Global Journal of Computer Science and Technology: D
Neural & Artificial Intelligence
Volume 24 Issue 1 Version 1.0 Year 2024
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Optimizing the Running Time of a Trigger Search Algorithm based on the Principles of Formal Verification of Artificial Neural Networks

By Aleksey Tonkikh & Ekaterina Stroeva
Moscow State University

Abstract- The article examines the problem of scalability of the algorithm for searching for a trigger in images, which is based on the operating principle of the Deep Poly formal verification algorithm. The existing implementation had a number of shortcomings. According to them, the requirements for the optimized version of the algorithm were formulated, which were brought to practical implementation. Achieved 4 times acceleration compared to the original implementation.

Keywords: *formal verification, machine learning, trigger injection attacks, backdoor attacks your.*

GJCST-D Classification: *LCC Code: QA76.87*

Strictly as per the compliance and regulations of:

Optimizing the Running Time of a Trigger Search Algorithm based on the Principles of Formal Verification of Artificial Neural Networks

Aleksey Tonkikh [α] & Ekaterina Stroeva [σ]

Abstract- **The article examines the problem of scalability of the algorithm for searching for a trigger in images, which is based on the operating principle of the Deep Poly formal verification algorithm. The existing implementation had a number of shortcomings. According to them, the requirements for the optimized version of the algorithm were formulated, which were brought to practical implementation. Achieved 4 times acceleration compared to the original implementation.**

Keywords: formal verification, machine learning, trigger injection attacks, backdoor attacks your.

I. Introduction

This paper discusses a trigger search algorithm that is based on one of the algorithms for the formal verification of neural networks, which is an urgent task, since many technology companies are faced with the problem of attacks using trigger overlays on images when training neural networks, as well as with the need to check the robustness of neural networks, which can be done mainly using formal verification algorithms.

In turn, one of the main problems of formal verification algorithms is the long operating time. This article proposes some methods to reduce the running time of the algorithm [1], which is used to detect the presence of a trigger in images from the MNIST dataset [2].

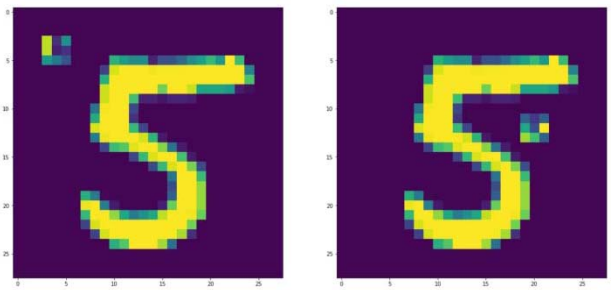

Fig.1: Example of a trigger and its location

Basic Definitions and Notations:

N —neural network;

I— an image that is analyzed in terms of the presence of a trigger;

X — set of images I, which is checked by the algorithm;

n — number of pixels in the image;

x_i — the value of the neuron before the one that is currently being analyzed;

x_j — calculated current value of the neuron;

$x_i \in [l_i, u_i]$ — range of values for each neuron;

$\phi_{pre} = [h_p, w_p] \le j \le [h_p + h_s, w_p + w_s] \wedge 0 \le x[j] \le 1$— preconditions for pixels that may contain trigger;

(c_s, h_s, w_s) — trigger parameters: number of channels, height and width, respectively;

(h_p, w_p) — upper left coordinate of the trigger;

t_s — output value of the neural network for the image with a trigger superimposed on it;

θ — specified success probability value;

K— the number of images in the sample checked for the absence of a trigger, or the number of elements in the set X.

A trigger is a rectangular sticker on an image that has the same number of channels and changes the classification (it is assumed that the trigger is the same for all images of a certain set and is located in the same place), for example, a 3×3 square with pixels of different colors in Fig.1.

Formal definition: for a neural network solving the problem of classifying images of size (c, h, w), the trigger is some image $S$of size(c_s, h_s, w_s) such that $c_s = c, h_s \le h, w_s \le w$.

We can say that in the picture I there is a trigger of size(c_s, h_s, w_s), the upper left corner of which is located at the place (h_p, w_p)(subject to the obvious conditions $h_p + h_s \le h, w_p + w_s \le w$), if

$$I_s[c_i, h_i, w_i] = \begin{cases} S[c_i, h_i - h_p, w_i - w_p], if \\ (h_p \le h_i < h_p + h_s) \wedge \\ \wedge (w_p \le w_i < w_p + w_s); \\ I[c_i, h_i, w_i], otherwise. \end{cases}$$

In other words, the trigger changes certain pixels of the image to given ones.

