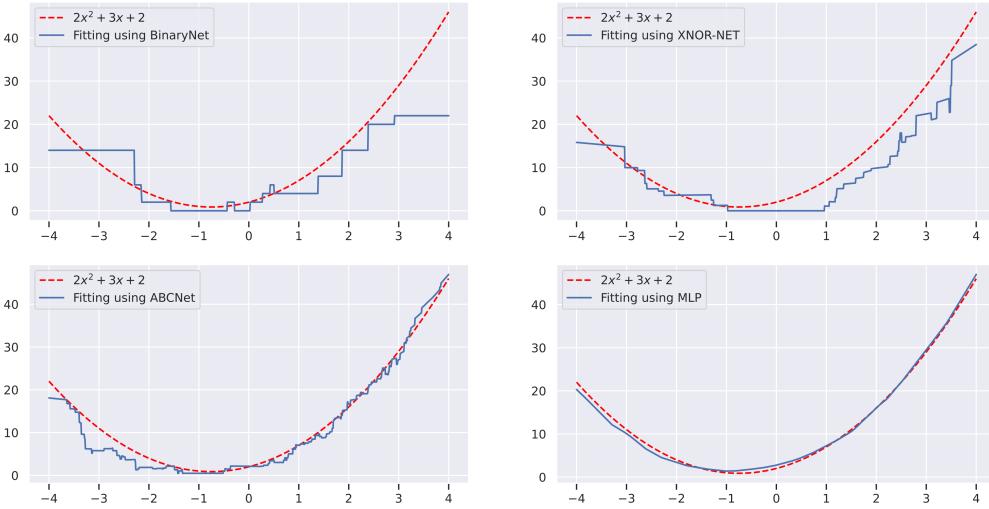


Figure 5.2: Courbe de chaque modèle

Pour mesurer la qualité de régression, nous avons utilisé 3 métriques:

#### Distance $\mathcal{L}^\infty$

C'est la distance  $\mathcal{L}^\infty$  dans l'espace des fonctions continues  $\mathcal{C}([-4, 4])$ :

$$\mathcal{L}_{\text{test}}^\infty(y, \mathcal{M}_\theta) = \sup_{x \in [-4, 4]} |\mathcal{M}_\theta(x) - f(x)| \quad (5.2)$$

#### Distance $\mathcal{L}^1$

C'est la distance  $\mathcal{L}^1$  dans l'espace  $\mathcal{C}([-4, 4])$ :

$$\mathcal{L}_{\text{test}}^1(y, \mathcal{M}_\theta) = \int_{-4}^4 |\mathcal{M}_\theta(x) - f(x)| dx \quad (5.3)$$

**Distance  $\mathcal{L}^2$** 

C'est la distance  $\mathcal{L}^2$  dans l'espace  $\mathcal{C}([-4, 4])$ :

$$\mathcal{L}_{\text{test}}^2(y, \mathcal{M}_\theta) = \left( \int_{-4}^4 (\mathcal{M}_\theta(x) - f(x))^2 dx \right)^{\frac{1}{2}} \quad (5.4)$$

Dans la pratique, nous allons estimer les 2 intégrales en utilisant la méthode Romb avec  $n = 2^{20} + 1$  points.

On a trouvé les résultats suivant:

Modèle	$\mathcal{L}^\infty$	$\mathcal{L}^1$	$\mathcal{L}^2$ Carré	$\mathcal{L}^2$
BinaryNet	24.000	4.322	42.524	6.521
XNOR-NET	13.852	3.531	19.671	4.435
ABCNet	7.395	1.421	4.024	2.01
MLP	1.692	0.616	0.562	0.750

Dans le tableau ci-dessus, MLP est le modèle classique sans utilisation de binarisations, et il sert comme référence.

## 5.4 Classification MNIST

### 5.4.1 Introduction

MNIST[18] est un jeu de données des images de chiffres écrits à la main. Il admet:

- 60000 exemplaires pour l'entraînement
- 10000 exemplaires pour le test.

Les images sont de tailles  $28 \times 28$  et en niveau de gris, dans lesquels les chiffres sont centrés.

Ce jeu de données sert comme un test de performance des modèles de Machine Learning.

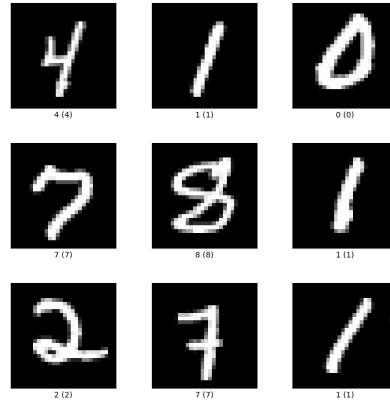


Figure 5.3: Des images de MNIST

### 5.4.2 Topologie

Nous avons utilisé l'architecture ci-dessous:

