Understanding Neural Networks Properties by Testing and Specification Mining

Xiaojie Xu, School of Computer Science
McGill University, Montreal
August, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

Under the supervision of Professor Xujie Si

Abstract

This thesis investigates neural network properties and behaviours from multiple perspectives, aiming to enhance our understanding of these complex models and improve their reliability, generalization, and interpretability. We focus on three main projects: neural activation patterns (NAPs), *Delta Debugging* for image reduction, and scalar invariant neural networks. First, we examine NAPs using a custom mining algorithm on simple Feed-Forward Neural Networks (FNNs) with the MNIST dataset, demonstrating their potential for formal verification and providing insights into neural network decision-making. Next, we adapt the *Delta Debugging* algorithm for image reduction, leveraging NAPs to generate minimal images that preserve recognizable features. This approach allows us to gain more insights into neural network properties and the decision-making process while validating our NAP findings. Finally, we introduce scalar invariant neural networks, which exploit the directional nature of image data to achieve scalar invariance. These networks demonstrate improved resilience to brightness and contrast adjustments and robust performance, with properties showcased by analysis via NAPs. This shows potential for more robust neural network applications in image classification tasks for real-world scenarios. Our work contributes to both the theoretical understanding and practical applications of neural networks, offering new aspects for analysis and novel architectural approaches.

Abrégé

Cette thèse examine les propriétés et comportements des réseaux neuronaux sous plusieurs angles, dans le but d'améliorer notre compréhension de ces modèles complexes et d'accroître leur fiabilité, leur capacité de généralisation et leur interprétabilité. Nous nous concentrons sur trois projets principaux : les motifs d'activation neuronale (NAP), le *Delta Debugging* pour la réduction d'images, et les réseaux neuronaux scalaires invariants. Tout d'abord, nous examinons les NAP en utilisant un algorithme de fouille personnalisé sur des réseaux neuronaux à propagation avant (FNN) simples avec le jeu de données MNIST, démontrant leur potentiel pour la vérification formelle et fournissant des insights sur la prise de décision des réseaux neuronaux. Ensuite, nous adaptons l'algorithme de *Delta Debugging* pour la réduction d'images, en utilisant les NAP pour générer des images minimales qui conservent des caractéristiques reconnaissables. Cette approche nous permet de mieux comprendre les propriétés des réseaux neuronaux et le processus de prise de décision tout en validant nos découvertes sur les NAP. Enfin, nous introduisons les réseaux neuronaux scalaires invariants, qui exploitent la nature directionnelle des données d'image pour atteindre une invariance scalaire. Ces réseaux démontrent une résilience accrue aux ajustements de luminosité et de contraste ainsi qu'une performance robuste, avec des propriétés mises en évidence par l'analyse via les NAP. Cela montre un potentiel pour des applications de réseaux neuronaux plus robustes dans les tâches de classification d'images pour des scénarios réels. Notre travail contribue à la fois à la compréhension théorique et aux applications pratiques des réseaux neuronaux, offrant de nouvelles perspectives d'analyse et des approches architecturales novatrices.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Xujie Si, for their unwavering support, guidance, and encouragement throughout the course of my research. Their insights and expertise have been invaluable to this work.

I extend my sincere thanks to my colleague, Chuqin Geng, for providing me with opportunities to participate in engaging projects that broadened my research horizons and enriched my learning experience.

I am also deeply thankful to my thesis reviewer, Prof.Jin Guo, for their valuable feedback and insightful suggestions. Their thoughtful review has been instrumental in enhancing the clarity and rigor of this work, and I deeply appreciate their time and expertise.

I am profoundly grateful to my family for their unconditional love, support, and patience throughout my academic journey. Special thanks to my parents, Kaige Ma and Guangming Xu, for their constant encouragement and unwavering belief in my abilities.

Lastly, I would like to acknowledge everyone who, in various ways, contributed to the successful completion of this thesis. Your support and encouragement have been instrumental in my academic journey.

To all mentioned and those whose names may not appear here but whose contributions are deeply appreciated, I offer my heartfelt thanks.

Table of Contents

List of Figures

List of Tables

Abbreviations

CNN Convolutional neural network

FNN Feedforward neural network

NAP Neural activation pattern

VNNCOMP Verified neural networks competition

Chapter 1

Introduction

Neural networks have revolutionized machine learning, achieving remarkable performance across various domains, such as computer vision [16, 35], natural language processing [8, 37], and healthcare [29, 40]. However, as these models are increasingly deployed in critical applications, understanding their inner workings has become crucial for ensuring reliability, generalization, and safety. This thesis addresses the pressing need to bridge the gap between neural networks' performance and our comprehension of their behaviours. We approach this challenge from three angles.

First, we investigate neural activation patterns (NAPs) and their mining process. Through comprehensive evaluations, we analyze the relationship between NAPs and datasets to understand neural network behaviors. We demonstrate potential applications of NAPs in formal verification, highlighting the inadequacies of previous specification diagrams used in the VNNCOMP.

Second, we further explore neural network behaviors using NAPs. We augment the software engineering algorithm *Delta Debugging* [44], applying it to image classification tasks to generate minimal recognizable images. By integrating NAPs from our previous work into this augmented algorithm, we provide insights into the decision-making processes of neural networks.

Finally, we shift our focus from exploring existing properties to modifying neural networks to create new properties, specifically scalar invariance. Based on our findings regarding the directional nature of image data, we introduce scalar invariant neural net-

works. We explore their potential in image processing applications by comparing them with state-of-the-art benchmarks. Leveraging NAPs, we study the robustness advantages of these networks compared to traditional models with bias terms.

The main contributions of this thesis can be summarized as follows:

- We propose our own definition of NAPs and implement a corresponding NAP-extract algorithm. Our work demonstrates the potential of using NAPs in formal verification of neural network robustness.

- We adapt the *Delta Debugging* algorithm to create an image reduction method. This approach, enhanced by our NAP findings, provides insights into neural network decision-making processes.

- We introduce a new neural network architecture called the Scalar Invariant Neural Network. We explore its potential in image classification tasks and investigate its unique robustness properties, leveraging our understanding of NAPs to provide a comprehensive analysis of this novel architecture.

Through these diverse approaches, we aim to contribute both to the theoretical foundations of neural networks and to their practical applications. Our work not only offers new tools for network analysis but also proposes novel architectural concepts. By examining neural networks from these multiple perspectives, we seek to enhance their interpretability, reliability, and performance in real-world scenarios.

This thesis is structured as follows: Chapter 2 provides essential background knowledge of related work in the field, including previously existing research that served as the basis for our work. Chapters 3 to 5 include the main body of this thesis. We begin by discussing our exploration of NAPs in Chapter 3, followed by our adaptation of *Delta Debugging* for image analysis in Chapter 4. We then present our work on scalar invariant neural networks in Chapter 5. Each section details our methodologies, findings, and their implications. Finally, in Chapter 6, we conclude by discussing the limitations of our approaches and outlining promising directions for future research in this rapidly evolving field.

Chapter 2

Background

2.1 Neural activation analysis

In recent years, deep neural networks have made significant successes and become increasingly important across various domains. However, the achievement of state-of-the-art performance comes with certain disadvantages. The complexity of deep neural networks, while a key factor in their high performance on many real-world tasks, also makes these models difficult to understand, often resembling black boxes [46]. This black-box nature can be risky for real-world applications, impacting safety and industrial liabilities [15]. For example, a mistake made by a self-driving car could result in the loss of several human lives. Therefore, it is crucial to understand the working mechanism of deep neural networks and create more reliable and trustworthy tools for the future.

One research direction focuses on studying neuron activations through visualization and analysis to gain more interpretability of neural network models. Olah et al. [26] studied feature visualization by optimization, aiming to trigger and visualize activation maximization in two ways: by finding examples in practice or creating inputs that most activate the neurons. The visualizations reveal the patterns learned by different neurons or layers of the model. They even observed that individual neurons could be considered *basis vectors* of activations, allowing arithmetic operations to combine different patterns. These visualizations provide deeper insights into the black-box nature of neural

networks. When combined with other tools, they can significantly enhance human understanding of neural network behavior.

Bäuerle et al. [2] examined neuron activation through layer-wise neural activation patterns. Instead of focusing on the high values of individual neurons triggered by specific inputs, they considered the entire activation distribution by analyzing the pattern formed by the values of all neurons in each layer. They collected and categorized images sharing similar neural activation patterns and observed that images following the same neuron activation patterns share common concepts. Moreover, the deeper the layers from which neuron activation patterns were extracted, the less abstract the shared concepts became. By associating neural activation patterns with image samples and their shared common concepts, this approach provides more straightforward visualizations of what models have learned. Additionally, by comparing the learned concepts and categorized images from the same dataset across different models, it helps observe learning differences. More advanced models tend to build classifications at earlier layers. Furthermore, the categorized images and their corresponding common concepts help identify potential biases in data collection.

Apart from helping in better understanding neural networks, the insights we obtained from neural activations also help model performance. Wang et al. [41] studied neuron activations from a statistical perspective and conducted layer-wise entropy analysis to examine how information flows between layers. They visualized feature maps of convolution-pooling layers, demonstrating the increased abstraction level as the layers get deeper. They also analyzed neuron activation patterns and their corresponding entropies, including the entropy of individual fully connected layers and the joint entropy of successive fully connected layers. Their observations showed that the gradient of entropy change between layers is related to the number of fully connected layers, which could potentially be an indicator for determining the right number of fully connected layers to improve model performance.

Although the above work represents only a small part of the research on model explainability through neuron activations, it highlights the significant value of studying

neural activations. This importance is what motivates us to conduct our study in this area.

2.2 Robustness Verification and VNNCOMP

Formal verification, as described by Meng et al. [24], involves mathematically proving the correctness of a software or hardware system against specified properties or specifications. Model checking is one of the classic verification techniques which includes two phases: specification and verification. In the specification phase, two key elements are required: property formalization, which defines the expected behaviour of the system, and a reduced framework, representing the system in a simplified form that preserves essential features for proper simulation during verification. In the verification phase, the verifier performs reasoning tasks and provides outputs indicating whether the specified properties hold.

Verification against the adversarial robustness of neural networks is a prominent direction within formal verification, driven by the need for trustworthy AI. For robustness verification, the neural network is considered a function mapping inputs to outputs, such as classification results. Within a maximum perturbation magnitude, usually denoted as ϵ, the network is considered non-robust if there exists an input x with classification y that can be perturbed within the range of ϵ to yield a different classification result.

VNN-COMP (Verified Neural Networks Competition) [38] is an annual competition focused on the formal verification of neural networks. Its goal is to provide standardized benchmarks for evaluating the performance of different neural network verification tools. With the provided benchmarks of datasets and models, participating tools need to verify the corresponding properties or specifications within given runtime limits. These tools are evaluated based on various metrics such as correctness, speed, scalability, and ease of use. In Chapter 3, we will use the benchmarks focusing on verifying adversarial robustness as references for some of our evaluations.

2.3 Delta Debugging

Delta Debugging, an algorithm firstly introduced by Andreas Zeller [44] in 1999, is rooted in the concept of regression containment. It addresses software regressions by automatically identifying changes made to a previously functional program that caused it to crash. This methodology was further developed in [44], laying the groundwork for the discussions presented in Chapter 4.

As a technique initially developed to identify the changes that cause a working program to fail, delta debugging focuses on analyzing the differences, or "deltas," between the working and broken versions of the program. Since an empty input can be considered a working program, the difference between an empty input and a failure input is the failure input itself, allowing this technique to be generalized to locating bugs in failure inputs, as demonstrated in [44].

Two versions of the *Delta Debugging* algorithm are discussed in [44]: $ddmin$ and dd, representing the *Minimizing Delta Debugging Algorithm* and *General Delta Debugging Algorithm*, respectively. The primary distinction between these versions lies in their objectives. $ddmin$ aims to identify the minimal test case that triggers program failure, ensuring that removing any additional atomic parts will eliminate the failure. On the other hand, dd, an extended version of $ddmin$, seeks to identify the minimal difference between a passing subset and a failing subset of the original test case.

This difference in objectives leads dd to address a significant weakness of $ddmin$: the substantial increase in complexity with larger input sizes. However, since we are studying the MNIST dataset, which consists of relatively small image sizes and uses a simple feedforward neural network, runtime is not a major concern for us. Therefore, we will focus on $ddmin$ as our baseline for this work, providing an overview of the algorithm for background knowledge.

Definition 2.3.1 (n-minimal test case)**.** A test case $c \subseteq c_{\text{✘}}$ is n-minimal if $\forall c' \subset c \cdot |c| - |c'| \leqslant n \Rightarrow (test((c') \neq \text{✘})$ holds. Consequently, c is 1-minimal if $\forall \delta_i \in c \cdot test(c - \{\delta_i\}) \neq \text{✘}$ holds, where δ_i denotes some arbitrary elementary change of $c_{\text{✘}}$ could be decomposed into.

Algorithm 1 Zeller and Hildebrandt's Minimizing Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\varnothing) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm ddmin*(c) is

$$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2)$$

where

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✘} \\ & \text{(“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘} \\ & \text{(“reduce to complement”)} \\ ddmin_2(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)) & \text{else if } n < |c'_{\text{✘}}| \text{ (“increase granularity”)} \\ c'_{\text{✘}} & \text{otherwise (“done”)} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i$, $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leqslant |c'_{\text{✘}}|$.

Algorithm 1 is the $ddmin$ algorithm adapted from [44]. The algorithm takes a failing test case, denoted as $c_{\text{✘}}$, and a testing function, $test$, which determines if a test case *pass* (denoted as ✔) or *fail* (denoted as ✘). An empty test case is considered a *pass* test case. The goal of the algorithm is to find the 1-minimal test case $c'_{\text{✘}}$ that is simplified from $c_{\text{✘}}$. According to Definition 2.3.1, a test case is 1-minimal if removing any atomic part eliminates the failure.

