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

foo bar

Abstract

foo bar

Keywords

bnn, verification, binarized neural networks answer set programming, asp

Contents

List of Tables

List of Figures

1 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 [1]. 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 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 [2, 3, 4]. 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 [5]. 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 [5, 6, 7]. As far as I know, 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. 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 [8].

While both qualitative and quantitative BNN verificators exist, they scale only up to millions of parameters for qualitative verification [9] and tens of thousands of parameters for quantitative verification [10].

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

In this thesis, I have implemented a quantitative BNN validator using the ASP paradigm in the framework Clingo from Potassco. I have also evaluated the speed of this validator against the state-of-the-art implementation BNNQuanalyst [10] on MNIST [11] and Fashion MNIST [12] datasets.

2 Used tools

2.1 Answer set programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems [13]. ASP is particularly suited for solving difficult combinatorial search problems [14]. 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.

2.1.1 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 [15]. 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

$$\mathrm{head}(r_i) \leftarrow \mathrm{body}(r_i) \tag{2.1}$$

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

$$p_0 \leftarrow p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots, \mathrm{not}\, p_n \tag{2.2}$$

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 $\forall r_i \in \Pi.\, \mathrm{body}^-(r_i) = \emptyset$, 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 $r_i \in \Pi.\, \mathrm{body}(r_i) \subseteq X \implies \mathrm{head}(r_i) \in X$. For general case the concept of *reduct of a program* Π *relative to a set* X *of atoms* is needed.

$$\Pi^X = \{\mathrm{head}(r_i) \leftarrow \mathrm{body}^+(r_i) \mid r_i \in \Pi, \mathrm{body}^-(r_i) \cap X = \emptyset\} \tag{2.3}$$

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

$$\mathrm{Cn}(\Pi) \supseteq \{\mathrm{head}(r_i) \mid r_i \in \Pi, \mathrm{body}(r_i) \subseteq \mathrm{Cn}(\Pi)\} \tag{2.4}$$

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

$$\mathrm{Cn}(\Pi^X) = X \tag{2.5}$$

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.

$$\Pi = \{p \leftarrow p, q \leftarrow \mathrm{not}\, p\} \tag{2.6}$$

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 $\{q\}$. 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.

2.1.2 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 I will describe 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 can be rewritten as following rule [14]:

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

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

Let's consider Π, $f \leftarrow \mathrm{not}\, f, p_1, \ldots, p_m, \mathrm{not}\, p_{m+1}, \ldots, \mathrm{not}\, p_n = r \in \Pi$. 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 ($\exists x \in X.\, x \in \mathrm{body}^-(r)$) 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, $f \notin X, f \notin \mathrm{Cn}(\Pi^X)$.

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

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

By adding the constraint $\leftarrow p_1, p_2$, any set that contains both p_1 and p_2 is not an answer set.

2.2 Clingo framework

Clingo is an integrated ASP system, consisting of a grounder Gringo and solver Clasp [16]. In the following section I will show basics of the clingo language, gringo as translation from clingo to aspif language and solving with clasp.

2.2.1 Clingo language

Clingo language 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 is of rule is consistent with the answer set. Fact does has 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 variables and arithmetic expressions in the parameters of atoms. 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 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 the place of the parameter in the body of rule. The wildcard behaves as anonymous variable.

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

Conditional literals

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. As intervals allows for writing sets of consecutive numbers, pooling allows for writing any set. Using pooling, previous example could be rewritten as following program.

