Verification of Unsupervised Neural Networks

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Fakultät für Informatik
der
Technischen Universität Dortmund

vorgelegt
von

Benedikt Böing

aus
Versmold

Bad Honnef, 2023

Angefertigt mit Genehmigung der Fakultät für Informatik
der Technischen Universität Dortmund

Gutachter: Prof. Dr. Emmanuel Müller
Gutachter: Prof. Dr. Barbara König

Tag der Promotion: 21.06.2023
Erscheinungsjahr: 2023

Declaration I, Benedikt Böing, confirm that this work is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (ideas, equations, figures, text, tables, programs) are properly acknowledged at the point of their use. A full list of the references employed has been included.

Acknowledgments This thesis would not have been possible without help from many different people. First of all, I would like to thank my supervisor Prof. Emmanuel Müller for his ongoing support, the fruitful discussions we had and for always believing in the potential value of my ideas. Moreover, I want to thank all my collaborators including Rajarshi Roy, Simon Klüttermann, Jelle Hüntelmann, Richard Stewing, Daniel Neider and Falk Howar without whom the publications would neither be of the same quality nor would they have been as much fun to write. Going further, I want to also thank Erik Scharwächter and Chiara Balestra for wonderful times in the same office sharing cups of coffee as well as insights on research and beyond. A special thanks does also go to Anton Tsitsulin. With him and with Erik I could spend a wonderful year at the University of Bonn before the pandemic forced us all to retreat to home office. I thoroughly enjoyed all the seminars and discussions we had in both Bonn and Dortmund. For this, I am grateful for having worked alongside Daniel Wilmes, Carina Newen and Bin Li as well as some of the previously mentioned colleagues. Beyond the research group in Dortmund I want to thank Florian Seiffarth with whom I often shared an office in Bonn and who made each visit to his office a fun and productive experience.
Each dissertation is born out of curiosity, perseverance and ongoing doubts. I am very grateful for all the support my family and friends gave me during the more difficult times. A very special "Thank you" goes to my fiancee Julia. She has not only shared home office with me but has supported me immensely throughout the entire doctorate. It is her I dedicate this thesis to.

Abstract

Neural networks are at the forefront of machine learning being responsible for achievements such as AlphaGo. As they are being deployed in more and more environments - even in safety-critical ones such as health care - we are naturally interested in assuring their reliability. However, the discovery of so-called adversarial attacks for supervised neural networks demonstrated that tiny distortions in the input space can lead to misclassifications and thus, to potentially catastrophic errors: Patients could be diagnosed wrongly, or a car might confuse stop signs and traffic lights. Thus, ideally, we would like to guarantee that these types of attacks cannot occur.

In this thesis we extend the research on reliable neural networks to the realm of unsupervised learning. This includes defining proper notions of reliability, as well as analyzing and adapting unsupervised neural networks with respect to this notion. Our definitions of reliability depend on the underlying neural networks and the problems they are meant to solve. However, in all our cases, we aim for guarantees on a continuous input space containing infinitely many points. Therefore we extend the traditional setting of testing against a finite dataset such that we require specialized tools to actually check a given network for reliability. We will demonstrate how we can leverage neural network verification for these purposes. Using neural network verification, however, entails a major challenge: It does not scale up to large networks. To overcome this limitation, we design a novel training procedure yielding networks that are both more reliable according to our definition as well as more amenable for neural network verification. By exploiting the piecewise affine structure of our networks, we can locally simplify them and thus decrease verification runtime significantly. We also take a perspective that complements a neural network's training by exploring how we can repair non-reliable neural network ensembles.

With this thesis, we paradigmatically show the necessity and the complications of unsupervised neural network verification. It aims to pave the way for more research to come and towards a safe usage of these simple-to-build yet difficult-to-understand models given by unsupervised neural networks.

Contents

Introduction 1

Machine learning has become one of the most promising and pervasive technologies of our time. As its applications range from medical diagnosis (KEE$^+$15) over image classification (LZX$^+$18) to chatbots (SLW$^+$20), it has the potential to not only relieve us from repetitive and undesired tasks but also to fuel scientific discovery and even create art. Unlike traditional software, it is not programmed to follow a predefined command pattern but is instead created implicitly by training on a given training dataset. This different paradigm allows machine learning to leverage on the ever-increasing amount of readily available data as well as on better computer chips enabling the training of larger models. Sparked by these developments, machine learning research has gained tremendous momentum designing novel ways of extracting information and models from raw data.
The major powerhouse of machine learning are neural networks being responsible for breakthroughs such as AlphaGo (SHM$^+$16), AlphaFold (JEP$^+$21) or Dall-e (RPG$^+$21). Though they first showed their exceptional performance on high-dimensional data such as images (KSH12) or texts (VSP$^+$17), they are now widespread even in safety-critical domains such as healthcare (ERR$^+$19) or infrastructure monitoring (YJY19). Their success can partly be attributed to their enormous versatility. They typically consist of a large amount layers of neurons through which the input is passed forward. These neurons allow them to implicitly learn very sophisticated features on which the eventual prediction is based. Thereby neurons enhance and sometimes even replace the manual task of feature engineering. As a result, given enough training data and neurons, neural networks can model any function, no matter how complicated (HSW89).
However, their predictive capability entails a price to pay: measured in the number of parameters, they are very complex models and the prediction of a given neural network does not follow a simple formula. Therefore neural networks are very often regarded as blackbox models (ZBW$^+$18; LLY$^+$21; OSF19) evading every attempt to be analyzed, described or understood globally. Moreover, as all its behavior is learned implicitly by means of training, we cannot directly encode safety properties into them - a severe downside in safety-critical applications. For example, we must handle constraints derived from the domain or infeasible

input-output pairs separately. Put simply: not only do we not know what the network does, we can also not define what it must not do.
This can become dangerous as the seminal paper by Szegedy et al in 2014 (SZS$^+$14) has shown. Their work revealed that neural networks usually exhibit undesired behavior in the form of adversarial attacks. These attacks are carefully crafted inputs to a given, already trained neural network which are classified wrongly. Starting from a valid input, for example from the training dataset, a very small amount of carefully designed noise is added such that it remains almost the same. The output of the neural network on this new input, however, changes drastically, resulting in a different prediction. For example, an image classifier can be made to believe that the picture of a turtle shows a gun (AEIK18) or vice versa yielding the network unsafe for gun detection at airport security. In other words, adversarial attacks are designed to fool the neural network and thus, from a more theoretical viewpoint, we know that neural networks fail to generalize over their entire input domain.
There have been many responses to this threat addressing several dimensions of the problem: on the one hand, there is a vast amount of research directed towards training models to be resistant against this sort of attack (BLZ$^+$21a). Methods such as adversarial training (MMS$^+$18) incorporate adversarial attacks into the training procedure to thereby fortify the neural network against them. On the other hand, we need proof that a neural network is safe once it has been trained. To be precise, we want to certify that for a given neural network no adversarial attack exists. We can achieve this using a process called neural network verification (KBD$^+$17; Ehl17; WOZ$^+$20). While usually machine learning models are tested against a finite amount of test data, this is not enough for neural network verification. Indeed, as the potential input space of neural networks is not limited, we must somehow cover infinitely many samples in the verification process.
Adversarial attacks have become a major research field for neural networks and with every defense against a particular type of adversarial attack, a new way of attacking is devised. This arms race between attackers and defenders leads to a better understanding of neural networks and raises many questions on if and how we can use neural networks in safety-critical environments. Its importance can both be seen by the number of publications in the field, which has grown exponentially in the recent years (HKR$^+$20a). Moreover, the VNN competition has been established (MBB$^+$23) recognizing the importance of neural network verification to overcome the threat posed by adversarial attacks.

1.1 Challenges

This thesis contributes to some of the challenges we identify in this field.
As of yet, almost all of the existing research in this area is conducted exclusively for the case of supervised learning. However, of course there exist neural networks for unsupervised learning such as anomaly detection or dimensionality reduction. Autoencoders (HHWB02) or the DeepSVDD model (RVG$^+$18) for example are commonly used neural networks to extract smaller representations of the data (autoencoder) and to do anomaly detection (autoencoder and DeepSVDD). Since adversarial attacks essentially exploit the unstable behavior inherent in neural networks, there is no reason to believe that these models should not be prone to them. However, it is as of yet not clear what exactly an adversarial attack on an unsupervised neural network is supposed to be. Unlike in the supervised approach, we have no labels at our disposal based on which we can define them. Therefore, we will first need to define useful notions of adversarial attacks for these types of networks.
The second challenge is given by scaling the verification process. The larger the neural network's complexity, the longer it takes to verify them. In fact, verification time grows exponentially with the number of neurons in the neural network (MMS$^+$22). The scientific approaches to overcome this problem are twofold. One direction tries to optimize the verification procedure by inventing new heuristics that exploit the neural network's structure and activation patterns. While some notable successes could be achieved (KHI$^+$19; WOZ$^+$20; KBD$^+$17; Ehl17), this endeavor is most likely limited to smaller-sized models because neural network verification is an NP-complete problem (KBD$^+$17). Therefore another approach relaxes the exact verification procedure sacrificing completeness for speed: they overapproximate the possible outputs of the input domain under scrutiny. As a result, these approaches may or may not certify a given neural network and in case they do not, we simply do not know whether the network is safe.
Given that we can solve the first challenge, the third challenge follows very naturally: we want to train neural networks in such a way that they become more robust against adversarial attacks. The typical way inspired by supervised neural networks would be given by adversarial training. Relying on an algorithm to quickly generate adversarial attacks (GSS15; MMS$^+$18), it simply adds them to the training dataset or even replaces the original samples with them. However, these methods usually yield models resistant to only that particular type of attack (TB19). We on the other hand want to make models more resistant to all these types of attacks by directly changing the neural network's structure.
Finally, assume that we are given a non-robust neural network: a neural network for which adversarial attacks exist. As neural network training can be very expensive, we would like to *post-robustify* it without the need to fully retrain.

However, current methods to increase robustness after training address the inputs to the neural network by preprocessing (XEQ18) or introducing randomness (XWZ^+17) into the neural network. None of the existing approaches directly adjusts the model's weights, biases or structure. A possible answer could lie in further adversarial training. However, this may be expensive depending on how much training needs to be done and - more importantly - we cannot guarantee robustness of the resulting model. Therefore we tackle the challenge of increasing robustness by making well-informed changes to the model itself. Of course, these changes must be small enough not to interfere with the model's performance and large enough to have a significant effect on the model's robustness.

Beyond the challenges we address in this thesis, there are of course more open challenges in neural network verification.
The first arises from an application point of view. Dependent on what the neural network is eventually supposed to do, it is most likely not enough to test for low-level functionality such as robustness to adversarial attacks. Instead, we require a high-level language encoding the semantics of a given problem and its safety specifications. For example in autonomous driving, we would like to ensure that stop signs are correctly classified by an image classifier. However, this entails somehow exhaustively defining all possible stop signs and their perturbations if we only have these low-level specifications at our disposal. If we could have that, we would likely not need any neural network in the first place as its strength lies exactly in implicitly, rather than explicitly, yielding an accurate classification function. Therefore, one challenge is to give safe and yet not overly explicit verification properties.
A second challenge which is largely unaddressed in the research community is the sound and complete verification of neural networks utilizing general activation functions. LSTM-based models (HS97) for example, used in natural language processing, build mostly on sigmoidal rather than piecewise linear activation functions. However, these types of models can so far only be verified using an overapproximation of their outputs.
The last open challenge of interest concerns the closeness of adversarial attack. Starting from an initial input usually from the training set, an adversarial attack is supposed to be as close as possible to this input. Indeed, many papers (MMS^+18; PMJ^+16; CW17a) phrase the problem of finding an adversarial attack as an optimization problem minimizing the distance to the original input constrained on being classified differently. Hence, methods to find adversarial attacks can be evaluated based on how well this minimization problem is solved. These methods give a scalable way to determine an upper bound on the distance to the closest adversarial attack. If they had guarantees on their approximation

gap to an optimal solution, they could even be considered verification methods. Thus one open challenge is to find the closest adversarial attack in a scalable way. Note here that exact verification procedures can be used to approximate the closest adversarial attack arbitrarily well. This shows how strongly neural network verification and adversarial attacks are intertwined.

1.2 Contributions

In this thesis, we address the aforementioned challenges with several contributions.
First of all, we bridge the gap between supervised and unsupervised learning with respect to adversarial attacks by introducing the notion of unsupervised adversarial attacks for autoencoders. Autoencoders are simple, feed-forward neural networks trained to reconstruct their input. Without labels, we will rely on exactly this property of every autoencoder: the reconstruction loss. Irrespective of what exactly a given autoencoder is used for - be it denoising, anomaly detection or dimensionality reduction - their loss function gives a difference between the input and the output. Therefore, utilizing an adversarial attack on its loss function, we can verify whether a given autoencoder deviates from its intended behavior for training samples. On the other hand, if the loss function in a given area is bounded, we may conclude that the model does not confound different classes in its latent space or prove that it denoises. Indeed many applications of autoencoders are based on its reconstruction error such that our novel type of adversarial attack is quite versatile.
We will furthermore adapt the existing notion of supervised adversarial attacks to the unsupervised use case of anomaly detection. Most anomaly detection methods produce an anomaly score for each point which, in conjunction with a fixed threshold, defines the binary label *anomalous* or *non-anomalous* for a given input. Therefore, if we can prove that the anomaly score never exceeds the threshold, we can prove that all points in a given area are non-anomalous. Thus, we can use the technique of neural network verification to prove a desired property of a given neural network.
In addition to the mere definition of new types of adversarial attacks, we also contribute to the challenge of verification scalability by introducing a new perspective on the problem. Given that almost all research introduces new heuristics to improve scalability of an NP-complete problem, we instead remark that very often, we also have control over the creation of the verification instance. To be precise, we often both train and verify the neural network. Hence we may incorporate our expertise on the verification process into the training procedure.

To allow for sound and complete verification, we consider only a subclass of neural networks: piecewise affine ones. This is a standard restriction and - as we will see in Chapter 2 - not a severe one. Based on this structure, we find the number of affine subfunctions to be crucial for verification runtime. Therefore, we reduce the number of subfunctions by adding a particular regularization term called *fctdist* to neural network training. This term essentially aims at making neighboring subfunctions similar to each other, effectively eliminating a large amount of them. As we will see, this leads to neural networks that are much faster to verify while at the same time retaining their predictive capabilities.
As a further effect, the *fctdist* term addresses the third challenge of training more stable neural networks. For autoencoders, we define the notion of ε-δ robustness indicating how strongly the input must be changed to result in a particular change of the output. As we will see, autoencoder trained with *fctdist* have a much higher ε-δ robustness. This can be useful to prove denoising or measure the amount of regularization.
A remarkable side result we obtained during our experiments is to never use dropout for training neural networks which afterward shall be verified. We show experimentally that dropout severely increases the runtime of verification procedures.
In the last chapter, we extend our work on autoencoders to another type of unsupervised neural network based model: the DEAN model (KM22). One way to think of it is as a large ensemble of small DeepSVDD models (RVG$^+$18). While each of the subnetworks is relatively small, in union they form a competitive anomaly detector. By employing feature bagging, the DEAN model ensures a constant size for each subnetwork and allows it to be used on datasets of any dimension. We explore how we can leverage this particular ensemble structure for neural network verification. In particular, we want to verify that all points surrounding a normal input will be classified as normal as well. Therefore we adapt the usual robustness notion from supervised learning to the case of anomaly detection by regarding a given anomaly detector as a binary classifier.
The first question is how to actually verify the ensemble. We solve this by a divide-and-conquer approach splitting up the model into its subnetworks, verifying each of them independently and merging the results. In fact, each subnetwork yields two results: an upper bound on the anomaly score proving that all surrounding points are normal as well and a lower bound via an input to the entire model designed to produce a high anomaly score. These two results also allow us to estimate the surprisingly small approximation gap, which stems from the combination of feature bagging and the divide-and-conquer approach.
Using ensemble models entails two major advantages. First, we replace verifying one large neural network by verifying many small neural networks. This turns

out to be much faster because it scales linearly in the number of subnetworks used and, thereby, also in the model's complexity. Moreover, this can be easily parallelized. Secondly, once we have verified each subnetwork, we can leave out the ones that yield a too large anomaly score and thus post-robustify a given, non-robust model. As we will see, this does not only ensure robustness for the input set we verified but increases robustness overall.

1.3 Outline

The remainder of this thesis is organized as follows. We first introduce theoretical foundations, including notation as well as properties of piecewise affine neural networks in Chapter 2. This chapter details what complicates unsupervised neural network verification and is thus necessary to understand the chapters thereafter. Moreover, we give an idea of how exact solvers - the ones used in this thesis - work as our solution frameworks are based on them. We cover related work in Chapter 3 discussing all relevant alternatives to the approaches we propose. Chapter 4 then serves as the starting point for our journey into unsupervised neural network verification. With the worst-case-error, it gives the first unsupervised problem specification and highlights its usefulness while at the same time hinting at problems that derive from it. Chapter 5 then builds on top of Chapter 4 introducing *fctdist*, which alleviates both the scalability issue as well as the severity of the worst-case-error. We expand into other types of unsupervised neural networks in Chapter 6 giving another solution to the scalability problem of general unsupervised neural networks. However, it also relies on Chapter 4 because we use the worst-case-error as a subroutine in our solution. Finally Chapter 7 concludes the thesis revisiting its contributions and pointing at future work that arises from it.

Table 1.1 shows how the different chapters add contributions to the challenges addressed in this thesis.

	Chapter 4	Chapter 5	Chapter 6
Problem Specification	✓	✓	✓
Scalability	✗	✓	(✓)
Robustness Training	(✓)	✓	✗
Post-Robustification	✗	✗	✓

Table 1.1: Overview of how the different chapters contribute to the different challenges.

1.4 List of Publications

This thesis relies mostly on the following papers jointly written with Rajarshi Roy, Simon Klüttermann, Jelle Hüntelmann, Richard Stewing, Daniel Neider, Falk Howar and Emmanuel Müller.

(Ben20): Benedikt Böing, Rajarshi Roy, Daniel Neider, Emmanuel Müller. Quality Guarantees for Autoencoders via Unsupervised Adversarial Attacks. ECML/PKDD 2020

(Ben22a): Benedikt Böing, Emmanuel Müller. On Training and Verifying Robust Autoencoders. DSAA 2022

(Ben22c): Benedikt Böing, Simon Klüttermann, Emmanuel Müller. Post-Robustifying Deep Anomaly Detection Ensembles by Model Selection. ICDM 2022

(Ben22b): Benedikt Böing, Falk Howar, Jelle Hüntelmann, Emmanuel Müller, Richard Stewing. Neural Network Verification with DSE. OVERLAY@AI*IA 2022

Theoretical Foundations of Neural Networks 2

This chapter introduces the core concepts of interest for the following chapters: neural networks and SMT solvers. Neural networks serve as the objects we want to ensure reliability for. Thus, we will describe them on a technical level and discuss some of their properties relevant for their analysis. These properties will also motivate the use of SMT solvers, which - in the context of this thesis - can be seen as the analysis tool for neural networks.

2.1 Neural Networks

Neural networks are responsible for some of the most important breakthroughs in machine learning of the last decades. They have shown exceptional performance on tasks as diverse as playing Go (SHM+16), creating images from text (RPG+21) and speech recognition (DXX18). These achievements stem from their enormous flexibility as they are capable of modeling any (!) function. While their main building blocks can be easily explained, the resulting models can become very complicated and thus are beyond the reach of a simple analysis. Therefore, to keep control of this rapidly expanding technology, we require more sophisticated analysis tools for neural networks.

2.1.1 Definition, Structure and Notation

We start by technically defining neural networks. A *neural network* $f : \mathbb{R}^N \to \mathbb{R}^M$ is a function consisting of *layers* of *neurons*. At each neuron, an affine function based on the output of the previous layer is calculated followed by an *activation function*. As neural networks are organized in layers, we can define their calculations as $f = f_L \circ ... \circ f_1$ where $L + 1$ is the number of layers (including the input layer). Each f_i calculates its output $x^{(i)}$ as

$$x^{(i)} = f_i(x^{(i-1)}) = \mathsf{act}(W^{(i)}x^{(i-1)} + b^{(i)})$$

where *act* is an activation function applied neuron-wise, usually introducing non-linearities into *f*. The first layer $x^{(0)}$ is called the *input layer*, the last layer $x^{(L)}$

the *output layer* and all the layers in between are called *hidden layers*.
Sometimes we will need the *preactivations* of each layer as well corresponding to $x^{(i)}$ before their activation function. Thus they are given by $y^{(i)} = W^{(i)}x^{(i-1)} + b^{(i)}$. Let $l_i \in \mathbb{N}$ be the number of neurons in layer i. Then the weight matrix $W^{(i)} \in \mathbb{R}^{l_i \times l_{i-1}}$ and the bias term $b^{(i)} \in \mathbb{R}^{l_i}$ determine the layer's affine behavior and are the key parameters learned during training.

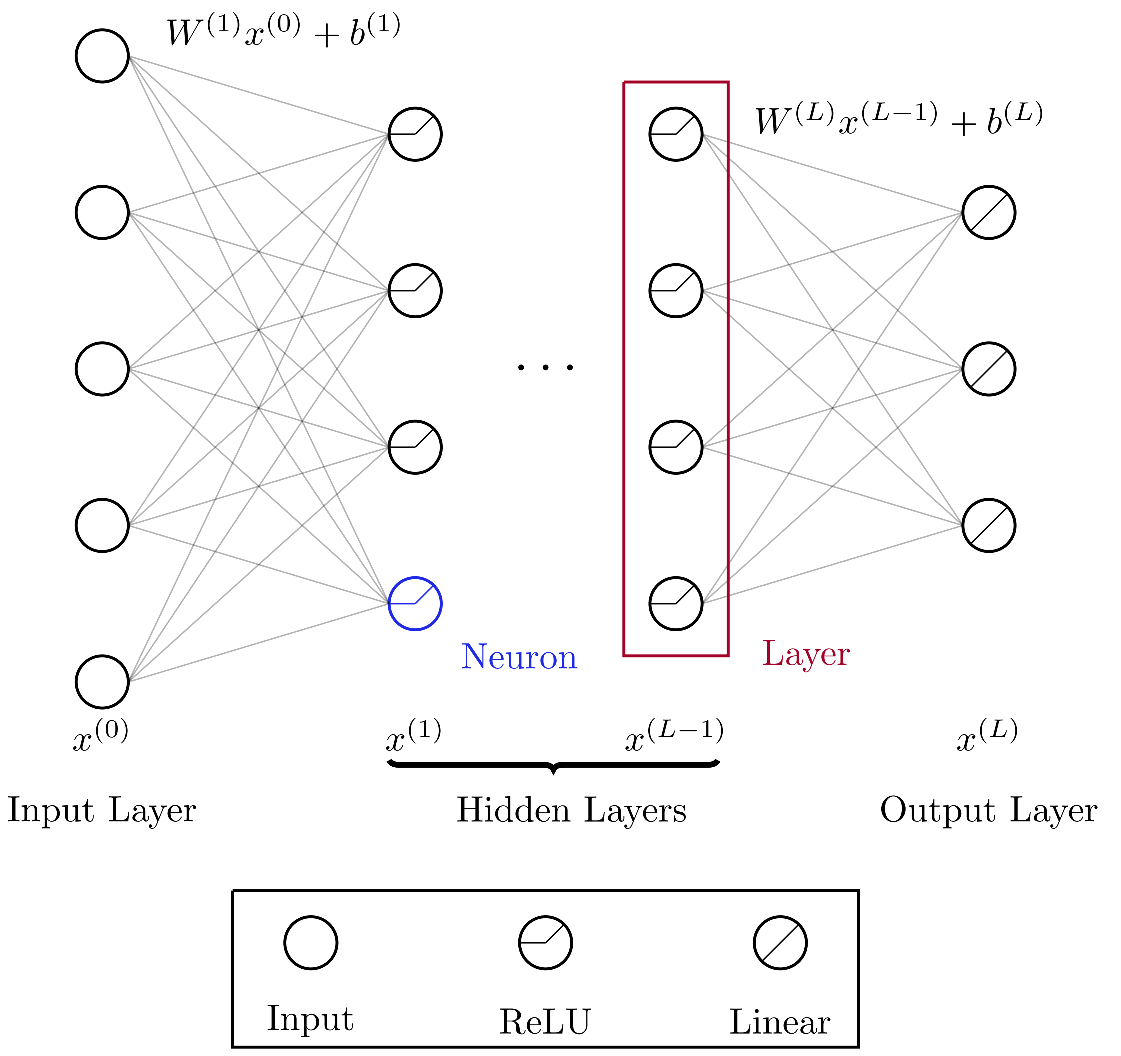


Figure 2.1: Depiction of a feed-forward neural network. The input is processed through several layers, which consist of neurons. On each neuron, an affine function followed by an activation function is calculated. Different activation functions are shown by different neuron fillings.

In this thesis, we will be using two types of activation functions exclusively: the Linear and the ReLU activation function. The Linear activation function is simply the identity function such that act$(x) = x$ while the ReLU activation function is given by

$$ReLU(x) = \max(0, x)$$

mapping negative inputs to 0 and positive inputs to themselves. Thus, they are piecewise linear functions with two regimes: for one particular input to the neural network, a ReLU neuron is either activated (> 0) or deactivated (≤ 0). This leads to an *activation pattern* over all ReLU neurons in the neural network. For k neurons, this is a function $actpat : \mathbb{R}^N \rightarrow \{0, 1\}^k$ indicating which ReLU neuron is activated and which is not for a given input.
These ReLU neurons, although looking innocent, are what, on the one hand, drastically increase a given neural network's expressive power and what, on the other hand, make them difficult to analyze.
There are other popular choices for activation functions, such as the $tanh$ or $sigmoid$ function. However, they do not yield piecewise affine neural networks and are therefore not included in this thesis. This is neither a severe restriction in practice - those are the most commonly used activation functions (GBC16) - nor in theory, as we will see in Theorem 2.1.1.
Neurons and layers are named by their respective activation functions resulting in, for example, *Linear neurons* or *ReLU layers*. ReLU networks, on the other hand, allow both ReLU and Linear neurons because Linear neurons are often required in the last layer to be able to predict negative values. The overall structure of a neural net, including the number of layers as well as the number and types of their neurons, is called the neural network's *architecture*. As can be seen, by this construction, neural networks alternate between affine functions reshaping the input and activation functions which introduce non-linearities. To ease understanding, Figure 2.1 shows an exemplary neural network.

2.1.2 Universal Approximation Theorem

One of the key properties shown by (HSW89) for neural networks is their ability to approximate any given function. Loosely speaking, given a function *g*, we can construct a neural network *f* such that the distance between *g* and *f* is smaller than a given threshold. Note though that the original theorem requires a particular type of activation function - so-called squashing functions - and gives no bound on the number of neurons required. Thus the resulting neural networks can become arbitrarily complicated.
Their work has been extended to more types of activation functions and refined

with respect to the number of neurons required. To give a precise statement, we adapt a more recent theorem by (LPW⁺17). Note that there are more general universal approximation theorems, as the following one holds only for ReLU networks; since this is the only type of network we consider in this thesis, that is no problem.

Theorem 2.1.1. *For any Lebesgue-integrable function $g : \mathbb{R}^n \to \mathbb{R}^m$ and any $\varepsilon > 0$, there exists a fully-connected ReLU network f such that*

$$\int_{\mathbb{R}^n} |f(x) - g(x)| dx < \varepsilon$$

In essence, this theorem states that we can use neural networks to model any function, making them one of the most versatile models that exist. However, their complexity comes with two caveats. On the one hand, it makes them very difficult to analyze and we cannot easily formulate global and understandable properties about a given model. In contrast to, example given, Decision Trees (BFOS84) or a Gaussian Naive Bayes model (HY01), we do not have a more concise representation of how a network processes its input than the network itself.
Moreover, due to the infinitely many functions we can realize with a fixed architecture, it is very challenging to obtain one of the parameter configurations that leads to a good performance on a given task. Thus the next section covers the main training procedure used for neural networks.

2.1.3 Training via Backpropagation

So far, we have established the skeleton of our model, which is the basic structure of the neural network as well as its universal approximation property. However, machine learning models have a predefined purpose, such as classification or regression. Thus, out of all the possible functions a given neural network architecture can convey, we want to find a particular function suitable for the task at hand. Therefore we must somehow encode the properties we want the function to have in the neural network.
This is typically done using a combination of a training set and a loss function. The training set $(\mathcal{X}, \mathcal{T})$ consists of pairs of input-output samples for the neural network where $\mathcal{X} = (x_1, \dots, x_n)$ are the input samples and $\mathcal{T} = (t_1, \dots, t_n)$ are their respective targets. It represents the information we have on the true function we wish to approximate. Note that in case of supervised learning, the targets are usually given in the form of labels or the true value of a regression task. However, we are not limited to such types of targets. Instead, for the case of autoencoders - an unsupervised model - each target t_i is simply given by the input x_i.

The loss function L, on the other hand, measures how strongly the neural network's output deviates from the target and incorporates the task the neural network is supposed to do. By design, loss functions are monotone, differentiable and penalize larger deviations more severely. Thus, by summing the loss over the training set, we can measure how well the neural network performs on it.
We can adjust the neural network f by changing its parameters $p = (p_i)_{i \in I}$ consisting of its weights and biases leading to the following minimization problem:

$$\min_p \sum_{i=1}^{n} L(f_p(x_i), t_i)$$

Examples of loss functions include cross-entropy for classification, mean-squared error for regression and many other functions reflecting the versatility of neural network use cases (GBC16).
Using these ingredients, we can follow the standard machine learning paradigm of training our model: instead of defining every parameter of the neural network, we sequentially show it training samples and update parameters according to how well the neural network performs on them as measured by the loss function. The exact update procedure is given by an algorithm called backpropagation (RHW86). It simply calculates the derivatives of the loss function with respect to the weights and biases and adjusts them such that the loss decreases.
To be precise let f, L, p, be as above, $\alpha \in \mathbb{R}_{>0}$ be a step size and (x, t) be an input-output pair. Then backpropagation proceeds as follows:

1. Propagate x through f obtaining $f(x)$ and all intermediate results
2. Calculate $L(f_p(x), t)$
3. Calculate $\Delta_i := \frac{\partial L}{\partial p_i}(f_p(x), t)$ for all $i \in I$.
4. Update p according to $p_i^{new} = p_i^{old} - \alpha \Delta_i$ for all $i \in I$

Even though there is a vast amount of literature (SCZZ20) on optimizing the neural network's training procedure, at their core, all methods rely on some form of this algorithm. As can be seen, one very important ingredient is the loss function's differentiability. Without it, we cannot calculate its partial derivative. This is to be kept in mind when designing new loss functions (see Chapter 5).

2.1.4 Piecewise Affine Neural Networks

As mentioned before, this thesis considers ReLU networks exclusively. On the one hand, according to Theorem 2.1.1, they can approximate any given function

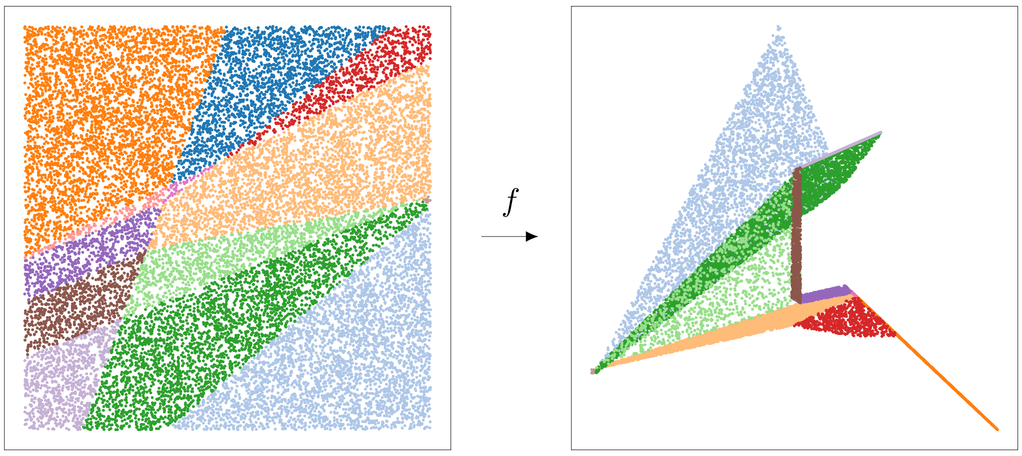


Figure 2.2: Depiction of a two-dimensional input (left side) as well as a two-dimensional output (right side) of a neural network f. The Figure is obtained by sampling inputs in $[-1, 1]^2$ and color-coding their linear subfunctions.

just as well as networks employing different activation functions. On the other hand, these networks exhibit a particular structure, making them easier to be analyzed: they are piecewise affine[1]. As each ReLU neuron splits up into the activated (> 0) and the non-activated (≤ 0) regime, these networks divide the input space into segments on each of which an affine subfunction is calculated, as can be seen in Figure 2.2.