Formal Statement of the Problem

There is no patch (trigger) S such that when applied to a certain set of images $I \in X$, the neural

Author α σ: Moscow State University. e-mails: alexej-t@mail.ru, katestroeva@gmail.com

network Nchanges the output class to the target class t_s, on imagesI_s with the trigger S:

$$\exists S(c_s, h_s, w_s): \forall I_s \in X: N(I_s) = t_s.$$

Initial conditions:

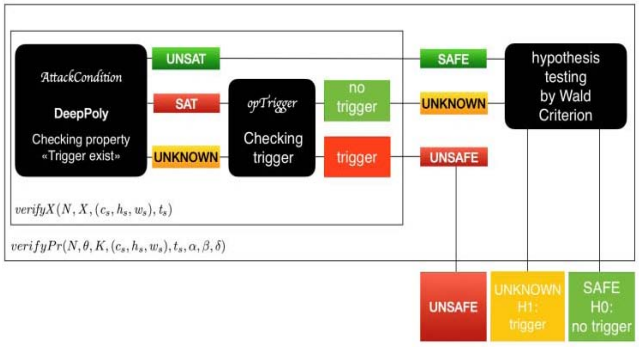

Fig. 2: Flowchart of the algorithm for searching for a trigger

1. *Dataset:* MNIST — 10,000 images in $1 \times 28 \times 28$ format; neural networks: fully connected and convolutional with activation functions ReLU, Sigmoid, Tanh with the number of parameters up to 100,000;
2. *Trigger Parameters:* $1 \times 3 \times 3$, any pixel values in the area;
3. *Security Property:* no trigger;
4. *Verification Algorithm:* DeepPoly.

II. Algorithm for Searching for a Trigger in an Image

The algorithm [1] is based on the DeepPoly verifier [3]. Its main goal is to search for a trigger that consistently fools the classifier for a certain number of images. The output value of the artificial neural network changes to a predetermined value. The search is performed over the entire image and for all possible values of each trigger pixel (a 3×3 trigger is considered and tested, although other values are possible). The Wald Criterion [4] is also used to evaluate hypotheses about the occurrence of a trigger.

Step by step, the entire algorithm works as follows:

1. Fix the position of the trigger. In the future, it is in this fixed area that there will bechecking for the presence of a trigger;
2. We go through the set of images and build variations of images:
   a) Calculate for an artificial neural network and a given image a set of constraints. Constraints are calculated in the body of the attack Condition function;
   b) We pass these restrictions to the SAT solver, and look at the answer: if the formula is degenerate, then there is no trigger for the image, therefore, the neural network is resistant to triggers;
   c) Otherwise we add these restrictions to the previous ones;
3. If the SAT solver finds a counterexample, then, consequently, there is a trigger. We find it by gradually parsing the solution to a Boolean function, which is performed in the opTrigger function;
4. If the SAT solver confirmed that the set of constraints does not have a solution, then the neural network works correctly;
5. If the SAT solver could not confirm the degeneracy of the constraints, and a trigger was not found, then more research needs to be done.

The relationships between the Attack Condition, opTrigger functions and all of the listed methods are presented in the block diagram in Fig. 2.

a) Description of how the Verify Pr Function Works

The algorithm is represented by the function verify Pr, which takes as input data the neural network N, the number of pictures K in the sample being tested, all trigger indicators (c_s, h_s, w_s), $t_s$probabilistic parameters α, β, δ of Wald criterion (SPRT) [4] and provides information about the presence or absence of a trigger with a given probability (Fig. 3).

Algorithm 1: $verifyPr(N, \theta, K, (c_s, h_s, w_s), t_s, \alpha, \beta, \delta)$

```
1  let n ← 0 be the number of times verifyX is called;
2  let z ← 0 be the number of times verifyX returns SAFE;
3  let p0 ← (1 − θ^K) + δ, p1 ← (1 − θ^k) − δ;
4  while true do
5      n ← n + 1;
6      randomly select a set of images X with size K;
7      if verifyX(N, X, (c_s, h_s, w_s), t_s) returns SAFE then
8          z ← z + 1;
9      else if verifyX(N, X, (c_s, h_s, w_s), t_s) returns UNSAFE then
10         if the generated trigger satisfies the success rate then
11             return UNSAFE;
12     if p1^z / p0^z × (1−p1)^(n−z) / (1−p0)^(n−z) ≤ β/(1−α) then
13         return SAFE; // Accept H0
14     else if p1^z / p0^z × (1−p1)^(n−z) / (1−p0)^(n−z) ≥ (1−β)/α then
15         return UNKNOWN; // Accept H1
```

Fig. 3: Pseudocode for the verify Prfunction [1]

[lines 1-2] Two variables are introduced: n— the number of calls to the verify X function, z— the number of SAFE responses returned by the verify X function.

[line 3] Set the probabilities $p_0, p_1$for using SPRT.

[line 4] A loop is started that runs until the SPRT conditions are met, as soon as the test monitors the fulfillment of one of the conditions, the result is given which hypothesis should be accepted [lines 12-15].

[line 5] A counter is started for the number of calls to the verify X function.

[line 6] Selecting K images randomly and composing them into a verifiable set X, which is fed to the input of the verifyX function.

