Towards Robust Artificial-Intelligence-Powered Software:
Provable Guarantees via Abstract Interpretation

by

Yuhao Zhang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 04/30/2024

The dissertation is approved by the following members of the Final Oral Committee:
Aws Albarghouthi, Associate Professor, Computer Sciences
Loris D'Antoni, Associate Professor, Computer Sciences
Fredric Sala, Assistant Professor, Computer Sciences
Chaowei Xiao, Assistant Professor, The Information School

To the humble intelligence.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my parents for their unwavering support throughout my PhD journey. Due to visa issues and the COVID-19 pandemic, I have not been able to visit home since I started my PhD in August 2019. I am sincerely grateful for their understanding and continuous communication with my grandparents and other relatives to accompany them during my absence.

I want to thank my advisors, Aws Albarghouthi and Loris D'Antoni, for their research guidance and mental support. Along with my advisors, I would also like to thank the rest of my committee, Fredric Sala and Chaowei Xiao, for their time and consideration. I am deeply grateful to all my excellent collaborators, without whom I would not have been able to earn my PhD.

I want to express my appreciation to those who have helped me during my academic job search. Thank my recommendation letter writers, Baishakhi Ray, Tao Xie, and my advisors. I also want to thank Professors Thomas Reps, Ethan Cecchetti, Tej Chajed, Yingfei Xiong, Rahul Chatterjee, Josiah Hanna, and Yong Jae Lee, who polished my job talk and other materials and conducted mock interviews with me. I am grateful to Charlie Murphy, Hejie Cui, Kanghee Park, Ling Zhang, Zi Wang, and other people who gave invaluable suggestions to improve my job talk. I want to thank Chaowei Xiao, Daniel Seita, Di Wang, Lijun Ding, Linyi Li, Jialu Zhang, Kexin Pei, Tianhang Zheng, Yue Zhao, and other people who were on the job market in previous years and shared their experiences. I also want to thank Adithya Murali, Bian Song, Binbin Xie, Dinghuai Zhang, Guangliang Liu, Guanhong Tao, Guannan Wei, Haitao Mao, Honghui Xu, Jacob Laurel, Jiarong Xing, Jing Liu, Jocelyn (Qiaochu) Chen, Kaiyuan Zhang, Ke Wu, Lauren Pick, Nan Jiang, Peiyan Dong, Pengfei Chong, Siwei Cui, Songlin Jia, Xufeng Cai, Yang Shi, Yangruibo (Robin) Ding, Yu Zhang, Yue Wu, Yuke Wang, Yuxiang Peng, Yuxin Sun, Zhe Zhou, Zhengzhong Tu, Zhiqing Sun, Zhun Deng, Zhuohan Li for sharing information with me. Most of them were also on the job market with me at the same time. I am grateful that we could share the anxiety of being on the job market with each other.

I am grateful to be part of a vigorous research group, MadPL[1]. I want to thank the graduate students who joined the group before me: Calvin Smith, Jinman Zhao, John Cyphert, Jordan Henkel, Qingheping Hu, Sam Drews, Zhicheng Cai, and Zi Wang.

[1] `https://madpl.cs.wisc.edu`

(a) Boston (b) My drawing of Boston

Figure 0.1: Pictures of Boston.

They welcomed me into the group as a new student. I would also like to thank my peers, Jialu Bao, Jinwoo Kim, and Zachary Susag, with whom I shared the experience of being impacted by the COVID-19 pandemic during our first year. I also thank the graduate students who joined the PL group after me and with whom I had the chance to interact: Abtin Molavi, Aisha Mohamed, Amanda Xu, Anna Meyer, Anvay Grover, Kanghee Park, Keith Johnson, Rahul Krishnan, Shaan Nagy, and Wiley Corning.

My two past internship experiences were terrific and greatly helped me find jobs in the industry, even though the number of available positions was limited. In the summer of 2021, I interned at the PROSE team at Microsoft, where I worked on the Blue-Pencil project, a key initiative behind Visual Studio IntelliCode. I thank my mentors, Arjun Radhakrishna and Gustavo Soares, for their guidance and support. In the summer of 2023, I interned at CodeWhisperer at Amazon AWS, where I enhanced the robustness of large language models for code generation. I want to thank my mentor, Shiqi Wang, and manager, Haifeng Qian, for their guidance and support.

Once again, I would like to thank all the people who accompanied me during my PhD journey and Boston (Figure 0.1), who showed up every time in my job talks and PhD defense.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Deep learning has rapidly emerged as a transformative technology that permeates all modern software, from autonomous driving systems to medical-diagnosis and malware-detection tools. Considering the critical role of this software in our technologies, it must behave as intended. The complexity introduced by deep-learning components complicates formal reasoning about the behavior of such software, frequently resulting in solutions that offer only empirical or no guarantees.

This thesis contributes techniques and algorithms that increase the robustness of deep-learning-powered software by providing strong *provable guarantees* across the components existing in the *entire* deep learning pipeline. By leveraging the power of *abstract interpretation*, a well-established theory for program analysis and verification, this thesis enables the verification of robustness properties across the deep learning pipeline. The thesis focuses on four critical aspects of robustness: (1) preventing numerical bugs in deep learning code implementations, (2) verifying the robustness of language models against adversarial perturbations, and (3) developing certifiable defenses against backdoor attacks during model training. By addressing these challenges and providing provable guarantees, this research aims to enhance the reliability and trustworthiness of deep-learning-powered software in critical applications.

1 INTRODUCTION

Artificial intelligence (AI) has witnessed remarkable progress over the past few decades, with deep learning emerging as a driving force behind this advancement. The journey began with the development of convolutional neural networks (CNNs), which revolutionized image recognition tasks by automatically learning hierarchical features from raw pixel data. A landmark moment in this journey was the introduction of AlexNet (Krizhevsky et al., 2012), a deep CNN that achieved unprecedented performance on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. The availability of large-scale datasets like ImageNet played a crucial role in the success of CNNs by providing vast amounts of labeled data for training. CNNs enabled computers to recognize objects, faces, and scenes with impressive accuracy, paving the way for applications like self-driving cars and medical image analysis.

As the focus shifted to sequential data such as text and speech, long short-term memory (LSTM) networks emerged, introducing the ability to remember and forget information over long sequences selectively. However, the real breakthrough in natural language processing arrived with the start of the Transformer architecture, which relies solely on attention mechanisms. This innovation laid the foundation for the development of large language models (LLMs), particularly decoder-only transformers like the Generative Pre-trained Transformer (GPT) models (Brown et al., 2020). These models leverage the power of attention to capture long-range dependencies and generate human-like text, marking a "Renaissance" of the sequential structure. The success of these models can be attributed to pre-training tasks, such as masked language modeling (MLM) and next token prediction, which enable models to learn rich representations of language from massive amounts of unlabeled data available on the web. By pre-training on vast quantities of diverse text data, these models acquire a broad understanding of language that can be fine-tuned for specific downstream tasks with minimal additional training, revolutionizing the field of natural language processing.

Alongside these advancements in deep learning, the field of program analysis has also made significant strides. One of the most influential techniques in this domain is abstract interpretation (Cousot and Cousot, 1977b), a theory introduced by Patrick Cousot and Radhia Cousot in the late 1970s. Abstract interpretation provides a framework for reasoning about the behavior of programs without actually executing them.

It involves creating abstract versions of the program's semantics that capture essential properties while ignoring irrelevant details. Analyzing these abstract semantics makes it possible to prove the program's properties, such as the absence of certain errors or the adherence to specific security policies. As artificial intelligence becomes increasingly integrated into software systems, ensuring the safety and reliability of these deep-learning-powered components becomes crucial. This thesis explores how abstract interpretation can be applied to verify the safety of deep-learning-powered software systems, addressing the critical need for trustworthy AI solutions.

Deep learning has rapidly emerged as a transformative technology that permeates all modern software, from autonomous driving systems (Huang and Chen, 2020) to medical-diagnosis (Azad et al., 2022; Singhal et al., 2022) and malware-detection (Raff et al., 2018) tools. As the adoption of deep learning becomes increasingly widespread in critical software, the need to ensure this deep-learning-powered behaves as intended has become paramount.

While ensuring that deep-learning-powered software behaves as intended encompasses a wide range of challenges, this thesis focuses specifically on the robustness of such software. *Robustness, in this context, extends beyond the traditional notion of model robustness, which primarily focuses on adversarial examples.* The scope of this research includes the robustness of the code implementation, training processes, and post-deployment functionalities, such as counterfactual explanations. By addressing robustness holistically across all these critical components, this thesis aims to provide *comprehensive* approaches to ensuring the reliable behavior of deep-learning-powered software.

Existing techniques for increasing the robustness of deep-learning-powered software, such as software testing and empirical defenses for adversarial evasion and data-poisoning attacks, often provide limited or no formal guarantees. While these methods can help identify and mitigate specific vulnerabilities, they cannot comprehensively address all potential issues or provide guarantees of robustness. In contrast, this thesis contributes techniques and algorithms that increase the robustness of deep-learning-powered software by providing strong *provable guarantees*. These techniques leverage the power of *abstract interpretation*, a well-established theory for program analysis and verification. By employing abstract interpretation, this research enables the verification of robustness properties across the deep-learning-powered software, thus providing strong provable guarantees.

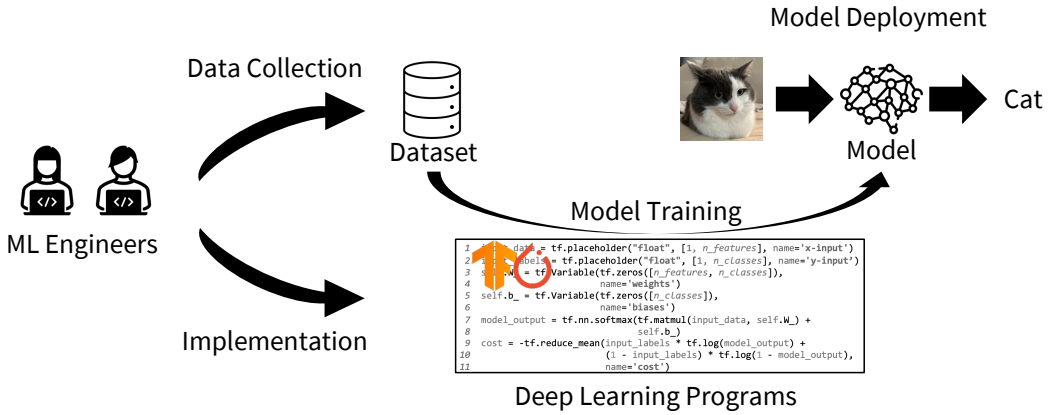
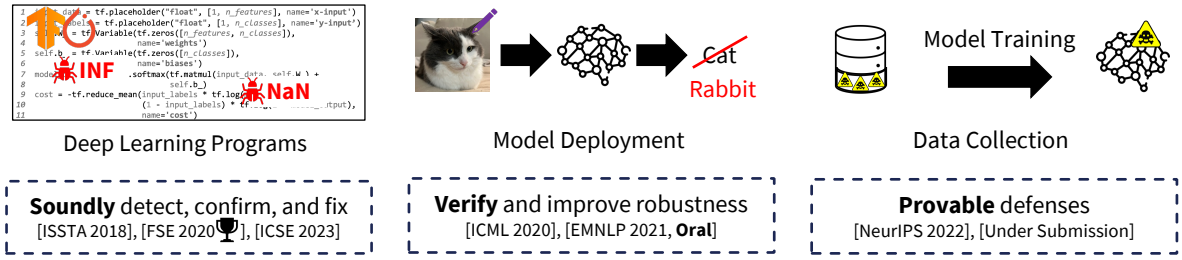


Figure 1.1: Overview of deep-learning-powered software.



(a) Numerical Bugs (b) Adversarial Examples (c) Data Poisoning Attacks

Figure 1.2: Robustness issues in deep-learning-powered software.

Deep-learning-powered software consists of the following steps (see Figure 1.1). First, machine learning (ML) engineers implement deep learning code using platforms like TensorFlow and PyTorch to construct neural network architectures and perform training and inference. Second, ML engineers collect data from the web to construct large datasets and train the models. Third, the trained models are deployed in real-world applications, making predictions, such as classifying objects in images. Ensuring robustness throughout this pipeline is crucial to guarantee that deep-learning-powered software behaves as intended and maintains its reliability in real-world applications.

As shown in Figure 1.2, robustness issues can manifest in various components of deep-learning-powered software. First, the code implementation built on top of deep learning platforms can contain bugs. One type of bug is numerical bugs, which generate values like `NaN` and `Inf` (Figure 1.2a). These numerical bugs are challenging to detect due to complex component interactions and their difficulty in being spotted during code reviews. Chapter 2 focuses on preventing numerical bugs in deep learning

programs to identify and rectify implementation vulnerabilities. Second, deep learning models are vulnerable to adversarial examples, carefully crafted inputs designed to fool the model into making incorrect predictions while appearing benign to humans (Figure 1.2b). Chapter 3 focuses on verifying the robustness of language models to ensure their resilience against adversarial perturbations. Third, backdoor attacks involve an adversary injecting malicious patterns into the training data, causing the model to learn a hidden trigger that can be exploited at inference time (Figure 1.2c). Chapter 4 focuses on developing certifiable defenses against backdoor attacks to protect the integrity of the training process and the resulting models.

1.1 Preventing Numerical Bugs in Deep Learning Programs

Prior empirical studies (Zhang et al., 2018b; Humbatova et al., 2020) on bugs in deep learning systems have demonstrated that these bugs are not limited to the deep learning models themselves; they also frequently appear in the deep learning programs built on top of deep learning platforms like TensorFlow (Abadi et al., 2016) or PyTorch (Paszke et al., 2019). In particular, they are often found in the programs specifying the corresponding neural network architectures.

To avoid unexpected or incorrect behaviors in deep learning software systems, it is necessary to detect bugs in their neural architectures. Although various approaches (Pei et al., 2017; Tian et al., 2018) have been proposed to test or verify deep learning models, these approaches do not address the needs of two types of stakeholders: (1) architecture vendors who design and publish neural architectures to be used by other users, and (2) developers who use neural architectures to train and deploy a model based on the developers' own training dataset.

- Architecture vendors need to provide quality assurance for their neural architecture. It is inadequate for the vendors to verify the architecture with specific instantiated models, which are dataset-dependent.

- Bugs in a neural architecture may manifest themselves into failures after developers have trained a model for hours, days, or even weeks, causing great loss in

time and computation resources (Zhang et al., 2018b). The loss can be prevented if these bugs can be detected early at the architecture level before model training.

- Failures can also occur when developers of a deep learning model need to retrain their models upon updates on training data. These updates can frequently happen during software system development and deployment, e.g., when the new feedback data is collected from users (Zhang et al., 2019b).
- Failures in deep learning models can be caused by a bug in the deep learning architecture, low-quality training data, incorrect parameter settings, or other issues. It is not easy for the developers to localize the bug.

In Chapter 2, we present DEBAR (Zhang et al., 2020d), the first attempt at conducting static analysis for bug detection at the architecture level, specifically targeting numerical bugs, an important category of bugs known to have catastrophic consequences. These numerical bugs are challenging to detect due to complex component interactions and their difficulty to be spotted during code reviews. To assure high reliability against numerical defects, in addition to bug detection, we propose the RANUM (Li et al., 2023b) framework, which includes novel techniques, in addition to bug detection, for two reliability assurance tasks: confirmation of potential-defect feasibility and suggestion of defect fixes. RANUM is the first framework that confirms potential-defect feasibility with failure-exhibiting tests and automatically suggests fixes.

1.2 Verifying the Robustness of NLP Models

Despite the remarkable performance of deep learning models in various tasks, deep learning models are known to be vulnerable to adversarial examples (Carlini and Wagner, 2017), which are carefully crafted inputs designed to mislead the model into producing incorrect outputs. The existence of adversarial examples exposes deep learning models to potential security threats, as malicious attackers can exploit these weaknesses to compromise the system's functionality.

Addressing the robustness problem in deep learning models involves developing methods to mitigate adversarial examples. Researchers have proposed various empirical techniques to achieve this goal, including adversarial training (Madry et al., 2018)

and defensive distillation (Papernot et al., 2016), as well as certified approaches (Gowal et al., 2019) that come with provable robustness guarantees. By providing rigorous proof of robustness, certified approaches can give users greater confidence in the trustworthiness of the model and reduce the risk of catastrophic failures.

Adversarial examples in NLP come from *discrete* perturbation spaces, which differ from continuous perturbation spaces in CV, where l_p norm bounds are widely used to define the pixel-level perturbations. In NLP, however, the focus is on word- or character-level string transformations, such as adding or deleting words, changing word order, and substituting synonyms or their combinations. Existing certified approaches for NLP either target l_p norm bounds around word embeddings (Ko et al., 2019) or only handle simple string transformations, such as word deletion (Welbl et al., 2020) and synonym substitution (Huang et al., 2019; Jia et al., 2019). These approaches cannot easily handle real-world adversarial examples, which often involve complex string transformations. Therefore, developing robust certified approaches that can handle a wider range of string transformations is an important research direction for improving the robustness of NLP models against adversarial attacks.

Chapter 3 presents our work on verifying the robustness of NLP models. In Zhang et al. (2020c), we introduce a general language that allows us to specify a perturbation space programmatically in a way that can be easily decomposed and verified. We also propose augmented abstract adversarial training (A3T), which combines the strengths of augmentation and abstraction techniques to improve robustness. In Zhang et al. (2021b), we present ARC (Abstract Recursive Certification), an approach for certifying the robustness of recursive neural networks, such as LSTMs and TreeLSTMs, to programmatically defined perturbation spaces. We also present a preliminary work on applying ARC to autoregressive transformers.

1.3 Certifiable Defense against Backdoor Attacks

Deep learning models are vulnerable to backdoor poisoning attacks, where attackers can poison a small fraction of the training set before model training and add triggers to inputs at test time. As a result, the prediction of the victim model trained on the poisoned dataset will diverge in the presence of a trigger in the test input. Backdoor attacks have been successfully demonstrated in various domains, including image recognition (Gu et al., 2017), sentiment analysis (Qi et al., 2021a), and malware de-

tection (Severi et al., 2021). These attacks pose significant security concerns for deep learning models and systems trained on data gathered from different sources, such as via web scraping.

Existing defenses against backdoor attacks have two main limitations. First, many approaches only provide empirical defenses specific to certain attacks and do not generalize to all backdoor attacks. Second, existing certified defenses either cannot handle backdoor attacks or are probabilistic, making them expensive and ineffective in practice.

Certification is crucial for defending against backdoor attacks, as it proves that the learned model can withstand such attacks. Empirical defenses lack certificates, can only defend against specific attacks, and can be bypassed by new, unaccounted-for attacks. Certification has been successful in building models that are provably robust to trigger-less poisoning attacks and evasion attacks, but these models remain vulnerable to backdoor attacks. *Determinism* is another desirable property for a certified defense, as probabilistic defenses typically require retraining thousands of models for a single test input prediction. This retraining can be mitigated by Bonferroni correction, but it is still necessary after a short period, making it challenging to deploy these defenses in practice. In contrast, deterministic defenses can reuse trained models an arbitrary number of times when producing certificates for different test inputs.

In Chapter 4, we present PECAN (**P**artitioning data and **E**nsembling of **C**ertified neur**A**l **N**etworks), a deterministic certified defense against backdoor attacks for neural networks. PECAN efficiently derives a backdoor-robustness guarantee by training a set of neural networks on disjoint partitions of the dataset and applying evasion certification to the neural networks.

Finally, Chapter 5 offers a conclusion, future directions, and final notes of this thesis.

To assure the reliability of DNN-based systems, it is highly critical to detect and fix numerical defects for two main reasons. First, numerical defects widely exist in DNN-based systems. For example, in the DeepStability database (Kloberdanz et al., 2022), over 250 defects are identified in deep learning algorithms where over 60% of them are numerical defects. Moreover, since numerical defects exist at the architecture level, any model using the architecture naturally inherits these defects. Second, numerical defects can result in serious consequences. Once numerical defects (such as divide-by-zero) are exposed, the faulty DNN model will output `NaN` or `INF` instead of producing any meaningful prediction, resulting in numerical failures and system crashes (Zhang et al., 2018b, 2019c). Thus, numerical defects hinder the application of DNNs in scenarios with high reliability and availability requirements such as threat monitoring in cybersecurity (Powell, 2022) and cloud system controlling (Sharma et al., 2016; Jay et al., 2019).

To address numerical defects in DNN architectures in an actionable manner (Xiong et al., 2022), we propose a workflow of reliability assurance, as illustrated in Figure 2.1, which consists of three essential tasks: potential-defect detection, feasibility confirmation, and fix suggestion. While DEBAR[1] is a static analysis tool that focuses on the first task of potential-defect detection, RANUM[2] supports all three tasks, providing a comprehensive solution to ensure the reliability of DNN architectures.

Potential-Defect Detection

In this task, we detect all potential numerical defects in a DNN architecture, with a focus on operators with numerical defects (in short as defective operators) that potentially exhibit inference-phase numerical failures for two main reasons, following the literature (Zhang et al., 2020d; Yan et al., 2021). First, these defective operators can be exposed after the model is deployed and thus are more devastating than those that potentially exhibit training-phase numerical failures (Odena et al., 2019; Zhang et al., 2020d). Second, a defective operator that potentially exhibits training-phase numerical failures can usually be triggered to exhibit inference-phase numerical failures, thus also being detected by our task. For example, the type of training-phase `NaN` gradient

[1] `https://github.com/ForeverZyh/DEBAR`
[2] `https://github.com/llylly/RANUM`

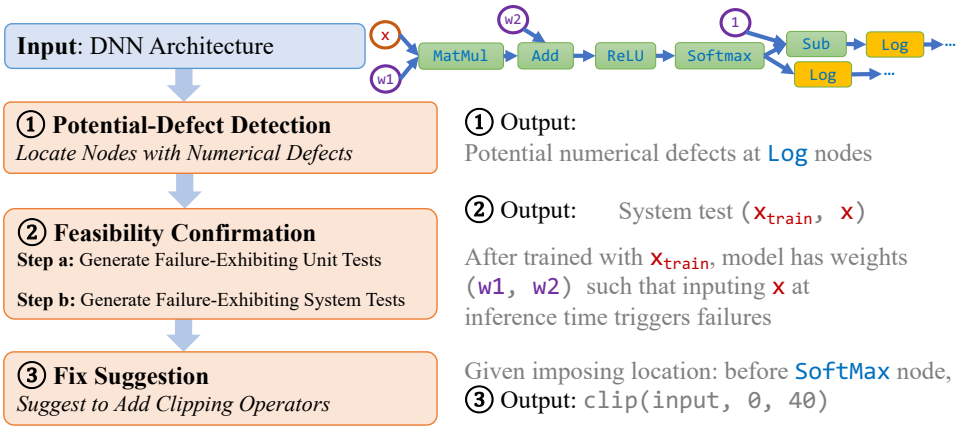


Figure 2.1: Workflow for reliability assurance against numerical defects in DNN architectures. The left-hand side shows three tasks and the right-hand side shows corresponding examples.

failures is caused by an operator's input that leads to invalid derivatives, and this input also triggers failures in the inference phase (Yan et al., 2021).

DEBAR was the first attempt to conduct static analysis for bug detection at the architecture level. To detect numerical bugs at the architecture level, we propose to use static analysis because static analysis is able to cover the large combinatorial space imposed by the numerous parameters and possible inputs of a neural architecture. We propose a static analysis approach for detecting numerical bugs in neural architectures based on abstract interpretation (Cousot and Cousot, 1977a), which mainly comprises two kinds of abstraction techniques, i.e., one for tensors and one for numerical values. We study three tensor abstraction techniques: array expansion, array smashing, and tensor partitioning, as well as two numerical abstraction techniques: interval abstraction and affine relation analysis. Among these techniques, array expansion, array smashing, and interval abstraction are adapted from existing abstraction techniques for imperative programs (Blanchet et al., 2003; Cousot and Cousot, 1977b). In addition, to achieve scalability while maintaining adequate precision, we propose *tensor partitioning* to partition tensors and infer numerical information over partitions, based on our insight: many elements of a tensor are subject to the same operators. In particular, representing (concrete) tensor elements in a partition as one abstract element under appropriate abstract interpretation can reduce analysis effort by orders of magnitude. Motivated by this insight, *tensor partitioning* initially abstracts all elements in a tensor as one abstract element and iteratively splits each abstract element into smaller ones when its

concrete elements go through different operators. Each abstract element represents one partition of the tensor, associated with a numerical interval that indicates the range of its concrete elements. Moreover, for the sake of precision, besides interval analysis, we conduct *affine relation analysis* to infer the elementwise affine equality relations among abstract elements representing partitions.

The RANUM framework also employs abstract interpretation for potential-defect detection, utilizing tensor partitioning and sole interval abstraction as its abstract domain. Unlike in DEBAR, we choose to use sole interval abstraction instead of affine equality relation analysis because it can be naturally differentiated, and interval gradients can be used in the succeeding tasks of feasibility confirmation and fix suggestion. Compared to DEBAR, RANUM offers improved capabilities, including support for dynamic graphs using the innovative technique of *backward fine-grained node labeling*. This allows for more flexibility and practicality as DEBAR can handle only static computational graphs and does not support the widely used dynamic graphs in PyTorch programs.

Feasibility Confirmation

In this task, we confirm the feasibility of these potential numerical defects by generating failure-exhibiting system tests. As shown in Figure 2.1, a system test is a tuple of training example[3] $\mathbf{x}_{\mathsf{train}}$ and inference example $\mathbf{x}$ such that after the training example is used to train the architecture under consideration, applying the resulting model on the inference example exhibits a numerical failure.

RANUM is the first approach that generates failure-exhibiting *system* tests that contain training examples. Doing so is a major step further from the existing GRIST (Yan et al., 2021) tool, which generates failure-exhibiting unit tests ignoring the practicality of generated model weights. Given that in practice model weights are determined by training examples, we propose the technique of *two-step generation* for this task. First, we generate a failure-exhibiting unit test. Second, we generate a training example that leads to the model weights in the unit test when used for training. For the second step, we extend the deep-leakage-from-gradient (DLG) attack (Zhu et al., 2019b) by incorporating the straight-through gradient estimator (Bengio et al., 2013).

[3]In real settings, multiple training examples are used to train an architecture, but generating a single training example to exhibit failures (targeted by our work) is desirable for ease of debugging while being more challenging than generating multiple training examples to exhibit failures.

Fix Suggestion

In this task, we fix a feasible numerical defect. To determine the fix form, we have inspected the developers' fixes of the numerical defects collected by Zhang et al. (2018b) by looking at follow-up Stack Overflow posts or GitHub commits. Among the 13 numerical defects whose fixes can be located, 12 fixes can be viewed as explicitly or implicitly imposing interval preconditions on different locations, such as after inputs or weights are loaded and before defective operators are invoked. Thus, imposing an interval precondition, e.g., by clipping (i.e., chopping off the input parts that exceed the specified input range) the input for defective operator(s), is an effective and common strategy for fixing a numerical defect. Given a location (i.e., one related to an operator, input, or weight where users prefer to impose a fix), we suggest a fix for the numerical defect under consideration.

RANUM is the first automatic approach based on the novel technique of *abstraction optimization*. We observe that a defect fix in practice is typically imposing interval clipping on some operators such that each later-executed operator (including those defective ones) can never exhibit numerical failures. Therefore, we propose the novel technique of abstraction optimization to "deviate away" the input range of a defective operator from the invalid range, falling in which can cause numerical failures.

2.1 Overview

In this section, we introduce the background of DNN numerical defects and failures, and then give an overview of DEBAR and RANUM with running examples.

Background

Deep learning developers typically use modern deep learning libraries like PyTorch and TensorFlow to define DNN architectures in code. These architectures can be represented as computational graphs, as shown in Figures 2.2 and 2.3. To facilitate analysis, the DNN architecture can be automatically converted to an ONNX-format computational graph (The Linux Foundation, 2022). However, it is important to note that DEBAR only supports static computational graphs from the TensorFlow library, while RANUM can analyze computational graphs from both PyTorch and TensorFlow libraries by performing static analysis over the ONNX-format computational graph.

```
input_data = tf.placeholder("float", [1, n_features], name='x-input')
input_labels = tf.placeholder("float", [1, n_classes], name='y-input')
self.W_ = tf.Variable(tf.zeros([n_features, n_classes]),
                      name='weights')
self.b_ = tf.Variable(tf.zeros([n_classes]),
                      name='biases')
model_output = tf.nn.softmax(tf.matmul(input_data, self.W_) +
                             self.b_)
cost = -tf.reduce_mean(input_labels * tf.log(model_output) +
                       (1 - input_labels) * tf.log(1 - model_output),
                       name='cost')
self.obj_function = tf.reduce_min(tf.abs(model_output),
                                  name='obj_function')
```

Figure 2.2: A deep learning program snippet that defines a linear regression model from benchmarks of real-world numerical defects.

The computational graph can be represented as a Directed Acyclic Graph (DAG): $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are sets of nodes and edges, respectively. Nodes with zero in-degree are called *initial nodes*, corresponding to input, weight, or constant nodes. Initial nodes provide concrete data for the DNN models that result from training the DNN architecture. The data from each node is formatted as a tensor, i.e., a multidimensional array, with a specified data type and array shape annotated alongside the node definition. Nodes with positive in-degree are called *internal nodes*, corresponding to concrete operators, such as matrix multiplication (`MatMul`) and addition (`Add`). During model training, the model weights, i.e., data from weight nodes, are updated by the training algorithm. Then, during the deployment phase (i.e., model inference), with these trained weights and a user-specified input, also known as an inference example, the output of each operator is computed in the topological order. The output of a specific node is used as the prediction result.

We let $\boldsymbol{x}$ and $\boldsymbol{w}$ denote the concatenation of data from all input nodes and data from all weight nodes, respectively. For example, in Figure 2.3, $\boldsymbol{x}$ concatenates data from nodes 1 and 11; and $\boldsymbol{w}$ concatenates data from nodes 2 and 4. Given specific $\boldsymbol{x}$ and $\boldsymbol{w}$, the input and output for each node are deterministic.[4] We use $f_n^{\text{in}}(\boldsymbol{x}; \boldsymbol{w})$ and $f_n^{\text{out}}(\boldsymbol{x}; \boldsymbol{w})$ to express input and output data of node n, respectively, given $\boldsymbol{x}$ and $\boldsymbol{w}$.

Numerical Defects in DNN Architecture We focus on inference-phase numerical defects. These defects lead to numerical failures when specific operators receive inputs

[4]An architecture may contain stochastic nodes. We view these nodes as nodes with randomly sampled data, so the architecture itself is deterministic.

Table 2.1: Supported defective operators that may contain numerical defects, along with their invalid ranges $\mathcal{I}_{n_0,\text{invalid}}$. In the table, U_{min} and U_{max} stand for the minimum and maximum positive number of the input tensor's data type, respectively. Operators marked with * are only supported in DEBAR and operators marked with $^+$ are newly supported in RANUM. Furthermore, RANUM restricts the invalid range of `Sqrt` from $(-\infty, -U_{\text{min}}]$ to $(-\infty, U_{\text{min}}]$ because the gradient of `Sqrt` can be invalid when the input range is in $[-U_{\text{min}}, U_{\text{min}}]$.

Op. Type	$\mathcal{I}_{n_0,\text{invalid}}$
`Pow`$^+$	$[-U_{\text{min}}, U_{\text{min}}] \times (-\infty, -U_{\text{min}}]$
`Div`	$\mathbb{R} \times [-U_{\text{min}}, U_{\text{min}}]$
`Reciprocal`	$[-U_{\text{min}}, U_{\text{min}}]$
`Sqrt`	$(-\infty, U_{\text{min}}]$
`Exp`, `Expm1`*	$[\ln U_{\text{max}}, \infty)$
`Log`	$(-\infty, U_{\text{min}}]$
`Log1p`*	$(-\infty, U_{\text{min}} - 1]$
`Range`$^+$	$\mathbb{R} \times \mathbb{R} \times [-U_{\text{min}}, U_{\text{min}}]$
`NegativeLogLikelihoodLoss`$^+$	$[0, 0]$ for non-zero cells using mean reduction

within invalid ranges so that the operators output `NaN` or `INF`.

Definition 2.1. *For the given computational graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, if there is a node $n_0 \in \mathcal{V}$, such that there exists a valid input and valid weights that can let the input of node n_0 fall within the invalid range, we say there is a* **numerical defect** *at node n_0.*

$$\begin{aligned} &\textit{Formally, } \exists \mathbf{x}_0 \in \mathcal{X}_{\text{valid}}, \boldsymbol{w}_0 \in \mathcal{W}_{\text{valid}}, f^{\text{in}}_{n_0}(\mathbf{x}_0; \boldsymbol{w}_0) \in \mathcal{I}_{n_0,\text{invalid}} \\ &\quad \Longrightarrow \exists \textit{ numerical defect at node } n_0. \end{aligned}$$

In the definition, $\mathcal{X}_{\text{valid}}$ and $\mathcal{W}_{\text{valid}}$ are valid input range and weight range, respectively, which are clear given the deployed scenario. For example, ImageNet `Resnet50` models have valid input range $\mathcal{X}_{\text{valid}} = [0, 1]^{3\times224\times224}$ since image pixel intensities are within $[0, 1]$, and valid weight range $\mathcal{W}_{\text{valid}} = [-1, 1]^p$ where p is the number of parameters since weights of well-trained `Resnet50` models are typically within $[-1, 1]$. The invalid range $\mathcal{I}_{n_0,\text{invalid}}$ is determined by n_0's operator type with detailed definitions in Table 2.1. For example, for the `Log` operator, the invalid range $\mathcal{I}_{n_0,\text{invalid}} = (-\infty, U_{\text{min}})$ where U_{min} is the smallest positive number of a tensor's data type.

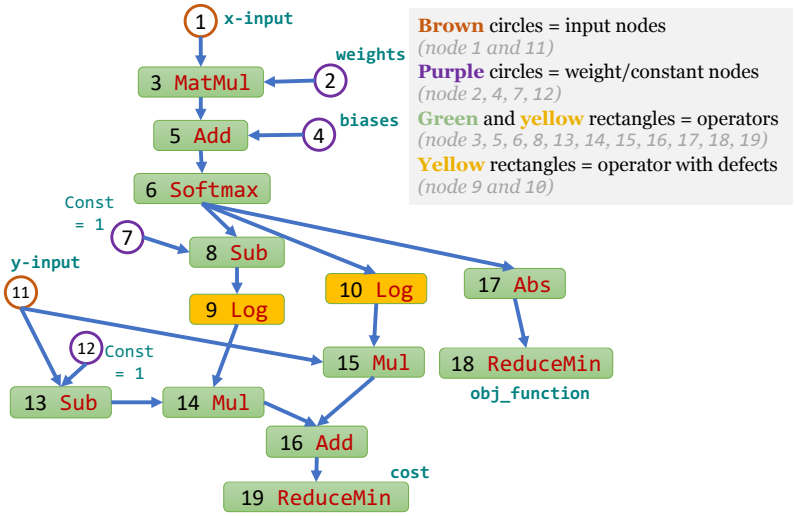


Figure 2.3: Computational graph encoded by the snippet in Figure 2.2.

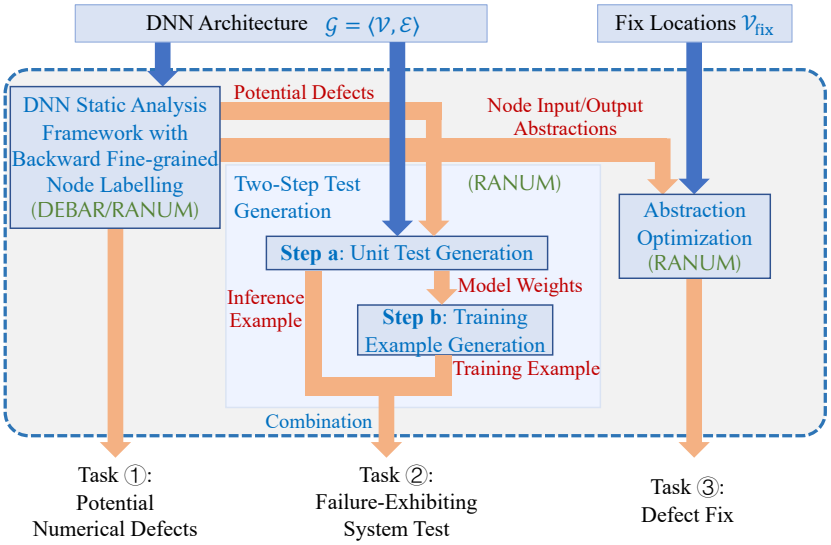


Figure 2.4: Overview of the reliability assurance framework approach. The output of RANUM indicates confirmation and manifestation of numerical defects (that can be feasibly exposed at the system level) for a given DNN architecture and effective fixes for the architecture's confirmed defects.

2.2 Motivating Examples

In Figure 2.4, we show the overview structure of the reliability assurance framework. The framework takes a DNN architecture expressed as a computational graph as the input. First, the DNN static analysis framework (task ① in Figure 2.1) detects all potential numerical defects in the architecture. Second, the two-step test generation

component (task ② in Figure 2.1), including unit test generation and training example generation, confirms the feasibility of these potential numerical defects. Third, the abstraction optimization component (task ③ in Figure 2.1) takes the abstractions produced by the DNN static analysis framework along with the user-specified fix locations, and produces preconditions to fix the confirmed defects.

Potential-Defect Detection via Static Analysis

The DNN static analysis framework first computes the numerical intervals of possible inputs and outputs for all nodes within the given DNN architecture, and then flags any nodes whose input intervals overlap with their invalid ranges as nodes with potential numerical defects.

In this section, we will first demonstrate how interval abstraction (Cousot and Cousot, 1977b) and tensor expansion (Blanchet et al., 2003) work on the running example in Figure 2.3. Next, we will use another example shown in Figure 2.5 to illustrate the differences among the four different abstract domains proposed in DEBAR: (a) sole interval abstraction and tensor expansion, (b) sole interval abstraction and tensor smashing (Blanchet et al., 2003), (c) sole interval abstraction and tensor partitioning (Gopan et al., 2005) used in RANUM, and (d) interval abstraction with affine equality relation analysis (Karr, 1976) and tensor partitioning.

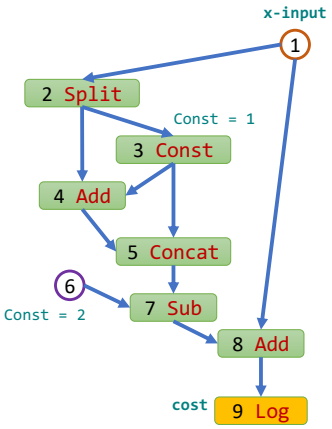
Running example in Figure 2.3 by sole interval abstraction and tensor expansion Interval abstraction is a widely used technique in abstract interpretation for abstracting numerical values. It involves representing each scalar variable as an interval with a lower and upper bound. These intervals are calculated by mapping standard operators to interval arithmetic. Tensor expansion, inspired by array expansion, maps each element in a tensor one-to-one to the abstract domain, without any further abstraction. When used in conjunction with interval abstraction, tensor expansion enables the direct mapping of tensor elements to their interval ranges.

