Synthèse de code virgule fixe pour les réseaux de neurones

Hanane Benmaghnia

THÈSE

Pour obtenir le grade de

Docteur

Délivré par

UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale

Énergie et Environnement ED305

Et de l'unité de recherche

LAMPS UR 4217

Spécialité : **Informatique**

Présentée par

BENMAGHNIA Hanane Zhor

Synthèse de Code Virgule Fixe

Pour les Réseaux de Neurones

Soutenue le 07 /10 / 2022 devant le jury composé de :

M. Mostafa EL HABIB DAHO, HDR, Université de Tlemcen Aboubekr Belkaid, Algérie	Rapporteur
M. Bertrand LE GAL, MCF, INP Bordeaux, France	Examinateur
M. Matthieu MARTEL, Professeur, Université de Perpignan Via Domitia, France	Directeur de Thèse
M. Claude MICHEL, HDR, Université de Nice Sophia Antipolis, France	Rapporteur
M. Guillaume REVY, MCF, Université de Perpignan Via Domitia, France	Examinateur
Mme Yassamine SELADJI, HDR, Université de Tlemcen Aboubekr Belkaid, Algérie	Co-Directrice de Thèse
M. Xavier THIRIOUX, Professeur, ISAE-SUPAERO Toulouse, France	Co-Encadrant

Projet cofinancé par le Fonds Européen de Développement Régional

À toi, à moi,
à la vie, à la science,
et à tout être vivant ayant croisé mon chemin...

Remerciements

♣ ◇ ♣

Cette thèse et ces trois années au LAMPS, n'auraient pu être menées à bien sans l'aide du *Pr. Matthieu MARTEL* et *HDR Yassamine SELADJI*.

Matthieu, je tiens à te remercier de m'avoir donné l'opportunité d'intégrer le monde de la recherche. Mes stages de Master sous ta direction et celle de Yasmine, m'ont ouvert l'appétit et donné envie de poursuivre en thèse. Je te suis reconnaissante de m'avoir embarqué dans cette aventure, qui m'a permis de grandir sur le plan professionnel et personnel. La thèse est un savoir-être après tout !

Yasmine, je te remercie chaleureusement d'avoir co-dirigé cette thèse. Tu as cru en moi et tu as su me motiver chaque matin, même pendant les périodes difficiles. Tes remaques exigeantes et pointilleuses, m'ont poussé à aller chercher la meilleure version de ce que je proposais. Ta positivité était contagieuse et cela faisait du bien. Je vous remercie *Matthieu* et toi-même d'avoir relu mon manuscrit et tous mes articles. Mille mercis à vous deux !

Ma gratitude va à *HDR Mostafa EL-HABIB DAHO* et *HDR Claude MICHEL*, qui ont accepté de rapporter ma thèse. Leurs retours et suggestions m'ont permis d'améliorer mon manuscrit et de déterminer les pistes à explorer au futur. J'adresse mes remerciements à *MCF Guillaume REVY* et *MCF Bertrand LE GAL* d'avoir accepté de faire partie de mon jury. Les échanges avec mon co-encadrant *Pr. Xavier THIRIOUX*, étaient très intéressants et constructifs pendant les événements et les réunions annuelles. Merci *Xavier*.

Ce travail n'aurait pas été possible sans le soutien de la Région Occitanie, qui m'a permis, grâce à une allocation de recherche pour le projet SYFI, de me consacrer sereinement à l'élaboration de ma thèse. Je suis également reconnaissante à l'Union Européenne, qui m'a octroyé une bourse d'excellence et permis de vivre une aventure à la française. Je remercie également L'École Doctorale ED305 pour son implication et son accompagnement pendant ces trois années, ainsi que l'UPVDoC. L'aventure associative avec l'UPVDoC fût très enrichissante pour moi sur tous les plans. J'ai appris à organiser un congré, à vulgariser mes travaux face à un public hétérogène et à développer ma créativité. Je remercie mes collègues : *Théo*, *Avilien*, *Christelle*, *Caroline*, *Diane*, *Moussa* et tous ceux que j'ai croisés.

Sylvia, *Michel*, *Robert*, et tous les membres du LAMPS, merci infiniment pour votre bienveillance et votre sympathie. Vous avez veillé à ce que l'environnement de travail soit agréable scientifiquement, techniquement et psychologiquement. Je remercie également

l'équipe VENUS, dans laquelle j'ai baigné ces trois années, ainsi que mes co-bureaux : *Farah*, les deux *Asma*, *Thuong*, *Dorra*, *Soufiane*, *Christophe*, *Nasrine* et *Sabrina* pour les échanges, le stress collectif et les délires pendant les événements.

Mes vifs remerciements vont aux *Delort* qui m'ont ouvert leurs portes pendant deux ans et demi et permis de vivre à la française, oups à la Catalane ! Grâce à vous, j'ai vaincu ma phobie des chiens. Je ne peux même plus m'en passer de *Yoda*, *Ramsès* et *Vodka*, maintenant. Mille mercis *Caro* et *Lolo*.

Imane, *Anaïs*, vous êtes les deux meilleures amies qu'il faut avoir dans sa vie. Vous m'avez toujours soutenu dans mes choix et cru en moi, quand même moi, je n'y croyais plus. L'aventure parisienne ne pourrait être que folle à vos côtés ! Je tiens à citer également mes alliées *Flore*, *Diana* et *Indiana*, dans mon long combat contre l'obésité. Vous êtes ma relation la plus intense et fusionnelle. Je suis tellement reconnaissante d'avoir croisé vos chemins, ainsi que ceux de *Arnaud*, *Julien* et *Jérémy*. Petit clin d'œil à mes petits poussins *MLMJALG*, tata vous aime plus que la Catalogne ! Je remercie également *Mélissa*, *Keren*, *Selma*, *Chihab*, *Mamie Nicole*, *Wassila* et *Noureddine* pour leur soutien et bienveillance.

Je garde le meilleur pour la fin et je remercie mes parents, mes frères et ma mamie. *Papa & Maman*, vous êtes les prunelles de mes yeux. Sans vous, je n'aurais jamais vu le jour ni connu la vie. Je tiens à remercier votre patience et votre compréhension en toutes circonstances, même quand je ne donnais pas signe de vie pendant des semaines... Sachez que je vous dois tout ! Chaque pas, chaque événement, chaque réussite n'est que le reflet de votre éducation et de votre image. Excusez-moi d'être dure parfois, souvent têtue mais c'est moi, votre enfant, votre petite fille, qui est devenue docteure maintenant ! *Djalal & Riad*, merci d'être vous et de me permettre de vivre le rôle de la grande sœur, pleinement. Réussissez, brillez ! *Mamie*, je sais que tu ne comprends rien à la science, mais tu m'as aidé à ta manière. Tu nous as appris à nous battre et à ne jamais baisser les bras. Tu en es l'exemple vivant. La victoire ne se donne pas, elle s'arrache ! Et pour finir, j'adresse mes remerciements à mes oncles, tantes, cousins et cousines sans exception ! Merci à tata *Wassila* et tonton *Mus* pour tous les bons moments ramadanesques perpignanais et tous les heureux événements.

Moltes mercès Perpinyà la catalana !

Table des matières

Table des figures

♣ ◇ ♣

Liste des tableaux

♣ ◇ ♣

Chapitre 1

« Les systèmes ont des sous-systèmes, et les sous-systèmes ont des sous-sous-systèmes et ainsi de suite à l'infini. C'est la raison pour laquelle nous recommençons chaque fois de zéro. »

— Alan Jay Perlis

Introduction

1.1 Contexte Général et Objectifs de la Thèse

De nos jours, les réseaux de neurones sont devenus de plus en plus populaires dans de nombreux domaines. Ils ont commencé à intégrer les systèmes embarqués et les systèmes critiques (*safety critical systems*). Ils jouent souvent un rôle important dans la prise de décision, par exemple dans les voitures à conduite autonome, les fusées, les robots, etc. Ces réseaux de neurones deviennent de plus en plus grands alors que les systèmes embarqués ont souvent des ressources limitées (mémoire, CPU, etc.) Par conséquent, l'utilisation et l'exécution des réseaux de neurones profonds [GBC16] sur des systèmes embarqués avec ressources limitées, présentent plusieurs nouveaux défis [GGSS19a, JDL+19, LTA16, LV20, JRZ+22, JGM+20, GPMG18, IM19, GMD+18] en termes de compression, de robustesse et de validation.

Nous avons d'une part, les réseaux de neurones qui sont connus pour leur consommation de ressources. Plus le réseau est profond, plus son besoin en ressources est élevé, c'est-à-dire que des ordinateurs puissants sont utilisés pour effectuer les calculs. Et d'autre part, nous avons les systèmes embarqués munis de matériels contraints (mémoire, CPU, etc.) Les micro-processeurs employés dans ces sytèmes embarqués ne disposent pas toujours d'unités de calcul flottant (FPU). Il revient aux développeurs de gérer la représentation des nombres qu'ils manipulent. Ces nombres sont représentés dans l'arithmétique à virgule fixe, où le nombre de bits devant et derrière la virgule doit être indiqué à la machine pour toutes les valeurs. Une des solutions proposées pour exécuter des réseaux de neurones sur des systèmes embarqués, ayant des ressources limitées, est de combiner l'arithmétique à virgule fixe et les réseaux de neurones. En d'autres termes, ça revient à porter ces réseaux de neurones sur des architectures à virgule fixe, car cette arithmétique est plus adaptée pour les systèmes embarqués, qui ont souvent un processeur fonctionnant uniquement avec des nombres entiers. L'arithmétique à virgule fixe permet d'utiliser des processeurs plus simples, au prix d'une montée en complexité de la partie logicielle du système. Intuitivement, une partie de la gestion des calculs prise en charge automatiquement par le matériel dans le cas de l'arithmétique flottante, doit être gérée manuellement par le développeur dans le cas de l'arithmétique fixe. Le développement logiciel s'en trouve alors considérablement alourdi.

Les réseaux de neurones sont entraînés sur des ordinateurs dotés d'une unité de calcul puissante utilisant la plupart du temps l'arithmétique à virgule flottante IEEE754 [ANS19]. Le portage de ces réseaux à l'aide de l'arithmétique à virgule fixe peut perturber ou modifier leur réponse (sorties) et leur comportement (classification, interpolation). Ces réseaux de neurones sont généralement sensibles à l'arithmétique des ordinateurs. Et donc, une nouvelle approche est nécessaire pour adapter les calculs lourds des réseaux de neurones aux processeurs plus simples des systèmes embarqués, en utilisant l'arithmétique fixe. Cette dernière est basée sur des nombres à virgule fixe représentés par un signe, une valeur entière et un format fixe. Ces formats fixes indiquent le nombre de bits de la partie

entière et fractionnaire de chaque nombre à virgule fixe. Ils sont gérés manuellement par le développeur pendant les opérations arithmétiques via des décalages, contrairement à l'arithmétique flottante, qui le fait de manière automatique grâce à l'exposant. Le développeur doit porter une attention particulière aux débordements pendant la gestion des formats fixes. Il doit s'assurer qu'il y a assez de bits alloués pour la partie entière afin d'éviter les *overflows*, en cas de retenue dans les calculs. La bonne gestion (détermination) de ces formats est très importante car, à la moindre erreur ou décalage en trop, le résultat de l'opération arithmétique est faussé.

Dans cette thèse, nous considérons le problème de réglage des formats fixes d'un réseau de neurones déjà entraîné, c'est-à-dire que nous nous intéressons à la détermination du nombre de bits minimal des parties entières et fractionnaires de chaque coefficient, et de chaque sortie de ce réseau. Dans le cas des classifieurs, le nouveau réseau de neurones (à virgule fixe) doit classer correctement les sorties dans la bonne catégorie, par rapport au réseau d'origine (flottant) avec un seuil d'erreur défini par l'utilisateur, à ne pas excéder. Et dans le cas des interpolateurs, c'est-à-dire les réseaux calculant une fonction mathématique, le réseau flottant et le réseau à virgule fixe doivent se comporter de manière identique. Si ces derniers calculent une fonction donnée f, l'erreur entre les résultats numériques calculés, doit être inférieure ou égale au seuil d'erreur fixé par l'utilisateur. Après avoir obtenu les formats fixes minimaux pour chaque coefficient, la synthèse de code à virgule fixe peut avoir lieu. Idéalement, pour générer un code basé que sur des entiers de la manière la plus efficace, il serait judicieux d'utiliser la précision mixte [DHS18, CBB+17] dans le code généré. En d'autres termes, il faut assigner le plus petit type de données à chaque variable du programme, pour consommer le moins de mémoire (bits) possible. Notons que la précision mixte est plus complexe à mettre en place mais, elle offre un gain en mémoire important qui, n'est pas négligeable pour les systèmes embarqués.

L'objectif de ma thèse est de proposer des approches permettant de régler les formats fixes de manière optimale pour chaque coefficient du réseau de neurones, en utilisant la technique de réglage de précision via la génération des contraintes. Régler les formats de manière optimale revient à trouver le nombre de bits minimal pour représenter ces coefficients. Un autre objectif de cette thèse est d'automatiser ce processus de génération des contraintes et des calculs en virgule fixe, long et fastidieux à travers un outil, appelé `SyFix`. Ce dernier optimise les calculs en virgule fixe et synthétise un code pour le réseau donné en entrée. Ce code synthétisé doit assurer une efficacité en termes de précision, de rapidité et de mémoire utilisée. `SyFix` doit également s'assurer que le code généré respecte le seuil d'erreur à ne pas dépasser, choisi par l'utilisateur. Soulignons que le code généré par `SyFix` contient uniquement du calcul entier et a pour objectif d'être exécuté sur des systèmes embarqués critiques (voitures intelligentes, radars, équipements médicaux...).

Pour conclure, résumons les principaux objectifs de cette thèse en ces trois points :

- Régler les formats fixes du réseau de neurones en générant des contraintes. Ces dernières calculent les formats fixes minimaux pour chaque neurone et pour chaque

coefficient du réseau de neurones, en tenant compte du seuil d'erreur et du type de données (maximal) dans le code synthétisé, requis par l'utilisateur.

- Convertir le réseau de neurones en utilisant les formats fixes calculés précédemment. Ces derniers sont obtenus via la résolution du système de contraintes via `Z3` [dMB08].
- Synthétiser un code à virgule fixe en C pour le réseau de neurones flottant, donné en entrée. Ce code synthétisé est basé sur des entiers et des opérations entières uniquement. Il garantit également le seuil d'erreur et le type de données, exigés par l'utilisateur.

1.2 Contributions de la Thèse

Les principales contributions de cette thèse sont présentées dans cette section. Ces contributions ont pour but de calculer le format fixe minimal pour chaque sortie de neurone et pour chaque coefficient du réseau de neurones, tout en respectant le type de données et le seuil d'erreur, à ne pas dépasser. Ces formats fixes sont requis pour synthétiser le code à virgule fixe correspondant au réseau de neurones donné en entrée.

1.2.1 Contribution 1 : Calcul des Erreurs et de leur Bits de Poids Fort dans un Réseau de Neurones

La première contribution consiste à borner les erreurs commises dans un réseau de neurones. Pour cela, nous définissons d'abord les opérations élémentaires (addition, multiplication), ainsi que les fonctions d'activation (*linéaire, unité linéaire rectifiée, sigmoïde, tangente hyperbolique*) en virgule fixe, requises dans les réseaux de neurones. Ensuite, nous calculons l'erreur correspondante à la somme pondérée des poids, basée sur les multiplications et les additions à virgule fixe. Après cela, nous bornons également l'erreur de chaque fonction d'activation ayant pour entrée la somme pondérée des poids. Et pour finir, cette contribution détermine le bit de poids fort des erreurs correspondantes à la somme pondérée des poids, et aux fonctions d'activation. Ce bit de poids fort servira dans la génération des contraintes calculant le nombre de bits minimal des parties fractionnaires des coefficients à virgule fixe.

1.2.2 Contribution 2 : Calcul des Formats Fixes via la Méthode `CubMeth`

La deuxième contribution concerne la méthode `CubMeth` [BMS22b]. Elle a pour but de calculer le format fixe minimal de chaque sortie de neurones et de chaque poids synaptique du réseau de neurones. Notons que, `CubMeth` attribue un format fixe pour chaque sortie de neurone et pour chaque poids synaptique dans la matrice des poids. Pour déterminer ces formats fixes, cette méthode calcule d'abord le nombre de bits de la partie entière des coefficients via une analyse dynamique du réseau. Cette analyse donne une sous-approximation des plages (*ranges*) des entrées, des sorties et des coefficients du réseau

de neurones. Ensuite, elle génère automatiquement des contraintes pour déterminer le nombre de bits minimal des parties fractionnaires des poids synaptiques et des neurones. Ces contraintes sont modélisées en prenant en considération le seuil d'erreur à ne pas dépasser, le type de données requis par l'utilisateur et la propagation des erreurs de calculs en avant[1] et en arrière[2]. `CubMeth` résout ces contraintes avec un solveur, et plus précisément le solveur `Z3` [dMB08]. Notons que le nombre de contraintes générées par cette méthode croît de manière cubique (complexité cubique) dans le pire des cas.

1.2.3 Contribution 3 : Calcul des Formats Fixes via la Méthode `QuadMeth`

La troisième contribution concerne la méthode `QuadMeth`. Cette méthode a le même but et suit le même principe que `CubMeth`. Contrairement à `CubMeth`, la méthode `QuadMeth` attribue un format fixe par neurone et un format fixe par ligne dans la matrice de poids synaptiques, ce qui rend la complexité du nombre de ses contraintes, quadratique, dans le pire des cas. La spécificité de `QuadMeth` est qu'elle génère moins de contraintes que `CubMeth`. La première approche génère un nombre de contraintes croissant de manière cubique et ne peut traiter que des réseaux de petites tailles, avec un temps de résolution assez élevé par le solveur `Z3` [dMB08]. Et cette méthode génère un nombre de contraintes croissant de manière quadratique, ce qui la rend plus rapide que `CubMeth`.

1.2.4 Contribution 4 : Calcul des Formats Fixes via la Méthode `LinMeth`

La quatrième contribution concerne la méthode `LinMeth`. Cette dernière a le même but et suit le même principe également que les deux méthodes `CubMeth` et `QuadMeth`, mais elle attribue un format fixe par couche de neurones et un format fixe par matrice de poids synaptiques. Le nombre de contraintes générées par `LinMeth` augmente de manière linéaire (complexité linéaire) dans le pire des cas, ce qui la rend plus rapide que les deux autres méthodes mais moins performante en termes de nombres bits optimisés.

1.2.5 Contribution 5 : Librairie Virgule Fixe

La cinquième contribution de cette thèse est l'implémentation de la librairie virgule fixe, appelée `SyFi`. Cette dernière implémente d'une part, les opérations élémentaires (addition, multiplication) requises dans les réseaux de neurones, ainsi que les versions approchées des fonctions d'activation. Et d'autre part, `SyFi` implémente les opérations et fonctions requises par `POPiX` (Contribution 1.2.7) telles que la division, la valeur absolue, la racine carrée et les fonctions trigonométriques.

[1] De la première couche vers la dernière couche du réseau de neurones.
[2] De la dernière couche vers la première couche du réseau de neurones.

1.2.6 Contribution 6 : Outil de Synthèse de Code à Virgule Fixe pour les Réseaux de Neurones

Les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, respectivement mentionnées dans les contributions 1.2.2, 1.2.3 et 1.2.4, sont implémentées dans l'outil `SyFix`. Cet outil a deux objectifs fondamentaux : le réglage des formats fixes et la synthèse de code à virgule fixe pour un réseau de neurones donné en entrée. Les principales étapes de `SyFix` pour générer un code à virgule fixe, respectant un seuil d'erreur et un type de données à ne pas dépasser sont : récupérer les valeurs des poids synaptiques, des biais et des entrées du réseau ; effectuer une analyse dynamique pour obtenir les intervalles des valeurs des sorties et des entrées du réseau ; récupérer le nombre de bits des parties entières de tous les coefficients ; générer automatiquement les contraintes pour trouver le nombre de bits minimal des parties fractionnaires des coefficients, et les résoudre avec `Z3` [dMB08] ; synthétiser le code à virgule fixe à l'aide de la librairie `SyFi`.

1.2.7 Contribution 7 : Synthèse de Code à Virgule Fixe pour les Programmes

La dernière contribution concerne l'outil `POPiX` [BKBM22]. Il résulte d'une collaboration entre *Sofiane Bessai*[3], *Dorra Ben Khalifa*[4], *Matthieu Martel*[5] et moi-même. Dans ce travail, l'outil `POP` [Ben21] et la librairie `SyFi` sont interférés. `POPiX` synthétise un code à virgule fixe pour un programme flottant et une précision, donnés en entrées. Une fois que le nombre de bits minimal requis pour chaque variable est obtenu, le code à virgule fixe est synthétisé à l'aide de la librairie `SyFi`. `POP` parcourt le programme en entrée, analyse les *ranges* des variables, règle la précision de ces dernières en générant des contraintes, et en les résolvant par le solveur `GLPK` [Mak12].

1.3 Plan de la Thèse

Ce manuscrit de thèse se compose principalement de quatre parties, où chacune englobe plusieurs chapitres. Il est organisé comme suit :

1.3.1 Première Partie : État de l'Art

Cette première partie intitulée : "*État de l'Art*" se compose de trois chapitres. Ces derniers présentent les notions pré-requises dans cette thèse, ainsi que quelques travaux se trouvant dans la littérature.

- Le Chapitre 2 concerne l'arithmétique des ordinateurs. Il met en avant les principales définitions et propriétés des deux arithmétiques à virgule flottante et à virgule

[3] https://www.linkedin.com/in/sofiane-bessai/
[4] https://dbenkhal.github.io/index.html
[5] https://perso.univ-perp.fr/matthieu.martel/

fixe. Les principales propriétés présentées pour les deux arithmétiques sont : la représentation des nombres à virgule flottante (respectivement à virgule fixe), les opérations arithmétiques, les valeurs spéciales et les modes arrondis. À la fin de ce chapitre, une comparaison entre l'arithmétique à virgule flottante et l'arithmétique à virgule fixe, est effectuée.

- Le Chapitre 3 donne un aperçu sur les réseaux de neurones, en présentant leurs composition, fonctionnement et types existants. Les fonctions d'activation, utilisées dans cette thèse (*linéaire, unité linéaire rectifiée, sigmoïde, tangente hyperbolique*), sont définies dans les deux arithmétiques à virgule flottante et à virgule fixe. Ce chapitre présente également quelques outils de compression de réseaux de neurones utilisant différentes techniques.

- Le Chapitre 4 est divisé en trois parties. Dans sa première partie, il met en avant la satisfiabilité modulo théories (SMT) ainsi que la programmation linéaire, permettant la résolution des problèmes SMT et des problèmes linéaires respectivement. La première partie de ce chapitre cite également quelques solveurs pour chaque technique. Les outils dédiés au réglage de précision des programmes et des réseaux de neurones, sont présentés dans la deuxième partie de ce chapitre. Parmi les outils présentés, il y a ceux qui règlent la précision pour des programmes ou réseaux de neurones flottants, et ceux qui règlent la précision pour des programmes ou réseaux de neurones à virgule fixe. La dernière partie du Chapitre 4 présente quelques outils synthétisant du code à virgule fixe pour les programmes, ainsi que pour les réseaux de neurones.

1.3.2 Deuxième Partie : Formats Fixes des Neurones et des Poids Synaptiques

La deuxième partie de ce manuscrit aborde la détermination et le réglage des formats fixes des neurones et des poids synaptiques, ainsi que le calcul des erreurs commises dans un réseau de neurones. Elle est constituée de quatre chapitres.

- Le Chapitre 5 modélise en premier lieu les erreurs résultantes aux opérations élémentaires (addition et multiplication) en virgule fixe et les erreurs correspondantes aux deux fonctions mathématiques *sigmoïde* et *tangente hyperbolique*. Ces erreurs sont requises pour calculer les erreurs de la somme pondérée des poids et des fonctions d'activation, qui sont également définies dans ce chapitre. Le bit de poids fort de l'erreur commise dans un réseau de neurones est donné dans ce chapitre aussi.

- Le Chapitre 6 concerne la méthode `CubMeth`, calculant les formats fixes des poids synaptiques et des sorties des neurones. Les contraintes générées par `CubMeth`, ainsi que le nombre et la complexité de ces contraintes, sont présentés dans ce chapitre.

- Le Chapitre 7 présente la technique utilisée par `QuadMeth` et `LinMeth`, pour calculer les formats fixes des poids synaptiques et des sorties des neurones. Il présente

également les contraintes générées par `QuadMeth` ainsi que le nombre et la complexité de ces dernières.

1.3.3 Troisième Partie : Évaluation des Performances de `SyFix`

La troisième partie de la thèse porte sur l'outil `SyFix` et ses performances. Elle contient deux chapitres.

- Le Chapitre 8 met en avant notre outil `SyFix` et ses fonctionnalités. Il décrit également notre librairie virgule fixe `SyFi`, ainsi que les fonctions qu'elle implémente.

- Le Chapitre 9 évalue les performances des trois méthodes `CubMeth`, `QuadMeth` et `LinMeth` via `SyFix`. Ces méthodes sont évaluées en termes de précision, du respect du seuil d'erreur et du type de données (fixés par l'utilisateur), de bits/bytes optimisés, et de distribution des types de données des variables dans le programme synthétisé. Les réseaux de neurones (*benchmarks*) utilisés pendant les expérimentations sont également présentés dans ce chapitre.

1.3.4 Quatrième Partie : Synthèse de Code à Virgule Fixe pour les Programmes Via `POPiX`

La dernière partie de ce manuscrit est constituée d'un seul chapitre portant sur l'outil `POPiX`.

- Le Chapitre 10 présente l'outil `POPiX`. Cet outil est une intégration des outils `POP` [Ben21] et notre librairie virgule fixe `SyFi`. Dans ce chapitre, nous décrivons le fonctionnement de `POP`, ensuite nous présentons la technique utilisée par `POPiX` pour synthétiser des programmes à virgule fixe, satisfaisant un certain besoin en précision. L'évaluation de `POPiX`, ainsi que les *benchmarks* utilisés, sont également mis en avant dans ce chapitre.

1.4 Publications et Manifestations Scientifiques

1.4.1 Conférences Internationales avec Comité de Lecture

- Hanane Benmaghnia, Matthieu Martel and Yassamine Seladji. **Fixed-Point Code Synthesis for Neural Networks.** *In 6th International Conference on Artificial Intelligence, Soft Computing and Applications, volume 12 of Computer Science & Information Technology, pages 11–30. CSCP, 2022*

- Sofiane Bessaï, Dorra Ben Khalifa, Hanane Benmaghnia, Matthieu Martel. **Fixed-Point Code Synthesis Based on Constraint Generation.** *In the 15th International Workshop on Design and Architectures for Signal and Image Processing, DASIP22.*

1.4.2 Articles dans des Revues Scientifiques

- Hanane Benmaghnia, Matthieu Martel and Yassamine Seladji. **Code Generation For Neural Networks Based On Fixed-Point Arithmetic**. *In Journal of Transactions on Embedded Computing Systems*, Accepted.

1.4.3 Résumés

- Hanane Benmaghnia, Matthieu Martel and Yassamine Seladji. **Synthèse de Code Virgule Fixe pour les Réseaux de Neurones**. *In Embedded-IA, Virtual, 2020*.

1.4.4 Posters

- Hanane Benmaghnia, Matthieu Martel and Yassamine Seladji. **Adapting Neural Networks to Fixed-Point and Integer Arithmetic**. *In IndabaXAlgeria. University of Tlemcen Algeria, 2019*.

- Hanane Benmaghnia, Matthieu Martel and Yassamine Seladji. **Adapting Neural Networks to Fixed-Point and Integer Arithmetic**. *In FEANICSES. ISAE-SUPAERO Toulouse, 2019*.

1.4.5 Événements Scientifiques

- **Organisation et Animation de la Fête de la Science autour du Thème : Informatique avec des Billes.**, *Village des Sciences, Université de Perpignan, 2019.*

- **Organisation du Neuvième Congrès des Doctorants**, *Association des Doctorants de Perpignan (UPVDoc), Virtuel, 2020.*

Première partie

État de l'Art

Chapitre 2

« En bonne arithmétique, un plus un égale tout et deux moins un égale rien. »

— Ninon de Lenclos

Arithmétique des Ordinateurs

♣ ◇ ♣

Les nombres réels ne peuvent être représentés exactement en mémoire sur machine. Cependant, il existe plusieurs arithmétiques, pour effectuer les calculs et représenter les nombres, telles que l'arithmétique à virgule flottante [ANS19, MBD+18], l'arithmétique à virgule fixe [MNR17, BS17, Yat09, MCS06, Lop14] et l'arithmétique des intervalles [Moo79]. Nous abordons uniquement les deux arithmétiques à virgule flottante et fixe dans ce chapitre. Ce dernier est organisé comme suit :

- Des définitions communes aux deux arithmétiques à virgule flottante et fixe, sont données dans la Section 2.1.
- La Section 2.2 donne un aperçu sur l'arithmétique à virgule flottante et la norme IEEE754, plus précisément. Les différentes propriétés de cette norme sont présentées telles que la représentation des nombres à virgule flottante et des valeurs spéciales, la gestion des exceptions, les arrondis et les opérations élémentaires.
- La Section 2.3 met en avant l'arithmétique à virgule fixe et ses différentes propriétés telles que la représentation des nombres à virgule fixe, les arrondis et les opérations arithmétiques. Quelques librairies et algorithmes existants sont également présentés dans cette section.
- Une comparaison entre les deux arithmétiques est effectuée dans la Section 2.4.
- La Section 2.5 conclut ce chapitre.

2.1 Définitions Communes

Dans cette section, nous présentons des définitions communes aux deux arithmétiques : à virgule fixe et à virgule flottante. La formule de calcul du bit de poids fort (respectivement de poids faible) d'un nombre est donnée par la Définition 2.1.1 (respectivement Définition 2.1.2). Les deux erreurs absolue et relative sont également présentées dans les définitions 2.1.3 et 2.1.4, respectivement. Les notations des ensembles des nombres utilisées dans ce chapitre sont : $\mathbb{N}$ l'ensemble des entiers, $\mathbb{Z}$ l'ensemble des entiers relatifs (négatifs et positifs), $\mathbb{R}$ l'ensemble des réels, $\mathbb{F}$ l'ensemble des nombres à virgule flottante et $\aleph$ l'ensemble des nombres à virgule fixe.

2.1.1 Ufp : Unit in the First Place

L'ufp, ou *unit in the first place*, est une unité de mesure de la précision des nombres à virgule flottante ou fixe. Elle correspond au bit de poids fort (premier bit non nul à gauche) d'un nombre à virgule flottante ou fixe.

Définition 2.1.1. *L'ufp d'un nombre à virgule flottante [MBD+18] est défini par*

$$\forall x \in \mathbb{F}, \quad \mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0, \\ 0 & \text{if } x = 0, \end{cases} \tag{2.1}$$

où $\lfloor . \rfloor$ représente la partie entière inférieure.

2.1.2 Ulp : Unit in the Last Place

L'ulp, ou *unit in the last place*, est une unité de mesure de la précision des nombres à virgule flottante ou fixe. Elle correspond au bit de poids faible (dernier bit significatif à droite) d'un nombre à virgule flottante ou fixe.

Définition 2.1.2. *L'ulp d'un nombre flottant dépend de son ufp. Cette unité est donnée avec*

$$\forall x \in \mathbb{F}, \quad \mathsf{ulp}(x) = \mathsf{ufp}(x) - w + 1, \tag{2.2}$$

où w représente le nombre de bits significatifs de x.

Remarque 1. *Cette définition n'est pas unique. Il existe d'autres définitions permettant de calculer l'ulp dans [Gol91, Mul05, Har99, Kah04].*

L'erreur absolue et l'erreur relative sont des mesures de quantification des erreurs. Informellement, ces deux mesures quantifient la différence entre le résultat d'une expression évaluée sur les réels et le résultat de la même expression calculée sur les nombres à virgule flottante (ou fixe), respectivement en valeur absolue et en valeur relative.

2.1.3 Erreur Absolue

L'erreur absolue est la distance entre une valeur exacte sur les réels et sa valeur approchée sur les nombres à virgule flottante (ou fixe). La Définition 2.1.3 donne l'expression pour calculer l'erreur absolue.

Définition 2.1.3. *La formule de l'erreur absolue, notée δ_{abs}, entre un nombre réel x et un nombre à virgule flottante $\tilde{x}$, est donnée avec*

$$\forall x \in \mathbb{R}, \quad \forall \tilde{x} \in \mathbb{F}, \quad |x - \tilde{x}| \leqslant \delta_{abs}. \tag{2.3}$$

2.1.4 Erreur Relative

L'erreur relative est égale à l'erreur absolue divisée par la valeur exacte. Contrairement à l'erreur absolue, l'erreur relative prend en compte la taille des valeurs mesurées et n'est pas dépendante de leur magnitude. La Définition 2.1.4 donne la formule correspondante à l'erreur relative.

Définition 2.1.4. *L'expression de l'erreur relative, notée δ_{rel}, entre un nombre réel x et un nombre à virgule flottante $\tilde{x}$, est donnée avec*

$$\forall x \in \mathbb{R}^*, \quad \forall \tilde{x} \in \mathbb{F}, \quad \left| \frac{x - \tilde{x}}{x} \right| \leqslant \delta_{rel}. \tag{2.4}$$

2.2 Arithmétique à Virgule Flottante

Les nombres à virgule flottante, appelés nombres flottants, sont utilisés dans les programmes informatiques pour approcher des valeurs de type réel. Jusque dans les années 80, chaque constructeur avait sa propre implémentation de l'arithmétique à virgule flottante (base de représentation, plage d'exposants, etc.), et donc un même programme informatique donnait différents résultats sur différentes architectures (avec les mêmes entrées). En 1985, la standardisation et l'homogénéisation de l'implémentation de l'arithmétique à virgule flottante (appelée arithmétique flottante par la suite) a eu lieu grâce à la norme IEEE754 [ANS85]. Cette dernière a été révisée en 2008 [ANS08], puis en 2019 [ANS19]. Cette norme a :

- Fixé précisément le format des données et leur encodage en machine (voir Table 2.1) ;
- Défini le comportement et la précision pour les opérateurs de base ;
- Défini les valeurs spéciales, les modes d'arrondi et la gestion des exceptions.

2.2.1 Représentation d'un Nombre à Virgule Flottante

Depuis son apparition, la norme IEEE754 est devenue le standard scientifique permettant de spécifier l'arithmétique flottante [ANS19, MBD+18]. Dans cette norme, chaque nombre flottant est représenté par un triplet (signe, exposant, mantisse) dans un format et une précision (simple, double, etc.), donnés dans la Table 2.1. La Définition 2.2.1 définit formellement un nombre flottant.

Définition 2.2.1. *Un nombre flottant $a \in \mathbb{F}$, en base 2, est défini par :*

$$a = (-1)^s \cdot M \cdot 2^e, \tag{2.5}$$

où s est le signe, M la mantisse et e l'exposant. Notons que $s \in \{0,1\}$, $e \in \{e_{min}, e_{max}\}$ et $M = d_0.d_1...d_i...d_{m-1}$, avec m la longueur de M (voir Table 2.1), $d_i \in \{0,1\}$ et $0 \leqslant i \leqslant m-1$. Les valeurs de e_{min}, e_{max} et m sont données dans la Table 2.1.

La Table 2.1 présente les différentes précisions et formats de représentation des nombres dans la norme IEEE754 [ANS19]. La première colonne donne le nombre total de bits sur lequel est représenté un nombre flottant et la deuxième colonne donne son appellation. La troisième, quatrième et cinquième colonne contiennent le nombre de bits sur lequel est

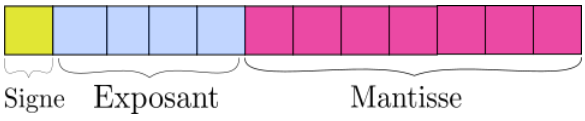

FIGURE 2.1: Représentation d'un Nombre à Virgule Flottante.

Format (bits)	Précision	Signe (s)	Mantisse (m)	Exposant (e)	e_{min}	e_{max}
16	Demie	1	10	5	-14	+15
32	Simple	1	23	8	-126	+127
64	Double	1	52	11	-1222	+1223
128	Quadruple	1	112	15	-16382	+16383

TABLE 2.1: Les Différents Formats de la Norme IEEE754.

représenté le signe, la mantisse et l'exposant, respectivement. Les deux dernières colonnes montrent les valeurs minimale et maximale que peut prendre l'exposant. Par exemple, un nombre flottant en simple précision est représenté sur 32 bits avec un signe sur un bit, une mantisse sur 23 bits et un exposant sur 8 bits, appartenant à la plage $[-126, 127]$.

La Figure 2.1 montre la représentation d'un nombre flottant composé d'un signe sur un bit, d'un exposant sur e bits et d'une mantisse représentée sur m bits. Les valeurs de m et de e peuvent être déterminées suivant le format et la précision, choisis. Elles correspondent respectivement à la quatrième et cinquième colonne de la Table 2.1.

Dans la norme IEEE754 [ANS19], les nombres flottants peuvent être représentés de manière normalisée ou dénormalisée. La normalisation des nombres flottants évite les représentations multiples du même nombre. Les nombres flottants dénormalisés [MBD+18] sont introduits pour éviter d'arrondir à zéro le résultat d'une opération flottante, qui serait plus petit que le plus petit nombre normalisé représentable dans le format utilisé. Cependant, dans ces travaux, nous nous intéressons aux nombres normalisés couvrants la plus grande partie de la plage de flottants disponibles en machine. Ces nombres ont une distribution non uniforme : la densité des flottants est plus grande vers zéro que vers les infinis [Gol91]. Un nombre normalisé (voir Définition 2.2.2) s'écrit avec son bit implicite d_0 à 1, et son exposant e prend une valeur entre zéro et sa valeur maximale (31, 255, 2047, 32767) suivant son format.

Définition 2.2.2. *Un nombre flottant $a \in \mathbb{F}$, en base 2, normalisé, est défini par :*

$$a = (-1)^s \times M \times 2^e, \tag{2.6}$$

où s est le signe, M la mantisse et e l'exposant. Notons que $s \in \{0, 1\}$, $M = 1.d_1...d_i...d_m$, avec m la longueur de M (voir Table 2.1), $d_i \in \{0, 1\}$, $1 \leqslant i \leqslant m$, et e prend une valeur entre zéro et sa valeur maximale (31, 255, 2047, 32767) suivant le format choisi (Table 2.1). Cette valeur maximale est obtenue en mettant tous les bits de l'exposant à 1.

Exemple 2.2.1. *Considérons le nombre flottant $a = 4.8125$ et montrons sa représentation machine en utilisant la norme IEEE754, la représentation normalisée et la simple précision (Table 2.1). Remarquons que a est positif, donc son bit de signe $s = 0$. Notons que, $4,8125 = (100.1101)_2$, en base 2. Ce nombre binaire peut également s'écrire : $(1.001101 \times 2^2)_2$ en décalant la virgule de deux positions vers la gauche. En simple précision, la mantisse est représentée sur 23 bits, l'exposant sur 8 bits et le signe sur 1 bit. La représentation en machine du flottant a est donnée par*

0 10000001 00110100000000000000000,

où 0 *correspond au signe,* 10000001 *est l'exposant et* 00110100000000000000000 *représente la mantisse. Ces valeurs sont obtenues via la Définition* 2.2.2.

2.2.2 Valeurs Spéciales et Exceptions

La norme IEEE754 [ANS19] définit des valeurs spéciales servant de représentations aux :

- Zéros : -0 et $+0$;
- Infinis : $-\infty$ et $+\infty$;
- Not-a-Number (`NaN`).

La Table 2.2 montre la représentation des valeurs spéciales existantes dans la norme IEEE754 [ANS19], en fonction du format des nombres flottants choisi dans la Table 2.1. Notons que, e_{max} (dans la Table 2.2) est obtenu en mettant tous les bits de l'exposant à 1 dans le format choisi (Table 2.1), et donc $e_{max} \in \{31, 255, 2047, 32767\}$. Un nombre flottant est égal à zéro quand son exposant et sa mantisse sont tous les deux égaux à zéro. À noter que, cette représentation du zéro peut avoir un signe positif, noté $+0$, ou un signe négatif, noté -0. La seule différence entre -0 et $+0$ est la propagation du signe à travers les autres opérations arithmétiques d'une expression.

L'infini, ou ∞, se caractérise par un exposant fixé à la valeur maximale et une mantisse à zéro. Un infini peut être positif, noté $+\infty$, ou négatif, noté $-\infty$. Ces valeurs sont obtenues lorsque le résultat d'une opération a un dépassement de capacité (*overflow*), c'est-à-dire que la valeur du résultat ne rentre pas et ne peut être représentée dans le format choisi.

Un `NaN`, est représenté avec un exposant à sa valeur maximale et une mantisse non nulle. Le signe n'est pas important pour cette valeur et peut être positif ou négatif. Un `NaN` est le plus souvent utilisé pour signaler le résultat d'une opération non définie sur les réels ($\sqrt{-4}$, $\frac{0}{0}$, $+\infty - \infty$, etc.).

En plus de ces valeurs spéciales, la norme IEEE754 impose de positionner certains drapeaux (*flags*) dans les cas suivants :

- Drapeau opération invalide (*invalid*) : si une opération donnant comme résultat `NaN` ;

Valeur Spéciale	Exposant (e)	Mantisse (m)
$+0$	$e = 0$	$m = 0$
-0	$e = 0$	$m = 0$
$\pm\infty$	$e = e_{max}$	$m = 0$
`NaN`	$e = e_{max}$	$m \neq 0$

TABLE 2.2: Valeurs Spéciales de la Norme IEEE754.

- Drapeau inexact (*inexact*) : lorsque le résultat d'une opération ne peut être exactement représenté, et il doit donc être arrondi ;
- Drapeau division par zéro ;
- Drapeau débordement vers l'infini (*overflow*) : lorsque le résultat d'une opération est trop grand par rapport à la borne supérieure de la plage des exposants représentables ;
- Drapeau débordement vers zéro (*underflow*) : lorsque le résultat d'une opération est trop petit par rapport à la borne inférieure de la plage des exposants représentables.

2.2.3 Opérations et Arrondis

Un nombre réel ne peut être exactement représenté en machine. Il doit donc être arrondi. La norme IEEE754 [ANS19] décrit quatre modes d'arrondi pour un nombre réel $a \in \mathbb{R}$:

- L'arrondi vers le haut ($+\infty$), noté ($\uparrow_{+\infty}(a)$), revient à arrondir vers le plus petit nombre flottant supérieur ou égal à a.
- L'arrondi vers le bas ($-\infty$), noté ($\uparrow_{-\infty}(a)$), revient à arrondir vers le plus grand nombre flottant inférieur ou égal à a.
- L'arrondi vers 0, ou troncature, noté ($\uparrow_0(a)$), est équivalent à un arrondi vers $-\infty$ (respectivement $+\infty$), si le nombre à arrondir est positif (respectivement négatif).
- L'arrondi au plus proche, noté ($\uparrow_\sim(a)$), revient à arrondir a vers le nombre flottant le plus proche.

Les arrondis vers $-\infty$, $+\infty$ et 0, sont appelés arrondis dirigés. Ces derniers arrondissent une valeur réelle $a \in \mathbb{R}$ vers un nombre flottant proche, à partir d'une direction donnée. Contrairement aux arrondis dirigés, l'arrondi au plus proche prend le nombre flottant à moindre distance de a, qu'il soit plus grand ou plus petit que a. Si la valeur à arrondir est équidistante des nombres flottants les plus proches, deux choix sont possibles : arrondir vers le nombre *pair* ou *loin de zéro*. Le premier va arrondir vers le nombre flottant avec une mantisse paire, tandis que l'arrondi *loin de zéro* va effectuer un arrondi vers le haut (respectivement vers le bas) dans le cas d'un nombre positif (respectivement négatif). Par défaut, les ordinateurs effectuent un arrondi au plus proche, avec égalité vers le nombre pair. Il est souvent noté comme un arrondi au plus proche pair.

L'implémentation d'une fonction en arithmétique flottante satisfait la propriété d'arrondi correct, si son résultat est égal à l'arrondi (suivant un mode d'arrondi donné) du résultat exact de la fonction. La Définition 2.2.3 définit formellement la sémantique des opérations élémentaires dans la norme IEEE754, pour les quatre modes d'arrondi.

Définition 2.2.3. *La sémantique des opérations élémentaires définie par la norme IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$, est donnée avec :*

$$\begin{array}{rccc} \uparrow_r : & \mathbb{R} & \longrightarrow & \mathbb{F} \\ & x \circledast_r y & = & \uparrow_r (x * y), \end{array} \tag{2.7}$$

où $\circledast_r \in \{+, -, \times, \div\}$ est l'une des quatre opérations élémentaires utilisées pour le calcul des nombres flottants, en utilisant le mode d'arrondi r et, $* \in \{+, -, \times, \div\}$ est l'opération exacte (opérations sur les réels).

Remarque 2. *Dans la norme IEEE754, la racine carrée ($\sqrt{}$) a un arrondi correct aussi.*

Remarque 3. *D'autres opérations mathématiques, comme les fonctions transcendantes (`sin`, `cos`, `tan`, `log`, etc.), n'ont pas obligation d'être arrondies correctement d'après la norme IEEE754 [ANS19]. Cet arrondi est laissé à l'appréciation de l'implémentation de l'arithmétique des nombres flottants et peut donc varier grandement d'un système à l'autre.*

L'arrondi correct garantit que l'addition et la multiplication dans l'arithmétique flottante sont commutatives, par contre l'associativité et la distributivité ne sont pas conservées [Gar21]. Pour plus de détails sur l'arithmétique flottante et la norme IEEE754, j'invite le lecteur à se référer au "Handbook of Floating-Point Arithmetic" de Muller et al. [MBD+18].

2.3 Arithmétique à Virgule Fixe

De nos jours, l'utilisation des systèmes embarqués ne cesse de croître malgré leurs ressources limitées. Ces systèmes ont des petites mémoires et souvent des CPUs compatibles avec le calcul entier uniquement, c'est pourquoi, l'utilisation de l'arithmétique à virgule fixe (appelée arithmétique fixe par la suite) reposant sur des nombres et opérations entières, est efficace. Cependant, cette arithmétique [MNR17, BS17, Yat09, MCS06, Gal21] est plus difficile à manier et à gérer pour les développeurs. Ces derniers doivent gérer la position de la virgule manuellement, mais aussi les débordements (*overflow* et *underflow*) et les exceptions (division par zéro, racine négative, etc.), contrairement à la norme IEEE754 [ANS19] qui gère tout de manière automatique. Pour éviter que les calculs soient erronés, une attention doit être accordée à la présence des retenues, aux nombres et aux sens des décalages effectués, aux longueurs des parties entières en sortie des calculs, etc. Dans cette section, nous montrons comment convertir un nombre flottant en un nombre à virgule fixe et présentons les opérations arithmétiques, les modes d'arrondis et nous citons quelques librairies et algorithmes à virgule fixe.

2.3.1 Représentation d'un Nombre à Virgule Fixe

Dans l'arithmétique fixe et selon la littérature [MNR17, BS17, MCS06, Lop14, Yat09], un nombre à virgule fixe est représenté par un **signe** $s \in \{0, 1\}$, une **valeur entière** $V \in \mathbb{N}$

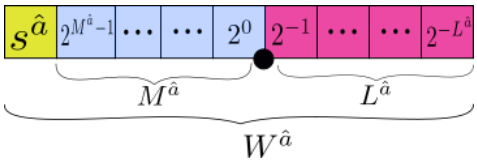

FIGURE 2.2: Représentation du Nombre à Virgule Fixe $\hat{a}$ dans un Format $< M^{\hat{a}}, L^{\hat{a}} >$.

et un **format** $< M, L >$, où $M \in \mathbb{Z}$ représente le nombre de bits avant la virgule fixe et $L \in \mathbb{N}$ représente le nombre de bits après la virgule fixe. En d'autres termes, un nombre à virgule fixe est un nombre entier relatif ($\in \mathbb{Z}$) multiplié par un facteur d'échelle 2^{-L}. Dans ce manuscrit, nous adoptons l'écriture et la représentation des nombres à virgule fixe, appelés nombres fixes par la suite, données dans les définitions 2.3.1 et 2.3.2, respectivement.

Définition 2.3.1. *Soit $\aleph$ l'ensemble des nombres fixes et, soit le nombre fixe $\hat{a} \in \aleph$ représenté sur $W^{\hat{a}}$ bits. Ce nombre $\hat{a}$ s'écrit sous la forme*

$$\hat{a} = (-1)^{s^{\hat{a}}} \times V^{\hat{a}}_{<M^{\hat{a}}, L^{\hat{a}}>}. \tag{2.8}$$

Son signe est $s^{\hat{a}} \in \{0, 1\}$, sa valeur entière $V^{\hat{a}} \in \mathbb{N}$ et son format $< M^{\hat{a}}, L^{\hat{a}} >$, tels que $M^{\hat{a}} \in \mathbb{Z}$ et $L^{\hat{a}} \in \mathbb{N}$. Ces deux derniers représentent respectivement le nombre de bits de la partie entière et fractionnaire. Ils sont calculés par

$$M^{\hat{a}} = \text{ufp}(a) + 1. \tag{2.9}$$

Et,

$$L^{\hat{a}} = W^{\hat{a}} - M^{\hat{a}} - 1. \tag{2.10}$$

La valeur entière $V^{\hat{a}}$ correspondante à $\hat{a}$ est obtenue via la formule suivante

$$(-1)^{s^{\hat{a}}} \times V^{\hat{a}} = \lfloor a \times 2^{L^{\hat{a}}} \rceil. \tag{2.11}$$

Notons que, a est la valeur flottante correspondante à $\hat{a}$ et ufp(a), utilisé dans l'Équation (2.9), représente le bit de poids fort de a et est défini dans la Définition 2.1.1. Dans l'Équation (2.11), $\lfloor . \rceil$ correspond à la partie entière, inférieure ou supérieure au plus proche.

La Figure 2.2 montre la représentation d'un nombre fixe $\hat{a}$ de longueur $W^{\hat{a}}$ bits. Son signe $s^{\hat{a}}$ est représenté sur un bit. Sa partie entière est représentée sur $M^{\hat{a}}$ bits et sa partie fractionnaire sur $L^{\hat{a}}$ bits.

Définition 2.3.2. *Dans une base de représentation $\beta = 2$, considérons un nombre fixe $\hat{a} \in \aleph$ et sa représentation flottante $a \in \mathbb{F}$. La valeur flottante correspondante au nombre fixe $\hat{a}$ est donnée avec*

$$a = (-1)^{s^{\hat{a}}} \times V^{\hat{a}} \times 2^{-L^{\hat{a}}}. \tag{2.12}$$

Dans la représentation à virgule fixe, nous remarquons que la gestion de la position de la virgule se fait manuellement par le développeur, ce qui la rend difficile à maintenir, contrairement à la virgule flottante [ANS19] qui le fait de manière automatique grâce à l'exposant.

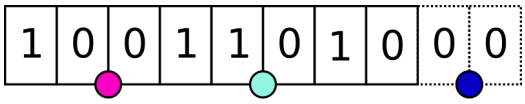

FIGURE 2.3: Nombre à Virgule Fixe avec Différents Facteurs d'Échelle.

Couleur Fact. D'Échelle	Format	Valeur Entière de $\hat{a}$	Valeur Flottante de $\hat{a}$
Magenta	$< 2, 6 >$	$(10011010)_2 = (154)_{10}$	$(10.011010)_2 = (\mathbf{2.40625})_{\mathbf{10}}$
Cyan	$< 5, 3 >$	$(10011010)_2 = (154)_{10}$	$(10011.010)_2 = (\mathbf{19.25})_{\mathbf{10}}$
Bleue	$< 9, 1 >$	$(10011010)_2 = (154)_{10}$	$(100110100.0)_2 = (\mathbf{308})_{\mathbf{10}}$

TABLE 2.3: Représentation du Nombre Fixe $\hat{a}$ dans Différents Formats.

