MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Verification of binarised neural networks using ASP

Bachelor's Thesis

JINDŘICH MATUŠKA

Brno, Spring 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Verification of binarised neural networks using ASP

Bachelor's Thesis

JINDŘICH MATUŠKA

Advisor: RNDr. Samuel Pastva, PhD.

Department of Computer Systems and Communications

Brno, Spring 2024

MUNI
FI

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jindřich Matuška

Advisor: RNDr. Samuel Pastva, PhD.

Acknowledgements

My thanks go to my family, which has provided me with a firm ground for my whole life, to my friends whom I could spend my free time and who were distracting me from the never-ending study duties, to all my teachers, who had patience with me and helped me grow, and finally to my advisor, who has provided me with this assignment and was my support when writing this thesis.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Abstract

Deep neural networks are state-of-the-art technology. Using them in critical real-life applications carries a risk of failure. For this, verification of their properties is needed. This thesis explores the possibility for the use of answer set programming paradigm in this task. It implements a quantitative verificator for binarized neural networks, a special case of deep neural networks, using this paradigm. It also demonstrates the use of this verificator on a network trained on the MNIST dataset. The verificator proves to be especially good when evaluating highly robust networks.

Keywords

verification, binarized neural networks, answer set programming, Clingo, robustness

Contents

List of Tables

Thesis assignment

ASP (Answer Set Programming) is a form of constraint programming designed for solving various NP-complete search problems. It is often used as an alternative to SAT/SMT solvers or symbolic algorithms based on BDDs (Binary Decision Diagrams). Meanwhile, SAT/SMT and BDDs are one of the key tools in current verification and validation workflows for binarised neural networks (BNNs) [1]. Conceptually, the goal of this thesis is therefore to explore the possibilities of applying ASP to reason about the behaviour of binarised neural networks. This should supplement the existing SAT/SMT/BDD-based approaches.

More concretely, the student should familiarise themselves with the problem of BNN robustness and the ASP method in general. They should then formulate an ASP encoding of the BNN robustness problem such that the robustness of a network can be validated using a suitable ASP solver (e.g. clingo [2]). The student should then create a prototype implementation of this encoding that will be tested on a collection of reasonable examples (such as the ones from [1]).

1 Introduction

2 Binarised neural networks

Binary neural networks are a type of Deep neural networks where instead of floating point numbers, binary values are used as inputs and outputs of layers. This reduction of available values can lead to high reduction in the computation time and energy consumption while achieving near state-of-the-art results [3].

Notation

Symbol	Definition
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of whole numbers
$\mathbb{B}$	set of ± 1-binarised values, $\mathbb{B} = \{-1, 1\}$
K^m	set of vectors on K of lenght m

Table 2.1: Used notation

2.1 Definition of Binarised neural network

2.1.1 Deep neural network

General neural network is a multilayer perceptron. It consists of perceptrons (neurons) in layers, these are further split into the input layer, the output layer and possibly multiple hidden layers. Each perceptron computes its inner potential ξ based on the outputs of the previous layer, its output is then determined by an activation function σ.

For the computation of the inner potential ξ weighted sum is used. There are many different used activation functions σ such as *unit step function*, *logistic sigmoid*, *hyperbolic tangens* or *ReLU*.

Multiple of these perceptrons are then assorted into layers. The input layer is not consisting of perceptrons but is straight composed of the inputs. The output layer is often represented by perceptrons with activation function from other layers, such as *argmax* for single-choice classification and *TODO* for the classification using the probability.

Further follows a more formal definition of the deep neural network corresponding to the one from [4].

Definition 2.1.1 (Perceptron)**.** Perceptron p is a function from vector of k real numbers to a real number. The function p is a composition of an inner potential ξ and an activation function σ. The inner potential ξ is weighted sum parametrized by static vector of real numbers $\vec{w}$ of size k and real bias b. The activation function σ can be instantized by any real-valued function.

$$p : \mathbb{R}^k \to \mathbb{R}$$

$$\xi : \mathbb{R}^m \to \mathbb{R},\ \sigma : \mathbb{R} \to \mathbb{R}$$

$$\xi(\vec{x}) = b + \sum_{i=1}^{m} w_i \cdot x_i$$

$$p = \sigma \circ \xi$$

Definition 2.1.2 (Single-layer perceptron)**.** Single-layer perceptron is a function t from vector of k real numbers to vector of l real numbers where each value in the result vector is computed by a signle perceptron.

$$t :\ \mathbb{R}^m \to \mathbb{R}^n$$

$$t(\vec{x}) = (p_1(\vec{x}), p_2(\vec{x}), \ldots, p_n(\vec{x}))$$

Definition 2.1.3 (Multi-layer perceptron)**.** Multi-layer perceptron (also called deep neural network) is a convolution of Single-layer perceptrons. The last applied layer t_{d+1} is called the output layer, all other layers are called hidden layers. The input of a multi-layer perceptron is called the input layer.

$$\mathcal{N} :\ \mathbb{R}^{n_0} \to \mathbb{R}^{n_{d+1}}$$

$$\mathcal{N} = t_{d+1} \circ t_d \circ \ldots \circ t_1$$

2.1.2 Binarised neural network

As the computation of general multi-layer perceptron depends on slow multiplication of floating-point numbers, came with an idea of binarized perceptron. This type of perceptron constrains the input vector, Binarized perceptron constraints the input space and vector of weights to vectors of binary values -1 and 1. The activation function is usually a heavyside step function H.

$$H(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

The computation of this constrained perceptron is faster than of the general perceptron as multiplication of two ±1-binarized values can be done with a single *XOR* gate (-1 is equivalent to 1, 1 is equivalent to 0).

From binarized perceptrons, multi-layered perceptrons can be built similiarly to the general perceptron. In the case of the categorisation problem, argmax layer consisting of weighted sums and *argmax* operator is used as the output layer to find the maximal output.

Further follows formal definition of binarised neural network. I also show that every binarised perceptron defined using parameters as in the definition from [1] can be translated into binarised perceptron according to my definition and vice versa. This shows the equivalence of my definition to that of [1].

Definition 2.1.4 (Binarised perceptron)**.** Binarised perceptron $p^{\mathbb{B}}$ is a function from vector of m ±1-binarised values to a single ±1-binarised value. The function $p^{\mathbb{B}}$ is a composition of an inner potential ξ and a heavyside step function H. The inner potential ξ is weighted sum parametrized by static vector of ±1-binarised values $\vec{w}$ of size k, and real bias b.

$$p^{\mathbb{B}} : \mathbb{B}^k \rightarrow \mathbb{B}$$

$$\xi : \mathbb{B}^k \rightarrow \mathbb{R},\ H : \mathbb{R} \rightarrow \mathbb{B}$$

$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot x_i$$

$$p^{\mathbb{B}} = H \circ \xi$$

Definition 2.1.5 (Binarised perceptron with batch normalization)**.** Binarised perceptron with batch normalization $\hat{p}^{\mathbb{B}}$ is a function from vector of m ±1-binarised values to a single ±1-binarised value. The function $p^{\mathbb{B}}$ is a composition of an inner potential ξ, batch normalization function ρ a heavyside step function H. The inner potential ξ is weighted sum parametrized by static vector of ±1-binarised values $\vec{w}$ of size k, and real bias b. The batch normalization function ρ is a function on real numbers parametrized by real values α, γ, μ, σ.

$$\hat{p}^{\mathbb{B}} : \mathbb{B}^k \to \mathbb{B}$$

$$\xi : \mathbb{B}^k \to \mathbb{R},\ \rho : \mathbb{R} \to \mathbb{R},\ H : \mathbb{R} \to \mathbb{B}$$

$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot x_i$$

$$\rho(x) = \alpha \cdot \left(\frac{x - \mu}{\sigma} \right) + \gamma$$

$$\hat{p}^{\mathbb{B}} = H \circ \rho \circ \xi$$

Lemma 2.1.1. *For every Binarised perceptron there is equivalent Binarised perceptron with batch normalization.*

Proof. If the parameters of batch normalization function ρ are set to be $\alpha = \sigma = 1, \gamma = \mu = 0$, function ρ is identity function. With parameters of inner potential ξ unchanged, the following holds

$$p^{\mathbb{B}} = H \circ \xi = H \circ \mathrm{id} \circ \xi = H \circ \rho \circ \xi = \hat{p}^{\mathbb{B}}$$

□

Lemma 2.1.2. *For every Binarised perceptron with batch normalization there is equivalent Binarised perceptron.*

Proof. The idea behind this construction comes from [1].

The value of $\hat{p}^{\mathbb{B}}(\vec{x})$ is only determined by the sign of expression $(\rho \circ \xi)(\vec{x})$. Lets thus analyse the inequality $(\rho \circ \xi)(\vec{x}) \geq 0$.

$$(\rho \circ \xi)(\vec{x}) = \alpha \cdot \left(\frac{b + \sum_{i=1}^{k} w_i \cdot x_i - \mu}{\sigma} \right) + \gamma$$

If $\alpha = 0$, then the expression $\rho \circ \xi$ is a constant function. In that case the perceptron is equivalent to the one with its bias bigger than the length of input vector for positive constant perceptron or with bias lower than the length of input vector for negative constant perceptron.

If $\alpha \neq 0$, the expression can be divided by the term $\frac{\alpha}{\sigma}$. In the case of this term being negative, the inequality switches and has to be corrected by further multiplying by -1.

$$(\rho \circ \xi)(\vec{x}) \cdot \frac{\sigma}{\alpha} = b + \sum_{i=1}^{k} w_i \cdot x_i - \mu + \frac{\sigma \cdot \gamma}{\alpha}$$

$$(\rho \circ \xi)(\vec{x}) \cdot \frac{\sigma}{\alpha} = (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i$$

$$\frac{\alpha}{\sigma} > 0 : (\rho \circ \xi)(\vec{x}) \geq 0 \iff (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i \geq 0$$

$$\frac{\alpha}{\sigma} < 0 : (\rho \circ \xi)(\vec{x}) \geq 0 \iff (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i \leq 0$$

$$\iff (-b + \mu - \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} -w_i \cdot x_i \geq 0$$

As can be seen, in the positive case the expression breaks into two parts, the new bias $(b - \mu + \frac{\sigma \cdot \gamma}{\alpha})$ and unchanged weighted sum $\sum_{i=1}^{k} w_i \cdot x_i$. In the negative case, both the bias and vector of weights are negative.

The perceptron without batch normalization for the positive resp. negative case consists of bias $(b - \mu + \frac{\sigma \cdot \gamma}{\alpha})$ resp. $(-b + \mu - \frac{\sigma \cdot \gamma}{\alpha})$ and weight vector of $\vec{w}$ resp. $-\vec{w}$. □

Remark. The proof of lemma above is constructive and is used for encoding of the quantitative verification problem as ASP problem.

Remark. As $\sum_{i=1}^{k} w_i \cdot x_i$ is always a whole number, bottom whole part of bias $\lfloor b \rfloor$ can be used in place of bias in inner layers of BNN. In the output layer however the fractional part can make difference when choosing the maximal input.

Definition 2.1.6 (Binarised single-layer perceptron)**.** Binarised single-layer perceptron is a function $t^{\mathbb{B}}$ from vector of m ±1-binarised numbers to vector of n ±1-binarised numbers where each value in the result vector is computed by a signle perceptron.

$$t^{\mathbb{B}} : \mathbb{B}^m \to \mathbb{B}^n$$
$$t^{\mathbb{B}}(\vec{x}) = (p_1^{\mathbb{B}}(\vec{x}), p_2^{\mathbb{B}}(\vec{x}), \ldots, p_n^{\mathbb{B}}(\vec{x}))$$

Definition 2.1.7 (Argmax layer)**.** Argmax layer is a function t^{am} which returns the mask of maximal value after the weighted sum. This mask has form of a one-hot vector, where only the first position with the maximal value after the weighted sum is assigned value 1, all the other positions are assigned value 0.

$$t^{am} : \mathbb{B}^m \to \{0,1\}^n$$
$$t^{am}(\vec{x}) = y, \, y_k = 1 \iff k = \arg\max_{i=1}^{n}(\zeta_i(\vec{x}))$$

Definition 2.1.8 (Binarised multi-layer perceptron)**.** Binarised multi-layer perceptron (also called binarised neural network) is a convolution of binarised single-layer perceptrons. The last applied layer t^{am} is called the output layer and takes form of argmax layer, all other layers are called hidden layers. The input of a binarised multi-layer perceptron is called the input layer.

$$\mathcal{N}^{\mathbb{B}} : \mathbb{B}^{n_0} \to \{0,1\}^{n_{d+1}}$$
$$\mathcal{N}^{\mathbb{B}} = t^{am} \circ t_d^{\mathbb{B}} \circ \ldots \circ t_1^{\mathbb{B}}$$

2.1.3 Examples of Binarised neural network

2.2 Robustness of Binarised neural network

There are two types of robustness problems. The qualitative robustness is a problem to determine wether the neural network gives the same (true) output for all inputs in some input region. The quantitative rubustness on the other hand determines the part of this input region, which has the same output as some chosen input of this region.

Further I provide formal definition of robustness on functions in general.

2.2.1 Definition of robustness

Definition 2.2.1. Quantitative robustness of function. Let F be a function, $F : P \to Q$. Let I be an input region of F, that is $I \subseteq P$. Let w be a function, $w : P \to \mathbb{R}, \forall x \in P : w(x) > 0$. Let h_p be a function for some base input $p \in P$, $h_p : Q \to \mathbb{R}$, that satisfies

$$h_p(F(p)) = 0$$

$$\forall x \in Q. 0 \leq h_p(x) \leq 1$$

Let $Q_{d,h_p}(I)$ be equal to

$$Q_{w,h_p}(I) = \frac{\int_I w(i) \cdot h_p(F(i))\, di}{\int_I w(i)\, di} \quad \text{or} \quad Q_{w,h_p}(I) = \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)}$$

if the input region is non-discrete or discrete respectively. Then $Q_{w,h_p}(I)$ is called the quantitative robustness of function F with a weight function w and evaluation function h_p on input region I.

For empty input region $I = \emptyset$ or non-discrete input regions I such that $\int_I w(i) di = 0$, quantitative robustness is equal to 0.

The quantitative robustness is an average of values aquired by aplying the evaluation function h_p on the input region weighted by the weight function w. The lower the value of robustness is, the more the function is robust on input region.