The essential idea of this algorithm is to split the input test case into subsets and remove as many subsets as possible to reach the 1-minimal test case while still preserving the original failure. The function $ddmin_2$ conducts the reduction process of delta debugging, taking two parameters: $c'_{\text{✘}}$, a *fail* test case, and n, the number of partitions or subsets. For each recursive step of running $ddmin_2(c'_{\text{✘}}, n)$, we first split the input failure test case $c'_{\text{✘}}$ into n subsets, denoted as $\Delta_1, \Delta_2, \ldots, \Delta_n$. Correspondingly, we have their complements, denoted as $\nabla_1, \nabla_2, \ldots, \nabla_n$, where $\nabla_i = c'_{\text{✘}} - \Delta_i$. Then, we have four possible outcomes of this run:

Reduce to subset. We run tests on each subset, denoted as $test(\Delta_i)$ for $i = 1, \ldots, n$. If any subset Δ_i fails the test, we set $n = 2$ and proceed with $ddmin_2(\Delta_i, 2)$ for the next recursive step.

Reduce to complement. If none of the subsets produce the original failure, we then test the complements of the subsets, denoted as $test(\nabla_i)$ for $i = 1, \ldots, n$. If any ∇_i fails the test, we proceed with $ddmin_2(\nabla_i, \max(2, n-1))$. In general, we set $n = n - 1$ to maintain the same level of granularity for the next step, but we also consider the $n = 2$ case to ensure a minimal granularity level of 2.

Increase granularity. If neither of the previous cases works, it might be because the current granularity level is insufficient. Therefore, we double the granularity and proceed with $ddmin_2(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n))$. However, this is only done if $n < |c'_{\text{✘}}|$ as a test case cannot be partitioned into infinitely small subsets.

Done. If we have already reached the highest level of granularity, i.e., $n = |c'_{\text{✘}}|$, and none of the previous cases work, we have found the 1-minimal test case.

The delta debugging process begins with $n = 2$ (i.e., $ddmin_2(c'_{\text{✘}}, 2)$) and concludes when the *Done* case is reached.

This algorithm provides a straightforward and easy-to-implement methodology for automated debugging [44]. As we know, debugging plays a significant role in developing a functional program, and this process can be divided into two main tasks: reproducing the failure and finding the root cause of the failure [39]. The debugging process to accomplish these tasks includes three detailed activities: fault localization, understanding, and correction [27].

Fault localization is the primary step, as we need to identify the program segment containing the failure before proceeding with the other two steps. However, in real-world cases, programs can be very large, requiring significant time and repetition during this process. Understanding the failure also demands much from programmers, such as their ability to comprehend the context around the bugs and their past experience dealing with similar issues. Consequently, debugging can be a time-consuming, challenging, and tedious task that requires substantial resource investment from both computational and human levels. The Delta Debugging approach significantly aids in the fault localization

step by removing irrelevant information from the input failure program, potentially saving a huge amount of work and resources.

The intrinsic idea of the *Delta Debugging* algorithm is fundamentally a *divide-and-conquer* process that reduces input while maintaining certain invariant properties [21]. This versatile approach has found applications beyond its original use in debugging program code. It has been successfully adapted to various domains, including precision tuning [14, 30], model signal awareness [33, 34], semantic history slicing [23], etc. The algorithm's ability to systematically reduce complex inputs while preserving specific characteristics makes it a valuable tool in diverse areas of computer science and software engineering.

2.4 Fixed-update Initialization

Before the invention of batch normalization, the changing distribution of inputs during the training process, also referred to as *internal covariate shift*, raised a significant problem. This shift could lead to issues such as saturation and vanishing gradients, complicating the model training process. To address these problems, a combination of *ReLU* [25], careful initialization [13, 31], and small learning rates was used as a solution. However, this approach made the training process very slow [19].

Batch normalization, proposed by Ioffe et al. [19], was introduced to stabilize the distribution of layer inputs. This innovation allowed the use of larger learning rates and reduced concerns about initialization. Consequently, batch normalization significantly improved training speed, stability, and overall performance and has been widely adopted in various state-of-the-art neural networks.

Although batch normalization has demonstrated significant success in model training, Zhang et al. [45] conducted a study revealing that the benefits of batch normalization are not exclusive to this technique. They proposed fixed-update initialization (Fixup), an initialization method that properly rescales standard initialization for deep residual networks to address the gradient explosion and vanishing problems. Their findings showed that models could be trained stably at a maximum learning rate without the need for nor-

malization. Additionally, their experiments indicated that with proper regularization, the performance of Fixup ResNets could achieve state-of-the-art test accuracies.

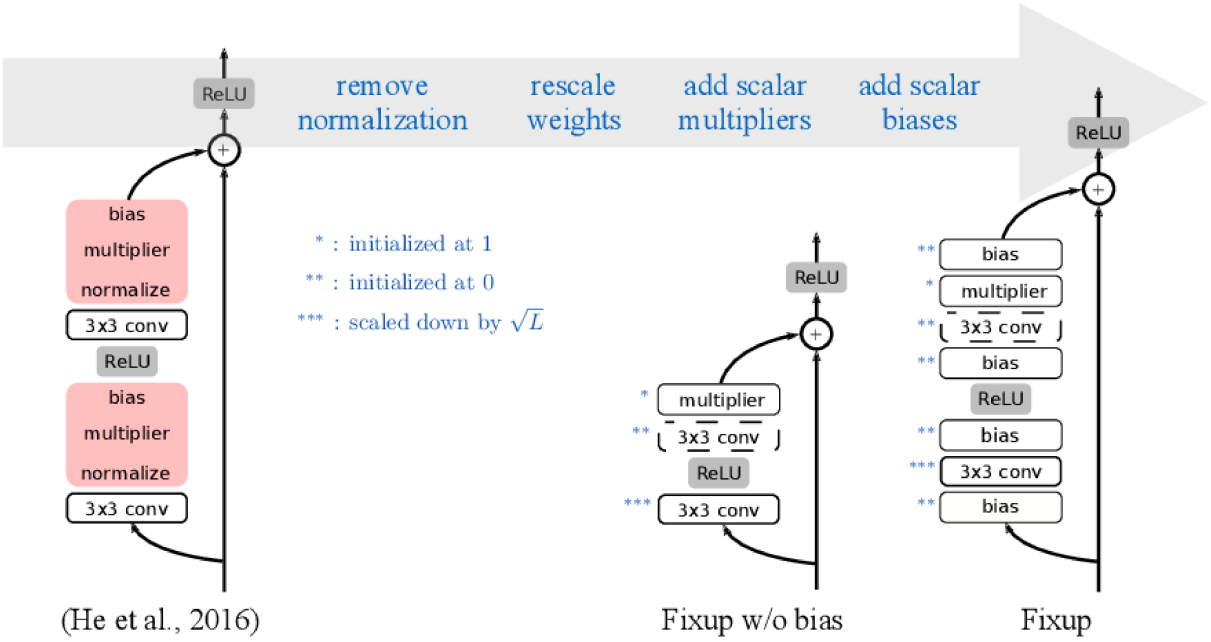


Figure 2.1: This figure illustrates the comparison between the original basic ResNet block and the ResNet block post-application of the fixup method. On the left, the original basic ResNet block is depicted, with Batch Normalization layers [19] highlighted in red. The middle and right structures depict the ResNet block subsequent to applying the fixup method, presented both without and with bias layers, respectively. *Note*: This figure is adapted from [45].

In summary, the Fixup initialization method proposed by Zhang et al. [45] includes the following three rules:

1. Initialize the classification layer and the last layer of each residual branch to 0.

2. Initialize every other layer using a standard method (e.g., the methods used by He et al. [17]), and scale only the weight layers inside residual branches by $L^{-\frac{1}{2m-2}}$, where L is the number of residual branches the model has, and m is number of layers inside a single residual branch.

3. Add a scalar multiplier (initialized at 1) in every branch and a scalar bias (initialized at 0) before each convolution, linear and element-wise activation layer.

Figure 2.1 illustrates the differences between a normal ResNet block and a Fixup ResNet block, including versions with and without bias terms. It also visualizes how the three rules mentioned above are applied to Fixup ResNet blocks.

This work draws our attention because the scalar invariant neural network we proposed in Chapter 5 relies on removing all bias terms from the network. The essential idea is that by eliminating bias terms, the network should remain invariant in its predictions when a positive scalar is applied to the input image. However, this approach encounters issues with removing bias terms from the batch normalization layers. Therefore, we need an alternative to batch normalization that can preserve its benefits while allowing us to construct scalar invariant neural networks.

Chapter 3

Neural Activation Pattern

With the development of the deep learning and it's success in performance, knowing the mechanism behind the black-box of neural networks has also get more and more significant. As we mentioned in Chapter 2.1, the study of neural activation is a popular area within model explainability. Beyond achieving good performance, it is crucial to understand neural networks better to improve the models. Consequently, this research direction has drawn our attention. In this chapter, we will be exploring the NAPs of models within a domain of image classification tasks and the potential of extending the application of NAPs to formal verifications which is sparked by our observation from evaluations.

3.1 NAP definition and properties

Before we dive into the details of all the experiments and evaluations, we will first provide essential information about NAPs. The Rectified Linear Unit (*ReLU*) [25] is a widely used activation function in neural networks, defined by the mathematical expression $ReLU(x) = \max(0, x)$. In simpler terms, *ReLU* outputs the input value x if it is positive or zero; otherwise, it returns zero. Consequently, each neuron can exhibit two activation states: *activated* if the output value is positive, and *deactivated* if the output is zero.

Definition 3.1.1 (Neural Activation Pattern)**.** A *Neural Activation Pattern (NAP)* of a neural network is a tuple $\mathcal{P} := (A, D)$, where A and D are two disjoint subsets of activated and deactivated neurons, respectively.

Definition 3.1.2 (Partially ordered NAP)**.** For any given two NAPs $\bar{\mathcal{P}} := (\bar{A}, \bar{D})$ and $\mathcal{P} := (A, D)$. We say $\bar{\mathcal{P}}$ subsumes $\mathcal{P}$ iff A, D are subsets of $\bar{A}, \bar{D}$ respectively. Formally, this can be defined as:

$$\bar{\mathcal{P}} \preccurlyeq \mathcal{P} \iff \bar{A} \supseteq A \; and \; \bar{D} \supseteq D \tag{3.1}$$

Moreover, two NAPs $\bar{\mathcal{P}}$ and $\mathcal{P}$ are equivalent if $\bar{\mathcal{P}} \preccurlyeq \mathcal{P}$ and $\mathcal{P} \preccurlyeq \bar{\mathcal{P}}$.

Definition 3.1.3 (NAP Extraction Function)**.** A NAP Extraction Function E takes a neural network $\mathcal{N}$ and an input x as parameters, and returns a NAP $\mathcal{P} := (A, D)$ where A and D represent all the activated and deactivated neurons of $\mathcal{N}$ respectively when passing x through $\mathcal{N}$.

Definition 3.1.4 (δ-relaxed NAP)**.** We introduce a relaxing factor $\delta \in [0, 1]$. We say a NAP is δ-relaxed with respect to the label ℓ, denoted as $\mathcal{P}^{\delta}_{\ell} := (A^{\delta}_{\ell}, D^{\delta}_{\ell})$, if it satisfies the following condition:

$$\exists S'_{\ell} \subseteq S_{\ell} \; s.t. \; \frac{|S'_{\ell}|}{|S_{\ell}|} \geqslant \delta \; \textbf{and} \; \forall x \in S'_{\ell}, \; E(\mathcal{N}, x) \preccurlyeq \mathcal{P}^{\delta}_{\ell} \tag{3.2}$$

3.2 NAP mining

In accordance with our specified definition of NAP (definition 3.1.1), we have implemented the preliminary version of the NAP mining algorithm. This algorithm systematically extracts activated and deactivated neurons based on their activation status in each sample from the training set. In the NAP, only neurons consistently activated across every training sample are included in the activated set, and similarly, neurons are considered as deactivated if they consistently remain inactive throughout all training samples.

However, our experiments are conducted on a relatively small-sized and simple-structured neural network, and the MNIST dataset is relatively small compared to many other ex-

isting image datasets. This means that gathering neurons by only selecting those that are consistently activated or deactivated throughout the entire training set might not be an effective mining algorithm. This limitation becomes particularly apparent when applying more complex neural network architectures to larger datasets. In such cases, the number of neurons increases significantly, and the datasets contain more samples for each class with greater diversity. This complexity makes it more challenging to identify consistently activated or deactivated neurons.

Therefore, we introduce a relaxation factor, denoted as δ, into our NAP, serving as a means to regulate its abstraction level. And we implemented the corresponding mining algorithm in Algorithm 2[1]. For a NAP associated with a specific label ℓ, when mining the NAP with $\delta = 1.0$ using Algorithm 2, denoted as $\mathcal{P}_\ell^{\delta=1.0}$, it covers all training samples labelled ℓ. This configuration yields the most precise NAP, yet concurrently, it is the least specific. In this context, $\mathcal{P}_\ell^{\delta=1.0}$ can be perceived as the highest level of abstraction for the common neural representation of label ℓ. However, excessive abstraction introduces the risk of under-fitting, thereby increasing the likelihood of Type II Errors in NAPs. As δ decreases, the probability of a neuron being selected to form a NAP increases, resulting in more specific NAPs. While this adjustment may help reduce Type II Errors, it could simultaneously increase the rate of Type I Errors, potentially impacting the recall rate.

