ERAN User Manual

Gagandeep Singh
Mislav Balunovic
Anian Ruoss
Christoph Müller
Jonathan Maurer
Adrian Hoffmann
Maximilian Baader
Matthew Mirman
Timon Gehr
Petar Tsankov
Dana Drachsler Cohen
Markus Püschel
Martin Vechev

ETH zürich

Abstract

Neural networks are becoming more and more important. They are applied in many real-world applications, such as speech recognition [Hinton et al. 2012], medical diagnostic [Al-Shayea 2011], and autonomous driving [Bojarski et al. 2016] among other fields. Thus it is critical that neural networks behave reliably. This has led to the need to verify the safety of these networks. A number of efforts have been made in this direction, from less scalable but complete verifiers based on SMT solving [Katz et al. 2017, Ehlers 2017, Bunel et al. 2017], mixed-integer linear programming [Tjeng and Tedrake 2017], or Lipschitz optimization [Ruan et al. 2018] to more scalable but incomplete verifiers based on abstract interpretation [Gehr et al. 2018, Singh et al. 2018, Singh et al. 2019a], duality [Dvijotham et al. 2018, Kolter and Wong 2017], semidefinite programming [Raghunathan et al. 2018, Dvijotham et al. 2019] or linear relaxations [Weng et al. 2018, Zhang et al. 2018, Boopathy et al. 2019]. All of the above works require specific network formats, which limits usability.

The Secure, Reliable, and Intelligent Systems Lab (SRI) at ETH aims to provide all of its state-of-the-art research on neural network verification in one tool with the ETH Robustness Analyzer for Neural Networks (ERAN). The goal of this work is to extend ERAN for handling more diverse network formats and architectures and add testing support to increase the trust in the verification process.

This thesis describes the newest version of the ERAN, which offers complete and incomplete verification methods for feedforward, convolutional and residual neural networks. ERAN can analyze ONNX and TensorFlow models directly, making it accessible and easy to use.

Contents

List of Figures

List of Tables

1

Introduction

ETH Robustness Analyzer for Neural Networks (ERAN) [Singh et al. 2020a] is a state-of-the-art precise and scalable verifier for the safety properties of neural networks. The properties include proving robustness against adversarial perturbations based on changes in pixel intensity [Singh et al. 2018, Singh et al. 2019a], geometric transformations of images [Singh et al. 2019a, Balunovic et al. 2020], and more. The verifier supports both exact and approximate verification of feedforward, convolutional, and residual networks based on abstract domains [Singh et al. 2018, Singh et al. 2019a] and Mixed Integer Linear Programming (MILP) solvers [Singh et al. 2019b]. ERAN is also the only verifier that is sound with respect to floating-point arithmetic. Figure 1.1 provides an overview of the capabilities of ERAN.

1.1 Structure of this Document

This document is structured as follows:

Chapter 2 provides the required background information for understanding the rest of the thesis.

Chapter 3 addresses the related works about neural network analysis frameworks.

Chapter 4 describes all options that are available in ERAN, their possible values and their interactions with each other.

Chapter 5 describes the implementation of the ERAN.

Chapter 6 describes the testing framework and how to use it.

Chapter 7 concludes the project and summarizes shortly the most important points. Future works will be proposed for further investigation.

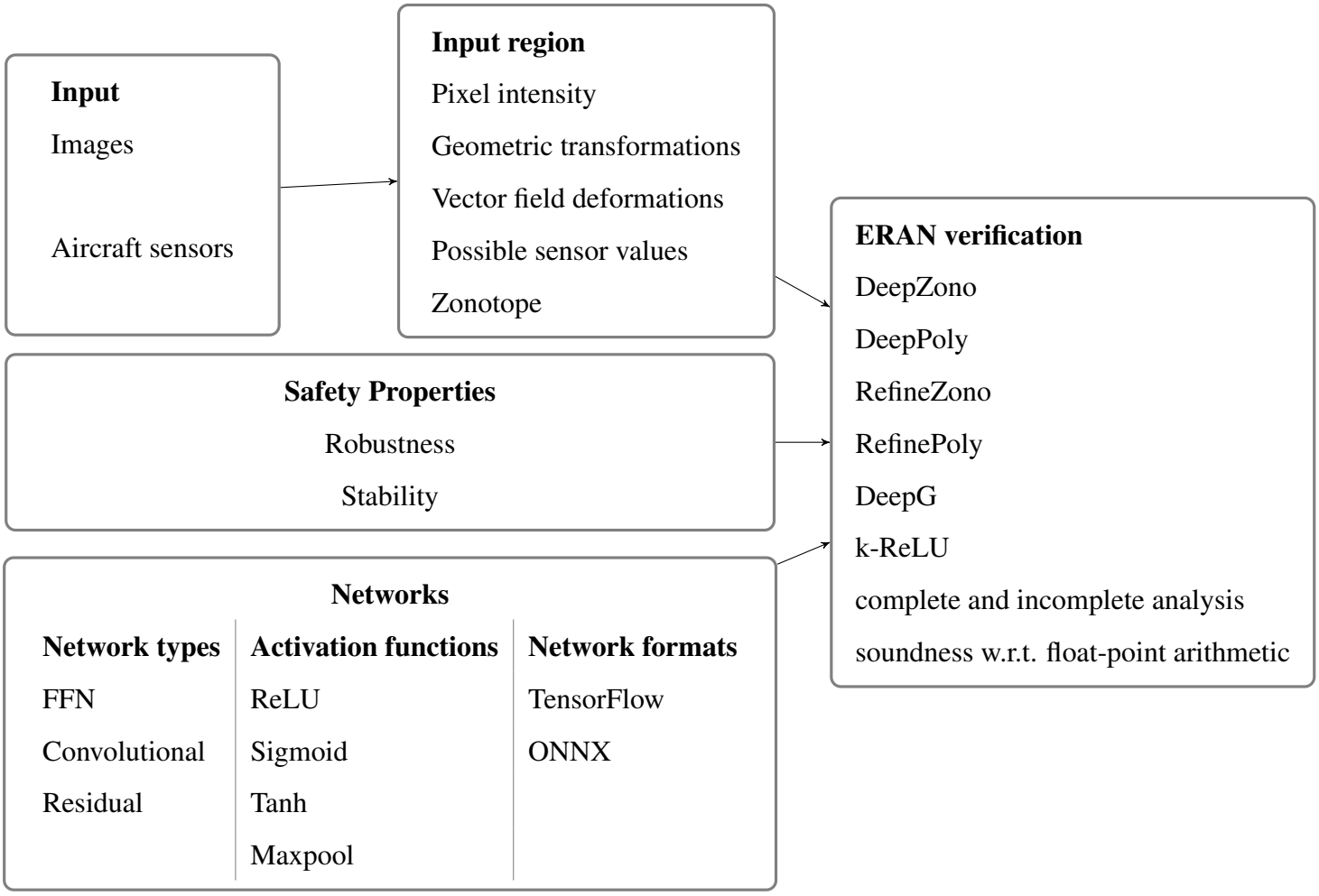

Figure 1.1: *ERAN network verification overview*

1.2 Conventions

Next, we introduce conventions made for the sake of readability and briefness of the complete document. This document adheres to the following conventions throughout:

- To increase readability, *italic* writing will be used for all names that are in the code.
- Nodes in the TensorFlow and ONNX graphs will be called nodes and nodes in the analyzer deep nodes.

2

Background

This chapter will look at the following building blocks of ERAN:

1. The abstract domains are the mathematical basis for network analysis.
2. The input formats and their intricacies.

2.1 Abstract domains

Abstract domains allow us to statically reason about the safety properties of neural networks. Using abstract domains, ERAN can prove that a neural network behaves as desired for all possible inputs within a convex set. These sets are in the form of domain-specific constraints.

2.1.1 DeepZono

DeepZono [Singh et al. 2018] is a specialized implementation of the Zonotope abstract domain tailored for handling neural network operations. The Zonotope domain uses affine arithmetic to keep track of relations between the values of neurons with respect to a convex input set. For a neuron x, the corresponding affine form x̂ is given by:

$$\hat{x} = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \eta_i \tag{2.1}$$

where $(\alpha_i^x)_{0 \leq i \leq n} \in \mathbb{R}$ and $(\eta_i^x)_{0 \leq i \leq n} \in [-1, 1]$.

The interval bounds for x can be computed as:

$$[\alpha_0^x - \sum_{i=1}^{n} |\alpha_i^x|, \alpha_0^x + \sum_{i=1}^{n} |\alpha_i^x|] \tag{2.2}$$

2.1.2 DeepPoly

DeepPoly [Singh et al. 2019a] is a specialized domain tailored for neural network analysis and can produce more precise and scalable results than DeepZono. The DeepPoly domain associates two polyhedral constraints $a_x^{\leq}, a_x^{\geq}$ and interval bounds $[l_x, u_x]$ with each neuron x where $l_x, u_x \in \mathbb{R}$. The polyhedral constraints relate the neuron x to the neurons in preceding layers via their linear combinations. The polyhedral constraints provide lower and upper bounds on the set of values that x can take, i.e., $a_x^{\leq} \leq x \leq a_x^{\geq}$ where $a_x^{\leq}, a_x^{\geq}$ are of the form $v + \sum_i w_i \cdot x_i$ where x_i is a neuron from a preceding layer and $v, w_i \in \mathbb{R}$.

2.1.3 DeepG

The set of images produced after geometrical transformations on images is not usually convex. DeepG [Balunovic et al. 2019] provides a method based on sampling and Lipschitz optimization to compute a convex approximation of the set produced after geometric transformations. To reduce the approximation error, DeepG splits the parameter space of the transformation into smaller pieces and then computes convex approximation for each piece separately. The resulting convex sets are analyzed using the DeepPoly domain.

2.1.4 Vector Field Deformations

Vector field transformations, i.e., moving pixels instead of manipulating pixel values, generally cause large ℓ_p-norm distances between original and deformed images, which makes noise-based distance measures unsuited for certification. Thus, [Ruoss et al. 2020] parametrize vector field deformations by the displacement magnitude, denoted by δ, and the smoothness, denoted by γ. This enables the derivation of convex relaxations that capture all images deformed by smooth vector fields, and thus allows for certification against vector field attacks. These convex relaxations can be efficiently integrated with the DeepPoly domain, the k-ReLU framework, and complete certification.