Figure 2.3 shows that on different segments - even neighboring ones - very different subfunctions can be applied even though all of them are affine: they allow for stretching, rotation and translation. In trying to verify a given neural network, this structure has proven to be very beneficial because it allows to consider the different subfunctions one by one: if we want to make a global statement about the entire neural network - the core task of neural network verification - it suffices to check that statement on each of the affine subfunctions. However, even if these networks are much simpler to analyze due to their piecewise affine structure, it still remains a challenging endeavor given that the number of subfunctions is exponential in the number of ReLU neurons (MMS$^+$22). Although we will eventually use off-the-shelf, state-of-the-art verification engines (WOZ$^+$20), which

[1]Note that also other types of layers, including convolutional layers, and pooling layers lead to piecewise affine neural networks. However, they are out of the scope of this thesis.

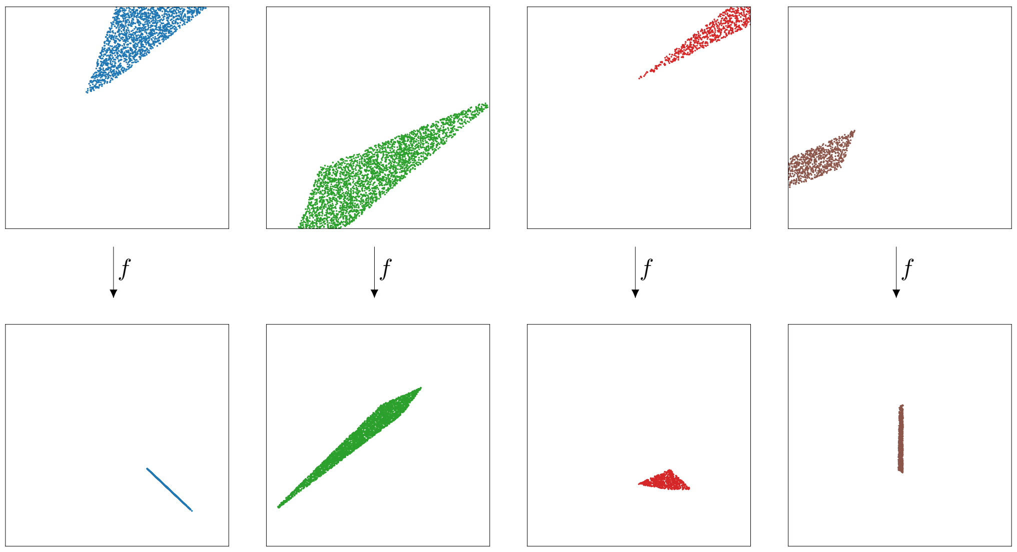


Figure 2.3: Exemplary mappings for different linear subfunctions from Figure 2.2. Different linear subfunctions can exhibit very different behavior.

implicitly handle all the calculations for us, we will give some geometric intuition of ReLU networks both for the sake of a better understanding as well as because we will exploit it to design a training procedure in Chapter 5.
To sequentially check each subfunction for a verification property we - loosely speaking - need to answer the following two questions: which affine subfunctions does the neural network calculate and where are they applied? We can answer both these questions using the following setup: Assume a given ReLU network $f : \mathbb{R}^N \to \mathbb{R}^M$ as well as an input point $x^* \in \mathbb{R}^N$. We can easily calculate the affine subfunction the network calculates on x^* as well as in its ε-environment given by

$$B_\varepsilon(x^*) := \{x \in \mathbb{R}^N : dist(x, x^*) < \varepsilon\}$$

for some metric $dist$ and some $\varepsilon > 0$. Here we assume that x^* is not on the border of two subfunctions (see Figure 2.4).
We will first consider the case of f consisting of Linear layers only and then reduce the more complicated case of ReLU layers to the case of Linear layers by showing how we can locally replace ReLU neurons with Linear neurons.

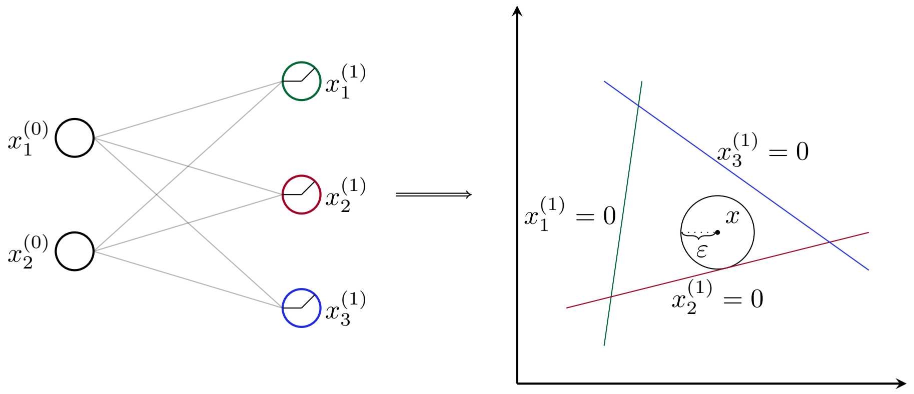


Figure 2.4: For every point x not on the border of two subfunctions, we can calculate the affine subfunction that is applied on x and in its ε-environment. The ReLU neurons $x_1^{(1)}, x_2^{(1)}, x_3^{(1)}$ being equal to zero define the borders on which the affine subfunction changes.

Theorem 2.1.2. *Let $f : \mathbb{R}^N \to \mathbb{R}^M$ be a neural network consisting of an input layer and L Linear layers with weight matrices and biases given by $W^{(1)}, \dots, W^{(L)}$ and $b^{(1)}, \dots, b^{(L)}$ respectively.*
Then f is a linear function with

$$f(x) = \underbrace{\left(\prod_{l=L}^{1} W^{(l)}\right)}_{V^{(L)} \in \mathbb{R}^{MxN}} x + \underbrace{\sum_{l=L-1}^{1} \left(\prod_{j=L}^{l+1} W^{(j)}\right) b^{(l)} + b^{(L)}}_{a^{(L)} \in \mathbb{R}^{M}}$$

for all $x \in \mathbb{R}^N$.

Proof. We prove the theorem by induction over the number of Linear layers L. First, consider the case where $L = 1$. Then the function f is given by

$$f(x) = W^{(1)}x + b^{(1)}$$

which is in line with the formula.
For the induction step $L - 1 \to L$, we assume that we have shown the formula for networks with $L - 1$ Linear layers. Moreover, we make use of the layer-wise definition of neural networks giving $f = f_L \circ \underbrace{f_{L-1} \circ \dots \circ f_1}_{f^{L-1}}$ where f^{L-1} is a neural network with $L - 1$ Linear layers. Thus we can apply that f_L is a Linear layer as

well as the induction hypothesis and obtain

$$\begin{aligned}
f(x) &= f_L \circ f_{L-1} \circ \dots \circ f_1 \\
&= W^{(L)}(V^{(L-1)}x + a^{(L-1)}) + b^{(L)} \\
&= W^{(L)} \left(\left(\prod_{l=L-1}^{1} W^{(l)} \right) x + \sum_{l=L-2}^{1} \left(\prod_{j=L-1}^{l+1} W^{(j)} \right) b^{(l)} \right) + b^{(L)} \\
&= \left(\prod_{l=L}^{1} W^{(l)} \right) x + \sum_{L-2}^{1} \left(\prod_{j=L}^{l+1} W^{(j)} \right) b^{(l)} + W^{(L)} b^{(L-1)} + b^{(L)} \\
&= \left(\prod_{l=L}^{1} W^{(l)} \right) x + \sum_{L-1}^{1} \left(\prod_{j=L}^{l+1} W^{(j)} \right) b^{(l)} + b^{(L)}
\end{aligned}$$

finishing the proof. □

Next, we will reduce the more challenging case of ReLU networks to Linear networks by iteratively replacing all ReLU neurons with Linear neurons. This in turn, allows us to apply the formula from Theorem 2.1.2 and to thereby calculate the local affine subfunction.
Consider a single ReLU neuron r and observe that for any particular input, the neuron r is either activated $(r > 0)$ or not $(r \leq 0)$. In the activated case, it already (locally) behaves as a Linear neuron such that we can safely replace the ReLU activation function with a Linear activation function for r and still obtain the same function. For the deactivated case, the ReLU neuron has to return 0, which we simulate by setting the respective weights and bias of the preceding layer to 0. To be precise, let r be the k-th ReLU neuron on layer j and assume that for a given x this ReLU neuron is deactivated. To obtain a locally functionally equivalent network with a Linear neuron, we replace the weights $W^{(j)}_{k\cdot}$ (the k-th row of the respective weight matrix) as well as $b^{(j)}_k$ (the corresponding bias term) by 0 (see Figure 2.5 for an example). This way, we can ensure that if we replace the ReLU neuron with a Linear neuron with the adjusted weights, the resulting neural network will calculate the same function for x and in its local environment. By replacing all ReLU neurons with Linear neurons, we reduce this case to the purely linear case and can calculate the local affine subfunction using Theorem 2.1.2.
As can be seen, by the previous description, different affine subfunctions loosely correspond to different ReLU activation patterns. On the one hand, different affine subfunctions certainly have different activation patterns. On the other hand, different activation patterns do not necessarily result in different affine subfunctions. If, for example, $W^{(L)}$ contains only zeros, the activation pattern of the previous layers does not matter. Again, note that a fixed ReLU activation

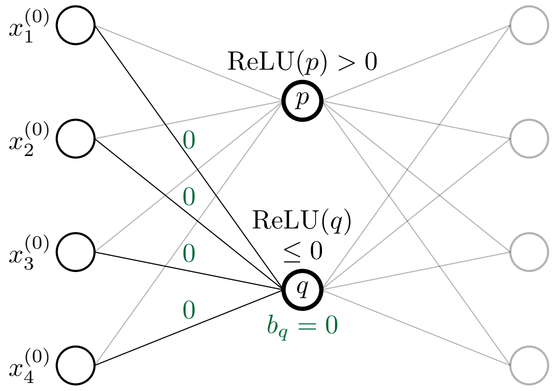


Figure 2.5: Adjusting a ReLU neuron such that the resulting network locally calculates the same function as the original network. For x in the original network, the preactivation of neuron q was negative. Thus we adjust the preceding weights and the bias to 0 and can replace both p and q by a linear neurons.

pattern does only yield the same affine subfunction in a local environment. However, we also need to know where exactly the neural network applies it.
We can give the subdomain on which the neural network f applies a particular subfunction as an intersection of half-spaces. To see this, we fix the ReLU activation pattern of an input x in the domain and describe it as a Linear Program (LP) (KVKV11). Note that we do not need an optimization objective but are just interested in describing the feasible region as linear constraints.
Let $\mathcal{N} = \{y_{k,j} \mid k \in \{1, \dots, L\}, j \in \{1, \dots, l_k\}\}$ be variables for the set of all non-input neurons and $\mathcal{D}_x \subset \mathcal{N}$ correspond to the deactivated ReLUs for a given input x. Moreover, we introduce variables $x_{k,j}$ for the j-th neuron in layer k. Then the variables $x_{0,j}$ of the feasible region of the following LP describe the inputs on which the neural network applies the same affine subfunction as on x.

$$\begin{aligned} y_{k,j} &= \sum_{i=1}^{l_{k-1}} W_{i,j}^{(k)} x_{k-1,i} + b_j^{(k)} && y_{k,j} \in \mathcal{N} \\ x_{k,j} &= 0 && y_{k,j} \in \mathcal{D}_x \\ x_{k,j} &= y_{k,j} && y_{k,j} \in \mathcal{N} \setminus \mathcal{D}_x \end{aligned} \tag{2.1}$$

The LP essentially describes all inputs whose forward passes through f lead to the same activation pattern as x. Therefore on all inputs for which the LP is feasible, the same affine function is applied. Of course, we do not necessarily need an input x to create an activation pattern. Instead, for k ReLU neurons, we can also just choose one of the 2^k many ReLU activation patterns. Note that in

this case, the LP does not need to be feasible at all if there is no input for this particular activation pattern.
The different constraints described by 2.1 define the borders to other affine subfunctions. Whenever the right-hand side of these equations cross 0, the activation pattern changes and thus, the resulting affine subfunction may change.
Let us consider the behavior of a single ReLU neuron in more detail. A ReLU neuron r without its activation function can be interpreted as a function $r : \mathbb{R}^N \to \mathbb{R}$ by defining it as the neural network's output. Then this subnetwork f_r given by all layers preceding r and r as the output neuron is - again - a ReLU network and thus it - again - defines an affine function f_{r,x^*} for any input x^*. To recap, f_{r,x^*} is the affine function that the neural network f_r applies on x^*. Its kernel $ker(f_{r,x^*}) = \{x \in \mathbb{R}^N : f_{r,x^*}(x) = 0\}$ defines a hyperplane separating the input space into two half-spaces: one where $f_{r,x^*} > 0$ and one where $f_{r,x^*} < 0$ corresponding to $r > 0$ or $r < 0$ in the original network as seen from the local affine subfunction on which x^* lives. Since the original network f does only change its affine subfunction if at least one ReLU neuron flips from being activated to deactivated or vice versa, these hyperplanes define the borders within which f applies the same affine subfunction. Note, though, that the half-spaces obtained as the kernel of f_{r,x^*} do indeed not only depend on the neuron r but also on the point x^*. This effect can be seen in Figure 2.6 demonstrating that the hyperplane defined by a particular ReLU neuron may change along with the affine subfunction of f. This is also intuitively clear, as, for example, a ReLU neuron becoming active in an early layer influences all subsequent ReLU neurons and their respective borders. As a result, the kernel $ker(f_{r,x^*}) = \{x \in \mathbb{R}^N : f_{r,x^*}(x) = 0\}$ and $ker(f_r) = \{x \in \mathbb{R}^N : f_r(x) = 0\}$ can differ significantly and the borders defined by ReLU neurons can become very complicated (see Figure 2.7).

2.2 Exact Neural Network Verification

Irrespective of the exact verification problem we intend to solve, we may need to consider all subfunctions a given neural network f comprises. We have seen how we can calculate a particular subfunction and where it is applied. However, we still need to describe how to consider them sequentially. As supervised neural networks are not in principle different to unsupervised neural networks with respect to their architecture, we can benefit from the vast body of research on optimizing this process for them (WOZ+20; KBD+17; Ehl17).

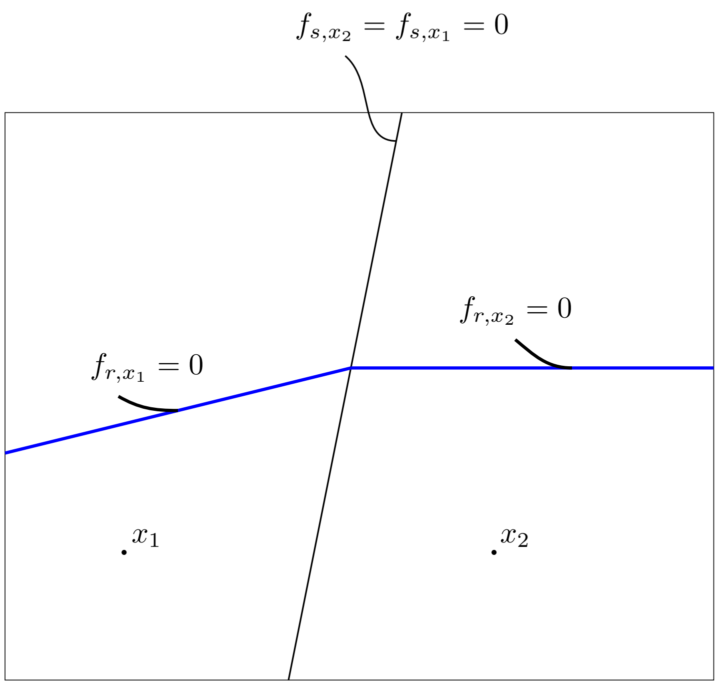


Figure 2.6: Depiction of possible different borders for different inputs (x_1 and x_2) for the same ReLU neuron r (blue). The borders $f_{r,x_1} = 0$ and $f_{s,x_2} = 0$ differ as neuron s influences neuron r.

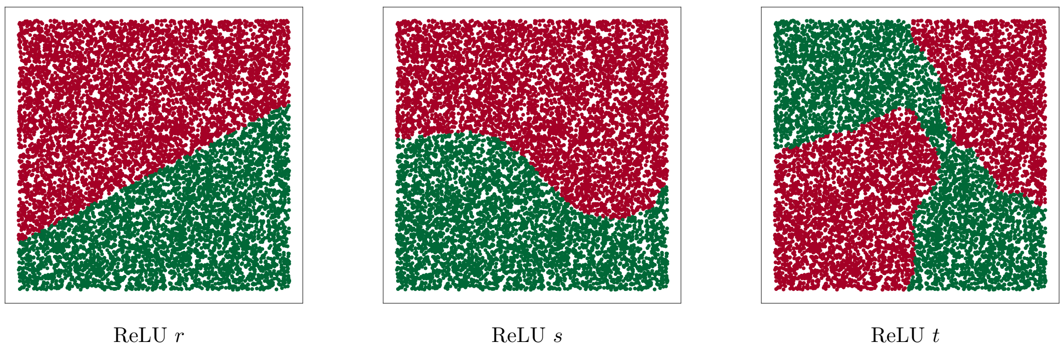


Figure 2.7: Implicit visualization of the border given by $ker(f_r)$ for different neurons r, s, t. Different colors on the inputs indicate whether the neuron is activated (green) or not (red) for that particular input. Neurons in the first non-input layer (neuron r) define a straight border, whereas neurons in later layers (neurons s and t) can define arbitrarily complex borders.

2.2.1 Satisfiability Modulo Theories (SMT)

At their core, all exact solvers formulate the problem as a logical conjunction of linear constraints. We will present one framework here - SMT solvers - as we believe it is easy to understand and as it allows us to highlight the difficulties of neural network verification. It separates the process of neural network verification into analyzing a particular subfunction and iterating over the different subfunctions. Therefore we can focus on the two steps independently. Put simply, SMT solvers abstract away linear constraints into variables of an SAT instance, repeatedly solve the SAT instance, which in turn implies the direction in which the constraints are to be satisfied and then use an LP solver to determine if the assignment is feasible. Next, we will give the details on these steps and introduce the required definitions.
Most problems of neural network verification can be solved by reducing it to a constraint satisfaction problem in first-order logic. A formula in first-order logic is formed using constants, variables, function and predicate symbols, logical connectives, and quantifiers. In this thesis, however, we require only a specific first-order logic, namely the *quantifier-free fragment of linear real arithmetic* (*LRA*), which we introduce next.
First, let $\mathcal{X} = \{x_0, x_1, ...\}$ be a set of *variables* which range over values in $\mathbb{R}$. Then, we define *terms* as follows: a term is either a constant $c \in \mathbb{R}$, a variable $x \in \mathcal{X}$, or a function application $t_1 \circ t_2$, where $\circ \in \{+, \cdot\}$ and t_1, t_2 are two terms. For instance, 5, x, and $3 \cdot x + 2 \cdot y$ are terms. To reflect the usual notation, we often drop the multiplication sign.
An *atomic formula* is a predicate symbol applied to terms. In LRA, we allow the usual binary predicates $<$, $\leq$, $=$, $\geq$, and $>$. For example, $3x + 2y > 5$ is an atomic formula. Moreover, a *formula* is inductively defined as follows: a formula is either an atomic formula, the negation $\neg\varphi$ of a formula φ, or the disjunction $\varphi_1 \vee \varphi_2$ of two formulas φ_1, φ_2. We also add syntactic sugar and allow the formulas $\varphi_1 \wedge \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$, which are defined as usual:

$$\varphi_1 \wedge \varphi_2 := \neg(\neg\varphi_1 \vee \neg\varphi_2) \tag{2.2}$$

$$\varphi_1 \rightarrow \varphi_2 := \neg\varphi_2 \vee \varphi_1 \tag{2.3}$$

$$\varphi_1 \leftrightarrow \varphi_2 := (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \tag{2.4}$$

To assign meaning to formulas, we introduce the concept of *interpretations*. An interpretation is a mapping $\mathcal{I} \colon \mathcal{X} \rightarrow \mathbb{R}$, which assigns a real value to each variable. Interpretations can easily be lifted to terms by calculating their value, and we write $\mathcal{I}(t)$ for the interpretation (the value) of the term t under $\mathcal{I}$. Finally, we can define when an interpretation $\mathcal{I}$ *satisfies* a formula φ which we denote by $\mathcal{I} \models \varphi$: we have $\mathcal{I} \models t_1 \diamond t_2$ for $\diamond \in \{<, \leq, =, \geq, >\}$ if and only if $\mathcal{I}(t_1) \diamond \mathcal{I}(t_2)$ is true, $\mathcal{I} \models \neg\varphi$ if $\mathcal{I} \not\models \varphi$, and $\mathcal{I} \models \varphi_1 \vee \varphi_2$ if and only if $\mathcal{I} \models \varphi_1$ or $\mathcal{I} \models \varphi_2$. We say that a formula φ is *satisfiable* if an interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi$ exists. The framework implementing highly-optimized procedures for deciding satisfiability of formulas in LRA is called *Satisfiability Modulo Theories* (*SMT*) (BFT17). As this is a general framework, it does not only allow checking satisfiability in LRA but also in many other fragments of first-order logic. This entails, however, that these solvers are not specifically designed for neural networks. Moreover, SMT solvers typically return an interpretation if the given formula is satisfiable. These will later turn out to be very useful as these will serve as counterexamples of a desired property.

2.2.2 Encoding the Neural Network

To encode the function computed by a neural network f in LRA, we introduce variables $x_{k,j}$ for each layer $k \in \{0, \dots, L\}$ and each neuron $j \in \{1, \dots, l_k\}$ in Layer k. Intuitively, each such variable captures the value of a neuron and is used as the input for other neurons. Correspondingly, variables $x_{0,1}, \dots, x_{0,l_0}$ represent the input to the neural network, while variables $x_{L,1}, \dots, x_{L,l_L}$ represents the output of the neural network. Note that we use $x_j^{(k)}$ to denote neurons and their associated values in the neural network and $x_{k,j}$ to denote the corresponding variable in the verification framework. To ensure that the variables $x_{k,j}$ actually have the desired meaning, we introduce constraints that describe the computation of each neuron. For a linear neuron, we construct

$$\psi_{k,j} := \Big[x_{k,j} = \big[\sum_{i=1}^{l_{k-1}} W_{i,j}^{(k)} x_{k-1,i}\big] + b_j^{(k)}\Big]. \tag{2.5}$$

On the other hand, for a ReLU neuron, we construct the constraint

$$\psi_{k,j} := \Big[\big[y_{k,j} = \sum_{i=1}^{l_{k-1}} W_{i,j}^{(k)} x_{k-1,i} + b_i^{(k)}\big] \wedge \big[x_{k,j} = \mathsf{ite}(y_{k,j} < 0, 0, y_{k,j})\big]\Big], \tag{2.6}$$

where ite (short for "if-then-else") is syntactic sugar for a conditional evaluation of terms. It is given by

$$\begin{aligned} &\big[(y_{k,j} < 0) \wedge (x_{k,j} = 0)\big] \\ \vee &\big[(y_{k,j} \geq 0) \wedge (x_{k,j} = y_{k,j})\big]. \end{aligned}$$

Finally, we define

$$\varphi^f := \bigwedge_{1 \leq k \leq L} \bigwedge_{1 \leq j \leq l_k} \psi_{k,j},$$

which collects the constraints for all individual neurons. By construction φ^f completely encodes the neural network f in the sense that $f(\mathcal{I}(x_{0,1}), \dots, \mathcal{I}(x_{0,l_0})) =$

$(\mathcal{I}(x_{L,1}), \dots, \mathcal{I}(x_{L,l_L}))$ holds for all satisfying interpretations $\mathcal{I} \models \varphi^f$.
The encoding of *f* in the SMT framework looks very similar to the LP formulation for a particular subfunction given in Section 2.1.4. Basically, it extends it with the ite operator representing the ReLU function and thus, implicitly, the SMT formulation iterates over all subfunctions. While this may not look like a big difference, it actually is the reason for the exponential runtime of neural network verification. Indeed, neural network verification is an NP-complete problem. To be precise, consider the following problem. For a given neural network *N* and linear constraints over its inputs and outputs, we want to decide whether there exists an input in the restricted input domain that gets mapped to an output in the restricted output domain by *N*. This problem is NP-complete as 3-SAT can be reduced to it (KBD+17).
To solve the formula, SMT solvers first abstract the linear inequalities in each subformula to variables. Thus, we obtain an SAT instance for which we can obtain multiple solutions using, for example, DPLL (DP60; DLL62). Each solution fixes the inequalities to be true or false, determining whether the inequality is to be satisfied (true) or violated (false). Thus each solution results in an LP as the conjunction over the (potentially adjusted) linear inequalities. If the LP is feasible, the SMT solver returns *sat* together with a feasible assignment to all variables. Else, the SMT solver iterates further through the solutions of the SAT instance.
These inner workings of a general-purpose solver reveal two difficulties. One of them is inherent in the problem. Assume we want to check satisfiability of the following formula: there exists a point in a given input domain such that a particular output neuron is smaller than any other output neuron. This is the typical query to an SMT solver to check for robustness of a classification neural network (TXT19). If the formula is satisfied, you immediately obtain a counterexample of the robustness property. Assume further that the formula is not satisfiable and that the neural network consists of 2^k many subfunctions in the input domain as given by the number of ReLU neurons k. Then the SMT solver will need to iterate over all of them to establish robustness. This is inefficient for mid-sized neural networks and quickly becomes impossible to calculate due to the problem's NP-completeness.
The other problem arises from the generality of SMT solvers. They do not incorporate dependencies between different ReLU neurons and thus always need to explore all possible ReLU activation patterns until they find a feasible solution to the formula or can return *unsat*. However, some neurons may never actually be activated and some pairs of ReLU neurons may only be activated together. Therefore, when iterating over all potential activation patterns, the SMT solver does a lot of unnecessary work because it does not exploit these neural network specifics. As a result, other methods built upon general-purpose solvers decreas-

ing the verification runtime substantially. A further discussion of these methods can be found in Chapter 3.

Related Work 3

This chapter embeds the novel methods developed in this thesis into its broader scientific background. We will give a brief history of adversarial attacks as well as of software verification as these are the subjects motivating our research. Moreover, we will explore the realm of verification methods that can be used for neural networks, as many of them tackle scalability. Note that this chapter is meant to give a general perspective. Therefore in each of the chapters describing a particular method, we add a further small chapter for its specifics.

3.1 Supervised Adversarial Attacks

The topic of adversarial attacks is not a new one. Though it has gained tremendous momentum in the machine learning community, its roots date back to at least as far as 2004 when Dalvi et al (DDM+04) introduced the term of *Adversarial Classification*. While their work regards any classification algorithm, the seminal paper "Intriguing properties of neural networks" by Szegedy et al. in 2013 (SZS+14) shifted the focus towards neural networks. They demonstrated that neural networks can exhibit incredibly unstable behavior with the following experiment.
Assume a neural network classifier on ImageNet (DDS+09). We can add noise to a correctly classified image of a panda to thereafter show a gibbon - at least according to the neural network. We as humans though still see the same picture of a panda because the noise is imperceptible to us. One important remark is to be made: adversarial attacks are a subjective notion in the following sense. It depends on humans not being able to tell the difference between the original image and the adversarial attack based on it. Therefore adversarial attacks are usually defined as differently classified objects which are close to the original one. The closeness then implicitly captures a human's incapability to spot a difference. However, adversarial attacks depend on humans unanimously declaring the adversarial attack as misclassified.
Of course, it is easy to find some other example that is classified differently. It is much more difficult, though, to find an input close to a given, correctly classified

one. Therefore several methods trying to find the closest adversarial attack have been proposed. Most of them are based in one way or another on the loss function's gradient with respect to the input variables determining the direction in which to search for an adversarial attack. Szegedy et al. (SZS+14) based their procedure on a constrained L-BFGS optimization incorporating not only the neural network's gradient with respect to the input but also the second derivatives via the Hessian matrix. Goodfellow et al. (GSS15), on the other hand, opted for a faster method - the **F**ast **G**radient **S**ign **M**ethod (FGSM) - projecting the gradient into the corner of a hypercube with radius ε. Other approaches focus on manipulating the neural network's logits (PMJ+16) instead of its loss function, projecting onto the decision boundary (MFF16), keeping the number of input dimensions changed small (SVS19) or rephrase finding adversarial attacks as a new optimization problem (CW17b) solved with the Adam (KB15) optimizer. None of the aforementioned methods yield guarantees on how close the adversarial attacks are to the original input. These can only be obtained by finding adversarial attacks utilizing a neural network verification method (KBD+17), which exhaustively searches for them and can thus approximate them arbitrarily well (see Chapter 4).

As can be seen by this brief overview, the topic of adversarial attacks for unsupervised neural networks has been mostly neglected. While there are neural networks for unsupervised tasks such as anomaly detection (SY14), clustering (SSL+18) or dimensionality reduction (HS06), almost all research effort is directed towards supervised classification neural networks. In these cases, adversarial attacks are based on labeled data defining classification borders which can be overstepped. For unsupervised neural networks, though, it is not so clear what an adversarial attack could be. What exactly does it mean to be an adversarial attack for, example given, the task of dimensionality reduction? This thesis suggests one possible answer for the paradigmatic case of autoencoders (HHWB02). We effectively lift the notion of adversarial attacks from labels to a more general concept of quality assessment. For the case of autoencoders, this means that we base our definition of unsupervised adversarial attack on one of their performance measure: the reconstruction error.

3.2 Defending Against Adversarial Attacks

Szegedy et al. (SZS+14) launched a large amount of research on adversarial attacks into different directions. On the one hand, there are papers crafting more and more types of attacks against any given neural network (HKR+20b; BCM+13) as shown above. But beyond those, the threat of adversarial attacks served as motivation to make neural networks resilient against them. Adversarial training (MMS$^+$18) trains the neural network on adversarial attacks created for this purpose. While this decreases the vulnerability against the type of adversarial attack it was trained on, other adversarial attacks are still undefended against. Masking (ACW18) intends to decrease the attacker's capability by revealing less information to the attacker. In essence, they step away from whitebox attacks and allow the attacker to rely solely on input-output pairs obtained by, for example, a web API. However, it turns out that a blackbox model suffices to create adversarial attacks as they transfer between different models (DLP$^+$18). Therefore we can create a substitute model based on which we can craft the adversarial attack (PMG$^+$17).

Our thesis contributes to this area in multiple ways. First, we address adversarial training in Chapter 4, reducing the worst-case-error. Secondly, our training regularization in Chapter 5 increases the model's provable robustness and decreases the amount of unsupervised adversarial attacks. This is in line with other approaches to increasing resilience through the training process. Finally, we address the question of how models can be made more resilient as a post-processing step. By building on the particular structure of ensemble models we can, similarly to (SK01), adjust the given model for our purposes. In our case, we change it in such a way that it becomes more robust against adversarial attacks.