Utilizing Algorithm 2, we mined NAPs for each class within the MNIST dataset across four distinct values of δ. The comprehensive examination involved the entire testing set, evaluating the coverage of samples for each class, as outlined in Table 3.1. For each NAP, we present the count of test samples sharing the same label as the NAP, denoted as $\mathcal{P}_\ell^\delta$, and those with a label different from ℓ following $\mathcal{P}_\ell^\delta$, designated as $\bar{\ell}$. The initial row identifies the label and total number of test samples for that label. Two columns are provided beneath each label, detailing the coverage of test samples. For instance, considering label 0, there are a total of 980 test samples. Among these, 976 adhere to $\mathcal{P}_{\ell=0}^{\delta=1.0}$, and 20 samples from classes 1 to 9 also follows $\mathcal{P}_{\ell=0}^{\delta=1.0}$. If we decrease δ to 0.99, we see that the number of test samples from class 0 decrease but at the same time, the number of sample from

[1]Note that this algorithm is an approximate method for mining δ-relaxed NAP, whereas δ should be greater than 0.5, otherwise, $A_\ell^\delta \bigcap D_\ell^\delta \neq \varnothing$. We leave more precise algorithms for future work.

Algorithm 2 NAP Mining Algorithm

Input: relaxing factor δ, neural network $\mathcal{N}$, subset of dataset containing all samples with same label S_ℓ
Initialize a counter c_k for each neuron v_k
for $x \in S_\ell$ **do**
  compute $E(\mathcal{N}, x)$
  for each k **do**
    if v_k is activated **then**
      $c_k \mathrel{+}= 1$
    end if
  end for
end for
$A_\ell \leftarrow \{v_k \mid \frac{c_k}{|S_\ell|} \geqslant \delta\}$
$D_\ell \leftarrow \{v_k \mid \frac{c_k}{|S_\ell|} \leqslant 1 - \delta\}$
$\mathcal{P}_\ell^\delta \leftarrow (A_\ell, D_\ell)$

other than 0 also decreases, which indicates that the mined NAP $\mathcal{P}_{\ell=0}^{\delta=0.99}$ is more specified towards class 0 but at the same time, it will cause more test samples from class 0 also failed to follow it. In summary, the effectiveness of NAPs greatly depends on the precision-recall trade-off. Therefore, selecting the appropriate δ value or the optimal level of abstraction becomes significant when leveraging NAPs in practical applications.

Table 3.1: The number of the test images in MNIST that follow a given NAP$^\delta$. For a label i, $\bar{i}$ represents images with labels other than i yet follow NAP$^\delta_{\ell=i}$. The leftmost column is the values of δ. The top row indicates how many images in the test set are of a label.

	0 (980)		1 (1135)		2 (1032)		3 (1010)		4 (982)		5 (892)		6 (958)		7 (1028)		8 (974)		9 (1009)	
	0	$\bar{0}$	1	$\bar{1}$	2	$\bar{2}$	3	$\bar{3}$	4	$\bar{4}$	5	$\bar{5}$	6	$\bar{6}$	7	$\bar{7}$	8	$\bar{8}$	9	$\bar{9}$
1.00	967	20	1124	8	997	22	980	13	959	25	874	32	937	26	1003	28	941	22	967	12
0.99	775	1	959	0	792	4	787	2	766	3	677	1	726	4	809	2	696	3	828	4
0.95	376	0	456	0	261	1	320	0	259	0	226	0	200	0	357	0	192	0	277	0
0.90	111	0	126	0	43	0	92	0	76	0	24	0	45	0	144	0	44	0	73	0

To look into the details of how the relaxation factor affects on the extraction of our NAPs, and to have a better view of how to manage the trade-off between the TYPE I and TYPE II errors as we mentioned above, we checked on the overlapping of neurons for each class under different values δs and displayed the results in Figure 3.1 and Table 3.2.

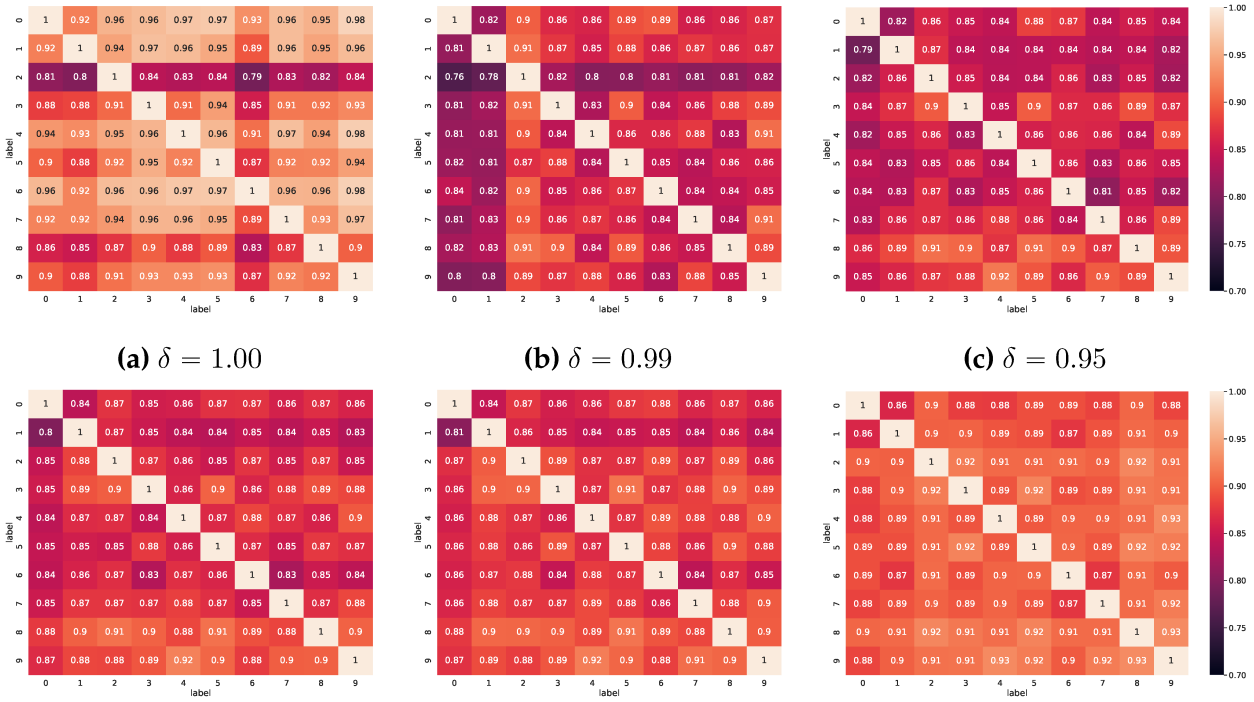
Table 3.2: The maximum overlap ratio for each label (class) on a given NAP$^{\delta}$ for MNIST. Each cell is obtained by $\max_i |N^{\delta}_{col} \bigcap N^{\delta}_i|/|N^{\delta}_{col}|$ where N^{δ}_{col} is the set of neurons in the dominant pattern for the label (class) in the header of the column of the selected cell with the given δ, N_i is the set of neurons in the dominant pattern for the label (class) i with the given δ.

	0	1	2	3	4	5	6	7	8	9
1.00	0.959	0.928	0.963	0.966	0.972	0.973	0.930	0.965	0.957	0.981
0.99	0.844	0.834	0.911	0.901	0.881	0.898	0.895	0.884	0.880	0.908
0.95	0.864	0.885	0.909	0.904	0.915	0.908	0.899	0.897	0.890	0.893
0.90	0.877	0.900	0.910	0.901	0.921	0.910	0.890	0.899	0.900	0.901
0.85	0.876	0.904	0.904	0.900	0.919	0.913	0.893	0.907	0.904	0.900
0.75	0.893	0.922	0.913	0.912	0.928	0.925	0.905	0.916	0.916	0.913
0.50	0.903	0.905	0.925	0.923	0.926	0.923	0.907	0.918	0.927	0.927

Figure 3.1 illustrates a heatmap depicting the overlap ratio between NAPs of any two classes across six different values of δ. Within each column of the heatmap, the overlap ratio is computed by dividing the number of overlapping neurons in the NAPs of the classes indicated via the row and column by the total number of neurons in the NAP of the class specified by the column. This asymmetry in values along the diagonal is due to the calculation methodology.

The varying shades of color in the heatmap reveal a trend where, as δ decreases, the overlap ratios initially decline before showing a general increase. This observation may be attributed to the relaxation of constraints on determining if a neuron is considered activated or deactivated. A slightly looser criterion, or lower δ value, leads to the inclusion of more neurons in the NAPs, which increases the total number of neurons included in the NAPs and lowers the overlap ratio. However, if we continue decreasing δ, the likelihood of more neurons appearing in both NAPs from different classes increases, resulting in higher overlap ratios.

Table 3.2 presents the maximum overlap ratio for a single class. Specifically, it represents the highest overlap ratio between a reference class and any other class. This table essentially extracts the maximum values from each column, excluding the diagonal ele-



(a) $\delta = 1.00$ **(b)** $\delta = 0.99$ **(c)** $\delta = 0.95$

(d) $\delta = 0.90$ **(e)** $\delta = 0.85$ **(f)** $\delta = 0.50$

Figure 3.1: Overlap ratio of the dominant pattern of two labels (classes) on a given NAP$^{\delta}$. Values in each grid are obtained by $|N^{\delta}_{col} \bigcap N^{\delta}_{row}|/|N^{\delta}_{col}|$ where N^{δ}_{col} is the set of neurons in the dominant pattern for the label (class) of the column of the selected grid with the given δ, N_{row} is the set of neurons in the dominant pattern for the label (class) of the row of the selected grid with the given δ.

ments, in our heatmap depicted in Figure 3.1. It is noteworthy that the values in each column of the table follow a pattern where the overlap ratio decreases initially and then increases as δ decreases.

3.3 Evaluations and potential applications

At this point, we have all the necessary components for mining an appropriate NAP from our model. Now, we can dig deeper into NAP itself and explore its potential applications to real-world topics and issues.

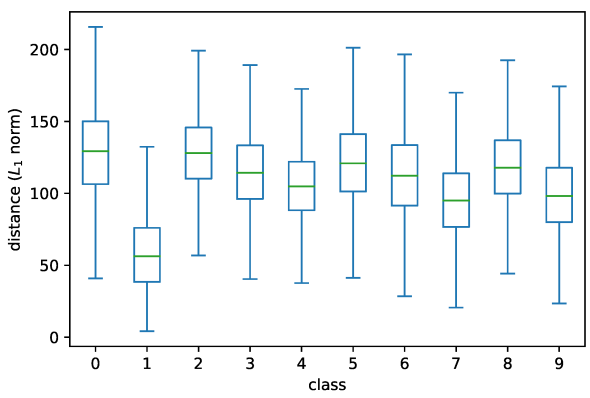
3.3.1 L_1, L_2 and L_∞ maximum verified bounds

We investigated the potential connections between our NAP and formal verification specifications for enhancing neural network robustness. One of existing methods in neural network verification is to construct specifications using a limited dataset, focusing on a small local neighbourhood surrounding known input samples. However, this approach fails to guarantee robustness for inputs lying beyond this restricted area. Additionally, these small changes might not be meaningful for neural networks performing image classification tasks, as augmented images with meaningful changes, i.e., with more semantic meanings, from the real world often originate from regions outside the verified neighbourhood [28]. This limitation highlights the need for a more comprehensive approach. Leveraging the inherent properties of our NAPs, we propose integrating them into the formal specification framework to address this issue.

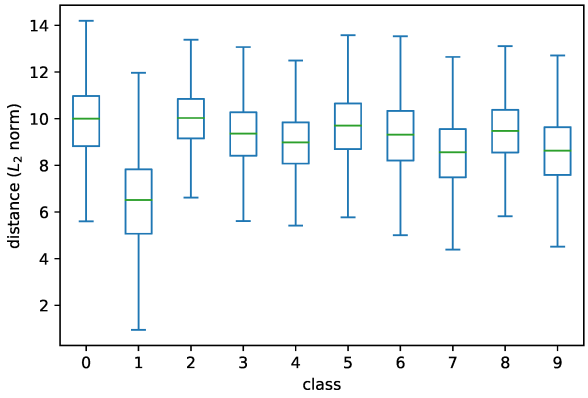
Primarily, we observe that the local neighbourhood size is insufficient for accommodating test samples and unseen inputs. To empirically substantiate this observation, we computed distances between all pairs of samples belonging to the same class across the entire MNIST dataset using L_1, L_2, and L_∞ norms. We employed the VNNCOMP-2021 [38] as a reference for our evaluation. While in VNNCOMP-2021, they primarily utilized the L_∞ norm for measuring distances between inputs, we also analyzed distributions with respect to L_1 and L_2 norms.

The result distributions, illustrated in Figure 3.2, indicate significant distances across all three measurement criteria, highlighting that it is insufficient for setting the local neighbourhood as the region to be verified. Our analysis reveals that the maximum verifiable bounds under L_1, L_2, and L_∞ norms are considerably smaller than the distances observed between real data instances. Particularly noteworthy is the finding that for each class, the smallest L_∞ distance between any two images sharing the same label exceeds 0.05. This value represents the largest perturbation magnitude employed in VNNCOMP-2021, as denoted by the red dotted line in Figure 3.2c.

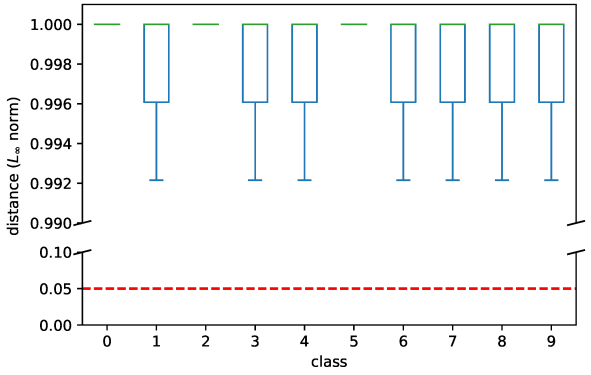
This implies that relying solely on the "data as specification" paradigm, wherein reference inputs with perturbations bounded in L_2 or L_∞ norms are utilized, is inadequate



(a) The distribution of L_1-norms between any two images from the same label. Images of digit (label) 1 are much similar than that of other digits.



(b) The distribution of L_2-norms between any two images from the same label. Images of digit (label) 1 are much similar than that of other digits.



