Composer une salade CAISAR avec des réseaux de neurones et de l'édition de graphe

Michele Alberti, François Bobot, Julien Girard-Satabin, Alban Grastien, Aymeric Varasse, Zakaria Chihani

Composer une salade CAISAR avec des réseaux de neurones et de l'édition de graphe

Michele Alberti, François Bobot, Julien Girard-Satabin, Alban Grastien, Aymeric Varasse et Zakaria Chihani

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

L'intégration de réseaux de neurones dans des systèmes logiciels critiques comporte un obstacle majeur : la sous-spécification intrinsèque de leur domaine d'entrée. En conséquence, la communauté de recherche se focalise sur la spécification de propriétés simples, à l'échelle de la fonction. Les méthodes formelles (et leur inséparable partenaire, les spécifications formelles) restent donc bornées à des problèmes peu expressifs. Plus spécifiquement, les langages de spécification existants peinent à exprimer des propriétés allant au-delà de la comparaison arithmétique entre variables flottantes ou impliquant la composition de fonctions. Afin de lever ces limites, nous étendons le langage de spécification de la plateforme de vérification Why3 afin de formaliser les propriétés impliquant plusieurs composants. Ce langage est implémenté au sein de la plateforme CAISAR. Nous augmentons son évaluation avec des techniques d'édition de graphe qui exploitent la structure simple du standard de représentation ONNX. L'édition automatique de graphe nous permet de créer automatiquement des réseaux encodant des spécifications directement dans leur flot de contrôle - et de les vérifier à l'aide de prouveurs dédiés sans aucune modification de leur code. Ces développements au sein de CAISAR offrent de nouvelles possibilités pour la spécification - et la vérification - de propriétés complexes, telles que des hyperpropriétés.

1 Introduction

L'objet de cette publication est l'étude des pratiques de vérification formelle des réseaux de neurones. Nous introduisons en guise préliminaires quelques notations et éléments de contexte.

Vérification formelle de réseaux de neurones

Soient $\mathcal{X} \subseteq \mathbb{R}^d$ un espace d'entrée, $\mathcal{Y} \subseteq \mathbb{R}^p$ un espace de sortie. On appelle *réseau de neurone* une fonction $f : \mathcal{X} \mapsto \mathcal{Y}$. Usuellement, f est constitué d'un enchaînement de fonctions linéaires f_l et non-linéaires f_n. Typiquement, f est constitué d'une séquence de compositions $f_l^k \circ f_n^{k-1} \circ f_l^{k-1} ... \circ f_n^0 \circ f_l^0(x)$ où k désigne le "nombre de couches". Parmi les opérations linéaires, on compte notamment la multiplication d'un vecteur par une matrice constante ou la convolution d'une matrice par un noyau constant. Une opération non-linéaire très répandue est la *Rectified Linear Unit (ReLu)* : étant donné un vecteur $x \in \mathcal{X}$ et x_i

sa $i-$ème coordonnée, $ReLu(x_i) = max(x_i, 0)$. Les paramètres des fonctions linéaires de f sont obtenus par un processus d'apprentissage automatique ; nous n'en parlerons pas ici, ce travail se concentrant sur des réseaux de neurones une fois l'entraînement terminé.

Un réseau neuronal peut-être représenté sous la forme d'un graphe de calcul ; c'est-à-dire un graphe dirigé acyclique où chaque noeud est une opération. Chaque noeud effectue un calcul sur un tableau de valeurs à plusieurs dimensions, appelé *tenseur*. ONNX [Com22] est un format de fichier standardisé pour représenter un tel graphe. Il spécifie les opérations autorisées et leur sémantique. Les calculs en ONNX supportent, entre autres, des types entiers et flottants. La taille importante des tenseurs utilisés, l'absence de régularité dans les constantes utilisés, et les non-linéarités introduites rendent l'analyse des réseaux de neurones difficiles, alors que le nombre de noeud du graphe peuvent rester raisonnable. Il a donc été nécessaire d'adapter des techniques de prouveurs existantes pour la vérification de réseaux de neurones.

La vérification formelle de réseaux de neurone est un champ scientifique florissant. Le premier travail dont nous trouvons la trace date de 2011 [PT11] ; en 2017 est présenté `Reluplex`, le premier solveur SMT dédié aux réseaux dotés de fonctions *ReLus* [Kat+17]. De ces prototypes à la peine sur des réseaux simples, une communauté s'est constituée et emparée du sujet. C'est ainsi qu'on trouve maintenant une profusion d'outils dédiés, capables de gérer des réseaux complexes et s'étendant à d'autres structures d'apprentissage machine, telles que les machines à vecteur de supports (MVS) [RZ19]. Les auteurs aiment y voir [Chi21] un parallèle avec l'évolution de la communauté des méthodes formelles. Similairement, d'outils prototypes et de langages d'entrée fragmentés, cette communauté en est arrivée à des outils capables de passer à l'échelle. Pour comparer ces outils, des compétitions telles que les *SMT* et Runtime Verification Competitions (introduisant le langage d'entrée SMT-LIB [BFT16]) se constituèrent. Ces outils matures servirent alors de base pour des plateformes de spécification et vérifications plus complexes, telles que Why3 [FP13], Frama-C [Bau+21] ou encore KeY [Ahr+16]. La compétition internationale de vérification de réseaux de neurones (*Verification of Neural Network Competition* - VNN-Comp) suit cette logique de structuration de communauté. La VNN-Comp [Bri+23] propose aux participants de soumettre des bancs d'essai au langage d'entrée normalisé, ou de lancer des prouveurs sur ces bancs d'essai. Les prouveurs sont classés en fonction de leur temps de vérification et leur correction. La VNN-Comp propose un langage de spécification commun, VNN-LIB [Dem+23], critiquable en plusieurs points.

Limitations du langage VNN-LIB

Les propriétés à vérifier au sein de la VNN-Comp sont toutes exprimées en VNN-LIB, faisant de ce langage un standard *de-facto* dont le respect est nécessaire pour participer à la compétition [1]. VNN-LIB est un sous-ensemble du langage SMT-LIB, un langage à la syntaxe basée sur les S-expressions, très largement usité dans la communauté de vérification Satisfaction Modulo Théorie (SMT). Soient $x \in \mathbb{R}^d$ (resp. $y \in \mathbb{R}^p$) un vecteur d'entrée (resp. de sortie) ; la notation x_c désignant la $c-$ième coordonnée de x, $f : \mathcal{X} \mapsto \mathcal{Y}$ un réseau de neurones, $\vec{K}$ un ensemble d'entiers (possiblement vide) et a_i, b_i, c_j, d_j des constantes réelles. Une spécification en VNN-LIB (représentée par les constantes réelles a_i, b_i, c_j, d_j et par la fonction f) est alors équivalente à la validité de la formule logique décrite par l'équation 1.

1. Dans la publication originale, une spécification est définie comme *"[...] la combinaison du modèle [au format ONNX] et la description de la propriété au format VNN-LIB"*. Dans cette publication, nous confondrons spécification et langage VNN-LIB

$$\forall x \in \mathbb{R}^d.\ y \in \mathbb{R}^p.\ \overbrace{f(x) = y}^{C} \implies$$
$$\underbrace{(\bigwedge_{i=0..d} a_i \leq x_i \leq b_i)}_{P} \implies \underbrace{\bigwedge_{j=0..p} \left(c_j \leq y_j \leq d_j \wedge \bigwedge_{k \in \vec{K}} y_p \leq y_k \right)}_{Q} \tag{1}$$

La comparaison arithmétique entre sorties n'est pas nécessaire, aussi $\vec{K}$ peut être vide. La formule précédente peut s'écrire sous la forme d'un triplet de Hoare $\{P\}C\{Q\}$, avec une petite subtilité. Bien que f soit nécessaire dans la spécification, VNN-LIB ne permet pas d'exprimer le flot de contrôle concret de f dans le fichier au format SMT-LIB. À la place, f est décrit dans un fichier séparé via le format ONNX [Com22]. Le lien entre symboles définis dans VNN-LIB et entrées et sorties concrètes du programme ONNX se fait via le nommage des variables (`x0`, ... pour les entrées et `y0`, ... pour les sorties). VNN-LIB attend des spécifications en nombres réels, ce qui en fait un fragment de la théorie SMT-LIB de l'arithmétique linéaire sans quantificateurs des nombres réels, QF_LRA. Ces choix de langage sont la conséquence directe d'une spécialisation sur un type de propriété très restreint (au détriment d'autres potentiellement d'intérêt équivalent) : les propriétés de robustesse locale autours d'un point, dont un exemple de définition est donnée équation 2.

$$\begin{aligned} \text{Soient } x \in \mathcal{X} \subseteq \mathbb{R}, \epsilon \in \mathbb{R}, \delta \in \mathbb{R}, f : x \mapsto y \text{ et une fonction de norme } \|\|_p. \\ \text{Alors } \forall x^{'} \in \mathcal{X}, \|x - x^{'}\|_p \leq \epsilon \implies \|f(x) - f(x^{'})\|_p \leq \delta \end{aligned} \tag{2}$$

Les critiques que portent les auteurs au VNN-LIB peuvent se résumer ainsi :

Limite 1 : Manque de compacité

La taille d'une spécification VNN-LIB croit linéairement avec la taille des entrées et sorties du programme, ce qui complique la composabilité de plusieurs spécifications.