Remarque 4. *Dans la littérature, il existe une autre notation pour les formats virgule fixe : la notation $Q_{M,L}$ [PKR13, Yat09]. Cette notation contient l'information sur le nombre de bits de la partie entière M (à gauche) et le nombre de bits de la partie fractionnaire L (à droite).*

Exemple 2.3.1. *Considérons la suite binaire représentée dans la Figure 2.3 correspondante à la valeur entière $V^{\hat{a}} = 154$ du nombre fixe $\hat{a} \in \aleph$. Dans la Table 2.3, nous considérons plusieurs formats différents pour la valeur entière $V^{\hat{a}}$, c'est-à-dire que nous la multiplions par plusieurs facteurs d'échelle, afin de montrer l'importance de la position de la virgule et de la bonne gestion des formats. La première colonne de la Table 2.3 correspond à la couleur associée à la position de la virgule (facteur d'échelle) dans la Figure 2.3. La deuxième colonne correspond au format attribué à $\hat{a}$, en d'autres termes, le nombre de bits avant et après la virgule fixe. La troisième colonne représente $V^{\hat{a}}$ la valeur entière de $\hat{a}$ (dans cet exemple $V^{\hat{a}}$ est égale à 154) en base 2 $(.)_2$ et en base 10 $(.)_{10}$. La dernière colonne montre la valeur flottante $a \in \mathbb{F}$ correspondante à $\hat{a}$. Cette dernière est obtenue par l'Équation (2.12) en multipliant la valeur entière $V^{\hat{a}}$ par les différents facteurs d'échelle. Si nous considérons la première ligne correspondante au format $< 2, 6 >$ et la couleur Magenta dans la Figure 2.3, nous obtiendrons la valeur flottante 2.40625 en multipliant 154×2^{-6}. La valeur flottante a change à chaque fois que le facteur d'échelle change même si la valeur entière $V^{\hat{a}}$ est la même.*

2.3.2 Opérations Arithmétiques

Dans l'arithmétique fixe, nous devons implémenter nos propres opérations arithmétiques : élémentaires (addition, soustraction, multiplication et division) et mathématiques (valeur absolue, racine carrée, fonctions trigonométriques). Nous devons aussi prêter attention à la bonne gestion (manuelle) des formats, des décalages et des exceptions du type *overflow*, contrairement à l'arithmétique flottante qui le fait de manière automatique. Dans ces travaux, nous nous intéressons uniquement à l'addition, la soustraction et à la multiplication, qui sont requises pour effectuer des calculs dans les réseaux de neurones (Chapitre 3). Nous allons à chaque fois considérer les nombres fixes $\hat{a}$, $\hat{b}$, $\hat{c}$, $\hat{r} \in \aleph$ et effectuer une opération virgule fixe élémentaire $\odot \in \{\oplus, \ominus, \otimes\}$, tel que $\hat{r} = \hat{a} \odot \hat{b}$, est le résultat de l'opération $\odot$,

et $\hat{c}$ contient le format requis par l'utilisateur pour cette opération.

Pour chaque opération, il existe trois manières de calculer les résultats [Lop14]. Elles reposent sur le choix et le calcul des formats en sortie : longueur optimale, longueur fixée et format fixé. Lors d'un calcul avec une longueur optimale, aucune troncature (voir Sous-Section 2.3.3) sur les opérandes n'est effectuée, c'est-à-dire que le résultat obtenu ne contient pas une erreur liée à la troncature. Dans le cas de la longueur fixée, l'utilisateur fixe le nombre total de bits du résultat. Le dernier cas est le format fixé, où l'utilisateur fixe le nombre de bits des deux parties : entière et fractionnaire du résultat. On s'intéressera et on utilisera le format fixé dans ces travaux. Nous avons cité précédemment que nous nous intéressons à l'addition, la soustraction et à la multiplication. Ces trois opérations effectuent des décalages. Nous introduisons ces derniers dans les définitions 2.3.3 et 2.3.4

Décalages

Soient $\ll$ et $\gg$, les opérateurs pour désigner les décalages binaires à gauche et à droite, respectivement. Décaler un nombre entier à gauche (respectivement à droite) de k bits revient à le multiplier par 2^k (respectivement effectuer une division entière par 2^k). Les définitions 2.3.3 et 2.3.4 définissent les décalages binaires à gauche et à droite, respectivement.

Définition 2.3.3. *Un décalage à gauche de k bits appliqué au nombre fixe $\hat{a} \in \aleph$, est défini par*

$$\begin{cases} V^{\hat{a}} = V^{\hat{a}} \ll k, \\ M^{\hat{a}} = M^{\hat{a}}, \\ L^{\hat{a}} = L^{\hat{a}} + k. \end{cases} \tag{2.13}$$

Définition 2.3.4. *Un décalage à droite de k bits appliqué au nombre fixe $\hat{a} \in \aleph$, est défini par*

$$\begin{cases} V^{\hat{a}} = V^{\hat{a}} \gg k, \\ M^{\hat{a}} = M^{\hat{a}}, \\ L^{\hat{a}} = L^{\hat{a}} - k. \end{cases} \tag{2.14}$$

Addition et Soustraction

Soit $\oplus$, l'opérateur pour désigner l'addition à virgule fixe et soit $\hat{r}$ le résultat de cette addition, tel que $\hat{r} = \hat{a} \oplus \hat{b}$. Le nombre de bits de la partie entière et fractionnaire de $\hat{r}$ sont donnés par la Définition 2.3.5.

Définition 2.3.5. *Le nombre de bits de la partie entière résultant d'une addition à virgule fixe sur $W^{\hat{r}}$ bits, est donné avec*

$$M^{\hat{r}} = max(M^{\hat{a}}, M^{\hat{b}}) + 1. \tag{2.15}$$

Et son nombre de bits de la partie fractionnaire, vaut

$$L^{\hat{r}} = W^{\hat{r}} - M^{\hat{r}} - 1. \tag{2.16}$$

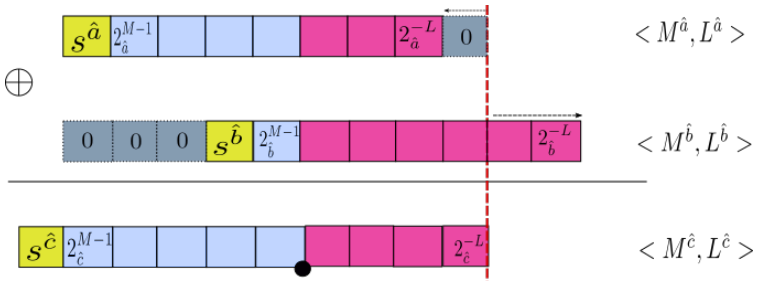

FIGURE 2.4: Addition de Deux Nombres à Virgule Fixe dans un Format Fixé $< M^{\hat{c}}, L^{\hat{c}} >$.

Dans ces travaux, nous nous intéressons à l'addition avec un format de sortie fixé, c'est-à-dire que le nombre de bits de la partie entière et fractionnaire sont fixés par l'utilisateur. Notons ce format requis $< M^{\hat{c}}, L^{\hat{c}} >$. Dans ce cas, il faut donc vérifier si le résultat de l'addition rentre bien dans le format donné $< M^{\hat{c}}, L^{\hat{c}} >$. Pour cela, il faut s'assurer que, le nombre de bits de la partie entière $M^{\hat{c}}$ requis, est supérieur ou égal à la valeur de $M^{\hat{r}}$, retournée par l'Équation (2.15).

La Figure 2.4 montre l'addition de deux nombres fixes $\hat{a}$ et $\hat{b}$ dans un format de sortie fixé $< M^{\hat{c}}, L^{\hat{c}} >$. Dans cette figure, $2^{M-1}_{\hat{a}}$ représente le bit de poids fort et $2^{-L}_{\hat{a}}$ représente le bit de poids faible de $\hat{a}$ (de même pour $\hat{b}$). Pour effectuer l'addition de $\hat{a}$ et $\hat{b}$ et obtenir un résultat dans le format requis $< M^{\hat{c}}, L^{\hat{c}} >$, nous devons aligner les longueurs des deux parties fractionnaires $L^{\hat{a}}$ et $L^{\hat{b}}$ avec $L^{\hat{c}}$ via des décalages. Notons que, les opérateurs $\ll$ et $\gg$, correspondent respectivement aux décalages binaires à gauche et à droite et sont respectivement définis dans les définitions 2.3.3 et 2.3.4. Le résultat de l'alignement de $\hat{a}$ (idem pour $\hat{b}$) avec le format requis $< M^{\hat{c}}, L^{\hat{c}} >$, est donné dans la Définition 2.3.6.

Définition 2.3.6. *Le résultat de l'alignement de $\hat{a}$ dans un format $< M^{\hat{a}}, L^{\hat{a}} >$ avec le format $< M^{\hat{c}}, L^{\hat{c}} >$ (nous supposons qu'il n'y a pas d'overflow : $M^{\hat{a}} \leqslant M^{\hat{c}}$), est défini par*

$$\forall s^{\hat{a}} \in \{0,1\}, \quad ((-1)^{s^{\hat{a}}} \times V^{\hat{a}}) = \begin{cases} ((-1)^{s^{\hat{a}}} \times V^{\hat{a}}) \ll (L^{\hat{c}} - L^{\hat{a}}) & si\ (L^{\hat{c}} > L^{\hat{a}}), \\ ((-1)^{s^{\hat{a}}} \times V^{\hat{a}}) \gg (L^{\hat{a}} - L^{\hat{c}}) & sinon. \end{cases} \tag{2.17}$$

et,

$$< M^{\hat{a}}, L^{\hat{a}} > = < M^{\hat{c}}, L^{\hat{c}} > . \tag{2.18}$$

Exemple 2.3.2. *Dans cet exemple et dans la Figure 2.5, nous illustrons l'addition à virgule fixe de $\hat{a} = 33_{<2,4>}$ et $\hat{b} = 33_{<4,2>}$. Nous fixons notre format de sortie à $< M^{\hat{c}}, L^{\hat{c}} > = < 6,3 >$, par exemple. D'après l'Équation (2.15), le résultat de cette addition peut tenir sur le format requis $< 6,3 >$ car $6 \geqslant 4$. Écrivons les valeurs de $\hat{a}$ et $\hat{b}$ en binaire, tels que :*

$$\hat{a} = 10.0001 \text{ et } \hat{b} = 1000.01.$$

Nous constatons que $s^{\hat{a}} = s^{\hat{b}} = 0$ et donc $s^{\hat{c}} = 0$. Pour calculer cette addition, il faut aligner $\hat{a}$ et $\hat{b}$ avec le format $< M^{\hat{c}}, L^{\hat{c}} >$, en appliquant la Définition 2.3.6. La Figure 2.5 montre les décalages

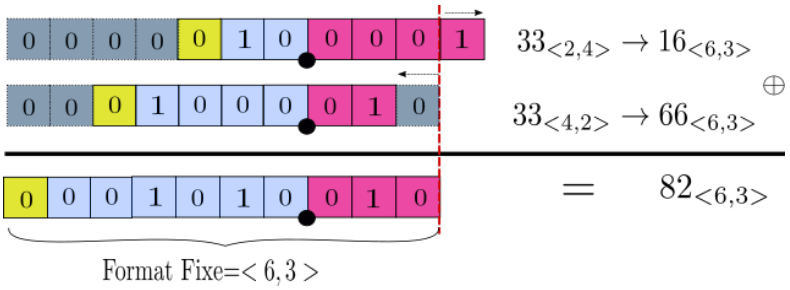

FIGURE 2.5: Addition Virgule Fixe avec Format Fixé $< 6, 3 >$.

effectués pour aligner $\hat{a}$ et $\hat{b}$. Un décalage d'un bit vers la droite est nécessaire pour $\hat{a}$, car le nombre de bits après la virgule requis en sortie est 3, et $\hat{a}$ en a 4. Un décalage d'un bit vers la gauche est effectué pour $\hat{b}$, car le nombre de bits après la virgule requis en sortie est 3, et $\hat{b}$ en a 2, cela revient à rajouter un 0 à droite. Après l'alignement des opérandes, nous obtenons $V^{\hat{a}} = 33 \gg 1 = 16$ et $V^{\hat{b}} = 33 \ll 1 = 66$. Maintenant, il suffit juste de sommer ces valeurs obtenues. Finalement, le résultat de cette addition dans le format fixé $< 6, 3 >$ obtenu est $\hat{c} = 82_{<6,3>}$.

Remarque 5. *Dans cet exemple, le format $< 4, 3 >$ aurait été suffisant pour cette addition, car en consacrant 6 bits pour la partie entière du résultat, nous avons deux bits nuls à gauche (voir Figure 2.5) qui ne sont pas nécessaires.*

Remarque 6. *La soustraction à virgule fixe se fait exactement de la même manière que l'addition à virgule fixe, définie ci-dessus. Il suffit de remplacer l'opérateur $\oplus$ par l'opérateur $\ominus$.*

Multiplication

Soit $\otimes$, l'opérateur pour désigner la multiplication à virgule fixe et soit $\hat{r}$ le résultat de cette multiplication, tel que $\hat{r} = \hat{a} \otimes \hat{b}$. Le nombre de bits de la partie entière et fractionnaire de $\hat{r}$ sont définis dans la Définition 2.3.7.

Définition 2.3.7. *Le nombre de bits de la partie entière d'une multiplication à virgule fixe sur $W^{\hat{r}}$ bits, est donné avec*

$$M^{\hat{r}} = M^{\hat{a}} + M^{\hat{b}} + 1. \tag{2.19}$$

Et son nombre de bits de la partie fractionnaire, vaut

$$L^{\hat{r}} = W^{\hat{r}} - M^{\hat{r}} - 1. \tag{2.20}$$

Supposons que le format fixé du résultat de la multiplication virgule fixe requis est $< M^{\hat{c}}, L^{\hat{c}} >$. La Figure 2.6 illustre la multiplication à virgule fixe dans le format requis $< M^{\hat{c}}, L^{\hat{c}} >$. Dans cette figure, $2^{M-1}_{\hat{a}}$ représente le bit de poids fort et $2^{-L}_{\hat{a}}$ le bit de poids faible de $\hat{a}$ (idem $\hat{b}$). Le résultat de cette multiplication est obtenu en calculant la somme binaire des produits binaires intermédiaires. Par exemple, la première multiplication intermédiaire (1) de la Figure 2.6 consiste à multiplier tous les bits de $\hat{a}$ par le premier bit de $\hat{b}$, en partant de la droite. Ensuite, on calcule le résultat de la multiplication (2) de tous les

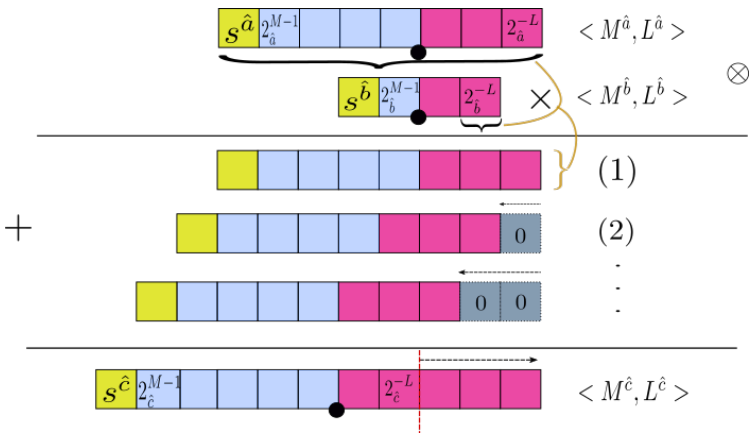

FIGURE 2.6: Multiplication de Deux Nombres à Virgule Fixe dans un Format Fixé $< M^{\hat{c}}, L^{\hat{c}} >$.

bits de $\hat{a}$ par le deuxième bit de $\hat{b}$, à partir de la droite et, on décale vers la gauche ce résultat d'une position (on effectue la même opération jusqu'au dernier bit de $\hat{b}$). Pour éviter les débordements (*overflow*), le nombre de bits de la partie entière $M^{\hat{c}}$ doit être, supérieur ou égal à, $M^{\hat{r}}$ défini dans l'Équation (2.19). Les algorithmes de la multiplication et de l'addition à virgule fixe sont donnés dans [MNR17, Lop14, Naj14].

2.3.3 Valeurs Spéciales et Arrondis

Dans chaque processus de conversion ou dans chaque calcul en précision finie, il y a une étape d'arrondi. Il existe plusieurs façons d'arrondir une valeur, on parle alors de modes d'arrondi. Les modes d'arrondi définis en Sous-Section 2.2.3 : $\uparrow_{+\infty}(.)$, $\uparrow_{-\infty}(.)$, $\uparrow_{0}(.)$ et $\uparrow_{\sim}(.)$, s'appliquent aussi à la virgule fixe. Cependant, nous nous intéressons principalement aux deux modes d'arrondi suivants : la troncature et l'arrondi au plus proche. Il faut noter que, la troncature correspond à l'arrondi vers le bas $\uparrow_{-\infty}(.)$, où on se contente d'effacer les bits au-delà de celui auquel on veut arrondir. En virgule flottante, c'est l'arrondi vers zéro qui s'obtient par troncature. La Définition 2.3.8 définit l'arrondi virgule fixe d'un nombre réel $a \in \mathbb{R}$ par troncature, noté $\uparrow_{-\infty}(a)$. Cet arrondi est le plus grand nombre fixe dont le nombre de bits de la partie fractionnaire est $L^{\hat{a}}$ et, qui est plus petit que a.

Définition 2.3.8. *L'arrondi virgule fixe de $a \in \mathbb{R}$ par troncature, s'obtient en calculant :*

$$\uparrow_{-\infty}(a) = \lfloor a \times 2^{L^{\hat{a}}} \rfloor \times 2^{-L^{\hat{a}}}, \tag{2.21}$$

où $\lfloor . \rfloor$ représente la partie entière inférieure.

Exemple 2.3.3. *Considérons la conversion en virgule fixe du nombre réel $a = \pi$ sur $W^{\hat{a}} = 8$ bits. D'après les équations (2.9) et (2.10), le format $< M^{\hat{a}}, L^{\hat{a}} >=< 2, 5 >$. Suivant la Définition 2.3.8, l'arrondi virgule fixe de a par troncature vaut*

$$\uparrow_{-\infty}(a) = \lfloor \pi \times 2^5 \rfloor \times 2^{-5} = 100 \times 2^{-5} = 3.125.$$

L'arrondi virgule fixe au plus proche du nombre réel $a \in \mathbb{R}$, noté $\uparrow_{\sim}(a)$, est le nombre fixe le plus proche de a dont le nombre de bits de la partie fractionnaire est $L^{\hat{a}}$. Cet arrondi est donné par la Définition 2.3.9.

Définition 2.3.9. *L'arrondi virgule fixe de $a \in \mathbb{R}$ au plus proche, s'obtient en calculant :*

$$\uparrow_{\sim}(a) = \lfloor a \times 2^{L^{\hat{a}}} \rceil \times 2^{-L^{\hat{a}}}, \tag{2.22}$$

où $\lfloor . \rceil$ représente la partie entière inférieure ou supérieure au plus proche.

Exemple 2.3.4. *Considérons la conversion en virgule fixe du nombre réel $a = \pi$ sur $W^{\hat{a}} = 8$ bits. D'après les équations (2.9) et (2.10), le format $< M^{\hat{a}}, L^{\hat{a}} >=< 2, 5 >$. Suivant la Définition 2.3.9, l'arrondi virgule fixe de a au plus proche vaut*

$$\uparrow_{\sim}(a) = \lfloor \pi \times 2^5 \rceil \times 2^{-5} = 101 \times 2^{-5} = 3.15625.$$

N.B. *Dans ces travaux, nous utilisons l'arrondi par troncature dans les opérations arithmétiques et l'arrondi au plus proche pour calculer la valeur entière d'un nombre fixe (conversion).*

Maintenant, nous souhaitons quantifier l'erreur commise entre le nombre réel a et son arrondi. Nous pouvons quantifier cette erreur en calculant les erreurs, absolue δ_{abs} et relative δ_{rel}, respectivement définies dans les définitions 2.1.3 et 2.3.3. Le Lemme 2.3.1 donne l'erreur absolue entre le réel a et son arrondi par troncature $\uparrow_{-\infty}(a)$, et le Lemme 2.3.2 présente l'erreur absolue entre le réel a et son arrondi au plus proche $\uparrow_{\sim}(a)$.

Lemme 2.3.1. *L'erreur absolue entre $a \in \mathbb{R}$ et son arrondi par troncature $\uparrow_{-\infty}(a)$, est*

$$\delta_{abs} < 2^{-L^{\hat{a}}}. \tag{2.23}$$

Lemme 2.3.2. *L'erreur absolue entre $a \in \mathbb{R}$ et son arrondi au plus proche $\uparrow_{\sim}(a)$, est*

$$\delta_{abs} \leqslant 2^{-L^{\hat{a}}-1}. \tag{2.24}$$

L'erreur relative correspondante à l'erreur effectuée par rapport à la valeur réelle de départ a telle que $a \neq 0$, est donnée par le Lemme 2.3.3.

Lemme 2.3.3. *L'erreur relative entre $a \in \mathbb{R}^*$ et son arrondi, est*

$$\delta_{rel} < 2^{-W^{\hat{a}}-1}. \tag{2.25}$$

N.B. *Les preuves des lemmes 2.3.1 et 2.3.2 sont données et détaillées dans [Gal21].*

Contrairement à l'arithmétique flottante, il n'existe pas une définition des infinis, du zéro négatif et des `NaN` dans l'arithmétique fixe. Les cinq drapeaux (*invalid, inexact, zero, overflow et underflow*) ne sont pas levés de manière automatique, comme c'est le cas dans l'arithmétique flottante. Le développeur doit gérer manuellement toutes les exceptions telles que les divisions par zéro, les nombres négatifs sous la racine carrée, etc.

2.3.4 Quelques Algorithmes et Librairies Virgule Fixe

Il existe plusieurs algorithmes et librairies pour l'arithmétique fixe utilisant différentes techniques pour, convertir un nombre de l'arithmétique flottante vers l'arithmétique fixe, et implémenter les algorithmes effectuant les opérations arithmétiques élémentaires (addition, soustraction, multiplication, division) et mathématiques (fonctions trigonométriques, racine carrée, valeur absolue, etc.) Ces opérations sont utilisées dans différents programmes de l'électronique embarquée tels que les filtres linéaires [Vol17, Lop14], traitement du signal et de l'image, etc. Dans la littérature, nous pouvons trouver plusieurs librairies à virgule fixe. Nous citons quelques-unes ci-dessous :

- `Libfixmath`[1] [Cod11a] est une librairie où les nombres fixes ayant pour format $< 16, 16 >$ sont implémentés en langage *C* [KR88]. Cette librairie contient principalement la fonction de conversion des nombres flottants en virgule fixe, les fonctions effectuant les opérations élémentaires et les opérations mathématiques (racine carrée, exponentielle, logarithme, valeur absolue, fonctions trigonométriques et leurs inverses). Cependant, les limites de `Libfixmath` sont, d'une part, la prise en charge des nombres fixes dans le format $< 16, 16 >$ uniquement et d'autre part, elle prend en considération l'*overflow* seulement pour les opérations élémentaires et mathématiques, mais pas pour l'interpolation linéaire et la fonction logarithmique. La librairie `Libfixmatrix` [2] [Cod11b] existe aussi et est basée sur `Libfixmath`. Elle permet d'effectuer des calculs en virgule fixe dans un format $< 16, 16 >$ sur des matrices.

- `Fixmath`[3] [NNA12] est développée en langage *C* [KR88]. Elle utilise des entiers sur 32 bits pour représenter les nombres fixes. Elle contient plusieurs fonctions pour effectuer les opérations élémentaires, les fonctions algébriques (`sqrt` et son inverse) et les fonctions transcendantes (`exp, log, log, pow, sin, cos`). `Fixmath` ne vérifie malheureusement pas les exceptions telles que les divisions par zéro et les débordements.

- `FPM`[4] [Lan19] est implémentée en langage *C++* [Str07]. Elle contient une variété de fonctions telles que les fonctions élémentaires, les fonctions basiques (`abs`, `fmod`, `remainder`, etc.), les fonctions trigonométriques (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`), les fonctions exponentielle et logarithmique, et les fonctions avec les puissances (`pow`, `sqrt`, etc.). Cette librairie est riche, mais a ses limites. Elle ne gère pas les problèmes d'*overflow* et d'*underflow* dans certains cas de figure, comme la fonction `pow` par exemple. `FPM` n'assure pas la précision (*accuracy*) pour les fonctions trigonométriques.

- `Compositional Numeric Library (CNL)`[5] [MBB15] est une librairie de classes numé-

[1] https://github.com/PetteriAimonen/libfixmath
[2] https://github.com/PetteriAimonen/libfixmatrix
[3] http://savannah.nongnu.org/projects/fixmath/
[4] https://github.com/MikeLankamp/fpm
[5] https://github.com/johnmcfarlane/cnl

riques en C++ [Str07] à précision fixe, offrant plusieurs types pour les entiers tels que, `safe integer`, `scaled integer`, `overflow integer`, etc. Elle contient les fonctions élémentaires ainsi que les fonctions racine carrée, puissance et valeur absolue. `CNL` assure l'absence des *overflows* dans les fonctions qu'elle implémente.

- **`C++11 Fixed Point Arithmetic Library`**[6] [Jon17] est une librairie implémentée en C++ [Str07]. Elle contient les fonctions élémentaires. Elle assure le non-débordement (absence des *overflows*) des résultats des calculs pour les opérations élémentaires seulement. Cette librairie est maintenant incluse dans la librairie `CNL` de John McFarlane.

Nous pouvons également citer **`CORDIC`** [Mul06, Vol59], qui est un ensemble d'algorithmes de calcul de fonctions élémentaires, trigonométriques, hyperboliques, exponentielle, etc. `CORDIC` utilise principalement l'addition et les décalages pour calculer ses fonctions. Les multiplications sont évitées, car elles sont plus coûteuses que les décalages. Les algorithmes `CORDIC` reposent sur des méthodes itératives, chiffre par chiffre (estimation linéaire).

2.4 Comparaison Entre les Arithmétiques : Flottante et Fixe

Le choix de l'arithmétique adéquate pour la résolution d'un problème donné est important. Nous devons adopter celle qui répond le mieux aux besoins (domaine d'application, matériel disponible, énergie consommée, etc.) Pour cela, nous devons déterminer les propriétés d'une bonne arithmétique. Une comparaison entre les propriétés citées ci-dessous aide à déterminer (choisir) l'arithmétique répondant aux besoins :

- Rapidité des calculs ;
- Précision des calculs ;
- Dynamique des valeurs représentées ;
- Portabilité des calculs ;
- Facilité d'utilisation et d'implémentation ;
- Énergie consommée ;
- Prix du CPU.

La Table 2.4 présente une comparaison entre l'arithmétique flottante et l'arithmétique fixe en matière de présence de norme, d'encodage, d'exposant, de distance entre deux nombres qui se suivent, de valeurs spéciales et de gestion d'exceptions. Les propriétés

[6] https://embeddedartistry.com/blog/2017/08/25/c11-fixed-point-arithmetic-library/

N° Ligne	Arith. / Propriété	Arith. Flottante	Arith. Fixe
1	**Norme**	+	-
2	**Encodage**	Signe, Exposant, Mantisse	Signe, Entier
3	**Exposant**	Explicite	Implicite
4	**Valeurs Spéciales**	+	-
5	**Gestion des Exceptions**	+	-
6	**Distance entre Deux Nombres Consécutifs**	Non-Équidistance	Équidistance
7	**Rapidité des Calculs**	-	+
8	**Précision des Calculs**	+	-
9	**Dynamique des Valeurs**	+	-
10	**Facilité d'Utilisation et d'Implémentation**	+	-
11	**Portabilité des Calculs**	+	+
12	**Énergie Consommée**	-	+
13	**Prix CPU**	-	+

TABLE 2.4: Comparaison entre l'Arithmétique Flottante et l'Arithmétique Fixe.

d'une bonne arithmétique (précision et rapidité de calculs, dynamique des valeurs représentées, énergie consommée, etc.), citées précédemment, sont également considérées dans ce tableau comparatif. La première ligne de la Table 2.4 montre l'existence d'une norme dans l'arithmétique flottante intitulée IEEE754 [ANS19], contrairement à l'arithmétique fixe qui n'en a pas, c'est pourquoi de nos jours, l'arithmétique flottante et en particulier la norme IEEE754 [ANS08], reste la plus utilisée. Dans l'arithmétique flottante, les nombres sont représentés par un signe, un exposant et une mantisse comme le montre la ligne 2 de la Table 2.4, tandis que la représentation d'un nombre fixe contient une valeur entière accompagnée d'un signe. L'arithmétique fixe nécessite la gestion de la position de la virgule manuellement par le développeur, à cause de son facteur d'échelle implicite (ligne 3 de la Table 2.4), contrairement à l'arithmétique flottante qui le fait de manière automatique, grâce à son facteur d'échelle explicite (exposant). Les lignes 4 et 5 de la Table 2.4 montrent que les valeurs spéciales (`NaN`, , $+0$, -0, $-\infty$, $+\infty$) et les exceptions (division par zéro, opération invalide, etc.) sont gérés automatiquement par la norme IEEE754 [ANS19], contrairement à l'arithmétique fixe où, le développeur doit gérer toutes les exceptions par lui-même. L'écart entre deux nombres représentables consécutifs (la différence entre deux nombres identiques, partout sauf sur le dernier bit de la mantisse) est équidistant dans l'arithmétique

fixe et, est égal à 2^{-L} (ligne 6 de la Table 2.4) pour un format fixe $< M, L >$. Contrairement à l'arithmétique fixe, il n'y a pas d'équidistance entre deux nombres consécutifs dans l'arithmétique flottante. Cette distance varie en fonction de l'exposant et des puissances de 2 positives ou négatives. Elle vaut soit un ulp ou un demi ulp [MBD+18, Gol91]. La ligne 7 de la Table 2.4 montre que l'arithmétique fixe est plus intéressante à utiliser en matière de rapidité de calculs, grâce aux opérations entières effectuées, à l'opposé de l'arithmétique flottante nécessitant un FPU (*Floating-Point Unit*) pour effectuer les calculs. La précision et la dynamique des calculs (ligne 8 et 9 de la Table 2.4) restent de qualité supérieure dans l'arithmétique flottante par rapport à l'arithmétique fixe [Naj14, Mé09]. Cette dernière a une dynamique limitée nécessitant le recadrage des données pour éviter les débordements. Notons que, plus la dynamique est importante, plus le débordement est faible. La dynamique augmente exponentiellement pour les flottants et linéairement dans le cas du fixe [Mén11, Cha14]. L'arithmétique fixe est plus difficile à implémenter et à utiliser (ligne 10 de la Table 2.4), car le développeur doit manier manuellement la position de la virgule ainsi que, les alignements des formats lors des calculs, dans son implémentation. L'utilisation de l'arithmétique fixe nécessite une connaissance du fonctionnement de cette dernière par l'utilisateur, contrairement à l'arithmétique flottante où, tout est géré de manière automatique. La ligne 11 de la Table 2.4 montre que les deux arithmétiques sont favorables à la portabilité. Avant l'apparition de la norme IEEE754, l'arithmétique fixe était plus favorable à la portabilité, car elle ne dépend pas d'un FPU, contrairement à l'arithmétique flottante qui, donnait des résultats différents sur différentes machines et architectures. L'arithmétique fixe est connue pour sa faible consommation d'énergie [Bar17, BS17], grâce à ses opérations entières tandis que, l'arithmétique flottante est plus gourmande en énergie, à cause de ses opérations complexes. À travers la ligne 12 de la Table 2.4, nous pouvons déduire qu'il serait plus intéressant d'utiliser l'arithmétique fixe, si nous souhaitons une faible consommation d'énergie. Le prix des processeurs est moins élevé dans le cadre de l'utilisation de l'arithmétique fixe (ligne 13 de la Table 2.4), car la surface du circuit est moins importante [NNF07, KYY15]. Cette arithmétique est basée uniquement sur du calcul entier avec des formats flexibles (n'importe quel format est possible tant que le développeur fait les ajustements appropriés). Par contre, l'arithmétique flottante effectue des calculs complexes utilisant les formats imposés par la norme IEEE754 (Table 2.1) et nécessitant des, CPUs sophistiqués coûtant plus cher [KYY15]. À travers la Table 2.4, nous pouvons conclure qu'il est intéressant d'utiliser l'arithmétique fixe dans les systèmes embarqués pour sa rapidité de calcul, sa portabilité, sa faible consommation en énergie et son prix moins élevé des CPUs. Si le besoin en précision est important et si l'utilisateur n'est pas familier avec l'arithmétique fixe, l'utilisation de l'arithmétique flottante sur des machines dotées d'un FPU serait plus adaptée.

2.5 Conclusion

Dans ce chapitre, nous avons présenté des définitions communes aux arithmétiques flottante et fixe. Ensuite nous avons donné un aperçu sur la norme IEEE754 et ses différentes propriétés (représentation des nombres, valeurs spéciales, exceptions, arrondis et opérations). Nous avons également abordé l'arithmétique fixe, basée sur les entiers, en explicitant ses propriétés (représentation des nombres, opérations arithmétiques, arrondis) et quelques librairies et algorithmes existants. Et en fin, une comparaison entre les deux arithmétiques a été faite en montrant les points forts et faibles de chacune. Dans le chapitre suivant, nous allons présenter les réseaux de neurones, en montrant leur composition, fonctionnement, fonctions d'activation ainsi que, quelques outils pour la compression de ces réseaux.

Chapitre 3

« En informatique, la miniaturisation augmente la puissance de calcul. On peut être plus petit et plus intelligent. »

— Bernard Werber

Réseaux de Neurones

♣ ◇ ♣

L'Intelligence Artificielle (IA), branche de l'informatique fondamentale, a été développée dans le but de simuler les comportements du cerveau humain. En 1943, McCulloch et Pitts ont proposé les premières notions de neurone formel [MP43]. Ils ont montré que, des réseaux de neurones (RNs) formels simples peuvent théoriquement réaliser des fonctions logiques et arithmétiques, mais en 1969, la recherche sur les RNs a perdu son intérêt, à cause de l'impossibilité de traitement des problèmes non linéaires. Ces dernières années, avec l'apparition des ordinateurs puissants (unités de calcul, mémoires, etc.), l'IA est de nouveau réapparue dans le monde de la recherche et, est mise sous le feu des projecteurs. Elle est en train d'émerger, de se développer et d'être utilisée dans plusieurs secteurs, tels que la médecine, la finance, la robotique, etc.

Dans ce chapitre, nous abordons les RNs, présentons leurs propriétés et citons quelques outils pour leur compression. Ce chapitre est organisé comme suit :

- La Section 3.1 donne un aperçu sur la composition (couches, entrées, sorties), la structure et le fonctionnement d'un RN (calculs effectués, etc.).
- La Section 3.2 montre les fonctions d'activation les plus utilisées pour activer (ou pas) les neurones du réseau. Dans cette section, nous présentons ces fonctions, ainsi que leurs approximations, en utilisant les deux arithmétiques flottante et fixe.
- Les différents types de RNs sont présentés dans la Section 3.3.
- Section 3.4 montre quelques outils existants pour la compression des RNs.
- La Section 3.5 conclut ce chapitre.

3.1 Composition et Fonctionnement d'un Réseau de Neurones

Les RNs traitent des problèmes de classification, de reconnaissance d'écriture et d'objets, etc. Ces RNs se composent principalement d'une couche d'entrée lisant les signaux entrants, des couches cachées effectuant des calculs intermédiaires et, une couche de sortie fournissant la réponse du problème traité. Un RN est composé de couches successives reliées entre elles. Une couche est un ensemble de neurones n'ayant pas de connexion entre eux.

La Figure 3.1 montre un RN composé d'une couche d'entrée, de trois couches intermédiaires cachées et d'une couche de sortie. Chaque couche est composée d'un ou plusieurs neurones. Par exemple, dans le RN de la Figure 3.1, la couche d'entrée se compose de deux neurones, les couches cachées contiennent trois neurones et la couche de sortie en a un seul. Dans un RN, un neurone d'une couche cachée est connecté en entrée à chacun des neurones de la couche précédente, et en sortie à chaque neurone de la couche suivante. La Figure 3.2 effectue un zoom sur un seul neurone du RN de la Figure 3.1. Ce neurone prend en entrée un vecteur $X = (x_1, ..., x_n)^t$, des poids synaptiques $W = (w_1, ..., w_n)$ et un biais b afin de calculer la somme pondérée des poids synaptiques $\sum_{i=1}^{n} w_i x_i + b$. Cette somme est appelée

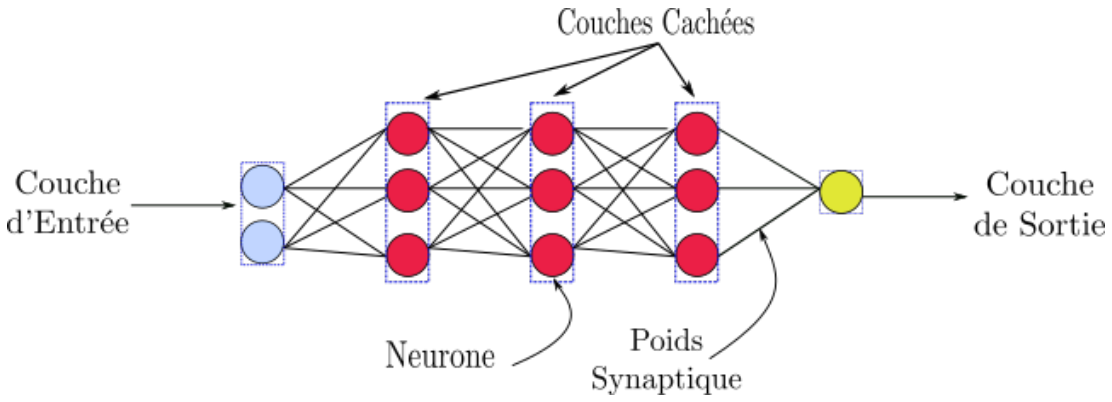

FIGURE 3.1: Représentation d'un Réseau de Neurones.

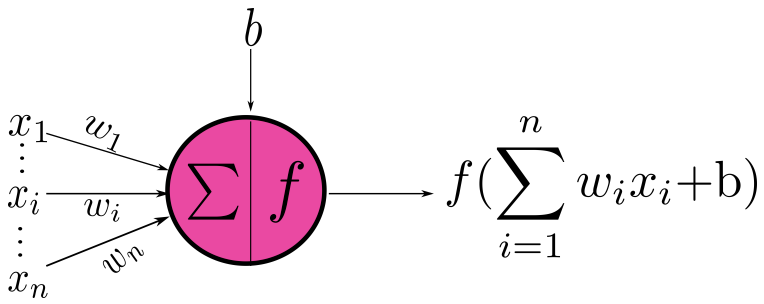

FIGURE 3.2: Zoom sur un Neurone.

fonction affine ou pré-activation. Ensuite, une fonction f, appelée fonction d'activation ou fonction de transfert, est appliquée à cette somme pondérée des poids synaptiques. Cette dernière est définie formellement dans la Définition 3.1.1. Une fonction d'activation sert à activer un neurone ou pas, suivant un seuil (en introduisant une non-linéarité). Il en existe plusieurs, nous présentons les plus utilisées dans la Section 3.2.

Définition 3.1.1. *Dans un RN composé de m couches et de n neurones par couche, la fonction calculant la sortie des neurones est définie par*

$$\begin{aligned} f: \ \mathbb{F}^n &\longrightarrow \mathbb{F} \\ X &\longmapsto u = f(X) = f(\sum_{i=1}^{n} w_i.x_i + b), \end{aligned} \tag{3.1}$$

où $X \in \mathbb{F}^n$ est le vecteur des entrées contenant les éléments x_i, $b \in \mathbb{F}$ est le biais, les $w_i \in \mathbb{F}$ sont les poids synaptiques et $u \in \mathbb{F}$ est la sortie du neurone.

3.2 Fonctions d'Activation

Une fonction d'activation (ou fonction de seuillage, ou encore fonction de transfert) est une fonction mathématique appliquée à un signal en sortie d'un neurone. Elle est spécifique à chaque couche. Il existe deux catégories de fonctions d'activation : les fonctions linéaires (*Linear*) et non linéaires (*ReLU*, *Sigmoid*, *Tanh*, *Softmax*, etc.) Une fonction non linéaire sert à introduire une non-linéarité dans le fonctionnement du neurone. Elle permet de l'activer ou pas. Les fonctions d'activation présentent généralement trois intervalles :

- En dessous du seuil [1], le neurone est non-actif ;
- Aux alentours du seuil, une phase de transition ;
- Au-dessus du seuil, le neurone est actif.

[1] Valeur numérique que doit atteindre ou dépasser un signal à la sortie d'un neurone afin qu'il soit activé.

Dans cette section, nous définissions les fonctions d'activation les plus utilisées dans les RNs, telles que l'*unité linéaire rectifiée*, le *sigmoïde*, la *tangente hyperbolique* et la fonction *linéaire*. Nous donnons également les formules permettant d'approcher le *sigmoïde* et la *tangente hyperbolique*. Ces fonctions d'activation sont définies dans les deux arithmétiques flottante et fixe.

3.2.1 Fonctions d'Activation dans l'Arithmétique Flottante

Les fonctions d'activation linéaires et non linéaires les plus sollicitées dans les RNs : *linéaire, unité linéaire rectifiée, sigmoïde et tangente hyperbolique* sont définies dans cette partie, en utilisant l'arithmétique flottante [ANS19]. Nous introduisons également des formules pour approcher le *sigmoïde* et la *tangente hyperbolique*.

Linéaire

La fonction d'activation *linéaire*, notée *Linear*, appelée également fonction identité, calcule l'identité d'une entrée flottante $a \in \mathbb{F}$. La fonction *Linear* est définie dans la Définition 3.2.1.

Définition 3.2.1. *Soit le nombre flottant $a \in \mathbb{F}$ et soit la fonction d'activation Linear. Cette dernière est définie par*

$$\forall a \in \mathbb{F}, \quad Linear(a) = a. \tag{3.2}$$

Unité Linéaire Rectifiée

La fonction d'activation *unité linéaire rectifiée*, notée *ReLU*, est une fonction d'activation non linéaire éliminant les valeurs négatives. Cette fonction est définie dans la Définition 3.2.2.

Définition 3.2.2. *Soit le nombre flottant $a \in \mathbb{F}$, et soit la fonction d'activation ReLU telle que*

$$\forall a \in \mathbb{F}, \quad ReLU(a) = max(0, a), \tag{3.3}$$

où max, est la fonction calculant le maximum entre deux nombres flottants.

Sigmoïde

La fonction d'activation *sigmoïde*, notée *Sig*, est une fonction d'activation non linéaire définie dans la Définition 3.2.3. Cette fonction peut être approchée de différentes manières [TR20, CRRS20, ÇTG15, Tom03, ACHG97], mais dans ces travaux nous adoptons l'approximation linéaire par morceaux de Çetin et al. [ÇTG15], notée *Sigmoid*. Cette dernière est donnée dans la Définition 3.2.4.

Définition 3.2.3. *Soit le nombre flottant $a \in \mathbb{F}$ et soit la fonction d'activation Sig telle que*

$$\forall a \in \mathbb{F}, \quad Sig(a) = \frac{1}{1 + e^{-a}}. \tag{3.4}$$

Définition 3.2.4. *L'approximation linéaire par morceaux de la fonction d'activation sigmoïde, dite Sigmoid, est définie par*

$$\forall a \in \mathbb{F}, \quad Sigmoid\,(a) = \begin{cases} 1 & \text{si} \quad a \geqslant 5, \\ 0.03125 \times a + 0.84375 & \text{si} \quad 2.375 \leqslant a < 5, \\ 0.125 \times a + 0.625 & \text{si} \quad 1 \leqslant a < 2.375, \\ 0.25 \times a + 0.5 & \text{si} \quad 0 \leqslant a < 1, \\ 1 - Sigmoid\,(a) & \text{si} \quad a < 0. \end{cases} \tag{3.5}$$

Nous allons utiliser cette fonction dans l'arithmétique fixe par la suite (Définition 3.2.9), c'est pourquoi nous avons choisi cette représentation. Cette dernière contient uniquement les opérations élémentaires (addition, soustraction et multiplication) et les coefficients sont exactement représentés dans l'arithmétique fixe, en utilisant cinq bits seulement dans le pire cas, pour représenter 0.84375 dans l'Équation (3.5).

Tangente Hyperbolique

La fonction d'activation *tangente hyperbolique*, notée *Th*, est une fonction d'activation non linéaire définie dans la Définition 3.2.5. Cette fonction peut être approchée en utilisant la relation entre les deux fonctions *sigmoïde* et *tangente hyperbolique*. La version approchée de *Th* est notée *Tanh*, et est donnée dans la Définition 3.2.6.

Définition 3.2.5. *Soit le nombre flottant $a \in \mathbb{F}$ et soit la fonction d'activation Th telle que*

$$\forall a \in \mathbb{F}, \quad Th(a) = \frac{1 - e^{-2a}}{1 + e^{-2a}}. \tag{3.6}$$

Définition 3.2.6. *La relation entre les deux fonctions d'activation Sigmoid et Tanh est donnée par*

$$\forall a \in \mathbb{F}, \quad Tanh\,(a) = -1 + 2 \times Sigmoid\,(2a). \tag{3.7}$$

3.2.2 Fonctions d'Activation dans l'Arithmétique Fixe

Dans cette partie, les versions en fixe des fonctions d'activation : *linéaire, unité linéaire rectifiée, sigmoïde et tangente hyperbolique* sont introduites. Rappelons que selon la Définition 2.3.1, un nombre fixe $\hat{a} \in \aleph$ s'écrit sous la forme $\hat{a} = (-1)^{s^{\hat{a}}} \times V^{\hat{a}}_{<M^{\hat{a}}, L^{\hat{a}}>}$, où $s^{\hat{a}} \in \{0, 1\}$, $M^{\hat{a}} \in \mathbb{Z}$ et $V^{\hat{a}}, L^{\hat{a}} \in \mathbb{N}$. Soulignons que $\oplus, \ominus$ et $\otimes$, représentent respectivement l'addition, la soustraction et la multiplication en fixe. Notons que, dans ce manuscrit les fonctions d'activation en fixe, sont appelées : $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ et $\hat{Tanh}$.

Linéaire

La fonction d'activation *linéaire* en fixe, notée $\hat{Linear}$, est la version en fixe de la fonction flottante *Linear*, donnée dans la Définition 3.2.1. $\hat{Linear}$ est définie dans la Définition 3.2.7.

Définition 3.2.7. *Soit le nombre fixe $\hat{a} \in \aleph$, et soit la fonction d'activation $\hat{Linear}$ telle que*

$$\forall \hat{a} \in \aleph, \quad \hat{Linear}(\hat{a}) = \hat{a}. \tag{3.8}$$

Unité Linéaire Rectifiée

La version en fixe du *ReLU* flottant (Définition 3.2.2) est donnée dans la Définition 3.2.8.

Définition 3.2.8. *Soit le nombre fixe $\hat{a} \in \aleph$ et soit le zéro en fixe $\hat{0} \in \aleph$, représenté par $0_{<0,0>}$. La fonction d'activation $\hat{ReLU}$ est donnée avec*

$$\forall \hat{a} \in \aleph, \quad \hat{ReLU}(\hat{a}) = \hat{max}(\hat{0}, \hat{a}) = \begin{cases} \hat{a} & \text{si} \quad s^{\hat{a}} = 0, \\ \hat{0} & \text{sinon.} \end{cases} \tag{3.9}$$

Notons que $\hat{max}$ est la fonction calculant le maximum entre deux nombres fixes et rappelons que, $s^{\hat{a}} \in \{0, 1\}$ est le signe de $\hat{a}$.

Sigmoïde

La version en fixe $\hat{Sigmoid}$ de l'approximation linéaire par morceaux du *Sigmoid*, définie dans la Définition 3.2.4, est donnée dans la Définition 3.2.9.

Définition 3.2.9. *Soit le nombre fixe $\hat{a} \in \aleph$, et soit l'approximation linéaire par morceaux en fixe du $\hat{Sigmoid}$ telle que*

$$\forall \hat{a} \in \aleph, \quad \hat{Sigmoid}(\hat{a}) = \begin{cases} 1_{<0,0>} & \text{si} \quad \hat{a} \geqslant 5_{<3,0>}, \\ (1_{<-5,5>} \otimes \hat{a}) \oplus 27_{<-1,5>} & \text{si} \quad 19_{<2,3>} \leqslant \hat{a} < 5_{<3,0>}, \\ (1_{<-3,3>} \otimes \hat{a}) \oplus 5_{<-1,3>} & \text{si} \quad 1_{<1,0>} \leqslant \hat{a} < 19_{<2,3>}, \\ (1_{<-2,2>} \otimes \hat{a}) \oplus 1_{<-1,1>} & \text{si} \quad 0_{<1,0>} \leqslant \hat{a} < 1_{<1,0>}, \\ 1_{<0,0>} \ominus \hat{Sigmoid}(\hat{a}) & \text{si} \quad \hat{a} < 0_{<1,0>}. \end{cases} \tag{3.10}$$

Tangente Hyperbolique

La version en fixe $\hat{Tanh}$, correspondante à la fonction d'activation *Tanh*, est obtenue via l'Équation (3.7). $\hat{Tanh}$ est définie dans la Définition 3.2.10.

Définition 3.2.10. *Soit le nombre fixe $\hat{a} \in \aleph$, et soit $\hat{Tanh}$, l'approximation permettant de calculer la fonction d'activation Tanh dans l'arithmétique fixe, telle que*

$$\forall \hat{a} \in \aleph, \quad \hat{Tanh}(\hat{a}) = \begin{cases} 1_{<0,0>} & \text{si} \quad \hat{a} \geqslant 5_{<3,0>}, \\ (1_{<-3,3>} \otimes \hat{a}) \oplus -5_{<-3,5>} & \text{si} \quad 19_{<2,3>} \leqslant \hat{a} < 5_{<3,0>}, \\ (1_{<-1,1>} \otimes \hat{a}) \oplus -3_{<-2,3>} & \text{si} \quad 1_{<1,0>} \leqslant \hat{a} < 19_{<2,3>}, \\ \hat{a} \oplus -1_{<-1,1>} & \text{si} \quad 0_{<1,0>} \leqslant \hat{a} < 1_{<1,0>}, \\ 1_{<0,0>} \ominus \hat{Tanh}(\hat{a}) & \text{si} \quad \hat{a} < 0_{<1,0>}. \end{cases} \tag{3.11}$$

Pour plus de détails sur les fonctions d'activation, le lecteur peut se référer à [SSA17].

3.3 Types de Réseaux de Neurones

Au fil du temps, plusieurs types de RNs ont vu le jour, dans le but de traiter des problèmes de différentes classes, allant d'un simple calcul, à des reconnaissances faciales et classifications complexes. Dans cette section, nous présentons brièvement les modèles les plus utilisés : le perceptron simple et multicouche, les RNs *feedforward*, convolutionnels et récurrents.

3.3.1 Réseaux de Neurones Feedforward

Les RNs *feedforward* sont des réseaux où, l'information se propage (se déplace) que dans une seule direction, de la première couche (d'entrée) vers la dernière couche (de sortie). Il n'y a pas de cycles ou de boucles, l'information ne se répète pas et n'est pas mémorisée dans ces RNs. Le perceptron simple, le perceptron multicouche et les RNs convolutionnels appartiennent à cette catégorie de RNs.

3.3.2 Perceptron Simple

Le perceptron est le plus ancien RN, créé par Rosenblatt en 1958 [Ros58]. Il possède un seul neurone (voir Figure 3.2), et constitue la forme la plus simple d'un RN. Le perceptron simple est dit simple parce qu'il dispose uniquement d'une couche en entrée, et d'une couche en sortie. Le perceptron a une seule matrice de poids synaptiques, ce qui le limite à traiter uniquement des fonctions linéaires séparables, où les résultats sont divisés en deux catégories.

3.3.3 Perceptron Multicouche

Le perceptron multicouche est un RN contenant plusieurs couches au sein desquelles, l'information entre par une couche d'entrée et sort par une couche de sortie. À la différence du perceptron simple, le perceptron multicouche dispose entre la couche en entrée et la couche en sortie d'une ou plusieurs couches, dites couches cachées. Le nombre de couches correspond aux nombres de matrices de poids synaptiques dont dispose le réseau. Un perceptron multicouche est donc mieux adapté pour traiter les fonctions non-linéaires. Le perceptron multicouche montré dans la Figure 3.1 est considéré comme un réseau à propagation directe (*feedforward*). Notons que, tous les neurones d'une couche cachée sont entièrement connectés (*fully connected*) avec ceux de la couche précédente et ceux de la couche suivante.

3.3.4 Réseaux de Neurones Convolutionnels

Les RNs convolutionnels (RNC) sont généralement utilisés pour le traitement et la reconnaissance d'images ainsi que, le traitement du langage naturel. Ces réseaux exploitent les

principes de l'algèbre linéaire [Hof71], en particulier la multiplication matricielle, pour identifier les modèles contenus dans une image. Contrairement au perceptron multicouche contenant qu'une partie classification, les RNCs se composent de deux parties :

- Une partie convolutionnelle : son objectif est d'extraire des caractéristiques propres à chaque image en les compressant de façon à réduire leur taille initiale. En résumé, l'image fournie en entrée passe à travers une succession de filtres, créant de nouvelles images appelées cartes de convolutions. Enfin, les cartes de convolutions obtenues sont concaténées dans un vecteur de caractéristiques, appelé code RNC.
- Une partie classification : Le code RNC obtenu en sortie de la partie convolutionnelle est fourni en entrée dans une deuxième partie, constituée de couches entièrement connectées (perceptron multicouche). Le rôle de cette partie est de combiner les caractéristiques du code RNC afin de classer l'image donnée en entrée.