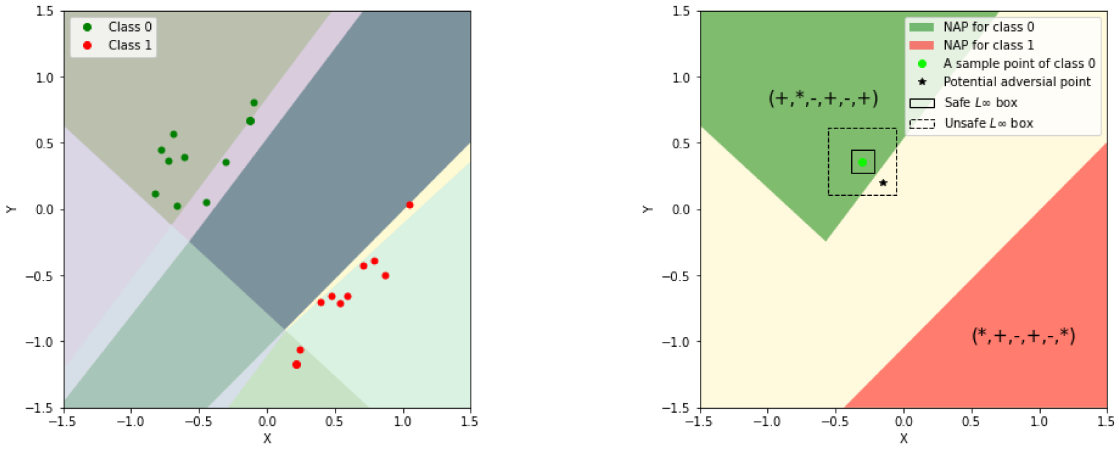
(c) The distribution of L_∞-norms between any two images from the same label. The red line is drawing at 0.05 – the largest ϵ used in [38].

Figure 3.2: Distances between any two images from the same label (class) are quite significant under different metrics of norm.

for verifying test set inputs or unseen data. The differences between training and testing data for each class typically exceed the perturbations permitted in specifications employing L_∞ norm-balls.

3.3.2 Case study: Visualing NAPs of a simple neural network

In addition to analyzing distance distributions, we conducted a straightforward case study that visually demonstrates the advantages of using NAPs to formulate specifications. In this case study, we employed a basic three-layer feedforward neural network (FNN) with the task of predicting a class of 20 points placed on a 2D plane. This simple FNN model has only 6 neurons and could achieve 100% accuracy in the prediction task.



(a) Linear regions in different colors are determined by weights and biases of the neural network. Points colored either red or green constitute the training set.

(b) NAPs are more flexible than L_∞ norm-balls (boxes) in terms of covering verifiable regions.

Figure 3.3: Visualization of linear regions and NAPs as specifications compared to L_∞ norm-balls.

In Figure 3.3a, we present the distribution of the 20 training data points in the input space, where it is, by colour, separated linear regions based on the weight and bias parameters of the trained FNN model. Each linear region represents a specific NAP including the states of all neurons. Table 3.3 provides the states of neurons for the samples, along with the number of samples following the same set of neuron states. The last column concludes the NAP for each class based on these neuron states. This table offers a clearer understanding of NAPs in the input space.

To visually represent NAPs for classes, we transpose them into image visualization as shown in Figure 3.3b. Here, the green and red regions correspond to the NAPs obtained

Table 3.3: The frequency of each ReLU and the NAPs for each label. Activated and deactivated neurons are denoted by $+$ and $-$, respectively, and $*$ denotes an arbitrary neuron state.

Label	Neuron states	#samples	NAP
0 (Green)	$(+,-,-,+,-,+)$	8	$(+,*,-,+,-,+)$
	$(+,+,-,+,-,+)$	2	
1 (Red)	$(+,+,-,-,+,-)$	7	$(*,+,-,+,-,*)$
	$(-,+,-,-,+,-)$	2	
	$(+,+,-,-,+,+)$	1	

for class 0 and 1, respectively, while the remaining area represents the unspecified region where label guarantees are absent. We selected a sample from class 0, denoted as a green dot in the figure, positioned close to the boundary of the NAP for class 0.

Considering the VNNCOMP framework, verification regions are L_∞-norm ϵ-balls centered around reference sample points. Successful verification within an ϵ-ball indicates a safe zone against adversarial attacks. We marked two distinct ϵ values and drew the corresponding neighbourhoods with boxes. The solid-line box represents a safe ϵ-ball where no adversarial examples exist, whereas the dotted-line box, larger in size, represents an unsafe zone, extending beyond the NAP region of class 0 into the unspecified area. Consequently, for this case, the verified region corresponds to the smaller solid-line box. However, by incorporating NAPs into specifications, we can verify a much broader and flexible region, extending our verification capabilities significantly.

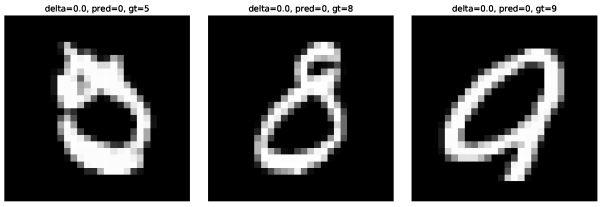
3.3.3 Misclassification examples

Up to this point, we have demonstrated that through the proper utilization of NAPs and an appropriate choice of the relaxation factor δ, we can derive effective representations or abstractions that capture patterns in the data. However, there remain unexplored scenarios that could provide further insights into how NAPs reveal patterns within groups of samples with shared attributes. In our context, this applies to sets of samples with the same label, contributing to a deeper understanding of neural network properties, particularly their decision-making processes.

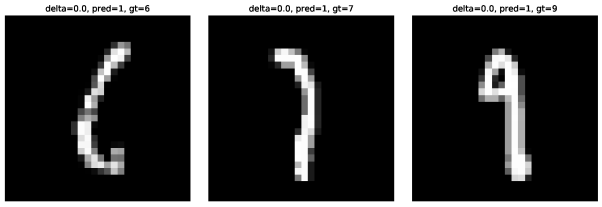
Examining misclassified samples that also follow the NAPs of the class of their incorrect prediction is an unexplored scenario that could provide more insights. As shown in Table 3.1, when δ=1.0, for each class, here are samples from other classes that also adhere to the NAPs of that class. We identified some instances and presented them visually in Figure 3.4. These examples are intriguing and, to some extent, provide evidence supporting the notion that NAPs serve as robust abstractions of their respective classes. These misclassified samples themselves are problematic, and it is more reasonable to assert that these images may have been assigned an inaccurate ground truth from a human perspective.

For instance, in Figure 3.4a, we present three instances where samples originally from classes other than 0 are misclassified as 0, following the NAP of label 0, that is, $\mathcal{P}_{\ell=0}^{\delta=1.0}$. Notably, the left-most image in Figure 3.4a presents a case with a ground truth of 5, which, from a human perspective, is challenging to identify as such. The neural network's decision to classify it as 0 seems reasonable. In the case of the middle and right-most images in Figure 3.4a, human observers may still recognize them as 8 and 9, respectively. However, it's acknowledged that in the middle image, the upper loop of the 8 is relatively small compared to the lower loop. In the right-most, the tail of the 9 is notably short, and the loop occupies a significant portion of its size. Despite being recognizable by humans, the neural network's decision to classify these instances as 0 is not surprising.

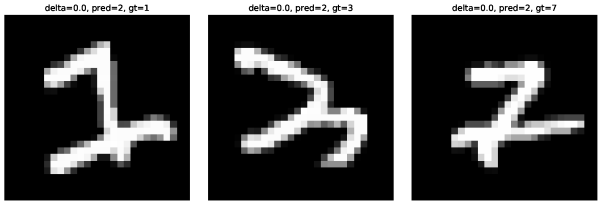
As seen in other examples, some cases are very problematic even from a human perspective. For instance, the first image in Figure 3.4b has a ground truth of 6 but is misclassified as 1, and the first image in Figure 3.4d has a ground truth of 5 but is misclassified as 3. Similarly, the middle image in Figure 3.4e has a ground truth of 6 but is misclassified as 4. Some samples are inherently problematic, while others, although recognizable by humans, align with the model's decision. Consequently, it is reasonable for these misclassified samples to follow the NAP of their erroneously predicted class. Moreover, the fact that misclassified samples also adhere to NAPs of other classes underscores the capability of NAPs to capture shared attributes among samples within the same class to a certain extent.



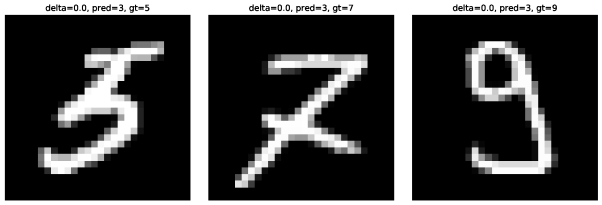
(a) Images are misclassified as 0 but have ground truths of 5, 8, and 9 from left to right.



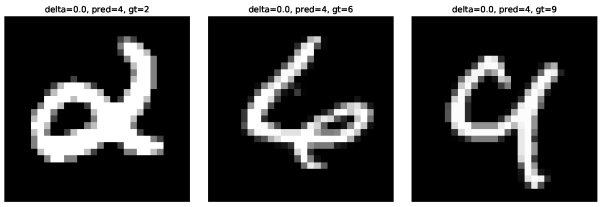
(b) Images are misclassified as 1 but have ground truths of 6, 7, and 9 from left to right.



(c) Images are misclassified as 2 but have ground truths of 1, 3, and 7 from left to right.

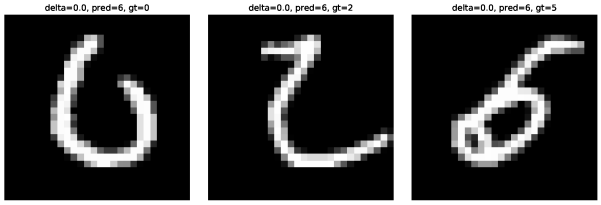


(d) Images are misclassified as 3 but have ground truths of 5, 7, and 9 from left to right.



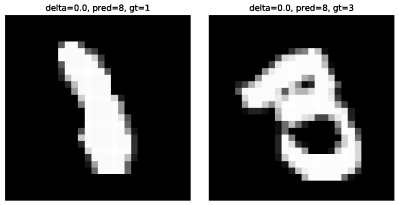
(e) Images are misclassified as 4 but have ground truths of 2, 6, and 9 from left to right.

(f) Images are misclassified as 5 but have ground truths of 3, 6, and 8 from left to right.

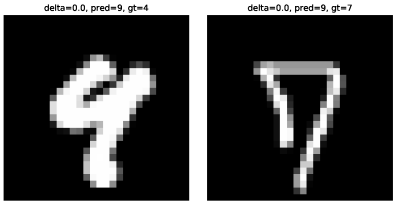


(g) Images are misclassified as 6 but have ground truths of 0, 2, and 5 from left to right.

(h) Images are misclassified as 7 but have ground truths of 2, 3, and 9 from left to right.



(i) Images are misclassified as 8 but have ground truths of 1 and 3 from left to right.



(j) Images are misclassified as 9 but have ground truths of 4 and 7 from left to right.

Figure 3.4: Some interesting MNIST test images misclassified as digits 0 to 9, respectively, which also follow the NAPs of their misclassified labels.

3.4 Discussion

In this chapter, we explored the behaviors of neural networks through NAPs. Using our custom implementation of the NAP mining algorithm, we conducted experiments on simple FNN with MNIST dataset to extract NAPs under varying relaxation constraints represented by δ. This allowed us to demonstrate the trade-off between NAP specificity and the coverage of test samples. In essence, we explored how NAPs can capture common features among test samples while balancing specificity.

Moreover, we observed the flexibility of NAPs in covering the input space, which inspired us to consider incorporating them into formal verification specifications. We illustrated this motivation through two experiments. First, we computed the L_1, L_2 and L_∞ distances between pairs of samples from the same class in the MNIST dataset to showcase the limited coverage of the original verification area defined by ϵ-balls, where we used the ϵ values from VNNCOMP-2021 as a reference. Our analysis revealed that this original verification area was insufficient to cover real-case inputs effectively.

Secondly, we presented a case study using a simple FNN and a dataset containing 20 points to highlight the broader and more flexible coverage of NAPs in the input space compared to ϵ-balls. This demonstrated the potential of integrating NAPs into formalized specification processes.

We also conducted additional analysis on the misclassified samples observed in previous experiments to gain deeper insights into neural networks. Specifically, we examined misclassified samples that adhere to the NAPs of their wrongly predicted labels. Our observations revealed that these incorrect predictions often appear reasonable because some samples pose challenges even from a human perspective, which partly explains why neural networks make these erroneous decisions.

The content discussed in this chapter is derived from the research outlined in [11]. Here, we mainly present an exploration of NAPs and the development of the idea of incorporating NAPs into formal verification specifications. The study detailed in [11] covers a comprehensive array of experiments conducted on established benchmarks such as MNIST and CIFAR-10 and detailed discussions on the models utilized and the Marabou

toolkit [20], which was employed in VNNCOMP-2021, to show the advantages of incorporating NAPs in formalizing specifications. These experiments collectively serve to show the potential of NAPs as a more reliable and extensible specification for neural network verification.

Chapter 4

Application of Delta Debugging in Image Classification Task

In this chapter, we explore a study involving the *Delta Debugging* Algorithm [43], a technique commonly used in software engineering to pinpoint the root causes of bugs in programs. This method iteratively tests and narrows down the input size to minimize the range causing issues. However, instead of focusing on program debugging, we apply this algorithm to the domain of machine learning tasks, particularly image classification. Our aim is to augment images using this algorithm to uncover intriguing features that may offer insights into the behaviour or properties of neural network models during prediction tasks.

4.1 Delta Debugging to images

For this part of work, we aim to apply the *Delta Debugging* Algorithm to image classification tasks to uncover interesting features of neural networks. Our technique retains the core concept of the *ddmin* method from [44], but since this is a different domain from programming debugging, we will make some modifications to the details.