Limite 2 : Limitation sur les propriétés à vérifier

Le format ONNX représente les réseaux de neurones comme des graphes orientés acycliques. Il décrit (parfois avec un curieux manque de précision [2]) un large spectre d'opérations de base. L'usage observé est de n'encoder qu'un seul réseau dans un fichier ONNX ; ce qui limite singulièrement l'expressivité de propriétés portant sur plusieurs traces d'exécutions, ou plusieurs réseaux à la fois. Prenons pour exemple une variation de la propriété 2, portant sur deux réseaux f_1 et f_2, décrite équation 3.

$$\begin{aligned} \text{Soient } x \in \mathcal{X} \subseteq \mathbb{R}, \epsilon \in \mathbb{R}, \delta \in \mathbb{R}, f_1 : x \mapsto y, f_2 : x \mapsto y, \text{ et une fonction de norme } \|\|_p. \\ \text{Alors } \forall x^{'} \in \mathcal{X}, \|x - x^{'}\|_p \leq \epsilon \implies \|f_1(x) - f_2(x^{'})\|_p \leq \delta \end{aligned} \tag{3}$$

Cette propriété n'est pas exprimable dans un seul fichier VNN-LIB, ce qui la rend concrètement hors de portée de la plupart des prouveurs existants.

Limite 3 : Manque d'expressivité sur l'espace des programmes vérifiables

Comme son nom l'indique, VNN-LIB se limite à la vérification de réseaux de neurones, excluant *a priori* les composants comme les MVS et les machines type *gradient boosting*.

2. La surprise des auteurs fut grande de ne pas trouver de spécification mathématique de l'opération de convolution dans l'opérateur Conv, pourtant très répandu dans les réseaux de neurones modernes `https://onnx.ai/onnx/operators/onnx__Conv.html`

Contributions

Dans ce travail, nous souhaitons lever ces limitations en permettant la spécification et la vérification effective de propriétés impliquant plusieurs réseaux de neurones ; ou *hyperpropriétés*. Nos contributions se structurent ainsi :

(i) nous présentons dans la section 2 un langage formel de plus haut niveau permettant de modéliser les constructions d'algèbre linéaire communes à l'apprentissage automatique et sa traduction vers la théorie QF_LRA, levant ainsi la **limitation 1** ;

(ii) nous présentons dans la section 3 des techniques automatiques d'édition de graphe bien définies, permettant d'encoder des compositions de réseaux de neurones dans un même fichier ONNX et procédons à une évaluation sur des prouveurs état de l'art des spécifications synthétisées par la plateforme dans la section 5 ; nous constatons la difficulté pratique de vérifier les propriétés ainsi générées et pointons vers des pistes d'amélioration, levant ainsi partiellement la **limitation 2** ;

(iii) nous présentons succinctement dans la section 4.4 une implémentation de la traduction de MVS vers un graphe ONNX dans la plateforme de vérification open-source CAISAR [Alb+22], levant ainsi la **limitation 3**.

2 Un langage de spécification pour les hyperpropriétés sur les modèles d'apprentissage automatique

2.1 Description du langage de spécification

En premier lieu, nous définissons la grammaire d'un langage de spécification en Figure 1. Ce langage est un fragment du langage de spécification et programmation WhyML, décrit dans la plateforme Why3 [FP13] de vérification déductive de programme. Il permet l'écriture de propriétés de plus haut niveau portant sur des modèles d'apprentissage automatique et des fonctions *abstraites* sur des vecteurs. Le langage est pur : les modèles sont considérés comme des fonctions sans effets de bord.

$$
\begin{aligned}
\langle decl\rangle &::= \textbf{type}\ \langle tId\rangle = \langle type\rangle \\
&\mid \textbf{predicate}\ \langle id\rangle\ \langle binder\rangle^* = \langle expr\rangle \\
&\mid \textbf{function}\ \langle id\rangle\ \langle binder\rangle^*\ \langle spec\rangle^* = \langle expr\rangle \\
\langle type\rangle &::= \langle tId\rangle \\
&\mid \langle type\rangle \to \langle type\rangle \\
&\mid (\langle type\rangle,\ldots,\langle type\rangle) \\
&\mid \texttt{vector}\ \langle type\rangle \\
&\mid \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{string} \\
&\mid \texttt{model} \\
\langle binder\rangle &::= \langle id\rangle \mid (\langle id\rangle : \langle type\rangle) \\
\langle spec\rangle &::= \textbf{requires}\ \{\langle expr\rangle\} \\
&\mid \textbf{ensures}\ \{\langle expr\rangle\} \\
\langle bop\rangle &::= \leq \mid \geq \mid < \mid > \\
&\mid + \mid - \mid \times \mid / \\
&\mid \wedge \mid \vee \mid \to
\end{aligned}
\qquad
\begin{aligned}
\langle expr\rangle &::= \langle id\rangle \\
&\mid \langle built\text{-}in\rangle \\
&\mid \langle expr\rangle\langle expr\rangle \\
&\mid (\langle expr\rangle,\ldots,\langle expr\rangle) \\
&\mid \textbf{let}\ \langle id\rangle = \langle expr\rangle\ \textbf{in} \\
&\mid \textbf{if}\ \langle expr\rangle\ \textbf{then}\ \langle expr\rangle \\
&\quad \textbf{else}\ \langle expr\rangle \\
&\mid \langle expr\rangle\langle bop\rangle\langle expr\rangle \\
&\mid \textbf{forall}\langle binder\rangle.\langle expr\rangle \\
&\mid \textbf{exists}\langle binder\rangle.\langle expr\rangle \\
&\mid \textbf{not}\langle expr\rangle \\
&\mid \texttt{i} \in \texttt{Integer} \\
&\mid \{\texttt{true}, \texttt{false}\} \in \texttt{Boolean} \\
&\mid \texttt{f} \in \texttt{Float} \mid \texttt{s} \in \texttt{String} \\
\langle built\text{-}in\rangle &::= \texttt{read\_model}\ \langle expr\rangle \\
&\mid \texttt{length}\ \langle expr\rangle \\
&\mid \texttt{has\_length}\ \langle expr\rangle\ \langle expr\rangle \\
&\mid \langle expr\rangle[\langle expr\rangle] \\
&\mid \langle expr\rangle @@ \langle expr\rangle
\end{aligned}
$$

Figure 1. Grammaire pour un langage de spécification de haut-niveau.

Spécifications Une spécification est une séquence de déclarations WhyML définissant types, prédicats et fonctions. En plus de ceux déjà présents en WhyML, nous considérons comme types primitifs les flottants 64 bits `float`, les tableaux de taille finie `vector`, et les fonctions représentant l'application de modèles `model`. WhyML opère une subtile distinction entre logique de programmes et expressions de programmes, distinction que nous ne présentons pas ici par simplicité. Les définitions de fonctions peuvent spécifier des contrats de fonction sous la forme de formules logiques du premier ordre en pré- et post-conditions. Nous introduisons des expressions prédéfinies supplémentaires pour faciliter l'écriture d'opérations :

— `read_model` **`s`** introduit un modèle stocké au chemin **`s`** dans une spécification ;
— `length` v renvoie la taille du vecteur v ;
— `has_length` v i est un prédicat décrivant si le vecteur v est bien de taille i ;
— $v[i]$ renvoie l'élément stocké à la $i-$ème position du vecteur v ;
— nn `@@` v renvoie un vecteur de sortie résultant de l'application du modèle nn sur le vecteur d'entrée v.

Typage Dérivant de WhyML, notre langage est typé. Il est possible de spécifier des annotations de type, mais elles ne sont pas nécessaires en règle générale, l'inférence de type étant conservée. Les règles de typage sont les mêmes qu'en WhyML, et nous ne les reproduirons pas ici. Précisons simplement que les contrats de fonctions ne peuvent être que des expressions de type `bool`. Les valeurs de type `vector` sont principalement manipulées par des opérations sur les valeurs de type `model`, aussi on considèrera très souvent des types `vector float`, soit des vecteurs contenant des valeurs flottantes. On notera qu'une limitation (commune à beaucoup de langages) porte sur l'unicité du type des vecteurs. Cela limite la possibilité de modéliser des entrées de type hétérogènes pour un modèle, mais simplifie grandement le typage et l'implémentation. Les expressions prédéfinies sont de type flèche :

— `read_model` prend un litéral de type `string` et renvoie un `model` ;
— `length` prend un `vector float` et renvoie un `int` ;
— `has_length` prend un `vector float` et un `int`, et renvoie un `bool` ;
— `_[_]` prend un `vector float` et un `int`, et renvoies un `float` ;
— `_@@_` prend un `model` et un `vector float`, et renvoies un `vector float`.

Sémantique Notre langage suit la sémantique classique donnée par la logique du premier ordre. Les expressions étant pures et totales, une spécification se réduit à une formule de logique du premier ordre. Les contrats de fonction ne sont ainsi qu'une manière de distinguer les hypothèses des conclusions dans une spécification, à la façon de la logique de Hoare pour la vérification de programmes. Les vecteurs sont des tableaux non-mutables de taille finie. La sémantique des expressions prédéfinies sont les suivantes :

— `read_model` **`s`** est une fonction des vecteurs dans les vecteurs associée au modèle stocké au chemin **`s`** (en cas de différence dans la taille des vecteurs, `read_model` **`s`** se comporte comme une fonction non-interprétée) ;
— `length` v est la valeur entière correspondant à la taille du vecteur v ;
— `has_length` v i est la valeur booléenne indiquant si le vecteur v est bien de taille i (i étant un entier) ;
— $v[i]$ est le $i-$eme élément de vecteur v, en supposant $0 \leq i \leq$ `length` v ;
— nn `@@` v est le vecteur résultant de l'application de nn sur le vecteur v.