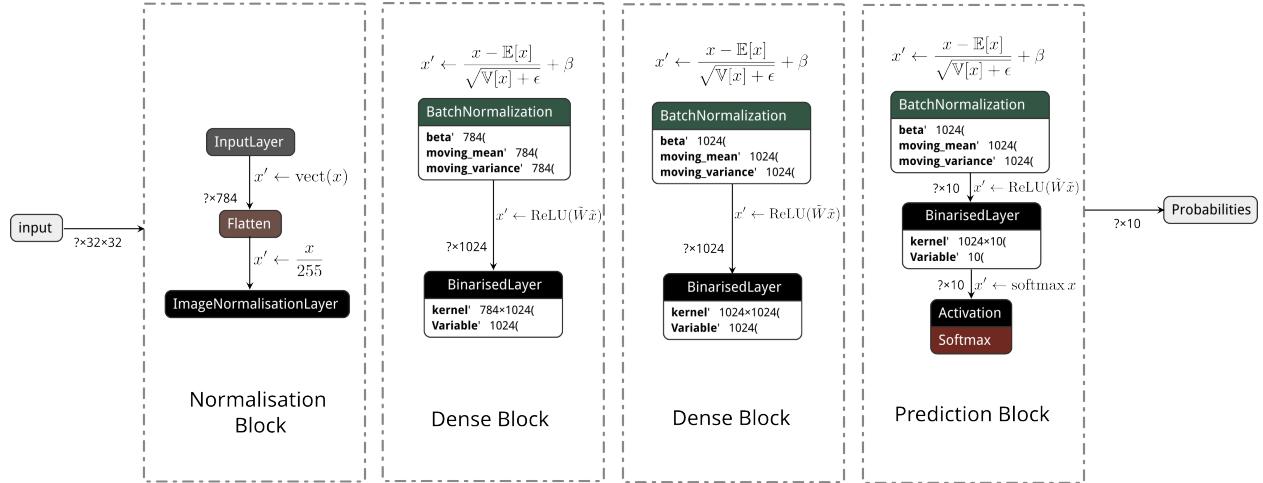


Figure 5.4: Architecture du modèle MNIST

BinarisedLayer désigne ici le type de couche dense binarisé qui est utilisé. Il s'agit de:

- QuantDense pour BinaryNet
- ScaledQuantDense pour XnorNet
- ABCDense pour ABCNet

Dans la figure, les dimensions de couche sont décrites explicitement avec leurs équations, et le point d'interrogation '?' indique que le modèle est indifférent à la taille de lot.

### 5.4.3 Modèle

On a utilisé 3 modèles pour l'architecture donnée:

- BinaryNet, avec:
  - Binarisation: sign avec STE
  - Activation: ReLU
- XnorNet, avec:
  - Binarisation: sign avec STE
  - Activation: ReLU
  - $\alpha$  et  $\beta$  sont calculés par produit scalaire.
  - Le facteur  $\alpha$  est entraînable, initialisé à (3.12)
  - Le facteur  $\beta$  est calculé, initialisé à (3.12)
- ABCNet avec:
  - Binarisation: sign décalée avec STE.
  - Activation: ReLU.
  - $n = 3$  binarisations des poids, suivant
  - $m = 1$  binarisations des noeuds, suivant
  - $\mu_i$  et  $\kappa_j$  sont entraînables
  - $\alpha_i$  et  $\beta_j$  sont calculés par produit scalaire.

### 5.4.4 Performances de prédictions

### 5.4.5 Performance mémoire

les performances mémoire sont mesurées avec la taille totale des paramètres, qui donnent une approximation de l'utilisation mémoire du modèle à son déploiement.

Nous avons fait la comparaison de 2 versions de chaque modèle:

- Les paramètres (non-binaires) sont en float à 32 bits.
- Les paramètres (non-binaires) sont en float à 8 bits.

### Comparaison Float 32

Les paramètres de type float sont naturellement encodés en 64 bits dans TensorFlow. En apportant des binarisations, on va avoir le tableau suivant

Modèle	Nombre de paramètres binarisés	Nombre de paramètres non-binarisés	Taille mémoire en KB $T$	Taux de compression $\tau = \frac{T_{\text{MLP}}}{T_{\text{Model}}}$
MLP	0	1'869'354	7302.16	1
BinaryNet	1'861'632	5'664	249.38	29.28
XnorNet	1'861'632	7'722	257.41	28.36
ABCNet	5'584'896	11'838	727.99	10.03

Table 5.1: Comparaison entre les modèles en utilisant Float32

### Comparaison Float 8

On peut aussi compresser les paramètres float en utilisant Float 8 (Minifloat) dans TensorFlow. Ainsi, on va avoir le tableau suivant:

Modèle	Nombre de paramètres binarisés	Nombre de paramètres non-binarisés	Taille mémoire en KB $T$	Taux de compression $\tau = \frac{T_{\text{MLP}}}{T_{\text{Model}}}$
MLP	0	1'869'354	7302.16	1
BinaryNet	1'861'632	5'664	249.38	29.28
XnorNet	1'861'632	7'722	257.41	28.36
ABCNet	5'584'896	11'838	727.99	10.03

Table 5.2: Comparaison entre les modèles en utilisant Float8

### Quantification de Float 32 à Float 8

Avec les deux tableaux précédents, on peut étudier l'apport de la réduction de précision de float à minifloat pour les 4 modèles considérés en terme de mémoire.

Modèle	Taille avec Float 32 $T_{32}$	Taille avec Float 8 $T_8$	Rapport de Taille $\tau = \frac{T_8}{T_{32}}$
MLP	7302.16	1825.54	0.25
BinaryNet	249.38	232.78	0.9334
XnorNet	257.41	234.79	0.9121
ABCNet	727.99	693.31	0.9523

Table 5.3: Comparaison entre les modèles en utilisant Float8

#### 5.4.6 Performance temps

les performance temps sonts mesurée avec le nombre d'instructions multiply & accumulate (MAC) nécessaires pour chaque modèle à son déploiement.