3.3 Neural Network Verification

As we have seen before, any attempt to defend against adversarial attacks may not result in enduring security: for every new defense, a new attack is created and most defenses work only for a subset of attacks (TB19). Therefore the only way to ensure lasting reliability for a neural network is to prove that there can be no adversarial attack on it. In this case, irrespective of the attacker's method, he can simply not craft an attack. To this end, we utilize neural network verification analyzing a given neural network with respect to a given property.

3.3.1 Use Cases of Neural Network Verification

Neural network verification is inspired by formal methods developed for general software verification (BM07) and essentially aims at proving that a given neural network works as intended. This can mean much more than just defending against adversarial attacks and includes properties such as denoising (Ben22a),

or correct airplane steering (OPM$^+$19).
Moreover, on the technical level more than just robustness against adversarial attacks can be verified. We have noted that adversarial attacks arise from the very unstable behavior of neural networks. Thus Lipschitz continuity (O'S06) is explored and verified (WZC$^+$18; ZZH19) as it limits the amount by which the neural network's output can change as its input changes. Furthermore, safety properties can be encoded via so-called reachability analysis (XTJ18), asking what are the potential outputs for a given set of inputs. Sometimes verification procedures are even embedded in other verification procedures. For example, one can use them to bound the values of particularly important ReLU neurons, thereby refining the analysis of the actual neural network. Also, one may use smaller, more specialized neural networks within a larger one and verify a controller, assuming that the smaller one works correctly (XKN22).

3.3.2 A Different Evaluation Paradigm

The most important distinction between neural network verification and other machine learning test schemes lies in the space considered. The usual way of testing machine learning models works using a finite test dataset for which some evaluation metric is calculated (GBC16; Bis07). However, from a finite test dataset, we can obviously not conclude a guarantee over an infinite value domain from which adversarial attacks can be crafted. Thus, the traditional test setting is not suited for formal proofs about a given neural network's behavior.
Instead, neural network verification requires a way to analyze the mathematical function the neural network conveys. However, neural networks are extremely complicated functions impeding a simple analysis of their behavior. Therefore specialized tools such as for example constraint-based solvers are required to model the network and allow for an automated verification.

3.4 Scaling Up the Verification Process

One severe caveat verification procedures have is their lack of scalability to large-scale neural networks. As the problem of neural network verification is NP-complete (WZC$^+$18) it becomes computationally intractable. However, the need to verify neural networks does, of course, not end with small-scale neural networks. Therefore considerable effort has been made to speed up the verification procedure resulting in different research branches[1]. This is further fueled by

[1]While we present these methods separately, current state-of-the-art methods such as (FMJV22) combine multiple approaches

the VNN competition (MBB$^+$23) rewarding the number of verification instances which can be solved correctly. This highlights again that scalability is regarded as one of the main challenges in neural network verification. In the following, we will briefly discuss the main approaches currently used and give examples for each idea to speed up the verification process.

3.4.1 Exact Verification Methods

Exact verification methods are both sound and complete. This means that they will either return a proof of the property under scrutiny or a counterexample violating that property. These two types of results make these methods quite versatile as both of them can be used. At their core, exact verification employs constraint-based methods, most notably SMT solvers (KBD$^+$17) and MILP (TXT19) solvers. They encode the neural network as a logical conjunction of constraints and can only be used for piecewise affine neural networks. On top of that, many heuristics have been implemented, speeding up the verification process significantly. Planet (Ehl17) transfers the ReLU function into constraints handled by the SAT solver and incorporates a linear approximation of the overall neural network behavior. ReluPlex (KBD$^+$17) on the other hand integrates the ReLU function into an LP formulation of the problem splitting on them only if the constraints cannot be satisfied by pivot operations. Marabou (WOZ$^+$20) builds on top of ReluPlex and combines bound tightening for ReLU neurons, a sophisticated branch-and-bound procedure as well as a heuristic on which ReLU neurons to branch.
As these tools were developed and released during the research on the different topics, our experiments are based on different solvers. The experiments done in Chapter 4 are based on the Z3 SMT solver (dMB08), whereas all experiments that followed are based on the Marabou framework due to its versatility and ease of use.

3.4.2 Approximate Verification Methods

The exactness of the aforementioned methods comes at the price of high computational complexity, making them suitable only for small networks. Therefore approximate methods have been developed, sacrificing completeness for speed. Many verification problems essentially ask which outputs the neural network can produce, given a set of inputs. If all these possible outputs share a particular property, such as being of the same class, the neural network is deemed safe. Therefore these methods overapproximate the outputs the neural network might produce. If, for a set of inputs, even the overapproximation of the outputs shares the property of interest, then - given those inputs - surely all possible outputs of the neural network will. The CROWN algorithm (ZWC$^+$18) extending the FastLin algorithm proposed by (WZC$^+$18), for example, yields an upper and a lower bound function for each output neuron by recursively propagating them through the network. The ERAN framework (SGPV19) on the other hand implements so-called *abstract interpretation* which propagates sets given by intervals, zonotopes or polytopes directly through the network. Finally, PRIMA (MMS$^+$22) calculates optimal bounds for a subset of highly interdependent ReLU neurons, thereby leaning towards exact verification methods. The main challenge in all these approximate verification methods is to cleverly capture dependencies among different neurons. Lastly, note that one may also emply an anytime algorithm: one that maintains and refines an approximate solution while converging to an exact one (RWS$^+$19).

3.4.3 Changing the Perspective

All of the aforementioned methods try to improve the verification process itself. That is, they do not regard the verification instance as under their control. While in many applications this may be a reasonable assumption, in this thesis, we explore how we can adjust the neural network such that is becomes easier to be verified.

By incorporating a regularization term into training, we effectively reduce the number of affine subfunctions - the main driver for neural network complexity. This results in a significant speedup to a non-regularized network using the same verification engine as we will demonstrate in Chapter 5.

Our second approach to alleviating the scalability issue is based on exploiting an ensemble structure. We propose to verify a large set of small neural networks instead of one large neural network as, in that case, verification runtime grows linearly with the number of networks and the verification process becomes more parallelizable.

Thus, by giving these two new perspectives, we can add to the vast body of research on scalability. As a further advantage, all the aforementioned approaches for given neural networks can, of course, be combined with our proposed solutions.

Adversarial Attacks on Autoencoders 4

Autoencoders are widely used for many unsupervised learning tasks such as cluster analysis (CGG19), compression (MCSK17), anomaly detection (SY14), as well as a variety of pre-processing steps (Gon16; MCSK17; PS14) in other machine learning pipelines. The general assumption is that data can be compressed into a lower dimensional latent space by an encoder function extracting the most relevant features of the data distribution. From this latent representation, the decoder tries to reconstruct the original input. As the latent representation is an information bottleneck, the autoencoder's input deviates from its output. Typically the autoencoder reconstructs better in dense regions (regions with many training examples) than in regions with few training examples (SY14), giving rise to its application in anomaly detection. Moreover, even the small errors in reconstruction that inevitably occur in dense regions are a desirable property as they allow it to be used, example given, for denoising. At the same time, it is necessary to control these errors for all points in dense regions because otherwise, the result - whether it is the latent representation or the reconstruction - is less useful. To this end, current approaches to assess autoencoders either measure the mean square error (MSE) internally on the unsupervised training data or external performance on some supervised downstream application, such as classification performance.

However, a major shortcoming of these approaches is that they cannot provide a formal guarantee in terms of the maximum deviation between input and output of the autoencoder as it is evaluated on a finite number of inputs. We are not aware of any existing scheme to calculate the largest error of an autoencoder in an infinite input space. This lack of formal quality guarantees for autoencoders leads to a very limited applicability of such unsupervised learning schemes for safety-critical applications. For instance, it is particularly important to consider the maximum deviation when working with data containing clusters. In such situations, the autoencoder should not mix up the clusters because otherwise, the autoencoder's results are meaningless. If the maximum deviation for the respective clusters is small enough, though, the autoencoder is guaranteed to keep the clusters separated.

To address this and other shortcomings of unsupervised learning with autoencoders, we provide the first methodology to bound an autoencoder's **w**orst-**c**ase-**e**rror (**wce**) in a safety-critical region. As a first step towards this goal, we define the notion of *unsupervised adversarial attacks*, which are inputs (not necessarily contained in the training or test data) on which the autoencoder's error exceeds a user-defined threshold. Then we define the worst-case-error of an autoencoder as the largest error that can possibly manifest. Since we cannot expect to find a global maximum of the error (as there is no reason for the error itself to be bounded), we restrict our search to user-defined regions with an infinite value domain of the input space. We leave this region as a parameter to be provided by the user as it clearly depends on knowledge about the use case at hand, characteristics of the training data, or other domain-specific information.
Based on the constraint-solving framework given in Chapter 2, we reduce the problem of finding an unsupervised adversarial attack to a satisfiability check of a formula in Real Arithmetic. Thus we apply highly-optimized, off-the-shelf satisfiability modulo theory (SMT) solvers, which can effectively reason about the infinite domains and, hence, can prove the existence or non-existence of unsupervised adversarial attacks. Once we have found an unsupervised adversarial attack, it serves as a lower bound for the worst-case-error. Moreover, a simple binary search allows us to approximate the worst-case-error arbitrarily well. Note that naive approaches, such as sampling, cannot provide an upper bound on the worst-case-error of an autoencoder as an exhaustive search of the input space is intractable. Moreover, our experimental evaluation shows that sampling often underestimates that worst-case-error.
We demonstrate the effectiveness of our QUGA (QUality Guarantees for Autoencoders) approach and evaluate our quality guarantees for unsupervised learning on a synthetically created dataset as well as on a real dataset. In both cases, we can find unsupervised adversarial attacks as well as formal quality guarantees by lower and upper error bounds in safety-critical regions. As the worst-case-error is a very general concept, we also discuss follow-up works based on it. First, we show how it can be leveraged to get so-called error landscapes describing how the reconstruction error differs across different regions. Secondly, for the task of anomaly detection, we extract a region in which all normal inputs are, thereby using the worst-case-error in an interpretation context. Lastly, we demonstrate how the worst-case-error can be reduced by means of adversarial training.

4.1 Related Work

Adversarial Attacks in Supervised vs. Unsupervised Learning:
In the area of supervised learning, adversarial attacks have been widely studied

(GSS15; DDM$^+$04; SZS$^+$14). While common definitions of adversarial attacks rely on the robust separation of class labels, we aim at unsupervised learning without given labels. Therefore supervised definitions do not cover the unsupervised learning case. Similarly existing approaches of adversarial attacks in unsupervised learning focus on a particular task such as image retrieval (FCDX20) assuming that there is a notion of a wrong output. In contrast to these approaches we define adversarial attacks directly in terms of the intrinsic learning objective of autoencoders which is - as reflected by its loss function - approximating the identity function.

Empirical Quality Assessment vs. Formal Guarantees:
Common evaluation schemes for autoencoders do an empirical quality assessment based on a given set of training data. The variety of quality measures ranges from simple average MSE to stability and robustness measures (LRM$^+$12; VLL$^+$10; MSY$^+$09). All of these measures have in common that they rely on the given training or test data. In contrast to such empirical evaluation, many safety-critical applications require formal guarantees explicitly also on unseen data. We propose such formal guarantees for trained autoencoders. Given a safety-critical data region, our method is able to either find an adversarial attack or prove that such an attack does not exist.

External vs. Internal Evaluation:
Common external evaluation uses, for example, the classification quality of a downstream task following the autoencoder as an indirect measure of quality for the autoencoder itself. However, by evaluating multiple tasks, one can never assign the quality assessment solely to a single task. This process is prone to mix-up and dependency effects between the different stages of the process. Therefore, we belief that the modular evaluation of individual tasks is an additional requirement for safety-critical systems. Such a design-by-contract has been successfully established in modular software verification (BM07). Similarly, we propose the first formal guarantee of an autoencoder given by an upper bound on the largest error in the entire input domain.

Verification of Neural Networks:
Our work is related to formal methods and verification of neural networks in general (see for example (BIL$^+$16; KBD$^+$17)). However, most of the research in this area focuses on the problem of finding adversarial attacks in supervised learning tasks and lacks formal insights for unsupervised learning. In contrast, our algorithm searches for unsupervised adversarial attacks. It does so by reducing the problem to a series of satisfiability checks in a Real Arithmetic and applies a highly-optimized Satisfiability Module Theories (SMT) solver as computational back-end to perform these checks. We have implemented a prototype of our algorithm on top of the Z3 SMT solver (dMB08) which provides a convenient

API and is one of the most popular tool in the domain of software verification. For larger, real-world scenarios, however, one would clearly use a solver that is optimized for constraints arising from feed-forward networks, such as ReluPlex (KBD$^+$17), Marabou (WOZ$^+$20) or Planet (Ehl17). Apart from constraint solving, other techniques from the area of deductive software verification have been used for finding adversarial attacks in supervised learning and proving robustness properties of feed-forward neural nets. The perhaps most popular approach is abstract interpretation (GMD$^+$18; SGPV19). However, abstract interpretation inherently overapproximates the behavior of the neural network and, hence, can only be used to prove safety properties. However, neither our unsupervised adversarial attacks nor our worst-case-error of an autoencoder can be achieved by their safety properties.

4.2 *QUGA*: Problem Statement

In general, an autoencoder tries to reproduce its input while propagating it through a latent space which typically has fewer dimensions than the input/output space. This latent space serves as an information bottleneck and, hence, introduces errors to the identity function the autoencoder is supposed to learn. However, most applications of autoencoders rely on a good approximation of the identity function, and we are naturally interested in quantifying its error. More precisely, our goal is to give formal guarantees in terms of the maximum deviation from the identity function.
As a first step towards this goal, we define the notion of *adversarial attacks on autoencoders*. Intuitively, such adversarial attacks are inputs on which the "distance" between the input and the output of the autoencoder is larger than a (user-defined) threshold $\varepsilon > 0$. Given the lack of definitions for adversarial attacks in unsupervised learning (and in particular for autoencoders), we define adversarial attacks based on an abstract metric *dist*, which maps two data points to a non-negative real number. However, we stress that the exact metric is not important for our definition (for example, any L_p-norm could be used) because all autoencoders share the goal of reconstructing the input.

Definition 4.2.1 (ε-adversarial attack). *Let $f\colon \mathbb{R}^N \to \mathbb{R}^N$ be an autoencoder, dist*$\colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}_+$ *a metric, and $\varepsilon > 0$. An ε-*adversarial attack *is a point $x \in \mathbb{R}^N$ such that*

$$\mathit{dist}(x, f(x)) > \varepsilon$$

(*a point on which the input and output of f deviate more than ε*).

Note that our definition of ε-adversarial attacks is not restricted to inputs in the training or test sets but allows any input $x \in \mathbb{R}^N$. This property makes finding ε-adversarial attacks a very challenging task and, in contrast to traditional internal evaluation (for example, the mean square error) on training data, to surely find an adversarial attacks or to conclude that no ε-adversarial attack exists is a computationally hard problem.
In the context of safety-critical systems, however, it is not enough to identify individual ε-adversarial attacks, but it is necessary to know the worst-case-error (the largest error) a given autoencoder produces. Of course, we cannot expect to find a global maximum of the error as there is no reason for the error itself to be bounded. Therefore, we restrict the region for which we want to find a bound on the error. This region depends on knowledge about the use case at hand, characteristics of the training data, or other domain-specific information. Thus, we leave it as a parameter to be provided by the user.

Definition 4.2.2 (Worst-case-error (*wce*) of autoencoders). *Let $f \colon \mathbb{R}^N \to \mathbb{R}^N$ be an autoencoder, $dist \colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}_+$ a metric, and $A \subseteq \mathbb{R}^N$ an (infinite) safety-critical region of inputs. Then, the* worst-case-error of *f* in *A is defined as*

$$wce(f, A) = \sup \{dist(x, f(x)) \in \mathbb{R}_+ \mid x \in A\}$$

(*the largest deviation of an input in the region A from the output*).

Definition 4.2.2 serves as our novel *quality criterion* for autoencoders that reflects how good the identity function is learned in the specific region of interest. Our *wce*-definition is inspired by many areas of reliable system design, including soft- and hardware verification, as $wce(f, A)$ guarantees that a system f employed in a safety-critical region A stays within its design parameters. Furthermore, our notion of *wce* overcomes limitations of classical quality metrics that are defined on finite training data only. We actively design *wce* for typically infinite data domains of safety-critical regions. In total, this leads us to the main problem statement, which we call *QUGA: QUality Guarantees for Autoencoders*.

Problem 4.2.1 (QUGA: Quality Guarantees for Autoencoders). *Given an autoencoder $f \colon \mathbb{R}^N \to \mathbb{R}^N$, a metric $dist \colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}_+$, and a region $A \subseteq \mathbb{R}^N$, compute $wce(f, A)$.*

In general, computing the worst-case-error is a very challenging problem as it involves reasoning about an infinite number of inputs (not just training data) and does not make any assumption about the autoencoder, the metric or the region. In the following section, we consider a restricted version of Problem 4.2.1 and show how a reduction to constraint solving can be leveraged for this restricted version.

4.3 Solution Framework

In this section, we provide a framework for computing ε-adversarial attacks and the worst-case-error of autoencoders. To make these problems computationally tractable, we consider a restricted version of Problem 4.2.1. The following restrictions are designed in such a way that the solution framework remains applicable to a wide range of autoencoders used in practice. Essentially, we restrict ourselves to piecewise affine neural networks and ensure that Problem 4.2.1 can be formulated using a logical conjunction of linear constraints:

1. We assume that the neurons of the autoencoder have Linear or ReLU (Rectified Linear Units) activation functions[1].
2. We assume the metric to be the L_1 or L_∞-norm.
3. We assume the safety-critical region A to be a finite union of convex compact polytopes (each polytope is bounded, closed and an intersection of half-spaces in the input space $\mathbb{R}^N$).
4. We approximate the worst-case-error up to a user-defined accuracy $\delta > 0$ because our framework can find ε-adversarial attacks for fixed ε only.

As the next step, we formally introduce autoencoders (Section 4.3.1). In Section 4.3.2, we then show how the existence of an ε-adversarial attack can be phrased as a satisfiability problem in Linear Real Arithmetic, one of the theories supported by the SMT framework. This allows us to use highly-optimized SMT solvers to do a symbolic search on the infinite input space. In Section 4.3.3, we finally provide an effective method to approximate the worst-case-error of an autoencoder by repeatedly solving the easier problem of determining the existence of ε-adversarial attacks for different values of ε.

4.3.1 Autoencoders

Intuitively, an *autoencoder* is a standard feed-forward neural network just as described in Section 2.1. However, they are supposed to reconstruct their input as reflected by the loss function on which they are trained[2]:

$$L_{recon}(x) = \|f(x) - x\|_2^2.$$

[1]Note that other activation functions used for convolutional and pooling layers could be used here too. We choose to use only Linear and ReLU activation functions for the sake of simplicity

[2]There are other loss functions as well, but we restrict ourselves to autoencoders trained with this reconstruction loss.

Therefore the dimensions of the input space and the output space are the same. Furthermore, autoencoders can be divided into the encoder mapping inputs to a so-called latent space and the decoder trying to inverse that map. However, as the latent space typically has fewer dimensions, this inversion does not work everywhere, introducing reconstruction errors. Note here, that for training we use the L_2 loss function which is common practice for autoencoders while for their analysis and the worst-case error we rely on the L_∞ loss function. Figure 4.1 on Page 39 shows a very small example of an autoencoder consisting of two-dimensional input/output layers and a single-dimensional latent space with no other hidden layers in between.
We briefly recall some notations from Section 2.1. For an autoencoder f, we use $L+1$ to denote the number of layers and $l_k \in \mathbb{N} \setminus \{0\}$ with $k \in \{0, \dots, L\}$ to represent the number of neurons in Layer k. Layer 0 is the *input layer*, layer L the *output layer*, and the remaining layers are the *hidden layers*. As stated above, we have that $l_0 = l_L = N$.

4.3.2 Identifying Adversarial Attacks

Let us now describe how to translate the problem of finding an ε-adversarial attack of an autoencoder f into LRA. At its core is a formula φ^f that encodes the function computed by f in LRA. This formula ranges over variables $x_{i,j}$ for neuron j in layer i. Its precise description can be found in Section 2.2.2. Moreover, we add further constraints in the form of formulas φ^A and $\varphi_\varepsilon^{dist}$ which encode the input region A and the metric (including the existence of an ε-adversarial attack) respectively. The resulting problem encoding is then given by the conjunction

$$\varphi_{\varepsilon,A}^{f} := \varphi^f \wedge \varphi^A \wedge \varphi_\varepsilon^{dist} \tag{4.1}$$

which is satisfiable if and only if an ε-adversarial attack exists. Moreover, recall that - if satisfiable - SMT solvers return an interpretation that assigns a value to each neuron in our case. This is very helpful because a satisfying interpretation of $\varphi_{\varepsilon,A}^{f}$ carries sufficient information to extract an ε-adversarial attack. Let us now describe $\varphi_\varepsilon^{dist}$ and φ^A in detail.

Encoding the Region:

Recall that we assume that the safety-critical region A in which to search for ε-adversarial attacks is provided as a finite union of compact convex polytopes. Formally, a convex polytope $\mathcal{P}$ is the finite intersection of half-spaces $\mathcal{H}_i$ of the form

$$\mathcal{H}_i = \{x \in \mathbb{R}^N : \sum_{j=1}^{N} a_{i,j} x_j \leq c_i\}$$

for $a_{i,j}, c_i \in \mathbb{R}$. Thus, restricting the search space for ε-adversarial attacks to a convex polytope $\mathcal{P}$ consisting of m half-spaces can simply be achieved by the formula

$$\psi_{\mathcal{P}} := \bigwedge_{1 \leq i \leq m} \Big[\sum_{1 \leq j \leq l_0} a_{i,j} x_{0,j} \leq c_i \Big].$$

Here, "$\wedge$" refers to the intersection of half-spaces given by the conjunction in the language of SMT solvers. Moreover, the final formula is then the disjunction

$$\varphi^A := \bigvee_{\mathcal{P} \in A} \psi_{\mathcal{P}}$$

for all polytopes $\mathcal{P}$ constituting to the given region A. Here, "$\vee$" gives the union over the different polytopes - again translated for SMT solvers.

Encoding the Existence of an ε-Adversarial Attack:

We will first describe $\varphi_{\varepsilon}^{dist}$ for *dist* being the L_∞-distance derived from the L_∞-norm:

$$dist(x, y) = \|x - y\|_\infty = \max_{i \in \{1,\dots,N\}} |x_i - y_i|$$

In the L_∞-norm, an input is an ε-adversarial attack if there exists a dimension $i \in \{1, \dots, N\}$ in which the absolute value of the difference of the input and the output in this dimension is larger than ε. This can be expressed in LRA by

$$\varphi_{\varepsilon}^{dist} := \bigvee_{1 \leq j \leq N} \Big[[x_{0,j} - x_{L,j} > \varepsilon] \vee [x_{L,j} - x_{0,j} > \varepsilon] \Big].$$

Before we continue with the final formula $\varphi^f_{\varepsilon,A}$, let us briefly illustrate the constraints generated so far using an example. Recall that the general encoding of the neural network can be found in Section 2.2.2.

Example 4.3.1. *Consider the simple autoencoder (with ReLU-activation) in Figure 4.1, consisting of two neurons in the input layer, one neuron in the single hidden layer, and two neurons in the output layer. Moreover, assume that we are given one polytope $\mathcal{P}$ consisting of the intersection of four half-spaces $-1 \leq x$, $x \leq 1$, $-1 \leq y$, and $y \leq 1$ (a*

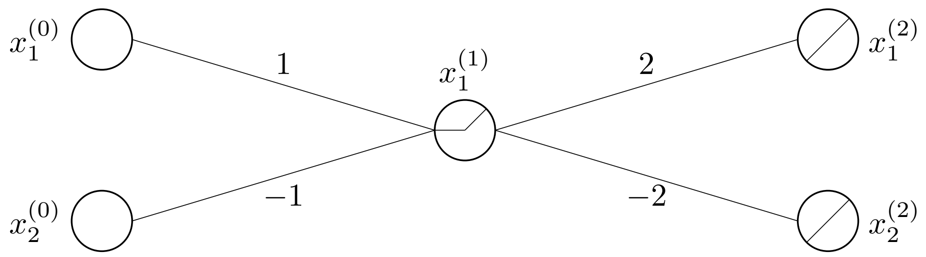


Figure 4.1: A small example of an autoencoder.

unit box around the origin). Then, the formulas φ^f, φ^A, and $\varphi_\varepsilon^{dist}$ are given by

$$\begin{aligned}\varphi^f := \; & [y_{1,1} = x_{0,1} + (-1)x_{0,2}] \\ & \wedge[x_{1,1} = \mathsf{ite}(y_{1,1} < 0, 0, y_{1,1})] \\ & \wedge[x_{2,1} = 2x_{1,1}] \\ & \wedge[x_{2,2} = (-2)x_{2,2}]\end{aligned}$$

$$\begin{aligned}\varphi^A := \; & [x_{0,1} \leq 1] \\ & \wedge[x_{0,1} \geq -1] \\ & \wedge[x_{0,2} \leq 1] \\ & \wedge[x_{0,2} \geq -1]\end{aligned}$$

$$\begin{aligned}\varphi_\varepsilon^{dist} := \; & [x_{0,1} - x_{2,1} > \varepsilon] \\ & \vee[x_{2,1} - x_{0,1} > \varepsilon] \\ & \vee[x_{0,2} - x_{2,2} > \varepsilon] \\ & \vee[x_{2,2} - x_{0,2} > \varepsilon]\end{aligned}$$

Finally, we combine all constraints generated so far into a single formula $\varphi_{\varepsilon,A}^f := \varphi^f \wedge \varphi^A \wedge \varphi_\varepsilon^{dist}$. As Theorem 4.3.1 states, this formula indeed expresses the existence of an ε-adversarial attack of the autoencoder f in the region A.

Theorem 4.3.1. *let f be an autoencoder, A a region, dist the L_∞ metric, $\varepsilon > 0$, and $\varphi_{\varepsilon,A}^f$ as defined above. Then, the following two properties hold:*

1. *If an ε-adversarial attack exists in the region defined by A, then $\varphi_{\varepsilon,A}^f$ is satisfiable.*
2. *If $\varphi_{\varepsilon,A}^f$ is satisfiable, say by the interpretation $\mathcal{I}$, then $(\mathcal{I}(x_{0,1}), \ldots, i(x_{0,l_0}))$ is an ε-adversarial attack in the region defined by A.*

Proof. To prove the first statement, we formulate a satisfying interpretation $\mathcal{I}$ for $\varphi^f_{\varepsilon,A}$, using an ε-adversarial attack, say $v^\varepsilon = (v_{0,1}, v_{0,2}, \cdots v_{0,l_0})$, in the region A. To do that, we assign values to each variable in $\varphi^f_{\varepsilon,A}$ in the following manner:

- We set $\mathcal{I}(x_{0,i}) = v_{0,i}$ for each $i \in \{1, \cdots, l_0\}$.
- For v^ε propagating through f, we set $\mathcal{I}(x_{k,j}) = v_{k,j}$ for each $i \in \{1, \cdots, l_k\}$ and $k \in \{1, \cdots, L\}$ where $v_{k,j}$ is the output of Neuron j in Layer k.

We prove that the above interpretation indeed satisfies $\varphi^f_{\varepsilon,A}$ as a conjunction of three formulas φ^f, φ^A and $\varphi^{dist}_\varepsilon$. First, we observe that for a linear neuron $v_{k,j}$

$$v_{k,j} = \sum_{i=1}^{l_k - 1} W^{(k)}_{i,j} v_{k-1,i} + b^{(k)}_j$$

for $k \in \{1, \cdots, L\}$, $j \in \{1, \cdots, l_k\}$. This indicates that the formulas $\psi_{k,j}$ as defined in Equation 2.5 are satisfied. Similarly, if $v_{k,j}$ is a ReLU neuron the formula $\psi_{k,j}$ as defined in Equation 2.6 is satisfied. Consequently, φ^f, being a conjunction of the former formulas, is satisfied.

Secondly, v^ε belongs to the region A and hence, satisfies φ^A. Lastly, following the definition of ε-adversarial attack, we have that at least in one dimension $i \in \{1, \dots, l_0\}$ the absolute difference between $v_{0,i}$ and $v_{L,i}$ is greater than ε. Therefore

$$\varphi^{dist}_\varepsilon = \bigvee_{1 \le i \le l_0} [v_{0,i} - v_{L,i} > \varepsilon] \vee [v_{L,i} - v_{0,i} > \varepsilon],$$

is satisfied as well.

To prove the second statement, we derive a ε-adversarial attack in the region defined by A from a satisfying interpretation $\mathcal{I}$ of $\varphi^f_{\varepsilon,A}$. For the sake of brevity, let $\mathcal{I}(x_{k,j}) = v_{k,j}$ for $k \in \{0, \cdots, L\}$ and $j \in \{1, \cdots, l_k\}$ and $v^\varepsilon = (v_{0,1}, v_{0,2}, \cdots, v_{0,l_0})$.

Now, we prove that when v^ε is propagated through f, the output of Neuron j in Layer k is $v_{k,j}$. We prove this fact by induction on the layers of f. For the base case, observe that the output of Neuron j in Layer 0 is $v_{0,j}$ since $(v_{0,1}, \cdots, v_{0,l_0})$ is an input to f. For the inductive step, observe that for a linear neuron $v_{k,j}$, we have

$$v_{k,j} = \sum_{i=1}^{l_k - 1} W^{(k)}_{i,j} v_{k-1,i} + b^{(k)}_j$$

because $\psi_{k,j}$ is satisfiable under $\mathcal{I}$. Now, due to induction hypothesis, $v_{k-1,i}$ is the output of Neuron i in Layer $k-1$ for $i \in \{1, \cdots, l_{k-1}\}$. The above relation exactly combines the output from Layer $k-1$ as an autoencoder would do and ensures that $v_{k,j}$ is the output of Neuron j in Layer k. Likewise, if $v_{k,j}$ is a ReLU neuron,

Algorithm 1: Computing *wce* up to accuracy δ

```
Input: Autoencoder $f$, Region $A$, metric $dist$, start value $\varepsilon_0 > 0$, accuracy
       $\delta > 0$
1 $\varepsilon_{\text{low}} = \varepsilon_{\text{up}} = \varepsilon_0$
2 Construct $\varphi^f_{\varepsilon_0,A}$ and check satisfiability using an SMT solver
3 if $\varphi^f_{\varepsilon_0,A}$ is satisfiable then
4     Increase $\varepsilon_{\text{up}}$ by $\varepsilon_{\text{up}} * 2$ until $\varphi^f_{\varepsilon_{\text{up}},A}$ becomes unsatisfiable
5 else
6     Decrease $\varepsilon_{\text{low}}$ by $\varepsilon_{\text{low}}/2$ until $\varphi^f_{\varepsilon_{\text{low}},A}$ becomes satisfiable or $\varepsilon_{\text{low}} < \delta$ (in
        which case return $\varepsilon_{\text{low}}$)
7 $\varepsilon^\star \leftarrow$ Binary-search$_{f,A,dist}(\varepsilon_{\text{low}}, \varepsilon_{\text{up}}, \delta))$          // involves calls to SMT
    solver
8 return $\varepsilon^\star$
```

the corresponding formula $\psi_{k,j}$ is satisfied, yielding the correct calculation for the autoencoder.

Moreover, φ^A being satisfiable under interpretation $\mathcal{J}$ implies that v^ε belongs to the region defined by A. Finally, since $\varphi^{dist}_\varepsilon$ is satisfiable under $\mathcal{J}$, we have

$$\bigvee_{1 \le i \le l_0} \Big[[v_{0,j} - v_{L,j} > \varepsilon] \vee [v_{L,j} - v_{0,j} > \varepsilon] \Big]$$

indicating that v^ε is an ε-adversarial attack.
Note that while we prove the theorem for the L_∞ metric only the same statements hold true for other metrics, which can be encoded for SMT solvers. $\square$

Theorem 4.3.1 now suggests a simple procedure to find ε-adversarial attacks: simply construct $\varphi^f_{\varepsilon,A}$, run an SMT solver, and return $(\mathcal{J}(x_{0,1}), \ldots, \mathcal{J}(x_{0,l_0}))$ if a satisfying assignment $\mathcal{J} \models \varphi^f_{\varepsilon,A}$ exists. However, the SMT solver might report that $\varphi^f_{\varepsilon,A}$ is unsatisfiable. In this case, Theorem 4.3.1 guarantees that no ε-adversarial attack exists. We exploit this property now to approximate the worst-case-error of an autoencoder.