2.2 Compilation des spécifications WhyML vers QF_LRA

Notre langage permet l'écriture de spécifications à un niveau d'abstraction du problème original. Cependant, ces spécifications de haut-niveau doivent être compilées vers des obligations de preuves équivalentes (ou au moins équisatisfiables) exprimables dans la théorie de l'arithmétique linéaire des nombres réels sans quantificateurs (QF_LRA), dont nous avons établi VNN-LIB être un fragment. Cette étape de compilation est formalisée ci-après comme une procédure à deux étapes. Tout d'abord, nous *simplifions* une spécification WhyML de φ vers φ' (*i.e.* $\varphi \rightsquigarrow^* \varphi'$). Ensuite, nous *traduisons* φ' en une expression équisatisfiable φ'' en QF_LRA (*i.e.* $\varphi' \simeq \varphi''$).

Simplification L'étape de simplification substitue les expressions prédéfinies que nous avons ajoutées par des primitives WhyML ; cette simplification est rendue possible par l'extension du moteur de réduction de Why3 des formules de premier ordre. Un aperçu des règles de réduction est disponible 2 ; ces règles s'appliquent dans tout contexte dans une formule. La règle 4a applique des quantificateurs universels sur des vecteurs $\langle id \rangle$ pour lesquels une expression `has_length` $\langle id \rangle$ `i` est présente dans la sous-expression immédiate. Cette règle résulte dans la sous-expression $\langle expr \rangle$ dans laquelle $\langle id \rangle$ est substituée par un vecteur concret $(x_0, \ldots, x_{i-1})$ de taille i, où $x_0, \ldots, x_{i-1}$ sont des variables fraiches universellement quantifiées [3]. La règle 4b réduit l'application d'un modèle d'une constante non-interprétée nn sur un vecteur $(x_0, \ldots, x_{m-1})$ vers un vecteur $(y_0, \ldots, y_{n-1})$, avec $y_0, \ldots, y_{n-1}$ des variables fraiches universellement quantifiées au *top-level* de la formule. Cette règle ne s'applique que quand le vecteur d'entrée a une longueur compatible au flot de contrôle concret du modèle. Des réductions comparables sont présentes dans d'autres outils tels que Z3 [dB08], CVC4 [Bar+11] et SMTCoq [Eki+17].

$$\forall \langle id \rangle.\ \texttt{has\_length}\ \langle id \rangle\ \texttt{i} \to \langle expr \rangle \rightsquigarrow \forall x_0, \ldots, x_{i-1}.\ \langle expr \rangle \{ \langle id \rangle := (x_0, \ldots, x_{i-1}) \} \quad (4a)$$

$$nn\ @@\ (x_0, \ldots, x_{m-1}) \rightsquigarrow (y_0, \ldots, y_{n-1}) \quad (4b)$$

Figure 2. Règles de réduction sur les expressions.

Traduction La deuxième étape vérifie d'abord, à l'aide d'un filtrage de motifs syntaxiques, si la formule simplifiée φ' contient des expressions arithmétiques linéaires et des quantificateurs non alternants, et effectue une simple traduction le cas échéant. En effet, les expressions linéaires quantifiées existentiellement sont trivialement équisatisfiables avec les expressions QF_LRA. Les expressions linéaires universellement quantifiées peuvent l'être en les niant d'abord. Enfin, les formules contenant les deux quantificateurs peuvent d'abord être décomposées en plusieurs formules disjointes, si les quantificateurs ne s'alternent pas, chacune étant soit quantifiée de manière existentielle, soit quantifiée de manière universelle, et pouvant ensuite être traduite individuellement.

Il est possible de démontrer que cette procédure de compilation est correcte. En raison de la granularité des motifs syntaxiques filtrés, elle n'est pas complète.

3 Intégrer la spécification dans le flot de contrôle par réécriture automatique de graphe

Une spécification VNN-LIB nécessite un réseau écrit au format ONNX. La majeure partie des prouveurs spécialisés réseaux de neurones supportent que des fragments de VNN-LIB (le plus souvent une conjonction d'inégalités entre constantes et variables d'entrée) ou d'ONNX.

3. Nous nous autorisons ici un léger abus de syntaxe, en écrivant les vecteurs comme des tuples bien qu'il s'agisse de tableaux de taille finie.

3.1 Préliminaires sur ONNX

ONNX est un format de représentation de réseaux de neurones décrit via le langage d'interface *Protocol Buffer*[4]. Le support ONNX est assuré au sein des bibliothèques d'apprentissage classiques telles que PyTorch ou TensorFlow, ou via un *runtime* dédié[5]. Un graphe ONNX peut se voir comme une fonction non-linéaire sur des tableaux multidimensionnels. On appellera ces tableaux des *tenseurs* ou *vecteurs*, de manière interchangeable, dans la suite de cette section.

ONNX décrit un graphe acyclique orienté dont les noeuds représentent des transformations sur les tenseurs (les *operateurs* suivant la terminologie ONNX), les arêtes décrivant le flot de données. Dans la suite, on appellera *fonction neuronale* une fonction représentable en ONNX. À l'heure de rédaction de ce travail,ONNX spécifie 193 opérateurs[6]. Nous fournissons ci-après une sémantique informelle de ceux nécessaires à la compréhension de notre travail. **`Add`**(x, y) (resp. **`Sub`**, **`Mul`**, **`Div`**) définit l'addition (resp. la soustraction, la multiplication et la division) éléments par éléments des tenseurs x et y supposés de même dimension. **`Concat`**(x, y) concatène au tenseur x le tenseur y. **`Gather`**(x, y) procède à une sélection d'indices de x suivant les valeurs de y, produisant un tenseur $z_i = x_{y_i}$. Enfin, **`Input`** marque l'entrée du graphe[7].

3.2 Exemple

Soient nn_1 et nn_2 deux fonctions neuronales avec une (resp. deux) entrées et une sortie.

Considérons l'expression suivante, où @@ représente l'application d'une fonction neuronale sur un tenseur de taille valide comme défini en Section 2 :

$$nn_2@@(nn_1@@(x_1), x_1 + \epsilon) + nn_1@@(x_0)$$

nn_1 est évalué sur deux entrées différentes x_0 et x_1. nn_2 est évalué sur une des évaluations de nn_1 et l'entrée $x_1 + \epsilon$.

Soit H une formule QF_LRA définissant des bornes sur x_1, x_2, ϵ. Une spécification que l'on pourrait souhaiter vérifier serait alors celle décrite en Équation 5.

$$\forall x_0, x_1, \epsilon.\ H(x_0, x_1, \epsilon) \rightarrow nn_2@@(nn_1@@(x_1), x_1 + \epsilon) + nn_1@@(x_0) > 0 \tag{5}$$

Cette formule présente plusieurs difficultés pour VNN-LIB et ONNX :

1. une des entrées est le résultat d'un calcul linéaire ;
2. elle comporte plusieurs réseaux ;
3. elle effectue une comparaison arithmétique sur une composition de fonctions neuronales.

Ces limitations résultent d'une tension entre la faible expressivité de VNN-LIB sur les entrées (item 1 et item 3) et le manque de composabilité de ONNX (item 2 et item 3).

Nous proposons de lever ces limites en réécrivant la spécification en la suivante Équation 6.

$$\forall x_0, x_1, \epsilon.\ H(x_0, x_1, \epsilon) \rightarrow nn_3@@(x_0, x_1, \epsilon) > 0 \tag{6}$$

Dans cette spécification, on définit une nouvelle fonction neuronale nn_3 qui intègre au sein de son flot de contrôle les calculs sur les entrées, la composition de réseau et la comparaison arithmétique, basé sur les entrées initiales x_1, x_2 et la constante ϵ. Une représentation graphique du flot de contrôle de nn_3 est présenté Figure 3. Plus précisément, nn_3 encode les opérations suivantes :

4. https://protobuf.dev/
5. `https://onnxruntime.ia/`
6. `https://github.com/onnx/onnx/blob/main/docs/Operators.md`
7. Dans le format ONNX original, les entrées du graphe de calcul peuvent être spécifiés dans les métadonnées du graphe. Nous définissons le noeud **`Input`** par simplicité des formules.

1. distribuer x_0 et x_1 sur des évaluations séparées de nn_1 ;
2. effectuer l'addition de ϵ à x_1 ;
3. créer un noeud combinant $nn_1@@(x_1)$ et $x_1 + \epsilon$;
4. effectuer le calcul de nn_1 et nn_2 sur leurs entrées respectives ;
5. ajout des sorties.

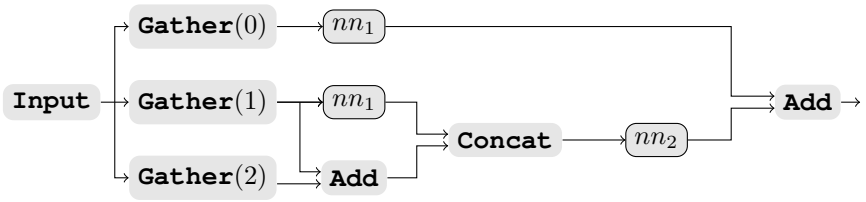


Figure 3. **Gather**(0) (resp. **Gather**(1)) extrait x_0 (resp. x_1) et **Gather**(2) extrait ϵ depuis le noeud d'entrée. Le premier noeud **Add** calcule $x_1+\epsilon$. Les noeuds nn_1 nn_2 représentent les flots de contrôle de nn_1 et nn_2 inlinés. Enfin, **Concat** prépare les deux entrées requis par nn_2.