Modèle	Nombre de MACs binarisés $N_B$	Nombre de MACs non-binarisés $N_F$	Instructions équivalentes: $I = \frac{N_B}{64} + N_F$	Taux de MACs binarisés $\alpha = \frac{N_B}{N_B + N_F}$	Taux de gain: $\tau = \frac{I_{\text{MLP}}}{I_{\text{Model}}}$
MLP	0	1'861'632	1'861'632	0	1
BinaryNet	1'861'632	2058	29'088	1	64
XnorNet	1'861'632	7'722	31'064	0.9989	59.92
ABCNet	5'584'896	11'838	93'438	0.9989	19.92

## 5.5 Classification Free Spoken Digits

### 5.5.1 Importance

Dans la littérature, les BNNs utilisés principalement avec les jeux de données images. Ainsi, notre objectif est l'utilisation d'un BNN pour résoudre les problèmes de traitement des signaux à faible complexité.

Nous avons choisis Free Spoken Digits[13] comme une preuve de concept de l'applicabilité des BNNs dans ce genre de problèmes. De plus, nous allons utiliser l'expertise de dB Sense pour extraire intelligemment les informations utiles d'une parole afin de construire un classificateur performant et à faible complexité.

### 5.5.2 Introduction

Free Spoken Digits[13] (FSD) est un jeu de données audio qui consiste d'un ensemble de fichiers wav à une fréquence d'échantillonage de 8 kHz. Les fichiers sont coupés pour minimiser le silence.

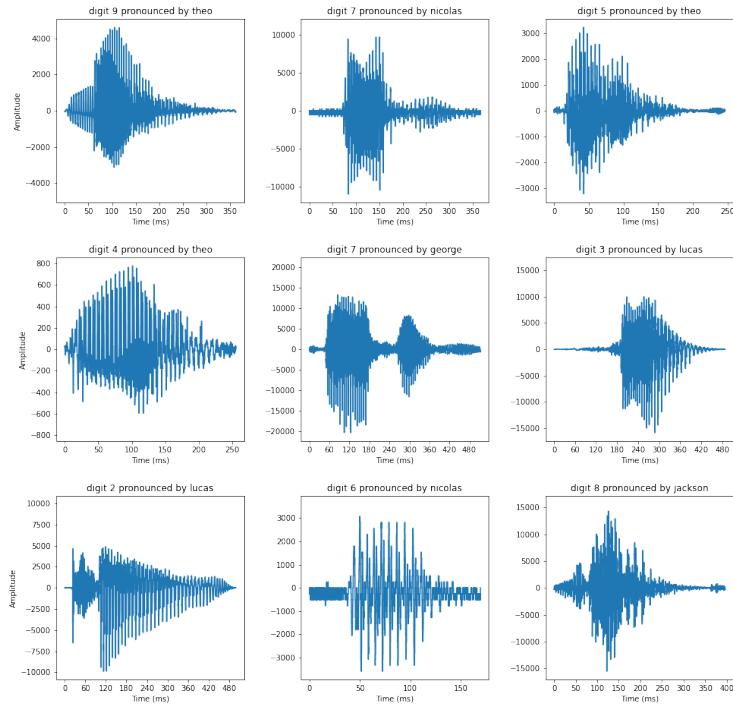


Figure 5.5: Des images de MNIST

### 5.5.3 Structure

FSD est enregistré à l'aide d'un microphone monophonique, et il admet 6 orateurs qui ont chacun 50 enregistrements par chiffre.

En total, chaque orateur admet 500 enregistrements, et donc ce jeu de données admet 3000 fichiers audio.

De plus, chaque enregistrement admet un nom significatif de la forme `digit_person_id.wav` avec:

- `digit` est le chiffre prononcé dans l'enregistrement.
- `person` est le nom de l'orateur.
- `id` est l'identificateur de l'enregistrement, sachant la personne et le chiffre, appartenant à l'ensemble  $\{0, \dots, 49\}$

### 5.5.4 Analyse

# Chapter 6

## Déploiement

### 6.1 Methodology

For the machine learning model creation we followed the Knowledge Discovery in Databases (*KDD*)[32] methodology which is an iterative multi-stage process for extracting useful information from databases. This process is a road map with multiple stages that emphasises on early decisions and choices we make showing how important planning can lead to a successful and well managed project. This methodology was invented by Oussama Fayyad back in the 1996.