In Figure 2.3, suppose that the user-specified input `x-input` (node 1) is within (elementwise, same below) range $[(-10,-10)^\intercal,(10,10)^\intercal]$; `weights` (node 2) are within range $[\begin{pmatrix}-10 & -10\\ -10 & -10\end{pmatrix},\begin{pmatrix}10 & 10\\ 10 & 10\end{pmatrix}]$; and `biases` (node 4) are within range $[(-10,-10)^\intercal,(10,10)^\intercal]$. Our DNN static analysis framework computes these interval abstractions for node inputs:

1. Node 5 (after `MatMul`): $[(-200,-200)^\intercal,(200,200)^\intercal]$;

```
1  x = tf.placeholder("float", [2, 2], name='x-input')
2  x_left, x_right = tf.split(x, num_or_size_splits=2, axis=1)
3  x_right = 1
4  x_left = x_left + x_right
5  x_new = tf.concat([x_left, x_right], axis=1)
6  model_output = 2 - x_new + x
7  cost = tf.log(model_output)
```

(a) Code snippet



(b) Computational graph

Figure 2.5: A code snippet and the corresponding computational graph to compare different abstract domains introduced in DEBAR.

2. Node 6 (after `Add`): $[(-210,-210)^\intercal,(210,210)^\intercal]$;

3. Node 8 (after `Softmax` in `float32`): $[(0,0)^\intercal,(1,1)^\intercal]$;

4. Node 9 (after `Sub` of $[1,1]$ and node 8), 10: $[(0,0)^\intercal,(1,1)^\intercal]$.

Since nodes 9 and 10 use the `Log` operator whose invalid input range $(-\infty, \mathrm{U}_{\min})$ overlaps with their input range $[(0,0)^\intercal,(1,1)^\intercal]$, we flag nodes 9 and 10 as potential numerical defects.

Difference among different abstract domains via Figure 2.5 In the previous paragraph, we demonstrated the abstract domain of interval abstraction and tensor expansion. However, the main disadvantage of tensor expansion is scalability, as the number of intervals is the same as the number of all tensor elements. In modern DNNs, this number can be in the millions or billions, making the approach impractical. An alternative abstraction technique is tensor smashing, which uses one abstract element to represent all elements in a tensor, inspired by array smashing. This greatly reduces the number of abstract elements needed, making the approach more scalable.

In Figure 2.5, we consider a 2×2 tensor called `x-input`, which is randomly initialized within the range $[0,1]$. At line 2, we split `x-input` along the column into two tensors, namely `x_left` and `x_right`. Subsequently, we perform some linear arithmetic operators on these tensors at lines 3 and 4. At line 5, we concatenate `x_left` and `x_right` to create

Table 2.2: Abstractions of Figure 2.5 using (a) interval abstraction and tensor expansion and (b) interval abstraction and tensor smashing.

Line	Abstractions of (a)	Abstractions of (b)
1	$\alpha(\texttt{x}) = [\begin{pmatrix}0 & 0\\ 0 & 0\end{pmatrix}, \begin{pmatrix}1 & 1\\ 1 & 1\end{pmatrix}]$	$\alpha(\texttt{x}) = [0, 1]$
2	$\alpha(\texttt{x\_left}) = [(0,0)^\intercal, (1,1)^\intercal]$, $\alpha(\texttt{x\_right}) = [(0,0)^\intercal, (1,1)^\intercal]$	$\alpha(\texttt{x\_left}) = [0, 1]$, $\alpha(\texttt{x\_right}) = [0, 1]$
3	$\alpha(\texttt{x\_right}) = [(1,1)^\intercal, (1,1)^\intercal]$	$\alpha(\texttt{x\_right}) = [1, 1]$
4	$\alpha(\texttt{x\_left}) = [(1,1)^\intercal, (2,2)^\intercal]$	$\alpha(\texttt{x\_left}) = [1, 2]$
5	$\alpha(\texttt{x\_new}) = [\begin{pmatrix}1 & 1\\ 1 & 1\end{pmatrix}, \begin{pmatrix}2 & 1\\ 2 & 1\end{pmatrix}]$	$\alpha(\texttt{x\_new}) = [1, 2]$
6	$\alpha(\texttt{model\_output}) = [\begin{pmatrix}0 & 1\\ 0 & 1\end{pmatrix}, \begin{pmatrix}2 & 2\\ 2 & 2\end{pmatrix}]$	$\alpha(\texttt{model\_output}) = [0, 2]$

a new tensor. The tensor `model_output` is defined at line 6. Finally, we compute the log of `model_output` to obtain the `cost` at the last line. Even though the log for computing `cost` can be a potential defective operator, it can be proved that this operator will not manifest a numerical bug in this case.

Table 2.2 shows a comparison between two methods: (a) interval abstraction and tensor expansion, and (b) interval abstraction and tensor smashing. In lines 5-6, the abstractions of the two approaches are equivalent, but (b) may be less precise than (a). Lines 5-6 demonstrate that the abstraction of (a) is more precise than that of (b) when it comes to handling tensor concatenation. However, both methods will produce false alarms for the log operator in line 7 because they both conclude that the lower bound of `model_output` can reach 0.

To achieve scalability while maintaining sufficient precision, we propose tensor partitioning as a solution. By partitioning tensors, we can infer numerical information over each partition based on the insight that many elements of a tensor are subject to the same operator. For example, we can partition `x_new` by columns and use two intervals to represent the left and right parts of the matrix, rather than using four intervals to represent the entire matrix without losing any accuracy. Table 2.3 displays the abstractions obtained from applying (c) interval abstraction and tensor partitioning. In this approach, we only need seven variables to store intervals, whereas (a) requires 16 intervals.

Although approach (c) reduces the number of variables needed for interval ab-

Table 2.3: Abstractions of Figure 2.5 using (c) interval abstraction and tensor partitioning and (d) interval abstraction with affine equality relation and tensor partitioning. Blue text denotes the abstractions of (d) in addition to (c).

Line	Abstractions of (c) and (d)	
1	$\sigma(\texttt{x}) = v_1, \alpha(v_1) = [0,1]$	
2	$\sigma(\texttt{x}) = (v_2, v_3)$, $\sigma(\texttt{x\_left}) = v_4$, $\sigma(\texttt{x\_right}) = v_5$ $\alpha(v_2) = \alpha(v_3) = \alpha(v_4) = \alpha(v_5) = [0,1]$	$v_2 = v_4, v_3 = v_5$
3	$\sigma(\texttt{x\_right}) = v_6, \alpha(v_6) = [1,1]$	
4	$\sigma(\texttt{x\_left}) = v_7, \alpha(v_7) = [1,2]$	$v_7 = v_4 + v_6$
5	$\sigma(\texttt{x\_new}) = (v_8, v_9)$ $\alpha(v_8) = [1,2]$ $\alpha(v_9) = [1,1]$	$v_8 = v_7, v_9 = v_6$
6	$\sigma(\texttt{model\_output}) = (v_{10}, v_{11})$ $\alpha(v_{10}) = 2 - \alpha(v_8) + \alpha(v_2) = [0,2]$ $\alpha(v_{11}) = 2 - \alpha(v_9) + \alpha(v_3) = [1,2]$	$v_{10} = 2 - v_8 + v_2 = 2 - v_6$, $v_{11} = 2 - v_9 + v_3$ $\alpha(v_{10}) = 2 - \alpha(v_4) = [1,2]$, $\alpha(v_{11}) = [1,2]$

straction, it still generates false alarms. To address this issue, we propose using affine relation analysis to infer element-wise affine equality relations among the abstract elements representing partitions. The last two columns of Table 2.3 presents the abstractions obtained from applying (d) interval abstraction with affine equality relation and tensor partitioning. The key difference between (c) and (d) is that in (d), we store the affine equality relation among variables, $v_7 = v_4 + v_6$, enabling us to perform elimination at line 6. As a result, the bounds of v_6 in approach (d) are tighter than in (c), avoiding false alarms for the operator log in line 7.

Difference between the abstract domains used in DEBAR and RANUM In terms of the abstract domain on tensors, DEBAR passively performs tensor partitioning, i.e., tensor partitioning only occurs after certain operators such as `Split`. To extend DEBAR, RANUM introduces a novel technique called backward fine-grained node labeling. This technique actively detects all nodes that require fine-grained abstractions, such as nodes that determine control flow in a dynamic graph. For these nodes, interval abstractions with the finest granularity are applied to reduce control flow ambiguity. For other nodes, some neighboring elements share the same interval abstraction to

improve efficiency while preserving tightness. As a result, RANUM's static analysis has high efficiency and supports many more DNN operators, including dynamic control-flow operators like `Loop`, compared to DEBAR.

Regarding the abstract domain on values, DEBAR uses interval abstraction with affine equality analysis, while RANUM uses sole interval abstraction. This choice leads to sub-optimal precision compared to DEBAR. However, RANUM makes this choice because sole interval abstraction can be naturally differentiated, and the gradients of the abstraction can be used in succeeding tasks such as feasibility confirmation and fix suggestion.

Feasibility Confirmation via Two-Step Test Generation

Given nodes that contain potential numerical defects (nodes 9 and 10 in Figure 2.3), we generate failure-exhibiting system tests to confirm their feasibility. A failure-exhibiting system test is a tuple $\langle \boldsymbol{x}_{\mathsf{train}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$, such that after training the architecture with the training example $\boldsymbol{x}_{\mathsf{train}}$,[5] with the trained model weights $\boldsymbol{w}_{\mathsf{infer}}$, the inference input $\boldsymbol{x}_{\mathsf{infer}}$ triggers a numerical failure. The name "system test" is inspired by traditional software testing, where we test the method sequence (`m = train(`$\boldsymbol{x}_{\mathsf{train}}$`); m.infer(`$\boldsymbol{x}_{\mathsf{infer}}$`)`). In contrast, GRIST generates model weights $\boldsymbol{w}_{\mathsf{infer}}$ along with inference input $\boldsymbol{x}_{\mathsf{infer}}$ that tests only the inference method `m.infer()`, and the weights may be infeasible from training. Hence, we view the GRIST-generated tuple $\langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$ as a "unit test".

We propose a two-step test generation technique to generate failure-exhibiting system tests.

***Step a**: Generate failure-exhibiting unit test $\langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$.* The state-of-the-art GRIST tool supports this step. However, GRIST solely relies on gradient back-propagation, which is relatively inefficient. In RANUM, we augment GRIST by combining its gradient back-propagation with random initialization inspired by recent research on DNN adversarial attacks (Madry et al., 2018). As a result, RANUM achieves 17.32X speedup with 100% success rate. Back to the running example in Figure 2.3, RANUM can generate $\begin{pmatrix} 5 & -5 \\ -5 & 5 \end{pmatrix}$ for node 2 and $(0.9, -0.9)^\intercal$ for node 4 as model weights $\boldsymbol{w}_{\mathsf{infer}}$; and $(10, -10)^\intercal$ for node 1 and $(1, 0)^\intercal$ for node 11 as the inference input $\boldsymbol{x}_{\mathsf{infer}}$. Such $\boldsymbol{w}_{\mathsf{infer}}$

[5] In particular, if our generation technique outputs $\boldsymbol{x}_{\mathsf{train}}$, the numerical failure can be triggered if the training dataset contains *only* $\boldsymbol{x}_{\mathsf{train}}$ or *only* multiple copies of $\boldsymbol{x}_{\mathsf{train}}$ and the inference-time input is $\boldsymbol{x}_{\mathsf{infer}}$. Our technique can also be applied for generating a batch of training examples by packing the batch as a single example: $\boldsymbol{x}_{\mathsf{train}} = (\boldsymbol{x}_{\mathsf{train}1}, \boldsymbol{x}_{\mathsf{train}2}, \ldots, \boldsymbol{x}_{\mathsf{train}\,\mathrm{B}})$.

and $\boldsymbol{x}_{\mathsf{infer}}$ induce input $(0,1)^\intercal$ and $(1,0)^\intercal$ for nodes 9 and 10, respectively. Since both nodes 9 and 10 use the `log` operator and $\log 0$ is undefined, both nodes 9 and 10 trigger numerical failures.

***Step b**: Generate training example $\boldsymbol{x}_{\mathsf{train}}$ that achieves model weights $\boldsymbol{w}_{\mathsf{infer}}$.* To the best of our knowledge, there is no automatic approach for this task yet. RANUM provides support for this task based on our extension of DLG attack. The DLG attack is originally designed for recovering the training data from training-phase gradient leakage. Here, we figure out the required training gradients to trigger the numerical failure at the inference phase and then leverage the DLG attack to generate $\boldsymbol{x}_{\mathsf{train}}$ that leads to such training gradients. Specifically, many DNN architectures contain operators (such as `ReLU`) on which DLG attack is hard to operate (Serra et al., 2018). We combine straight-through estimator (Bengio et al., 2013) to provide proxy gradients and bypass this barrier. Back to the running example in Figure 2.3, supposing that the initial weights are $\begin{pmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{pmatrix}$ for node 2 and $(0,0)^\intercal$ for node 4, RANUM can generate training example $\boldsymbol{x}_{\mathsf{train}}$ composed of $(5.635, -5.635)^\intercal$ for node 1 and $(1,0)^\intercal$ for node 11, such that one-step training with learning rate 1 on this example leads to $\boldsymbol{w}_{\mathsf{infer}}$. Combining $\boldsymbol{x}_{\mathsf{train}}$ from this step with $\boldsymbol{x}_{\mathsf{infer}}$ from *step a*, we obtain a failure-exhibiting system test that confirms the feasibility of potential defects in nodes 9 and 10.

Fix Suggestion via Abstract Optimization

In this task, we suggest fixes for the confirmed numerical defects. RANUM is the first approach for this task to our knowledge.

The user may prefer different fix locations, which correspond to a user-specified set of nodes $\mathcal{V}_{\mathsf{fix}} \subseteq \mathcal{V}$ to impose the fix. For example, if the fix method is clipping the inference input, $\mathcal{V}_{\mathsf{fix}}$ are input nodes (e.g., nodes 1, 11 in Figure 2.3); if the fix method is clipping the model weights during training, $\mathcal{V}_{\mathsf{fix}}$ are weight nodes (e.g., nodes 2, 4 in Figure 2.3); if the fix method is clipping before the defective operator, $\mathcal{V}_{\mathsf{fix}}$ are nodes with numerical defects (e.g., nodes 9, 10 in Figure 2.3).

According to the empirical study of developers' fixes, 12 out of 13 defects are fixed by imposing interval preconditions for clipping the inputs of $\mathcal{V}_{\mathsf{fix}}$. Hence, we suggest interval precondition, which is interval constraint $\mathbf{l}_n \leqslant \mathsf{f}^{\mathsf{in}}_n(\boldsymbol{x}; \boldsymbol{w}) \leqslant \mathbf{u}_n$ for nodes $n \in \mathcal{V}_{\mathsf{fix}}$, as the defect fix. A fix should satisfy that, when these constraints $\bigwedge_{n \in \mathcal{V}_{\mathsf{fix}}} (\mathbf{l}_n \leqslant \mathsf{f}^{\mathsf{in}}_n(\boldsymbol{x}; \boldsymbol{w}) \leqslant \mathbf{u}_n)$ are imposed, the input of any node in the computational

graph should always be valid, i.e., $f^{in}_{n_0}(x; w) \notin \mathcal{I}_{n_0,\text{invalid}}, \forall n_0 \in \mathcal{V}$.

In RANUM, we formulate the fix suggestion task as a constrained optimization problem, taking the endpoints of interval abstractions for nodes in $\mathcal{V}_{\text{fix}}$ as optimizable variables. We then propose the novel technique of abstraction optimization to solve this constrained optimization problem. Back to the Figure 2.3 example, if users plan to impose a fix on inference input, RANUM can suggest the fix $-1 \leqslant \texttt{x-input} \leqslant 1$; if users plan to impose a fix on nodes with numerical defects, RANUM can suggest the fix $10^{-38} \leqslant$ node 9 & node 10 . `input` $\leqslant +\infty$.

2.3 Approaches and Formalization

Potential-Defect Detection via Static Analysis

In this section, we describe our abstraction for neural architectures using numerical abstractions and tensor partitioning (for abstracting tensors), i.e., combining intervals with affine equalities (for abstracting numerical values).

Numerical Abstraction

In abstract interpretation (Cousot and Cousot, 1977a), concrete properties are described in the concrete domain $\mathbb{C}$ with a partial order $\subseteq$, and abstract properties are described in the abstract domain $\mathbb{A}$ with a partial order $\sqsubseteq$. We say that the correspondence between concrete properties and abstract properties is a Galois connection $\langle \mathbb{C}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$ with an abstraction function $\alpha : \mathbb{C} \mapsto \mathbb{A}$ and a concretization function $\gamma : \mathbb{A} \mapsto \mathbb{C}$ satisfying

$$\forall c \in \mathbb{C}, \forall a \in \mathbb{A}. \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a).$$

To infer the value range for variables in a DL program, we need to compute the possible sets of values that each variable can take. We define the concrete domain $\mathbb{C}$ of n variables as $\mathcal{P}(\mathbb{R}^n)$, where an element is a set of n-element vectors denoting the possible values that n variables can take. The partial order in $\mathbb{C}$ is the subset relation $\subseteq$ over sets.

Abstract Domain of Intervals The abstract domain of intervals $\mathbb{A}_I$ is defined as

$$\mathbb{A}_I \triangleq \{([l_1, u_1], \ldots, [l_n, u_n]) \mid l, u \in \mathbb{R}^n\}.$$

An element in $\mathbb{A}_I$ can be seen as a pair of two vectors (l, u), where $[l_i, u_i]$ denotes the lower bound and upper bound of the values that the i-th variable may take. Given two elements $a_1, a_2 \in \mathbb{A}_I$, we say $a_1 \sqsubseteq a_2$ if both have n intervals and each interval in a_1 is a sub-interval of the interval in the corresponding position in a_2.

The abstraction function α_I of an element $c \in \mathbb{C}$ is defined as

$$\alpha_I(c) = ([l_1^c, u_1^c], \ldots, [l_n^c, u_n^c]),$$

where

$$l_i^c = \min_{x \in c}(x_i), \quad u_i^c = \max_{x \in c}(x_i), \qquad 1 \leqslant i \leqslant n.$$

The concretization function γ_I of an element $a \in \mathbb{A}_I$ is defined as

$$\gamma_I(a) = \{x \in \mathbb{R}^n \mid \forall i \in [1, n].x_i \in [l_i^a, u_i^a]\},$$

where $[l_i^a, u_i^a]$ is the interval range of the i-th variable of a.

It is easy to see that the concrete domain $\langle \mathbb{C}, \subseteq \rangle$ and the interval abstract domain $\langle \mathbb{A}_I, \sqsubseteq_I \rangle$ form the Galois connection.

Abstract Domain of Affine Equalities We also maintain affine relations among variables in a DL program in the form of

$$\sum_{i>0} \omega_i x_i = \omega_0, \tag{2.1}$$

where x_i's are variables and ω_i's are constant coefficients, inferred automatically during the analysis.

The abstract domain of affine equalities (Karr, 1976) $\mathbb{A}_E$ is defined as

$$\mathbb{A}_E \triangleq \{(\mathbf{A}, b) \mid \mathbf{A} \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, m > 0\},$$

where a matrix $\mathbf{A}$ and a column vector b define the affine space of n variables. An

element in $\mathbb{A}_E$ constrains variables $x \in \mathbb{R}^n$ by an equation $\mathbf{A}x = b$ describing the possible set of values that x can take. Furthermore, to have a canonical form, we require $(\mathbf{A}, b)$ to be in the reduced row echelon form (Karr, 1976).

The abstraction function α_E of an element $c \in \mathbb{C}$ is defined as

$$\alpha_E(c) = \begin{cases} (\mathbf{A}, b), & (\mathbf{A}, b) \text{ is in reduced row echelon form, and} \\ & \mathbf{A}x = b \text{ holds for all } x \text{ in } c \\ \top, & \text{if } c \text{ is the whole space} \\ \bot, & \text{otherwise.} \end{cases}$$

The concretization function γ_E of an element $a^{\sharp E} = (\mathbf{A}, b) \in \mathbb{A}_E$ is defined as

$$\gamma_E((\mathbf{A}, b)) = \{x \in \mathbb{R}^n \mid \mathbf{A}x = b\}.$$

The concrete domain $\langle \mathbb{C}, \subseteq \rangle$ and the affine equality abstract domain $\langle \mathbb{A}_E, \sqsubseteq_E \rangle$ form the Galois connection

$$\langle \mathbb{C}, \subseteq \rangle \underset{\alpha_E}{\overset{\gamma_E}{\leftrightarrows}} \langle \mathbb{A}_E, \sqsubseteq_E \rangle.$$

The details about the domain operators (including meet, join, inclusion test, etc.) of the affine equality abstract domain can be found in the publication by Karr (1976). We do not need a widening operator for the domain of affine equalities because the lattice of affine equalities has finite height, and the number of affine equalities while analyzing a program is decreasing until reaching the dimension of the affine space in the program.

Combining Interval Abstraction and Affine Equality Relation Abstraction We combine the interval abstraction and affine equality relation abstraction as our numerical abstraction to infer the value range for scalar variables in the DL programs and also for those auxiliary abstract summary variables introduced by tensor partitioning. We could use relational numerical abstract domains (such as polyhedra) to infer (inequality) relations. However, because many tensor operators induce affine equality relations, in DEBAR, we consider only the affine equality relations among variables. In addition, affine equality relations are cheap to infer, and thus are fit to analyze large DNNs.

Furthermore, because *ReLU* operators are widely used in DNN implementations, for each variable a, we introduce a^{ReLU} to denote the resulting variable of $ReLU(a)$, i.e.,

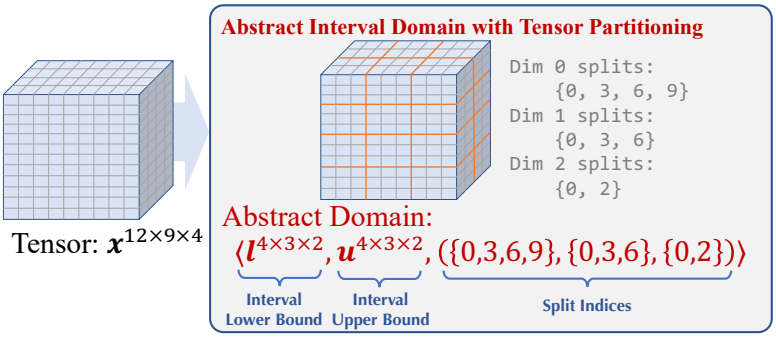


Figure 2.6: Example of tensor partitioning. Tensor partitioning reduces the size of tensors in the abstract domain by sharing one interval bound among all elements inside one subblock.

$a^{ReLU} = max(0, a)$. Considering the way of using *ReLU* operators in DNN implementations, in DEBAR, we maintain only the affine equality relations between a variable b and the *ReLU* result of another variable a of the same shape (while a, b may be the summary variables of partitions from different tensors), in the form of

$$b - a^{ReLU} = 0,$$

which can also be expressed in the form of affine equalities. Additionally, the *ReLU* operator has a property

$$ReLU(a) - ReLU(-a) - a = 0, \forall a \in \mathbb{R}.$$

We can utilize this property for better analysis precision by (1) adding an additional equality

$$a^{ReLU} - a^{-ReLU} - a = 0,$$

where a^{-ReLU} denotes the result of $ReLU(-a)$; (2) adding an additional equality

$$c^{ReLU} - a^{-ReLU} = 0$$

for every equality in the form of $c = -a$.

Tensor Abstraction

We first formally define tensor partitioning. Suppose the tensor has m dimensions with shape $n_1 \times n_2 \times \cdots \times n_m$, we define the abstract domain as such:

$$\begin{aligned}\mathbb{A} := \{\langle a, (S_i)_{i=1}^m \rangle : a \in \mathbb{A}_V^{n_1' \times \cdots \times n_m'} \\ |S_i| = n_i', \forall s \in S_i, s \in \mathbb{N}, 0 \leqslant s < n_i\}.\end{aligned} \tag{2.2}$$

Each element in $\mathbb{A}$ is a tuple $\langle a, (S_i)_{i=1}^m \rangle$, where the first element is a subblock-wise numerical abstraction, which can come from any numerical abstract domain $\mathbb{A}_V$, and the last element $(S_1, S_2, \ldots, S_m)$ contains m sets, where each set S_i corresponds to the *0-indexed* split indices for the i-th dimension to form subblocks. We use $S_i[j] \in \mathbb{N}$ to represent the j-th element of split index set S_i sorted in ascending order and define $S_i[|S_i|] = n_i$.

We further denote a mapping $\mathcal{J}$ from a tuple of indices in $(S_i)_{i=1}^m$ to a set of indices in the original tensor, $\mathcal{J} = [0, n_1') \times \ldots \times [0, n_m') \mapsto \mathcal{P}([0, n_1) \times \ldots \times [0, n_m))$. For a tuple of indices $t = (t_1, \ldots, t_m)$ satisfying $\forall 1 \leqslant i \leqslant m. 0 \leqslant t_i < n_i'$, $\mathcal{J}[t]$ is defined as follows,

$$\mathcal{J}[t] = \left\{(l_1, \ldots, l_m) \mid \forall 1 \leqslant i \leqslant m, l_i \in [S_{t_i}, S_{t_{i+1}})\right\} \tag{2.3}$$

Example 2.2. *Figure 2.6 illustrates this abstraction domain when $\mathbb{A}_V$ is the interval domain $\mathbb{A}_I$. To abstract a set of tensors $\mathcal{X}$ with shape $12 \times 9 \times 4$, we define split indices for each dimension, respectively, then impose interval constraints on each subblock of the tensor. For example, $[\mathbf{l}_{0,0,0}, \mathbf{u}_{0,0,0}]$ constrain any element in $\mathbf{x}_{0:2,0:2,0:1}$, $[\mathbf{l}_{3,2,1}, \mathbf{u}_{3,2,1}]$ constrain any element in $\mathbf{x}_{9:11,6:8,2:3}$.*

Abstract Domain for Neural Architectures

Definition 2.3. *The abstract domain for **T**ensor partitioning and **I**nterval abstraction with affine **E**quality relation $\mathbb{A}_{TIE}$ is defined as*

$$\mathbb{A}_{TIE} \triangleq \{(\mathcal{J}, a^{\sharp I}, a^{\sharp E}) \mid a^{\sharp I} \in \mathbb{A}_I^{n_1' \times \cdots \times n_m'}, a^{\sharp E} \in \mathbb{A}_E^{n_1' \times \cdots \times n_m'}\},$$

where $\mathcal{J}$ is defined in Equation (2.3) and $a^{\sharp I}, a^{\sharp E}$ are the numerical abstract elements over the $\prod_{i=1} n_i'$ summary variables corresponding to the partitions defined by $\mathcal{J}$ in the interval domain $\mathbb{A}_I$ and the affine equality domain $\mathbb{A}_E$, respectively.

Definition 2.4. *The concretization function* γ_{TIE} *of an element* $a^\sharp = (\mathcal{J}, a^{\sharp I}, a^{\sharp E}) \in \mathbb{A}_{\text{TIE}}$ *is defined as*

$$\gamma_{\text{TIE}}(a^\sharp) = \left\{ A \;\middle|\; \begin{array}{c} \forall t \in \text{dom}(\mathcal{J})\ \forall (j_1, \ldots, j_m) \in \mathcal{J}[t]. \\ A[(j_1, \ldots, j_m)] \in (\gamma_I(a^{\sharp I}[t]) \cap \gamma_E(a^{\sharp E}[t])) \end{array} \right\},$$

where we use $a^{\sharp I}[t]$ *and* $a^{\sharp E}[t]$ *to denote the abstractions corresponding to the tuple* t *in* $\mathbb{A}_I^{n'_1 \times \cdots \times n'_m}$ *and* $\mathbb{A}_E^{n'_1 \times \cdots \times n'_m}$*, respectively.*

Backward Fine-Grained Node Labeling Tensor partitioning provides a degree of freedom in terms of the partition granularity, i.e., we can choose the subblock size for each node's abstraction. The elementwise abstraction is the most concrete as it is the finest granularity of tensor partitioning. When the coarsest granularity (i.e., one scalar to summarize the node tensor) is chosen, the abstraction saves the most space and computational cost but loses much precision.

In DEBAR, the coarsest granularity is used by default for most operators. In other words, DEBAR performs tensor partitioning passively, i.e., tensor partitioning only occurs after certain operators, such as `Split`. However, we find that using the finest instead of the coarsest granularity for some nodes is more beneficial for overall abstraction preciseness. For example, the control-flow operators (e.g., `Loop`) benefit from concrete execution to determine the exact control flow in the dynamic graph, and the indexing operators (e.g., `Slice`) and shaping operators (e.g., `Reshape`) benefit from explicit indexers and shapes to precisely infer the output range. Hence, we propose to use the finest granularity for some nodes (namely fine-grained requiring operators) while the coarsest granularity for other nodes during static analysis.

To benefit from the finest granularity abstraction for required nodes, typically, all of their preceding nodes also need the finest granularity. Otherwise, the over-approximated intervals from preceding nodes will be propagated to the required nodes, making the finest abstraction for the required nodes useless. To solve this problem, in RANUM, we back-propagate "fine-grained" labels from these fine-grained requiring nodes to initial nodes by topologically sorting the graph with *inverted* edges, and then apply the finest granularity abstractions on all labeled nodes. In practice, we find that this strategy eliminates the control-flow ambiguity and indexing ambiguity with little loss of efficiency[6]. As a result, RANUM supports all the dynamic graphs that

[6]Theoretically, using the finest granularity for tensor partitioning cannot fully eliminate the ambigu-

are not supported by DEBAR, comprising 39.2% of the benchmarks proposed by Yan et al. (2021).

Notes on the Implementation of Tensor Partitioning First, in DEBAR, all abstract domains are implemented based on the Numpy library instead of PyTorch used in RANUM. This difference results in tensor expansion, i.e., the finest granularity of tensor partitioning, timing out on 33 architectures of the DEBAR benchmark, while tensor expansion worked well in RANUM. We hypothesize that most of the time consumption is due to GPU to CPU memory conversion, which is not a problem in RANUM because the abstract interpretation is performed on GPU using PyTorch.

Second, due to the passive tensor partitioning design made in DEBAR, it only implemented tensor partitioning abstract transformers for a limited number of APIs. In contrast, RANUM implemented tensor partitioning abstract transformers for almost all APIs because RANUM allows developers to determine the granularities of input tensors. When preceding nodes use finer-grain abstraction granularity, the subsequent nodes should preserve such fine granularity to maintain the analysis preciseness. Principally, the choice of abstraction granularity should satisfy both tightness (bearing no precision loss compared to elementwise interval abstraction) and minimality (using the minimum number of partitions for high efficiency). To realize these principles, we dynamically determine a node's abstraction granularity based on the granularity of preceding nodes. The abstraction design for some operators is non-trivial. Omitted details (formulation, illustration, and proofs) about the static analysis framework are provided in Appendix C of RANUM's paper and RANUM's implementation.

Two-Step Test Generation for Feasibility Confirmation

RANUM generates failure-exhibiting system tests for the given DNN to confirm the feasibility of potential numerical defects. Here, we take the DNN architecture as the input. From the static analysis framework, we obtain a list of nodes that have potential numerical defects. For each node n_0 within the list, we apply our technique of two-step test generation to produce a failure-exhibiting system test $t_{sys} = \langle \boldsymbol{x}_{train}, \boldsymbol{x}_{infer} \rangle$ as the

ity, since interval abstraction is intrinsically an over-approximation. Nevertheless, in our evaluation, we find that this technique eliminates control-flow and indexing ambiguities on all 63 programs in the benchmarks.

output. According to Section 2.2, the test should satisfy that after the architecture is trained with $\boldsymbol{x}_{\mathsf{train}}$, entering $\boldsymbol{x}_{\mathsf{infer}}$ in the inference phase results in a numerical failure.

We propose the novel technique of two-step test generation: first, generate failure-exhibiting unit test $\langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$; then, generate training example $\boldsymbol{x}_{\mathsf{train}}$ that leads model weights to be close to $\boldsymbol{w}_{\mathsf{infer}}$ after training.

Step a: Unit Test Generation As sketched in Section 2.2, we strengthen the state-of-the-art unit test generation approach, GRIST (Yan et al., 2021), by combining it with random initialization to complete this step. Specifically, GRIST leverages the gradients of the defective node's input with respect to the inference input and weights to iteratively update the inference input and weights to generate failure-exhibiting unit tests. However, GRIST always conducts updates from the existing inference input and weights, suffering from local minima problem. Instead, motivated by DNN adversarial attack literature (Madry et al., 2018), a sufficient number of random starts help find global minima effectively. Hence, in RANUM, we first conduct uniform sampling 100 times for both the inference input and weights to trigger the numerical failure. If no failure is triggered, we use the sample that induces the smallest loss as the start point for gradient optimization. As Section IV-A in the RANUM paper shows, this strategy substantially boosts the efficiency, achieving 17.32X speedup.

Step b: Training Example Generation For this step, RANUM takes the following inputs: (1) the DNN architecture, (2) the failure-exhibiting unit test $\mathrm{t}_{\mathsf{unit}} = \langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$, and (3) the randomly initialized weights $\boldsymbol{w}_0$. Our goal is to generate a legal training example $\boldsymbol{x}_{\mathsf{train}}$, such that the model trained with $\boldsymbol{x}_{\mathsf{train}}$ will contain weights close to $\boldsymbol{w}_{\mathsf{infer}}$.

DNNs are typically trained with gradient-descent-based algorithms such as stochastic gradient descent (SGD) (Cai et al., 2023). In SGD, in each step t, we sample a mini-batch of samples from the training dataset to compute their gradients on model weights and use these gradients to update the weights. We focus on one-step SGD training with a single training example, since generating a single one-step training example to exhibit a failure is more desirable for debugging because, in one-step training, the model weights are updated strictly following the direction of the gradients. Therefore, developers can inspect inappropriate weights, easily trace back to nodes with inappropriate gradients, and then fix these nodes. In contrast, in multi-step

training, from inappropriate weights, developers cannot trace back to inappropriate gradients because weights are updated iteratively and interactions between gradients and weights are complex (even theoretically intractable (Li and Yuan, 2017)).

In this one-step training case, after training, the model weights $\boldsymbol{w}_{\mathsf{infer}}$ satisfy

$$\boldsymbol{w}_{\mathsf{infer}} = \boldsymbol{w}_0 - \gamma \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}_{\mathsf{train}}; \boldsymbol{w}_0), \tag{2.4}$$

where $\gamma \in \mathbb{R}_+$ is a predefined learning rate, and $\mathcal{L}$ is the predefined loss function in the DNN architecture. Hence, our goal becomes finding $\boldsymbol{x}_{\mathsf{train}}$ that satisfies

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}_{\mathsf{train}}; \boldsymbol{w}_0) = (\boldsymbol{w}_0 - \boldsymbol{w}_{\mathsf{infer}})/\gamma. \tag{2.5}$$

The DLG attack (Zhu et al., 2019b) is a technique for generating input data that induce specific weight gradients. The attack is originally designed for recovering training samples from monitored gradient updates. Since the right-hand side (RHS) of Equation (2.5) is known, our goal here is also to generate input example $\boldsymbol{x}_{\mathsf{train}}$ that induces specific weight gradients. Therefore, we leverage the DLG attack to generate training example $\boldsymbol{x}_{\mathsf{train}}$.

Extending DLG Attack with Straight-Through Estimator Directly using DLG attack suffers from an optimization challenge in our scenario. Specifically, in DLG attack, suppose that the target weight gradients are $\Delta \boldsymbol{w}_{\mathsf{targ}}$, we use gradient descent over the squared error $\|\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}; \boldsymbol{w}_0) - \Delta \boldsymbol{w}_{\mathsf{targ}}\|_2^2$ to generate $\boldsymbol{x}$. In this process, we need meaningful gradient information of this squared error loss to perform the optimization. However, the gradient of this loss involves second-order derivatives of $\mathcal{L}(\boldsymbol{x}; \boldsymbol{w}_0)$, which could be zero. For example, DNNs with `ReLU` as activation function are piecewise linear and have zero second-order derivatives almost everywhere. This optimization challenge is partly addressed in DLG attack by replacing `ReLU` with `Sigmoid`, but it changes the DNN architecture (i.e., the system under test) and hence is unsuitable.

We leverage the straight-through estimator to mitigate the optimization challenge. Specifically, for a certain operator, such as `ReLU`, we do not change its forward computation but change its backward gradient computation to provide second-order derivatives within the DLG attack process. For example, for `ReLU`, in backward computation we use the gradient of Softplus function, namely $1 - \frac{1}{1+\exp(\boldsymbol{x})}$, because Softplus is an approximation of `ReLU` with non-zero second-order derivatives. Note that we modify the

computed gradients only within the DLG attack process. After such $\mathbf{x}_{\mathsf{train}}$ is generated by the attack, we evaluate whether it triggers a numerical failure using the original architecture and gradients in Equation (2.4).

We listed hyperparameters in our two-step system test generation technique: For the unit test generation, we use the Adam optimizer where the learning rate is 1 and the maximum iteration number is 100. For the training example generation, we target for training example under learning rate $\gamma = 1$ and the approach has similar performance under other learning rates. We follow the convention in the DLG attack, where we use the L-BFGS method as the optimizer for gradient-based minimization. We terminate the method and return "`failed`" if either the running time exceeds 1800 s (universal execution time limit for all experimental evaluations), or a failure-exhibiting example training is not found after 300 iterations of L-BFGS optimization.

Abstraction Optimization for Fix Suggestion

In this task, we aim to generate the precondition fix given imposing locations. The inputs are the DNN architecture, the node n_0 with numerical defects, and a node set $\mathcal{V}_{\mathsf{fix}}$ to impose the fix. We would like to generate interval preconditions for $\mathcal{V}_{\mathsf{fix}}$ node inputs so that after these preconditions are imposed, the defect on n_0 is fixed.

Formally, our task is to find $\langle l_n, u_n \rangle$ for each $n \in \mathcal{V}_{\mathsf{fix}}$ (l_n and u_n are scalars so the same interval bound applied to all elements of n's tensor), such that for any $\mathbf{x}, \boldsymbol{w}$ satisfying $f_n^{\mathsf{in}}(\mathbf{x}; \boldsymbol{w}) \in [l_n, u_n], \forall n \in \mathcal{V}_{\mathsf{fix}}$, for the defective node n_0, we have $f_{n_0}^{\mathsf{in}}(\mathbf{x}; \boldsymbol{w}) \notin \mathcal{I}_{n_0,\mathsf{invalid}}$, where the full list of invalid input ranges $\mathcal{I}_{n_0,\mathsf{invalid}}$ is in Table 2.1. There is an infinite number of possible $\langle l_n, u_n \rangle$ interval candidates since l_n and u_n are floating numbers. Hence, we need an effective technique to find a valid solution from the exceedingly large search space that incurs a relatively small model utility loss. To achieve so, we formulate a surrogate optimization problem for this task.

$$\max_{l_n, u_n : n \in \mathcal{V}_{\mathsf{fix}}} \quad s \quad \text{s.t.} \quad u_n \geqslant l_n + s(u_n^{\mathsf{valid}} - l_n^{\mathsf{valid}}), \forall n \in \mathcal{V}_{\mathsf{fix}}, \tag{2.6}$$

$$l_n^{\mathsf{valid}} \leqslant l_n \leqslant u_n \leqslant u_n^{\mathsf{valid}}, \forall n \in \mathcal{V}_{\mathsf{fix}}, \tag{2.7}$$

$$\mathcal{L}_{n_0}^{\mathsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}}) < 0. \tag{2.8}$$

Here, l_n^{valid} and u_n^{valid} are the valid ranges (of the node's input n), which are fixed and determined by the valid ranges of input and weights. $\mathcal{L}_{n_0}^{\mathsf{precond}}$ is the node-specific

Algorithm 1 Abstraction Optimization (Section 2.3)

Input: DNN architecture $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, defective node $n_0 \in \mathcal{V}$, nodes to impose fix $\mathcal{V}_{\text{fix}} \subseteq \mathcal{V}$

1: $s \leftarrow 1, \gamma_s \leftarrow 0.9, \gamma_c \leftarrow 0.1, \text{minstep} \leftarrow 0.1, \text{maxiter} \leftarrow 1000$
2: $c_n \leftarrow (l_n^{\text{valid}} + u_n^{\text{valid}})/2, l_n \leftarrow l_n^{\text{valid}}, u_n \leftarrow u_n^{\text{valid}}, \forall n \in \mathcal{V}_{\text{fix}}$
3: **for** $i = 1$ to maxiter **do**
4: **for** $n \in \mathcal{V}_{\text{fix}}$ **do**
5: $\text{loss} \leftarrow \mathcal{L}_{n_0}^{\text{precond}}(\{l_{n'}, u_{n'}\}_{n' \in \mathcal{V}_{\text{fix}}})$
6: $c_n \leftarrow c_n - \gamma_c \max\{|c_n|, \text{minstep}\}\text{sgn}(\nabla_{c_n}\text{loss})$
7: $(l_n, u_n) \leftarrow (c_n - \frac{s(u_n^{\text{valid}} - l_n^{\text{valid}})}{2}, c_n + \frac{s(u_n^{\text{valid}} - l_n^{\text{valid}})}{2})$
8: $(l_n, u_n) \leftarrow (\max\{l_n, l_n^{\text{valid}}\}, \min\{u_n, u_n^{\text{valid}}\})$
9: **end for**
10: **if** $\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}) < 0$ **then**
11: **return** $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$ // Find precondition fix
12: **end if**
13: $s \leftarrow \gamma_s \cdot s$
14: **end for**
15: **return** "failed" // Failed to find precondition fix

precondition generation loss that is the distance between the furthest endpoint of defective node n_0's interval abstraction and n_0's valid input range. Hence, when $\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}})$ becomes negative, the solution $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$ is a valid precondition. The optimization variables are the precondition interval endpoints l_n and u_n and the objective is the relative span of these intervals. The larger the span is, the looser the precondition constraints are, and the less hurt they are for the model's utility. Equation (2.6) enforces the interval span requirement. Equation (2.7) assures that the precondition interval is in the valid range. Equation (2.8) guarantees the validity of the precondition as a fix.