Cet exemple illustre comment nous levons les limitations sous-jacentes à VNN-LIB et ONNX. Par l'intégration de plusieurs réseaux au sein d'un même graphe ONNX, nous autorisons des spécifications comportant plusieurs fonctions neuronales. En intégrant des fragments de formules dans le flot de contrôle du programme, nous autorisons des spécifications plus complexes. Enfin, les entrées et sorties des fonctions neuronales n'ont plus besoin d'être quantifiées explicitement dans les formules.

3.3 Composition de fonctions neuronales

Étant donné une formule initiale exprimée dans le langage défini Section 2, l'objectif est d'en transformer une partie en sous-graphes ONNX. On suppose des formules WhyML qui ne contiennent que des variables quantifiées universellement au *top-level* après simplification. Ces variables constituent les entrées et les sorties des fonctions neuronales. Dans la suite, on considère des fonctions neuronales pures, sans effet de bord. Une première passe identifie les applications de fonction neuronale de la forme $s_i = f@@x$. Chaque sous-expression s_i se voit assigner un indice unique, comme illustré Figure 4.

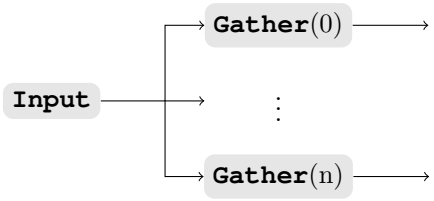


Figure 4. Décomposition d'une entrée tensorielle en scalaires, correspondant chacun à une variable définie dans une expression WhyML.

Une fois ces sous-expressions identifiées, il s'agit de créer des noeuds ONNX qui leur sont sémantiquement équivalentes. Dans ce travail, nous nous limitons à des opérations QF_LRA, aussi il existe une transcription directe de ces calculs. Enfin, les entrées des réseaux de neurones originaux sont édités pour intégrer les noeuds ainsi créés, comme illustré Figure 5. Dans le cas de la composition de réseaux de neurones $nn_i@@nn_j@@...$, on substitue le noeud d'entrée de nn_i par le noeud de sortie de nn_j.

La Figure 6 décrit de manière synthétique la transformation des sous-expressions WhyML en noeuds ONNX.

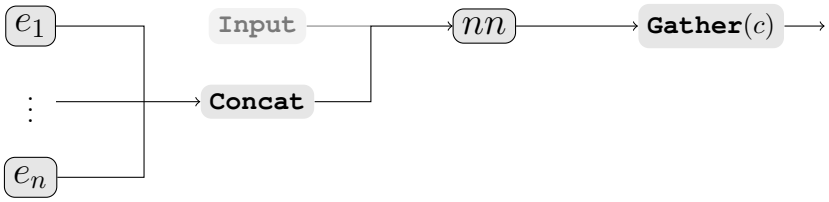


Figure 5. Conversion de l'expression WhyML $nn@@(e_1, \ldots, e_n)[c]$, avec c un litéral entier. Ici, le noeud (e_i) décrit le résultat de la transformation de l'expression Why3 e_i. Le noeud **Input** original est remplacé par la concaténation des résultats de transformations $e_1, \ldots, e_n$.

$$
\begin{aligned}
[\![e_1 + e_2]\!]_M &= \texttt{Add}([\![e_1]\!]_M, [\![e_2]\!]_M) \\
[\![e_1 - e_2]\!]_M &= \texttt{Sub}([\![e_1]\!]_M, [\![e_2]\!]_M) \\
[\![e_1 * e_2]\!]_M &= \texttt{Mul}([\![e_1]\!]_M, [\![e_2]\!]_M) \\
[\![e_1/e_2]\!]_M &= \texttt{Div}([\![e_1]\!]_M, [\![e_2]\!]_M) \\
[\![-e_1]\!]_M &= \texttt{Mul}([\![-1.]\!]_M, [\![e_2]\!]_M) \\
[\![\langle id \rangle]\!]_M &= \texttt{Gather}(\texttt{Input}, M(\langle id \rangle)) \\
[\![nn@@(e_1, \ldots, e_n)[c]]\!]_M &= \texttt{Gather}(\texttt{Apply}(nn, \texttt{Concat}([\![e_1]\!]_M, \ldots, [\![e_n]\!]_M)), c) \\
\texttt{Apply}(nn, g) &= Parser(nn)[\texttt{Input} \leftarrow g]
\end{aligned}
$$

Figure 6. Transformations d'expressions WhyML en noeuds ONNX. Tous les noeuds créés considèrent des tenseurs de taille 1. Une transformation $[\![\cdot]\!]_M$ est paramétrisée par un *mapping* des symboles d'entrée $\langle id \rangle$ vers les indices des entrées. Pour des réseaux nn_1 et nn_2, $nn_1[\texttt{Input} \leftarrow nn_2]$ construit une copie de nn_1 où le noeud d'entrée de g_1 est remplacé par nn_2. On suppose l'existence d'une fonction globale *Parser* qui associe un nom de fichier ONNX à une fonction neuronale.

Cette transformation permet d'élargir les propriétés qu'il est possible de vérifier avec les prouveurs états de l'art. Elle permet d'étendre VNN-LIB avec les propriétés intégrant des calculs QF_LRA sur les entrées et sorties et la composition de fonctions neuronales, en encodant directement ces éléments dans le graphe ONNX. Des expressions intégrant des structures de contrôle pourraient aisément être encodées ; à titre d'exemple, ONNX fournit un opérateur **If** encodant les conditionnelles. Cet opérateur n'est cependant pas supporté par la plupart des prouveurs, aussi les techniques à base de *splitting* semblent mieux indiquées pour effectuer des raisonnements plus précis. Dans le même ordre, la traduction de formules Why3 (qui travaillent sur une représentation plate des données) se fait par l'adjonction de nombreux noeuds dans le graphe ONNX ; générant beaucoup d'opération différente sur une seule dimension. Les solveurs spécialisés réseaux de neurones étant généralement optimisés pour des graphes comptant peu de noeuds avec des grandes dimensions, on rencontre une limitation. Ce choix de conception de la réécriture est à raffiner : savoir quand regrouper les opérations (avant ou après transformations) n'est pas évident.

Nous nous sommes limités aux opérations arithmétiques simples typiques de QF_LRA. ONNX fournissant une grande quantité d'opérations, notre transformation peut aisément être étendue. Cependant l'existence de l'opérateur en ONNX n'implique pas leur traitement dans les prouveurs spécialisés, encore moins leur traitement efficace. Donc, un équilibre doit être trouvé entre, d'une part, l'expressivité offerte à l'utilisateurůice dans la spécification et, d'autre part, la performance des prouveurs spécialisés sur les réseaux encodant ladite spécification. Les termes de cet équilibre constituent une piste de recherche que nous jugeons pertinente à poursuivre avec les développeurs de prouveurs.

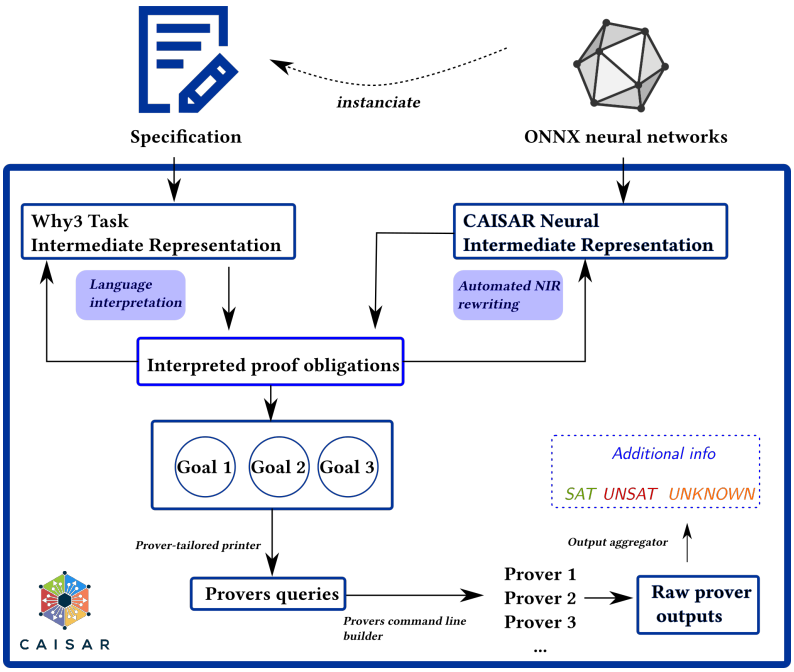


Figure 7. Architecture fonctionnelle gros grains de CAISAR. Les parties *Language interpretation* et *Automated NIR rewriting* sont des extensions significatives de la plateforme Why3.

4 Implémentation dans la plateforme CAISAR

CAISAR (Characterization of Artificial Intelligence Safety and Robustness) est une plateforme logicielle développée au CEA List. Son objectif est de fournir un point d'entrée unique pour la spécification et vérification de programmes synthétisés par apprentissage automatique. À cette fin, CAISAR se repose sur la plateforme de vérification de programmes Why3. Elle étend le langage de spécification originel comme décrit Section 2 et bénéficie de toute la machinerie de traduction automatique de formule vers les prouveurs. En plus des prouveurs originellement supportés par Why3, CAISAR supporte dix outils de vérification dédiés aux programmes appris.