4.3.3 Approximating the Worst-Case-Error

We now provide an algorithm for approximating the worst-case-error of an autoencoder. Our algorithm, which is sketched in pseudocode in Algorithm 1, is based on the method for finding ε-adversarial attacks from Section 4.3.2. Apart

from the autoencoder itself, the safety-critical region, and a metric, it expects two additional arguments: a start value $\varepsilon_0 > 0$ for the search and an accuracy value $\delta > 0$. The start value ε_0 is used as an initial estimate for $wce(f, A)$ and can be either initialized arbitrarily or based on domain knowledge. The accuracy, on the other hand, is a measure of how close the output of Algorithm 1 is to the actual value of $wce(f, A)$. A smaller δ results in a more precise approximation of $wce(f, A)$, but it also increases the computation time.
Algorithm 1 uses a binary search to find a sufficiently close approximation of $wce(f, A)$ (see line 7). To this end, it uses two values $\varepsilon_{\text{low}} < \varepsilon_{\text{up}}$ for which it maintains the invariant that (a) there exists an ε_{low}-adversarial attack and (b) there does not exist an ε_{up}-adversarial attack in the given region. Hence, $wce(f, A)$ is guaranteed to lie in the interval $[\varepsilon_{\text{low}}, \varepsilon_{\text{up}}]$ (unless there exists only an ε-adversarial attack for $\varepsilon < \delta$; we leave out this side case in our analysis). The initial values for ε_{low} and ε_{up} are obtained by starting with ε_0 and increasing ε_{up} or decreasing ε_{low} until the invariant is established (see lines 1 to 6). Subsequently, the binary search then repeatedly runs the procedure for finding ε-adversarial attacks and updates the bounds ε_{low} and ε_{up} accordingly. If it finds an adversarial attack, it increases ε_{low} to the respective value, else it decreases ε_{up}. Algorithm 1 stops once the interval $[\varepsilon_{\text{low}}, \varepsilon_{\text{up}}]$ is small enough (less than 2δ). In summary, Algorithm 1 provides an effective procedure to compute the worst-case-error of an autoencoder up to a user-defined accuracy $\delta > 0$, as formalized in the theorem below.

Theorem 4.3.2. *Let f be an autoencoder, A a region, and $\delta > 0$. Then, Algorithm 1 terminates eventually and outputs a value $\varepsilon^\star \in [wce(f, A) - \delta, wce(f, A) + \delta]$.*

Proof. In Lines 1 to 6 in Algorithm 1, we compute ε_{low} and ε_{up} (which are inputs to the binary search in Line 7) in such a way that, $\varphi^f_{\varepsilon_{low}}$ is satisfiable, while $\varphi^f_{\varepsilon_{up}}$ is not. Note that these initial searches terminate because the reconstruction error in A is bounded as it is a continuous function on a compact set. We refer to $wce(f, A)$ as ε_{wce} for brevity.
Once we enter the binary search, we have the following loop invariants: $\varphi^f_{\varepsilon_{low},A}$ is satisfiable; $\varphi^f_{\varepsilon_{up},A}$ is not satisfiable; and $\varepsilon_{low} \leq \varepsilon_{wce} \leq \varepsilon_{up}$. While the first two invariants are a consequence of the updates made during the loop in binary search, the third invariant can be proved using contradiction.

For the proof, we make use of the following property of wce : $\varphi^f_{\varepsilon',A}$ is satisfiable for any $\varepsilon' \leq \varepsilon_{wce}$ and $\varphi^f_{\tilde{\varepsilon},A}$ is not satisfiable for any $\tilde{\varepsilon} > \varepsilon_{wce}$. Now, towards contradiction, assume that $\varepsilon_{wce} < \varepsilon_{low}$ or $\varepsilon_{up} < \varepsilon_{wce}$. However, if $\varepsilon_{wce} < \varepsilon_{low}$, the above property of wce is violated because $\varphi^f_{\varepsilon_{low},A}$ is satisfiable. Similarly, if $\varepsilon_{up} < \varepsilon_{wce}$, the property of wce is violated since $\varphi^f_{\varepsilon_{up},A}$ is unsatisfiable. Thus,

$\varepsilon_{low} \leq \varepsilon_{wce} \leq \varepsilon_{up}$.

The binary search terminates when $|\varepsilon_{up} - \varepsilon_{low}| < 2\delta$ and $\varepsilon^* = (\varepsilon_{up} + \varepsilon_{low})/2$ is returned as an output. Now, in the final step, we have

$$\varepsilon^* - \varepsilon_{wce} \leq \varepsilon^* - \varepsilon_{low} \leq \frac{\varepsilon_{up} - \varepsilon_{low}}{2} \leq \delta, \text{ and}$$
$$\varepsilon^* - \varepsilon_{wce} \geq \varepsilon^* - \varepsilon_{up} \geq \frac{\varepsilon_{low} - \varepsilon_{up}}{2} \geq -\delta$$

Thus, $\varepsilon^* \in (wce(f, A) - \delta, wce(f, A) + \delta)$. □

As we can see Theorem 4.3.2 follows from Theorem 4.3.1 and the fact that the binary search of Algorithm 1 narrows down the interval $[\varepsilon_{\text{low}}, \varepsilon_{\text{up}}]$ until it is smaller than 2δ.

The complexity of Algorithm 1 consists mainly of two parts: the binary search and the SMT solver. The number of steps in the binary search is in $\mathcal{O}\left(log\left(\frac{wce(f,A)}{\delta}\right)\right)$. In each step, the SMT solver is called once with a runtime that mainly depends on the number of atomic formulas in (the respective) $\varphi^f_{\varepsilon,A}$. Under the restrictions in Section 4.3, there are $\mathcal{O}(n+m)$ many atomic formulas where n is the number of neurons in the autoencoder and m is the number of half-spaces used to construct the safety-critical region. Note that the number of atomic formulas arising from the L_∞ distance depends linearly on the dimension of the input/output space of the autoencoder and is hence in $\mathcal{O}(n)$. Even though encoding the problem as a formula is inexpensive, the SMT solver itself is an exponential algorithm, as discussed in Section 2.2.2.

4.4 Empirical Evaluation

We evaluate both concepts presented within our QUGA solution: (1) extracting an adversarial attack and (2) calculation of quality bounds. For our evaluation, we use both synthetic and real-world datasets. For future comparison and reproducibility of our experiments, we provide our implementation[3] with the off-the-shelf SMT solver Z3. As Z3 is not specialized for neural networks, our approach is not scalable enough to deal with benchmark datasets such as MNIST or CIFAR-10. Therefore later parts of this thesis are devoted to overcoming this issue.

[3]`https://github.com/KDD-OpenSource/QUGA`

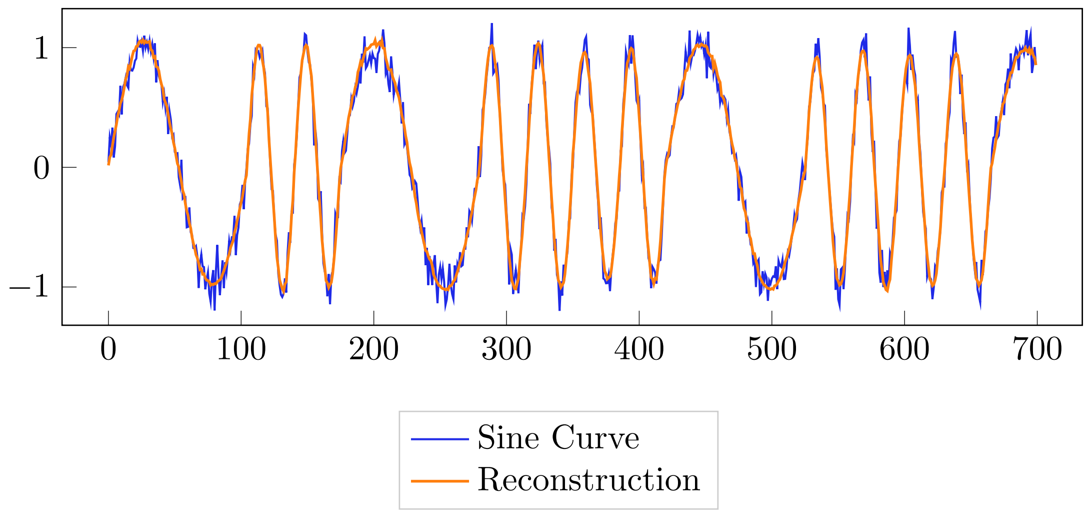


Figure 4.2: Synthetic sine curve with two frequencies and added noise with its reconstruction by the autoencoder.

4.4.1 Experiment Setup

We use synthetic time series data generated by sine curves with two different frequencies (35 and 105 timesteps per cycle) and random Gaussian noise ($\sigma = 0.1$) per time-point. Additionally, we use ECG5000 data from the UCR time series repository (CKH+15). We train autoencoders with an architecture of $L = (35, 5, 35)$ with five hidden ReLU nodes and 35 linear output nodes using the MSE loss function. Training data consists of time windows of length 35 without overlap. For the sine curve, the time windows correspond to 4 clusters: The full sine curve with 35 timesteps and the beginning, the middle and the end of the large sine curve. We denote them by $C_{full}, C_{beg}, C_{mid}$ and C_{end} respectively. For the ECG5000 dataset, we obtain 8 clusters arising from 2 classes and 4 time windows. We call them C_{i_x} where $i \in \{1, 2, 3, 4\}$ and $x \in \{u, b\}$, indicating the upper or lower part of the respective time window. As critical region A, we evaluate a hypercube around the two sine curves of width 0.2 in every dimension. This region contains, by construction, the majority of training data. For the ECG5000 dataset, we extract representative time series for the two main classes via their mean and add a margin of 0.25. We visualize the regions along with the training data in Figure 4.3.

4.4.2 Extracting an Adversarial Attack

The first observation is that our QUGA approach successfully extracts adversarial attacks. We depict the adversarial attacks obtained in Figure 4.4 for the Sine

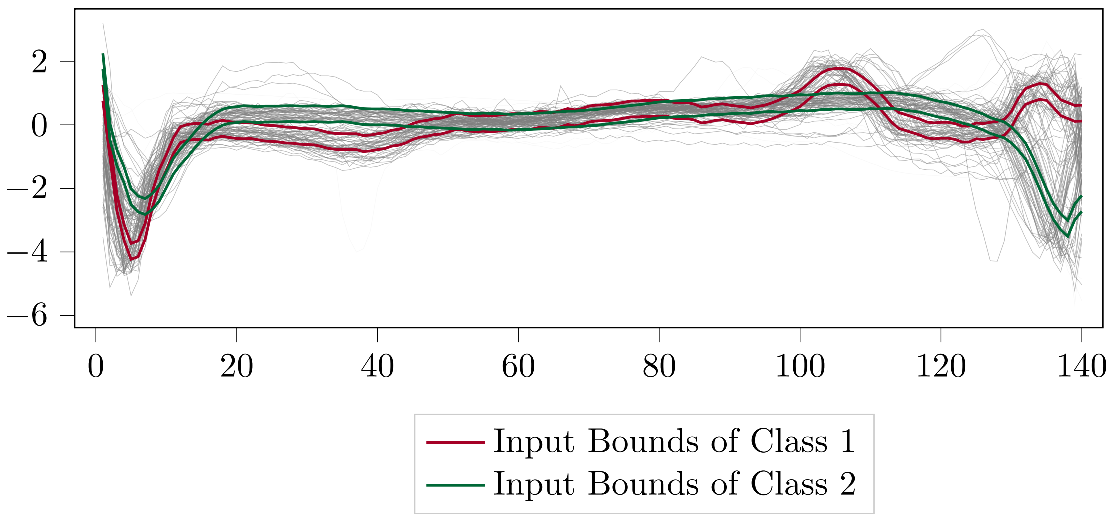


Figure 4.3: ECG5000 dataset with two safety-critical regions (red and green) obtained by extracting prototypes for two classes and adding a margin of 0.25.

Curve dataset and in Figure 4.5 for the two safety-critical regions in the ECG5000 dataset. As can be seen, our choice of *dist* leads to an optimization procedure in which the difference in one single dimension is maximized. The inputs for these adversarial attacks are usually found at the corner of the region A under scrutiny. This is reasonable because the autoencoder has been trained on inputs within the region A. Thus the error should increase with the distance from this region. Ideally, an autoencoder should extract a denoised version of the input. With the adversarial attacks, we have an indication of whether the autoencoder succeeds in doing so. For the sine curve, the outputs of the autoencoder on adversarial attacks in C_{full}, C_{mid} and C_{end} are much smoother than for the adversarial attacks in C_{beg}, suggesting that the autoencoder does not denoise as well in C_{beg}. For the ECG5000 dataset, the autoencoder seems to denoise for all clusters very well.

4.4.3 Comparing Quality Bounds with Sampling

We compare the quality bounds obtained by the QUGA approach with accuracy 0.025 to quality bounds obtained by a simple sampling approach. As a competitor to the QUGA approach, we sample points in the region, calculate their L_∞ errors and take the maximum as an estimator for the $L_\infty - wce$. Tables 4.1 and 4.2 sum up the results. First of all note, that the QUGA $L_\infty - wce$ bounds are much more precise. The $L_\infty - wce$ bound obtained by the sampling approach yields no upper bound at all, and furthermore, the lower bound is much weaker than

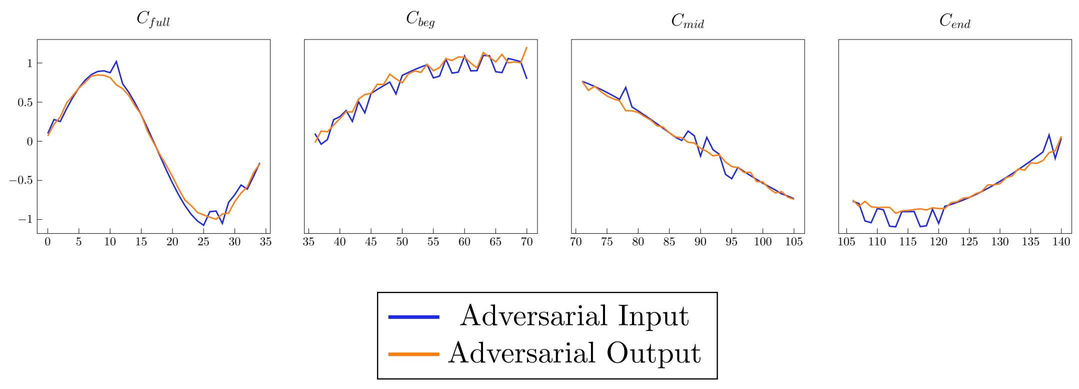


Figure 4.4: Adversarial attacks for different parts of the Sine Curve dataset obtained by the QUGA approach maximizing the L_∞-distance between the input and the output of the autoencoder in the respective safety-critical region. The adversarial attack on the second plot from the left indicates that this part is denoised less.

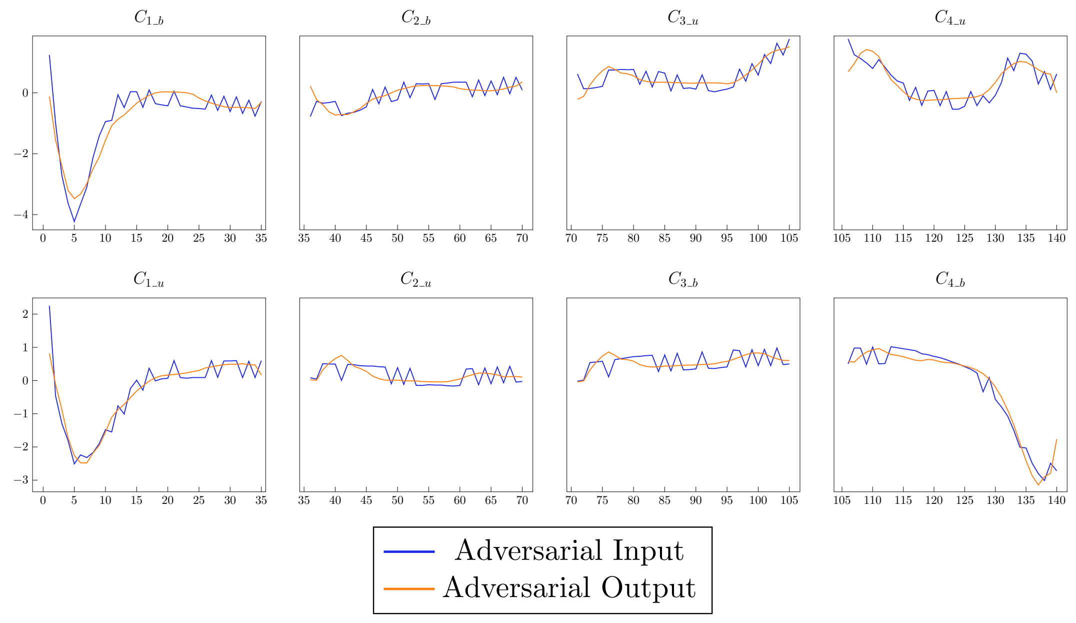


Figure 4.5: Adversarial attacks for two classes and different time windows of the ECG5000 dataset obtained by the QUGA approach maximizing the L_∞-distance between the input and the output of the autoencoder in the respective safety-critical region. No difference in denoising quality between the different plots can be seen.

the lower bound obtained by QUGA via the adversarial attack in all cases. A clear drawback of sampling is the large number of samples required to reach our QUGA estimation. In Figure 4.6 we show the runtime of QUGA vs. sampling with their respective error estimations. QUGA as a systematic search scheme is more efficient, while sampling is shown to underestimate worst-case-errors.

Sine Curve

Cluster	C_{full}	C_{beg}	C_{mid}	C_{end}
QUGA	0.297	0.422	0.297	0.266
Sampling	0.211	0.255	0.214	0.189

Table 4.1: Worst-case-errors as estimated by sampling and QUGA approach for the Sine Curve dataset. The accuracy for the QUGA approach is 0.025.

ECG

Cluster	C_{1_b}	C_{2_b}	C_{3_u}	C_{4_u}	C_{1_u}	C_{2_u}	C_{3_b}	C_{4_b}
QUGA	1.359	1.016	0.828	1.078	1.453	0.766	0.766	0.953
Sampling	1.189	0.829	0.651	0.908	1.255	0.563	0.546	0.774

Table 4.2: Worst-case-errors as estimated by sampling and QUGA approach for the ECG dataset. The accuracy of the QUGA approach is 0.025.

4.4.4 Safety Critical Application

We demonstrate a use case of our QUGA framework on the ECG5000 dataset by evaluating the unsupervised training based on two time series clusters. The goal of a traditional evaluation would be to show that all training objects are clearly separated in the latent space. In contrast, we care about all possible (infinitely many) objects in two safety-critical areas that need to be distinguishable in the latent space. In Figure 4.7, we see the resulting corridor into which points from the critical regions can be mapped. For the first three time windows, we cannot guarantee that the autoencoder keeps points from the two regions distinguishable in the latent space. Both clusters mix up as the upper bound of the lower cluster is higher than the lower bound of the upper cluster. For the last 35 time steps, though, a guaranteed separation of all infinitely many points in the critical regions is possible by the autoencoder. With this result, we can give a formal quality guarantee of the trained autoencoder. It securely extracts a latent representation for each time series in the safety-critical area that guarantees the separation of

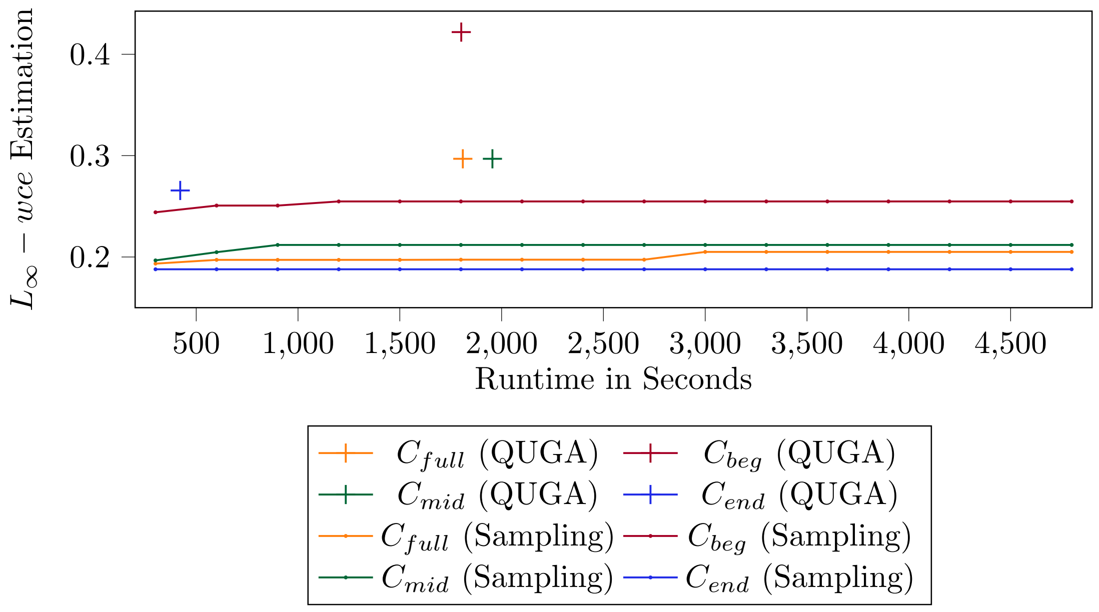


Figure 4.6: Depiction of runtime against wce estimation for the Sine Curve dataset. The sampling approach always underestimates the wce and does not increase over time.

both clusters. Please note that one could not have used the latent space to check separability directly. We have no control over where the autoencoder maps the safety-critical regions in the latent space. In contrast, our QUGA method solves this by symbolic representation of the autoencoder and the systematic search of possible unsupervised adversarial attacks that lead to a mix-up of two clusters. With this, we can prove separability for all infinitely many points in the safety-critical regions and not just on the finite training set.

4.4.5 Follow-Up Work

Based on the definitions and approach in this chapter, we conducted follow-up work highlighting the versatility of the worst-case-error. First, we use it to derive a so-called error landscape dissecting the input space into regions of different reconstruction errors (Rie20). Secondly, we use the unsupervised adversarial attacks found by our approach to describe the anomalous or the normal region as defined by the autoencoder in the context of anomaly detection (Atr21; Ell22). Lastly, we investigate whether adversarial training can help reduce the severity of worst-case-errors (Vu23).

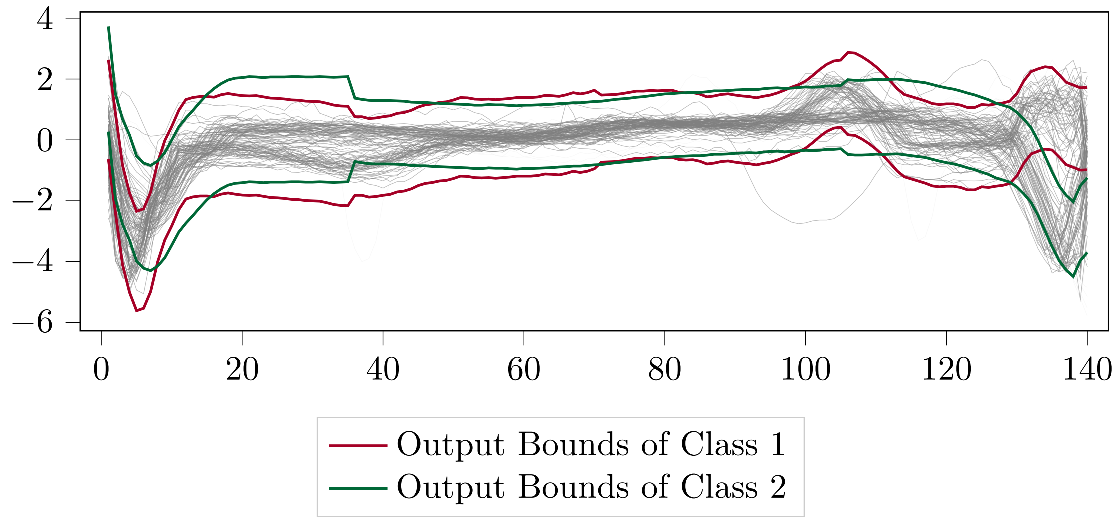


Figure 4.7: Image spaces (red and green) into which points from the respective safety-critical regions in Figure 4.3 can theoretically be mapped by the autoencoder.

Error Landscapes:
As first follow-up work, we divided the input space into different subregions in each of which the reconstruction loss is bounded from above and below. These regions enable us to determine where a given autoencoder has learned to reconstruct well with a level of certainty that cannot be attained by mere sampling. We call this dissection the error landscape of the autoencoder, as it can be thought of as a map of the reconstruction error.
First, we must be able to determine not only the worst-case-error but also the minimum reconstruction error in a given region. To do that, we make the necessary changes to the verification problem: we replace $\varphi_{\varepsilon}^{dist}$ from Equation 4.1 with

$$\varphi_{\varepsilon}^{dist,up} := \bigwedge_{1\leq j\leq N} \Big[[x_{0,j} - x_{L,j} < \varepsilon] \wedge [x_{L,j} - x_{0,j} < \varepsilon] \Big].$$

essentially asking for an input with a reconstruction error of less than ε. Based on this new verification problem, we can adjust the binary search to approximate the smallest reconstruction error in a region.
Having bound the error in a given region from both above and below, we can split the region into two subregions such that in the resulting subregions, the bounds are closer to each other. If this procedure is repeated often enough, we end up with regions in which the difference between the upper and the lower bound on the reconstruction error is smaller than a predefined threshold. Then we end up with a region in which all points have approximately the same error.

The described idea calls for a good splitting heuristic based on which a given region is divided. We tried several ones based on different use cases: axis-parallel cuts are used for an easy interpretation and description of the resulting subregions. Splitting right between the largest and the smallest error, on the other hand, tries to yield the best error-landscape with the fewest possible divisions. The question of which region to split is answered easily: we always choose the subregion to be divided based on the largest difference between the maximum and minimum reconstruction loss.
Error landscapes can be used to get a general overview of the reconstruction error. However, dependent on the autoencoder and the dimensionality, a large amount of neural network verifications may be necessary to obtain an accurate error landscape.

Description of Anomalous Region:
A more application-guided method based on the worst-case-error is the description of an anomalous or normal region. Autoencoders can serve as anomaly detectors by assigning each input an anomaly score based on the reconstruction error and comparing it to a threshold. Thus - similarly to our approach for error landscapes - we bound the reconstruction error from below for a given input region. If that bound is larger than the threshold, we can guarantee that the autoencoder will always predict *anomalous* in that region.
Similarly, we may describe the region in which points can be classified as normal (Ell22). To this end, we iteratively search for normal points and always extend the region in which normal points may occur. This region is usually bounded because autoencoders are trained on a bounded dataset.
Using these methods may help in providing a concise description of what exactly a given autoencoder does. Note, though, that, similar to all verification problems, we must not confuse this description of the autoencoder with a description of where normal or anomalous points are to be found. The verification problem can only yield a description of the autoencoder and never of the data on which it was trained.

Adversarial Training:
A common strategy to mitigate the problem of adversarial attacks for supervised neural networks is adversarial training (MMS+18). We adapt this strategy for the case of autoencoders by training them with adversarial attacks created using adapted versions of FGSM (GSS15) and PGD (MMS+18). To be precise, we used the reconstruction loss $\|x - f(x)\|_2^2$ to guide the construction of adversarial attacks. Moreover, we investigated two different autoencoder loss functions: the usual reconstruction loss $\|x - f(x)\|_2^2$ and an adaptation of the denoising loss (VLL+10)

for adversarial attacks given by $\|x - f(\tilde{x})\|_2^2$. In (VLL+10) $\tilde{x}$ is created by adding Gaussian noise to x. In our case $\tilde{x}$ is simply given by an adversarial attack on x. Our results are in line with adversarial training for supervised neural networks (MMS+18). By employing adversarial training, the worst-case-errors reduce substantially. This result was shown on all datasets and irrespective of the exact adversarial attack method.
Furthermore, not only did the worst-case-error decrease, but the distribution of the reconstruction error, in general, shifted towards smaller errors. We conclude that training on inputs with higher reconstruction errors, as given by the adversarial attacks, substantially improves reconstruction performance. It appears that adversarial training eliminates the weak spots of autoencoders by guiding the training towards more difficult regions.

4.5 Summary

QUGA overcomes major shortcomings of unsupervised learning with autoencoders. We provide the first methodology to bound the error of an autoencoder in a safety-critical region. With our solution framework based on SMT solvers, we propose to search for adversarial attacks in the infinite search space of a safety-critical region. Therefore, we have defined *unsupervised adversarial attacks* as inputs that show maximal error even if these objects are not contained in the training or test data. Our QUGA approach formulates the autoencoder, the safety-critical region, and the reconstruction error with a logical conjunction of linear constraints. Once we have found an unsupervised adversarial attack, it serves as a lower bound for the error, while binary search allows to derive an upper bound. We demonstrate the effectiveness of our approach on both a synthetically created and a real dataset. We show that QUGA finds unsupervised adversarial attacks, provides quality guarantees with lower and upper bounds, and outperforms sampling schemes that underestimate the worst-case-error.
As this is the first work for unsupervised adversarial attacks on autoencoders, we conducted some follow-up research. We showed how the general procedure presented can be extended to other bounds and other use cases. The major shortcoming of our approach is clearly given by scalability. Therefore the next chapters are concerned with overcoming this issue.

Training Autoencoders for Robustness and Scalability 5

As we have already seen in the last chapter, autoencoders are widely employed neural networks with a variety of use cases such as denoising (VLL$^+$10), dimensionality reduction (HS06) and anomaly detection (SY14). Due to their versatility and unsupervised nature, they can be used on almost any dataset and can serve either as a standalone solution or as a subroutine in a machine learning pipeline. Yet, as with any neural network, they are prone to overfitting, and thus, a lot of research effort has been directed towards making them robust (ZP17; QWZW14) and work reliably.
Additionally, as we have shown in the previous chapter, they can show a clearly undesired behavior: a significant change in the output given just a small perturbation of the input. Inspired by this phenomenon, neural network verification, which tries to prove or disprove properties of a given neural network, has emerged as its own research field.
Current notions of robustness for autoencoders do not address the unpredictable behavior neural networks can exhibit but merely evaluate a robustness measure on a predefined test set. However, as can be seen in Figure 5.1, finite test sets do not cover all possible inputs and thus may not find all flawed ones.
Therefore we introduce a new notion of robustness for autoencoders. At its core, it closely resembles the Epsilon-Delta criterion for continuity of functions, while it is also the natural extension of the supervised learning notion of robustness. It is designed to be applicable for different use cases, evaluates the entire $(L_\infty-)$neighborhood of radius ε for a given sample and is verifiable using SMT solvers (KBD$^+$17; WOZ$^+$20) or other constraint based verification methods. To use SMT solvers efficiently, though, we have to restrict ourselves to autoencoders using ReLU or Linear neurons[1] such that the autoencoder is a piecewise affine function.
Similar to the *wce* and in contrast to other notions of robustness for autoencoders, our new problem specification is developed with verifiability using SMT solvers in mind. Thus, in this chapter, we will recall how to solve the problem with SMT

[1]Note that our restrictions allow for convolutional and pooling layers as well because they can be represented using matrix multiplication.

solvers by translating it into an instance of the SMT problem. Thereafter, we can make use of any state-of-the-art solver as a blackbox tool to prove or disprove robustness of a given autoencoder.
However, recall that SMT solver-based verification has to iterate over all affine subfunctions in the given neighborhood. This slows down the verification procedure significantly and therefore inhibits scalability (c.f. Figure 5.2). Moreover, the more subfunctions we have in a given subset, the more unstable the autoencoder becomes, reducing robustness as we defined it.
To counteract this deficiency, we introduce a new regularization term aimed at reducing the number of affine subfunctions in a given neighborhood. The advantages are twofold: first, we obtain autoencoders that are more robust according to our definition. Secondly, we thereby train the autoencoder to be more amenable for its subsequent verification.
We experimentally compare our approach with respect to both robustness and verification runtime to other popular regularization schemes, including ridge regression (KH91), dropout (SHK$^+$14) and denoising (VLL$^+$10). Our approach yields the most robust autoencoders with the fastest verification while at the same time keeping the same predictive capacity as shown in a downstream experiment. Also, we do not only test verification runtime for the problem specification of robustness. We furthermore demonstrate how different regularization strategies affect the scalability issue of wce verification (compare further Chapter 4). We can show that the faster verification runtime we obtain for our regularization scheme is not due to the improved robustness but, indeed, tackles the verification process itself.
Moreover, we can show that some popular regularization schemes actually harm the process of verification. Dropout, for example, does not only *not* help in the verification process but actually slows it down significantly because it increases the number of affine subfunctions.
In summary, this chapter contains two major contributions. First, we give a **definition of robustness for autoencoders that permits verification**, show how to verify it and demonstrate its use. Secondly, we **design a training procedure** such that the resulting autoencoders are easier to verify.
Please note that these contributions correspond to a two-stage process: the regularization of autoencoders happens during the training phase, whereas the verification takes place for a given, already-trained autoencoder. Thus verification can be seen as a post-processing step.