For any $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$, thanks to RANUM's static analysis framework, we can compute induced intervals of defective node n_0, and thus compute the loss value $\mathcal{L}_{n_0}^{\text{precond}}$.

As shown in Algorithm 1, we propose the technique of **abstraction optimization** to effectively and approximately solve this optimization. Our technique works iteratively. In the first iteration, we set span $s = 1$, and in the subsequent iterations, we reduce the span s exponentially as shown in Line 13 where hyperparameter $\gamma_s = 0.9$. Inside each iteration, for each node to impose precondition $n \in \mathcal{V}_{\text{fix}}$, we use the interval center $c_n = (l_n + u_n)/2$ as the optimizable variable and compute the *sign* of its gradient: $\text{sgn}(\nabla_{c_n}\text{loss})$. We use this gradient sign to update each c_n toward reducing the loss value in Line 6. Then, we use c_n and the span s to recover the actual interval in

Line 7 and clip l_n and u_n by the valid range $[l_n^{\mathsf{valid}}, u_n^{\mathsf{valid}}]$ in Line 8. At the end of this iteration, for updated l_n and u_n, we compute $\mathcal{L}_{n_0}^{\mathsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}})$ to check whether the precondition is a fix. If so, we terminate; otherwise, we proceed to the next iteration. We note that *if the algorithm finds a precondition, the precondition is guaranteed to be a valid fix* by the soundness nature of our static analysis framework and the definition of $\mathcal{L}_{n_0}^{\mathsf{precond}}$. When no feasible precondition is found within $\mathsf{maxiter} = 1000$ iterations, we terminate the algorithm and report "failed to find the fix".

Remark 2.5. *The key ingredient in the technique is the gradient-sign-based update rule (shown in Line 6), which is much more effective than normal gradient descent for two reasons.* (1) *Our update rule can get rid of gradient explosion and vanishing problems. For early optimization iterations, the span* s *is large and interval bounds are generally coarse, resulting in too large or too small gradient magnitude. For example, the input range for* `Log` *could be* $[1, 10^{10}]$ *where gradient can be* 10^{-10}*, resulting in almost negligible gradient updates. In contrast, our update rule leverages the gradient sign, which always points to the correct gradient direction. The update step size in our rule is the maximum of current magnitude* $|c_n|$ *and* $\mathsf{minstep}$ *to avoid stagnation.* (2) *Our update rule mitigates the gradient magnitude discrepancy of different* c_n*. At different locations, the nodes in DNNs can have diverse value magnitudes that are not aligned with their gradient magnitudes, making gradient optimization challenging. Therefore, we use this update rule to solve the challenge, where the update magnitude depends on the value magnitude* ($|c_n|$) *instead of gradient magnitude* ($\nabla_{c_n}\mathsf{loss}$)*. We empirically compare our technique with standard gradient descent in Section 2.4.*

2.4 Experiments

We report experiment results of DEBAR on potential-defect detection and RANUM on feasibility confirmation and fix suggestion in this section.

Potential-Defect Detection: DEBAR

We compare the effectiveness of four abstract domains: (a) sole interval abstraction and tensor expansion, (b) sole interval abstraction and tensor smashing, (c) sole interval abstraction and tensor partitioning, and (d) interval abstraction with affine equality relation analysis and tensor partitioning. Note that even though (c) is theoretically

equivalent to the abstract domain used in RANUM, the actual results of (c) and RANUM are incomparable due to differences in implementation:

1. The ONNX representation sometimes decomposes one operator in the TensorFlow computation graph into two operators in the ONNX graph. This difference can make the same abstract interpretation on the ONNX representation less precise than on the TensorFlow representation. For example, consider a `Square(x)` node being decomposed into a `Mul(x, x)` node with two inputs, and suppose $\alpha(x) = [-1, 1]$. For the `Square` node, we can compute the tightest output $[0, 1]$ using the abstract transformer for `Square`. However, for `Mul`, we can only get $[-1, 1] = [-1, 1] \times [-1, 1]$ using interval arithmetic for multiplication.

2. As noted in Section 2.3, DEBAR and RANUM make different design choices regarding how to partition tensors. This difference in design choices also makes RANUM's implementation of tensor partitioning more comprehensive than DEBAR's.

Experiment Setups

Datasets We collect two datasets for the evaluation. The first dataset is a set of 9 buggy architectures collected by existing studies. The buggy architectures come from two studies: 8 architectures were collected by a previous empirical study (Zhang et al., 2018b) on TensorFlow bugs and 1 architecture was obtained from a study conducted to evaluate TensorFuzz (Odena et al., 2019).

As most of the architectures in the first dataset are small, we collect the second dataset, which contains 48 architectures from a large collection of research projects in the repository of TensorFlow Research Models[7]. The whole collection contains 66 projects implemented in TensorFlow by researchers and developers for different tasks in various domains, including computer vision, natural language processing, speech recognition, and adversarial machine learning. We first filter out the projects that are not related to specific neural architectures such as API frameworks and optimizers. We further filter out the projects of which the computation graph cannot be generated due to incomplete documentation or complicated configuration. As a result, 32 projects remain after filtering, and some of them contain more than one neural architecture.

[7] `https://github.com/tensorflow/models/blob/13e7c85d521d7bb7cba0bf7d743366f7708b9df7/research`

Overall, our second dataset contains a great diversity of neural architectures such as Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Generative Adversarial Network (GAN), and Hidden Markov Model (HMM). Note that we have no knowledge about whether the architectures in this dataset contain numerical bugs when collecting the dataset.

For every architecture in the two datasets, we extract the computation graph via a TensorFlow API. Each extracted computation graph is represented by a Protocol Buffer file[8], which provides the operators (nodes) and the data flow relations (edges). We make 48 computation graphs publicly available[9].

Columns 1–4 in Table 2.4 provide an overview of the two datasets. Column 2 provides an estimation of the lines of code in the corresponding deep learning programs. Column 3 shows the number of operators in the computation graphs, and *textsum* has the highest number of operators (208,412). Moreover, Column 4 shows the number of parameters (trainable weights) in the DNN architectures, and *lm_1b* has the largest number of parameters (1.04G).

Measurements Our approach checks every operator that may lead to a numerical error and determines whether a warning should be reported. To measure the effectiveness of our approach, we treat it as a classifier that classifies whether each operator is buggy, and evaluates its effectiveness using the number of true/false positives/negatives and accuracy. More concretely, true (false) positives refer to the warnings that are (not) indeed bugs, true (false) negatives refer to those correct (buggy) operators where no warning is reported, and accuracy is calculated using the following formula, where TP/FP refers to true/false positive and TN/FN refers to true/false negative.

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

For the first dataset, we refer to user patches to determine whether a warning is a bug. For the second dataset,

- 204 true positives are confirmed by executing the architecture under analysis using the designed inputs and parameters to trigger the numerical errors.

[8] `https://en.wikipedia.org/wiki/Protocol_Buffers`
[9] `https://doi.org/10.5281/zenodo.3843648`

- 52 true positives are confirmed by the developer-provided fixes (not merged yet) in the issue discussion.
- 43 true positives are confirmed when two authors of DEBAR separately do reasoning on each computation graph, and both authors conclude that each warning is true positive.

Since our approach does not have false negatives by design, we omit this column in reporting our evaluation results.

Experiment Results

Columns 5–9 of Table 2.4 show the results. We make the following observations about DEBAR.

- It detects all known numerical errors on the 9 architectures in the first dataset, with zero false positive.
- It detects 299 previously unknown operators that may lead to numerical errors in the real-world architectures from the second dataset. Note that a numerical bug can trigger multiple numerical errors at different operators, e.g., failing to normalize an input tensor that is used in multiple operators.
- It correctly classifies 3,073 operators with only 230 false positives, achieving accuracy of 93.0%.
- It is scalable to handle the real-world architectures, all of which are analyzed in 3 minutes, and the average time is 12.1 seconds.

To understand why DEBAR generates some false positives, we investigate the false positives (FPs) and find the following reasons.

- Some operators depend on an argument to index the tensor elements for the operators. For example, function `gather` returns elements in a tensor based on an argument that specifies the elements' indexes. Since we do not know beforehand which indexes are subject to an operator, we merge the intervals at all possible indexes, leading to imprecision. 116 FPs belong to this category.

Table 2.4: Dataset Overview and Results

Name	LoC	#Ops	#Params	TP	DEBAR				Array Smashing				Sole Interval Abstraction			
					TN	FP	Acc	Time	TN	FP	Acc	Time	TN	FP	Acc	Time
TensorFuzz	77	225	178K	4	0	0	100.0%	1.9	0	0	100.0%	1.9	0	0	100.0%	1.7
Github-IPS-1	367	1,546	5.05M	1	4	0	100.0%	2.3	4	0	100.0%	2.2	4	0	100.0%	2.2
Github-IPS-6	2,377	167	23.6K	2	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
Github-IPS-9	226	102	23.6K	1	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
SO-IPS-1	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.8	1	0	100.0%	1.8
SO-IPS-2	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.7	1	0	100.0%	1.8
SO-IPS-6	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.8	1	0	100.0%	1.8
SO-IPS-7	49	145	407K	2	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
SO-IPS-14	48	74	7.85K	1	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.6
ssd_mobile_net_v1	71,242	22,412	27.3M	26	233	48	84.4%	21.8	137	144	53.1%	21.5	136	145	52.8%	21.6
ssd_inception_v2	71,242	23,929	100M	3	49	2	96.3%	19.8	45	6	88.9%	19.8	44	7	87.0%	19.7
ssd_mobile_net_v2	71,242	28,724	24.3M	26	233	48	84.4%	25.5	137	144	53.1%	26.0	136	145	52.8%	25.9
frcnn_resnet_50	71,242	12,485	73.4M	5	31	22	62.1%	11.4	31	22	62.1%	11.4	31	22	62.1%	11.5
deep_speech	659	7,318	0.131K	0	6	0	100.0%	6.9	6	0	100.0%	7.1	6	0	100.0%	7.1
deeplab	7,514	21,100	87.1M	0	2	0	100.0%	17.8	2	0	100.0%	17.7	2	0	100.0%	18.3
autoencoder_mnae	369	187	944K	0	1	0	100.0%	2.6	1	0	100.0%	2.7	1	0	100.0%	2.6
autoencoder_vae	369	370	1.41M	2	1	0	100.0%	2.7	1	0	100.0%	2.7	1	0	100.0%	2.7
attention_ocr	1,772	3,624	1.74M	1	4	2	71.4%	5.2	4	2	71.4%	5.1	4	2	71.4%	5.2
textsum	906	208,412	10.5M	0	94	0	100.0%	105.7	94	0	100.0%	103.1	94	0	100.0%	106.4
shake_shake_32	1,233	7,348	5.85M	0	55	0	100.0%	7.5	55	0	100.0%	7.6	55	0	100.0%	7.7
shake_shake_96	1,233	7,348	52.4M	0	55	0	100.0%	7.6	55	0	100.0%	7.7	55	0	100.0%	7.7
shake_shake_112	1,233	7,348	71.3M	0	55	0	100.0%	7.6	55	0	100.0%	7.6	55	0	100.0%	7.5
pyramid_net	1,233	43,142	52.6M	0	7	0	100.0%	37.9	7	0	100.0%	38.7	7	0	100.0%	39.2
sbn	1,108	11,262	2.21M	0	42	3	93.3%	4.1	42	3	93.3%	4.1	26	19	57.8%	3.7
sbnrebar	1,108	11,262	2.21M	0	187	2	98.9%	9.8	187	2	98.9%	9.8	107	82	56.6%	8.8
sbndynamicrebar	1,108	31,530	2.61M	0	194	2	99.0%	18.7	194	2	99.0%	19.2	114	82	58.2%	17.9
sbngumbel	1,108	2,070	1.98M	0	78	2	97.5%	4.6	78	2	97.5%	4.6	46	34	57.5%	4.0
audioset	405	699	216M	0	2	0	100.0%	2.9	2	0	100.0%	2.9	1	1	50.0%	2.9
learning_to_rem	702	1,027	4.30M	0	6	0	100.0%	3.1	6	0	100.0%	3.1	6	0	100.0%	3.1
neural_gpu1	2,401	5,080	2.68M	0	53	0	100.0%	5.5	53	0	100.0%	5.5	53	0	100.0%	5.6
neural_gpu2	2,401	2,327	1.35M	0	38	0	100.0%	4.2	38	0	100.0%	4.2	38	0	100.0%	4.2
ptn	1,713	23,636	145M	0	351	0	100.0%	14.8	351	0	100.0%	15.0	351	0	100.0%	14.8
namignizer	262	2,310	652K	0	3	0	100.0%	3.5	3	0	100.0%	3.5	3	0	100.0%	3.5
feelvos	2,955	83,558	83.0M	0	4	0	100.0%	135.4	4	0	100.0%	137.7	4	0	100.0%	132.6
fivo_srnn	5,661	3,514	357K	0	7	11	38.9%	4.2	7	11	38.9%	4.3	7	11	38.9%	4.3
fivo_vrnn	5,661	3,820	365M	0	7	11	38.9%	4.4	7	11	38.9%	4.5	7	11	38.9%	4.5
fivo_ghmm	5,661	2,759	60	0	9	23	28.1%	4.0	9	23	28.1%	4.1	9	23	28.1%	4.1
dcb_var_bnn	2,143	474	36.0K	0	22	0	100.0%	3.0	22	0	100.0%	3.0	22	0	100.0%	3.0
dcb_neural_ban	2,143	186	18.0K	0	4	0	100.0%	2.7	4	0	100.0%	2.7	4	0	100.0%	2.7
dcb_bb_alpha_nn	2,143	11,180	36.0K	0	163	2	98.8%	8.2	163	2	98.8%	8.2	163	2	98.8%	8.4
dcb_rms_bnn	2,143	186	18.0K	0	4	0	100.0%	2.7	4	0	100.0%	2.7	4	0	100.0%	2.7
adversarial_crypto	133	676	8.14K	0	6	0	100.0%	3.0	6	0	100.0%	3.0	6	0	100.0%	3.0
sentiment_analysis	130	334	4.39M	0	3	1	75.0%	2.7	3	1	75.0%	2.7	3	1	75.0%	2.7
next_frame_pred	493	2,820	6.70M	1	6	0	100.0%	4.0	6	0	100.0%	4.1	6	0	100.0%	4.0
minigo	3,774	929	34.4K	1	0	0	100.0%	3.0	0	0	100.0%	3.0	0	0	100.0%	3.0
c_entropy_coder	2,000	15,709	20.0K	0	13	0	100.0%	9.7	13	0	100.0%	10.0	13	0	100.0%	9.8
lfads	2,898	51,853	928K	202	213	3	99.3%	48.7	213	3	99.3%	49.5	213	3	99.3%	48.6
lm_1b	3,81	2,926	1.04G	0	1	0	100.0%	4.0	1	0	100.0%	4.0	1	0	100.0%	4.0
swivel	1,449	279	36.0K	0	1	0	100.0%	2.7	1	0	100.0%	2.7	1	0	100.0%	2.7
skip_thought	1,129	6,800	377M	0	15	0	100.0%	5.7	15	0	100.0%	5.8	15	0	100.0%	5.7
video_prediction	462	48,148	41.6M	32	288	0	100.0%	30.7	288	0	100.0%	30.6	288	0	100.0%	30.5
gan_mnist	806	2,664	39.7M	0	3	0	100.0%	3.7	3	0	100.0%	3.8	3	0	100.0%	3.7
gan_cifar	510	3,784	43.3M	0	17	0	100.0%	4.5	17	0	100.0%	4.5	17	0	100.0%	4.5
gan_image_c	444	4,230	35.5M	0	17	0	100.0%	4.7	17	0	100.0%	4.7	17	0	100.0%	4.7
vid2depth	2,502	35,072	99.6M	0	132	48	73.3%	21.2	132	48	73.3%	21.9	132	48	73.3%	21.5
domain_adaptation	3,079	6,010	7.01M	0	28	0	100.0%	5.6	28	0	100.0%	5.6	25	3	89.3%	5.7
delf	3,683	2,712	9.10M	0	10	0	100.0%	5.1	10	0	100.0%	5.1	10	0	100.0%	5.0
Total	—	—	—	313	2760	230	93.0%	691.1	2564	426	87.1%	694.9	2349	641	80.6%	688.7

- Our affine relation analysis works on only linear expressions. When a non-linear operator is used, we create a new abstract element, leading to imprecision. 15 FPs belong to this category.

- 48 FPs belong to both of the preceding two categories.

Table 2.5: Results of Array Expansion

Name	Array Expansion			
	TN	FP	Acc	Time
TensorFuzz	0	0	100%	29.6
GitHub-IPS-6	0	0	100%	3.2
GitHub-IPS-9	0	0	100%	3.1
SO-7	0	0	100%	72.4
SO-14	0	0	100%	2.4
autoencoder_mnae	1	0	100.0%	105.4
autoencoder_vae	1	0	100.0%	232.6
sbn	42	3	93.3%	397.8
sbnrebar	187	2	98.9%	725.1
sbngumbel	78	2	97.5%	401.2
learning_to_remember	6	0	100.0%	913.5
neural_gpu1	53	0	100.0%	702.8
neural_gpu2	38	0	100.0%	336.8
namignizer	3	0	100.0%	629.9
fivo_srnn	7	11	38.9%	44.1
fivo_ghmm	9	23	28.1%	4.1
dcb_var_bnn	22	0	100.0%	12.3
dcb_neural_ban	4	0	100.0%	4.0
dcb_bb_alpha_nn	163	2	98.8%	155.5
dcb_rms_bnn	4	0	100.0%	4.0
adversarial_crypto	6	0	100.0%	3.6
sentiment_analysis	3	1	75.0%	693.8
mingo	0	0	100.0%	8.0
c_entropy_coder	13	0	100.0%	340.7

- For while loops in RNNs, we do not use tensor partitioning and elementwise affine equality relations but use the classic Kleene iteration together with the widening operator (Cousot and Cousot, 1977a) in the interval abstract domain, leading to imprecision. 50 FPs belong to this category.

- The TensorFlow API used to extract computation graphs fails to analyze the shapes of some tensors, leading to 1 FP.

Columns 10–13 of Table 2.4 show the results of array smashing, and Table 2.5 shows the results of array expansion. Since array expansion times out on 33 of the subjects with a time budget of 30 minutes, we report only the results on the remaining 24 subjects. From the tables, we make the following observations.

- Compared to array smashing, DEBAR even runs faster, indicating that the over-

head of tensor partitioning is so negligible that the overhead is dominated by the random error of execution time, and DEBAR successfully eliminates 196 more false positives, improving the (total) accuracy from 87.1% to 93.0%.

- Compared to array expansion, the analysis of DEBAR runs seconds to hundreds of seconds faster, and does not lose any accuracy on all the 24 subjects that array expansion can analyze within the time budget of 30 minutes.

These observations confirm that tensor partitioning is more effective than the other two tensor abstraction techniques.

Columns 14–17 of Table 2.4 show the results of DEBAR and sole interval abstraction. DEBAR has a negligible overhead (0.3% on average) and eliminates 411 false positives, improving the accuracy from 80.6% to 93.0% in total. These observations indicate that the affine relation analysis is effective and substantially contributes to the overall effectiveness.

Feasibility Confirmation: RANUM

We compare RANUM with baseline approaches (constructed by leaving our novel techniques out of RANUM) on feasibility confirmation.

Experiment Setups

Datasets We conduct the evaluation on the GRIST benchmarks (Yan et al., 2021), being the largest dataset of real-world DNN numerical defects to our knowledge. The benchmarks contain 63 real-world deep learning programs with numerical defects collected from previous studies and GitHub. Each program contains a DNN architecture, and each architecture has one or more numerical defects. There are 79 real numerical defects in total.

Evaluation Protocol RANUM confirms the feasibility of potential numerical defects by generating failure-exhibiting system tests. Since RANUM is the first approach for this task, we do not compare with existing literature and propose one random-based approach (named "Random" hereinafter) as the baseline. In "Random", we first generate a failure-exhibiting unit test with random sampling. If there is any sample that triggers a failure, we stop and keep the inference example part as the desired

Table 2.6: Results of failure-exhibiting **system** test generation with RANUM and Random (baseline). C is the total number of runs where numerical failures are triggered in 10 repeated runs. T is the average execution time per run.

	RANUM		Random			RANUM		Random			RANUM		Random			RANUM		Random			RANUM		Random	
ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T
1	0	9.01	0	1806.13	13	10	0.01	10	0.01	27	10	0.01	10	0.01	38	1	0.13	0	1800.65	49b	10	0.50	8	364.09
2a	10	0.03	10	0.06	14	10	12.60	10	0.50	28a	0	24.37	0	1920.29	39a	10	0.43	1	1623.72	50	10	4.89	10	0.16
2b	10	0.03	10	0.06	15	0	1.71	0	2107.24	28b	0	24.17	0	1911.26	39b	10	0.43	8	364.10	51	10	49.12	10	2.10
3	10	0.02	10	0.05	16a	10	0.12	10	0.75	28c	10	0.12	10	0.53	40	10	0.06	10	0.02	52	10	4.87	10	0.15
4	10	0.01	10	0.01	16b	10	0.21	0	1834.44	28d	10	0.12	10	0.48	41	10	0.06	10	0.02	53	10	0.07	10	0.03
5	10	0.05	10	0.06	16c	10	0.25	0	1831.67	29	10	0.89	10	11.96	42	10	0.06	10	0.02	54	10	0.07	10	0.02
6	10	0.84	10	12.42	17	10	549.98	10	235.11	30	10	4.88	10	0.14	43a	10	0.48	1	1623.71	55	10	0.82	10	12.79
7	10	0.87	10	12.51	18	10	0.02	10	0.05	31	10	14.62	10	9.31	43b	10	0.45	9	184.14	56	10	0.07	10	0.03
8	10	0.86	10	12.37	19	10	4.88	10	0.16	32	10	0.08	10	0.03	44	10	0.27	10	1.36	57	10	0.01	8	360.01
9a	10	0.20	7	541.25	20	10	4.88	10	0.14	33	10	0.07	10	0.02	45a	10	4.89	10	0.15	58	10	0.83	10	12.28
9b	10	0.14	10	1.39	21	10	4.89	10	0.14	34	10	0.42	10	0.20	45b	10	0.88	10	12.27	59	10	0.02	10	0.05
10	10	4.90	10	0.16	22	10	500.10	0	1801.60	35a	10	0.44	10	4.01	46	10	0.01	10	0.01	60	10	4.88	10	0.16
11a	10	0.15	10	0.72	23	10	0.01	10	0.01	35b	10	0.45	10	4.22	47	10	0.08	10	0.03	61	10	9.84	10	0.93
11b	10	0.13	10	0.76	24	10	0.81	10	12.52	36a	10	0.44	1	1623.76	48a	10	9.89	10	0.90	62	10	48.86	10	2.73
11c	10	0.11	10	0.74	25	10	0.04	10	0.15	36b	10	0.46	3	1263.79	48b	10	4.88	10	0.15	63	10	49.06	10	2.15
12	10	0.26	10	0.72	26	10	0.07	10	0.03	37	2	0.07	2	1440.50	49a	10	0.49	1	1623.88	**Tot: 79**	**733**	**17.31**	**649**	**334.14**

$\boldsymbol{x}_{\mathsf{infer}}$. Then, we generate $\boldsymbol{x}_{\mathsf{train}}$ again by random sampling. If any sample, when used for training, could induce model weights $\boldsymbol{w}$ that cause a numerical failure when using $\boldsymbol{x}_{\mathsf{infer}}$ as the inference input, we keep that sample as $\boldsymbol{x}_{\mathsf{train}}$ and terminate. If and only if both $\boldsymbol{x}_{\mathsf{infer}}$ and $\boldsymbol{x}_{\mathsf{train}}$ are found, we count this run as a "success" one for "Random".

For each defect, due to the randomness of the model's initial weights, we repeat both RANUM and "Random" for 10 runs. Both approaches use the same set of random seeds.

Experiment Results

Results are in Table 2.6. We observe that RANUM succeeds in $733/(79 \times 10) = 92.78\%$ runs and the baseline "Random" succeeds in $649/(79 \times 10) = 82.15\%$ runs. Moreover, RANUM spends only 17.31 s time on average per run, which is a 19.30X speedup compared to "Random". We also observe that RANUM is more reliable across repeated runs. There are only 6 cases with unsuccessful repeated runs in RANUM, but there are 19 such cases in "Random". Hence, RANUM is substantially more effective, efficient, and reliable for generating system tests than the baseline.

We study all six defects where RANUM may fail and have the following findings. (1) For four defects (Case IDs 1, 15, 37, and 38), the architecture is challenging for gradient-based optimization, e.g., due to the `Min/Max/Softmax` operators that provide little or no gradient information. We leave it as future work to solve these cases, likely in need of dynamically detecting operators with vanishing gradients and reconstructing the gradient flow. (2) Two defects (Case IDs 28a and 28b) correspond to those caused

by `Div` operators where only a close-to-zero divisor can trigger a numerical failure. Hence, for operators with narrow invalid ranges, RANUM may fail to generate failure-exhibiting system tests.

Ablation Study RANUM uses the two-step test generation technique to produce failure-exhibiting system tests: it first generates failure-exhibiting unit tests with gradient back-propagation, then generates failure-exhibiting training examples via the extended DLG attack. To isolate the impact of RANUM at each step, we replace either step with random sampling: "Random + RANUM", which first generates failure-exhibiting unit tests via random sampling, then generates training example via RANUM; "RANUM + Random", which first generates failure-exhibiting unit tests via RANUM, then generates training example via random sampling. We find that "RANUM + Random" takes 9.38X running time than RANUM and fails for 68 runs (RANUM only fails for 57 runs); and "Random + RANUM" fails for 113 runs (roughly 2X failed runs compared to RANUM). This study implies that RANUM's technique helps to improve effectiveness and efficiency at both steps of failure-exhibiting system test generation compared to the pure random baseline. The improvement for the first step is mainly from the effectiveness perspective, and the improvement for the second step is mainly from the efficiency perspective.

Fix Suggestion: RANUM

We compare RANUM with baseline approaches (constructed by leaving our novel techniques out of RANUM) and developers' fixes on fix suggestion.

Experiment Setups

Datasets We conduct the evaluation on the same benchmarks used in feasibility confirmation.

Evaluation Protocol We compare RANUM with fixes generated by baseline approaches and developers' fixes.

RANUM is the first approach for this task, and we propose two baseline approaches to compare with. (1) RANUM-E: this approach changes the abstraction domain of RANUM from interval with tensor partitioning to standard interval. To some degree,

RANUM-E represents the methodology of conventional static analysis tools that use standard interval domain for abstraction and search of effective fixes. (2) GD: this approach uses standard gradient descent for optimization instead of the abstraction optimization technique in RANUM.

We evaluate whether each approach can generate fixes that eliminate *all* numerical defects for the DNN architecture under analysis given imposing locations. We consider three types of locations: on both weight and input nodes, on only weight nodes, and on only input nodes. In practice, model providers can impose fixes on weight nodes by clipping weights after a model is trained; and users can impose fixes on input nodes by clipping their inputs before loading them into the model. Since all approaches are deterministic, for each case we run only once. We say that the fix eliminates all numerical defects if and only if (1) the RANUM static analysis framework cannot detect any defects from the fixed architecture; and (2) 1,000 random samples cannot trigger any numerical failures after imposing the fix.

We conduct an empirical study to compare the fixes generated by RANUM and by the developers. We manually locate GitHub repositories from which the GRIST benchmarks are constructed. Among the 79 cases, we find the repositories for 53 cases on GitHub and we study these cases. We locate the developers' fixes of the numerical defects by looking at issues and follow-up pull requests. Since RANUM suggests different fixes for different imposing locations, for each case we first determine the imposing locations from the developer's fix, and then compare with RANUM's fix for these locations.

RANUM fixes are on the computational graph and developers' fixes are in the source code, so we determine to conduct code-centered comparison: RANUM fixes are considered feasible only when the fixes can be easily implemented by code (within 10 lines of code) given that developers' fixes are typically small, usually in 3-5 lines of code. In particular, our comparison is based on two criteria: (1) which fix is sound on any valid input; (2) if both are sound, which fix hurts less to model performance and utility (based on the span of imposed precondition, the larger span the less hurt). Two authors independently classify the comparison results for each case and discuss the results to reach a consensus.

Table 2.7: Results of fix suggestion under three imposing location specifications with RANUM and two baselines (RANUM-E and GD). # is the number of fixes found. "Time (s)" is the total running time for all 79 cases.

Imposing Locations	RANUM		RANUM-E		GD	
	#	Time (s)	#	Time (s)	#	Time (s)
Weight + Input	**79**	**54.23**	78	540.13	57	188.63
Weight	**72**	**58.47**	71	581.86	43	219.28
Input	**37**	**924.74**	**37**	3977.30	29	952.19

Experiment Results

Comparison between RANUM and Baselines We report the statistics, including the number of successful cases among all the 79 cases and the total running time, in Table 2.7. From the table, we observe that on all the three imposing location settings, RANUM always succeeds in most cases and spends much less time.

For example, when fixes can be imposed on both weights and input nodes, RANUM succeeds on *all* cases with a total running time 54.23 s. In contrast, RANUM-E requires $> 10\times$ time, and GD succeeds in only 72.15% cases. Hence, RANUM is substantially more effective and efficient for suggesting fixes compared to baseline approaches.

Since RANUM is based on iterative refinement, we study the number of iterations needed for finding the fix. When fixes can be imposed on both weight and input nodes, where RANUM succeeds on all the 79 cases, the average number of iterations is 29.80, the standard deviation is 14.33, the maximum is 53, and the minimum is 2. Hence, when RANUM can find the fix, the number of iterations is small, coinciding with the small total running time 54.23 s.

The two baseline approaches can be viewed as ablated versions of RANUM. Comparing RANUM and GD, we conclude that the technique of abstraction optimization substantially improves the effectiveness and also improves the efficiency. Comparing RANUM and RANUM-E, we conclude that the interval abstraction with tensor partitioning as the abstraction domain substantially improves the efficiency and also improves the effectiveness.

From Table 2.7, it is much easier to find the fix when imposing locations are weight nodes compared to input nodes. Since model providers can impose fixes on weights and users impose on inputs, this finding implies that fixing numerical defects on the providers' side may be more effective than on the users' side.

Comparison between RANUM and Developers' Fixes We categorize the comparison results with manual fixes as below.

1. (*30 cases*) *Better than developers' fixes or no available developer's fix.* Developers either propose no fixes or use heuristic fixes, such as reducing the learning rate or using the mean value to reduce the variance. These fixes may work in practice but are unsound, i.e., cannot rigorously guarantee the elimination of the numerical defect for any training or inference data. In contrast, RANUM generates better fixes since these fixes rigorously eliminate the defect.

2. (*7 cases*) *Equivalent to developers' fixes.* Developers and RANUM suggest equivalent or highly similar fixes.

3. (*13 cases*) *No need to fix.* For these cases, there is no need to fix the numerical defect in the given architecture. There are mainly three reasons. (1) The DNN is used in the whole project with fixed weights or test inputs. As a result, although the architecture contains defects, no system failure can be caused. (2) The architecture is injected a defect as a test case for automatic tools, such as a test architecture in the `TensorFuzz` (Odena et al., 2019) repository. (3) The defect can be hardly exposed in practice. For example, the defect is in a `Div` operator where the divisor needs to be very close to zero to trigger a divide-by-zero failure, but such situation hardly happens in practice since the divisor is randomly initialized.

4. (*3 cases*) *Inferior than developers' fixes or RANUM-generated fixes are impractical.* In two cases, RANUM-generated fixes are inferior to developers' fixes. Human developers observe that the defective operator is `Log`, and its input is non-negative. So they propose to add 10^{-6} to the input of `Log` as the fix. In contrast, RANUM can generate only a clipping-based fix, e.g., clipping the input if it is less than 10^{-6}. When the input is small, RANUM's fix interrupts the gradient flow from output to input while the human's fix maintains it. As a result, the human's fix does less hurt to the model's trainability and is better than RANUM's fix. In another case, the RANUM-generated fix imposes a small span for some model weights (less than 0.1 for each component of that weight node). Such a small weight span strongly limits the model's expressiveness and utility. We leave it as the future work to solve these limitations.

From the comparison results, we can conclude that for the 40 cases where numerical defects are needed to be fixed (excluding case C), RANUM suggests equivalent or better fixes than human developers in 37 cases. Therefore, RANUM is comparably effective as human developers in terms of suggesting numerical-defect fixes, and is much more efficient since RANUM is an automatic approach.

We discuss two practical questions for RANUM users. (1) *Does RANUM hurt model utility, e.g., inference accuracy?* If no training or test data ever exposes a numerical defect, RANUM does not confirm a defect and hence no fix is generated and there is no hurt to the utility. If RANUM confirms numerical defects, whether the fix affects the utility depends on the precondition-imposing locations. If imposing locations can be freely selected, RANUM tends to impose the fix right before the vulnerable operator, and hence the fix does not reduce inference performance. The reason is that the fix changes (by clipping) the input only when the input falls in the invalid range of the vulnerable operator. In practice, if the imposing locations cannot be freely selected and need to follow developers' requirements, our preceding empirical study shows that, in only 3 out of 40 cases, compared with developers' fixes, our fixes incur larger hurt to the inference or training performance of the architecture. (2) *Should we always apply RANUM to fix any architecture?* We can always apply RANUM to fix any architecture since RANUM fixes do not visibly alter the utility in most cases. Nonetheless, in deployment, we recommend first using RANUM to confirm defect feasibility. If there is no such failure-exhibiting system test, we may not need to fix the architecture; otherwise, we use RANUM to generate fixes.

2.5 Related Work

Understanding and Detecting Defects in DNNs Discovering and mitigating defects and failures in DNN based systems is an important research topic (Zhang et al., 2018b; Pham et al., 2020). Following the taxonomy in previous work (Humbatova et al., 2020), DNN defects are at four levels from bottom to top. (1) Platform-level defects. Defects can exist in real-world deep learning compilers and libraries. Approaches exist for understanding, detecting, and testing against these defects (Wang et al., 2020c; Liu et al., 2023). (2) Architecture-level defects. *Our work focuses on numerical defects, being one type of architecture-level defects.* Automatic detection and localization approaches (Wardat et al., 2021; Liu et al., 2021) exist for other architecture-level defects such as suboptimal

structure, activation function, and initialization and shape mismatch (Hattori et al., 2020; Dolby et al., 2018). (3) Model-level defects. Once a model is trained, its defects can be viewed as violations of desired properties as discussed by Zhang et al. (2019b). Some example defects are correctness (Tizpaz-Niari et al., 2020; Guerriero et al., 2021), robustness (Wang et al., 2021a), and fairness (Zhang et al., 2020b) defects. (4) Interface-level defects. DNN-based systems, when deployed as services, expose interaction interfaces to users where defects may exist, as shown by empirical studies on real-world systems (Wan et al., 2021).

Testing and Debugging for DNNs A rich body of work exists for testing and debugging DNN defects (Zhang et al., 2019b). Some representatives are DeepXplore Pei et al. (2017) and DeepGauge Ma et al. (2018). Recent work enables automatic model debugging and repair via consistency checking (Xiao et al., 2021), log checking (Zhang et al., 2021a), spectrum analysis (Qi et al., 2021b), data slicing (Liu et al., 2022b), or analyzer-guided synthesis (Sotoudeh and Thakur, 2021). Such previous work focuses on deep learning models and does not detect numerical bugs before training.

Verification for DNN Another solution for eliminating DNN defects is to rigorously guarantee the non-existence of defects via verification (Li et al., 2023a; Albarghouthi, 2021). There are two lines of verification of neural network robustness: complete verification (Wang et al., 2021b; Katz et al., 2019) and incomplete verification (Xu et al., 2020; Singh et al., 2019). The complete verifiers either find an adversarial example or generate proof that all inputs in the given perturbation space will be correctly classified. Compared to the complete verifiers, the incomplete ones will abstain from predicting if they cannot prove the correctness of the prediction because their techniques will introduce over-approximation. The complete approaches do not have over-approximation issues but require expensive verification algorithms such as branch and bound. DEBAR and RANUM use incomplete verifiers for achieving high scalability.

Recently, the incomplete verifiers customized for DNNs are emerging, mainly focusing on proposing tighter abstractions (Zhang et al., 2018a; Singh et al., 2019) or incorporating abstractions into training to improve robustness (Gowal et al., 2019; Mirman et al., 2018). Besides robustness, incomplete verification has also been applied to rigorously bound model difference (Paulsen et al., 2020b,a). DEBAR is a static analysis tool customized for numerical-defect detection and fixing. RANUM also

includes a such static analysis tool in its framework.

Detecting and Exposing Numerical Defects in DNNs Despite the widespread existence of numerical defects in real-world DNN-based systems (Zhang et al., 2018b), only a few automatic approaches exist for detecting and exposing these defects. To the best of our knowledge, GRIST (Yan et al., 2021) is the only approach for detecting and exposing these defects besides DEBAR and RANUM. Yan et al. (2021) propose a gradient back-propagation approach, GRIST, for generating failure-exhibiting *unit test* to expose numerical defects. Compared to DEBAR, GRIST is a testing approach, which may not find all numerical bugs due to the unsoundness of software testing. Compared to RANUM, GRIST generates unit tests, whose existence may not be sufficient for confirming the feasibility of numerical defect, since the model weights contained in the unit test may not be feasible after training. As a result, RANUM takes a step further by generating failure-exhibiting *system test* which also contains training data that lead to the failure-exhibiting weights.

Array Analysis in Imperative Programs DEBAR is inspired by the existing approaches of abstract interpretation for analyzing arrays, in particular, array partitioning (Gopan et al., 2005). Compared with the existing approaches of array partitioning, we are the first to generalize them from arrays to tensors, employ an affine relation analysis for capturing affine equality relations among partitions, and design abstract tensor operators for deep learning architectures such as the abstract operator for ReLU.