The weight function can be used to make part of the inputs more prominent. For instance, by the use of weight function such as $w(x) = \frac{1}{1+||p-x||}$, inputs closer to the base input will have higher weight.

The evaluation function can be used to encode dissimiliarity of an input from the base input p. That could prove useful in case of external metric.

The simplest example of the weight function is a constant function $w_1(x) = 1$. With constant function, every input is assigned the same weight, which leads to the quantitative robustness being an average of evaluation function h_p applied over all inputs from the input region.

More complex weight functions can be used to give some areas of the input region higher priority. Such weight function may reflect requirement for the robustness closer to the base input p.

Lemma 2.2.1. *For input region consisting only of the base input $I = \{p\}$, the quantitative rubustness is equal to 0*

Proof.

$$Q_{w,h_p}(\{p\}) = \frac{w(p) \cdot h_p(p)}{w(p)} = \frac{w(p)}{w(p)} \cdot 0 = 0$$

□

Lemma 2.2.2. *Quantitative robustness is between 0 and 1 for every possible combination of functions and input regions.*

Proof. Let me first show that the quantitative robustness is nonnegative. Both w and h_p are nonnegative on P and Q respectively, thus $w(x) \cdot h_p(y)$ is nonnegative for all $(x, y) \in P \times Q$. For that also values of integrals and summations over both $w(x)$ and $w(x) \cdot h_p(y)$ are nonnegative. The fraction of two nonnegative values is nonnegative and the quantitative robustness is nonnegative.

By definition holds $0 \leq h_p(y) \leq 1$, thus also $0 \leq h_p(F(x)) \leq 1$. As $w(x) \geq 0$,

$$0 = w(x) \cdot 0 \leq w(x) \cdot h_p(F(x)) \leq w(x) \cdot 1$$

Again, integration or summation over any subset $I \subseteq P$ can be applied onto the latter two expressions.

$$0 \leq \int_I w(i) \cdot h_p(F(i))\, di \leq \int_I w(i)\, di$$

$$0 \leq \sum_{i \in I} w(i) \cdot h_p(F(i)) \leq \sum_{i \in I} w(i)$$

If the right-hand side term is equal to 0, the statement holds by definition of qualitative robustness. Else by division by the right-hand side term (nonnegative), following holds

$$0 \leq \frac{\int_I w(i) \cdot h_p(F(i))\, di}{\int_I w(i)\, di} \leq 1$$

$$0 \leq \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)} \leq 1$$

□

While the quantitative robustness does tell us something about how much of the input region is evaluated wrong (or even how much is it wrong), the qualitative robustness does only say if there is any wrong output. This may seem less useful, however it can lead to lower computational expenses.

Definition 2.2.2. Qualitative robustness of function F on input region I Function $F : P \rightarrow Q$ is (qualitatively) robust on input region $I \subseteq P$ with respect to evaluation function h_p if and only if $Q_{w,h_p}(I) = 0$.

Lemma 2.2.3. *The property of qualitative robustness is independent of the function w.*

Proof. For empty input region, the lemma holds trivially by definition.

Otherwise as weight function w is by definition positive, all statements of the following chain are equivalent.

$$Q_{w,h_p}(I) = \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)} = 0$$

Since $\sum_{i \in I} w(i) > 0$:

$$\sum_{i \in I} w(i) \cdot h_p(F(i)) = 0$$

As both w and h_p are non-negative

$$\forall i \in I.\, w(i) \cdot h_p(F(i)) = 0$$

$$\forall i \in I.\, w(i) = 0 \lor h_p(F(i)) = 0$$

From the definition $\forall x \in P.\, w(x) > 0$

$$\forall i \in I.\, h_p(F(i)) = 0$$

This statement is independent of the function w, thus the lemma holds for the discrete variation.

The non-discrete variant can be proven similarly. □

The robustness according to my definition can be contraintuitive when applied to functions that are not continuous on the input interval. An extreme example of such function is the Dirichlet function $D : \mathbb{R} \to \mathbb{R}$.

$$D(x) = \begin{cases} 1 & \text{if } x \in \mathbb{Q} \\ 0 & \text{if } x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

Lemma 2.2.4. *The Dirichlet function D is robust on any interval $\langle k, l \rangle$ where k, l are rational numbers, $k \neq l$.*

Proof. As shown in the Example 3.1.1 of [5], the Dirichlet function has Lebesgue integral on interval $[0, 1]$ with a value equal to 0.

Lets prove that for every rational number $a \in \mathbb{Q}$ and real number $b \in \mathbb{R}$ following statements are equivalent:

1. b is rational
2. $a + b$ is rational

1. $\implies$ 2.:
Since both a and b are rational, by definition they can be written as a fraction of integers

$$a = \frac{p}{q}, b = \frac{p'}{q'}$$

The sum can be expressed as a fraction of integers, thus is also rational.

$$a + b = \frac{pq' + p'q}{qq'}$$

2. $\implies$ 1.:
The b can be written using the rational a and the sum of a and b.

$$b = (a + b) + (-a)$$

Now since both $(-a)$ and $(a + b)$ are rational, b was already proven to be rational in the opposite side implication.

Lets show that the integral of Dirichlet function D is equal to 0 on any interval bounded by rational numbers. The limits of the definite integral can be transformed by removing k and adding it to the argument of D. As was proven previously, for rational number k, $D(x+k) = D(x)$.

$$\int_k^l D(x)dx = \int_0^{l-k} D(x+k)dx = \int_0^{l-k} D(x)dx$$

As the dirichlet function is nonnegative, following is true.

$$0 \leq \int_0^{l-k} D(x)dx \leq \int_0^{\lceil l-k \rceil} D(x)dx$$

The right-hand side integral can be split into unit-long parts.

$$\int_0^{\lceil l-k \rceil} D(x)dx = \int_0^1 D(x)dx + \int_1^2 D(x)dx + \ldots + \int_{\lceil l-k \rceil -1}^{\lceil l-k \rceil} D(x)dx$$

Finally, as every unit integral has rational bounds, it can be transformed to integral with 0 as lower bound like already shown.

$$\int_u^{u+1} D(x)dx = \int_0^1 D(x+u)dx = \int_0^1 D(x)dx = 0$$

$$0 \leq \int_k^l D(x)dx \leq \int_0^{\lceil l-k \rceil} D(x)dx = 0$$

To show the robustness of Dirichlet function, the quantitative robustness with respect to constant weight function w_1 and identity as the evaluation function can be used.

$$Q_{w_1,id}(\langle k,l \rangle) = \frac{\int_k^l 1 \cdot D(x)dx}{\int_k^l 1} = \frac{0}{l-k} = 0$$

□

Since this thesis focuses on robustness of the discrete input regions, I will further assume only discrete version of the robustness problem.

Definition 2.2.3. Strict qualitative robustness of function F on input region I. Function $F : P \to Q$ is strictly (qualitatively) robust on input region $I \subseteq P$ if and only if for some $p \in P$ it is robust on this region with respect to evaluation function $\overline{h_p}$ defined as follows:

$$\overline{h_p}(q) = \begin{cases} 0 & F(p) = q \\ 1 & F(p) \neq q \end{cases}$$

Lemma 2.2.5. *Function F is strictly robust on input region I if and only if for each two inputs $i, j \in I$, the function F assigns the same value to them.*

Proof. Proof of equivalence in this lemma is done by prooving corresponding implications.

For empty input region $I = \emptyset$, the statement holds trivially. Further in the proof I will always assume nonempty input region.

First the left-to-right implication. Let function F be strictly robust on input region I, $\overline{h_p}$ being the evaluation function. As shown in the proof of Lemma 2.2.3, for all instances from input region $i \in I$ holds

$$\overline{h_p}(F(i)) = 0$$

By definition of evaluation function $\overline{h_p}$ it also holds

$$F(p) = F(i)$$

As this holds for every input $i \in I$, for every two inputs $i, j \in I$:

$$F(i) = F(p) = F(j)$$

Now for the opposite implication: Let for every $i, j \in I$, $F(i) = F(j)$. Let $p \in I$. As $\overline{h_p}(F(p)) = 0$ and for every input $i \in I$ it holds that $F(i) = F(p)$, $\overline{h_p}(F(i)) = 0$ for every input from the input region I. The F is thus strictly robust on input region I. □

The strict robustness is equivalent to the robustness as defined in [6]. The t-target robustness from this article is equivalent to my definition of robustness with respect to evaluation function $h_t : Q \to \mathbb{R}$

$$h_t(q) = \begin{cases} 0 & \text{if } q \neq t \\ 1 & \text{if } q = t \end{cases}$$

Finally the term $Pr(R(u, \tau))$ is equal to quantitative robustness with respect to weight function w_1 and evaluation function h_u on input region $R(u, \tau)$.

2.2.2 Definition of input regions

As both the qualitative and quantitative robustness rely on subsets of feasible inputs, definition of these is needed.

I provide definition of two classes of input regions, input regions based on the Hamming distance and input regions with fixed indices. The input region based on the Hamming distance $R(\vec{u}, r)$ contains all input vectors that have at most r bits changed. The input region with fixed indices $R(\vec{u}, I)$ specifies set of indices I on which the input vector may differ form I. Definition are taken from [1].

Definition 2.2.4. Input region based on the Hamming distance. For an input $\vec{u} \in \mathbb{B}_{\pm 1}^{n_1}$ and an integer $r \geq 0$, let $R(\vec{u}, r) := \{\vec{x} \in \mathbb{B}_{\pm 1}^{n_1} \mid HD(\vec{x}, \vec{u}) \leq r\}$, where $HD(\vec{x}, \vec{u})$ denotes the Hamming distance between $\vec{x}$ and $\vec{u}$.

Intuitively, $R(\vec{u}, r)$ includes input vectors that differ from $\vec{u}$ on at most r positions. Examples of such input regions are:

$$\begin{aligned} R((1,1,1,1),1) = \{&(1,1,1,1), \\ &(-1,1,1,1),(1,-1,1,1),(1,1,-1,1),(1,1,1,-1)\} \end{aligned}$$

$$\begin{aligned} R((-1,1,-1),2) = \{&(-1,1,-1), \\ &(1,1,-1),(-1,-1,-1),(-1,1,1), \\ &(-1,-1,1),(1,1,1),(1,-1,-1)\} \end{aligned}$$

Lemma 2.2.6. *Let $\vec{u}$ be a vector from $\mathbb{B}_{\pm 1}^{d}$. Then $||R(\vec{u}, r)|| = \sum_{i=0}^{\min(r,d)} \binom{d}{i}$*

Proof. $R(\vec{u}, r)$ is union of sets

$$R(\vec{u}, r) = \bigcup_{i=0}^{r} \{\vec{x} \mid HD(\vec{x}, \vec{u}) = i\}$$

These sets are disjoint as elements of each have different number of positions changed. The size of each of these sets is equal to

$$||\{\vec{x} \mid HD(\vec{x}, \vec{u}) = i\}|| = \binom{d}{i}$$

because they consist of d positions, out of which i are choosen to be altered. Finally, as for i larger than d, $\binom{d}{i} = 0$, the statement holds. □

Definition 2.2.5. Input region based on fixed bits. For an input $\vec{u} \in \mathbb{B}_{\pm 1}^{n_1}$ and set of indices $I \subseteq [n_1]$, let $R(\vec{u}, I) := \{\vec{x} \in \mathbb{B}_{\pm 1}^{n_1} \mid \forall i \in I.\, x_i = u_i\}$.

The input region based on fixed bits $R(\vec{u}, I)$ does specify the positions which are fixed to the values of the base vector $\vec{u}$. Examples of such input regions are:

$$R((1,1,-1,1),\{1,3,4\}) = \{(1,1,-1,1),(1,-1,-1,1)\}$$

$$\begin{aligned} R((1,1,1,1),\{3\}) = \{&(1,1,1,1),(-1,1,1,1),(1,-1,1,1),(-1,-1,1,1),\\ &(1,1,1,-1),(-1,1,1,-1),(1,-1,1,-1),(-1,-1,1,-1)\} \end{aligned}$$

Lemma 2.2.7. *Let $\vec{u}$ be a vector from $\mathbb{B}_{\pm 1}^{d}$, $I \subseteq [n_1]$. Then $||R(\vec{u}, I)|| = 2^{d-||I||}$*

Proof. Each element of I does fix a single position in the input vector. The number of variable positions is $d - ||I||$, each being instance of $+1$ or -1. This makes the number of variants $2^{d-||I||}$. □

Remark. Lemmas 2.2.6 and 2.2.7 allow for fast computation of the size of the input region.

3 Answer set programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems [7]. ASP is particularly suited for solving difficult combinatorial search problems [8]. ASP is somewhat closely related to propositional satisfability checking (SAT) in sense that the problem is represented as logic program. Difference is in computational mechanism of finding solution.

3.1 Extended logic program

Extended logic program is a finite set of facts, rules and constraints each consisting literals. A literal is an atom or its negation. An atom is the elementary construct for representing knowledge [9]. Each atom constitutes a single variable, it can be seen as a possible feature of solution.

In order to be able to distinguish between a query which fails in the sense that it does not succeed and a query which fails in the stronger sense that its negation succeeds, extended logic programs allow for the *classical negation* $\neg$ in addition to the *negation-as-failure not* [10].

Further I provide formal definition of extended logic programs according to [10, 11]. I will assume the notation of signature as defined in [12].

3.1.1 Syntax of Extended logic program

Definition 3.1.1. Terms [12]. Let Σ be a signature. The set of Σ-terms of sort σ is the smallest set of expressions satisfying the following properties:

- Each variable x of sort σ is a term of sort σ, provided that $\sigma \in \Sigma^S$.
- Each constant symbol $c \in \Sigma^C$ of sort σ is a Σ-term of sort σ.
- If $f \in \Sigma^F$ is a function symbol of arity $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ and t_i is a Σ term of σ_i, for $i = 1, \ldots, n$, then $f(t_1, \ldots, t_n)$ is a term of sort σ.

Definition 3.1.2. Atoms, literals [12]. Let Σ be a signature. A Σ-atom is an expression of the form

$$p(t_1, \ldots, t_n)$$

where $p \in \Sigma^P$ is a predicate symbol of arity $\sigma_1 \times \cdots \times \sigma_n$ and for $i = 1, \ldots, n$, t_i is a Σ-term of sort σ_i.