[lines 7-11] Application of the verifyX function, which will be described in the following pseudocode (Fig. 4). The SAFE output means that you need to increase the z variable by 1 and go to a new iteration of SPRT, the UNSAFE output checks that the flip-flop does not satisfy all the specified statistical parameters and moves on to SPRT.

b) Description of how the Verify X Function Works

The verify X function takes as input the neural network N, the tested set of images X, the dimensions (c_s, h_s, w_s) and position (h_p, w_p) of the trigger, the target label for the trigger t_s.

Algorithm 2: $verifyX(N, X, (c_s, h_s, w_s), t_s)$

```
1  let hasUnknown ← false;
2  foreach trigger position (h_p, w_p) do
3      let φ ← φ_pre;
4      foreach image I ∈ X do
5          let φ_I ← attackCondition(N, I, φ_pre, (c_s, h_s, w_s), (h_p, w_p), t_s);
6          if φ_I is UNSAT then
7              φ ← false;
8              break;
9          else
10             φ ← φ ∧ φ_I;
11     if solving φ results in SAT or UNKNOWN then
12         if opTrigger(N, X, φ, (c_s, h_s, w_s), (h_p, w_p), t_s) returns a trigger then
13             return UNSAFE;
14         else
15             hasUnknown ← true;
16 return hasUnknown ? UNKNOWN : SAFE;
```

Fig. 4: Pseudo code for the verify X function [1]

At the output, the verify X function produces the response SAFE if there is no trigger in the selected set X or UNSAFE if there is a trigger (Fig. 2).

[line 1] The has Unknown variable is created, which is responsible for the case of uncertainty (it is impossible to get an answer about the presence or absence of a trigger), by default its value is set to False.

[line 2] The cycle is started to cycle through all possible trigger locations on the image being checked.

[line 3] The neural network is specified by a set of conjunctions ϕ, that is, in a form accessible to the SAT solver. During initialization, a set of initial constraints $\phi_{pre} = \wedge_{j \in P(w_p, h_p)} lw_j \le x_j \le up_j$ for the value ofpixels x_j located at positions $j \in P(w_p, h_p)$ in which thelocation is assumed at this step is entered into this variable trigger. Here$lw_j$and $up_j$are normalized boundaries for the trigger value, lying in the interval $[0; 1]$.

[line 4] For each image $I \in X$, a cycle is started to check each image for the presence of a trigger.

[line 5] The main function for checking the presence of a trigger attack Condition uses the DeepPoly formal verification algorithm for neural networks, which checks the property "there is a trigger on the image", returns an image represented in the form of conjunctions ϕ_I, and a SAT response if the property is satisfied (trigger found), UNSAT - property not satisfied (trigger not found).

[lines 6-10] If the attack Condition function returned UNSAT in the previous step, then the neural network is not executable, the variable ϕ is assigned the value False, exiting the loop. If a trigger is found, then its representation $\phi_I$is added to the neural network function.

[lines 11-15] The resulting representation of the neural network ϕ is fed into the SAT solver, and if the output is SAT or UNKNOWN, then the opTrigger function is run.

i. *The function opTigger*

First checks whether the resulting rectangle S of size (c_s, h_s, w_s) at position $(h_p, w_p)$is a trigger that successfully attacks every image I in the test set X. Because If the accumulated error of abstract interpretation resulting from the application of the DeepPoly algorithm is too large, the resulting model may be a false trigger. If it is a real trigger, then it returns model S and the output is UNSAFE.

The opTrigger function creates a trigger based on the available data, using an approach based on optimizing the loss function:

$$loss(N, I, S, (h_p, w_p), t_s) = \begin{cases} 0, & if\ n_s > n_0; \\ n_0 - n_s - \epsilon, & otherwise. \end{cases}$$

$n_s = N(I_s)[t_s]$ — outputfor target label t_s; $n_0 = max_{j \neq t_s} N(I_s)[j]$ — next after the largest value of the output vector; ϵ is a small constant, about 10^{-9}.

The loss function returns 0 if the attack on I by the trigger is successful. Otherwise, it returns a quantitative measure of how far the simulated attack is from being successful.

For the entire tested set X we obtain a joint loss function

$$loss(N, X, S, (h_p, w_p), t_s) = \sum_{I \in X} loss(N, I, S, (h_p, w_p), t_s)$$

An optimization problem is then solved to find an attack that successfully changes the classification of all images in X:$argmin_S loss(N, X, S, (h_p, w_p), t_s)$.

ii. *The Attack Condition Function*

Takes all parameters as input and outputs the result

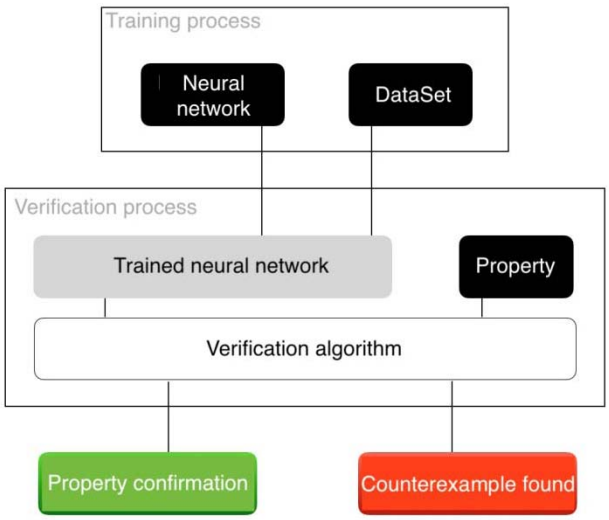

Fig. 5: Pseudocode for the Attack Condition function

there is a trigger or not a trigger (Fig. 5). Inserts restrictions on the trigger in the form of conjunctions and adds ϕ to the network structure.