La plateforme est un logiciel libre disponible publiquement (licence LGPL-v2) [8]. Cette section décrit l'implémentation concrète de la réduction de spécification WhyML (*cf.* Section 2) et l'édition automatique de graphe (*cf.* Section 3) au sein de CAISAR ; qui représentent des améliorations substantielles de la plateforme. La Figure 7 présente l'architecture de la plateforme mise à jour, en soulignant les apports de nos travaux.

4.1 Implémentation de la stratégie de réduction

Why3 représente les spécifications WhyML comme des structures `task`. Une `task` comprend notamment la spécification (comprise comme la formule écrite par l'utilisateurůice et toutes les définitions importées), des informations additionnelles (`meta`) utilisées pour guider les prouveurs et un registre de symboles. Why3 permet la réécriture de `tasks` via des modules OCaml `transformation`. Un module définit les transformations à effectuer sur la tâche par filtrage de motif. En accord avec un style fonctionnel, il est possible de composer les `transformations` et de les appliquer à différentes `task` [9].

8. `https://git.frama-c.com/pub/caisar/`
9. Cf. `https://www.why3.org/api/Trans.html` pour l'API complète.

La réduction de formule est implémentée via une série de `transformations`. Why3 fournit un moteur d'exécution, permettant d'évaluer des fragments des spécifications. Nous avons ajouté à ce moteur la définition de symboles `built-in` décrits Figure 1. Ainsi, la fonction `length` est évaluée comme sa valeur de retour concrète, valeur qui est alors substituée dans la formule. Les symboles d'entrée et de sortie des réseaux sont stockés (et mis à jour) comme des `meta`.

4.2 Représentation intermédiaire neuronale

Pour représenter un réseau ONNX, nous définissons une représentation neuronale intermédiaire (*Neural Intermediate Representation (NIR)*). Une NIR est un graphe orienté acyclique immutable implémenté comme un type variant OCaml pur, rendant aisé son parcours et l'application de transformation. Chaque noeud définit son prédécesseur, le graphe étant alors défini récursivement depuis son noeud de sortie (supposé unique). Les opérations des noeuds décrits imitent ceux définis dans la référence ONNX version 8 [Com22].

On construit une NIR via une interface générée automatiquement à l'aide de la bibliothèque ocaml-protoc-plugin. Cette NIR permet une représentation de formats variés de réseaux de neurones (dont un sous-ensemble de ONNX), ainsi que des machines à vecteur de support.

4.3 Édition automatique de graphes

Une fois la `task` simplifiée par le moteur d'exécution de Why3 et les NIRs construites pour chaque fonction neuronale, nous identifions les sous-expressions pouvant être converties par filtrage de motif. Plus spécifiquement, Why3 offre des fonctions de parcours des formules définies dans une `task`. Les expressions d'une `task` sont décrites via un type variant défini dans le module Why3 `Term`, sur lequel s'opère le filtrage.

De ces sous-expressions, on construit des noeuds NIR adaptés, noeuds qui sont ensuite ajoutés aux NIR existantes pour en produire une nouvelle. Les sous-expressions ainsi intégrées à la NIR sont retirées de la `task` originale.

Initialement, nous utilisions le noeud ONNX **`Gather`** de manière intensive. Cependant, des expériences préliminaires ont montré que les prouveurs ne supportent pas encore cet opérateur. Dans l'implémentation actuelle, **`Gather`** est encodé comme une succession de multiplications matricielles. Les réseaux ONNX résultants sont alors de très grosse taille, ce qui peut entraîner une sous-performance des prouveurs. Nous comptons signaler cette limitation aux auteurs des principaux prouveurs état de l'art.

4.4 Conversion depuis des machines à vecteur de supports

L'outil SAVer accepte les MVS dans le format OVO. Ce format utilise le format CSV et est seulement spécifié dans les grandes lignes. À l'aide d'expérimentations, nous avons pu documenter le format de sorte que CAISAR soit capable de lire des OVO ayant un noyau linéaire ou polynomial. La transformation en un graphe ONNX est assez directe et utilise les fonctions arithmétiques simples ainsi que le noeud **Sign**. La transformation ne respecte pas forcément l'ordre des opérations flottantes, l'ordre des opérations n'étant pas spécifié dans le format initial.

5 Évaluation

Nous présentons dans cette section la capacité effective de CAISAR à générer des obligations de preuve comprenant des ONNX édités selon la procédure décrite Sections 3 et 4 à destination des prouveurs état de l'art spécialisés en réseaux de neurones. Cette capacité effective, comme on le verra le long de la partie, reste limitée. Les prouveurs évalués sont le plus souvent incapables d'évaluer les propriétés fournies, respectant pourtant le standard ONNX.

```
1   let constant eps : t
2   predicate valid_input (i: input) =
3     (-5.0:t) .≤ i[0] .≤ (5.0:t)
4     ∧ (-5.0:t) .≤ i[1] .≤ (5.0:t)
5     ∧ (-5.0:t) .≤ i[2] .≤ (5.0:t)
6
7   let P (i: input) : t
8     requires { has_length i 3 }
9     requires { valid_input i }
10    ensures { result .- i[0] .+ i[1] .+ i[2] .≤ eps } =
11    (nn @@ i)[0]
```

Figure 8. Une spécification WhyML décrivant la propriété $y - x_1 - x_2 - x_3 \leq \varepsilon$ et sa satisfaction par une fonction neuronale *nn*.

Nous effectuons cette évaluation sur les prouveurs suivantsă: nnenum [Bak21], Marabou [Kat+19] et PyRAT [Dur+22] ; ces outils sont tous des participants depuis plusieurs années à la VNN-Comp et sont déjà supportés par CAISAR. Nous souhaitions évaluer le vainqueur des éditions 2023 et 2024 de la VNN-Comp, *i.e.* α-β-CROWN [Wan+21]. Malheureusement, l'outil s'est avéré incapable de *parser* les réseaux modifiés par nos soins. Nous avons identifié l'origine du soucià: un package `onnx2pytorch` non mis à jour depuis 2021. Il est prévu que nous contactions les auteurs de α-β-CROWN afin de trouver une solution. Dans la suite de cette section, les configurations spécifiques que nous avons utilisé pour les prouveurs sont disponibles sur notre forge logicielle. Nous ne présenterons que des fragments jugés importants des spécifications WhyML ; nous ne mentionnerons pas les fonctions de lecture de réseaux de neurones, jeu de données ou initialisation de constantes. Nous avons mené toutes les expériences sur un ordinateur Precision 5530 doté d'un processeur Intel Core i7-8850H CPU @ 2.60GHz et de 32 GiO de mémoire vive.

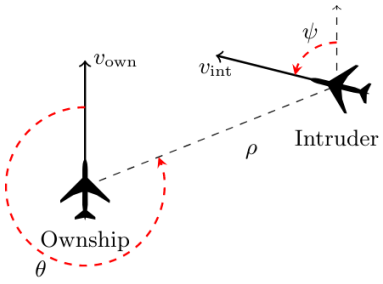
5.1 Calculs sur les entrées et sorties

Comme vu plus tôt, VNN-LIB limite les spécifications sur les entrées et sorties comme des séries de contraintes arithmétiques avec des constantes. Il n'est pas possible de spécifier des opérations arithmétiques simples entre entrées et sorties. Avec CAISAR, nous pouvons écrire ces calculs dans une postcondition WhyML ; l'édition automatique de graphe intègrera cette postcondition directement dans le flot de contrôle du réseau ONNX.

Notre première expérience porte sur un réseau jouet, appris sur un jeu de donnée synthétique vérifiant la propriété $y = x_1 - x_2 - x_3$. Nous reformulons la propriété pour prendre en compte d'éventuelles imprécisions sur les entrées. Soit $\varepsilon \in \mathbb{R}$, alors $y - x_1 - x_2 - x_3 \leq \varepsilon$.

La Figure 8 décrit un fragment d'un module WhyML décrivant le problème. Le type `t` désigne les flottants respectant le standard `IEEE 754`, préexistant dans Why3. Le type `input` désigne des vecteurs de type `t`. Le **`let`** ligne `1` définit une constante `eps` de type `t`, à la OCaml. Le prédicat `valid_input` ligne `2` assure que les trois composantes de l'entrée `input` sont comprises entre `-5.0` et `5.0`. La fonction `P` lignes `7`–`11` renvoie la première composante du résultat de l'application de la fonction neuronale `nn` sur l'entrée `i`. Cette spécification s'assure que le résultat de `P` reste proche de la valeur attendue, bornée par une erreur `eps`. Les préconditions sont définies par les clauses **`requires`**. Ligne `8`, `has_length i 3` assure que le vecteur `i` a la bonne longueur (ici, 3). Ligne `9`, `valid_input` assure que les composantes de `i` sont bien dans les bornes. La postcondition définie ligne `10`, dans la clause **`ensures`**, s'assure que le résultat de `P` vérifie la propriété. Enfin, la ligne `11` décrit le corps de la fonctionă: l'application de la fonction neuronale `nn` à l'entrée `i` et la sélection de sa première composante.

nnenum et Marabou échouent à lire le réseau ONNX généré, à cause d'une levée d'exception portant sur les noms des noeuds au sein du graphe. PyRAT prouve la violation de la propriété en 34.78 secondes en moyenne (sur trois évaluations).



Property ϕ_1.