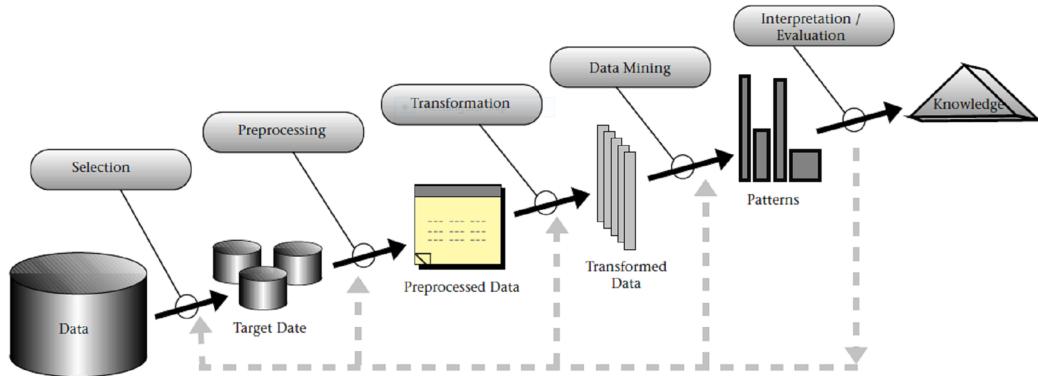


Figure 6.1: Knowledge Discovery in Databases Methodology

### 6.1.1 Selection

The initial graphs were gathered from the network repository . In addition to the different characteristics of each graph.

We then used these graphs to run our work product on different environments, gather the results of each test and save them into different files using the *Writer* module.

We also gathered the characteristics from each machine to be used in our model.

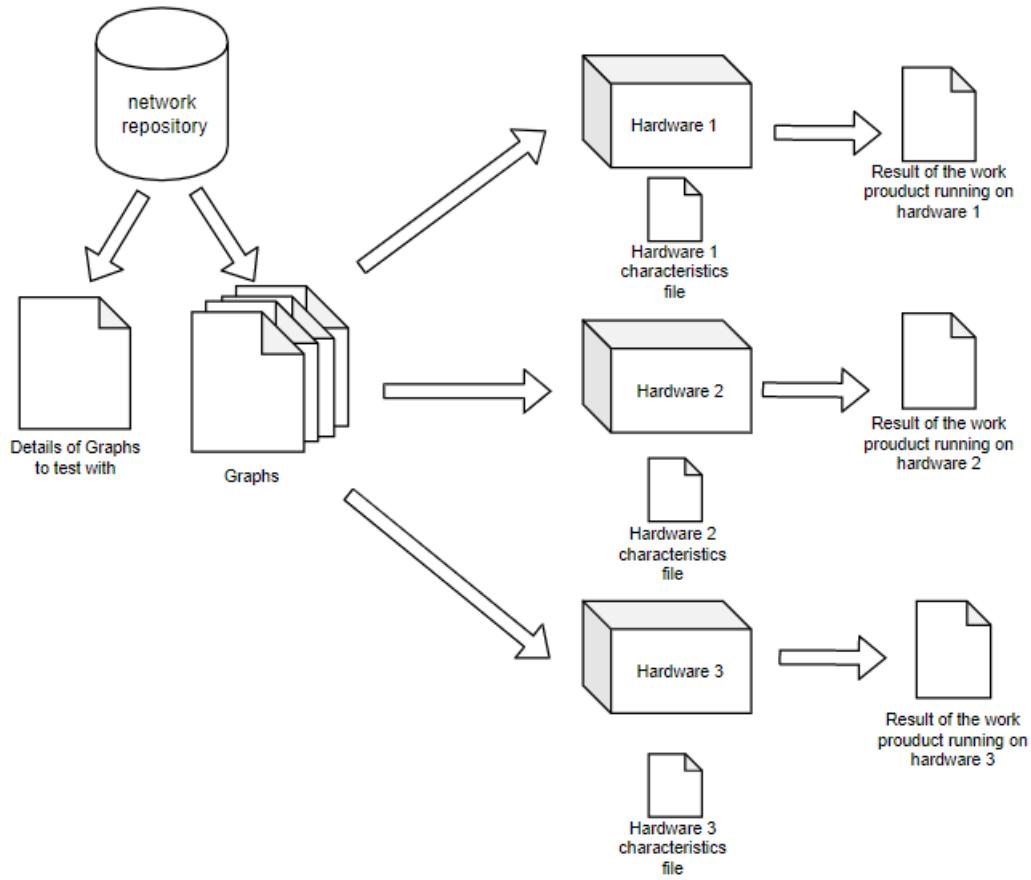


Figure 6.2: Selection stage

### 6.1.2 Preprocessing and feature selection

#### Data integration

After the selection phase, we joined our results into one data store from which we are going to extract knowledge.

We joined files containing details on graphs and files containing results of our work product by graph name.

We joined files containing information of the hardware characteristics and our result files by hardware name.