3.3.5 Réseaux de Neurones Récurrents

Les RNs récurrents (RNR) se caractérisent par leurs boucles de retour d'informations. La force des RNRs réside dans leur capacité à prendre en compte des informations contextuelles suite à la récurrence du traitement de la même information (conserver des informations en mémoire). Ils sont principalement utilisés lors de l'utilisation de données de séries temporelles pour prévoir les résultats, par exemple les prévisions des transactions boursières ou des ventes, reconnaissance vocale, reconnaissance du texte, etc. Les RNRs se composent d'une ou plusieurs couches. Le modèle de Hopfield (réseau temporel) [Hop82] est le RNR monocouche le plus connu. Les RNRs multicouche revendiquent quant à eux la particularité de posséder des couples (entrée/sortie) comme les perceptrons, entre lesquels la donnée véhicule à la fois en propagation en avant[2] (*feedforward propagation*) et en rétro[3] propagation (*backpropagation*).

Pour plus de détails sur les types de RNs, le lecteur peut se référer à [GBC16].

3.4 Compression des Réseaux de Neurones

La compression des RNs consiste à réduire la complexité calculatoire et/ou la taille des RNs. Les principales techniques de compression [Ber21] visant à réduire la taille et l'exigence en ressources de calcul sont :

- Hachage : regrouper les paramètres dans un RN pour d'une part, éviter les redondances et, d'autre part, accéder plus rapidement aux données.
- Élagage : supprimer les paramètres d'un modèle qui sont jugés comme non nécessaires à la bonne inférence du réseau.

[2] De la première couche vers la dernière couche du RN.
[3] De la dernière couche vers la première couche du RN.

- Quantification : réduire le nombre de bits requis pour représenter chaque poids synaptique.
- Réduction de la précision numérique : diminuer la complexité des calculs en limitant la précision numérique des données.
- Binarisation : dans un réseau binaire, les poids synaptiques et les activations, sont contraints de prendre une valeur de $+1$ ou -1.

Dans cette section, nous montrons deux outils de compression de RNs `DeepSZ` [JDL+19] et `Condensa` [JGM+20] dans l'arithmétique flottante [ANS19], et les deux outils `F8Net` [JRZ+22] et `Ristretto` [GPMG18], utilisant l'arithmétique fixe (voir Chapitre 2).

Ristretto

`Ristretto`[4] [GPMG18] est un outil d'approximation de RNs convolutionnels [GBC16], basé sur Caffe[5] [JSD+14]. Il réduit les besoins en mémoire, la zone de traitement des éléments et la consommation énergétique globale des accélérateurs matériels [MGG17]. `Ristretto` prend en entrée un modèle entraîné et renvoie une version compressée de ce modèle. L'entrée et la sortie de cet outil sont des fichiers de description du RN et ses paramètres. `Ristretto` compresse des modèles en utilisant l'arithmétique et la représentation en virgule fixe (voir Chapitre 2), au lieu de la virgule flottante [ANS19]. Plus précisément, cet outil analyse un RN convolutionnel donné par rapport à la plage numérique requise pour représenter les poids synaptiques, les activations, et les résultats intermédiaires des couches convolutionnelles et entièrement connectées. Ensuite, il simule l'impact d'un format réduit (arithmétique fixe) ou d'opérateurs arithmétiques de moindre précision sur la précision du modèle. Le flux de quantification (*quantization*) de `Ristretto`, c'est-à-dire compresser n'importe quel RN convolutionnel à virgule flottante sur 32 bits en virgule fixe dynamique [Yat09], minifloat[6] [ANS19] ou arithmétique sans multiplieur (remplacer toutes les multiplications par des décalages), comporte quatre étapes. Ces étapes sont : l'analyse des poids synaptiques, l'analyse des activations, la réduction de la largeur des formats (représentation en virgule fixe) avec test de précision et la dernière étape consiste à retourner un modèle compressé du RN initial.

DeepSZ

`DeepSZ`[7] [JDL+19] est un outil de compression de RNs à perte de précision attendue. Afin d'obtenir un RN compressé, cet outil passe par quatre étapes clés : l'élagage (*pruning*) du réseau, l'évaluation des bornes d'erreur, l'optimisation de la configuration des bornes d'erreur et la génération du modèle compressé. Pour se faire, les bornes d'erreur possibles pour

[4] https://github.com/yuzeng2333/Ristretto-caffe
[5] https://github.com/BVLC/caffe
[6] Représentés sur moins de 32 bits.
[7] https://github.com/szcompressor/DeepSZ

chaque couche doivent être déterminées. Ensuite, un modèle pour estimer la perte globale de précision d'inférence en fonction de la dégradation de la précision causée par les couches décompressées individuelles, est construit. Par la suite, un algorithme d'optimisation dynamique est proposé par cette approche, afin de déterminer la configuration des bornes d'erreur la mieux adaptée. Cet algorithme a pour but de maximiser le taux de compression sous la contrainte de précision d'inférence, définie par l'utilisateur. `DeepSZ` est basé sur l'outil `Caffe` [JSD+14] et le compresseur `SZ`[8] [TDCC17], et s'intéresse principalement à la compression des couches entièrement connectées.

Condensa

L'idée présentée dans [JGM+20] concerne l'utilisation de la quantification (*quantization*) et de l'élagage (*pruning*) des poids synaptiques pour la compression des RNs profonds [GBC16], via l'outil `Condensa`[9]. Cet outil compresse le RN et optimise ses hyperparamètres en utilisant une formulation d'optimisation sous contraintes, et les optimiseurs : *Bayesian Optimization* [JSW98] et *L-C algorithm* [CI18] (la méthode *Augmented Langrangian*). `Condensa` prend en entrée un modèle pré-entraîné et une fonction objective (par exemple minimisation du temps d'exécution, mémoire, etc.) donnés par l'utilisateur, et renvoie un modèle compressé de ce modèle.

F8Net

`F8Net`[10] [JRZ+22] est un outil de quantification (*quantization*) de RNs utilisant uniquement des multiplications à virgule fixe sur 8 bits. Il utilise une analyse statistique pour déterminer les formats fixes, et plus précisément les longueurs fractionnaires, pour les poids synaptiques et les activations, en utilisant leur variance. Le format de chaque couche est déterminé de manière automatique lors de l'apprentissage du RN par l'algorithme introduit dans cette approche.

Discussion

Ces outils de compression visent à réduire les ressources consommées en réduisant le nombre de bits pour représenter les poids synaptiques, les activations, etc. Ces optimisations sont effectuées pendant l'entraînement du RN afin d'obtenir un modèle compressé. Le point en commun avec nos méthodes est le calcul des formats optimaux (le plus petit format fixe) des poids synaptiques, des activations et des calculs dans un RN. La différence est que, dans ces outils de compression, un modèle condensé pour le RN est généré, mais dans notre cas un code contenant les calculs avec des formats fixes minimaux, pour prédire le résultat du RN (classification, fonctions mathématiques, etc.), est synthétisé. Notons que, notre approche prend un RN déjà entraîné en entrée. Il existe bien évidemment

[8] https://github.com/szcompressor/SZ
[9] https://github.com/NVlabs/condensa
[10] https://github.com/snap-research/F8Net

d'autres outils et techniques de compression de RNs, basés sur l'arithmétique fixe tels que [RRV+18, MHNW19, JGWD20, KK21, SYK21, WHC+21, SFN+22].

3.5 Conclusion

Dans ce chapitre, nous avons abordé les RNs en explicitant leur fonctionnement et structure dans la Section 3.1, ainsi que les fonctions d'activation les plus utilisées, dans la Section 3.2. Dans la Section 3.3, différents types de RNs ont été brièvement cités. La dernière Section 3.4 a donné un aperçu sur différents outils et techniques pour compresser les RNs. Dans le chapitre suivant, nous allons présenter la programmation linéaire et la satisfiabilité modulo théories, en citant quelques solveurs existants. Ensuite, quelques outils pour le réglage de précision et la synthèse de code pour les RNs et pour les programmes, sont mis en avant.

Chapitre 4

« L'arithmétique, c'est être capable de compter jusqu'à vingt sans enlever ses chaussures. »

— Walt Disney

Solveurs, Synthèse de Code & Réglage de Précision

♣ ◇ ♣

Ces dernières années, les RNs commencent à être utilisés dans les systèmes embarqués tels que, les voitures autonomes, les caméras, etc. Ces systèmes ont des ressources limitées, tandis que les RNs sont consommateurs de mémoire et de calculs. L'une des solutions pour exécuter ces RNs sur des systèmes embarqués, est d'utiliser l'arithmétique fixe (voir Chapitre 2). Cependant, les formats fixes (position de la virgule fixe) des nombres et variables fixes, doivent être déterminés et gérés manuellement par le développeur, comme nous l'avons vu au Chapitre 2. Pour calculer ces formats de façon optimale, nous devons résoudre des problèmes d'optimisation. Ces problèmes peuvent être résolus via des approches telles que, la programmation linéaire [Kar08] ou les solveurs de satisfiabilité modulo théories [Mon16]. La technique permettant de trouver le format fixe pour chaque variable du programme, satisfaisant les contraintes de l'utilisateur, est intitulée *réglage de précision* [DM18]. Cette dernière permet de produire un code avec des

formats réglés selon les exigences de l'utilisateur. Le but de ce chapitre est de passer en revue quelques techniques et outils existants dans la littérature sur, le réglage de précision et la synthèse de code. Il est organisé comme suit :

- La Section 4.1 donne, d'une part, un aperçu sur la programmation linéaire, ses cas d'usage et l'écriture standard d'un problème d'optimisation linéaire. Et d'autre part, elle présente les solveurs SAT et les solveurs de satisfiabilité modulo théories.
- Quelques outils de réglage de précision, pour les programmes et pour les RNs, sont mis en avant dans la Section 4.2, ainsi que les techniques qu'ils utilisent.
- La Section 4.3 présente quelques outils de synthèse de code virgule fixe pour les programmes ainsi que pour les RNs, basés sur différentes méthodes telles que, le réglage de précision et le calcul des bornes d'erreur.
- La Section 4.4 conclut ce chapitre.

4.1 Solveurs

Dans cette thèse, nous générons des contraintes afin de calculer les formats (Section 2.3) des nombres fixes. Ces contraintes doivent être résolues via un solveur. Cependant, il existe plusieurs solveurs et plusieurs techniques pour trouver une solution (si elle existe) à ces contraintes. Dans cette section, nous nous intéressons uniquement à la programmation linéaire (PL) et à la satisfiabilité modulo théories (SMT).

4.1.1 Programmation Linéaire

La Programmation Linéaire (PL) [Kar08] est utilisée pour résoudre des problèmes d'optimisation. Les domaines d'application de ces problèmes sont très nombreux aussi bien dans la nature des problèmes abordés (planification et contrôle de production, distribution dans des réseaux, etc.) que dans les secteurs industriels : énergie, transports (aériens, routiers et ferroviaires), télécommunications, industrie forestière, finance. On appelle PL, le problème mathématique qui consiste à optimiser (maximiser ou minimiser) une fonction linéaire de plusieurs variables reliées par des relations linéaires, appelées contraintes. Formellement, une solution qui satisfait toutes les contraintes est appelée *solution admissible*. Si cette dernière renvoie la valeur maximale (ou minimale) de la fonction de coût (objective), alors cette solution est dite *solution admissible optimale*. Le terme PL suppose que les solutions à trouver doivent être représentées en variables réelles. S'il est nécessaire d'utiliser des variables entières dans la modélisation du problème, on parle alors de programmation linéaire en nombres entiers (PLNE) [Pap81]. Il est important de savoir que les problèmes PLNE sont NP-difficile et nettement plus compliqués à résoudre que les problèmes PL à variables continues (réelles). La Définition 4.1.1 définit formellement un problème d'optimisation linéaire.

Définition 4.1.1. *Un problème d'optimisation linéaire sous forme standard s'écrit sous la forme :*

$$\begin{array}{ll} \text{maximiser (ou minimiser)} & Z(x) = c^T x, \\ \text{sous les contraintes} & Ax \leqslant b, \quad (\text{ou } Ax \geqslant b) \\ & x \geqslant 0. \end{array} \tag{4.1}$$

où $c = (c_1, \ldots, c_n)^T$ et (la variable) $x = (x_1, \ldots, x_n)^T$ sont des vecteurs colonnes à n lignes, $A = (a_{ij})_{1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n}$ est une matrice à m lignes et n colonnes, et $b = (b_1, \ldots, b_m)^T$ est un vecteur colonne à m lignes. Notons que $Z(x)$ est la fonction de coût (objective) du problème à maximiser (ou à minimiser) et $Ax \leqslant b$ (ou $Ax \geqslant b$) est l'ensemble des contraintes linéaires avec des coefficients réels.

Ces problèmes d'optimisation sont résolus à l'aide d'un solveur, dit solveur PL. En pratique, il existe plusieurs solveurs PL tels que, `GLPK` [Mak12], `Linprog` [WB20], `LP_SOLVE` [BEN16], etc. Ces solveurs sont libres d'accès. Il existe également d'autres solveurs n'étant pas libres d'accès tels que, `CPLEX` [Cpl09] et `Gurobi` [GO20]. Ces solveurs sont basés sur la méthode du *simplexe* [DOW55], la méthode des *points intérieurs* [Dik67] ou la méthode des *ellipsoïdes* [BGT81].

Exemple 4.1.1. *Une jeune entreprise a le projet de fabriquer deux produits nécessitant l'utilisation de deux machines. La première machine (A) ne peut travailler que* 400 *heures par mois et la deuxième machine (B),* 600 *heures. La fabrication du premier produit P_1 nécessite une heure de la machine A et une heure de la machine B, et il est vendu à* 16000 *euros l'unité. Le second produit P_2 nécessite deux heures de la machine A et une heure de la machine B, il est vendu à* 10000 *euros l'unité. Cette jeune entreprise cherche à définir le programme de fabrication lui permettant de rendre maximal son bénéfice. Ce problème peut être représenté comme un problème d'optimisation avec des contraintes linéaires, où les variables à maximiser sont le nombre de produits P_1 et P_2. En utilisant la Définition 4.1.1, ce problème s'écrit sous la forme*

$$\begin{array}{llcll} \text{maximiser} & Z(P_1, P_2) & = & 16000P_1 + 10000P_2, & (i) \\ \text{sous les contraintes} & P_1 + P_2 & \leqslant & 400, & (ii) \\ & 2P_1 + P_2 & \leqslant & 600, & (iii) \\ & P_1 & \geqslant & 0, & (iv) \\ & P_2 & \geqslant & 0. & (v) \end{array} \tag{4.2}$$

Notons que (i) représente la fonction de coût à maximiser, c'est-à-dire le bénéfice maximal de l'entreprise. Les contraintes (ii) et (iii) répondent respectivement aux exigences des machines. Les deux dernières contraintes (iv) et (v) permettent d'assurer la condition sur la positivité de P_1 et P_2. Une solution optimale à ce problème est

$$P_1 = P_2 = 200, \text{ et } Z(P_1, P_2) = 5200000.$$

Ces valeurs sont obtenues à l'aide d'un solveur PL basé sur la méthode du simplexe [DOW55].

Pour plus d'informations sur la programmation linéaire, j'invite le lecteur à se référer au livre de Karloff [Kar08].

4.1.2 Satisfiabilité Modulo Théories (SMT)

Dans cette sous-section, nous présentons en premier lieu les solveurs SAT, ensuite les solveurs de satisfiabilité modulo théories (SMT), ainsi que leurs fonctionnements respectifs.

Résolution SAT

De manière générale, à partir d'une formule de logique propositionnelle F sous forme normale conjonctive (*FNC*), un algorithme de résolution SAT explore l'espace des valeurs et construit progressivement un modèle M pour F. Notons que, F est composée d'une conjonction de clauses, où chaque clause est une disjonction de littéraux (voir Exemple 4.1.2). Par la suite, le modèle M est prolongé par décision ou par déduction de la valeur d'un littéral l de M et F, qui n'est pas encore dans M. Si une décision produit un échec (c'est-à-dire que tous les littéraux sont définis sur "$faux$"), l'algorithme revient en arrière et inverse la décision. L'architecture Davis-Putnam-Logemann-Lovelan (DPLL) [BM07] contient trois étapes fondamentales : propagation (*Propag*), décision (*Decide*) et retour en arrière (*Backtrack*). Le principe de l'algorithme DPLL est la construction incrémentale d'un modèle M, pour la formule de logique propositionnelle F en FNC. Pendant la recherche :

- Une variable (un littéral) peut avoir la valeur "*vrai/ faux/ non assigné*".
- Une clause peut être :
  - SAT si et seulement si au moins un de ses littéraux est assigné à "$vrai$";
  - $Unit$ si et seulement si tous ses littéraux sauf un sont assignés à "$faux$";
  - $Conflict$ si et seulement si tous ses littéraux sont affectés à "$faux$";
  - $Undef$ si et seulement si ce n'est pas SAT, $Unit$ ou $Conflict$.
- Une formule FNC est SAT si toutes ses clauses sont SAT.

Exemple 4.1.2. *Vérifions la satisfiabilité de la formule suivante :*

$$(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge x_1. \tag{4.3}$$

Dans la Table 4.1 de cet exemple, les littéraux en violet, ayant un exposant v, ont pour valeur "vrai" et les littéraux en bleu, ayant un exposant f, la valeur "$faux$" leur a été assignée. La négation de x est représentée par $\bar{x}$. La Table 4.1 montre la manière dont est construit le modèle M, pour la formule donnée dans l'Équation (4.3). La deuxième ligne de la table montre la formule initiale. Dans la troisième ligne (respectivement quatrième ligne), une propagation de x_1^v (respectivement x_2^f) est effectuée. Dans la cinquième ligne, la valeur "vrai" est affectée à x_3. Ensuite, une opération de décision est menée. Dans la cinquième ligne, on propage x_4 qui a pour valeur "vrai" aussi. À ce stade, toutes les clauses renvoient la valeur SAT sauf $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$, et donc la formule de l'Équation (4.3), ne sera jamais SAT avec ces valeurs de x_1, x_2, x_3 et x_4. Pour remédier à cela, un retour en arrière s'impose en attribuant la valeur "$faux$" à x_3 cette fois-ci (septième et huitième

Opération	Modèle (M)	Formule
-	-	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Propag x_1^v	x_1^v	$\{x_1^v, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1^f, \bar{x}_2\}, \{\bar{x}_1^f, \bar{x}_3, \bar{x}_4\}, \{x_1^v\}$
Propag x_2^f	x_1^v, x_2^f	$\{x_1^v, x_2^f\}, \{x_2^f, \bar{x}_3, x_4\}, \{\bar{x}_1^f, \bar{x}_2^v\}, \{\bar{x}_1^f, \bar{x}_3, \bar{x}_4\}, \{x_1^v\}$
Decide x_3^v	x_1^v, x_2^f, x_3^v	$\{x_1^v, x_2^f\}, \{x_2^f, \bar{x}_3^f, x_4\}, \{\bar{x}_1^f, \bar{x}_2^v\}, \{\bar{x}_1^f, \bar{x}_3^f, \bar{x}_4\}, \{x_1^v\}$
Propag x_4^v	$x_1^v, x_2^f, x_3^v, x_4^v$	$\{x_1^v, x_2^f\}, \{x_2^f, \bar{x}_3^f, x_4^v\}, \{\bar{x}_1^f, \bar{x}_2^v\}, \{\bar{x}_1^f, \bar{x}_3^f, \bar{x}_4^f\}, \{x_1^v\}$
Backtrack x_3^v	x_1^v, x_2^f	$\{x_1^v, x_2^f\}, \{x_2^f, \bar{x}_3, x_4\}, \{\bar{x}_1^f, \bar{x}_2^v\}, \{\bar{x}_1^f, \bar{x}_3, \bar{x}_4\}, \{x_1^v\}$
Decide x_3^f	x_1^v, x_2^f, x_3^f	$\{x_1^v, x_2^f\}, \{x_2^f, \bar{x}_3^v, x_4\}, \{\bar{x}_1^f, \bar{x}_2^v\}, \{\bar{x}_1^f, \bar{x}_3^v, \bar{x}_4\}, \{x_1^v\}$

TABLE 4.1: Construction d'un Modèle M pour la Formule de l'Équation (4.3).

ligne). Avec ces valeurs de x_1 : "vrai", x_2 : "faux" et x_3 : "faux", le modèle sera SAT quelle que soit la valeur attribuée à x_4. Finalement, le modèle M construit pour cette formule est :

$$M = \{x_1^v, x_2^f, x_3^f\} = \{x_1, \neg x_2, \neg x_3\}.$$

Les problèmes SAT sont NP-Complet à partir de trois littéraux seulement par clause. C'est pour cela, que maints travaux ont été menés dans ce sens pour trouver des solutions à moindre coût [Mon16, Coq19, BHvMW09].

Solveurs SMT

Les problèmes de satisfiabilité sont souvent rencontrés dans la bio-informatique, le génie logiciel, la compilation, l'optimisation, la simulation, etc. La plupart des problèmes de ces disciplines peuvent être modélisés ou représentés à travers des expressions booléennes contenant des variables avec des valeurs ("$vrai/faux$"), et donc des solveurs SAT sont utilisés pour leur résolution. Par contre, il existe d'autres types de problèmes tels que les problèmes issus de la théorie des égalités, de l'arithmétique linéaire [JdM13] et non-linéaire [JdM12], de l'arithmétique flottante [ANS19], nécessitant la théorie de la satisfiabilité pour leur résolution ou des solveurs de satisfiabilité modulo théories [dMDS07, BHvMW09, Mon16, dMB11], dits solveurs SMT. Ces solveurs SMT, sont généralement basés sur un solveur SAT qui en forme le noyau, combiné à un solveur de théories et un moteur d'instanciation. Le solveur SAT effectue le raisonnement sur les variables booléennes. Une fois ces variables assignées, elles sont transmises aux solveurs de théories pour qu'elles soient résolues. Ces solveurs combinent différentes procédures de décision sur des théories via deux méthodes historiquement utilisées : la méthode dite de Shostak [Sho84], et celle de Nelson-Oppen [NO80].

Exemple 4.1.3. *Considérons la formule suivante :*

$$\forall x, y \in \mathbb{R}, \qquad ((x > 0) \vee (y + x \leqslant 0)) \wedge (x - 1 \leqslant 0) \wedge (y \geqslant 0). \tag{4.4}$$

La formule de l'Équation (4.4) n'admet pas de solution, car $y + x \leqslant 0$ ne pourra jamais être négatif (la somme de deux valeurs positives est toujours positive). Soulignons que, les valeurs possibles pour x appartiennent à l'intervalle $]0; 1]$ et y est supérieur ou égal à 0. Si on remplace par exemple $x > 0$ par $x \geqslant 0$, une solution à cette formule sera $x = 0$ et $y = 0$.

Il existe plusieurs solveurs SMT tels que `Z3` [dMB08], `Alt-Ergo` [Coq19], `CVC4` [BCD+11], `MathSat` [CGSS13], `VeriT` [DDF13], etc. Ces derniers combinent la capacité des solveurs SAT à trouver des solutions pour des formules complexes, avec la capacité des solveurs de théories spécialisés à trouver des solutions, aux systèmes de contraintes par rapport aux théories spécifiques du premier ordre. Pour plus de détails sur ces solveurs, le lecteur peut se référer à [BHvMW09, Mon16].

4.2 Réglage de Précision

Ces dernières années, le réglage de précision (ou la précision ajustée) est devenu un axe de recherche important. Il vise l'économie des ressources utilisées (processeurs, mémoire, etc.) dans les équipements ou applications avec des ressources limitées, tels que les systèmes embarqués. Le réglage de précision est envisageable dans le cas où la perte de qualité ou la présence d'une certaine erreur de calcul, sont tolérées dans des applications [CA20] de traitement des médias (vidéo, image, etc.). L'étude menée dans [CA20] a présenté les avantages et les inconvénients de plusieurs outils de réglage de précision. Elle a souligné qu'il existait peu d'outils efficaces pour régler des programmes. Précisons que, le réglage de précision n'est pas une simple tâche limitée à changer le type de données, dans le code source aléatoirement. C'est une technique plus complexe qui analyse la sémantique des programmes. Pour cette raison, divers outils ont été proposés pour aider les développeurs à sélectionner les représentations et les types de données les plus appropriés. De tels outils peuvent intégrer différentes approches, mais leur but est commun : adapter automatiquement ou semi-automatiquement un code original donné en assurant une certaine précision, et en utilisant des types de données flottants ou des formats fixes, plus petits que ceux du programme initial. Dans cette section, nous présentons, en premier lieu, l'outil de réglage de précision pour les RNs d'Ioualalen et Martel [IM19]. Ensuite, nous donnons un aperçu sur quelques outils permettant de régler la précision des programmes (donnés en entrée). Ces outils renvoient les plus petits types de données pour les flottants [RNN+13, Ben21] ou les plus petits formats dans le cadre de l'arithmétique fixe [CCCA20, Ben21], tout en respectant des contraintes de précision.

4.2.1 Outils de Réglage de Précision pour les Programmes

Il existe plusieurs outils de réglage de précision. Dans cette partie, nous présentons ces trois outils de réglage de précision pour les programmes : `Precimonious` [RNN+13], `TAFFO` [CCCA20] et `POP` [Ben21].

Precimonious

`Precimonious`[1] [RNN+13] est un outil de réglage de précision de programmes flottants, fonctionnant par analyse dynamique. Cet outil a été implémenté à l'aide du compilateur *Low Level Virtual Machine* (LLVM) [LA04]. Il a pour but de proposer de nouveaux types de données et de prototyper des configurations de précision mixte[2] [CBB+17, DHS18] pour les variables flottantes, en respectant le seuil d'erreur fixé par l'utilisateur. `Precimonious` prend en entrée un programme annoté avec la précision requise par le développeur. Il effectue une analyse dynamique pour proposer une nouvelle configuration (types de données) de ce programme, utilisant une précision inférieure et satisfaisant les contraintes de performance. Cet outil est basé sur l'algorithme de recherche *delta-debugging* [ZH02], qui garantit de trouver un 1-minimum local s'il en existe. `Precimonious` peut régler n'importe quel programme, y compris les programmes contenant des boucles.

TAFFO

`TAFFO`[3] (Tuning Assistant for Floating-point to Fixed-point Optimization) [CCCA20] est un outil basé sur LLVM [LA04] et conçu pour le réglage de la précision des logiciels, en utilisant l'analyse statique. Il convertit automatiquement un code flottant en code fixe et ajuste la précision de ce dernier. Sa stratégie consiste à collecter statiquement des annotations à partir du code source (flottant) et à les convertir en métadonnées LLVM-IR, dans le but de remplacer les opérations flottantes par des opérations fixes. Cette analyse statique permet de projeter sur la sortie l'erreur introduite par chaque instruction en virgule fixe. `TAFFO` est basé sur l'arithmétique affine [SdF03]. L'avantage de `TAFFO` est qu'il prend en charge les programmes C et C++.

POP

`POP`[4] [Ben21] est un outil de réglage de précision par analyse statique. Son approche est basée sur une modélisation sémantique de la propagation des erreurs numériques à travers le programme, en générant un système de contraintes dont la solution minimale, donne le meilleur réglage de précision de ce programme. Dans les travaux de Ben Khalifa [Ben21], deux méthodes basées sur une approche d'analyse statique ont été proposées. La première méthode combine une analyse d'erreurs en avant et en arrière. Ensuite, ces analyses sont exprimées sous la forme d'un ensemble de contraintes linéaires vérifiées par un solveur SMT [dMB08]. La deuxième méthode consiste à générer un problème de programmation linéaire en nombres entiers (PLNE) [Pap81] à partir du code source du programme, en raisonnant sur le bit de poids fort et le nombre de bits significatifs des valeurs des variables. La solution entière à ce problème, calculée en temps polynomial par un solveur de

[1] https://github.com/corvette-berkeley/precimonious
[2] Utiliser des types de données avec des largeurs différentes tout en assurant une certaine précision.
[3] https://github.com/TAFFO-org/TAFFO
[4] https://github.com/benkhelifadorra/POP-v2.0

programmation linéaire classique [Mak12], donne une optimisation des types de données en nombre de bits. Un ensemble plus fin d'équations sémantiques est également proposé dans cette thèse, basé sur la méthode d'itération sur les politiques pour trouver les nouvelles précisions. Contrairement à `TAFFO`, `POP` est capable de renvoyer des solutions en nombre de bits adaptées (*bit level*) à l'arithmétique flottante (IEEE754), à l'arithmétique fixe et à la bibliothèque `MPFR` pour une précision non standard.

Discussion

Les deux outils `Precimonious` [RNN+13] et `POP` [Ben21], proposent un réglage de précision pour les programmes flottants donnés en entrée, tout en respectant la précision requise par l'utilisateur. Le premier outil est basé sur une analyse dynamique et le second sur une analyse statique. Les approches utilisées dans ces outils, en particulier celle de `POP` (borner les erreurs, contraintes linéaires pour calculer la précision) ressemble à la notre, sauf que `POP` est dédié au réglage de la précision des programmes et nous, nous intéressons au réglage des formats fixes des RNs. `TAFFO` convertit un programme flottant en fixe, tout en réglant la précision. Notre approche utilise également l'arithmétique fixe, mais génère un RN en fixe au lieu d'un programme en fixe. Il existe bien évidemment d'autres outils dédiés au réglage de la précision des programmes, basés sur différentes approches telles que l'analyse statique (`FPTaylor` [SJRG15], `FPTuner` [CBB+17], `DAISY` [DIN+18], etc.) ou l'analyse dynamique (`HiFPTuner` [GR18], `ADAPT` [MLO+18], `PROMISE` [GJP+19], etc.), pour les flottants et les fixes.

4.2.2 Outils de Réglage de Précision pour les Réseaux de Neurones

Récemment, la communauté scientifique commence à s'intéresser au réglage de la précision des RNs, afin d'économiser les ressources consommées par ces derniers. Dans cette partie, l'outil de réglage de précision pour les RNs d'Ioualalen et Martel [IM19], est présenté.

L'outil d'Ioualalen et Martel [IM19]

L'idée clé derrière l'outil d'Ioualalen et Martel [IM19] est de réduire la taille des types de données d'un RN flottant (IEEE754 [ANS19]), sans changer le comportement de ce dernier. Ce nouveau RN, avec des types de données plus petits (un type de données par neurone), se comporte comme le RN initial, en tolérant un seuil d'erreur choisi par l'utilisateur pour les résultats (classification, calculs, etc.). Les plages des valeurs des variables correspondantes aux entrées et aux calculs du RN, sont calculées via une analyse dynamique, en effectuant plusieurs exécutions avec plusieurs jeux de données. Une propagation de la précision en allant de la première couche du RN vers la dernière couche, ainsi qu'une propagation de la précision de la dernière couche vers la première, sont effectuées, pour générer des contraintes. Ces dernières ont pour but de calculer le plus petit type de données possible pour chaque neurone. Ces contraintes sont résolues avec la programmation linéaire [Kar08].

Discussion

L'outil d'Ioualalen et Martel [IM19], présenté ci-dessus, est dédié aux RNs flottants. Cet outil prend en entrée un RN entraîné et un seuil d'erreur à ne pas dépasser, et propose une configuration pour les types de données de ce RN. Notre approche effectue un raisonnement similaire à celui-ci sauf que, nous nous intéressons à l'arithmétique fixe et au réglage des formats fixes, au lieu des flottants. Il existe également plusieurs outils cités dans le Chapitre 3 : `Ristretto` [GPMG18], `DeepSZ` [JDL+19], `Condensa` [JGM+20], `F8Net` [JRZ+22], effectuant la compression des RNs via différentes méthodes comme la quantification, l'élagage, etc. Ces outils effectuent le réglage des formats fixes/flottants pendant l'entraînement du RN.

4.3 Synthèse de Code

La synthèse de code [Gul10, SGF10] consiste à produire un code répondant à des exigences exprimées sous forme de spécifications. Les synthétiseurs acceptent une variété de spécifications et réalisent des recherches sur un ensemble de codes, pour déterminer celui qui satisfait au mieux les contraintes imposées par l'utilisateur, contrairement aux compilateurs qui prennent en entrée des codes de haut niveau et, effectuent des traductions vers du code bas niveau. Cette section présente quelques outils [NSW18, GGSS19b, LV20] avec différentes approches permettant de générer du code ou un modèle de RN, utilisant moins de bits pour représenter les variables et les calculs. Nous donnons également un aperçu sur quelques outils [Jha11, AOS12, MRS+12a, DKMS13, CBC+18] synthétisant du code virgule fixe et répondant aux exigences et contraintes données par l'utilisateur.

4.3.1 Outils de Synthèse de Code pour les Réseaux de Neurones

Quand les RNs ont commencé à être utilisés dans les systèmes et équipements avec des ressources limitées (petite mémoire, etc.) tels que les systèmes embarqués, le besoin de réduire la taille des calculs et de la représentation des données dans les RNs a eu lieu, et a donné naissance à plusieurs approches et outils. Nous présentons ci-dessous quelques outils permettant de synthétiser du code ou des modèles pour les RNs avec différentes techniques.

LeFlow

`LeFlow`[5] [NSW18] est un outil open-source pour synthétiser des modèles de calcul numérique écrits dans Tensorflow[6] [AAB+16] à du matériel (*hardware*) synthétisable, via le compilateur *Accelerated Linear Algebra*[7] (XLA) de Google. Ce dernier émet du code

[5] https://github.com/danielholanda/LeFlow
[6] https://www.tensorflow.org/
[7] https://www.tensorflow.org/xla

LLVM [LA04] directement à partir d'une spécification Tensorflow. `LeFlow` permet aux utilisateurs de générer des RNs profonds en Python [vR07] et de prototyper des algorithmes d'apprentissage automatique [GBC16] sur des FPGAs (*Field-Programmable Gate Array*) [Tri12], sans avoir à se soucier des détails de la création d'une conception matérielle, ou d'un code C [KR88] optimisé à l'aide de directives matérielles. Cet outil permet à l'utilisateur de spécifier des optimisations de dépliage de boucles, d'intégration (*inlining*) et de partitionnement de la mémoire dans le code Python d'origine.

SeeDoT

L'outil `SeeDoT` [GGSS19b] génère un code à virgule fixe pour les algorithmes d'inférence d'apprentissage automatique [Bis07, GBC16], qui peuvent s'exécuter sur des appareils contraints en matière de ressources tels que les IoT (internet des objets) [AIM10], les microcontrôleurs et les FPGA [Tri12]. Cet outil présente une stratégie de compilation qui réduit l'espace de recherche pour certains paramètres clés, en particulier les paramètres d'échelle pour la représentation des nombres fixes, utilisés dans le code à virgule fixe généré. Dans cette approche, une implémentation de certaines opérations coûteuses est proposée (multiplication, exponentielle, argmax, etc.). `SeeDot` compile des modèles sur des équipements contraints, c'est-à-dire des équipements ayant une mémoire de taille KB (Kilo-Byte) et n'ayant pas de support matériel pour les opérations en virgule flottante. Il surpasse : l'émulation logicielle de la virgule flottante [ANS19] (Arduino [Ban11]), la virgule fixe à largeur de bits élevée (MATLAB [BBH+03]), la quantification (*quantization*) post-entraînement (TensorFlow-Lite[8]) et les FPGA à virgule flottante et fixe générés à l'aide d'outils de synthèse de haut niveau.

L'outil de Lauter et al. [LV20]

Un outil pour l'analyse semi-automatique des erreurs en virgule flottante [ANS19], pour la phase d'inférence d'apprentissage profond (*deep learning*) [GBC16], est présenté dans [LV20]. Cet outil est compatible avec les modèles Tensorflow/Keras [AAB+16, GP17], et s'appuie sur la bibliothèque Python/C++ [vR07, Str07] pour transformer un RN en code C++, afin d'analyser le besoin de précision du réseau. Cet outil d'analyse de précision semi-automatique a été implémenté en s'appuyant sur des packages existants pour les RNs profonds (*frugally-deep*[9]). Ces packages ont été couplés avec une implémentation C++ à partir de zéro d'une combinaison des deux arithmétiques affine [SdF03] et d'intervalle [Moo79], appelée CAA (*Combined Affine & Interval Arithmetic*). Cette implémentation de CAA est basée sur une analyse d'erreur rigoureuse pour les opérations arithmétiques, ainsi que pour les fonctions d'activation (*Tanh*, etc.), et permet de calculer les bornes d'erreurs relatives et absolues pour les RNs profonds.

[8] https://www.tensorflow.org/lite/guide?hl=fr
[9] https://github.com/Dobiasd/frugally-deep

Discussion

Les outils présentés ci-dessus synthétisent des modèles ou des programmes pour les RNs. `LeFlow` [NSW18] synthétise des RNs en Python à du *hardware*, contrairement à notre approche qui s'intéresse au *software*. L'outil de Lauter et al. [LV20] analyse les erreurs en virgule flottante et transforme un RN en code C/C++, mais dans notre cas nous bornons les erreurs de calcul en virgule fixe et synthétisons un code C (en respectant un seuil d'erreur défini par l'utilisateur). L'outil `SeeDoT` [GGSS19b] génère un code fixe en C, correspondant à un RN dédié aux microcontrôleurs. Cet outil s'intéresse au calcul des formats fixes via une stratégie de compilation. `SeeDoT` teste plusieurs valeurs des paramètres d'échelle et garde la meilleure valeur en termes de précision. Notre approche synthétise également du code fixe en C pour les RNs, mais utilise une stratégie différente (contraintes linéaires) à celle de `SeeDoT`, pour calculer les formats fixes. D'autant plus, notre méthode respecte un seuil d'erreur, défini par l'utilisateur, tandis que `SeeDoT` ne le fait pas.

4.3.2 Outils et Algorithmes de Synthèse de Code pour l'Arithmétique Fixe

Les outils de synthèse de code virgule fixe ont besoin de connaître les formats fixes de chaque nombre et de chaque variable, afin de synthétiser un code, satisfaisant les contraintes imposées par l'utilisateur. Ces formats optimaux[10] sont déterminés de différentes manières, en se basant sur des techniques de réglage de précision et de calcul de bornes d'erreurs. Nous présentons ci-dessous quelques outils permettant de synthétiser du code virgule fixe, tout en respectant les exigences de l'utilisateur.

L'algorithme de Jha [Jha11]

Le but de la thèse de Jha [Jha11] est de donner un algorithme de synthèse optimale d'expressions en virgule fixe [BS17, Yat09], basé sur la synthèse inductive. Son objectif est de trouver la meilleure implémentation en virgule fixe pour une expression donnée (ne considère pas la réécriture des expressions). L'inconvénient de cette approche est qu'il faut plusieurs minutes pour synthétiser un programme en virgule fixe correspondant à une expression.

L'outil de Darulova et al. [DKMS13]

Darulova et al. [DKMS13] ont proposé un outil de synthèse de programmes en virgule fixe, basée sur la réécriture d'expressions et la programmation génétique [PLM08]. Leur algorithme utilise une interprétation abstraite [CC92] pour estimer les bornes d'erreurs d'une implémentation en virgule fixe. Pour évaluer l'adéquation d'une solution proposée, leur approche utilise une analyse statique basée sur l'arithmétique affine [SdF03], pour calculer la borne supérieure de l'erreur. L'objectif est de minimiser cette borne supérieure de l'erreur. Ainsi, les bornes supérieures les moins coûteuses sont utilisées pour comparer deux expressions, par rapport à la précision. Contrairement à la méthode de Jha [Jha11], la

[10] Plus petits formats possibles.

technique de Darulova et al. [DKMS13] peut synthétiser un programme pour une expression en quelques secondes. Cependant, les deux techniques fournissent des bornes pessimistes pour les expressions non linéaires et sont limitées aux programmes linéaires.

Le projet DEFIS

Dans le contexte des polynômes, des filtres linéaires et des algorithmes de traitement de signal, les membres du projet `DEFIS`[11] [MRS+12a] ont présenté de nombreuses approches de synthèse de code en virgule fixe. Pour n'en citer que quelques-unes, l'idée décrite dans [DYSD14] fonctionne en inférant des opérations de convolution de haut niveau à partir du code source d'origine. De plus, Najahi et al. [MNR14, MNR17] ont présenté une approche automatisée pour synthétiser des codes en arithmétique fixe pour certains blocs de base d'algèbre linéaire. Ils prennent une description mathématique du problème ainsi que la plage des variables d'entrée, et synthétisent un code à virgule fixe. Lopez [Lop14] traite la transformation des filtres linéaires et des contrôleurs, en utilisant l'arithmétique fixe. Sa contribution principale est une analyse d'erreur complète, en ce qui concerne les longueurs des mots (nombres fixes), et la formulation de l'optimisation de la longueur des mots comme un problème d'optimisation convexe non linéaire d'entiers. Une extension de ce travail à la classe complète des algorithmes linéaires invariants dans le temps a été proposée dans [Vol17].

L'outil de Cattaneo et al. [CBC+18]

L'outil de Cattaneo et al. [CBC+18] introduit une solution, qui repose sur une passe de transformation de compilateur autonome implémentée dans LLVM [LA04], pour effectuer la conversion des nombres flottants en nombres fixes. Leur outil est spécialement dédié à MIOSIX[12], un système d'exploitation temps réel ciblant les systèmes embarqués. Le but de cette approche est de transformer un bout de code donné en virgule flottante [ANS19], en code sémantiquement équivalent utilisant des calculs en virgule fixe. Le développeur annote le code source (une sélection) via des attributs d'annotation personnalisés sur des variables à virgule flottante. Cela lui permet de se concentrer uniquement sur la partie critique du code, sans affecter l'exactitude du reste du programme. Le compilateur collecte ces annotations et les propage aux valeurs intermédiaires. Ensuite, il crée des instructions basées sur l'arithmétique fixe, qui sont sémantiquement équivalentes au code flottant d'origine. Après la conversion, le code converti doit être compilé pour l'architecture cible et le code objet intégré dans le système de construction MIOSIX.

[11] http://defis.lip6.fr/pmwiki/pmwiki.php?n=Project.Description
[12] https://miosix.org/

L'outil de Aslan et al. [AOS12]

Contrairement aux outils [Jha11, DKMS13, DYSD14, Lop14, Naj14, Vol17, CBC+18] effectuant la conversion des nombres flottants en nombres fixes, Aslan et al. [AOS12] ont développé un outil, qui prend en entrée un code en virgule fixe sur n bits et génère une sortie en virgule flottante sur m bits (norme IEEE754 et formats personnalisés). Cet outil génère du code Verilog RTL (*Register Transfer Level*) [BF01] et son *testbench*, qui peut être implémenté dans les systèmes FPGA [Tri12] et VLSI (*Very Large Scale Integration*) [Che99]. Une approche HLS (*High Level Synthesis*) [CM10] est utilisée pour concevoir et vérifier le code Verilog HDL (*Hardware Description Language*) [Pal03]. Ce dernier est vérifié à l'aide de Modelsim[13] et MATLAB [BBH+03], et synthétisé à l'aide de l'outil de conception Xilinx ISE (*Integrated Synthesis Environment*) [CH17]. L'objectif principal de cet outil proposé est de réduire le TTM[14] (*Time To Market*) et d'augmenter la productivité en vérifiant le matériel pendant le processus de conception, et en réduisant le temps de conception et de vérification.

Discussion

Les outils proposés dans cette partie synthétisent du code pour des programmes. Jha [Jha11] et Darulova et al. [DKMS13], proposent des méthodes basées sur la réécriture des expressions, en estimant les bornes d'erreurs en virgule fixe. Le point en commun avec notre approche est l'utilisation des bornes d'erreurs afin de calculer les formats fixes du RN. Tous les travaux du projet DEFIS [MRS+12a] synthétisent des programmes en fixe. Nos travaux ont des étapes communes avec ceux de Lopez [Lop14] en particulier, qui sont l'analyse des erreurs des calculs et la formulation du problème de calcul des formats fixes comme un problème d'optimisation, sauf que Lopez s'intéresse aux filtres linéaires et aux contrôleurs et nous, nous intéressons aux RNs. Il existe également d'autres outils pour la synthèse de code fixe tels que `TAFFO` [CCCA20] (voir Section 4.2), `POPiX` (voir Chapitre 10), les outils de [MSCS03, MHSN12], etc. Contrairement aux outils [Jha11, DKMS13, CBC+18], au projet `DEFIS` [MRS+12b] et à notre approche, l'outil de Aslan et al. [AOS12] convertit un code fixe en un code flottant.

4.4 Conclusion

Dans ce chapitre, nous avons donné un aperçu sur la programmation linéaire dans la Section 4.1, en montrant son modèle et ses cas d'usage. Ensuite, nous avons abordé les solveurs SMT, qui sont eux mêmes basés sur les solveurs SAT. Les sections 4.2 et 4.3 ont montré quelques outils de réglage de précision, de synthèse de code virgule fixe pour les programmes, et de génération de code et de modèles pour les RNs. Dans la deuxième partie de ce manuscrit, nous mettons en avant le calcul des erreurs commises

[13] `https://eda.sw.siemens.com/en-US/ic/modelsim/`
[14] Le temps qui s'écoule entre la conception d'un produit et sa mise en vente.

dans un RN. Ensuite, nous générons des contraintes afin de calculer les formats fixes minimaux des poids synaptiques, des biais et des calculs intermédiaires dans un RN avec trois méthodes : `CubMeth`, `QuadMeth` et `LinMeth`.

Deuxième partie

Réglage des Formats Fixes des Neurones et des Poids Synaptiques

Chapitre 5

« Une année de travail sur l'intelligence artificielle est suffisante pour vous faire croire en Dieu. »

— Alan Jay Perlis

Calcul d'Erreur d'un Réseau de Neurones à Virgule Fixe

♣◇♣

Dans l'arithmétique fixe, des erreurs de calcul sont générées pendant que des opérations arithmétiques sont effectuées, mais des erreurs sur les entrées (les opérandes) sont également présentes. Ces erreurs sont dues aux arrondis (Section 2.3), pendant le calcul du résultat des opérations arithmétiques et la conversion des nombres flottants en nombres fixes. Dans ce chapitre, nous abordons les erreurs commises dans un RN pendant le calcul de la somme pondérée des poids synaptiques et l'application d'une fonction d'activation (Chapitre 3). Cette somme pondérée et ces fonctions d'activation sont basées sur des opérations arithmétiques (addition, soustraction, multiplication et fonctions mathématiques) en fixe, c'est pourquoi nous commençons d'abord par déterminer les erreurs correspondantes à ces opérations dans l'arithmétique fixe. Ensuite, nous calculons le bit de poids fort de chaque erreur, dans l'arithmétique fixe, indiquant à partir de quel bit l'erreur débute. Ce chapitre est organisé comme suit :

- Les erreurs engendrées par une addition (une soustraction) et une multiplication en fixe, sont présentées dans la Section 5.1. Cette dernière montre également les erreurs commises dans les fonctions mathématiques *sigmoïde* et *tangente hyperbolique* en fixe.
- La Section 5.2 met en avant les erreurs commises dans un RN, plus précisément celles de la somme pondérée des poids synaptiques, et de l'application des fonctions d'activation sur cette somme pondérée.
- La Section 5.3 calcule le bit de poids fort des erreurs, définies dans la Section 5.2, dues à la somme pondérée des poids synaptiques et aux fonctions d'activation .
- La Section 5.4 conclut ce chapitre.

5.1 Modélisation des Erreurs dans l'Arithmétique Fixe

Cette section définit quatre propositions importantes concernant les erreurs commises dans l'addition (la soustraction) et la multiplication en fixe, ainsi que dans les deux fonctions mathématiques *sigmoïde* et *tangente hyperbolique* en fixe. Ces propositions sont nécessaires pour calculer les erreurs commises dans un RN, plus précisément dans le calcul des sorties des neurones (Section 5.2). Soulignons que, l'erreur due à une soustraction en fixe est la même que, celle due à une addition en fixe, c'est pourquoi nous présentons uniquement l'erreur de l'addition fixe dans cette section. Rappelons que, selon la Définition 2.3.1, un nombre fixe $\hat{x} \in \aleph$, s'écrit sous la forme $\hat{x} = (-1)^{s^{\hat{x}}} \times V^{\hat{x}}_{<M^{\hat{x}},L^{\hat{x}}>}$, où $s^{\hat{x}} \in \{0, 1\}$, $M^{\hat{x}} \in \mathbb{Z}$ et $V^{\hat{x}}, L^{\hat{x}} \in \mathbb{N}$. Notons également que, $\oplus$ et $\otimes$, font référence à l'addition et à la multiplication en fixe, définies dans le chapitre 2, et $+$, $\times$ sont utilisés pour l'addition et la multiplication flottantes, respectivement. Dans ce chapitre, $\hat{x} \in \aleph$ est utilisé pour représenter un nombre fixe, $x \in \mathbb{F}$ représente un nombre flottant, $\hat{X} \in \aleph^n$ représente un vecteur contenant n nombres fixes et $\bar{X} \in \mathbb{F}^n$ est un vecteur de n nombres flottants.

5.1.1 Erreur de l'Addition et de la Multiplication

L'erreur correspondante à une opération arithmétique est définie par la différence entre le résultat flottant et le résultat fixe (en valeur absolue). La Proposition 5.1.1 définit l'erreur d'addition de deux nombres fixes, et la Proposition 5.1.2 définit l'erreur d'une multiplication en fixe.

Proposition 5.1.1. *Soient les nombres fixes $\hat{x}$, $\hat{y}$, $\hat{z} \in \aleph$, ayant pour formats $< M^{\hat{x}}, L^{\hat{x}} >$, $< M^{\hat{y}}, L^{\hat{y}} >$ et $< M^{\hat{z}}, L^{\hat{z}} >$ respectivement, et soient x, y, $z \in \mathbb{F}$, leurs représentations flottantes. Notons $\epsilon_{\oplus} \in \mathbb{R}$, l'erreur entre l'addition en fixe $\hat{z} = \hat{x} \oplus \hat{y}$ et l'addition flottante $z = x + y$, définies dans le Chapitre 2. Cette erreur est donnée avec*

$$\epsilon_{\oplus} \leqslant 2^{-L^{\hat{x}}} + 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \tag{5.1}$$

Preuve. *Soient $\epsilon_{\hat{x}}, \epsilon_{\hat{y}}, \epsilon_{\hat{z}} \in \mathbb{R}$, les erreurs de troncature (Chapitre 2) de $\hat{x}$, $\hat{y}$ et $\hat{z}$. Ces erreurs sont respectivement bornées par $2^{-L^{\hat{x}}}$, $2^{-L^{\hat{y}}}$ et $2^{-L^{\hat{z}}}$, car $L^{\hat{x}}$, $L^{\hat{y}}$ et $L^{\hat{z}}$ représentent respectivement, le dernier bit correct de $\hat{x}$, $\hat{y}$ et $\hat{z}$. Notons que*

$$x = \hat{x} + \epsilon_{\hat{x}}, \qquad \text{(de même pour } \hat{y} \text{ et } \hat{z}\text{)}. \tag{5.2}$$

Si nous injectons l'Équation (5.2) dans $z = x + y$, nous obtenons $\hat{z} = \hat{x} \oplus \hat{y}$, et l'erreur $\epsilon_{\oplus}$ est

$$\epsilon_{\oplus} \leqslant \epsilon_{\hat{x}} + \epsilon_{\hat{y}} + \epsilon_{\hat{z}}. \tag{5.3}$$

Nous savons aussi que, l'erreur de $\hat{x}$, est bornée par

$$\epsilon_{\hat{x}} \leqslant 2^{-L^{\hat{x}}}, \qquad \text{(de même pour } \hat{y} \text{ et } \hat{z}\text{)}. \tag{5.4}$$

En combinant les équations (5.3) et (5.4), nous obtenons

$$\epsilon_{\oplus} \leqslant 2^{-L^{\hat{x}}} + 2^{-L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \qquad \blacksquare$$

Proposition 5.1.2. *Soient les nombres fixes $\hat{x}$, $\hat{y}$, $\hat{z} \in \aleph$, ayant pour formats $< M^{\hat{x}}, L^{\hat{x}} >$, $< M^{\hat{y}}, L^{\hat{y}} >$ et $< M^{\hat{z}}, L^{\hat{z}} >$ respectivement, et soient x, y, $z \in \mathbb{F}$, leurs représentations flottantes. Notons $\epsilon_{\otimes} \in \mathbb{R}$, l'erreur entre la multiplication en fixe $\hat{z} = \hat{x} \otimes \hat{y}$ et la multiplication flottante $z = x \times y$, définies dans le Chapitre 2. Cette erreur est donnée avec*

$$\epsilon_{\otimes} \leqslant \hat{y} \times 2^{-L^{\hat{x}}} + \hat{x} \times 2^{-L^{\hat{y}}} + 2^{-L^{\hat{x}} - L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \tag{5.5}$$

Preuve. *Soient $\epsilon_{\hat{x}}, \epsilon_{\hat{y}}, \epsilon_{\hat{z}} \in \mathbb{R}$, les erreurs de troncature (Chapitre 2) de $\hat{x}$, $\hat{y}$ et $\hat{z}$. Ces erreurs sont respectivement bornées par $2^{-L^{\hat{x}}}$, $2^{-L^{\hat{y}}}$ et $2^{-L^{\hat{z}}}$, car $L^{\hat{x}}$, $L^{\hat{y}}$ et $L^{\hat{z}}$ représentent respectivement, le dernier bit correct de $\hat{x}$, $\hat{y}$ et $\hat{z}$. En injectant l'Équation (5.2) dans $z = x \times y$, nous obtenons $\hat{z} = \hat{x} \otimes \hat{y}$, et l'erreur $\epsilon_{\otimes}$. Cette dernière est bornée par*

$$\epsilon_{\otimes} \leqslant \hat{y} \times \epsilon_{\hat{x}} + \hat{x} \times \epsilon_{\hat{y}} + \epsilon_{\hat{x}} \times \epsilon_{\hat{y}} + \epsilon_{\hat{z}}. \tag{5.6}$$

Notons que l'Équation (5.4) permet de borner $\epsilon_{\hat{x}}$, $\epsilon_{\hat{y}}$ et $\epsilon_{\hat{z}}$, et donc en combinant les deux équations (5.4) et (5.6), nous obtenons

$$\epsilon_{\otimes} \leqslant \hat{y} \times 2^{-L^{\hat{x}}} + \hat{x} \times 2^{-L^{\hat{y}}} + 2^{-L^{\hat{x}} - L^{\hat{y}}} + 2^{-L^{\hat{z}}}. \qquad \blacksquare$$

5.1.2 Erreur du Sigmoïde et de la Tangente Hyperbolique

L'erreur commise dans l'approximation en fixe de la fonction mathématique *sigmoïde*, définie dans la Définition 3.2.9, est présentée dans la Proposition 5.1.3. Et l'erreur correspondante à l'approximation en fixe de la *tangente hyperbolique*, définie dans la Définition 3.2.10, est donnée dans la Proposition 5.1.4.