- Description: If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold.
- Tested on: all 45 networks.
- Input constraints: $\rho \geq 55947.691$, $v_{\text{own}} \geq 1145$, $v_{\text{int}} \leq 60$.
- Desired output property: the score for COC is at most 1500.

Property ϕ_2.

- Description: If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal.
- Tested on: $N_{x,y}$ for all $x \geq 2$ and for all y.
- Input constraints: $\rho \geq 55947.691$, $v_{\text{own}} \geq 1145$, $v_{\text{int}} \leq 60$.
- Desired output property: the score for COC is not the maximal score.

Property ϕ_3.

- Description: If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.
- Tested on: all networks except $N_{1,7}$, $N_{1,8}$, and $N_{1,9}$.
- Input constraints: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi \geq 3.10$, $v_{\text{own}} \geq 980$, $v_{\text{int}} \geq 960$.
- Desired output property: the score for COC is not the minimal score.

Figure 9. Illustration de ACAS Xu et quelques propriétés à vérifier. Les valeurs θ, ρ et ψ représentent des angles relatifs entre un appareil *ownship* et un appareil *intrus* ; v_{own} la vitesse de l'appareil *ownship* et v_{int} la vitesse de l'intrus. Les propriétés décrivent des comportements attendus de *ownship* par rapport à l'intrus. Figure et image tirées de [Kat+17].

5.2 Spécifier un processus de normalisation sur des propriétés fonctionnelles

La vérification formelle de réseaux de neurone compte quelques bancs d'essai classiques. ACAS Xu (*Aicraft Collision Avoidance System - Unmanned*) est l'un d'entre eux. Originellement présenté dans [Kat+17], il consiste à vérifier des propriétés fonctionnelles sur un réseau embarqué dans un aéronef. Le réseau dispose de coordonnées relatives d'un autre aéronef (l'intrus), et doit fournir une consigne de changement de direction. Différentes propriétés fonctionnelles sont formalisées et prouvées dans la publication originelle ; une section de la VNN-Comp consiste à vérifier les propriétés ACAS Xu le plus rapidement possible (en restant évidemment correct). On pourra se référer à la Figure 9 et à la publication originale pour plus d'informations.

Nous considérons ACAS Xu avec une différence subtile, mais de taille. La publication originale présente les propriétés dans un espace de valeurs non-normalisé. À titre d'exemple, la propriété ϕ_1 s'exprime : $\mathcal{P}(x) = x[\rho] \geq 55947.691 \wedge x[vown] \geq 1145 \wedge x[vint] \leq 60 \implies \mathcal{Q}(y) = y[coc] < 1500$. Cependant, la propriété effectivement encodée dans Marabou sont des valeurs normalisées [10] comprises entre $-0,5$ et $3,9911256459$. Nous soulignons qu'il ne s'agit pas d'une faute de la part des concepteurûices du système. L'algorithme de rétropropagation du gradient qui sous-tend l'apprentissage profond perd en efficacité dès lors que les quantités calculées demeurent un large intervalle de valeur. Il est donc très souvent nécessaire de renormaliser les entrées et sorties d'un réseau de neurone. Plus formellement, soient des domaines numérique de *spécification* et *vérification* sur les entrées (resp. sur les sorties) $\mathcal{S}_x, \mathcal{V}_x$ (resp. $\mathcal{S}_y, \mathcal{V}_y$). On peut définir $n : \mathcal{S}_x \mapsto \mathcal{V}_x$ une fonction de *normalisation* et $d : \mathcal{V}_y \mapsto \mathcal{S}_y$ une fonction de dénormalisation. Pour vérifier une spécification $\{P\}f\{Q\}$ sur une fonction neuronale f, il est nécessaire d'y intégrer les fonctions n et d, soient de vérifier effectivement $\{P\}d \circ f \circ n\{Q\}$. Ne pas le faire créerait un écart entre une spécification définie par des experts et celle effectivement vérifiée.

Nous montrons qu'il est possible d'utiliser WhyML pour modéliser n et d, et notre édition

10. On peut se référer au dépôt git de Marabou `https://github.com/NeuralNetworkVerification/Marabou/blob/215828c64e624be7917e69e4e873c746d8df25a2/resources/properties/acas_property_1.txt` pour les valeurs exactes.

Property ϕ_4.

- Description: If the intruder is directly ahead and is moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.
- Tested on: all networks except $N_{1,7}$, $N_{1,8}$, and $N_{1,9}$.
- Input constraints: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi = 0$, $v_{\text{own}} \geq 1000$, $700 \leq v_{\text{int}} \leq 800$.
- Desired output property: the score for COC is not the minimal score.

Property ϕ_6.

- Description: If the intruder is sufficiently far away, the network advises COC.
- Tested on: $N_{1,1}$.
- Input constraints: $12000 \leq \rho \leq 62000$, $(0.7 \leq \theta \leq 3.141592) \vee (-3.141592 \leq \theta \leq -0.7)$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{\text{own}} \leq 1200$, $0 \leq v_{\text{int}} \leq 1200$.
- Desired output property: the score for COC is the minimal score.

Figure 10. Propriétés ACAS Xu ϕ_4 et ϕ_6.

automatique de graphe pour les encoder directement dans une fonction neuronale. On peut ainsi immédiatement écrire $\{P\}d \circ f \circ n\{Q\}$, explicitant ainsi un processus de pré et post-traitement de données.

La Figure 11 présente un fragment d'une telle spécification WhyML. La fonction `normalize_t` normalise une entrée suivant une valeur moyenne et un écart-type en suivant une sémantique d'arithmétique flottante. La fonction `normalize_input` applique cette opération sur chaque élément d'un vecteur. `runP1` est la fonction qui décrit la normalisation de l'entrée (ligne `17`), le calcul effectif de la fonction neuronale `nn` sur lesdites entrées (ligne `18`) et l'accès à une composante (ici, `clear_of_conflict` est une valeur entière). Enfin, ligne `19`, `runP1` replonge les sorties dans le domaine de spécification. Nous soulignons que la spécification ainsi exprimée correspond exactement à la spécification mathématique exprimée dans la publication originale.

Nous consignons en Table 1 le temps de vérification sur les propriétés ACAS Xu normalisées, ainsi que sur les version non-normalisées (celles générées par CAISAR). Fort heureusement, nous n'introduisons pas d'incorrection dans PyRAT. Ce prouveur est capable de vérifier les propriétés générées par CAISAR avec un surcoût en temps de calcul. À notre grand regret, certaines propriétés prennent beaucoup plus de temps à vérifier, au point de dépasser la limite de temps que nous nous sommes imposés. Nous pensons que l'ajout des matrices linéaires pour encoder les noeuds **`Gather`** entraîne une baisse drastique de la précision de PyRAT.

Pour nnenum, nous notons une différence sur la réponse du prouveur sur les propriétés ϕ_4 et ϕ_6. Nous avons identifié l'origine de cette différence comme un bug imputable au *parser* ONNX de nnenum qui sera remonté aux auteurs. Pour référence, les propriétés ϕ_4 et ϕ_6 sont présentées Figure 10.

Plus grave, Marabou trouve un contre-exemple sur les propriétés ϕ_4 et ϕ_6, là où les réseaux employés ne devraient pas en avoir selon les résultats originels. Une étude préliminaire du contre-exemple généré par Marabou montre que l'évaluation de ce contre-exemple dans le réseau ONNX modifié n'entraîne pas de violation de propriété. Il est, de notre avis, trop tôt pour conclure à un bug de Marabou.

5.3 Propriétés sur plusieurs fonctions neuronales

Comparaison entre sorties de réseaux Nous présentons une propriété inspirée de l'exemple de l'introduction (*cf.* Propriété 3).

Il est fréquent en apprentissage automatique d'élaguer un réseau de neurone, c'est à dire passer les poids inutilisés à zéro et se reposer sur des représentations de matrices creuses. Il est possible de spécifier la différence de performance induite par ce processus. Soit un réseau de neurone $g_1 : \mathcal{X} \mapsto \mathcal{Y}$ et g_2 sa version élaguée, on souhaite vérifier $\{\varphi_{pre}(\vec{x})\}\ \|g_1(\vec{x}) - g_2(\vec{x})\|_\infty \leq \varepsilon\ \{\varphi_{post}(\vec{z})\}$, où $\varepsilon \in \mathbb{R}$.

Nous avons entraîné un réseau de neurone sur l'ensemble d'apprentissage MNIST [Lec+98], dont nous avons élagué 20% des poids les plus inutilisés. Hélas, Marabou et nnenum échouent à lire le réseau ONNX. PyRAT réussit là où les deux autres prouveurs échouent, mais dépasse sa limite de temps.

```
1   let function normalize_t i mean range = (i .- mean) ./ range
2
3   let function denormalize_t i mean range = (i .* range) .+ mean
4
5   let function normalize_input i = Vector.mapi i normalize_by_index
6
7   let function denormalize_output_t o =
8     denormalize_t o
9       (7.518884020100597531666153372498229146003723144531 25:t)
10        (373.94992000000002008164301514625549316406 25:t)
11
12  let runP1 (i: input) : t
13    requires { has_length i 5 }
14    requires { valid_input i }
15    requires { intruder_distant_and_slow i }
16    ensures { result .≤ (1500.0:t) }  =
17      let j = normalize_input i in
18      let o = (nn @@ j)[clear_of_conflict] in
19      (denormalize_output_t o)
```

Figure 11. Spécification WhyML pour la propriété ACAS Xu ϕ_1 non normalisée.