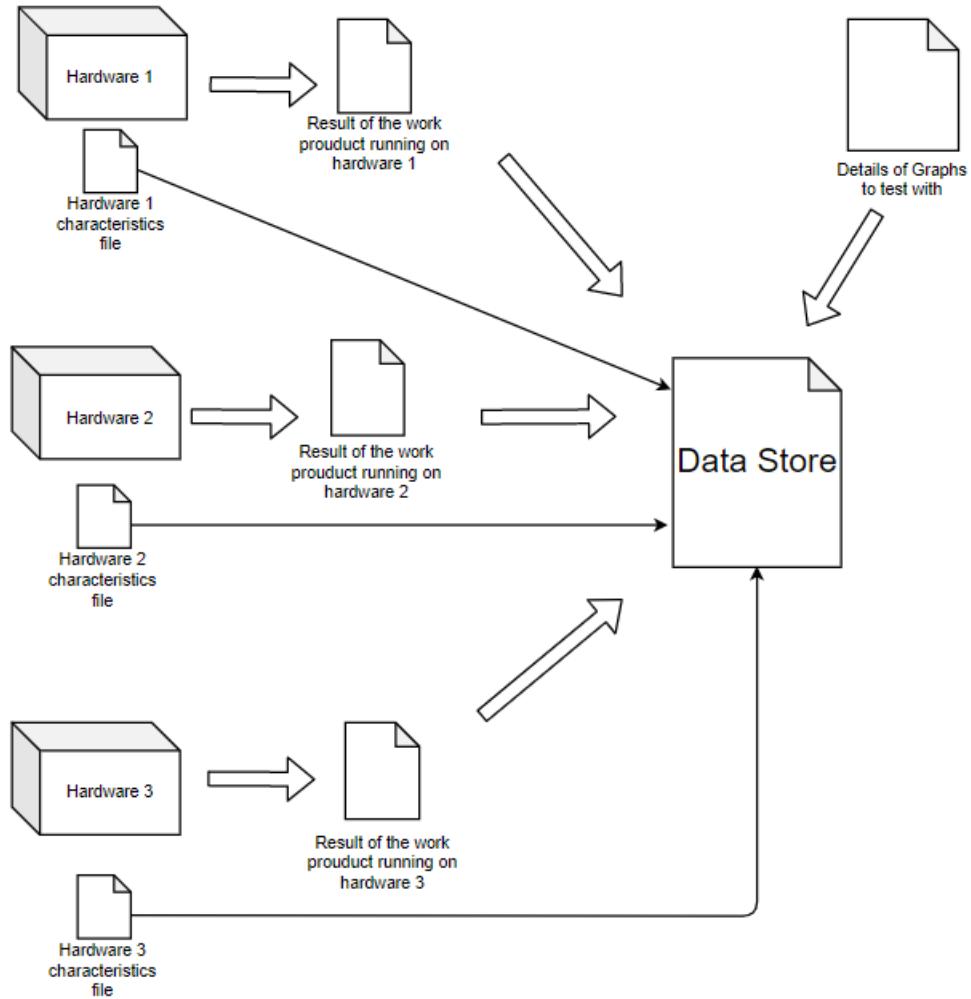


Figure 6.3: Data integration

### Naming conventions

We started by naming our different features using a certain convention ( all our features are written in *Pascal Case* with spaces between words. Each column not following this convention will be renamed.

### Dividing features based on complexity

After naming our features we started classifying them based on their complexity. We divided our features into three groups

- Available features : these are features that can be easily determined and are generally gathered from our initial data set, such as :
  - *Number Of Threads*: This feature is used implicitly by the Graph Strategy prediction model
  - *Number Of Edges*  $|\mathcal{E}|$ : This can be calculated by only counting the lines on the file, and can even be estimated in  $\mathcal{O}(1)$  by viewing the graph file size and reading some random lines.
  - *Number Of Nodes*  $|\mathcal{V}|$ : We set  $n \leftarrow 0$ , we then read each vertex  $u$  on the file, and set  $n \leftarrow \max(n, u + 1)$ . The number of nodes is then  $n$
  - *Density*: It is equal to  $\frac{|\mathcal{E}|}{|\mathcal{V}|(|\mathcal{V}|-1)}$
- Calculable features : these features can be calculated from the graph file without creating the graph, such as :
  - Contains all Available features.
  - *Max Degree*: This can be calculated by creating a vector of size  $|\mathcal{V}|$  and for each line  $u \sim v$  of the graph file we set  $T[u] \leftarrow T[u] + 1$ . the max degree is then  $\max_u T[u]$
  - *Min Degree*: Same as the max degree, but we return  $\min_u T[u]$
  - *Average Degree*: Same as the max degree, but we return  $\frac{\sum_u T[u]}{|\mathcal{V}|}$
- Hard features : These are features requires the creation of the graph, or are simply computationally not feasible, such as :
  - *Assortativity*
  - *Number Triangles*: The fastest possible algorithm[23] has complexity  $\mathcal{O}(|\mathcal{V}|^\omega)$ <sup>1</sup>. So it is not feasible to calculate the number of triangles given the size of considered graphs, we can only do some estimations of this features.
  - *Average Number Of Triangles, Max Number Of Triangles, Average Clustering Coefficient, Fraction Closed Triangles*: Are all directly related to the number of triangles, so they share its computational complexity.
  - *Max K Core*: can be calculated in linear time  $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ , but only if the graph is created, which is not possible in our case, because we are going to estimate the best strategy without creating the graph itself!.

### Imputing memory missing values

By knowing at least the result when we run our test on a specific hardware with a graph using a fixed number of threads, we can predict the ram usage of other test results on the same hardware and same graph using linear regression with a good precision. This has been observed initially in the previous chapter.

For that reason, we will complete missing memory values using linear regression for each graph and hardware couple, these diagrams represent that process :

---

<sup>1</sup> $\omega \approx 2.376$  is the fast matrix multiplication exponent.