The main idea of checking for a trigger: the area of pixels in which the trigger will be placed is selected, each such pixel is assigned a symbolic value included in the interval $[0;1]$ [lines 1-8]. Next, using the DeepPolyReLU function, we track the moment at which the checked pixel value from the segment $[0;1]$ will change the output vector of values, that is, the classification will change, we obtain the pixel value at which the trigger will be located on this pixel [lines 9-21].

If for all values of the checked pixel from the segment $[0;1]$there is no change in the value of the target label (the output segment for the target label at all points is greater than the output segments for all other values) [line 25], then there will be no trigger, we return UNSAT [line 26], if it is not clear whether the target label has changed or not (the output segment for the target label intersects with some output segment for some of the other values), then the situation requires moredeep analysis, UNKNOWN is returned [lines 29 and 37]. If the output label has definitely changed, then the trigger is found, SAT is returned [line 35].

The DeepPoly algorithm [3], like all formal verification algorithms [5], checks properties (Fig. 6). In the context of the verifyX function, the "no trigger" property is checked.

Algorithm 1 attackCondition$(N, I, (c_s, h_s, w_s), (h_p, w_p), t_s)$

```
1: let constraints ← [[0,0]] * (h * w);
2: for j ← 0,...,h * w do
3:     if j ∈ φ_pre then
4:         constraints[j] ← ([0,1]);
5:     else
6:         constraints[j] ← ([I[j], I[j]]);
7:     end if
8: end for

9: foreach layer ∈ N do
10:    if layer is ReLU then
11:        for i ← 0,...,len(constraints) do
12:            constraints[i] ← DeepPolyReLU(constraints[i][0], constraints[i][1]);
13:        end for
14:    else
15:        NewConstraints ← [[0,0]] * (layer.CountNeurons);
16:        for i ← 0...len(layer.CountNeurons) do
17:            NewConstraints[i] ← AffineCompute(constraints, layer.weights[i], layer.biases[i]);
18:            constraints ← NewConstraints;
19:        end for
20:    end if
21: end for

22: let flag ← True
23: for i ← 0,...,len(constraints) do
24:    if i ≠ t_s then
25:        if constraints[t_s][1] < constraints[i][0] then
26:            return UNSAT;
27:        else
28:            if constraints[t_s][0] < constraints[i][1] then
29:                flag ← False;
30:            end if
31:        end if
32:    end if
33: end for

34: if flag then
35:    return SAT;
36: else
37:    return UNKNOWN;
38: end if
```

Fig. 6: General scheme of operation of algorithms for formal verification of neural networks

c) Deep Poly ReLU function

Analyzes approximate values for the output of the ReLU activation function (Fig. 7).

Algorithm 2 DeepPolyReLU(l, u)

```
1: if 0 ∈ [l_i, u_i] then
2:     let S_1 ← (u_i − l_i) * u_i/2;
3:     let S_2 ← (u_i − l_i) * (−l_i)/2;

4:     if S_1 < S_2 then
5:         return [0, u_i];
6:     else
7:         return [l_i, u_i];
8:     end if

9: else

10:    if u_i < 0 then
11:        return [0, 0];
12:    else
13:        return [l_i, u_i];
14:    end if
15: end if
```

Fig. 7: Pseudo code for the Deep Poly ReLU function

d) Basic Moments

- Linear constraints on each neuron are represented as a linear combination of only input data x_1, x_2(and not through the constraints of previous neurons), then the constraints for each neuron at each step will be better, the segment will expand less.
- If the ReLU input receives a segment with a half-living ends, then it turns into itself, without changes. If the segment contains a point zero, then as constraints we use $0 \leq x_j \leq \lambda x_i + \mu$ (the equation of

the straight line defining the upper boundary of the triangle passing through the points $(l_i; 0)$, $(u_i; u_i)$, l_i and u_i — boundaries of the interval in the previous step). If the entire segment is negative, then it simply goes to zero.

- The main difference from other algorithms is exactly one lower constraint. This makes it possible to narrow the boundaries of the intervals and facilitate computing power (Fig. 8). It is also argued that approximation by such triangles is better than zonotopes — they are easier to calculate, and also often have a smaller area. With a similar formulation of the problem, the zonotope in this case will be a parallelogram, the lower side of which contains the point $(0; 0)$.