2.1.5 ELINA

The above mentioned abstract domains are implemented in the ETH Library for Numerical Analysis (ELINA) [Singh et al. 2020b]. The DeepZono (implemented as *zonoml*) and DeepPoly (implemented as *fppoly*) domains in ELINA are sound for arithmetic operations implemented according to the IEEE floating-point standard for 64-bit floating-point numbers. This means that the result from ELINA captures all possible values that the neurons can take under any rounding mode and any order of execution of operations.

2.2 Network format

ERAN can analyze neural networks in ONNX and TensorFlow formats directly. Both formats are based on a computation graph. The nodes of this directed acyclic graph save the operations and parameters needed for the execution. One of the main differences between the two formats is in the representation of tensors. TensorFlow uses NHWC (batch, height, width, channel) while ONNX uses NCHW representation. For a better understanding of the code snippets shown later, this section will show the relevant parts of the model, graph, and nodes of both frameworks.

2.2.1 ONNX

ONNX stands for Open Neural Network Exchange format. It is developed to make moving models between different formats easier. In ERAN it is mainly used to import models trained in PyTorch, but the format is also supported by Caffe2, Microsoft Cognitive Toolkit, MXNet and many more. These frameworks provide an export function that gives an ONNX representation of the model. By adding ONNX support, ERAN can import models from any of these frameworks.

Of the ONNX model, only the field *graph* is relevant for understanding the ERAN code. It is a representation of the execution graph. The *graph* has the fields *node*, *initializer*, *input*, and *output*, which are accessed by the ERAN code. The field *node* is the list of nodes, ordered by input/output dependencies. A node in the list *node* has the fields *input*, *output*, *op_type*, and *attribute*. Both *node.input* and *node.output* are lists of strings representing the names of the node inputs and outputs respectively. Two nodes are connected if a name in *node.output* of one node matches a name in *node.input* of the other. The *op_type* is a string representation of the operation, e.g. "Conv" for convolution. These strings are sometimes different from the ones TensorFlow uses, e.g. "Conv2D". The *attribute* is a list, the content of which is operation dependent. The graphs *initializer* is also a list and contains all named tensors of the model, e.g. weight matrices, etc. At last, the *graph.input* and *graph.output* fields contain the input and output parameters of the graph. To increase the readability, every access of an *input* field in the ERAN code and reference to these fields from here on reads either *graph.input* or *node.input*. The same applies to *output*.

The ONNX operation types encountered in this thesis are shown in Table 2.1.

2.2.2 TensorFlow

TensorFlow is an open-source machine learning framework developed by Google. It was the first model format ERAN supported.

As before, the only relevant part of the TensorFlow model is the *graph*. On the *graph*, ERAN calls two methods *as_graph_def* and *get_operations*. The *as_graph_def* method is used to set the graph in the TensorFlow session. The *get_operations* method returns the list of nodes, similar to the *node* field of the ONNX graph. Every node has the fields *type*, *name*, *inputs*,

Table 2.1: *Relevant ONNX operations*

Name	Description
Constant	Is constant tensor
MatMul	Calculates matrix multiplication
Gemm	Calculates general matrix multiplication, Y = alpha * A' * B' + beta * C
Add	Applies element-wise binary addition
Sub	Applies element-wise binary subtraction
Mul	Applies element-wise binary multiplication
Conv	Applies Convolution with an optional bias addition
MaxPool	Applies max pooling across input tensor, according to kernel, stride, and padding
Relu	Applies the rectified linear function element-wise, y = max(0, x)
Sigmoid	Applies sigmoid function element-wise, y = 1 / (1 + exp(-x))
Tanh	Applies the hyperbolic tangent function element-wise
Gather	Gathers elements from tensor according to indices
Shape	Returns the tensors shape
Reshape	Reshapes tensor
Concat	Concatenates multiple tensors into a single tensor
Unsqueeze	Inserts dimensions of value 1

Table 2.2: *Relevant TensorFlow operations*

Name	Description
Placeholder	The placeholder tensor for the input
Constant	Is constant tensor
MatMul	Calculates matrix multiplication
Add	Applies element-wise binary addition
BiasAdd	Applies element-wise binary addition
Conv2D	Applies Convolution
MaxPool	Applies max pooling across input tensor, according to kernel, stride, and padding
Relu	Applies the rectified linear function element-wise, y = max(0, x)
Sigmoid	Applies sigmoid function element-wise, y = 1 / (1 + exp(-x))
Tanh	Applies the hyperbolic tangent function element-wise

outputs, and *shape* which are relevant to ERAN. Additionally, the *get_attr* method is available to get the attributes needed for the execution of the model. *Type* is the string representation of the operation, like *op_type*. The *name* is comparable to the *node.output* in ONNX. The *inputs* and *outputs* are the lists of input and output nodes. Two nodes are connected in the graph if they are in each other's *inputs* and *outputs*, respectively. The *shape* describes the shape of the tensor after the operation in NHWC format.

The TensorFlow operation types used in this thesis are listed in Table 2.2.

3

Configuring ERAN for network analysis

This chapter describes all the options that ERAN provides for analyzing neural networks. The first section provides an overview of the options providing their names, default values, and a short description. The following sections give a more detailed description of these options. Finally, the last section provides examples of how these options can be used for tuning the neural network analysis with ERAN. ERAN is a generic neural network analyzer and can serve as a backend for a range of network verification tasks. It can be accessed via command-line or a configuration file. The configuration file config.py defines the default option values. If no default value is set for an option, it has to be set to *None*. In the command-line, the option value can be defined by the option name prefixed with a double dash and followed by the value. For flags, the presence of the option name means true and its absence means keeping the default. The command-line option has priority if both the configuration file and command-line define conflicting values for an option.

3.1 Option Overview

Table 3.1 provides an overview of all the options available for analyzing neural networks with ERAN.

Table 3.1: *ERAN Option Description*

Name	Default	Short description
netname	None	The network name, the extension can be .pb, .tf, .pyt, .meta, or .onnx
dataset	None	The dataset, can be either mnist, cifar10, acasxu[1], or fashion[2]
mean	0.5, 0.5, 0.5[3]	The mean(s) used to normalize the data
std	1, 1, 1	The standard deviation(s) used to normalize the data
num_tests	None	Number of the last image to test
from_test	0	From which image to test
debug	False	Whether to display debug info
epsilon	0	The epsilon for L_∞ perturbation
domain	None	Either deepzono, refinezono, deeppoly or refinepoly
input_box	None	The input box(es) with which the output_constraints are verified
output_constraints	None	The constraints verified, ignored if dataset is defined
zonotope	None	The file to specify the zonotope matrix
complete	False	The flag specifying whether to use complete verification or not
timeout_complete	60	The timeout in seconds for the complete verifier
timeout_lp	1	The timeout in seconds for the LP solver
timeout_milp	1	The timeout in seconds for the MILP solver
use_default_heuristic	True	Whether to use area heuristic for ReLU approximation
use_milp	True	Whether to use MILP or not
dyn_krelu	False	Whether to dynamically select parameter k for k-ReLU
use_2relu	False	Whether to use 2-ReLU
use_3relu	False	Whether to use 3-ReLU
refine_neurons	False	Whether to refine intermediate neurons
numproc	#CPU	The number of processes to use for MILP / LP / k-ReLU solver
geometric	False	Whether to do geometric analysis
geometric_config	None	The geometric configuration file location
data_dir	None	The geometric data location
num_params	0	The number of transformation parameters
attack	False	Whether to verify attack images
spatial	False	Whether to do analysis of vector field deformations
t-norm	inf	Vector field norm, can be 1, 2, or inf
delta	0.3	Vector field displacement magnitude
gamma	∞	Vector field smoothness constraint

3.2 Options applicable to all types of analysis

The options described next can be used for all types of analysis.

3.2.1 netname

The *netname* specifies the absolute or relative path to the network file. It must be present for all forms of analysis. The supported network types are TensorFlow (.pb), TensorFlow checkpoint (.meta), a text-based version of TensorFlow (.tf), a text-based version of PyTorch (.pyt), and ONNX (.onnx). To analyze a network trained with PyTorch it has to be either translated into the text-based version by hand or converted to ONNX using the integrated ONNX export of PyTorch.

3.2.2 dataset

For the *dataset*, the values of mnist (MNIST [LeCun and Cortes 2010]), cifar10 (CIFAR-10 [Krizhevsky et al.]), acasxu (ACAS Xu [Manfredi and Jestin 2016]), and fashion (Fashion-MNIST [Xiao et al. 2017]) can be chosen. The acasxu dataset is only available for analysis with box inputs, while fashion is only available for geometric analysis. The dataset defines the size and dimensions of the input to the neural network. Choosing one of these options will load the corresponding dataset and pass the adversarial region through the network. Note that for image-classification networks, the adversarial region is only passed through the network if the original image is first classified correctly by the network. This option is mutually exclusive with the option *zonotope*, but one of them must be given.

3.2.3 mean

This option needs a float for every channel in the input. The *mean*(s) is used to normalize the data. For .pyt networks, this option is ignored, as the normalization is already defined in the .pyt file. The default is 0.5, 0.5, and 0.5, for all datasets, except mnist without *geometric*, where it is 0.

3.2.4 std

This option needs a float for every channel in the input. The *std* values are used as the standard deviation to normalize the data. For .pyt networks, this option is ignored, as the normalization

[1] Only for box inputs.
[2] Only for geometric analysis.
[3] For MNIST dataset and non-geometric analysis, the default is 0.

is already defined in the .pyt file. The default is 1, 1, and 1, for all datasets.

3.2.5 num_tests

If present, the *num_tests* option defines the index of the last image in test set that is analyzed.

3.2.6 from_test

If present, the *from_test* option defines the index of the first image in test set that is analyzed.

3.2.7 debug

If this flag is present, then additional debug output is displayed.

3.3 Box input options

In this section, the options for analysis with box inputs are presented. These options are ignored if *geometric* is true.