The essential idea of *ddmin*, detailed in Chapter 2.3, is to iteratively remove parts from a failure input and test the reduced input. This process continues until removing any

additional atomic part causes the failure to disappear, thereby achieving a *1-minimal* state. This helps to identify the most relevant parts responsible for the failure.

In our case, we will use an image as the input and iteratively mask out pixels from the input image. At each iteration, we will test pre-defined conditions. This iterative process will continue until masking out even one more pixel from the image no longer satisfies these conditions, which corresponds to the *1-minimal* state in *ddmin*. We refer to our technique as *ddImage*.

The crucial aspect of this process is the conditions set for testing at each iteration, as they are the primary factors influencing the output and affecting the results of our observations. The details of these conditions and the different experiments we conducted will be discussed in Section 4.2. Different conditions were set for different experiments.

For masking out pixels, we will follow the *ddmin* approach of partitioning the program into subsets by splitting the pixels into subsets. Given a number n, we will evenly split the unmasked pixels into n subsets. The details of our technique are presented in Algorithm 3.

Algorithm 3 High level algorithm for applying delta debugging for image augmentation *ddImage*. The algorithm is initiated by $ddImage(I_o, l_o, \mathcal{N}, 2)$

Input: Image I; label l_I; neural network $\mathcal{N}$; number of subsets n

Split I into n pixelwise disjoint subsets Δ_1,...,Δ_n where $\forall|\Delta_i| \approx |I|/n$
if $\exists\Delta_i$ such that $\mathcal{N}(I - \Delta_i) == l_I$ **and** $Conditions(I - \Delta_i)$ is **true then**
  $i = \text{argmax}_{j \in n}\, Prob(l_I|\mathcal{N}(I - \Delta_j) == l_I)$
  $ddImage(I - \Delta_i, l_I, \mathcal{N} \max(n - 1, 2))$
else if $n < |I|$ **then**
  $ddImage(I - \Delta_i, l_I, \mathcal{N}, \min(2n, |I|))$
else
  return I
end if

We use I_o and l_o to denote an original input image and its label, respectively. The neural network used for prediction is denoted as $\mathcal{N}$. I and l_I denote an arbitrary image and a label, respectively, that are input into the algorithm. n denotes the number of partitions we want to split the image into by grouping the pixels of the image evenly into n subsets. *Conditions* is a function representing all the conditions or constraints we want

an image to satisfy to proceed as the input for the next iteration step. It returns *True* if all the conditions are satisfied; otherwise, it returns *False*. The symbol $-$ represents the operation of masking out a subset of pixels. For example, $I - \Delta_i$ means masking out all the pixels in subset Δ_i from image I. The function $|\cdot|$ gives the size of an image. In the original application of the *Delta Debugging* Algorithm, the size of a program is considered the number of atomic parts it has [44]. Here, as we are dealing with images, we consider an atomic part of an image to be one pixel. Therefore, $|\cdot|$ gives the number of pixels left in the input image, or more specifically, the number of unmasked pixels in the input image.

We specifically mention unmasked pixels because, when we pass an arbitrary image I to test for the conditions, we still pass all the pixels, including both masked and unmasked ones. It is impossible to remove the pixels completely from an image while keeping the remaining pixels in their original positions. Therefore, at each iteration step, we keep a mask to track which pixels are still unmasked. For the masked pixels, we simply set the pixel value to 0, making it pure black. [1]

For each iteration step, there are three possible outcomes, indicated by the *if-elseif-else* block. The *if* corresponds to the *Reduce to complement* case, *elseif* corresponds to the *Increase granularity* case, and *else* corresponds to the *Done* case. Here are the details for each case:

Reduce to complement. If there exists a subset of pixels Δ_i such that masking out these pixels from the input image I, i.e., $I - \Delta_i$, results in a prediction matching the label l_I and satisfies the *Conditions*, we will reduce to the complement that gives the highest prediction probability. We want to ensure that the augmented image maintains the same prediction as the ground truth because observing patterns would be meaningless if the augmented image does not retain the same label as the ground truth. This approach differs from *ddmin*, which can reduce to any qualified complement, because we believe that the confidence level of a neural network in its predictions is significant.

Increase granularity. If no complements qualify as the input for the next iteration step, we will increase the granularity level, similar to the approach in *ddmin*.

[1] We chose to set masked pixels to black because the MNIST dataset consists of black-and-white images. The digits are presented by white strokes, while black serves as the background, containing no information. Thus, it is reasonable to set our masked pixels to black.

Done. As in *ddmin*, if we have reached the highest granularity level, i.e., each subset contains exactly one pixel, it means we have reached the *1-minimal* state and have found the minimal image.

There is one thing worth nothing that in our implementation of *ddImage*, we exclude one case from *ddmin*, namely *Reduce to subset*. In *ddmin*, the goal is to quickly find the minimal failure test case by reducing the input as much as possible at each step, focusing solely on preserving the original failure without other constraints. However, in our scenario, we are dealing with images and considering both the predictions and corresponding confidence levels for each possible augmentation. Our objective is to mask out pixels while maintaining a correct prediction with a high confidence level.

Intuitively, masking out more pixels erases more information from the image, leading to a greater drop in prediction confidence. Given that $n \geqslant 2$ throughout the process, it's clear that $\forall i \in [1, n] \cdot |I - \Delta_i| \geqslant |\Delta_i|$. Therefore, unlike *ddmin*, we prioritize *Reduce to complement* over *Reduce to subset*. Also, because of this rationale, we find it is not very necessary to check the *Reduce to subset* case, and thus, we have omitted it from our *ddImage* algorithm.

4.2 Evaluations

4.2.1 Evaluation of applying Delta Debugging to images

For our experiment, we worked with a relatively small-sized FNN with four layers and the MNIST dataset [7]. Since we consider correct prediction as a condition to be checked, we first filtered out all the test images that could be predicted correctly by our neural network. We then applied our method to these correctly predicted test images individually, as each process is image-dependent, making batch processing impractical. Our methodology, detailed in the previous section, leaves one crucial part to be specified: the conditions for the iteration steps, which are essential for our experiments.

Initially, we set no additional conditions, meaning we continued the reduction process until the prediction of the augmented image no longer matched the ground truth.

However, this approach did not yield very meaningful observations. For most images, the resulting minimal images contained only a few pixels, making it difficult to discover any obvious patterns.

To address this, we added a constraint by setting a threshold for the prediction probability. This adjustment aimed to ensure that the augmented images maintained a high confidence level in their predictions, thereby producing more informative result minimal images for our analysis:

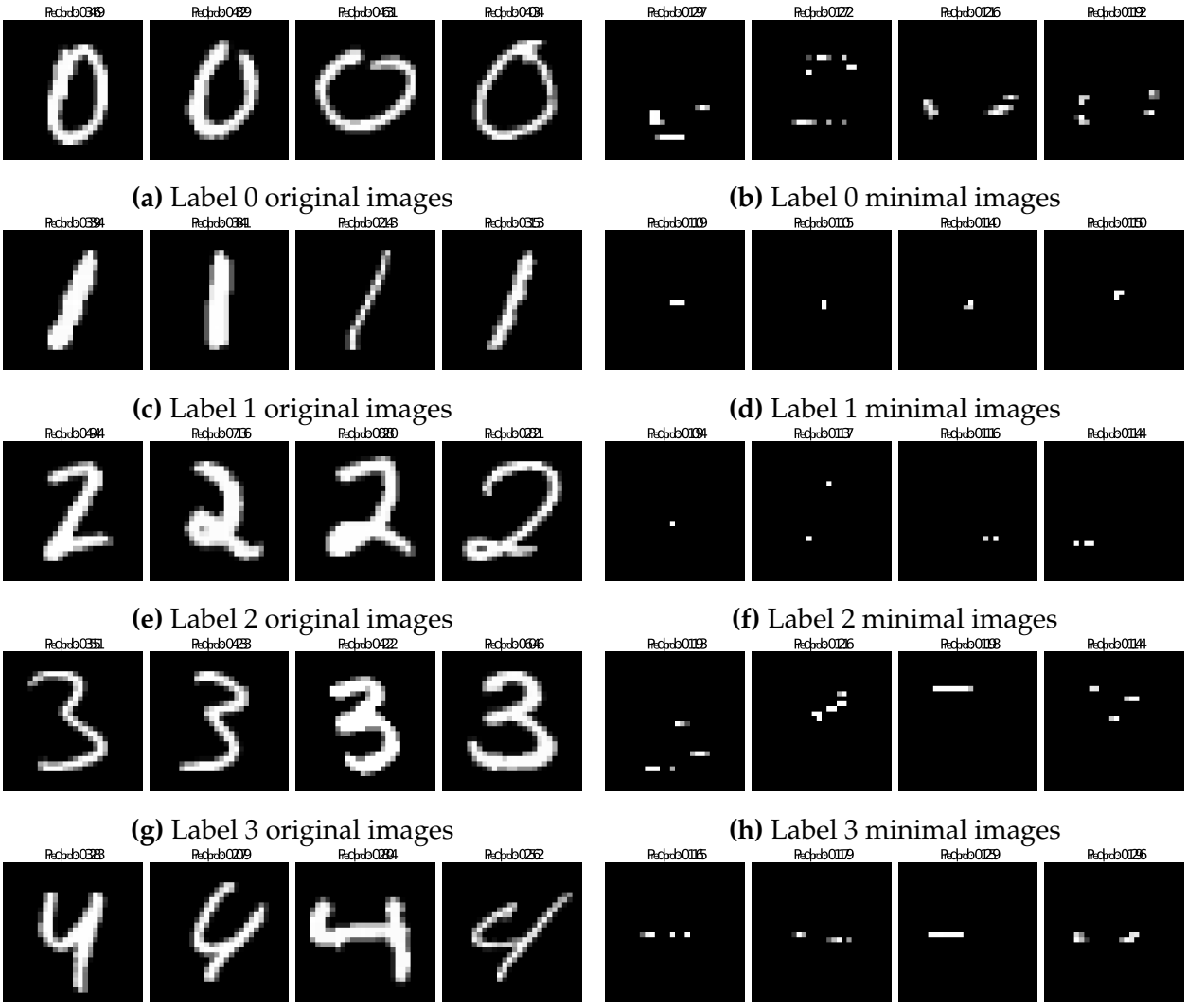
Condition case 1. $\exists \Delta_i$ such that $\mathcal{N}(I - \Delta_i) == l_I$ and $Prob(l_I | I - \Delta_i) \geqslant t$ where t is a given threshold and $t \in [0, 1]$.

In this scenario, if we set $t \leqslant 0.1$, it is effectively the same as having no additional conditions beyond ensuring the prediction remains correct throughout the entire process. This is because the sum of probabilities for all labels is 1. If the probability of the ground truth label is less than 0.1, another label must have a higher probability than the ground truth, resulting in an incorrect prediction and thus failing to satisfy the conditions.

Figures 4.1 and 4.2 display examples selected by running *Condition case 1* without setting a threshold t on probability, which is equivalent to setting $t \leqslant 0.1$. The images on the left are the original images, while the images on the right are the minimal images resulting from applying our *ddImage* method. As evident from the result minimal images, only a few pixels remain compared to their original counterparts.

For each label, we can observe some similarities between their minimal images. For instance, in Figure 4.1d, images labeled 1 tend to have only a dot in the middle, and images labeled 4 tend to have pixels left in the middle in the shape of a dotted line, as shown in Figure 4.1j. However, these observed patterns are not uniquely consistent across all classes. For example, images labeled 5 in Figure 4.2b also result in minimal images with dots, some of which are located in the middle of the image, similar to images labeled 1.

Consequently, it is challenging to identify a clear and distinctive pattern across these samples. Additionally, given that there are 10 classes, a random guess would result in a uniform prediction probability of 0.1. When the prediction probability is lower, the result is closer to a random guess. From this perspective, even if we observe some patterns, they



(a) Label 0 original images **(b)** Label 0 minimal images

(c) Label 1 original images **(d)** Label 1 minimal images

(e) Label 2 original images **(f)** Label 2 minimal images

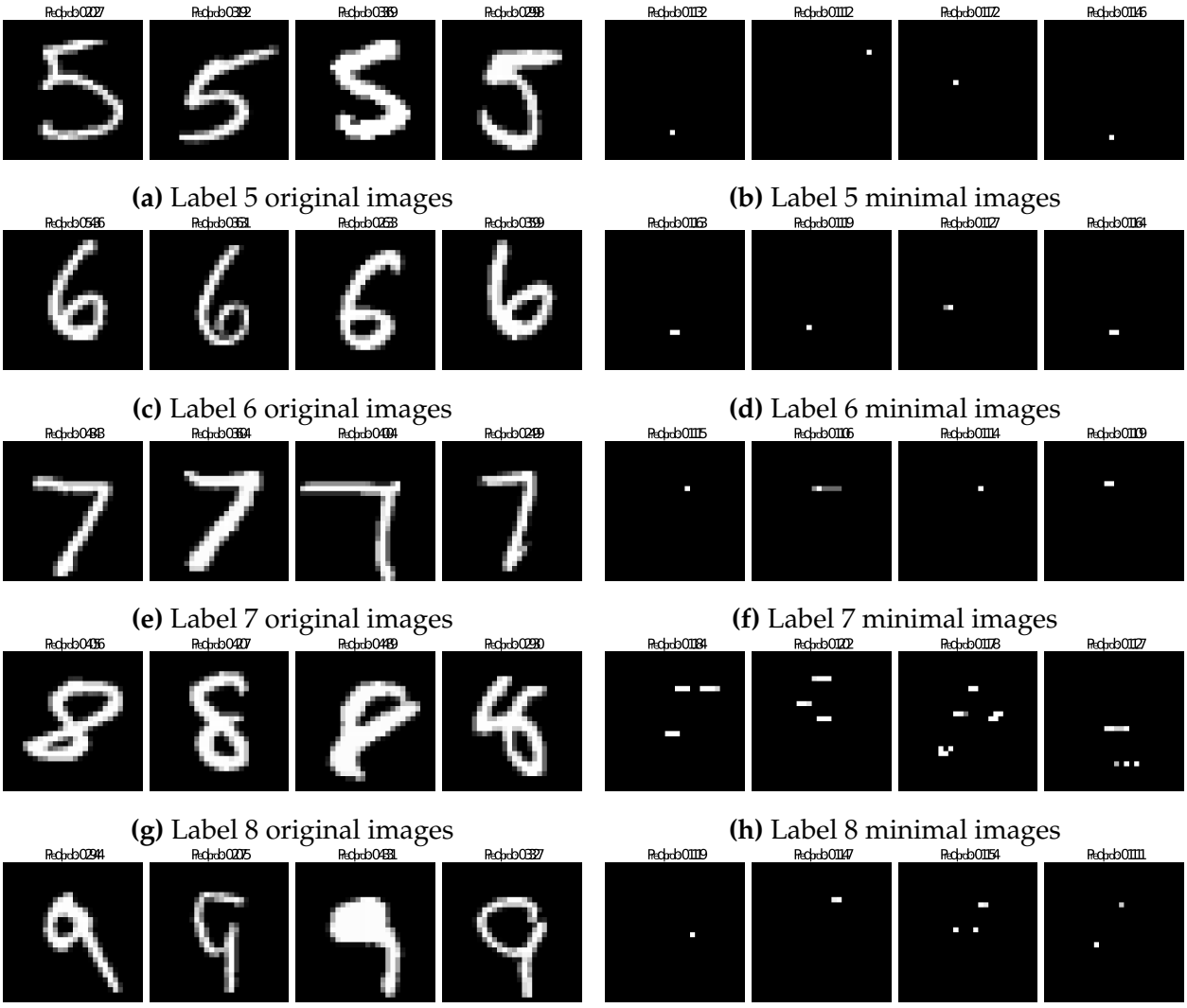
(g) Label 3 original images **(h)** Label 3 minimal images