A Σ-literal is a formula of the form

$$\varphi \text{ or } \neg\varphi$$

where φ is a Σ-atom.

For both function and predicate symbols infix notation can be used if was estabilished (e.g. $+, -, =, \leq, \ldots$). If this notation would result in ambiguity, brackets should be used.

Example 3.1.1. *Let Σ_1 be a signature with sorts*

$$\sigma_{\mathbb{B}} = \{True, False\}, \sigma_{\mathbb{N}} = \mathbb{N}_0$$

and with function symbols $\vee, \wedge$ of arity $\sigma_{\mathbb{B}} \times \sigma_{\mathbb{B}} \rightarrow \sigma_{\mathbb{B}}$, and function symbols $=, <$ of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{N}} \rightarrow \sigma_{\mathbb{B}}$. Let p be a predicate of arity $\sigma_{\mathbb{B}}$, and q be a predicate of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{B}}$ Also let variables X, Y, Z be of sort $\sigma_{\mathbb{N}}$ and let there be no variables of sort $\sigma_{\mathbb{B}}$.
Then expressions

$$\begin{gathered} True,\ False, \\ True \vee False,\ False \wedge True, \\ (42 = 42),\ (X = 17),\ (3 < Y) \vee (6 = 6), \\ [(5 = 1) \wedge True] \vee False \end{gathered}$$

are some of the Σ-terms of sort $\sigma_{\mathbb{B}}$.
Expressions

$$\begin{gathered} p(True),\ p(False), \\ p([(5 = 1) \wedge True] \vee False), \\ q(0, True),\ q(42, False), \\ q(123, (42 = 42)),\ q(93, True \vee (Z < 17)) \end{gathered}$$

are some of the Σ-atoms.

Definition 3.1.3. Rule [11]. Let Σ be a signature. A Σ-rule (Σ-formula) r is an expression of the form

$$l_0 \, or \, \ldots \, or \, l_k \leftarrow l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n.$$

where each l_i is a Σ-literal. The following notation is used:

$$\begin{aligned} \text{head}(r) &= \{l_0, \ldots, l_k\} \\ \text{body}(r) &= \{l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n\} \\ \text{body}^+(r) &= \{l_{k+1}, \ldots, l_m\} \\ \text{body}^-(r) &= \{\text{not}\, l_{m+1}, \ldots, \text{not}\, l_n\} \end{aligned}$$

Further, if $\text{head}(r) = \emptyset$, the rule is called a constraint and is written as

$$\leftarrow l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n.$$

If $\text{body}(r) = \emptyset$, the rule is called a fact and is written as

$$l_0 \, or \, \ldots \, or \, l_k.$$

The term not is called default negation.

Definition 3.1.4. Logic program [11]. Logic program is a pair (Σ, Π) where Σ is a signature and Π is a collection (set) of Σ-rules.

In this thesis, I will be mostly working with logic programs that have zero or one literal in the head of each of their rules. While allowing for multiple literals in the head of a rule can allow for otherwise impossible to achieve expressions, it does introduce a large portion of nontriviality and even computational cost. General logic programs are in the complexity class Σ_2^P [13], logic programs without disjunction in head are NP-complete [14] and logic programs with neither disjunction nor default negation belong to P [11].

Example 3.1.2. *An example of a logic program with signature Σ_1 from Example 3.1.1 is a pair (Σ_1, Π_1) where Π_1 is a set*

$$\Pi_1 = \begin{cases} p(True). \\ q(5, True). \\ p(False) \leftarrow \text{not}\, p(True). \\ q(Y, True) \leftarrow q(X, (Y < X)). \end{cases}$$

The logic program is often denoted only by its second element Π. In that case its signature consists of symbols occuring in the program.

3.1.2 Semantics of extended logic program

For the definition of the extended logic programs to be usefull, definition of stable models is needed. Many definitions have been created [15]. Here I will show the definition using the reduct of a logic program [11], which was used for the ASP solver I am using for the implementation of verification, Clingo [16].

First of all, the logic program needs to be grounded. Grounding is a process in which we substitute each rule in the program for its equivalent rules with constant symbols only.

The semantics of function symbols is usually defined in the grounder (the program that does the grounding) thus the programs can be grounded before the start of solving of the logic program.

Definition 3.1.5. Ground program [11]. Terms, literals, and rules of program Π with signature σ are called ground if they contain no variables and no function symbols. A program is called ground if all its rules are ground. A rule r' is called a ground instance of a rule r of Π if it is obtained from r by:

- replacing r's variables by properly typed ground terms of σ;
- replacing r's function terms by their values.

A program $gr(\Pi)$ consisting of all ground instances of all rules of Π is called the ground instantiation of Π.

Example 3.1.3. *The ground instantiation of the logic program* (Σ_1, Π_1) *defined in the Example 3.1.2 can be found as follows:*

The fact $p(True)$. *does not contain any variable nor function symbol. For this, it is ground. The same holds for the fact* $q(5, True)$. *and the rule* $p(False) \leftarrow \text{not}\, p(True)$.

To ground the rule $q(Y, True) \leftarrow q(X, (Y < X))$.*, this rule should be first substitued for each rule with pair* (X, Y) *substitued for every element from* $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{N}}$*, that is every element from* $\mathbb{N}_0 \times \mathbb{N}_0$*. Then in the second parameter of atom in the body, the expression substitued for* $(Y < X)$ *would be replaced with True or False, depending on the substitution for X and Y. This would lead to the ground instantiation* $gr(\Sigma_1, \Pi_1)$ *being infinite.*

To fight this, we can allow only the instantiation of rule r in which only the substitutions of variables in the $\text{body}^{+}(r)$ *leading to the already existing* *atoms are allowed. After the stable model will be defined, it will be easy to see that this constraint does not remove any stable models of program, as every atom in it has to also be in the head of some rule.*

In this example, we already know that the atom $p(5, True)$ does exist. This means we can substitute $X \equiv 5$ and $(Y < X) \equiv True$. Also as $(Y < X) \equiv (Y < 5) \equiv True$, to successfully instantiate the rule, it has to hold that $Y < 5$. There are no other constraints on the rule, thus the rule instantiates into rules:

$$\begin{aligned}
Y = 0 &: q(0, True) \leftarrow q(5, True).\\
Y = 1 &: q(1, True) \leftarrow q(5, True).\\
Y = 2 &: q(2, True) \leftarrow q(5, True).\\
Y = 3 &: q(3, True) \leftarrow q(5, True).\\
Y = 4 &: q(4, True) \leftarrow q(5, True).
\end{aligned}$$

But now we have introduced new atoms on predicate q. We have to also add all instances of the rule, in which these atoms are allowed. As the individual atoms only differ in the first term, the instances are seen easily. The instances added by the new rules also do not introduce new atoms on predicate q, the ground instance is thus in this case finite.

$$\begin{aligned}
X = 4, Y = 0 &: q(0, True) \leftarrow q(4, True).\\
X = 4, Y = 1 &: q(1, True) \leftarrow q(4, True).\\
X = 4, Y = 2 &: q(2, True) \leftarrow q(4, True).\\
X = 4, Y = 3 &: q(3, True) \leftarrow q(4, True).\\
X = 3, Y = 0 &: q(0, True) \leftarrow q(3, True).\\
X = 3, Y = 1 &: q(1, True) \leftarrow q(3, True).\\
X = 3, Y = 2 &: q(2, True) \leftarrow q(3, True).\\
X = 2, Y = 0 &: q(0, True) \leftarrow q(2, True).\\
X = 2, Y = 1 &: q(1, True) \leftarrow q(2, True).\\
X = 1, Y = 0 &: q(0, True) \leftarrow q(1, True).
\end{aligned}$$

The full ground instantiation $gr((\Sigma_1, \Pi_1))$ is thus the program:

$$gr((\Sigma_1, \Pi_1)) = \begin{cases} p(True). \\ q(5, True). \\ p(False) \leftarrow \text{not}\, p(True). \\ q(0, True) \leftarrow q(5, True). \\ q(1, True) \leftarrow q(5, True). \\ q(2, True) \leftarrow q(5, True). \\ q(3, True) \leftarrow q(5, True). \\ q(4, True) \leftarrow q(5, True). \\ q(0, True) \leftarrow q(4, True). \\ q(1, True) \leftarrow q(4, True). \\ q(2, True) \leftarrow q(4, True). \\ q(3, True) \leftarrow q(4, True). \\ q(0, True) \leftarrow q(3, True). \\ q(1, True) \leftarrow q(3, True). \\ q(2, True) \leftarrow q(3, True). \\ q(0, True) \leftarrow q(2, True). \\ q(1, True) \leftarrow q(2, True). \\ q(0, True) \leftarrow q(1, True). \end{cases}$$

The Example 3.1.3 also illustrates the need for the logic programs to be well written. As was shown in this example, due to a single flawed rule, the size of the ground instantiation of the logic program is quadratic given the highest number in the first parameter of atoms of the predicate q.

Grounders differ in the exact algorithm of the grounding as well as in the allowed signature of the logic program. The exact full syntax and semantics of the ASP grounder GRINGO can be found in [17].

Definition 3.1.6. Partial interpretation of a signature. A partial interpretation S of a signature Σ is a set of Σ-literals in which for each Σ-atom φ there is either φ, $\neg\varphi$ or neither of them.

A partial interpretation can be used to denote the solution of a logic program. It allows for 3-valued logic — an atom either is true, is false or is unknown. The atom being unknown can specify an atom, that was not derived, while the atom being false strictly says it cannot be true.

For a partial interpretation to be a solution (answer set) of the program, it further needs to be consistent with the logic program and all its literals have to be founded in the program (the set has to be minimal).

Each rule r is seen as a statement $\text{head}(r)$ holds if all the literals from the $\text{body}^+(r)$ hold and at the same time no literal from the $\text{body}^-(r)$ hold. The facts have empty body, thus their head has to hold in every solution. On the other hand, the body of constraints can never be consistent with the solution as its head is empty thus can not be consistent with solution. Further follows a more formal definition.

Definition 3.1.7. Consistent partial interpretation. Partial interpretation S is said to be consistent with the body of rule r if and only if $\text{body}^+(r) \subseteq S$ and $\text{body}^-(r) \cap S = \emptyset$.

Partial interpretation S is said to be consistent with the head of rule r if and only if $\text{head}(r) \cap S \neq \emptyset$

Partial interpretation S is said to be consistent with a rule r if and only if S is not consistent with $\text{body}(r)$ or S is consistent with $\text{head}(r)$.

Partial interpretation S is said to be consistent with a logic program (Σ, Π) if and only if it is consistent with all of its rules.

Definition 3.1.8. Basic logic program. A logic program (Σ, Π) is called basic if there is no negative atom in body of any of its rules nor constraints, that is

$$\forall r \in \Pi.\ \text{body}^-(r) = \emptyset$$

Now for the answer sets (partial interpretations that are solutions) of a logic program a reasonable constraints on them are to satisfy (be consistent with) the logic program and to only contain information that needs to be true (each answer set has to be minimal).

Definition 3.1.9. Answer set of a basic logic program [11]. Let (Σ, Π) be a basic logic program and S be a partial interpretation of Σ. S is an answer set (solution) for Π if S is minimal (in the sense of set-theoretic inclusion) among the partial interpretations consistent with Π.

For a grounded basic logic program, finding its answer set (solution) is easy. It can be built incrementally. Heads of facts have to be included in the partial interpretation for it to be an answer set. Then in each step all heads of rules, that have its bodies consistent with the previous set, are added into the set. This is done as long as any literal is added into the partial interpretation.

Lemma 3.1.1. *Incremental building of the answer set of the basic logic program. Let (Σ, Π) be a basic logic program with at most one literal in the head of every rule. Let*

$$S_0 = \emptyset,$$

$$S_{i+1} = S_i \cup \{\varphi \mid \{\varphi\} = \mathrm{head}(r), \mathrm{body}^+(r) \subseteq S_i\} \textit{ for each } i \geq 0.$$

Denote by S_∞ the union of all S_i for $i \geq 0$. If it holds that:

- *for every Σ-atom φ, S_∞ contains at most one of literals $\varphi, \neg\varphi$, and*
- *S_∞ is consistent with every constraint of Π,*

then S_∞ is the only answer set of program (Σ, Π). Else (Σ, Π) has no answer set.

Proof. S_∞ does contain only (some of) heads of rules from Π, each of these being Σ-literals. If S_∞ does also not contain both φ and $\neg\varphi$ for any Σ-atom φ, S_∞ is a partial interpretation of Σ.

By definition S_∞ contains the literal from the head of each rule, that has its body consistent with the S_∞. For this the S_∞ is consistent with every non-constraint rule of the logic program.

Let $rank : S_\infty \to \mathbb{N}_0$ be a function that assigns to every literal $\varphi \in S_\infty$ the lowest value n such that $\varphi \in S_n$. Let there be some answer set A of (Σ, Π) such that $S_\infty \setminus A = \Delta \neq \emptyset$. Let α be a literal from Δ with the lowest $rank$. By definition, $rank(\alpha) \geq 1$ as S_0 is empty. As the literal $\alpha \in S_{rank(\alpha)}$ and $\alpha \notin S_{rank(\alpha)-1}$, either α is in a head of some fact (thus $rank(\alpha) = 1$ and α needs to be in A for it to be consistent), or there must be some rule $r \in \Pi$ such that $\mathrm{body}^+(r)$ is not consistent with $S_{rank(\alpha)-2}$ and is consistent with $S_{rank(\alpha)-1}$. (The rule r is what made the literal α in $S_{rank(\alpha)}$.) But A does contain all the literals from $S_{rank(\alpha)-1}$. (Literal α has the lowest rank of all literals from S_∞ not included in A.) Thus A is not consistent with r and is not an answer set of (Σ, Π). By contradiction, for every answer set B of (Σ, Π), $S_\infty \subseteq B$. As the answer set is by definition minimal, the S_∞ is the only possible answer set of (Σ, Π). □

Example 3.1.4. *Let (Σ, Π) be a logic program,*

$$\Pi = \begin{cases} p(a). \\ p(b) \leftarrow p(a). \\ p(c) \leftarrow p(b). \\ p(x). \\ p(y) \leftarrow p(x), p(a). \\ p(z) \leftarrow p(x), p(c). \\ p(n) \leftarrow p(a), p(k). \end{cases}$$

When building the solution, we first start with an empty set.

$$S_0 = \{\}$$

In the first step only the facts have bodies consistent with the partial solution S_0. We can add literals in their heads into the partial solution.

$$S_1 = \{p(a), p(x)\}$$

With these two literals in the partial solution, rules $p(b) \leftarrow p(a)$. and $p(y) \leftarrow p(x), p(a)$. have their bodies consistent with the partial solution S_1. For this we add literals in heads of these rules into the partial solution.

$$S_2 = \{p(a), p(x), p(b), p(y)\}$$

In the next step, another rule $p(c) \leftarrow p(b)$. has its body consistent with the partial solution S_2. We add the literal from its head into the partial solution.

$$S_3 = \{p(a), p(x), p(b), p(y), p(c)\}$$

This addition into the partial solution makes the body of yet another rule $p(z) \leftarrow p(x), p(c)$. to be consistent with the partial solution.

$$S_4 = \{p(a), p(x), p(b), p(y), p(c), p(z)\}$$

No other literals have to be added into the partial solution. S_4 is the only solution of logic program (Σ, Π).