3.3.1 epsilon

Epsilon specifies the radius of the L_∞ ball. Note that *epsilon* denotes α_i^{xi} in the affine form of DeepZono input. In DeepPoly the lower and upper bounds for the input are calculated by taking the image and subtracting or adding *epsilon* respectively. This option is also mutually exclusive with the option *zonotope*. If no epsilon is provided, an epsilon of 0 is used for the analysis.

3.3.2 domain

The *domain* can be either deepzono (DeepZono), refinezono (RefineZono), deeppoly (DeepPoly) or refinepoly (RefinePoly). Refinezono and refinepoly build on deepzono and deeppoly respectively. The refining part is done by based on linear programming with triangle ReLU approximation (LP), mixed-integer linear programming (MILP), or k-ReLU approximations to narrow the lower and upper bounds in neurons. The choice of the refine method depends on *use_milp*, *use_2relu*, *use_3relu*, *dyn_krelu*, and *refine_neurons* options. As mentioned in Section 3.4 only DeepZono and RefineZono can be used in combination with *zonotope*. If the *geometric* flag is set, the analysis will be done with the DeepPoly domain and this option is ignored.

3.3.3 intput_box

The *intput_box* defines one or more box inputs to be verified. This option is mutually exclusiv with the option *epsilon*. The format of the input box is as follows: Every line describes a dimension and can have multiple intervals in it. Every interval is desctibed by a opening square bracket followed by the lower bound, a comma, the upper bound and a closing square bracket (e.g. [1.2, 3.4]). The combination of every interval combination will be checked.

3.3.4 output_constraints

The *output_constraints* defines the properties ERAN tries to verify. This option is ignored, if the dataset (and spec_number) is defined. The format for the constraints is: On the first line is the number of output labels, the following lines start with one or more output labels, followed by a constraint. The labels are described by the letter y followed by the label index (e.g. y0 for the first label). The supported constraints are: min (one of the previous labels is the minimum), max (one of the previous labels is the maximum), notmin (this label is not the minimum, this constraint makes only sense for a single label), notmax (this label is not the minimum, this constraint makes only sense for a single label), $<$ or $>$ label (all previous labels are smaller or bigger then following label, only one label can follow $<$ or $>$).

3.3.5 use_default_heuristic

This value can either be true or false and defines whether to use default heuristic for the ReLU approximation for the DeepZono and DeepPoly domains. The default is true.

3.3.6 complete

This value can be either true or false. If true, complete verification is performed when incomplete verification fails. For geometric analysis, the value of this option is ignored. The default is false.

3.3.7 timeout_complete

This option is the timeout in seconds for the complete verifier (above). The default is 60.

3.4 Option for zonotope input (zonotope)

For DeepZono and RefineZono, a zonotope can be used as an input for the neural network. As with *netname*, *zonotope* is the path to the file specifying the zonotope. The *zonotope* file has two integers followed by a number of floats. The numbers can be separated by spaces,

commas, or newlines. The first integer denotes the input dimension (e.g. 784 for MNIST). The second integer is one plus the number of error terms of the zonotope. The number of floats is the two integers multiplied. Looking at an input of i = [x0, x1, ... , xk], the file has all α values of the affine forms following each other. The file would have the format: $k, 1+n, \alpha_0^{x0}, \alpha_1^{x0}, ..., \alpha_n^{x0}, \alpha_0^{x1}, ..., \alpha_n^{xk}$.

3.5 Options for refined analysis

The options in this section can be used in RefineZono and RefinePoly analysis. They are ignored for all other domains.

3.5.1 use_milp

This value can either be true or false. If true, the MILP solver is used otherwise the LP solver is employed. The default value is true.

3.5.2 timeout_lp

With this option, the timeout in seconds used by the linear program (LP) solver can be defined. The default value is 1.

3.5.3 timeout_milp

This option is the timeout in seconds for the mixed-integer linear program (MILP) solver. The default is 1.

3.5.4 dyn_krelu

If the *dyn_krelu* flag is present, the value is true and otherwise, the default is kept. With this option, the analyzer chooses parameter k in the k-ReLU analysis dynamically. This flag can be used together with *use_2relu* and *use_3relu*. The default is false.

3.5.5 use_2relu

If the *use_2relu* flag is present, the value is true and otherwise, the default is kept. The k-ReLU analysis is done with k = 2 if this option is active. This flag can be used together with *dyn_krelu* and *use_3relu*. The default is false.

3.5.6 use_3relu

If the *use_3relu* flag is present, the value is true and otherwise, the default is kept. The k-ReLU analysis is done for k = 3 if this option is active. This flag can be used together with *dyn_krelu* and *use_2relu*. The default is false.

3.5.7 refine_neurons

If the *refine_neurons* flag is present, the value is true and otherwise, the default is kept. Intermediate neurons are refined by calling the solver if this option is active. This flag can be used together with *dyn_krelu*,

3.5.8 use_3relu

,

3.5.9 use_milp

and *use_2relu*. The default is false.

3.5.10 numproc

This defines the number of processes the LP/MILP solver and k-ReLU analysis will run on. The default is the number of processors on the machine.

3.6 Geometric analysis options

This section describes all the options for neural network verification against geometric attacks. Every option here is ignored if *geometric* is false. A look at the DeepG repository [Balunovic et al. 2020] is recommended, for more information and details on *geometric_config*.

3.6.1 geometric

If the *geometric* flag is present, the value is true and otherwise, the default is kept. The geometric analysis is done with the DeepPoly domain and the domain option is ignored. There are two ways to do geometric analysis in ERAN. The first is to pass a geometric configuration (*geometric_config*) and the second is to create the necessary files as described in DeepG and pass the folder location (*data_dir*). For both, the number of transform parameters (*num_params*) has to be given. If the *attack* flag is true, then random attack images will also be tested. The default for *geometric* is false.

3.6.2 geometric_config

This is the path to the geometric configuration file. If this value is present, then *data_dir* is ignored. From this file, the transformed images and attack images are generated.

3.6.3 data_dir

This is the location of the generated files with the transformed images and attack images. If *geometric_config* is present, then this option is ignored.

3.6.4 num_params

This is the number of parameters for the geometric transformation.

3.6.5 attack

This flag is true if it is present and keeps its default value otherwise. With this option, images are passed through the network. These images are generated by taking an original image and adding random noise. The number of images and the amount of noise is defined in the *geometric_config* file or the attack file in the *data_dir* folder.

3.7 Options for Vector Field Analysis

This section describes all the options for neural network verification against vector field attacks. Every option here is ignored if *spatial* is false.

3.7.1 spatial

If the *spatial* flag is present, the value is true and otherwise, the default is kept. The default for *spatial* is false. For certification against non-smooth vector fields ($\gamma = \infty$) any domain can be used. However, certification against smooth vector fields ($\gamma < \infty$) is only compatible with the DeepPoly and RefinePoly domains. Finally, to increase precision, the k-ReLU framework or complete certification can be employed (with the corresponding parameters).

3.7.2 t-norm

The T-norm is the vector field norm that characterizes the pixel displacement magnitude (more details are provided in [Ruoss et al. 2020]). Acceptable values are *1*, *2*, or *inf*.

3.7.3 delta

This option corresponds to the maximum pixel displacement magnitude. Any float larger than zero is an acceptable input, and a value of 1 corresponds to a displacement by one grid length.

3.7.4 gamma

This parameter characterizes the smoothness of the deforming vector field. Any float larger than zero is an acceptable input, but only values smaller than 2δ will effectively constrain the vector fields to be smooth.

3.8 Call examples

This section shows examples of how ERAN can be used for neural network verification. The script to access ERAN is *__main__.py* in the *tf_verify* folder.

As a first use case, the user wants to verify robustness for the network cifarNet.onnx against L_∞ norm perturbations. The dataset is CIFAR-10. The means and standard deviations the network was trained with are 0.5, 0.5, and 0.4 and 0.2, 0.2, and 0.2 respectively.

First, he wants a fast result, so he uses DeepZono with an epsilon of 0.1 as shown in Listing 3.1.

```
$ python3 . --netname /path/cifarNet.onnx --dataset cifar10 --mean 0.5 0.5 0.4
--std 0.2 0.2 0.2 --domain deepzono --epsilon 0.1
```

Listing 3.1: *Run DeepZono analysis*

For a more precise result, he uses RefinePoly, without MILP, but with area heuristic and k-ReLU. For k-ReLU, he runs the analysis with k = 2 and a dynamically chosen k. Listing 3.2 shows the command to run this analysis.

```
$ python3 . --netname /path/cifarNet.onnx --dataset cifar10 --mean 0.5 0.5 0.4
--std 0.2 0.2 0.2 --domain refinepoly --epsilon 0.1 --use_milp false
--use_area_heuristic true --dyn_krelu --use_2relu
```

Listing 3.2: *Run RefinePoly analysis*

Now he has the assumption that changes in pixels are completely dependent. Meaning if there is a change in the red channel the same change is affecting the green and blue channel. He can use the *zonotope* input for that. First, he takes a CIFAR-10 image and creates a zonotope out of it. The following Listing 3.3 shows a mock example of such a script.

```
def create_cifar_zonotope(image, epsilon):
    normalize(image)
    # image is in NHWC format
    channel_length = len(image)/3
    zonotope = []

    # write input size and 1 + number of dimension of the zonotope
    zonotope.append(len(image))
    zonotope.append(1 + channel_length))
    for i in range(channel_length):
        # Calculate alpha 1 though 1024
        alphas = [epsilon if i == j else 0 for j in range(channel_length)]

        # Prepend pixel value as alpha 0
        red = [image[i * 3]] + alphas
        green = [image[i * 3 + 1]] + alphas
        blue = [image[i * 3 + 2]] + alphas

        # Add alphas to zonotope
        zonotope.extend(red)
        zonotope.extend(green)
        zonotope.extend(blue)
    save(zonotope)
```

Listing 3.3: *Script for creating zonotope*

Of course, the analysis would now include values outside of $[0, 1]$ for a color, where only values inside have meaning. This should be corrected before the zonotope is utilized. For a working example of a zonotope-creation script see *create_zonotope.py* in folder *data*.