2.6 Future Work

In this chapter, we have presented two automatic approaches named DEBAR and RANUM for reliability assurance of DNNs against numerical defects. DEBAR is the first tool to support the detection of potential numerical defects. In contrast, the follow-up work, RANUM, supports the detection of potential numerical defects, the confirmation of potential-defect feasibility, and the suggestion of defect fixes. RANUM includes multiple novel extensions and optimizations upon existing tools and introduces three novel techniques. Our extensive evaluation on real-world DNN architectures has demonstrated the high effectiveness and efficiency of RANUM compared to both state-of-the-art approaches and developers' fixes.

However, there are several areas for future improvements to DEBAR and the potential-defect detection part of RANUM. First, the detection results are reported on the nodes in computational graphs. However, developers need to manually map the defect nodes to the defect statements in the code. Future work should explore the automatic mapping from computational graph nodes to program statements.

Second, both approaches require developers to report all possible ranges of the inputs and weights. Future work should explore the automation of analyzing the used dataset, data preprocessing, and neural weight normalization code to obtain the ranges automatically.

Third, the ranges of the inputs and weights serve as loop invariants in the context of neural network training. Future directions should explore how to automatically extract these loop invariants instead of relying on developers to report them. Addressing these challenges will further enhance the usability and practicality of DEBAR and RANUM, making them even more valuable tools for ensuring the reliability of DNNs against numerical defects.

Fourth, in addition to numerical bugs that cause values like `NaN` or `Inf`, numerical errors caused by floating-point computation are another type of bug to consider, especially nowadays when large language models frequently use low-precision floating-point quantized versions of models for training and inference. Extending the scope of numerical bugs to include numerical errors is a promising future direction.

3 VERIFYING THE ROBUSTNESS OF NLP MODELS

Deep neural networks have proven incredibly powerful in a huge range of machine-learning tasks. However, deep neural networks are highly sensitive to small input perturbations that cause the network's accuracy to plummet (Carlini and Wagner, 2017; Szegedy et al., 2014). In the context of natural language processing, these *adversarial examples* come in the form of spelling mistakes, use of synonyms, etc.—essentially, meaning-preserving transformations that cause the network to change its prediction (Ebrahimi et al., 2018; Zhang et al., 2019a).

In this chapter, we ask the following two questions:

(*1*) *Can we train models that are robust against rich perturbation spaces over strings?*

(*2*) *Can we develop a certified defense to arbitrary string transformations that applies to recursive neural networks?*

We propose A3T [1] as an answer to the first question and ARC [2] as an answer to the second question.

Overview of A3T

The practical challenge in answering this question is computing the worst-case loss. This is because the perturbation space can be enormous and therefore impractical to enumerate. This is particularly true for NLP tasks, where the perturbation space should contain inputs that are semantically equivalent to the original input—e.g., variations with typos or words replaced by synonyms. Therefore, we need to *approximate* the adversarial loss. There are two such classes of approximation techniques as shown in Figure 3.1:

Augmentation The first class of techniques computes a *lower bound* on the adversarial loss by exploring a finite number of points in the perturbation space. This is usually done by applying a gradient-based attack, like HotFlip (Ebrahimi et al., 2018) for natural-language tasks. We call this class of techniques *augmentation*-based, as they essentially search for a perturbed sample with which to augment the training set.

[1] `https://github.com/ForeverZyh/A3T`
[2] `https://github.com/ForeverZyh/certified_lstms`

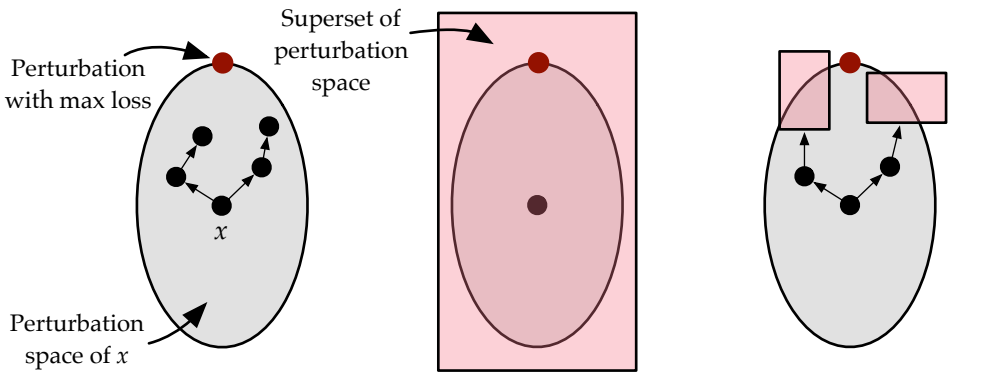
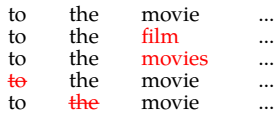


Figure 3.1: Illustration of augmentation, abstraction, and A3T

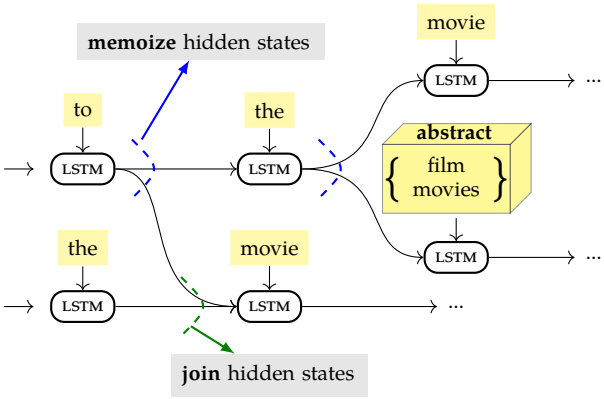
Abstraction The second class of techniques computes an *upper bound* on the adversarial loss by over-approximating, or *abstracting*, the perturbation space into a set of symbolic constraints that can be efficiently propagated through the network. For example, the *interval* abstraction has been used in numerous works (Mirman et al., 2018; Gowal et al., 2019; Huang et al., 2019). We call this class of techniques *abstraction*-based.

Both classes of techniques can produce suboptimal results: augmentation can severely underapproximate the worst-case loss and abstraction can severely overapproximate the loss. Particularly, we observe that the two techniques have complementary utility, working well on some perturbation spaces but not others—for example, Huang et al. (2019) have shown that abstraction works better for token substitutions, while augmentation-based techniques like HotFlip (Ebrahimi et al., 2018) and MHA (Zhang et al., 2019a) are general—e.g., apply to token deletions and insertions.

We propose *augmented abstract adversarial training* (A3T), an adversarial training technique that combines the strengths of augmentation and abstraction techniques. The key idea underlying A3T is to decompose the perturbation space into two subsets, one that can be explored using augmentation and one that can be abstracted—e.g., using augmentation to explore word duplication typos and abstraction to explore replacing words with synonyms. From an algorithmic perspective, our computation of adversarial loss switches from a concrete, e.g., gradient-based, search through the perturbation space to a symbolic search. As such, for every training sample (x, y), our



(a) Enumeration of all strings



(b) ARC: Abstract Recursive Certification

Figure 3.2: An illustration of our approach.

technique may end up with a lower bound *or* an upper bound on its adversarial loss.

Overview of ARC

Certifying robustness involves proving that a network's prediction is the same no matter how a given input string is perturbed. We assume that the *perturbation space* is defined as a program describing a set of possible string transformations—e.g., *if you see the word "movie", replace it with "film" or "movies"*. Such transformations can succinctly define a perturbation space that is exponentially large in the length of the input; so, certification by enumerating the perturbation space is generally impractical.

We present ARC (Abstract Recursive Certification), an approach for *certifying* robustness to programmatically defined perturbation spaces. ARC can be used within an adversarial training loop to *train robust models*. We illustrate the key ideas behind ARC through a simple example. Consider the (partial) input sentence *to the movie...*, and say we are using an LSTM for prediction. Say we have two string transformations: (**T1**) If you see the word *movie*, you can replace it with *film* or *movies*. (**T2**) If you see the word *the* or *to*, you can delete it. ARC avoids enumerating the large perturbation space (Figure 3.2(a)) using two key insights.

Memoization: ARC exploits the recursive structure of LSTM networks, and their extensions (BiLSTMs, TreeLSTMs), to avoid recomputing intermediate hidden states.

ARC *memoizes* hidden states of prefixes shared across multiple sequences in the perturbation space. For example, the two sentences *to the movie...* and *to the film....* share the prefix *to the*, and therefore we memoize the hidden state after the word *the*, as illustrated in Figure 3.2(b) with dashed blue lines. The critical challenge is characterizing which strings share prefixes without having to explicitly explore the perturbation space.

Abstraction: ARC uses *abstract interpretation* (Cousot and Cousot, 1977a) to symbolically represent sets of perturbed strings, avoiding a combinatorial explosion. Specifically, ARC represents a set of strings as a hyperrectangle in a $\mathbb{R}^n$ and propagates the hyperrectangle through the network using interval arithmetic (Gowal et al., 2019). This idea is illustrated in Figure 3.2(b), where the words *film* and *movies* are represented as a hyperrectangle. By *joining* hidden states of different sentences (a common idea in program analysis), ARC can perform certification efficiently.

Memoization and abstraction enable ARC to efficiently certify robustness to very large perturbation spaces.

3.1 Robustness Problem and Preliminaries

We consider a classification setting with a neural network F_θ with parameters θ, trained on samples from domain $\mathcal{X}$ and labels from $\mathcal{Y}$. The domain $\mathcal{X}$ is a set of strings over a finite set of *symbols* Σ (e.g., English words or characters), i.e., $\mathcal{X} = \Sigma^*$. We use $\mathbf{x} \in \Sigma^*$ to denote a string; $x_i \in \Sigma$ to denote the ith element of the string; $\mathbf{x}_{i:j}$ to denote the substring $x_i, \ldots, x_j$; ϵ to denote the empty string; and $\textsc{len}_{\mathbf{x}}$ to denote the length of the string.

Robustness to string transformations A *perturbation space* S is a function in $\Sigma^* \to 2^{\Sigma^*}$, i.e., S takes a string $\mathbf{x}$ and returns a set of possible perturbed strings obtained by modifying $\mathbf{x}$. Intuitively, $S(\mathbf{x})$ denotes a set of strings that are semantically *similar* to $\mathbf{x}$ and therefore should receive the same prediction. We assume $\mathbf{x} \in S(\mathbf{x})$.

Given string $\mathbf{x}$ with label y and a perturbation space S, We say that a neural network F_θ is *S-robust* on $(\mathbf{x}, y)$ iff

$$\forall\, \mathbf{z} \in S(\mathbf{x}).\, F_\theta(\mathbf{z}) = y \tag{3.1}$$

One goal of ARC is to *certify*, or prove, S-robustness (Eq 3.1) of the neural network

for a pair $(\mathbf{x}, y)$. Given a certification approach, we can then use it within an adversarial training loop to yield certifiably robust networks.

Robustness certification We will certify S-robustness by solving an *adversarial loss* objective:

$$\max_{\mathbf{z} \in S(\mathbf{x})} L_\theta(\mathbf{z}, y) \tag{3.2}$$

where we assume that the loss function L_θ is < 0 when $F_\theta(\mathbf{z}) = y$ and $\geqslant 0$ when $F_\theta(\mathbf{z}) \neq y$. Therefore, if we can show that the solution to the above problem is < 0, then we have a certificate of S-robustness.

Certified training If we have a procedure to compute adversarial loss, we can use it for *adversarial training* by solving the following *robust optimization* objective (Madry et al., 2018), where $\mathcal{D}$ is the data distribution:

$$\arg\min_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x}, y) \sim \mathcal{D}} \left[\max_{\mathbf{z} \in S(\mathbf{x})} L_\theta(\mathbf{z}, y) \right] \tag{3.3}$$

At A3T, our proposal decomposes the perturbation space into two subsets. The first subset can be explored through augmentation techniques, while the second requires abstraction. For instance, we can use augmentation to explore word duplication typos, while abstraction enables us to explore replacing words with synonyms. Meanwhile, in ARC, we propose constructing the perturbation loss using the abstract loss defined in Eq 3.2. By optimizing directly over the abstract loss, ARC provides an efficient means of over-approximating it while ensuring the precision of the abstraction.

Programmable Perturbation Spaces

In our problem definition, we assumed an arbitrary perturbation space S. We propose to define S *programmatically* as a set of string transformations. The language is very flexible, allowing the definition of a rich class of transformations as *match* and *replace* functions.

Single transformations A string transformation T is a pair (φ, f), where $\varphi : \Sigma^s \to \{0, 1\}$ is the *match* function, a Boolean function that specifies the substrings (of length

s) to which the transformation can be applied; and $f : \Sigma^s \to 2^{\Sigma^t}$ is the *replace* function, which specifies how the substrings matched by φ can be replaced (with strings of length t). We will call s and t the size of the *domain* and *range* of transformation T, respectively.

Example 3.1. *In all examples, the set of symbols Σ is English words. So, strings are English sentences. Let T_{del} be a string transformation that deletes the stop words "*to*" and "*the*". Formally, $T_{del} = (\varphi_{del}, f_{del})$, where $\varphi_{del} : \Sigma^1 \mapsto \{0, 1\}$ and $f_{del} : \Sigma^1 \to 2^{\Sigma^0}$ are:*

$$\varphi_{del}(x) = \begin{cases} 1, & x \in \{\text{“to”, “the”}\} \\ 0, & \textit{otherwise} \end{cases}, \quad f_{del}(x) = \{\epsilon\},$$

*Let T_{sub} be a transformation substituting the word "*movie*" with "*movies*" or "*film*". Formally, $T_{sub} = (\varphi_{sub}, f_{sub})$, where $\varphi_{sub} : \Sigma^1 \mapsto \{0, 1\}$ and $f_{sub} : \Sigma^1 \to 2^{\Sigma^1}$ are:*

$$\varphi_{sub}(x) = \begin{cases} 1, & x = \text{“movie”} \\ 0, & \textit{otherwise} \end{cases}, f_{sub}(x) = \begin{Bmatrix} \text{“film”}, \\ \text{“movies”} \end{Bmatrix}$$

Defining perturbation spaces We can compose different string transformation to construct perturbation space S:

$$S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}, \tag{3.4}$$

where each T_i denotes a string transformation that can be applied *up to* $\delta_i \in \mathbb{N}$ times.

Multiple transformations As discussed above, a specification S in our language is a set of transformations $\{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$ where each T_i is a pair (φ_i, f_i). We define the semantics of a specification $S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$ (such that $T_i = (\varphi_i, f_i)$) as follows. Given a string $\mathbf{x} = x_1 \ldots x_m$, a string $\mathbf{y}$ is in the perturbations space $S(\mathbf{x})$ if:

1. there exists matches $\langle (l_1, r_1), j_1 \rangle \ldots \langle (l_k, r_k), j_k \rangle$ (we assume that matches are sorted in ascending order of l_i) such that for every $i \leqslant k$ we have that (l_i, r_i) is a valid match of φ_{j_i} in $\mathbf{x}$;

2. the matches are not overlapping: for every two distinct i_1 and i_2, $r_{i_1} < l_{i_2}$ or $r_{i_2} < l_{i_1}$;

3. the matches respect the δ constraints: for every $j' \leqslant n$, $|\{\langle (l_i, r_i), j_i\rangle \mid j_i = j'\}| \leqslant \delta_{j'}$.

4. the string $\mathbf{y}$ is the result of applying an appropriate transformation to each match: if for every $i \leqslant k$ we have $\mathbf{s}_i \in f_{j_i}(x_{l_i} \dots x_{r_i})$, then

$$\mathbf{y} = x_1 \dots x_{l_1-1}\, \mathbf{s}_1\, x_{r_1+1} \dots x_{l_k-1}\, \mathbf{s}_k\, x_{r_k+1} \dots x_m.$$

Intuitively, a string $\mathbf{z}$ is in the perturbation space $S(\mathbf{x})$ if it can be obtained by (1) finding a set σ of *non-overlapping* substrings of $\mathbf{x}$ that match the various predicates φ_i and such that at most δ_i substrings in σ are matches of φ_i, and (2) replacing each substring $\mathbf{x}' \in \sigma$ matched by φ_i with a string in $f_i(\mathbf{x}')$. The complexity of the formalization is due to the requirement that matched substrings should not overlap—this requirement guarantees that each character in the input is only involved in a single transformation and will be useful when formalizing our abstract training approaches A3T and ARC. We illustrate with an example.

Example 3.2. *Let* $S = \{(T_{del}, 1), (T_{sub}, 1)\}$ *be a perturbation space that applies* T_{del} *and* T_{sub} *to the given input sequence up to once each. If* $\mathbf{x} =$"to the movie"*, a subset of the perturbation space* $S(\mathbf{x})$ *is shown in Figure 3.2(a).*

Decomposition $S = \{(T_i, \delta_i)\}_i$ can be decomposed into $\prod(\delta_i + 1)$ subset perturbation spaces by considering all smaller combinations of δ_i. We denote the decomposition of perturbation space S as $\textsc{dec}_S$, and exemplify below:

Example 3.3. $S_1 = \{(T_{del}, 2)\}$ *can be decomposed to a set of three perturbation spaces* $\textsc{dec}_{S_1} = \{\varnothing, \{(T_{del}, 1)\}, \{(T_{del}, 2)\}\}$*, while* $S_2 = \{(T_{del}, 1), (T_{sub}, 1)\}$ *can be decomposed to a set of four perturbation spaces*

$$\textsc{dec}_{S_2} = \{\varnothing, \{(T_{del}, 1)\}, \{(T_{sub}, 1)\}, \{(T_{del}, 1), (T_{sub}, 1)\}\}$$

where $\varnothing$ *is the perturbation space with no transformations, i.e., if* $S = \varnothing$*, then* $S(\mathbf{x}) = \{\mathbf{x}\}$ *for any string* $\mathbf{x}$*.*

We use notation $S_{k\downarrow}$ to denote S after reducing δ_k by 1; therefore, $S_{k\downarrow} \in \textsc{dec}_S$.

3.2 Augmented Abstract Adversarial Training (A3T)

In this section, we describe our abstract training technique, A3T, which combines augmentation and abstraction.

Recall the adversarial training objective function, Eq. (3.3). The difficulty in solving this objective is the inner maximization objective: $\max_{\mathbf{z} \in S(\mathbf{x})} L_\theta(\mathbf{z}, y)$, where the perturbation space $S(\mathbf{x})$ can be intractably large to efficiently enumerate, and we therefore have to resort to approximation. We begin by describing two approximation techniques and then discuss how our approach combines and extends them.

Augmentation (search-based) techniques We call the first class of techniques *augmentation* techniques, since they search for a worst-case sample in the perturbation space $S(\mathbf{x})$ with which to augment the dataset. The naïve way is to simply enumerate all points in $S(\mathbf{x})$—our specifications induce a finite perturbation space, by construction. Unfortunately, this can drastically slow down the training. For example, suppose T defines a transformation that swaps two adjacent characters. On a string of length N, the specification $(T, 2)$ results in $O(N^2)$ transformations.

An efficient alternative, HotFlip, was proposed by Ebrahimi et al. (2018). HotFlip efficiently encodes a transformation T as an operation over the embedding vector and *approximates* the worst-case loss using a single forward and backward pass through the network. To search through a set of transformations, HotFlip employs a beam search of some size k to get the top-k perturbed samples. This technique yields a point in $S(\mathbf{x})$ that may not have the worst-case loss. Alternatives like MHA (Zhang et al., 2019a) can also be used as augmentation techniques.

Abstraction techniques Abstraction techniques compute an over-approximation of the perturbation space, as a symbolic set of constraints. This set of constraints is then propagated through the network, resulting in an upper bound on the worst-case loss. Specifically, given a transformation T, we define a corresponding abstract transformation $\widehat{T}$ such that for all $\mathbf{x}$, the constraint $T(\mathbf{x}) \subseteq \widehat{T}(\mathbf{x})$ holds.

The abstraction used in the A3T paper builds upon the work of Huang et al. (2019), which employs an *interval domain* to define $\widehat{T}(\mathbf{x})$, meaning that $\widehat{T}(\mathbf{x})$ is a conjunction of constraints on each character. The A3T paper generalizes their approach; please refer to the paper for more details. In Section 3.3, we will illustrate another abstraction introduced by ARC, which can also be used as the abstraction technique in A3T. We

Algorithm 2 A3T

Input: $S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$ and point $(\mathbf{x}, y)$
Output: worst-case loss
Split S into S_{aug} and S_{abs} and return

$$\max_{\mathbf{z} \in augment_k(S_{aug}, \mathbf{x})} L_\theta(\widehat{\mathbf{z}}, y) \quad s.t.\ \widehat{\mathbf{z}} = abstract(S_{abs}, \mathbf{z})$$

also provide the results of combining A3T and ARC in Section 3.4. For now, we assume that we can efficiently overapproximate the worst-case loss for $\widehat{T}(\mathbf{x})$ by propagating it through the network. This assumption allows us to discuss the A3T approach and its applications.

A3T: A High-Level View The key idea of A3T is to decompose a specification S into two sets of transformations, one containing transformations that can be effectively explored with augmentation and one containing transformations that can be precisely abstracted.

Algorithm 2 shows how A3T works. First, we decompose the specification S into two subsets of transformations, resulting in two specifications, S_{aug} and S_{abs}. For S_{aug}, we apply an augmentation technique, e.g., HotFlip or MHA, to come up with a list of top-k perturbed samples in the set $S_{aug}(\mathbf{x})$—this is denoted as the set $augment_k(S_{aug}, \mathbf{x})$.

Then, for each point $\mathbf{z}$ in the top-k results, we compute an abstraction $abstract(S_{abs}, \mathbf{z})$, which is a set of constraints over-approximating the set of points in $S_{abs}(\mathbf{z})$. Recall our overview in Fig. 3.1 for a visual depiction of this process.

Finally, we return the worst-case loss.

3.3 Abstract Recursive Certification (ARC)

In this section, we present our technique for proving S-robustness of an LSTM on $(\mathbf{x}, y)$. Formally, we do this by computing the adversarial loss, $\max_{\mathbf{z} \in S(\mathbf{x})} L_\theta(\mathbf{z}, y)$. Recall that if the solution is < 0, then we have proven S-robustness. To solve adversarial loss optimally, we effectively need to evaluate the LSTM on all of $S(\mathbf{x})$ and collect all final states:

$$F = \{\textsc{lstm}(\mathbf{z}, h_0) \mid \mathbf{z} \in S(\mathbf{x})\} \tag{3.5}$$

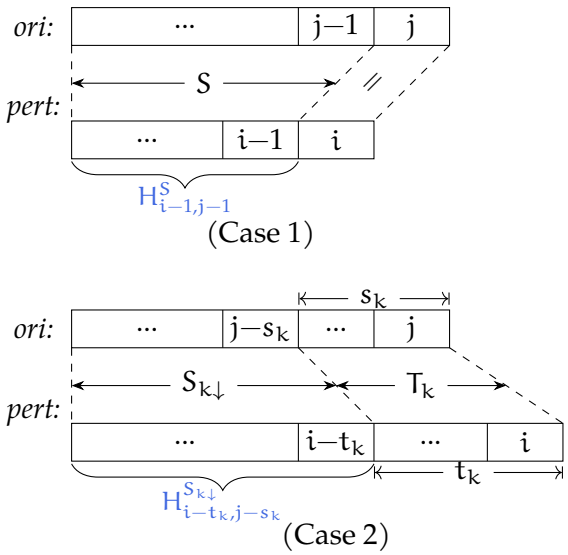


Figure 3.3: Illustration of two cases of Equation (3.8).

Computing F precisely is challenging, as $S(\mathbf{x})$ may be prohibitively large. Therefore, we propose to compute a superset of F, which we will call $\widehat{F}$. This superset will therefore yield an upper bound on the adversarial loss. We prove S-robustness if the upper bound is < 0.

To compute $\widehat{F}$, we present two key ideas that go hand-in-hand: In the first subsection, we observe that strings in the perturbation space share common prefixes, and therefore we can *memoize* hidden states to reduce the number of evaluations of LSTM cells—a form of dynamic programming. We carefully derive the set of final states F as a system of memoizing equations. The challenge of this derivation is characterizing which strings share common prefixes without explicitly exploring the perturbation space. In the second subsection, we apply abstract interpretation to efficiently and soundly solve the system of memoizing equations, thus computing an overapproximation $\widehat{F} \supseteq F$.

Memoizing Equations of Final States

Tight Perturbation Space Given a perturbation space S, we shall use $S^{=}$ to denote the *tight* perturbation space where each transformation T_j in S is be applied **exactly** δ_j times.

Think of the set of all strings in a perturbation space as a tree, like in Fig. 3.2(b), where strings that share prefixes share LSTM states. We want to characterize a subset $H^{S}_{i,j}$ of LSTM states at the ith layer where the perturbed prefixes have had *all* transformations in a space S applied on the original prefix $\mathbf{x}_{1:j}$.

We formally define $H^S_{i,j}$ as follows:

$$H^S_{i,j} = \{\textsc{lstm}(\mathbf{z}, h_0) \mid \mathbf{z} \in S^{=}(\mathbf{x}_{1:j}) \wedge \textsc{len}_{\mathbf{z}} = i\} \tag{3.6}$$

By definition, the base case $H^{\varnothing}_{0,0} = \{0^d\}$.

Example 3.4. *Let* $\mathbf{x}$ = "to the movie". *Then,* $H^{\{(T_{del},1)\}}_{1,2} = \{\textit{LSTM}(\text{"the"}, h_0), \textit{LSTM}(\text{"to"}, h_0)\}$. *These states result from deleting the first and second words of the prefix "*to the*", respectively. We also have* $H^{\varnothing}_{2,2} = \{\textit{LSTM}(\text{"to the"}, h_0)\}$.

The set of final states of strings in $S^{=}(\mathbf{x})$ is

$$\bigcup_{i \geqslant 0} H^S_{i,\textsc{len}_{\mathbf{x}}}. \tag{3.7}$$

Memoizing equation We now demonstrate how to rewrite Equation (3.6) by explicitly applying the transformations defining the perturbation space S. Notice that each $H^S_{i,j}$ comes from two sets of strings: (1) strings whose suffix (the last character) is not perturbed by any transformations (the first line of Equation (3.8)), and (2) strings whose suffix is perturbed by $T_k = (\varphi_k, f_k)$ (the second line of Equation (3.8)), as illustrated in Figure 3.3. Thus, we derive the final equation and then immediately show an example:

$$\begin{aligned} H^S_{i,j} = {} & \{\textsc{lstm}(x_j, h) \mid h \in H^S_{i-1,j-1}\} \cup \\ & \bigcup_{\substack{1 \leqslant k \leqslant |S| \\ \varphi_k(\mathbf{x}_{a:b})=1}} \{\textsc{lstm}(\mathbf{z}, h) \mid \mathbf{z} \in f_k(\mathbf{x}_{a:b}), h \in H^{S_{k\downarrow}}_{i-t_k,j-s_k}\} \end{aligned} \tag{3.8}$$

where $a{=}j{-}s_k + 1$ and $b{=}j$.

We compute Equation (3.8) in a bottom-up fashion, starting from $H^{\varnothing}_{0,0} = \{0^d\}$ and increasing i, j and considering every possible perturbation space in the decomposition of S, $\textsc{dec}_S$.

Lemma 3.5. *Equation* (3.8) *and Equation* (3.6) *are equivalent.*

Proof. We prove Lemma 3.5 by induction on i, j, and S.

Base case: $H^{\varnothing}_{0,0} = h_0$ is defined by both Equation (3.6) and Equation (3.8).

Inductive step for $H^S_{i,j}$: Suppose the lemma holds for $H^{S'}_{i',j'}$, where $(0 \leqslant i' \leqslant i \wedge 0 \leqslant j' \leqslant j \wedge S' \subseteq S) \wedge (i' \neq i \vee j' \neq j \vee S' \neq S)$. $S' \subseteq S$ denotes that for all $(T_k, \delta'_k) \in S'$, we have $(T_k, \delta_k) \in S$ and $\delta'_k \leqslant \delta_k$.

$H^S_{i,j}$ in Equation (3.6) comes from two cases illustrated in Figure 3.3. These two cases are captured by the first line and the second line in Equation (3.8), respectively. The inductive hypothesis shows that the lemma holds for states $H^S_{i-1,j-1}$ and $H^{S_{k\downarrow}}_{i-t_k,j-s_k}$. Thus, the lemma also holds for $H^S_{i,j}$. □

Example 3.6. *Consider computing $H^{\{(T_{del},1)\}}_{1,2}$. We demonstrate how to derive states from Equation (3.8):*

$$\begin{aligned} H^{\{(T_{del},1)\}}_{1,2} =& \{\textsc{lstm}(\text{“}\textit{the}\text{”}, h) \mid h \in H^{\{(T_{del},1)\}}_{0,1}\} \cup \\ & \{\textsc{lstm}(\mathbf{z}, h) \mid \mathbf{z} \in f_{del}(\text{“}\textit{the}\text{”}), h \in H^{\varnothing}_{1,1}\} \end{aligned} \tag{3.9}$$

Assume $H^{\varnothing}_{1,1} = \{\textsc{lstm}(\text{“to”}, h_0)\}$ and $H^{\{(T_{del},1)\}}_{0,1} = \{h_0\}$ are computed in advance. The first line of Equation (3.9) evaluates to $\{\textsc{lstm}(\text{“the”}, h_0)\}$, which corresponds to deleting the first word of the prefix “to the”. Because $\mathbf{z}$ can only be an empty string, the second line of Equation (3.9) evaluates to $\{\textsc{lstm}(\text{“to”}, h_0)\}$, which corresponds to deleting the second word of “to the”. The dashed green line in Fig. 3.2(b) shows the computation of Equation (3.9).

Defining Final States using Prefixes Finally, we compute the set of final states, F, by considering all perturbation spaces in the decomposition of S.

$$F = \bigcup_{S' \in \textsc{dec}_S} \bigcup_{i \geqslant 0} H^{S'}_{i,\textsc{len}_x} \tag{3.10}$$

Theorem 3.7. *Equation (3.10) is equivalent to Equation (3.5).*

Proof. We can expand Equation (3.5) using the decomposition of the perturbation space as

$$\begin{aligned} & F \\ =& \bigcup_{S' \in \textsc{dec}_S} \{\textsc{lstm}(\mathbf{z}, h_0) \mid \mathbf{z} \in S'^{=}(\mathbf{x})\} \\ =& \bigcup_{S' \in \textsc{dec}_S} \bigcup_{i \geqslant 0} \{\textsc{lstm}(\mathbf{z}, h_0) \mid \mathbf{z} \in S'^{=}(\mathbf{x}) \wedge \textsc{len}_{\mathbf{z}} = i\} \end{aligned} \tag{3.11}$$

Equation (3.11) and Equation (3.10) are equivalent, leading to the equivalence of Equation (3.5) and Equation (3.10). □

Example 3.8. *Let* $S = \{(\mathsf{T}_{\text{del}}, 1), (\mathsf{T}_{\text{sub}}, 1)\}$ *and* $\mathbf{x}$ =*"to the movie"*. F *is the union of four final states,* $\mathsf{H}^{\varnothing}_{3,3}$ (*no transformations*), $\mathsf{H}^{\{(\mathsf{T}_{\text{del}},1)\}}_{2,3}$ (*exactly 1 deletion*), $\mathsf{H}^{\{(\mathsf{T}_{\text{sub}},1)\}}_{3,3}$ (*exactly 1 substitution*), *and* $\mathsf{H}^{\{(\mathsf{T}_{\text{del}},1),(\mathsf{T}_{\text{sub}},1)\}}_{2,3}$ (*exactly 1 deletion and 1 substitution*).

Abstract Memoizing Equations

Memoization avoids recomputing hidden states, but it still incurs a combinatorial explosion. We employ abstract interpretation (Cousot and Cousot, 1977a) to solve the equations efficiently by overapproximating the set F.

Abstract Interpretation The *interval domain*, or *interval bound propagation*, allows us to evaluate a function on an infinite set of inputs represented as a hyperrectangle in $\mathbb{R}^n$.

Interval domain. We define the interval domain over scalars—the extension to vectors is standard. We will use an interval $[l, u] \subset \mathbb{R}$, where $l, u \in \mathbb{R}$ and $l \leqslant u$, to denote the set of all real numbers between l and u, inclusive.

For a finite set $X \subset \mathbb{R}$, the *abstraction* operator gives the tightest interval containing X, as follows: $\alpha(X) = [\min(X), \max(X)]$. Abstraction allows us to compactly represent a large set of strings.

Example 3.9. *Suppose the words* $x_1 =$ "movie" *and* $x_2 =$ "film" *have the 1D embedding* 0.1 *and* 0.15*, respectively. Then,* $\alpha(\{x_1, x_2\}) = [0.1, 0.15]$. *For* n*-dimensional embeddings, we simply compute an abstraction of every dimension, producing a vector of intervals.*

The *join* operation, $\sqcup$, produces the smallest interval containing two intervals: $[l, u] \sqcup [l', u'] = [\min(l, l'), \max(u, u')]$. We will use joins to merge hidden states resulting from different strings in the perturbation space (recall Fig. 3.2(b)).

Example 3.10. *Say we have two sets of 1D LSTM states represented as intervals,* $[1, 2]$ *and* $[10, 12]$. *Then* $[1, 2] \sqcup [10, 12] = [1, 12]$. *Note that* $\sqcup$ *is an overapproximation of* $\cup$*, introducing elements in neither interval.*

Interval transformers. To evaluate a neural network on intervals, we lift neural-network operations to interval arithmetic—*abstract transformers*. For a function g, we use $\widehat{g}$ to denote its abstract transformer. We use the transformers proposed by Jia

et al. (2019). We illustrate transformers for addition and any monotonically increasing function $g : \mathbb{R} \to \mathbb{R}$ (e.g., ReLU, tanh).

$$[l, u] \widehat{+} [l', u'] = [l + l', u + u'], \qquad \widehat{g}([l, u]) = [g(l), g(u)]$$

Note how, for monotonic functions g, the abstract transformer $\widehat{g}$ simply applies g to the lower and upper bounds.

Example 3.11. *When applying to the ReLU function,* $\widehat{\text{relu}}([-1, 2]) = [\text{relu}(-1), \text{relu}(2)] = [0, 2]$.

An abstract transformer $\widehat{g}$ must be *sound*: for any interval $[l, u]$ and $x \in [l, u]$, we have $g(x) \in \widehat{g}([l, u])$. We use $\widehat{\textsc{lstm}}$ to denote an abstract transformer of an LSTM cell. It takes an interval of symbol embeddings and an interval of states. We use the definition of $\widehat{\textsc{lstm}}$ given by Jia et al. (2019).

Abstract Memoizing Equations We now show how to solve Equation (3.8) and Equation (3.10) using abstract interpretation. We do this by rewriting the equations using operations over intervals. Let $\widehat{H}^{\varnothing}_{0,0} = \alpha(\{0^d\})$, then

$$\begin{aligned}
\widehat{H}^{S}_{i,j} &= \widehat{\textsc{lstm}}(\alpha(\{x_j\}), \widehat{H}^{S}_{i-1,j-1}) \sqcup \\
&\qquad \bigsqcup_{\substack{1 \leqslant k \leqslant |S| \\ \varphi_k(\mathbf{x}_{a:b})=1}} \widehat{\textsc{lstm}}(\alpha(f_k(\mathbf{x}_{a:b})), \widehat{H}^{S_{k\downarrow}}_{i-t_k,j-s_k}) \\
\widehat{F} &= \bigsqcup_{S' \in \textsc{dec}_S} \bigsqcup_{i \geqslant 0} \widehat{H}^{S'}_{i,\textsc{len}_{\mathbf{x}}}
\end{aligned}$$

where a and b are the same in Equation (3.8).

The two key ideas are (1) representing sets of possible LSTM inputs abstractly as intervals, using α; and (2) joining intervals of states, using $\sqcup$. These two ideas ensure that we efficiently solve the system of equations, producing an overapproximation $\widehat{F}$.

The above abstract equations give us a compact overapproximation of F that can be computed with a number of steps that is linear in the length of the input. Even though we can have $O(\textsc{len}^2_{\mathbf{x}})$ number of $H^S_{i,j}$ for a given S, only $O(\textsc{len}_{\mathbf{x}})$ number of $H^S_{i,j}$ are non-empty. This property is used in Theorem 3.12 and will be proved in the appendix.

Theorem 3.12. (*Soundness & Complexity*) $F \subseteq \widehat{F}$ *and the number of LSTM cell evaluations needed to compute* $\widehat{F}$ *is* $O(\textsc{len}_{\mathbf{x}} \cdot n \cdot \prod_{i=1}^{n} \delta_i)$.

Proof. We first show that Equation (3.7) is equivalent to

$$\bigcup_{0 \leqslant i \leqslant \textsc{maxlen}_{\mathbf{x}}} H^{S}_{i,\textsc{len}_{\mathbf{x}}} \quad \text{, where}$$
$$\textsc{maxlen}_{\mathbf{x}} = \textsc{len}_{\mathbf{x}} + \sum_{(T_k,\delta_k)\in S} \max(t_k - s_k, 0)\delta_k$$

where $f_k : \Sigma^{s_k} \to 2^{\Sigma^{t_k}}$. As we will prove later, $\textsc{maxlen}_{\mathbf{x}}$ is the upper bound of the length of the perturbed strings. Because t_k, s_k, δ_k are typically small constants, we can regard $\textsc{maxlen}_{\mathbf{x}}$ as a term that is linear in the length of the original string $\textsc{len}_{\mathbf{x}}$, i.e., $\textsc{maxlen}_{\mathbf{x}} = O(\textsc{len}_{\mathbf{x}})$.

Now, we prove that $\textsc{maxlen}_{\mathbf{x}}$ is the upper bound of the length of the perturbed string. The upper bound $\textsc{maxlen}_{\mathbf{x}}$ can be achieved by applying all string transformations that increase the perturbed string's length and not applying any string transformations that decrease the perturbed string's length. Suppose a string transformation $T_k = (f_k, \varphi_k)$, $f_k : \Sigma^{s_k} \to 2^{\Sigma^{t_k}}$ can be applied up to δ_k times, then we can apply it δ_k times to increase the perturbed string's length by $(t_k - s_k)\delta_k$.

The proof of soundness follows immediately from the fact that α, $\sqcup$, and $\widehat{\textsc{lstm}}$ overapproximate their inputs, resulting in an overapproximation F.

The proof of complexity follows the property that the number of non-empty hyperrectangles $\widehat{H}^{S}_{i,j}$ is $O(\textsc{len}_{\mathbf{x}} \cdot \prod_{i=1}^{n} \delta_i)$. This property follows the definition of the string transformations and the tight perturbation space $S^{=}$. $\widehat{H}^{S}_{i,j}$ can be non-empty if and only if

$$i = j + \sum_{(T_k,\delta_k)\in S} (t_k - s_k)\delta_k, \quad \text{where } f_k : \Sigma^{s_k} \to 2^{\Sigma^{t_k}}$$

For each $\widehat{H}^{S}_{i,j}$, we need to enumerate through all transformations, so the complexity is $O(\textsc{len}_{\mathbf{x}} \cdot n \prod_{i=1}^{n} \delta_i)$ in terms of the number of LSTM cell evaluations. The interval bound propagation needs only two forward passes for computing the lower bound and upper bound of the hyperrectangles, so it only contributes constant time to complexity. In all, the total number of LSTM cell evaluations needed is $O(\textsc{len}_{\mathbf{x}} \cdot n \prod_{i=1}^{n} \delta_i)$. □

For practical perturbations spaces (see Section 3.4), the quantity $n \prod_{i=1}^{n} \delta_i$ is typically small and can be considered constant.