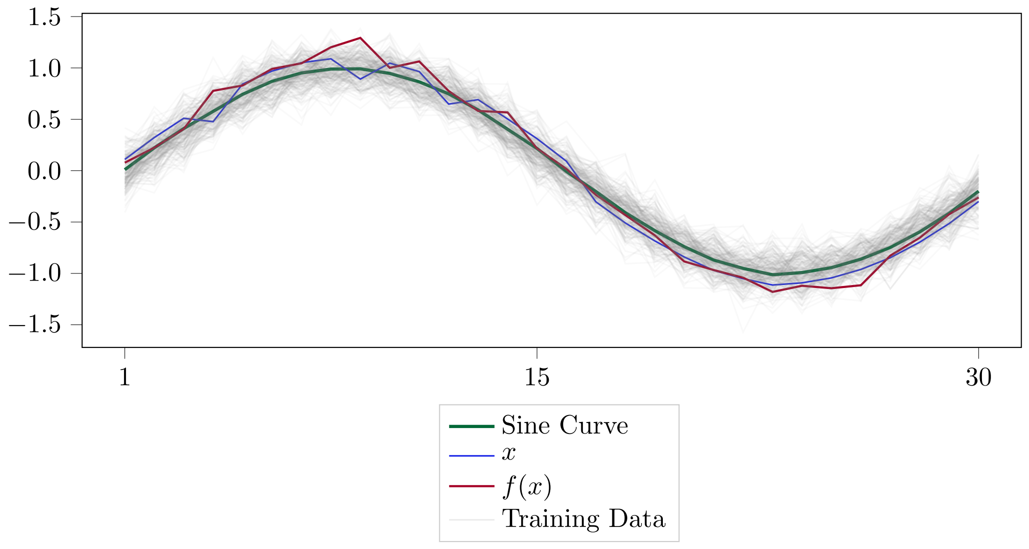


Figure 5.1: Inputs violating the desired behavior of an autoencoder. The autoencoder trained on the 30-dimensional samples should denoise and map all points surrounding the original sine curve closer to it. However, we found an input x clearly violating this behavior as $f(x)$ differs strongly from the denoised sine curve.

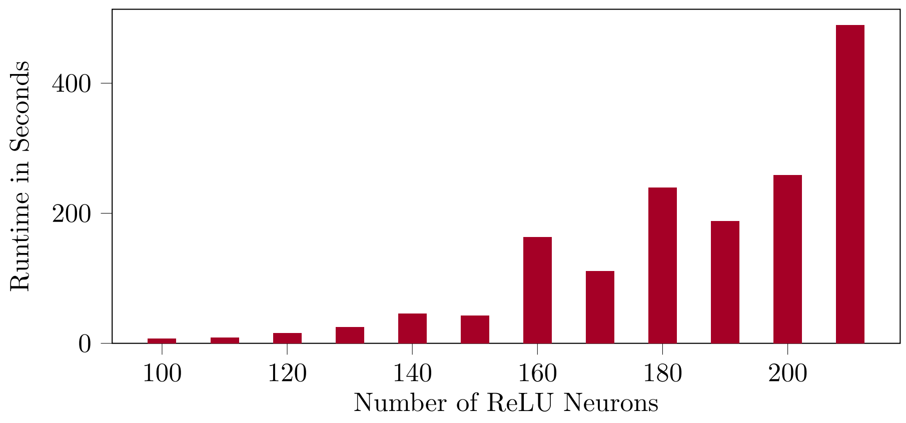


Figure 5.2: Runtime comparison of robustness verification with different numbers of ReLU neurons. More ReLU neurons correspond to more affine subfunctions that need to be checked during the verification process. Hence the runtime increases with more ReLU neurons. Results are averaged over 100 runs.

5.1 Related Work

Neural Network Verification:
Since the seminal paper by (SZS$^+$14) introducing the notion of adversarial attacks, the research community is aware of the unstable behavior neural networks can exhibit. Starting from a given input x with label y, adversarial attacks are inputs close to x yet with a different label $\hat{y} \neq y$.
To prove the (non-)existence of such adversarial attacks in a given neighborhood - or another predefined property about a neural network - is called neural network verification (XTSM19; Ehl17). Contrary to the classical way of evaluating neural networks using a test set, it has to implicitly consider all infinitely many points in the neighborhood by some form of symbolic reasoning.
However, verification of unsupervised neural networks has so far been mostly neglected due to missing labels. Without labels, we require different problem specifications that can be verified. In particular, if we want to prove robustness for a given autoencoder in a given neighborhood, we need to define this in terms of a verifiable property.
We presented the only other work on unsupervised neural network verification (Ben20) in the previous chapter. It introduces the notion of the worst-case-error of an autoencoder as the largest reconstruction error that can manifest in a given region. In contrast, this chapter introduces the notion of robustness based on a reference point asking how strongly we need to perturb it in order to significantly change the outcome.
Verification Approaches and Scalability:
As discussed in Chapter 3, approaches for neural network verification fall into two main categories: on the one hand, there are approximation methods (SGM$^+$18; ZWC$^+$18; WZC$^+$18) which base their verification on calculating limits on the neural network's output. While these approaches are both fast and sound, they are not complete: if the network fulfills the given property, these methods may prove it for you. However, if they fail to do so, the property might still be true and the methods just could not prove it.
In our experiment, we will make use of the other branch of research consisting of exact (meaning both sound and complete) verification methods (KBD$^+$17; Ehl17; TXT19). In particular, we will use SMT based approaches (Ehl17). They translate the verification problem into an instance of the SMT problem encoding the neural network and the property to be checked. These approaches are exact, which means that they guarantee the given property if and only if it holds true. However, this comes at the cost of two disadvantages. First, it allows only for ReLU/Linear/Convolutional/Pooling layers to be used, ensuring that the neural network consists of a collection of affine subfunctions. Secondly, the verification does not scale up to large networks because it essentially has to iterate over all of

these affine subfunctions in a given neighborhood.
To speed up the verification process by focusing on the improvement of the SMT solver is an ongoing research effort (KBD$^+$17; WOZ$^+$20; MH21). However, SMT-based verification of ReLU networks is slow as the underlying problem is NP-complete (WZC$^+$18). Its runtime grows exponentially in the number of ReLU neurons. Thus, as observed by Xiao et al. (XTSM19), it is useful to design a training procedure that makes the verification problem more amenable for the SMT solver. By introducing a regularization term for training, we reduce the number of affine subfunctions in a given neighborhood resulting in an instance the SMT solver can solve more efficiently. Note that our method can thus be combined with any improvement on the SMT solver itself.
Autoencoder Regularization:
Numerous different regularization schemes have been proposed both for neural networks in general and for autoencoders in particular. For our comparison, we consider general schemes such as ridge regression (KH91) and dropout (SHK$^+$14) as well as autoencoder-specific schemes such as denoising (VLL$^+$10) and robust autoencoders (ZP17). Moreover, we include the MMR regularization (CAH19) as it is closest in nature to our scheme by exploiting the ReLU structure. This form of regularization intends to push the boundaries between different subfunctions such that the subfunctions around training samples occupy a larger space. We, on the other hand, join different subfunctions with our regularization term.
Beyond these general regularization strategies, some methods to increase verifiable robustness have been proposed (BLZ$^+$21b). Most notably, (MMS$^+$18) add adversarial attacks to their training data, thereby increasing the robustness of their models. However, these are no general regularization schemes and we cannot expect that verification runtime will decrease using these methods. Our method, though, has two goals: to increase the robustness of a given autoencoder and to decrease the runtime of SMT solvers.

5.2 Robustness Verification for Autoencoders

This section builds upon the definition of an autoencoder given in Section 4.3.1; we will briefly recall some of its properties relevant for this chapter.
As discussed in Chapter 4, an autoencoder $f : \mathbb{R}^N \to \mathbb{R}^N$ is a feed-forward neural network consisting of an encoder and a decoder. The encoder maps the input to the latent space while the decoder, in turn, maps the latent space back to the input space. Autoencoders are trained using a reconstruction loss which, for a

given input x and autoencoder f, is defined as:

$$L_{recon}(x) = \|f(x) - x\|_2^2$$

However, autoencoders cannot perfectly reconstruct the input because the latent space usually has fewer dimensions than the input space and thus, it serves as an information bottleneck. Therefore autoencoders can be used to extract denoised versions of the data as they can only learn the important patterns present in it. Note that while there are other types of autoencoders with different loss functions, we will use only those which are built upon this type of reconstruction loss and architecture.

5.2.1 Verifiable Robustness for Autoencoders

Currently, almost all existing verification problems are designed for supervised neural networks and are based on a change in labels. They usually verify whether in a neighborhood around a given input x^* with a presumably correct label y there exists another input $\hat{x}$ with a different label. However, for autoencoders, we have no such labels at our disposal. Therefore we need other problem specifications capturing desirable properties of an autoencoder.
Our research aims at filling this gap and is inspired by the robustness property for supervised neural networks (MMS+18). It refers to how strongly a given input must be changed to cause a change of labels. Similarly, our notion of robustness for autoencoders checks how strongly the input must be changed such that the output differs by a predefined amount.
Another way to think about our new definition is by comparing it to the reachability verification problem (XTJ18). In essence, it asks what the possible outcomes a neural network exhibits for a predefined input set are.
Moreover, our notion of robustness resembles the definition of continuity for functions making it a general-purpose problem specification. We give the unsupervised problem specification in the following definition:

Definition 5.2.1 (ε-δ robustness). *Let $f : \mathbb{R}^N \to \mathbb{R}^N$ be an autoencoder, $dist : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}_+$ a metric and $x^* \in \mathbb{R}^N$ an input sample. We say that x^* is ε-δ robust if*

$$dist(x^*, x) < \varepsilon \Rightarrow dist(f(x^*) - f(x)) < \delta.$$

Essentially x^* is ε-δ robust if all points close to x^* map to points close to $f(x^*)$. The metric will usually be given by an L_p-norm and for our solution framework, we require it to be the L_∞-norm.
From this definition, we can moreover derive a local stability measure for a fixed δ given by the largest ε such that ε-δ robustness holds true.

Definition 5.2.2 (rob_δ)**.** *In the above setting with δ fixed, we define the robustness measure as*

$$rob_\delta(f, x^*) = \sup\{\varepsilon \in \mathbb{R}_+ \mid x^* \textit{ is } \varepsilon\text{-}\delta \textit{ robust under } f\}.$$

This definition extends Definition 5.2.1 by asking for the largest change in the input space such that the change in the output space remains bounded by δ. Clearly, two problems follow from these definitions. First, for fixed ε and δ, we want to determine whether a given point x^* is ε-δ robust and secondly, we want to calculate the robustness measure for a fixed δ. The ε-δ robustness check will be solved repeatedly as a subroutine to determine rob_δ.
We highlight the use of these definitions in our experiment section, where we a) prove that a given autoencoder denoises and b) use it as a robustness measure for known regularization techniques. Yet the definition is not limited to these use cases. In the case of anomaly detection, one could, for instance, prove that all points around a given anomaly are predicted to be anomalous as well. This shows that our definition is quite general and can be used in a versatile way.
However, high robustness is not always a desirable property. For example, an autoencoder mapping all inputs to a single output is ε-δ robust for any choice of ε, δ and $dist$. Thus, as in the case of supervised robustness, it must be considered jointly with other metrics.

5.2.2 Comparison to Worst-Case-Error

We compare our notion of robustness for autoencoders to the worst-case-error *wce* introduced in Chapter 4. While the two problems are similar to each other, they differ in the following sense: the robustness measure searches for a point $\hat{x}$ in the input space such that $f(\hat{x})$ differs from $f(x^*)$. Thus it searches for another input-output pair $(\hat{x}, f(\hat{x}))$ for an already given input-output pair $(x^*, f(x^*))$. The worst-case-error, on the other hand, has no reference input-output pair but instead searches for a point x in a predefind region such that the distance between x and $f(x)$ is large. Thus both problem specifications measure a form of behavioral stability. The wce is mostly inspired by the reconstruction loss as the predominant quality criterion. The robustness measure, on the other hand, is more inspired by the denoising property of autoencoders as well as the manifold assumption (GBC16), which means that the autoencoder maps all points to a small dimensional manifold.
Nevertheless, both notions yield valid verification problems and hence we will check the speedup of our regularization approach (see Section 5.4) on both of them in our experiments.

5.3 SMT Problem Encoding

Similar to Section 4.3.2 (wce problem encoding) we will now describe how to solve the ε-δ robustness verification problem using SMT solvers. We will - again - encode the entire formula for the SMT solver as a conjunction of three formulas: φ^f encoding the autoencoder, φ_ε encoding the distance in the input space and φ_δ encoding the distance in the output space.

$$\varphi^f_{\varepsilon,\delta} := \varphi^f \wedge \varphi_\varepsilon \wedge \varphi_\delta$$

As a result of the verification process, we will obtain either $unsat$ corresponding to a robust autoencoder, or sat, together with a sample violating the robustness property.
This approach entails similar restrictions as in Section 4.3.2; we can only make use of certain types of both metrics and autoencoders. To be precise, we use L_∞ as our metric and autoencoders consisting of Linear and ReLU nodes.
The precise description of φ^f can be found in Section 2.2.2. For the input space and a particular point $x^* \in \mathbb{R}^N$ we add

$$\varphi_\varepsilon = \bigwedge_{1 \leq i \leq N} [[x_{0,i} \geq x_i^* - \varepsilon] \wedge [x_{0,i} \leq x_i^* + \varepsilon]]$$

to the formula. Likewise for the output space and $f(x^*) \in \mathbb{R}^N$ we add

$$\varphi_\delta = \bigvee_{1 \leq i \leq N} [[x_{L,i} \geq f(x^*)_i + \delta] \vee [x_{L,i} \leq f(x^*)_i - \delta]]$$

to the formula. Basically the formula φ_ε restricts the SMT solver to search for a point x in an $\varepsilon-$ environment of x^* while φ_δ is satisfied if and only if the error between $f(x^*)$ and $f(x)$ is large enough. If the SMT solver returns $unsat$ for this formula, we know that for all points in $\{x \in \mathbb{R}^N : \|x - x^*\|_\infty < \varepsilon\}$ the difference of $f(x)$ to $f(x^*)$ is less than δ thereby proving robustness. If, instead, the solver returns sat and a solution $\tilde{x}$ which satisfies $\|\tilde{x} - x^*\|_\infty < \varepsilon$ and $\|f(\tilde{x}) - f(x^*)\|_\infty > \delta$, we directly obtain an example indicating where robustness failed.
To further ease understanding, we will present a small example: For the simple autoencoder given in Figure 5.3, $x^* = (0.5, 0)$, $f(x^*) = (1, 0)$, $\delta = 0.2$ and $\varepsilon = 0.1$ the SMT formula looks as follows:

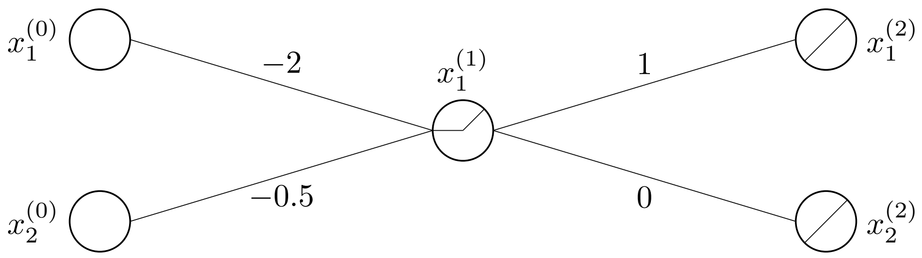


Figure 5.3: Exemplary autoencoder f for explaining the construction of SMT formulas.

Example 5.3.1.

$$\begin{aligned}
\varphi^f := \; & [y_{1,1} = 2x_{0,1} + (-0.5)x_{0,2}] \\
& \wedge[x_{1,1} = \mathsf{ite}(y_{1,1} < 0, 0, y_{1,1})] \\
& \wedge[x_{2,1} = 1x_{1,1}] \\
& \wedge[x_{2,2} = 0x_{1,1}]
\end{aligned}$$

$$\begin{aligned}
\varphi_{\varepsilon=0.1} := \; & [x_{0,1} \geq 0.4] \\
& \wedge[x_{0,1} \leq 0.6] \\
& \wedge[x_{0,2} \geq -0.1] \\
& \wedge[x_{0,2} \leq 0.1]
\end{aligned}$$

$$\begin{aligned}
\varphi_{\delta=0.2} := \; & [x_{2,1} \geq 1 + 0.2] \\
& \vee[x_{2,1} \leq 1 - 0.2] \\
& \vee[x_{2,2} \geq 0 + 0.2] \\
& \vee[x_{2,2} \leq 0 - 0.2]
\end{aligned}$$

The solution for this instance is sat with, e.g., the following variable assignments:
$\{x_{0,1} = 0.6, x_{0,2} = -0.1, x_{1,1} = y_{1,1} = x_{2,1} = 1.25, x_{2,2} = 0\}$

Finally, in order to calculate rob_δ for a given δ, we apply binary search over the values of ε to obtain the maximum robustness up to a user-defined accuracy acc. This is similar to the binary search applied in Section 4.3.3 to calculate the wce. Starting from an upper bound on ε one repeatedly checks for ε-δ robustness and increases/decreases ε accordingly. Thus, dependent on the accuracy, one needs $\mathcal{O}\left(log\left(\frac{rob_\delta}{acc}\right)\right)$ many checks to determine rob_δ.

5.3.1 Understanding SMT solvers

As we have discussed in Chapter 2, exact verification methods such as SMT solvers are inefficient. Recall that in order to incorporate ReLU neurons into φ^f we need to add $\vee$-conjunctions to it, slowing down the verification process. SMT solvers then need to transform this formula into its boolean abstraction replacing each inequality with a boolean variable and solving the resulting SAT instance. The solution of this instance is, in turn, translated to a Linear Program corresponding to one particular ReLU activation pattern. As we can see, the SMT solver needs to repeatedly solve the SAT problem as well as the LP problem, usually using DPLL (DP60; DLL62) and Simplex (DdM06). How often it needs to solve these problems strongly depends on the number of $\vee$ clauses in the formula $\varphi^f_{\varepsilon,\delta}$. Thus if we can find a way to reduce this number, we expect the SMT solver to run more efficiently, speeding up the process of verification. This is the goal of the next section.

5.4 Regularization Strategies for Autoencoders

This section introduces our new regularization term *fctdist* with which we train autoencoders to be more amenable for verification and which simultaneously increases their robustness. Moreover, we will recall the necessary theoretical foundation and give an overview of other regularization strategies against which we compare our new approach.

Preliminaries:
According to Section 2.1.4, autoencoders using only ReLU and Linear neurons can be seen as piecewise affine functions. Our regularization term *fctdist* makes use of this particular structure because it is based on the difference between neighboring affine subfunctions. Therefore we need to calculate the current affine subfunction for a particular point x as well as the borders to other, close affine subfunctions. For the current affine subfunction, we employ Theorem 2.1.2 and for the distance $d_B(x)$ to the nearest different subfunction, we recall that each ReLU neuron before its activation function can be seen as a function $f_{r,x}$ on its own (see Section 2.1.4). These functions define borders to other affine subfunctions via their kernel $ker(f_{r,x}) = \{y \in \mathbb{R}^N | f_{r,x}(y) = 0\}$. After employing Theorem 2.1.2 again, the function $f_{r,x}$ is locally given by $f_{r,x} = V_{r,x} + a_{r,x}$. We calculate the distances $d_r(x)$ to the border defined by its kernel by projecting onto it with the following formula (see (CAH19)):

$$d_r(x) = \frac{|\langle V_{r,x}, x\rangle + a_{r,x}|}{||V_{r,x}||_2}$$

Of course, once we overstep one border, the other borders change (see Figure 2.6). However, for the calculation of the closest border, this phenomenon does not occur. Thus we can calculate $d_B(x)$ via

$$d_B(x) = \min_{\substack{r \\ r \text{ is ReLU neuron}}} \frac{|\langle V_{r,x}, x\rangle + a_{r,x}|}{||V_{r,x}||_2}.$$

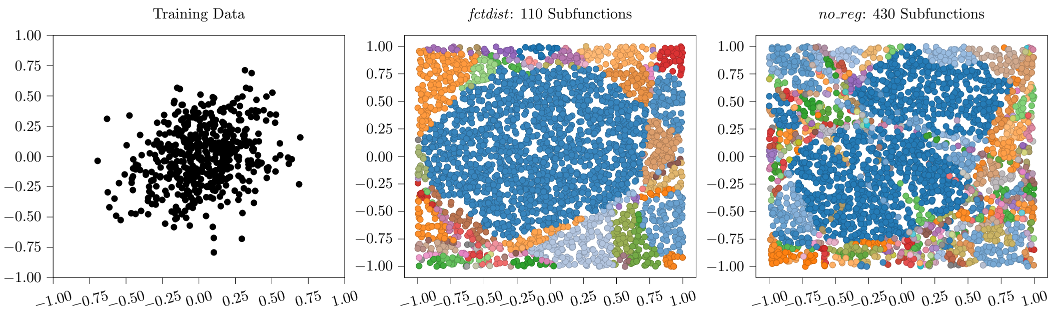


Figure 5.4: Estimating the resulting subfunction structure of an autoencoder with a two-dimensional input for given training data (left) once with *fctdist* regularization (middle) and once without regularization (right). Different subfunctions are shown by (20) different colors. The figure was obtained by sampling 3000 inputs and calculating their respective subfunction. The regularized version better resembles the training data and the non-regularized version employs many more subfunctions.

5.4.1 Function Distance Regularization Term

Equipped with the theoretical foundation on the structure of autoencoders, we can now define the regularization term we propose to increase their verification scalability. Essentially for any given x, we train the network to make the surrounding subfunctions similar to the one the autoencoder applies to x. This effectively reduces the number of subfunctions that need to be checked during verification (see Figure 5.4).

More precisely, let $k \in \mathbb{N}$ be the number of subfunctions we want to adjust and let $\{(U_i, B_i)\}_{i=1,\ldots,k}$ be the k affine subfunctions closest to x (see Figure 5.5). Moreover, let (U, B) be the affine subfunction for x. Then we define the regularization term *fctdist*(x, λ) to be:

$$\textit{fctdist}(x,\lambda) = \lambda \frac{1}{k} \sum_{i=1}^{k} (\|U_i - U\|_F + \|B_i - B\|_2)$$

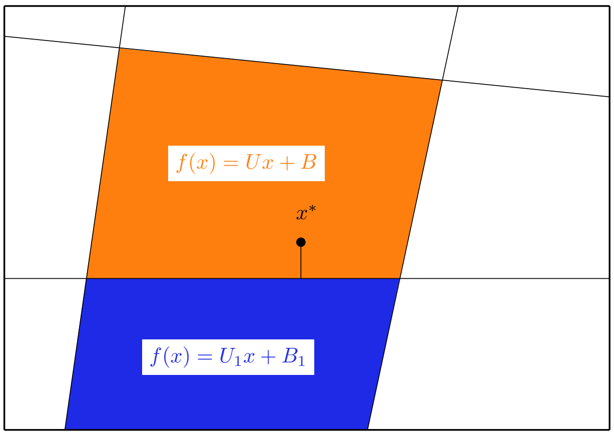


Figure 5.5: For $k = 1$ we have have the loss function $\mathit{fctdist}(x^*) = \lambda(\|U - U_1\|_F + \|B - B_1\|_2)$ because (U_1, B_1) is the closest different affine subfunction.

where $\|\cdot\|_F$ denotes the Frobenius norm for matrices and $\|\cdot\|_2$ denotes the Euclidean norm for vectors. The total loss function for training is then given by:

$$L_{\mathit{fctdist},\lambda}(x) = \|x - f(x)\|_2^2 + \mathit{fctdist}(x, \lambda)$$

Note that, of course, these extra calculations add an overhead to the loss calculation and backpropagation. However, it adds roughly the same time as the mmr regularization scheme by (CAH19) and - dependent on how often verification is done - is outweighed by the reduction in verification time. Note also that the resulting loss function is clearly differentiable.
Internally SMT solvers keep a boolean abstraction in which the inequalities of the encoding become binary variables. On this abstraction, the SMT solver has to repeatedly solve the SAT Problem whose runtime is governed by how often the conjunction $\vee$ occurs. On the neural network side, each ReLU node contributes one $\vee$ to the SMT solver's SAT problem. However, if in the target region the ReLU node is either always activated or always zero, the respective $\vee$ is essentially irrelevant. This results in an easier problem instance for the SMT solver and hence in faster runtime.

5.4.2 Existing Autoencoder Regularization Schemes

Beyond our newly defined loss function, we will use other loss functions and regularization schemes to compare our method against and hence we briefly recall them here. In most cases, they differ from the usual autoencoder model by adjusting the loss function L.

- No regularization serving as the baseline model:

$$L_{no\_reg}(x) = L_{recon}(x) = \|x - f(x)\|_2^2$$

- *mmr* regularization (CAH19) in which short distances to the k closest affine subfunctions are penalized:

$$L_{mmr} = \|x - f(x)\|_2^2 + \frac{1}{k}\sum_{i=1}^{k} \max\left(0, 1 - \frac{d_B^i(x)}{\gamma_B}\right)$$

  where $d_B^i(x)$ is the distance to the i-th closest affine subfunction from x and γ_B is a regularization parameter describing the distance until which the distance shall be penalized. Similarly to our approach, this method exploits the piecewise affine structure of ReLU neural networks. It is supposed to increase the area around x in which a particular affine subfunction is applied.

- Ridgeregression by (KH91) penalizing large network weights:

$$L_{L2}(x) = \|x - f(x)\|_2^2 + \lambda \sum_{i=1}^{L} \|W_i\|_F^2$$

  where λ denotes the regularization parameter.

- Dropout by (SHK+14), which randomly chooses a subset of nodes that are deactivated for each training sample. It uses L_{no_reg} as metric.

- Denoising by (VLL+10) adding noise to each input yet keeping the original sample as reconstruction target:

$$L_{denoising}(x) = \|x - f(\tilde{x})\|_2^2$$

  where $\tilde{x} = x + \gamma$ and γ is given by gaussian noise.

- RobustAEs by (ZP17) working with the datamatrix X whose rows are the training samples. It is split into $X = L_D + S$ where S is supposed to represent the noisy/anomalous samples. Then, iteratively, the following equation is minimized:

$$\min_{\theta} \|L_D - f(L_D)\|_2 + \lambda\|S\|_1$$

  maintaining $L_D + S = X$ after every iteration.

Both the $L2$ and the dropout regularization can be used on any neural network, whereas denoising is a strategy designed specifically for autoencoders. The mmr regularization was originally designed for supervised neural networks. Hence we slightly adjust it to make it applicable for autoencoders. Moreover, our newly proposed loss function $L_{fctdist}$ can also be used for supervised neural networks - a direction we postpone for future research.
Lastly, note that all of these regularization schemes are used in the training phase only. For measuring robustness/worst-case-error we will rely exclusively on the L_∞ metric.

5.5 Experiments

This section is divided into three parts: first, we will demonstrate potential use cases of the robustness property. Next, we will focus on scalability showing the significant reduction of verification time achieved by the *fctdist* regularization term. Lastly, we will compare different regularization schemes for autoencoders with respect to their influence on robustness, downstream classification capability and runtime[2].
For each of the following experiments, we have trained 10 autoencoders with their different regularization schemes to account for randomness in the training procedure and evaluated each of them 10 times to account for randomness in the SMT solver's runtime.

5.5.1 Use Cases of Robustness

To start with, we will show how we can verify that a given autoencoder denoises in a particular region. To this end, we consider a very simple dataset consisting of a sine curve with noise on which a denoising autoencoder (VLL+10) has been trained (see Figure 5.6). Since we know how the dataset has been (synthetically) constructed, we know how to define denoising: reconstructions of the autoencoder are supposed to be closer to the original sine curve.
We could verify that the robustness (ε) around the actual sine curve is 0.288 with $\delta = 0.1$. That means that the autoencoder maps all samples within the denoising area (for x^* being the sine curve this is given by $x^* \pm 0.288 := \{x \in \mathbb{R}^{30} : \|x - x^*\|_\infty \leq 0.288\}$) to reconstructions within the target area ($x^* \pm 0.1$), proving that the autoencoder actually denoises.

[2]The code, as well as implementation details used to generate the results, can be found online at `www.github.com/KDD-OpenSource/robust_AE`

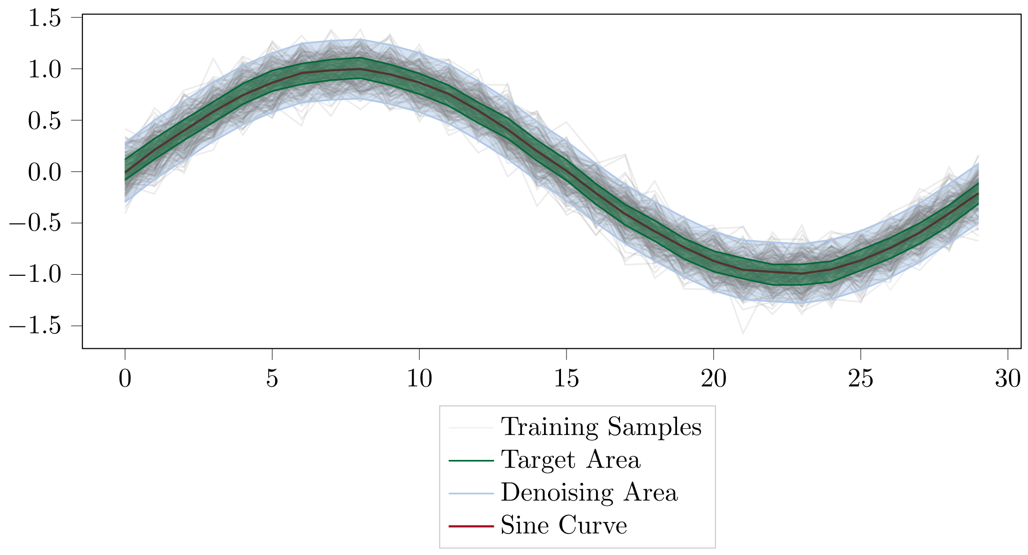


Figure 5.6: The autoencoder provably denoises because all points within the denoising area (blue) are mapped to points within the target area (green).