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

In this case, each pool must be enclosed in braces, 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).
```

2.2.2 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. Specification of aspif language is written in

Aspif (ASP intermediate format) language consists of lines. First line of file is header of form

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

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

$$\texttt{h} \;\; \texttt{m} \;\; a_1 \;\; \ldots \;\; a_m$$

where $\texttt{h} \in \{0, 1\}$, $\texttt{m} \geq 0$, $\forall i \in \{1, \ldots, m\} a_i \in \mathbb{N}^+$. Parameter $\texttt{h}$ determines wherther head of this rule is disjunction (0) or choice (1), $\texttt{m}$ determines number of literals and a_i are positive literals. Body B is called normal if it has form of

$$0 \;\; \texttt{n} \;\; l_1 \;\; \ldots \;\; l_n$$

in which case it is called normal body (literals are in conjunction). Parameter $\texttt{n} \geq 0$ 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

$$1 \;\; \texttt{l} \;\; \text{n} \;\; l_1 \;\; w_1 \;\; \ldots \;\; l_n \;\; w_n$$

Parameter $\texttt{l} \geq 0$ determines lower bound, $\texttt{n}$ 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.

Show statement

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

$$4 \;\; \text{m} \;\; \text{s} \;\; \text{n} \;\; l_1 \;\; \ldots \;\; l_n$$

where `m` is length of string `s`, `s` is string with name, `n` is number length of condition and l_i are literals. If no `#show` directive is in clingo file, all atoms are to be shown.

Example

Let's illustrate the gringo parsing on some programs[1]. First the program $\Pi = \{p \leftarrow \mathrm{not}\, q, q \leftarrow \mathrm{not}\, p\}$[2].

```
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`).

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 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.

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

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).
```

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

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. 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 literal, 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 that, the atoms for the solver are only annonymous 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).
```

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

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

2.2.3 Extensions in Clingo language

Further I will show some of extensions of Clingo language usefull for the implementation part of this thesis.

Disjunctive logic programs

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

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

where $A_0; \ldots ; A_m$ are atoms forming rule head. Semantically the rule means if body holds, than at least one of $A_0, \ldots , A_m$ holds. Additionaly, set of atoms devised by this rule has to be minimal. The disjunctive logic programs are not commonly used as they are making the computational complexity higher.

```
{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

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

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

Cardinality constraint is so called conditional literal. It's semantics is that at least l and at most u of literals $p_1; \ldots ; p_m; \text{not } p_{m+1}; \ldots ; \text{not } p_n$ has to be in the answer set for this literal to hold. Cardinality constraint can be also further unboudned on either one or both sides. 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 [16].

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. This is especially usefull for defining 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.
```

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

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.
```

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

The main difference is in aspif file, line 8. Instead of positive building literals as in previous example on line 6, 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 =, the formula is rewritten using two <=. The operator != is evaluated two <= in disjunction.

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.

```
{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 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 use modified program equivalent to the one above with 20 literals. The results are written in table below.

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

Aggregates

$$\texttt{l} \diamond_1 \texttt{\#agg}\{p_1; \ldots; p_m; \text{not } p_{m+1}; \ldots; \text{not } p_n\} \diamond_2 \texttt{u}$$

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.

2.2.4 Clasp

3 Můj přínos

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 $\{(s_1, s_2, \dots, s_k) \mid s_1, s_2, \dots, s_k \in S\}$
$\mathcal{N}$	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 $\vec{v}$
$\lfloor x \rfloor$	bottom whole part
	$\lfloor x \rfloor = y \iff x = y + q \wedge 0 \le q < 1$

Table 3.1: Used notation

3.1 Binary neural network

I have been working with deterministic binarized neural networks (BNN) as defined in [17] and [10]. 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 $(t_1, t_2, \dots, t_d, t_{d+1})$.

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

$$\mathcal{N} = t_{d+1} \circ t_d \circ \cdots \circ t_1, \tag{3.1}$$

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}.

$$\begin{aligned} t_i &: \mathbb{B}_{\pm 1}^{n_i} \to \mathbb{B}_{\pm 1}^{n_{i+1}} \\ t_i &= t_i^{bin} \circ t_i^{bn} \circ t_i^{lin} \end{aligned} \tag{3.2}$$

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

$$\begin{aligned} t_{d+1} &: \mathbb{B}_{\pm 1}^{n_{d+1}} \to \mathbb{N}_0 \\ t_{d+1} &= t_{d+1}^{am} \circ t_{d+1}^{lin} \end{aligned} \tag{3.3}$$

Each layer is a function with parameters defined in Table 3.2.