Property	**PyRAT**				**nnenum**				**Marabou**			
	T_n	A_n	T_u	A_u	T_n	A_n	T_u	A_u	T_n	A_n	T_u	A_u
ϕ_1 (1_1)	9.00	(✓)	13.00	(✓)	2.00	(✓)	2.67	(✓)	3.00	(✓)	900.00	(🕐)
ϕ_2 (1_1)	22.33	(✓)	61.00	(✓)	2.67	(✓)	4.67	(?)	35.67	(✓)	900.00	(🕐)
ϕ_3 (1_1)	381.00	(✓)	607.33	(🕐)	3.33	(✓)	11.00	(✓)	344.00	(✓)	900.00	(🕐)
ϕ_4 (1_1)	32.33	(✓)	607.00	(🕐)	2.33	(✓)	5.00	(✓)	49.00	(✓)	1.00	(✗)
ϕ_5 (1_1)	607.00	(🕐)	607.00	(🕐)	4.00	(✓)	10.33	(✓)	120.33	(✓)	7.00	(✗)
ϕ_6 (1_1)	190.33	(✓)	917.67	(🕐)	18.33	(✓)	54.00	(?)	660.00	(✓)	900.00	(🕐)
ϕ_7 (1_9)	607.00	(🕐)	607.00	(🕐)	2.00	(✗)	2.00	(✗)	5.00	(✗)	4.33	(✗)
ϕ_8 (2_9)	607.00	(🕐)	607.00	(🕐)	2.00	(✗)	2.00	(✗)	637.33	(✗)	900.00	(🕐)
ϕ_9 (3_3)	59.00	(✗)	175.33	(✗)	5.67	(✗)	11.00	(✗)	181.00	(✗)	6.00	(✗)
ϕ_{10} (4_5)	7.00	(✓)	7.33	(✓)	2.00	(✓)	2.00	(✓)	8.00	(✓)	900.00	(🕐)

Table 1. Résultats du banc d'essai ACAS Xu sur les prouveurs appelés par CAISAR. La moyenne est faite sur trois évaluations. La mention à droite de la propriété indique quel réseau de neurone Reluplex a été utilisé pour la vérifier. La colonne T_n (resp. T_u) indique le temps (en secondes) nécessaire au prouveur pour donner une réponse sur la propriété normalisée (resp. non-normalisée). La colonne A_n (resp. A_u) indique le résultat de la vérification sur la propriété normalisée (resp. non-normalisée). Une ✓ indique qu'une propriété est satisfiable, une ✗ qu'un contre-exemple a été trouvé, une 🕐 indique le dépassement de la limite de temps, et un ? une erreur fatale du prouveur.

Composition de fonctions neuronales Une autre classe de propriétés commune implique plusieurs calculs de fonctions neuronales en séquence. Soient deux réseaux $g_1 : \mathcal{X} \mapsto \mathcal{Y}$ et $g_2 : \mathcal{Y} \mapsto \mathcal{Z}$, on cherche à vérifier $\{\varphi_{pre}(\vec{x})\}\ f(\vec{x})\ \{\varphi_{post}(\vec{z})\}$, où f est $g_2 \circ g_1(\vec{x})$.

Ici, tous les prouveurs évalués réussissent à lire le fichier ONNX généré, mais dépassent tous leur limite de temps.

5.4 À spécification valide, vérification impossible ?

La plupart des prouveurs comportent des *bugs* bloquant les empêchant de traiter nos graphes ONNX, pourtant conformes à la spécification. Nous avons vérifié cette conformité, d'une part, en nous assurant de n'utiliser que des opérateurs connus et identifiés comme supportés (vérification effectuée en consultant le code source des prouveurs) ; d'autre part, nos fichiers ONNX générés sont conformes au onnxruntime, l'implémentation officielle du standard ONNX. Toutefois, du fait de ces défaillances, il semble prématuré de conclure que notre approche permet effectivement une vérification formelle sur des propriétés plus

étendues.

Les résultats présentés ici sont à notre sens à interpréter comme permettant la spécification de nouvelles propriétés. Nous restons confiants sur la possibilité de les vérifier effectivement dans le futur : les limitations des prouveurs étant très vraisemblablement temporaires et dues à des contraintes de développement qui sont celles du temps académique plutôt que de l'industrie.

Des travaux futurs pour le support d'opérateurs ONNX tels que **`Gather`** permettront, nous l'espérons, un encodage plus efficace des expressions WhyML dans ONNX.

6 Travaux antécédents

Les dernières années ont vu s'accroître le nombre d'outils visant à la vérification formelle de réseaux de neurones [Kat+19 ; Wan+21 ; Bak21 ; Dur+22 ; RZ19]. L'objet de cet article n'étant pas de faire une revue de littérature exhaustive, on pourra se référer à une telle revue pour plus de précisions [UM21]. À l'exception notable de SAVer [RZ19], ces outils ciblent exclusivement les réseaux de neurones, et offrent un support partiel de ONNX et VNN-LIB. Nous avons évalué nnenum [Bak21], PyRAT [Dur+22] et Marabou [Kat+19], du fait de leur classement à la VNN-Comp.

Il existe déjà des possibilités d'encoder des propriétés impliquant plusieurs réseaux. Ainsi, l'équivalence est encodée dans les outils ReluDiff [PWW20] et NeuroDiff [Pau+21]. Ces outils ne gèrent toutefois pas les calculs arbitraires sur les entrées et sorties de réseaux.

Nous ne sommes pas les premier à explorer les limites de VNN-LIB (voir Section 1), ni à proposer une alternative de plus haut-niveau, sous la forme de langages spécifiques au domaine (LSD) Le LSD DNNP [SED21], intégré dans l'outil DNNV, est un sous-ensemble du langage Python. De façon similaire à CAISAR, DNNV propose une réimplémentation partielle de la théorie QF_LRA et propose un mécanisme pour l'édition automatique de réseaux. Ce dernier ne sert toutefois qu'à retirer des opérateurs non-supportés par les prouveurs intégrés à DNNV, ou pour concaténer une séquence d'opération en une seule. Il n'est pas question d'intégrer la spécification au sein du réseau. À notre avis, l'emploi de Python qui limite la sûreté due au typage constitue une autre différence notable. NNV [Lop+23] est une boîte à outil MatLab proposant des capacités de vérification pour de multiples architectures de réseaux de neurones. Lesdites propriétés portent sur des réseaux uniques, ou des systèmes en rétroaction à boucle fermée, et s'expriment exclusivement sous la forme d'ensembles atteignables. NeSAL [XKN22] définis des réseaux de neurones spécialisés, appelés *réseaux de spécification*, ainsi qu'un fragment de logique du premier ordre pour satisfaire des propriétés. De notre compréhension, NeSAL n'existe en implémentation que sous la forme de scripts Python déjà définis. Si la publication originelle propose de réécrire une partie de la spécification au sein du réseau, elle ne le fait pas de manière automatique (l'édition est faite manuellement). Vehicle [Dag+22] propose un langage de haut niveau pour exprimer des propriétés sur un réseau de neurone. Ce langage est implémenté dans l'outil du même nom, qui offre une compilation vers Marabou et le prouveur interactif Agda. De l'opinion des auteurs, le typage fort (incluant des types dépendants) de Vehicle représente un atout certain. Toutefois, Vehicle ne supporte que Marabou en tant que prouveur dédié, et n'a aucune capacité d'édition de réseau ONNX. Enfin, il existe des bibliothèques logicielles dédiées à l'édition de graphe ONNX, telles que onnx-simplifier et GraphSurgeon ; elles sont générique et ne visent pas à l'intégration de spécifications au sein d'un réseau de neurone.

7 Conclusion

Nous avons présenté un langage de spécification formelle pour les fonctions synthétisées par apprentissage automatique. Nous pouvons spécifier des propriétés impliquant plusieurs de ces fonctions neuronales grâce à l'implémentation de ce langage au sein de CAISAR. Nous

avons présenté une approche d'édition automatique de graphe nous permettant de dépasser les limitations des langages de spécification existant en intégrant directement dans le graphe de calcul une partie de la spécification. Malgré le respect des langages d'entrée théoriques des prouveurs, nous constatons que la plupart n'arrivent pas à vérifier les propriétés générées par CAISAR ; soit à cause de *bugs*, soit de soucis de passage à l'échelle.

Un premier travail essentiel déjà entamé consistera à remonter aux auteurûices des prouveurs les limitations rencontrées. Nous sommes en train d'investiguer comment réduire l'empreinte mémoire et temporelle des transformations générées par CAISAR.

Nous envisageons plusieurs extensions pour notre langage. Nous pouvons par exemple imaginer une théorie Why3 dédiée à la spécification de traitement de données, notamment en encodant des transformations géométriques directement sous la forme d'opérateurs ONNX. Le support de logiques temporelle linéaire est également à l'étude.

Nous souhaitons étendre le langage de CAISAR pour encoder des obligations de preuve permettant [MI22 ; BK23] l'interprétabilité des réseaux de neurones.

Les résultats présents ont été financés avec le soutien de l'Agence Nationale de la Recherche (ANR), projet ANR-23-DEGR-0001.
Des remerciements chaleureux à Serge Durand pour son aide avec les idiosyncrasies de PyRAT et Marabou, ainsi qu'à Augustin Lemesle et Julien Lehmann pour leur disponibilité et leur soutien.

Références

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt et Mattias Ulbrich. "Deductive Software Verification The KeY Book". In : *Lecture Notes in Computer Science*. 2016. URL : https://api.semanticscholar.org/CorpusID:34552701 (cf. p. 2).