Proposition 5.1.3. *Soient $\hat{x}$, $\hat{z} \in \aleph$, deux nombres fixes ayant pour formats $< M^{\hat{x}}, L^{\hat{x}} >$ et $< M^{\hat{z}}, L^{\hat{z}} >$ respectivement, et soient $x, z \in \mathbb{F}$, leurs représentations flottantes. Notons $\epsilon_{\hat{Sigmoid}} \in \mathbb{R}$, l'erreur entre l'approximation linéaire par morceaux du $\hat{Sigmoid}$ en fixe (Définition 3.2.9), notée*

$\hat{z} = \hat{Sigmoid}(\hat{x})$, et l'approximation linéaire par morceaux du Sigmoid flottant (Définition 3.2.4), notée $z = Sigmoid(x)$, telle que

$$\epsilon_{\hat{Sigmoid}} \leqslant \begin{cases} 0 & \text{si} \quad \hat{x} \geqslant 5_{<3,0>}, \\ 2^{-5} \times (1 + \hat{x}) + 2^{-L^{\hat{x}}-4} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 19_{<2,3>} \leqslant \hat{x} < 5_{<3,0>}, \\ 2^{-3} \times (1 + \hat{x}) + 2^{-L^{\hat{x}}-2} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 1_{<1,0>} \leqslant \hat{x} < 19_{<2,3>}, \\ 2^{-1} + 2^{-2} \times \hat{x} + 2^{-L^{\hat{x}}-2} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 0_{<1,0>} \leqslant \hat{x} < 1_{<1,0>}, \\ \epsilon_{\hat{Sigmoid}} + 2^{-L^{\hat{z}}} & \text{si} \quad \hat{x} < 0_{<1,0>}. \end{cases} \tag{5.7}$$

Preuve. *Soient $\epsilon_{\hat{x}}, \epsilon_{\hat{z}} \in \mathbb{R}$, les erreurs de troncature (Chapitre 2) de $\hat{x}$ et $\hat{z}$, telles que $x = \hat{x} + \epsilon_{\hat{x}}$ et $z = \hat{z} + \epsilon_{\hat{z}}$ (Équation (5.2)). Ces erreurs sont respectivement bornées par $2^{-L^{\hat{x}}}$ et $2^{-L^{\hat{z}}}$, car $L^{\hat{x}}$ et $L^{\hat{z}}$ représentent respectivement, le dernier bit correct de $\hat{x}$ et $\hat{z}$. En appliquant les propositions 5.1.1 et 5.1.2 sur l'approximation linéaire par morceaux du Sigmoid flottant, défini dans la Définition 3.2.4, nous obtenons*

$$\epsilon_{\hat{Sigmoid}} \leqslant \begin{cases} 0 & \text{si } \hat{x} \geqslant 5_{<3,0>}, \\ 0.03125.\epsilon_{\hat{x}} + \hat{x}.\epsilon_{0.03125} + \epsilon_{\hat{x}}.\epsilon_{0.03125} + \epsilon_{0.84375} + 2\epsilon_{\hat{z}} & \text{si } 19_{<2,3>} \leqslant \hat{x} < 5_{<3,0>}, \\ 0.125.\epsilon_{\hat{x}} + \hat{x}.\epsilon_{0.125} + \epsilon_{\hat{x}}.\epsilon_{0.125} + \epsilon_{0.625} + 2\epsilon_{\hat{z}} & \text{si } 1_{<1,0>} \leqslant \hat{x} < 19_{<2,3>}, \\ 0.25.\epsilon_{\hat{x}} + \hat{x}.\epsilon_{0.25} + \epsilon_{\hat{x}}.\epsilon_{0.25} + \epsilon_{0.5} + 2\epsilon_{\hat{z}} & \text{si } 0_{<1,0>} \leqslant \hat{x} < 1_{<1,0>}, \\ \epsilon_{\hat{Sigmoid}} + \epsilon_{\hat{z}} & \text{si } \hat{x} < 0_{<1,0>}. \end{cases} \tag{5.8}$$

Substituons dans l'Équation (5.8), $\epsilon_{\hat{x}}$ par $2^{-L^{\hat{x}}}$ (Équation (5.2)) et 0.03125 par son ufp (calcul du bit de poids fort), en utilisant la Définition 2.1.1 (de même pour les autres coefficients). Le résultat obtenu est

$$\epsilon_{\hat{Sigmoid}} \leqslant \begin{cases} 0 & \text{si} \quad \hat{x} \geqslant 5_{<3,0>}, \\ 2^{-5} \times (1 + \hat{x}) + 2^{-L^{\hat{x}}-4} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 19_{<2,3>} \leqslant \hat{x} < 5_{<3,0>}, \\ 2^{-3} \times (1 + \hat{x}) + 2^{-L^{\hat{x}}-2} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 1_{<1,0>} \leqslant \hat{x} < 19_{<2,3>}, \\ 2^{-1} + 2^{-2} \times \hat{x} + 2^{-L^{\hat{x}}-2} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 0_{<1,0>} \leqslant \hat{x} < 1_{<1,0>}, \\ \epsilon_{\hat{Sigmoid}} + 2^{-L^{\hat{z}}} & \text{si} \quad \hat{x} < 0_{<1,0>}. \end{cases}$$

■

Proposition 5.1.4. *Soient $\hat{x}, \hat{z} \in \aleph$, deux nombres fixes ayant pour formats $< M^{\hat{x}}, L^{\hat{x}} >$ et $< M^{\hat{z}}, L^{\hat{z}} >$ respectivement, et soient $x, z \in \mathbb{F}$, leurs représentations flottantes. Notons $\epsilon_{\hat{Tanh}} \in \mathbb{R}$, l'erreur entre l'approximation en fixe de $\hat{Tanh}$ (Définition 3.2.10), notée $\hat{z} = \hat{Tanh}(\hat{x})$, et l'approximation flottante de Tanh (Définition 3.2.6), notée $z = Tanh(x)$, telle que*

$$\epsilon_{\hat{Tanh}} \leqslant \begin{cases} 0 & \text{si} \quad \hat{x} \geqslant 5_{<3,0>}, \\ 2^{-5} + 2^{-4} \times \hat{x} + 2^{-L^{\hat{x}}-3} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 19_{<2,3>} \leqslant \hat{x} < 5_{<3,0>}, \\ 2^{-3} + 2^{-2} \times \hat{x} + 2^{-L^{\hat{x}}-1} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 1_{<1,0>} \leqslant \hat{x} < 19_{<2,3>}, \\ 2^{-1} + 2^{-1} \times \hat{x} + 2^{-L^{\hat{x}}} + 2^{-L^{\hat{z}}+1} & \text{si} \quad 0_{<1,0>} \leqslant \hat{x} < 1_{<1,0>}, \\ \epsilon_{\hat{Tanh}} + 2^{-L^{\hat{z}}} & \text{si} \quad \hat{x} < 0_{<1,0>}. \end{cases} \tag{5.9}$$

Remarquons que la démonstration du calcul d'erreur correspondante à la $\hat{Tanh}$ se fait exactement de la même manière que pour le $\hat{Sigmoid}$.

5.2 Modélisation des Erreurs dans un Réseau de Neurones

Les résultats théoriques sur les erreurs numériques commises à l'intérieur d'un RN entièrement connecté, en utilisant l'arithmétique fixe, sont donnés dans cette section. Il existe deux types d'erreurs : la propagation de l'erreur du vecteur d'entrée et les erreurs dues au calcul de la somme pondérée des poids synaptiques (fonction affine), présentée dans l'Équation (5.10), et des fonctions d'activation, définies dans les définitions 3.2.7, 3.2.8, 3.2.9 et 3.2.10. La Proposition 5.2.1 calcule l'erreur de la somme pondérée des poids synaptiques en fixe. Les erreurs résultantes des fonctions d'activation $\hat{Sigmoid}$ et $\hat{Tanh}$ en fixe, sont respectivement présentées dans les propositions 5.2.2 et 5.2.3. Rappelons que, la formule calculant la sortie d'un neurone, est donnée dans la Définition 3.1.1.

5.2.1 Erreur de la Somme Pondérée des Poids Synaptiques

La Proposition 5.2.1 a pour but de donner la formule d'erreur correspondante à la somme pondérée des poids synaptiques. Rappelons que, dans un RN entièrement connecté de type perceptron multicouche (Section 3.3), ayant m couches et n neurones par couche, la somme pondérée des poids synaptiques pour un neurone i appartenant à la couche k, notée $u_{ki} \in \mathbb{F}$, est donnée avec

$$\begin{aligned} g_{k,i} : \ & \mathbb{F}^n \longrightarrow \mathbb{F} \\ & \bar{X}_{k-1} \longmapsto u_{k,i} = g_{k,i}(\bar{X}_{k-1}) = \sum_{j=0}^{n-1} (w_{k-1,i,j} \times x_{k-1,j}) + b_{k-1,i}. \end{aligned} \tag{5.10}$$

Notons que, $\forall\, k,\ 1 \leqslant k \leqslant m,\ \ \forall\, i,\ 0 \leqslant i < n$, $u_{k,i}$, est le résultat de la somme pondérée des poids synaptiques du neurone i de la couche k, $\bar{X}_{k-1}$ est le vecteur des entrées ($x_{k-1,j} \in \mathbb{F}$), le biais $b_{k-1,i} \in \mathbb{F}$ est l'élément i du vecteur $\bar{B}_{k-1}$, et $w_{k-1,i,j} \in \mathbb{F}$ appartient à la ligne i et la colonne j de la matrice des poids synaptiques W_{k-1}.

La version en fixe de $g_{k,i}$, définie l'Équation (5.10), est notée $\hat{g}_{k,i} \in \aleph$. Cette dernière calcule la somme pondérée des poids synaptiques en fixe, notée $\hat{u}_{k,i} \in \aleph$. Elle est définie par

$$\begin{aligned} \hat{g}_{k,i} : \ & \aleph^n \longrightarrow \aleph \\ & \hat{X}_{k-1} \longmapsto \hat{u}_{k,i} = \hat{g}_{k,i}(\hat{X}_{k-1}) = \sum_{j=0}^{n-1} (\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}, \end{aligned} \tag{5.11}$$

où $\forall\, k,\ 1 \leqslant k \leqslant m,\ \ \forall\, i,j\, 0 \leqslant i,j < n$, $\hat{u}_{k,i}$ est le résultat de la somme pondérée du neurone i de la couche k en fixe, $\hat{b}_{k-1,i}$ est le biais, $\hat{w}_{k-1,i,j}$ sont les poids synaptiques et $\hat{x}_{k-1,j}$ sont les entrées.

Remarque 7. *Dans cette section, tous les opérateurs $\sum$ font référence à l'addition en fixe, sauf celui de l'Équation (5.10), faisant référence à l'addition flottante.*

Proposition 5.2.1. *Soit $\theta_{k,i} \in \mathbb{R}$, l'erreur due aux calculs numériques de $\hat{g}_{k,i}$ et aux conversions des coefficients flottants vers des coefficients en fixe. L'erreur $\theta_{k,i}$ résultante pour chaque neurone i de chaque couche k, est donnée avec*

$$\forall k, 1 \leqslant k \leqslant m,\ \forall i, j, 0 \leqslant i, j < n,$$

$$\theta_{k,i} \leqslant (3n-2) \times (2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}}) + (4n-1) \times 2^{-L^{\hat{u}}_{k,i}}. \tag{5.12}$$

Preuve. *Notre objectif est de borner $\theta_{k,i}$, l'erreur résultante au calcul de la somme pondérée des poids synaptiques ($\hat{g}_{k,i}$), pour chaque neurone de chaque couche. Pour se faire, il suffit de borner la formule de l'Équation (5.11). Commençons d'abord par calculer l'erreur de $\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}$. Ensuite, bornons l'erreur de la somme des multiplications en fixe $\sum_{j=0}^{n-1} (\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j})$, et enfin calculons l'erreur de $\sum_{j=0}^{n-1} (\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}$, à l'aide des propositions 5.1.1 et 5.1.2.*

Soit $e\hat{r}r_{\otimes} \in \mathbb{R}$, l'erreur de la multiplication en fixe de $\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}$, et soit $2^{-L^{\hat{u}}_{k,i}}$, l'erreur de troncature de la sortie du neurone $\hat{u}_{k,i}$ (Équation (5.4)). En utilisant la Proposition 5.1.2, nous obtenons

$$e\hat{r}r_{\otimes} \leqslant 2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{u}}_{k,i}}. \tag{5.13}$$

Ensuite, considérons $e\hat{r}r_{\oplus} \in \mathbb{R}$, l'erreur d'une addition en fixe et $e\hat{r}r_s \in \mathbb{R}$, l'erreur de la somme des multiplications en fixe $\sum_{j=0}^{n-1} (\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j})$. Cette erreur est calculée à travers le résultat de l'Équation (5.13) et la Proposition 5.1.1, telle que

$$e\hat{r}r_s \leqslant n \times e\hat{r}r_{\otimes} + (n-1) \times e\hat{r}r_{\oplus}. \tag{5.14}$$

Alors,

$$e\hat{r}r_s \leqslant n \times e\hat{r}r_{\otimes} + (n-1) \times (2 \times e\hat{r}r_{\otimes} + 2^{-L^{\hat{u}}_{k,i}}). \tag{5.15}$$

En simplifiant les calculs, nous obtenons

$$e\hat{r}r_s \leqslant (3n-2) \times e\hat{r}r_{\otimes} + (n-1) \times 2^{-L^{\hat{u}}_{k,i}}. \tag{5.16}$$

Maintenant, considérons $e\hat{r}r_{\hat{u}_{k,i}} \in \mathbb{R}$, l'erreur de $\sum_{j=0}^{n-1} (\hat{w}_{k-1,i,j} \otimes \hat{x}_{k-1,j}) \oplus \hat{b}_{k-1,i}$, la somme pondérée des poids synaptiques en fixe. Cette erreur est bornée en utilisant l'Équation (5.16) et la Proposition 5.1.1, telle que

$$e\hat{r}r_{\hat{u}_{k,i}} \leqslant e\hat{r}r_s + 2^{-L^{\hat{b}}_{k-1,i}} + 2^{-L^{\hat{u}}_{k,i}}. \tag{5.17}$$

Notons que, $\hat{b}_{k-1,i}$ prend le format de sortie de $\hat{u}_{k,i}$ (alignement des formats avec celui exigé en sortie pendant l'addition fixe dans le Chapitre 2), donc

$$e\hat{r}r_{\hat{u}_{k,i}} \leqslant (3n-2) \times e\hat{r}r_{\otimes} + (n+1) \times 2^{-L^{\hat{u}}_{k,i}}. \tag{5.18}$$

Finalement,

$$e\hat{r}r_{\hat{u}_{k,i}} \leqslant (3n-2) \times (2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}}) + (4n-1) \times 2^{-L^{\hat{u}}_{k,i}}. \tag{5.19}$$

En combinant les équations (5.12) et (5.19), nous obtenons

$$e\hat{r}r_{\hat{u}_{k,i}} = \theta_{k,i}. \qquad \blacksquare$$

5.2.2 Erreur des Fonctions d'Activation

Les erreurs commises dans les fonctions d'activation *sigmoïde* et *tangente hyperbolique* en fixe (Chapitre 3) dans un RN, sont respectivement définies dans les propositions 5.2.2 et 5.2.3. Afin de calculer ces erreurs dans le cadre d'un RN, il suffit de substituer l'entrée $\hat{x}$ par $\hat{u}_{k,i}$ et $2^{-L_{\hat{x}}}$ par $\theta_{k,i}$, dans les propositions 5.1.3 et 5.1.4. Notons que, $\hat{u}_{k,i}$ correspond à la somme pondérée des poids synaptiques en fixe (Équation (5.11)), et $\theta_{k,i}$ est l'erreur commise lors du calcul de $\hat{u}_{k,i}$ (Proposition 5.2.1). L'erreur de troncature $2^{-L_{\hat{z}}}$ dans les propositions 5.1.3 et 5.1.4 correspond à $2^{-L^{\hat{u}}_{k,i}}$ dans ce cas. Dans cette section, considérons les intervalles I_1, I_2, I_3, I_4 et I_5, utilisés dans les propositions tels que

- $I_1 : \hat{u}_{k,i} \geqslant 5_{<3,0>}$,
- $I_2 : 19_{<2,3>} \leqslant \hat{u}_{k,i} < 5_{<3,0>}$,
- $I_3 : 1_{<1,0>} \leqslant \hat{u}_{k,i} < 19_{<2,3>}$,
- $I_4 : 0_{<1,0>} \leqslant \hat{u}_{k,i} < 1_{<1,0>}$,
- $I_5 : \hat{u}_{k,i} < 0_{<1,0>}$.

Ces derniers correspondent aux intervalles de l'approximation linéaire par morceaux du *Sigmoid* et de la *Tanh* en fixe, dans la Section 3.2.

Proposition 5.2.2. *Soient $\hat{u}_{k,i} \in \aleph$, la somme pondérée des poids synaptiques en fixe, définie dans l'Équation (5.11), $\theta_{k,i}$ l'erreur commise sur $\hat{u}_{k,i}$, définie dans l'Équation (5.12), et $2^{-L^{\hat{u}}_{k,i}}$ l'erreur de troncature de $\hat{u}_{k,i}$. Considérons $\gamma_{k,i} \in \mathbb{R}$, l'erreur correspondante à l'approximation linéaire par morceaux du Sigmoid en fixe, telle que*

$$\gamma_{k,i} \leqslant \begin{cases} 0 & \text{si} \quad I_1, \\ 2^{-5} \times (1 + \hat{u}_{k,i}) + \theta_{k,i} \times 2^{-4} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si} \quad I_2, \\ 2^{-3} \times (1 + \hat{u}_{k,i}) + \theta_{k,i} \times 2^{-2} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si} \quad I_3, \\ 2^{-1} + 2^{-2} \times \hat{u}_{k,i} + \theta_{k,i} \times 2^{-2} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si} \quad I_4, \\ \gamma_{k,i} + 2^{-L^{\hat{u}}_{k,i}} & \text{si} \quad I_5. \end{cases} \tag{5.20}$$

Proposition 5.2.3. *Soient $\hat{u}_{k,i} \in \aleph$, la somme pondérée des poids synaptiques en fixe, définie dans l'Équation (5.11), $\theta_{k,i}$ l'erreur commise sur $\hat{u}_{k,i}$, définie dans l'Équation (5.12), et $2^{-L^{\hat{u}}_{k,i}}$ l'erreur de troncature de $\hat{u}_{k,i}$. Considérons $\xi_{k,i} \in \mathbb{R}$, l'erreur de Tânh en fixe, telle que*

$$\xi_{k,i} \leqslant \begin{cases} 0 & \text{si } I_1 \\ 2^{-5} + 2^{-4} \times \hat{u}_{k,i} + \theta_{k,i} \times 2^{-3} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si } I_2, \\ 2^{-3} + 2^{-2} \times \hat{u}_{k,i} + \theta_{k,i} \times 2^{-1} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si } I_3, \\ 2^{-1} + 2^{-1} \times \hat{u}_{k,i} + \theta_{k,i} + 2^{-L^{\hat{u}}_{k,i}+1} & \text{si } I_4, \\ \xi_{k,i} + 2^{-L^{\hat{u}}_{k,i}} & \text{si } I_5 \end{cases} \tag{5.21}$$

Remarque 8. *Dans le pire cas, les erreurs correspondantes aux deux fonctions d'activation Linear et ReLU en fixe, définies respectivement dans les définitions 3.2.7 et 3.2.8, sont inférieures ou égales à $\theta_{k,i}$, présentée dans l'Équation (5.12), car Linear calcule l'identité, et ReLU renvoie $\hat{0}$ si la sortie est négative, ou l'identité sinon.*

5.3 Calcul de l'Ufp de l'Erreur d'un Réseau de Neurones

Cette section traite le calcul du bit de poids fort, dit ufp, des erreurs correspondantes à la somme pondérée des poids synaptiques et des fonctions d'activation en fixe *Sigmoid* et *Tânh*, respectivement, présentées dans les propositions 5.2.1, 5.2.2, 5.2.3. Ces expressions de calcul de bit de poids fort des erreurs, seront utilisées dans la génération des contraintes dans les chapitres 6 et 7, afin de calculer les formats fixes correspondants à $\hat{u}$, $\hat{w}$ et $\hat{x}$, utilisés dans l'Équation (5.11). Ils sont respectivement définies dans les propositions 5.3.1, 5.3.2 et 5.3.3.

5.3.1 Ufp de l'Erreur de la Somme Pondérée des Poids Synaptiques

Dans la Proposition 5.2.1, nous avons donné la formule correspondante à l'erreur de calcul de la somme pondérée des poids synaptiques $\theta_{k,i}$, pour chaque neurone i de la couche k. Maintenant, nous nous intéressons au calcul du bit de poids fort de cette erreur $\theta_{k,i}$, dans la Proposition 5.3.1. En d'autres termes, nous calculons $\text{ufp}(\theta_{k,i})$, à l'aide de la Définition 2.1.1 de l'ufp.

Proposition 5.3.1. *Soit $\text{ufp}(\theta_{k,i}) \in \mathbb{Z}$, le bit de poids fort de l'erreur correspondante à la somme pondérée des poids synaptiques du neurone i de la couche k, et soient $\nu, \mu \in \mathbb{Z}$ deux constantes, telles que*

$$\nu = \text{ufp}(3n-2) \quad \text{et} \quad \mu = \text{ufp}(4n-1), \quad \forall n \geqslant 1, \text{ le nombre de neurones par couche.}$$

Le bit de poids fort de l'erreur du neurone i de la couche k, noté $\text{ufp}(\theta_{k,i})$, est défini par
$\forall\, k, 1 \leqslant k \leqslant m, \forall\, i, j, 0 \leqslant i, j < n,$

$$\text{ufp}(\theta_{k,i}) \leqslant max(\nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\, \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}\,;\, \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\, \mu - L^{\hat{u}}_{k,i}) + 2, \tag{5.22}$$

où max est la fonction calculant le maximum entre des valeurs relatives ($\in \mathbb{Z}$), et $< M^{\hat{x}}_{k-1,j}, L^{\hat{x}}_{k-1,j} >$, $< M^{\hat{w}}_{k-1,i,j}, L^{\hat{w}}_{k-1,i,j} >$, et $< M^{\hat{u}}_{k,i}, L^{\hat{u}}_{k,i} >$ sont les formats fixes des coefficients $\hat{x}_{k-1,j}$, $\hat{w}_{k-1,i,j}$ et $\hat{u}_{k,i}$, respectivement.

Preuve. *Considérons la formule de calcul de l'erreur $\theta_{k,i}$, définie dans l'Équation (5.12) et calculons son ufp, c'est-à-dire son bit de poids fort, en utilisant l'Équation (2.1). Rappelons que, $\forall n \geqslant 1$,*

$$\theta_{k,i} \leqslant (3n-2) \times (2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}}) + (4n-1) \times 2^{-L^{\hat{u}}_{k,i}}. \quad (5.12)$$

Ensuite, écrivons $\theta_{k,i}$ telle que,

$$\theta_{k,i} \leqslant 2^{\mathrm{ufp}(3n-2)+1} \times (2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}}) + 2^{\mathrm{ufp}(4n-1)+1} \times 2^{-L^{\hat{u}}_{k,i}}. \quad (5.23)$$

Remplaçons ν et μ dans l'Équation (5.23),

$$\theta_{k,i} \leqslant 2^{\nu+1} \times (2^{M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}} + 2^{M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}} + 2^{-L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}}) + 2^{\mu+1} \times 2^{-L^{\hat{u}}_{k,i}}. \quad (5.24)$$

Donc,

$$\theta_{k,i} \leqslant 2^{\nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j} + 1} + 2^{\nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 1} + 2^{\nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 1} + 2^{\mu - L^{\hat{u}}_{k,i} + 1}. \quad (5.25)$$

Ensuite,

$$\theta_{k,i} \leqslant 2^{max(\nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j} + 1\,;\, \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 1\,;\, \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 1\,;\, \mu - L^{\hat{u}}_{k,i} + 1) + 1}. \quad (5.26)$$

Finalement,

$$\mathrm{ufp}(\theta_{k,i}) \leqslant max(\nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}\,;\ \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\ \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\ \mu - L^{\hat{u}}_{k,i}) + 2.$$

■

5.3.2 Ufp de l'Erreur des Fonctions d'Activation

Les erreurs du neurone i de la couche k, dues aux fonctions d'activation $\hat{Sigmoid}$ ou $\hat{Tanh}$ en fixe sont, respectivement, notées $\gamma_{k,i}$ et $\xi_{k,i}$, et présentées dans les propositions 5.2.2 et 5.2.3. Cependant, nous nous intéressons au calcul du bit de poids fort de $\gamma_{k,i}$ et $\xi_{k,i}$, dans les propositions 5.3.2 et 5.3.3, respectivement.

Ufp de l'Erreur du Sigmoïde

La Proposition 5.3.2 a pour but de montrer le calcul de l'ufp de l'erreur due à la fonction d'activation $\hat{Sigmoid}$, appliquée à chaque neurone i de la couche k. En d'autres termes, cette proposition calcule le bit de poids fort de l'erreur $\gamma_{k,i}$, donnée dans l'Équation (5.20).

Proposition 5.3.2. *Soit $\mathrm{ufp}(\gamma_{k,i}) \in \mathbb{Z}$, le bit de poids fort de l'erreur correspondante à la fonction d'activation $\hat{Sigmoid}$, appliquée au neurone i de la couche k. Et soient $\nu, \mu \in \mathbb{Z}$ deux constantes, respectivement définies par*

$\nu = \text{ufp}(3n-2)$ *et* $\mu = \text{ufp}(4n-1)$, $\forall n \geqslant 1$, le nombre de neurones par couche.

Le bit de poids fort de l'erreur $\gamma_{k,i}$ du neurone i de la couche k, noté $\text{ufp}(\gamma_{k,i})$, *est donné par* $\forall\, k,\ 1 \leqslant k \leqslant m,\ \forall\, i, j,\ 0 \leqslant i, j < n$,

$$\text{ufp}(\gamma_{k,i}) \leqslant \begin{cases} \text{ufp}(\theta_{k,i}) & \text{si } I_1, \\ max(M^{\hat{u}}_{k,i} - 4;\ V_{k-1,i,j} - 2;\ Y_{k-1,i,j} - 2;\ Z_{k-1,i,j} - 2;\ H_{k,i} - 2;\ G_{k,i}) + 1 & \text{si } I_2, \\ max(M^{\hat{u}}_{k,i} - 2;\ V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};\ G_{k,i}) + 1 & \text{si } I_3, \\ max(M^{\hat{u}}_{k,i} - 1;\ V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};\ G_{k,i}) + 1 & \text{si } I_4, \\ max(\text{ufp}(\gamma_{k,i});\ G_{k,i}) + 1 & \text{si } I_5. \end{cases} \tag{5.27}$$

où $G_{k,i} = -L^{\hat{u}}_{k,i} + 1$, $H_{k,i} = \mu - L^{\hat{u}}_{k,i}$, $V_{k-1,i,j} = \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}$, $Y_{k-1,i,j} = \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}$ *et* $Z_{k-1,i,j} = \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}$.

Preuve. *Considérons la formule de calcul de l'erreur $\gamma_{k,i}$, définie dans l'Équation (5.20) et calculons son ufp, c'est-à-dire son bit de poids fort. Rappelons que, $M^{\hat{u}}_{k,i}$ est le nombre de bits de la partie entière (bit de poids fort) de $\hat{u}_{k,i}$, et remplaçons dans l'Équation (5.20), $\hat{u}_{k,i}$ par $2^{M^{\hat{u}}_{k,i}}$ et $\theta_{k,i}$ par son expression donnée dans l'Équation (5.26). Nous obtenons*

$$\gamma_{k,i} \leqslant \begin{cases} 0 & \text{si } I_1, \\ 2^{M^{\hat{u}}_{k,i}-4} + 2^{max(V_{k-1,i,j}-2;\ Y_{k-1,i,j}-2;\ Z_{k-1,i,j}-2;\ H_{k,i}-2)} + 2^{G_{k,i}} & \text{si } I_2, \\ 2^{M^{\hat{u}}_{k,i}-2} + 2^{max(V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i})} + 2^{G_{k,i}} & \text{si } I_3, \\ 2^{M^{\hat{u}}_{k,i}-1} + 2^{max(V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i})} + 2^{G_{k,i}} & \text{si } I_4, \\ \gamma_{k,i} + 2^{G_{k,i}} & \text{si } I_5. \end{cases} \tag{5.28}$$

Donc,

$$\gamma_{k,i} \leqslant \begin{cases} 0 & \text{si } I_1, \\ 2^{max(M^{\hat{u}}_{k,i}-4;V_{k-1,i,j}-2;\ Y_{k-1,i,j}-2;\ Z_{k-1,i,j}-2;\ H_{k,i}-2;G_{k,i})+1} & \text{si } I_2, \\ 2^{max(M^{\hat{u}}_{k,i}-2;V_{k-1,i,j};Y_{k-1,i,j};Z_{k-1,i,j};H_{k,i};G_{k,i})+1} & \text{si } I_3, \\ 2^{max(M^{\hat{u}}_{k,i}-1;V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};G_{k,i})+1} & \text{si } I_4, \\ \gamma_{k,i} + 2^{G_{k,i}} & \text{si } I_5. \end{cases} \tag{5.29}$$

Finalement,

$$\text{ufp}(\gamma_{k,i}) \leqslant \begin{cases} \text{ufp}(\theta_{k,i}) & \text{si } I_1, \\ max(M^{\hat{u}}_{k,i} - 4;\ V_{k-1,i,j} - 2;\ Y_{k-1,i,j} - 2;\ Z_{k-1,i,j} - 2;\ H_{k,i} - 2;\ G_{k,i}) + 1 & \text{si } I_2, \\ max(M^{\hat{u}}_{k,i} - 2;\ V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};\ G_{k,i}) + 1 & \text{si } I_3, \\ max(M^{\hat{u}}_{k,i} - 1;\ V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};\ G_{k,i}) + 1 & \text{si } I_4, \\ max(\text{ufp}(\gamma_{k,i});\ G_{k,i}) + 1 & \text{si } I_5. \end{cases}$$

■

Ufp de l'Erreur de la Tangente Hyperbolique

Le calcul du bit de poids fort de l'erreur $\xi_{k,i}$, due à une fonction d'activation $\hat{Tanh}$ en fixe, est présenté dans la Proposition 5.3.3. Rappelons que la formule de l'erreur $\xi_{k,i}$ est donnée dans la Proposition 5.2.3.

Proposition 5.3.3. *Soit* $\text{ufp}(\xi_{k,i}) \in \mathbb{Z}$, *le bit de poids fort de l'erreur correspondante à la fonction d'activation* $\hat{Tanh}$, *appliquée au neurone i de la couche k. Et soient* $\nu, \mu \in \mathbb{Z}$ *deux constantes, telles que*

$$\nu = \text{ufp}(3n-2) \quad \textit{et} \quad \mu = \text{ufp}(4n-1), \quad \forall n \geqslant 1, \text{ le nombre de neurones par couche.}$$

Le bit de poids fort de l'erreur $\xi_{k,i}$ *du neurone i de la couche k, noté* $\text{ufp}(\xi_{k,i})$, *est donné avec* $\forall\, k, 1 \leqslant k \leqslant m, \forall\, i, j, 0 \leqslant i, j < n,$

$$\text{ufp}(\xi_{k,i}) \leqslant \begin{cases} \text{ufp}(\theta_{k,i}) & \text{si } I_1, \\ max(-5;\ M^{\hat{u}}_{k,i} - 4;\ V_{k-1,i,j} - 3;\ Y_{k-1,i,j} - 3;\ Z_{k-1,i,j} - 3;\ H_{k,i} - 3;\ G_{k,i}) + 1 & \text{si } I_2, \\ max(-3;\ M^{\hat{u}}_{k,i} - 2;\ V_{k-1,i,j};\ Y_{k-1,i,j};\ Z_{k-1,i,j};\ H_{k,i};\ G_{k,i}) + 1 & \text{si } I_3, \\ max(M^{\hat{u}}_{k,i} - 1;\ V_{k-1,i,j} - 1;\ Y_{k-1,i,j} - 1;\ Z_{k-1,i,j} - 1;\ H_{k,i} - 1;\ G_{k,i}) + 1 & \text{si } I_4, \\ max(\text{ufp}(\gamma_{k,i});\ G_{k,i} + 1) & \text{si } I_5. \end{cases} \tag{5.30}$$

où $G_{k,i} = -L^{\hat{u}}_{k,i} + 1$, $H_{k,i} = \mu - L^{\hat{u}}_{k,i}$, $V_{k-1,i,j} = \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}$, $Y_{k-1,i,j} = \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}$ *et* $Z_{k-1,i,j} = \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}$.

N.B. *La démonstration du calcul de* $\text{ufp}(\gamma_{k,i})$, *le bit de poids fort de l'erreur commise lors de l'application de la fonction d'activation* $\hat{Tanh}$, *sur le neurone i de la couche k, se fait exactement de la même manière que, pour la fonction d'activation* $\hat{Sigmoid}$ *dans la Proposition 5.3.2.*

Remarque 9. *Dans le pire cas, le bit de poids fort (ufp) de l'erreur des fonctions d'activation* $\hat{Linear}$ *et* $\hat{ReLU}$ *est le même que celui de la somme pondérée des poids synaptiques, défini dans la Proposition 5.3.1.*

5.4 Conclusion

Dans ce chapitre, nous avons défini les erreurs de l'addition, la multiplication et les fonctions mathématiques *sigmoïde*, et *tangente hyperbolique* en fixe, dans la Section 5.1. Ces dernières sont requises dans la Section 5.2, pour calculer dans un RN, l'erreur de la somme pondérée des poids synaptiques $\theta_{k,i}$, et les erreurs des fonctions d'activation en fixe $\gamma_{k,i}$ et $\xi_{k,i}$, correspondantes au $\hat{Sigmoid}$ et à la $\hat{Tanh}$, respectivement. Ensuite, dans la Section 5.3, le bit de poids fort des erreurs $\theta_{k,i}$, $\gamma_{k,i}$ et $\xi_{k,i}$ est calculé. Il est utilisé dans la génération des contraintes dans les chapitres 6 et 7, pour déterminer le format optimal[1] $< M, L >$ de chaque neurone, de chaque entrée et de chaque poids synaptique du RN.

[1] Le plus petit format possible.

Chapitre 6

« La preuve de la valeur d'un système informatique est son existence. »

— Alan Jay Perlis

CubMeth : un Format Fixe par Neurone & par Poids Synaptique

Dans ce travail, notre objectif est de synthétiser un RN fixe, pour un RN flottant donné en entrée, respectant les exigences de l'utilisateur (seuil d'erreur et type de données). Nous savons également que, l'arithmétique fixe présentée dans le Chapitre 2, représente un nombre fixe avec un signe, une valeur entière et un format. Pour utiliser ces RNs dans des systèmes embarqués, nous devons économiser au maximum la mémoire consommée. Pour se faire, nous devons représenter les coefficients fixes avec un nombre de bits minimal, c'est pourquoi, nous présentons dans ce chapitre la méthode CubMeth, basée sur la génération de contraintes automatiquement. Le nombre de ces dernières augmente cubiquement, dans le pire des cas. Cette méthode permet de trouver les formats optimaux (plus petits formats), pour représenter les coefficients fixes, tout en respectant le type de données à ne pas dépasser pour représenter ces coefficients (le plus

grand type de données se trouvant dans le code synthétisé), ainsi qu'un seuil d'erreur à ne pas excéder entre le RN flottant et fixe. Ce chapitre est organisé de la manière suivante :

- Le principe de la méthode CubMeth est présenté dans la Section 6.1.
- La Section 6.2 met en avant les contraintes générées par la somme pondérée des poids de CubMeth, afin de calculer le nombre de bits minimal des parties fractionnaires des sorties des neurones, et des poids synaptiques.
- Les contraintes générées par CubMeth, correspondantes aux fonctions d'activation montrées dans la Section 3.2, sont présentées dans la Section 6.3.
- Le nombre de contraintes générées par CubMeth, ainsi que la complexité de cette méthode, sont calculés dans la Section 6.4.
- La Section 6.5 conclut ce chapitre.

6.1 Principe de CubMeth

La méthode CubMeth a pour but de régler et de calculer les formats fixes des coefficients $\hat{u}$, $\hat{w}$, $\hat{x}$ du RN, par rapport au seuil d'erreur et au type de données à ne pas excéder, tous les deux fixés par l'utilisateur. Cette méthode attribue un format pour chaque neurone, pour chaque poids synaptique de la matrice $\hat{W}$ et pour chaque entrée. CubMeth génère des contraintes pour déterminer le nombre de bits minimal des parties fractionnaires des neurones et des coefficients du RN (Exemple 6.1.1), et effectue une analyse dynamique du RN avec plusieurs vecteurs d'entrées, pour calculer le nombre de bits des parties entières. Cette analyse dynamique permet la sous-approximation des *ranges* (plages) des entrées et sorties des neurones de chaque couche, mais nous prévoyons de calculer une sur-approximation des *ranges* par analyse statique dans le futur.

La Figure 6.1 illustre le principe de répartition des formats par la méthode CubMeth. Le RN présenté dans cette figure se compose d'une couche d'entrée, d'une couche intermédiaire et d'une couche de sortie. Ces couches contiennent trois neurones, et sont entièrement connectées entre elles. CubMeth attribue un format à chaque neurone $\hat{u}$ de chaque couche (quadrillage autour de chaque neurone). Nous constatons également que les arêtes sortantes de chaque neurone, correspondantes aux poids synaptiques $\hat{w}$, sont connectées aux neurones de la couche suivante. Ce qui équivaut à dire qu'un seul format fixe est attribué à chaque poids synaptique de chaque matrice $\hat{W}$. Maintenant que le nombre de bits des parties entières $M^{\hat{u}}$, $M^{\hat{w}}$ et $M^{\hat{x}}$, est calculé via l'analyse dynamique du RN, CubMeth a pour but de trouver le nombre de bits optimal[1] des parties fractionnaires $L^{\hat{u}}$, $L^{\hat{w}}$ de chaque sortie de neurone $\hat{u}$ et de chaque poids synaptique $\hat{w}$, en générant des contraintes. Ces contraintes sont produites en fonction des fonctions d'activation en fixe utilisées $\hat{Linear}$,

[1] Le plus petit L répondant aux exigences de l'utilisateur.

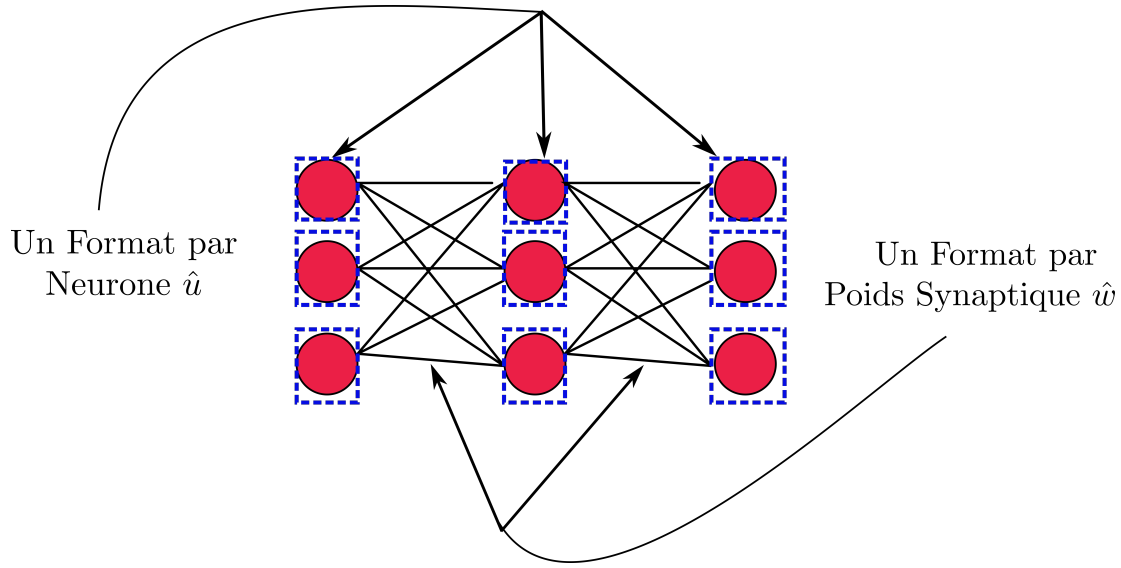

FIGURE 6.1: Attribution et Répartition des Formats dans CubMeth.

ReLU, *Sigmoid* ou *Tanh*, et sont résolues grâce à un solveur (Section 4.1). Dans notre cas, nous avons utilisé le solveur Z3 [dMB08], afin de résoudre les contraintes générées automatiquement par CubMeth.

Pour récapituler, dans la Figure 6.2, CubMeth prend en entrée un RN flottant déjà entraîné dans une certaine précision (simple précision dans notre cas, définie dans la Table 2.1), et synthétise un RN en fixe, ayant le même comportement que le RN initial, en générant un nombre de contraintes avec une complexité cubique dans le pire des cas. Ce RN basé sur l'arithmétique fixe (Section 2.3), respecte un seuil d'erreur à ne pas dépasser pour la couche de sortie, ainsi qu'un nombre de bits maximal à ne pas excéder pour représenter les coefficients. Ils sont tous les deux définis par l'utilisateur et, respectivement notés *Threshold* et T. Le *Threshold* est une valeur entre zéro et un, permettant d'assurer que l'erreur absolue (Équation (2.3)) entre le RN initial et le RN en fixe, ne dépasse pas le seuil défini par l'utilisateur. Et le type de données T peut prendre une valeur quelconque, mais dans ces travaux, nous avons utilisé $T \in \{8, 16, 32\}$ bits pour la compatibilité avec les types de données du langage C [KR88] pour la synthèse de code.

Exemple 6.1.1. *À travers cet exemple, nous présentons le principe de CubMeth. Considérons le RN entièrement connecté de la Figure 6.3, se composant de trois couches ($m = 3$) et de deux neurones par couche ($n = 2$). Fixons le seuil d'erreur, Threshold à $0,01$, et contraignons le type de données T à 32 bits. Notre but est de synthétiser un RN fixe, ayant le même comportement que le RN flottant initial, donné en entrée. Autrement dit, l'erreur commise sur tous les neurones de la couche de sortie (u_{30}, u_{31} dans la Figure 6.3), doit être inférieure à (ou égale à) 0.01, pendant que nous utilisons des nombres entiers respectant le type de donnée imposé ($T \leqslant 32$ bits). Les vecteurs des biais, des matrices de poids synaptiques et le vecteur d'entrées du RN de la Figure 6.3, sont définis par*

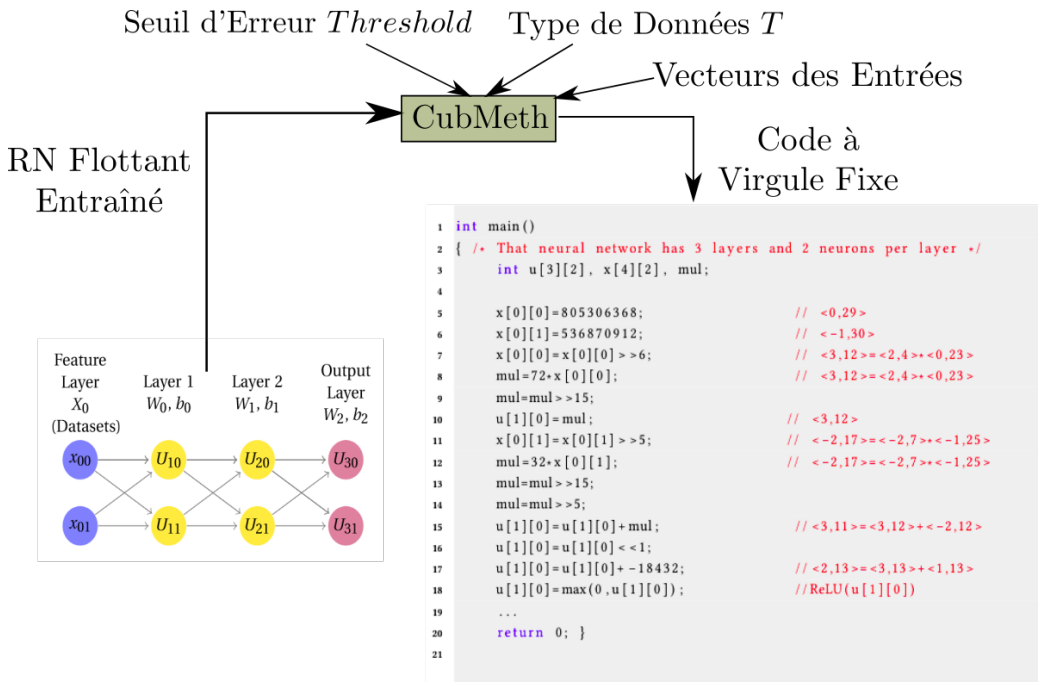

FIGURE 6.2: Entrées et Sorties de `CubMeth`.

$$b_0 = \begin{pmatrix} -2.25 \\ 0.5 \end{pmatrix}, b_1 = \begin{pmatrix} 1.2 \\ 0.5 \end{pmatrix}, b_2 = \begin{pmatrix} 3.1 \\ 1.1 \end{pmatrix}, W_0 = \begin{pmatrix} 4.5 & 0.25 \\ -1.06 & 3.37 \end{pmatrix}, W_1 = \begin{pmatrix} -0.75 & 0.85 \\ 2.1 & 0.48 \end{pmatrix},$$
$$W_2 = \begin{pmatrix} -1.5 & 0.57 \\ 0.21 & -2.61 \end{pmatrix}, X_0 = \begin{pmatrix} 1.5 \\ 0.5 \end{pmatrix}.$$

Notre objectif à cette étape est de calculer les formats minimaux $< M, L >$ des coefficients du RN et des sorties des neurones, respectant les exigences de l'utilisateur. Ensuite, d'évaluer le RN de la Figure 6.3, en calculant l'Équation (5.11) avec une fonction d'activation Linéar (Définition 3.2.7), pour simplifier les calculs. Notons que dans cet exemple, nous utilisons la méthode `CubMeth` pour déterminer le nombre de bits des parties fractionnaires minimal des sorties des neurones et des coefficients du RN. Cette méthode génère des contraintes pour calculer ces formats minimaux et plus précisément le nombre de bits des parties fractionnaires. Elles sont formellement définies dans les sections 6.2 et 6.3. Montrons les contraintes générées par cette méthode pour le neurone u_{31}

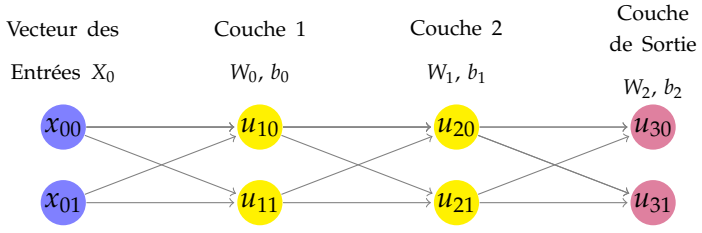

FIGURE 6.3: RN Entièrement Connectée avec 3 Couches et 2 Neurones par Couche.

N.	Rés. Flottant	Rés. Fixe	Erreur Absolue	Type de Données
u_{10}	4.625	$37888_{<3,13>}$ = 4.625	/	int32_t
u_{11}	0.595	$4856_{<-1,13>}$ = 0.5927734375	/	int16_t
u_{20}	-1.763	$-3619_{<1,11>}$ = -1.76708984375	/	int16_t
u_{21}	10.4981	$21492_{<4,11>}$ = 10.494140625	/	int32_t
u_{30}	9.083917	$4648_{<4,9>}$ = 9.078125	$5.79 \times 10^{-3} \approx 2^{-8}$	int16_t
u_{31}	-26.300041	$-13461_{<5,9>}$ = -26.291015625	$9.02 \times 10^{-3} \approx 2^{-7}$	int32_t

TABLE 6.1: Comparaison entre les Résultats Flottants et les Résultats Fixes du RN de la Figure 6.3 pour un *Threshold* = 0.01 et un Type de Données $T \leqslant 32$ bits.

de la Figure 6.3, telles que

$$C = \left\{ \begin{array}{ccc} L^u_{31} \geqslant 7, & L^u_{31} + M^u_{31} \leqslant 31, & L^u_{31} \leqslant L^x_{21}, \\ L^w_{210} + M^w_{210} \leqslant 31, & L^w_{211} + M^w_{211} \leqslant 31, & L^x_{20},\ L^x_{21},\ L^w_{210},\ L^w_{211},\ L^u_{31} \geqslant 0, \\ L^u_{31} \leqslant L^x_{20}, & L^w_{210} + L^x_{20} + M^w_{210} + M^x_{20} \leqslant 63, & L^w_{211} + L^x_{21} + M^w_{211} + M^x_{21} \leqslant 63, \\ L^x_{20} - L^u_{31} \geqslant 6 + M^w_{210}, & L^x_{21} - L^u_{31} \geqslant 6 + M^w_{211}, & L^w_{210} - L^u_{31} \geqslant 6 + M^x_{20}, \\ L^w_{211} - L^u_{31} \geqslant 6 + M^x_{21}, & L^w_{210} + L^x_{20} - L^u_{31} \geqslant 6, & L^w_{211} + L^x_{21} - L^u_{31} \geqslant 6, \\ L^u_{21} - L^u_{31} \geqslant 9. & & \end{array} \right\} \tag{6.1}$$

Dans l'Équation (6.1), le nombre de bits des parties entières de u_{31}, x_{20}, x_{21}, w_{210} et w_{211} est M^u_{31}, M^x_{20}, M^x_{21}, M^w_{210} et M^w_{211}, respectivement, et leurs nombre de bits des parties fractionnaires est L^u_{31}, L^x_{20}, L^x_{21}, L^w_{210} et L^w_{211}, respectivement . Par exemple, la contrainte $L^u_{31} \geqslant 7$, donne une borne inférieure pour L^u_{31} indiquant que, la sortie u_{31} doit respecter le seuil d'erreur fixé par l'utilisateur. Après avoir résolu ces contraintes et obtenu les formats minimaux des coefficients et des sorties (formats dans la troisième colonne de la Table 6.1), nous passons à l'évaluation du RN.