The only literal from the heads of rules that have not been included into the partial solution is $p(n)$. This literal relies on some unknown literal $p(k)$ that is not in any head of rules of Π. Another case of literal that is not in a solution of logic program is shown in the following example.

Example 3.1.5. *Let* (Σ, Π) *be a logic program,*

$$\Pi = \begin{cases} p(a) \leftarrow p(b). \\ p(b) \leftarrow p(a). \end{cases}$$

This logic program a single solution, that is an empty set. While the partial interpretation $\{p(a), p(b)\}$ *is consistent with the logic program, it is not the minimal set.*

The simple way of building answer sets of a basic programs, can be extended to all logic programs. In basic programs we have assumed that no rule with negative literals does exist in the logic program. To fight the negative literals in a general extended logic program, we can first define a partial interpretation of Σ that constitutes the possible answer set, then take all rules $r \in \Pi$ that have no intersection of their $\text{body}^-(r)$ with our partial interpretation and into the reduced logic program include only the nonnegative $\text{body}^+(r)$ from them. If this reduced program yields the same answer set as the partial interpretation we have defined, it is an answer set.

Definition 3.1.10. Reduct of a logic program [11]. Let (Σ, Π) be a logic program, S be a partial interpretation of Σ. Let Π^S be a set of rules such that

$$\Pi^S = \{\text{head}(r_i) \leftarrow \text{body}^+(r_i). \mid r_i \in \Pi, \text{body}^-(r_i) \cap S = \emptyset\}$$

Then Π^S is called a reduct of Π relative to the partial interpretation S.

Example 3.1.6. *In the reduct can be encountered a new type of rule that was not yet discussed here. Let* $\Pi = \{\leftarrow \text{not}\, \varphi.\}$, $S = \{\}$ *for some atom* φ. *The reduct* Π^S *contains a single rule with both head and body empty. Based on the Definition 3.1.7, no partial interpretation can be consistent with such rule as no partial interpretation can be consistent with an empty head and every partial interpretation is consistent with an empty body. This is however a desired behaviour, the interpretation S was defined in a way it was not consistent with the rule so it is only right the reduct can not have any answer set.*

Definition 3.1.11. Answer set of general grounded logic program [11]. A partial interpretation S of Σ is an answer set for (Σ, Π) if S is an answer set for Π^S.

Definition 3.1.11 gives a direct way of computing answer sets of any logic programs. For every possible subset S of grounded atoms in program Π, the reduct Π^S can be constructed and evaluated for its answer set. If the found answer set is equal to the subset of grounded atoms S, it is said to be also an answer set of the program Π.

Let's illustrate the computation of an answer set of general logic program on two examples.

Example 3.1.7. *Let* (Σ, Π), *where*

$$\Pi = \{\varphi \leftarrow \varphi., \psi \leftarrow \text{not}\, \varphi.\},$$

be a logic program with two atoms φ, ψ. *There are 4 subsets of set of all atoms. First the reduct relative to the subset is made, on it the calculation of its answer set* (AS) *is made. If the answer set of reduct is equal to the subset, it is an answer set of the whole logic program.*

S	Π^S	AS *of* Π^S
$\{\}$	$\varphi \leftarrow \varphi.$ $\psi \leftarrow .$	$\{\psi\}$
$\{\varphi\}$	$\varphi \leftarrow \varphi.$	$\{\}$
$\{\psi\}$	$\varphi \leftarrow \varphi.$ $\psi \leftarrow .$	$\{\psi\}$
$\{\varphi, \psi\}$	$\varphi \leftarrow \varphi.$	$\{\}$

There is only a single answer set of (Σ, Π), *that is* $\{\psi\}$. *Similiar to Example 3.1.5, in the reduct* $\Pi^{\{\varphi\}}$ *partial interpretation* $\{\varphi\}$ *is not an answer set as it is not minimal.*

Example 3.1.8. *Let* (Σ, Π), *where*

$$\Pi = \{\varphi \leftarrow \text{not}\, \psi., \psi \leftarrow \text{not}\, \varphi.\},$$

be a logic program with two atoms φ, ψ. *Again, there are 4 subsets of set of all atoms.*

S	Π^S	*AS of* Π^S
$\{\}$	$\varphi \leftarrow .$ $\psi \leftarrow .$	$\{\varphi, \psi\}$
$\{\varphi\}$	$\varphi \leftarrow .$	$\{\varphi\}$
$\{\psi\}$	$\psi \leftarrow .$	$\{\psi\}$
$\{\varphi, \psi\}$		$\{\}$

This time there are two answer sets of (Σ, Π), $\{\varphi\}$ *and* $\{\psi\}$.

3.2 Clingo

Clingo is an integrated ASP system, consisting of a grounder Gringo and solver Clasp [18]. In the following section I will show basics of the Clingo language, gringo as translation from Clingo to Aspif language and solving with clasp.

3.2.1 Clingo language

Clingo language [19] is used for transcribing ASP programs and their extended versions.

4 ASP encoding of BNN robustness

5 Evaluation

6 Conclusion

7 Introduction

Deep neural networks (DNN) are state-of-the-art technology. They are used in many real-world applications e.g. medicine, self-driving cars, autonomous systems, many of which are critical. Deep neural networks used in natural language processing have up to hundreds of billions of parameters [20]. That is too many for people to comprehend. Tools for the automation of DNN verification are needed.

Verification of neural networks is split into two categories. First, qualitative verification, searches through input space looking for any adversarial input, e.g. an input which results in wrong output. Second, quantitative verification, searches through input space determining the size of the part of the input space giving adversarial outputs.

For the qualitative disproving of the verity of general DNNs, many algorithms for finding adversarial inputs have already been developed [21, 22, 23]. The DNN can be qualitatively disproven by finding any adversarial input. Proving the nonexistence of adversarial input is usually made by generating constraints on adversarial inputs and then showing there can be no such input [24]. This is computationally much more difficult and prone to errors emerging from inaccuracies of floating point numbers operations. It is often made with the use of a simplex algorithm combined with the satisfiability modulo theories (SMT) paradigm [24, 25, 26]. As far as I am aware, there is no reasonably quick quantitative validator of general DNNs yet.

As DNNs generally base their computations on floating point numbers, the computation of output is hard. For this, quantization, a new branch of DNN development, where instead of 32 or 64-bit long floating point numbers, low-bit-width (eg. 4-bit) fixed point numbers are used. This mitigates the high cost of computation and the need for high-end devices while also lowering the power consumption. Extreme examples of quantization are Binary neural networks (BNN). These use binary parameters for their computations, resulting in much cheaper computational price, both when learning and in production, while maintaining near state-of-the-art results [27].

While both qualitative and quantitative BNN verificators exist, they scale only up to millions of parameters for qualitative verification [28] and tens of thousands of parameters for quantitative verification [6].

These verificators can even compute on natural numbers only, without errors.

In this thesis, I have implemented a quantitative BNN verificator using the answer set programming paradigm in the framework Clingo from Potassco. In this implementation I have derived an efficient encoding of BNN computation and of input regions under the hamming distance and under fixed indices. Further I have encapsulated this verificator together with a program for parsing BNNs into a Clingo-readable format in an easy to use framework. As far as I am concerned, this is the first such implementation using the ASP paradigm.

I have also evaluated the speed of this verificator on BNN models of various architectures based of PyTorch deep learning platform provided by NPAQ [29] trained in [6] on MNIST [30] dataset. The verificator works especially well on robust networks when the input region is specified by fixed indices.

8 Used tools

8.1 Answer set programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems [7]. ASP is particularly suited for solving difficult combinatorial search problems [8]. ASP is somewhat closely related to propositional satisfability checking (SAT) in sense that the problem is represented as logic program. Difference is in computational mechanism of finding solution.

8.1.1 Syntax and semantics of ASP

Logic program Π in ASP is a set of rules consisting of atoms. An atom is the elementary construct for representing knowledge [9]. Each atom constitutes a single variable which either is or is not in the answer set (solution). An atom can be seen as a possible feature of the answer set. Each rule $r_i \in \Pi$ has form of

$$\mathrm{head}(r_i) \leftarrow \mathrm{body}(r_i). \tag{8.1}$$

$\mathrm{head}(r_i)$ is then a single atom p_0, while $\mathrm{body}(r_i)$ is a set of zero or more literals (atoms or negated atoms) $\{p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots \mathrm{not}\, p_n\}$.

$$p_0 \leftarrow p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots, \mathrm{not}\, p_n. \tag{8.2}$$

Further the $\mathrm{body}(r_i)$ can be split between $\mathrm{body}^+(r_i) = \{p_1, \ldots, p_m\}$ and $\mathrm{body}^-(r_i) = \{p_{m+1} \ldots p_n\}$. If $\forall r_i \in \Pi. \mathrm{body}^-(r_i) = \emptyset$, meaning there are no negative literals in the program, then Π is called *basic*.

Semantically rule (2.2) means *If all atoms from* $\mathrm{body}^+(r_i)$ *are included in answer set and no atom from* $\mathrm{body}^-(r_i)$ *is in answer set, then* $\mathrm{head}(r_i)$ *has to be in the set.*

Set of atoms X is closed under a basic program Π if for any rule $r_i \in \Pi. \mathrm{body}(r_i) \subseteq X \implies \mathrm{head}(r_i) \in X$. For general case the concept of *reduct of a program* Π *relative to a set* X *of atoms* is needed.

$$\Pi^X = \{\mathrm{head}(r_i) \leftarrow \mathrm{body}^+(r_i) \mid r_i \in \Pi, \mathrm{body}^-(r_i) \cap X = \emptyset\} \tag{8.3}$$

This program is always basic as it only contains positive atoms.

Let's denote $\mathrm{Cn}(\Pi)$ the minimal set of atoms closed under a basic program Π, that is

$$\mathrm{Cn}(\Pi) \supseteq \{\mathrm{head}(r_i) \mid r_i \in \Pi, \mathrm{body}(r_i) \subseteq \mathrm{Cn}(\Pi)\} \tag{8.4}$$

Such set is said to constitute program Π. Set X of atoms is said to be answer set of Π iff

$$\mathrm{Cn}(\Pi^X) = X \tag{8.5}$$

In other words, the answer set is a model of Π such that its every atom is grounded in Π.

Let's illustrate this property on an example.

$$\Pi = \{p \leftarrow p., q \leftarrow \mathrm{not}\, p.\} \tag{8.6}$$

There are 4 subsets of set of all atoms. First the reduct relative to the subset is made, on it the calculation of constituting set is made. If $X = \mathrm{Cn}(\Pi^X)$, then X is one of answer sets.

X	Π^X	$\mathrm{Cn}(\Pi^X)$
$\{\}$	$p \leftarrow p.$ $q \leftarrow .$	$\{q\}$
$\{p\}$	$p \leftarrow p.$	$\{\}$
$\{q\}$	$p \leftarrow p.$ $q \leftarrow .$	$\{q\}$
$\{p, q\}$	$p \leftarrow p.$	$\{\}$

There is only a single answer set of Π, that is $\{q\}$. Note that $\{p\}$ is not an answer set as it is not minimal. Also, there is an interesting type of rule in the table, $p \leftarrow$ .. This type of rule is commonly reffered as *fact* and ensures that atom p is always included in the answer set.

Another toy example is $\Pi = \{p \leftarrow \mathrm{not}\, q., q \leftarrow \mathrm{not}\, p.\}$. Again, there are 4 subsets of set of all atoms.

X	Π^X	$\mathrm{Cn}(\Pi^X)$
$\{\}$	$p \leftarrow .$ $q \leftarrow .$	$\{p, q\}$
$\{p\}$	$p \leftarrow .$	$\{p\}$
$\{q\}$	$q \leftarrow .$	$\{q\}$
$\{p, q\}$		$\{\}$

This time there are two answer sets of Π, $\{p\}$ and $\{q\}$. As we can see, the double not forms a XOR — exactly one of atoms p, q has to be in the answer set.

8.1.2 Language extensions

Writing logic programs only using rules of form (2.2) would be very hard. For this, many language extensions have been developed to shorten programs, simplifying both readability and computation of stable models.

In this section I will only show the constraint, as it can be very easily transformed into base ASP rule. I will describe more language extensions provided with Clingo framework in section (2.2).

Constraint

$$\leftarrow p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots, \mathrm{not}\, p_n.$$

Constraint is a type of language extension with semantics that its body can not be in the answer set. It is defined as the following rule [8]:

$$f \leftarrow \mathrm{not}\, f, p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots, \mathrm{not}\, p_n. \tag{8.7}$$

where f is a new atom (atom not used anywhere else).

Let's consider a logic program Π, with a constraint rule (8.7) call it r. Let X be any set of atoms s.t. $\{p_1, \ldots, p_m\} \subseteq X, \{p_{m+1}, \ldots, p_n\} \cap X = \emptyset$. If $f \in X$, then the rule $f \leftarrow \mathrm{body}^+(r)$. is not in Π^x, thus $f \notin \mathrm{Cn}(\Pi^X)$ and $X \neq \mathrm{Cn}(\Pi^X)$. If $f \notin X$, then the rule $f \leftarrow \mathrm{body}^+(r)$. is in Π^X, thus $f \in \mathrm{Cn}(\Pi^X)$ and $X \neq \mathrm{Cn}(\Pi^X)$. This shows that no set consistent with constraint rule can be answer set.

