Indian Institute Of Technology - Delhi

Amar Nath and Shashi Khosla School of Information Technology (SIT)
Centre of Excellence in Cyber Security and Information Assurance (CoE-CSIA)


# Finding Minimal Modifications in Deep Neural Networks.

*Submitted by -*
Tooba Khan (2021JCS2245)

*Submitted to -*
Dr. Subodh Vishnu Sharma
Dr. Kumar Madhukar

# Contents

# 1 Problem-1: Minimal Modifications in DNNs

The problem discussed in this section is to find the minimal modification in deep neural networks. By minimal modifications we mean, given an edge with weight $w_i$ , we want to know if we can add an $\epsilon$ to $w_i$ such that on an Input I, the output changes. More specifically, for certain inputs, we know that the output is wrong. We want to know the smallest change required in the network to correct this misprediction.

## 1.1 Motivation

This technique can be used to correct known faults in the neural network by making minimum modifications in the network itself without retraining it using an adversarial data set. This technique can also be used to prove the robustness of watermarks and find minimum change required to break the watermarks. This is a white box approach to correct faults in a DNN.

## 1.2 Implementation

The existing work discusses about changing the weights of the last layers i.e the weights connecting the last hidden layer to the output layer.
Two approaches have been implemented as a part of this project:

### 1.2.1 Finding Minimal modification in the last layer by restricting the upper and lower bounds of epsilons:

In the neural network present in figure 1, for Inputs [3,5] the output is [-1,1] i.e the v3,1<v3,2. We want to find the smallest change in the neural network such that v3,1>v3,2. To solve this problem, we add an $\epsilon$ to all the weights in the last layer. Since v3,1 has to increase, we call it as the increment neuron and v3,2 as the decrement neuron because we want it's value to decrease. If v2,1 is a positive value neuron on our input, then weights of edges from v2,1 to an increment neuron should only increase and weights of edges from v2,1 to a decrement neuron should only decrease. Similarly, If v2,1 is a negative value neuron on our input, then weights of edges from v2,1 to an increment neuron should

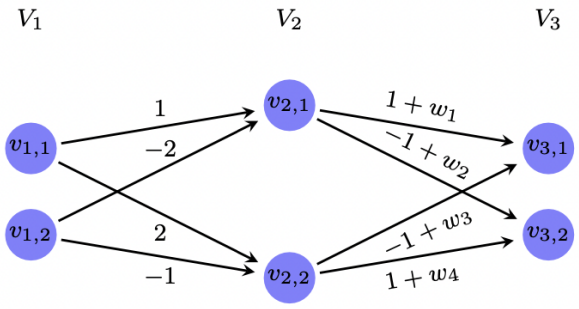


Fig 1: Modifications in last layer.

only decrease and weights of edges from v2,1 to a decrement neuron should only increase. This is summarised below, in table 1. Using this approach, we restricted the possible

Table 1.1: Assignment of lower and upper bounds to epsilon.

| **Neuron in layer (l-1)** | **Neuron in layer (l)** | **Lower bound, Upper bound for epsilon.** |
|---|---|---|
| Positive | Increment | [0, inf] |
| Positive | Decrement | [-inf, 0] |
| Negative | Increment | [-inf, 0] |
| Negative | Decrement | [0, inf] |
| 0 | Increment | [0, 0] |
| 0 | Decrement | [0, 0] |

values epsilon can have and thus reduced the time taken to find a minimal modification. This problem was modelled in Marabou and Gurobi.

### 1.2.2 Finding Minimal modification across all layers in the Network:

Another way of finding minimal modifications, is to consider the entire network and find the edges which can be modified such that the output of the network changes and the modification remains minimum. This approach requires us to model the entire Deep neural network as a optimization query. Marabou does not support putting constraints on the product of two Marabou variables, so this approach could not be implemented in Marabou.
It is implemented using Gurobi and Z3. However, Z3 did not scale and could not produce an output(SAT/UNSAT) after running for 24 hours. Gurobi produced a minimal modification. The objective used, was not only to minimize the total change introduced in the network(sum of the absolute values of all epsilons) but to minimize the change introduced in a particular edge weight as well. This ensures that the overall network

does not change drastically and also the modifications done to each edge weight remains minimum.
A Deep Neural Network can have several layers where each layer can then have multiple neurons and this can result in a large number of epsilons which we will have to optimize. To control this, we add/subtract epsilons only to a certain category of edge weights. The edges which qualify this category should have one or more of the properties listed below:

1. Weight of the edge should be greater than the average weight of all edges in the network.
2. Value of neuron on the given input should be greater than average of all neurons in the network.

The above criteria reduces the number of values to be optimized and helps Gurobi to find a better solution.

## 1.3 Example

Tests were conducted on MNIST and ACAS-Xu Networks. This section discusses the implementation with an example using the approach mentioned in 1.2.2 for correcting a fault in trained ACAS-Xu Network. The network has 1 input layer, 6 hidden layers and 1 output layer. Each hidden layer has 50 neurons and the input and output layers have 5 neurons each. The network has 13,000 edges which can be modified to correct a fault.
This network is known to mispredict on a certain input. Our objective was to correct this misprediction such that on the given input, the network assigns Class 0 or Class 1. Out of the 13,000 edges, the tool modified 14 edges:

1. 2 between layer 0 i.e Input Layer and layer 1.
2. 4 between layer 2 and layer 3.
3. 2 between layer 3 and layer 4.
4. 1 between layer 4 and layer 5.
5. 5 between layer 6 and layer 7 i.e Output Layer.