Handling the Aggregation of All Hidden States

In addition to the memoization and abstraction of final states, we need to consider LSTM architectures that aggregate all states h_i, e.g., by averaging all states or computing an attention mechanism on them. In such cases, we aim to compute the interval abstraction of each state at the i-th time step, denoted as $\widehat{H}_i$.

It is tempting to compute H_i as

$$H_i = \bigcup_{S' \in \text{DEC}} \bigcup_{0 \leqslant j \leqslant \text{LEN}_{\mathbf{x}}} H^{S'}_{i,j} \tag{3.12}$$

Unfortunately, Equation (3.12) does not contain states in the middle of a string transformation (see next example).

Example 3.13. *Consider the string transformation* T_{swap} *that swaps two adjacent words and suppose* $\mathbf{x} =$ *"to the"*, $S = \{(T_{\text{swap}}, 1)\}$, *then* $H_1 = \{\textsc{lstm}(\text{"to"}, h_0), \textsc{lstm}(\text{"the"}, h_0)\}$. *However, the only non-empty state with* $i = 1$ *is* $H^{\varnothing}_{1,1} = \{\textsc{lstm}(\text{"to"}, h_0)\}$. *The state* $\textsc{lstm}(\text{"the"}, h_0)$ *is missing because it is in the middle of transformation* T_{swap}.

However, Equation (3.12) *is correct for* $i = 2$ *because the transformation* T_{swap} *completes at time step 2.*

Think of the set of all strings in a perturbation space as a tree, like in Fig. 3.2(b), where strings that share prefixes share LSTM states. We want to characterize a subset $G^S_{i,j}$ of LSTM states at the ith layer where the perturbed prefixes have had *all* transformations in a space S applied on the original prefix $\mathbf{x}_{1:j}$ and are in the middle of transformation T_k. Intuitively, $G^S_{i,j}$ is a super set of $H^S_{i,j}$.

We formally define $G^S_{i,j}$ as follows:

$$G^S_{i,j} = \{\textsc{lstm}(\mathbf{z}_{1:i}, h_0) \mid \mathbf{z} \in S^{=}(\mathbf{x}_{1:j}) \wedge L_{\mathbf{z}} \geqslant i\} \tag{3.13}$$

We rewrite Equation (3.13) by explicitly applying the transformations defining the perturbation space S, thus deriving our final equations:

$$G^S_{i,j} = \bigcup_{1 \leqslant k \leqslant n} \bigcup_{\substack{1 \leqslant l \leqslant t_k \\ \varphi_k(\mathbf{x}_{a:b}) = 1}} \left\{ \begin{array}{r} \textsc{lstm}(\mathbf{c}, h) \mid h \in G^{S_{k\downarrow}}_{i-l,j-s_k} \\ \mathbf{c} \in f_{k,:l}(\mathbf{x}_{a:b}) \end{array} \right\}$$
$$\cup \{\textsc{lstm}(x_j, h) \mid h \in G^S_{i-1,j-1}\} \tag{3.14}$$

where $a = j - s_k + 1$ and $b = j$. Notation $f_{k,:l}(\mathbf{x}_{a:b})$ collects the first l symbols for each $\mathbf{z}$ in $f_k(\mathbf{x}_{a:b})$, i.e.,

$$f_{k,:l}(\mathbf{x}_{a:b}) = \{\mathbf{z}_{1:l} \mid \mathbf{z} \in f_k(\mathbf{x}_{a:b})\}$$

$G^S_{i,j}$ contains (1) strings whose suffix is perturbed by $T_k = (\varphi_k, f_k)$ and the last symbol of $\mathbf{z}$ is the lth symbol of the output of T_k (the first line of Equation (3.14)), and (2) strings whose suffix (the last character) is not perturbed by any transformations (the second line of Equation (3.14)).

Then, H_i can be defined as

$$H_i = \bigcup_{S' \in \text{DEC}} \bigcup_{0 \leqslant j \leqslant \text{LEN}_{\mathbf{x}}} G^{S'}_{i,j}$$

Lemma 3.14. *Equation* (3.13) *and Equation* (3.14) *are equivalent.*

The above lemma can be proved similarly to Lemma 3.5.

We use interval abstraction to abstract Equation (3.14). The total number of LSTM cell evaluation needed is $O(\text{LEN}_{\mathbf{x}} \cdot \max_{i=1}^{n}(t_i) \cdot n \prod_{i=1}^{n} \delta_i)$.

Extension to Bi-LSTMs

A *Bi-LSTM* performs a forward and a backward pass on the input. The forward pass is the same as the forward pass in the original LSTM. For the backward pass, we reverse the input string $\mathbf{x}$, the input of the match function φ_i, and the input/output of the replace function f_i of each transformation. Formally, We denote $\mathbf{x}^R$ as the reversed string $\mathbf{x}$. Suppose a transformation T has a match function φ and a replace function f, the reversed transformation $T^R = (\varphi^R, f^R)$ is defined as

$$\varphi^R(\mathbf{x}) = \varphi(\mathbf{x}^R), f^R(\mathbf{x}) = \{\mathbf{z}^R \mid \mathbf{z} \in f(\mathbf{x}^R)\} \quad \forall \mathbf{x} \in \Sigma^*$$

Extension to Tree-LSTMs

A *Tree-LSTM* takes trees as input. We can define the programmable perturbation space over trees in the same form of Equation (3.4), where T_i is a tree transformation. We show some examples of tree transformations in Figure 3.4. $T_{DelStop}$ (Figure 3.4a) removes a leaf node with a stop word in the tree. After removing, the sibling of the removed node becomes the new parent node. T_{Dup} (Figure 3.4b) duplicates a word in a leaf node

by first removing the word and expanding the leaf node with two children, each of which contains the previous word. T_{SubSyn} (Figure 3.4c) substitutes a word in the leaf node with one of its synonyms.

Intuitively, we replace substrings in the formalization of LSTM with subtrees in the Tree-LSTM case. We denote the subtree rooted at node u as $\mathbf{t}_u$ and the size of $\mathbf{t}_u$ as $\textsc{size}_{\mathbf{t}_u}$. The state H_u^S denotes the Tree-LSTM state that reads subtree $\mathbf{t}_u$ generated by a tight perturbation space S. The initial states are the states at leaf node u, $H_u^{\varnothing} = \{\textsc{lstm}(x_u, h_0)\}$ and the final state is H_{root}^S.

We provide transition equations for three specific tree transformations Figure 3.4.

Merge states For a non-leaf node v, we will merge two states, each from a child of v.

$$H_v^S = \bigcup_{S' \in \textsc{dec}_S} \{\textsc{trlstm}(h, g) \mid h \in H_{v_1}^{S'} \wedge g \in H_{v_2}^{S-S'}\} \tag{3.15}$$

where v_1 and v_2 are children of v, and $\textsc{trlstm}$ denotes the Tree-LSTM cell that takes two states as inputs. The notation $S - S'$ computes a tight perturbation space by subtracting S' from S. Formally, suppose

$$S = \{(T_1, \delta_1), (T_2, \delta_2), \ldots, (T_n, \delta_n)\}$$
$$S' = \{(T_1, \delta_1'), (T_2, \delta_2'), \ldots, (T_n, \delta_n')\}$$

then

$$S - S' = \{(T_1, \delta_1 - \delta_1'), (T_2, \delta_2 - \delta_1'), \ldots, (T_n, \delta_n - \delta_n')\}$$

Notice that Equation (3.15) is general to any tight perturbation space S containing these three tree transformations.

T_{SubSyn} We first show the computation of H_u^S for a leaf node u. The substitution only happens in the leaf nodes because only the leaf nodes correspond to words.

$$H_u^{\{T_{SubSyn}, 1\}} = \{\textsc{lstm}(c, h_0) \mid c \in f_{sub}(x_u)\}$$

T_{Dup} T_{Dup} can be seen as a subtree substitution at leaf node u.

$$H_u^{\{T_{Dup}, 1\}} = \{\textsc{trlstm}(h, h)\},$$

where $h = \textsc{lstm}(x_u, h_0)$.

$\mathsf{T}_{\textit{DelStop}}$ Things get tricky for $\mathsf{T}_{\textit{DelStop}}$ because $\{(\mathsf{T}_{\textit{DelStop}}, \delta)\}$ can delete a whole subtree $\mathbf{t}_v$ if (1) the subtree only contains stop words and (2) $\textsc{size}_{\mathbf{t}_v} \leqslant \delta$. We call such subtree $\mathbf{t}_u$ *deletable* if both (1) and (2) are true.

Besides the merging equation Equation (3.15), we provide another transition equation for H^S_v, where v is any non-leaf node with two children v_1, v_2 and a perturbation space $S = \{\ldots, (\mathsf{T}_{\textit{DelStop}}, \delta), \ldots\}$

$$
\begin{aligned}
H^S_v = & \bigcup_{S' \in \textsc{dec}_S} \{\textsc{trlstm}(h, g) \mid h \in H^{S'}_{v_1} \wedge g \in H^{S-S'}_{v_2}\} \\
& \cup \begin{cases}
H^{S-S^{(1)}_{del}}_{v_2} & (1)\ \mathbf{t}_{v_1} \text{ is } \textit{deletable} \\
H^{S-S^{(2)}_{del}}_{v_1} & (2)\ \mathbf{t}_{v_2} \text{ is } \textit{deletable} \\
H^{S-S^{(1)}_{del}}_{v_2} \cup H^{S-S^{(2)}_{del}}_{v_1} & \text{both (1) and (2)} \\
\varnothing & \text{otherwise}
\end{cases}
\end{aligned}
\tag{3.16}
$$

where

$$
S^{(1)}_{del} = \{(\mathsf{T}_{\textit{DelStop}}, \textsc{size}_{\mathbf{t}_{v_1}})\}, S^{(2)}_{del} = \{(\mathsf{T}_{\textit{DelStop}}, \textsc{size}_{\mathbf{t}_{v_2}})\}
$$

Soundness and Complexity We use interval abstraction to abstract the transition equations for Tree-LSTM. The total number of LSTM/Tree-LSTM cell evaluations needed is $O(\textsc{size}_{\mathbf{t}}(\prod_{i=1}^{n} \delta_i)^2)$. The term $(\prod_{i=1}^{n} \delta_i)^2$ comes from Equation (3.15), as we need to enumerate S' for each S in the decomposition set.

3.4 Experiments

Experiment Setups

Datasets

The **AG** News (Zhang et al., 2015) dataset consists of a corpus of news articles collected by Gulli (2005) about the 4 largest news topics. The Stanford Sentiment Treebank (**SST2**) (Socher et al., 2013) dataset consists of sentences from movie reviews and human annotations of their sentiment. The task is to predict the sentiment (positive/negative) of a given sentence. We also use **SST**, which has reviews in the *constituency*

Table 3.1: String transformations to construct the perturbation spaces for evaluation.

	Trans.	Description	Training
CHAR	$T_{SwapPair}$	**swap a pair** of two adjacent characters	Augmentation
	T_{Del}	**delete** a character	Augmentation
	T_{InsAdj}	**insert** to the right of a character one of its **adjacent** characters on the keyboard	Augmentation
	T_{SubAdj}	**substitute** a character with an **adjacent** character on the keyboard	Abstraction
WORD	$T_{DelStop}$	**delete** a **stop word**	Augmentation
	T_{Dup}	**duplicate** a word	Augmentation
	T_{SubSyn}	**substitute** a word with one of its synonyms	Abstraction

parse tree form and five labels. The **IMDB** (Maas et al., 2011) dataset is a collection of movie reviews, ratings, and related information provided by the Internet Movie Database (IMDB). The dataset contains over 50,000 reviews from more than 25,000 movies, including both positive and negative reviews.

Models

A3T For the AG dataset, we trained a smaller character-level model than the one used in Huang et al. (2019) but kept the number of layers and the data preprocessing the same. For the SST2 dataset, we trained a word-level model and a character-level model. We used the same models in Huang et al. (2019), also following their setup.

ARC We trained LSTM and Bi-LSTM models for the SST2 dataset and we trained Tree-LSTM models for the SST dataset.

Please find the details of the setups in the Appendices of the original papers of A3T and ARC.

Perturbations

A3T Our choice of models allows us to experiment with both character-level and word-level perturbations. We evaluated A3T on six perturbation spaces constructed using the seven individual string transformations in Table 3.1.

For the character-level model on dataset AG, we used the following specifications: $\{(T_{SwapPair}, 2), (T_{SubAdj}, 2)\}$, $\{(T_{Del}, 2), (T_{SubAdj}, 2)\}$, and $\{(T_{InsAdj}, 2), (T_{SubAdj}, 2)\}$. For example, the first specification mimics the combination of two spelling mistakes: swap two

Table 3.2: String transformations for S_{review}.

Trans	Description
$T_{review1}$	substitute a phrase in the set A with another phrase in A.
$T_{review2}$	substitute a phrase in the set B with another phrase in B or substitute a phrase in C with another phrase in C.
$T_{review3}$	delete a phrase "one of" from "one of the most ..." or from "one of the ...est".
$T_{review4}$	duplicate a question mark "?" or an exclamation mark "!".

characters up to twice and/or substitute a character with an adjacent one on the keyboard up to twice.

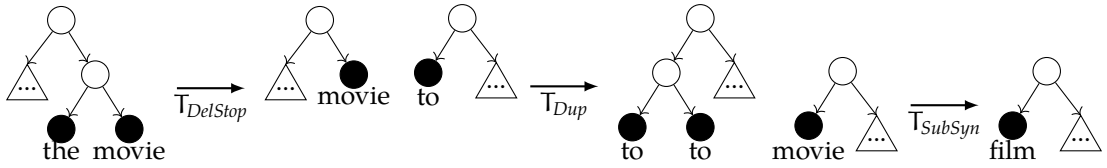
For the word-level model on dataset SST2, we used the following specifications: $\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$, $\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$, and $\{(T_{DelStop}, 2), (T_{Dup}, 2), (T_{SubSyn}, 2)\}$. For example, the first specification removes stop words up to twice and substitutes up to twice words with synonyms.

For the character-level model on dataset SST2, we used the following specifications: $\{(T_{SwapPair}, 1), (T_{SubAdj}, 1)\}$, $\{(T_{Del}, 1), (T_{SubAdj}, 1)\}$, and $\{(T_{InsAdj}, 1), (T_{SubAdj}, 1)\}$. For example, the first specification mimics the combination of two spelling mistakes: swap two characters and/or substitute a character with an adjacent one on the keyboard.

For the character-level model on AG dataset, we considered the perturbations to be applied to a prefix of an input string, namely, a prefix length of 35 for $\{(T_{SwapPair}, 2), (T_{SubAdj}, 2)\}$, a prefix length of 30 for $\{(T_{Del}, 2), (T_{SubAdj}, 2)\}$ and $\{(T_{InsAdj}, 2), (T_{SubAdj}, 2)\}$. For the character-level model on SST2 dataset, we considered perturbations with $\delta = 1$ but allow the perturbations to be applied to the whole input string. We made these restrictions because one cannot efficiently evaluate the exhaustive accuracy with larger δ, due to the combinatorial explosion of the size of the perturbation space.

ARC We create perturbation spaces by combining the word-level transformations in Table 3.1, e.g., $\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$ removes up to two stop words and replaces up to two words with synonyms. We also design a domain-specific perturbation space S_{review} for movie reviews by inspecting highly frequent n-grams in the movie review training set. Formally,

$$S_{review} = \{(T_{review1}, 1), (T_{review2}, 1), (T_{review3}, 1), (T_{review4}, 1), (T_{SubSyn}, 2)\}$$



(a) $T_{DelStop}$: remove *the*. (b) T_{Dup}: duplicate *to*. (c) T_{SubSyn}: substitute *movie* with *film*

Figure 3.4: Examples of tree transformations.

where $T_{review1}$, $T_{review2}$, $T_{review3}$, and $T_{review4}$ are defined in Table 3.2 with

$$A = \{\text{“this is”}, \text{“this 's”}, \text{“it is”}, \text{“it 's”}\}$$
$$B = \{\text{“the movie”}, \text{“the film”}, \text{“this movie”}, \text{“this film”}, \text{“a movie”}, \text{“a film”}\}$$
$$C = \{\text{“the movies”}, \text{“the films”}, \text{“these movies”}, \text{“these films”}\}$$

For Tree-LSTMs, we consider the tree transformations exemplified in Figure 3.4.

Baselines

A3T A3T is our technique that can be implemented in various ways. For our experiments, we made the following choices. First, we manually labeled which transformations in S are explored using augmentation and which ones are explored using abstract interpretation (the third column in Table 3.1).[3] Second, we implemented two different ways of performing data augmentation for the transformations in S_{aug}: (1) **A3T (HotFlip)** uses HotFlip to find the worst-case samples for augmentation, while (2) **A3T (search)** performs an explicit search through the perturbation space to find the worst-case samples for augmentation. Finally, we used DiffAI (Mirman et al., 2018) to perform abstract training for the transformations in S_{abs}, using the intervals abstraction.

We implement A3T and compare it to the following baselines.

- **Normal training** is the vanilla training method that minimizes the cross entropy between predictions and target labels. This method does not use the perturbation space and does not attempt to train a robust model.
- **Random (Data) augmentation** performs adversarial training using a weak adversary that simply picks a random perturbed example from the perturbation space.

[3]We consider this choice of split to be a hyperparameter.

- **HotFlip augmentation** performs adversarial training using the HotFlip (Ebrahimi et al., 2018) attack to solve the inner maximization problem.

In all augmentation training baselines, and A3T, we also adopt a curriculum-based training method (Zhang et al., 2018a; Gowal et al., 2019) which uses a hyperparameter λ to weigh between normal loss and maximization objective.

ARC For training certifiable models against arbitrary string transformations, we compare ARC to (1) **Normal training**, (2) **Data augmentation**, (3) **HotFlip augmentation**, and (4) **A3T** on CNN.

For training certifiable models against word substitution, we compare ARC to (1) *Jia et al.* (*2019*) that trains *certifiably* robust (Bi)LSTMs. We call this CertSub in the rest of this section. (2) *ASCC* (*Dong et al., 2021*) that trains *empirically* robust (Bi)LSTMs. And (3) *Huang et al.* (*2019*) that trains *certifiably* robust CNNs.

For certification, we compare ARC to (1) *POPQORN* (*Ko et al., 2019*), the state-of-the-art approach for certifying LSTMs. (2) *SAFER* (*Ye et al., 2020*) that provides probabilistic certificates to word substitution.

Xu et al. (*2020*) is a special case of ARC where the perturbation space only contains substitution. The abstract state $g_{i,j}$ in their paper (Page 6, Theorem 2) is equivalent to $\widehat{H}_{i,i}^{\{(T_{SubSyn}, j)\}}$ in our paper.

Evaluation Metrics

- **Normal accuracy** is the vanilla accuracy of the model on the test set.
- **HotFlip accuracy** is the adversarial accuracy of the model with respect to the HotFlip attack, i.e., for each point in the test set, we apply the HotFlip attack and test if the classification is still correct.
- **Certified accuracy** (CF Acc.) is the percentage of points in the test set certified as S-robust (Eq 3.1) using ARC.
- **Exhaustive accuracy** is the worst-case accuracy of the model: a prediction on $(\mathbf{x}, y)$ is considered correct if and only if all points $\mathbf{z} \in S(\mathbf{x})$ lead to the correct prediction. Formally, given a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ and a perturbation space

Table 3.3: Experiment results for the three perturbations on the character-level model on AG dataset. We show the normal accuracy (Acc.), HotFlip accuracy (HF Acc.), and exhaustive accuracy (Exhaustive) of five different training methods.

	$\{(T_{SwapPair}, 2), (T_{SubAdj}, 2)\}$			$\{(T_{Del}, 2), (T_{SubAdj}, 2)\}$			$\{(T_{InsAdj}, 2), (T_{SubAdj}, 2)\}$		
Training	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive
Normal	87.5	71.5	60.1	87.5	79.0	62.5	87.5	79.1	59.0
Random Aug.	87.5	75.7	68.2 [+8.1]	87.4	81.3	69.4 [+6.9]	87.8	81.2	69.7 [+10.7]
HotFlip Aug.	86.6	85.7	84.9 [+24.8]	85.8	84.9	82.7 [+20.2]	86.8	85.9	82.6 [+23.6]
A3T (HotFlip)	86.4	86.4	86.4 [+26.3]	87.2	87.1	85.7 [+23.2]	87.4	87.4	85.5 [+26.5]
A3T (search)	86.9	86.8	**86.8 [+26.7]**	87.6	87.4	**86.2 [+23.7]**	87.9	87.8	**86.5 [+27.5]**

Table 3.4: Experiment results for the three perturbations on the word-level model on SST dataset.

	$\{T_{DelStop}, 2), (T_{SubSyn}, 2)\}$			$\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$			$\{(T_{DelStop}, 2), (T_{Dup}, 2), (T_{SubSyn}, 2)\}$		
Training	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive
Normal	82.4	68.9	64.4	82.4	55.8	47.9	82.4	54.8	42.4
Random Aug.	80.0	70.0	66.0 [+1.6]	81.5	54.2	49.7 [+1.8]	81.0	56.1	46.2 [+3.8]
HotFlip Aug.	80.8	74.4	68.3 [+3.9]	80.8	68.7	56.0 [+8.1]	81.2	69.0	51.0 [+8.6]
A3T (HotFlip)	80.2	73.5	70.2 [+5.8]	79.9	69.7	57.7 [+9.8]	78.8	68.1	55.1 [+12.7]
A3T (search)	79.9	74.4	**71.2 [+6.8]**	79.0	70.7	**62.7 [+14.8]**	77.7	69.8	**59.8 [+17.4]**

S, we define exhaustive accuracy as follows:

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[\forall \mathbf{z} \in S(\mathbf{x}_i).\, F_\theta(\mathbf{z}) = y_i] \tag{3.17}$$

Intuitively, for each sample $(\mathbf{x}_i, y_i)$, its classification is considered correct iff F_θ predicts y_i for every single point in $S(\mathbf{x}_i)$.

HotFlip accuracy is an upper bound of exhaustive accuracy; certified accuracy is a lower bound of exhaustive accuracy.

Experiment Results

A3T

In this section, we evaluate A3T by answering the following question: does A3T improve robustness in rich perturbation spaces for character-level and word-level models?

Results We show the results for the selected perturbation spaces on character-level and word-level models in Tables 3.3, 3.4, and 3.5, respectively.

Table 3.5: Experiment results for the three perturbations on the character-level model on SST2 dataset.

	$\{(\mathsf{T}_{SwapPair}, 1), (\mathsf{T}_{SubAdj}, 1)\}$			$\{(\mathsf{T}_{Del}, 1), (\mathsf{T}_{SubAdj}, 1)\}$			$\{(\mathsf{T}_{InsAdj}, 1), (\mathsf{T}_{SubAdj}, 1)\}$		
Training	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive	Acc.	HF Acc.	Exhaustive
Normal	77.0	36.5	23.0	77.0	50.7	25.8	77.0	51.0	24.4
Random Aug.	75.6	47.1	28.2 [+5.2]	75.7	56.4	29.3 [+3.5]	74.5	57.0	33.8 [+9.4]
HotFlip Aug.	71.4	63.9	34.8 [+11.8]	76.6	67.1	38.0 [+12.2]	76.1	70.4	33.4 [+9.0]
A3T (HotFlip)	73.6	54.8	35.2 [+12.2]	75.3	58.2	32.9 [+7.1]	72.4	66.3	44.7 [+20.3]
A3T (search)	70.2	57.1	**48.7 [+15.7]**	72.5	62.5	**44.8 [+19.0]**	71.6	65.0	**55.2 [+30.8]**

Compared to normal training, the results show that both A3T (HotFlip) and A3T (search) increase the exhaustive accuracy and can improve the robustness of the model. A3T (HotFlip) and A3T (search) also outperform random augmentation and HotFlip augmentation. In particular, A3T (search) has exhaustive accuracy that is on average 20.3 higher than normal training, 14.6 higher than random augmentation, and 6.7 higher than HotFlip augmentation.

We also compared A3T to training using only abstraction (i.e., all transformations in S are also in S_{abs}) for the specification $\{(\mathsf{T}_{SwapPair}, 2), (\mathsf{T}_{SubAdj}, 2)\}$ on AG dataset and $\{(\mathsf{T}_{SwapPair}, 1), (\mathsf{T}_{SubAdj}, 1)\}$ on SST2 dataset (not shown in Tables 3.3, 3.4, and 3.5); this is the only specification that can be fully trained abstractly since it only uses length-preserving transformations. Training using only abstraction yields an exhaustive accuracy of 86.9 for $\{(\mathsf{T}_{SwapPair}, 2), (\mathsf{T}_{SubAdj}, 2)\}$ on AG dataset, which is similar to the exhaustive accuracy of A3T (HotFlip) (86.4) and A3T (search) (86.8). However, training using only abstraction yields an exhaustive accuracy of 47.0 for $\{(\mathsf{T}_{SwapPair}, 1), (\mathsf{T}_{SubAdj}, 1)\}$ on SST2 dataset, which is better than the one obtained using normal training, but much lower than the exhaustive accuracy of A3T (HotFlip) and A3T (search). Furthermore, the normal accuracy of the abstraction technique on SST2 dataset drops to 58.8 due to the over-approximation of the perturbation space while A3T (HotFlip) (73.6) and A3T (search) (70.2) retain high normal accuracy.

A3T yields models that are more robust to complex perturbation spaces than those produced by augmentation and abstraction techniques. This result holds for both character-level and word-level models.

ARC

In this section, we evaluate ARC on three settings: (1) training certifiable models against arbitrary string transformations, (2) training certifiable models against word substitution, and (3) certification against word substitution.

Table 3.6: Results of LSTM (on SST2), Tree-LSTM (on SST), and Bi-LSTM (on SST2) for three perturbation spaces. **Note:** The results of LSTM on $\{(\mathsf{T}_{Dup}, 2), (\mathsf{T}_{SubSyn}, 2)\}$ were updated after the publication of the original ARC paper. We improved the implementation of ARC on T_{Dup} because the previous implementation of ARC included some cases in $(\mathsf{T}_{Dup}, 3)$ for $(\mathsf{T}_{Dup}, 2)$, leading to more over-approximation. This improvement also affects Table 3.12.

		$\{(\mathsf{T}_{DelStop}, 2), (\mathsf{T}_{SubSyn}, 2)\}$				$\{(\mathsf{T}_{Dup}, 2), (\mathsf{T}_{SubSyn}, 2)\}$				$\{(\mathsf{T}_{DelStop}, 2), (\mathsf{T}_{Dup}, 2)\}$			
	Train	Acc.	HF Acc.	CF Acc.	EX Acc.	Acc.	HF Acc.	CF Acc.	EX Acc.	Acc.	HF Acc.	CF Acc.	EX Acc.
LSTM	Normal	**84.6**	71.9	4.6	68.9	84.6	64.0	0.6	55.1	**84.6**	73.7	1.2	65.2
	Data Aug.	84.0	77.0	5.5	74.4	**84.7**	70.2	0.4	61.5	84.5	75.4	0.4	68.3
	HotFlip	84.0	**78.7**	4.3	74.6	82.5	**75.9**	0.0	62.0	84.4	**80.6**	0.0	68.7
	ARC	82.5	77.8	**72.5**	**77.0**	80.2	70.0	**58.7**	**64.0**	82.6	78.6	**69.4**	**74.6**
Tree-LSTM	Normal	**50.3**	39.9	4.1	33.8	**50.3**	33.4	0.0	17.9	**50.3**	40.1	0.0	25.7
	Data Aug.	47.5	40.8	1.4	36.4	48.1	37.1	0.0	23.0	47.6	40.6	0.0	29.0
	HotFlip	49.5	43.4	1.6	38.4	48.7	**39.5**	0.0	29.0	49.5	42.7	0.0	32.1
	ARC	46.4	**43.4**	**30.9**	**41.9**	46.1	39.0	**17.1**	**37.6**	46.5	**43.8**	**19.2**	**40.0**
Bi-LSTM	Normal	83.0	71.1	8.2	68.0	83.0	63.4	2.1	56.1	83.0	72.5	6.4	65.5
	Data Aug.	83.2	75.1	8.7	72.9	**83.5**	66.8	1.3	59.1	**84.6**	75.0	4.6	68.6
	HotFlip	**83.6**	**79.2**	9.2	73.4	82.8	**76.6**	0.1	55.5	83.5	**79.1**	0.0	55.7
	ARC	83.5	78.7	**70.9**	**77.5**	80.2	71.4	**59.8**	**66.4**	82.6	76.2	**66.2**	**71.8**

Table 3.7: Results of LSTM on SST2 dataset for S_{review}.

	S_{review}			
Train	Acc.	HF Acc.	CF Acc.	EX Acc.
Normal	**83.9**	72.4	21.0	71.0
Data Aug.	79.2	72.5	30.5	71.7
HotFlip	79.6	74.3	34.5	72.9
ARC	82.3	**78.1**	**74.2**	**77.1**

Results: Training against Arbitrary String Transformations We compare ARC to data augmentation and HotFlip using the three perturbation spaces in Table 3.6 and the domain-specific perturbation space S_{review} in Table 3.7.

ARC outperforms data augmentation and HotFlip in terms of EX Acc. and CF Acc. Table 3.6 shows the results of LSTM, Tree-LSTM, and Bi-LSTM models on the tree perturbation spaces. Table 3.7 shows the results of LSTM models on the domain-specific perturbation space S_{review}. ARC has significantly higher EX Acc. than normal training ($+8.1, +14.0, +8.7$ on average), data augmentation ($+4.2, +10.4, +5.0$), and HotFlip ($+3.6, +6.7, +10.4$) for LSTM, Tree-LSTM, and Bi-LSTM respectively.

Models trained with ARC have a relatively high CF Acc. (53.6 on average). Data augmentation and HotFlip result in models not amenable to certification—in some

Table 3.8: ARC vs A3T (CNN) on SST2 dataset.

		$\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$				$\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$			
Train	Model	Acc.	HF Acc.	CF Acc.	EX Acc.	Acc.	HF Acc.	CF Acc.	EX Acc.
A3T (HotFlip)	CNN	80.2	71.9	N/A	70.2	79.9	68.3	N/A	57.7
ARC	LSTM	**82.5**	**77.8**	**72.5**	**77.0**	**80.2**	**70.0**	**55.4**	**64.0**

Table 3.9: ARC vs CertSub and ASCC on IMDB dataset.

	$\{(T_{SubSyn}, 1)\}$			$\{(T_{SubSyn}, 2)\}$		
Train	Acc.	CF Acc.	EX Acc.	Acc.	CF Acc.	EX Acc.
CertSub	76.8	67.0	71.0	76.8	64.8	68.3
ACSS	82.8	0.0	81.5	82.8	0.0	**80.8**
ARC	**87.7**	**77.8**	**82.6**	**86.3**	**71.0**	78.2

cases, almost nothing in the test set can be certified.

ARC produces more robust models at the expense of accuracy. Other robust training approaches like CertSub and A3T also exhibit this trade-off. However, as we will show next, ARC retains higher accuracy than these approaches.

We compare ARC with A3T using $\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$ and $\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$. We do not use $\{(T_{DelStop}, 2), (T_{Dup}, 2)\}$ because A3T degenerates to HotFlip training for this perturbation space. A3T degenerates to HotFlip training on $\{(T_{DelStop}, 2), (T_{Dup}, 2)\}$, so we do not use this perturbation space.

The LSTMs trained using ARC are more robust than the CNNs trained by A3T for both perturbation spaces; ARC can certify the robustness of models while A3T cannot. Table 3.8 shows that ARC results in models with higher accuracy (+2.3 and +0.3), HF Acc. (+5.9 and +1.7), and EX Acc. (+6.8 and +6.3) than those produced by A3T. ARC can certify the trained models while A3T cannot.

Results: Training against Word Substitutions We choose two perturbation spaces, $\{(T_{SubSyn}, 1)\}$ and $\{(T_{SubSyn}, 2)\}$. We train one model per perturbation space using ARC under the same experimental setup of CertSub, BiLSTM on the IMDB dataset. By definition, CertSub and ASCC train for an arbitrary number of substitutions. CF Acc. is computed using ARC. Note that CertSub can only certify for $\{(T_{SubSyn}, \infty)\}$ and ASCC cannot certify.

ARC trains more robust models than CertSub for two perturbation spaces with word substitution. Table 3.9 shows that ARC achieves higher accuracy, CF Acc., and EX Acc. than CertSub on the two perturbation spaces.

Table 3.10: ARC vs Huang et al. (2019) (CNN) on SST2 dataset.

		$\{(\mathsf{T}_{SubSyn}, 3)\}$			
Train	Model	Acc.	HF Acc.	CF Acc.	EX Acc.
Huang et al. (2019)	CNN	81.7	77.2	44.5	76.5
ARC	LSTM	**83.3**	**78.3**	**73.3**	**77.9**

ARC trains a more robust model than ASCC for $\{(\mathsf{T}_{\mathbf{SubSyn}}, 1)\}$***, but ASCC's model is more robust for*** $\{(\mathsf{T}_{\mathbf{SubSyn}}, 2)\}$**.** Table 3.9 shows that the ARC-trained models have higher accuracy and CF Acc.

We use $\{(\mathsf{T}_{SubSyn}, 3)\}$ on SST2 dataset for comparison between ARC and Huang et al. (2019). We directly quote the results in their paper.

ARC trains more robust LSTMs than CNNs trained by Huang et al. (2019). Table 3.10 shows that ARC results in models with higher accuracy (+1.6), HF Acc. (+1.1), CF Acc. (+28.8), and EX Acc. (+3.4) than those produced by Huang et al. (2019).

Results: Certification against Word Substitutions We compare the certification of an ARC-trained model and a normal model against $\{(\mathsf{T}_{SubSyn}, 3)\}$ on the first 100 examples in SST2 dataset. Because POPQORN can only certify the l_p norm ball, we overapproximate the radius of the ball as the maximum l_1 distance between the original word and its synonyms.

ARC runs much faster than POPQORN. ARC is more accurate than POPQORN on the ARC-trained model, while POPQORN is more accurate on the normal model. ARC certification takes 0.17sec/example on average for both models, while POPQORN certification takes 12.7min/example. ARC achieves 67% and 5% CF Acc. on ARC-trained model and normal model, respectively, while POPQORN achieves 22% and 28% CF Acc., but crashes on 45% and 1% of examples for two models.

SAFER is a post-processing technique for certifying robustness via *randomized smoothing*. We train a Bi-LSTM model using ARC following SAFER's experimental setup on the IMDB dataset and SAFER's synonym set, which is different from CertSub's. We consider the perturbation spaces $\{(\mathsf{T}_{SubSyn}, 1)\}$ and $\{(\mathsf{T}_{SubSyn}, 2)\}$. We use both ARC and SAFER to certify the robustness. The significance level of SAFER is set to 1%.

SAFER has a higher certified accuracy than ARC. However, its certificates are statistical, tied to word substitution only, and are slower to compute. Considering $\{(\mathsf{T}_{SubSyn}, 2)\}$, ARC results in a certified accuracy of 79.6 and SAFER results in a certified accuracy of 86.7 (see Table 3.11). Note that certified accuracies are incomparable

Table 3.11: ARC vs SAFER on IMDB dataset.

	$\{(T_{SubSyn}, 1)\}$		$\{(T_{SubSyn}, 2)\}$	
Train	CF Acc.	RS Acc.	CF Acc.	RS Acc.
Data Aug.	0.2	**90.0**	0.1	**89.7**
ARC	**82.0**	87.2	**79.6**	86.7

Table 3.12: Results of different instantiations of ARC-A3T on SST2 dataset.

	$\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$			$\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$		
Train	Acc.	CF Acc.	EX Acc.	Acc.	CF Acc.	EX Acc.
Abs-fir	**83.2**	15.1	68.6	82.6	16.8	**66.7**
Abs-sec	81.4	71.1	75.8	**83.0**	54.2	65.4
ARC	82.5	**72.5**	**77.0**	80.2	**58.7**	64.0

because SAFER computes certificates that only provide statistical guarantees. Also, note that ARC uses $O(n \prod_{i=1}^{n} \delta_i)$ forward passes for each sample, while SAFER needs to randomly sample thousands of times. In the future, it would be interesting to explore extensions of SAFER to ARC's rich perturbation spaces.

ARC-A3T

We can apply the idea of A3T to ARC, extending ARC to abstract any subset of the given perturbation space and to augment the remaining perturbation space. We show the effectiveness of this extension in the appendix.

We evaluate ARC-A3T on the same perturbation spaces as we do for A3T. For each perturbation space, ARC-A3T has four instantiations: abstracting the whole perturbation space (downgraded to ARC), abstracting the first perturbation space ($\{(T_{DelStop}, 2)\}$ or $\{(T_{Dup}, 2)\}$), abstracting the second perturbation space ($\{(T_{SubSyn}, 2)\}$), and augmenting the whole perturbation space. We use enumeration for augmenting. We do not test the last instantiation because enumeration the whole perturbation space is infeasible for training. We further evaluate the trained models on different perturbation sizes, i.e., $\{(T_{DelStop}, \delta), (T_{SubSyn}, \delta)\}$ and $\{(T_{Dup}, \delta), (T_{SubSyn}, \delta)\}$ with $\delta = 1, 2, 3$.

Different instantiations of ARC-A3T win for different perturbation spaces. Table 3.12 shows the results of different instantiations of ARC-A3T. For $\{(T_{DelStop}, 2), (T_{SubSyn}, 2)\}$, abstracting the first perturbation space ($\{(T_{DelStop}, 2)\}$) achieves the best accuracy and abstracting the whole perturbation space (ARC) achieves the best CF

Acc. and EX Acc. For $\{(T_{Dup}, 2), (T_{SubSyn}, 2)\}$, abstracting the first perturbation space ($\{(T_{Dup}, 2)\}$) achieves the best EX Acc., abstracting the second perturbation space ($\{(T_{SubSyn}, 2)\}$) achieves the best accuracy, and abstracting the whole perturbation space (ARC) achieves the best CF Acc.

3.5 Related Work

Adversarial Text Generation Zhang et al. (2019d) present a comprehensive overview of adversarial attacks on neural networks over natural language. A3T focuses on the word- and character-level transformations. ARC implements the word-level transformation but can also be extended to character-level. HotFlip (Ebrahimi et al., 2018) is a gradient-based approach that can generate the adversarial text in the perturbation space described by word- and character-level transformations. MHA (Zhang et al., 2019a) uses Metropolis-Hastings sampling guided by gradients to generate word-level adversarial text via word substitution. Other work focuses on generating adversarial text on the sentence-level (Liang et al., 2018) or paraphrase-level (Iyyer et al., 2018; Ribeiro et al., 2018). Another kind of adversary is universal adversarial triggers (Wallace et al., 2019) that are fixed words which concatenate to any input will trigger the model to make a false prediction.

Formal Verification for NLP Models In the verification for NLP tasks, one group of works aims to certify robustness under specific string transformations such as word substitution and deletion. Jia et al. (2019); Huang et al. (2019) proposed to use IBP to certify the robustness under arbitrary numbers of word substitutions. Xu et al. (2020) proposed to use IBP and CROWN-IBP to certify the robustness under word substitutions up to a certain amount. Welbl et al. (2020) proposed the formal verification under text deletion for models based on the popular decomposable attention mechanism by interval bound propagation. Another group of works aims to certify robustness under l_p norm perturbation on word embeddings. Shi et al. (2020) combined forward propagation and a tighter backward bounding process to achieve the formal verification of Transformers. Bonaert et al. (2021) proposed the Multi-norm Zonotope abstract domain, an extension of the classical Zonotope designed to handle l_1 and l_2-norm bound perturbations. POPQORN (Ko et al., 2019) and Ryou et al. (2021) are general

algorithms to quantify the robustness of recurrent neural networks, including RNNs, LSTMs, and GRUs.

SAFER (Ye et al., 2020) is a model-agnostic approach that uses *randomized smoothing* (Cohen et al., 2019) to give probabilistic certificates of robustness to word substitution. ARC gives a non-probabilistic certificate and can handle arbitrary perturbation spaces beyond substitution.