If X is not consistent with rule r and $f \notin X$, then X contains some negative literal of r ($\exists x \in X. x \in \text{body}^-(r)$) and the rule r is not in the reduct Π^X, or X does not contain some positive literal of r, thus the rule r is not applied. Either way, $f \notin X, f \notin \text{Cn}(\Pi^X)$.

Such rule is usefull for constraining the answer set. Lets assume expansion of example from the previous section.

$$\Pi = \{p_1 \leftarrow \text{not}\, q_1., \ q_1 \leftarrow \text{not}\, p_1., \ p_2 \leftarrow \text{not}\, q_2., \ q_2 \leftarrow \text{not}\, p_2.\}$$

By adding the constraint $\leftarrow p_1, p_2.$, any set that contains both p_1 and p_2 is not an answer set. Program Π thus has 4 answer sets, while $\Pi \cup \{\leftarrow p_1, p_2.\}$ has only 3.

8.2 Clingo framework

Clingo is an integrated ASP system, consisting of a grounder Gringo and solver Clasp [18]. In the following section I will show basics of the Clingo language, gringo as translation from Clingo to Aspif language and solving with clasp.

8.2.1 Clingo language

Clingo language [19] is used for transcribing ASP programs and their extended versions. There are 3 possible forms of rules as shown in the table bellow.

Type	Form	ASP rule
Fact	head(r).	head$(r) \leftarrow$.
Rule	head(r) :- body(r).	head$(r) \leftarrow$ body(r).
Constraint	:- body(r).	$\leftarrow$ body(r).

Semantically these rules mean to use head of rule if whole body of rule is consistent with the answer set. Fact does have an empty body, thus its head is used always. No answer set is consistent with body of constraint.

Example

Program (2.6) can be rewritten into Clingo language as following program.

```
p :- p.
q :- not p.
```

Each rule consists of head and body, separated by `:-` operator. Every atom starts with small letter of english alphabet. Every statements ends with a dot.

Clingo language also allows for the use of parametric atoms called functions[1]. Such atoms take form of `name(parameters)`, the name starts with a small letter of english alphabet, parameters can be of multiple types including integers, variables and arithmetic expressions (for comprehensive list see the Potassco guide [19]). Multiple parameters may be present, in that case they are separated by comma (`,`). Variable always starts with a big letter of english alphabet. Take the following program as an example.

```
1 a(1). a(2). a(3). b(2).
2 b(X+3) :- a(X), b(X).
```

This program contains four facts on line 1 and single substituable rule with arithmetic expression on line 2. As can be seen on line 1, multiple statements can share single line as long as they are all formed correctly. In the same manner, a statement can span over multiple lines.

If the exact parameter is not needed, one can use a wildcard (`_`) in place of the parameter in the body of rule. The wildcard behaves as anonymous variable meaning it is assigned a fresh parameter in the process of grounding.

```
a(1, 0). a(2, 4). a(3, -3).
b(X) :- a(X, _).
```

In the process of grounding parametric atoms, Gringo iteratively searches through the space of possible atoms of the same name and parameter count. If it finds suitable combination of atoms, it adds a new atom with the corresponding parameters. For instance, in the previous example atoms `b(1)`, `b(2)`, resp. `b(3)` would be added for atoms `a(1, 0)`, `a(2, 4)`, `a(3, -3)`.

[1] In the case of Clingo language functions are the ones in the matematic sense. Here, the function is a mere finitary relation.

Arithmetic functions and comparison predicates

As was already shown in one of the previous examples, Clingo language allows for the usage of some arithmetic functions for working with parameters. The following symbols are used for these functions: + (addition), - (subtraction, unary minus), * (multiplication), / (integer division), \ (modulo), ** (exponentiation), || (absolute value), & (bitwise AND), ? (bitwise OR), ^ (bitwise exclusive OR), and ~ (bitwise complement). The built-in predicates to compare terms are = (equal), != (not equal), < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). [19].

Intervals, pooling, wild cards

Another usefull constructs in Clingo language are intervals and pooling. Consider the following example of intervals (programs are equivalent):

```
a(1..3, 0..1).
```

```
a(1, 0). a(2, 0). a(3, 0).
a(1, 1). a(2, 1). a(3, 1).
```

Each combination of intervals is evaluated as a single rule. The first parameter, `1..3`, gives 3 choices, the second, `0..1`, 2 choices, thus $6 = 3 * 2$ rules can be derived.

For a more complex example consider the following program.

```
comp(X*Y) :- X = 2..20, Y = 2..20, X*Y <= 20.
prime(X) :- not comp(X), X = 2..20.
```

This program calculates prime numbers. A number is composite if it can be written as a product of two numbers greater than 1. Else it is prime.

Pooling is very similiar to intervals. It also allows for compact writing of rules. Similiar to intervals allowing for writing sets of consecutive numbers, pooling allows for writing any set. Using pooling, the example from the start of this section could be rewritten as following program.

```
a((1;2;3), (0;1)).
```

The pooling (`;`) has lower precedence than the splitting of parameters (`,`). For this, each pool must be enclosed in parentheses, else this program would be equivalent to:

```
a(1). a(2). a(3, 0). a(1).
```

Pooling can be also done on nonnumeric parameters.

```
a(foo; bar).
```

8.2.2 Gringo

Gringo is a software that grounds program in Clingo language into the format Aspif that is readable in Clasp. Gringo first resolves every rule with variables into (possibly multiple) variable-free rules and then changes format of the program into Clasp-readable Aspif. Gringo thus can introduce new atoms that were not obvious from the Clingo program. Full specification of Aspif language is written in [18].

Aspif (ASP intermediate format) language consists of statements, each on its own line. First line of file is header of form

$$\texttt{asp}\ \ v_m\ \ v_n\ \ v_r\ \ t_1\ \ \dots\ \ t_k$$

where v_m, v_n and v_r are versions of major, minor and revision numbers respectively and each t_i is a tag. Then follow lines with statements translated from program. For this thesis only rule and show statements are relevant. Last line of Aspif format file is a single 0.

Rule statement

Rule statement in Aspif has form of

$$1\ \ H\ \ B$$

in which head H has form of

$$h\ \ m\ \ a_1\ \ \dots\ \ a_m$$

where $h \in \{0, 1\}$, $m \geq 0$, $\forall i \in \{1, \dots, m\} . a_i \in \mathbb{N}^+$. Parameter h determines wherther head of this rule is disjunction (0) or choice (1), m determines number of literals and a_i are positive literals.

Body B is called normal if it has form of

$$0 \;\; n \;\; l_1 \;\; \ldots \;\; l_n$$

The normal body literals are in conjunction. For the head of statement with this body to be used, all its literals have to be consistent with the answer set. Parameter $n \geq 0$ determines the length of rule body and each l_i is literal. If the literal is negative, inversed value of its index is used.

The other type of body B is called weight body. Its form is

$$1 \;\; b \;\; n \;\; l_1 \;\; w_1 \;\; \ldots \;\; l_n \;\; w_n$$

Parameter $b \geq 0$ determines lower bound, n the length of rule body, each l_i is literal and $w_i \geq 1$ its weight. If the literal is negative, inversed value of its index is used. In this case, the head of the rule is used if the sum of weights from literals consistent with the answer set is greater or equal to the lower bound.

Show statement

Show statement is for specification of output, they result from `#show` directive. Each show statement is of form

$$4 \;\; m \;\; s \;\; n \;\; l_1 \;\; \ldots \;\; l_n$$

where m is length of string s, s is string with name, n is number length of condition and l_i are literals. The show statement prints the string s if all literals l_i are cosistent with the answer set.

Example

Let's illustrate the gringo parsing on some programs[2]. First the program $\Pi = \{p \leftarrow \text{not}\, q., q \leftarrow \text{not}\, p.\}$[3].

[2] To prevent gringo from making optimalizations,I have run all examples in this section with option `--preserve-facts=all`

[3] In fact the Aspif file generated by Gringo would have lines 2 and 3 swapped. In this thesis I have made this swap to better illustrate the translation.

```
1 p :- not q.
2 q :- not p.
```

```
asp 1 0 0            1
1 0 1 2 0 1 -1       2
1 0 1 1 0 1 -2       3
4 1 q 1 1            4
4 1 p 1 2            5
0                    6
```

First line of Aspif file is headder. Lines 1 and 2 of Clingo file are equivalent to lines 2 and 3 of Aspif file. There you can see transcription of negative literals `not q` and `not p` as -1 and -2 respectively. Let's now decompose the line 2 of Aspif program.

$$\begin{array}{ccccccc} 1 & 0 & 1 & 2 & 0 & 1 & -1 \\ S & h & m & a_1 & f & n & l_1 \end{array}$$

Begining with $S = 1$, this statement is a rule. Head of this rule is composed of $h = 0$, head of this rule is disjunction. As $m = 1$, if body is consistent with answer set, then the literal in the head has to be in the answer set. Finally, the literal on $a_1 = 2$ is alias for `p`. The body is then composed from $f = 0$, the literals are in conjunction. Then $n = 1$ means only one literal is in the body, that is $l_1 = -1$, meaning negative literal with alias 1 (`not q`). Line 3 is dual to this line.

Next, on lines 4 and 5, there are two statements for showing atoms.

$$\begin{array}{ccccc} 4 & 1 & \mathtt{q} & 1 & 1 \\ S & m & s & n & l_1 \end{array}$$

First is a show statement $(S = 4)$ for a single-lettered $(m = 1)$ atom `q`. It is printed out if single $(n = 1)$ literal numbered $l_1 = 1$ is in the answer set. Similiarly, the single-lettered atom `p` is printed out if single literal numbered 2 is in the answer set.

To illustrate the variable resolving, let me show you another Clingo program beeing parsed by gringo.

```
1 a(1).
2 a(2).
3 a(3).
4 b(2).
5 b(X+3) :- a(X), b(X).
```

```
asp 1 0 0                1
1 0 1 1 0 0              2
1 0 1 2 0 0              3
1 0 1 3 0 0              4
1 0 1 4 0 0              5
1 0 1 5 0 2 4 2          6
4 4 b(2) 1 4             7
4 4 b(5) 1 5             8
4 4 a(1) 1 1             9
4 4 a(2) 1 2             10
4 4 a(3) 1 3             11
0                        12
```

On lines 1–4 there are 4 facts `a(1)`, `a(2)`, `a(3)` and `b(2)`. Each of this fact instantiates a new atom with a single parameter. Then on line 5 is rule with variable X. To resolve this rule, gringo first looks through all known atoms of `a` and `b` and finds tuples that satisfy the positive literals of the body of the rule, that is functions `a` and `b` each with the same parameter. There is only a single such instantiation, that is `a(2)` and `b(2)`. Thus new atom `b(5)` is added to known atoms. This resolution of parametric rules continues until there is no rule that could add any new atoms.

Now, when we know all possibly needed atoms, Clingo adds rules to the program. The 4 facts from lines 1–4 of Clingo ended up as rules on lines 2–5 of Aspif. As can be seen, each fact got rewritten as rule with no parameter, thus applying every time. Rule from line 5 of Clingo got transcribed as a single rule on line 6 in Aspif - there is only a single possible assignment of atoms that fit the rule. Literal 5, corresponding to atom `b(5)`, is in the answer set if both literals 4 and 2 corresponding to `b(2)` and `a(2)` resp. are in the answer set. This rule can not be applied on any other tuple of literals, thus the grounding of rules is over. Further in Aspif file there are only show statements.

It should be noted that the parameters of atoms are only used in the grounding process. After grounding, the atoms lose their names and parameters and are for the solver represented by only assigned numbers.

One problem gringo has is that it is prone to generating endless number of atoms. Take the following program as an example.

```
a(1).
a(X+1) :- a(X).
```

```
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 1 1
1 0 1 3 0 1 2
1 0 1 4 0 1 3
1 0 1 5 0 1 4
...
```

As can be seen, in this case searching for atoms ends up with new atoms endlessly emerging. When writing Clingo programs, one has to be aware of this behaviour.

Another possibly unexcpected behaviour is that gringo works only on 32 or 64-bit integers. This is better shown on the following example.

```
a(1).
a(2*X) :- a(X).
```

```
asp 1 0 0                          1
1 0 1 1 0 0                        2
1 0 1 2 0 1 1                      3
...
1 0 1 33 0 1 32                    34
4 4 a(1) 1 1                       35
4 4 a(2) 1 2                       36
...
4 12 a(536870912) 1 30             64
4 13 a(1073741824) 1 31            65
4 14 a(-2147483648) 1 32           66
4 4 a(0) 1 33                      67
0                                  68
```

As seen on lines 65–67, the parameter of a has overflown to negative values and then to zero.

8.2.3 Extensions in Clingo language

Further I will show some of extensions of Clingo language usefull either in general or directly for the implementation part of this thesis.

Disjunctive logic programs

Disjunctive logic programs allow of use of disjunction in the rule head.

Type	Form
Fact	$A_0; \dots ; A_m.$
Rule	$A_0; \dots ; A_m$:- $L_0, \dots , L_n.$

Semantically the rule means that if body holds, than at least one of $A_0, \dots , A_m$ holds. Additionaly, set of atoms devised by this rule has to be minimal. This minimality criterion distinguish the disjunctive logic programs from the cardinality constraints. The disjunctive logic programs are not commonly used as they may increase the computational complexity [31].

```
{q}.
a; b; c :- q.
```

```
asp 1 0 0
1 1 1 1 0 0
1 0 3 2 3 4 0 1 1
4 1 q 1 1
4 1 a 1 2
4 1 b 1 3
4 1 c 1 4
0
```

Transcription from Clingo to Aspif is straightforward. Each disjunctive head is transcribed as a single disjunctive rule.

Cardinality constraints

$$\texttt{l}\ \{p_1; \dots ; p_m; \text{not}\, p_{m+1}; \dots ; \text{not}\, p_n\}\ \texttt{u}$$

$$\texttt{l <=}\ \{p_1; \dots ; p_m; \text{not}\, p_{m+1}; \dots ; \text{not}\, p_n\}\ \texttt{<= u}$$

Cardinality constraint's semantics is that at least l and at most u of literals $p_1; \dots ; p_m; \text{not}\, p_{m+1}; \dots ; \text{not}\, p_n$ has to be in the answer set for this literal to hold. The literals are percieved as set, thus multiple occurences of the same literal count only as one. Cardinality constraints could be encoded into ASP using oriented binary decision diagram. However, probably due to the complexity of this transformation size, Gringo encodes this conditional literal as (possibly) multiple rules with weight bodies [18].