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 3.2: Definition of BNN layers [10]

I have transformed the parameters to lower their count and make them integer only.

$$\begin{aligned} t_i(\vec{x}) &= (t_i^{bin} \circ t_i^{bn} \circ t_i^{lin})(\vec{x}) = y \\ y_j &= \text{sign}_{\pm 1}\left(\alpha_j \cdot \frac{\langle \vec{x}, \mathbf{W}_{:,j} \rangle + b_j - \vec{\mu}_j}{\vec{\sigma}_j} + \vec{\gamma}_j \right) \end{aligned}$$

The argument of $\text{sign}_{\pm 1}$ 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 $\frac{\vec{\alpha}_j}{\vec{\sigma}_j}$:

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

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

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

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 also encoding inputs as binary values $\{0, 1\}$.

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

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

$$\langle \vec{x}^{(b)}, \mathbf{W}'_{:,j} \rangle + \left\lfloor \frac{-\langle \vec{1}, \mathbf{W}'_{:,j} \rangle + \vec{b}'_j}{2} \right\rfloor \geq 0 \tag{3.4}$$

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]

3.2 Encoding of BNN in Clingo

Input for the Clingo was parsed from the model using Python program. This program also computed the transformation from Section 3.1. This program then outputs the model of BNN as multiple rules. Semantics are as follows.

`layer(`L, N`).`

Layer L consists of N nodes. Layer 0 is input layer, thus its number of nodes corresponds to the length of input vector. Highest layer is the output layer.

`weight(`L, P, N, W`).`

Weight from node P of layer $L - 1$ to node N of layer L is W.

`bias(`L, N, B`).`

Bias of node N of layer L is B.

`outpre(`N, P`).`

Output node N has a precedence of P. If tied, the output with higher precedence is the result.

The encoding of input region is written in Section 3.3.

I have transcribed the computation of BNN as the following clingo program.

```
1 % output layer
2 output_layer(L) :- L = #max{ X : layer(X, _) }.

3 % any combination of input bits is possible
4 {on(0, 0..K-1)} :- layer(0, K).

5 % hidden layers
6 on(L, N) :-
7     #sum{ W,I : on(L-1, I), weight(L, I, N, W) } >= B,
8     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 3.4.

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.

Encoding of the output layer in this case 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.

3.3 Encoding of input regions

An input region of BNN is some subset of inputs on which the analysis is performed. I have implemented both of input region types from [10], 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.

To perform the quantitative analysis on some input region the encoding of such region is needed.

3.3.1 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 input 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 vector $\vec{u}$.

In my implementation, full input 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.

```
% input space 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.

3.3.2 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 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 $i \in I$ is added. Note that this defintion is dual to the one in [10].

```
% 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.

3.3.3 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 feature for the output to be adversarial, I have used the same encoding as for the computing of the output of the network.

```
% 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).
```

4 Evaluation

5 Discussion

Bibliography

1. 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`.
2. 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`.
3. 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]`.
4. 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`.
5. 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. Available also from: `https://api.semanticscholar.org/CorpusID:249240518`.
6. 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.

7. 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.

8. 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`.

9. 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.

10. 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`.

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

12. XIAO, Han; RASUL, Kashif; VOLLGRAF, Roland. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*. 2017.

13. LIFSCHITZ, V. What is answer set programming? In: *Proc. 23rd AAAI Conf. on Artificial Intelligence, 2008*. 2008, pp. 1594–1597.

14. ANGER, Christian; KONCZAK, Kathrin; LINKE, Thomas; SCHAUB, Torsten. A Glimpse of Answer Set Programming. *Künstliche Intell.* 2005, vol. 19, no. 1, p. 12.

15. 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`.

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

17. HUBARA, Itay; COURBARIAUX, Matthieu; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Binarized Neural Networks. *ArXiv*. 2016, vol. abs/1602.02505. Available also from: `https://api.semanticscholar.org/CorpusID:6453539`.