CS781 Project Report

Kalvala Vivek (210050077)
Savaram Divakar Sai (210050143)

November 17, 2024

1 Introduction

Problem Statement and Relevance: In the context of verifying neural networks for image classification tasks, tools such as Alpha, Beta-CROWN, and DeepPoly are commonly used to assess the robustness of networks against input perturbations. These verifiers typically evaluate the network's performance across all possible perturbations of the input image, ensuring its resilience against adversarial attacks.

To address this, we aim to restrict the perturbation model such that only a $k \times k$ patch of the image is perturbed by a disturbance of magnitude δ, where k and δ are defined inputs. The challenge lies in modifying the input modeling or adjusting the verifier's codebase to efficiently handle this restricted perturbation without resorting to computationally expensive brute-force methods.

This problem has significant relevance in real-life applications. In domains like autonomous driving, healthcare diagnostics, and security systems, localized attacks such as adversarial patches pose a tangible threat. For instance, a small manipulated patch on a stop sign could mislead an autonomous vehicle's perception system or an adversarial perturbation on a medical image could cause diagnostic errors. Ensuring that neural networks are robust to such localized disturbances is crucial for their safe and reliable deployment. By addressing this challenge, the work not only enhances the practical utility of robustness verifiers but also contributes to the broader goal of building trustworthy AI systems.

2 Choice of Tool and Its Limitations:

For the verification process, we utilized the Alpha-Beta-CROWN tool. While this tool is effective in evaluating the robustness of neural networks against perturbations applied across the entire image, it lacks the capability to specifically assess the impact of localized perturbations restricted to a $k \times k$ square region. This limitation poses a challenge in scenarios where the focus is on analyzing robustness against targeted, localized adversarial disturbances.

We are now working on modifying the existing model to enable the evaluation of localized perturbations confined to a $k \times k$ region. This adaptation aims to address the current limitation by allowing the verification process to specifically assess the robustness of neural networks against small, targeted disturbances within a restricted area of the input image.

3 Our Approach

We are aiming to reduce the number of calls to the $\alpha\beta$-CROWN verifier by eliminating the need to check all $(n-k)^2$ potential locations of the localized $k \times k$ perturbation. Instead of evaluating every possible position where the perturbation could occur, we seek to optimize the process and focus only on the most relevant or critical regions, thus minimizing computational overhead.

Claim: Let there be an $m \times m$ sub-patch of the image, where $m > k$. If this sub-patch is perturbed by δ and the neural network classifier produces the same output as the original image, then all $(m-k)^2$ possible $k \times k$ sub-patches within this $m \times m$ region are guaranteed to produce the same output when perturbed by δ.

Using the claim, we begin by checking the entire image (i.e., when $m = n$) with a perturbation of magnitude δ. If the neural network produces the same output after applying the perturbation, we conclude the process. If the results differ, we reduce the size of the region to $m = n/2$ and repeat the verification for 9 overlapping $n/2 \times n/2$ squares.

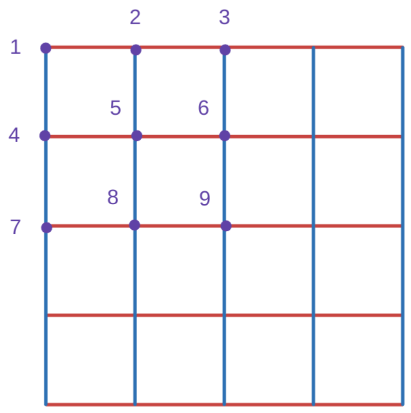

Figure 1: 9 $n/2 \times n/2$ squares in a single $n \times n$ square

Consider the starting points of the 9 squares, each of size $\frac{n}{2} \times \frac{n}{2}$, as indicated by the positions in the figure above. Each numbered point represents the top-left corner of one of these smaller squares. From each of these starting points, the corresponding $n/2 \times n/2$ squares can be observed, as shown in the figure.

Among these squares, some may produce the same result as the original image. We discard those squares and only consider the ones that yield different results. This process is repeated for each iteration, where $m = \frac{n}{2^i}$ and $m > 2k$, with i representing the current iteration step. The process continues until $m = 2k$.