Listing 3.4 shows the command to run the analysis for the zonotope input with RefineZono, the refinement is performed with MILP, and 3-ReLU.

```
$ python3 . --netname /path/cifarNet.onnx --zonotope /path/zonotope
--domain refinezono --use_3relu
```

Listing 3.4: *Run Zonotope analysis*

The next example shows verification against geometric perturbations. For this, the user needs a geometric config file to generate the transformed images. Two examples are provided in the folder *deepg/code/examples*. An example geometric configuration file is shown in Listing 3.5.

```
dataset          cifar10
noise            0
chunks           1
 inside_splits   500
method           polyhedra
 spatial_transform  Rotation(-1,1)
num_tests        100
ub_estimate      Triangle
num_attacks      20
poly_eps         0.0001
num_threads      20
max_coeff        20
lp_samples       1000
num_poly_check   50
set              test
```

Listing 3.5: *Geometric configuration file*

The reason for the redundant information, e.g. cifar10, is so the same geometric configuration file can be used for generating the files containing the transformed and attack images. The command to run the *geometric* analysis with additional attack images is shown in Listing 3.6.

```
$ python3 . --netname /path/cifarNet.onnx --dataset cifar10 --geometric
--geometric_config /path/config.txt --num_params 1 --attack
```

Listing 3.6: *Run geometric analysis*

If the *attack* flag is not set, then *num_attacks* is treated as 0. While the *geometric_config* option is the more convenient way to do geometric analysis, for repeated use of the same configuration file, it is faster to first generate the files and then read them repeatedly. The same is also suggested for reproducibility.

After generating the files, Listing 3.7 shows a call to run geometric analysis by reading the files.

```
$ python3 . --netname /path/cifarNet.onnx --dataset cifar10 --geometric
--data_dir /path/ --num_params 1 --attack
```

Listing 3.7: *Run geometric analysis by reading files*

Listing 3.8 shows the command for replicating the complete verification of Acas Xu of the paper [Singh et al. 2019b]. The dataset is Acas Xu and the network acasxu_prop9_net.tf is provided in the folder *data/acasxu/nets* of ERAN.

```
$ python3 . --netname ../data/acasxu/nets/acasxu_prop9_net.tf --dataset acasxu
--domain deepzono --complete True
```

Listing 3.8: *Complete verification of a Acas Xu network*

The next two examples show certification against vector field deformations. Recall that we can certify both smooth ($\gamma < \infty$) and non-smooth ($\gamma = \infty$) vector fields. Listing 3.9 shows the command for incomplete certification of smooth vector fields. Spatial certification is also com-

patible with more precise methods including the k-ReLU framework and complete certification. To invoke these methods, the user can just add the corresponding flags, as can be seen in Listing 3.10 for the complete certification case.

```
$ python3 . --netname ../path/ cifarNet .onnx --dataset cifar10 --domain deeppoly
--spatial --t-norm inf --delta 0.3 --gamma 0.1
```

Listing 3.9: *Incomplete verification of smooth vector fields*

```
$ python3 . --netname ../path/ cifarNet .onnx --dataset cifar10 --domain deeppoly
--spatial --t-norm inf --delta 0.3 --gamma 0.1 --complete True
```

Listing 3.10: *Complete verification of smooth vector fields*

4

Implementation

This chapter describes the implementation of ERAN. The first section gives a top-down view of the pipeline for verifying neural network properties. The following sections describe the different components of ERAN in detail. Figure 4.1 shows the folder structure of the ERAN.

4.1 Pipeline

In the first step, the network is read in by the *__main__.py* script and a TensorFlow or ONNX model is created. Second, *ERAN* class *eran.py* is initialized and the model is passed to a translator, which takes this model and puts it in an intermediate representation (IR). Third, the IR is used by the *optimizer* to create domain-specific nodes. In the end, the *analyzer* passes input through the deep nodes to verify network properties. Figure 4.2 shows a rough overview of how classes interact with each other and where they are defined.

4.2 Reader

For saved TensorFlow and ONNX models, the reader is trivial. It only uses the framework's load methods, since the representation of both already is a directed acyclic graph.

For the TensorFlow files .meta and .pb a TensorFlow session is created with *tensorflow.Session()* and the model graph is loaded into it. Listing 4.1 shows the calls to load the TensorFlow checkpoint files, which have the extension .meta. The TensorFlow Protobuf files, extension .pb, are loaded as demonstrated in Listing 4.2. For ONNX, the method is a literal load method as shown in Listing 4.3.

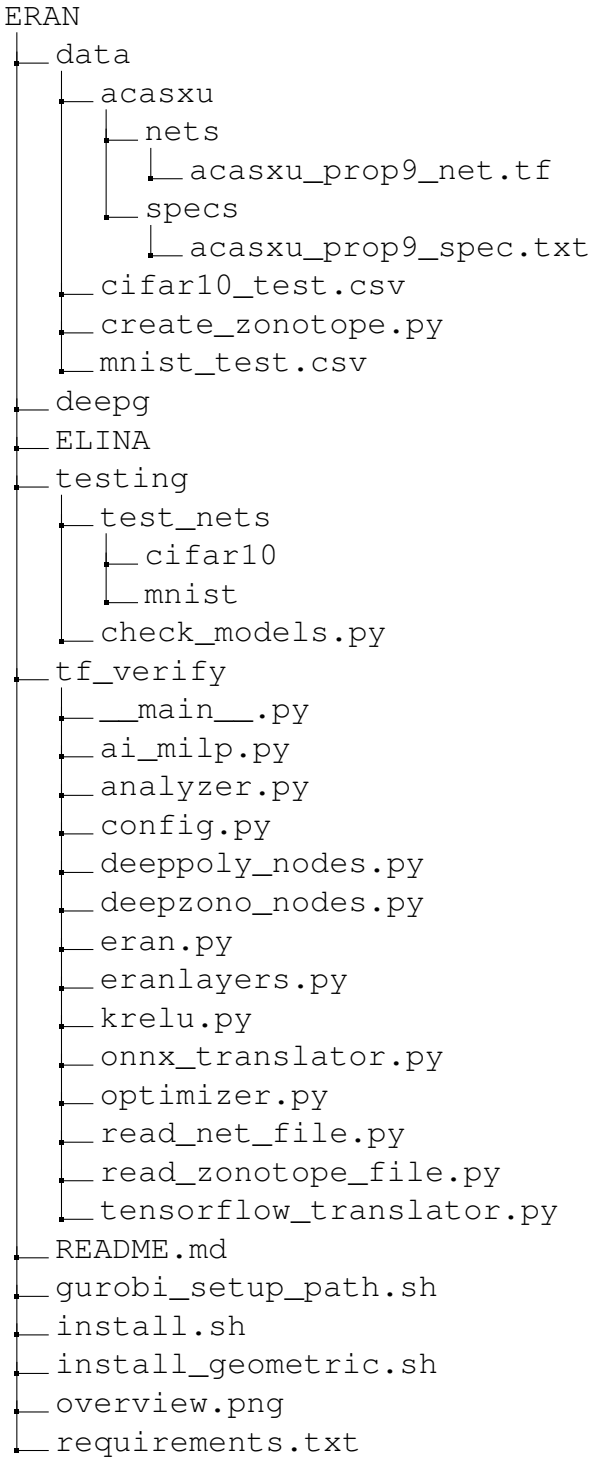

Figure 4.1: *ERANs folder structure*

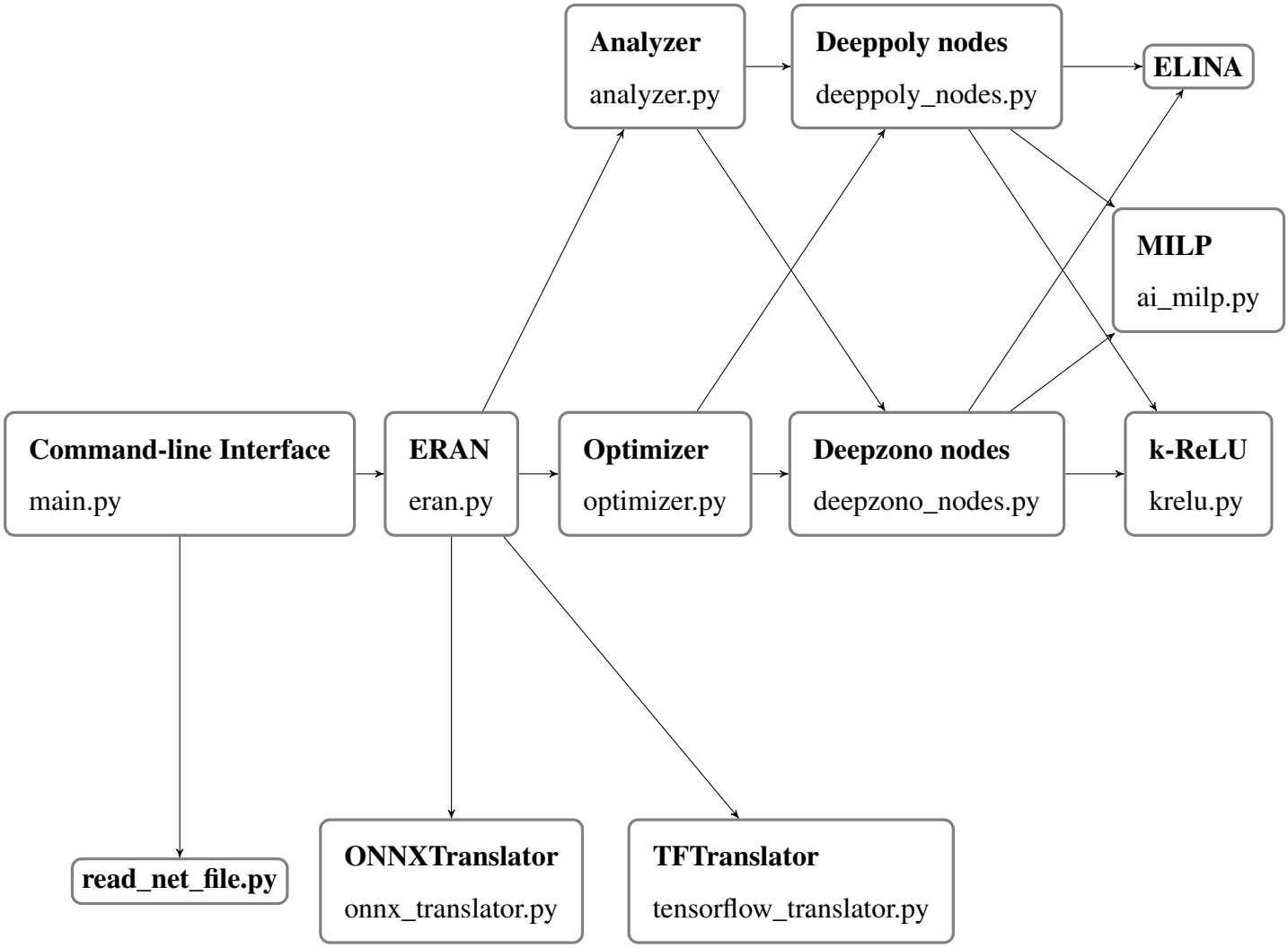

Figure 4.2: *Class Dependency Graph*

```
saver = tensorflow.train.import_meta_graph(netname)
saver.restore(sess, tensorflow.train.latest_checkpoint(netfolder+'/'))
```