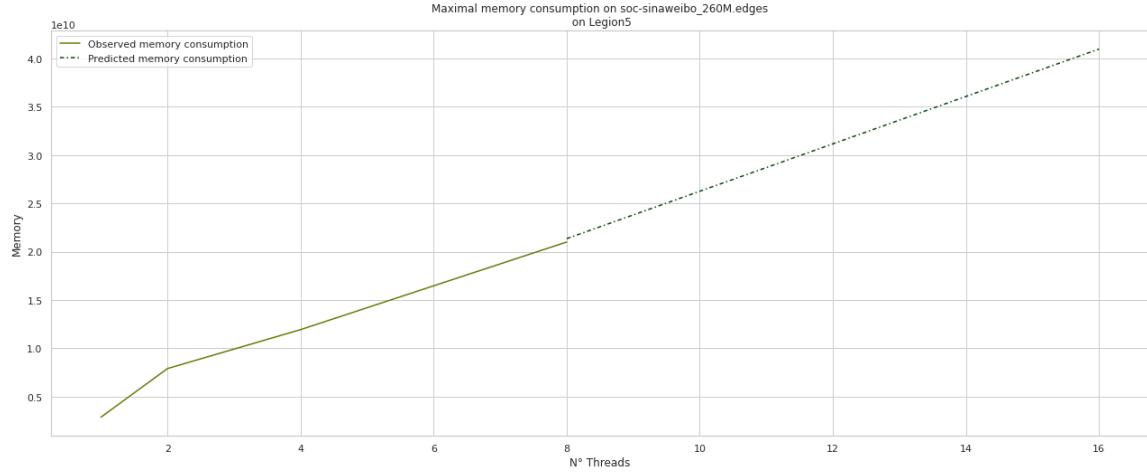


Figure 6.4: Linear regression on ideapad machine

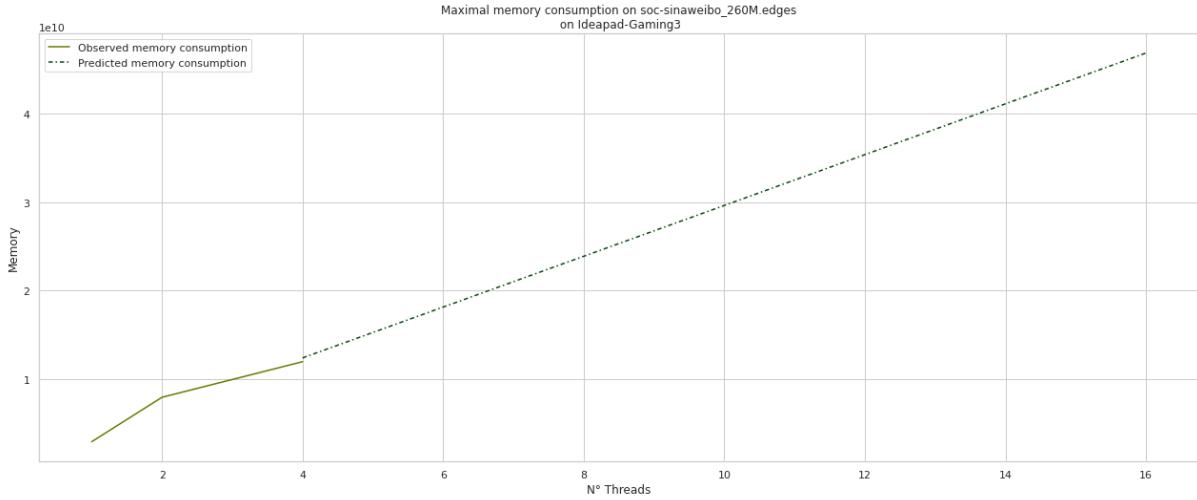


Figure 6.5: Linear regression on legion5 machine

### 6.1.3 Transformation

In this step we transformed a few of our values to ensure easier manipulation and readability later on.

- we converted all time fields into *seconds*
- we converted the memory to *Gigabyte*

### 6.1.4 Data Mining

#### Data Mining Memory and time

We started by building two models. One to predict the approximate time the graph creation and the other to predict the memory used. These models will later be used in our final model to predict the best number of threads we can use.

### Predicting number of threads that maximizes memory usage

Having the Time Model and the memory model, we can now search for the optimal number of threads that minimizes the graph creation time without exhausting the memory. So we created a Graph Strategy Prediction model that predicts the number of threads using the following algorithm:

---

**Algorithm 1** Number of threads prediction

---

```
nbThreadsBest ← 1
timeUsedMin ← +∞
while nbThreads ≤ limit do
    memoryUsed ← memoryModel(nbThreads)
    timeUsed ← timeModel(nbThreads)
    if timeUsed < timeUsedMin then
        if MemoryUsed < memoryLimit then
            timeUsedMin ← timeUsed
            nbThreadsBest ← nbThreads
        end if
    end if
end while
```

---

#### 6.1.5 Model Selection

Initially We selected the following 4 models:

- Linear Support vector Machine: It gives us the flexibility to define how much error is acceptable in our model and will find an appropriate line (or hyperplane in higher dimensions) to fit the data.
- Linear Regression: The model tries to fit the data into one linear function and make certain predictions based on that linear function.
- Decision tree: is a model that generates a tree that represents rules to make certain predictions.
- Random Forest: is an ensemble learning method based on decision tree which is derived from the bagging method to reduce the over-fitting of the decision tree model.

Also We will consider the following 3 feature sets:

- Available Features
- Calculable Features
- Calculable + Hard Features

Which will give us  $3 \times 4 = 12$  possible models. We will then test the performance of each one.