La Table 6.1 montre les résultats obtenus pour le RN de la Figure 6.3. La première colonne donne les noms des neurones. La deuxième colonne présente les résultats des sorties des neurones, calculés par l'Équation (5.10), en utilisant l'arithmétique flottante avec simple précision (Table 2.1). La troisième colonne montre les résultats de l'évaluation en virgule fixe de l'Équation (5.11). Enfin, dans la quatrième colonne, nous calculons l'erreur absolue (Définition 2.1.3) entre les résultats en virgule flottante et les résultats en virgule fixe. La dernière colonne montre le plus petit type de données attribuable à chaque neurone dans le code à virgule fixe synthétisé. Le Threshold défini par l'utilisateur est 0,01 *(7 bits significatifs dans les parties fractionnaires, au minimum, pour les sorties de la dernière couche). Comme nous pouvons le constater, la pire erreur absolue de la couche de sortie dans la Table 6.1, est inférieure à la valeur du seuil d'erreur requis. Au départ, tous les neurones sont représentés sur* 32 *bits, mais des types de données plus petits peuvent leur être assignés, merci aux contraintes générées. Dans cet exemple, trois neurones sur six peuvent être représentés avec* 16 *bits au lieu de* 32 *bits, ce qui permet de réduire la taille de la mémoire utilisée de* 25%.

La Figure 6.4 montre quelques lignes de code synthétisé par notre outil `SyFix` *(Chapitre 8), pour les neurones u_{10} et u_{11} de la Table 6.1 et de la Figure 6.3. Ce code calcule les résultats présentés*

```
1  int main()
2    { /* That neural network has 3 layers and 2 neurons per layer */
3         int32_t u10, x10;  int16_t u11, x11;
4         int16_t u20, x20;  int32_t u21, x21;
5         int16_t u30, x30;  int32_t u31, x31;
6         int mul,x0[2];
7        x0[0]=805306368;              // <1,29>
8        x0[1]=536870912;              // <-1,30>
9        x0[0]=x0[0]>>6;               // <1,23>
10       mul=72*x0[0];                 // <4,27>=<2,4>*<1,23>
11       mul=mul>>15;                  // <4,12>
12       u10=mul;                       // <4,12>
13       x0[1]=x0[1]>>5;               // <-1,25>
14       mul=32*x0[1];                 // <-2,32>=<-2,7>*<-1,25>
15       mul=mul>>15;                  // <-2,17>
16       mul=mul>>5;                   // <-2,12>
17       u10=u10+mul;                    // <4,12>=<4,12>+<-2,12>
18       u10=u10<<1;                    // <3,12>
19       u10=u10+-18432;                // <3,13>=<3,12>+<1,13>
20       x10=u10;                      // <3,13>
21       x0[0]=x0[0]>>3;               // <1,20>;
22       mul=-543*x0[0];              // <1,29>=<0,9>*<1,20>
23       mul=mul>>18;                  // <1,11>
24       u11=mul;                       // <1,11>
25       x0[1]=x0[1]>>3;               // <-1,22>
26       mul=431*x0[1];                  // <1,29>=<1,7>*<-1,22>
27       mul=mul>>18;                  // <1,11>
28       u11=u11+mul;                   // <1,11>=<1,11>+<1,11>
29       u11=u11<<2;                   // <-1,11>
30       u11=u11+4096;                  // <-1,13>=<-1,11>+<-1,13>
31       ...
32       return 0; }
33
```

FIGURE 6.4: Code à Virgule Fixe Synthétisé pour les Neurones u_{10} et u_{11} de la Figure 6.3 pour un *Threshold* $= 0.01$ et un Type de Données $T \leqslant 32$ bits.

dans la troisième colonne de la Table 6.1, et utilise les types de données de la cinquième colonne. Par exemple, la ligne 7 représente l'entrée $x_{00} = 1,5$*, dans la représentation en virgule fixe. Cette valeur est décalée vers la droite de 6 positions (ligne 9), afin d'être alignée et utilisée dans la multiplication par* $4,5$ *(Ligne 10), représentée par 72 dans l'arithmétique fixe. Le résultat à virgule fixe* $u_{10} = 37888$ *dans la Table 6.1, est renvoyé par la ligne 19 et utilise int32_t comme type de données.*

6.2 Modélisation des Contraintes de la Somme Pondérée des Poids

Cette section met en avant les contraintes générées par CubMeth, pour calculer le nombre de bits minimal de la partie fractionnaire de chaque sortie de neurone ainsi que, le nombre de bits minimal de la partie fractionnaire des poids synaptiques. Ces deux dernières représentent les variables du système linéaire à optimiser (minimiser, voir Section 4.1). Nous ne mentionnons pas le nombre de bits de la partie fractionnaire du biais comme variable du système ici, car le format de la sortie $\hat{u}_{k,i}$, lui sera attribué. Dans tous les cas, le biais sera aligné avec le format exigé en sortie du neurone. Dans ce manuscrit, notons que $< M^{\hat{var}}_{k,i}, L^{\hat{var}}_{k,i} >$ est le format fixe de $\hat{var}_{k,i} \in \{\hat{x}_{k,i}, \hat{u}_{k,i}\}$, où $M^{\hat{var}}_{k,i}$ est le nombre de bits de la partie entière de $\hat{var}_{k,i}$, et $L^{\hat{var}}_{k,i}$ le nombre de bits de sa partie fractionnaire. Le format fixe des coefficients des poids synaptiques $\hat{w}_{k,i,j}$ est $< M^{\hat{w}}_{k,i,j}, L^{\hat{w}}_{k,i,j} >$. Le nombre de bits de la partie entière est $M^{\hat{w}}_{k,i,j}$ et le nombre de bits de la partie fractionnaire est $L^{\hat{w}}_{k,i,j}$. L'indice k correspond à la couche k du RN. L'indice i correspond à la ligne i de la matrice des poids synaptiques de la couche k, et l'indice j représente la colonne j de la ligne i de la matrice k du RN. La génération de ces contraintes consiste en cinq points : l'assertion du type de données requis, la positivité du nombre de bits des parties fractionnaires, la condition aux limites, la propagation des calculs vers l'avant et vers l'arrière, et la borne sur l'erreur de la somme pondérée des poids synaptiques.

Assertion du Type de Données Requis

Le type de données requis, T, est fixé par l'utilisateur. Il représente le nombre de bits maximal à ne pas dépasser dans la représentation fixe. Il représente également le type de données maximal qu'on peut trouver dans le code synthétisé du RN fixe. Les contraintes définies dans les équations (6.2), (6.3) et (6.4), affirment respectivement que, les coefficients en fixe $\hat{x}$, $\hat{u}$, $\hat{w}$ dans l'Équation (5.11), ne doivent pas dépasser le type de données $T \in \{8, 16, 32\}$ bits, défini par l'utilisateur. Ce dernier est égal au nombre de bits maximal pour représenter les coefficients des poids synaptiques, des entrées du RN et des neurones. Ces contraintes sont définies par

$$M^{\hat{x}}_{k,i} + L^{\hat{x}}_{k,i} \leqslant T - 1, \qquad 0 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.2}$$

$$M^{\hat{u}}_{k,i} + L^{\hat{u}}_{k,i} \leqslant T - 1, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.3}$$

$$M^{\hat{w}}_{k,i,j} + L^{\hat{w}}_{k,i,j} \leqslant T - 1, \qquad 0 \leqslant k < m,\ 0 \leqslant i, j < n, \tag{6.4}$$

Dans les équations (6.2), (6.3) et (6.4), la somme des longueurs de la partie entière et fractionnaire, $M^{\hat{x}}_{k,i} + L^{\hat{x}}_{k,i}$ (respectivement $M^{\hat{u}}_{k,i} + L^{\hat{u}}_{k,i}$ et $M^{\hat{w}}_{k,i,j} + L^{\hat{w}}_{k,i,j}$), ne doit pas excéder la longueur du type de données T (défini par l'utilisateur), pour ne pas engendrer un *overflow*. Notons que dans ces équations, nous utilisons $T - 1$, car nous réservons un bit pour le

signe. Par exemple, dans l'Équation (6.2), si nous considérons $T = 32$ bits et $M^{\hat{x}}_{k,i} = 4$, alors la valeur maximale que peut prendre $L^{\hat{x}}_{k,i}$ est 27, c'est-à-dire 27 bits après la virgule fixe. La contrainte définie dans l'Équation (6.5) concerne la multiplication en fixe $\hat{w}_{k,i,j} \otimes \hat{x}_{k,j}$, se trouvant dans l'Équation (5.11). Elle est donnée avec

$$M^{\hat{w}}_{k,i,j} + L^{\hat{w}}_{k,i,j} + M^{\hat{x}}_{k,j} + L^{\hat{x}}_{k,j} \leqslant 2T - 1, \qquad 0 \leqslant k < m,\, 0 \leqslant i, j < n. \tag{6.5}$$

L'Équation (6.5) permet d'assurer que le nombre de bits total utilisé pendant la multiplication en fixe, n'excède pas le double du nombre de bits du type de données requis T. Nous avons choisi d'utiliser le double de T dans les calculs intermédiaires dans la multiplication, pour avoir plus de bits corrects en sortie. Notons que dans l'Équation (6.5), nous utilisons $2T - 1$, car nous réservons un bit pour le signe.

Nombre de Bits des Parties Fractionnaires

Dans ce travail, nous considérons le nombre de bits des parties fractionnaires positif ou nul, c'est-à-dire que nous pouvons avoir au minimum zéro (ou plus) bits corrects après la virgule fixe. Les contraintes correspondantes aux équations (6.6), (6.7) et (6.8) assurent respectivement que, le nombre de bits des parties fractionnaires $L^{\hat{x}}_{k,i}$, $L^{\hat{u}}_{k,i}$ et $L^{\hat{w}}_{k,i,j}$, ne peut être négatif. Elles sont données par

$$L^{\hat{x}}_{k,i} \geqslant 0, \qquad 0 \leqslant k \leqslant m,\, 0 \leqslant i < n, \tag{6.6}$$

$$L^{\hat{u}}_{k,i} \geqslant 0, \qquad 1 \leqslant k \leqslant m,\, 0 \leqslant i < n, \tag{6.7}$$

$$L^{\hat{w}}_{k,i,j} \geqslant 0, \qquad 0 \leqslant k < m,\, 0 \leqslant i, j < n. \tag{6.8}$$

Condition aux Limites

La condition aux limites pour les neurones de la couche de sortie est modélisée dans la contrainte de l'Équation (6.9). Cette dernière permet de satisfaire le seuil d'erreur à ne pas dépasser pour la couche de sortie du RN, c'est-à-dire que l'erreur (absolue) commise entre les sorties du RN flottant et du RN fixe, est inférieure ou égale à ce seuil d'erreur. Ce dernier est défini par l'utilisateur, et est dit *Threshold*. Cette contrainte est donnée avec

$$L^{\hat{u}}_{m,i} \geqslant \left|\mathrm{ufp}(\mathit{Threshold})\right|, \qquad 0 \leqslant i < n,\, m : \text{ dernière couche du RN}. \tag{6.9}$$

L'Équation (6.9) donne une borne inférieure à $L^{\hat{u}}_{m,i}$, correspondant au nombre de bits de la partie fractionnaire des neurones de la dernière couche m du RN. Dans cette équation, ufp(*Threshold*), correspond au bit de poids fort du seuil d'erreur défini par l'utilisateur. Il est calculé en utilisant l'Équation (2.1). Cet ufp donne le nombre de bits corrects (minimal) des parties fractionnaires des neurones de la couche de sortie, pour satisfaire le seuil d'erreur exigé.

Propagation des Calculs vers l'Avant et vers l'Arrière

Cette partie concerne la propagation des calculs en avant et en arrière, afin de déterminer le nombre de bits des parties fractionnaires des neurones, dans les deux sens. L'Équation (6.10) présente la contrainte, où la propagation du calcul du nombre de bits de la partie fractionnaire se fait vers l'avant, c'est-à-dire de la première couche du RN vers la dernière couche. Cette propagation est dite *forward propagation*. Et l'Équation (6.11) montre la contrainte, où la propagation du calcul du nombre de bits de la partie fractionnaire se fait vers l'arrière, en allant de la dernière couche du RN vers la première couche. Cette propagation est dite *backward propagtion*. Ces contraintes bornent le nombre de bits des parties fractionnaires, et sont données avec

$$\forall j : \; L^{\hat{u}}_{k,i} \leqslant L^{\hat{x}}_{k-1,j} \, , \qquad 1 \leqslant k \leqslant m, \, 0 \leqslant i, j < n, \tag{6.10}$$

$$L^{\hat{x}}_{k,i} \leqslant L^{\hat{u}}_{k,i} \, , \qquad 1 \leqslant k \leqslant m, \, 0 \leqslant i < n. \tag{6.11}$$

La contrainte de l'Équation (6.10) a pour but de donner une borne supérieure pour $L^{\hat{u}}_{k,i}$. Elle assure que, le nombre de bits de la partie fractionnaire $L^{\hat{u}}_{k,i}$ de la sortie du neurone i de la couche k, soit inférieur (ou égal) à toutes ses entrées, $\forall j$, $0 \leqslant j < n$, $L^{\hat{x}}_{k-1,j}$. La contrainte de l'Équation (6.11) donne une borne supérieure pour $L^{\hat{x}}_{k,i}$. Cette contrainte garantit que, le nombre de bits de la partie fractionnaire $L^{\hat{x}}_{k,i}$, de l'entrée $\hat{x}$ du neurone i de la couche suivante, est inférieur (ou égal) au nombre de bits de la partie fractionnaire $L^{\hat{u}}_{k,i}$, de la sortie du neurone i de la couche courante.

Borne sur l'Erreur de la Somme Pondérée des Poids Synaptiques

Dans cette sous-section, nous donnons les contraintes correspondantes aux erreurs de calcul de la somme pondérée des poids (Proposition 5.2.1). Les contraintes définies dans les équations (6.12) , (6.13), (6.14) et (6.15), visent à borner le nombre de bits de la partie fractionnaire $L^{\hat{u}}_{k,i}$, de la sortie $\hat{u}_{k,i}$ de chaque neurone i de la couche k, ainsi que le nombre de bits de la partie fractionnaire $L^{\hat{w}}_{k-1,i,j}$, de chaque poids synaptique $\hat{w}_{k-1,i,j}$, appartenant à la matrice des poids $\hat{W}_{k-1}$. Elles sont obtenues via le calcul du bit de poids fort de l'erreur commise lors du calcul de la somme pondérée des poids synaptiques. Ce bit de poids fort, dit ufp, est donné dans la Proposition 5.3.1. Notons que, CubMeth attribue un format par neurone et un format par poids synaptique, et génère les contraintes suivantes

$$\forall j : \; L^{\hat{x}}_{k-1,j} - L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{w}}_{k-1,i,j} + 2, \qquad 1 \leqslant k \leqslant m, \, 0 \leqslant i, j < n. \tag{6.12}$$

$$\forall j : \; L^{\hat{w}}_{k-1,i,j} - L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{x}}_{k-1,j} + 2, \qquad 1 \leqslant k \leqslant m, \, 0 \leqslant i, j < n. \tag{6.13}$$

$$\forall j : \; L^{\hat{w}}_{k-1,i,j} + L^{\hat{x}}_{k-1,j} - L^{\hat{u}}_{k,i} \geqslant \nu + 2, \qquad 1 \leqslant k \leqslant m, \, 0 \leqslant i, j < n. \tag{6.14}$$

$$L^{\hat{u}}_{k-1,i} - L^{\hat{u}}_{k,i} \geqslant \mu + 2, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n. \tag{6.15}$$

avec $\nu = \text{ufp}(3n-2)$ et $\mu = \text{ufp}(4n-1)$, $\forall n \geqslant 1$.
Soulignons que, chaque contrainte correspond à un élément de la fonction *max* dans l'Équation (5.22), donnée avec $\forall\, k,\ 1 \leqslant k \leqslant m,\ \forall\, i,j,\ 0 \leqslant i,j < n$,

$$\text{ufp}(\theta_{k,i}) \leqslant max(\nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\ \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j}\,;\ \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j}\,;\ \mu - L^{\hat{u}}_{k,i}) + 2.$$

Par exemple, la contrainte de l'Équation (6.12) est obtenue de la manière suivante

$$L^{\hat{u}}_{k,i} \leqslant -(\nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 2),$$

Notons que, $L^{\hat{u}}_{k,i}$ est positif, $0 \leqslant \theta_{k,i} < 1$ et $\text{ufp}(\theta_{k,i}) \leqslant 0$, c'est pourquoi nous avons utilisé le signe "$-$" devant $\nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + 2$.
Donc,

$$L^{\hat{u}}_{k,i} - L^{\hat{x}}_{k-1,i} \leqslant -(\nu + M^{\hat{w}}_{k-1,i,j} + 2),$$

Finalement,

$$L^{\hat{x}}_{k-1,i} - L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{w}}_{k-1,i,j} + 2. \qquad \blacksquare$$

Les équations (6.13), (6.14) et (6.15) sont obtenues de la même manière.

6.3 Modélisation des Contraintes des Fonctions d'Activation

Dans cette section, nous définissons les contraintes générées automatiquement par `CubMeth` quand l'une de ces fonctions d'activation en fixe, $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ ou $\hat{Tanh}$, est sélectionnée. Ces contraintes ont pour but de calculer le nombre de bits des parties fractionnaires des neurones et des poids synaptiques.

6.3.1 Fonctions d'Activation Linéaire et Unité Linéaire Rectifiée

Les contraintes générées suite à une fonction d'activation en fixe $\hat{Linear}$ ou $\hat{ReLU}$ (Chapitre 3), sont présentées dans la Figure 6.5. Les contraintes générées pour ces deux fonctions d'activation, correspondent à celles générées par la somme pondérée des poids, car la fonction $\hat{Linear}$, définie dans la Définition 3.2.7, renvoie l'identité, ce qui revient à calculer la somme pondérée des poids synaptiques. Et la fonction $\hat{ReLU}$, définie dans la Définition 3.2.8, renvoie l'identité si la valeur donnée en entrée est positive et zéro sinon. Cette fonction, génère une erreur égale à celle de la somme pondérée des poids dans le pire des cas, c'est-à-dire quand la valeur donnée en entrée est positive. Dans la Figure 6.5, les équations (6.2), (6.3), (6.4) et (6.5), permettent d'assurer l'assertion du type de données T requis par l'utilisateur, c'est-à-dire que le nombre total de bits utilisés pour représenter les coefficients en fixe $\hat{u}$, $\hat{w}$ et $\hat{x}$, ne doit pas excéder T, et le nombre total de bits pour effectuer une multiplication en fixe, doit tenir sur $2T$ (bit du signe inclus). Les équations (6.6), (6.7)

$$M^{\hat{x}}_{k,i} + L^{\hat{x}}_{k,i} \leqslant T - 1, \qquad 0 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.2}$$

$$M^{\hat{u}}_{k,i} + L^{\hat{u}}_{k,i} \leqslant T - 1, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.3}$$

$$M^{\hat{w}}_{k,i,j} + L^{\hat{w}}_{k,i,j} \leqslant T - 1, \qquad 0 \leqslant k < m,\ 0 \leqslant i,j < n, \tag{6.4}$$

$$M^{\hat{w}}_{k,i,j} + L^{\hat{w}}_{k,i,j} + M^{\hat{x}}_{k,j} + L^{\hat{x}}_{k,j} \leqslant 2T - 1, \qquad 0 \leqslant k < m,\ 0 \leqslant i,j < n. \tag{6.5}$$

$$L^{\hat{x}}_{k,i} \geqslant 0, \qquad 0 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.6}$$

$$L^{\hat{u}}_{k,i} \geqslant 0, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.7}$$

$$L^{\hat{w}}_{k,i,j} \geqslant 0, \qquad 0 \leqslant k < m,\ 0 \leqslant i,j < n, \tag{6.8}$$

$$L^{\hat{u}}_{m,i} \geqslant |\mathrm{ufp}(\mathit{Threshold})|, \qquad 0 \leqslant i < n,\ m : \text{ dernière couche du RN}, \tag{6.9}$$

$$\forall j :\ L^{\hat{u}}_{k,i} \leqslant L^{\hat{x}}_{k-1,j}, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i,j < n, \tag{6.10}$$

$$L^{\hat{x}}_{k,i} \leqslant L^{\hat{u}}_{k,i}, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.11}$$

$$\forall j :\ L^{\hat{x}}_{k-1,j} - L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{w}}_{k-1,i,j} + 2, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i,j < n, \tag{6.16}$$

$$\forall j :\ L^{\hat{w}}_{k-1,i,j} - L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{x}}_{k-1,j} + 2, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i,j < n, \tag{6.17}$$

$$\forall j :\ L^{\hat{w}}_{k-1,i,j} + L^{\hat{x}}_{k-1,j} - L^{\hat{u}}_{k,i} \geqslant \nu + 2, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i,j < n, \tag{6.18}$$

$$L^{\hat{u}}_{k-1,i} - L^{\hat{u}}_{k,i} \geqslant \mu + 2, \qquad 1 \leqslant k \leqslant m,\ 0 \leqslant i < n, \tag{6.19}$$

FIGURE 6.5: Contraintes Générées par `CubMeth` pour le Calcul du Nombre de Bits L Minimal des Coefficients du RN Suite à une Fonction d'Activation en Fixe $\hat{Linear}$ ou $\hat{ReLU}$.

et (6.8) permettent d'assurer que, le nombre de bits des parties fractionnaires de $\hat{u}$, $\hat{w}$ et $\hat{x}$, soit positif. L'Équation (6.9) concerne le respect du seuil d'erreur *Threshold*, exigé par l'utilisateur, pour les neurones de la couche de sortie. La propagation des calculs vers l'avant et vers l'arrière est modélisée dans les équations (6.10) et (6.11). Et enfin, les équations (6.12), (6.13),(6.14) et (6.15) sont obtenues via le calcul du bit de poids fort de l'erreur de calcul de la somme pondérée des poids, défini dans la Proposition 5.3.1.

6.3.2 Fonction d'Activation Sigmoïde

Les contraintes générées suite à une fonction d'activation $\hat{Sigmoid}$, avec une approximation linéaire par morceaux, définie dans la Définition 3.2.9, sont présentées fans la Figure 6.6. L'approximation linéaire par morceaux du $\hat{Sigmoid}$ contient cinq morceaux, c'est-à-dire cinq intervalles de valeurs. Rappelons que ces intervalles I_1, I_2, I_3, I_4 et I_5 sont définis précédemment dans la Section 5.2 par

- $I_1 : \hat{u}_{k,i} \geqslant 5_{<3,0>}$,
- $I_2 : 19_{<2,3>} \leqslant \hat{u}_{k,i} < 5_{<3,0>}$,
- $I_3 : 1_{<1,0>} \leqslant \hat{u}_{k,i} < 19_{<2,3>}$,
- $I_4 : 0_{<1,0>} \leqslant \hat{u}_{k,i} < 1_{<1,0>}$,
- $I_5 : \hat{u}_{k,i} < 0_{<1,0>}$.

où $\hat{u}_{k,i}$ représente la sortie en fixe du neurone i de la couche k, définie dans l'Équation (5.11). Notons que, les contraintes correspondantes à la somme pondérée des poids synaptiques, illustrées dans la Figure 6.5, sont générées pour I_1, I_2, I_3, I_4 et I_5, car nous calculons cette somme pondérée quelque soit la fonction d'activation et quelque soit l'intervalle concerné. Les contraintes dans les équations (6.20), (6.21), (6.22), (6.23), (6.24) et (6.25), sont obtenues via le calcul du bit de poids fort de l'erreur commise, suite à une fonction d'activation $\hat{Sigmoid}$ avec approximation linéaire par morceaux, défini dans la Proposition 5.3.2. Ces contraintes concernent I_2, I_3, I_4 et I_5. Dans le cas où,

- I_2 est activé, les deux paramètres ψ et λ prennent respectivement les valeurs -3 et 1 ;
- I_3 est activé, les valeurs de ψ et λ sont -1 et 1 ;
- I_4 est activé, ψ et λ valent 0 et 1, respectivement ;
- Par contre, dans le cas où, I_5 est activé, cela se traduit par un signe, $s^{\hat{u}_{k,i}} = 1$, c'est-à-dire que la valeur de l'entrée en fixe $\hat{u}_{k,i}$ du $\hat{Sigmoid}$, est négative. Il faut donc considérer la valeur absolue de $\hat{u}_{k,i}$ et ce, en mettant, $s^{\hat{u}_{k,i}} = 0$. Ensuite, il faut vérifier si la valeur absolue de $\hat{u}_{k,i}$ appartient à I_1, I_2, I_3 ou I_4, puis activer les contraintes correspondantes.

- Si I_1, Activer les contraintes de la Figure 6.5.
- Sinon, $\forall k, i, j,\ 1 \leqslant k \leqslant m,\ 0 \leqslant i, j < n,\ \ \psi \in \{-3, -2, -1, 0, 1\}\,,\ \lambda \in \{-1, 0, 1, 2\}$,

$$L^{\hat{u}}_{k,i} \geqslant M^{\hat{u}}_{k,i} + \psi, \tag{6.20}$$

$$\forall j: \quad L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j} + \lambda, \tag{6.21}$$

$$\forall j: \quad L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + \lambda, \tag{6.22}$$

$$\forall j: \quad L^{\hat{u}}_{k,i} \geqslant \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + \lambda, \tag{6.23}$$

$$L^{\hat{u}}_{k,i} \geqslant \mu - L^{\hat{u}}_{k,i} + \lambda, \tag{6.24}$$

$$L^{\hat{u}}_{k,i} \geqslant -L^{\hat{u}}_{k,i} + 2, \tag{6.25}$$

Activer les contraintes de la Figure 6.5.

  - Si I_2,
$$\psi = -3 \qquad \text{et} \qquad \lambda = -1,$$
  - Si I_3,
$$\psi = -1 \quad \text{et} \qquad \lambda = 1,$$
  - Si I_4,
$$\psi = 0 \qquad \text{et} \qquad \lambda = 1,$$
  - Si I_5 , il faut considérer la valeur absolue de $\hat{u}_{k,i}$ et tester
    - Si I_1,

      Activer les contraintes de la Figure 6.5.
    - Si I_2,
$$\psi = -2 \qquad \text{et} \qquad \lambda = 0,$$
    - Si I_3,
$$\psi = 0 \qquad \text{et} \qquad \lambda = 2,$$
    - Si I_4,
$$\psi = 1 \qquad \text{et} \qquad \lambda = 2,$$

FIGURE 6.6: Contraintes Générées par CubMeth pour le Calcul du Nombre de Bits L Minimal des Coefficients du RN Suite à une Fonction d'Activation $\hat{Sigmoid}$ en Fixe.

Soulignons que les contraintes de plusieurs (ou tous) intervalles peuvent être activées simultanément, si nous considérons $I^{\hat{u}}_{k,i}$, l'intervalle où appartient $\hat{u}_{k,i}$, tel que

$$I^{\hat{u}}_{k,i} = \bigcup_{j=1}^{h} I_j \,, \qquad \text{avec } h \in \{1,2,3,4\}. \tag{6.26}$$

6.3.3 Fonctions d'Activation Tangente Hyperbolique

Les contraintes générées suite à une fonction d'activation $\hat{Tanh}$, avec une approximation linéaire par morceaux, définie dans la Définition 3.2.10, sont présentées dans la Figure 6.7. Cette approximation linéaire par morceaux contient cinq morceaux, c'est-à-dire cinq intervalles de valeurs, notés I_1, I_2, I_3, I_4 et I_5. Ces derniers sont les mêmes que ceux utilisés précédemment dans le $\hat{Sigmoid}$ (Sous-Section 6.3.2). Notons que, les contraintes correspondantes à la somme pondérée de poids synaptiques, illustrées dans la Figure 6.5, sont générées pour I_1, I_2, I_3, I_4 et I_5, car nous calculons cette somme pondérée quelque soit la fonction d'activation et quelque soit l'intervalle concerné. Les contraintes dans les équations (6.27), (6.28), (6.29), (6.30), (6.31), (6.32) et (6.33), sont obtenues via le calcul du bit de poids fort de l'erreur commise, suite à une fonction d'activation $\hat{Tanh}$, défini dans la Proposition 5.3.3. Ces contraintes concernent I_2, I_3, I_4 et I_5, sauf la contrainte de l'Équation (6.33), ne concernant pas I_4. Dans le cas où,

- I_2 est activé, les trois paramètres ρ, ψ et λ valent respectivement -4, -3 et -2 ;
- I_3 est activé, les valeurs de ρ, ψ et λ sont -3, -1 et 1 ;
- I_4 est activé, ψ et λ valent tous les deux 0, et ρ ne prend aucune valeur, car la contrainte de l'Équation (6.33), n'est pas activée (ne concerne pas I_4) ;
- Par contre, dans le cas où, I_5 est activé, cela se traduit par un signe négatif ($s^{\hat{u}_{k,i}} = 1$) de la valeur de l'entrée en fixe $\hat{u}_{k,i}$ de la $\hat{Tanh}$. Il faut donc considérer la valeur absolue de $\hat{u}_{k,i}$ et ce, en mettant, $s^{\hat{u}_{k,i}} = 0$. Ensuite, il faut vérifier si la valeur absolue de $\hat{u}_{k,i}$ appartient à I_1, I_2, I_3 ou I_4, puis activer les contraintes correspondantes. Soulignons également que, les contraintes de plusieurs (ou tous) intervalles peuvent être activées simultanément pour la $\hat{Tanh}$, comme le montre l'Équation (6.26).

6.4 Nombre de Contraintes Générées par Couche

Dans cette section, nous nous intéressons au nombre de contraintes générées par couche, dans le pire cas par `CubMeth`, ainsi que leur complexité. Les propositions 6.4.1, 6.4.2 et 6.4.3 donnent le nombre de contraintes générées par couche, suite à une fonction d'activation $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ ou $\hat{Tanh}$, dans le meilleur et dans le pire cas.

Proposition 6.4.1. *Le nombre de contraintes générées par couche, suite à une fonction d'activation Lin̂ear ou ReL̂U, est défini par*

$$nbr = 7n^2 + 7n, \tag{6.34}$$

où $nbr \in \mathbb{N}$, est le nombre de contraintes générées et $n \in \mathbb{N}$, est le nombre de neurones par couche.

Preuve. *Les équations (6.2), (6.3), (6.6), (6.7), (6.9), (6.11) et (6.19), génèrent chacune n contraintes pour chaque couche k, ayant Lin̂ear ou ReL̂U comme fonction d'activation. Et les équations (6.4), (6.5), (6.8), (6.10), (6.16), (6.17) et (6.18), génèrent chacune n^2 contraintes. En additionnant le nombre de toutes ces contraintes générées, nous obtenons*

$$nbr = 7n^2 + 7n.$$ ■

Proposition 6.4.2. *Soient $n, nbr \in \mathbb{N}$, tels que, n est le nombre de neurones par couche, et nbr est le nombre de contraintes générées par couche. Le nombre de contraintes générées par le Sig̃moid dans le meilleur des cas, est donné avec*

$$nbr = 7n^2 + 7n. \tag{6.35}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 25n^2 + 25n. \tag{6.36}$$

Preuve. *Dans le meilleur des cas, c'est-à-dire quand I_1 ou I_5 avec des valeurs dans I_1 uniquement sont activés, le nombre de contraintes générées est égal à celui de la somme pondérée des poids synaptiques, d'un ReL̂U ou d'un Lin̂ear. Et dans le pire des cas, les intervalles I_1, I_2, I_3, I_4, et I_5 avec des valeurs dans I_1, I_2, I_3, I_4, sont activés. Et donc, en activant tous les intervalles de la Figure 6.6, nous obtenons*

$$nbr = 7n^2 + 7n + 6 \times (3n^2 + 3n) = 25n^2 + 25n.$$ ■

Proposition 6.4.3. *Soient $n, nbr \in \mathbb{N}$, tels que, n est le nombre de neurones par couche, et nbr est le nombre de contraintes générées par couche. Le nombre de contraintes générées suite à une fonction d'activation Tân̂h dans le meilleur des cas, est donné avec*

$$nbr = 7n^2 + 7n. \tag{6.37}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 25n^2 + 29n. \tag{6.38}$$

N.B. *La preuve concernant le calcul du nombre de contraintes générées par la Tân̂h, se fait exactement de la même manière que celle du Sig̃moid.*

Afin de calculer le nombre total de contraintes générées par un RN, il suffit d'additionner le nombre de contraintes générées par l'ensemble des couches du RN. Pour obtenir le

nombre de contraintes générées suite à une fonction d'activation $\hat{Linear}$ ou $\hat{ReLU}$ (respectivement $\hat{Sigmoid}$ et $\hat{Tanh}$), il faut multiplier le nombre de couches utilisant cette fonction d'activation par le nombre de contraintes générées pour une seule couche. Le nombre de contraintes générées par `CubMeth`, afin de calculer le nombre de bits minimal des parties fractionnaires du RN, est de complexité cubique $O(n) = n^3$, dans le pire des cas, c'est-à-dire quand le nombre de couches (m) est égal au nombre de neurones (n). Notons que dans la réalité m est plus petit que n.

6.5 Conclusion

Dans ce chapitre, nous avons présenté la méthode `CubMeth`. Le principe de cette dernière est donné dans la Section 6.1. `CubMeth` attribue un format fixe pour chaque neurone et pour chaque poids synaptique du RN. Cette méthode génère des contraintes dans les sections 6.2 et 6.3, pour calculer le format fixe optimal de chaque coefficient (poids synaptiques et sorties des neurones), plus précisément, le nombre de bits minimal des parties fractionnaires L, satisfaisant le seuil d'erreur *Threshold* et le type de données T, requis par l'utilisateur. Enfin, le nombre de contraintes générées par `CubMeth`, a été présentés dans la Section 6.4. Les expérimentations effectuées sur cette méthode sont mises en avant dans le Chapitre 9. Ce dernier montre que les limites de `CubMeth` sont le nombre élevé de contraintes générées et le temps important pour leur résolution, c'est pourquoi nous définissons les deux méthodes `QuadMeth` et `LinMeth` dans le chapitre suivant. Ces deux méthodes attribuent, respectivement, un format fixe par neurone et par ligne de poids synaptiques, et un format fixe par couche de neurones et par matrice de poids synaptique et, génèrent moins de contraintes par rapport à `CubMeth`.

- Si I_1, Activer les contraintes de la Figure 6.5.
- Sinon, $\forall k, i, j, 1 \leqslant k \leqslant m, 0 \leqslant i, j < n$, $\psi \in \{-3, -2, -1, 0, 1\}$, $\lambda \in \{-1, 0, 1, 2\}$, $\rho \in \{-4, -3, -2\}$,

$$L^{\hat{u}}_{k,i} \geqslant M^{\hat{u}}_{k,i} + \psi, \tag{6.27}$$

$$L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{x}}_{k-1,j} - L^{\hat{w}}_{k-1,i,j} + \lambda, \tag{6.28}$$

$$L^{\hat{u}}_{k,i} \geqslant \nu + M^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + \lambda, \tag{6.29}$$

$$L^{\hat{u}}_{k,i} \geqslant \nu - L^{\hat{w}}_{k-1,i,j} - L^{\hat{x}}_{k-1,j} + \lambda, \tag{6.30}$$

$$L^{\hat{u}}_{k,i} \geqslant \mu - L^{\hat{u}}_{k,i} + \lambda, \tag{6.31}$$

$$L^{\hat{u}}_{k,i} \geqslant -L^{\hat{u}}_{k,i} + 2, \tag{6.32}$$

$$L^{\hat{u}}_{k,i} \geqslant \rho \tag{6.33}$$

Activer les contraintes de la Figure 6.5.

  - Si I_2,

$$\rho = -4, \quad \psi = -3 \quad \text{et} \quad \lambda = -2,$$

  - Si I_3,

$$\rho = -3, \quad \psi = -1 \quad \text{et} \quad \lambda = 1,$$

  - Si I_4,

$$\rho = -, \quad \psi = 0 \quad \text{et} \quad \lambda = 0,$$

  - Si I_5, il faut considérer la valeur absolue de $\hat{u}_{k,i}$ et tester
    * Si I_1,

      Activer les contraintes de la Figure 6.5.

    * Si I_2,

$$\rho = -4, \quad \psi = -2 \quad \text{et} \quad \lambda = -1,$$

    * Si I_3,

$$\rho = -3, \quad \psi = 0 \quad \text{et} \quad \lambda = 2,$$

    * Si I_4,

$$\rho = -, \quad \psi = 1 \quad \text{et} \quad \lambda = 1,$$

FIGURE 6.7: Contraintes Générées par CubMeth pour le Calcul du Nombre de Bits L Minimal des Coefficients du RN Suite à une Fonction d'Activation $\hat{Tanh}$ en Fixe.

Chapitre 7

« Un programme informatique fait ce que vous lui avez dit de faire, pas ce que vous voulez qu'il fasse. »

— Loi de Murphy

De CubMeth vers QuadMeth & LinMeth

♣ ◇ ♣

Le nombre de contraintes générées pour calculer le nombre de bits des parties fractionnaires des coefficients en fixe du RN, dépend de la taille du RN donné en entrée. Plus le RN est grand, plus le nombre de contraintes augmente. Ceci représente un réel problème combinatoire, car le temps de résolution de ces contraintes explose, et souvent les solveurs sont limités par un nombre de contraintes à résoudre. La méthode CubMeth, vue précédemment dans le Chapitre 6, génère un nombre de contraintes croissant de manière cubique dans le pire des cas, et donc les solveurs explosent et ne parviennent à résoudre que des problèmes de petite taille, avec un temps de résolution conséquent (Chapitre 9). Dans ce chapitre, nous mettons en avant les deux méthodes QuadMeth et LinMeth. Ces dernières calculent le nombre de bits minimal des parties fractionnaires des coefficients du RN, en générant un nombre de contraintes croissant, respectivement, de manière quadratique et linéaire, dans le pire cas. QuadMeth attribue un format fixe par neurone et par ligne de poids synaptiques, et QuadMeth assigne un

format fixe par couche de neurones et par matrice de poids synaptiques, contrairement à CubMeth, qui attribue un format fixe pour chaque neurone et pour chaque poids synaptique. QuadMeth et LinMeth sont mises en place pour générer les RNs fixes, en utilisant moins de contraintres que CubMeth, et avec un temps de résolution des ces dernières moins important (Chapitre 9), tout en respectant les exigences de l'utilisateur (seuil d'erreur et type de données). Ce chapitre est organisé comme suit :

- La méthode QuadMeth est présentée dans la Section 7.1. Son principe, ses contraintes générées par couche, ainsi que le nombre de ces dernières, sont également mis en avant dans cette section.
- La Section 7.2 présente le principe de LinMeth, les contraintes générées par cette méthode, ainsi que le nombre de ces dernières par couche.
- La Section 7.3 conclut ce chapitre.

7.1 La Méthode QuadMeth

Dans cette section, la méthode de réglage des formats fixes QuadMeth est mise en avant. Elle a pour but de calculer le nombre de bits minimal des parties fractionnaires des neurones et des lignes de poids synaptiques du RN. Le principe de cette méthode, les contraintes générées par couche, ainsi que le nombre de ces dernières, sont donnés dans cette section.

7.1.1 Principe de QuadMeth

La méthode QuadMeth a pour but de calculer les formats fixes des coefficients $\hat{u}$, $\hat{w}$, $\hat{x}$ du RN, avec un nombre de contraintes croissant quadratiquement dans le pire des cas, en attribuant un format pour chaque neurone et un format pour chaque ligne de poids synaptiques dans la matrice $\hat{W}$. Soulignons que, nous nous intéressons à la génération des contraintes, pour calculer le nombre de bits minimal des parties fractionnaires, et que nous calculons le nombre de bits des parties entières, en effectuant une analyse dynamique du RN, avec différents vecteurs d'entrées. Cette analyse dynamique permet la sous-approximation des *ranges* (plages) des entrées et sorties des neurones de chaque couche.

La Figure 7.1 montre le principe de répartition des formats par la méthode QuadMeth. Le RN présenté dans cette figure, se compose de trois couches entièrement connectées, chacune munie de trois neurones. Comme nous pouvons le constater dans la Figure 7.1, à travers le quadrillage (bleu) autour de chaque neurone, QuadMeth attribue un format à chaque neurone $\hat{u}$ de chaque couche. Nous constatons également que les arêtes sortantes de chaque neurone, correspondantes aux poids synaptiques $\hat{w}$, ont la même couleur et sont connectées aux neurones de la couche suivante. Ce qui équivaut à dire qu'un seul format est attribué à chaque ligne de poids synaptiques, de la matrice $\hat{W}$. Ce format est attribué à tous les poids de la même ligne. Pour le calculer, nous récupérons d'abord la valeur maximale en valeur

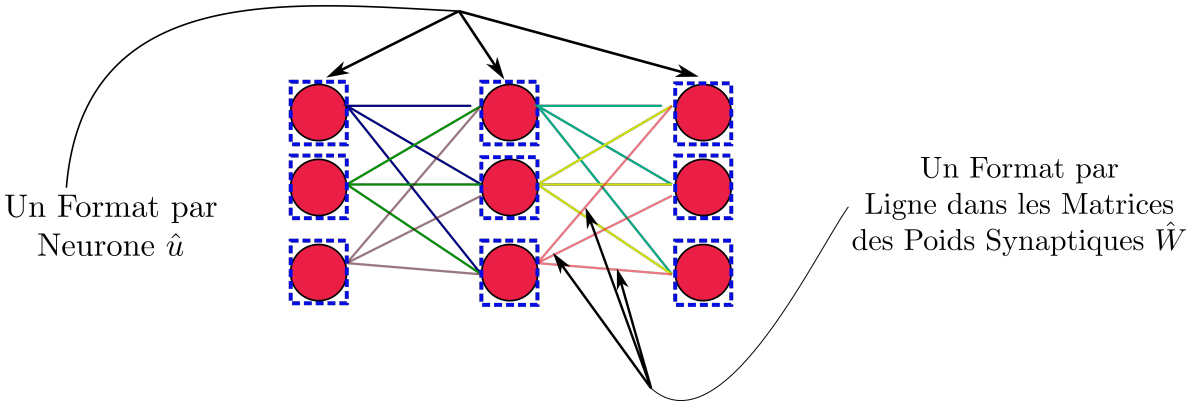

FIGURE 7.1: Répartition des Formats dans QuadMeth.

absolue de chaque ligne, afin de déterminer le nombre de bits de la partie entière maximal, permettant d'éviter les *overflows*. Notons que, la valeur ayant le plus grand nombre de bits de la partie entière, est notée $\hat{w}_{k,i}^{max}$, dans un format $< M_{k,i}^{\hat{w}_{max}}, L_{k,i}^{\hat{w}_{max}} >$, telle que

$$\forall j,\ 0 \leqslant j < n,\ \forall \hat{w}_{k,i,j} \in \hat{w}_{k,i}, \quad \hat{w}_{k,i,j} \leqslant \left|\hat{w}_{k,i}^{max}\right|, \tag{7.1}$$

où $\hat{w}_{k,i}$ est l'ensemble des poids synaptiques de la ligne i de la matrice $\hat{W}_k$, correspondante à la couche k. Le coefficient $\hat{w}_{k,i,j}$ représente le poids synaptique situé à la colonne j et à la ligne i de la matrice $\hat{W}_k$, et $\left|\hat{w}_{k,i}^{max}\right|$ est la valeur ayant le plus grand bit de poids fort dans la ligne i de la matrice $\hat{W}_k$. La valeur $\hat{x}_{k,i}$, dans un format $< M_{k,i}^{\hat{x}}, L_{k,i}^{\hat{x}} >$, représente la valeur de l'entrée maximale du neurone i de la couche k. Maintenant que le nombre de bits des parties entières $M^{\hat{u}}$, $M^{\hat{w}_{max}}$ et $M^{\hat{x}}$, est calculé via l'analyse dynamique[1] du RN, QuadMeth a pour but de trouver le nombre de bits optimal[2] des parties fractionnaires $L^{\hat{u}}$,$L^{\hat{w}_{max}}$ de chaque sortie de neurone $\hat{u}$ et de chaque ligne de poids synaptiques $\hat{w}^{max}$, en générant des contraintes. Ces dernières sont générées en fonction des fonctions d'activation en fixe utilisées, *Linear*, *ReLU*, *Sigmoid* ou *Tanh*, et sont résolues grâce à un solveur (Section 4.1). Dans notre cas, nous avons utilisé le solveur Z3 [dMB08], afin de résoudre les contraintes générées automatiquement par QuadMeth.

Pour récapituler, QuadMeth prend en entrée un RN flottant déjà entraîné dans une certaine précision (simple précision dans notre cas, définie dans la Table 2.1), et synthétise un RN en fixe, ayant le même comportement (sorties) que le RN initial, en générant des contraintes augmentant de manière quadratique (sous-section 7.1.3), dans le pire des cas. Ce RN basé sur l'arithmétique fixe (Section 2.3), respecte un seuil d'erreur à ne pas dépasser pour la couche de sortie, ainsi qu'un nombre de bits maximal à ne pas excéder pour représenter les coefficients. Ils sont tous les deux définis par l'utilisateur et respectivement notés, *Threshold* et T.

[1] Exécuter le RN flottant avec plusieurs vecteurs d'entrées et récupérer les intervalles des valeurs (sous-approximation) des entrées, des sorties et des poids synaptiques de chaque couche.

[2] Le plus petit L répondant aux exigences de l'utilisateur.

Soulignons également que QuadMeth a exactement les mêmes entrées et sorties que CubMeth (Figure 6.2), et que les RNs utilisés dans ces deux méthodes sont les mêmes. Il n'y a que les contraintes générées, ainsi que le nombre de ces dernières, qui changent.

7.1.2 Contraintes Générées par QuadMeth

La méthode QuadMeth a également pour but de calculer le nombre de bits minimal de la partie fractionnaire des sorties des neurones, et des poids synaptiques. Rappelons que dans cette méthode, un format $< M_{k,i}^{\hat{u}}, L_{k,i}^{\hat{u}} >$ est assigné à chaque neurone $\hat{u}_{k,i}$ et un format $< M_{k,i}^{\hat{w}_{max}}, L_{k,i}^{\hat{w}_{max}} >$ est assigné à chaque ligne dans la matrice des poids synaptiques. Notons que $M_{k,i}^{\hat{u}}$, est le nombre de bits de la partie entière de $\hat{u}_{k,i}$, le neurone i de la couche k. Et $L_{k,i}^{\hat{u}}$, est le nombre de bits de sa partie fractionnaire. Selon l'Équation (7.1), $\hat{w}_{k,i}^{max}$ représente le plus grand coefficient de la ligne i, dans la matrice $\hat{W}_k$, en valeur absolue. Le nombre de bits de sa partie entière est $M_{k,i}^{\hat{w}_{max}}$ et le nombre de bits de sa partie fractionnaire est $L_{k,i}^{\hat{w}_{max}}$. Soulignons également que $L_{k,i}^{\hat{u}}$ et $L_{k,i}^{\hat{w}_{max}}$, sont les variables du système de contraintes à minimiser (Section 4.1). Comme pour CubMeth, QuadMeth génère également des contraintes pour déterminer $L_{k,i}^{\hat{u}}$ et $L_{k,i}^{\hat{w}_{max}}$ minimaux, dans le cas d'une fonction d'activation $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ ou $\hat{Tanh}$. La génération des contraintes de la somme pondérée des poids consiste également en cinq points pour cette méthode : l'assertion du type de données requis, la positivité des parties fractionnaires, la condition aux limites, la propagation des calculs vers l'avant et vers l'arrière, et la borne sur l'erreur de la somme pondérée des poids synaptiques.

Pour obtenir les contraintes correspondantes à la somme pondérée des poids et les fonctions d'activation $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ et $\hat{Tanh}$, il suffit de substituer $\hat{w}_{k,i,j}$ par $\hat{w}_{k,i}^{max}$, dans les figures 6.5, 6.6 et 6.7. Cela correspond à l'attribution d'un format pour chaque ligne de la matrice des poids synaptiques. Par exemple, dans la Figure 6.5, les contraintes définies dans les équations (6.4) et (6.5), permettant l'assertion du type de données T, deviennent

$$M_{k,i}^{\hat{w}_{max}} + L_{k,i}^{\hat{w}_{max}} \leqslant T - 1, \qquad 0 \leqslant k < m,\ 0 \leqslant i < n, \tag{7.2}$$

$$M_{k,i}^{\hat{w}_{max}} + L_{k,i}^{\hat{w}_{max}} + M_{k,i}^{\hat{x}} + L_{k,i}^{\hat{x}} \leqslant 2T - 1, \qquad 0 \leqslant k < m,\ 0 \leqslant i, j < n. \tag{7.3}$$

7.1.3 Nombre de Contraintes Générées par Couche

Dans cette sous-section, nous nous intéressons au nombre de contraintes générées par couche dans le pire cas par QuadMeth, ainsi que leur complexité. Les propositions 7.1.1, 7.1.2 et 7.1.3 présentent le nombre de contraintes générées par couche, suite à une fonction d'activation $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ ou $\hat{Tanh}$, dans le meilleur et dans le pire cas.

Proposition 7.1.1. *Le nombre de contraintes générées par couche, suite à une fonction d'activation $\hat{Linear}$ ou $\hat{ReLU}$, est défini par*

$$nbr = 14n, \tag{7.4}$$

où $nbr \in \mathbb{N}$, est le nombre de contraintes générées et, $n \in \mathbb{N}$ est le nombre de neurones par couche.

Preuve. *Dans le cas de* QuadMeth*, selon l'Équation (7.1), tous les* $\hat{w}_{k,i,j}$ *sont substitués par* $\hat{w}_{k,i}^{max}$ *dans la Figure 6.5. Toutes les contraintes de cette méthode génèrent n contraintes pour chaque couche k (ayant Linear ou ReLU comme fonction d'activation), car ces dernières dépendent uniquement de l'indice i. En additionnant le nombre de toutes ces contraintes générées, nous obtenons*

$$nbr = 14n.$$ ■

Proposition 7.1.2. *Soient* $n, nbr \in \mathbb{N}$*, tels que, n est le nombre de neurones par couche et nbr est le nombre de contraintes générées par couche. Le nombre de contraintes générées par le Sigmoid dans le meilleur des cas, est donné avec*

$$nbr = 14n. \tag{7.5}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 50n. \tag{7.6}$$

Preuve. *Dans le meilleur des cas, c'est-à-dire quand* I_1 *ou* I_5 *(Chapitre 6) avec des valeurs dans* I_1 *uniquement sont activés, le nombre de contraintes générées est égal à celui de la somme pondérée des poids synaptiques, d'un ReLU ou d'un Linear. Et dans le pire des cas, les intervalles* I_1*,* I_2*,* I_3*,* I_4*, et* I_5 *avec des valeurs dans* I_1*,* I_2*,* I_3*,* I_4*, sont activés. Notons que dans le cas de* QuadMeth*, selon l'Équation (7.1), tous les* $\hat{w}_{k,i,j}$ *sont substitués par* $\hat{w}_{k,i}^{max}$*, dans les figures 6.5 et 6.6. Le nombre de contraintes générées par chaque équation de la Figure 6.5 est égal à n. Et donc, en activant tous les intervalles de la Figure 6.6, nous obtenons*

$$nbr = 14n + 6 \times (6n) = 50n.$$ ■

Proposition 7.1.3. *Soient* $n, nbr \in \mathbb{N}$*, tels que, n est le nombre de neurones par couche et nbr est le nombre de contraintes générées par couche. Le nombre de contraintes générées suite à une fonction d'activation Tanh dans le meilleur des cas, est donné avec*

$$nbr = 14n. \tag{7.7}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 54n. \tag{7.8}$$

N.B. *La preuve concernant le calcul du nombre de contraintes générées par la Tanh, se fait exactement de la même manière que celle du Sigmoid.*

Afin de calculer le nombre total de contraintes générées par un RN, il suffit d'additionner le nombre de contraintes générées par l'ensemble des couches du RN. Pour obtenir le nombre de contraintes générées suite à une fonction d'activation *Linear* ou *ReLU* (respectivement *Sigmoid* et *Tanh*), il faut multiplier le nombre de couches utilisant cette fonction d'activation par le nombre de contraintes générées pour une seule couche. Le nombre de contraintes générées par QuadMeth afin de, calculer le nombre de bits minimal des parties

fractionnaires du RN, est de complexité quadratique $O(n) = n^2$ dans le pire cas, c'est-à-dire quand le nombre de couches (m) est égal au nombre de neurones (n). Notons que dans la réalité m est plus petit que n. Le nombre de contraintes générées par cette méthode augmente moins vite que celui de `CubMeth`, présentée dans le Chapitre 6. En revanche, cette méthode optimise moins de bits par rapport à `CubMeth` (Chapitre 9), car elle assigne un format par ligne de poids synaptiques, et ce en utilisant le M maximal pour toute la ligne de la matrice $\hat{W}$ (bits à gauche inutilisés dans la représentation fixe).

7.2 La Méthode `LinMeth`

Afin de calculer le nombre de bits minimal pour représenter les coefficients $\hat{u}$, $\hat{x}$, $\hat{w}$ du RN, en utilisant l'arithmétique fixe, vue dans le Chapitre 2, la méthode `LinMeth` génère des contraintes, pour calculer le nombre de bits des parties fractionnaires de ces coefficients. Dans cette section, nous mettons en avant la méthode `LinMeth`. Cette dernière calcule le nombre de bits minimal des parties fractionnaires des coefficients du RN, en attribuant un format fixe par couche de neurones et un format fixe par matrice de poids synaptiques, contrairement à `CubMeth` (Chapitre 6) et `QuadMeth` (Section 7.1), qui attribuent un format fixe pour chaque neurone et pour chaque poids synaptique, et un format fixe pour chaque neurone et pour chaque ligne de poids synaptiques, respectivement.