Listing 4.1: *Load .meta*

```
graph_def = tf.GraphDef()
graph_def.ParseFromString(f.read())
sess.graph.as_default()
tf.graph_util.import_graph_def(graph_def, name='')
```

Listing 4.2: *Load .pb*

```
onnx_model = onnx.load( net_file )
onnx.checker.check_model(onnx_model)
```

Listing 4.3: *Load .onnx*

For the text-based .tf and .pyt format, the reader translates them into a TensorFlow model and pass it on for further analysis. The .pyt format additionally has mean(s) and standard deviation(s) to normalize the input before passing it through the network. With the new options *mean* (subsection 3.2.3) and *std* (subsection 3.2.4), this can be accomplished without going through this format.

4.3 ERAN

This section explains the implementation of class *ERAN*. First, the initialization method is looked at and, then the method *analyze_box*, where the method *analyze_zonotope* is analogous.

On initialization, ERAN gets a model as an argument. With it, ERAN calls the appropriate translator and gets the IR, which contains *operations* and *resources*. The IR is then used to initialize the *optimizer*. Figure 4.3 shows an overview of the initialization process. At the end of the initialization process, the number of neurons in the network is printed. The number of neurons is defined as the sum of the output size of all activation layers in the network.

The method *analyze_box* takes as input the lower and upper bounds of the input box, the domain, the timeouts for MILP and LP solvers, a boolean determining whether to use area heuristic, a boolean indicating testing mode and six arguments for geometric analysis, that will be ignored in this chapter. ERAN uses this input to get the domain-specific nodes from the *optimizer*. The list of deep nodes is then given to the *analyzer* and analyzed by it. The result is a tuple with the following:

- The class with a higher lower bound than the upper bound of every other class, or minus one if no such class exists.
- The special network representation
- The list of lower bounds
- The list of upper bounds

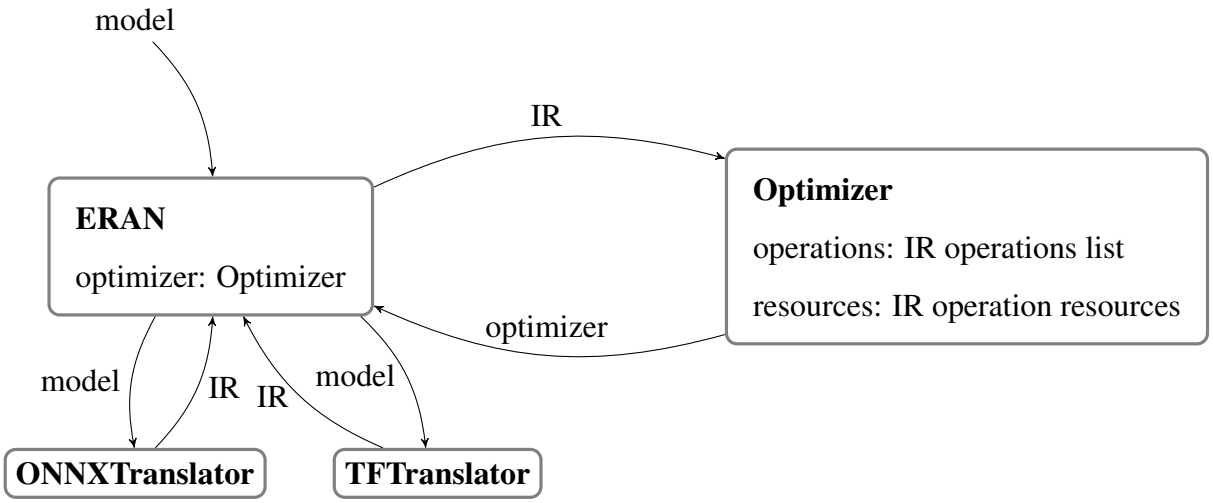

Figure 4.3: *ERAN Initialization*

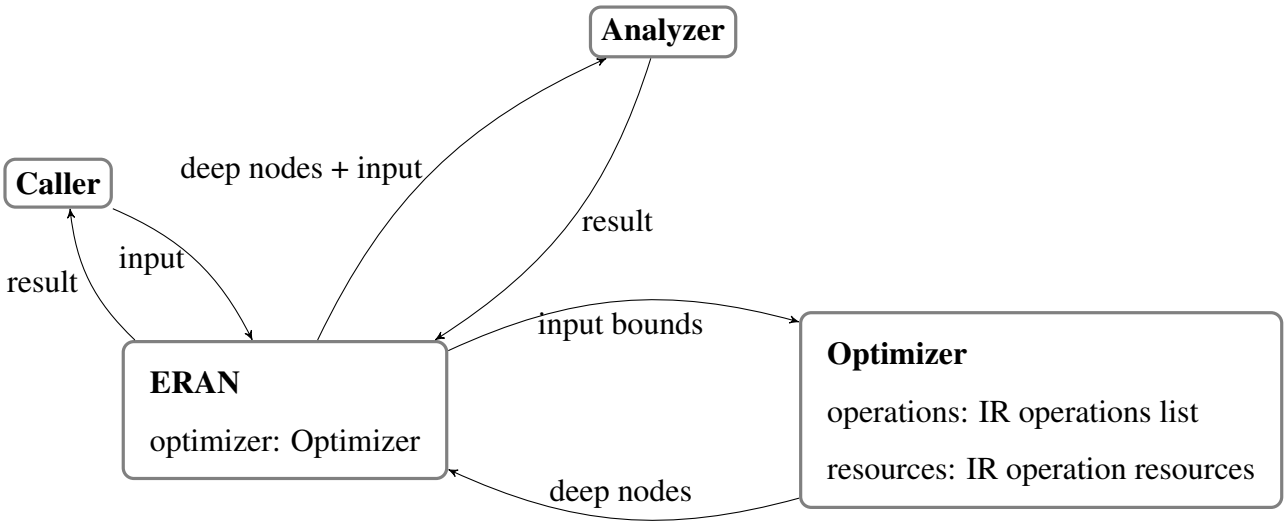

Figure 4.4: *ERAN analyze_box*

- If testing is true, the list of tensor names and shapes

Figure 4.4 illustrates this method.

ERAN creates new deep nodes for every input region. Input node changes are unavoidable, as they encode the input bounds, but reusing the rest is an option. Reusing might be optimal from a performance perspective, but the effect is negligible. The advantage of this approach is that the same instance of ERAN can analyze in all domains. The testing framework makes use of this possibility.

4.4 Translators

The TensorFlow translator *tensorflow_translator.py* and ONNX translator *onnx_translator.py* are similar. Both first prepare the model and then translate it.

Model preparation The model preparation of the TensorFlow translator consists of creating a session and using the TensorFlow *graph_util* to convert variables into constants with *convert_variables_to_constants* and removing training nodes with *remove_training_nodes*.

There are three reasons the ONNX translator has to do more to prepare the model. First, as mentioned in section 2.2, ONNX works with NCHW format while TensorFlow uses NHWC. Due to the TensorFlow legacy, the ERAN IR also uses NHWC. The second reason is that ERAN has to do shape inference on the graph nodes. The last reason is that the nodes in the ONNX model graph are only connected by name and not reference like the TensorFlow nodes. This makes traversing the graph difficult. These problems are addressed in the *prepare_model* method of the ONNX translator.

In the *prepare_model* method, four maps are created. The first map *shape_map* is from a node's output name to its output shape. The second map *constants_map* has also output names as keys and the corresponding constant as value. The third and fourth maps are *output_node_map* and *input_node_map*, respectively. The map from output and input names of a node to the node itself. Those two maps are used to make graph traversal easier.

To get the shapes for the *shape_map*, the built-in shape inference of ONNX could not be used. The reason for this is the common practice in PyTorch to use the operation Reshape(N, -1) to flatten the input. This is translated to Shape, Gather, Concat, Unsqueeze, and Reshape in ONNX. While the output of the Shape operation is a constant, for ONNX it is a tensor and only calculated at runtime. Therefore, the shape inference includes constant propagation, for which the *constants_map* is created. If all inputs to an operation are in the map, the translator calculates the result using the *NumPy* library [Oliphant 2006] and puts it in as well. The calculation of the output shape of every operation is based on the constant inputs and input shapes. In the TensorFlow model, all the shapes are already saved in the nodes.

Listing 4.4 is the matrix multiplication and element-wise operation cases of *prepare_model* and stands as an example of the shape inference and constant propagation implementation.

The conversion from NCHW to NHWC is done every time an array is put into the *constants_map* by the *nchw_to_nhwc* method. This method transposes every array with four dimensions and returns the unchanged array otherwise.