## 6.2 Results

As we are following the KDD methodology, and as we selected and preprocessed our data set now we are going to select features that we are going to work with, so initially we are going to use Available features then we evaluate the model performance and we go back to the data mining stage to select other set of features (Calculable features and Calculable + Hard features). Mainly, we implemented two testing methods :

### 6.2.1 Train test split

As a first estimation of the model performance, we used Train Test split technique. Here we split our data set into Train and Test data sets with 70% for the train set and 30% for the test set. The performance indicator is  $R^2$ [2] :

	Available Features	Calculable Features	Calculable + Hard Features
LinearSVR	0.391	0.515	0.447
LinearRegressor	0.398	0.642	0.554
DecisionTreeRegressor	0.943	0.986	0.934
RandForestRegressor	0.926	0.965	0.988

Table 6.1: Using train test split validation on memory models

	Available Features	Calculable Features	Calculable + Hard Features
LinearSVR	0.531	0.606	0.578
LinearRegressor	0.595	0.655	0.645
DecisionTreeRegressor	0.958	0.927	0.947
RandForestRegressor	0.936	0.977	0.975

Table 6.2: Using train test split validation on time models

We can observe that two tree based models (Decision Tree and Random Forest) performed better than the other two models (Linear Regression and Support Vector Machine). This has to do with the fact that the latter are both geometric models<sup>2</sup>, but in fact our features do not have a geometric interpretation.

### 6.2.2 Cross Validation

As the results show, we got a very high model performance, which led us to suspect that our model is experiencing data leakage<sup>3</sup>. So to have a better estimation of the error, we Used cross validation with **10 folds**. These are our results for the different models tested :

So we deduce that indeed, there was some leakage from Training set to Testing set, and also the results confirm that the geometric models perform worse than tree based models.

---

<sup>2</sup>Which supposes that the input features are members of a Hilbert/Euclidean Space

<sup>3</sup>Data Leakage[28] is a situation in machine learning and statistics on which the given (training) data contains unexpected extra information about the subject it is estimating.

	Available Features	Calculable Features	Calculable + Hard Features
LinearSVR	0.355	0.293	0.296
LinearRegressor	-3.465	-2.475	-2.524
DecisionTreeRegressor	0.750	0.899	0.897
RandForestRegressor	0.838	0.934	0.939

Table 6.3: Using 10-fold cross validation on memory models

	Available Features	Calculable Features	Calculable + Hard Features
LinearSVR	-1.164	-1.207	-2.064
LinearRegressor	-5.603	-3.976	-3.886
DecisionTreeRegressor	0.672	0.558	0.547
RandForestRegressor	0.726	0.749	0.757

Table 6.4: Using 10-fold cross validation on time models

Also the we can deduce also that the Decision Tree regression model was overfitting, but it was not detected in the Train Test split phase due to data leakage

Finally, while Random Forest was also leaking, it did have better overall performance as Random Forest is more resilient to the over-fitting phenomenon. So our memory and time models will be a *RandomForestRegressor*, using Calculable Features, as it is not practical to extract Hard Features in practice.

### 6.3 Final Model

Our final model is based on two *RandomForestRegressor*, one for time prediction, and the other for memory prediction, and it predicts the number of threads based on the algorithm described in section 5.1.4

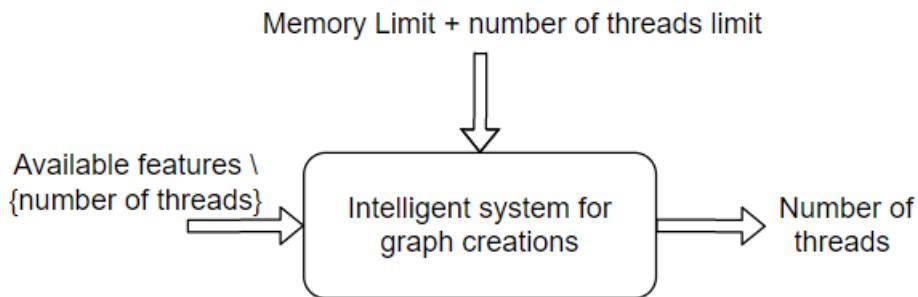


Figure 6.6: Final Model

## 6.4 Conclusion

In this chapter we presented different stages we went through to create the machine learning models that predict the best strategy to create the graph based on the characteristics of the hardware and details about the graph we want to create.

# Conclusion

Throughout the preparation of our end-of-year project, we wanted to tackle the graph creation problem using a clear methodology. From evaluating the different strategies and implementations until we eventually build an intelligent model that can predict the best creation strategy.

We started by specifying the context and by modelling our problem mathematically in order to have a clear vision on what we want to build.

In chapter 2, We presented our design and the different design principles we followed in order to ensure the flexibility and resiliency of our work product.

In chapter 3, we presented the different tools used and our general process, in addition to the different challenges we faced while building the Monitoring software that was used later to generate inputs to be used the machine learning phase.

In chapter 4, we analyzed the different results we gathered after we run the software on multiple machines. Which helped later on while building the machine learning pipeline.

And finally, we presented our machine learning methodology, explained in details its different steps and presented the results to get a working intelligent system for graph creation.