(i) Label 4 original images **(j)** Label 4 minimal images

Figure 4.1: Some examples from classes 0 to 4 before and after applying *ddImage* without setting a threshold on prediction probability. The left columns display the original images, while the right columns show the corresponding minimal images after applying *ddImage*.

might not reflect the true intrinsic logic behind the decision-making process. Therefore, we believe it would be beneficial to increase the threshold probability to better observe interesting features.

We increased the threshold for the prediction probability and experimented with several different values, including 0.2, 0.3, and 0.4, as presented in Figure 3.4. Each group of three images shows the same input image processed by *ddImage* with thresholds $t \geqslant 0.2$,



(a) Label 5 original images **(b)** Label 5 minimal images

(c) Label 6 original images **(d)** Label 6 minimal images

(e) Label 7 original images **(f)** Label 7 minimal images

(g) Label 8 original images **(h)** Label 8 minimal images

(i) Label 9 original images **(j)** Label 9 minimal images

Figure 4.2: Some examples from classes 5 to 9 before and after applying *ddImage* without setting a threshold on prediction probability. The left columns display the original images, while the right columns show the corresponding minimal images after applying *ddImage*.

$\geqslant 0.3$, and $\geqslant 0.4$ from left to right. Each row includes four groups of images of the same class. From top to bottom, images are from class 0 to class 9, respectively. As the probability threshold increased, more pixels were preserved in the resulting images, which was expected. Unlike when we set the threshold to $\leqslant 0.1$, the digits are relatively more recognizable after reduction, and the higher the threshold, the more recognizable the resulting images are.

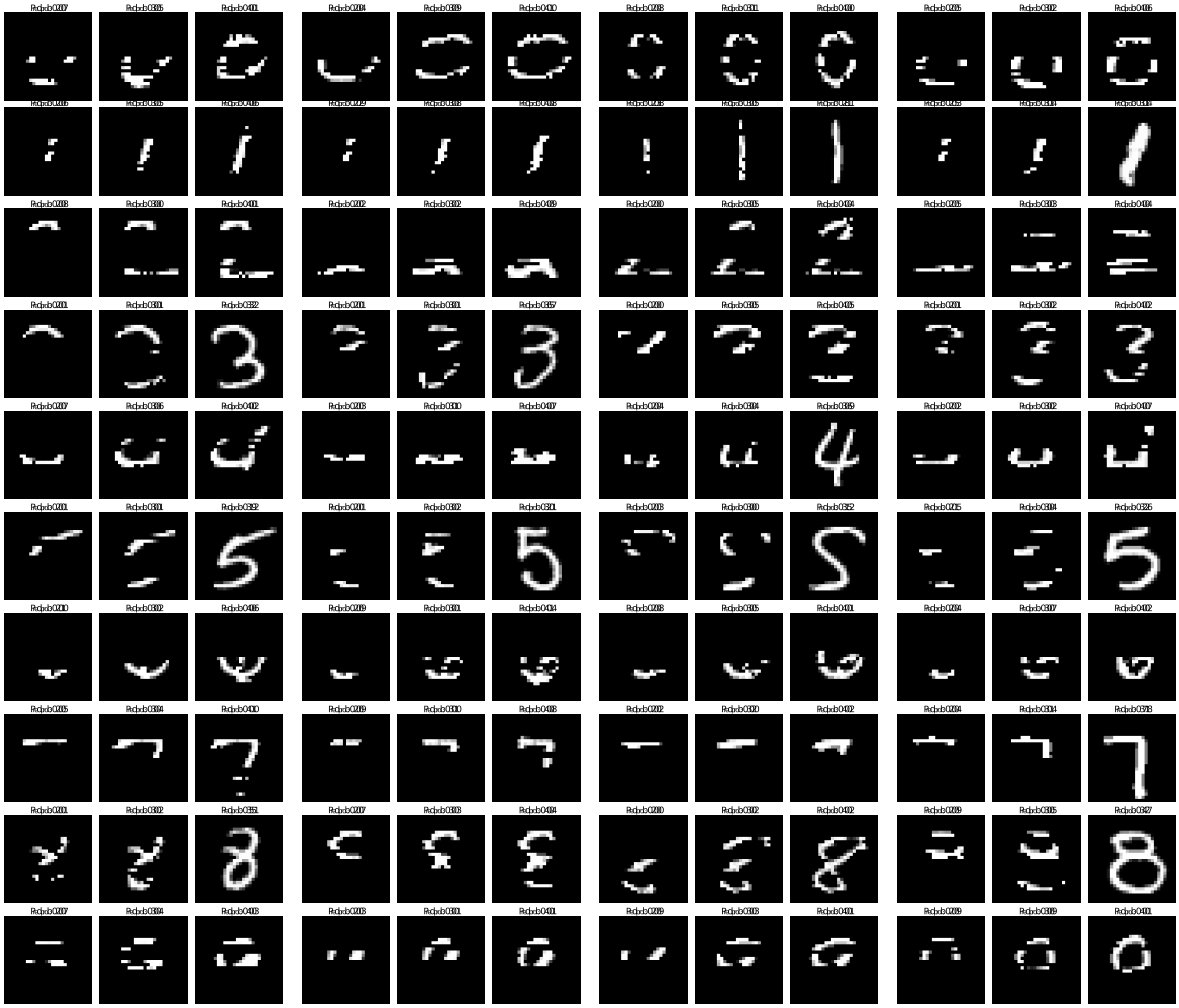


Figure 4.3: Some examples of results image under three different probability thresholds. For each group of three images, from left to right, they are under threshold $t \geqslant 0.2, \geqslant 0.3$ and $\geqslant 0.4$ as conditions. Each row contains images from one class. From top to bottom, images are from class 0 to class 9, respectively.

There are discernible patterns in the reduction process for each class. Certain areas tended to be reduced first, leaving the remaining pixels clustered in specific regions. For example, for the digit 0, the lower half of the hoop tends to be retained during reduction. However, these patterns are not always unique for each class. For instance, the shapes after the reduction process for classes 4 and 6 exhibit some similarities, with both retaining an upward curve as the remaining part.

In conclusion, our initial reduction process using probability as the threshold revealed intriguing patterns in the resulting minimal images. As we increased the prediction prob-

ability threshold, the results progressed from completely unrecognizable to relatively more identifiable. However, the patterns observed across different classes lacked uniqueness.

The masked-out pixels demonstrated that not all pixels are essential for the neural network's decision-making process, with some features being more important than others. This varying importance of features also exposes the neural network's vulnerability, as there exists a critical point where prediction fails.

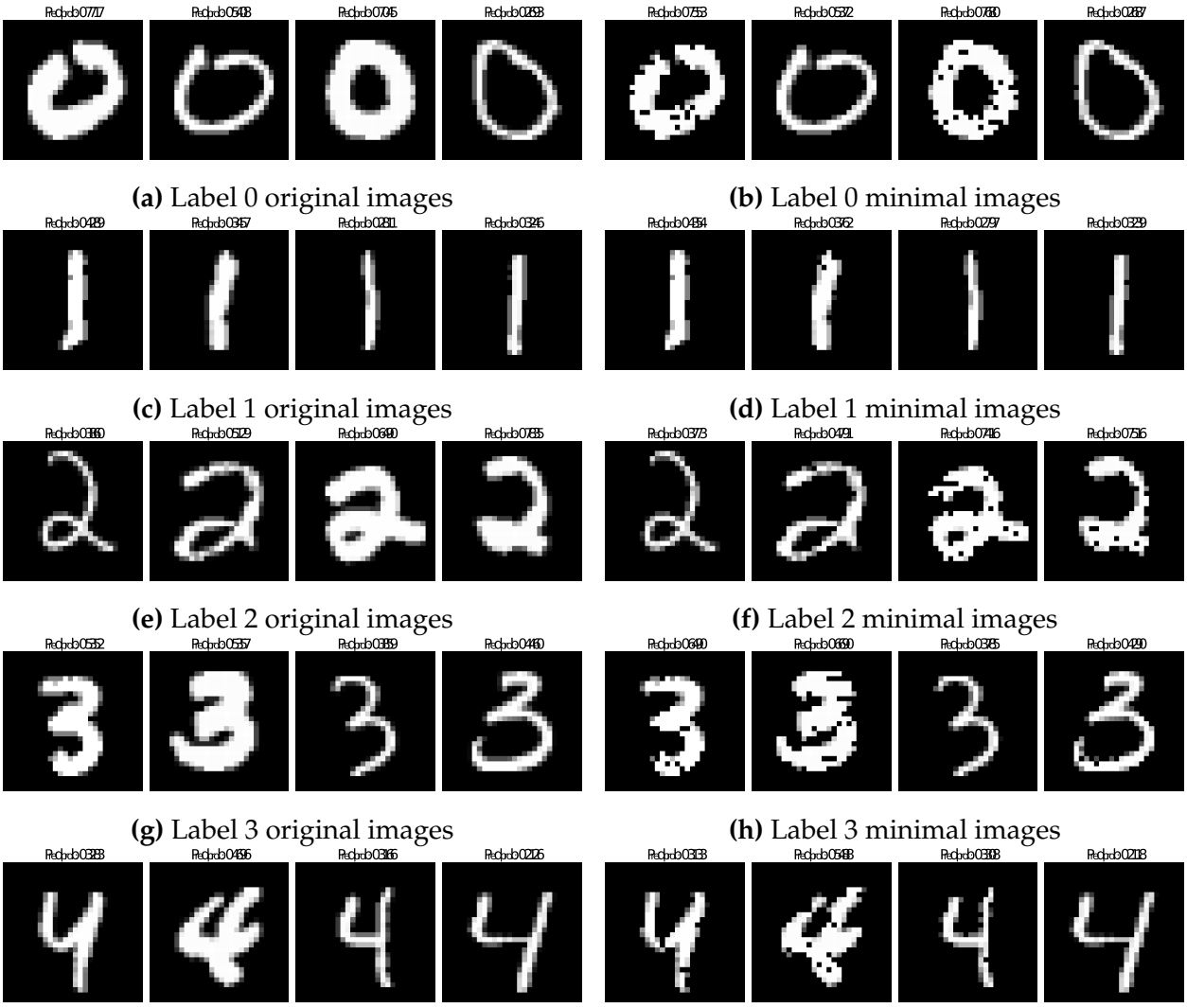
Moreover, the unrecognizable minimal images highlight the gap difference between human and machine perception, underscoring the challenges in neural network interpretability. These images also raise potential reliability concerns, suggesting that networks may classify seemingly unrelated inputs into a given class—a finding that deserves further investigation.

These insights led us to explore another conditioning criterion. As discussed in Chapter 3, our study on NAPs demonstrated their potential to abstract features of the neural network. We aim to incorporate NAPs into our conditions, using them instead of probability, to evaluate how well the neural network preserves image information during the pixel reduction process. Therefore, we set our condition as follows:

Condition case 2. $\exists \Delta_i$ such that $\mathcal{N}(I - \Delta_i) == l_I$ and $\mathcal{P}_I == \mathcal{P}_{I_o}$ where $\mathcal{P}_I$ is the NAP of I and $\mathcal{P}_{I_o}$ is the NAP of I_o, i.e., the NAP of the original image we input at the start of delta debugging process.

In this scenario, we first extract the NAP of the original input image I_o, denoted as $\mathcal{P}_{I_o}$. Given that our neural network is relatively small, we include all the neurons without incorporating the relaxation factor δ mentioned in Chapter 3.1. At each step, we extract the NAP of the augmented image, denoted as $\mathcal{P}_I$, and compare it with $\mathcal{P}_{I_o}$ until it reaches the *1-minimal* state, meaning that removing even one more pixel will cause it to no longer follow $\mathcal{P}_{I_o}$.