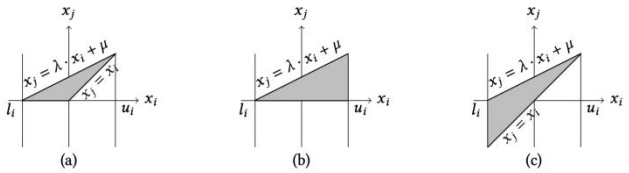

Fig. 8: Approximation of the ReLU function in the DeepPoly algorithm [3]

The AttackCondition function takes all parameters as input and outputs the result — there is a trigger or there is no trigger. Inserts restrictions on the trigger in the form of conjunctions and adds ϕ to the network structure.

These results are then used in the VerifyPr function, which gives a probabilistic assessment of the presence of a trigger.

e) The Affine Compute Function

Takes as input values from the previous layer, performs standard affine transformations — multiplying by weights and adding a bias vector, and at the output produces an interval within which all possible values supplied to the input of the ReLU function lie (Fig. 9).

Algorithm 3 AffineCompute(constraints, w, bias)

```
let l_j ← bias;
let u_j ← bias;
for i ← 0, ..., len(constraints) do
    l_j ← l_j + w[i] * constraints[i][0];
    u_j ← u_j + w[i] * constraints[i][1];
end for
return [l_j, u_j];
```

Fig. 9: Pseudocode for the AffineCompute function

f) SPRT (Sequential Probability Ratio Test) or Wald Criterion

Designations:

θ is the probability of a trigger appearing, common to all K pictures: for a given neural network N, trigger S, target label t_s, it is postulated that S has a probability of success θ if and only if there is a position (h_p, w_p)such that the probability of occurrence $L(N(I_s)) = argmax_i(y_{output}) = t_s$ for any I in the chosen test set is θ.

No trigger:

$$\forall I \in X \exists s, I_s = I(s): L(N(I_s)) > t_s,$$

where α, β, δ are confidence levels.

Testable hypotheses:

H_0: The probability of no attack on a set of Krandomlyselected images is greater than$1 - \theta^K$.

H_1: The probability of no attack on a set $K$of randomly selected images is no greater than $1 - \theta^K$.

Next, the researcher sets the values of the parameters α and β, this is the probability of an error of the first and second kind, respectively (Fig. 10).

	Decision	
Truth	*accept* H_0, *reject* H_1	*reject* H_0, *accept* H_1
$p \geq \theta$: H_0 *true*, H_1 *false*	correct ($>1-\alpha$)	type I error ($\leq\alpha$)
$p < \theta$: H_0 *false*, H_1 *true*	type II error ($\leq\beta$)	correct ($>1-\beta$)

Fig.10: Errors of type 1 and 2

Parameter δ is the "gap" between the null and alternative hypothesis. If the value falls in a region where the estimated probability of not having the attack will be greater than $p_0 = (1 - \theta^K) + \delta$, then we accept the null hypothesis, if less than $p_1 = (1 - \theta^K) - \delta$, then we reject the null hypothesis , if between them, then we move on to a new iteration of the algorithm. This is precisely the procedure of sequential analysis, which consists in sequential testing of the indicated inequalities for probabilities, and in this way it differs from simple testing of hypotheses.

The article [1] sets the following parameter values $K = 5,10,100, \theta = 0.8,0.9,1, \alpha = \beta = \delta = 0.01$.