When $m = 2k$, we start by checking whether the perturbation of this $2k \times 2k$ region changes the output. If the output remains the same, we discard the region. If the output changes, we continue the verification process by checking all $k \times k$ squares within this $2k \times 2k$ region. This method is both sound and complete.

4 Main Implementation

To apply the perturbation to a $m \times m$ region at the position (x, y) of an $n \times n$ image, the bounds for each pixel are calculated as follows: For each pixel in the region, we compute the range by taking input_image $- \epsilon$ and input_image $+ \epsilon$. Specifically, instead of using a global perturbation, we calculate the bounds as input_image $- \epsilon \cdot A$ and input_image $+ \epsilon \cdot A$, where A is an $n \times n$ matrix. In this matrix, all elements are zeros except for the $m \times m$ region starting at (x, y), which consists of ones. This localized perturbation allows us to apply changes only to the desired sub-region of the image.

For implementing the above algorithm, a basic limitation arises when n is not a power of 2. If n is not of the form 2^{α}, the implementation of this method becomes difficult. To address this issue, we first find the closest m such that $2^{\beta} \times k = m$ and $m < n$, where m is the modified size of the image.

After determining the value of m, we can identify four $m \times m$ sub-squares within the $n \times n$ image. These four $m \times m$ squares are positioned in each of the four corners of the image. By verifying these four $m \times m$ sub-squares, we can verify the entire $n \times n$ image. If we visualize the $n \times n$ image and identify these four $m \times m$ regions, it becomes clear that these four regions together encompass the whole $n \times n$ image. Therefore, verifying these four $m \times m$ sub-images separately is equivalent to verifying the entire $n \times n$ image.

We have also maintained a matrix to track the verification status of each $m \times m$ region. When a $m \times m$ region is verified and produces the same output as the original image, we update the matrix accordingly. If a neighboring $m \times m$ region produces an error, and we need to verify all the 9 $\frac{m}{2} \times \frac{m}{2}$ squares within that region, we check the matrix to see if any of these smaller squares have already been verified. If a smaller square has already been verified as part of a neighboring $m \times m$ region that produced the same result, we can skip the re-verification of these squares, saving computational resources and reducing redundancy in the verification process.

5 Modifications to Files and Functions

For the implementation of the functionality of the above parts, we have modified two files: `abcrown.py` and `specifications.py`.

In `abcrown.py`, we made changes to the main function within the class `ABCROWN` to accommodate the new verification process. In `specifications.py`, we altered the `construct_vnnlib` function and introduced a custom function called `custom_construct_vnnlib`, in which we implemented the $m \times m$ region perturbation. This function is responsible for applying the perturbation to the specified $m \times m$ regions within the $n \times n$ image.

Additionally, we made minor changes in `loading.py`, specifically in the `parse_run_mode` function, which we renamed to `custom_parse_run_mode`. However, this change does not implement any algorithmic modifications; it was made primarily to facilitate easier code writing.

6 Installation and Running

α, β-CROWN is tested on Python 3.11 and PyTorch 2.2.1. Lower versions, including Python 3.7 and PyTorch 1.11, may also work. It can be easily installed into a conda environment. If you do not have conda, you can install miniconda.

The same setup works for our repository as well. Follow the steps below to clone the repository and set up the environment:

- Clone the repository and setup the conda environment:

```
https://github.com/Aaavek/modified-abcrown/blob/main/complete_verifier/modified_ab.py

# Remove the old environment, if necessary
conda deactivate; conda env remove --name alpha-beta-crown

# Install all dependencies into the alpha-beta-crown environment
conda env create -f complete_verifier/environment.yaml --name alpha-beta-crown

# Activate the environment
conda activate alpha-beta-crown
```

7 Running the Code

To run the main implementation, use the script `modified_ab.py`. The following steps outline how to run the code:

- Activate the conda environment and navigate to the `complete_verifier` directory:

```
conda activate alpha-beta-crown
cd complete_verifier
```

- Run the script with the required arguments:

```
python modified_ab.py --config <path-to-config> --k <value-of-k>
--imagepath <path-to-image> --epsilon <eps> --device cpu
```

- Description of the command-line arguments:
  - `--config <path-to-config>`: Path to the configuration file.
  - `--k <value-of-k>`: Integer for perturbation region size (must be smaller than image size).
  - `--epsilon <eps>`: Value of epsilon for perturbations (can also be in config file).
  - `--device cpu`: Specifies the device for running the model (remove if using GPU).