The cardinality constraint can also be used without either one or both of l or u. That way it is unbounded from bottom or top respectively. The unbounded cardinality constraint is especially usefull for definition of the input space.

When transcripting from Clingo language to Aspif, each cardinality constraint rule translates to up to two weighted body rules and up two normal rules. When used in head, additional choice rule is used.

```
1 {a; b; c}.
2 q :- 1 {a; b; c} 2.
```

```
asp 1 0 0                      1
1 1 3 1 2 3 0 0                2
1 0 1 4 1 1 3 1 1 2 1 3 1      3
1 0 1 5 1 3 3 1 1 2 1 3 1      4
1 0 1 6 0 2 4 -5               5
1 0 1 7 0 1 6                  6
4 1 a 1 1                      7
4 1 b 1 2                      8
4 1 c 1 3                      9
4 1 q 1 7                      10
0                              11
```

On line 1 in Clingo and 2 in Aspif, solver can choose any number of atoms `a`, `b`, `c`. Then rule on line 2 in Clingo translates into lines 3–6 in Aspif. Rules on lines 3, 4 create new literals 4, 5. These are set if lower bound is filled (sum is at least 1) or upper bound overshot (sum is more than 2, that is at least 3). Then on line 5 a new literal 6 is set if 4 is set and 5 is not set. This corresponds to filling the lower bound and not overshooting upper bound simultaneously. Rule on line 6 is then used to set literal corresponding to `q`. In this case the rule is redundant, but generally this rule would implement the conjunction of body literals in the Clingo rule.

Use of the cardinality constraint in the head is very similiar.

```
1 {q}.
2 1 {a; b; c} 2 :- q.
```

```
asp 1 0 0                        1
1 1 1 1 0 0                      2
1 0 1 2 0 1 1                    3
1 1 3 3 4 5 0 1 2                4
1 0 1 6 1 1 3 3 1 4 1 5 1        5
1 0 1 7 1 3 3 3 1 4 1 5 1        6
1 0 1 8 0 2 6 -7                 7
1 0 0 0 2 2 -8                   8
4 1 q 1 1                        9
4 1 a 1 5                        10
4 1 b 1 4                        11
4 1 c 1 3                        12
0                                13
```

The main difference is in Aspif file, line 8. Instead of positive building literals as in previous example on line 6, a constraint is used. Thus either the rule is not used resulting in nonexistence of literal 2 or the literal 8 is set, corresponding to the cardinality constraint holding. Another difference is on the line 4. In the example with cardinality constraint in the body, the choice rule was added by another statement, while here it is part of transcribtion of this rule.

$$\mathtt{l} \diamond_1 \{p_1; \dots; p_m; \text{not } p_{m+1}; \dots; \text{not } p_n\} \diamond_2 \mathtt{u}$$

Similiarly to usage of operator <= in the statement, operators <, >=, >, =, != can be used in its place. In case of = or !=, the formula is rewritten using two <= operators.

In the body of statement, the operator = can be also used to assign value to variable. In that case, for each possible value, a new literal is created which is in the answer set if the equality holds. Thus the following two Clingo programs are parsed into the same Aspif file.

File	Best of 3	Average of 3
With redundancy	0.933 s	0.970 s
Without redundancy	3.278 s	3.287 s

Table 8.1: Comparison between redundant and nonredundant grounding

```
{a; b}.
q(X) :- X = {a; b}.
```

```
{a; b}.
q(2) :- 2 = {a; b}.
q(1) :- 1 = {a; b}.
q(0) :- 0 = {a; b}.
```

```
asp 1 0 0
1 1 2 1 2 0 0
1 0 1 3 1 1 2 1 1 2 1
1 0 1 4 0 1 -3
1 0 1 5 1 1 2 1 1 2 1
1 0 1 6 1 2 2 1 1 2 1
1 0 1 7 0 2 5 -6
1 0 1 8 0 1 7
1 0 1 9 1 2 2 1 1 2 1
1 0 1 10 0 1 9
4 1 a 1 1
4 1 b 1 2
4 4 q(0) 1 4
4 4 q(1) 1 8
4 4 q(2) 1 10
0
```

It can be seen that the generated program is highly redundant. For instance pairs of literals 3 and 5, 6 and 9, 7 and 8, 9 and 10 are the same. This might be on purpose as it somehow lowers the computational time of Clasp. I have measured the time to go through all solutions of both file generated by the gringo and equivalent file without redundancy. The files used modified program equivalent to the one above with 20 literals. The results are written in Table 8.1.

Conditional literals

$$L : L_1, \ldots, L_n$$

Conditional literals allow for a more compact writing of conditions. Semantically, the notation represents either literal L if the condition $L_1, \ldots, L_n$ is consistent with the answer set, In the case of inconsistency with the condition the conditional literal appear to not be present. In rule bodies, it is equivalent to a condition L is consistent with the answer set or conjunction of $L_1, \ldots, L_n$ is not consistent with the answer set.

Conditional literals are especially usefull when used inside of aggregates as their use help significantly to compact the programs.

Aggregates

$$\texttt{l} \diamond_1 \texttt{\#agg}\{l_1 : L_1; \ldots; l_n : L_n\} \diamond_2 \texttt{u}$$

Similiar to cardinality constraints are the aggregates. These work as functions applied on sets of tuples. Usable aggregate functions are `#count`, `#sum`, `#sum+`, `#min` and `#max`. As in cardinality constraints, both $\diamond_1, \diamond_2$ default to <= if ommited.

Usual use of aggregates is by substituting some number parameter for l_i based on the parametric literal L_i. This way the aggregate works as function over all instances of such literal. For example the following program

```
{a(1); a(2); a(5); a(10)}.
s(S) :- S = #sum{ X : a(X) }.
```

allows for assigning to parameter of atom `s` only values which can be aquired as a sum of some numbers from list 1, 2, 5 and 10.

The `#count`, `#sum` and `#sum+` aggregates are rewritten simmiliarly to the cardinality constraint as up to two weighted body rules with the values l_i as weights. The `#min` and `#max` aggregates are rewritten by sorting the possible values. The aggregate then returns the value iff some instance of this value is present and no instance of bigger value is present at the same time resulting in many statements in the Aspif file.

8.2.4 Clasp

Clasp is the solver of the Clingo framework. It takes Aspif file on input to search for answer sets of specified problem. This solver approaches the inference using the unit propagation of nogoods [32]. It is also able to search for all answer sets in one pass making it good option for the task of verification.

9 Binary neural network

Notation

Let me first introduce you to the notation for this chapter.

Symbol	Definition
$\mathbb{N}_0$	Set of nonnegative integers, that is $\{0, 1, 2, \dots\}$
$\mathbb{B}$	$\{0, 1\}$
$\mathbb{B}_\pm$	$\{-1, 1\}$
$\mathbb{R}$	Real numbes
S^k	Set of vectors $\{(s_1, s_2, \dots, s_k) \mid s_1, s_2, \dots, s_k \in S\}$
$\mathcal{N}^{\mathbb{B}}$	Binary network as a function
$\mathbf{M}$	Matrix
$\vec{v}$	Vector
$\vec{1}$	Vector of 1's
$[k]$	Set of numbers up to k, that is $\{1, 2, \dots, k\}$
$\mathbf{M}_{:,j}$	j-th row of matrix $\mathbf{M}$
$\vec{v}_j$	j-th entry of vector $\vec{v}$
$\langle \vec{x}, \vec{y} \rangle$	Scalar product $\langle \vec{x}, \vec{y} \rangle = \sum_i \vec{x}_i + \vec{y}_i$
$\mathrm{sign}_{\pm 1}(x)$	Sign function using values +1, -1 $\mathrm{sign}_{\pm 1}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$
$\lfloor x \rfloor$	Bottom whole part $\lfloor x \rfloor = y \iff x = y + q \wedge 0 \leq q < 1$

Table 9.1: Used notation

9.1 Definition of BNN

I have been working with deterministic binarized neural networks (BNN) as defined in [3] and [6]. For convinience I have rewritten the output as a natural number instead of one-hot vector. This article defines deterministic BNN as a convolution of layers. The input is encoded as vector $\mathbb{B}_{\pm 1}^{n_1}$. The BNN is then encoded as tuple of blocks $(t_1, t_2, \ldots, t_d, t_{d+1})$.

$$\mathcal{N}^{\mathbb{B}} : \mathbb{B}_{\pm 1}^{n_1} \to \mathbb{N}_0$$

$$\mathcal{N}^{\mathbb{B}} = t_{d+1} \circ t_d \circ \cdots \circ t_1, \tag{9.1}$$

where for each $i \in \{1, 2, \ldots, d\}$, t_i is inner block consisting of LIN layer t_i^{lin}, BN layer t_i^{bn} and BIN layer t_i^{bin}.

$$t_i : \mathbb{B}_{\pm 1}^{n_i} \to \mathbb{B}_{\pm 1}^{n_{i+1}}$$

$$t_i = t_i^{bin} \circ t_i^{bn} \circ t_i^{lin} \tag{9.2}$$

Output block t_{d+1} then consists of LIN layer t_{d+1}^{lin} and ARGMAX layer t_{d+1}^{am}.

$$t_{d+1} : \mathbb{B}_{\pm 1}^{n_{d+1}} \to \mathbb{N}_0$$

$$t_{d+1} = t_{d+1}^{am} \circ t_{d+1}^{lin} \tag{9.3}$$

Each layer is a function with parameters as defined in Table 9.2.

Layer	Function	Parameters	Definition
LIN	$t_i^{lin} : \mathbb{B}_{\pm 1}^{n_i} \to \mathbb{R}^{n_{i+1}}$	Weight matrix: $\mathbf{W} \in \mathbb{B}_{\pm 1}^{n_i \times n_{i+1}}$ Bias (row) vector: $\vec{b} \in \mathbb{R}^{n_{i+1}}$	$t_i^{bn}(\vec{x}) = \vec{y}$, where $\forall j \in [n_{i+1}]$ $\vec{y}_j = \langle \vec{x}, \mathbf{W}_{:,j} \rangle + \vec{b}_j$
BN	$t_i^{bn} : \mathbb{R}^{n_{i+1}} \to \mathbb{R}^{n_{i+1}}$	Weight vector: $\vec{\alpha} \in \mathbb{R}^{n_{i+1}}$ Bias vector: $\vec{\gamma} \in \mathbb{R}^{n_{i+1}}$ Mean vector: $\vec{\mu} \in \mathbb{R}^{n_{i+1}}$ Std. dev. vector: $\vec{\sigma} \in \mathbb{R}^{n_{i+1}}$	$t_i^{bn}(\vec{x}) = \vec{y}$, where $\forall j \in [n_{i+1}]$ $\vec{y}_j = \vec{\alpha}_j \cdot \frac{\vec{x}_j - \vec{\mu}_j}{\vec{\sigma}_j} + \vec{\gamma}_j$
BIN	$t_i^{bin} : \mathbb{R}^{n_{i+1}} \to \mathbb{B}_{\pm 1}^{n_{i+1}}$	-	$t_i^{bin}(\vec{x}) = \vec{y}$, where $\forall j \in [n_{i+1}]$, $\vec{y}_j = \begin{cases} +1, & \text{if } \vec{x}_j \geq 0; \\ -1, & \text{otherwise} \end{cases}$
ARGMAX	$t_{d+1}^{am} : \mathbb{R}^{n_{d+1}} \to \mathbb{N}_0$	-	$t_{d+1}^{am}(\vec{x}) = \arg\max(\vec{x})$

Table 9.2: Definition of BNN layers [6]

9.2 Transformation of BNN

In this section, I will show a possible trasformation of parameters to lower their count and make them integer only. This transformation is similiar to the one introduced in [6]. The transformation to integer-only parameters is crucial as Clingo can not compute on real numbers.

$$t_i(\vec{x}) = (t_i^{bin} \circ t_i^{bn} \circ t_i^{lin})(\vec{x}) = y$$

$$y_j = \operatorname{sign}_{\pm 1}\left(\alpha_j \cdot \frac{\langle \vec{x}, \mathbf{W}_{:,j} \rangle + b_j - \vec{\mu}_j}{\vec{\sigma}_j} + \vec{\gamma}_j\right)$$

The argument of $\operatorname{sign}_{\pm 1}$ can be further analysed.

$$\vec{\alpha}_j \cdot \frac{\langle \vec{x}, \mathbf{W}_{:,j} \rangle + \vec{b}_j - \vec{\mu}_j}{\vec{\sigma}_j} + \vec{\gamma}_j \geq 0$$

There are three possible cases of value $\frac{\vec{\alpha}_j}{\vec{\sigma}_j}$:

$$\frac{\vec{\alpha}_j}{\vec{\sigma}_j} > 0:$$
$$\langle \vec{x}, \mathbf{W}_{:,j} \rangle + \vec{b}_j - \vec{\mu}_j + \frac{\vec{\sigma}_j}{\vec{\alpha}_j} \cdot \vec{\gamma}_j \geq 0$$
$$\mathbf{W}'_{:,j} = \mathbf{W}_{:,j},\ \vec{b}'_j = \vec{b}_j - \vec{\mu}_j + \frac{\vec{\sigma}_j}{\vec{\alpha}_j} \cdot \vec{\gamma}_j$$
$$\langle \vec{x}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0$$

$$\frac{\vec{\alpha}_j}{\vec{\sigma}_j} < 0:$$
$$\langle \vec{x}, \mathbf{W}_{:,j} \rangle + \vec{b}_j - \vec{\mu}_j + \frac{\vec{\sigma}_j}{\vec{\alpha}_j} \cdot \vec{\gamma}_j \leq 0$$
$$\langle \vec{x}, -\mathbf{W}_{:,j} \rangle - \vec{b}_j + \vec{\mu}_j - \frac{\vec{\sigma}_j}{\vec{\alpha}_j} \cdot \vec{\gamma}_j \geq 0$$
$$\mathbf{W}'_{:,j} = -\mathbf{W}_{:,j},\ \vec{b}'_j = -\vec{b}_j + \vec{\mu}_j - \frac{\vec{\sigma}_j}{\vec{\alpha}_j} \cdot \vec{\gamma}_j$$
$$\langle \vec{x}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0$$

$$\vec{\alpha}_j = 0:$$
$$\vec{\gamma}_j \geq 0$$
$$\mathbf{W}'_{:,j} = \vec{0},\ \vec{b}'_j = \vec{\gamma}_j$$
$$\langle \vec{x}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0$$