From the resulting minimal images, we observed that using NAP as the sole condition for the reduction process produces more ideal minimal images, as shown in Figure 4.4 and 4.5. The shape of digits and the information of the original input images are well-preserved under this condition case. For images with thinner strokes, fewer pixels are



(a) Label 0 original images **(b)** Label 0 minimal images

(c) Label 1 original images **(d)** Label 1 minimal images

(e) Label 2 original images **(f)** Label 2 minimal images

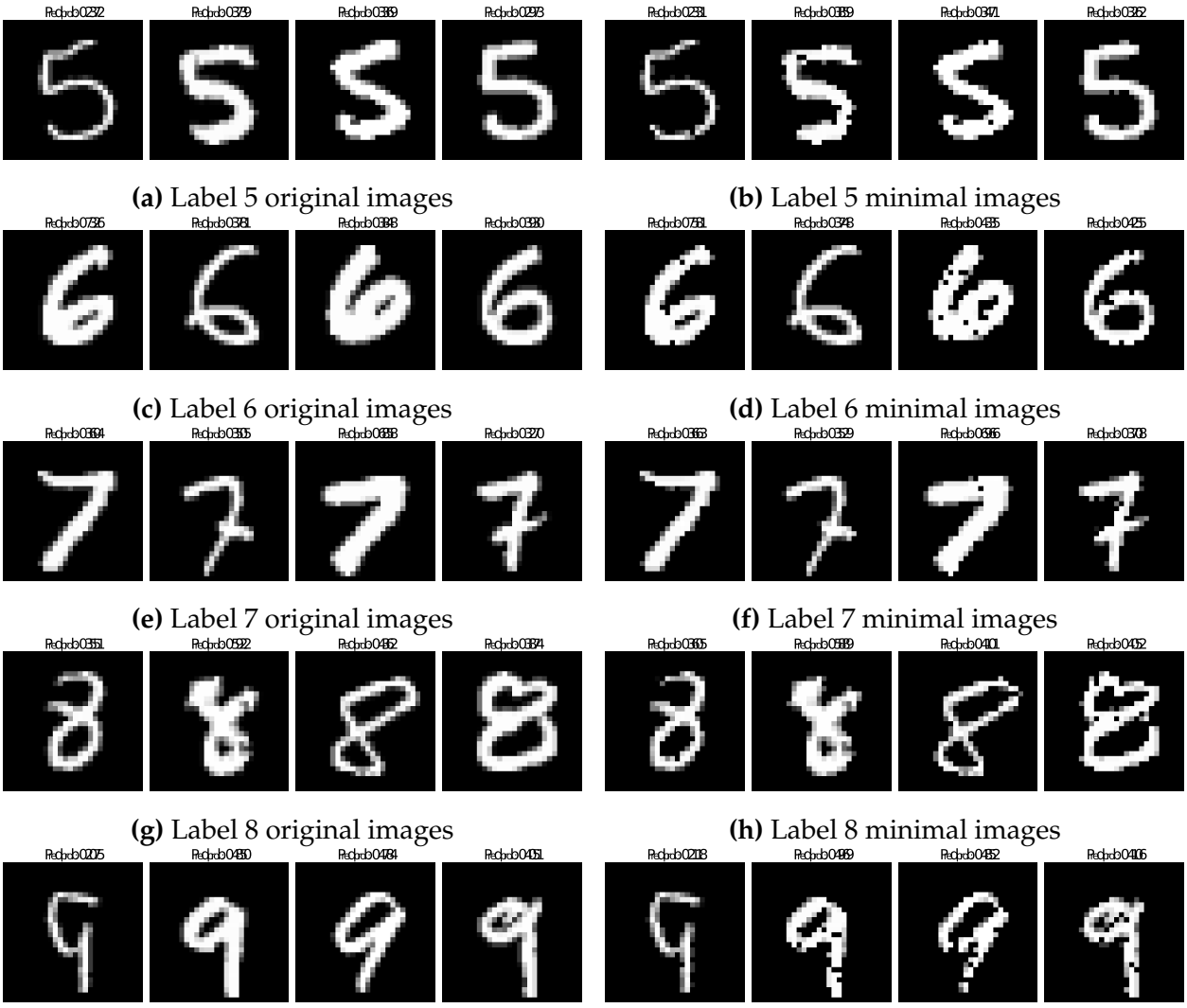
(g) Label 3 original images **(h)** Label 3 minimal images

(i) Label 4 original images **(j)** Label 4 minimal images

Figure 4.4: Some examples from classes 0 to 4 before and after applying *ddImage* with coorporating NAP in the condition.

masked out, leading to only minor changes compared to the original images. For those with bolder strokes, more pixels are removed from the strokes, creating small holes, but the majority of the stroke pixels are retained, making the digits easily recognizable and preserving the important information of the images.

In general, the strokes of the digits are preserved in a more continuous manner, maintaining most of the significant parts, whereas the images conditioned on prediction prob-



(a) Label 5 original images **(b)** Label 5 minimal images

(c) Label 6 original images **(d)** Label 6 minimal images

(e) Label 7 original images **(f)** Label 7 minimal images

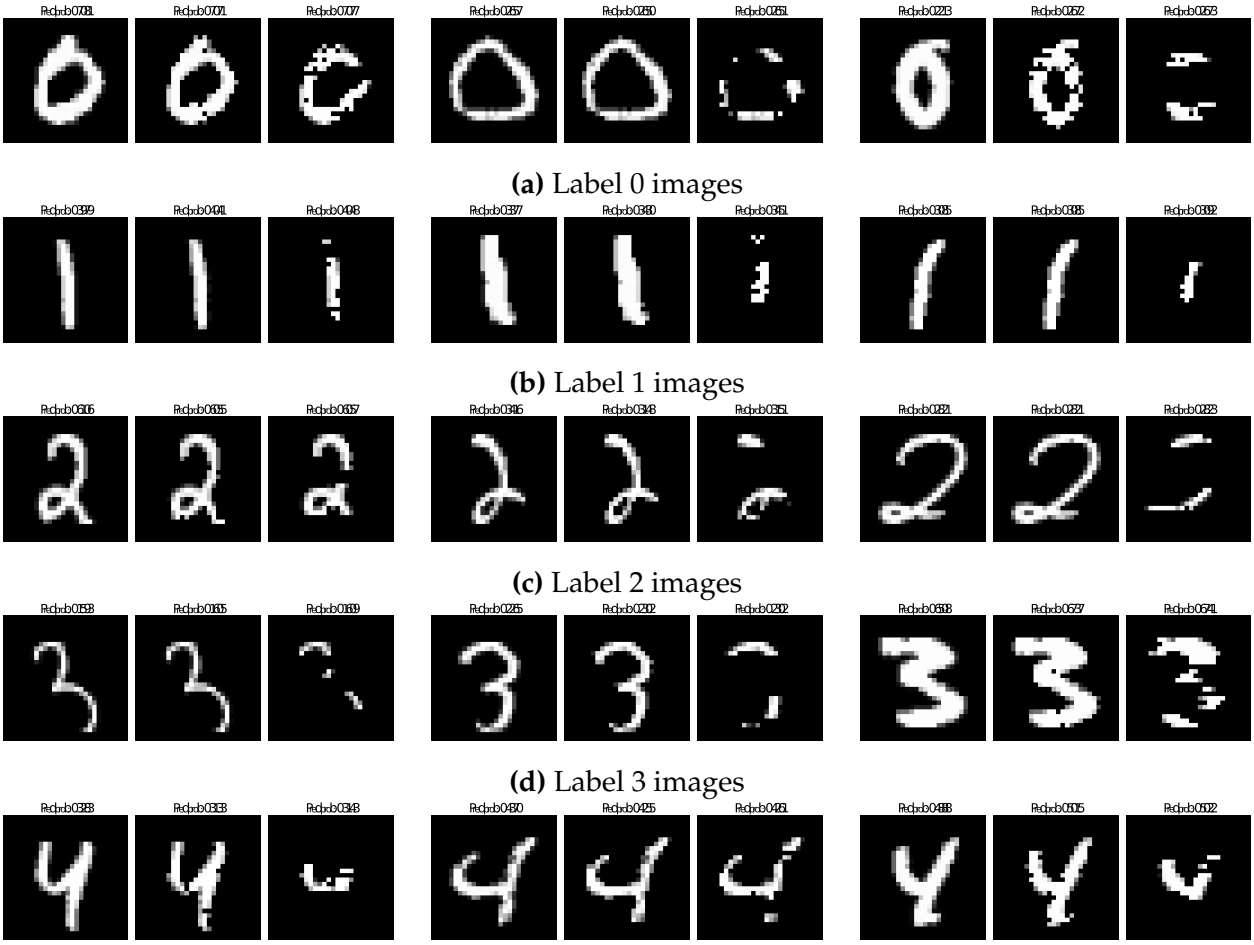
(g) Label 8 original images **(h)** Label 8 minimal images

(i) Label 9 original images **(j)** Label 9 minimal images

Figure 4.5: Some examples from classes 5 to 9 before and after applying *ddImage* with coorporating NAP in the condition.

ability have more segmented strokes. This demonstrates that NAP helps preserve important information in an image selectively and smartly.

To further verify this observation, we conducted an evaluation comparing the resulting images from these two conditions in a more obvious manner. For each image, we first performed the reduction process using NAP as the condition. We then used the prediction probability of the NAP-conditioned minimal image as the threshold for the reduction process conditioned on prediction probability, and compared the two resulting



(a) Label 0 images

(b) Label 1 images

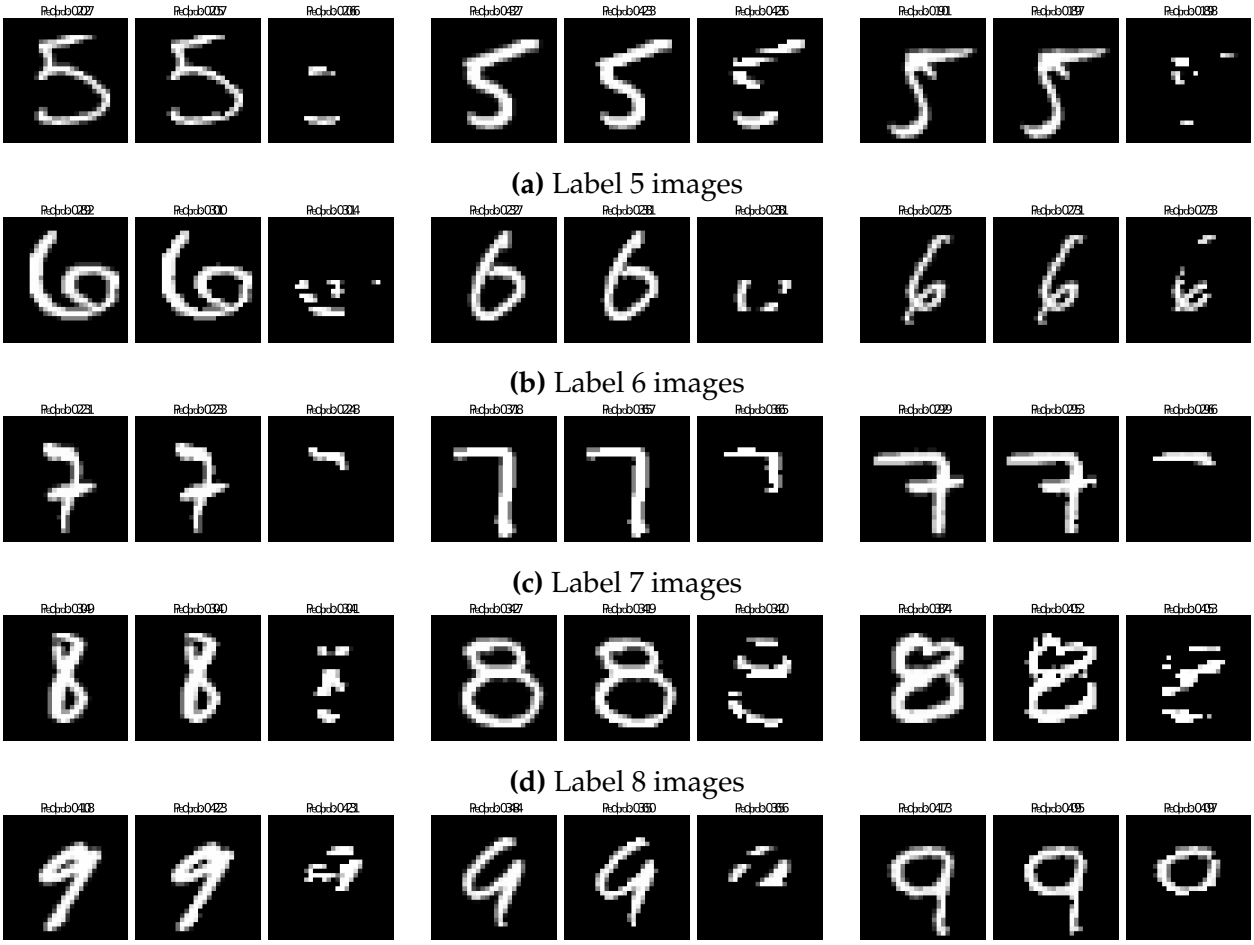
(c) Label 2 images

(d) Label 3 images

(e) Label 4 images

Figure 4.6: Some examples from classes 0 to 4 between NAP-conditioned and probability-conditioned result images of the same level of prediction probability. For each group of three images, the left one is the original image, middle one is NAP-conditioned, the right one is probability-conditioned.

images. The examples of this comparison are shown in Figure 4.6 and 4.7. As you can see, with the same level of prediction probability, the NAP-conditioned result minimal images could preserve more information and better shape of the digits. In some examples, the probability-conditioned result image has a significant part of the stroke masked out, such as in Figure 4.6b. Some images even become unrecognizable from a human perspective, such as in Figure 4.6d. In contrast, the NAP-conditioned result images of the same original images have the digit shape well-preserved.



(a) Label 5 images

(b) Label 6 images

(c) Label 7 images

(d) Label 8 images

(e) Label 9 images

Figure 4.7: Some examples from classes 5 to 9 between NAP-conditioned and probability-conditioned result images of the same level of prediction probability. For each group of three images, the left one is the original image, middle one is NAP-conditioned, the right one is probability-conditioned.