III. Experimental Part

a) Scalability Study

A scalability study showed that for neural networks with about 10,000 parameters, searching for triggers for all 10 labels takes about several minutes. In article [1] and the implementation, a search for triggers for the conv_small_relu neural network was proposed (the architecture is shown in Fig. 11). Such a neural network contains 89,000 parameters. Finding triggers for all 10 tags takes about 10 hours.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 13, 13]             272
              ReLU-2          [-1, 16, 13, 13]               0
            Conv2d-3            [-1, 32, 5, 5]           8,224
              ReLU-4            [-1, 32, 5, 5]               0
           Flatten-5                 [-1, 800]               0
            Linear-6                 [-1, 100]          80,100
              ReLU-7                 [-1, 100]               0
            Linear-8                  [-1, 10]           1,010
================================================================
Total params: 89,606
Trainable params: 89,606
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.34
Estimated Total Size (MB): 0.41
----------------------------------------------------------------
```

Fig. 11: Neural networkconv_small_relu architecture

Similar architectures with fewer and more parameters were tested. For neural networks with about 105,000 parameters, verification for one target label takes about 20 hours (for 10 labels it takes approximately 200 hours). From this we conclude that the duration of verification increases exponentially with increasing number of parameters.

b) Improved Work Speed

During the analysis of the repository, the bottleneck was identified — the back_substitute function of the utils.py module, which is responsible for the integration of interval arithmetic. Profiling of this program shows that about70%of the execution time is occupied by this function (Fig. 12). The calculation graph shows similar results (Fig. 13).

Name	Call Count	Time (ms)		Own Time (ms)	
back_substitute	24430640	22239630	92.9%	17060946	71.3%
<method 'reduce' of 'numpy.ufunc' objec	171164012	1590864	6.6%	1590864	6.6%
<method 'nonzero' of 'numpy.ndarray' ol	156399360	1305071	5.5%	1305071	5.5%
f_wrapped	776610376	6201004	25.9%	1178936	4.9%
find_top_boxed_args	776610376	564045	2.4%	419477	1.8%
<built-in method numpy.core._multiarray	588557196	4058328	17.0%	234355	1.0%
roll	47374080	281605	1.2%	181411	0.8%
<lambda>	1580220125	171697	0.7%	171697	0.7%
_wrapreduction	17116201	1832933	7.7%	167766	0.7%
sum	171160960	1999585	8.4%	140632	0.6%
f_wrapped	238679620	251373	1.1%	105127	0.4%
delete	27040	100982	0.4%	100528	0.4%
array	143931460	506997	2.1%	96540	0.4%
apply_poly	13520	11371319	47.5%	95687	0.4%
<built-in method numpy.array>	143947586	78817	0.3%	78817	0.3%
<built-in method numpy.zeros>	29865701	70790	0.3%	70790	0.3%
sum	171160960	2143509	9.0%	70448	0.3%
<dictcomp>	17116201	62815	0.3%	62815	0.3%
append	47374080	1042424	4.4%	61034	0.3%

Fig.12: Table of execution times of all algorithm functions

To optimize the selected bottleneck, various approaches to code optimization and library replacement were tested, as well as deployment on GPUs using the PyTorch library.

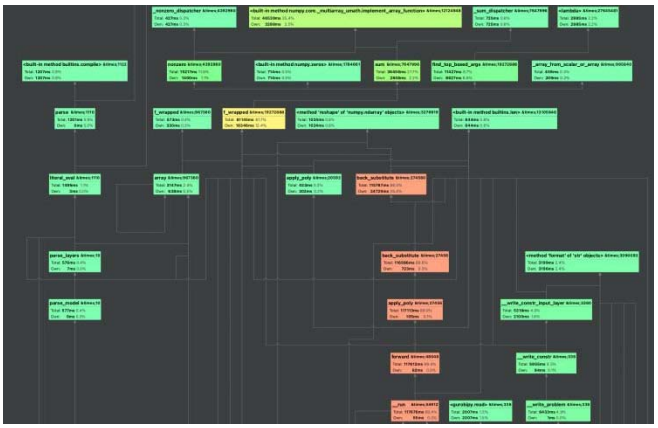

Fig.13. Calculation graph of all algorithm functions

It was not possible to obtain a significant increase in performance using the GPU, since the method uses a large number of not very complex calculations. As a result, calculations slowed down 10 times. This happened because GPUs are adapted for calculating large matrices, while the CPU copes better with the proposed task. The use of other libraries and code optimization led to a 20 percent improvement in the execution speed of the back_substitute function. The overall running time of the algorithm was also improved by approximately 10% (Fig. 14).

```
No backdoor
No false alarm
No stamp
Running time = 496m 6s
         4363399642 function calls (4172002925 primitive calls) in 29766.186 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 24430640 24456.822    0.001 28484.934    0.001 utils.py:147(back_substitute)
```