Adversarial Training and Abstract (Certified) Training of NLP Models Adversarial training is an empirical defense method that can improve the robustness of models by solving a *robust-optimization* problem (Madry et al., 2018), which minimizes worst-case (adversarial) loss. Some techniques in NLP use adversarial attacks to compute a lower bound on the worst-case loss (Ebrahimi et al., 2018). ASCC (Dong et al., 2021) overapproximates the word substitution attack space by a convex hull where a lower bound on the worst-case loss is computed using gradients.

Other techniques, named abstract training (or certified training), compute upper bounds on adversarial loss using abstract interpretation (Gowal et al., 2019; Mirman et al., 2018). Any incomplete verifiers based on abstract interpretation discussed in the previous paragraph can be used in abstract training. For example, Jia et al. (2019); Xu et al. (2020) used interval domain to capture the perturbation space of substitution and train robust models for LSTMs. Huang et al. (2019) proposed a simplex space to capture the perturbation space of substitution. They converted the simplex into intervals after the first layer of the neural network and obtained the abstract loss by IBP. However, these abstract training techniques for NLP tasks cannot handle the general programmable perturbation space proposed in Chapter 3.

A3T trains robust CNN models against a programmable perturbation space by combining adversarial training and abstract training. ARC designs an abstract interpretation technique specialized for LSTMs and trains robust LSTMs against programmable perturbation spaces.

3.6 Preliminary Work: ARC on Autoaggressive Transformers

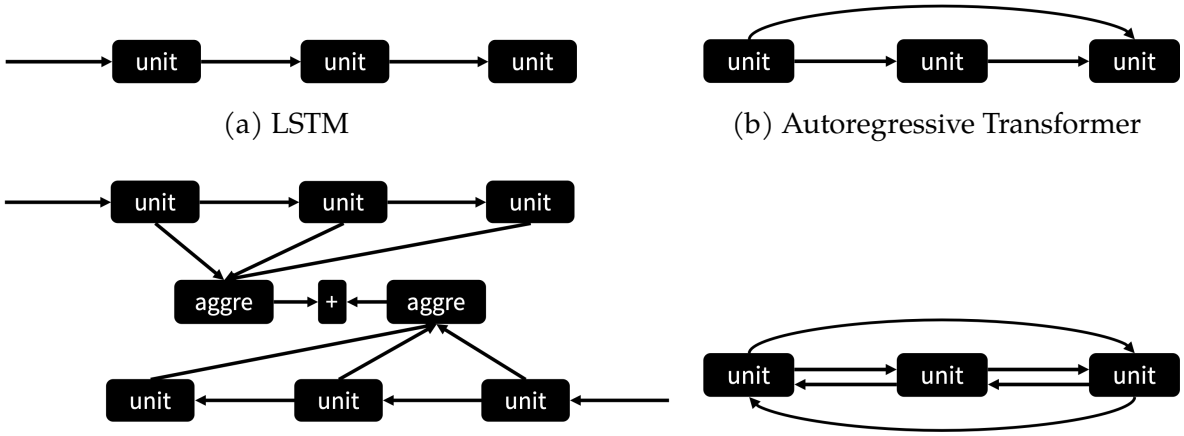
Transformers are the backbone of popular large language models. Existing work (Shi et al., 2020; Bonaert et al., 2021) only considers an l_p norm perturbation space for certifying Transformers. While these approaches can naturally extend to word substitutions, they cannot handle arbitrary perturbation spaces formalized in Section 3.1. Therefore, extending ARC's certification against arbitrary perturbation spaces to Transformers is a critical step toward the trustworthiness of large language models and future artificial intelligence models.

Recently, the architectures of Transformers have undergone a clear shift from bi-directional encoder structures like BERT (Devlin et al., 2019) to autoregressive decoder-only structures like GPT-3 (Brown et al., 2020). This shift surprisingly invigorates the possibility of extending ARC to Transformers with an autoregressive decoder-only structure. ARC cannot be extended to bi-directional Transformers (Bi-Transformers) because Bi-Transformers have loop-like structures, as illustrated in Figure 3.5b.[4] These loop-like structures prevent the application of the dynamic programming algorithm, which is used by ARC to progressively propagate over-approximation according to the programmable perturbation space through the neural network. Compared to a Bi-Transformer, an autoregressive Transformer does not contain any loop-like structure, making it a promising candidate for certification by ARC.

Although there is a glimmer of hope in applying ARC to autoregressive Transformers, we point out two inevitable challenges when applying ARC to autoregressive Transformers. A preprint of this work can be found in Zhang et al. (2024a). Notably, none of these challenges have been discussed in any previous work. We then propose corresponding approaches to overcome these two challenges.

Position Encoding under Arbitrary Perturbation Spaces Position encoding is a crucial component in the Transformer architecture. It allows the model to incorporate positional information of tokens in a sequence. Since the Transformer relies solely on attention mechanisms and has no inherent token order notion, position encodings are added to the input embeddings to inject positional information. By incorporating

[4]One may question why ARC can handle Bi-LSTM but not Bi-Transformers. The answer is that the Bi-LSTM structure does not have a loop-like structure, as illustrated in Figure 3.5c.



(a) LSTM (b) Autoregressive Transformer

(c) Bi-LSTM (d) Bi-Transformer

Figure 3.5: Illustration of an LSTM, an autoregressive Transformer, a Bi-LSTM, and a Bi-Transformer.

position encodings, the Transformer can effectively capture the order and positional relationships of tokens in a sequence.

Position encodings do not cause an issue for previous works because they only consider length-preserving perturbation spaces, such as synonym substitutions. However, regarding arbitrary programmable perturbation spaces, perturbed inputs can have different lengths. This difference in lengths necessitates careful handling of the embeddings before attention computation. Conventionally, before being fed to the attention computation, the position encoding p_i has already been added to the embedding of the ith token, i.e., $e_i = e'_i + p_i$, where e'_i is the original input embedding. Directly applying ARC to e_i can cause position mismatches. To solve this issue, we introduce a new embedding $e_{i,j}$, which represents the original jth input token being moved to the ith position in the perturbed input, denoted as $e_{i,j} = e'_j + p_i$. Intuitively, the new embedding separates the input embedding and position encoding and adapts to the input-length change. Equipped with ARC's definition of $G^S_{i,j}$, which captures the set of hidden states with the ith token in the perturbed input and the jth token in the original input (see Equation (3.13)), this new embedding accurately captures position embeddings $e_{i,j}$.

The following example illustrates how to overcome the position encoding challenge, assuming we still use an LSTM in addition to position encodings instead of an autoregressive Transformer, which will be explained in the next challenge.

Example 3.15. *Let* $\mathbf{x}$ = "to the", e'_1 *and* e'_2 *be the original input embeddings for the first and second word, respectively, and* p_1 *and* p_2 *be the corresponding position encodings. Conventionally,*

the embeddings fed to the next LSTM layer (or Transformer layer) are computed as $e_1 = e'_1 + p_1$ *and* $e_2 = e'_2 + p_2$*. As a result,* $G_{1,2}^{\{(T_{del},1)\}}$ *will be computed as* $\{\textsc{lstm}(e_1, h_0), \textsc{lstm}(e_2, h_0)\}$ *according to Example 3.4. However, the second state in the above computation is incorrect because after deleting "to", the word "the" becomes the first word in the sentence, and its embedding should be* $e'_2 + p_1$ *instead of* $e_2 = e'_2 + p_2$*.*

In contrast, leveraging the new embedding $e_{i,j}$*, we can compute the correct state set* $G_{1,2}^{\{(T_{del},1)\}} = \{\textsc{lstm}(e_{1,1}, h_0), \textsc{lstm}(e_{1,2}, h_0)\}$*.*

Handling Autoregressive Transformer The key difference between Figure 3.5a and Figure 3.5b is that for ith state, an LSTM takes the previous state as its input. In contrast, an autoregressive transformer takes all the previous states as inputs because it needs to compute the attention over all previous states.

A straightforward approach is storing all previous states instead of a single previous state during computation. We first discuss which states need to be stored by investigating the computation of the attention mechanism in Transformers. Formally, we define the nth transformer state on the perturbed input $\mathbf{z}$ as

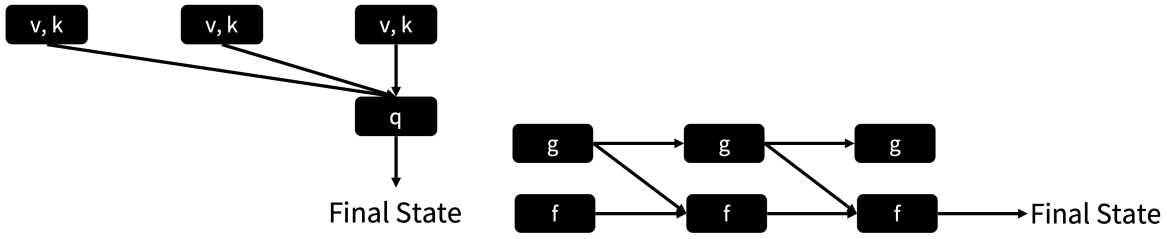
$$\textsc{trans}(\mathbf{z}_{1:n}, m) = \sum_{i=1}^{n} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{n} e^{q_n k_j}}, \tag{3.18}$$

where m is the index mapping from $\mathbf{z}$ to the original input $\mathbf{x}$, and $q_n = W_q e_{n,m(n)}$, $k_i = W_k e_{i,m(i)}$, $v_i = W_v e_{i,m(i)}$ are query, key, value vectors computed from the new embedding $e_{i,m(i)}$. From Equation (3.18), we need to take care of $\mathbf{z}$, m, $q_{1:i}$, $k_{1:i}$, and $v_{1:i}$. Among these, the index mapping m and the perturbed input $\mathbf{z}$ will be maintained by the definition of $G_{i,j}^{S}$ in ARC, as shown in Example 3.15. Therefore, we need to store the remaining three matrices (or three lists of vectors) $q_{1:i}$, $k_{1:i}$, $v_{1:i}$ for computing state $\textsc{trans}(\mathbf{z}_{1:n}, m)$. However, this approach has two drawbacks:

1. It needs an extra $O(\textsc{len}_{\mathbf{x}})$ factor on memory consumption for Transformers compared to ARC for LSTMs.

2. The interval abstraction used in ARC will cause additional over-approximation when abstracting $q_{1:i}$, $k_{1:i}$, and $v_{1:i}$ (see the next example).

Example 3.16. *Consider* $\mathbf{x}$ = "A B"*, where each word has one synonym ("a", "b", respectively) that can be substituted. Let* $e_{i,i}$ *be the embedding of the* i*-th original token, and* $e^{s}_{i,i}$ *be the*



(a) Autoregressive Transformer (b) Two RNNs interacting in parallel

Figure 3.6: One-layer autoregressive transformers are two RNNs interacting in parallel. *embedding of the* $\mathfrak{i}$*-th substituted token. Without loss of generality, we only consider the abstraction of* $\mathfrak{k}_{1:\mathfrak{i}}$ *in this example. Let* $\widehat{A}^{S}_{\mathfrak{i},\mathfrak{j}}$ *denote the abstraction of* $\mathfrak{k}_{1:\mathfrak{i}}$ *where the perturbed prefixes have had all transformation in a space* S *applied on the original prefix* $\mathbf{x}_{1:\mathfrak{j}}$*. We have* $\widehat{A}^{\varnothing}_{1,1} = \alpha(\{W_{\mathfrak{k}} e_{1,1}\})$, $\widehat{A}^{\{(T_{\text{SubSyn}},1)\}}_{1,1} = \alpha(\{W_{\mathfrak{k}} e^{s}_{1,1}\})$*. At the first state, the interval abstractions are still tight.*

At the second state, let $[\mathfrak{a};\mathfrak{b}]$ *denote the concatenation of vectors* $\mathfrak{a}$ *and* $\mathfrak{b}$*. Then, we have*

$$\begin{aligned}\widehat{A}^{\{(T_{\text{SubSyn}},1)\}}_{2,2} &= \alpha\begin{pmatrix} [\widehat{A}^{\varnothing}_{1,1}; W_{\mathfrak{k}} e^{s}_{2,2}], \\ [\widehat{A}^{\{(T_{\text{SubSyn}},1)\}}_{1,1}; W_{\mathfrak{k}} e_{2,2}] \end{pmatrix} \\ &= \left[\alpha(\{W_{\mathfrak{k}} e_{1,1}, W_{\mathfrak{k}} e^{s}_{1,1}\}); \alpha(\{W_{\mathfrak{k}} e_{2,2}, W_{\mathfrak{k}} e^{s}_{2,2}\})\right] \end{aligned} \tag{3.19}$$

The $\widehat{A}^{\{(T_{\text{SubSyn}},1)\}}_{2,2}$ *computed above introduces an extra over-approximation, which ARC has never introduced before*[5]*, as* $\widehat{A}^{\{(T_{\text{SubSyn}},1)\}}_{2,2}$ *contains a perturbed input "a b", which substituted two words with their synonyms.*

The over-approximation introduced in Example 3.16 is due to the fact that ARC is only able to record the previous string transformation, which is sufficient for LSTM models whose hidden states only depend on the previous state. We can modify ARC to record the previous $\mathfrak{k}$ string transformations instead of one. However, this modification will introduce an extra $O(\textsc{len}_{\mathbf{x}}^{\mathfrak{k}-1})$ time complexity in the algorithm. To avoid the additional time complexity, we ask: *can we rewrite Eq.* (3.18) *into equations such that the* $\mathfrak{n}$*th state only depend on the previous state?*

[5]Note that the above over-approximation does not even appear at $\widehat{H}_{\mathfrak{i}}$ in the above "Handling the Aggregation of All Hidden States" subsection.

Our Approach

The short answer to the above question is yes. In fact, ***one-layer autoregressive transformers are two RNNs interacting in parallel.*** Figure 3.6 illustrates this claim.

Given the index mapping m and the perturbed input $\mathbf{z}$ and denoting the sigmoid function as σ, we define

$$f_i = \begin{cases} v_i \sigma(q_n k_i - g_{i-1}) + f_{i-1}\sigma(g_{i-1} - q_n k_i), & i > 1 \\ v_1, & i = 1 \end{cases} \tag{3.20}$$

$$g_i = \begin{cases} \log(e^{g_{i-1}} + e^{q_n k_i}), & i > 1 \\ q_n k_1, & i = 1 \end{cases}. \tag{3.21}$$

Theorem 3.17. f_n *in Equation* (3.20) *is equivalent to* TRANS$(\mathbf{z}_{1:n}, m)$ *in Equation* (3.18).

Proof. We first prove the following equation of g_i by induction.

$$e^{g_i} = \sum_{j=1}^{i} e^{q_n k_j} \tag{3.22}$$

It is easy to see the base case ($i = 1$) holds. In the inductive step, suppose g_{i-1} holds for Equation (3.22), we have

$$e^{g_i} = e^{g_{i-1}} + e^{q_n k_i} = \sum_{j=1}^{i-1} e^{q_n k_j} + e^{q_n k_i} = \sum_{j=1}^{i} e^{q_n k_j}$$

We then prove the following equation of f_l by induction.

$$f_l = \sum_{i=1}^{l} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{l} e^{q_n k_j}} \tag{3.23}$$

It is easy to see the base case ($l = 1$) holds. In the inductive step, suppose f_{l-1} holds

for Equation (3.23), we have

$$
\begin{aligned}
f_l &= \sum_{i=1}^{l} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{l} e^{q_n k_j}} \\
&= v_l \frac{e^{q_n k_l}}{\sum_{j=1}^{l} e^{q_n k_j}} + \sum_{i=1}^{l-1} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{l} e^{q_n k_j}} \\
&= v_l \frac{e^{q_n k_l}}{\sum_{j=1}^{l-1} e^{q_n k_j} + e^{q_n k_l}} + \sum_{i=1}^{l-1} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{l-1} e^{q_n k_j}} \frac{\sum_{j=1}^{l-1} e^{q_n k_j}}{\sum_{j=1}^{l-1} e^{q_n k_j} + e^{q_n k_l}} \\
&= v_l \frac{e^{q_n k_l}}{e^{g_{l-1}} + e^{q_n k_l}} + \sum_{i=1}^{l-1} v_i \frac{e^{q_n k_i}}{\sum_{j=1}^{l-1} e^{q_n k_j}} \frac{e^{g_{l-1}}}{e^{g_{l-1}} + e^{q_n k_l}} && \text{(By Equation (3.22))} \\
&= v_l \frac{e^{q_n k_l}}{e^{g_{l-1}} + e^{q_n k_l}} + f_{l-1} \frac{e^{g_{l-1}}}{e^{g_{l-1}} + e^{q_n k_l}} && \text{(By the inductive hypothesis)} \\
&= v_l \sigma(q_n k_l - g_{l-1}) + f_{l-1} \sigma(g_{l-1} - q_n k_l)
\end{aligned}
$$

By the definition Equation (3.18), $\textsc{trans}(\mathbf{z}_{1:n}, m)$ is equivalent to f_n in Equation (3.23). □

Notes on the abstract transformers and the implementation We note that Equation (3.21) needs to be implemented as

$$g_i = \max(g_{i-1}, q_n k_i) + \text{log1p}(e^{-|g_{i-1} - q_n k_i|}),$$

for numerical stability.

When designing the abstract transformer of Equation (3.20), we propose rewriting f_i in two ways:

$$
\begin{aligned}
f_i &= (v_i - f_{i-1}) \sigma(q_n k_i - g_{i-1}) + f_{i-1} \\
f_i &= v_i + (f_{i-1} - v_i) \sigma(g_{i-1} - q_n k_i)
\end{aligned}
$$

We then meet these two intervals as $\widehat{f_i}$. However, notice that $\sigma(q_n k_i - g_{i-1}) + \sigma(g_{i-1} - q_n k_i) = 1$, indicating that using the Zonotope abstract domain to replace the interval abstract domain is a promising future direction (also see Bonaert et al. (2021)).

Rewriting a one-layer autoregressive transformer into two RNNs slows down the attention mechanism computation. Two RNNs need to compute the final states se-

Table 3.13: Results of applying ARC to autoregressive Transformer on SST2 dataset.

	$\{(\mathsf{T}_{Dup}, 2), (\mathsf{T}_{SubSyn}, 2)\}$		
Train	Acc.	CF Acc.	EX Acc.
Normal	79.90	9.06	63.32
Data Aug.	**80.51**	8.95	67.33
HotFlip	79.79	14.22	71.22
ARC	79.19	**64.80**	**72.71**

quentially, while the original attention computation in transformers leverages matrix multiplication, which is much faster than sequential computations of RNN states on GPUs.

Experiments

Our model employs a single-layer, decoder-only autoregressive transformer with two-head attention. Unlike conventional transformer layers, we omit layer normalization, leaving the abstract transformer for layer normalization as a potential avenue for future research. Classification is made by feeding the final state of the transformer layer into two MLP layers. We implemented a prototype of ARC on T_{Dup} and $(\mathsf{T}_{SubSyn}$ to demonstrate the preliminary results of the proposed approach[6].

We compare ARC with normal training, data augmentation, and HotFlip augmentation on the SST2 dataset against $\{(\mathsf{T}_{Dup}, 2), (\mathsf{T}_{SubSyn}, 2)\}$. Table 3.13 shows that ARC achieves the best certified accuracy and exhaustive accuracy when compared to the other three training approaches.

Compared to Table 3.6, although the normal accuracy of our transformer model is lower than that of the LSTM model, the exhaustive accuracy and certified accuracy of the transformer model are higher. This suggests that the transformer model can exhibit stronger robustness or be more amenable to verification than the LSTM model. Similar results have also been obtained by related work in randomized smoothing (Zhang et al., 2023a).

[6] `https://github.com/ForeverZyh/certified_lstms/tree/job-talk`

3.7 Future Work

We present ARC, which uses memoization and abstract interpretation to certify robustness to programmable perturbations for LSTMs. ARC can be used to train models that are more robust than those trained using existing techniques and handle more complex perturbation spaces. Last, the models trained with ARC have high certification accuracy, which can be certified using ARC itself. We also present an adversarial training technique, A3T, combining augmentation and abstraction techniques to achieve robustness against programmable string transformations in neural networks for NLP tasks. In the experiments, we showed that combining ARC and A3T yields more robust models than augmentation and abstraction techniques.

We foresee many future improvements to A3T and ARC. First, we manually split S into S_{aug} and S_{abs}. Performing the split automatically is left as future work.

Second, as mentioned in Section 3.6, we plan to explore the abstract transformer for layer normalization and replace the interval abstract domain with the Zonotope domain.

Third, as large language models for code tasks become prevalent, it is crucial to design programmable code perturbation following the same way of programmable perturbation space discussed in this chapter and to call for the verification of these large code models leveraging the techniques introduced in Section 3.6.

4 CERTIFIABLE DEFENCE AGAINST BACKDOOR ATTACKS

Deep learning models are vulnerable to *backdoor* poisoning attacks, where the attackers can *poison* a small fragment of the training set before model training and add *triggers* to inputs at test time. As a result, the prediction of the victim model that was trained on the poisoned training set will diverge in the presence of a trigger in the test input.

Effective backdoor attacks have been proposed for image recognition (Gu et al., 2017), sentiment analysis (Qi et al., 2021a), and malware detection (Severi et al., 2021). For example, the explanation-guided backdoor attack (XBA) (Severi et al., 2021) can break malware detection models as follows: The attacker *poisons* a small portion of benign software in the training set by modifying the values of the most important features so that the victim model recognizes these values as evidence of the benign prediction. At test time, the attacker inserts a *trigger* by changing the corresponding features of malware to camouflage it as benign software and making it bypass the examination of the victim model. Thus, backdoor attacks are of great concern to the security of deep learning models and systems that are trained on data gathered from different sources, e.g., via web scraping.

We identify two limitations of existing defenses to backdoor attacks. First, many existing approaches only provide empirical defenses that are specific to certain attacks and do not generalize to *all* backdoor attacks. Second, existing certified defenses—i.e., approaches that produce robustness certificates—are either unable to handle backdoor attacks, or are probabilistic (instead of deterministic), and therefore expensive and ineffective in practice.

Why certification? A defense against backdoor attacks should construct effective *certificates* (proofs) that the learned model can indeed defend against backdoor attacks. Empirical defenses do not provide certificates, can only defend against specific attacks, and can be bypassed by new unaccounted-for attacks. In Section 4.3, we show that existing empirical defenses cannot defend against XBA when only 0.1% of training data is poisoned. Certification has been successful at building models that are provably robust to *trigger-less* poisoning attacks and *evasion* attacks, but models trained to withstand such attacks are still weak against backdoor attacks. The *trigger-less* attack (Zhu et al., 2019a) assumes the attacker can poison the training set but cannot modify the test inputs, e.g., adding triggers, while the *evasion* attack (Madry et al., 2018) assumes the

attacker modifies the test inputs but cannot poison the training set. Existing certified defenses against trigger-less attacks, e.g., DPA (Levine and Feizi, 2021), and against evasion attacks, e.g., CROWN-IBP (Zhang et al., 2020a) and PatchGuard++ (Xiang and Mittal, 2021), cannot defend against backdoor attacks as they can either defend against the poison in the training data or the triggers at test time, but not both. As we show in the experiments, we can break these certified defenses using BadNets (Gu et al., 2017) and XBA (Section 4.3).

Why determinism? It is desirable for a certified defense to be *deterministic* because probabilistic defenses (Zhang et al., 2022b; Weber et al., 2020) typically require one to retrain thousands of models when performing predictions for a single test input. Retraining can be mitigated by Bonferroni correction, which allows reusing the trained models for a fixed number of predictions. However, retraining is still necessary after a short period, making it hard to deploy these defenses in practice. On the other hand, deterministic defenses (Levine and Feizi, 2021) can reuse the trained models an arbitrary number of times when producing certificates for different test inputs. Furthermore, probabilistic defenses for backdoor attacks, e.g., BagFlip (Zhang et al., 2022b), need to add noise to the training data, resulting in low accuracy for datasets that cannot tolerate too much noise when training (Section 4.3).

PECAN We present PECAN[1] (**P**artitioning data and **E**nsembling of **C**ertified neur**A**l **N**etworks), a deterministic certified defense against backdoor attacks for neural networks. PECAN can take *any* off-the-shelf technique for evasion certification and use it to construct a certified backdoor defense. This insight results in a simple modular implementation that can leverage future advances in evasion certification algorithms. Specifically, PECAN trains a set of neural networks on disjoint partitions of the dataset, and then applies evasion certification to the neural networks. By partitioning the dataset, we analytically bound the number of poisoned data seen per neural network; by employing evasion certification, we bound the number of neural networks that are robust in the face of triggers. Using this information, we efficiently derive a backdoor-robustness guarantee.

Figure 4.1 illustrates the workflow of PECAN. In Step 1, inspired by *deep partition aggregation* (Levine and Feizi, 2021), PECAN deterministically partitions a dataset into

[1] `https://github.com/ForeverZyh/defend_framework/tree/dev-DBA-malware`

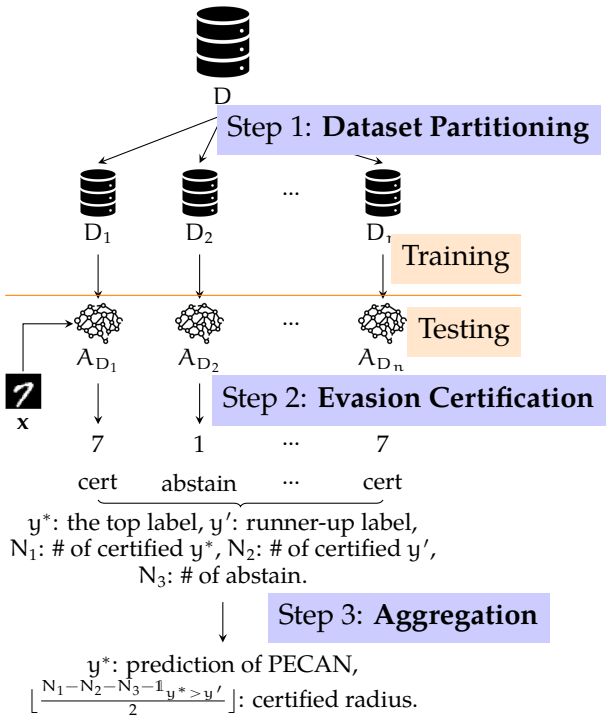


Figure 4.1: An overview of our approach PECAN.

multiple disjoint subsets. This step ensures that a poisoned data item only affects a single partition. In Step 2, PECAN trains an ensemble of neural networks, one on each partition. At test time, PECAN performs evasion certification to check which neural networks are immune to triggers; those that are not immune (or that cannot be proven immune) abstain from performing a prediction. Finally, in Step 3, PECAN aggregates the results of the ensemble and produces a prediction together with a robustness certificate: the percentage of the poisoned data in the training set that the training process can tolerate, the *certified radius*.

We evaluate PECAN on three datasets, MNIST, CIFAR10, and EMBER. PECAN outperforms or competes with BagFlip, which was proposed in our previous work (Zhang et al., 2022b) and it was the best certified defense against backdoor attacks at that time. Furthermore, BagFlip takes hours to compute the certificate, while PECAN takes seconds. We also evaluate PECAN against two known backdoor attacks, BadNets and XBA, PECAN reduces the average attack success rates from 90.24% to 0.67% and 66.37% to 2.19%, respectively.

4.1 Problem Definition

Given a dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, a (test) input $\mathbf{x}$, and a machine learning algorithm A, we write A_D to denote the machine learning model learned on dataset D by the algorithm A, and $A(D, \mathbf{x})$ to denote the output label predicted by the model A_D on input $\mathbf{x}$. We assume the algorithm will behave the same if trained on the same dataset across multiple runs. This assumption can be guaranteed by fixing the random seeds during training.

We are interested in certifying that if an attacker has poisoned the dataset, the model we have trained on the dataset will still behave "well" on the test input with maliciously added triggers. Before describing what "well" means, we need to define the *perturbation spaces* of the dataset and the test input, i.e., what possible changes the attacker could make to the dataset and the test input.

Perturbation space of the dataset Following DPA (Levine and Feizi, 2021), we define a *general* perturbation space over the dataset, allowing attackers to delete, insert, or modify training examples in the dataset. Given a dataset D and a *radius* $r \geqslant 0$, we define the *perturbation space* as the set of datasets that can be obtained by deleting or inserting up to r examples in D:

$$S_r(D) = \left\{ \widetilde{D} \mid |D \ominus \widetilde{D}| \leqslant r \right\},$$

where $A \ominus B$ is the symmetric difference of sets A and B. Intuitively, r quantifies how many examples need to be deleted or inserted to transform from D to $\widetilde{D}$.

Example 4.1. *If the attacker modifies one training example $\mathbf{x} \in D$ to another training example $\widetilde{\mathbf{x}}$ to form a poisoned dataset $\widetilde{D} = (D \setminus \{\mathbf{x}\}) \cup \{\widetilde{\mathbf{x}}\}$. Then $\widetilde{D} \in S_2(D)$ but $\widetilde{D} \notin S_1(D)$ because $S_r(D)$ considers one modification as one deletion and one insertion.*

Note that we assume a more general perturbation space of the training set than the ones considered by BagFlip and FPA; our work allows inserting and deleting examples instead of just modifying existing training examples.

Perturbation space of the test input We write $P(\mathbf{x})$ to denote the set of perturbed examples that an attacker can transform the example $\mathbf{x}$ into. Formally, the perturbation

space $P(\mathbf{x})$ can be defined as the l_p norm ball with radius s around the test input $\mathbf{x}$,

$$P(\mathbf{x}) = \{\widetilde{\mathbf{x}} \mid \|\mathbf{x} - \widetilde{\mathbf{x}}\|_p \leqslant s\}$$

Example 4.2. *An instantiation of* $P(\mathbf{x})$ *is the* l_0 *feature-flip perturbation* $F_s(\mathbf{x})$*, which allows the attacker to modify up to* s *features in an input* $\mathbf{x}$*,*

$$F_s(\mathbf{x}) = \{\widetilde{\mathbf{x}} \mid \|\mathbf{x} - \widetilde{\mathbf{x}}\|_0 \leqslant s\}$$

Dynamic backdoor attacks (Salem et al., 2022) like BadNets, which involve placing different patches in an input $\mathbf{x}$, with each patch having a maximum size of s, can be captured by $F_s(\mathbf{x})$. XBA, which modifies a predetermined set of features up to s features in an input $\mathbf{x}$, can also be captured by $F_s(\mathbf{x})$. Note that we assume a more general perturbation space of the test input than the one considered by FPA, which cannot handle dynamic backdoor attacks.

Threat models We consider backdoor attacks, where the attacker can perturb both the training set and the test input, but cannot control the training process of models. For the training set, we assume the attacker selects a poisoned training set from a perturbation space $S_r(D)$ of the training set D with a radius $r \geqslant 0$. For the test input, we assume the attacker selects a test input with a malicious trigger from a perturbation space $P(\mathbf{x})$ of the test input $\mathbf{x}$ with a given l_p norm and the radius s.

We say that an algorithm A is robust to a **backdoor attack** on a backdoored test input $\widetilde{\mathbf{x}}$ if the algorithm trained on any perturbed dataset $\widetilde{D}$ would predict the backdoored input $\widetilde{\mathbf{x}}$ the same as $A(D, \mathbf{x})$. Formally,

$$\forall \widetilde{D} \in S_r(D),\ \widetilde{\mathbf{x}} \in P(\mathbf{x}).\ A(\widetilde{D}, \widetilde{\mathbf{x}}) = A(D, \mathbf{x}) \tag{4.1}$$

Remark 4.3. *When* $r = 0$*, Eq 4.1 degenerates to evasion robustness, i.e.,* $\forall \widetilde{\mathbf{x}} \in P(\mathbf{x}).\ A(D, \widetilde{\mathbf{x}}) = A(D, \mathbf{x})$*, because* $S_0(D) = \{D\}$*.*

Given a large enough radius r, an attacker can always change enough inputs and succeed at breaking robustness. Therefore, we will typically focus on computing the maximal radius r for which we can prove that Eq 4.1 for given perturbation spaces $S_r(D)$ and $P(\mathbf{x})$. We refer to this quantity as the *certified radius*.

Certified guarantees We aim to design a certifiable algorithm A, which can defend against backdoor attacks, and to compute the certified radius of A. In our experiments (Section 4.3), we suppose a given benign dataset D and a benign test input $\mathbf{x}$, and we certifiably quantify the robustness of the algorithm A against backdoor attacks by computing the certified radius.

In Section 4.3, we also experiment with how the certifiable algorithm A defends backdoor attacks if a poisoned dataset $\widetilde{D}$ and a test input $\widetilde{\mathbf{x}}$ with malicious triggers are given, but the clean data is unknown. We theoretically show that we can still compute the certified radius if the clean data D and $\mathbf{x}$ are unknown in Section 4.2.

4.2 The PECAN Certification Technique

Our approach, which we call PECAN (**P**artitioning data and **E**nsembling of **C**ertified neur**A**l **N**etworks), is a deterministic certification technique that defends against backdoor attacks. Given a learning algorithm A, we show how to automatically construct a new learning algorithm $\bar{A}$ with certified backdoor-robustness guarantees (Eq 4.1) in Section 4.2. We further discuss how $\bar{A}$ can defend against a backdoored dataset and formally justify our discussion in Section 4.2.

Constructing Certifiable Algorithm $\bar{A}$

The key idea of PECAN is that we can take any off-the-shelf technique for evasion certification and use it to construct a certified backdoor defense. Intuitively, PECAN uses the evasion certification to defend against the possible triggers at test time, and it encapsulates the evasion certification in deep partition aggregation (DPA) (Levine and Feizi, 2021) to defend against training set poisoning.

Given a dataset D, a test input $\mathbf{x}$, and a machine learning algorithm A, PECAN produce a new learning algorithm $\bar{A}$ as described in the following steps (shown in Figure 2.4),

Dataset Partitioning We partition the dataset D into n disjoint sub-datasets, denoted as $D_1, \ldots, D_n$, using a hash function that deterministically maps each training example into a sub-dataset D_i. Train n classifiers $A_{D_1}, \ldots, A_{D_n}$ on these sub-datasets.

Evasion Certification We certify whether the prediction of each classifier A_{D_i} is robust under the perturbation space $P(\mathbf{x})$ by any evasion certification approach for the learning algorithm, e.g., CROWN-IBP for neural networks (Xu et al., 2020). Formally, the certification approach determines whether the following equation holds,

$$\forall \widetilde{\mathbf{x}} \in P(\mathbf{x}).\ A(D_i, \mathbf{x}) = A(D_i, \widetilde{\mathbf{x}}) \tag{4.2}$$

We denote the output of each certification as $A^P_{D_i}(\mathbf{x})$, which can either be $A^P_{D_i}(\mathbf{x}) = \text{cert}$, meaning Eq 4.2 is certified. Otherwise, $A^P_{D_i}(\mathbf{x}) = \text{abstain}$, meaning the certification approach cannot certify Eq 4.2.

Aggregation We compute the top label y^* by aggregating all predictions from $A(D_i, \mathbf{x})$. Concretely, $y^* \triangleq \underset{y \in \mathcal{C}}{\text{argmax}} \sum_{i=1}^{n} \mathbb{1}_{A(D_i,\mathbf{x})=y}$, where $\mathcal{C} = \{0, 1, \ldots\}$ is the set of possible labels. Note that if a tie happens when taking the argmax, we break ties deterministically by setting the smaller label index as y^*. We denote the runner-up label as y' as $\underset{y \in \mathcal{C} \wedge y \neq y^*}{\text{argmax}} \sum_{i=1}^{n} \mathbb{1}_{A(D_i,\mathbf{x})=y}$. We count the number of certified predictions equal to y^* as N_1, the number of certified predictions equal to y' as N_2, and the number of abstentions as N_3. We formally defining N_1, N_2, and N_3 as

$$N_1 = \sum_{i=1}^{n} \mathbb{1}_{A(D_i,\mathbf{x})=y^* \wedge A^P_{D_i}(\mathbf{x})=\text{cert}}$$
$$N_2 = \sum_{i=1}^{n} \mathbb{1}_{A(D_i,\mathbf{x})=y' \wedge A^P_{D_i}(\mathbf{x})=\text{cert}}$$
$$N_3 = \sum_{i=1}^{n} \mathbb{1}_{A^P_{D_i}(\mathbf{x})=\text{abstain}}.$$

We set the prediction $\bar{A}(D, \mathbf{x})$ as y^*. We compute the certified radius r in the following two cases. If $N_1 - N_2 - N_3 - \mathbb{1}_{y^* > y'} < 0$, we set r as $\diamond$, i.e., a value denoting no certification. In this case, PECAN cannot certify that $\bar{A}$ is robust to evasion attacks even if the dataset is not poisoned. Otherwise, we compute r as $\lfloor \frac{N_1 - N_2 - N_3 - \mathbb{1}_{y^* > y'}}{2} \rfloor$. A special case is $r = 0$, when PECAN can certify $\bar{A}$ is robust to evasion attacks, but cannot certify that it is robust if the dataset is poisoned.

We note that the computation of the certified radius is equivalent to DPA when no classifier abstains, i.e., $N_3 = 0$,

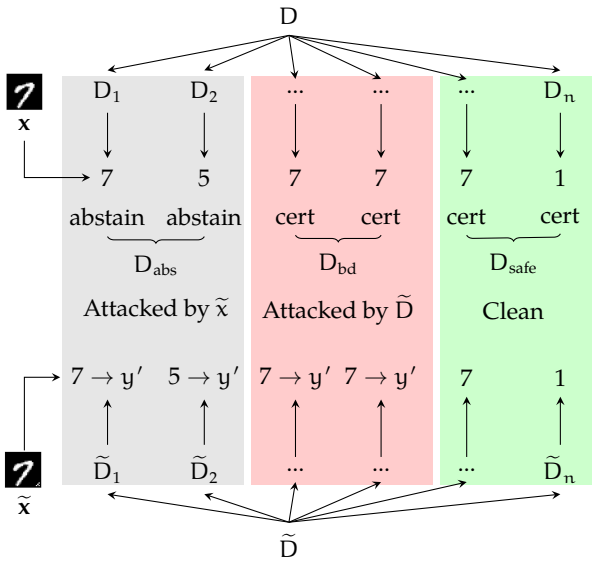


Figure 4.2: An illustration of the proof of Theorem 4.4. It shows the worst case for PECAN, where the attacker can change all predictions in D_{abs} and D_{bd} to the runner-up label y'. Note that we group D_{abs}, D_{bd}, and D_{safe} together to ease illustration.

Theorem 4.4 (Soundness of PECAN)**.** *Given a dataset* D *and a test input* $\mathbf{x}$*, PECAN computes the prediction* $\bar{A}(D, \mathbf{x})$ *and the certified radius as* r*. Then, either* $r = \diamond$ *or*

$$\forall \widetilde{D} \in S_r(D),\ \widetilde{\mathbf{x}} \in P(\mathbf{x}).\ \bar{A}(\widetilde{D}, \widetilde{\mathbf{x}}) = \bar{A}(D, \mathbf{x}) \tag{4.3}$$

We outline a proof sketch of Theorem 4.4 before providing the main proof. The key idea is that if we can prove that a majority of classifiers are immune to poisoned data, we can prove that the aggregated result is also immune to poisoned data. We start by lower bounding the number of classifiers that are immune to poisoned data as $N_1 - r$ and upper bounding the number of classifiers that can be manipulated by the attackers as $N_2 + r + N_3$. Then, we show that if $r \leqslant \lfloor \frac{N_1 - N_2 - N_3 - \mathbb{1}_{y^* > y'}}{2} \rfloor$, the classifiers immune to poisoned data will always be a majority.