7.2.1 Principe de `LinMeth`

La méthode `LinMeth` a pour but de calculer les formats fixes des coefficients $\hat{u}$, $\hat{w}$, $\hat{x}$, en assignant un format fixe pour chaque couche de neurones et chaque matrice de poids synaptiques $\hat{W}$. Soulignons que, nous nous intéressons à la génération des contraintes pour calculer le nombre de bits minimal des parties fractionnaires, et que nous calculons le nombre de bits des parties entières, en effectuant une analyse dynamique du RN avec différents vecteurs d'entrée. Ensuite, nous récupérons le nombre de bits de poids fort maximal de chaque matrice, des sorties de chaque couche et de chaque vecteur des entrées. Cette analyse dynamique permet la sous-approximation des intervalles de valeurs.

La Figure 7.2 illustre le principe de calcul et de répartition des formats par la méthode `LinMeth`. Le RN présenté dans cette figure se compose de trois couches entièrement connectées entre elles, contenant trois neurones chacune. Comme nous pouvons le constater dans la Figure 7.2, à travers les quadrillages autour de chaque couche de neurones, `LinMeth` attribue un format à chaque couche du RN. Pour le déterminer, nous calculons d'abord le nombre de bits de la partie entière maximale, des entrées et sorties de la couche, grâce à l'analyse dynamique du RN. Notons que, la sortie ayant le plus grand nombre de bits de la partie entière, est notée $\hat{u}_k^{max}$ dans un format $< M_k^{\hat{u}_{max}}, L_k^{\hat{u}_{max}} >$, telle que

$$\forall i,\ 0 \leqslant i < n,\ \forall \hat{u}_{k,i} \in \hat{U}_k, \quad \hat{u}_{k,i} \leqslant |\hat{u}_k^{max}|\,, \tag{7.9}$$

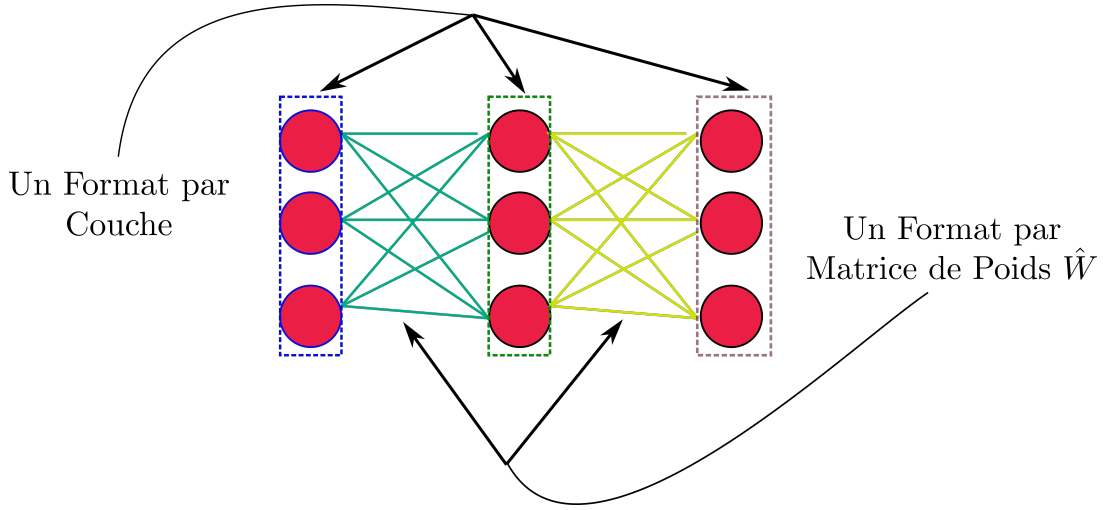

FIGURE 7.2: Répartition des Formats dans `LinMeth`.

où $\hat{U}_k$ est le vecteur des sorties correspondant à la couche k. Le coefficient $\hat{u}_{k,i}$ représente la sortie du neurone i de la couche k, et $\left|\hat{u}_k^{max}\right|$ est la valeur ayant le plus grand bit de poids fort dans le vecteur des sorties $\hat{U}_k$.
Nous constatons également que les arêtes sortantes de chaque neurone de chaque couche, correspondantes aux poids synaptiques $\hat{w}$, ont la même couleur et sont connectées aux neurones de la couche suivante. Ce qui équivaut à dire qu'un seul format est attribué à chaque matrice de poids synaptiques $\hat{W}$. Pour le calculer, nous récupérons d'abord la valeur maximale en valeur absolue de chaque matrice, afin de déterminer le nombre de bits de la partie entière maximal, permettant d'éviter les *overflows*. Notons que, la valeur ayant le plus grand nombre de bits de la partie entière, est notée $\hat{w}_k^{max}$, dans un format $< M_k^{\hat{w}_{max}}, L_k^{\hat{w}_{max}} >$, telle que

$$\forall j,\, 0 \leqslant j < n,\, \forall \hat{w}_{k,i,j} \in \hat{W}_k, \quad \hat{w}_{k,i,j} \leqslant \left|\hat{w}_k^{max}\right|, \tag{7.10}$$

où $\hat{W}_k$ est la matrice des poids synaptiques, correspondante à la couche $k+1$. Le coefficient $\hat{w}_{k,i,j}$ représente le poids synaptique situé à la colonne j et à la ligne i de la matrice $\hat{W}_k$, et $\left|\hat{w}_k^{max}\right|$ est la valeur ayant le plus grand bit de poids fort dans la matrice $\hat{W}_k$.
Nous effectuons la même opération pour les vecteurs des entrées $\hat{X}_k$, c'est-à-dire nous récupérons le nombre de bits de poids fort maximal de l'entrée de chaque couche. Cette entrée est notée $\hat{x}_k^{max}$, dans un format $< M_k^{\hat{x}_{max}}, L_k^{\hat{x}_{max}} >$, telle que

$$\forall i,\, 0 \leqslant i < n,\, \forall \hat{x}_{k,i} \in \hat{X}_k, \quad \hat{x}_{k,i} \leqslant \left|\hat{x}_k^{max}\right|, \tag{7.11}$$

où $\hat{x}_k$ est le vecteur des entrées de la couche $k+1$. Le coefficient $\hat{x}_{k,i}$ représente l'entrée i de la couche $k+1$, et $\left|\hat{x}_k^{max}\right|$ est la valeur ayant le plus grand bit de poids fort dans le vecteur des entrées $\hat{X}_k$. Maintenant que le nombre de bits des parties entières $M^{\hat{u}_{max}}$, $M^{\hat{w}_{max}}$ et $M^{\hat{x}_{max}}$, est calculé via une analyse dynamique du RN, `LinMeth` a pour but de trouver le nombre de bits optimal[3] des parties fractionnaires $L^{\hat{u}_{max}}$,$L^{\hat{w}_{max}}$ de chaque couche k et de

[3]Le plus petit L répondant aux exigences de l'utilisateur.

chaque matrice de poids synaptiques $\hat{W}$, en générant des contraintes automatiquement. Ces contraintes sont générées en fonction des fonctions d'activation en fixe utilisées *Linear*, *ReLU*, *Sigmoid* ou *Tanh*, et sont résolues grâce à un solveur (Section 4.1). Dans notre cas, nous avons utilisé le solveur `Z3` [dMB08], afin de résoudre les contraintes générées automatiquement par `LinMeth`.

Pour récapituler, `LinMeth` prend en entrée un RN flottant déjà entraîné dans une certaine précision (simple précision dans notre cas, définie dans la Table 2.1), et synthétise un RN en fixe, ayant le même comportement que le RN initial, en générant un nombre de contraintes avec une complexité linéaire, dans le pire cas. Ce RN basé sur l'arithmétique fixe (Section 2.3), respecte un seuil d'erreur à ne pas dépasser pour la couche de sortie, ainsi qu'un nombre de bits maximal à ne pas excéder pour représenter les coefficients. Ils sont tous les deux définis par l'utilisateur et respectivement notés, *Threshold* et T.

Soulignons également que `LinMeth` a exactement les mêmes entrées et sorties que `CubMeth` (Figure 6.2) et `QuadMeth` (Section 7.1), et que les RNs utilisés dans ces trois méthodes sont les mêmes. Il n'y a que les contraintes générées, ainsi que le nombre de ces dernières, qui changent.

7.2.2 Contraintes Générées par `LinMeth`

La méthode `LinMeth` a le même but que `CubMeth` et `QuadMeth`, qui est de calculer le nombre de bits minimal de la partie fractionnaire des sorties des neurones et des poids synaptiques. Rappelons que dans cette méthode, un format $< M_k^{\hat{u}_{max}}, L_k^{\hat{u}_{max}} >$ est assigné à chaque couche de neurones, un format $< M_k^{\hat{w}_{max}}, L_k^{\hat{w}_{max}} >$ est attribué à chaque matrice de poids synaptiques et un format $< M_k^{\hat{x}_{max}}, L_k^{\hat{x}_{max}} >$ est affecté à chaque vecteur des entrées. Notons que selon l'Équation (7.9), $M_k^{\hat{u}_{max}}$ est le nombre de bits de la partie entière maximale des sorties de la couche k. Et $L_k^{\hat{u}_{max}}$ est le nombre de bits de sa partie fractionnaire. Selon l'Équation (7.10), $\hat{w}_k^{max}$ représente le plus grand coefficient de la matrice $\hat{W}_k$, en valeur absolue. Le nombre de bits de sa partie entière est $M_k^{\hat{w}_{max}}$ et le nombre de bits de sa partie fractionnaire est $L_k^{\hat{w}_{max}}$. Et suivant l'Équation (7.11), $\hat{x}_k^{max}$ représente le plus grand coefficient du vecteur des entrées $\hat{X}_k$, en valeur absolue. Le nombre de bits de sa partie entière est $M_k^{\hat{x}_{max}}$ et le nombre de bits de sa partie fractionnaire est $L_k^{\hat{x}_{max}}$. Soulignons également que, $L_{k,i}^{\hat{u}}$ et $L_{k,i}^{\hat{w}_{max}}$, sont les variables du système de contraintes à minimiser (Section 4.1). Comme c'est le cas pour `CubMeth` et `QuadMeth`, `LinMeth` génère également des contraintes pour déterminer $L_k^{\hat{u}_{max}}$ et $L_k^{\hat{w}_{max}}$ minimaux, dans le cas d'une fonction d'activation *Linear*, *ReLU*, *Sigmoid* ou *Tanh*. La génération des contraintes de la somme pondérée des poids consiste également en cinq points pour cette méthode : l'assertion du type de données requis, la positivité des parties fractionnaires, la condition aux limites, la propagation des calculs vers l'avant et vers l'arrière, et la borne sur l'erreur de la somme pondérée des poids synaptiques.

Pour obtenir les contraintes correspondantes à la somme pondérée des poids et les fonctions d'activation *Linear*, *ReLU*, *Sigmoid* et *Tanh*, il suffit de substituer $\hat{u}_{k,i}$ par $\hat{u}_k^{max}$, $\hat{w}_{k,i,j}$ par $\hat{w}_k^{max}$ et $\hat{x}_{k,j}$ par $\hat{x}_k^{max}$, dans les figures 6.5, 6.6 et 6.7. Cela correspond à l'attribution

d'un format pour chaque couche de neurones, chaque matrice de poids synaptiques et chaque vecteur des entrées. Par exemple, dans la Figure 6.5, les contraintes définies dans les équations (6.2), (6.3), (6.4) et (6.5), permettant l'assertion du type de données T, deviennent

$$M_k^{\hat{x}_{max}} + L_k^{\hat{x}_{max}} \leqslant T - 1, \qquad 0 \leqslant k \leqslant m, \tag{7.12}$$

$$M_k^{\hat{u}_{max}} + L_k^{\hat{u}_{max}} \leqslant T - 1, \qquad 1 \leqslant k \leqslant m, \tag{7.13}$$

$$M_k^{\hat{w}_{max}} + L_k^{\hat{w}_{max}} \leqslant T - 1, \qquad 0 \leqslant k < m, \tag{7.14}$$

$$M_k^{\hat{w}_{max}} + L_k^{\hat{w}_{max}} + M_k^{\hat{x}_{max}} + L_k^{\hat{x}_{max}} \leqslant 2T - 1, \qquad 0 \leqslant k < m. \tag{7.15}$$

7.2.3 Nombre de Contraintes Générées par Couche

Dans cette sous-section, nous nous intéressons au nombre de contraintes générées par couche dans le pire cas par `LinMeth`, ainsi que leur complexité. Les propositions 7.2.1, 7.2.2 et 7.2.3 donnent le nombre de contraintes générées par couche, suite à une fonction d'activation *Linear*, *ReLU*, *Sigmoid* ou *Tanh*, dans le meilleur et dans le pire cas.

Proposition 7.2.1. *Le nombre de contraintes générées par couche, suite à une fonction d'activation Linear ou ReLU, est défini par*

$$nbr = 14, \tag{7.16}$$

où $nbr \in \mathbb{N}$, est le nombre de contraintes générées par couche.

Preuve. *Dans le cas de LinMeth, selon les équations (7.9), (7.10), et (7.11), tous les $\hat{u}_{k,i}$, $\hat{w}_{k,i,j}$ et $\hat{x}_{k,j}$, sont respectivement substitués par, $\hat{u}_k^{max}$, $\hat{w}_k^{max}$ et $\hat{x}_k^{max}$, dans la Figure 6.5. Toutes les contraintes de cette méthode génèrent chacune une contrainte pour chaque couche k, ayant Linear ou ReLU comme fonction d'activation. En additionnant le nombre de toutes ces contraintes générées, nous obtenons*

$$nbr = 14. \qquad \blacksquare$$

Proposition 7.2.2. *Soit $nbr \in \mathbb{N}$ le nombre de contraintes générées par couche, suite à une fonction d'activation Sigmoid. Dans le meilleur des cas, ce nombre de contraintes est donné avec*

$$nbr = 14. \tag{7.17}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 50. \tag{7.18}$$

Preuve. *Dans le meilleur des cas, c'est-à-dire quand I_1 ou I_5 (Chapitre 6) avec des valeurs dans I_1 uniquement sont activés, le nombre de contraintes générées est égal à celui de la somme pondérée des poids synaptiques, d'un ReLU ou d'un Linear. Et dans le pire des cas, les intervalles I_1, I_2, I_3, I_4, et I_5 avec des valeurs dans I_1, I_2, I_3, I_4, sont activés. Chacune des équations (6.20), (6.21), (6.22), (6.23), (6.24), (6.25) génère une contrainte par couche dans LinMeth.*

Le nombre de contraintes résultant pour chaque intervalle vaut 6. *Et donc, en activant tous les intervalles de la Figure* 6.6 *avec la substitution des équations* (7.9), (7.10), *et* (7.11) *dans cette dernière, nous obtenons*

$$nbr = 14 + 6 \times 6 = 50. \quad \blacksquare$$

Proposition 7.2.3. *Soit* $nbr \in \mathbb{N}$ *le nombre de contraintes générées par couche, suite à une fonction d'activation* $\hat{Tanh}$. *Dans le meilleur des cas, ce nombre de contraintes est donné avec*

$$nbr = 14. \tag{7.19}$$

Et le nombre de contraintes générées dans le pire des cas vaut

$$nbr = 54. \tag{7.20}$$

N.B. *La preuve concernant le calcul du nombre de contraintes générées par la* $\hat{Tanh}$, *se fait exactement de la même manière que celle du* $\hat{Sigmoid}$.

Pour calculer le nombre total de contraintes générées par un RN, il suffit d'additionner le nombre de contraintes générées par l'ensemble des couches du RN. Afin d'obtenir le nombre de contraintes générées suite à une fonction d'activation $\hat{Linear}$ ou $\hat{ReLU}$ (respectivement $\hat{Sigmoid}$ et $\hat{Tanh}$), il faut multiplier le nombre de couches utilisant cette fonction d'activation par le nombre de contraintes générées pour une seule couche. La complexité de la méthode `LinMeth` pour générer les contraintes, afin de calculer le nombre de bits minimal des parties fractionnaires du RN, est de complexité linéaire $O(n) = n$, dans le pire des cas, c'est-à-dire quand le nombre de couches (m) est égal au nombre de neurones (n). Soulignons qu'en réalité, m est plus petit que n. Le nombre de contraintes générées par `LinMeth` augmente moins vite (Chapitre 9) que celui des méthodes `CubMeth` et `QuadMeth`, présentées respectivement dans le Chapitre 6 et la Section 7.1. En revanche, cette méthode optimise moins de bits (Chapitre 9) par rapport à `CubMeth` et `QuadMeth`, car elle assigne un format par matrice de poids synaptiques (respectivement couche et vecteur des entrées), et ce en utilisant le M maximal de la matrice $\hat{W}$ (respectivement sorties des couches et vecteurs $\hat{X}$).

Discussion

Les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, présentées respectivement dans le Chapitre 6 et les sections 7.1 et 7.2, ont le même objectif : synthétiser un RN fixe, satisfaisant les exigences de l'utilisateur (*Threshold* et T). Elles prennent exactement les mêmes entrées et utilisent les mêmes RNs.

Les différences entre les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, sont la technique utilisée pour calculer les formats fixes et, le nombre de contraintes générées. Le nombre de contraintes de la première croît cubiquement dans le pire des cas (Chapitre 6), car elle attribue un format à chaque neurone et à chaque poids synaptique, et celui de la deuxième méthode augmente quadratiquement (Section 7.1) dans le pire des cas, car elle assigne un format à chaque neurone et un format à chaque ligne de poids synaptiques, mais `LinMeth`

est de complexité linéaire dans le pire cas, car elle attribue un format à chaque couche du RN et un format à chaque matrice de poids synaptiques. Notons également que, le nombre de contraintes générées par un RN, dépend du nombre de couches (m) du RN et du carré du nombre de neurones (n) par couches (Section 6.4) dans CubMeth, tandis que dans QuadMeth, il dépend du nombre de couches (m) du RN et du nombre de neurones (n) et dans LinMeth, il ne dépend que du nombre de couches (m). Les expérimentations en termes de nombre et temps de résolution des contraintes générées, de précision et *Threshold*, de bits/bytes optimisés et de répartition des types de données dans le programme synthétisé, sont présentées dans le Chapitre 9.

7.3 Conclusion

Ce chapitre a présenté les deux méthodes QuadMeth et LinMeth. Les principes de ces dernières sont respectivement donnés dans les sections 7.1 et 7.2. QuadMeth assigne un format fixe pour chaque neurone et chaque ligne dans la matrice des poids synaptiques et, LinMeth attribue un format fixe pour chaque couche de neurones et pour chaque matrice de poids synaptiques du RN. Ces méthodes génèrent des contraintes pour calculer les formats fixes minimaux des coefficients (poids synaptiques et sorties des neurones), plus précisément, le nombre de bits minimal des parties fractionnaires L, satisfaisant le seuil d'erreur *Threshold* et le type de données T, requis par l'utilisateur. Enfin, la complexité de QuadMeth et LinMeth, ainsi que et le nombre de contraintes générées par ces dernières, ont été présentés dans les sous-sections 7.1.3 et 7.2.3. Les expérimentations effectuées sur ces deux méthodes sont mises en avant dans le Chapitre 9. La troisième partie de ce manuscrit porte sur les expérimentations effectuées sur les trois méthodes CubMeth, QuadMeth et LinMeth, via notre outil de synthèse de code à virgule fixe pour les RNs SyFix, et notre librairie virgule fixe SyFi, qui seront présentés dans le chapitre suivant.

Troisième partie

Évaluation des Performances de `SyFix`

Chapitre 8

« Vous croyez savoir quand vous apprenez, vous en êtes sur quand vous écrivez, persuadé quand vous enseignez, mais certain seulement quand vous programmez... »

— Alan Jay Perlis

L'Outil SyFix

♣ ◇ ♣

Dans ce chapitre, nous mettons en avant notre outil de **Sy**nthèse de code à virgule **Fix**e pour les RNs, dit SyFix. Ce dernier fait appel à la librairie SyFi, afin d'utiliser les fonctions à virgule fixe qu'elle implémente. SyFix englobe les trois méthodes de calcul des formats fixes minimaux pour les sorties des neurones et les coefficients du RN. Ces trois méthodes sont CubMeth, QuadMeth et LinMeth. Elles sont, respectivement, définies dans les chapitres 6 et 7. Ce chapitre est organisé comme suit :

- La Section 8.1 présente la librairie virgule fixe SyFi et ses fonctionnalités.
- Notre outil de synthèse de code à virgule fixe pour les RNs, SyFix, est mis en avant dans la Section 8.2. Cette dernière présente le *workflow* et les principales fonctionnalités de SyFix [BMS22b, BMS22a].
- La Section 8.3 conclut ce chapitre.

8.1 Librairie Virgule Fixe SyFi

Notre librairie virgule fixe, dite SyFi, est mise en avant dans cette section. Elle a pour but d'implémenter les opérations élémentaires (addition, soustraction, multiplication et

```
1 /************** Format de l'Addition ******************/
2  inline int msb_add_different_format(int m1, int m2)
3     { return 1+((m1>=m2) ? m1:m2);}
4  inline int lsb_add_different_format(int msb, int length)
5     { return length-msb ; }
6 /************** Addition Fixe ******************/
7  FIX Add(FIX x1,FIX x2,int length);
8/***************** Alignement Fixe ******************/
9  FIX align_length(FIX x,int ufp_max,int length);
10
```

FIGURE 8.1: Addition Virgule Fixe par SyFi.

division), les fonctions d'activation des RNs requises par SyFix (Section 8.2), ainsi que les fonctions requises par l'outil de synthèse de code à virgule fixe pour les programmes POPiX [BKBM22], dans le Chapitre 10. Les opérations arithmétiques à virgule fixe ne sont pas standardisées, comme c'est le cas pour l'arithmétique flottante [ANS19]. Le développeur doit gérer manuellement les formats fixes, c'est-à-dire le nombre de bits avant et après la virgule, contrairement à l'exposant qui est géré de manière automatique dans les flottants. Ces formats fixes sont manipulés via des décalages. La mauvaise gestion de ces derniers, peut provoquer des dépassements de capacité de type *overflow* ou *underflow*, ce qui va fausser tous les résultats des calculs. C'est pourquoi, nous devons porter une attention particulière à la bonne gestion des formats fixes pendant les calculs à virgule fixe.

8.1.1 Calcul des Formats Fixes

Le calcul et la bonne gestion des formats fixes est une tâche primordiale pour le bon déroulement des calculs en virgule fixe. C'est pourquoi, SyFi implémente les fonctions permettant de calculer ces formats, c'est-à-dire le nombre de bits des parties entières et fractionnaires de chaque opération arithmétique en fixe. Rappelons que, ces formats fixes sont formellement définis dans le Chapitre 2. La librairie SyFi est implémentée en C [KR88] (dédiée à POPiX [BKBM22]), mais nous avons aussi une version de cette librairie en Python [vR07] (dédiée à SyFiX). Considérons par exemple l'addition à virgule fixe. Dans la Figure 8.1, les deux fonctions *msb_add_different_format()* et *lsb_add_different_format()*, définies dans la ligne 2 et 4, calculent respectivement le nombre de bits de la partie entière et fractionnaire du résultat de l'addition fixe. La première fonction prend en paramètres $m1$ et $m2$, les nombres de bits des parties entières des opérandes à additionner, et renvoie le maximum entre les deux en ajoutant un bit supplémentaire. Ce dernier a pour but d'éviter les *overflows* en cas de présence de retenue. La seconde fonction prend en paramètres, le nombre de bits de la partie entière du résultat de l'addition fixe (calculé par la première fonction ou donné manuellement), noté *msb*, ainsi que le nombre de bits

total pour représenter le résultat de l'addition fixe, noté *length*. Ainsi, le nombre de bits de la partie fractionnaire est calculé en soustrayant *msb* de *length*. Si nous faisons la relation entre $m1$, $m2$, *length* et les chapitres 6 et 7, nous constatons que $m1$ et $m2$ correspondent aux M des coefficients du RN, déterminés via l'analyse dynamique. Et la valeur de *length* est obtenue en additionnant M, L et un bit pour le signe. Rappelons que L, représente la solution des contraintes générées, c'est-à-dire le nombre minimal de bits de la partie fractionnaire des coefficients du RN.

8.1.2 Opérations Élémentaires

Un calcul contient au moins une opération élémentaire. Notons également que chaque fonction complexe[1] est composée (ou approchée) de plusieurs opérations élémentaires. La librairie `SyFi` implémente les quatre opérations élémentaires principales : l'addition, la soustraction, la multiplication et la division. Considérons par exemple la fonction *Add()* présentée dans la ligne 7 de la Figure 8.1. Cette fonction calcule le résultat d'une addition fixe. Elle prend en paramètres deux nombres fixes $x1$ et $x2$, ainsi que le nombre de bits total (somme du nombre de bits de la partie entière, fractionnaire et le signe) sur lequel sera représenté le résultat de cette addition. Cette fonction fera appel aux deux fonctions *msb_add_different_format()* et *lsb_add_different_format()*, définies également dans cette figure, pour calculer le format en sortie de l'addition de $x1$ et $x2$. Ensuite, elle appellera la fonction *align_length()* (ligne 9), pour aligner $x1$ et $x2$ avec le format de la sortie calculé. Et enfin, elle additionnera les deux opérandes et renvoie le résultat de l'addition fixe.

8.1.3 Fonctions d'Activation

Les opérations élémentaires sont essentielles dans le calcul des fonctions d'activation, définies dans le Chapitre 3. `SyFi` implémente les fonctions calculant les activations $\hat{Linear}$, $\hat{ReLU}$, $\hat{Sigmoid}$ et $\hat{Tanh}$ en fixe. Rappelons également que, nous considérons les versions approchées des deux fonctions d'activation $\hat{Sigmoid}$ et $\hat{Tanh}$. Ces deux dernières sont basées sur une approximation linéaire par morceaux et sont respectivement définies dans les définitions 3.2.9 et 3.2.10. Soulignons également que cette approximation est basée sur des multiplications et des additions seulement.

8.1.4 Fonctions Requises par `POPiX`

Les fonctions définies précédemment concernent principalement les RNs. Cette sous-section met en avant les fonctions fixes utilisées par l'outil `POPiX` [BKBM22] dans le Chapitre 10. `POPiX` a besoin des fonctions calculant la racine carrée, la valeur absolue et les fonctions trigonométriques (sinus, cosinus et arctangente) en fixe. Ces dernières sont implémentées dans `SyFi`. Notons que pour la racine carrée *sqrt*, `SyFi` utilise l'algorithme de [MNR17]. Et pour approcher les fonctions trigonométriques, `SyFi` utilise *Power*

[1] Fonction étant difficile à calculer ou à approcher sa valeur exacte en virgule fixe, comme le sigmoïde.

series [Moi12]. Soulignons que l'ordre de développement de *Power series* dépend du nombre de bits significatifs, requis au résultat.

8.2 L'Outil SyFix

Dans cette section, notre outil de synthèse de code à virgule fixe pour les RNs est présenté. Il se nomme SyFix. Ce dernier est implémenté en Python [vR07] et synthétise du code en C [KR88]. SyFix fait appel à la librairie SyFi, définie dans la Section 8.1, pour effectuer les opérations arithmétiques en fixe. Soulignons que notre outil passe par six étapes essentielles afin de synthétiser un RN à virgule fixe. Ces étapes sont illustrées dans le *workflow* de la Figure 8.2.

Récupérer les Entrées

La première étape de SyFix consiste à importer le RN flottant entraîné et, récupérer les informations et les paramètres donnés par l'utilisateur. Comme mentionné dans la Figure 8.2, les entrées de SyFix sont : la valeur du *Threshold*, le type de données T, la méthode choisie pour calculer les formats fixes, le RN flottant entraîné (par TensorFlow [AAB+16] par exemple) et les vecteurs des entrées en float. Notons que le *Threshold* représente le seuil d'erreur à ne pas dépasser, entre les sorties flottantes et fixes de la dernière couche du RN. Le calcul du bit de poids fort de ce seuil nous donne le nombre de bits corrects que doivent avoir au minimum les parties fractionnaires des sorties des neurones de la dernière couche. Le type de données T, est le plus grand type de données qu'on peut avoir dans le code fixe synthétisé. Rappelons que ce dernier peut prendre une valeur quelconque, mais dans ces travaux, nous avons utilisé $T \in \{8, 16, 32\}$ bits pour la compatibilité avec les types de données du langage C [KR88]. SyFix propose les trois méthodes CubMeth, QuadMeth et LinMeth pour régler le RN fixe. Ces méthodes sont formellement définies dans les chapitres 6 et 7.

Effectuer une Analyse Dynamique du RN

Dans la Figure 8.2, la deuxième étape consiste à effectuer une analyse dynamique du RN flottant donné en entrée, en considérant multiples vecteurs d'entrées. Cette analyse nous permet d'avoir une sous-approximation des *ranges*, des entrées et des sorties de chaque couche, ainsi que le nombre de bits des parties entières M des poids synaptiques et des biais. Par exemple, si le *range* des entrées du RN appartient à $[-2.1, 5.2]$, alors le M maximal des entrées (égal à 3) est celui de 5. Actuellement, nous utilisons l'analyse dynamique pour déterminer ces *ranges*, mais nous prévoyons de calculer une sur-approximation de ces derniers, en effectuant une analyse statique, dans le futur.

Générer les Contraintes et les Résoudre

La génération des contraintes pour régler les formats fixes des coefficients et des sorties des neurones, représente l'étape la plus importante de `SyFix`. Ce dernier propose trois méthodes pour déterminer ces formats fixes : `CubMeth`, `QuadMeth` et `LinMeth`. `CubMeth` consiste à attribuer un format pour chaque neurone et pour chaque poids synaptique. `QuadMeth` assigne un format pour chaque neurone et pour chaque ligne dans la matrice des poids synaptiques, et `LinMeth` attribue un format par couche de neurones et par matrice de poids synaptiques. Les contraintes générées par ces trois méthodes sont, respectivement, modélisées dans les chapitres 6 et 7. Pour résoudre ces dernières, `SyFix` fait appel au solveur `Z3` [dMB08]. Ce dernier renvoie le nombre de bits des parties fractionnaires minimal des coefficients, si la solution existe. Grâce à ces résultats, nous obtenons un réglage des formats fixes pour le RN donné en entrée, satisfaisant le *Threshold*, le type de données T et permettant d'évaluer le RN en fixe.

Évaluer le RN

La quatrième étape de `SyFix` évalue le RN donné en entrée, dans les deux arithmétiques fixe et flottante. Dans l'évaluation fixe du RN, ce dernier doit être d'abord converti en fixe, en utilisant les formats obtenus dans l'étape précédente. Ensuite, cette évaluation du RN dans l'arithmétique fixe, consiste à calculer la somme pondérée des poids, définie dans l'Équation (5.11), sur laquelle une fonction d'activation doit être appliquée (Chapitre 3) par la suite. Notons que `SyFix` sollicite `SyFi` (Section 8.1), pour effectuer cette évaluation en virgule fixe. En parallèle, l'évaluation flottante de ce RN (initial) est effectuée de la même manière que l'évaluation fixe, afin de calculer l'erreur absolue (Définition 2.1.3) entre les deux évaluations fixe et flottante dans la dernière étape de la Figure 8.2.

Synthétiser le Code du RN

La dernière étape de `SyFix` est celle de la synthèse de code à virgule fixe pour le RN flottant (en simple précision (Table 2.1)), donné en entrée. Notre outil synthétise un code C [KR88] basé sur des entiers uniquement (Figure 6.3). `SyFix` a la possibilité de synthétiser un code avec un type de données unique (celui exigé par l'utilisateur "T"), mais il peut aussi synthétiser un code avec précision mixte [CBB+17, DHS18], en attribuant le plus petit type de données possible pour chaque variable. La précision mixte n'est pas facile à mettre en place, car nous devons nous assurer qu'il n'y ait pas d'*overflows* dans les calculs intermédiaires. Pour déterminer le type de données minimal de chaque variable, nous calculons, $M + L + 2$, la somme des nombres de bits de la partie entière, de la partie fractionnaire, du signe et du bit de sécurité, en cas de retenue dans les calculs intermédiaires. Soulignons également que `SyFix` propose également la synthèse de code en virgule flottante du RN donné en entrée, si l'utilisateur le souhaites.

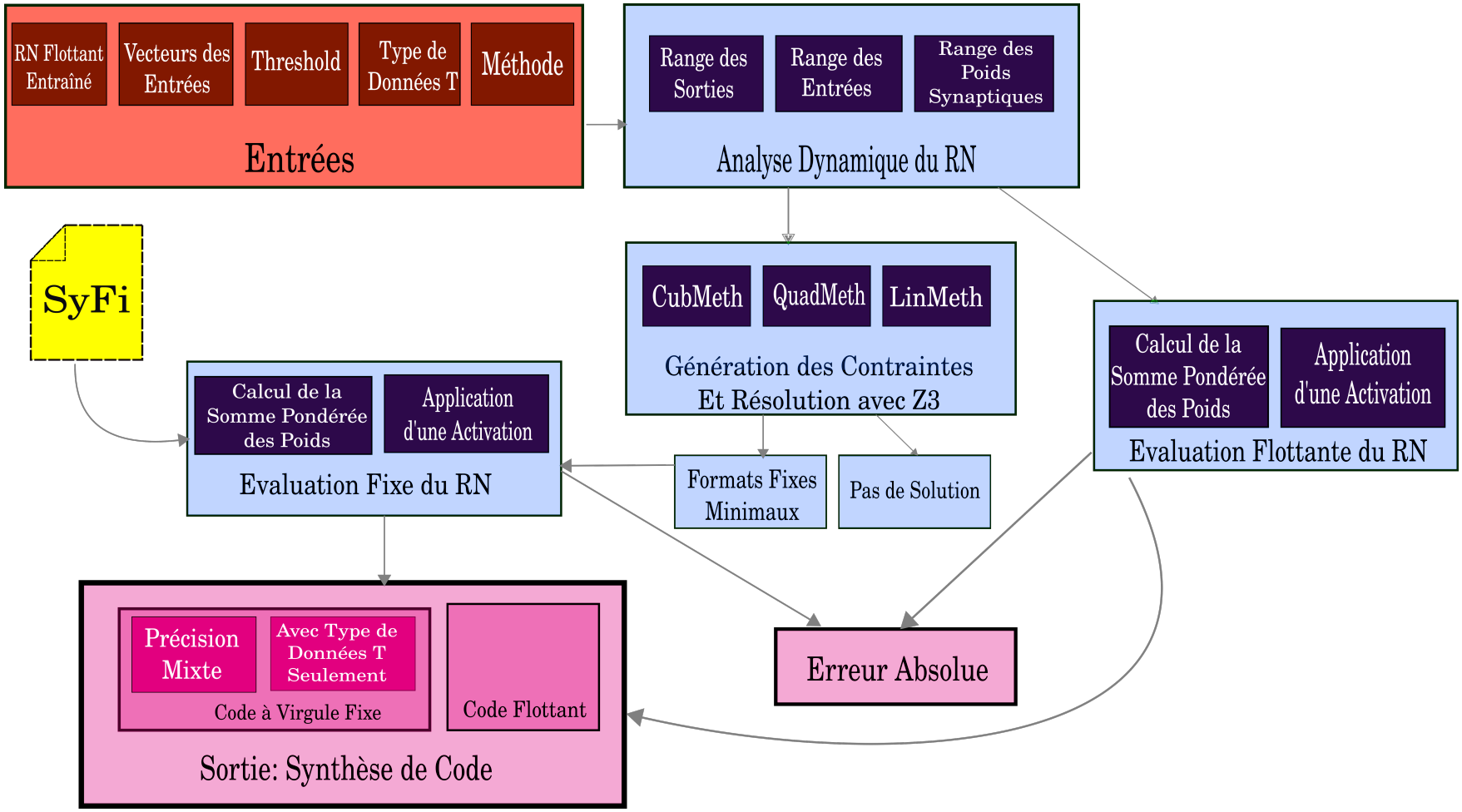

FIGURE 8.2: Workflow de l'Outil SyFix.

Calculer l'Erreur Absolue

Cette dernière étape consiste à calculer l'erreur absolue entre, les sorties fixes et flottantes de la dernière couche du RN, en utilisant l'Équation (2.3). Cette erreur absolue obtenue doit être impérativement inférieure ou égale au *Threshold*, requis par l'utilisateur.

8.3 Conclusion

Dans ce chapitre, nous avons présenté en premier lieu dans la Section 8.1, notre librairie `SyFi` ainsi que, les fonctions qu'elle implémente. `SyFi` est utilisée par `SyFix`, notre outil synthétisant du code à virgule fixe pour un RN flottant, donné en entrée. Dans la Section 8.2, les fonctionnalités et les différentes étapes de `SyFix`, sont mises en avant. Dans le chapitre suivant, nous évaluons les performances de notre outil en utilisant `CubMeth`, `QuadMath` et `LinMeth`, en termes de bits/bytes optimisés, de précision et seuil d'erreur, de nombre et de temps de résolution des contraintes générées et en termes de distribution des types de données dans le programme synthétisé.

Chapitre 9

« Dans un ordinateur, le langage naturel n'est pas naturel. »

— Alan Jay Perlis

Évaluation des Performances de SyFix

♣ ◇ ♣

L'évaluation des trois méthodes CubMeth, QuadMeth et LinMeth, via notre outil SyFix, est mise en avant dans ce chapitre. Soulignons d'abord que, les expérimentations sont effectuées sur les mêmes RNs flottants (donnés en entrées) représentés en simple précision (Table 2.1), pour les trois méthodes. Ces expérimentations concernent la précision numérique et la garantie du non dépassement du seuil d'erreur exigé, le nombre et le temps de résolution des contraintes générées par CubMeth, QuadMeth et LinMeth. Le nombre de bits/bytes optimisés grâce au réglage des formats fixes par ces trois méthodes, ainsi que la répartition des types des données dans le code synthétisé, sont également présentés dans ce chapitre. Ce dernier est organisé comme suit :

- La Section 9.1 décrit les *benchmarks* utilisés dans nos expérimentations, afin d'évaluer les performances de notre outil SyFix.
- La Section 9.2 concerne la précision numérique et le respect du seuil d'erreur requis. Elle présente les résultats de SyFix en considérant différents seuils d'erreur et types de données, exigés par l'utilisateur.

- Le nombres de contraintes générées ainsi que la durée de leur résolution par le solveur `Z3` [dMB08], sont mis en avant dans la Section 9.3.
- La Section 9.4 présente le nombre de bits/bytes optimisés après le réglage des formats fixes par `CubMeth`, `QuadMeth` et `LinMeth`, respectivement définies dans les chapitres 6 et 7.
- La Section 9.5 montre le pourcentage des types de données présents dans les codes synthétisés par `SyFix`, pour les RNs de la Section 9.1.
- Une comparaison entre `SyFix` et les outils de l'état de l'art, présentés dans les chapitres 3 et 4, est effectuée dans la Section 9.6.
- La Section 9.7 conclut ce chapitre.

9.1 Benchmarks

Cette section présente les RNs utilisés dans l'évaluation des trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, par notre outil `SyFix`. Ces méthodes sont définies formellement dans les chapitres 6 et 7. Énumérons dans la Table 9.1 ces RNs.

RN \ Attributs	Couches	Neurones	Connexions
Iris	3	33	363
CosFun	4	40	400
Hyper	4	48	576
Wine	2	52	1352
Bumps	2	60	1800
Cancer	3	150	7500
AFun	2	200	10000

TABLE 9.1: Description des RNs Utilisés dans les Expérimentations.

La première colonne de la Table 9.1 présente les noms des RNs décrits. La deuxième colonne donne le nombre de couches de chaque RN et la troisième montre le nombre de neurones. La dernière colonne présente le nombre de connexions entre les neurones de ces réseaux, ayant des couches entièrement connectées.

Iris

La première ligne de la Table 9.1 présente le RN *Iris*. Ce dernier est un classifieur. Il classifie les plantes *Iris* [SDDM12] en trois classes : *Iris-Setosa*, *Iris-Versicolour* et *Iris-Virginica*. Il prend en entrée quatre valeurs numériques correspondantes à la longueur et la largeur

des sépales[1], ainsi que la longueur et la largeur des pétales en centimètres. *Iris* se compose de trois couches, 33 neurones et 363 connexions entre les neurones.

CosFun

CosFun est un RN de type interpolateur, c'est-à-dire qu'il calcule une fonction mathématique. Cet interpolateur calcule la fonction $f(x, y) = x \times cos(y)$. *CosFun* se compose de quatre couches, 40 neurones et 400 connexions entre ces derniers.

Hyper

Le RN *Hyper* présenté dans la Table 9.1 est un interpolateur. Ce RN calcule le sinus hyperbolique du point (x, y). Il est composé de quatre couches et 48 neurones entièrement connectés, ce qui engendre 576 connexions. Ce RN est le même que celui utilisé dans [IM19].

Wine

Le RN *Wine* est un classifieur. Il classifie les vins en trois classes [ACdV92] : classe 0, classe 1 et classe 2. Ce RN prend en entrée treize paramètres (alcool, acide malique, cendres, etc.). *Wine* est composé de deux couches entièrement connectées, 52 neurones et 1352 relations entre ces derniers.

Bumps

Bumps est un RN de type interpolateur calculant la fonction *bump*. Ce RN est le même que celui utilisé dans [IM19]. Il se compose de 60 neurones, deux couches entièrement connectées et 1800 relations entre les neurones.

Cancer

Le RN *Cancer* est un classifieur. Il diagnostique et classifie le cancer du sein en deux catégories : bénin ou malin. Ce classifieur prend en entrée trente attributs mentionnés dans [WSM92]. Il se compose de trois couches, 150 neurones et 7500 connexions entre les neurones.

AFun

La fonction affine $f(x) = 4 \times x + \frac{3}{2}$ est calculée par l'interpolateur *AFun*. Ce RN se compose de deux couches entièrement connectées, de 200 neurones et de 10000 connexions entre ces derniers. Soulignons que nous aurions pu entraîner ce RN avec moins de neurones, mais nous l'avons fait ainsi, pour tester l'architecture maximale tolérée par `SyFix`.

[1] Chaque pièce (foliole) du calice d'une fleur.

Threshold		$2^{-1} = 0.5$			$2^{-4} \approx 10^{-1}$			$2^{-7} \approx 10^{-2}$			$2^{-10} \approx 10^{-3}$			$2^{-14} \approx 10^{-4}$		
Type de Données T	**Méthode** / **RN**	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin
32 bits	***Iris***	√	√	√	√	√	√	√	√	√	√	√	√	√	×	√
	CosFun	√	√	√	√	√	√	√	√	√	√	√	√	×	×	×
	Hyper	√	×	√	√	×	√	√	×	√	√	×	×	×	×	×
	Wine	√	√	√	√	√	√	√	√	√	√	√	√	√	×	√
	Bumps	√	×	√	√	×	√	√	×	×	√	×	×	×	×	×
	Cancer	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	AFun	√	√	√	√	√	√	√	√	√	√	√	√	×	×	×
16 bits	***Iris***	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	CosFun	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	Hyper	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	Wine	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	Bumps	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	Cancer	√	×	×	√	×	×	×	×	×	×	×	×	×	×	×
	AFun	√	×	×	×	×	×	×	×	×	×	×	×	×	×	×
8 bits	***Iris***	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	CosFun	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Hyper	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Wine	√	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Bumps	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Cancer	√	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	AFun	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

TABLE 9.2: Erreurs des Sorties des RNs Versus *Threshold* en Considérant Différents T.

9.2 Précision et Seuil d'Erreur

Cette section concerne la précision et le seuil d'erreur. À travers la Table 9.2 et la Figure 9.1, nous indiquons si les RNs, définis dans la Section 9.1, satisfont le seuil d'erreur *Threshold* en utilisant le type de données T, définis par l'utilisateur. Si le RN est un interpolateur, cela signifie que la sortie de la fonction mathématique f, a une erreur inférieure ou égale au *Threshold*. Si le RN est un classifieur, cela signifie que l'erreur de classification de ce RN est inférieure ou égale à $(Threshold \times 100)\%$. Rappelons également que le *Threshold* est une valeur entre 0 et 1 et que $T \in \{8, 16, 32\}$ bits.

La Table 9.2 montre si les RNs définis dans la Section 9.1, garantissent les *Threshold* : 2^{-1}, 2^{-4}, 2^{-7}, 2^{-10}, 2^{-14} et les types de données T : 8, 16, 32 bits, respectivement. La première ligne de cette table montre les différentes valeurs de *Threshold* utilisées. Chaque méthode (`CubMeth`, `QuadMeth`, `LinMeth`) est évaluée pour une valeur de *Threshold* et un type de données T. Rappelons que ce dernier (première colonne) représente le type de données maximal, qu'on peut trouver dans le code synthétisé. Par exemple, si T vaut 16 bits, seulement les types de données *int*_16 et *int*_8, peuvent se trouver dans le code synthétisé. La deuxième colonne de la Table 9.2 donne les noms des RNs utilisés dans les expérimentations. Le symbole $\times$ fait référence à l'infaisabilité de la solution de la méthode utilisée (`CubMeth`, `QuadMeth`, `LinMeth`), c'est-à-dire que le solveur `Z3` [dMB08] ne parvient pas à trouver une solution (formats fixes minimaux) ou que nous ne pouvons pas satisfaire le seuil d'erreur, en utilisant le type de données T. Le symbole $\surd$ signifie que `Z3` a trouvé une solution (formats minimaux) au système de contraintes de la méthode utilisée (Chapitre 6, et 7). Chaque ligne de la Table 9.2 correspond à un RN et chaque colonne correspond à une méthode utilisée pour un *Threshold* défini. Par exemple, dans la sixième ligne de la deuxième colonne, le RN *Cancer* satisfait tous les *Threshold* (jusqu'à $10^{-4} \approx 2^{-14}$) quelle que soit la méthode utilisée, en utilisant un type de données $T \leqslant 32$ *bits*. En d'autres termes, au minimum 14 bits corrects dans les parties fractionnaires de la couche de sortie du RN sont assurés. En utilisant la méthode `CubMeth`, les RNs *Iris, Wine et Cancer*, garantissent un *Threshold* jusqu'à 2^{-14}, correspondant à quatorze bits corrects dans la partie fractionnaire de la pire sortie du RN. Et les RNs *CosFun, Hyper, Bumps* et *AFun*, assurent au minimum 10 bits corrects dans les parties fractionnaires des sorties. Au-delà de cette valeur de *Threshold*, le solveur `Z3` ne trouve pas de solution en utilisant un type de données $T \leqslant 32$ bits pour ces RNs. Les deux méthodes `QuadMeth` et `LinMeth` garantissent une erreur inférieure ou égale à 2^{-10} pour les RNs *CosFun* et *AFun*. En utilisant le type de données $T \leqslant 16$ bits et la méthode `CubMeth`, tous les RNs sauf *AFun* ont une erreur inférieure ou égale à 2^{-4}, et assurent l'exactitude d'au moins quatre bits dans la partie fractionnaire de la pire sortie de la dernière couche. Seuls les RNs *Wine* et *Cancer* peuvent assurer un bit significatif dans la partie fractionnaire, en utilisant le type de données $T \leqslant 8$ bits et la méthode `CubMeth`. Les deux autres méthodes `QuadMeth` et `LinMeth`, ne peuvent assurer ces

Threshold en utilisant ce type de données (8 bits), car le solveur `Z3` ne trouve pas de solution.

Les résultats peuvent varier en fonction de plusieurs paramètres : les vecteurs d'entrées, les coefficients des poids synaptiques w et des biais b, les fonctions d'activation, le seuil d'erreur *Threshold* et le type de données T. Généralement, lorsque les coefficients sont compris entre -1 et 1, les résultats sont plus précis car leurs ufp (Définition 2.1.1) sont négatifs, et donc on peut avoir plus de bits significatifs dans les parties fractionnaires. L'infaisabilité des solutions dépend également de la taille du type de données T, par exemple, si nous avons un petit type de données T et un nombre conséquent de bits dans la partie entière des coefficients (grande valeur de M), nous ne pouvons pas avoir suffisamment de bits après la virgule fixe pour satisfaire les grands seuils d'erreur.

La Figure 9.1 représente un zoom sur la colonne correspondante au *Threshold* $= 2^{-10}$ et les lignes 1, 4 et 6 correspondantes aux RNs : *Iris, Wine, Cancer*, en utilisant un type de données $T \leqslant 32$ bits. Cette figure illustre l'erreur absolue (Définition 2.1.3) entre le RN flottant et le RN fixe, en utilisant *Threshold* $= 2^{-10}$ et $T \leqslant 32$ bits. L'axe horizontal représente les différents vecteurs d'entrées et l'axe vertical représente la pire erreur absolue de la couche de sortie du RN. Nous constatons que les courbes des erreurs absolues des trois méthodes, sont toujours en-dessous de celle du *Threshold* pour tous les RNs, cela signifie que ce *Threshold* est respecté. Remarquons également que la courbe de l'erreur de `CubMeth` est celle qui se rapproche le plus de celle du *Threshold*, car les formats fixes sont calculés pour chaque neurone et chaque poids synaptique dans le Chapitre 6. La courbe de la méthode `LinMeth` est celle qui s'éloigne le plus de la courbe du *Threshold*. Elle est plus précise car elle utilise plus de bits dans les parties fractionnaires des coefficients (elle attribue un format par couche et un format par matrice de poids synaptiques, en considérant le plus grand M à chaque fois).

9.3 Nombre et Durée de Résolution des Contraintes Générées

Les expérimentations de cette section concernent le nombre de contraintes générées par les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, ainsi que le temps pris par `Z3` [dMB08] pour résoudre ces contraintes.