Note, though, that this does not directly verify that denoising happens *within* the target area. For this, we would have to choose another target area by adjusting δ.
Next, we want to validate that, using robustness, we can measure a form of regularization. To this end, we check robustness for increased levels of the well-known denoising regularization scheme by (VLL+10). As can be seen in Figure 5.7, we obtain more robust autoencoders for higher levels of denoising. Even though the general trend is clearly visible, we also observe that each denoising level exhibits a large variance among its 10 runs.
As our problem specification is deterministic, this must be due to the training procedure of the autoencoder. Therefore, if we want to ensure robustness, verification is mandatory each time you train an autoencoder and you cannot rely solely on a good training procedure. Thus we can show that training autoencoders with the same level of denoising multiple times can still lead to substantially different robustness results underlining the importance of verifying *any* given network.

5.5.2 Scalability of Autoencoder Verification

In this section, we will see how the *fctdist* regularization term decreases verification runtime. Figure 5.8 compares both verification problems (robustness and worst-case-error) for different autoencoder architectures with and without the

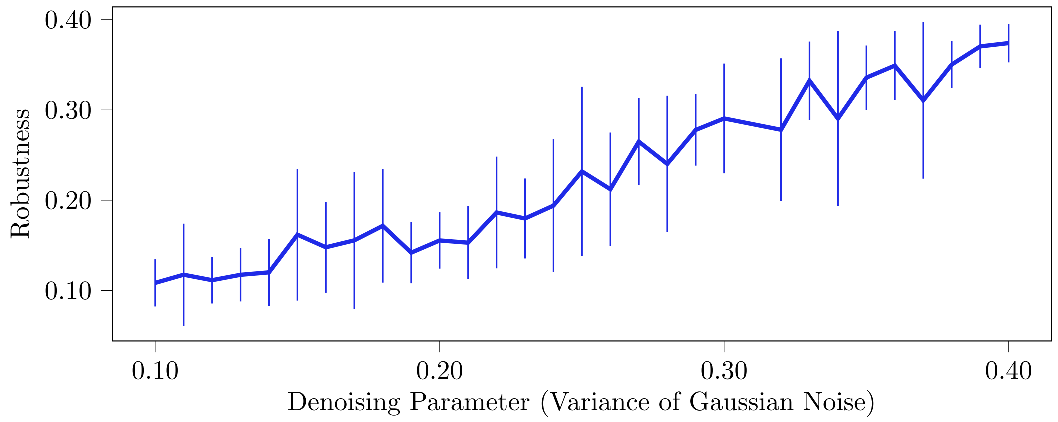


Figure 5.7: Average robustness of different autoencoder models given by different levels of denoising on Sine Curve dataset. For larger levels of denoising the robustness increases as well. For each level of denoising we show the mean and the standard deviation over ten different models.

fctdist regularization on the Sine Curve dataset. Moreover, it shows an estimate of the number of affine subfunctions surrounding the actual sine curve. As we can see, the mean runtime, the runtime's standard deviation and the number of surrounding subfunctions increase significantly for larger architectures in the case of non-regularized autoencoders. In contrast, both the runtime and the number of surrounding subfunctions increase far less for the regularized autoencoder. The largest average speedup is by a factor of 21 in case of the robustness verification and 16 in case of the worst-case-error verification. This experiment confirms that easier models - as measured in the number of affine subfunctions - are verified faster as in the verification process any SMT solver needs to iterate over fewer affine subfunctions in the given neighborhood. Thus with fewer subfunctions, the verification problem is solved faster.
Lastly, we found that the high variance in verification time can be attributed solely to the SMT solver and not to the randomness in autoencoder training. Fixing one particular trained autoencoder and repeatedly running the SMT solver over it yields the same variance in runtime as when aggregated over different models.

5.5.3 Empirical Results on Affine Subfunctions

We can use the analysis tools at our disposal to count the true number of subfunctions a neural network actually consists of. Recall that in theory, a network with k number of ReLU neurons may comprise 2^k many subfunctions. Experimentally we could show though (Ben22b) that most of these subfunctions are infeasible:

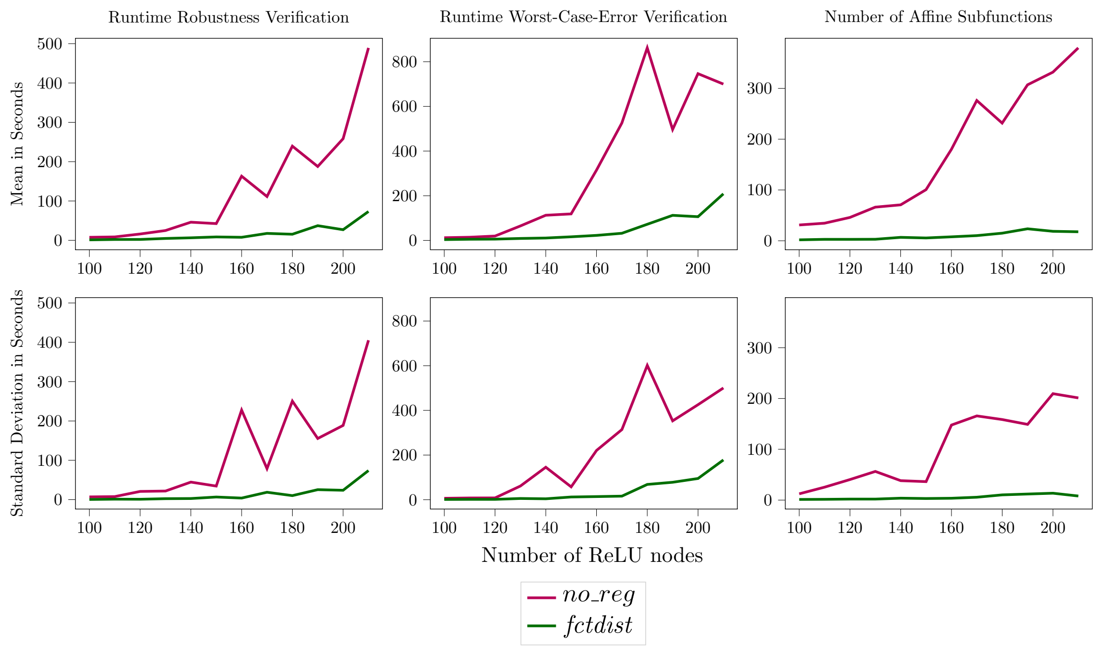


Figure 5.8: Mean and standard deviation over 100 runs for different numbers of ReLU neurons. For both verification problems both the mean runtime as well as standard deviation increase significantly with the estimated number of surrounding affine subfunctions.

there is no input for the respective activation pattern. In fact, our experiment on two autoencoders resulted in just 1% of the possible subfunctions being feasible. A lot of open research questions spawn from this observation. On what does the number of feasible subfunctions depend? How may we manipulate it? How many subfunctions are needed to retain a good performance? Please note that while it sounds promising to be able to discard 99% of the affine subfunctions, we are still left with a substantial number; a fact confirmed by the inability of verification methods to analyze large neural networks.
However, we might leverage on the preceding observation. Constraint-based verification methods (see Chapter 3) implicitly cover all possible ReLU activation patterns resulting in a massive overhead as they consider the infeasible activation patterns also. Thus, we may be able to filter out the infeasible ones as a preprocessing step and thereafter let the verification method process only the remaining subfunctions. This would be particularly beneficial if multiple verifications on the same neural network are carried out as, for example, in the VNN-competition (MBB+23) on the ACAS Xu system (OPM+19).

5.5.4 Comparison of Different Regularization Schemes

Now we turn to the experimental comparison of the different loss functions both with respect to the robustness measure as well as the worst-case-error measure introduced in (Ben20).

Experimental Setup

We used five different real datasets, all taken from (CKH+15) as given in Table 5.1. For each of them we have used an autoencoder with an architecture given by the right hand side column. For example the architecture $65-25-5-25-65$ refers to an autoencoder with 65 inputs/outputs and three hidden ReLU layers with 25, 5, and 25 neurons respectively. As they contain labels, we use the class means as reference points x^* to calculate robustness. Note that we could have obtained such points in a purely unsupervised manner by using a clustering algorithm (for example K-Means (M+67)) as well. The results are shown in Figure 5.9. Let

Dataset Information	
Dataset	Autoencoder Architecture
sonyAIBORobotSurface2 (S2)	$65-25-5-25-65$
sonyAIBORobotSurface1 (S1)	$70-25-5-25-70$
PhalanxOutlinesCorrect (P)	$80-25-5-25-80$
TwoLeadECG (T)	$82-25-5-25-82$
MoteStrain (M)	$84-25-5-25-84$

Table 5.1: Information on different real datasets used. All datasets were taken from (CKH+15) and have 2 classes.

us now discuss them in more detail.

Results on *fctdist* Regularization:

First, we see that *fctdist* consistently produces among the highest robustness values as well as among the lowest worst-case-errors in most cases. In fact, the robustness is about twice as large as the ones of the other regularization schemes. Note, though, that even with *fctdist* we did not prove denoising for any dataset because ε is always smaller than δ chosen to be 0.1. We would need to increase the hyperparameter λ in order to obtain autoencoders which surely denoise. However, this may impede downstream classification.
Interestingly, models regularized with *fctdist* also show the lowest worst-case-errors. Thus *fctdist* seems to work as a general-purpose regularizer.

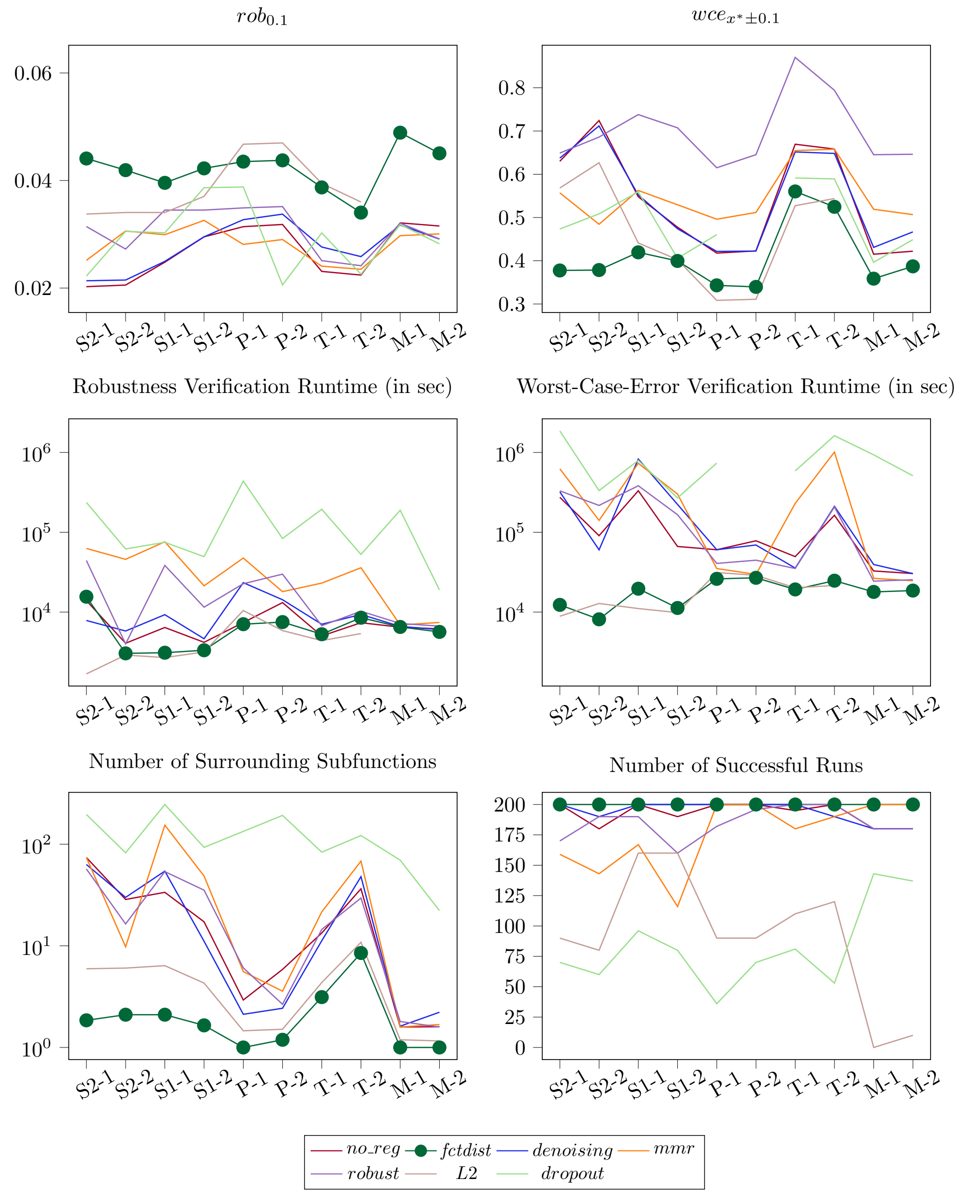


Figure 5.9: Comparison of different regularization schemes with respect to mean robustness, mean worst-case-error, total runtime, mean number of surrounding subfunctions and total number of successful runs. Using the *fctdist* regularization scheme leads to models with the highest robustness as well as the lowest worst-case-error (top row). Moreover, they have the smallest runtime in both the robustness and worst-case-error verification (middle row) resulting from the lowest number of surrounding subfunctions (bottom left). Note that dropout and $L2$ regularization often lead to unsuccessful runs (bottom right). This may be due to an unsound floating point implementation as observed by (DJST18).

Beyond the qualitative improvements, models trained with *fctdist* also exhibit the lowest runtimes for both verification problems as well as the lowest number of surrounding subfunctions. This again confirms the theoretical foundation that simpler models can be verified faster. Also, it is the only approach that could successfully be verified in all cases.

Results on $L2$ and $dropout$:

Next, we observe that both $L2$ and $dropout$ often fail to produce successful runs (bottom right). We found that for $L2$ this can be caused by numerical imprecisions of the SMT solver as the weights become too small and might be due to our particular choice of solver. This phenomenon has also been observed by (DJST18). We also note, though, that this issue may become obsolete with improved verification engines.
For $dropout$, on the other hand, this is due to the timeout of 300 seconds. Again, this result is in line with the estimated number of surrounding subfunctions which is significantly larger than for any other regularization scheme and thus impedes a fast verification. In both experiments, $dropout$ showed by far the largest runtime of all regularization schemes.
These issues render both regularization schemes unattractive if verification of a model is required.

Results on mmr and $robust$:

To our surprise, the mmr regularization did not decrease the number of surrounding subfunctions. Even though it is supposed to push the ReLU boundaries away from the training data, this seems not to affect the simplicity of the resulting model. Therefore we could neither find an increase in robustness nor a decrease in verification runtime.
Also, the $robustAE$ regularization scheme proposed in (ZP17) does not lead to models that are verifiably more robust. It actually seems to destabilize the resulting autoencoder as the worst-case-error becomes larger. However, this method aims to make models robust against noise and anomalies, which we do not evaluate, highlighting the importance of precisely defining what type of robustness one refers to.

Downstream Classification Performance

Finally, we want to compare whether the different regularization schemes lead to different downstream classification performances. We extracted latent space

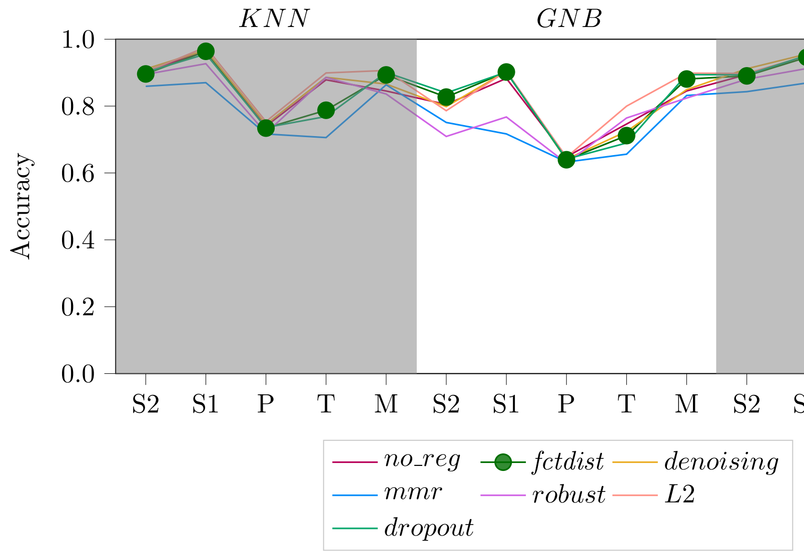


Figure 5.10: Downstream classification performance for three methods (KNN, Gaussian Naive Bayes and Random Forest) of all regularization schemes on all datasets.

representations for all datasets and predicted class labels (recall that the datasets were originally supervised ones) with KNN (GWB$^+$03), Gaussian Naive Bayes (HY01) and Random Forest (Bre01). Figure 5.10 shows that there is no significant difference in performance between the different regularization schemes. Thus even though autoencoders endowed with the *fctdist* regularization scheme are much simpler, there is no decline in performance.
Of course, all of our results depend on many hyperparameters for which we had to make choices. Therefore the magnitude of these results changes according to these hyperparameters. However, these experiments show the general trend towards which each of the regularization schemes heads.

5.6 Summary

As autoencoders gain more and more relevance due to their widespread use, this chapter again addresses the need to formally verify their behavior. In line with similar developments for supervised neural networks, we give the first verifiable robustness definition for autoencoders. We show its use cases ranging from denoising to measuring generalization and build upon the previously introduced SMT-based framework to check whether the robustness property is satisfied.
Moreover, we address the weak spot of verification frameworks: scalability. Yet,

in contrast to improving the SMT solver directly, we instead focus on simplifying the autoencoder. By adjusting its training procedure, we obtain autoencoders with less affine subfunctions leading to a more efficient verification.
Our experiments clearly show the benefits of our approach. Using our approach, we obtain more robust autoencoders and shorter verification runtimes without sacrificing predictive capability. Moreover, we outperform other regularization techniques with respect to these quality criteria. Interestingly, we even show that dropout as a regularization technique is not only *not* beneficial but actually harmful for the verification process.
Based on this chapter, there are several directions for future work. The proposed regularization technique can be transferred to other architectures as well as supervised learning. Also, it would be interesting to apply approximate verification techniques to autoencoders trained with *fctdist*. We conjecture that the approximation gap to exact verification methods would not be as severe as in non-regularized neural networks. Finally, one could also try to exploit the resulting simpler autoencoders for explanatory purposes extending into the realm of interpretable machine learning.
This chapter tries to improve the scalability of verification methods by addressing the training side of neural networks. In the next chapter, we will take another viewpoint: we will see how we can leverage on a different neural network architecture to speed up verification. Thus, we will also consider another type of neural network.

6 Post-Robustifying Existing Models

Anomaly Detection has been an actively researched machine learning problem for several decades. Its use cases range from fault detection in machines (BSBD21) to credit card fraud detection (WW21) as well as to safety-critical areas such as medical diagnosis (FGD$^+$22) or infrastructure control(LBGM18). More recently, outstanding performance was attained by anomaly detectors combining ensemble methods and deep learning.
While researchers have made a lot of progress in designing algorithms for increased detection performance, so far, formal verification of these deep anomaly detectors has been neglected. Yet, especially in safety-critical areas, it is of utmost importance to formally state and prove guarantees about a model's behavior. This need is exacerbated by the discovery of adversarial attacks: inputs designed to fool the model into a wrong prediction (SZS$^+$14). For the anomaly detection task, this corresponds to false positive samples indistinguishable from normal samples in the training dataset.
From these adversarial attacks, a research branch defending against such attacks has emerged. Methods such as adversarial training (MMS$^+$18; GSS15) make neural networks more robust against adversarial attacks by adjusting their training. If, however, these methods yield a non-robust model, there is no method to repair it by robustifying it as a post-processing step.
Moreover, tools to verify that a given model is not prone to adversarial attacks are notoriously slow. Scalability is a major challenge for neural network verification. We address the aforementioned challenges by using the inherent properties of ensemble methods. Given an already trained ensemble model, we first **assess its robustness by a divide-and-conquer approach**. By splitting it up into its submodels, solving a verification problem for each submodel and merging the intermediate results, we obtain both upper and lower bounds for the largest anomaly score in a predefined region.
Moreover, the intermediate results allow us to distinguish between submodels that either do or do not harm the ensemble's robustness. Thereafter we can create a new, robust ensemble model by using only the non-problematic submodels. Thus we obtain a **post-processing method that** - within certain bounds - **robusti-**

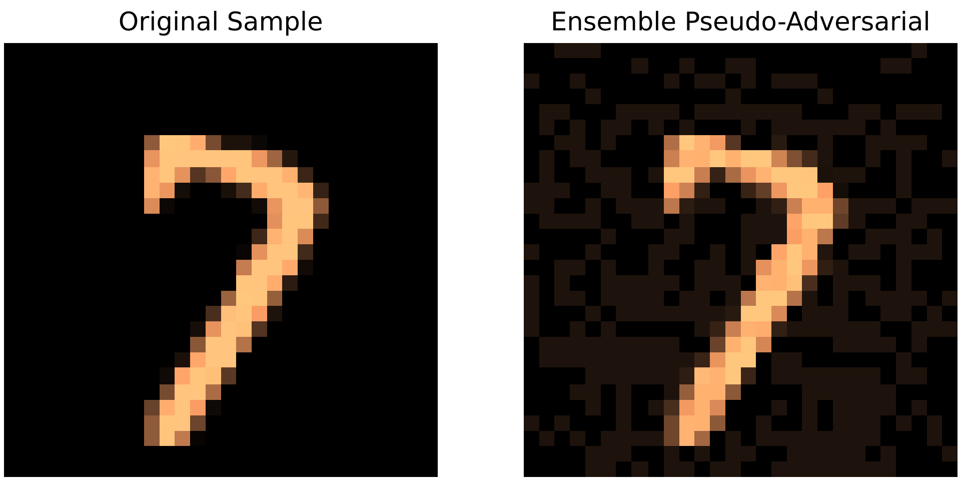


Figure 6.1: Original MNIST sample (left) and a pseudo-adversarial attack (right). Differences between the original and the adversarial attack have been enlarged to make them visible. The original sample is predicted as normal. The pseudo-adversarial attack *p-adv*, on the other hand, is predicted as anomalous even though it is nearly indistinguishable from the original.

fies any such ensemble to the desired degree.
Beyond post-robustification, our method is the **first to produce** a so-called **pseudo-adversarial attack** for an ensemble method as shown in Figure 6.1. These are inputs to the ensemble that are most likely being predicted as anomalous by the ensemble model and narrow down the approximation gap between the upper and lower bound of the anomaly score. They differ from non-pseudo adversarial attacks in that they are combined from adversarial attacks on the submodels and thus, there is no guarantee that the pseudo-adversarial attack is an adversarial attack to the ensemble model.
Using ensemble methods **alleviates another major issue of neural network's formal verification: scalability**. As shown by (XTSM19), the runtime of a verification method increases exponentially with the complexity of neural networks. However, the use of ensemble methods allows us to elegantly circumvent this problem as it is much faster to verify many small neural networks compared to one large neural network.
We highlight the use of our robustification method on the *DEAN* (*D*eep *E*nsemble *A*nomaly *D*etection) ensemble method (KM22). This model combines several advantageous properties: since it employs feature bagging with a large set of simple submodels, we can scale up the verification to datasets of any dimension. Moreover, due to its simple architecture, any particular submodel can be verified

efficiently. Note that we introduce an approximation into the verification process by our divide-and-conquer approach. However, we show that the approximation gap between our estimation of the largest anomaly score and the actual largest anomaly score is very small as our bounds on it are very tight.
With our experiments, we show that we can successfully post-robustify a given *DEAN* ensemble without impairing its predictive performance. Moreover, we compare the robustness of the *DEAN* model to other well-known deep anomaly detectors. As it yields the best performance both in terms of scalability and robustness, we deem it most suitable for verifiable anomaly detection.

6.1 Related Work

Deep Anomaly Detection Methods:
Neural networks have shown exceptional performance in the task of anomaly detection. They assign an anomaly score to each input and compare it to a threshold to determine anomalousness. Due to their implicit feature learning, they are particularly well-suited for complicated distributions on high dimensional datasets (GBC16). For the purpose of post-robustification, we must combine this property with formal verification. However, formal verification proves to be a notoriously difficult task for large-scale neural networks. Therefore we need models that are complicated enough to model complex distributions and simple enough to be verified.
In this work, we primarily employ the *DEAN* model (KM22) because it is an ensemble of many simple submodels. Thereby each particular submodel can be verified while the entire ensemble ensures a good predictive capability.
Moreover, we compare *DEAN* to two representative alternatives: an autoencoder-based ensemble called *RandNet* (CSAT17) and *DeepSVDD* (RVG$^+$18). Both models show state-of-the-art performance in anomaly detection, yet as we will see, they are not competitive in terms of both robustness and scalability.
Neural Network Verification:
Formal verification of neural networks can be categorized into exact approaches such as SMT solvers (Ehl17; KBD$^+$17) and approximation approaches such as abstract interpretation (SGM$^+$18; ZWC$^+$18; WZC$^+$18). For our purposes, we will make use of SMT solvers as this allows us to obtain two types of results for a predefined region: an upper bound on the anomaly score as well as a lower bound derived from a pseudo-adversarial attack. Thus we can estimate the approximation gap to the true largest anomaly score in that region. Abstract interpretation based approaches, on the other hand, could only provide us with an upper bound and do not yield a pseudo-adversarial attack for ensembles.
However, exact solvers are notorious for being much slower than approximation

approaches. We counteract this deficiency by feature bagging and limiting the complexity of each submodel. Overall this leads to an acceptable runtime while producing well-approximated robustness results.
Robustifying against Adversarial Attacks:
Most existing methods, including (MMS$^+$18; ZCH19; CAH19; GSS15), try to incorporate robustness into their training methods. However, if this training does not yield a robust model, one can only retrain it without the guarantee of obtaining a robust model thereafter. Instead, we aim to adjust an already trained network to become provably robust.

6.2 Preliminaries: Anomaly Detection

This section gives the necessary background to understand adversarial attacks on anomaly detection models. This includes general deep anomaly detectors as well as the *DEAN* model (KM22) we use to highlight the use of post-robustification. Beyond being on par with other state-of-the-art deep anomaly detectors, *DEAN* has some very favorable properties for robustness verification.

6.2.1 Deep Anomaly Detectors

First, we need to define what we mean by a deep anomaly detector D. It consists of a (set of) neural network(s) F and an anomaly score $Anom_F : \mathbb{R}^N \to \mathbb{R}$ representing the degree of anomalousness for the input x: the larger $Anom_F(x)$ is, the more likely it is to be an anomaly according to the model. We will usually omit the subscript F for the sake of simplicity.
Moreover, to eventually distinguish between normal and abnormal points, a threshold τ needs to be set such that the anomaly detector becomes a binary classifier: if $Anom(x) > \tau$ we consider x to be an anomaly, else we consider it to be normal. This threshold does also allow us to consider anomaly detection as a verification problem: we either want to verify that in a given set of input points, everything is classified as normal or anomalous. For the models we train, we choose τ such that $Anom(x) \leq \tau$ for 80% of training data points. Next, we will go into detail on how F and $Anom_F$ look like for the particular case of the *DEAN* model.

6.2.2 DEAN Model

The *DEAN* model is a particular type of deep anomaly detector where F is given by an ensemble $F = (f_1, \dots, f_m)$ of relatively simple neural networks. Each submodel f_i consists of a fully-connected neural network with ReLU activations in each hidden layer and a single Linear output neuron. Moreover, there is no bias in any neuron. The input dimension b of f_i is set as a hyperparameter because *DEAN* employs feature bagging on the input: for an input $x \in \mathbb{R}^N$ to D and submodel f_i, the input $\tilde{x}_i \in \mathbb{R}^b$ denotes the projection of x onto the features for f_i. Moreover, these features are chosen randomly for each submodel. The networks f_i are trained to map normal points close to a one-dimensional constant $q_i \in \mathbb{R}$ using the following loss function:

$$L_{f_i}(\tilde{x}_i) = (f_i(\tilde{x}_i) - q_i)^2$$

As (KM22) suggest, we set this constant $q_i = 1$ for training and use the mean on the training set $\mathcal{X}$ given by

$$q_i = \frac{1}{|\mathcal{X}|} \sum_{x_i \in \mathcal{X}} f_i(\tilde{x}_i)$$

for evaluation.
Since the submodels do not have constant bias terms, they cannot simply learn the constant function $f_i(\cdot) = q_i$. Instead, the network needs to learn parameters in such a way that normal inputs result in a low deviation from q_i while all other inputs exhibit large deviations. Therefore this loss can be used to measure anomalousness for a given input similar to (RVG$^+$18).
Finally, we define the anomaly score obtained by *DEAN* given by

$$Anom(x) = \sqrt{\frac{1}{m} \sum_{i=1}^{m} L_{f_i}(\tilde{x}_i)}$$

combining all the outputs and thereby incorporating all input features of x. Moreover, it will average out too large deviations occurring in, for example, just one submodel, resulting in a statistically robust model.

6.3 Problem Setting

There are several notions of robustness both for deep anomaly detectors (NBL$^+$21; ZP17) and for neural networks, in general, (ZSLG16). While a lot of emphasis has been put on training models to be robust, once a model exists, it can only be shown or measured whether it is robust. However, given a non-robust model, it would be useful to just slightly adapt it in order to make it robust instead of retraining a new model from scratch.
Therefore this section poses the challenge to post-process a given model such that it becomes robust. To this end, we will formally introduce the post-robustification problem and provide the necessary definitions.

6.3.1 Post-Robustification

Inspired by verifiable robustness against adversarial attacks in the realm of supervised learning, we want to locally post-robustify models for anomaly detection around a given input x^*. However, the problem can be formulated more generally as it could be transferred to supervised learning as well:

Problem 6.3.1 (Post-Robustification)**.** *Given an evaluation metric m and a non-robust model-input pair (D, x^*), create a new model D^* such that (D^*, x^*) is robust and $m(D^*) - m(D)$ is maximized.*

To work with this general problem formulation, we need to make it more concrete by giving the model, the evaluation metric and the notion of robustness: in this thesis, we choose the previously defined *DEAN* model evaluated by the standard AUC score (Faw06) reflecting the predictive capability in the task of anomaly detection. The precise definition of robustness will be given in the next section.
The given problem definition reflects that we do not only want to robustify our model on x^*, but that we also do not want to trade off too severely with respect to a given evaluation metric. Otherwise, depending on the definition of robustness, post-robustification might result in a degenerate model that maps all inputs to the same output: a model that is very robust but not at all useful.
Note that, assuming a monotone m, we do not want to minimize $|m(D^*) - m(D)|$: if D^* performs even better according to the evaluation metric, we do not consider this a problem.

6.3.2 Adversarial Robustness

Our robustness definition is the direct adaptation of adversarial robustness from supervised learning to anomaly detection. Essentially, we consider the anomaly detector as a binary classifier and apply the definition of (MMS$^+$18) to it.

Definition 6.3.1 (ε-adv-rob)**.** *Let D be an anomaly detector, $dist$ a metric and $x^* \in \mathbb{R}^N$ such that $Anom(x^*) < \tau$. We say that D is ε-adversarial-robust at x^* if and only if for all inputs in $B_\varepsilon(x^*) := \{x \in \mathbb{R}^N : dist(x, x^*) \leq \varepsilon\}$ the **L**argest **A**nomaly **S**core is less than τ:*

$$LAS(D, B_\varepsilon(x^*)) := \max_{x \in B_\varepsilon(x^*)} Anom(x) < \tau$$

According to this definition, a normal input x^* is robust if and only if all surrounding inputs are normal as well. Thus if we can prove ε-adversarial-robustness, we know that there cannot be an adversarial attack in $B_\varepsilon(x^*)$. As in

the previous chapters, $dist$ will be given by the L_∞ distance.
Please note that this is a much stronger notion of robustness than testing against a finite set of test samples. In contrast, we aim to verify the model against infinitely many points defined by the ε environment of x^*. It corresponds to the same notion of robustness against adversarial attacks in supervised learning (MMS+18).
However, this stronger notion of robustness also entails a new problem, as coming up with a new, robustified model D^* is futile if we do not *know* that it is robust. Thus we give a slightly adapted problem formulation.