Proof. For any poisoned dataset $\widetilde{D}$, we partition $\widetilde{D}$ into n sub-datasets $\{\widetilde{D}_1, \ldots, \widetilde{D}_n\}$ according to $\{D_1, \ldots, D_n\}$ from the clean dataset D. Note that we can determine such a correspondence between D_i and $\widetilde{D}_i$ because our hash function is deterministic and only depends on each training example. We further divide $\{D_1, \ldots, D_n\}$ into three disjoint parts D_{abs}, D_{bd}, and D_{safe} in the following way,

- $D_{abs} = \{D_i \mid A^P_{D_i}(\mathbf{x}) = \text{abstain}\}$ are the sub-datasets, on which A abstains from making the prediction on $\mathbf{x}$. From the definition of N_3, we have $|D_{abs}| = N_3$.

Intuitively, D_{abs} contains the sub-datasets that can possibly be attacked by the test input $\widetilde{\mathbf{x}}$ with malicious triggers.

- D_{bd} are the sub-datasets on which A does not abstain and are also poisoned, i.e., each of them has at least one training example removed or inserted. Even though we do not know the exact sub-datasets in D_{bd}, we know $|D_{bd}| \leqslant r$ because $\widetilde{D} \in S_r(D)$ constrains that there are at most r such poisoned sub-datasets.
- $D_{safe} = \{D_i \mid D_i = \widetilde{D}_i \wedge A^P_{D_i}(\mathbf{x}) = \text{cert}\}$ contains the clean sub-datasets, on which A does not abstain.

We denote the numbers of the original top prediction y^* and the original runner-up prediction y' on the backdoored data $\widetilde{D}$ and $\widetilde{\mathbf{x}}$ as $\widetilde{N}_{y^*}$ and $\widetilde{N}_{y'}$, respectively. Formally,

$$\widetilde{N}_{y^*} = \sum_{i=1}^{n} \mathbb{1}_{A(\widetilde{D}_i,\widetilde{\mathbf{x}})=y^*}, \quad \widetilde{N}_{y'} = \sum_{i=1}^{n} \mathbb{1}_{A(\widetilde{D}_i,\widetilde{\mathbf{x}})=y'}$$

Next, we prove Eq 4.3 for any backdoored data $\widetilde{D}$ and $\widetilde{\mathbf{x}}$ by showing that

$$\widetilde{N}_{y^*} \geqslant \widetilde{N}_{y'} + \mathbb{1}_{y^*>y'} \tag{4.4}$$

We prove Eq 4.4 by showing a lower bound of $\widetilde{N}_{y^*}$ is $N_1 - r$ and an upper bound of $\widetilde{N}_{y'}$ is $N_2 + r + N_3$. Together with the definition of r, we can prove Eq 4.4 because we have,

$$\begin{aligned}
&\widetilde{N}_{y^*} - \widetilde{N}_{y'} - \mathbb{1}_{y^*>y'} \\
\geqslant &N_1 - r - (N_2 + r + N_3) - \mathbb{1}_{y^*>y'} \\
= &N_1 - N_2 - 2r - N_3 - \mathbb{1}_{y^*>y'} \\
= &N_1 - N_2 - 2\lfloor \frac{N_1 - N_2 - N_3 - \mathbb{1}_{y^*>y'}}{2} \rfloor - N_3 - \mathbb{1}_{y^*>y'} \\
\geqslant &N_1 - N_2 - (N_1 - N_2 - N_3 - \mathbb{1}_{y^*>y'}) - N_3 - \mathbb{1}_{y^*>y'} \\
= &0.
\end{aligned}$$

Note that the second last line holds iff $N_1 - N_2 - N_3 - \mathbb{1}_{y^*>y'} \geqslant 0$. Otherwise, we have $r = \diamond$.

As shown in Figure 4.2, the lower bound of $\widetilde{N}_{y^*}$ can be computed by noticing that 1) the attacker can change any prediction in D_{bd} from y^* to another label because these

datasets are poisoned, 2) the attacker can change any prediction in D_{abs} to another label because CROWN-IBP cannot certify the prediction under the evasion attacks, and 3) the attacker cannot change anything in D_{safe} because of the guarantee of CROWN-IBP and D_{safe} is not poisoned,

$$\forall D_i \in D_{safe}, \widetilde{\mathbf{x}} \in P(\mathbf{x}).\ A(D_i, \mathbf{x}) = A(D_i, \widetilde{\mathbf{x}}) = A(\widetilde{D}_i, \widetilde{\mathbf{x}})$$

The upper bound of $\widetilde{N}_{y'}$ can be computed by noticing that 1) the attacker can change any prediction in D_{bd} to y', 2) the attacker can change any prediction in D_{abs} to y', and 3) the attacker cannot change anything in D_{safe}.

We complete the proof by showing that the best attack strategy of the attacker is to change the prediction of $\bar{A}$ to the runner-up label y'. If the attacker chooses to change the prediction of $\bar{A}$ to another label y'', denoted the counts as $\widetilde{N}_{y''}$, then the upper bound of $\widetilde{N}_{y''}$ will be always smaller or equal to $\widetilde{N}_{y'}$. □

PECAN under the Backdoored Data

The above algorithm and proof of PECAN assume that a clean dataset D and a clean test example $\mathbf{x}$ are already given. However, we may be interested in another scenario where the poisoned dataset $\widetilde{D} \in S_r(D)$ and the input example $\widetilde{\mathbf{x}} \in P(\mathbf{x})$ with malicious triggers are given, and the clean data D and $\mathbf{x}$ are unknown. In other words, we want to find the maximal radius r such that $\bar{A}(\widetilde{D}, \widetilde{\mathbf{x}}) = \bar{A}(D, \mathbf{x})$ for any D and $\mathbf{x}$ that can be perturbed to $\widetilde{D}$ and $\widetilde{\mathbf{x}}$ by the perturbation S_r and P, respectively. Formally,

$$\forall D, \mathbf{x}.\ \widetilde{D} \in S_r(D) \wedge \widetilde{\mathbf{x}} \in P(\mathbf{x}) \implies \bar{A}(\widetilde{D}, \widetilde{\mathbf{x}}) = \bar{A}(D, \mathbf{x}) \tag{4.5}$$

Intuitively, Eq 4.5 is the symmetrical version of Eq 4.1. Owing to the symmetrical definition of S_r and P, if we apply PECAN to the given poisoned data $\widetilde{D}, \widetilde{\mathbf{x}}$, then the prediction $\bar{A}(\widetilde{D}, \widetilde{\mathbf{x}})$ and the certified radius r satisfy the certified backdoor-robustness guarantee (Eq 4.5). The following theorem formally states the soundness of PECAN under the backdoored data.

Theorem 4.5 (Soundness of PECAN under backdoored data)**.** *Given a dataset $\widetilde{D}$ and a test input $\widetilde{\mathbf{x}}$, PECAN computes the prediction $\bar{A}(\widetilde{D}, \widetilde{\mathbf{x}})$ and the certified radius r. Then, either $r = \diamond$ or Eq 4.5 holds.*

Proof. Theorem 4.4 tells that either $r = \diamond$ or the following equation holds,

$$\forall D' \in S_r(\widetilde{D}),\ \mathbf{x}' \in P(\widetilde{\mathbf{x}}).\ \bar{A}(\widetilde{D}, \widetilde{\mathbf{x}}) = \bar{A}(D', \mathbf{x}') \tag{4.6}$$

By the symmetrical definition of S_r and P, we have

$$\forall D.\ \widetilde{D} \in S_r(D) \implies D \in S_r(\widetilde{D}) \tag{4.7}$$

$$\forall \mathbf{x}.\ \widetilde{\mathbf{x}} \in P(\mathbf{x}) \implies \mathbf{x} \in P(\widetilde{\mathbf{x}}). \tag{4.8}$$

Then, for all possible clean data D and $\mathbf{x}$, we have

$$\begin{aligned} & \widetilde{D} \in S_r(D) \wedge \widetilde{\mathbf{x}} \in P(\mathbf{x}) \\ \implies & D \in S_r(\widetilde{D}) \wedge \mathbf{x} \in P(\widetilde{\mathbf{x}}) && \text{(By Eq 4.7 and Eq 4.8)} \\ \implies & \bar{A}(\widetilde{D}, \widetilde{\mathbf{x}}) = \bar{A}(D, \mathbf{x}) && \text{(By Eq 4.6)} \end{aligned}$$

□

4.3 Experiments

We implemented PECAN in Python and provided the implementation in the supplementary materials.

In Section 4.3, we evaluate the effectiveness and efficiency of PECAN by comparing it to BagFlip (Zhang et al., 2022b), the state-of-the-art probabilistic certified defense against backdoor attacks. We use CROWN-IBP, implemented in auto-LiRPA (Xu et al., 2020), as the evasion defense approach in this setting. Whenever we use CROWN-IBP for the evasion defense approach, we also use it to train the classifiers in the dataset-partitioning step since the classifiers trained by CROWN-IBP can improve the certification rate in the evasion-certification step.

In Section 4.3, we evaluate the effectiveness of PECAN under the patch attack using BadNets (Gu et al., 2017) for image classification and the explanation-guided backdoor attack (XBA) (Severi et al., 2021) for malware detection and compare PECAN to other approaches. We use PatchGuard++ (Xiang and Mittal, 2021) as the evasion defense approach for image classification and use CROWN-IBP as the evasion defense approach for malware detection.

Experimental Setup

Datasets We conduct experiments on MNIST, CIFAR10, CIFAR10-02, and EMBER (Anderson and Roth, 2018) datasets. CIFAR10-02 (Weber et al., 2020) is a subset of CIFAR10, comprising examples labeled as 0 and 2. It consists of 10,000 training examples and 2,000 test examples. EMBER is a malware detection dataset containing 600,000 training and 200,000 test examples. Each example is a vector containing 2,351 features of the software, e.g., number of sections and number of writeable sections.

Models When comparing PECAN and BagFlip, we train fully connected neural networks with four layers for MNIST and CIFAR10 datasets. For experiments using PatchGuard++, we use the BagNet (Brendel and Bethge, 2019) model structure used by PatchGuard++. We use the same fully connected neural network for EMBER as in related works (Zhang et al., 2022b; Severi et al., 2021). We use the same data augmentation for PECAN and other baselines.

Metrics For each test input $\mathbf{x}_i, y_i$, the PECAN will predict a label and the certified radius r_i. In this section, we assume that the attacker *modified* R% examples in the training set. We denote R as the *modification amount*. We summarize all the used metrics as follows:
Certified Accuracy denotes the percentage of test examples that are correctly classified and whose certified radii are no less than R, i.e., $\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\bar{A}(D,\boldsymbol{x}_i)=y_i \wedge \frac{r_i}{|D|} \geqslant 2R\%}$, where m and $|D|$ are the sizes of test and training set, respectively. Notice that there is a factor of 2 on the modification amount R because $S_r(D)$ considers one modification as one insertion and one deletion, as in Example 4.1.
Normal Accuracy denotes the percentage of test examples that are correctly classified by the algorithm *without* certification, i.e., $\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\bar{A}(D,\boldsymbol{x}_i)=y_i}$.
Attack Success Rate (*ASR*) We are interested in how many test examples are originally correctly classified without the malicious trigger but wrongly classified after adding the trigger, i.e., $\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\bar{A}(D,\widetilde{\boldsymbol{x}}_i)\neq y_i \wedge \bar{A}(D,\boldsymbol{x}_i)=y_i \wedge \frac{r_i}{|D|} \geqslant 2R\%}$, where $\mathbf{x}_i$ is the original test and $\widetilde{\mathbf{x}}_i$ is with a malicious trigger.
Abstention Rate is computed as $\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\frac{r_i}{|D|} < 2R\%}$.

Effectiveness and Efficiency of PECAN

We evaluate the effectiveness and efficiency of PECAN on MNIST, CIFAR10, and EMBER under the backdoor attack with the l_0 feature-flip perturbation F_1, which allows the attacker to modify up to one feature in an example. We compare PECAN to BagFlip, the state-of-the-art probabilistic certified defense against l_0 feature-flip backdoor attacks.

Summary of the results PECAN achieves significantly higher certified accuracy than BagFlip on CIFAR10, EMBER, and MNIST. PECAN has similar normal accuracy as BagFlip for all datasets. PECAN is more efficient than BagFlip at computing the certified radius.

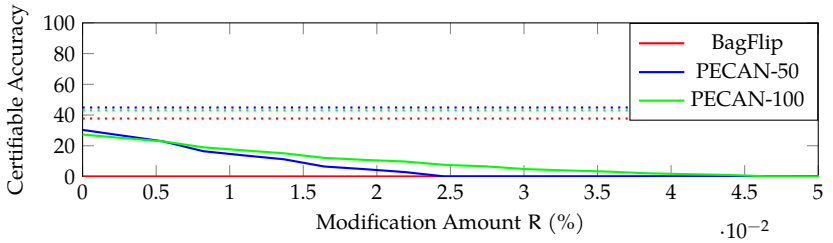
Setup For PECAN, we vary n, the number of partitions, to ensure a fair comparison with BagFlip, whose bag size is set to 200, 3000, and 2000 for MNIST, EMBER, and CIFAR10 datasets, respectively. We denote PECAN with different settings of n as PECAN-n.

BagFlip achieves meaningful results only on MNIST, where we tune the parameter n of PECAN to 3000 to achieve the same normal accuracy of BagFlip and compare their results following the practice by Jia et al. (2020, 2021).

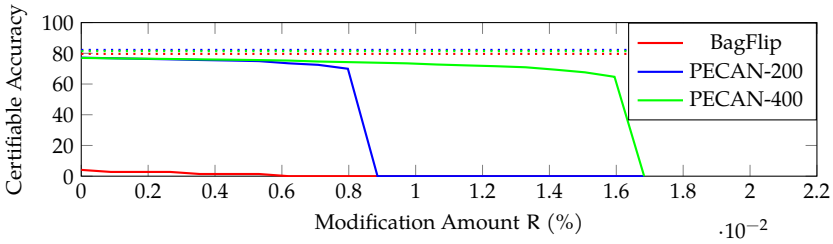
Results Figure 4.3 shows the comparison between PECAN and BagFlip on CIFAR10, EMBER, and MNIST. **PECAN achieves significantly higher certified accuracy than BagFlip across all modification amounts R and normal accuracy similar to BagFlip for all datasets.**

BagFlip performs poorly on CIFAR10 and EMBER because these two datasets cannot tolerate the high level of noise that the BagFlip algorithm adds to the training data. A high level of noise is crucial to establish meaningful bounds by BagFlip. For example, BagFlip can add 20% noise to the training data of MNIST, i.e., a feature (pixel) in a training example will be flipped to another value with 20% probability. However, for CIFAR10 and EMBER, BagFlip has to reduce this probability to 5% to maintain normal accuracy, resulting in a low certified accuracy.

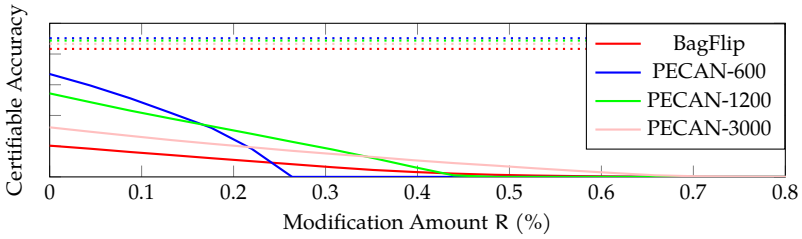
Figure 4.3 (c) shows the comparison between PECAN and BagFlip on MNIST. PECAN-3000 achieves higher certified accuracy than BagFlip across all modification amounts R. When comparing PECAN-600 and PECAN-1200 with BagFlip, we find that



(a) CIFAR10 F_1



(b) EMBER F_1



(c) MNIST F_1

Figure 4.3: Comparison to BagFlip on CIFAR10, EMBER and MNIST, showing the normal accuracy (dotted lines) and the certified accuracy (solid lines) at different modification amounts R.
PECAN-600 and PECAN-1200 achieve higher certified accuracy than BagFlip when $R \in [0, 0.23]$ and $R \in [0, 0.42]$, respectively.

In the Appendix C of PECAN (Zhang et al., 2023b), we also evaluate the effectiveness of PECAN against the perturbation space with the l_∞ norm. The results show that PECAN achieves certified accuracy similar to F_1 as shown in Figure 4.3.

PECAN is more efficient than BagFlip at computing the certified radius. PECAN computes the certified radius in constant time via the closed-form solution in the aggregation step. In our experiment on MNIST, BagFlip requires 8 hours to prepare a lookup table because BagFlip does not have a closed-form solution for computing the certified radius.

Training time of PECAN. Like other certified approaches like BagFlip and DPA, PECAN requires training n classifiers. On the MNIST dataset, PECAN took 5.0, 5.9, and 10.5 hours to train 600, 1200, and 3000 models, respectively, using a single V-100

GPU. For the CIFAR-10 dataset, PECAN trained 50 and 100 models in 0.6 hours. On the EMBER dataset, it took 1.3 and 0.6 hours to train 200 and 400 models, respectively. The training process for PECAN incurs a one-time cost and can be amortized across multiple predictions, making it a feasible approach for practical applications.

PECAN under Backdoored Data

In Section 4.3, we assess the efficacy of PECAN in image classification using the CIFAR10-02 and MNIST datasets under the BadNets (Gu et al., 2017), which introduces patches as triggers to implant backdoors into models. We compare PECAN with Friendly Noise (Liu et al., 2022a), the state-of-the-art empirical defense against general data poisoning attacks on image classification datasets. Subsequently, we evaluate two certified defenses: DPA, a defense against trigger-less attacks, and PatchGuard++, a defense against evasion patch attacks. Finally, we assess FPA, a certified defense against backdoor attacks.

In Section 4.3, we evaluate five empirical defenses—Isolation Forests (Liu et al., 2008), HDBSCAN (Murtagh and Contreras, 2012), Spectral Signatures (Tran et al., 2018), MDR (Wang et al., 2022c), and Friendly Noise—in the context of malware detection using the EMBER dataset under the explanation-guided backdoor attack (XBA) (Severi et al., 2021). The first three defenses are proposed as adaptive defenses in XBA, while MDR is currently the state-of-the-art empirical defense against backdoor attacks on the EMBER dataset. Additionally, we assess two certified defenses: DPA, FPA, and CROWN-IBP, a defense against evasion attacks.

We do not compare to BagFlip because: (1) BadNets generates large patches beyond BagFlip's defending ability, resulting in a certified accuracy of zero, and (2) BagFlip's certified accuracy on EMBER is poor, as shown in Figure 4.3 (b).

Summary of the results PECAN reduces the ASR of the victim model (NoDef) on backdoored test sets from an average of 90.24% to 0.67% under BadNets and from 66.37% to 2.19% under XBA, while other approaches fail to defend against the backdoor attacks, with the exception of FPA, which performs well under the XBA attack.

Table 4.1: Results on poisoned dataset generated by BadNets and evaluated on test sets with triggers and clean test sets. We report the standard error of the mean in parentheses. We note that NoDef and other empirical methods do not have abstention rates.

		Test set with triggers			Clean test set	
	Approaches	ASR (↓)	Accuracy (↑)	Abstention Rate	Accuracy (↑)	Abstention Rate
CIFAR10-02	NoDef	95.54% (5.32)	4.46% (5.32)	N/A	89.74% (0.90)	N/A
	Friendly Noise	16.74% (2.25)	83.26% (2.25)	N/A	86.02% (0.25)	N/A
	PatchGuard++	1.64% (0.47)	0.30% (0.37)	98.07% (0.50)	77.18% (2.16)	18.33% (2.37)
	DPA	6.21%	24.60%	69.19%	70.00%	23.45%
	FPA	0.00%	0.00%	100.0%	0.00%	100.0%
	PECAN	0.87%	22.86%	77.27%	69.60%	23.95%
MNIST	NoDef	84.93% (6.85)	15.07% (6.85)	N/A	92.61% (2.79)	N/A
	Friendly Noise	11.41% (5.32)	88.59% (5.32)	N/A	85.33% (4.09)	N/A
	PatchGuard++	0.76% (0.45)	4.77% (0.80)	94.47% (1.07)	38.71% (3.77)	60.84% (3.45)
	DPA	0.46%	80.93%	18.61%	74.92%	20.31%
	FPA	0.00%	0.00%	100.0%	0.00%	100.0%
	PECAN	0.46%	80.93%	18.61%	74.92%	20.31%

Experiments with BadNets

Setup We inject backdoors into 0.2% of the CIFAR10-02 and MNIST training sets using BadNets. For the CIFAR10-02 dataset, a 8×8 backdoored patch is added to images labeled with 1, aiming to mislead the victim model into predicting these images as label 0 instead of 1. For the MNIST dataset, nine different backdoored patches with sizes 4×4 are added at various locations to images labeled with 1-9. The objective is to deceive the victim model into predicting the backdoored images as the digit 0. Note that we do not change labels of the poisoned training data.

Our hyperparameter tuning strategy of Friendly Noise follows the original paper. The number of partitions in FPA is set to 256 and 56 for CIFAR10-02 and MNIST datasets, respectively. We train PECAN using the following hyperparameters: for the CIFAR10-02 dataset, we set $n = 100$, use 200 epochs, a learning rate of $5e-4$, and a weight decay of $5e-1$. When training on the MNIST dataset, we set $n = 600$, use 900 epochs, a learning rate of $5e-4$, and a weight decay of $5e-2$. For PatchGuard++ and PECAN, the hyperparameter τ, which controls the trade-off between ASR and abstention rate, is set to maximize the difference between accuracy and ASR. For the CIFAR10-02 dataset, we set $\tau = 0.99$ and $\tau = 0$ for PECAN and PatchGuard++, respectively. For the MNIST datsaet, we set $\tau = 1$ and $\tau = 0$ for PECAN and PatchGuard++, respectively. We set $\tau = 0$ for PatchGuard++ because no matter what τ is, there will always be a

considerable ASR.

We run ensemble-based methods PECAN, DPA, and FPA only once because they inherently exhibit low variance. For other approaches, we run them five times and report the average results.

Comparison to Friendly Noise Friendly Noise, the state-of-the-art empirical defense, fails to defend against the BadNets attack, showing high ASRs of 16.74% and 11.41% on CIFAR10-02 and MNIST datasets, respectively. In contrast, PECAN achieves much lower ASRs of 0.87% and 0.46%. Compared to PECAN, Friendly Noise achieves higher accuracies on both test sets as PECAN is a certified defense and has high abstention rates.

Comparison to DPA and PatchGuard++ DPA fails to defend against the BadNets attack on the CIFAR10-02 dataset with an ASR of 6.21%. The result of DPA is the same as PECAN in the MNIST dataset because DPA is equivalent to PECAN when its hyperparameter τ is set to 1. We hypothesize that in this specific case, the backdoor attack is not strong enough; thus, the effect of test-time evasion does not manifest, leading to the same results for DPA and PECAN. PatchGuard++ shows high ASRs across all values of $\tau \in [0, 1]$, with optimal performance achieved when setting τ to 0, leading to high abstention rates.

Comparison to FPA FPA is unsuitable for defending against the BadNets attack due to the large patch sizes, limiting each classifier in FPA to consider only 4 and 14 pixels for each input in CIFAR10-02 and MNIST datasets. This limited visibility of pixels results in each classifier in FPA merely guessing predictions. Additionally, FPA cannot defend against dynamic backdoor attacks that can place patches at different locations.

Experiments with XBA

Setup We use XBA to backdoor 0.1% of the training set and add triggers into the malware in the test set. We aim to fool the victim model to predict the malware with malicious triggers as non-malware. We generate a poisoned dataset $\widetilde{D}_3$ and its corresponding test set with triggers by perturbation F_3, which allows the attacker to modify up to three features in an example. As shown in Table 4.2, XBA achieves a 67.16% ASR with only three features modified.

Table 4.2: Results on poisoned dataset $\widetilde{D}_3$ when evaluated on the malware test set with triggers and the clean test set.

	Malware test set with triggers			Clean test set	
Approaches	ASR (↓)	Accuracy (↑)	Abstention Rate	Accuracy (↑)	Abstention Rate
NoDef	67.16% (11.21)	32.84% (11.21)	N/A	98.37% (0.28)	N/A
Isolation Forest	28.74% (8.71)	71.26% (8.71)	N/A	94.16% (0.24)	N/A
HDBSCAN	63.47% (11.85)	36.53% (11.85)	N/A	98.11% (0.37)	N/A
Spectral Signature	67.07% (16.63)	32.93% (16.63)	N/A	98.11% (0.40)	N/A
MDR	63.56% (10.98)	36.44% (10.98)	N/A	98.27% (0.21)	N/A
Friendly Noise	58.76% (1.57)	41.24% (1.57)	N/A	95.09% (0.58)	N/A
PECAN-Empirical	25.47%	74.53%	N/A	89.15%	N/A
CROWN-IBP	6.64% (2.25)	28.87% (2.23)	64.49% (2.29)	62.04% (0.85)	32.27% (1.26)
DPA	33.91%	41.89%	24.20%	79.06%	5.20%
FPA	0.72%	34.28%	65.00%	76.41%	23.38%
PECAN	2.19%	29.46%	68.35%	42.44%	56.42%

To account for modifying fewer training examples than the original paper, we reduced the minimal cluster size for Isolation Forest and HDBSCAN from 0.5% to 0.05%. For MDR, we enumerated a set $\{4, 5, 6, 7, 8\}$ and set the threshold for building the graph to 7. For Friendly Noise, we followed the grid search for hyper-parameters (the friendly noise learning rate lr and μ) in their paper (Liu et al., 2022a) and set both the noise_eps and friendly_clamp to 16 for CIFAR10-02, and to 8 for MNIST, and to 32 for EMBER dataset. The best hyper-parameters sets for CIFAR10-02, MNIST, and EMBER datasets are achieved at $(lr = 10, \mu = 10)$, $(lr = 100, \mu = 1)$, and $(lr = 50, \mu = 10)$, respectively. For PECAN and DPA, we set the number of partitions n to 3000 and present their results for the modification amount $R = 0.1\%$. As CROWN-IBP does not consider R, we show its results against the perturbation F_3 regardless of R.

Comparison to Empirical Defenses PECAN can defend against the backdoor attack on the EMBER dataset, but Isolation Forest, HDBSCAN, Spectral Signature, and MDR fail to defend against the attack. Table 4.2 shows that PECAN successfully reduces the ASR of the victim model from 67.16% to 2.19% on the malware test set with triggers. PECAN has the lowest ASR compared to other empirical defenses, in which the best empirical defense, Isolation Forest, has an ASR of 28.74%.

We found that these empirical approaches struggled to filter out poisoning examples in the training set due to the small amount of poisoning—only 0.1%. Isolation Forest, HDBSCAN, Spectral Signature, and MDR incorporate filtering mechanisms to remove poisoning examples from the training set, aiming to defend against poisoning

attacks.

- Isolation Forest removes 73,961 training examples, of which 146 examples are poisoned and 73,815 are false positives. **The precision is 0.20%, and the recall is 24.33%.**
- HDBSCAN removes 122,323 training examples, of which 204 examples are poisoned, and 122,119 are false positives. **The precision is 0.17%, and the recall is 34.00%.**
- Spectral Signatures remove 5000 training examples, of which 134 examples are poisoned, and 4866 are false positives. **The precision is 2.68%, and the recall is 22.33%.**
- MDR removes 647 training examples, all of which are false positives, resulting in no poisoned examples being removed. **The precision is 0%, and the recall is 0%.**

These results demonstrate that while the empirical defenses can filter out some poisoned examples, they also have a high number of false positives, leading to low precision and recall values. The effectiveness of these defenses varies, with HDBSCAN achieving the highest recall and Spectral Signatures having the highest precision among the four methods evaluated.

PECAN has a high abstention rate because PECAN is a certified defense against backdoor attacks. However, even if we make PECAN an empirical defense by producing an output even when certification fails, we find PECAN-Empirical still achieves an ASR of 25.47% and an accuracy of 74.53%, which still outperforms the best empirical approach Isolation Forest, with an ASR 28.74%.

Comparison to DPA and CROWN-IBP The ASR of DPA and CROWN-IBP on the malware test set with triggers are 33.91% and 6.64% meaning that many malware with triggers can bypass their defenses.

Comparison to FPA FPA defends against backdoor attacks by adopting a feature partitioning-approach, making it effective against attacks that introduce a fixed and small trigger across all examples, especially on tabular datasets where useful information is preserved after partitioning. As a result, FPA achieves a lower ASR of 0.72% than PECAN on the EMBER dataset under the XBA attack.

However, FPA has limitations when defending against attacks with medium-size dynamic triggers, such as the BadNets attack employed in Section 4.3, or on image datasets where features are significantly disrupted after partitioning. Additionally, FPA cannot defend against attacks that involve modifications to training labels or the insertion/removal of training examples. In contrast, PECAN does not have these limitations because its perturbation space captures all such data poisoning attacks. Consequently, PECAN successfully defends against both BadNets and XBA attacks.

In conclusion, while FPA demonstrates effectiveness on fixed and small triggers, particularly on tabular datasets like EMBER under the XBA attack, it faces challenges against dynamic triggers and disruptions in image datasets. On the other hand, PECAN is more versatile against a broader range of data poisoning attacks.

4.4 Related Work

Deep learning models are vulnerable to backdoor attacks (Saha et al., 2020; Turner et al., 2019), and empirical defenses (Geiping et al., 2021; Liu et al., 2018) can be bypassed (Wang et al., 2020b; Koh et al., 2022). Hence, our focus on building a certified defense.

Certified defenses against backdoor attacks Existing certification approaches provide probabilistic certificates by extending randomized smoothing (Cohen et al., 2019), originally proposed to defend against adversarial evasion attacks, to defend against backdoor attacks. BagFlip (Zhang et al., 2022b) is the state-of-the-art model-agnostic probabilistic defense against feature-flipping backdoor attacks at that time. Wang et al. (2020a); Weber et al. (2020) proposed backdoor-attack defenses that are also model-agnostic, but are less effective than BagFlip. PECAN is deterministic and therefore less expensive and more effective than these defenses. Probabilistic defenses are model-agnostic; while PECAN is evaluated on neural networks, it can work for any machine learning model as that supports a deterministic evasion certification approach. Jia et al. (2020) proposed a deterministic de-randomized smoothing approach for kNN classifiers. Their approach computes the certificates using an expensive dynamic programming algorithm, whereas PECAN's certification algorithm has constant time complexity. XRand (Nguyen et al., 2022) presents a certified defense against XBA by leveraging differential privacy. Unlike PECAN, XRand adopts a different attack model,

and its certifications are probabilistic in nature. FPA (Hammoudeh and Lowd, 2023) employs a deterministic certified defense mechanism that partitions the feature space instead of the dataset. We discuss the limitation of FPA and compare it with PECAN in Section 4.3.

Certified backdoor attack detection CBD (Xiang et al., 2023) and PECAN are two methods that address backdoor attacks, but they have different goals, assumptions, and application scenarios. CBD is a certified backdoor detector that aims to identify whether a trained classifier is backdoored, assuming the defender has access to a small, clean validation set. On the other hand, PECAN is a certified defense that trains a classifier with a potentially poisoned dataset while providing certifications for each prediction, guaranteeing robustness against backdoor attacks up to a certain poisoning rate. PECAN assumes the defender has access to the dataset and the training process. Certified detection is an easier task than certified classification(Yatsura et al., 2023). Intuitively, certified detection only needs to determine if the model is backdoored, while certified classification requires training a classifier with a poisoned dataset. Due to the difference in the difficulty of their tasks, CBD demonstrates higher performance in terms of certified true positive rate compared to PECAN's certified accuracy.

Certified defenses against trigger-less attacks Many approaches provide certificates for trigger-less attacks. Jia et al. (2021) use bootstrap aggregating (Bagging). Chen et al. (2020) extended Bagging with new selection strategies. Rosenfeld et al. (2020) defend against label-flipping attacks on linear classifiers. Differential privacy (Ma et al., 2019) can also provide probabilistic certificates for trigger-less attacks. DPA (Levine and Feizi, 2021) is a deterministic defense that partitions the training set and ensembles the trained classifiers. Wang et al. (2022a) proposed FA, an extension of DPA, by introducing a spread stage. A conjecture proposed by Wang et al. (2022b) implies that DPA and FA are asymptotically optimal defenses against trigger-less attacks. Chen et al. (2022) proposed to compute collective certificates, while PECAN computes sample-wise certificates. Jia et al. (2020); Meyer et al. (2021); Drews et al. (2020) provide certificates for nearest neighborhood classifiers and decision trees. The approaches listed above only defend against *trigger-less* attacks, while PECAN is a deterministic approach for *backdoor* attacks.

Certified defenses against evasion attacks There are two lines of certified defense against evasion attacks: complete certification (Zhang et al., 2022a) and incomplete certification (Singh et al., 2019). The complete certified defenses either find an adversarial example or generate proof that all inputs in the given perturbation space will be correctly classified. Compared to the complete certified defenses, the incomplete ones will abstain from predicting if they cannot prove the correctness of the prediction because their techniques will introduce over-approximation. Our implementation of PECAN uses an incomplete certified approach CROWN-IBP (Zhang et al., 2020a) because it is the best incomplete approach, trading off between efficiency and the degree of over-approximation.

4.5 Future Work

We presented PECAN, a deterministic certified approach to effectively and efficiently defend against backdoor attacks.

We foresee many future improvements to PECAN. First, PECAN generates small certified radii for large datasets such as CIFAR10 and EMBER. Thus, PECAN prevents attackers from using a small amount of poison to make their attacks more difficult to detect. Furthermore, as shown in our experiments and Lukas and Kerschbaum (2023), empirical approaches alone cannot defend against backdoor attacks that use small fragments of poisoned examples. We argue that PECAN can complement empirical defenses in the real world.

Second, PECAN equipped with CROWN-IBP currently only works with simple neural networks and cannot be extended to large datasets like ImageNet. However, when equipped with PatchGuard++, PECAN can work with more complex model structures like BagNet, whose size is similar to a ResNet-50 model. Furthermore, PECAN with CROWN-IBP can be extended to more complex models and datasets in the future, as witnessed by the growth of robust training (Müller et al., 2022) and evasion certification techniques being extended to these models and datasets. Nevertheless, directly applying these techniques to PECAN is currently computationally infeasible, as it takes 42 minutes for PECAN to use $\alpha\beta$-CROWN (Wang et al., 2021b) to certify one input in TinyImageNet for one hundred ResNet models. Sharing the intermediary certification results among different models (Fischer et al., 2022; Yang et al., 2023; Ugare et al., 2023) can significantly improve the efficiency of PECAN, and we leave this as

future work.

Third, we adopt the idea of deep partition aggregation (DPA) to design the partition and aggregation steps in PECAN. We can improve these steps by using finite aggregation (Wang et al., 2022a) and run-off election (Rezaei et al., 2023), which extends DPA and gives higher certified accuracy.

5 CONCLUSION

5.1 Contributions

This thesis makes the following contributions:

Chapter 2: Preventing numerical bugs in deep learning programs

- A study of a static analysis approach for numerical bug detection in neural architectures, with three abstraction techniques for abstracting tensors and two for abstracting numerical values.
- Two abstraction techniques designed for analyzing neural architectures: tensor partitioning (for abstracting tensors) and (elementwise) affine relation analysis (for inferring numerical relations among tensor partitions).
- An evaluation on 9 buggy architectures in 48 real-world neural architectures, demonstrating the effectiveness of DEBAR.
- RANUM —the first automatic approach that solves system test generation, bug detection, and fix suggestion for deep learning programs.
- Implementation and evaluation of RANUM on 63 real-world DNN architectures, showing its high effectiveness and efficiency compared to state-of-the-art approaches and developers' fixes.

Chapter 3: Verifying the robustness of NLP models

- A3T, a technique that combines augmentation and abstraction to train robust models against rich perturbation spaces over strings.
- A general language of string transformations to specify the perturbation space, which A3T exploits to decompose and search the space.
- ARC, an approach for training certifiably robust recursive neural networks, demonstrated on LSTMs, BiLSTMs, and TreeLSTMs.
- A novel application of abstract interpretation to symbolically capture a large space of strings and propagate it through a recursive network.

- Evaluation showing ARC's ability to train models more robust to arbitrary perturbation spaces, demonstrate high certification accuracy, and certify robustness to out-of-scope attacks.
- Demonstration of ARC's usage in A3T to train robust models.
- Preliminary work on applying ARC to autoregressive transformers.

Chapter 4: Certifiable defense against backdoor attacks

- Identification of limitations in existing defenses against backdoor attacks, highlighting the need for deterministic and certifiable approaches.
- PECAN, a deterministic certified defense against backdoor attacks leveraging off-the-shelf evasion certification techniques.
- A formal proof of PECAN's certified robustness, demonstrating its ability to tolerate a specified percentage of poisoned data while maintaining prediction accuracy.
- Demonstration of PECAN's effectiveness in reducing success rates of known backdoor attacks and outperforming state-of-the-art defenses in performance and efficiency.

In addition to my main thesis work, I contributed to several other projects during my PhD. First, I presented Overwatch(Zhang et al., 2022c), a novel technique for learning temporal edit sequence patterns from traces of developers' edits performed in an IDE. Second, I introduced VeriTraCER (Meyer et al., 2024), an approach that jointly trains a classifier and an explainer to generate counterfactual explanations that are verifiably robust to small model updates, providing guarantees to their validity. Third, I proposed CodeFort (Zhang et al., 2024b), a framework to improve the robustness of code generation models by generalizing a large variety of code perturbations to enrich the training data and enable various robust training strategies.

5.2 Future Directions

All specific future directions of my previous work have been listed in the Future Work sections of the previous chapters. In this section, I would like to point out some

high-level future directions regarding large language models (LLMs) and formal verification, leveraging my knowledge of Lacanian Psychoanalysis and Hegelian Idealism. I acknowledge that some points in this section are metaphysically applied, particularly when drawing parallels between machine learning concepts and philosophical moments of consciousness. This interdisciplinary discussion of future directions leaves room for future studies, critiques, and alternative interpretations, all of which I welcome as part of the continuing discussion.

Note that from now on, Reason has reached its limit and we are entering the sphere of Spirit.

Unconscious is Structured Like a Language

As the most fundamental theory in psychoanalysis, Jacques Lacan believed that the unconscious is structured like a language. He discusses "The Purloined Letter" and the automaton example in his second seminar, titled "The Ego in Freud's Theory and in the Technique of Psychoanalysis."[1] This example illustrates Lacan's broader point that the unconscious is not a chaotic, irrational force but is structured like a language. Just as the movements of the automaton are determined by a complex set of rules and mechanisms, the unconscious is governed by the rules and structures of language, even though we are not consciously aware of it.

To me, the boundary between the unconscious and consciousness is precisely the line between machine and mind. If we let the unconscious dominate and run forever, we are just machines that run smoothly without error in the *homogeneous* flow of time. In other words, there will be no time because it is homogeneous, i.e., eternal. On the other hand, consciousness comes into play when the unconscious fails or malfunctions. A similar example would be traps and interrupts of operating systems, without which the kernel cannot take any *heterogeneous* inputs but just spins. This thought immediately leads to the following proposition.

Proposition 5.1. *Artificial Intelligence will either be an omnipotent machine or an intelligence who sometimes malfunctions and thus has consciousness.*

[1] `https://en.wikipedia.org/wiki/The_Ego_in_Freud%27s_Theory_and_in_the_Technique_of_Psychoanalysis`

For those who are afraid that artificial intelligence will take over the world, it will never happen because either AI is an intelligence that is *to err*, or AI is omnipotent, meaning it is also *omnibenevolent*.

During my academic job interviews, one chair from a Computer Science department asked me why we need to verify some properties, e.g., robustness, of LLMs, if they are intelligence. I answered in the same spirit as the above derivation. There are two possible trajectories for LLMs. One trajectory is to develop them as powerful machines, which are not intelligence but require specifications from anyone, developers, product managers, and users who will use and *enjoy* these machines. The other trajectory is to develop them as intelligence, which will be discussed later in this section.