Translation After all required information is collected, the translator then creates a list of operation types *operation_types* and a list with the resources needed for these operations *operation_resources*. This is done by iterating over the list of operations in the *graph*. For TensorFlow, that means calling *get_operations* and for ONNX, accessing the *node* field. The first operation of a TensorFlow model is always Placeholder node, while for ONNX the input is not considered a node. At the start of the ONNX model translation, a Placeholder operation is appended to *operation_types* and *graph.input[0]* is used to get the necessary information to put

```
for node in model.graph.node:
    ...

    # matrix multiplication
    elif node.op_type in ["MatMul", "Gemm"]:

        # Find out if A or B are transposed
        transA = 0
        transB = 0
        for attribute in node.attribute:
            if 'transA' == attribute.name:
                transA = attribute.i
            elif 'transB' == attribute.name:
                transB = attribute.i

        # Calculate shape
        M = shape_map[node.input[0]][transA]
        N = shape_map[node.input[1]][1 - transB]

        # Save shape for value
        shape_map[node.output[0]] = [M, N]

    # element-wise operations
    elif node.op_type in ["Add", "Sub", "Mul"]:
        # Shape stays the same
        shape_map[node.output[0]] = shape_map[node.input[0]]

        # Propagate constant if possible
        if node.input[0] in constants_map and node.input[1] in constants_map:

            # Calculate the resulting constant
            if node.op_type == "Add":
                result = np.add(constants_map[node.input[0]],
                                constants_map[node.input[1]])
            elif node.op_type == "Sub":
                result = np.subtract(constants_map[node.input[0]],
                                constants_map[node.input[1]])
            elif node.op_type == "Mul":
                result = np.multiply(constants_map[node.input[0]],
                                constants_map[node.input[1]])

            # Save constant
            constants_map[node.output[0]] = result

    ...
```

Listing 4.4: *prepare_model excerpt*

in *operation_resources*. Depending on the operation type, the translators handle the node by ignoring it or extracting all necessary information out of it. To ignore a node, the translators put the node in the *reshape_map*. The ignored nodes output name is the key and its input name the value. Later, if another node has an input contained in the *reshape_map*, the input is replaced with the mapped value, the ignored nodes input. The information extraction is of course operation type dependent. The information required from every relevant node is input names, output name, and output dimensions.

For ONNX, there is an interesting mixed case in the operation Reshape. Because of the difference in format, between NHWC in ERAN and NCHW in ONNX, the Reshape operation is relevant. Equations 4.1 and 4.2 illustrate how the Reshape operation is handled in TensorFlow and ONNX format respectively.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \xrightarrow{\text{Reshape}} \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \xrightarrow{\text{Format change}} \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \xrightarrow{\text{Reshape}} \begin{pmatrix} 1 & 3 & 2 & 4 \end{pmatrix} \tag{4.2}$$

The Reshape operation is relevant and treated like a Gather operation, except if a matrix multiplication follows, then the matrix is adjusted to correct the problem and the Reshape is ignored.

The other case, which is not exactly standard is Add. If one of the addends is constant the operation type is Add, but if no addend is constant, then the operation type is Resadd.

As previously stated, every relevant operation needs information on input names, output name, and output shape. Table 4.1 shows for every operation the additionally needed resources. For operations that are ignored, the entry Ignored is used.

Table 4.1: *Resources needed for operations*

Operation	Resources
Placeholder	None
Constant	Ignored without $reshape\_map$ entry
MatMul	The matrix
Gemm	The matrix and the bias
Add	The addend if it is constant, otherwise none[1]
BiasAdd	The bias
Sub	The constant one of minuend or subtrahend and a boolean denoting the choice
Mul	The array with the elements to multiply
Conv	The filters, bias, input shape, strides, padding[2], and kernel shape
Conv2D	The filters, input shape, strides, padding[2]
MaxPool	The input shape, window size, strides, padding[2]
Relu	None
Sigmoid	None
Tanh	None
Gather	The input shape, indexes, axis
Shape	Ignored
Reshape	Ignored, if followed by a matrix multiplication, otherwise like Gather
Concat	Ignored
Unsqueeze	Ignored

4.5 Optimizer

The optimizer is defined in *optimizer.py*. On initialization, the optimizer is given the *operation_types* and *operation_resources* of the translator. The optimizer takes these and creates a list of domain-specific deep nodes *execute_list* and a list of tensor names and shapes *output_info*. The list of deep nodes will be used by the analyzer to verify the network property. The list of tensor names and shapes is for testing purposes. These lists are returned by

[0]In this case, the translation is Resadd.
[0]The padding is divided into vertical and horizontal.

the *get_deepzono* or *get_deeppoly* method. The *get_deepzono* takes 3 arguments, a special representation of the neural network *nn*, the lower and the upper bound of the input. The special representation *nn* is used for the LP or MILP part of the refined analysis. To analyze a zonotope as an input, *get_deepzono* is called without providing the lower or upper bound, but passing the zonotope instead. The *get_deeppoly* method takes ten arguments. The first three are the same as for *get_deepzono*, and the additional ones are all only important for the input of the geometric analysis. To create the list of nodes the optimizer iterates over the *operation_types* and uses *operation_resources* to create the deep nodes. In DeepZono, every operation has its node, while in DeepPoly a matrix multiplication or convolution followed by an activation layer is put together. Additionally, DeepPoly differentiates between first, intermediate, and last layer nodes. Table 4.2 shows translation from operations to nodes. For DeepZono and DeepPoly, only the deep nodes are relevant, while for RefineZono and RefinePoly the optimizer also has to provide the information necessary to do the refined analysis. For that purpose, the special representation of the neural network *nn* is filled with this information, while iterating over *operation_types*. The information put into *nn* is very similar to Table 4.1, which is expected as the refined analysis is trying to do the same as the nodes.

For DeepZono and RefineZono, the optimizer has to add a DeepzonoDuplicate node to the *execute_list* for every Resadd operation present. For residual network, the DeepzonoDuplicate node creates a copy of the node that is input to the two branches.

Table 4.2: *IR to deep nodes*

Operation sequence	DeepZono nodes[1]	DeepPoly nodes[2]
Placeholder	Input[3]	Input
MatMul	Matmul	-
MatMul → Add/BiasAdd	Affine	Relu[4]
Gemm	Affine	Relu[4]
Add	Add	-
BiasAdd	Add	-
Resadd	Resadd[5]	Resadd[6]
Sub	Sub	Sub
Mul	Mul	Mul
Conv2D	Conv	-
Conv2D → BiasAdd	Convbias	Conv2d[6]
Conv	Convbias	Conv2d[6]
MaxPool	Maxpool	Maxpool
Relu	Relu	-
Sigmoid	Sigmoid	-
Tanh	Tanh	-
Gather	Gather	Gather
Reshape	Gather	Gather
Gemm[7] → ReLU	Affine → Relu	Relu[8]
Gemm[7] → Sigmoid	Affine → Sigmoid	Sigmoid[8]
Gemm[7] → Tanh	Affine → Tanh	Tanh[8]

[1] All names have Deepzono prepended.
[2] All names have Deeppoly prepended and Node appended.
[3] This can also be InputZonotope.
[4] This is only true if this is the last node.
[5] This also adds a DeepzonoDuplicate node to the *execute_list*.
[6] This is the same node if a Relu operation follows. The node has a boolean has_relu, which denotes just that.
[7] Gemm is interchangeable with MatMul → Add/BiasAdd
[8] This node has First, Intermediate or Last appended depending on its position

4.6 Deep nodes

As previously stated the deep nodes are domain-specific. DeepZono nodes are defined in *deepzono_nodes.py* and DeepPoly nodes defined in *deeppoly_nodes.py*. All deep nodes have an initialization and a *transformer* method. In the initialization method, all parameters needed in the *transformer* method are saved. If a parameter is an array it is put in continuous memory. The *transformer* method is mainly a wrapper for a call to ELINA. The *transformer* method takes 10 and 11 arguments for DeepZono and DeepPoly respectively. The arguments are all the same except for the additional parameter *use_area_heuristic* for DeepPoly. The *transformer* arguments are:

- *nn*, a special network representation
- *man*, the *zonoml* or *fppoly* manager
- *element*, the abstract element
- *nlb*, the lower bounds for refined analysis
- *nub*, the upper bounds for refined analysis
- *relu_groups*, a list needed for k-ReLU
- *refine*, the boolean to decide whether to do refined analysis
- *timeout_lp*, the timeout for the LP solver
- *timeout_milp*, the timeouts for the MILP solver
- *testing*, the boolean determining whether to return additional test bounds
- *use_area_heuristic*, the boolean for DeepPoly to decide whether to use area heuristic

A special case, are the Input nodes as their *transformer* call initializes *element* and has only the domain manager *man* as input. The *transformer* method of all other deep nodes returns the manipulated *element* or a tuple with the *element* and the lower and upper bound for the current node if *testing* is true. The lower and upper bounds returned for testing are numerically the same as the bounds used to do refined analysis with, but the list of bounds could not be reused because for refined analysis, not every node's bounds are needed, e.g. Relu node. This will be further elaborated in Section 4.8.

4.7 Analyzer

The analyzer *analyzer.py* is where the network is finally analyzed. The analyzer is initialized with the list of deep nodes provided by the optimizer *ir_list* and *nn*, the special representation of the neural network. To analyze the network property the *analyze* method is called. In it, the analyzer iterates over the deep nodes and calls *transformer* on every node. At the end of this iteration, the analyzer has an abstract element representing the network properties and the list of lower and upper bounds. For RefinePoly, these bounds are then used to create an instance of the LP solver. In the next step, the analyzer iterates over every class in the output and checks

if its score is bigger than that of every other class in the output. This is accomplished by calling ELINAs domain-specific *is_greater* method, which takes the manager *man*, the abstract element *element*, the indexes of the two classes, and for DeepPoly and RefinePoly additionally *use_area_heuristic*. The result of *is_greater* is true if the class with the first index has a higher value for the whole analyzed input region. If a class is found that is greater than every other, then the iteration stops and the index of the class and the bounds are returned. When testing, the *analyze* method additionally returns *output_info*, the list of testing-relevant tensor names and shapes.