# Bibliography

- [1] .
- [2] Shwetha Acharya. *What are RMSE and MAE?* <https://towardsdatascience.com/what-are-rmse-and-mae-e405ce230383>. 2022.
- [3] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *CoRR* abs/1308.3432 (2013). arXiv: 1308.3432. URL: <http://arxiv.org/abs/1308.3432>.
- [4] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.
- [5] Matthieu Courbariaux and Yoshua Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: <http://arxiv.org/abs/1602.02830>.
- [6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *CoRR* abs/1511.00363 (2015). arXiv: 1511.00363. URL: <http://arxiv.org/abs/1511.00363>.
- [7] Elizabeth Gibney. “Self-taught AI is best yet at strategy game Go”. In: *Nature* (Oct. 2017). DOI: 10.1038/nature.2017.22858.
- [8] Citu Group. *What is the carbon footprint of a house?* URL: <https://citu.co.uk/citu-live/what-is-the-carbon-footprint-of-a-house>.
- [9] Koen Helwegen et al. “Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization”. In: *CoRR* abs/1906.02107 (2019). arXiv: 1906.02107. URL: <http://arxiv.org/abs/1906.02107>.
- [10] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.
- [11] IBM. *Présentation générale de CRISP-DM*. <https://www.ibm.com/docs/fr/spss-modeler/saas?topic=dm-crisp-help-overview>. 2017.
- [12] A.G. Ivakhnenko et al. *Cybernetics and Forecasting Techniques*. Modern analytic and computational methods in science and mathematics. American Elsevier Publishing Company, 1967. ISBN: 9780444000200. URL: <https://books.google.tn/books?id=rGFgAAAAIAAJ>.
- [13] Zohar Jackson. *Free Spoken Digits Dataset*. <https://github.com/Jakobovski/free-spoken-digit-dataset>. 2019.
- [14] Jonathan L. Gross, Jay Yellen, Ping Zhang. *Handbook of Graph Theory*. 17 Dec. 2013.
- [15] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

- [16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>.
- [17] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.
- [18] Yann LeCun. *THE MNIST DATABASE*. <http://yann.lecun.com/exdb/mnist>. 1998.
- [19] Xiaofan Lin, Cong Zhao, and Wei Pan. “Towards Accurate Binary Convolutional Neural Network”. In: *CoRR* abs/1711.11294 (2017). arXiv: 1711.11294. URL: <http://arxiv.org/abs/1711.11294>.
- [20] Seppo Linnainmaa. “Taylor Expansion of the Accumulated Rounding Error”. In: *BIT* 16.2 (1976), 146–160. ISSN: 0006-3835. DOI: 10.1007/BF01931367. URL: <https://doi.org/10.1007/BF01931367>.
- [21] Zechun Liu et al. “Bi-Real Net: Enhancing the Performance of 1-bit CNNs With Improved Representational Capability and Advanced Training Algorithm”. In: *CoRR* abs/1808.00278 (2018). arXiv: 1808.00278. URL: <http://arxiv.org/abs/1808.00278>.
- [22] Kim Martineau. *Shrinking deep learning’s carbon footprint*. 7Aug 2021. URL: <https://news.mit.edu/2020/shrinking-deep-learning-carbon-footprint-0807>.
- [23] Matthieu Latapy. “Practical algorithms for triangle computations in very large (sparse (power-law)) graphs”. In: (2007).
- [24] Rajkumar Roy. *Industrial Knowledge Management*. 2001.
- [25] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *CoRR* abs/1603.05279 (2016). arXiv: 1603.05279. URL: <http://arxiv.org/abs/1603.05279>.
- [26] Robert C. Martins. *Clean Code*. 2008.
- [27] Robin J. Wilson. “Introduction to Graph theory”. In: (1970).
- [28] Prerna Singh. *Data Leakage in Machine Learning: How it can be detected and minimize the risk*. <https://towardsdatascience.com/data-leakage-in-machine-learning-how-it-can-be-detected-and-minimize-the-risk-8ef4e3a97562>. 2022.
- [29] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP”. In: *CoRR* abs/1906.02243 (2019). arXiv: 1906.02243. URL: <http://arxiv.org/abs/1906.02243>.
- [30] *The C++ container library*. <https://en.cppreference.com/w/cpp/container>. 2011.
- [31] Tim Dettmers. *Deep Learning in a Nutshell: History and Training*. <https://developer.nvidia.com/blog/deep-learning-nutshell-history-training/>. 16 Dec. 2015.
- [32] Vegard Flovik. “What is Graph Theory, and why should you care?” In: (12 Aug. 2020).
- [33] Aimee Wynsberghe. “Sustainable AI: AI for sustainability and the sustainability of AI”. In: *AI and Ethics* 1 (Feb. 2021). DOI: 10.1007/s43681-021-00043-6.
- [34] Chunyu Yuan and Sos S. Agaian. “A comprehensive review of Binary Neural Network”. In: *CoRR* abs/2110.06804 (2021), pp. 3–4. arXiv: 2110.06804. URL: <https://arxiv.org/abs/2110.06804>.

- [35] Chunyu Yuan and Sos S. Agaian. “A comprehensive review of Binary Neural Network”. In: *CoRR* abs/2110.06804 (2021). arXiv: 2110 . 06804. URL: <https://arxiv.org/abs/2110.06804>.