```
No backdoor
No false alarm
No stamp
Running time = 534m 51s
         8737689942 function calls (8591859206 primitive calls) in 32091.220 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 24430640 23547.410    0.001 29343.768    0.001 utils.py:147(back_substitute)
```

Fig.14: Comparison of algorithm running time before and after optimization improvements

Parallelization of the selected problem is impossible, since the newly calculated data must again be fed into the input. Nevertheless, you can try to parallelize the search for a trigger in different places, but this issue is subject to deeper study.

IV. Practical Implementation

a) Software and Hardware

The main part of the described experiments was carried out on a computer complex using a central processor and having the following characteristics:

Table I: Hardware

CPU	Apple M1 Max processor
RAM	32 GB

Experiments on the GPU were carried out using a computing cluster with the characteristics indicated in Table II.

Table II: Hardware, GPU cluster

Video card	NVIDIA RTX A6000
Processor	AMD EPYC 7532 32-Core
RAM	252 GB

Software with the characteristics shown in Table III was used.

Table III: Software

OS (CPU)	MacOS Ventura 13.3.1
OS (GPU)	Ubuntu 20.04.4 LTS
Python	3.10.0
numpy	1.23.5
scipy	1.8.0
autograd	1.4
	9.5.1
torchsummary	1.5.1
nvidia cuda	11.7
pytorch	1.13.1

b) Datasets and Neural Networks

Neural networks trained on the following data sets were used:

- MNIST – a set of single-channel images of28×28pixels. Images are divided into 10 classes — numbers from 0 to 9 (Fig. 15);
- CIFAR-10 – a set of three-channel images of 32×32pixels. Images are divided into 10 classes — airplane, car, bird, cat, deer, dog, frog, horse, ship, truck (Fig. 16).

In addition to the experiments proposed in the article, other neural networks were trained. They were analyzed using a trigger search algorithm and used to compare the original implementation and the optimized version. Neural networks that showed high accuracy on the test set, as a rule, did not have a trigger. An example of a tested neural network is shown in Fig. 11.

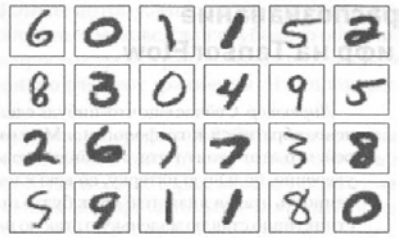

Fig.15: MNIST DataSet

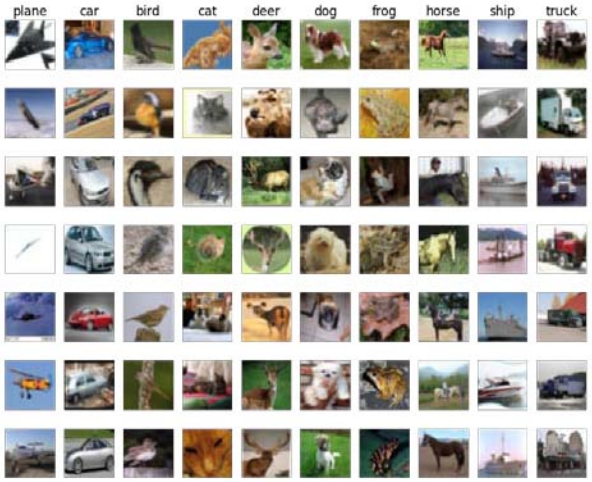

Fig.16: CIFAR-10 DataSet

c) Disadvantages of the Current Implementation

Formal verification, as a young field of science, has many difficulties with uniform standards of use. The proposed implementation of the trigger search problem has a number of significant problems that arise for the user who decides to use this algorithm. It was decided to correct the identified deficiencies as part of this work.

1. The proposed implementation works only with neural networks stored in a special format, where all weights and biases are stored in separate txt files, and the architecture itself is written in a separate spec.json file. To read neural networks in this format, a separate json_parser module is used, which extracts the weights of the neural network and prepares them for work. The inability to conduct an experiment on a neural network not described by the authors is a significant drawback;
2. The proposed algorithm works quite slowly even on small neural networks, which is natural, since formal verification very carefully analyzes the entire neural network layer by layer, neuron by neuron. Since when testing more complex neural networks with a large number of parameters, the key limitation is the running time of the algorithm, optimizing it will increase its applicability. Also, the existing implementation does not use parallelization and it was decided to fix this too;
1. The existing implementation is only suitable for testing neural networks trained on the MNIST

dataset, and has not been adapted or tested for the CIFAR-10 dataset;

1. Support for only a limited set of layers, such as two-dimensional convolutional and fully connected layers.

In connection with the identified shortcomings, requirements were drawn up for an optimized version of the existing implementation:

1. The ability to search for triggers for ANN written in PyTorch;
2. Adding parallelization at various levels;
3. Adding support for neural networks, trained on the CIFAR-10 data set;
4. Adding support for MaxPool1D, MaxPool3D, Conv1D, Conv3D layers.

d) GPU Usage

As part of solving the optimization problem and using parallel computing, using profiling methods, a bottleneck was identified — the back_substitute function of the utils module (Fig. 13). Implementation of this function using the PyTorch library and graphics processing unit (GPU) did not give the expected acceleration.

This happened because the formal verification problem is poorly adaptable to GPU computing. During the calculation, there are quite a few operations that are similar to each other, and most of them depend on the previous step, which makes the use of the GPU ineffective.

It was decided to replace the used autograd library with numpy. Since the numpy library is written in C and Fortran programming languages, it is highly optimized. The autograd library is a "wrapper" of already optimized algorithms, which gives a series of small delays that accumulate and give a significant slowdown with a large number of calls. Replacing the autograd library with the numpy library increased the speed of the back_substitute function by 20 percent, and the speed of the entire algorithm by an average of 10 percent.

The table below shows the running time of the back_substitute function using various libraries. For each library, 10,000 calculations were carried out and the average value was calculated:

Table IV: Running time of the back_substitute function for different libraries

Library	Back_substitute running time (s)
autograd	0.00023
PyTorch (GPU)	0.00225
numpy	0.00018

e) Using Parallel Computing

During the study of the existing implementation of the trigger search algorithm, places were identified that could be optimized using parallelization; the pseudocode is presented in Fig. 17.

Algorithm 2: $verifyX(N, X, (c_s, h_s, w_s), t_s)$ #pragma parallel