4.8 Refinement

The refinement is only done for RefineZono and RefinePoly domain, as previously stated. If the network is analyzed with these domains, then the *transformer* method's *refine* argument is true. The process is distributed over the deep nodes. In deep nodes, that map from MatMul, Gemm, Conv2D, or Conv in the IR, the lower and upper bounds are appended to *nlb* and *nub* respectively. Those bounds are then improved in nodes that map from Relu in the IR. This means the DeeppolyReluNodes (First, Intermediate, and Last) and DeeppolyConv2dNodes do both add bounds and improve them. Like the options suggest there are two different methods employed to improve these bounds. The first is the triangle approximation of ReLU with LP or MILP and the second is the k-ReLU analysis. The triangle approximation of ReLU is defined in *ai_milp.py*. The solver used for LP and MILP is Gurobi [Gurobi Optimization 2019]. For convolutional and residual networks analyzed with RefinePoly, refinement is only done once the analyzer has finished iterating over the deep nodes. k-ReLU is implemented in *krelu.py*.

5

Testing Framework

This chapter describes the testing framework of ERAN. The first section defines the use cases of this testing framework. The second section describes the changes needed for this framework. The third section describes the parts of ERAN that can be tested with this framework. The fourth section provides example calls. Finally, the last section describes all test output in detail.

Correctness is imperative for a neural network verifier. The testing framework presented here ensures that the analysis for concrete points is correctly implemented, by comparing the result of the model with the bounds from ERAN. This is implemented in the file *check_models.py* in the folder *testing*. Running *check_models.py* tests multiple networks with multiple domains, which is enabled by the ERAN initialization and *analyze_box* being in try clauses. All tests are printed to a single output file.

5.1 Layout of the Testing Framework

This section describes the testing framework and how it is built. The testing framework can test the ERAN initialization and the method *analyze_box*. In the following, a parser test will refer to the test of the ERAN initialization, a normal test will refer to a test of *analyze_box* and a test refers to both. Since the testing framework should continue testing even if a test fails with an exception, both the ERAN initialization and the call of *analyze_box* are in try clauses.

Running *check_models.py* without arguments is described by Algorithm 1.

Algorithm 1 Testing Framework

```
1:  procedure CHECK_MODELS.PY
2:      output file ← open("tested.txt")
3:      for folder ∈ testing/test_nets do
4:          dataset ← folder name
5:          for network ∈ folder do
6:              model ← load(network)
7:              eran ← ERAN(model)
8:              input ← load(dataset)
9:              for domain ∈ [DeepZono, DeepPoly, RefineZono, RefinPoly] do
10:                 lower bounds, upper bounds ← eran.analyze_box(input, domain)
11:                 eran_results ← average(lower bounds, upper bounds)
12:                 results ← model(input)
13:                 if end eran_result - end result < acceptable difference then
14:                     output file ← "success"
15:                 else
16:                     for eran_result, result ∈ bounds, results do
17:                         if eran_result - result > acceptable difference then
18:                             output file ← exact layer the differences started
19:                             goto 9
```

5.2 Modifications to ERAN for testing

The model is loaded in *read_net_file.py*, as is explained in Section 4.2. The testing framework uses this model to run the same input through ERAN and the model. If the results differ, then the intermediate results are compared layer by layer to help determine the location where the divergence occurs. To get these intermediate results, additions to ERAN were necessary. First, the optimizer returns the additional *output_info*. The *output_info* is a list, which for deep nodes in *execute_list* has the name and shape of the tensor, corresponding to the last operation covered by the deep node (see Table 4.2). For deep nodes that do not correspond to an operation, e.g. *DeepzonoDuplicate*, there is no entry in *output_info*. Second, the *analyzer* takes an additional argument *testing* in his initialization method. Third, most nodes give back the bounds while testing, instead of just returning the abstract element, as shown in Listing 5.1.

```
class Deepnode:
    ...

    def transformer( ... , testing ):
        ...

        if testing :
            return element, lower_bound, upper_bound
        return element
```

Listing 5.1: *Deepnode transformer return*

To calculate the bounds, the methods *add_bounds* and *calc_bounds* were implemented in *deepzono_nodes.py* and *deeppoly_nodes.py* respectively. These are the same bounds that are used for refined analysis, therefore, the last element of refinement bounds *nlb* and *nub* are returned.

5.3 ERAN code covered by the testing framework

This section describes the parts of the ERAN code that can be tested with the testing framework.

Chapter 4 showed the components involved with the network analysis in detail. The first component is the reader. The framework cannot test this part. For .onnx, .pb and .meta files, the ONNX and TensorFlow frameworks load the model. In this case, the methods used are tested by the framework developers and testing them is outside the scope of our framework. Testing for .tf and .pyt files could be added but was omitted because ONNX support was explicitly added to replace the need for these formats. The translators can be tested separately with the option *parser*, discussed later. When testing the translator the resulting *operations* list is shown as a result.

The other part that can be tested is the *analyze_box* method of ERAN end to end. This is done by comparing the average of the output bounds of *analyze_box* with the output of the model. For ONNX the input for the model has to be transposed to fit the NCHW format. To run the ONNX models ONNX Runtime [Microsoft 2020] is used. The *output_info* added to the result of *analyze_box*, mentioned in Section 5.2 is used to add the output to the model. Listing 5.2 and 5.3 show how this is managed for TensorFlow and ONNX models, respectively. If the bounds of ERAN and the result of the model do not match, the additional output from the models is compared with the respective bounds of ERAN to find the exact layer, where the results diverged. For examples of test outputs, see Section 5.5.

```
# TensorFlow only needs the names
output_names = [e[0] for e in output_info]

# Get the output for every named tensor in TensorFlow model
pred = sess.run([sess.graph.get_tensor_by_name(name[7:]) for name in out_names],
                {sess.graph.get_operations()[0].name + ':0': input})
```

Listing 5.2: *Get output for relevant layers in TensorFlow*

```
# Make input NCHW for ONNX
input = input.transpose(0, 3, 1, 2)

for name, shape in output_info:
    # Create a new output node
    out_node = helper.ValueInfoProto(type = helper.TypeProto())
    out_node.name = name
    out_node.type.tensor_type.elem_type =
            model.graph.output[0].type.tensor_type.elem_type

    # Make shape NCHW if necessary
    if len(shape)==4:
        shape = [shape[0], shape[3], shape[1], shape[2]]

    # Add dimensions to ONNX shape
    for dim_value in shape:
        dim = out_node.type.tensor_type.shape.dim.add()
        dim.dim_value = dim_value

    # Make model return additional output
    model.graph.output.append(out_node)

# Get output of ONNX model for input
runnable = rt.prepare(model, 'CPU')
pred = runnable.run(input)
```

Listing 5.3: *Get output for relevant layers in ONNX*

5.4 Using the testing framework

This section will first show all options the *check_models.py* script provides and then go over use cases.

The testing options are shown in Table 5.1. In this subsection, a test references running the *analyze_box* method and model once. Tests are run for every *network* and every *domain* given by the user. For flags that tune the analysis, e.g. *use_area_heuristic* or *use_2relu*, the testing framework is referencing the same config.py as the ERAN in normal use.

5.4.1 Testing options

This subsection gives an overview of the testing options with Table 5.1 and describes the option in more detail in the respective paragraphs.

[1] deepzono, deeppoly, refinezono, and refinepoly
[1] If this option is taken, then dataset MUST be given.

Table 5.1: *Testing options*

Name	Default	Description
dataset	None	The dataset for the networks
domain	all[1]	The domains tested
network	None	The networks to be tested[2]
out	tested.txt	The name of the output file
parser	False	The flag denoting to only the parser will be tested
continue	False	The flag denoting to only to run tests without a previous result
failed	False	The flag denoting to only to run tests without previous success

dataset The *dataset* can be defined as an option and multiple networks can be tested with this *dataset*. This has the drawback, that all networks tested have to be for the same *dataset*. The testing framework uses folders to solve the problem of testing networks with different datasets. Looking in the *testing/testing_nets* folder, every folder name inside is used as the *dataset* for the networks inside the folder itself. This option is provided, to enable the *network* option.

domain One or more domains can be given as an option when running *check_models.py*. The default is to test all of them, which means DeepZono, DeepPoly, RefineZono, and RefinePoly. The purpose of this option is to cut down testing time during the development of domain-specific code.

network With the *network* option, one can specify one or more relative or absolute paths to networks. If no network is provided, then, as previously stated, the networks in the folders inside *testing/testing_nets* are tested. This option is provided to test specific models without moving them.

out *Out* defines the name of the output file. The format of the output file will be described in Section 5.5. The default is tested.txt. The *check_models.py* script creates a new result file if it does not already exist. If the file already exists the file is appended and not overridden. The output of every test is written in this file.

parser The *parser* flag signifies, that only the translators are tested. It can be used to check if a model can be analyzed using ERAN.

continue This option can be used to continue aborted test runs. The script is also stopped when a segmentation fault occurs.

failed With the *failed* flag, only tests that have not yet succeeded, are run.

5.4.2 Testing example calls

This subsection presents use cases for the tesing framework and the commands to solve them.

In the first use case, the user has networks of different types and classifying different datasets in the folders *testing/test_nets/mnist/* and *testing/test_nets/cifar10/*. With this setup, the user can just run the command, as described in Listing 5.4. To test only with DeepZono and RefineZono, e.g. there are changes in *deepzono_nodes.py*, the user can execute the command in Listing 5.5. To add testing support for a dataset A, the user can put an A_test.csv file in the *data* folder and then test networks with that input by putting them in the *testing/test_nets/A/* folder.

```
$ python3 check_models.py
```

Listing 5.4: *Run all tests*

```
$ python3 check_models.py --domain deepzono refinezono
```

Listing 5.5: *Run tests for DeepZono and RefineZono*

In the second use case, the user has many models exported from PyTorch and wants to check if ERAN can parse them. The dataset predicted by the models is CIFAR-10, so he can put them in the *testing/test_nets/cifar10* folder. Listing 5.6 is the call to run the tests.

```
$ python3 check_models.py --parser
```

Listing 5.6: *Run parser only tests*

In the third use case, the user wants to check two large models, which he does not want to move. The option *network* provides just that. Listing 5.7 shows this.

```
$ python3 check_models.py --network /path/cifarNet1.onnx /path/cifarNet2.onnx
--dataset cifar10
```

Listing 5.7: *Run test for a single network*

The fourth use case sees the user developing a feature in ERAN. He has three models in the *testing/test_nets/cifar10* folder, which failed the tests previously because the feature was not implemented correctly. Listing 5.8 is the command to only rerun the tests that failed.

```
$ python3 check_models.py --failed
```

Listing 5.8: *Rerun tests that failed before*

In the fifth use case, the user crashed his laptop during a test run because he forgot to plug it in. Listing 5.9 shows the command to continue, where it left off.

```
$ python3 check_models.py --continue
```

Listing 5.9: *Continue aborted test run*

5.5 Format of the test output

This section explains the possible results of a test. All results are printed to the output file *out*. A test can be run with the option *parser* or without. For tests run with this option the positive result states, the *dataset*, the *network*, the string "ERAN parsed successfully" and the *operation_types* of the translator. An example can be found in Listing 5.10.

```
mnist, convMedGTANH__Point.pyt, ERAN parsed successfully
, [' Placeholder ', 'Conv2D', 'BiasAdd', 'Tanh', 'Conv2D', 'BiasAdd', 'Tanh',
 'MatMul', 'BiasAdd', 'Relu', 'MatMul', 'BiasAdd'],
```

Listing 5.10: *Success message for parser*

The failure message for the translator, no matter the option, is printed in the except part of the try clause and consists of the *dataset*, the *network*, and the stack trace with "ERAN parse error trace: " prepended.