For this example, the following observations were made:
Input: [-0.2677621, -0.12416982, -0.06393118, 0.375, -0.01954727]

Output vector for original network: [-0.010152077638595547, -0.015996924597676286, 0.02646589423200745, -0.018612992745719894, 0.035624006665884264]

Output vector for modified network: [0.03662438, 0.03662371, 0.0264659, -0.018613, 0.03562403]

The total modification done in the network was: 0.01101
And the maximum modification done in any edge was: 0.0028
The original network assigned class 4 to the input while the modified network assigned class 0 to the input.

## 1.4 Results

Using the existing implementation which add/subtracts values from every edge weight in the last layer, 24 modifications were required. The total change required in the network was 0.01143 and the maximum modification done in any edge was 0.00495.
Using the existing implementation discussed in section 1.2.2, 14 modifications were required. The total change required in the network was 0.01079 and the maximum modification done in any edge was 0.00283.
Using the existing implementation discussed in section 1.2.1, 6 modifications were required. The total change required in the network was 0.00569 and the maximum modification done in any edge was 0.001896. The number of modifications required in terms of the number of edges which had to be modified, maximum modification done in any edge and the total change required in the network remained least in approach discussed in 1.2.1.

# 2 Problem-2: Adversarial Image Generation

This section discusses the problem of finding adversarial images for a given Deep Neural Network and a set of Input images. The problem is to model the DNN as an optimization query and find the minimum change required in an input image such that the DNN gives a wrong output.

## 2.1 Implementation

This problem has been implemented as a separate tool which takes a Deep Neural Network, a dataset of images and produces a dataset containing adversarial images. It minimizes the overall change done in the input image which preserves it's naturalness and also minimizes the change done in any individual pixel value which reduces the L-inf distance between the input image and generated adversarial image.

## 2.2 Example

For MNIST Network, the Input vector is a 1-D array of 784 pixels, where each pixel ranges between 0 and 1. The tool modified 47 pixels out of 784. Each modification was below 0.1. The output label for original image was 4 and output label for adversarial image was 6. The total change introduced in the image was 4.64 and the L-inf distance between original and adversarial image was 0.1.

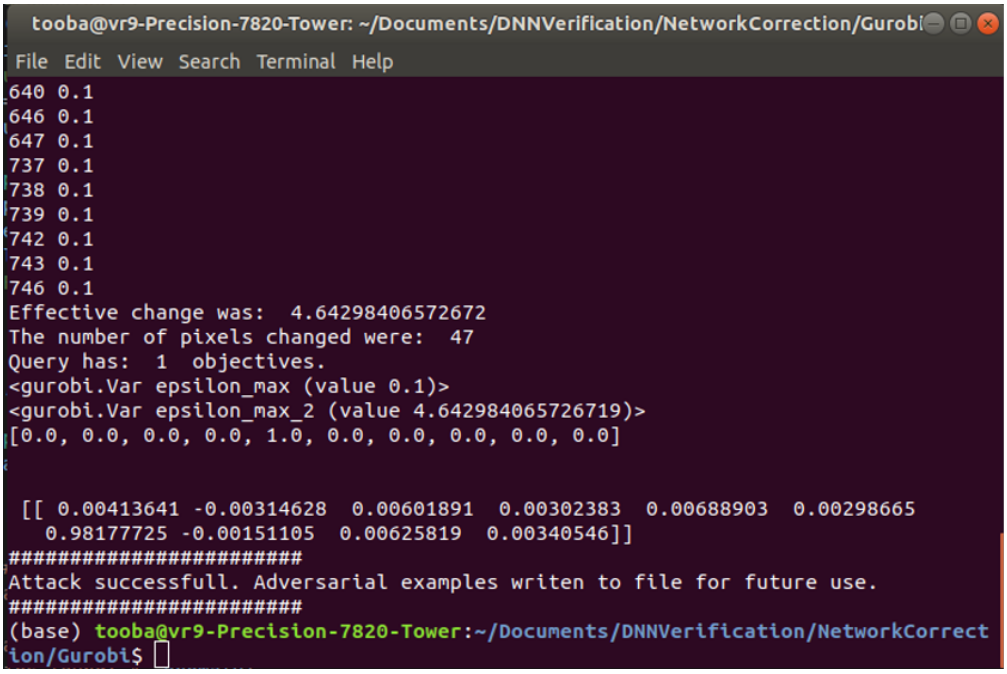


Fig 2 : Adversarial image generation.

## 2.3 Results

This approach was tested on a deep neural network trained on MNIST dataset. The dataset has 60,000 images where each image is of size 28*28. The tool could successfully attack 70.21% of these images(42126/60000). On an average, only 40 out of 784 pixel values had to be changed to get a misprediction. Out of the 42126 images on which the attack was successful, only 75.14% of the images were found to be actually adversarial i.e for 31,654 images, the network actually gave a wrong output. The tool took 4.5 hours to produce adversarial images for 60,000 input images.

# 3 Limitations

Z3, Marabou and Gurobi have been used for all the implementations. As discussed above, Marabou does not support putting constraints on the product of two Marabou variables, so it can not be used to model a Deep Neural Network. Z3 is slow and does not terminate in 24 hours. Gurobi is a LP solver and does not allow implementation of ReLU activation function. Hence, this implementation restricts each node of the network to a particular phase based on prior knowledge. This means that for a neuron whose value is in the positive phase, it's value can only increase and for a neuron whose value is in the negative phase, it's value will not be changed.
This limitation gives rise to the question whether a better solution is possible if we could have allowed the neurons to cross the negative-positive boundary by modelling ReLU activation function.