This shows that each inner layer can be computed using weight matrix and one bias parameter. However if there were some parameter $\alpha_j = 0$, $\mathbf{W}'$ would not have values only from ±1 but from $\{-1, 0, 1\}$. This is not an issue as Clingo works by default on integers. If 0-values were issue, they could be removed either by offsetting bias parameter or by modifying the BNN structure to remove the constant node. (This has not been implemented in my work.)

$$\vec{\alpha}_j = 0 : \; W'_{:,j} = W_{:,j}, \; \vec{b}'_j = \begin{cases} \langle \vec{1}, \vec{1} \rangle + 1 & \text{if } \vec{\gamma}_j \geq 0; \\ -\langle \vec{1}, \vec{1} \rangle - 1 & \text{otherwise} \end{cases}$$

In my implementation I am encoding inputs as binary values $\{0, 1\}$. This helps to speed up the computation in Clingo as it removes the need for multiplication inside of `#sum` aggregates.

$$\begin{gathered} \vec{x} = 2\vec{x}^{(b)} - \vec{1} \\ \langle \vec{x}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0 \\ \langle 2\vec{x}^{(b)} - \vec{1}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0 \\ 2\langle \vec{x}^{(b)}, \mathbf{W}'_{:,j} \rangle - \langle \vec{1}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j \geq 0 \\ \langle \vec{x}^{(b)}, \mathbf{W}'_{:,j} \rangle + \frac{-\langle \vec{1}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j}{2} \geq 0 \end{gathered}$$

Still each block could be computed using one weight matrix and one bias vector. Additionally, as $\langle \vec{x}^{(b)}, \mathbf{W}'_{:,j} \rangle$ is an integer value, equation is equivalent to

$$\langle \vec{x}^{(b)}, \mathbf{W}'_{:,j} \rangle + \left\lfloor \frac{-\langle \vec{1}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j}{2} \right\rfloor \geq 0 \tag{9.4}$$

As for the output block, it has already just one weight matrix and one bias vector. The implementation of ARGMAX layer is mainly Clingo-based. The only needed transformation is to split bias to whole and decimal part.

$$\vec{b} = \vec{b}' + \vec{p}, \; \vec{b}'_j = \lfloor \vec{b}_j \rfloor$$

$\vec{p}$ is then used as main order of outputs.

Implementation of these transformations in Python is added as [appendix]

9.3 Encoding of BNN in Clingo

9.3.1 Base encoding

Input for the Clingo was parsed from the model using Python program. This program also computed the transformation from Section 9.1. This program then outputs the model of BNN as multiple rules. Semantics are as defined in Table 9.3.

Fact	Param.	Semantics
`layer(`L, N`).`	L	layer number
		$L = 0$ corresponds to input layer
		highest layer corresponds to output layer
	N	number of nodes in layer L
`weight(`L, P, N, W`).`	L	layer
	P	node in layer $L - 1$
	N	node in layer L
	W	weight of node P to node N
`bias(`L, N, B`).`	L	layer
	N	node in layer L
	B	bias for node N in layer L
`outpre(`N, P`).`	N	output node
	P	N has precedence P
		if two outputs are tied, highest P is used

Table 9.3: Semantics of encoding BNN into Clingo-readable file

With this notation of literals, I have transcribed the computation of BNN as the following Clingo program.

```
% output layer
output_layer(L) :- L = #max{ X : layer(X, _) }.

% any combination of input bits is possible
{on(0, 0..K-1)} :- layer(0, K).

% hidden layers
on(L, N) :-
    #sum{ W,I : on(L-1, I), weight(L, I, N, W) } >= B,
    bias(L, N, B), not output_layer(L).
```

```
 9 % output layer before arg max layer
10 outnode(N, S + B) :-
11     S = #sum{ W,I : on(L-1, I), weight(L, I, N, W) },
12     output_layer(L), bias(L, N, B).

13 % output the node with highest sum
14 % if tied on sum, then with highest precedence
15 % if tied on both sum and precedence, then with lowest
     number
16 output(Node) :-
17     (Sum, Order, -Node) = #max{ (S, O, -N) : outnode(N, S
     ), outpre(N, O) }.
```

On line 2, it computes the output layer. Line 4 allows for arbitrary assignment of input vector. Then next layers are computed on lines 6–8. This computation follows the Equation 9.4.

Computation of the output is performed in two steps. First, on lines 10–12, sum of weighted inputs and bias is computed into the `outnode`, then on lines 16–17, the maximal of these values is choosen to be the output.

Grounding of most of this program is straightforward. Line 2 expands to a single fact. Line 4 expands to number of choice rules equal to size of input vector. Lines 6–8 expand to a number of rules equal to hidden nodes of the network.

9.3.2 Improved encoding of the output layer

Encoding of the output layer as shown above is problematic. Rule on lines 10–12 generate for each of the output bits one more atom than is the size of last hidden layer, each with equality constraint. Then the rule on lines 16–17 generate exponentaly many rules for `output`.

To mitigate this issue I have created an alternative way for computation of the output. I have used the choice rule while constraining the output to be the maximal. This way I have removed the need to use an intermediate step of `outnode`. The following lines replace lines 9–17.

```
 9 % there is single output
10 1 {output(0..N-1)} 1 :- layer(L, N), output_layer(L).

11 % the sum of another is not higher
12 :- output(N), O = 1..M-1, output_layer(L), layer(L, M), O
      != N,
13     #sum{ W, I, this  : on(L-1, I), weight(L, I, N, W);
14          -W, I, other : on(L-1, I), weight(L, I, O, W) }
     < BO - BN,
15     bias(L, N, BN), bias(L, O, BO).

16 % the sum of another is not same or order is not higher
17 :- output(N), O = 1..M-1, output_layer(L), layer(L, M), O
      != N,
18     #sum{ W, I, this  : on(L-1, I), weight(L, I, N, W);
19          -W, I, other : on(L-1, I), weight(L, I, O, W) }
     = BO - BN,
20     bias(L, N, BN), bias(L, O, BO),
21     outpre(N, PN), outpre(O, PO), PO > PN.
```

The grounding generates only a single atom for each output and then for each of them a finitely many constraints. Line 10 asserts there is exactly single output. Then the two constraints assure that there is no other output that would have higher biased sum of previous layer and if equal, the one with higher precedence is used. There is no need for the evaluation of output number as the precedence already is nonredundant.

The use of choice rule and constraints has greatly reduced both size of the Aspif file generated by gringo and the total time needed for computation of answer sets as shown in the table below.

9.4 Encoding of input regions

An input region of BNN is some subset of inputs on which the analysis is performed. I have implemented two input region types from [6], that is input region based on Hamming distance (R_H) and input region with fixed indices (R_I).

The quantitative analysis of said BNN is a problem to say how many inputs in the input region differ from ground truth (or in this case the base input) in the outputs. Given BNN $\mathcal{N}^{\mathbb{B}}$, an input region R and base input vector $\hat{v}$, the number of missclassified inputs is $\|\{v \mid v \in R, \mathcal{N}^{\mathbb{B}}(v) \neq \mathcal{N}^{\mathbb{B}}(\hat{v})\}\|$.

To perform the quantitative analysis on some input region, the encoding of such region and of a missclassified (adversarial) input is needed.

9.4.1 Hamming distance

$$R_H(\vec{u}, r) = \{\vec{v} \mid \vec{u}, \vec{v} \in \mathbb{B}^n, \|\{i \mid \vec{u}_i \neq \vec{v}_i\}\| \leq r\}$$

Space of inputs under hamming distance $r \in \mathbb{N}_0$ from base vector $\vec{u} \in \mathbb{B}^n$, $R_H(\vec{u}, r)$ is a set of input vectors, that differs on at most r entries from the base vector $\vec{u}$.

In my implementation, full base vector is parsed by the Python parser. If the entry of vector represents active state, fact `input(`a`).`, where a corresponds to the index of that entry, is included in the model. Else nothing is added to the model. The maximal allowed hamming distance is represented as fact `hamdist(`r`).`, where r corresponds to the allowed radius of this distance.

```
% base vector is at most hamdist from on
:- #count{ N : not input(N), on(0, N);
           N : input(N), not on(0, N) } > H,
    hamdist(H).
```

The input region is then encoded in the Clingo language as a single constraint.

9.4.2 Fixed indices

$$R_I(\vec{u}, I) = \{\vec{v} \in \mathbb{B}^n \mid \forall i \in I \,.\, \vec{u}_i = \vec{v}_i\}$$

Input region $R_I(\vec{u}, I)$ given by fixed indices is a set of input vectors that do not differ from the base vector $\vec{u} \in \mathbb{B}^{n_1}$ on entries on indices from $I \subseteq [n_1]$.

Like in the input region based of hamming distance, the full base vector $\vec{u}$ is added to the model using facts `input(`a`).`. Then the fixed indices in the form of fact `inpfix(`i`).` for every index $i \in I$ are added. Note that this defintion is dual to the one in [6] where I corresponds to the free indices.

```
% input does not differ from base on fixed indices
:- inpfix(N), on(0, N), not input(N).
:- inpfix(N), not on(0, N), input(N).
```

The encoding into Clingo is again simple. If the index is fixed, then the input `on(0, N)` and base input `input(N)` can not differ.

9.4.3 Encoding of adversarial output

For the output to differ from the one of base input, the program has to be further constrained. To implement the rule for the output to be adversarial, I have used the same encoding as for the computing of the output of the network.

I have also tried encoding the rule by constraining the output to not be pre-computed value given by evaluating the base input outside the Clingo framework but this did not yield better results.

```
% Show only inputs with output nonequal to that of input vector
inputOn(0, N) :- input(N).
inputOn(L, N) :-
    #sum{ W,I : inputOn(L-1, I), weight(L, I, N, W) } >= B,
    bias(L, N, B), not output_layer(L).

1 {inputOutput(0..N-1)} 1 :- layer(L, N), output_layer(L).

% the sum of another is not higher
:- inputOutput(N), O = 1..M-1, output_layer(L), layer(L, M), O != N,
    #sum{ W, I, this  : inputOn(L-1, I), weight(L, I, N, W);
          -W, I, other : inputOn(L-1, I), weight(L, I, O, W) } < BO - BN,
    bias(L, N, BN), bias(L, O, BO).
% the sum of another is not same or order is not higher
:- inputOutput(N), O = 1..M-1, output_layer(L), layer(L, M), O != N,
    #sum{ W, I, this  : inputOn(L-1, I), weight(L, I, N, W);
          -W, I, other : inputOn(L-1, I), weight(L, I, O, W) } = BO - BN,
    bias(L, N, BN), bias(L, O, BO),
    outpre(N, PN), outpre(O, PO), PO > PN.

:- output(Node), inputOutput(Node).
```

10 Evaluation

10.1 Methodology of evaluation

For the evaluation of my model, I have used inputs from MNIST dataset [30], the same as were used for the evaluation of BNNQuanalyst [6]. In the next chapter I will be reffering to instances of inputs 0 to 9 as I0 to I9. The architectures of different models is written in Table 10.1. These trained models have been taken from [6].

The framework was evaluated using the computational sources of MetaVO, virtual organization providing computing resources for academic community. The framework was evaluated on physical machine kirke36.meta.zcu.cz. The framework was evaluated on 8 CPU cores with 16 GB of RAM.

10.2 Evaluation over different inputs

I have evaluated BNN models M1, M2, M6, M7 agnist all inputs I0-I9 with hamming distance 3. The results are shown in the Table 10.2. As can be seen, the time to find all adversary inputs is independent on the number of all adversary inputs. Time to find 1st model (adversary input) is dependent on the number of models (adversary inputs). The more models exist for the problem, the higher chance is to find one by assigning values for literals at random. The table also shows that the

Model	Architecture	Model	Architecture
M1	100:100:10	M7	100:50:20:10
M2	100:50:10	M8	16:25:20:10
M3	400:100:10	M9	36:15:10:10
M4	64:10:10	M10	16:64:32:20:10
M5	784:100:10	M11	25:25:25:20:10
M6	100:100:50:10	M12	784:50:50:50:50:10

Table 10.1: Architectures of models

Values correspond to the sizes of layers size of Aspif intermediate file does not change much between different inputs. The quantitative verification using ASP seems to scale better with depth of the evaluated BNN than with width.

10.3 Evaluation over different hamming distance

I have evaluated models M2, M3, M7, M11 on input I0 for hamming distance from 0 to 4. The results are shown in the Table 10.3. As can be seen, the hamming distance does not affect the size of the Aspif intermediate file at all. This is because the hamming distance does only change weight in a single constraint corresponding to the encoding of input region by hamming distance. The time to find all models (adversarial inputs) scales badly with hamming distance for big input layers. This is expected as the input space of inputs under hamming distance grows for small distances close to exponentially with size of the input layer in the base. The effect of size of the base is apparent in the evaluation of model M11, here the time grows relatively slowly.