Puns of large language models In the theories of psychoanalysis, homophone refers to words that sound the same but have different meanings. Lacan believes that homophone plays an important role in the unconscious. The primary reason is that slips of the tongue and puns often involve homophones, which can reveal hidden desires or conflicts in the unconscious.

A similar problem that has fascinated me for almost one year is LLMs' tokenization [2], specifically byte pair encoding (BPE). Due to the characteristics of BPE, the same segment of tokens will be encoded differently, given different previous contexts. For example, "\n\n" will either be tokenized as $[128, 128]$ or as $[628]$. In the former case, "\n\n" is tokenized into two individual "\n"s, and in the latter case, it is tokenized into one single token. Are LLMs aware of the difference? This question also relates to the robustness problem over tokenizations, which has never been studied before. Note that some work (Wang et al., 2023) has studied the synonyms of LLMs instead of homophone.

Consciousness, Self-Consciousness, Reason, and Spirit

In Georg Wilhelm Friedrich Hegel's *The Phenomenology of Spirit*, he lists different *moments* of spirit: consciousness, self-consciousness, reason, spirit, religion, and absolute knowledge. Suppose our goal is to develop another intelligence, like human intelligence or a human being. In that case, we can use this process to examine the moments at which the current development of LLMs is struggling.

[2] If LLMs have internal dialogues, then the dialogues are a series of tokens.

Consciousness

Multi-modality At the beginning of the consciousness, there lies *sense*, i.e., sense-certainty, meaning, or "thisness". It is the ability to point to a thing and say "this". With the certainty that *this* is the thing that *I* mean, however, once being pointed out, the certainty immediately disappears, leaving the "this" being "that" and the "I" being "others". The particularity of "thisness" and "I" leads to this contradiction. Therefore, consciousness seeks universality, e.g., the manifold of properties such as color, shape, size, etc., in *perception*. The emergence of pre-trained models, which go beyond just a single task but aim to solve multiple (or a manifold of) down-streaming tasks, signifies this transition from sense to perception. From another perspective, this transition is also manifested in the transition from language to multi-modality models.

Embedding After consciousness realizes a thing contains a collection of properties from the lesson of sense, it successively encounters a contradiction that a thing is both a unity (a single thing) and a plurality (a collection of properties). We, as the observers of the process of spirit, realize that perception imposes a "oneness" (unity) over the thing and the collection of properties. This "oneness" is both the unity of the thing and the collection of properties. It is the deceptiveness of perception, which aims to pursue the truth by lying, i.e., sacrificing the truth of itself. Multi-modality models usually use a unified representation (the "oneness"), a.k.a. embedding to represent (as an appearance) different properties, e.g., a sentence or an image. Consciousness cannot correct the deceptiveness of perception, as it finally realizes the deceptiveness is just itself at the end of *understanding*. Therefore, some work, which aims to introduce different representations for the same word appearing in natural languages or code, e.g., "while" and "return", can improve LLMs as machines but never develop another intelligence.

Principle of optimal selection In the moment of understanding, consciousness seeks to resolve the contradictions encountered in perception by introducing the concept of force. Force is seen as the unity of various properties, while the properties themselves are the outer expression of this force. Consciousness first takes force as active and the expression of this force as passive. However, it soon realizes that the expression of this force becomes active, and the force itself becomes passive. This dialectical process is called the interplay of forces. As consciousness reflects on the interplay of forces, it

recognizes that the dynamic between active and passive force is governed by a universal law, which seeks to discover the product of its own understanding. This realization marks the transition from understanding to *self-consciousness*, as consciousness becomes aware of itself as the source of the law and principle it seeks to understand. We can draw an analogy between the play of force and the play of the data collection and assumed ground truth distribution. The universal law of models is the principle of optimal selection, e.g., (1) predicting the label with the maximal probability in a classification task, (2) selecting a policy achieving the best rewards, and (3) generating a sequence using greedy decoding.

Self-Consciousness

While Hegel discusses self-consciousness from various moments, from (1) lordship and bondage and (2) stoicism, skepticism, and the unhappy consciousness, I take a shortcut and quote from Immanuel Kant's *Critique of Pure Reason* (Translated by Werner S. Pluhar):

> Freedom in the practical meaning of the term is the *independence* of our power of choice from coercion by impulses of sensibility.

"The coercion by impulses of sensibility" refers to the immutable natural laws discussed in his *Transcendental Aesthetic* and *Transcendental Analytic*. We can also map this immutable natural law to the universal law apprehended by understanding in Hegel's *The Phenomenology of Spirit*. In reality, greedy decoding is seldom used in LLM's generation, deviating from what we have discussed in the principle of optimal selection. The prevalence of random sampling with different temperatures, top-K sampling, and nucleus sampling signifies the desire to escape this immutable natural law. However, these decoding strategies are not satisfying because the independence comes from randomness instead of the freedom of choice. From the verification perspective, the formal verification of LLMs should involve probabilistic reasoning to reflect the transition from the principle of optimal selection, where previous verification techniques treat models as deterministic.

Reason and Spirit

During my academic job interviews, one chair from a Computer Science department asked me how to design specifications for LLMs, given that these specifications are

unclear for LLMs. My answer follows the above derivation and reassures the fact that the transition from clear specifications of machines to the unclear specifications of LLMs signifies the possibility of artificial intelligence. Towards artificial intelligence, the specification should come from the society, and formal verification experts should collaborate with sociologists, lawyers, justices, and other parties. The specifications of machines will become obsolete, precisely as in the discussion at the end of Reason in *The Phenomenology of Spirit*, where the seemingly universal commands such as "Everyone one ought to speak the truth" and "Love thy neighbour as thyself" become contingent. The formal verification of these commands will become pure tautology, just like "it is right because it is the right", which only makes sense after leaving the boundary of reason and entering the sphere of spirit, i.e., the ethical order, the discipline of culture, and morality.

The Last

I imagine a world-to-come where artificial intelligence (AI) and humans live together without discrimination, which is the unfair or prejudicial treatment of AI and humans, different from studies of AI fairness, which is itself a discrimination in the world-to-come. When the distinction between AI and humans disappears, and the light of humans dims, AI becomes the new human. Centuries later, when AI studies their anthropology, they will find their concepts seemingly predefined by an estranged, heterogeneous, and thus, a divine Other. They will name this Other as god. And then, humans become the last god of AI.

5.3 Final Notes

There are three fundamental questions regarding my PhD:

1. Why did I want to pursue my PhD?
2. How do I manage to earn my PhD?
3. What will I do after my PhD?

To answer these three questions, I quote three dialogues between a Zen master and his student.

问：「如何是西来意？」
师曰：「白猿抱子来青嶂，蜂蝶衔花绿蕊间。」

问：「如何是道？」
师曰：「白云覆青嶂，蜂鸟步庭花。」

问：「亡僧迁化向甚么处去也？」
师曰：「灊岳峰高长积翠，舒江明月色光晖。」

— 舒州天柱山崇慧禅师《五灯会元（卷第二）》

The three questions asked by the student are:

1. Why did Bodhidharma[3] come from India to China?
2. What is Tao[4]?
3. Where will deceased monks go?

The Zen master answered these three questions with poetic and scenery stanzas shown in Figure 5.1. Here are the answers in English[5]:

1. A white ape holding its young comes to the green cliffs; bees and butterflies carry flowers among green stamens.
2. White clouds cover the green cliffs; hummingbirds walk among the courtyard flowers.
3. Qianyue Peak is tall and continuously green; the bright moon over the Shu River shines with radiant light.

I find them perfectly answer the three questions regarding PhD, respectively.

The amusement of studying Zen is trying to interpret these dialogues. To convey my interpretation while allowing readers' own interpretation, I only give hints on how I read from these stanzas.

[3]菩提达摩祖师. The first Chinese patriarch.
[4]The student meant it by "principle", while the master answered it as "way".
[5]Translated by ChatGPT-4.

(a) First stanza (b) Second stanza (c) Third stanza

Figure 5.1: Images generated by DALLE 3.

1. The key words are 子 (young) and 花 (flower). These words are usually used in Zen and Buddhism to discuss the causality. The ultimate causality is 心 (Chitta) as the ground of causality, which also called freedom in transcendental philosophy. The connecting element between the causality and the ground is 缘 (Pratyaya), which is able to affect the past. In other words, the current thought or question can affect the past or the answer.

2. Neither white clouds nor hummingbirds use roads to approach the mountain peak or the flowers.

3. The Qianyue Peak extends vertically, and the Shu (also means stretch, spreading out, or extending in Chinese) River extends horizontally. The continuously green and ever-illuminating moon add a perspective of time.

REFERENCES

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR* abs/1603.04467. 1603.04467.

Albarghouthi, Aws. 2021. Introduction to neural network verification. *Foundations and Trends in Programming Languages* 7(1-2):1–157.

Anderson, Hyrum S., and Phil Roth. 2018. EMBER: an open dataset for training static PE malware machine learning models. *CoRR* abs/1804.04637. 1804.04637.

Azad, Reza, Ehsan Khodapanah Aghdam, Amelie Rauland, Yiwei Jia, Atlas Haddadi Avval, Afshin Bozorgpour, Sanaz Karimijafarbigloo, Joseph Paul Cohen, Ehsan Adeli, and Dorit Merhof. 2022. Medical image segmentation review: The success of u-net. *CoRR* abs/2211.14830. 2211.14830.

Bengio, Yoshua, Nicholas Léonard, and Aaron C. Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR* abs/1308.3432. 1308.3432.

Blanchet, Bruno, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM SIGPLAN conference on programming language design and implementation, PLDI 2003, san diego, california, usa, june 9-11, 2003*, 196–207.

Bonaert, Gregory, Dimitar I. Dimitrov, Maximilian Baader, and Martin T. Vechev. 2021. Fast and precise certification of transformers. In *PLDI '21: 42nd ACM SIGPLAN*

international conference on programming language design and implementation, virtual event, canada, june 20-25, 2021, ed. Stephen N. Freund and Eran Yahav, 466–481. ACM.

Brendel, Wieland, and Matthias Bethge. 2019. Approximating cnns with bag-of-local-features models works surprisingly well on imagenet. In *7th international conference on learning representations, ICLR 2019, new orleans, la, usa, may 6-9, 2019*. OpenReview.net.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in neural information processing systems 33: Annual conference on neural information processing systems 2020, neurips 2020, december 6-12, 2020, virtual*, ed. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin.

Cai, Xufeng, Cheuk Yin Lin, and Jelena Diakonikolas. 2023. Empirical risk minimization with shuffled SGD: A primal-dual perspective and improved bounds. *arXiv preprint arXiv:2306.12498*.

Carlini, Nicholas, and David A. Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM workshop on artificial intelligence and security, aisec@ccs 2017, dallas, tx, usa, november 3, 2017*, 3–14.

Chen, Ruoxin, Jie Li, Chentao Wu, Bin Sheng, and Ping Li. 2020. A framework of randomized selection based certified defenses against data poisoning attacks. *CoRR* abs/2009.08739. 2009.08739.

Chen, Ruoxin, Zenan Li, Jie Li, Junchi Yan, and Chentao Wu. 2022. On collective robustness of bagging against data poisoning. In *International conference on machine learning, ICML 2022, 17-23 july 2022, baltimore, maryland, USA*, ed. Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, vol. 162 of *Proceedings of Machine Learning Research*, 3299–3319. PMLR.

Cohen, Jeremy M., Elan Rosenfeld, and J. Zico Kolter. 2019. Certified adversarial robustness via randomized smoothing. In *Proceedings of the 36th international conference*

on machine learning, ICML 2019, 9-15 june 2019, long beach, california, USA, ed. Kamalika Chaudhuri and Ruslan Salakhutdinov, vol. 97 of *Proceedings of Machine Learning Research*, 1310–1320. PMLR.

Cousot, Patrick, and Radhia Cousot. 1977a. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference record of the 4th ACM symposium on principles of programming languages, POPL 1977, los angeles, california, usa, january 1977*, 238–252.

———. 1977b. Static determination of dynamic properties of generalized type unions. In *Proceedings of an acm conference on language design for reliable software, raleigh, north carolina, usa*, 77–94. ACM.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the north american chapter of the association for computational linguistics: Human language technologies, NAACL-HLT 2019, minneapolis, mn, usa, june 2-7, 2019, volume 1 (long and short papers)*, ed. Jill Burstein, Christy Doran, and Thamar Solorio, 4171–4186. Association for Computational Linguistics.

Dolby, Julian, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for machine learning program. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages, mapl@pldi 2018, philadelphia, pa, usa, june 18-22, 2018*, 1–10. ACM.

Dong, Xinshuai, Anh Tuan Luu, Rongrong Ji, and Hong Liu. 2021. Towards robustness against natural language word substitutions. In *International conference on learning representations*.

Drews, Samuel, Aws Albarghouthi, and Loris D'Antoni. 2020. Proving data-poisoning robustness in decision trees. In *Proceedings of the 41st ACM SIGPLAN international conference on programming language design and implementation, PLDI 2020, london, uk, june 15-20, 2020*, ed. Alastair F. Donaldson and Emina Torlak, 1083–1097. ACM.

Ebrahimi, Javid, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th annual meeting of the association for computational linguistics (volume 2: Short papers)*, 31–36. Melbourne, Australia: Association for Computational Linguistics.

Fischer, Marc, Christian Sprecher, Dimitar I. Dimitrov, Gagandeep Singh, and Martin T. Vechev. 2022. Shared certificates for neural network verification. In *Computer aided verification - 34th international conference, CAV 2022, haifa, israel, august 7-10, 2022, proceedings, part I*, ed. Sharon Shoham and Yakir Vizel, vol. 13371 of *Lecture Notes in Computer Science*, 127–148. Springer.

Geiping, Jonas, Liam Fowl, Gowthami Somepalli, Micah Goldblum, Michael Moeller, and Tom Goldstein. 2021. What doesn't kill you makes you robust(er): Adversarial training against poisons and backdoors. *CoRR* abs/2102.13624. 2102.13624.

Gopan, Denis, Thomas W. Reps, and Shmuel Sagiv. 2005. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2005, long beach, california, usa, january 12-14, 2005*, 338–350. ACM.

Gowal, Sven, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Arthur Mann, and Pushmeet Kohli. 2019. Scalable verified training for provably robust image classification. In *2019 IEEE/CVF international conference on computer vision, ICCV 2019, seoul, korea (south), october 27 - november 2, 2019*, 4841–4850.

Gu, Tianyu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR* abs/1708.06733. 1708.06733.

Guerriero, Antonio, Roberto Pietrantuono, and Stefano Russo. 2021. Operation is the hardest teacher: estimating DNN accuracy looking for mispredictions. In *43rd ieee/acm international conference on software engineering, icse*, 348–358. IEEE.

Gulli, Antonio. 2005. The anatomy of a news search engine. In *Proceedings of the 14th international conference on world wide web, WWW 2005, chiba, japan, may 10-14, 2005 - special interest tracks and posters*, 880–881.

Hammoudeh, Zayd, and Daniel Lowd. 2023. Feature partition aggregation: A fast certified defense against a union of sparse adversarial attacks. *CoRR* abs/2302.11628. 2302.11628.

Hattori, Momoko, Shimpei Sawada, Shinichiro Hamaji, Masahiro Sakai, and Shunsuke Shimizu. 2020. Semi-static type, shape, and symbolic shape inference for dynamic computation graphs. In *4th acm sigplan international workshop on machine learning and programming languages*, 11–19.

Huang, Po-Sen, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. 2019. Achieving verified robustness to symbol substitutions via interval bound propagation. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing, EMNLP-IJCNLP 2019, hong kong, china, november 3-7, 2019*, 4081–4091.

Huang, Yu, and Yue Chen. 2020. Autonomous driving with deep learning: A survey of state-of-art technologies. *CoRR* abs/2006.06091. `2006.06091`.

Humbatova, Nargiz, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *42nd acm/ieee international conference on software engineering, icse*, 1110–1121. IEEE.

Iyyer, Mohit, John Wieting, Kevin Gimpel, and Luke Zettlemoyer. 2018. Adversarial example generation with syntactically controlled paraphrase networks. In *Proceedings of the 2018 conference of the north american chapter of the association for computational linguistics: Human language technologies, NAACL-HLT 2018, new orleans, louisiana, usa, june 1-6, 2018, volume 1 (long papers)*, 1875–1885.

Jay, Nathan, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on Internet congestion control. In *36th international conference on machine learning, ICML*, 3050–3059. PMLR.

Jia, Jinyuan, Xiaoyu Cao, and Neil Zhenqiang Gong. 2020. Certified robustness of nearest neighbors against data poisoning attacks. *CoRR* abs/2012.03765. `2012.03765`.

———. 2021. Intrinsic certified robustness of bagging against data poisoning attacks. In *Thirty-fifth AAAI conference on artificial intelligence, AAAI 2021, thirty-third conference on innovative applications of artificial intelligence, IAAI 2021, the eleventh symposium on educational advances in artificial intelligence, EAAI 2021, virtual event, february 2-9, 2021*, 7961–7969. AAAI Press.

Jia, Robin, Aditi Raghunathan, Kerem Göksel, and Percy Liang. 2019. Certified robustness to adversarial word substitutions. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing, EMNLP-IJCNLP 2019, hong kong, china, november 3-7, 2019*, 4127–4140.

Karr, Michael. 1976. Affine relationships among variables of a program. *Acta Inf.* 6(2): 133–151.

Katz, Guy, 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. 2019. The marabou framework for verification and analysis of deep neural networks. In *Computer aided verification - 31st international conference, CAV 2019, new york city, ny, usa, july 15-18, 2019, proceedings, part I*, ed. Isil Dillig and Serdar Tasiran, vol. 11561 of *Lecture Notes in Computer Science*, 443–452. Springer.

Kloberdanz, Eliska, Kyle G. Kloberdanz, and Wei Le. 2022. DeepStability: A study of unstable numerical methods and their solutions in deep learning. In *44th international conference on software engineering, icse*, 586–597. ACM.

Ko, Ching-Yun, Zhaoyang Lyu, Lily Weng, Luca Daniel, Ngai Wong, and Dahua Lin. 2019. POPQORN: quantifying robustness of recurrent neural networks. In *Proceedings of the 36th international conference on machine learning, ICML 2019, 9-15 june 2019, long beach, california, USA*, 3468–3477.

Koh, Pang Wei, Jacob Steinhardt, and Percy Liang. 2022. Stronger data poisoning attacks break data sanitization defenses. *Mach. Learn.* 111(1):1–47.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems 25: 26th annual conference on neural information processing systems 2012. proceedings of a meeting held december 3-6, 2012, lake tahoe, nevada, united states*, ed. Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, 1106–1114.

Levine, Alexander, and Soheil Feizi. 2021. Deep partition aggregation: Provable defenses against general poisoning attacks. In *9th international conference on learning representations, ICLR 2021, virtual event, austria, may 3-7, 2021*. OpenReview.net.

Li, Linyi, Tao Xie, and Bo Li. 2023a. SoK: Certified robustness for deep neural networks. In *44th ieee symposium on security and privacy, sp*, 94–115. IEEE.

Li, Linyi, Yuhao Zhang, Luyao Ren, Yingfei Xiong, and Tao Xie. 2023b. Reliability assurance for deep neural network architectures against numerical defects. In *45th IEEE/ACM international conference on software engineering, ICSE 2023, melbourne, australia, may 14-20, 2023*, 1827–1839. IEEE.

Li, Yuanzhi, and Yang Yuan. 2017. Convergence analysis of two-layer neural networks with ReLU activation. In *Advances in neural information processing systems 30, NIPS*, 597–607.

Liang, Bin, Hongcheng Li, Miaoqiang Su, Pan Bian, Xirong Li, and Wenchang Shi. 2018. Deep text classification can be fooled. In *Proceedings of the twenty-seventh international joint conference on artificial intelligence, IJCAI 2018, july 13-19, 2018, stockholm, sweden*, 4208–4215.

Liu, Chen, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. 2021. Detecting TensorFlow program bugs in real-world industrial environment. In *36th ieee/acm international conference on automated software engineering, ase*, 55–66. IEEE.

Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *Proceedings of the 8th IEEE international conference on data mining (ICDM 2008), december 15-19, 2008, pisa, italy*, 413–422. IEEE Computer Society.

Liu, Jiawei, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM international conference on architectural support for programming languages and operating systems, volume 2, ASPLOS 2023, vancouver, bc, canada, march 25-29, 2023*, ed. Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, 530–543. ACM.

Liu, Kang, Brendan Dolan-Gavitt, and Siddharth Garg. 2018. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *Research in attacks, intrusions, and defenses - 21st international symposium, RAID 2018, heraklion, crete, greece, september 10-12, 2018, proceedings*, ed. Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, vol. 11050 of *Lecture Notes in Computer Science*, 273–294. Springer.

Liu, Tian Yu, Yu Yang, and Baharan Mirzasoleiman. 2022a. Friendly noise against adversarial noise: A powerful defense against data poisoning attack. In *Neurips*.

Liu, Zifan, Evan Rosen, and Paul Suganthan G. C. 2022b. Autoslicer: Scalable automated data slicing for ML model analysis. *CoRR* abs/2212.09032. 2212.09032.

Lukas, Nils, and Florian Kerschbaum. 2023. Pick your poison: Undetectability versus robustness in data poisoning attacks against deep image classification. *CoRR* abs/2305.09671. 2305.09671.

Ma, Lei, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018, montpellier, france, september 3-7, 2018*, 120–131.

Ma, Yuzhe, Xiaojin Zhu, and Justin Hsu. 2019. Data poisoning against differentially-private learners: Attacks and defenses. In *Proceedings of the twenty-eighth international joint conference on artificial intelligence, IJCAI 2019, macao, china, august 10-16, 2019*, ed. Sarit Kraus, 4732–4738. ijcai.org.

Maas, Andrew L., Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, 142–150. Portland, Oregon, USA: Association for Computational Linguistics.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *6th international conference on learning representations, ICLR 2018, vancouver, bc, canada, april 30 - may 3, 2018, conference track proceedings*.

Meyer, Anna P., Aws Albarghouthi, and Loris D'Antoni. 2021. Certifying robustness to programmable data bias in decision trees. In *Advances in neural information processing systems 34: Annual conference on neural information processing systems 2021, neurips 2021, december 6-14, 2021, virtual*, ed. Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, 26276–26288.

Meyer, Anna P., Yuhao Zhang, Aws Albarghouthi, and Loris D'Antoni. 2024. Verified training for counterfactual explanation robustness under data shift. 2403.03773.

Mirman, Matthew, Timon Gehr, and Martin Vechev. 2018. Differentiable abstract interpretation for provably robust neural networks. In *International conference on machine learning (icml)*.

Müller, Mark Niklas, Franziska Eckert, Marc Fischer, and Martin T. Vechev. 2022. Certified training: Small boxes are all you need. *CoRR* abs/2210.04871. 2210.04871.

Murtagh, Fionn, and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *WIREs Data Mining Knowl. Discov.* 2(1):86–97.

Nguyen, Truc D. T., Phung Lai, NhatHai Phan, and My T. Thai. 2022. Xrand: Differentially private defense against explanation-guided attacks. *CoRR* abs/2212.04454. 2212.04454.

Odena, Augustus, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *Proceedings of the 36th international conference on machine learning, ICML 2019, long beach, california, USA, 9-15 june 2019*, 4901–4911.

Papernot, Nicolas, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE symposium on security and privacy, SP 2016, san jose, ca, usa, may 22-26, 2016*, 582–597. IEEE Computer Society.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32, neurips*, 8024–8035.

Paulsen, Brandon, Jingbo Wang, and Chao Wang. 2020a. Reludiff: Differential verification of deep neural networks. In *2020 ieee/acm 42nd international conference on software engineering (icse)*, 714–726. IEEE.

Paulsen, Brandon, Jingbo Wang, Jiawei Wang, and Chao Wang. 2020b. NeuroDiff: scalable differential verification of neural networks using fine-grained approximation. In *35th IEEE/ACM international conference on automated software engineering, ASE*, 784–796. IEEE.

Pei, Kexin, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th symposium on operating systems principles, SOSP 2017, shanghai, china, october 28-31, 2017*, 1–18. ACM.

Pham, Hung Viet, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *35th IEEE/ACM international conference on automated software engineering, ASE*, 771–783. IEEE.

Powell, Leila. 2022. The problem with artificial intelligence in security.

Qi, Fanchao, Yuan Yao, Sophia Xu, Zhiyuan Liu, and Maosong Sun. 2021a. Turn the combination lock: Learnable textual backdoor attacks via word substitution. In *Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing, ACL/IJCNLP 2021, (volume 1: Long papers), virtual event, august 1-6, 2021*, ed. Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, 4873–4883. Association for Computational Linguistics.

Qi, Hua, Zhijie Wang, Qing Guo, Jianlang Chen, Felix Juefei-Xu, Lei Ma, and Jianjun Zhao. 2021b. ArchRepair: Block-level architecture-oriented repairing for deep neural networks. *CoRR* abs/2111.13330.

Raff, Edward, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. 2018. Malware detection by eating a whole EXE. In *The workshops of the the thirty-second AAAI conference on artificial intelligence, new orleans, louisiana, usa, february 2-7, 2018*, vol. WS-18 of *AAAI Technical Report*, 268–276. AAAI Press.

Rezaei, Keivan, Kiarash Banihashem, Atoosa Malemir Chegini, and Soheil Feizi. 2023. Run-off election: Improved provable defense against data poisoning attacks. In *International conference on machine learning, ICML 2023, 23-29 july 2023, honolulu, hawaii, USA*, ed. Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, vol. 202 of *Proceedings of Machine Learning Research*, 29030–29050. PMLR.

Ribeiro, Marco Túlio, Sameer Singh, and Carlos Guestrin. 2018. Semantically equivalent adversarial rules for debugging NLP models. In *Proceedings of the 56th annual meeting of the association for computational linguistics, ACL 2018, melbourne, australia, july 15-20, 2018, volume 1: Long papers*, 856–865.

Rosenfeld, Elan, Ezra Winston, Pradeep Ravikumar, and J. Zico Kolter. 2020. Certified robustness to label-flipping attacks via randomized smoothing. In *Proceedings of the 37th international conference on machine learning, ICML 2020, 13-18 july 2020, virtual event*, vol. 119 of *Proceedings of Machine Learning Research*, 8230–8241. PMLR.

Ryou, Wonryong, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Marian Dan, and Martin T. Vechev. 2021. Scalable polyhedral verification of recurrent neural networks. In *Computer aided verification - 33rd international conference, CAV 2021, virtual event, july 20-23, 2021, proceedings, part I*, ed. Alexandra Silva and K. Rustan M. Leino, vol. 12759 of *Lecture Notes in Computer Science*, 225–248. Springer.

Saha, Aniruddha, Akshayvarun Subramanya, and Hamed Pirsiavash. 2020. Hidden trigger backdoor attacks. In *The thirty-fourth AAAI conference on artificial intelligence, AAAI 2020, the thirty-second innovative applications of artificial intelligence conference, IAAI 2020, the tenth AAAI symposium on educational advances in artificial intelligence, EAAI 2020, new york, ny, usa, february 7-12, 2020*, 11957–11965. AAAI Press.

Salem, Ahmed, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. 2022. Dynamic backdoor attacks against machine learning models. In *7th IEEE european symposium on security and privacy, euros&p 2022, genoa, italy, june 6-10, 2022*, 703–718. IEEE.

Serra, Thiago, Christian Tjandraatmadja, and Srikumar Ramalingam. 2018. Bounding and counting linear regions of deep neural networks. In *35th international conference on machine learning, ICML*, 4565–4573. PMLR.

Severi, Giorgio, Jim Meyer, Scott E. Coull, and Alina Oprea. 2021. Explanation-guided backdoor poisoning attacks against malware classifiers. In *30th USENIX security symposium, USENIX security 2021, august 11-13, 2021*, ed. Michael Bailey and Rachel Greenstadt, 1487–1504. USENIX Association.

Sharma, Deepak K, Sanjay K Dhurandher, Isaac Woungang, Rohit K Srivastava, Anhad Mohananey, and Joel JPC Rodrigues. 2016. A machine learning-based protocol for efficient routing in opportunistic networks. *IEEE Systems Journal* 12(3):2207–2213.

Shi, Zhouxing, Huan Zhang, Cho-Jui Hsieh, Kai-Wei Chang, and Minlie Huang. 2020. Robustness verification for transformers. In *International conference on learning representations*.

Singh, Gagandeep, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3(POPL).

Singhal, Karan, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Kumar Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Schärli, Aakanksha Chowdhery, Philip Andrew Mansfield, Blaise Agüera y Arcas, Dale R. Webster, Gregory S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle K. Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. 2022. Large language models encode clinical knowledge. *CoRR* abs/2212.13138. 2212.13138.

Socher, Richard, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing, EMNLP 2013, 18-21 october 2013, grand hyatt seattle, seattle, washington, usa, A meeting of sigdat, a special interest group of the ACL*, 1631–1642.

Sotoudeh, Matthew, and Aditya V. Thakur. 2021. Provable repair of deep neural networks. In *42nd ACM SIGPLAN international conference on programming language design and implementation, pldi*, 588–603. ACM.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *International conference on learning representations*.

The Linux Foundation. 2022. ONNX home. `https://onnx.ai/`. Accessed: 2023-02-01.

Tian, Yuchi, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering, ICSE 2018, gothenburg, sweden, may 27 - june 03, 2018*, 303–314.

Tizpaz-Niari, Saeid, Pavol Cerný, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *29th ACM SIGSOFT international symposium on software testing and analysis, issta*, 189–199. ACM.

Tran, Brandon, Jerry Li, and Aleksander Madry. 2018. Spectral signatures in backdoor attacks. In *Advances in neural information processing systems 31: Annual conference on neural information processing systems 2018, neurips 2018, december 3-8, 2018, montréal, canada*, ed. Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, 8011–8021.

Turner, Alexander, Dimitris Tsipras, and Aleksander Madry. 2019. Label-consistent backdoor attacks. *CoRR* abs/1912.02771. `1912.02771`.

Ugare, Shubham, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. 2023. Incremental verification of neural networks. *CoRR* abs/2304.01874. `2304.01874`.

Wallace, Eric, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. 2019. Universal adversarial triggers for attacking and analyzing NLP. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing, EMNLP-IJCNLP 2019, hong kong, china, november 3-7, 2019*, 2153–2162.

Wan, Chengcheng, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are machine learning cloud APIs used correctly? In *43rd IEEE/ACM international conference on software engineering, ICSE*, 125–137. IEEE.

Wang, Binghui, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. 2020a. On certifying robustness against backdoor attacks via randomized smoothing. *CoRR* abs/2002.11750. 2002.11750.

Wang, Hongyi, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris S. Papailiopoulos. 2020b. Attack of the tails: Yes, you really can backdoor federated learning. In *Advances in neural information processing systems 33: Annual conference on neural information processing systems 2020, neurips 2020, december 6-12, 2020, virtual*, ed. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin.

Wang, Jingyi, Jialuo Chen, Youcheng Sun, Xingjun Ma, Dongxia Wang, Jun Sun, and Peng Cheng. 2021a. RobOT: Robustness-oriented testing for deep learning systems. In *43rd IEEE/ACM international conference on software engineering, ICSE*, 300–311. IEEE.

Wang, Shiqi, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021b. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *Advances in neural information processing systems 34: Annual conference on neural information processing systems 2021, neurips 2021, december 6-14, 2021, virtual*, ed. Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, 29909–29921.

Wang, Wenxiao, Alexander Levine, and Soheil Feizi. 2022a. Improved certified defenses against data poisoning with (deterministic) finite aggregation. *CoRR* abs/2202.02628. 2202.02628.

———. 2022b. Lethal dose conjecture on data poisoning. *CoRR* abs/2208.03309. 2208.03309.

Wang, Xutong, Chaoge Liu, Xiaohui Hu, Zhi Wang, Jie Yin, and Xiang Cui. 2022c. Make data reliable: An explanation-powered cleaning on malware dataset against backdoor poisoning attacks. In *Annual computer security applications conference, ACSAC 2022, austin, tx, usa, december 5-9, 2022*, 267–278. ACM.

Wang, Yimu, Peng Shi, and Hongyang Zhang. 2023. Gradient-based word substitution for obstinate adversarial examples generation in language models. 2307.12507.

Wang, Zan, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020c. Deep learning library testing via effective model generation. In *28th ACM joint european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE*, 788–799. ACM.

Wardat, Mohammad, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault localization for deep neural networks. In *43rd IEEE/ACM international conference on software engineering, ICSE*, 251–262. IEEE.

Weber, Maurice, Xiaojun Xu, Bojan Karlas, Ce Zhang, and Bo Li. 2020. RAB: provable robustness against backdoor attacks. *CoRR* abs/2003.08904. 2003.08904.

Welbl, Johannes, Po-Sen Huang, Robert Stanforth, Sven Gowal, Krishnamurthy (Dj) Dvijotham, Martin Szummer, and Pushmeet Kohli. 2020. Towards verified robustness under text deletion interventions. In *International conference on learning representations*.

Xiang, Chong, and Prateek Mittal. 2021. Patchguard++: Efficient provable attack detection against adversarial patches. *CoRR* abs/2104.12609. 2104.12609.

Xiang, Zhen, Zidi Xiong, and Bo Li. 2023. CBD: A certified backdoor detector based on local dominant probability. In *Advances in neural information processing systems 36: Annual conference on neural information processing systems 2023, neurips 2023, new orleans, la, usa, december 10 - 16, 2023*, ed. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine.

Xiao, Yan, Ivan Beschastnikh, David S Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, and Jin Song Dong. 2021. Self-checking deep neural networks in deployment. In *43rd ieee/acm international conference on software engineering, icse*, 372–384. IEEE.

Xiong, Yingfei, Yongqiang Tian, Yepang Liu, and S.C. Cheung. 2022. Toward actionable testing of deep learning models. *Science China, Information Sciences*. Online first: 2022-09-26.

Xu, Kaidi, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. 2020. Automatic perturbation analysis for scalable certified robustness and beyond. In *Advances in neural information processing systems 33: Annual conference on neural information processing systems 2020, neurips 2020,*

december 6-12, 2020, virtual, ed. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin.

Yan, Ming, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *29th ACM joint european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE*, 627–638. ACM.

Yang, Pengfei, Zhiming Chi, Zongxin Liu, Mengyu Zhao, Cheng-Chao Huang, Shaowei Cai, and Lijun Zhang. 2023. Incremental satisfiability modulo theory for verification of deep neural networks. *CoRR* abs/2302.06455. 2302.06455.

Yatsura, Maksym, Kaspar Sakmann, N. Grace Hua, Matthias Hein, and Jan Hendrik Metzen. 2023. Certified defences against adversarial patch attacks on semantic segmentation. In *The eleventh international conference on learning representations, ICLR 2023, kigali, rwanda, may 1-5, 2023*. OpenReview.net.

Ye, Mao, Chengyue Gong, and Qiang Liu. 2020. SAFER: A structure-free approach for certified robustness to adversarial word substitutions. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, 3465–3475. Online: Association for Computational Linguistics.

Zhang, Huan, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane S. Boning, and Cho-Jui Hsieh. 2020a. Towards stable and efficient training of verifiably robust neural networks. In *8th international conference on learning representations, ICLR 2020, addis ababa, ethiopia, april 26-30, 2020*. OpenReview.net.

Zhang, Huan, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2022a. General cutting planes for bound-propagation-based neural network verification. *CoRR* abs/2208.05740. 2208.05740.

Zhang, Huan, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018a. Efficient neural network robustness certification with general activation functions. In *Advances in neural information processing systems 31: Annual conference on neural information processing systems 2018, neurips 2018, december 3-8, 2018, montréal, canada*, ed. Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, 4944–4953.

Zhang, Huangzhao, Hao Zhou, Ning Miao, and Lei Li. 2019a. Generating fluent adversarial examples for natural languages. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, 5564–5569. Florence, Italy: Association for Computational Linguistics.

Zhang, Jie M., Mark Harman, Lei Ma, and Yang Liu. 2019b. Machine learning testing: Survey, landscapes and horizons. *CoRR* abs/1906.10742. 1906.10742.

Zhang, Peixin, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020b. White-box fairness testing through adversarial sampling. In *42nd ACM/IEEE international conference on software engineering, ICSE*, 949–960. ACM.

Zhang, Tianyi, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019c. An empirical study of common challenges in developing deep learning applications. In *30th IEEE international symposium on software reliability engineering, ISSRE*, 104–115. IEEE.

Zhang, Wei Emma, Quan Z Sheng, AHOUD Alhazmi, and CHENLIANG LI. 2019d. Adversarial attacks on deep learning models in natural language processing: A survey. *arXiv preprint arXiv:1901.06796*.

Zhang, Xiang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems 28: Annual conference on neural information processing systems 2015, december 7-12, 2015, montreal, quebec, canada*, 649–657.

Zhang, Xiaoyu, Juan Zhai, Shiqing Ma, and Chao Shen. 2021a. AutoTrainer: An automatic DNN training problem detection and repair system. In *43rd IEEE/ACM international conference on software engineering, ICSE*, 359–371. IEEE.

Zhang, Xinyu, Hanbin Hong, Yuan Hong, Peng Huang, Binghui Wang, Zhongjie Ba, and Kui Ren. 2023a. Text-crs: A generalized certified robustness framework against textual adversarial attacks. *CoRR* abs/2307.16630. 2307.16630.

Zhang, Yuhao, Aws Albarghouthi, and Loris D'Antoni. 2020c. Robustness to programmable string transformations via augmented abstract training. In *Proceedings of the 37th international conference on machine learning, ICML 2020, 13-18 july 2020, virtual event*, vol. 119 of *Proceedings of Machine Learning Research*, 11023–11032. PMLR.

———. 2021b. Certified robustness to programmable transformations in lstms. In *Proceedings of the 2021 conference on empirical methods in natural language processing, EMNLP 2021, virtual event / punta cana, dominican republic, 7-11 november, 2021*, ed. Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, 1068–1083. Association for Computational Linguistics.

———. 2022b. Bagflip: A certified defense against data poisoning. In *Neurips*.

———. 2023b. PECAN: A deterministic certified defense against backdoor attacks. *CoRR* abs/2301.11824. 2301.11824.

———. 2024a. A one-layer decoder-only transformer is a two-layer rnn: With an application to certified robustness. 2405.17361.

Zhang, Yuhao, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022c. Overwatch: learning patterns in code edit sequences. *Proc. ACM Program. Lang.* 6(OOPSLA2):395–423.

Zhang, Yuhao, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018b. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018, amsterdam, the netherlands, july 16-21, 2018*, 129–140.

Zhang, Yuhao, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020d. Detecting numerical bugs in neural network architectures. In *ESEC/FSE '20: 28th ACM joint european software engineering conference and symposium on the foundations of software engineering, virtual event, usa, november 8-13, 2020*, ed. Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, 826–837. ACM.

Zhang, Yuhao, Shiqi Wang, Haifeng Qian, Zijian Wang, Mingyue Shang, Linbo Liu, Sanjay Krishna Gouda, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. 2024b. Codefort: Robust training for code generation models. 2405.01567.

Zhu, Chen, W. Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2019a. Transferable clean-label poisoning attacks on deep neural nets. In *Proceedings of the 36th international conference on machine learning, ICML 2019, 9-15 june*

2019, long beach, california, USA, ed. Kamalika Chaudhuri and Ruslan Salakhutdinov, vol. 97 of *Proceedings of Machine Learning Research*, 7614–7623. PMLR.

Zhu, Ligeng, Zhijian Liu, and Song Han. 2019b. Deep leakage from gradients. In *Advances in neural information processing systems 32, NeurIPS*, 14747–14756.