La Table 9.3 montre dans sa deuxième (respectivement quatrième et sixième) colonne, le nombre de contraintes générées pour chaque RN de la première colonne par `CubMeth` (respectivement `QuadMeth` et `LinMeth`). Sa troisième (respectivement cinquième et septième) colonne donne le temps d'exécution du solveur `Z3`, pour résoudre ces contraintes. Comme nous pouvons le voir, le nombre de contraintes et le temps d'exécution du solveur augmentent très rapidement dans le cas de `CubMeth`. Par exemple, le RN *Hyper* prend 37 secondes pour résoudre les 4368 contraintes générées par `CubMeth` et le RN *bumps* prend environ 15 minutes pour résoudre les 19530 contraintes générées. On peut aussi remarquer que le RN *Cancer* a 150 neurones et 81600 contraintes. Le nombre de contraintes a vite

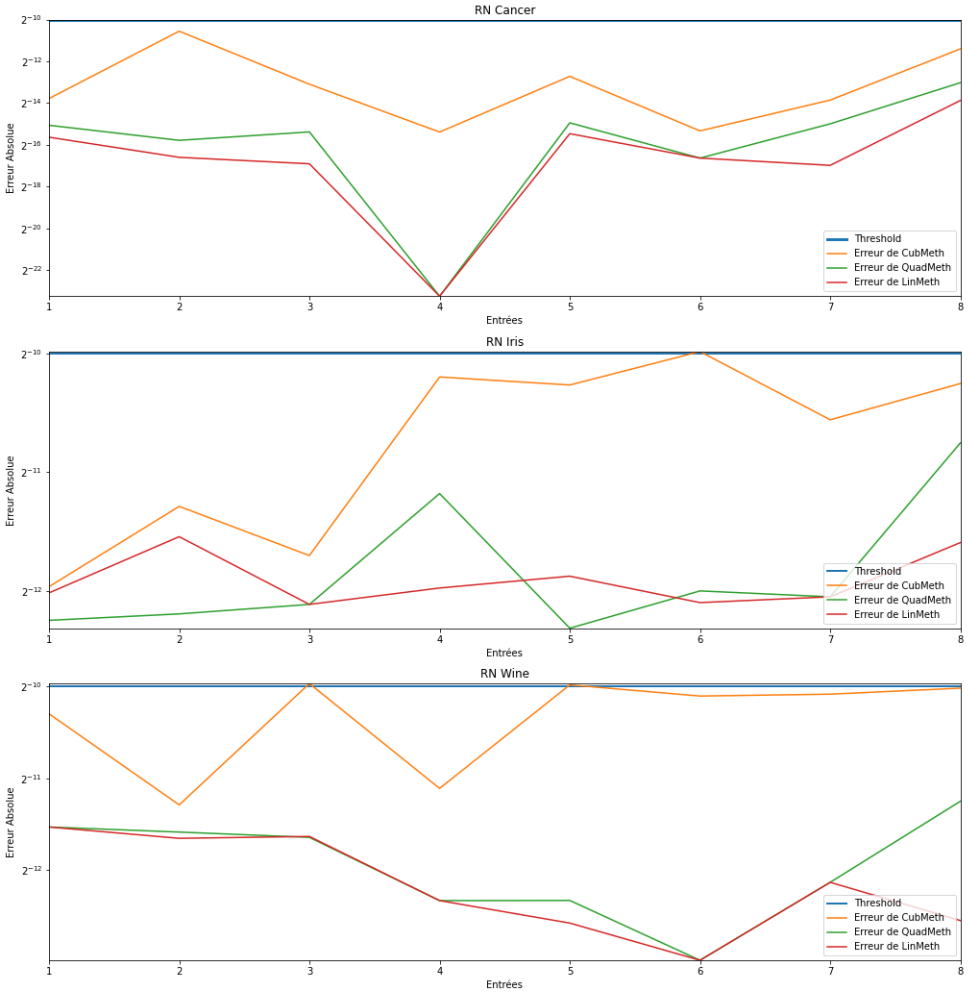

FIGURE 9.1: Erreur Absolue entre le RN Flottant et le RN Fixe en considérant un *Threshold* à 2^{-10} et un type de données $T \leqslant 32$ bits.

augmenté et le temps d'exécution pour les résoudre est supérieur à une heure. Comme nous l'avons souligné dans le Chapitre 6, le nombre de contraintes est de complexité cubique pour `CubMeth` dans le pire cas, c'est pourquoi nous avons mis en place `QuadMeth` et `LinMeth`. La Table 9.3 montre que le nombre de contraintes générées par `LinMeth` est petit par rapport à celui des deux autres méthodes, car ce dernier dépend uniquement du nombre de couches (Chapitre 7), contrairement à `QuadMeth` qui dépend du nombres de couches et du nombre de neurones par couche (Chapitre 7), et `CubMeth` dépendant du

	CubMeth		QuadMeth		LinMeth	
Att. / RN	Nbr Cont.	Durée Résol.	Nbr Cont.	Durée Résol.	Nbr Cont.	Durée Résol.
Wine	1716	10 s	728	5 s	108	< 1 s
Iris	2772	22 s	462	1 s	162	< 1 s
CosFun	3080	28 s	560	2 s	216	< 1 s
Hyper	4368	37 s	672	3 s	216	< 1 s
Bumps	19530	14 min 50 s	840	8 s	108	< 1 s
AFun	70700	1 h 13 min 26 s	2800	24 s	108	< 1 s
Cancer	81600	1 h 38 min 12 s	5700	41 s	162	< 1 s

TABLE 9.3: Nombres de Contraintes Générées et leur Temps de Résolution.

nombre de couches et du carré de nombre de neurones par couche (Chapitre 6). En utilisant `LinMeth`, le solveur `Z3` renvoie une réponse en moins d'une seconde pour tous les RNs et le temps de réponse due à l'utilisation de `QuadMeth` reste raisonnable (secondes) par rapport à celui de `CubMeth` (heures). Par exemple, le nombre de contraintes de *AFun* vaut 70700 lors de l'utilisation de `CubMeth`, contre 2800 et 108 lors de l'utilisation de `QuadMeth` et `LinMeth`, respectivement. Le temps de leur résolution passe d'une heure à 24 s pour `QuadMeth` et moins d'une seconde pour `LinMeth`. Certes, `LinMeth` est celle qui génère le moins de contraintes et est la plus rapide à les résoudre, mais elle utilise plus de mémoire, en utilisant plus de bits pour représenter les coefficients fixes (sections 9.4 et 9.7).

9.4 Bits/Bytes Optimisés

Dans cette section, les expérimentations présentées concernent la Figure 9.2 et les tables 9.4, 9.5 et 9.6. Elles ont pour but de montrer que notre approche économise des bits/bytes, grâce au calcul des formats minimaux pour les neurones via les méthodes, `CubMeth`, `QuadMeth` et `LinMeth`, respectivement définies dans les chapitres 6 et 7. Au départ, tous les neurones sont représentés en $T \in \{8, 16, 32\}$ bits, et après l'étape de réglage des formats via l'une des trois méthodes citées, nous réduisons le nombre de bits/bytes pour représenter un RN, tout en respectant le *Threshold* et le T, fixés par l'utilisateur.

La Figure 9.2 montre le nombre total de bits requis pour chaque neurone de chaque couche de *CosFun*, après le réglage des formats $< M,L >$ via `CubMeth` (Chapitre 6), en considérant $Threshold = 2^{-10}$ et $T \leqslant 32$ bits. Il est intéressant d'utiliser ces résultats dans les FPGA [Tri12] ou dans la synthèse de code C (Figure 6.4) avec précision mixte [DHS18, CBB+17] (Section 9.5), c'est-à-dire en attribuant le plus petit type de données à chaque variable. Par exemple, nous attribuons le type de données $int16\_t$ au premier neurone de la première couche qui n'a besoin que de 13 bits. Soulignons que la taille de tous les neurones était 32 bits initialement, et après la résolution des contraintes de `CubMeth` et l'obtention

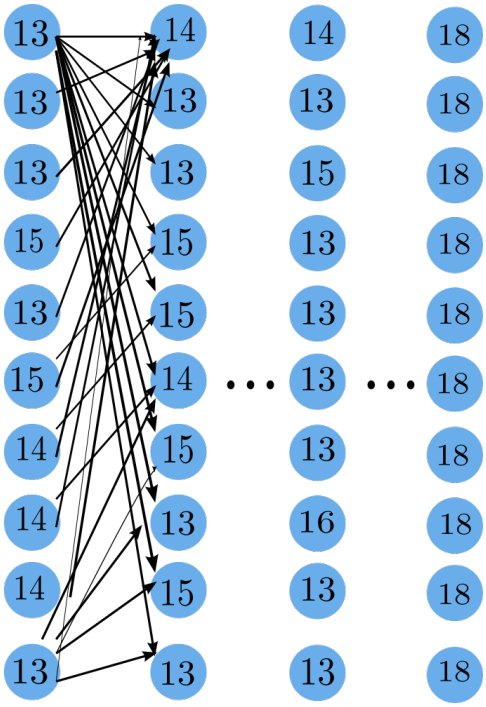

FIGURE 9.2: Nombre de Bits de Chaque Neurone de *CosFun* après le Réglage des Formats via `CubMeth` pour un $Threshold = 2^{-10}$ et un $T \leqslant 32$ bits.

des formats minimaux, nous n'avons besoin que de 18 bits pour les neurones de la couche de sortie, afin de satisfaire le seuil d'erreur 2^{-10}. On gagne 14 bits dans chaque neurone de la couche de sortie, ce qui représente un gain de 43.75%.

Les tables 9.4, 9.5 et 9.6 montrent la taille de chaque RN en bytes avant et après le réglage des formats via `CubMeth`, `QuadMeth` et `LinMeth`, respectivement. Elles présentent également le nombre de bytes optimisés, ainsi que le pourcentage de gain pour chaque RN de la Table 9.1, pour différentes valeurs de *Threshold* (2^{-1}, 2^{-4}, 2^{-7} 2^{-10}, et 2^{-14}) et de types de données ($T \leqslant 32$, $T \leqslant 16$ bits). Par exemple, dans la Table 9.4 concernant `CubMeth`, la taille (nombre de bytes) du RN *Cancer* est réduite de 6×, par rapport à sa taille initiale (on gagne 84, 21%), en utilisant un type de données $T \leqslant 32$ bits et un $Threshold = 2^{-1}$. On peut remarquer que, le nombre de bytes économisés diminue lorsque le *Threshold* requis augmente, car il faut être plus précis dans la représentation en virgule fixe (le nombre de bits L après la virgule fixe augmente). Notons que Le symbole ×, dans les tables 9.4, 9.5 et 9.6, signifie que le solveur `Z3` [dMB08] ne trouve pas de solution pour les contraintes de `CubMeth`, `QuadMeth` et `LinMeth`, respectivement.

Ces tables montrent que les trois méthodes économisent des bits/bytes en prenant en compte le seuil d'erreur et le type de données, requis par l'utilisateur. `CubMeth` (Table 9.4) est la méthode la moins gourmande en termes de mémoire, car elle calcule le format fixe minimal de chaque neurone et de chaque poids synaptique du RN (elle règle tous les coefficients). `QuadMeth` (Table 9.5) consomme plus de bytes que `CubMeth` et moins de bytes que `LinMeth`. Cette méthode attribue un format minimal pour chaque neurone et pour

			Type de Données T ⩽ 32 bits				Type de Données T ⩽ 16 bits			
Thresh.	**Méth.**	**Att. / RN**	**SBO**	**SAO**	**BS**	**Gain (%)**	**SBO**	**SAO**	**BS**	**Gain (%)**
$2^{-1} = 0.5$	CubMeth	***Iris***	132	20	112	**85.22**	66	12	54	**82.76**
		CosFun	160	27	133	**83.67**	80	16	64	**80.78**
		Hyper	192	54	138	**72.07**	96	30	66	**69.01**
		Wine	208	38	170	**82.15**	104	19	85	**82.45**
		Bumps	480	177	303	**63.25**	240	84	156	**65**
		Cancer	400	64	336	**84.21**	200	32	168	**84.37**
		AFun	800	326	474	**59.37**	320	220	100	**31.25**
$2^{-4} \approx 10^{-1}$	CubMeth	***Iris***	132	25	107	**81.15**	66	17	49	**75**
		CosFun	160	32	128	**80.15**	80	21	59	**74.68**
		Hyper	192	61	131	**68.29**	96	35	61	**63.93**
		Wine	208	49	159	**76.8**	104	31	73	**70.19**
		Bumps	480	196	284	**59.16**	240	98	142	**59.21**
		Cancer	400	84	316	**79.12**	200	56	144	**72**
		AFun	800	376	424	**53.08**	×	×	×	×
$2^{-7} \approx 10^{-2}$	CubMeth	***Iris***	132	41	91	**66.66**	×	×	×	×
		CosFun	160	59	101	**63.12**	×	×	×	×
		Hyper	192	84	108	**56, 25**	×	×	×	×
		Wine	208	77	131	**62.98**	×	×	×	×
		Bumps	480	206	274	**57.16**	×	×	×	×
		Cancer	400	105	295	**73.96**	×	×	×	×
		AFun	800	450	350	**43.75**	×	×	×	×
$2^{-10} \approx 10^{-3}$	CubMeth	***Iris***	132	35	97	**74.05**	×	×	×	×
		CosFun	160	43	117	**73.67**	×	×	×	×
		Hyper	192	72	120	**62.69**	×	×	×	×
		Wine	208	68	140	**67.54**	×	×	×	×
		Bumps	480	208	272	**56.66**	×	×	×	×
		Cancer	400	125	275	**68.9**	×	×	×	×
		AFun	800	475	325	**40.62**	×	×	×	×
$2^{-14} \approx 10^{-4}$	CubMeth	***Iris***	×	×	×	×	×	×	×	×
		CosFun	×	×	×	×	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	79	129	**62.19**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	178	222	**55.5**	×	×	×	×
		AFun	×	×	×	×	×	×	×	×

TABLE 9.4: Gain de Bytes après le Réglage des Formats par `CubMeth` pour Différentes Valeurs de *Threshold* et de Type de Données T.

SBO : Size Before formats Optimization (Bytes), **SAO :** Size After formats Optimization (Bytes),
BS : Bytes Saved.

			Type de Données T ⩽ 32 bits				Type de Données T ⩽ 16 bits			
Thresh.	**Méth.**	**Att. / RN**	**SBO**	**SAO**	**BS**	**Gain (%)**	**SBO**	**SAO**	**BS**	**Gain (%)**
$2^{-1} = 0.5$	QuadMeth	***Iris***	132	23	109	**82.67**	×	×	×	×
		CosFun	160	36	124	**77.57**	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	56	152	**73.25**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	93	307	**76.78**	×	×	×	×
		AFun	800	388	422	**52.86**	×	×	×	×
$2^{-4} \approx 10^{-1}$	QuadMeth	***Iris***	132	31	101	**76.98**	×	×	×	×
		CosFun	160	54	106	**66.32**	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	72	136	**65.5**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	120	280	**70.12**	×	×	×	×
		AFun	800	416	384	**48**	×	×	×	×
$2^{-7} \approx 10^{-2}$	QuadMeth	***Iris***	132	38	94	**71.3**	×	×	×	×
		CosFun	160	63	97	**60.70**	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	88	120	**57.75**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	147	253	**63.46**	×	×	×	×
		AFun	800	505	295	**36.87**	×	×	×	×
$2^{-10} \approx 10^{-3}$	QuadMeth	***Iris***	132	46	86	**65.62**	×	×	×	×
		CosFun	160	43	117	**73.67**	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	104	104	**50**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	173	227	**56.81**	×	×	×	×
		AFun	800	490	310	**38.75**	×	×	×	×
$2^{-14} \approx 10^{-4}$	QuadMeth	***Iris***	×	×	×	×	×	×	×	×
		CosFun	×	×	×	×	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	×	×	×	×	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	209	191	**47.93**	×	×	×	×
		AFun	×	×	×	×	×	×	×	×

TABLE 9.5: Gain de Bytes après le Réglage des Formats par `QuadMeth` pour Différentes Valeurs de *Threshold* et de Type de Données T.

SBO : Size Before formats Optimization (Bytes), **SAO :** Size After formats Optimization (Bytes), **BS** : Bytes Saved.

			Type de Données T ⩽ 32 bits				Type de Données T ⩽ 16 bits			
Thresh.	**Méth.**	**Att. / RN**	**SBO**	**SAO**	**BS**	**Gain (%)**	**SBO**	**SAO**	**BS**	**Gain (%)**
$2^{-1} = 0.5$	LinMeth	***Iris***	132	25	107	**81.43**	66	12	54	**82.76**
		CosFun	160	39	121	**75.62**	×	×	×	×
		Hyper	192	73	119	**61.97**	×	×	×	×
		Wine	208	49	159	**76.8**	×	×	×	×
		Bumps	480	185	295	**61.51**	×	×	×	×
		Cancer	400	105	295	**73.87**	×	×	×	×
		AFun	800	413	387	**48.37**	×	×	×	×
$2^{-4} \approx 10^{-1}$	LinMeth	***Iris***	132	32	100	**75.75**	×	×	×	×
		CosFun	160	60	100	**62.5**	×	×	×	×
		Hyper	192	88	104	**54.16**	×	×	×	×
		Wine	208	65	143	**69.05**	×	×	×	×
		Bumps	480	224	255	**53.22**	×	×	×	×
		Cancer	400	132	268	**67.21**	×	×	×	×
		AFun	800	435	365	**45.62**	×	×	×	×
$2^{-7} \approx 10^{-2}$	LinMeth	***Iris***	132	40	92	**70.07**	×	×	×	×
		CosFun	160	72	88	**55**	×	×	×	×
		Hyper	192	103	89	**46.35**	×	×	×	×
		Wine	208	81	127	**61.29**	×	×	×	×
		Bumps	480	265	215	**44.94**	×	×	×	×
		Cancer	400	158	242	**60.56**	×	×	×	×
		AFun	800	557	243	**30.37**	×	×	×	×
$2^{-10} \approx 10^{-3}$	LinMeth	***Iris***	132	47	85	**64.39**	×	×	×	×
		CosFun	160	66	94	**58.75**	×	×	×	×
		Hyper	192	72	120	**62.69**	×	×	×	×
		Wine	208	97	111	**53.54**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	185	215	**53.9**	×	×	×	×
		AFun	800	650	250	**25**	×	×	×	×
$2^{-14} \approx 10^{-4}$	LinMeth	***Iris***	132	77	55	**42.32**	×	×	×	×
		CosFun	×	×	×	×	×	×	×	×
		Hyper	×	×	×	×	×	×	×	×
		Wine	208	94	114	**55.22**	×	×	×	×
		Bumps	×	×	×	×	×	×	×	×
		Cancer	400	239	161	**40.24**	×	×	×	×
		AFun	×	×	×	×	×	×	×	×

TABLE 9.6: Gain de Bytes après le Réglage des Formats par `LinMeth` pour Différentes Valeurs de *Threshold* et de Type de Données T.

SBO : Size Before formats Optimization (Bytes), **SAO :** Size After formats Optimization (Bytes),
BS : Bytes Saved.

chaque ligne de poids synaptiques, en considérant le plus grand M des coefficients de la ligne de poids. Et pour finir, la méthode `LinMeth` (Table 9.6) est celle qui réduit le moins le nombre de bytes utilisés, car elle attribue un format par couche et un format par matrice de poids synaptiques (en considérant le plus grand M à chaque fois). Par exemple, dans la Table 9.6, le RN *Cancer* gagne 73.87% en termes de bytes, tandis que dans les tables 9.5 et 9.4 le gain pour ce RN est de 76.78% et 84.21%, respectivement.

9.5 Types de Données avec Précision Mixte

La dernière partie des expérimentations concerne la Table 9.7. Elle a pour but de montrer la distribution des types de données des variables dans le code synthétisé, pour chaque RN de la Table 9.1, en considérant de différentes valeurs de *Threshold* et de type de données T. La première ligne indique le type de données à ne pas dépasser ($T \leqslant 32$, $T \leqslant 16$ bits). La deuxième ligne indique les *Threshold* à garantir. La troisième ligne montre les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth` évaluées pour chaque *Threshold*. La première colonne donne les noms des RNs et les différents types de données inférieurs ou égaux à T. Les colonnes de 2 à 13, montrent le pourcentage de variables représentées en *int*32_*t*, *int*16_*t* et *int*8_*t*, pour $T \leqslant 32$ bits et différentes valeurs de *Threshold*. Et les colonnes de 14 à 19, montrent le pourcentage de variables représentées en *int*16_*t* et *int*8_*t*, pour $T \leqslant 16$ bits et un *Threshold*, respectivement fixé à, 2^{-1} et 2^{-4}. Par exemple, le RN *Iris* a dans son code synthétisé 33,33% de variables en 32 bits, 42.42% en 16 bits et 24.24% en 8 bits, où le seuil d'erreur requis est 2^{-1}, le type de données $T \leqslant 32$ bits et la méthode utilisée est `CubMeth`. Le même RN *Iris* avec le même seuil d'erreur et la même méthode, mais avec un type de données $T \leqslant 16$ bits, a 60,60% de variables représentées en 16 bits et 39,40% en 8 bits.

Le symbole $\times$ dans le tableau 9.7, fait référence à l'infaisabilité de la solution des contraintes, calculant les formats minimaux par l'une des trois méthodes (chapitres 6, et 7), selon le *Threshold* donné et le type de données T requis. Notons également que le symbole - veut dire que le type de données *int*32_*t* n'est pas concerné, quand $T \leqslant 16$ bits. Dans cette table, nous remarquons que plus le seuil d'erreur est petit, plus le type de données des variables est élevé, car nous avons besoin de plus de bits dans les parties fractionnaires. Par exemple, le RN *Hyper* a 58.33% de ses variables en *int*16_*t* et 41.66% en *int*8_*t*, pour un *Threshold* valant 2^{-1} et un $T \leqslant 16$ bits, tandis que pour un *Threshold* égal à 2^{-4}, toutes les variables sont représentées en *int*16_*t*. Les types de données des variables sont obtenus grâce au calcul des formats minimaux, par l'une des trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`. Soulignons que le type de données pour chaque variable est déterminé en calculant la somme du nombre de bits de la partie entière et fractionnaire, puis en ajoutant à ce résultat deux bits ($M + L + 2$). Le premier bit est pour le signe et le second est pour éviter les *overflows*, à cause des retenues dans les calculs intermédiaires.

Thr. / RN/Types	Type de Données Requis ⩽ 32 bits												Type de Données Requis ⩽ 16 bits					
	$2^{-1} = 0.5$			$2^{-4} \approx 10^{-1}$			$2^{-7} \approx 10^{-2}$			$2^{-10} \approx 10^{-3}$			$2^{-1} = 0.5$			$2^{-4} \approx 10^{-1}$		
	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin	Cub	Quad	Lin
Iris																		
int32_t	33.33%	33.33%	33.33%	33.33%	33.33%	66.66%	75.75%	72.72%	66.66%	66.66%	100%	100%	-	-	-	-	-	-
int16_t	42.42%	66.66%	66.66%	66.66%	66.66%	33.33%	24.24%	27.27%	33.33%	33.33%	0%	0%	60.60%	×	×	100%	×	×
int8_t	24.24%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	39.40%	×	×	0%	×	×
CosFun																		
int32_t	27.5%	50%	50%	35%	65.0%	75%	47.5%	85%	100%	72.5%	100%	100%	-	-	-	-	-	-
int16_t	47.5%	50%	50%	65%	35%	25%	52.5 %	15%	0%	27.5%	0%	0%	52.5%	×	×	97.5%	×	×
int8_t	25%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	47.5%	×	×	2.5%	×	×
Hyper																		
int32_t	31.25%	×	50%	45.83 %	×	75%	58.33%	×	100%	75%	×	×	-	-	-	-	-	-
int16_t	45.83 %	×	50%	54.16%	×	25%	41.66%	×	0%	25%	×	×	58.33%	×	×	100%	×	×
int8_t	22.91%	×	0%	0%	×	0%	0%	×	0%	0%	×	×	41.66%	×	×	0%	×	×
Wine																		
int32_t	32.69%	13.46%	0%	42.3%	61.53%	50%	48.07%	78.84%	100%	50%	100%	100%	-	-	-	-	-	-
int16_t	25%	86.53%	100%	57.69%	38.46%	50%	51.92%	21.15%	0%	50%	0%	0%	40.38%	×	×	92.30%	×	×
int8_t	42.3%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	59.61%	×	×	7.69%	×	×
Bumps																		
int32_t	64.16%	×	66.66%	64.16%	×	66.66%	70%	×	100%	93.33%	×	×	-	-	-	-	-	-
int16_t	30%	×	33.33%	34.16%	×	33.33%	30%	×	0%	6.66%	×	×	74.16%	×	×	95%	×	×
int8_t	5.83%	×	0%	1.66%	×	0%	0%	×	0%	0%	×	×	25.83%	×	×	5%	×	×
Cancer																		
int32_t	39%	50%	50%	50%	50%	50%	51%	100%	100%	51%	100%	100%	-	-	-	-	-	-
int16_t	12%	50%	50%	50%	50%	50%	49%	0%	0%	49%	0%	0%	50%	×	×	100%	×	×
int8_t	49%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%	×	×	0%	×	×
AFun																		
int32_t	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100%	100 %	100%	-	-	-	-	-	-
int16_t	0 %	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	×	×	×	×	×
int8_t	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	×	×	×	×	×

TABLE 9.7: Types de Données Utilisés dans le Code Synthétisé en fonction du *Threshold* et du Type de Données T.

Discussion

Pour conclure, LinMeth est la plus rapide (Table 9.3) en termes de temps de résolution des contraintes générées, car c'est celle qui génère le moins de contraintes. Ensuite QuadMeth s'en suit et CubMeth reste la plus pessimiste, en termes de nombre de contraintes générées. LinMeth est plus précise que CubMeth et QuadMeth (Figure 9.1), c'est-à-dire que l'erreur commise dans la couche de sortie est plus petite que celle des deux autres méthodes. En revanche, cette méthode est plus gourmande en termes de mémoire consommée (Table 9.6), car elle utilise plus de bits pour représenter les coefficients (pire cas). CubMeth est la plus lente (Table 9.3) et la moins gourmande en termes de mémoire (Table 9.4). En termes d'erreurs, c'est celle qui se rapproche le plus du *Threshold* (Figure 9.1), car elle attribue un format fixe à chaque neurone et chaque poids synaptique (de manière ajustée). Et QuadMeth est un compromis entre CubMeth et LinMeth, par contre le solveur Z3 [dMB08] échoue à trouver des solutions pour cette méthode plus souvent que les deux autres précédentes.

9.6 Comparaison de SyFix avec les Outils de l'État de l'Art

Dans cette section, nous présentons une comparaison entre notre outil SyFix et les outils et techniques [IM19, GGSS19a, LV20, GPMG18, JDL+19, JRZ+22, JGM+20], présentés dans les chapitres 3 et 4. Malheureusement, nous ne pouvons pas effectuer des comparaisons expérimentalement, à cause de l'absence de plusieurs facteurs et paramètres cités ci-dessous par rapport à SyFix, mais nous présentons une comparaison entre ces outils et SyFix via les caractéristiques mentionnées dans la Table 9.8.

La première colonne de cette Table donne les outils avec lesquels nous comparons SyFix. La deuxième colonne montre les entrées de chaque outil. La technique et l'arithmétique utilisées sont, respectivement, mentionnées dans la troisième et la quatrième colonne. La cinquième colonne indique si ces outils satisfont ou non un seuil d'erreur (*Threshold*), requis par l'utilisateur. Les sixième et septième colonnes indiquent si les outils supportent les couches entièrement connectées (C. EC) et convolutionnelles (C. Conv). Les deux dernières colonnes montrent si les outils synthétisent un code ou génèrent un modèle pour les RNs, donnés en entrée.

La première ligne de la Table 9.8 concerne SyFix. Notre outil prend en entrée un RN entraîné en simple précision (Table 2.1), un *Threshold*, un type de données T à ne pas excéder et des vecteurs d'entrées, comme le montre la Figure 8.2. SyFix utilise l'arithmétique fixe et calcule les formats fixes via les méthodes CubMeth, QuadMeth ou LinMeth (chapitres 6 et 7), en générant des contraintes automatiquement. Actuellement, SyFix ne prend en charge que des RNs avec des couches entièrement connectées, mais nous prévoyons d'ajouter les couches convolutionnelles dans le futur. L'objectif de notre outil est la synthèse de code, basé uniquement sur des entiers, dédié aux équipements contraints, tels que les systèmes embarqués.

La technique d'Ioualalen et Martel [IM19] prend également en entrée un RN entraîné, un *Threshold* à ne pas dépasser et des vecteurs d'entrées. Cette méthode utilise l'arithmétique flottante [ANS19] et réduit également la taille des types de données, grâce à des contraintes linéaires générées. Cette méthode ne prend en charge que les couches entièrement connectées et ne génère pas de modèle ni de code pour le RN, donné en entrée. Il y a quelques fonctionnalités communes avec `SyFix`, mais nous ne pouvons pas effectuer une comparaison entre les deux, en raison de la différence de l'arithmétique utilisée dans le réglage de précision et de l'absence de synthèse de code pour cette approche.

L'outil `SEEDOT` [GGSS19a] synthétise un code à virgule fixe pour les algorithmes d'inférence d'apprentissage automatique (ML), qui peut s'exécuter sur du matériel contraint. Cet outil utilise une stratégie de compilation qui réduit l'espace de recherche pour certains paramètres clés, en particulier les paramètres d'échelle (formats fixes) pour la représentation des nombres à virgule fixe, utilisés dans le code à virgule fixe synthétisé. `SEEDOT` et `SyFix` génèrent tous les deux un code à virgule fixe, mais notre outil garantit un seuil d'erreur et un type de données, requis par l'utilisateur. `SEEDOT` trouve un facteur d'échelle pour le nombre fixe à représenter, tandis que notre outil résout des contraintes pour trouver le format fixe minimal, pour chaque neurone et chaque coefficient du RN. Encore une fois, nous ne pouvons pas comparer les deux approches en raison de l'absence du *Threshold* et du type de données T dans `SEEDOT`.

Lauter et Volkova [LV20] présentent un outil pour l'analyse semi-automatique des erreurs en virgule flottante pour la phase d'inférence d'apprentissage profond. Cet outil prend en entrée un RN entraîné et le transforme en code C++, tout en analysant le besoin en précision de ce RN. Les arithmétiques affine et d'intervalles sont utilisées pour calculer les bornes d'erreurs relatives et absolues. Les principales différences entre `SyFix` et cet outil sont l'arithmétique utilisée et les deux paramètres *Threshold* et type de données T, requis. C'est pourquoi la comparaison entre ces deux outils n'est pas pertinente.

Les outils `Ristretto` [GPMG18] et `F8Net` [JRZ+22], utilisent tous les deux l'arithmétique fixe et la quantification comme technique, pour générer un modèle compressé pour le RN donné en entrée. `Ristretto` prend en entrée un RN entraîné, ainsi que les *ranges* des valeurs numériques des coefficients, tandis que `F8Net` prend un RN pré-entraîné en entrée. Les deux outils prennent en charge les couches convolutionnelles et entièrement connectées. Ces outils sont difficilement comparables à `SyFix`, à cause de l'absence des paramètres : *Threshold* et type de données T. Ajoutons à cela que, l'optimisation des formats fixes n'est pas effectuée lors de l'apprentissage du RN par `SyFix`.

`DeepSZ` [JDL+19] est un outil de compression de RNs. Il comprime les poids synaptiques (*sparse*) de ces derniers, en utilisant l'arithmétique flottante. `DeepSZ` passe par quatre étapes clés : l'élagage du RN, l'évaluation des bornes d'erreurs, l'optimisation de la configuration des bornes d'erreurs et la génération du modèle compressé, et `Condensa` [JGM+20] est un outil utilisant l'élagage et la quantification pour la compression des RN profonds. Il prend en entrée un RN pré-entraîné ainsi qu'une fonction objective (sur la minimi-

sation de la mémoire, le temps d'exécution, etc.), et génère un modèle compressé. Cet outil utilise l'arithmétique flottante et prend en charge les couches convolutionnelles et entièrement connectées. `Condensa` et `DeepSZ` ne satisfont pas un *Threshold* et un type de données T requis, et ils n'utilisent pas la même arithmétique que `SyFix`, pour cela que nous n'avons pas pu les comparer.

9.7 Conclusion

Dans ce chapitre, nous avons présenté en premier lieu dans la Section 9.1, les RNs utilisés pour évaluer notre outil `SyFix`. Ensuite, nous avons présenté dans la Section 9.2, les résultats expérimentaux concernant la précision et la garantie du *Threshold*. Nous nous sommes également intéressés, dans la Section 9.3, au nombre et à la durée de résolution des contraintes générées, par les trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`. La troisième partie des expérimentations (Section 9.4) portait sur le gain en mémoire (bits/bytes) après le réglage des formats fixes, par les trois méthodes citées précédemment. Et en dernier lieu, la Section 9.5 a mis en avant la répartition des types de données des variables dans les programmes synthétisés, en considérant de différentes valeurs de *Threshold* et de T. Pour conclure, Nous avons effectué dans la Section **??** une comparaison entre `SyFix` et les outils de l'état de l'art, présentés dans les chapitres 3 et 4. La partie suivante de ce manuscrit présentera l'outil `POPiX` [BKBM22], combinant `POP` [Ben21] et `SyFi` (Chapitre 8), et ayant pour but de synthétiser des programmes à virgule fixe.

Carac. / Outil	Entrées	Technique	Arith.	Thresh. Err.	C. EC	C. Conv	Synth. Code	Gen. Mod.
SyFix	RN Entraîné + Vect. d'Entr. +*Threshold* + Type de Données T	Réglage de Préc. Avec Gen. de Contr.	Arith. Fixe	√	√	×	√	×
[IM19]	RN Entraîné + Vect. D'Entr. + *Threshold*	Réglage de Préc. Avec Contr. Linéaires	Arith. Flottante	√	√	×	×	×
SEEDOT [GGSS19b]	RN Entraîné + Jeux de Données d'Entraînement	Calcul de Param. Avec Strat. de Compil.	Arith. Fixe	×	√	√	√	×
[LV20]	RN Entraîné	Calcul des Bornes d'Erreurs	Arith. Affine + d'Intervalles	×	√	√	√	×
Ristretto [GPMG18]	RN Entraîné + Ranges des Valeurs Numériques	Quantification	Arith. Fixe	×	√	√	×	√
DeepSZ [JDL+19]	RN Entraîné + Précision Requise	Élagage + Éval. des Bornes d'Erreurs	Arith. Flottante	×	√	√	×	√
F8Net [JRZ+22]	RN Pré-entraîné	Quantification	Arith. Fixe	×	√	√	×	√
Condensa [JGM+20]	RN Pré-entraîné + Fct. Objective	Élagage + Quantification	Arith. Flottante	×	√	√	×	√

TABLE 9.8: Comparaison entre `SyFix` et les Outils de l'État de l'Art.

Quatrième partie

POPiX : Synthèse de Code Virgule Fixe pour les Programmes

Chapitre 10

« Lorsqu'on s'occupe d'informatique il faut faire comme les canards... Paraître calme en surface et pédaler comme un forcené par en dessous. »

— RICHARD LALLEMENT

POPiX : Interface entre POP & SyFi

♣◇♣

Dans les systèmes embarqués, l'arithmétique à virgule fixe (Chapitre 2) est une alternative à l'arithmétique flottante [ANS19]. Elle est utilisée pour sa réduction de coût de calculs en termes de mémoire et de consommation d'énergie, et sa rapidité en termes de temps d'exécution. Par exemple, dans les voitures, les robots, les dispositifs médicaux, etc. Dans ce chapitre, nous présentons POPiX [BKBM22], l'outil de synthèse de code à virgule fixe pour les programmes. Ce dernier est une interface entre l'outil POP [Ben21] et la librairie à virgule fixe SyFi (Section 8.1). Il résout un système de contraintes linéaires pour obtenir les formats fixes minimaux des variables du programme. Ce Chapitre est organisé comme suit :

- La Section 10.1 met en avant le principe de la méthode de réglage des formats fixes pour les programmes par POP [Ber21]. Elle donne également un exemple illustratif pour cette méthode.

- ⁎ La Section 10.2 décrit l'outil POPiX [BKBM22]. Elle présente l'interface entre l'outil POP [Ben21] et la librairie SyFi, présentée dans la Section 8.1.
- ⁎ Les expérimentations évaluant les performances de POPiX sont présentées dans la Section 10.3. Cette dernière donne le nombre de bits optimisés, l'énergie consommée par les programmes flottants et fixes, ainsi que leurs temps d'exécution et la répartition des types de données avec précision mixte.
- ⁎ La Section 10.4 présente quelques outils effectuant également la synthèse de code à virgule fixe pour les programmes.
- ⁎ La Section 10.5 conclut ce chapitre.

10.1 Réglage des Formats Fixes des Programmes

Le principe de réglage de formats fixes par POP [Ben21] et de synthèse de code par POPiX [BKBM22], ainsi qu'un exemple illustratif, sont présentés dans cette section. Cette approche prend en entrée un programme flottant, muni d'une exigence en précision (en sortie) pour une variable (ou plusieurs), et synthétise un code à virgule fixe en C avec précision mixte, respectant l'exigence en précision donnée par l'utilisateur.

POP [Ben21] est une méthode de réglage de précision statique, basée sur une modélisation sémantique de la propagation des erreurs numériques à travers le programme. Concrètement, elle dépend de deux valeurs entières :

- L'ufp des valeurs (Définition 2.1.1),
- Le *require* défini par l'utilisateur, indiquant la précision finale souhaitée pour les sorties (une ou plusieurs variables).

Notons que, le terme précision fait référence au nombre de bits significatifs requis par l'utilisateur sur une variable du programme, noté nsb [ABM21]. Formellement, soit $\bar{x}$ l'approximation de x en précision finie, et soit $\epsilon(x)$ l'erreur absolue (Définition 2.1.3). Donc, si $\mathsf{nsb}(x) = k$, pour $x \neq 0$, alors

$$\varepsilon(x) \leqslant 2^{\mathsf{ufp}(x)-k+1} \ . \tag{10.1}$$

[Ben21] a montré qu'un problème PLNE (Section 4.1) peut être généré à partir du code source du programme, et peut être résolu de manière optimale par un solveur PL. Notons que, POP utilise GLPK [Mak12]. Concernant les types de données résultants, la caractéristique clé de POP consiste à trouver directement le nombre minimal de bits nécessaires à chaque point de contrôle du programme original. Ensuite, ces précisions peuvent être approchées au nombre supérieur de bits correspondant à un format existant int16_t, int32_t, etc. À titre d'illustration, si une variable x a $\mathsf{nsb}(x) = 18$ bits , alors x est adaptée

au format int32_t. Notons que cette technique a montré son efficacité dans des travaux antérieurs liés aux outils de réglage de précision [ABM21, Ben21]. Après avoir résolu le problème PLNE, POP obtient les formats fixes $< M, L >$, introduits dans le Chapitre 2. Enfin, nous appelons notre librairie à virgule fixe, SyFi (Section 8.1), pour synthétiser une version en virgule fixe du programme avec des entiers uniquement. Grâce à cette technique, il est possible de réaliser des économies de mémoire jusqu'à plus du double et des économies d'énergie pouvant atteindre $3,5\times$ (Section 10.3).

Langage Impératif

Comme détaillé dans [Ben21], POP utilise le langage impératif suivant :

$$x \in Id \quad \ell \in Lab \quad \odot \in \{+, -, \times, \div\} \quad math \in \{\sin, \cos, \tan, \arcsin, \log, \ldots\}$$
$$\textbf{Expr} \ni \mathbf{e} : \mathrm{e} ::= \mathrm{c\#}p \mid x \mid e_1^{\ell_1} \odot e_2^{\ell_2} \mid math(e^{\ell_1}) \mid sqrt(e^{\ell_1})$$
$$\textbf{Cmd} \ni \mathbf{c} : c ::= c_1^{\ell_1}; c_2^{\ell_2} \mid x = e^{\ell_1} \mid \texttt{while}\ b^{\ell_0}\ \texttt{do}\ c_1^{\ell_1} \mid \texttt{if}\ b^{\ell_0}\ \texttt{then}\ c_1^{\ell_1}\ \textbf{\textit{else}}\ c^{\ell_2} \mid$$
$$\texttt{create\_vector}(v, s) \mid \texttt{create\_matrix}(m, r, c) \mid \texttt{require\_nsb}(x, n)$$

On note Id l'ensemble des identifiants et Lab l'ensemble des points de contrôle du programme, utilisés pour assigner à chaque élément $e \in$ **Expr** et $c \in$ **Cmd** un point de contrôle unique $\ell \in Lab$. POP, est capable de gérer les boucles, les conditions et les tableaux. La déclaration des vecteurs est exprimée par l'instruction `create_vector`(v,s), où s désigne la taille du vecteur v. La déclaration d'une matrice m est exprimée par l'instruction `create_matrix` (m, r, c) , où r et c désignent respectivement le nombre de lignes et de colonnes de la matrice. L'instruction `require_nsb`(x,n) indique le minimum nsb n qu'une variable x doit avoir à un point de contrôle. Le reste de la grammaire est standard. Notons que les fonctions élémentaires mathématiques habituelles sont prises en charge. Pour plus de détails, le lecteur peut se référer à [Ben21, ABM21].

Fonctions de Coût

Des fonctions de coût sont données comme objectif d'optimisation au solveur linéaire. Selon la fonction de coût utilisée, différents critères peuvent être pris en compte pour la phase de réglage. POP gère actuellement les fonctions de coût suivantes [BM22] :

- **CF1** Optimise la somme du nombre de bits significatifs de toutes les variables à chaque point de contrôle du programme.
- **CF2** Optimise uniquement la somme des précisions des variables assignées dans le programme. Par rapport à **CF1**, cette fonction de coût minimise la taille des variables et le nombre de bits nécessaires aux opérateurs pour stocker les résultats intermédiaires. Notons que **CF2** est utilisée dans les expérimentations de la Section 10.3.
- **CF3** Minimise la précision maximale nécessaire dans le programme, c'est-à-dire la pire précision à un point de contrôle du programme réglé. Cette fonction est utile pour faire tenir un programme dans un certain format (par exemple toutes les variables en 16 ou 32 bits.)

```
1   [...]
2 NumTuples|10| = 40.0|5,10|;
3 create_vector(x,40);
4 create_vector(y,40);
5 create_vector(z,40);
6   [...]
7 while(i<NumTuples) {
8  x2|-6,10|=x[i]|-3,10| * x[i]|-3,10|;
9  y2|-1,11|=y[i]|-1,11| * y[i]|-1,11|;
10 z2|-6,10|=z[i]|-3,10| * z[i]|-3,10|;
11 tm|-1,10| = x2|-6,6| + y2|-1,11|
12 + z2|-6,5|;
13 magSqrt[i]|-1,10|=sqrt(tm)|-1,10|;
14   i|5,10| = i|5,11| + 1.0|0,6|;} ;
15  [...]
16 while(n<NumTuples) {
17  res|0,11| = 0.0|0,11|;
18  i|0,10| = 0.0|0,10|;
19  while (i<LpfFiltLen) {
20    if (n-i>=0.0) {
21      aux0|2,10| = lpfCoeffs[i]|2,10| *
22      magSqrt[n-i]|-1,10|;
23      res|8,10| = res|8,11| + aux0|2,9|;};
24    i|3,9| = i|3,10| + 1.0|0,7|;} ;
25    n|5,8| = n|5,9| + 1.0|0,4|;};
26    [...]
27 require_nsb(res,8);
```

```
1    [...]
2 int NumTuples; // <0,10>
3 int16_t aux0;  // <2,10>
4 int16_t res; // <8,11>
5 int16_t tm; // <1,10>
6 int8_t derivCoeffs[5]; int16_t x[40];
7 int16_t y[40]; int16_t z[40];
8    [...]
9 while(i<NumTuples) {
10  x2 = (int64_t) (x[i] * x[i]);//<-6, 20>
11  x2 = x2 >> 10;//<-6,10>
12  y2 = (int64_t) (y[i] * y[i]);//<-2,22>
13  y2 = y2 >> 10;//<-2,11>
14  y2 = y2 << 1;//<-1,11>
15  z2 = (int64_t) (z[i] * z[i]);//<-6,20>
16  z2 = z2 >> 10;//<-6,10>
17  x2 = x2 << 5;//<-1,10>
18  y2 = y2 >> 0;//<-1,10>
19  tm = x2 + y2 +z2;  // <-1,10>
20  z2 = z2 << 5; // <-6,10>
21  magSqrt[i]=sqrt_fix(tm,-1,9,8);//<-1,10>
22  i = i + 1;}
23 while(n<NumTuples ) {
24  while(i<LpfFiltLen) {
25    if(n - i >= 0) {
26      aux0=(int64_t)(lpfCoeffs[i]
27      *magSqrt[n-i]);//<1,20>
28      aux0=aux0 >> 9; // <2,10>
29      res = res >>0; // <8,10>
30      aux0 = aux0 << 6; // <8,10>
31      res = res + aux0;} //<8,10>
32 i = i + 1;} n = n + 1;} [...]
```

FIGURE 10.1: À Gauche : Programme Généré Réglé Renvoyant la Paire |ufp, nsb| en Bleue pour Chaque Variable. À Droite : Programme C Généré avec les Formats Fixes.

- **CF4** N'optimise que la somme des précisions des opérateurs arithmétiques et des fonctions élémentaires. Cette fonction est pertinente d'un point de vue matériel, par exemple pour limiter la taille des opérateurs dans les FPGA [GC15].

- **CF5** Minimise le nombre de conversions de type. En effet, les conversions de type introduites par le réglage en précision mixte peuvent ralentir l'exécution des programmes et on peut préférer un compromis entre économie de mémoire et temps d'exécution. Cette fonction résout ce problème.

Soulignons que ces fonctions de coût sont modifiées lorsqu'il s'agit des tableaux : l'outil multiplie la précision par le nombre d'éléments et ce processus n'est effectué qu'une seule fois pour chaque tableau, au lieu de plusieurs fois pour chaque utilisation de tableaux. Notons qu'une discussion et une comparaison entre ces fonctions de coût a été effectué dans [BM22].

Exemple 10.1.1. *L'exemple de filtre passe-bas FIR [BKBM22] illustre la méthode de réglage des formats fixes, décrite précédemment. Le code correspondant à ce programme est donné dans la Figure 10.1. Le point de départ de l'analyse par* `POP` *est de supposer que, toutes les variables sont dans une précision simple (Table 2.1) et qu'un range de valeurs est donnée pour les entrées du*

programme. Rappelons que, l'instruction `require_nsb(res,8)` *est une postcondition ajoutée par l'utilisateur pour spécifier que,* `res` *doit avoir au minimum* 8 *bits significatifs à la sortie du programme.* POP *effectue d'abord une détermination des ranges de toutes les variables du programme par analyse dynamique, à chaque point de contrôle. Basée sur des équations sémantiques, l'outil* POP *génère un problème PLNE (Section 4.1) à partir du code source du programme, annoté avec les résultats de l'analyse des ranges et l'exigence de précision (require). En d'autres termes, il génère un système de contraintes linéaires. Par exemple, le système S de l'Équation (10.2) décrit les contraintes générées pour les instructions d'addition et d'affectation dans la ligne* 23 *de la Figure 10.1 (côté gauche).*

$$S = \left\{ \begin{array}{l} \mathsf{nsb}(+)^{\ell_{548}} >= \mathsf{nsb}(res)^{\ell_{549}}, \mathsf{nsb}(res)^{\ell_{549}} >= \mathsf{nsb}(if)^{\ell_{551}}, \\ \mathsf{nsb}(res)^{\ell_{511}} >= \mathsf{nsb}(res)^{\ell_{545}}, \mathsf{nsb}(res)^{\ell_{545}} >= \mathsf{nsb}(+)^{\ell_{548}} + 8 + carry() - 8 \\ \mathsf{nsb}(res)^{\ell_{545}} >= \mathsf{nsb}(+)^{\ell_{548}} + 8 + carry() - 8, \mathsf{nsb}(res)^{\ell_{548}} >= \mathsf{nsb}(res)^{\ell_{549}}, \\ \mathsf{nsb}(aux0)^{\ell_{547}} >= \mathsf{nsb}(+)^{\ell_{548}} + 6 + carry() - 8, \mathsf{nsb}(aux0)^{\ell_{543}} >= \mathsf{nsb}(aux0)^{\ell_{547}} \end{array} \right\} \tag{10.2}$$

Dans un premier temps, chaque variable du programme est assignée à un point de contrôle unique $\ell \in Lab$*, afin de déterminer facilement leur nombre de bits significatifs. Par la suite, la fonction* $carry()$ *est utilisée pour calculer si un bit de retenue peut avoir lieu pendant l'opération. Elle renvoie* 0 *ou* 1*. Concernant la scalabilité, nous générons un nombre linéaire de contraintes et de variables dans la taille du programme analysé (*$\approx$ 500 *pour le code filtre passe-bas FIR). La solution à notre système de contraintes donne le nombre minimal de bits nécessaires, avec une garantie de précision sur les résultats (surlignés en bleu dans la partie gauche de la Figure 10.1). Par exemple, dans la ligne* 23*, le couple* $|8, 10|$ *dénote que la variable* `res` *a* 10 *bits significatifs et que son ufp est égal à* 8*. Les contraintes générées sont données de manière formelle dans [ABM21].*

Après avoir réglé le programme et obtenu les paires $|ufp, \mathsf{nsb}|$ *de toutes les variables, le code C donné dans la partie droite de la Figure 10.1, est synthétisé. Tout d'abord, le meilleur format (int8_t, int16_t, int32_t) pour chaque variable (appelée, précision mixte), doit être déterminé. Par exemple, dans les lignes 6 et 7, les vecteurs* `x`*,* `y` *et* `z`*, le type de données int16_t leur a été attribué, tandis que le vecteur* `derivCoeffs` *requiert int8_t comme type de données. Soulignons que le type de données sélectionné pour chaque variable est le type minimal permettant d'encoder les coefficients à virgule fixe, suivant les formats correspondants à la solution du problème PLNE. Par exemple, la variable* `res` *a* 10 *bits significatifs (lignes* 29 *et* 31 *de la Figure 10.1) et donc, elle peut avoir int16_t comme type de données. Notons que les formats* $< M, L >$ *des variables apparaissant dans le code sont déterminés, et que les alignements nécessaires pendant les opérations sont effectués. Par exemple, les décalages effectués dans les lignes* 29 *et* 30*, permettent d'aligner l'opérande de l'addition de la ligne* 31*. De même, le décalage de la ligne* 28 *est effectué, afin d'obtenir le bon format pour le résultat de la multiplication de la ligne* 27*. Actuellement, les opérations en virgule fixe sont générées séquentiellement et certaines optimisations supplémentaires pourraient être faites, par exemple en n'utilisant qu'un seul décalage pour les lignes* 27 *et* 29*.*

10.2 L'Outil POPiX

Dans cette section, nous mettons en avant l'outil de synthèse de code à virgule fixe, POPiX [BKBM22], ainsi que ces différentes étapes. Soulignons que POPiX est une interface entre l'outil POP [Ben21], présenté dans la Section 4.2 et la librairie à virgule fixe SyFi, présentée dans la Section 8.1. POPiX a été réalisé par *Sofiane Bessai*[1] [Bes22]. Il est implémenté en utilisant les deux langages : JAVA [AGH05] et C++ [Str07]. POPiX prend en entrée un programme flottant écrit en langage impératif et renvoie un code à virgule fixe en C [KR88], avec précision mixte. Il passe par quatre étapes fondamentales : entrées et *parseur*, analyse dynamique des *ranges*, génération et résolution des contraintes linéaires, synthèse de code fixe. Les étapes principales et le *workflow* de POPiX, sont donnés dans la Figure 10.2. Toutes ces étapes sont réalisées par POP [Ben21], sauf celle de la synthèse de code à virgule fixe en utilisant SyFi.

Entrées et Parseur

Dans la Figure 10.2, la première étape consiste à récupérer le programme flottant donné en entrée, muni des *require* pour une ou plusieurs variables. Ensuite, POPiX utilise l'outil ANTLR v4.7.1 [Par13] pour *parser* cette entrée.

Analyse Dynamique des Ranges

La deuxième étape consiste à effectuer une analyse dynamique pour obtenir une sous-approximation des *ranges* des valeurs, et obtenir leur ufp (Définition 2.1.1). Cette analyse garantit l'absence des *overflows* dans les calculs.

Génération et Résolution des Contraintes

Cette troisième étape est celle qui permet d'obtenir les formats fixes optimaux (minimaux) des variables. POPiX génère des contraintes linéaires pour trouver ces formats. Elles sont formellement définies dans [Ben21]. Elles sont obtenues à travers la formulation d'un problème PLNE, basé sur une modélisation sémantique des propagations des erreurs numériques dans le code. Ce problème est résolu de manière optimale avec la programmation linéaire (GLPK [Mak12]). La solution à ce problème, si elle existe, elle donne le nombre de bits minimal requis pour chaque nombre fixe, afin de satisfaire le *require* exigé par l'utilisateur.

[1] https://www.linkedin.com/in/sofiane-bessai/

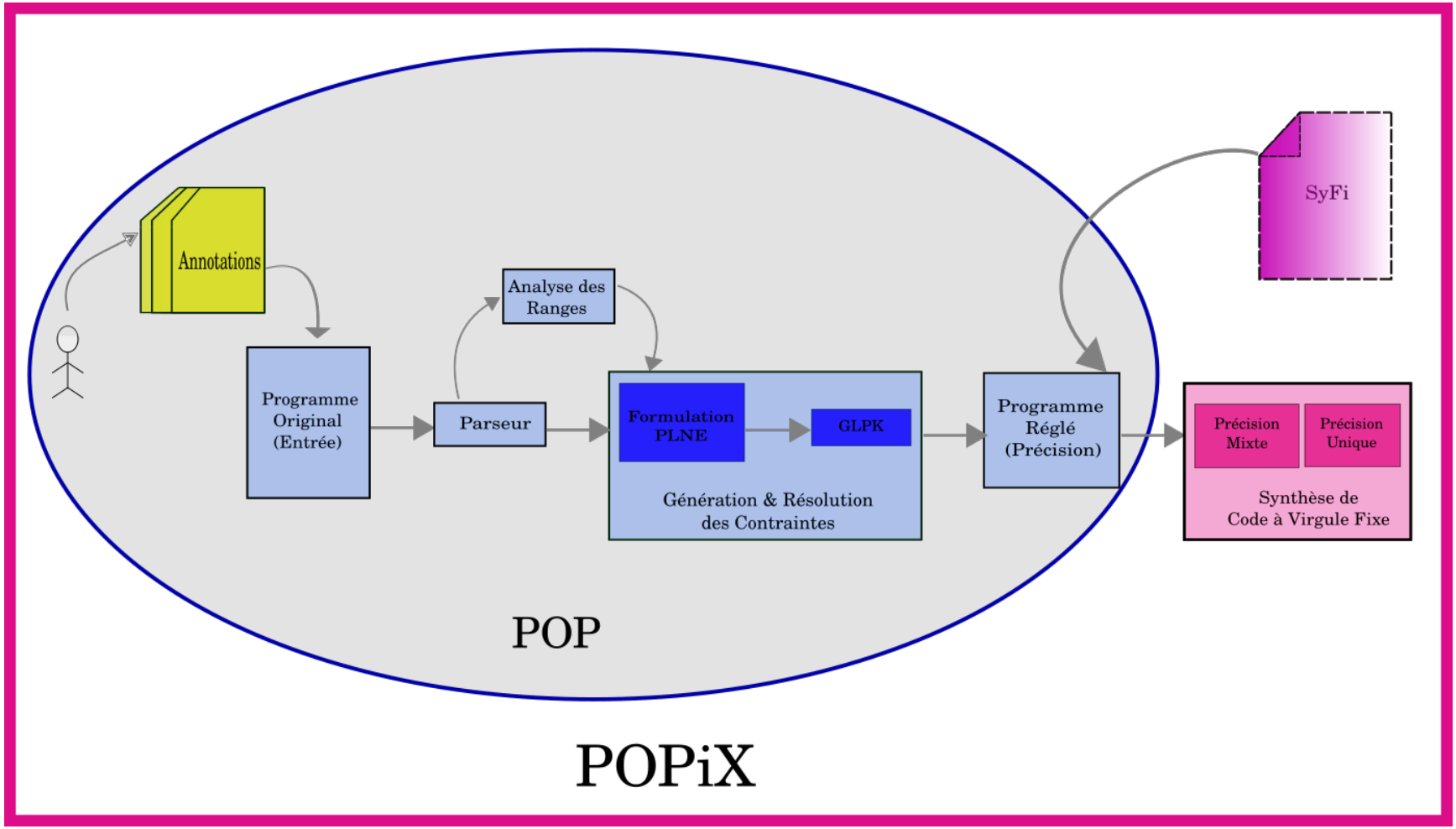

FIGURE 10.2: Workflow de l'Outil POPiX.