The success message for the end-to-end test is given in Listing 5.11. It contains the *dataset*, the *network*, the *domain*, and the string "success".

```
mnist, convMedGTANH__Point.pyt, deepzono, success
```

Listing 5.11: *Success message for end-to-end test*

In case a test fails because of an exception in ERAN, the "success" string is exchanged with "ERAN analyze error trace: " and the stack trace. Listing 5.12 is such a failure message. This error message is genuine as ERAN cannot handle Conv2D $\rightarrow$ BiasAdd $\rightarrow$ Sigmoid for DeepPoly analysis.

```
mnist, convMedGTANH__Point.pyt, deeppoly, trace: Traceback (most recent
call last):
  File "check_models.py", line 195, in <module>
    label, nn, nlb, nub, output_info = eran.analyze_box(specLB, specUB, domain,
    1, 1, True, testing=True)
  File "../tf_verify/eran.py", line 73, in analyze_box
    execute_list, output_info = self.optimizer.get_deeppoly(nn, specLB, specUB,
    lexpr_weights, lexpr_cst, lexpr_dim, uexpr_weights, uexpr_cst, uexpr_dim,
    expr_size)
  File "../tf_verify/optimizer.py", line 539, in get_deeppoly
    assert 0, "the Deeppoly analyzer doesn't support the operation: '" +
    self.operations[i] + "' of this network: " + str(self.operations)
AssertionError: the Deeppoly analyzer doesn't support the operation: 'Tanh' of
this network: ['Placeholder', 'Conv2D', 'BiasAdd', 'Tanh', 'Conv2D', 'BiasAdd',
'Tanh', 'MatMul', 'BiasAdd', 'Relu', 'MatMul', 'BiasAdd']
```

Listing 5.12: *Failure message for convMedGTANH__Point.pyt, the operation Tanh is not supported*

The last error message is the one, where the result of ERAN and the model do not match. Listing 5.13 is produced, by taking the bounds of ERAN before ReLU is applied. The message states dataset, network name and domain as before and on the following lines, the result ERAN gave back, the result of the model, at which layer the differences started, what is the name inside the model and the difference.

```
cifar10, cifar_conv_maxpool.tf, deeppoly,
eran, [ 1.06886569 -1.70941519 -0.2204093  5.92306311 0.34591173 0.94349005
  1.88718244 0.19593985 -0.5731354 -1.04429471],
model, [1.06886603 0.         0.         5.92306201 0.34591181 0.94348935
 1.88718205 0.19593944 0.         0.        ]
cifar10, cifar_conv_maxpool.tf, deeppoly, started divergence at layer, 0,
outputname, import/Relu:0,
difference, [-3.37952500e-01 -7.57158626e-01 1.23336668e-08 ... -1.60802317e+00
 -1.52837109e+00 -3.22496391e-01]
```

Listing 5.13: *Failure message stating the first layer at which the results differ*

6

Conclusion and Future Work

In this thesis, the state-of-the-art neural network verifier ERAN has been expanded with following features:

- A new network format ONNX was supported, with custom shape inference and constant propagation.
- The abstract domain RefinePoly was implemented.
- A file format was designed for encoding Zonotope inputs which involved creating a file reader and a special input node.
- From DeepG, the geometric perturbations involving translation, rotation, and scaling, were integrated, with both on the fly input region generation and file reading support.
- k-ReLU was added as an option for refining analysis.
- A testing framework, where the output of ERAN and the analyzed model are directly compared and divergences located, was developed to ensure the correctness of the verifier.

In the future this work can be expanded by:

- Integrating GPU implementation of DeepPoly called GPUPoly. This will allow running the DeepPoly analysis on the GPU.
- Support for additional network layers can be added, e.g. LSTM, Conv3D, Maxpool3D.
- Input batching can be added, as the analysis is done one image at a time at the moment.
- Support for additional datasets can be added, e.g. FaceScrub [Ng and Winkler 2015].
- The analysis for L_1 and L_2 norm based perturbations can be implemented.

- Support for the verification of user defined network properties can be added.

Bibliography

AL-SHAYEA, Q. 2011. Artificial neural networks in medical diagnosis. *Int J Comput Sci Issues 8* (02), 150–154.

BALUNOVIC, M., BAADER, M., SINGH, G., GEHR, T., AND VECHEV, M. 2019. Certifying geometric robustness of neural networks. In *Advances in Neural Information Processing Systems 32*.

BALUNOVIC, M., BAADER, M., SINGH, G., GEHR, T., AND VECHEV, M., 2020. Deepg. `https://github.com/eth-sri/deepg`.

BOJARSKI, M., TESTA, D., DWORAKOWSKI, D., FIRNER, B., FLEPP, B., GOYAL, P., JACKEL, L., MONFORT, M., MULLER, U., ZHANG, J., ZHANG, X., ZHAO, J., AND ZIEBA, K. 2016. End to end learning for self-driving cars.

BOOPATHY, A., WENG, T.-W., CHEN, P.-Y., LIU, S., AND DANIEL, L. 2019. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In *AAAI*.

BUNEL, R., TURKASLAN, I., TORR, P. H. S., KOHLI, P., AND KUMAR, M. P. 2017. Piecewise linear neural network verification: A comparative study. *CoRR abs/1711.00455*.

DVIJOTHAM, K., STANFORTH, R., GOWAL, S., MANN, T., AND KOHLI, P. 2018. A dual approach to scalable verification of deep networks.

DVIJOTHAM, K., STANFORTH, R., GOWAL, S., QIN, C., DE, S., AND KOHLI, P. 2019. Efficient neural network verification with exactness characterization. In *UAI*.

EHLERS, R. 2017. Formal verification of piece-wise linear feed-forward neural networks. *CoRR abs/1705.01320*.

GEHR, T., MIRMAN, M., DRACHSLER-COHEN, D., TSANKOV, P., CHAUDHURI, S., AND VECHEV, M. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. 3–18.

GUROBI OPTIMIZATION, L., 2019. Gurobi optimizer reference manual.

HINTON, G., DENG, L., YU, D., DAHL, G., MOHAMED, A.-R., JAITLY, N., SENIOR, A., VANHOUCKE, V., NGUYEN, P., SAINATH, T., AND KINGSBURY, B. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE 29* (11), 82–97.

KATZ, G., BARRETT, C. W., DILL, D. L., JULIAN, K., AND KOCHENDERFER, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR abs/1702.01135*.

KOLTER, J., AND WONG, E. 2017. Provable defenses against adversarial examples via the convex outer adversarial polytope.

KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-10 (canadian institute for advanced research).

LECUN, Y., AND CORTES, C. 2010. MNIST handwritten digit database.

MANFREDI, G., AND JESTIN, Y. 2016. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 1–9.

MICROSOFT, C., 2020. Onnxruntime.

NG, H.-W., AND WINKLER, S. 2015. A data-driven approach to cleaning large face datasets. *2014 IEEE International Conference on Image Processing, ICIP 2014* (01), 343–347.

OLIPHANT, T. E. 2006. *A guide to NumPy*, vol. 1. Trelgol Publishing USA.

RAGHUNATHAN, A., STEINHARDT, J., AND LIANG, P., 2018. Semidefinite relaxations for certifying robustness to adversarial examples, 11.

RUAN, W., HUANG, X., AND KWIATKOWSKA, M. 2018. Reachability analysis of deep neural networks with provable guarantees. *CoRR abs/1805.02242*.

RUOSS, A., BAADER, M., BALUNOVIĆ, M., AND VECHEV, M., 2020. Efficient certification of spatial robustness.

SINGH, G., GEHR, T., MIRMAN, M., PÜSCHEL, M., AND VECHEV, M. 2018. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 10802–10813.

SINGH, G., GEHR, T., PÜSCHEL, M., AND VECHEV, M. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang. 3*, POPL (Jan.).

SINGH, G., GEHR, T., PÜSCHEL, M., AND VECHEV, M. 2019. Robustness certification with refinement. In *International Conference on Learning Representations*.

SINGH, G., MAURER, J., MIRMAN, M., GEHR, T., HOFFMANN, A., TSANKOV, P., DRACHSLER COHEN, D., PÜSCHEL, M., AND VECHEV, M., 2020. Eran. `https://github.com/eth-sri/eran`.

SINGH, G., PÜSCHEL, M., AND VECHEV, M., 2020. Elina. `https://github.com/eth-sri/ELINA`.

TJENG, V., AND TEDRAKE, R. 2017. Verifying neural networks with mixed integer programming. *CoRR abs/1711.07356*.

WENG, T.-W., ZHANG, H., CHEN, H., SONG, Z., HSIEH, C.-J., BONING, D., DHILLON, I., AND DANIEL, L., 2018. Towards fast computation of certified robustness for relu networks, 04.

XIAO, H., RASUL, K., AND VOLLGRAF, R., 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

ZHANG, H., WENG, T., CHEN, P., HSIEH, C., AND DANIEL, L. 2018. Efficient neural network robustness certification with general activation functions. *CoRR abs/1811.00866*.