Problem 6.3.2 (Post-Robustification (ε-adv-rob)). *Given an evaluation metric m and a non-robust model-input pair (D, x^*) according to Definition 6.3.1, create a new model D^* such that $m(D^*) - m(D)$ is maximized and prove that (D^*, x^*) is robust.*

We have now defined the problem we want to solve for the *DEAN* model. However, we need a further subproblem which we will solve for each submodel of *DEAN* to establish robustness on a given *DEAN* model.

6.3.3 Adapting the Worst-Case-Error Problem

Given that *DEAN*, among other deep anomaly detectors, bases its anomaly score on the deviation to a desired output, we introduce a slightly adapted version of the worst-case-error problem (Problem 4.2.1 given in Section 4.2). If we can bound the worst-case-error of all submodels in the region of interest $B_\varepsilon(x^*)$, we can also bound the anomaly score of the *DEAN* model and thus know whether it will ever predict *anomalous* for any input in that region.

Definition 6.3.2 (*DEAN*-submodel wce). *Let $f : \mathbb{R}^b \to \mathbb{R}^1$ be a DEAN submodel, $A \subset \mathbb{R}^b$ an input region and q the target value of f. The worst-case-error is given by:*

$$wce(f, A) = \sup_{x \in A} \|q - f(x)\|_\infty$$

Moreover, we want to calculate an input that realizes wce up to a predefined accuracy. Sticking to the notation introduced in Section 4.2 we call these *wce adversarial attacks*.

Definition 6.3.3 ($wce - adv$). *In the context of definition 6.3.2 we define a wce adversarial attack (denoted wce-adv) as an input x such that*

$$\big|\|q - f(x)\|_\infty - wce(f, A)\big| < acc$$

where acc is a predefined accuracy.

Now that we have all the required definitions, we can build our way backward to solve each of these problems. This is the content of the next section.

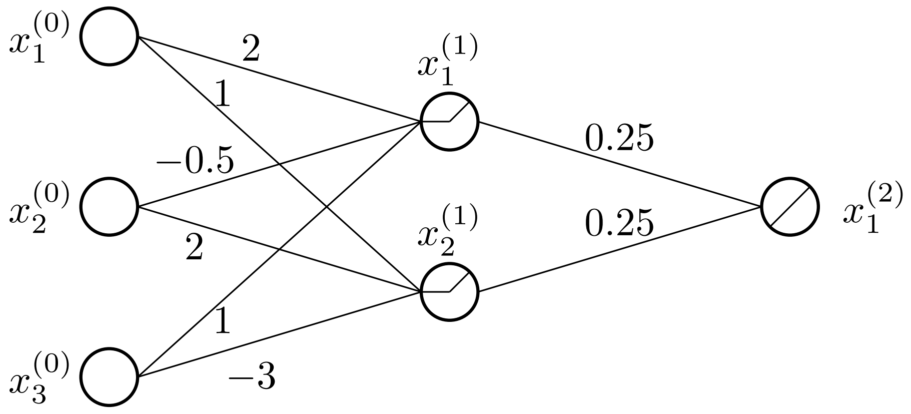


Figure 6.2: Exemplary *DEAN* submodel for explaining the construction of SMT formulas.

6.4 Solution Framework

In this section, we provide a solution framework for post-robustifying a deep anomaly detection ensemble. Reversing the structure of the previous section, we start by calculating wce on a *DEAN* submodel followed by verifying the *DEAN* ensemble and finally post-robustifying it.

6.4.1 Calculating wce for a DEAN submodel

To calculate wce for a given *DEAN* submodel *f* we will slightly adapt the solution provided in Section 4.3 and make use of so-called SMT solvers. In essence $wce(f, B_\varepsilon(x^*))$ is obtained by repeatedly checking existence of an input x in $B_\varepsilon(x^*)$ such that $\|f(x) - q\| > \delta$. Using this δ-check as a subroutine we can approximate $wce(f, B_\varepsilon(x^*))$ with a predefined accuracy using binary search over δ. Therefore in the following, we only need to describe how to use an SMT solver for a particular value of δ.
While an SMT solver can be used in a very versatile way, in this section we will build upon previous chapters, explaining only what we need to provide and what in return we obtain as a solution. For a more detailed description of SMT solvers, please refer to Section 2.2.2 or (Ehl17).
For our specific use case, we will provide a boolean combination ($\wedge$, $\vee$, $\neg$) of linear inequalities, the *formula*, as input to our SMT solver of choice (WOZ$^+$20). These linear inequalities will encode the *DEAN* submodel as well as the δ-check, similar to Section 4.3.
As a result of the verification process, we will obtain either $unsat$ proving that no input resulting in a distance to q exceeding δ exists, or sat together with a sample

$x \in B_\varepsilon(x^*)$ such that $\|q - f(x)\|_\infty > \delta$. We will keep track of the last sample obtained during the binary search as this will be the *wce-adv*.
As stated in the previous chapters, SMT solvers allow only for piecewise affine activation functions. In this sense, *DEAN* is a favorable choice because it only employs ReLU and Linear layers.
Similar to the problem formulation of *wce* for autoencoder we require three formulas, φ^f, φ_δ and φ_ε, encoding the δ-check. The neural network will again be encoded in φ^f while φ_δ and φ_ε are given by

$$\varphi_\delta = [x_{L,1} \geq q + \delta] \vee [x_{L,1} \leq q - \delta]$$

and

$$\varphi_\varepsilon = \bigwedge_{1 \leq k \leq N} [x_{0,k} \leq x_k^* + \varepsilon] \wedge [x_{0,k} \geq x_k^* - \varepsilon]$$

respectively. Basically, the formula φ_δ is satisfied if and only if the error between $f(x) = x_1^{(L)}$ and q is large enough, while φ_ε restricts the SMT solver to search for a point x in an $\varepsilon-$environment of x^*. Eventually, the formula presented to the SMT solver is given by

$$\varphi^f_{\delta, B_\varepsilon(x^*)} := \varphi^f \wedge \varphi_\varepsilon \wedge \varphi_\delta.$$

If the SMT solver returns *unsat* for this formula, we know that for all points in $\{x \in \mathbb{R}^n : \|x - x^*\|_\infty < \varepsilon\}$ the distance between $f(x)$ and q is less than δ. If on the other hand the solver returns *sat* and a solution $\tilde{x}$ which satisfies $\|\tilde{x} - x^*\|_\infty < \varepsilon$ and $\|f(\tilde{x}) - q\|_\infty > \delta$, we directly obtain an example resulting in a large deviation.
To further ease understanding, we will present a small example: for the simple *DEAN* submodel given in Figure 6.2, $x^* = (1, 2, 1)$, $f(x^*) = 1$, $\varepsilon = 0.1$ and $\delta = 0.1$ the SMT formula looks as follows:

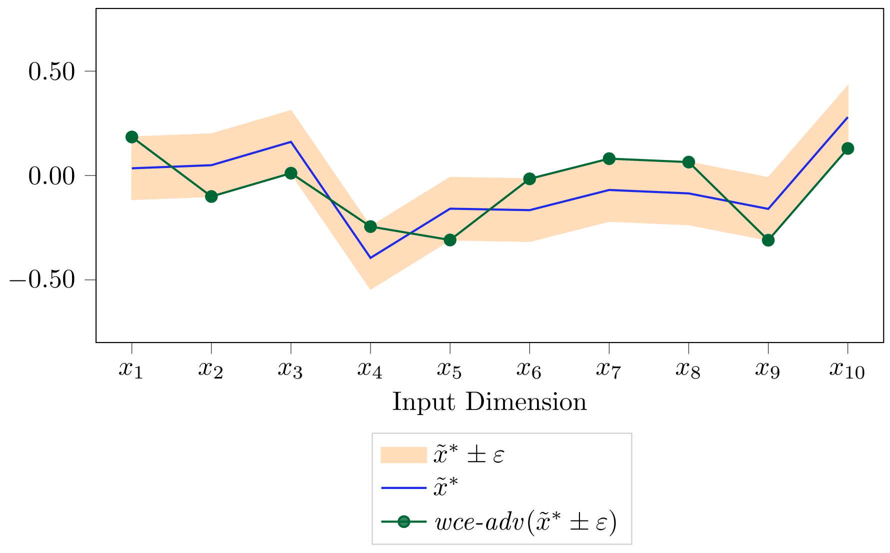


Figure 6.3: Visualization of *wce-adv*. For each dimension (x-axis) *wce-adv* is usually given by either $\tilde{x}^* - \varepsilon$ or $\tilde{x}^* + \varepsilon$. Thus *wce-adv* is a *corner* of $B_\varepsilon(\tilde{x}^*)$

Example 6.4.1.

$$\begin{aligned}\varphi^f := \;&(y_{1,1} = 2x_{0,1} - 0.5x_{0,2} + x_{0,3})\\ &\wedge(y_{1,2} = x_{0,1} + 2x_{0,2} - 3x_{0,3})\\ &\wedge(x_{1,1} = \mathsf{ite}(y_{1,1} < 0, 0, y_{1,1}))\\ &\wedge(x_{1,2} = \mathsf{ite}(y_{1,2} < 0, 0, y_{1,2}))\\ &\wedge(x_{2,1} = 0.25x_{1,1} + 0.25x_{1,2})\end{aligned}$$

$$\begin{aligned}\varphi_{\varepsilon=0.1} :=&(x_{0,1} \geq 0.9)\\ &\wedge(x_{0,1} \leq 1.1)\\ &\wedge(x_{0,2} \geq 1.9)\\ &\wedge(x_{0,2} \leq 2.1)\\ &\wedge(x_{0,3} \geq 0.9)\\ &\wedge(x_{0,3} \leq 1.1)\end{aligned}$$

$$\begin{aligned}\varphi_{\delta=0.1} :=&(x_{2,1} \geq 1.1)\\ &\vee(x_{2,1} \leq 0.9)\end{aligned}$$

The solution for this instance is sat with, for example, the following variable assignments:

$$\begin{aligned}
x_{0,1} &= 1.1\\
x_{0,2} &= 2.1\\
x_{0,3} &= 0.9\\
x_{1,1} &= y_{1,1} = 2.05\\
x_{1,2} &= y_{1,2} = 2.6\\
x_{2,1} &= 1.1625
\end{aligned}$$

Note however, that for $\delta = 0.2$ no solution exists proving that wce must be between 0.1 and 0.2.

6.4.2 Robustness Verification

Having established a procedure to calculate both the wce value as well as *wce-adv* on a *DEAN* submodel, we now show how to verify the entire *DEAN* model. The general procedure of our verification framework is given by splitting up the ensemble model, calculating wce and *wce-adv* on each submodel and merging the results. In the following, we provide details on these steps.

Splitting:

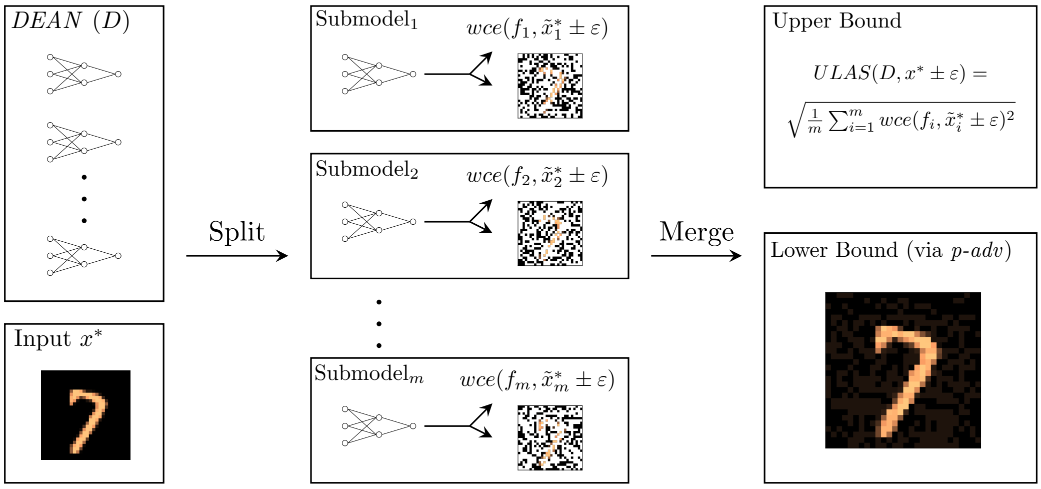
We split the *DEAN* model into each of the submodels it consists of. Thus, if $D = (f_1, ..., f_m)$ is the *DEAN* model, we consider each f_i separately in the next step. Note also that for the next step, we need to respect the feature bagging. If we want to check robustness for a given input x^* each submodel f_i is verified on $B_\varepsilon(\tilde{x}_i^*)$.

Submodel Verification:

For each submodel f_i we solve the adapted version of the worst-case-error problem as described in Section 6.4.1. Thereby we obtain two outputs. On the one hand, we obtain a value wce of each submodel and, on the other hand, an input *wce-adv* that realizes wce up to a predefined accuracy. Note, though, that the input is not for the entire *DEAN* model but just for a single f_i model.

Merging Outputs:

From the two outputs obtained for each submodel, we will extract an upper and a lower bound for the largest anomaly score $LAS(D, B_\varepsilon(x^*))$. Recall that if the



$DEAN$ (D)
Input x^*
Split
Submodel$_1$
$wce(f_1, \tilde{x}_1^* \pm \varepsilon)$
Submodel$_2$
$wce(f_2, \tilde{x}_2^* \pm \varepsilon)$
Submodel$_m$
$wce(f_m, \tilde{x}_m^* \pm \varepsilon)$
Merge
Upper Bound
$ULAS(D, x^* \pm \varepsilon) =$
$\sqrt{\frac{1}{m}\sum_{i=1}^{m} wce(f_i, \tilde{x}_i^* \pm \varepsilon)^2}$
Lower Bound (via p-adv)

Figure 6.4: Verification Process of the DEAN model. We first split DEAN and the input (left) up into the submodels and their features. Thereafter on each submodel, both *wce* and *wce-adv* are calculated (middle), followed by merging the results into the upper bound and the pseudo-adversarial attack *p-adv* (right). Note that due to feature bagging, each submodel's *wce-adv* does not cover all input dimensions.

upper bound is lower than the anomaly threshold τ we prove local robustness. The lower bound, on the other hand, is used to estimate the approximation gap to $LAS(D, B_\varepsilon(x^*))$.
We **U**pper bound the **L**argest **A**nomaly **S**core by:

$$ULAS(D, B_\varepsilon(x^*)) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} wce(f_i, B_\varepsilon(\tilde{x}_i^*))^2}$$

We replace the error of each model with the largest error that can possibly manifest for each submodel. This is an overapproximation of $LAS(D, B_\varepsilon(x^*))$ because a single feature of x^* might be shared by multiple submodels. These submodels, however, might realize their *wce-adv* with different values for this particular feature. Yet inputs to the *DEAN* model can, of course, only have one value for every dimension.
Therefore we construct the lower bound of the anomaly score by combining the adversarial attacks *wce-adv* of each submodel f_i into a pseudo-adversarial attack for *DEAN*. To this end, we leverage a property of the adversarial attacks *wce-adv* obtained by our subroutine: usually they are at a corner of the input

space[1] $B_\varepsilon(\tilde{x}_i^*)$. Thus for $(\tilde{x}_i^*)_k$ being the k'th dimension of $\tilde{x}_i^*$ they are given by $y_k^{(i)} \in \{(\tilde{x}_i^*)_k + \varepsilon, (\tilde{x}_i^*)_k - \varepsilon\}$ (see Figure 6.3).
Taking the perspective of a particular dimension p, there are several submodels that have this dimension as input due to feature bagging. To combine the adversarial attacks, we simply employ a majority vote among these submodels to determine which side of the corner we choose.
Thus let J be the index set of submodels using feature p, $l(j)$ be the respective index corresponding to feature p in each of these submodels and $\{y_{l(j)}^{(j)} : j \in J\}$ be the corner points obtained for dimension p by their respective unsupervised adversarial attacks. Then we construct a pseudo-adversarial attack for the ensemble as

$$p\text{-}adv(B_\varepsilon(x^*))_p := \text{mode}\{y_{l(j)}^{(j)} : j \in J\}.$$

From this pseudo-adversarial attack, we extract a lower bound for $LAS(D, B_\varepsilon(x^*))$ by simply calculating $Anom(p\text{-}adv(B_\varepsilon(x^*)))$. Essentially, we try to combine the adversarial attacks of each submodel in such a way that many of the submodel's errors become large, thereby tailoring a pseudo-adversarial attack for the ensemble model. We call it a pseudo-adversarial attack because we cannot guarantee that it is indeed an adversarial attack to the *DEAN* model. After all, it is just a lower bound for the anomaly score. We will show experimentally, though, that it is close to the upper bound resulting in a very small approximation gap.

Properties of Ensemble Verification:

Robustness verification of an ensemble comes with both advantages and caveats. One major advantage is with respect to scalability. Using an ensemble of simpler models allows us to break down the difficult problem of verifying a large neural network into smaller subproblems, each of which can be solved in a much shorter time. Since verification time increases exponentially in the number of ReLU nodes (XTSM19), the size of networks that can be verified is limited. Thus ensemble methods of simpler models are a natural aid for scalability.
Secondly, feature bagging - a method only applicable to ensemble methods - allows us to scale up the verification procedure to datasets of any dimension.
Finally, as we will see in the experiments, the *DEAN* ensemble is more robust in the previously defined sense than other deep anomaly methods making them the go-to choice for robust anomaly detection.
Still, there is a trade-off we take by verifying each of the submodels independently:

[1]Of course, this depends strongly on the value of ε. The larger the space in which we search for the $wce - adv$, the larger the chance that it is not at a corner. We conducted follow-up experiments of that for autoencoder in (Vu23).

Algorithm 2: Post-Robustify

```
  Input: $\tau$, $L = \left((f_i, wce(f_i, B_\varepsilon(\tilde{x}_i^*)))\right)_{i=1,\dots,m}$
1 $L_{sort} = sort(L)$ by $wce$
2 while $ULAS(L_{sort}, B_\varepsilon(x^*)) > \tau$ do
3     $L_{sort} \leftarrow L_{sort} \setminus max(L_{sort})$
4 return $models(L_{sort})$
```

we only approximate the largest anomaly score on the entire net because we are not able to consider inputs to different submodels jointly. We will see empirically, though, that our approximation is very close.

6.4.3 Robustify

Endowed with the capability to determine robustness, we will now present a simple procedure with which we can post-process a non-robust model-input pair based on each submodel's wce such that it becomes robust: we sort all submodels by their wce and remove them one by one starting with the largest wce until the upper bound on $LAS(F, B_\varepsilon(x^*))$ of the remaining models is below the anomaly threshold τ. This way, we can guarantee that in $B_\varepsilon(x^*)$ the resulting ensemble will never predict *anomalous* ensuring robustness.

The exact procedure is given in Algorithm 2 and is based on the assumption that there are a few submodels whose worst-case-errors exceed τ by a substantial margin (see Figure 6.5). In that case, we only need to remove a small portion of the submodels, thus retaining a lot of the ensemble's predictive capability.

This seemingly simple procedure has to be executed with care given the following limitations: first, by reducing the number of models, we might impair the predictive capability of the ensemble. We will show experimentally that this trade-off is not severe, but of course, this depends on the level of robustness one wants to achieve. Secondly, there is a limit of robustness that we cannot overcome simply given by the smallest worst-case-error of any submodel. Thus, if ε is too large (if we want too much robustness) post-robustification by removing submodels becomes impossible and Algorithm 2 returns an empty list.

6.5 Experiments

This section highlights the use of post-robustification for a given *DEAN* model. We will start by looking into a single ensemble model trained on MNIST (Den12), showing what useful results can be obtained with our method. Thereafter we will

compare *DEAN* with a) an autoencoder ensemble (*RandNet*) (CSAT17) and b) the *DeepSVDD* model (RVG+18) on 8 other real-world datasets highlighting that *DEAN* models are more robust from the beginning, allow for post-robustification on each of these datasets, and - in contrast to RandNet and DeepSVDD - keep a constant runtime across all datasets.
The code and more details on our experiments can be found at `www.github.com/KDD-OpenSource/Robustify`.

6.5.1 Deep Dive MNIST

Our first experiments are conducted on the MNIST dataset. Here we train an ensemble of 1000 *DEAN* submodels with feature bagging of size 32 to highlight properties of the *Post-Robustify* method on a single ensemble model.

Remaining Predictive Capability:

Assuming that we started with a powerful predictor, our first experiment addresses whether, by post-robustifying, the model loses its predictive capability. As shown in Figure 6.5, after post-robustification based on one point, we can keep 856 of the original models and sacrifice almost no AUC. Indeed we could have deleted more than 50% of the submodels before witnessing a severe drop in AUC score.

Approximation of the Largest Anomaly Score:

As we cannot directly calculate LAS, we must approximate it. Recall that we obtained an upper bound on the LAS by aggregating over each submodel's *wce* and a lower bound by combining each submodel's *wce-adv* into a pseudo-adversarial attack for the *DEAN* model. We can use both bounds to test how accurate the approximation of LAS for the *DEAN* ensemble is. Since we have no theoretical guarantee on the size of the approximation gap, we empirically evaluate it for a given *DEAN* model on 20 samples in Figure 6.6. Surprisingly, the relative error between the approximation gap and the actual LAS is always less than 2% showing that our approximation scheme is very precise.

Local vs. Global Robustness:

Finally, we investigate the global effects of local post-robustification. Figure 6.7 shows that we significantly decrease $ULAS$ for other normal samples by local post-robustification. It appears that the models we delete by post-robustifying one sample also cause high worst-case-errors on other samples. Therefore our

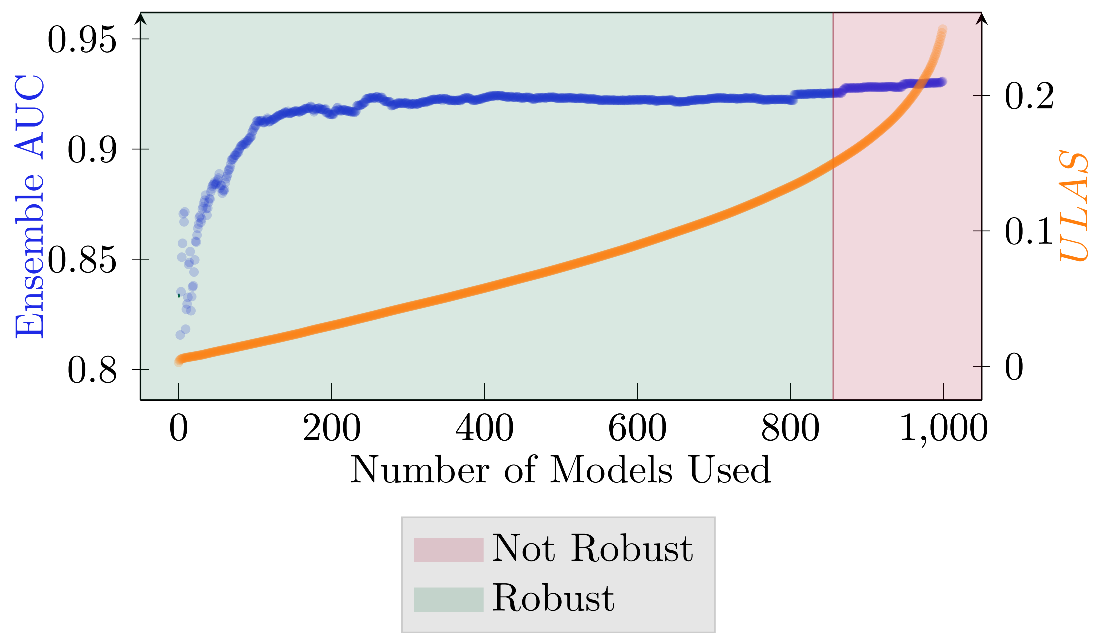


Figure 6.5: ROC-AUC Score and $ULAS$ of the *DEAN* Ensemble as a function of the number of models used to generate it. Here we use only the submodels with the lowest error and consider an ensemble with $ULAS \leq 0.15$ to be robust as defined by τ. Calculating these individual errors does not require labels, so our process is still unsupervised.

method increases robustness not only locally but also globally.
However, an important remark is that in order to be truly robust at multiple points, one needs to successively apply the post-robustifying step in each of which the trade-off with predictive capability is made. As seen in Figure 6.11 though, the trade-off becomes less severe with a growing number of local post-robustifications.

6.5.2 Comparison to RandNet and DeepSVDD

This section compares *DEAN* to two representative, alternative models: *RandNet* (CSAT17) and *DeepSVDD* (RVG$^+$18). While *RandNet* consists of an ensemble of autoencoders, thereby being directly comparable to *DEAN*, *DeepSVDD* consists of a single, large neural network. Thus even though we cannot directly apply our post-processing method to *DeepSVDD*, we highlight how *DEAN* outperforms it with respect to runtime and robustness.

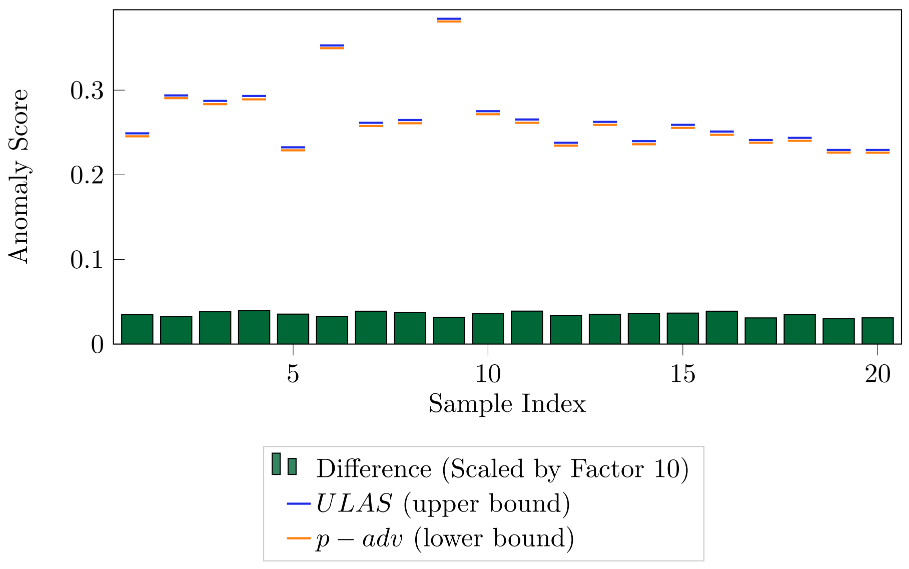


Figure 6.6: LAS Approximations (upper bound through averaging submodel's *wce* and lower bound through combining *wce-adv*) for 20 different samples. For every sample tested, we know the value of LAS up to a relative error of less than 2%.

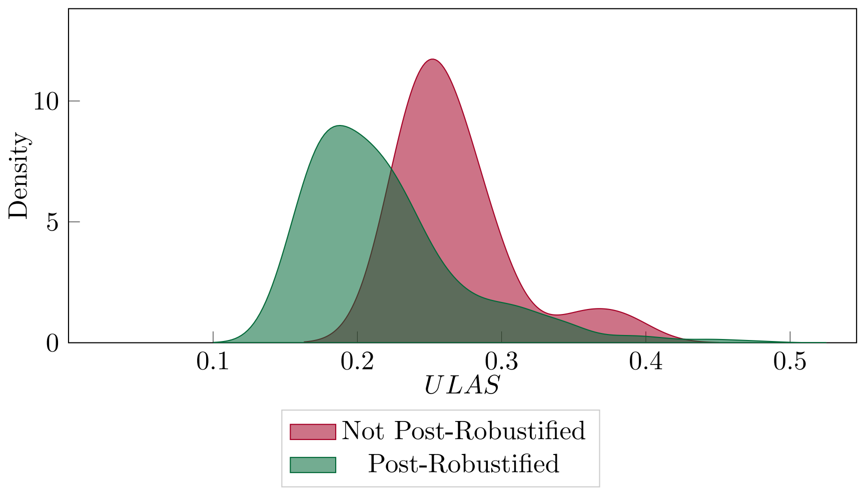


Figure 6.7: Gaussian Kernel Density Estimation plot of $ULAS$ of 20 samples before (red) and after (green) application of Post-Robustify. After post-robustifying on a particular sample, we calculated $ULAS$ on different samples to obtain the green density.

Dataset	Features	RandNet	DEAN	DeepSVDD
pageblocks	10	0.9231	0.9577	0.8748
segment	18	0.9982	0.9998	0.9981
steelplates (oSoC10)	27	0.7521	0.7329	0.718
wbc	30	0.941	0.9751	0.9336
satellite	36	0.8321	0.8196	0.8233
qsarbiodeg (MRB+13)	39	0.8734	0.7425	0.821
gasdrift (VVA+12)	128	0.9791	0.9319	0.9562
har (AGO+13)	561	0.9786	0.9529	0.9371
Average		0.9097	0.8890	0.8828

Table 6.1: ROC-AUC Scores on the 8 datasets used in this thesis.

Datasets:

We choose eight different datasets with varying numbers of features. These are chosen from (Ray16) and (VvRBT13) such that each algorithm achieves a similar AUC score as shown in Table 6.1. For each dataset, we performed the following experiments based on 20 samples for post-robustification. Moreover, we repeated each experiment with 10 different models using 100 submodels each. Thus, for example, we have a total of $20 \cdot 10 = 200$ post-robustifications for the *DEAN* model on the *pageblocks* dataset.