Synthèse de Code à Virgule Fixe

Cette dernière étape a pour but de synthétiser un code fixe correspondant au programme flottant donné en entrée, en respectant le *require* exigé par l'utilisateur. Après avoir obtenu le nombre de bits minimal pour chaque variable grâce à POP dans l'étape précédente, le format fixe peut-être déterminé. Le nombre de bits avant la virgule fixe M, est déterminé via l'analyse dynamique du code dans la deuxième étape. Et le nombre de bits après la virgule fixe L, est calculé en soustrayant M du nombre de bits total (solution du problème PLNE). Ensuite, POPiX fait appel à la librairie SyFi pour effectuer les calculs fixes. En dernier, POPiX synthétise un code fixe avec précision mixte, respectant le *require*. POPiX a également la possibilité de générer un code avec précision unique, c'est-à-dire que le même type de données est attribué à toutes les variables.

10.3 Évaluation des Performances de POPiX

Dans cette section, nous présentons les performances de POPiX, introduit dans la Section 10.2, en termes de précision mixte, de gain en mémoire et de consommation d'énergie, qui sont des mesures importantes pour valider cette méthode de synthèse de code à virgule fixe. Nous mesurons également le temps d'analyse passé par POPiX et le temps d'exécution du programme en virgule fixe par rapport au programme en virgule flottante, dans lequel nous supposons que toutes les variables sont en simple précision (Table 2.1) avant l'analyse. L'environnement expérimental et les *benchmarks* utilisés sont également présentés dans cette section. Soulignons que cette étude a été présentée dans [BKBM22, Bes22] .

10.3.1 Benchmarks et Environnement

Les *benchmarks* utilisés pour évaluer POPiX sont des programmes issus de FPBench[2]. Chaque programme est exécuté avec trois exigences de précision, choisies arbitrairement par l'utilisateur : 4, 8 et 16 bits, qui bornent l'erreur relative (Définition 2.1.4) du résultat. Nous la notons, *require*, par la suite. Toutes les expérimentations présentées dans cette section ont été effectuées sur deux machines : Ubuntu 20.04 LTS, avec un cœur $i5$ à $2,7$ GHz et 16 Go de RAM et Ubuntu 20.04 LTS, avec un CPU AMD $Ryzen$ 5 $3500u$ et $5,7$ Go de RAM. Précisons que la raison d'utilisation de la machine *Intel*, est d'exploiter l'outil Jouleit[3], afin d'estimer la consommation d'énergie du CPU, de la RAM et du GPU intégré.

[2] https://fpbench.org/
[3] https://github.com/powerapi-ng/jouleit

10.3.2 Types de Données avec Précision Mixte et Bits Optimisés

Cette sous-section présente les expérimentations concernant la mémoire optimisée (nombre de bits) et la distribution des types de données avec précision mixte dans les programmes.

La Table 10.1 montre les configurations de précision mixte obtenues après analyse (par POPiX) pour un *require* donné, en termes de nombre de variables ou d'opérations que l'on peut représenter sur int8_t, int16_t et int32_t, ainsi que le pourcentage de gain en mémoire (nombre de bits). La première colonne de la Table 10.1 donne le nom du programme passé à POPiX. La deuxième colonne, intitulée "*Nbr Fct*" fait référence au nombre de fonctions élémentaires dans le code et la colonne suivante intitulée "*Nbr Op*" indique le nombre d'opérations élémentaires. Dans ces expérimentations, 100% est supposé être le pourcentage de toutes les variables initialement représentées en simple précision (Table 2.1). Concrètement, le gain en mémoire par rapport au nombre de bits initial pour la majorité des programmes originaux est considérable, atteignant 75% pour le programme "CRadius" pour un *require* de 4 bits. Par exemple, le programme "carbonGas" a au total 13 variables, toutes représentées en simple précision initialement (avant analyse par POPiX). Et dans le code synthétisé par POPiX, il y a 7 variables représentées en int8_t et 6 variables en int16_t. Ceci engendre un gain en nombre de bits de 63, 5%. Concernant le programme "azimut", POPiX n'a pas réussi à déduire une précision mixte pour un *require* de 16 bits, alors qu'il optimise 4, 2% et 2, 8% de bits, respectivement, pour des *require* de 4 bits et 8 bits. Remarquons également que pour un *require* de 4 bits, seulement 17 variables sont représentées en int32_t. Une explication possible à ce résultat est l'appel aux fonctions élémentaires dans le code (*Nbr Fct* = 7), qui peuvent utiliser des variables intermédiaires qui demandent une plus grande précision que l'exigence de précision de l'utilisateur.

10.3.3 Énergie Consommée

Les expérimentations effectuées dans cette sous-section concernent la mesure d'énergie consommée par les programmes flottants et fixes.

La Figure 10.3 montre l'énergie consommée par l'exécution des programmes synthétisés pour chaque *benchmark*, pour un *require* de 4 bits et 12 bits, respectivement. Soulignons que les codes en virgule fixe synthétisés par POPiX, consomment significativement moins d'énergie que les codes en virgule flottante. Par exemple, l'énergie économisée sur le CPU et la DRAM atteint $\approx$ 43% pour le programme "carbonGas" et plus de 75% pour le programme "jetEngine", quand le *require* vaut 4. Notons que ce constat est également valable pour le reste des *require* avec de légères variations d'économies d'énergie (exemple : *require*=12 bits).

10.3.4 Temps d'Exécution des Programmes

Cette dernière partie des expérimentations concerne les temps d'exécution des programmes flottants et fixes ainsi que le temps d'analyse et de synthèse par POPiX.

			4 bits				8 bits				16 bits			
Programme	**Nbr Fct**	**Nbr Op**	**int8_t**	**int16_t**	**int32_t**	**%**	**int8_t**	**int16_t**	**int32_t**	**%**	**int8_t**	**int16_t**	**int32_t**	**%**
azimuth	7	7	1	0	17	**4.2**	0	1	17	**2.8**	-	-	-	**-**
carbonGas	0	7	7	6	0	**63.5**	2	11	0	**53.8**	1	1	11	**9.6**
CRadius	1	3	6	0	0	**75.0**	0	6	0	**50.0**	0	0	6	**0.0**
CTheta	1	3	4	0	3	**42.9**	0	4	3	**28.6**	0	0	7	**0.0**
doppler1	0	7	9	1	0	**72.5**	1	9	0	**52.5**	0	1	9	**5.0**
doppler2	0	7	9	1	0	**72.5**	3	7	0	**57.5**	0	3	7	**15.0**
doppler3	0	7	9	1	0	**72.5**	2	8	0	**55.0**	0	2	8	**10.0**
instantCurrent	3	18	7	14	7	**43.8**	3	18	7	**40.2**	0	3	25	**5.4**
jetEngine	0	29	7	18	5	**47.5**	3	15	12	**32.5**	0	3	27	**5.0**
LeadLagSystem	1	17	2	29	2	**48.5**	0	31	2	**47.0**	0	0	33	**0.0**
LowPassFilter	0	0	0	330	0	**50.0**	0	330	0	**50.0**	0	4	326	**0.6**
CX	1	3	2	0	4	**25.0**	0	2	4	**16.7**	0	0	6	**0.0**
CY	1	3	2	0	4	**25.0**	0	2	4	**16.7**	0	0	6	**0.0**
triangle12	1	9	7	6	0	**63.5**	0	12	1	**46.2**	0	0	13	**0.0**
turbine1	0	14	4	13	0	**55.9**	0	17	0	**50.0**	0	0	17	**0.0**
turbine2	0	10	10	3	0	**69.2**	0	13	0	**50.0**	0	0	13	**0.0**
turbine3	0	14	3	14	0	**54.4**	0	17	0	**50.0**	0	0	17	**0.0**

TABLE 10.1: Pourcentage de Bits Gagnés et Distribution des Types de Données dans le Programme Synthétisé pour un Besoin en Précision de 4, 8 et 16 bits.

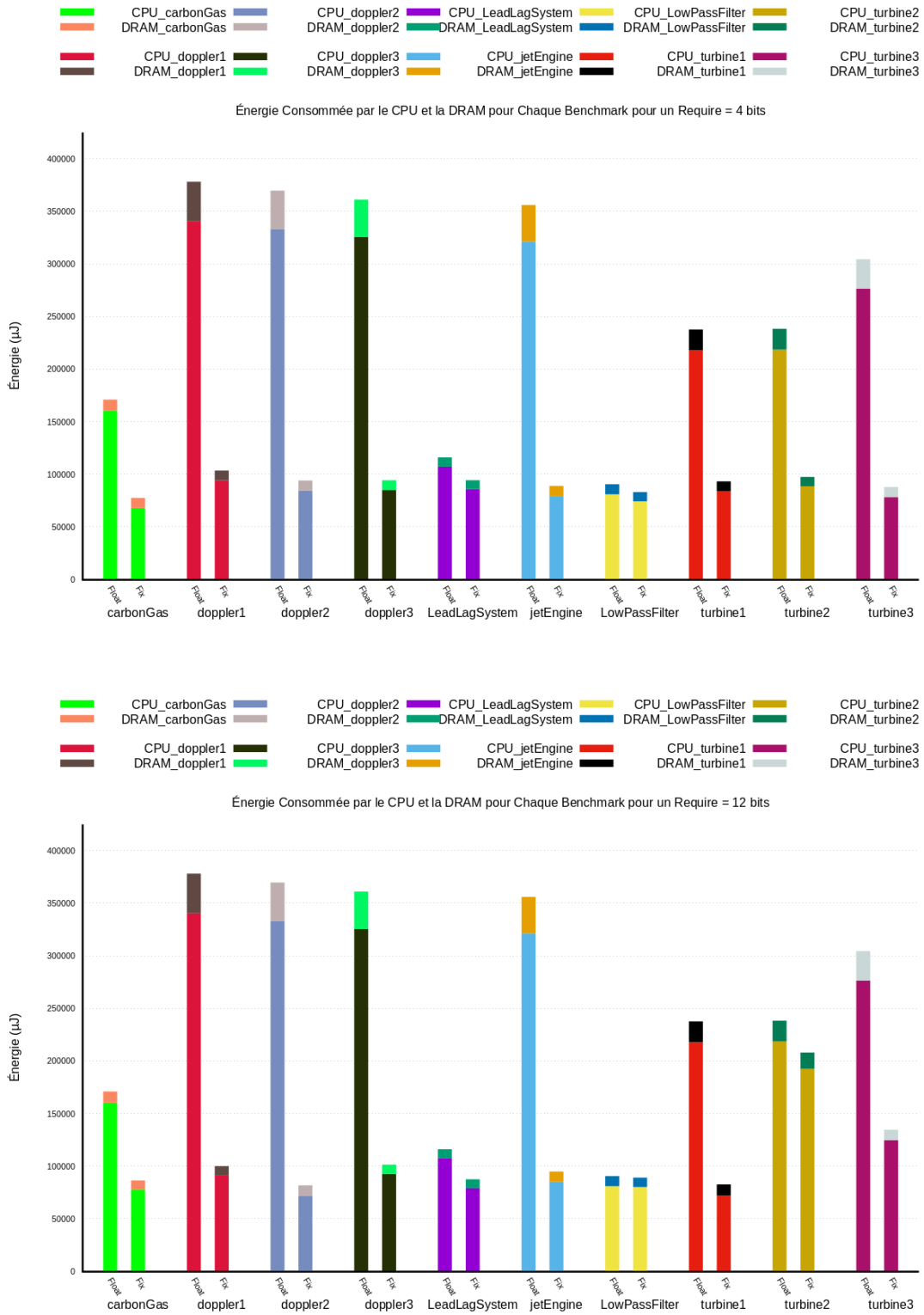

FIGURE 10.3: Mesure de l'Énergie Consommée (CPU and DRAM) par les Programmes Flottants et Fixes correspondants aux *Benchmarks*.

Programme	t_{float}	4 bits		8 bits		16 bits	
		$t_{synthesis}$	t_{fix}	$t_{synthesis}$	t_{fix}	$t_{synthesis}$	t_{fix}
azimuth	1.91	340	1.8	301	2.6	233	3.2
carbonGas	0.17	342	0.29	233	0.31	201	0.46
CRadius	0.10	240	1.31	192	1.33	186	1.36
CTheta	0.38	203	0.56	223	0.57	273	0.57
doppler1	0.16	205	0.53	249	0.54	225	0.57
doppler2	0.24	188	0.28	225	0.29	207	0.30
doppler3	0.17	243	0.30	195	0.32	207	0.34
instantCurrent	1.01	264	1.99	265	2.36	273	2.73
jetEngine	0.34	294	1.18	264	1.24	276	1.82
LeadLagSystem	0.53	352	0.71	394	0.86	346	1.09
CX	0.43	178	0.29	208	0.37	175	0.52
CY	0.39	203	0.41	197	0.55	197	0.68
triangle12	0.17	199	1.57	208	1.63	203	2.53
turbine1	0.24	220	0.31	228	0.35	269	0.49
turbine2	0.28	246	0.49	222	0.61	212	1.01
turbine3	0.48	247	0.47	253	0.49	217	0.83

TABLE 10.2: Mesure des Temps d'Exécution des Programmes Flottants et Fixes.

La Table 10.2 met en avant les résultats des temps d'exécution de chaque *benchmark*. La première colonne de cette Table donne le nom des programmes. la deuxième colonne, notée "$\mathbf{t}_{float}$", met en avant le temps d'exécution des programmes flottants. Et les colonnes "$\mathbf{t}_{synthesis}$" et "$\mathbf{t}_{fix}$" montrent respectivement, le temps de synthèse total de POPiX et le temps d'exécution des programmes en virgule fixe. Soulignons que tous les temps sont donnés en millisecondes dans la Table 10.2. Nous visualisons que le temps de synthèse de codes par POPiX pour la majorité des *benchmarks* est négligeable, n'excédant pas 352 ms pour le programme "LeadLagSystem" ($\approx$ 30 LOCs), dans le pire cas. Bien que la synthèse de code via POPiX soit rapide (quelques secondes), nous observons que le temps d'exécution reste le même pour les codes à virgule flottante et fixe avec un ralentissement négligeable pour certains benchmarks, à cause de quelques opérations nécessitant plus de cycles d'horloge dans le cas de l'entier (multiplication sur 8 bits non disponible sur *Intel* [Int22], par exemple).

Discussion

POPiX est un outil de synthèse de code fixe basé sur la génération de contraintes linéaires, pour ajuster la précision du programme donné en entrée, suivant un *require*. Il optimise jusqu'à 75% de mémoire en termes de nombre de bits utilisés pour représenter les variables, et jusqu'à 3.5$\times$ l'énergie consommée par le programme (CPU et DRAM). Bien que le temps d'exécution des programmes fixes reste le même que le flottant (ou parfois négligemment un peu élevé par rapport au flottant), le gain en mémoire et en énergie est considérable.

10.4 Outils de l'État de l'Art et POPiX

Comme décrit dans [BKBM22], l'automatisation de la synthèse de code et le réglage de précision est un sujet d'actualité, qui intéresse plusieurs chercheurs. Dans la littérature, il existe plusieurs travaux mentionnés dans le Chapitre 4, dédiés à la synthèse de code à virgule fixe et le réglage des formats fixes, tels que : TAFFO [CCCA20], DEFIS [MRS+12a], Cattaneo et al. [CBC+18], Jha [Jha11] et Darulova et al. [DKMS13]. Le point commun de POPiX avec ces travaux est la conversion des flottants en fixes de manière optimale (utiliser le moins de bits possible), ainsi que la synthèse de codes à virgule fixe, souvent dédiés aux systèmes embarqués.

TAFFO [CCCA20] est également un outil de réglage de précision, convertissant les programmes flottants en fixe. Il est basé sur l'analyse statique. Le point en commun entre POPiX et TAFFO, est l'estimation des erreurs via le processus de réglage de précision. Cependant, POPiX est plus rapide et a besoin de quelques secondes uniquement pour synthétiser un code à virgule fixe.

Le projet DEFIS [MRS+12a] propose plusieurs approches de synthèse de code fixe dédiées aux filtres linéaires, aux algorithmes de traitement de signal et aux polynômes. Par exemple, Najahi et al. [MNR14, MNR17, Naj14] ont présenté une approche automatisée pour la synthèse de code fixe pour certains blocs de base d'algèbre linéaire. Ils prennent une description mathématique du problème et la plage des variables d'entrée et, synthétisent un code fixe pour ce problème. Lopez [Lop14] s'intéresse à la transformation des filtres linéaires et des contrôleurs en fixe. Sa contribution est une analyse d'erreur complète, en ce qui concerne les nombres fixes. Il considère le calcul des formats fixes comme un problème d'optimisation convexe non linéaire d'entiers. Ensuite, Volkova [Vol17] a proposé une extension à ce travail, en s'intéressant à la classe complète des algorithmes linéaires invariants dans le temps.

Cattaneo et al. [CBC+18] proposent une approche permettant de transformer un bout de code donné en virgule flottante en un code sémantiquement équivalent, basé sur les calculs fixes. Pour effectuer la conversion des nombres flottants en fixe, cette méthode repose sur une passe de transformation de compilateur autonome implémentée dans LLVM [LA04]. Soulignons que cet outil est dédié à MIOSIX, un système d'exploitation temps réel ciblant les systèmes embarqués.

Jha [Jha11] propose un algorithme de synthèse optimale d'expressions fixes. Cet algorithme est basé sur la synthèse inductive, tandis que la méthode de réécriture d'expressions fixes proposée par Darulova et al. [DKMS13], est basée sur la programmation génétique [PLM08].

10.5 Conclusion

Dans ce chapitre, nous avons présenté `POPiX`, l'outil de synthèse de code à virgule fixe pour les programmes, réalisé par *Sofiane Bessai* à partir de `POP` et `SyFi`. Dans la Section 10.1, nous avons donné un aperçu sur le principe de la méthode de réglage des formats fixes. Ensuite, dans la Section 10.2, nous avons montré le *workflow* de `POPiX`, ainsi que ses différentes étapes pour synthétiser du code fixe. Les résultats expérimentaux ont été mis en avant dans la Section 10.3, en termes de gain en mémoire, de précision mixte, de consommation d'énergie et de temps d'exécution des programmes. En dernier, nous avons présenté quelques outils de l'état de l'art s'intéressant à la synthèse de code fixe, dans la Section 10.4.

Chapitre 11

« Pas de patience, pas de science. »

— Jean-Perre Jarroux

Conclusion et Perspectives

♣ ◇ ♣

11.1 SyFix

11.1.1 Conclusion

Les travaux de cette thèse ont porté sur la synthèse de code à virgule fixe avec précision mixte [DHS18], pour des RNs flottants entraînés, donnés en entrée.

Nous avons également vu que, la manipulation des nombres à virgule fixe ne se fait pas de manière automatique, comme pour les flottants (norme IEEE754 [ANS19]), et que leur gestion est fastidieuse. C'est pourquoi nous avons implémenté la librairie `SyFi` (Chapitre 8), permettant de convertir les nombres flottants en fixe et d'effectuer des opérations arithmétiques sur ces derniers.

L'automatisation de la synthèse de code à virgule fixe se fait également par notre outil `SyFix`, présenté dans le Chapitre 8. Ce dernier prend en entrée un seuil d'erreur (*Threshold*) et un type de données T, à ne pas excéder. Ensuite, il effectue d'abord une analyse dynamique du RN, pour déterminer les *ranges* et le nombre de bits des parties entières des coefficients, puis il génère des contraintes via l'une des trois méthodes `CubMeth`, `QuadMeth` et `LinMeth`, respectivement définies dans les chapitres 6 et 7. Ces contraintes sont résolues par le solveur `Z3` [dMB08]. Elles calculent les formats fixes $< M, L >$ minimaux, et plus précisément le nombre de bits des parties fractionnaires des neurones et des poids

synaptiques (si la solution existe). La synthèse de code à virgule fixe en C [KR88] pour le RN donné en entrée, a lieu grâce à `SyFi` et aux formats fixes minimaux, obtenus par l'une des trois méthodes. Ce nouveau RN (fixe) a le même comportement que le RN initial (flottant). Autrement dit, les sorties de la dernière couche du RN fixe, doivent être identiques à celles du RN flottant, avec une erreur de calcul maximale inférieure ou égale à la valeur du *Threshold* fixé.

Les expérimentations dans le Chapitre 9 ont également montré que la méthode de calcul des formats `CubMeth` est, la moins gourmande en termes de mémoire (elle consomme moins de bytes), mais son nombre de contraintes générées et leurs temps de résolution restent conséquents. Contrairement à `CubMeth`, les deux méthodes `QuadMeth` et `LinMeth`, consomment plus de bytes par rapport à `CubMeth`, car elles attribuent, respectivement, un format par neurone et par ligne de poids synaptiques, et un format par couche de neurones et par matrice de poids synaptiques. `QuadMeth` attribue le plus grand M de la ligne, comme nombre de bits de la partie entière à tous les poids de de la ligne, et `LinMeth` assigne le plus grand M de la matrice des poids synaptiques, comme nombre de bits de la partie entière à tous les poids de cette matrice (de même pour la couche des neurones). En revanche, cette méthode génère moins de contraintes par rapport à `CubMeth` et `QuadMeth`, et est plus précise (erreurs plus petites) que les deux autres méthodes (Figure 9.1).

Les trois méthodes proposées dans cette thèse prennent en compte la propagation des erreurs de calcul, et les erreurs de conversion du flottant au fixe, dans les contraintes qu'elles génèrent. Ces dernières sont basées sur des variables entières. Grâce à leur réglage des formats fixes, le code en virgule fixe synthétisé répond aux critères exigés (*Threshold* et type de données T) par l'utilisateur. Ce code est basé sur la précision mixte [DHS18, CBB+17], où le plus petit type de données possible, est attribué aux variables. Dans le cas de `CubMeth` et `QuadMeth`, un type de données (minimal) est attribué à chaque neurone. Par contre, dans le cas où `LinMeth` est utilisée, un type de données (minimal) est assigné à tous les neurones de la même couche.

Récapitulons les principaux problèmes résolus dans cette thèse dans ces points :

- Le réglage des formats fixes pour les RNs flottants donnés en entrée, tout en respectant le *Threshold* et le type de données T requis, avec trois méthodes `CubMeth`, `QuadMeth` ou `LinMeth`.

- Le choix de l'approximation des fonctions d'activation $\hat{Sigmoid}$ et $\hat{Tanh}$, en fixe.

- Le calcul des erreurs commises dans un RN lors de l'utilisation de l'arithmétique fixe.

- L'automatisation de la synthèse de code à virgule fixe en C avec précision mixte, pour un RN flottant donné en entrée.

11.1.2 Perspectives

Dans cette sous-section, nous mettons en avant quelques perspectives pour la continuité des travaux de cette thèse, telles que

- ⁎ S'intéresser davantage au côté *hardware* et répéter les mêmes expérimentations, présentées dans le Chapitre 9, sur des FPGA [Tri12].

- ⁎ Utiliser un autre solveur que `Z3` [dMB08] pouvant résoudre un plus grand nombre de contraintes, afin de prendre en charge les grands RNs (profonds), et tester nos approches sur des RNs industriels de taille réelle.

- ⁎ Effectuer une analyse statique pour déterminer une sur-approximation des *ranges* des entrées et des coefficients du RN, au lieu de l'analyse dynamique utilisée actuellement.

- ⁎ Considérer les autres architectures des RNs, tels que les convolutionnels, et les intégrer dans notre outil `SyFix`.

- ⁎ Proposer une autre méthode de réglage des formats fixes du RN. Par exemple, en créant des groupes de neurones dans la même couche et des groupes de poids synaptiques dans la même matrice, dans le but de trouver un compromis entre précision, mémoire consommée et nombre et temps de résolution des contraintes générées.

- ⁎ Ajouter d'autres fonctions d'activation dans `SyFi`, comme la fonction $Softmax$ [SSA17], utilisée dans les problèmes de classification multi-classe.

- ⁎ Considérer d'autres méthodes d'approximation des fonctions d'activation [ÇTG15] et les comparer avec l'approximation linéaire par morceaux, définie dans le Chapitre 3.

- ⁎ Comparer les codes flottant et fixe synthétisés par `SyFix` en termes de temps d'exécution, en considérant les cycles d'horloge pour chaque opération également.

- ⁎ Appliquer une méthode de compression (`DeepSZ` [JDL+19], `Condensa` [JGM+20], etc.) sur le RN flottant avant de le faire passer sur `SyFix`, et comparer les résultats avec et sans compression.

- ⁎ Étudier la différence entre deux versions de RNs entraînés traitant le même problème de classification ou d'interpolation, et les comparer en termes de précision et de mémoire consommée.

11.2 POPiX

11.2.1 Conclusion

Le but de POPiX est de proposer un nouvel outil permettant de transformer un programme numérique en virgule flottante donné en entrée, en un programme sémantiquement équivalent. Ce dernier est basé sur des calculs en virgule fixe avec des entiers uniquement. L'idée clé de ce travail repose sur une modélisation sémantique de la propagation des erreurs numériques à travers le programme en virgule flottante. Afin de réaliser la conversion en virgule fixe, POPiX combine deux étapes fondamentales. La première étape consiste à générer un problème PLNE à partir du programme d'origine, afin d'obtenir les formats minimaux (nombre de bits minimal avant et après la virgule fixe), qui satisfont les exigences de précision définies par l'utilisateur. Concrètement, cela se fait en raisonnant sur le bit de poids fort (ufp) et le nombre de bits significatifs des valeurs. Le problème PLNE peut être résolu de manière optimale en une seule fois par un solveur de programmation linéaire classique (PL) sans itération. À notre connaissance, ce travail est le premier à s'intéresser à la synthèse de code en virgule fixe avec analyse statique, à l'aide d'une formulation PLNE du problème. La deuxième étape de ce travail consiste à appeler en interne la librairie à virgule fixe, SyFi, pour convertir les nombres et calculs flottants en fixe, en utilisant le nombre de bits minimal des parties entières et fractionnaires. Ces derniers sont renvoyés par la première étape. POPiX a été évalué sur un ensemble de *benchmarks* provenant de la communauté FPBench [DMP+16]. Les expérimentations ont montré des résultats remarquables en termes de gain en mémoire et d'énergie économisée. Le code source de POPiX, ainsi que toutes les données et résultats présentés sont publiquement disponibles sur : https://github.com/sbessai/popix.

11.2.2 Perspectives

Comme mentionné dans [BKBM22], POPiX peut être développé dans différentes directions. Nous citons quelques une ci-dessous :

- Valider cette méthode en considérant des architectures plus courantes dans les systèmes embarqués.
- Comparer POPiX aux outils de l'état de l'art en matière de temps d'analyse, de mémoire consommée et d'économie d'énergie.
- S'intéresser au matériel plutôt qu'aux implémentations logicielles en virgule fixe, en utilisant des FPGA [Tri12]. Cibler les FPGA pour deux raisons : ce type de matériel devient de plus en plus populaire aujourd'hui et il a l'avantage de permettre des conceptions entièrement personnalisées.

- Adopter notre méthode dans des applications du monde réel en étendant `POPiX` afin qu'il puisse gérer des programmes C via une intégration à LLVM [LA04].

–Toute science crée une nouvelle ignorance.–
Henri Michaux.

Bibliographie

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow : Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[ABM21] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. Fast and efficient bit-level precision tuning. In Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi, editors, *Static Analysis - 28th International Symposium, SAS*, volume 12913 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2021.

[ACdV92] Stefan Aeberhard, Danny Coomans, and Olivier de Vel. The classification performance of RDA. *Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland, Tech. Rep*, pages 92–01, 1992.

[ACHG97] Hesham Amin, K Memy Curtis, and Barrie R Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings-Circuits, Devices and Systems*, 144(6) :313–317, 1997.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things : A survey. *Comput. Networks*, 54(15) :2787–2805, 2010.

[ANS85] Institute of Electrical and Electronics Engineers. *IEEE standard for binary floating-point arithmetic*, Std 754-1985 edition, 1985.

[ANS08] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.

[ANS19] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-point Arithmetic*, Std 754-2019 edition, 2019.

[AOS12] Semih Aslan, Erdal Oruklu, and Jafar Saniie. A high-level synthesis and verification tool for fixed to floating point conversion. In *55th IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2012, Boise, ID, USA, August 5-8, 2012*, pages 908–911. IEEE, 2012.

[Ban11] Massimo Banzi. *Getting Started with Arduino, Second Edition*. Make projects. O'Reilly, 2011.

[Bar17] Benjamin Barrois. *Methods to Evaluate Accuracy-Energy Trade-Off in Operator-Level Approximate Computing. (Méthodes pour évaluer le compromis précision-énergie dans le calcul approximatif au niveau opérateur)*. PhD thesis, University of Rennes 1, France, 2017.

[BBH+03] Prithviraj Banerjee, Debabrata Bagchi, Malay Haldar, Anshuman Nayak, Victor Kim, and R. Uribe. Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003), 8-11 April 2003, Napa, CA, USA, Proceedings*, pages 263–264. IEEE Computer Society, 2003.

[BCD+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[BEN16] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5. 2.0. open source (mixed-integer) linear programming system, 2016.

[Ben21] Dorra Ben Khalifa. *Fast and efficient bit-level precision tuning. (Analyse statique pour le réglage de la précision numérique)*. PhD thesis, University of Perpignan, France, 2021.

[Ber21] Anthony Berthelier. *Etudes techniques de compression de réseaux de neurones pour sa mise en place dans une architecture embarquée de type Smartphone*. PhD thesis, Université Clermont Auvergne, 2021.

[Bes22] Sofiane Bessai. *Synthèse de Code Virgule Fixe pour les Systèmes Embarqués*. Master's thesis, Université de Perpignan Via Domitia, 2022.

[BF01] Lionel Bening and Harry Foster. *Principles of verifiable RTL design - a functional coding style supporting verification processes in Verilog*. Kluwer, 2001.

[BGT81] Robert G. Bland, Donald Goldfarb, and Michael J. Todd. Feature article - the ellipsoid method : A survey. *Oper. Res.*, 29(6) :1039–1091, 1981.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[BKBM22] Sofiane Bessai, Dorra Ben Khalifa, Hanane Benmaghnia, and Matthieu Martel. Fixed-point code synthesis based on constraint generation. In *DASIP '22 : Workshop on Design and Architectures for Signal and Image Processing (15th edition) - in conjunction with HiPEAC 2022, Budapest, Hungary, 2022*, pages 26–34. Springer, 2022.

[BM07] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[BM22] Dorra Ben Khalifa and Matthieu Martel. Constrained precision tuning. In *8th International Conference on Control, Decision and Information Technologies, CoDIT 2022, Istanbul, Turkey, May 17-20, 2022*, pages 230–236. IEEE, 2022.

[BMS22a] Hanane Benmaghnia, Matthieu Martel, and Yassamine Seladji. Code generation for neural networks based on fixed-point arithmetic. *ACM Trans. Embed. Comput. Syst.*, 2022. Just Accepted.

[BMS22b] Hanane Benmaghnia, Matthieu Martel, and Yassamine Seladji. Fixed-point code synthesis for neural networks. In *6th International Conference on Artificial Intelligence, Soft Computing and Applications*, volume 12 of *Computer Science & Information Technology*, pages 11–30. CSCP, 2022.

[BS17] Benjamin Barrois and Olivier Sentieys. Customizing fixed-point and floating-point arithmetic - A case study in k-means clustering. In *2017 IEEE International Workshop on Signal Processing Systems, SiPS 2017, Lorient, France, October 3-5, 2017*, pages 1–6. IEEE, 2017.

[CA20] Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation : A survey. *ACM Comput. Surv.*, 53(2) :33 :1–33 :35, 2020.

[CBB+17] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Rigorous floating-point mixed-precision tuning. In Giuseppe Castagna and Andrew D. Gordon, editors,

Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 300–315. ACM, 2017.

[CBC+18] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. Embedded operating system optimization through floating to fixed point compiler transformation. In Martin Novotný, Nikos Konofaos, and Amund Skavhaug, editors, *21st Euromicro Conference on Digital System Design, DSD 2018, Prague, Czech Republic, August 29-31, 2018*, pages 172–176. IEEE Computer Society, 2018.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3) :103–179, 1992.

[CCCA20] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2) :10 :1–10 :26, 2020.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[CH17] Sanjay Churiwala and I Hyderabad. Designing with xilinx® fpgas. In *Circuits &Systems*. Springer, 2017.

[Cha14] Aymen Chakhari. *Evaluation analytique de la précision des systèmes en virgule fixe pour des applications de communication numérique. (Analytical accuracy evaluation of fixed-point systems for digital communication applications)*. PhD thesis, University of Rennes 1, France, 2014.

[Che99] Wai-Kai Chen, editor. *The VLSI Handbook*. CRC Press, 1999.

[CI18] Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev. "learning-compression" algorithms for neural net pruning. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8532–8541. Computer Vision Foundation / IEEE Computer Society, 2018.

[CM10] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[Cod11a] Google Code. Libfixmath. Available at https://github.com/PetteriAimonen/libfixmath, 2011.

[Cod11b] Google Code. Libfixmatrix. Available at https://github.com/PetteriAimonen/libfixmatrix, 2011.

[Coq19] Albin Coquereau. *[ErgoFast] Amélioration de performances du solveur SMT Alt-Ergo grâce à l'intégration d'un solveur SAT efficace. ([ErgoFast] Improving performance of the SMT solver Alt-Ergo with a better integration of efficient SAT solver)*. PhD thesis, University of Paris-Saclay, France, 2019.

[Cpl09] IBM ILOG Cplex. V12. 1 : User's manual for cplex. *International Business Machines Corporation*, 46(53) :157, 2009.

[CRRS20] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. Fast approximations of activation functions in deep neural networks when using posit arithmetic. *Sensors*, 20(5) :1515, 2020.

[ÇTG15] Onursal Çetin, Feyzullah Temurtaş, and Şenol Gülgönül. An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function. *Dicle Medical Journal/Dicle Tip Dergisi*, 42(2), 2015.

[DDF13] David Déharbe, Pablo Federico Dobal, and Pascal Fontaine. Approches formelles dans l'assistance au développement de logiciels. In *afadl*, 2013.

[DHS18] Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. In Chris Gill, Bruno Sinopoli, Xue Liu, and Paulo Tabuada, editors, *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, pages 208–219. IEEE Computer Society / ACM, 2018.

[Dik67] II Dikin. Iterative solution of problems of linear and quadratic programming. In *Doklady Akademii Nauk*, volume 174, pages 747–748. Russian Academy of Sciences, 1967.

[DIN+18] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2018.

[DKMS13] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In Rolf Ernst and Oleg Sokolsky, editors, *Proceedings of the International Conference on Embedded Software, EMSOFT*, pages 22 :1–22 :10. IEEE, 2013.

[DM18] Nasrine Damouche and Matthieu Martel. Mixed precision tuning with salsa. In Luis Gomes, Andreas Ahrens, César Benavente-Peces, and Mohammad S. Obaidat, editors, *Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, July 29-30, 2018*, pages 185–194. SciTePress, 2018.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dMB11] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories : introduction and applications. *Commun. ACM*, 54(9) :69–77, 2011.

[dMDS07] Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2007.

[DMP+16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. 2016.

[DOW55] George B Dantzig, Alex Orden, and Philip Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2) :183–195, 1955.

[DYSD14] Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In Yao-Wen Chang, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, pages 726–733. IEEE, 2014.

[Gal21] Diane Gallois-Wong. *Formalisation en Coq des algorithmes de filtre numérique calculés en précision finie. (Coq formalization of digital filter algorithms computed*

using finite precision arithmetic). PhD thesis, University of Paris-Saclay, France, 2021.

[Gar21] Rémy Garcia. *Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes. (Floating-point numbers round-off error analysis by constraint programming)*. PhD thesis, University of Côte d'Azur, Nice, France, 2021.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[GC15] Xitong Gao and George A. Constantinides. Numerical program optimization for high-level synthesis. In George A. Constantinides and Deming Chen, editors, *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 210–213. ACM, 2015.

[GGSS19a] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. Compiling KB-sized machine learning models to tiny IoT devices. In *Programming Language Design and Implementation, PLDI 2019*, pages 79–95. ACM, 2019.

[GGSS19b] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. Compiling kb-sized machine learning models to tiny iot devices. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 79–95. ACM, 2019.

[GJP+19] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. Auto-tuning for floating-point precision with discrete stochastic arithmetic. *J. Comput. Sci.*, 36, 2019.

[GMD+18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2 : safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy*, pages 3–18. IEEE Computer Society, 2018.

[GO20] LLC Gurobi Optimization. Gurobi optimizer reference manual. gurobi optimization inc, 2020.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, 1991.

[GP17] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[GPMG18] Philipp Gysel, Jon J. Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto : A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 29(11) :5784–5789, 2018.

[GR18] Hui Guo and Cindy Rubio-González. Exploiting community structure for floating-point precision tuning. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 333–343. ACM, 2018.

[Gul10] Sumit Gulwani. Dimensions in program synthesis. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24. ACM, 2010.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 1999.

[Hof71] Kenneth Hoffman. *Linear algebra*. Englewood Cliffs, NJ, Prentice-Hall, 1971.

[Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8) :2554–2558, 1982.

[IM19] Arnault Ioualalen and Matthieu Martel. Neural network precision tuning. In David Parker and Verena Wolf, editors, *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*, volume 11785 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2019.

[Int22] Intel. Intel instrinsics guide. Available at https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX,AVX2,FMA&ig_expand=95,104,124,4913&cats=Arithmetic, 2022.

[JDL+19] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. Deepsz : A novel framework to compress deep neural networks by using error-bounded lossy compression. In *In International Symposium on High-Performance Parallel and Distributed Computing, HPDC*, pages 159–170. ACM, 2019.

[JdM12] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Commun. Comput. Algebra*, 46(3/4) :104–105, 2012.

[JdM13] Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reason.*, 51(1) :79–108, 2013.

[JGM+20] Vinu Joseph, Ganesh Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5) :17–25, 2020.

[JGWD20] Sambhav R. Jain, Albert Gural, Michael Wu, and Chris Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[Jha11] Susmit Jha. *Towards Automated System Synthesis Using SCIDUCTION*. PhD thesis, University of California, Berkeley, USA, 2011.

[Jon17] Philippe Jonhston. C++11 Fixed Point Arithmetic Library. Available at https://embeddedartistry.com/blog/2017/08/25/c11-fixed-point-arithmetic-library/, 2017.

[JRZ+22] Qing Jin, Jian Ren, Richard Zhuang, Sumant Hanumante, Zhengang Li, Zhiyu Chen, Yanzhi Wang, Kaiyuan Yang, and Sergey Tulyakov. F8net : Fixed-point 8-bit only multiplication for network quantization. In *International Conference on Learning Representations*, 2022.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe : Convolutional architecture for fast feature embedding. In Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu, editors, *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 675–678. ACM, 2014.

[JSW98] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *J. Glob. Optim.*, 13(4) :455–492, 1998.

[Kah04] William Kahan. A logarithm too clever by half. Available at https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT, 2004.

[Kar08] Howard Karloff. *Linear programming*. Springer Science & Business Media, 2008.

[KK21] Sungrae Kim and Hyun Kim. Zero-centered fixed-point quantization with iterative retraining for deep convolutional neural network-based object detectors. *IEEE Access*, 9 :20828–20839, 2021.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[KYY15] Baris Karakaya, Ramazan Yeniceri, and Müstak E. Yalçin. Wave computer core using fixed-point arithmetic. In *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, Portugal, May 24-27, 2015*, pages 1514–1517. IEEE, 2015.

[LA04] Chris Lattner and Vikram S. Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[Lan19] Mike Lankamp. FPM. Available at https://github.com/MikeLankamp/fpm, 2019.

[Lop14] Benoit Lopez. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (Optimal implementation of linear filters in fixed-point arithmetic)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2014.

[LTA16] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2849–2858. JMLR.org, 2016.

[LV20] Christoph Quirin Lauter and Anastasia Volkova. A framework for semi-automatic precision and accuracy analysis for fast and rigorous deep learning. In *27th IEEE Symposium on Computer Arithmetic, ARITH 2020*, pages 103–110. IEEE, 2020.

[Mak12] Andrew Makhorin. GLPK (GNU Linear Programming Kit). Available at https://www.gnu.org/software/glpk/, 2012.

[MBB15] John McFarlane, Heikki Berg, and Torfinn Berset. Compositional Numeric Library. Available at https://github.com/johnmcfarlane/cnl, 2015.

[MBD+18] Jean-Michel Muller, Nicolas Brunie, Florent De Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic (2nd Ed.)*. Springer, 2018.

[MCS06] Daniel Ménard, Daniel Chillet, and Olivier Sentieys. Floating-to-fixed-point conversion for digital signal processors. *EURASIP J. Adv. Signal Process.*, 2006, 2006.

[Mén11] Daniel Ménard. *Contribution à la conception de systèmes en virgule fixe. (Contribution to fixed-point system design)*. 2011.

[MGG17] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. PLACID : A platform for fpga-based accelerator creation for dcnns. *ACM Trans. Multim. Comput. Commun. Appl.*, 13(4) :62 :1–62 :21, 2017.

[MHNW19] Norbert Mitschke, Michael Heizmann, Klaus-Henning Noffz, and Ralf Wittmann. A fixed-point quantization technique for convolutional neural networks based on weight scaling. In *2019 IEEE International Conference on Image Processing, ICIP 2019, Taipei, Taiwan, September 22-25, 2019*, pages 3836–3840. IEEE, 2019.

[MHSN12] Daniel Ménard, Nicolas Hervé, Olivier Sentieys, and Hai-Nam Nguyen. High-level synthesis under fixed-point accuracy constraint. *J. Electr. Comput. Eng.*, 2012 :906350 :1–906350 :14, 2012.

[MLO+18] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. ADAPT : algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 48 :1–48 :13. IEEE / ACM, 2018.

[MNR14] Matthieu Martel, Amine Najahi, and Guillaume Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In Eduardo de la Torre and Sébastien Pillement, editors, *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing, DASIP*, pages 1–8. IEEE, 2014.

[MNR17] Matthieu Martel, Amine Najahi, and Guillaume Revy. Trade-offs of certified fixed-point code synthesis for linear algebra basic blocks. *J. Syst. Archit.*, 76 :133–148, 2017.

[Moi12] Mircea Moise. A fixed point arithmetic library for spinnaker. *Masters, The University of Manchester*, 2012.

[Mon16] David Monniaux. A survey of satisfiability modulo theory. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC*

2016, Bucharest, Romania, September 19-23, 2016, Proceedings, volume 9890 of *Lecture Notes in Computer Science*, pages 401–425. Springer, 2016.

[Moo79] Ramon E. Moore. *Methods and applications of interval analysis*. SIAM studies in applied mathematics. SIAM, 1979.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4) :115–133, 1943.

[MRS+12a] Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoit Lopez, Eric Goubault, Sylvie Putot, Franck Védrine, Amine Najahi, Guillaume Revy, L. Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti. Design of fixed-point embedded systems (DEFIS) french ANR project. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25, 2012*, pages 1–2. IEEE, 2012.

[MRS+12b] Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoit Lopez, Eric Goubault, Sylvie Putot, Franck Védrine, Amine Najahi, Guillaume Revy, L. Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti. Design of fixed-point embedded systems (DEFIS) french ANR project. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25, 2012*, pages 1–2. IEEE, 2012.

[MSCS03] Daniel Ménard, Taofik Saïdi, Daniel Chillet, and Olivier Sentieys. Implantation d'algorithmes spécifiés en virgule flottante dans les DSP virgule fixe. *Tech. Sci. Informatiques*, 22(6) :783–803, 2003.

[Mul05] Jean-Michel Muller. On the definition of ulp(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, 2005.

[Mul06] Jean-Michel Muller. *Elementary functions - algorithms and implementation (2. ed.)*. Birkhäuser, 2006.

[Mé09] Daniel Ménard. Conversion en virgule fixe pour les applications en traitement numérique du signal. ARCHI, 2009.

[Naj14] Mohamed Amine Najahi. *Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks. (Synthèse de programmes certifiés en arithmètique à virgule fixe, et son application à des briques de base d'algèbre linéaire)*. PhD thesis, University of Perpignan, France, 2014.

[NNA12] Johan Nyström, Hans-Peter Nilsson, and Johan Almbladh. Fixmath - fixed point library. Available at http://savannah.nongnu.org/projects/fixmath/, 2012.

[NNF07] Zoran Nikolic, Ha Thai Nguyen, and Gene Frantz. Design and implementation of numerical linear algebra algorithms on fixed point dsps. *EURASIP J. Adv. Signal Process.*, 2007, 2007.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2) :356–364, 1980.

[NSW18] Daniel H. Noronha, Bahar Salehpour, and Steven J. E. Wilton. Leflow : Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks. *CoRR*, abs/1807.05317, 2018.

[Pal03] Samir Palnitkar. *Verilog HDL : a guide to digital design and synthesis*, volume 1. Prentice Hall Professional, 2003.

[Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4) :765–768, 1981.

[Par13] Terence Parr. The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, pages 1–326, 2013.

[PKR13] Tapan Pradhan, Bibek Kabi, and Aurobinda Routray. Fixed-point hestenes algorithm for singular value decomposition of symmetric matrices. In *Proceedings of the 2013 International Conference on Electronics, Signal Processing and Communication Systems*, 2013.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008.

[RNN+13] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious : tuning assistant for floating-point precision. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 27 :1–27 :12. ACM, 2013.

[Ros58] Frank Rosenblatt. The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6) :386, 1958.

[RRV+18] Vasanthakumar Rajagopal, Chandra Kumar Ramasamy, Ashok Vishnoi, Raj Narayana Gadde, Narasinga Rao Miniskar, and Sirish Kumar Pasupuleti. Accurate and efficient fixed point inference for deep neural networks. In *2018*

IEEE International Conference on Image Processing, ICIP 2018, Athens, Greece, October 7-10, 2018, pages 1847–1851. IEEE, 2018.

[SDDM12] Madhusmita Swain, Sanjit Kumar Dash, Sweta Dash, and Ayeskanta Mohapatra. An approach for iris plant classification using neural network. *International Journal on Soft Computing*, 3(1) :79, 2012.

[SdF03] Jorge Stolfi and Luiz Henrique de Figueiredo. An introduction to affine arithmetic. *Trends in Computational and Applied Mathematics*, 4(3) :297–312, 2003.

[SFN+22] Sangeetha Siddegowda, Marios Fournarakis, Markus Nagel, Tijmen Blankevoort, Chirag Patel, and Abhijit Khobare. Neural network quantization with AI model efficiency toolkit (AIMET). *CoRR*, abs/2201.08442, 2022.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326. ACM, 2010.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1) :1–12, 1984.

[SJRG15] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In Nikolaj S. Bjørner and Frank S. de Boer, editors, *FM 2015 : Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2015.

[SSA17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12) :310–316, 2017.

[Str07] Bjarne Stroustrup. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.

[SYK21] Heming Sun, Lu Yu, and Jiro Katto. Learned image compression with fixed-point arithmetic. In *Picture Coding Symposium, PCS 2021, Bristol, United Kingdom, June 29 - July 2, 2021*, pages 1–5. IEEE, 2021.

[TDCC17] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 1129–1139. IEEE Computer Society, 2017.

[Tom03] MT Tommiska. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings-Computers and Digital Techniques*, 150(6) :403–411, 2003.

[TR20] Nicholas Gerard Timmons and Andrew Rice. Approximating activation functions. *CoRR*, abs/2001.06370, 2020.

[Tri12] Stephen M Trimberger. *Field-programmable gate array technology*. Springer Science & Business Media, 2012.

[Vol59] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electron. Comput.*, 8(3) :330–334, 1959.

[Vol17] Anastasia Volkova. *Towards reliable implementation of digital filters. (Vers une implémentation fiable des filtres numériques)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2017.

[vR07] Guido van Rossum. Python programming language. In Jeff Chase and Srinivasan Seshan, editors, *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*. USENIX, 2007.

[WB20] Pascal Welke and Christian Bauckhage. Ml2r coding nuggets : Solving linear programming problems. Technical report, Technical Report. MLAI, University of Bonn, 2020.

[WHC+21] Peisong Wang, Xiangyu He, Qiang Chen, Anda Cheng, Qingshan Liu, and Jian Cheng. Unsupervised network quantization via fixed-point factorization. *IEEE Trans. Neural Networks Learn. Syst.*, 32(6) :2706–2720, 2021.

[WSM92] William H Wolberg, W Nick Street, and Olvi L Mangasarian. Breast cancer wisconsin (diagnostic) data set [http ://archive.(ics. uci. edu/ml/], 1992.

[Yat09] Randy Yates. Fixed-point arithmetic : An introduction. *Digital Signal Labs*, 81(83) :198, 2009.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2) :183–200, 2002.

Résumé

Minimiser la précision avec laquelle les neurones d'un réseau de neurones calculent, est un objectif souhaitable pour limiter les ressources nécessaires à son exécution. Ceci est particulièrement important pour les réseaux de neurones utilisés dans les systèmes embarqués, ayant des ressources limitées (petites mémoires, CPUs effectuant du calcul entier, etc.) Malheureusement, les réseaux de neurones sont très sensibles à la précision avec laquelle ils ont été entraînés et changer cette précision, dégrade généralement la qualité de leurs réponses. L'utilisation de l'arithmétique à virgule fixe, basée sur les entiers et les opérations entières uniquement, reste un bon compromis entre les ressources limitées des systèmes embarqués et les calculs importants des réseaux de neurones. Cependant, cette dernière est difficile à maintenir. La gestion de la virgule fixe doit se faire manuellement contrairement aux flottants, où l'exposant est géré automatiquement. Dans cette thèse, nous introduisons trois nouvelles techniques pour ajuster les formats fixes et synthétiser un code à virgule fixe pour un réseau de neurones entraîné, donné en entrée, utilisant uniquement des entiers et ayant le même comportement que le réseau flottant initial. Pour se faire, le nombre de bits avec lequel est représenté chaque neurone et chaque poids synaptique, doit être ajusté (la plus petite valeur possible), de telle sorte à ce que le nouveau réseau calcule une sortie, où l'erreur commise n'excède pas un seuil d'erreur, choisi par l'utilisateur. D'un point de vue technique, nous générons un système de contraintes avec des variables entières pouvant être résolues par un solveur SMT. La solution à ce système, si elle existe, donnera le nombre de bits minimal requis pour chaque neurone et chaque poids synaptique.

Mot clés : Arithmétique à virgule fixe, Synthèse de code, Réseaux de neurones, Précision numérique, Génération de Contraintes, Analyse statique, Solveur SMT.

Abstract

Minimizing the precision in which the neurons of a neural network compute, is a desirable objective to limit the resources needed to execute it. This is specially important for neural networks used in embedded systems, which have limited resources (small memories, CPUs performing integer computations, etc.) Unfortunately, neural networks are very sensitive to the precision in which they have been trained and changing this precision, generally degrades the quality of their answers. Using the fixed-point arithmetic, based on integers and integer operations only, remains a good trade-off between the limited resources of embedded systems and the big number of computations of neural networks. However, this arithmetic is difficult to maintain. The management of the fixed-point position must be done manually, unlike the floating-point numbers, where the exponent is managed automatically. In this thesis, we introduce three new techniques to tune the fixed-point formats and synthesize a fixed-point code for a trained neural network, given as input, using integers only and having the same behavior as the initial floating-point network. For that, the number of bits with which each neuron and each synaptic weight is represented, have to be tuned (the smallest format), so the error committed on the outputs of the synthesized neural network, must not exceed an error threshold, chosen by the user. From a technical point of view, we generate a constraint system with integer variables that can be solved by an SMT solver. The solution to this system, if it exists, gives the minimal number of bits required for each neuron and each synaptic weight.

Keywords: Fixed-point arithmetic, Code synthesis, Neural networks, Numerical accuracy, Constraints generation, Static analysis, SMT solver.