The configuration file should be formatted in YAML. Below is an example configuration file, `mnist_cnn_a_adv.yaml`:

```
model:
  name: mnist_cnn_4layer
  path: models/sdp/mnist_cnn_a_adv.model
  data_min: 0.0
  data_max: 1.0
data:
  dataset: None
specification:
  epsilon: 0.7
  norm: .inf
attack:
  pgd_restarts: 50
solver:
  batch_size: 1024
  beta-crown:
    iteration: 20
bab:
  timeout: 10
```

In the configuration file, the parameters `data_min` and `data_max` represent the minimum and maximum possible values for each pixel in the input image. For example, these values can be 0.0 and 255.0 for RGB images, depending on the model used.

`model:path` specifies the path to the model to be used.

The `epsilon`: can be provided either in the configuration file or through the command line.

The `bab:timeout` parameter allows modification of the timeout value to optimize the running speed of the model during a particular step.

8 Experimental Results and Analysis

We have analyzed the verifier count for a 28×28 image of the digit "1" from the MNIST dataset under varying ϵ values. For $\epsilon = 0.1$ and $\epsilon = 0.2$, the verifier count remained consistently at 4. These results are excluded from the table below for brevity, as they demonstrate that small ϵ values have minimal impact on the verifier count. The table instead focuses on summarizing the results for $\epsilon \geq 0.3$.

ϵ	Verifier Count (`bab:timeout` = 3s)	Verifier Count (`bab:timeout` = 10s)
0.3	4	4
0.31	40	4
0.32	40	40
0.34	44	40
0.36	84	44
0.38	196	64
0.4	246	121

Table 1: Verifier count comparison for varying ϵ with `bab:timeout` values of 3s and 10s.

By the brute force method, the number of verifier calls required would be 576. However, our approach aims to significantly reduce this count. From the table, it can be observed that for $\epsilon = 0.3$, the verifier count is 4, which is a substantial reduction compared to brute force. As ϵ increases, the verifier count rises steadily: for $\epsilon = 0.36$, the count reaches 84, and for $\epsilon = 0.38$, it increases to 196. At $\epsilon = 0.4$, the verifier count becomes 246. These results demonstrate that while our method reduces the verifier count significantly for smaller ϵ, the efficiency decreases as ϵ grows. Beyond certain ϵ values, further optimization becomes ineffective as the number of calls approaches or exceeds the brute force count. Typically, ϵ values are small in practical scenarios, making our method highly effective within this range.

We have also analyzed the verifier count by varying the value of k while keeping $\epsilon = 0.35$ and the BAB timeout fixed at 5 seconds.

The following table shows the verifier count for various values of k. As observed, for small values of k such as 2, 4, 5, and 7, as k increases, the verifier count also increases. However, this increase is not monotonic. Consequently, as k grows, the efficiency of the modified verifier tends to decrease. .

Value of k	**Verifier Count (VC)**	**Brute Force VC**	**Status**
2	13	729	Safe
4	13	625	Safe
5	40	576	Safe
7	94	484	Safe
11	25	324	Timeout
12	11	289	Timeout
13	10	256	Timeout

Table 2: Verifier count analysis for varying k values with $\epsilon = 0.35$ and bab:timeout = 5.

For medium values of k, such as $k = 11, 12, 13$, the verifier results in a timeout. This behavior indicates a limitation of the current implementation and suggests that future improvements are needed to handle such cases efficiently.

For high values of k, where $k > \frac{n}{2}$, our algorithm is unable to be applied effectively, resulting in no significant improvement in efficiency. Thus, there is a clear need for optimization when dealing with values of k that approach or exceed half the image size.

9 Future Work

A key area for future work is optimizing the verification process for medium and high values of k, where the current implementation faces challenges such as timeouts. This can be achieved by refining the algorithm to better handle larger perturbation regions, potentially through the use of parallelization, efficient data structures, or adaptive region sizing techniques that adjust based on the image's characteristics. These improvements would allow the verification process to scale more effectively with larger perturbations and more complex models.