[Alb+22] Michele Alberti, François Bobot, Zakaria Chihani, Julien Girard-Satabin et Augustin Lemesle. "CAISAR : A platform for Characterizing Artificial Intelligence Safety and Robustness". In : *AISafety*. CEUR-Workshop Proceedings. Vienne, Austria, juill. 2022. URL : https://hal.archives-ouvertes.fr/hal-03687211 (cf. p. 4).

[Bak21] Stanley Bak. "Nnenum : Verification of ReLU Neural Networks with Optimized Abstraction Refinement". In : *NASA Formal Methods*. Sous la dir. d'Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz et Ivan Perez. Lecture Notes in Computer Science. Cham : Springer International Publishing, 2021, p. 19-36. ISBN : 978-3-030-76384-8. DOI : 10.1007/978-3-030-76384-8_2 (cf. p. 12, 16).

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds et Cesare Tinelli. "CVC4". In : *Computer Aided Verification (CAV)*. Sous la dir. de Ganesh Gopalakrishnan et Shaz Qadeer. T. 6806. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 171-177. ISBN : 978-3-642-22110-1. DOI : 10.1007/978-3-642-22110-1_14. URL : http://link.springer.com/10.1007/978-3-642-22110-1_14 (visité le 19/05/2021) (cf. p. 6).

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles et Nicky Williams. "The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform". In : *Communications of the ACM* 64.8 (août 2021), p. 56-68. ISSN : 0001-0782, 1557-7317. DOI : 10.1145/3470569 (cf. p. 2).

[BFT16] Clark Barrett, Pascal Fontaine et Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016 (cf. p. 2).

[BK23] Shahaf Bassan et Guy Katz. "Towards Formal XAI : Formally Approximate Minimal Explanations of Neural Networks". In : *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, 2023, p. 187-207. isbn : 9783031308239. doi : 10.1007/978-3-031-30823-9_10. eprint : 2210.13915 (cs.LG) (cf. p. 17).

[Bri+23] Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T. Johnson et Changliu Liu. *First Three Years of the International Verification of Neural Networks Competition (VNN-COMP)*. 2023. arXiv : 2301.05815 [cs.LG] (cf. p. 2).

[Chi21] Zakaria Chihani. "Formal Methods for AI : Lessons from the past, promises of the future". In : *CAID 2021 : Conference on Artificial Intelligence for Defense*. Rennes (FR), France, nov. 2021. url : https://hal.science/hal-04479570 (cf. p. 2).

[Com22] Community. *ONNX : Open Neural Network eXchange Format standard for machine learning interoperability*. Open Neural Network Exchange. Avr. 2022 (cf. p. 2, 3, 11).

[Dag+22] Matthew L. Daggitt, Wen Kokke, Robert Atkey, Luca Arnaboldi et Ekaterina Komendantskya. *Vehicle : Interfacing Neural Network Verifiers with Interactive Theorem Provers*. 2022. doi : 10.48550/ARXIV.2202.05207. url : https://arxiv.org/abs/2202.05207 (cf. p. 16).

[dB08] Leonardo de Moura et Nikolaj Bjørner. "Z3 : An Efficient SMT Solver". In : *Tools and Algorithms for the Construction and Analysis of Systems*. Sous la dir. de C. R. Ramakrishnan et Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2008, p. 337-340. isbn : 978-3-540-78800-3. doi : 10.1007/978-3-540-78800-3_24 (cf. p. 6).

[Dem+23] S. Demarchi, D. Guidotti, L. Pulina et A. Tacchella. "Supporting Standardization of Neural Networks Verification with VNN-LIB and CoCoNet." In : *6th Workshop on Formal Methods for ML-Enabled Autonomous Systems*. Juill. 2023 (cf. p. 2).

[Dur+22] Serge Durand, Augustin Lemesle, Zakaria Chihani, Caterina Urban et François Terrier. "ReCIPH : Relational Coefficients for Input Partitioning Heuristic". In : *WFVML 2022* (juill. 2022). Poster. url : https://inria.hal.science/hal-03926281 (cf. p. 12, 16).

[Eki+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds et Clark W. Barrett. "SMTCoq : A Plug-In for Integrating SMT Solvers into Coq". In : *Proceedings of the 28th International Conference on Computer Aided Verification, Part II*. Springer, 2017, p. 126-133. isbn : ['978-3-319-63389-3', '978-3-319-63390-9']. doi : 10.1007/978-3-319-63390-9_7 (cf. p. 6).

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich. "Why3 - Where Programs Meet Provers". In : *Programming Languages and Systems*. Sous la dir. de Matthias Felleisen et Philippa Gardner. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2013, p. 125-128. isbn : 978-3-642-37036-6. doi : 10.1007/978-3-642-37036-6_8 (cf. p. 2, 4).

[Kat+17] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian et Mykel J. Kochenderfer. "Reluplex : An Efficient SMT Solver for Verifying Deep Neural Networks". In : *Computer Aided Verification*. Springer International Publishing, 2017, p. 97-117. isbn : 9783319633879. doi : 10.1007/978-3-319-63387-9_5 (cf. p. 2, 13).

[Kat+19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zelji, David L. Dill, Mykel J. Kochenderfer et Clark Barrett. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". en. In : *Computer Aided Verification*. Sous la dir. d'Isil Dillig et Serdar Tasiran. T. 11561. Cham : Springer International Publishing, 2019, p. 443-452. isbn : 978-3-030-25539-8 978-3-030-25540-4. doi : 10.1007/978-3-030-25540-4_26. (Visité le 18/07/2019) (cf. p. 12, 16).

[Lec+98] Y. Lecun, L. Bottou, Y. Bengio et P. Haffner. "Gradient-based learning applied to document recognition". In : *Proceedings of the IEEE* 86.11 (1998), p. 2278-2324. issn : 0018-9219. doi : 10.1109/5.726791 (cf. p. 14).

[Lop+23] Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran et Taylor T. Johnson. "NNV 2.0 : The Neural Network Verification Tool". In : *Computer Aided Verification*. Sous la dir. de Constantin Enea et Akash Lal. Cham : Springer Nature Switzerland, 2023, p. 397-412. isbn : 978-3-031-37703-7 (cf. p. 16).

[MI22] Joao Marques-Silva et Alexey Ignatiev. "Delivering Trustworthy AI through Formal XAI". In : *Proceedings of the AAAI Conference on Artificial Intelligence* 36.11 (juin 2022), p. 12342-12350. issn : 2159-5399. doi : 10.1609/aaai.v36i11.21499. url : https://ojs.aaai.org/index.php/AAAI/article/view/21499 (cf. p. 17).

[Pau+21] Brandon Paulsen, Jingbo Wang, Jiawei Wang et Chao Wang. "NeuroDiff : scalable differential verification of neural networks using fine-grained approximation". In : *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia : Association for Computing Machinery, 2021, p. 784-796. isbn : 9781450367684. doi : 10.1145/3324884.3416560. url : https://doi.org/10.1145/3324884.3416560 (cf. p. 16).

[PT11] Luca Pulina et Armando Tacchella. "NeVer : a tool for artificial neural networks verification". In : *Annals of Mathematics and Artificial Intelligence* 62.34 (juill. 2011), p. 403-425. issn : 1012-2443. doi : 10.1007/s10472-011-9243-0. url : https://doi.org/10.1007/s10472-011-9243-0 (cf. p. 2).

[PWW20] Brandon Paulsen, Jingbo Wang et Chao Wang. "ReluDiff : differential verification of deep neural networks". In : *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE 20. ACM, juin 2020. doi : 10.1145/3377811.3380337. url : http://dx.doi.org/10.1145/3377811.3380337 (cf. p. 16).

[RZ19] Francesco Ranzato et Marco Zanella. "Robustness Verification of Support Vector Machines". In : *Static Analysis*. Sous la dir. de Bor-Yuh Evan Chang. Cham : Springer International Publishing, 2019, p. 271-295. isbn : 978-3-030-32304-2 (cf. p. 2, 16).

[SED21] David Shriver, Sebastian Elbaum et Matthew B. Dwyer. "DNNV : A Framework for Deep Neural Network Verification". In : *Computer Aided Verification*. Sous la dir. d'Alexandra Silva et K. Rustan M. Leino. Lecture Notes in Computer Science. Cham : Springer International Publishing, 2021, p. 137-150. isbn : 978-3-030-81685-8. doi : 10.1007/978-3-030-81685-8_6 (cf. p. 16).

[UM21] Caterina Urban et Antoine Miné. "A Review of Formal Methods Applied to Machine Learning". In : *arXiv:2104.02466 [cs]* (avr. 2021). arXiv : 2104.02466 [cs] (cf. p. 16).

[Wan+21] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh et J. Zico Kolter. *Beta-CROWN : Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Robustness Verification*. 31 oct. 2021. arXiv : 2103.06624 [cs, stat]. URL : http://arxiv.org/abs/2103.06624 (visité le 04/03/2022) (cf. p. 12, 16).

[XKN22] Xuan Xie, Kristian Kersting et Daniel Neider. "Neuro-Symbolic Verification of Deep Neural Networks". In : *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. Sous la dir. de Lud De Raedt. Main Track. International Joint Conferences on Artificial Intelligence Organization, juill. 2022, p. 3622-3628. DOI : 10.24963/ijcai.2022/503. URL : https://doi.org/10.24963/ijcai.2022/503 (cf. p. 16).