10.4 Evaluation over fixed input bits

Fixed bits vectors were created from the base input instances corresponding to the number 0 using the following bash script. The script creates for each input instance 4 fixed bits vectors by removing the first 0, 8, 16, resp. 24 positions from the list of fixed bits.

```
for l in 25 100 400; do
  for pos in $(eval echo {0..$(( $l-1 ))}); do
    echo -n "$pos "
  done | head --bytes=-1 "inpbits_0_${l}_0.txt"
  for free in 8 16 24; do
        cat "inpbits_0_${l}_$(( $free-8 )).txt" \
    | cut -d’ ’ -f’9-’ > "inpbits_0_${l}_${free}.txt"
  done
done
```

For created fixed bits vectors models M2, M3, M7, M11 were evaluated on instance I0. Results are included in the Table 10.4. This task seems to be better suited for my framework as the solver, Clasp, can prune large subspaces of the input space at once. In the model M3, where only a small subspace of the input space is adversarial for this

Input	$\#M$	T_M	T_1	N_r	N_w	$\#M$	T_M	T_1	N_r	N_l
			M1					M2		
I0	40821	30.419	0.15	11151	44386	16403	19.114	0.09	5601	22236
I1	10945	33.523	0.18	11138	44373	34288	29.625	0.04	5588	22223
I2	22518	31.482	0.13	11154	44389	26628	14.982	0.06	5604	22239
I3	20666	32.190	0.16	11152	44387	10955	15.271	0.08	5602	22237
I4	114392	25.190	0.09	11149	44384	70244	13.806	0.06	5599	22234
I5	36405	31.883	0.14	11147	44382	140021	13.695	0.04	5597	22232
I6	0	27.498	–	11153	44388	487	16.880	0.35	5603	22238
I7	53242	28.737	0.13	11144	44379	15055	16.309	0.08	5594	22229
I8	139399	24.955	0.12	11151	44386	47558	14.072	0.08	5601	22236
I9	42591	27.802	0.14	11150	44385	74443	14.335	0.05	5600	22235
			M6					M7		
I0	51423	68.897	0.96	15702	62538	3330	25.382	0.16	6322	25098
I1	196	82.353	2.17	15689	62525	35548	21.913	0.07	6309	25085
I2	45315	48.831	0.66	15705	62541	20752	37.921	0.11	6325	25101
I3	6725	71.345	0.61	15703	62539	8238	26.670	0.10	6323	25099
I4	24909	87.996	1.17	15700	62536	74224	21.288	0.16	6320	25096
I5	22733	59.329	0.86	15698	62634	11053	21.608	0.07	6318	25094
I6	6283	77.874	0.70	15704	62540	15919	26.313	0.15	6324	25100
I7	7091	76.299	0.80	15695	62531	8361	23.145	0.29	6315	25091
I8	49700	129.550	0.85	15702	62538	146724	15.858	0.05	6322	25098
I9	80704	50.981	0.82	15701	62537	24300	20.125	0.15	6321	25097

Table 10.2: Evaluation of M1, M2, M6, M7 over inputs I0 — I9

$\#M$ number of adv. inputs, T_M time to all adv. inputs [sec], T_1 time to 1st adv. input [sec]
N_r number of lines (approx. rules) of Aspif, N_l number of words (approx. literals) of Aspif

Distance	$\#M$	T_M	T_1	N_r	N_w	$\#M$	T_M	T_1	N_r	N_l
	M2					M3				
0	0	0.078	–	5601	22236	0	0.494	–	41205	164440
1	0	0.177	–	5601	22236	0	2.316	–	41205	164440
2	91	0.775	0.13	5601	22236	0	156.865	–	41205	164440
3	16404	19.316	0.16	5601	22236	$[TO]$ 0	$[TO]$ 300.006	–	41205	164440
4	$[TO]$ 447801	$[TO]$ 300.055	0.05	5601	22236	$[TO]$ 0	$[TO]$ 300.025	–	41205	164440
	M7					M11				
0	0	0.103	–	6322	25098	0	0.056	–	2052	8079
1	0	0.295	–	6322	25098	12	0.141	0.06	2052	8079
2	73	1.471	0.14	6322	25098	207	0.188	0.07	2052	8079
3	3330	31.686	0.16	6322	25098	1813	0.398	0.06	2052	8079
4	$[TO]$ 167451	$[TO]$ 300.008	0.13	6322	25098	11830	0.818	0.07	2052	8079

Table 10.3: Evaluation of M2, M3, M7, M11 over hamming distance 0 — 4

$\#M$ number of adv. inputs, T_M time to all adv. inputs [sec], T_1 time to 1st adv. input [sec],
N_l number of lines (approx. rules) of Aspif, N_w number of words (approx. literals) of Aspif
$[TO]$ solving has timed out

input, the framework is significantly faster than on the other models. On the others — M2, M7 and M11 — Clasp probably spends most of the computation time on the enumeration of models (adversarial inputs). As far as I am aware, Clasp is unable to count models any other way. If it could count them in batches like it can prune sets, where it knows no models are present, the evaluation of models M2, M7 and M11 might be speeded up significantly.

Free	$\#M$	T_M	T_1	N_r	N_w	$\#M$	T_M	T_1	N_r	N_l
	M2					M3				
0	0	0.076	–	5700	22335	0	0.443	–	41604	164839
8	215	0.099	0.01	5692	22327	0	0.455	–	41596	164831
16	50517	0.779	0.01	5684	22319	0	0.493	–	41588	164823
24	12467600	130.907	0.02	5676	22311	2704	1.688	0.04	41580	164815
	M7					M11				
0	0	0.099	–	6421	25197	0	0.058	–	2076	8103
8	58	0.167	0.04	6413	25189	226	0.113	0.05	2068	8095
16	38187	1.327	0.05	6405	25181	60144	2.229	0.06	2060	8087
24	11819735	282.620	0.08	6397	25183	$[TO]$ 9343147	$[TO]$ 300.001	0.09	2052	8079

Table 10.4: Evaluation of M2, M3, M7, M11 over 0 — 24 free bit positions

$\#M$ number of adv. inputs, T_M time to all adv. inputs [sec], T_1 time to 1st adv. input [sec],
N_l number of lines (approx. rules) of Aspif, N_w number of words (approx. literals) of Aspif
$[TO]$ solving has timed out

11 Conclussion

In this thesis I have created and implemented framework for the quantitative analysis of binary neural networks using the answer set paradigm. My framework uses Python to parse the neural network into Clingo-readable format and then ASP-paradigm-based framework Clingo, combination of parser Gringo and solver Clasp, to find adversarial inputs. In this framework I have implemented functionality for representing input regions based on Hamming distance and input regions based on fixed indices.

I have evaluated the framework on binary neural networks trained on the MNIST database of handwritten digits. The framework scales very well when evaluating robust neural networks with input region constrained by the fixed indices. When many adversarial inputs are present, the limiting factor proved to be the Clasp as it relies on enumeration of the models (adversarial inputs). To eliminate this shortcomming, another solver will have to be used. When evaluating input regions based on hamming distance, the framework has shown better performance on networks with smaller input layer.

Answer set programming has the potential to make verification of neural networks both easier and faster. ASP paradigm has high expressive power while its solvers have strong heuristics to find a fast way for the computation.

Appendices

Code of BNN verificator

The full code of the verificator implemented in this thesis together with the examples can be found either in the Thesis archive in the IS MU or in the Github repository https://github.com/Ardnij123/BNN_verification

Bibliography

1. ZHANG, Yedi; ZHAO, Zhe; CHEN, Guangke; SONG, Fu; CHEN, Taolue. BDD4BNN: a BDD-based quantitative analysis framework for binarized neural networks. In: *International Conference on Computer Aided Verification*. Springer, 2021, pp. 175–200.
2. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; SCHAUB, Torsten. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*. 2019, vol. 19, no. 1, pp. 27–82.
3. HUBARA, Itay; COURBARIAUX, Matthieu; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Binarized Neural Networks. *ArXiv*. 2016, vol. abs/1602.02505.
4. BISHOP, Christopher M. Neural networks for pattern recognition. In: 1995. Available also from: `https://api.semanticscholar.org/CorpusID:60563397`.
5. HONG, Don; WANG, Jianzhong; GARDNER, Robert. *Real Analysis with an Introduction to Wavelets and Applications*. Burlington: Academic Press, 2005. ISBN 978-0-12-354861-0. Available from DOI: `https://doi.org/10.1016/B978-012354861-0/50003-8`.
6. ZHANG, Yedi; ZHAO, Zhe; CHEN, Guangke; SONG, Fu; CHEN, Taolue. Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach. *ACM Trans. Softw. Eng. Methodol.* 2023, vol. 32, no. 3. ISSN 1049-331X. Available from DOI: `10.1145/3563212`.
7. LIFSCHITZ, V. What is answer set programming? In: *Proc. 23rd AAAI Conf. on Artificial Intelligence, 2008*. 2008, pp. 1594–1597.
8. ANGER, Christian; KONCZAK, Kathrin; LINKE, Thomas; SCHAUB, Torsten. A Glimpse of Answer Set Programming. *Künstliche Intell.* 2005, vol. 19, no. 1, p. 12.
9. DELGRANDE, Jim. *Answer Set Programming* [online]. [N.d.]. [visited on 2024-04-27]. Available from: `https://www2.cs.sfu.ca/CourseCentral/721/jim/ASP1.Intro.pdf`.

10. GELFOND, Michael; LIFSCHITZ, Vladimir. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*. 1991, vol. 9, pp. 365–385. Available from DOI: `10.1007/BF03037169`.

11. HARMELEN, Frank; LIFSCHITZ, Vladimir; PORTER, Bruce. The Handbook of Knowledge Representation. *Elsevier Science San Diego, USA*. 2007, p. 1034.

12. ZARBA, Calogero G. *Many-Sorted Logic* [online]. 2006. [visited on 2024-10-26]. Available from: `https://web.archive.org/web/20070929131504/http://react.cs.uni-sb.de/~zarba/snow/ch01.pdf`.

13. EITER, Thomas; GOTTLOB, Georg; MANNILA, Heikki. Disjunctive datalog. *ACM Trans. Database Syst.* 1997, vol. 22, no. 3, pp. 364–418. ISSN 0362-5915. Available from DOI: `10.1145/261124.261126`.

14. SCHLIPF, J.S. The Expressive Powers of the Logic Programming Semantics. *Journal of Computer and System Sciences*. 1995, vol. 51, no. 1, pp. 64–86. ISSN 0022-0000. Available from DOI: `https://doi.org/10.1006/jcss.1995.1053`.

15. LIFSCHITZ, Vladimir. Twelve Definitions of a Stable Model. In: GARCIA DE LA BANDA, Maria; PONTELLI, Enrico (eds.). *Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 37–51. ISBN 978-3-540-89982-2.

16. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; SCHAUB, Torsten. Multi-shot ASP solving with clingo. *CoRR*. 2017, vol. abs/1705.09811.

17. GEBSER, MARTIN; HARRISON, AMELIA; KAMINSKI, ROLAND; LIFSCHITZ, VLADIMIR; SCHAUB, TORSTEN. Abstract gringo. *Theory and Practice of Logic Programming*. 2015, vol. 15, no. 4–5, pp. 449–463. ISSN 1475-3081. Available from DOI: `10.1017/s1471068415000150`.

18. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; OSTROWSKI, Max; SCHAUB, Torsten; WANKO, Philipp. *Theory Solving Made Easy with Clingo 5 (Extended Version)*. 2016.

19. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; LINDAUER, Marius; OSTROWSKI, Max; ROMERO, Javier; SCHAUB, Torsten; THIELE, S; WANKO, P. *Potassco guide version 2.2. 0*. 2019.

20. WEI, Jason; BOSMA, Maarten; ZHAO, Vincent Y.; GUU, Kelvin; YU, Adams Wei; LESTER, Brian; DU, Nan; DAI, Andrew M.; LE, Quoc V. Finetuned Language Models Are Zero-Shot Learners. *arXiv e-prints*. 2021, arXiv:2109.01652. Available from DOI: `10.48550/arXiv.2109.01652`.

21. CARLINI, Nicholas; WAGNER, David. Towards Evaluating the Robustness of Neural Networks. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 39–57. Available from DOI: `10.1109/SP.2017.49`.

22. CHEN, Guangke; CHEN, Sen; FAN, Lingling; DU, Xiaoning; ZHAO, Zhe; SONG, Fu; LIU, Yang. *Who is Real Bob? Adversarial Attacks on Speaker Recognition Systems*. 2020. Available from arXiv: `1911.01840 [eess.AS]`.

23. EYKHOLT, Kevin; EVTIMOV, Ivan; FERNANDES, Earlence; LI, Bo; RAHMATI, Amir; XIAO, Chaowei; PRAKASH, Atul; KOHNO, Tadayoshi; SONG, Dawn. Robust Physical-World Attacks on Deep Learning Visual Classification. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1625–1634. Available from DOI: `10.1109/CVPR.2018.00175`.

24. ISAC, Omri; BARRETT, Clark W.; ZHANG, M.; KATZ, Guy. Neural Network Verification with Proof Production. *2022 Formal Methods in Computer-Aided Design (FMCAD)*. 2022, pp. 38–48.

25. KATZ, Guy; HUANG, Derek A.; IBELING, Duligur; JULIAN, Kyle; LAZARUS, Christopher; LIM, Rachel; SHAH, Parth; THAKOOR, Shantanu; WU, Haoze; ZELJIĆ, Aleksandar; DILL, David L.; KOCHENDERFER, Mykel J.; BARRETT, Clark. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In: DILLIG, Isil; TASIRAN, Serdar (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2019, pp. 443–452. ISBN 978-3-030-25540-4.

26. KATZ, Guy; BARRETT, Clark; DILL, David L.; JULIAN, Kyle; KOCHENDERFER, Mykel J. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: MAJUMDAR, Rupak; KUNČAK, Viktor (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2017, pp. 97–117. ISBN 978-3-319-63387-9.

27. AGRAWAL, Tanay. *Binarized Neural Network (BNN) and its implementation in machine learning* [online]. 2023-08. [visited on 2024-04-11]. Available from: `https://neptune.ai/blog/binarized-neural-network-bnn-and-its-implementation-in-ml`.

28. CHENG, Chih-Hong; NÜHRENBERG, Georg; HUANG, Chung-Hao; RUESS, Harald. Verification of Binarized Neural Networks via Inter-neuron Factoring. In: PISKAC, Ruzica; RÜMMER, Philipp (eds.). *Verified Software. Theories, Tools, and Experiments*. Cham: Springer International Publishing, 2018, pp. 279–290. ISBN 978-3-030-03592-1.

29. BALUTA, Teodora; SHEN, Shiqi; SHINDE, Shweta; MEEL, Kuldeep S.; SAXENA, Prateek. *Quantitative Verification of Neural Networks And its Security Applications*. 2019. Available from arXiv: `1906.10395 [cs.CR]`.

30. LECUN, Yann. *THE MNIST DATABASE of handwritten digits* [online]. [visited on 2024-04-11]. Available from: `http://yann.lecun.com/exdb/mnist/`.

31. EITER, Thomas; GOTTLOB, Georg. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*. 1995, vol. 15, pp. 289–323.

32. GEBSER, Martin; KAUFMANN, Benjamin; SCHAUB, Torsten. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 2012, vol. 187, pp. 52–89.