Verification of RandNet and DeepSVDD:

For both models, we need to define how to verify their robustness. To this end, we need to upper bound their anomaly score (via $ULAS$) to thereafter compare this bound to τ.
For the *RandNet* model $RN = (ae_1, \dots, ae_m)$ the anomaly score is given by the median of the reconstruction error $\|x - ae_j(x)\|_2$ they used for each model ae_j. However, as described before, SMT solvers cannot handle the L_2 norm. Thus, we proceed similar to the *DEAN* model: we calculate wce for each submodel by

$$wce(ae_j, B_\varepsilon(x^*)) = \max_{x \in B_\varepsilon(x^*)} \|x - ae_j(x)\|_\infty$$

and join the results. We can bound the L_2 norm in $B_\varepsilon(x^*)$ by the worst-case-error

$$\max_{x \in B_\varepsilon(x^*)} \|x - ae_j(x)\|_2 = \max_{x \in B_\varepsilon(x^*)} \sqrt{\sum_{i=1}^{N} (x_i - (ae_j(x))_i)^2} \leq \sqrt{N} \cdot wce(ae_j, B_\varepsilon(x^*)).$$

Eventually, using the median as their aggregation function, we get

$$ULAS(RN, B_\varepsilon(x^*)) = \underset{i\in 1,\ldots,m}{\text{median}}(\sqrt{N} \cdot wce(ae_i, B_\varepsilon(x^*))).$$

Note that we probably strongly overestimate the largest anomaly score given that we replace each dimension in the reconstruction error by the worst-case-error yielding a factor of $\sqrt{N}$ for the approximation. However, this is the best way in which we can bound the anomaly score and highlights that the *DEAN* model is much more suitable for post-robustifcation due to its architecture with just one output neuron.
For a *DeepSVDD* model DS, the anomaly score is given by the Euclidian distance (L_2-norm) to a (given) parameter c. Similar to *RandNet* we instead need to calculate the L_∞-norm as wce:

$$wce(DS, B_\varepsilon(x^*)) = \max_{x\in B_\varepsilon(x^*)} \|DS(x) - c\|_\infty$$

We can ensure robustness only via the largest L_∞-ball fully contained in the $L2$-ball of radius c. Therefore, if p is the model's output dimension, the upper bound of the anomaly score is given by

$$ULAS(DS, B_\varepsilon(x^*)) = \sqrt{p \cdot wce(DS, B_\varepsilon(x^*))^2}.$$

Ratio Comparison:

As the anomaly scores of different models can have different scales, equation 6.1 defines the so-called Change Ratio (CR) to make the results on different models comparable. It indicates by what factor the model differs from a robust one: the model is already robust for $CR \leq 1$ while for $CR > 1$ we need to reduce $ULAS$ by a factor CR to obtain a robust model.

$$CR(D, x^*, \varepsilon) := \frac{ULAS(D, B_\varepsilon(x^*))}{\tau(D)} \tag{6.1}$$

Figure 6.8 shows that for every dataset, *DEAN* is already the most robust, often having a $CR \leq 1$. While *DeepSVDD* is already less robust, *RandNet* has change ratios more than 100 times higher than those of *DEAN*. At least partly this is due to a larger approximation error of $ULAS$ as the different algorithms use different norms (L_2 or L_∞) for their anomaly scores. Therefore the simple architectures of each of *DEAN*'s submodels favor a precise calculation of $ULAS$. Moreover, the given change ratios render the *RandNet* model unfit for post-robustification as we cannot reduce $ULAS$ enough to be below the anomaly threshold τ. For the

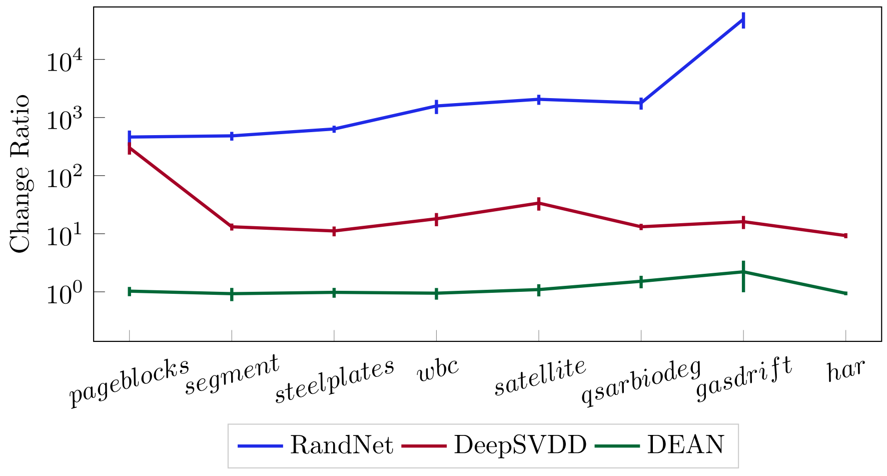


Figure 6.8: Change Ratio on eight datasets averaged over ten runs and twenty samples for post-robustification each. DEAN has the lowest overall change ratio. Thus we do not need to adjust it significantly to post-robustify one sample.

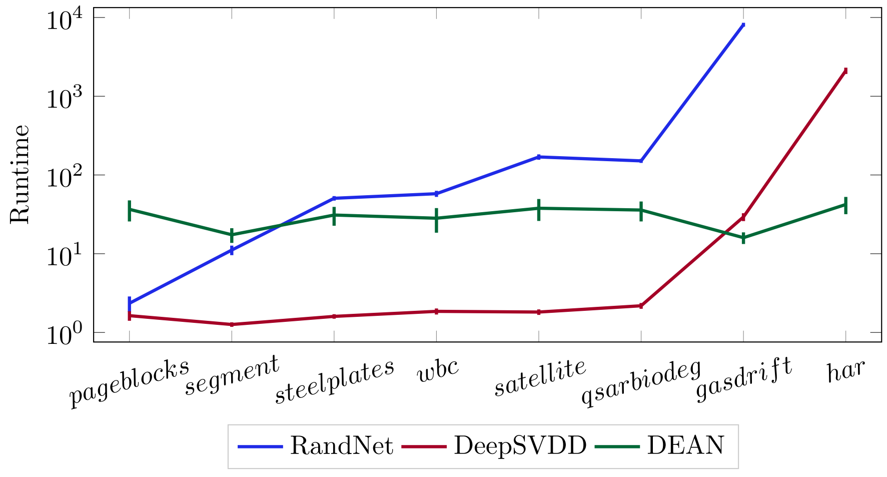


Figure 6.9: Runtime on eight datasets averaged over ten runs and twenty samples each. The computational complexity for *RandNet* and *DeepSVDD* increases with the number of features (note that the datasets are sorted by their number of features). For the highest dimensional dataset "har" with 561 features, it was impossible to verify the autoencoder due to timeout.

DeepSVDD model, on the other hand, post-robustification may be valid in some cases. However, in this case, we have no methodology to actually reduce $ULAS$. Figure 6.9 shows the runtime of the verification algorithm. As the x-axis is sorted by the number of features in each dataset, we show the drastic increase in required verification time both for *RandNet* and, to a lesser extent, also for *DeepSVDD*. In contrast, the required verification time for the *DEAN* model stays constant over all datasets as feature bagging allows keeping the number of ReLU nodes in each submodel the same.
Even though *DEAN* seems to have a larger runtime on lower-dimensional datasets, note that - unlike *DeepSVDD* - its verification process can be parallelized along the submodels. Moreover, it depends linearly on the number of submodels verified. Therefore the eventual runtime of *DEAN* verification can be controlled both with the number of submodels to be verified and with the number of CPU cores available.

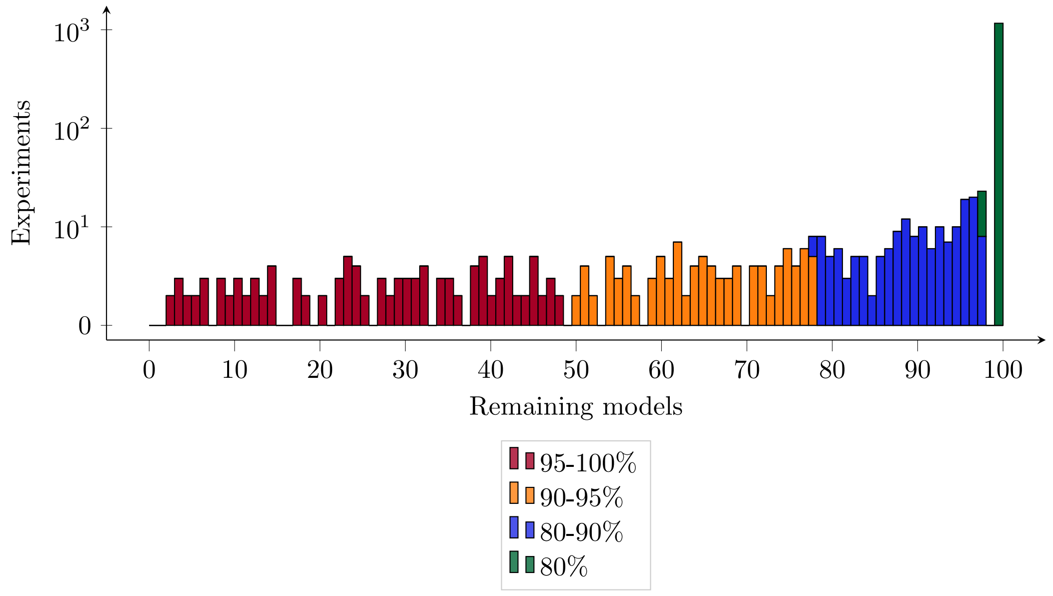


Figure 6.10: Number of models deleted by post-robustification. To reach robustness around a single sample, we must remove less than 10% of models on average. For 90% of verified samples, we need to remove at most 22% of submodels. Just a few samples are hard to post-robustify and require removing more submodels.

Towards Global Robustness:

Finally, we want to build on the results obtained in section 6.5.1. As we have seen, local post-robustification impacts robustness globally. Therefore Figure 6.11 shows the number of remaining models after *repeatedly* applying post-robustification on different samples. It appears that on all datasets, we can post-robustify up to 10 different samples without sacrificing too many models.
However, there are a few samples that would delete more than 50% of the models, as can be seen on the left-hand side. For these cases, post-robustification would have a severe impact on the remaining AUC score.

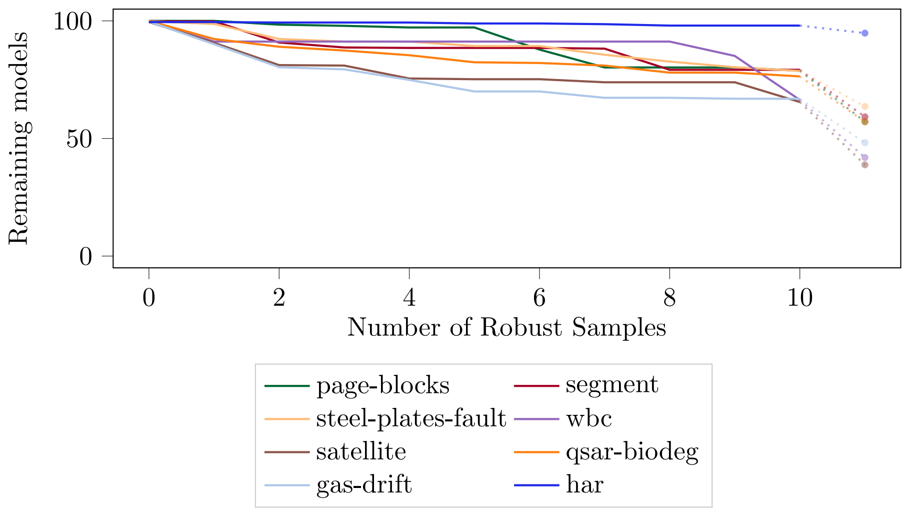


Figure 6.11: Number of remaining models after *repeated* post-robustification. We test if we can post-robustify a given ensemble for multiple samples. We show the number of remaining models on the right side after post-robustifying up to 10 samples averaged over ten different ensembles and samples. A few hard-to-verify samples affect this curve strongly. Hence we choose the ten samples that are easiest to post-robustify for every model. However, we show the effect of also including the final sample as point 11.

6.6 Summary

In this chapter, we study the formal verification of ensembles for anomaly detection. We first develop the necessary theoretical notions and thereafter show not only how to assess a given anomaly detection ensemble with respect to robustness but also how to repair a non-robust model.

In our extensive experiment section, we demonstrate our new method - Post-Robustify - on different datasets highlighting that post-robustification is almost always possible, even without a sacrifice in prediction performance. Moreover, we compare different deep anomaly detectors. The *DEAN* model seems particularly suitable for post-robustification, as it outperforms common alternatives in terms of scalability and the degree of change required to obtain a robust model. Also, using feature bagging, *DEAN* achieves a verification time independent of the number of features of the dataset. Thereby, using the ensemble structure of the *DEAN* model, we overcome the major challenge of the previous chapters - albeit only for this type of model.
Although we also show that local post-robustification affects robustness globally, there is a limit of points for which we can ensure it. Thus, to achieve global robustness, more sophisticated algorithms and perhaps even another definition of robustness is needed.

7 Summary and Future Work

This thesis bridges the gap between supervised and unsupervised neural networks in neural network verification. We define new problem specifications to be verified, show their usefulness, analyze and tackle challenges in scalability and demonstrate how to increase a given model's robustness. This section gives both a summary of the research contributions as well as an outlook into future work and remaining open challenges.

7.1 Contributions

As a first major challenge, we need to give ourselves precise problem definitions. While for supervised neural networks - among other properties - robustness against adversarial attacks is a well-established verification problem, unsupervised neural networks lack such problem definitions. Thus, we paradigmatically introduce such new verification problems for the case of autoencoders. First, we introduce the worst-case-error, which at its core builds upon the innate training property of every autoencoder: the reconstruction of the input. It determines the largest reconstruction error in a given region of interest. Secondly, we define ε-δ robustness for the autoencoder. It asks the question of how much (ε) we must change the input to achieve a particular change (δ) in the output. Since the autoencoder is a very versatile model with many use cases and both verification problems describe very general properties, we can apply them in different settings. For example, we can use them to show that classes do not get mixed up in the latent space when we use the autoencoder for dimensionality reduction. Moreover, for the use case of anomaly detection, we can describe the region in which, according to the autoencoder, the normal or anomalous points can be found. Finally, we show how using the presented verification problems, we can prove that a given autoencoder denoises. Note that these applications are in stark contrast to supervised neural network verification, which in most cases, is based on a change of classes. Similarly, our use cases are not based on changes of classes but are inspired by what a particular unsupervised neural network is used for. The second major challenge we tackle in the thesis is verification scalability.

Similar to the supervised case, the runtime of unsupervised neural network verification grows exponentially as the network's size increases. Neural network verification is NP-complete. Therefore we add to the current body of research on verification scalability. However, we deviate from the most common approach to tackle it. Instead of directly improving on the verification method itself, we train neural networks to be amenable for verification with a regularizer called *fctdist*. By exploiting the theoretical foundation of the piecewise affine structure of our neural networks, we achieve a significant reduction in the number of subfunctions. As constraint-based verification methods must, in one form or another, iterate over these subfunctions, this drastically decreases verification runtime. Another approach to decrease verification runtime is given by using ensemble methods. This effectively circumvents the need to verify one large neural network and instead relies on dividing the problem into many subproblems. In this case, the verification runtime does not increase exponentially with the size of the neural network but instead just linearly with the number of submodels used in the ensemble.
The third challenge we address in the thesis is to train models for more robustness. The regularizer *fctdist* does not only reduce verification runtime but instead also increases robustness of the resulting model. The simplification reached by reducing the number of affine subfunctions yields models which locally have a more linear behavior and are thus more stable. Moreover, we demonstrate that adversarial training - as inspired by the case of supervised learning - improves resilience against the adversarial attacks we define in the thesis.
The fourth challenge we address in the thesis is post-robustification. Motivated by the potentially high costs of training, it asks to repair a model that is broken. More precisely, we want to make a non-robust model robust without retraining from scratch. We show how to solve that problem for ensemble models. In particular, we consider the DEAN model, which is an anomaly detection ensemble of many small DeepSVDD models employing feature bagging. We aim to bound the model's anomaly score in an area to prove that all points are predicted as *normal* in it. By splitting the ensemble and verifying each subnetwork separately, we can identify which subnetworks harm robustness. Merging the results gives an approximate verification procedure for the entire model. It is only an approximate solution because we search for the largest anomaly score for each submodel individually. However, the combined anomaly score overestimates the actual largest anomaly score. Therefore we estimate the approximation gap by giving a lower bound. Since the approximation gap is surprisingly small, we obtain a reasonable alternative to large-scale non-ensemble models.

7.2 Future Work

Based on the presented contributions, many open research questions both of practical and theoretical interest arise. While we made significant progress on the question of scalability, large scale neural networks still remain out of reach for exact verification methods. This is due to their enormous size making an exhaustive exploration virtually impossible. However, current methods only optimize the mathematical framework into which the problem has been translated. We believe that this could be improved upon if a more geometrical rather than optimization viewpoint was taken.
Any ReLU activation pattern corresponds to a particular subset of the input space. While current methods base the order in which they consider the different activation patterns on, for example, lower or upper bounds on the ReLU neurons, they do not consider how close other subspaces of other ReLU activation patterns are. However, if one wants to check, example given, for robustness, not all activation patterns need to be considered and only those in close vicinity to the input point matter. Therefore we might achieve a verification speedup by exploring the surrounding of x in a more geometrical way. Chapter 5 paves the way for this research direction. Indeed, starting from one activation pattern, it calculates the distances to the borders of other activation patterns, which we hope to exploit in future work.
The second way to speed up verification is by approximate methods. Whether it is through an ensemble technique or through relaxation, as in abstract interpretation, it is one way in which verification methods can be scaled up to more complex models. However, there is still a lot of open research in bounding the approximation gap between approximate and exact methods. While some efforts have been made to estimate it (SYZ^{+}19), there is more research to be done in finding guarantees to achieve a certain fraction of the optimal solution with an approximation algorithm. Moreover, similar to our approach in Chapter 5 of training for easier verification, one might train for neural networks exhibiting a small approximation gap.
Another natural extension of our thesis consists in applying approximation algorithms to autoencoders. While we see some hurdles to applying, for example, abstract interpretation directly to approximate the worst-case-error, it could potentially help in approximating the output deviation for ε-δ robustness and, in this way, determine the denoising capability of a given autoencoder. Moreover, it could be used to check whether separate input sets remain separate in the latent space, thus allowing for a lower dimensional representation on which a downstream task can safely be executed.
Although we have started creating new verification problems for unsupervised neural networks, there are likely many more to be discovered. In particular, we

search for more low-level specifications aiming at a particular task of the neural network, such as ε-δ robustness for denoising. But building on top of that, we also need to reconcile these low-level specifications with high-level demands of a particular user. He should be able to state his request in a non-technical domain-specific language. For example, in autonomous driving, we have no way of covering all possible perturbations of a stop sign in an image. Therefore, we cannot verify the high-level demand of safely identifying a stop sign with just a camera. This has severe implications not only for the safety of autonomous cars but also for ethics and responsibility in case of a fault showing that safe machine learning systems concern a wide range of research directions.
Just as supervised neural networks were a fruitful inspiration for our research, we expect that our contributions to unsupervised neural networks will, in turn, benefit neural network verification in general. In fact, both the *fctdist* regularizer as well as the ensemble idea can likely be applied to supervised neural networks as well. They are mainly based on the two changes of perspective, including the training process as well as the neural network's structure in the verification process. Both these ideas are not exclusive to unsupervised neural networks.
Also, the challenge of post-robustification is still open for general neural networks and may well be solved by cleverly adjusting some neural networks' internals, such as their weights or biases. Moreover, one might be able to incorporate knowledge obtained from the verification process into the post-robustification process in a counterexample-guided inductive synthesis manner (STB+06). Then, by successively refining an existing model, we may eventually obtain safe and reliable neural networks.

Bibliography

[ACW18] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning - ICML*, 2018.

[AEIK18] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *International Conference on Machine Learning - ICML*, 2018.

[AGO+13] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. Training computationally efficient smartphone–based human activity recognition models. *Artificial Neural Networks and Machine Learning – ICANN*, 2013.

[Atr21] Maurice Atrops. Explainable anomaly descriptions for autoencoders. 2021. Unpublished Masterthesis - University of Bonn.

[BCM+13] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Srndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *European Conference on Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, 2013.

[Ben20] Benedikt Böing and Rajarshi Roy and Emmanuel Müller and Daniel Neider. Quality guarantees for autoencoders via unsupervised adversarial attacks. In *European Conference on Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, 2020.

[Ben22a] Benedikt Böing and Emmanuel Müller. On training and verifying robust autoencoders. In *IEEE International Conference on Data Science and Advanced Analytics - DSAA*, 2022.

[Ben22b] Benedikt Böing and Falk Howar and Jelle Hüntelmann and Emmanuel Müller and Richard Stewing. Neural network verification with DSE. In *Workshop on Artificial Intelligence and Formal Verification,* *Logic, Automata, and Synthesis hosted by the International Conference of the Italian Association for Artificial Intelligence - AIxIA*, 2022.

[Ben22c] Benedikt Böing and Simon Klüttermann and Emmanuel Müller. Post-robustifying deep anomaly detection ensembles by model selection. In *IEEE International Conference on Data Mining - ICDM*, 2022.

[BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[BIL$^+$16] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems - NeurIPS*, 2016.

[Bis07] Christopher M. Bishop. *Pattern Recognition and Machine Learning, 5th Edition*. Springer, 2007.

[BLZ$^+$21a] Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. Recent advances in adversarial training for adversarial robustness. In *International Joint Conference on Artificial Intelligence - IJCAI*, 2021.

[BLZ$^+$21b] Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. Recent advances in adversarial training for adversarial robustness. In *International Joint Conference on Artificial Intelligence - IJCAI*, 2021.

[BM07] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 2001.

[BSBD21] Lucas Costa Brito, Gian Antonio Susto, Jorge Nei Brito, and Marcus Antonio Viana Duarte. Fault detection of bearing: An unsupervised machine learning approach exploiting feature extraction and dimensionality reduction. *Informatics*, 2021.

[CAH19] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. Provable robustness of relu networks via maximization of linear regions. In *International Conference on Artificial Intelligence and Statistics - AISTATS*, 2019.

[CGG19] Shlomo E. Chazan, Sharon Gannot, and Jacob Goldberger. Deep clustering based on A mixture of autoencoders. In *International Workshop on Machine Learning for Signal Processing*, 2019.

[CKH$^+$15] Yanping Chen, Eamonn Keogh, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, and Gustavo Batista. The ucr time series classification archive, 2015. `www.cs.ucr.edu/~eamonn/time_series_data/`.

[CSAT17] Jinghui Chen, Saket Sathe, Charu C. Aggarwal, and Deepak S. Turaga. Outlier detection with autoencoder ensembles. In *International Conference on Data Mining - ICDM*, 2017.

[CW17a] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Workshop on Artificial Intelligence and Security*, 2017.

[CW17b] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Symposium on Security and Privacy - SP*, 2017.

[DDM$^+$04] Nilesh N. Dalvi, Pedro M. Domingos, Mausam, Sumit K. Sanghai, and Deepak Verma. Adversarial classification. In *International Conference on Knowledge Discovery and Data Mining - SIGKDD*, 2004.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification - CAV*, 2006.

[DDS$^+$09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition - CVPR*, 2009.

[Den12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 2012.

[DJST18] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods - 10th International Symposium - NFM*, 2018.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.

[DLP$^+$18] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Conference on Computer Vision and Pattern Recognition - CVPR*, 2018.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS*, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 1960.

[DXX18] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing - ICASSP*, 2018.

[Ehl17] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis - ATVA*, 2017.

[Ell22] Daniel Ellefred. Exhaustive description of non-anomalous data found by an autoencoder. 2022. Unpublished Bachelorthesis - Technical University of Dortmund.

[ERR$^+$19] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 2019.

[Faw06] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 2006.

[FCDX20] Yan Feng, Bin Chen, Tao Dai, and Shu-Tao Xia. Adversarial attack on deep product quantization network for image retrieval. In *Conference on Artificial Intelligence - AAAI*, 2020.

[FGD$^+$22] Tharindu Fernando, Harshala Gammulle, Simon Denman, Sridha Sridharan, and Clinton Fookes. Deep learning for medical anomaly detection - A survey. *ACM Computing Surveys*, 2022.

[FMJV22] Claudio Ferrari, Mark Niklas Müller, Nikola Jovanovic, and Martin T. Vechev. Complete verification via multi-neuron relaxation guided branch-and-bound. In *International Conference on Learning Representations - ICLR*, 2022.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2016. `http://www.deeplearningbook.org`.

[GMD^{+}18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy*, 2018.

[Gon16] Lovedeep Gondara. Medical image denoising using convolutional denoising autoencoders. In *IEEE International Conference on Data Mining Workshops*, 2016.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations - ICLR*, 2015.

[GWB^{+}03] Gongde Guo, Hui Wang, David A. Bell, Yaxin Bi, and Kieran Greer. KNN model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM*, 2003.

[HHWB02] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery - DaWaK*, 2002.

[HKR^{+}20a] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 2020.

[HKR^{+}20b] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Compututer Science Review*, 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[HS06] Geoffrey E. Hinton and Ruslan Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

[HSW89] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[HY01] David J Hand and Keming Yu. Idiot's bayes—not so stupid after all? *International Statistical Review*, 2001.

[JEP$^+$21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations - ICLR*, 2015.

[KBD$^+$17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification - CAV*, 2017.

[KEE$^+$15] Konstantina Kourou, Themis P. Exarchos, Konstantinos P. Exarchos, Michalis V. Karamouzis, and Dimitrios I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 2015.

[KH91] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems - NeurIPS*, 1991.

[KHI$^+$19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification - CAV*, 2019.

[KM22] Simon Klüttermann and Emmanuel Müller. Dean: Deep ensemble anomaly detection. 2022. https://github.com/psorus/DEAN.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems - NeurIPS*, 2012.

[KVKV11] Bernhard H Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial Optimization*. Springer, 2011.

[LBGM18] Van-Khoa Le, Pierre Beauseroy, and Edith Grall-Maes. Abnormal trajectory detection for security infrastructure. In *International Conference on Digital Signal Processing - DSP*, 2018.

[LLY+21] Yu Liang, Siguang Li, Chungang Yan, Maozhen Li, and Changjun Jiang. Explaining the black-box model: A survey of local interpretation methods for deep neural networks. *Neurocomputing*, 2021.

[LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems - NeurIPS*, 2017.

[LRM+12] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning - ICML*, 2012.

[LZX+18] Ying Li, Haokui Zhang, Xizhe Xue, Yenan Jiang, and Qiang Shen. Deep learning for remote sensing image classification: A survey. *WIREs Data Mining Knowledge Discovery*, 2018.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[MBB+23] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. The third international verification of neural networks competition (vnn-comp 2022): Summary and results, 2023.

[MCSK17] Qinxue Meng, Daniel R. Catchpoole, David Skillicom, and Paul J. Kennedy. Relational autoencoder for feature extraction. In *International Joint Conference on Neural Networks - IJCNN*, 2017.

[MFF16] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *Conference on Computer Vision and Pattern Recognition - CVPR*, 2016.

[MH21] Malte Mues and Falk Howar. Data-driven design and evaluation of SMT meta-solving strategies: Balancing performance, accuracy, and cost. In *36th IEEE/ACM International Conference on Automated Software Engineering - ASE*, 2021.

[MMS+18] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations - ICLR*, 2018.

[MMS+22] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proceedings of the ACM on Programming Languages - POPL*, 2022.

[MRB+13] Kamel Mansouri, Tine Ringsted, Davide Ballabio, Roberto Todeschini, and Viviana Consonni. Quantitative structure–activity relationship models for ready biodegradability of chemicals. *Journal of Chemical Information and Modeling*, 2013.

[MSY+09] Martin Renqiang Min, David A. Stanley, Zineng Yuan, Anthony J. Bonner, and Zhaolei Zhang. A deep non-linear feature mapping for large-margin knn classification. In *International Conference on Data Mining - ICDM*, 2009.

[NBL+21] Naji Najari, Samuel Berlemont, Grégoire Lefebvre, Stefan Duffner, and Christophe Garcia. Radon: Robust autoencoder for unsupervised anomaly detection. In *International Conference on Security of Information and Networks - SIN*, 2021.

[OPM+19] Michael P Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *Digital Avionics Systems Conference - DASC*, 2019.

[O'S06] Mícheál O'Searcoid. *Metric spaces*. Springer Science & Business Media, 2006.

[OSF19] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, 2019.

[oSoC10] Semeion Research Center of Sciences of Communication. Steel plates faults data set, 2010. http://archive.ics.uci.edu/ml/datasets/steel+plates+faults.

[PMG+17] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Asia Conference on Computer and Communications Security - AsiaCCS*, 2017.

[PMJ+16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *European Symposium on Security and Privacy - EuroS&P*, 2016.

[PS14] Luca Pasa and Alessandro Sperduti. Pre-training of recurrent neural networks via linear autoencoders. In *Advances in Neural Information Processing Systems - NeurIPS*, 2014.

[QWZW14] Yu Qi, Yueming Wang, Xiaoxiang Zheng, and Zhaohui Wu. Robust feature learning by stacked autoencoder with maximum correntropy criterion. In *International Conference on Acoustics, Speech and Signal Processing - ICASSP*, 2014.

[Ray16] Shebuti Rayana. Odds library, 2016. http://odds.cs.stonybrook.edu.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 1986.

[Rie20] Karl Rieländer. Deriving error landscapes from autoencoders in the application of anomaly detection. 2020. Unpublished Bachelorthesis - University of Bonn.

[RPG+21] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning - ICML*, 2021.

[RVG+18] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. In *International Conference on Machine Learning - ICML*, 2018.

[RWS+19] Wenjie Ruan, Min Wu, Youcheng Sun, Xiaowei Huang, Daniel Kroening, and Marta Kwiatkowska. Global robustness evaluation of deep neural networks with provable guarantees for the hamming distance. In *International Joint Conference on Artificial Intelligence - IJCAI*, 2019.

[SCZZ20] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE Transactions on Cybernetics*, 2020.

[SGM+18] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems - NeurIPS*, 2018.

[SGPV19] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations - ICLR*, 2019.

[SHK+14] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[SK01] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *International Conference on Knowledge Discovery and Data Mining - SIGKDD*, 2001.

[SLW+20] Prissadang Suta, Xi Lan, Biting Wu, Pornchai Mongkolnam, and Jonathan H Chan. An overview of machine learning in chatbots. *International Journal of Mechanical Engineering and Robotics Research*, 2020.

[SSL+18] Uri Shaham, Kelly P. Stanton, Henry Li, Ronen Basri, Boaz Nadler, and Yuval Kluger. Spectralnet: Spectral clustering using deep neural networks. In *International Conference on Learning Representations - ICLR*, 2018.

[STB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2006.

[SVS19] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Compututation*, 2019.

[SY14] Mayu Sakurada and Takehisa Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Workshop on Machine Learning for Sensory Data Analysis - MLSDA*, 2014.

[SYZ+19] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A convex relaxation barrier to tight robustness verification of neural networks. In *Neural Information Processing Systems - NeurIPS*, 2019.

[SZS+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations - ICLR*, 2014.

[TB19] Florian Tramer and Dan Boneh. Adversarial training and robustness for multiple perturbations. 2019.

[TXT19] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations - ICLR*, 2019.

[VLL+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 2010.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems - NeurIPS*, 2017.

[Vu23] Minh Bang Vu. Adversarial training for unsupervised neural networks. 2023. Unpublished Masterthesis - Technical University of Dortmund.

[VVA⁺12] Alexander Vergara, Shankar Vembu, Tuba Ayhan, Margaret A. Ryan, Margie L. Homer, and Ramón Huerta. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical*, 2012.

[VvRBT13] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 2013.

[WOZ⁺20] Haoze Wu, Alex Ozdemir, Aleksandar Zeljic, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina S. Pasareanu, and Clark W. Barrett. Parallelization techniques for verifying neural networks. In *Formal Methods in Computer Aided Design - FMCAD*, 2020.

[WW21] Tung-Yu Wu and Youting Wang. Locally interpretable one-class anomaly detection for credit card fraud detection. In *International Conference on Technologies and Applications of Artificial Intelligence - TAAI*, 2021.

[WZC⁺18] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning - ICML*, 2018.

[XEQ18] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Network and Distributed System Security Symposium - NDSS*, 2018.

[XKN22] Xuan Xie, Kristian Kersting, and Daniel Neider. Neuro-symbolic verification of deep neural networks. In *International Joint Conference on Artificial Intelligence - IJCAI*, 2022.

[XTJ18] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE Transactions on NEural Networks and Learning Systems*, 2018.

[XTSM19] Kai Yuanqing Xiao, Vincent Tjeng, Nur Muhammad (Mahi) Shafiullah, and Aleksander Madry. Training for faster adversarial robustness verification via inducing relu stability. In *International Conference on Learning Representations - ICLR*, 2019.

[XWZ+17] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. 2017.

[YJY19] XW Ye, T Jin, and CB Yun. A review on deep learning-based structural health monitoring of civil infrastructures. *Smart Struct Syst*, 2019.

[ZBW+18] Zhongheng Zhang, Marcus W Beck, David A Winkler, Bin Huang, Wilbert Sibanda, Hemant Goyal, et al. Opening the black box of neural networks: methods for interpreting neural network models in clinical applications. *Annals of Translational Medicine*, 2018.

[ZCH19] Huan Zhang, Minhao Cheng, and Cho-Jui Hsieh. RobBoost: A provable approach to boost the robustness of deep model ensemble. *Safe Machine Learning workshop at International Conference of Learning Representations - ICLR*, 2019.

[ZP17] Chong Zhou and Randy C. Paffenroth. Anomaly detection with robust deep autoencoders. In *International Conference on Knowledge Discovery and Data Mining - KDD*, 2017.

[ZSLG16] Stephan Zheng, Yang Song, Thomas Leung, and Ian Goodfellow. Improving the robustness of deep neural networks via stability training. In *IEEE Conference on Computer Vision and Pattern Recognition - CVPR*, 2016.

[ZWC+18] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in Neural Information Processing Systems - NeurIPS*, 2018.

[ZZH19] Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. Recurjac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications. In *Conference on Artificial Intelligence - AAAI*, 2019.