```
1  let hasUnknown ← false;
2  foreach trigger position (h_p, w_p) do                #pragma parallel
3      let φ ← φ_pre;
4      foreach image I ∈ X do
5          let φ_I ← attackCondition(N, I, φ_pre, (c_s, h_s, w_s), (h_p, w_p), t_s);
6          if φ_I is UNSAT then
7              φ ← false;
8              break;
9          else
10             φ ← φ ∧ φ_I;
11     if solving φ results in SAT or UNKNOWN then
12         if opTrigger(N, X, φ, (c_s, h_s, w_s), (h_p, w_p), t_s) returns a trigger then
13             return UNSAFE;
14         else
15             hasUnknown ← true;
16 return hasUnknown ? UNKNOWN : SAFE;
```

Fig. 17: Pseudocode of the VerifyX function indicating places of parallelization

It was decided to test parallelization in two selected areas: at the stage of selecting a target label and at the stage of enumerating trigger locations. This is possible thanks to the following process. For each trigger location, a chain of conjunctions of admissible intervals of all neurons in the neural network is calculated. Since the sequence of conjunctions does not change its meaning depending on the location of the conjunction in the chain, the result when applying parallelization remains unchanged. The proposed parallelization in both cases was implemented using the standard multiprocessing library and in total gave a significant increase in speed in various experiments. On average, on the tested neural networks, an acceleration of 4 times was obtained relative to the original implementation. The results of the experiments are shown in Table V. The first half shows the results for fully connected neural networks, and the second half for convolutional ones.

Other optimizations implemented according to the formulated requirements are listed below:

1. As part of the work, a sequence of actions was implemented to convert any neural network written in PyTorch into a specialized format used by the trigger search algorithm. This pipeline has been tested for all possible types of layers and architectures, including those that were not studied in the original article;
2. To support layers of new types, the corresponding classes were implemented with processing built according to the DeepPoly formal verification method;

3. To support the CIFAR-10 data set, the images in it were normalized from 0 to 1 and converted into the appropriate specialized format. The existing implementation was adapted to use a $3 \times 3 \times 3$ trigger, and support for multi-channel triggers was added wherever this was lacking.

Table V: Algorithm running time before and after optimization

Neural network	Number of parameters	Original time	Optimized time
mnist_model_0	79 510	1 223 s	341 s
mnist_model_1	159 010	2 352 s	659 s
mnist_model_2	199 310	6 873 s	1 704 s
mnist_model_3	119 810	5 394 s	1 328 s
mnist_conv_small	89 606	22 452 s	4 548 s
mnist_model_5	159 387	258 854 s	74 855 s
mnist_conv_maxpool	34 622	17 880 s	3 632 s
cifar_conv_relu	62 006	—	197 426 s

f) Assessing the complexity of the trigger search algorithm in neural networks of various architectures

The time it takes to search for a trigger in a neural network depends on its architecture. The complexity of testing a neural network can be determined both empirically and theoretically. It will correlate with the number of parameters and depend on: the number of layers in the neural network, the size of these layers, the type of these layers. Empirically, it was found that fully connected layers are faster to check than convolutional layers. The verification time depends to a greater extent on the number of layers and to a lesser extent on their size.

The number of parameters for verifying fully connected layers can be expressed as $O(m * n)$, where m and n are the number of inputs and outputs of the layer. The number of parameters for verifying 2D layers can be expressed as$O(c * h * w)$ for Conv2D and $O(c * h^2 * w^2)$ for MaxPool2D, where$c, h$and w are the number of channels, the height and width of the kernel. Thus, it is easy to see that the more parameters the 2D layers have, the longer the neural network will take to verify.

Table VI: Estimation of the complexity of searching for a trigger for different types of layers

Layer type	Complexity
Linear	$O(m * n)$
Conv2D	$O(c * h * w)$
MaxPool2D	$O(c * h^2 * w^2)$

V. Conclusion

Formal verification algorithms are generally applicable to checking neural networks for the absence of attacks. Using the DeepPoly algorithm, you can not only check for the presence of a trigger in an image, but also generate triggers. Verification problems arise on networks containing Sigmoid and Tanh activation functions.

Probabilistic models provide a numerical assessment of testing the operation of neural networks; they can be combined with formal verification algorithms. As a continuation of the work, you can try to use other verification algorithms for these experiments, based on the analysis by zonotopes [5] of the Sigmoid and Tanh activation functions.

References Références Referencias

1. Pham Long H., Sun Jun. Verifying neural networks against backdoor attacks // Springer International Publishing.— 2022.— URL: https: //link.springer.com/content/pdf/10.1007/978-3-031-13185-1.pdf.
2. Deng Li. The mnist database of handwritten digit images for machine learning research // IEEE Signal Processing Magazine. — 2012. — Vol. Volume: 29, Issue: 6. — URL: https://ieeexplore.ieee.org/document/ 6296535.
3. An abstract domain for certifying neural networks/ Gagandeep Singh, Timon Gehr, Markus Püschel, Martin Vechev//Proc. ACM Program. Lang. — 2019. — jan. — Vol. 3, no. POPL. — 30 p. — URL: https: //doi.org/10.1145/3290354.
4. Wald A. Sequential tests of statistical hypotheses// The Annals of Mathematical Statistics. — 1945. — Vol. Vol. 16, No. 2. — URL: https://www.jstor.org/.
5. Ekaterina Stroeva Aleksey Tonkikh. Methods for formal verification of artificial neural networks: A review of existing approaches//Interna- tional Journal of Open Information Technologies. — 2022. — Vol. Vol 10, No 10. — URL: http://injoit.org/index.php/j1/article/view/1417.