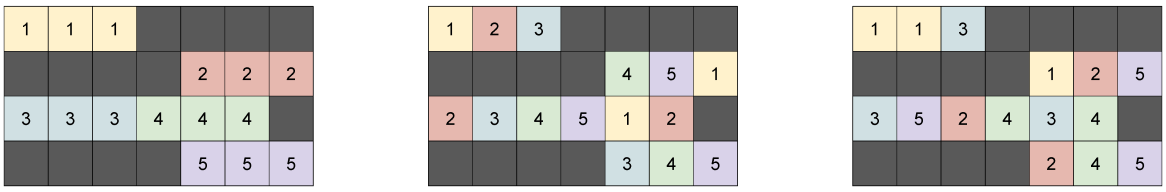
The evaluations above demonstrate that neural networks encode significant information in their NAPs, effectively preserving crucial information learned from the dataset during the reduction process. This suggests a better alignment between the network's internal representation and human perception, indicating that the NAP captures more semantically meaningful features. It also highlights that studying NAPs could be a more promising direction for neural network interpretability, as it is meaningful for both neural networks and humans. Considering our findings in Chapter 3, where we showed that NAPs could capture features from images, these results further support our point to

some extent. Additionally, the resilience of NAPs to significant image changes indicates that neural networks develop stable, high-level representations of input data.

4.2.2 Subsetting Techiniques for Masking

There is one minor part we have not yet explained in detail: the method for the step of splitting an input into subsets before the recursive reduction steps in Algorithm 3. We explored three approaches, as shown in Figure 4.8. These approaches are splitting into chunks, splitting sequentially, and splitting randomly. According to Algorithm 3, we need to split the unmasked pixels evenly into n subsets. Suppose, after calculation, each subset should contain k pixels. Then, the subsetting works as follows for the three methods:

Split into chunks. We will take k successive unmasked pixels from the array for each subset. As indicated in Figure 4.8a, the gray grid represents masked pixels, which are out of our consideration. The grids with numbers marked on them indicate the unmasked pixels and their corresponding group numbers, where pixels with the same group number belong to the same subset.



(a) Split into chunk **(b)** Split sequentially **(c)** Split randomly

Figure 4.8: Three methods for splitting image pixels into subsets are illustrated. Each grid represents a pixel in the image. Gray grids represent masked pixels, which are not considered for subsetting. Grids with colors and numbers indicate unmasked pixels, with the numbers representing the group number. Pixels with the same color and number are split into the same subset. For easier visualization, we show five subsets, each containing three pixels.

Split sequentially. In this method, pixels are assigned to group numbers 1 through n, and this assignment process is repeated. As indicated in Figure 4.8b, pixels with the same group number belong to the same subset.
Split randomly. For each subset, we randomly pick k unmasked pixels to form one subset. These k pixels are then considered selected and are out of consideration for the next subset-picking step, as indicated in Figure 4.8c.

Upon comparing the results of these three methods, we did not notice any significant differences. Each method displays its own pattern regarding the distribution of remaining unmasked pixels in the resulting minimal images. The *split into chunks* method tends to produce images where the remaining unmasked pixels are clustered together, forming larger dots or lines. In contrast, the results from the *split sequentially* and *split randomly* methods have more scattered distributions of unmasked pixels, making the digits relatively easier to recognize. However, when the probability threshold is set very low, such as 0, it remains difficult for humans to recognize the digits. Therefore, we believe that the choice of subsetting method does not significantly impact the experiments in the previous section. Consequently, we focused on the results of one subsetting method, choosing *split into chunks* for running Algorithm 3 and the evaluations.

We explored different methods because the form of the input can play an important role in the subsetting procedure, especially for the input type (e.g., programs) of the original application of the *Delta Debugging* algorithm, where the syntactic correctness of the result subset must be considered [21]. While image data is relatively straightforward to handle, as there are no syntactic or semantic issues, we still wanted to assess the extent to which the subsetting method would affect the evaluation results.

4.3 Discussion

In this chapter, we explored a case study applying a software engineering algorithm called *Delta Debugging* to perform image reduction and observe patterns that could potentially provide insights into the logic behind neural networks' decision-making processes. We adopted the *ddmin* method from [44] to develop our method, *ddImage*. Ini-

tially, we used prediction probability as the criterion for the reduction process. However, we found that relying solely on this criterion was insufficient to make significant observations. From a decision-making perspective, the resulting images revealed the varying significance of the features of input data in contributing to decision-making or prediction results. This variation also highlights the vulnerability and reliability issues of neural networks. The unrecognizable digits in the images after the reduction process indicate a different logic behind decision-making between humans and neural networks, as the patterns observed were not easily interpretable to humans. Given our previous study of NAPs, which demonstrated their potential to abstract learned features from the data, we incorporated NAPs into the reduction process criteria. As expected, relying solely on NAPs produced highly effective minimal images, preserving the digits in a well-defined shape, even from a human perspective. This not only shows that NAPs capture information for inference at a similar level to humans but also further validates the points discussed in Chapter 3.

There are two important points not addressed in [44] regarding the original Delta Debugging algorithm *ddmin*, which were proposed by [21]. These points include the order of examining subsets and complements, which might affect the results based on the use case, and the necessity of considering the format of input during subsetting. In our case, for efficiency purposes, we removed the step of examining the subsets. Regarding input format, our experiments on different subsetting methods revealed that, due to the nature of image data (which has no semantic or syntactic issues), there was no major effect on the results. However, for future work or for variants of Delta Debugging in other applications, these two points are definitely worth noting.

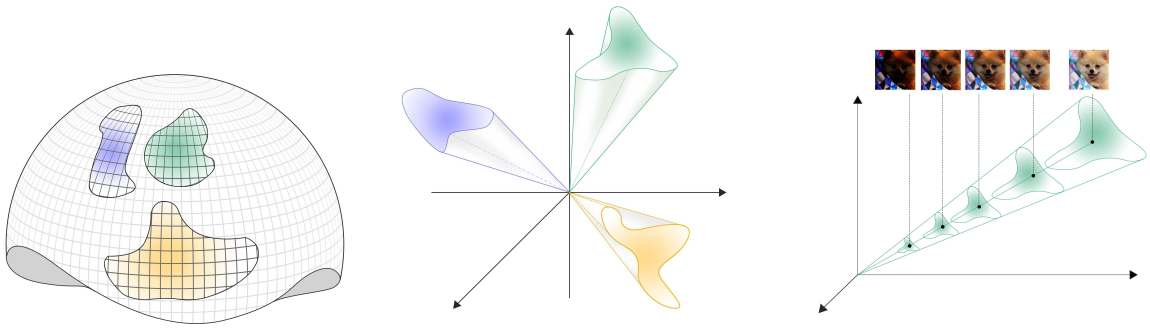
Chapter 5

Scalar Invariant Neural Network

In this chapter, rather than focusing on the existing properties of current models, we observed an intriguing property in the distribution of image data. Based on this observation, we modified the neural network to create a new property—scalar invariance. This new property is better suited for image classification tasks and has the potential to address real-world issues in applications involving image processing.

5.1 Property of the distribution of image domain

Image data is typically stored in computers using a pixel-based format. In most cases, each pixel in an image is represented by three numeric values, corresponding to the intensity of red, green, and blue required to produce the color of that pixel. These numeric values are usually represented as integers ranging from 0 to 255, or as floating-point values ranging from 0.0 to 1.0. When we flatten out the pixel matrix, it essentially forms a high-dimensional vector. Ignoring the clipping issue, modifying any point along this vector direction mainly alters the brightness or contrast of the image. While such modifications may alter the appearance of the image, they generally do not change its category or label from a human perspective. In other words, a dog remains recognizable as a dog even if there are significant changes in image brightness or contrast. Therefore, based on this observation, it suggests the distribution of image datasets incorporates directionality, as displayed in Figure 5.1.



(a) Image classes can be thought of as manifolds in n-dimensional input space.

(b) Image classes can be conceptualized as corn-shaped if considering the varying contrast.

(c) An image specifes a direction in input space, and copies of that image with varying contrast lie along the same direction.

Figure 5.1: The directionality (varying contrast) manifests in the intrinsic distribution of images.

This discover of the inherent directional nature of image data has made us to rethink the design of the neural network structure and start to question whether conventional neural network architecture adequately account for this special property of image data. As we look into the neural networks, the bias terms play a significant role in shaping the responses or making predictions for neural networks to input data. However, these bias terms may introduce biases or preferences that conflict with the directional characteristics inherent in image data. Hence, we create scalar invariant neural network by removing the bias terms and evaluate the accountability of this new-structured neural network.

5.2 Scalar invariant neural networks

5.2.1 Basic architect of convolutional neural network

A neural network consists of an input layer, hidden layers, and an output layer. For CNNs, commonly used for image-related tasks, a fundamental component is the convolutional layer, specifically designed for processing grid-like data. This layer plays a crucial role in extracting meaningful features from input images through a process called convolution.

In a convolutional layer, there are multiple filters or kernels. Each kernel slides through the input volume spatially, computing the dot product between each weight in the kernel and each value of the input column it is currently covering. Each dot product results in a single value. By sliding through the entire input column, a feature map is formed from these values. This feature map is then passed through an activation function, usually *ReLU*. Next, a pooling layer performs downsampling to reduce the spatial dimensions of the feature map.

After several convolutional and pooling layers, the feature maps are flattened into a 1D vector. This flattening process combines all the spatial information from the feature maps into a single vector, which serves as the input to the fully connected layers. The flattened vector is passed through one or more fully connected layers, where transformations on the input vector are performed using learnable parameters (weights and biases), ultimately producing an output vector. Finally, the output vector is passed through the output layer, which typically uses an appropriate activation function, such as *Softmax*, depending on the task.

To further investigate the scalar invariant property, we formally denote the input sample as X and a convolutional neural network as $\mathcal{N}$. Then $\mathcal{N}$ is composed of convolutional layers $\mathcal{F}_i$, pooling layers $\mathcal{P}_i$, and fully connected layers $\mathcal{L}_j$, where $i, j \in \mathbb{N}$. And we denote the final activation function as $\mathcal{A}$ and refer to *ReLU* as $\mathcal{R}$. Considering layers and activation functions as transformations on the input X, then the output of the network before the final activation function $\mathcal{A}$ is represented by:

$$\mathcal{O}(X) = \underbrace{\mathcal{L}_j \circ \mathcal{R} \circ ... \circ \mathcal{R} \circ \mathcal{L}_1}_{\text{j Linear layers}} \circ \underbrace{\mathcal{P}_i \circ \mathcal{R} \circ \mathcal{F}_i ... \circ \mathcal{P}_1 \circ \mathcal{R} \circ \mathcal{F}_1}_{\text{i Convolutional layers}} \circ X \tag{5.1}$$

The FNNs with simple architects could also use this representation, where FNNs could be considered as a special case with 0 convolutional layers or blocks.

And the final prediction class is determined by the one with the highest probability over all classes $\mathcal{C}$, that is:

$$\mathcal{N}(X) = \underset{c \in \mathcal{C}}{\operatorname{argmax}} \{\mathcal{A} \circ \mathcal{O}(X)\} \tag{5.2}$$

5.2.2 Scalar associative transformations

Now that we have a clear understanding of the structure of typical CNNs and FNNs, we aim to demonstrate that the associative property holds for each type of layer. This theoretical proof will establish the feasibility of creating a scalar invariant neural network. It's important to note that in our analysis, we will only consider positive scalar values. This decision is based on the nature of image data, as multiplying by negative scalars could easily lead to pixel values exceeding their range, resulting in colour or intensity inversions that significantly alter the image's appearance. Furthermore, negative scalars can change the sign of neurons, potentially causing issues in subsequent layers such as pooling and *ReLU* activation. Therefore, our focus will solely be on positive scalars for this analysis.

When examining the operation within a convolutional layer $\mathcal{F}$ using a kernel $\mathcal{K}$, it is evident that the associative property holds for convolution operations. To formalize this observation, consider a positive scalar denoted by s such that $s \in \mathbb{R}^+$. We can express this property as follows:

$$\begin{aligned} \mathcal{F} \circ (sX) =& \sum_m \sum_n sX(i+m, j+n)\mathcal{K}(m,n) \\ =& s \sum_m \sum_n X(i+m, j+n)\mathcal{K}(m,n) \\ =& s(\mathcal{F} \circ X) \end{aligned} \tag{5.3}$$

Next, we examine the pooling layer, where the above property also remains valid, for both max pooling and average pooling operations. This assertion stems from the fact that both the max and average pooling operations are expected to maintain scalar multiplication. It is similar for the ReLU function. Hence, we can conclude:

$$\begin{aligned} \mathcal{P} \circ (sX) &= s(\mathcal{P} \circ X) \\ \mathcal{R} \circ (sX) &= s(\mathcal{R} \circ X) \end{aligned} \tag{5.4}$$

Finally, passing the input X to a fully connected layer $\mathcal{L}$ can be thought of as applying a linear transformation $(\mathcal{W}, \mathcal{B})$ on X. If we set the bias term $\mathcal{B}$ to $\mathbf{0}$. We will have the scalar

associative property. That is:

$$\mathcal{L} \circ (sX) = (sX)\mathcal{W}^T = sX\mathcal{W}^T = s(\mathcal{L} \circ X) \tag{5.5}$$

It's worth noting that our proofs also rely on the commutative property, which typically holds for matrix and vector multiplications involving a scalar. Consequently, when biases are set to zero, we establish that the scalar multiplication associative property holds for the output function, specifically: $\mathcal{O}(sX) = s\mathcal{O}(X)$.

5.2.3 Scalar invariant convolutional neural networks

Having established the multiplication associativity of the preceding hidden layers, we now turn our attention to the output layers preceding the final predictions of the network $\mathcal{N}$. In classification tasks, the last activation function $\mathcal{A}$ is typically the *Softmax*. When we multiply the input X by a scalar s ($s \in \mathbb{R}^+$) and pass the product through the *Softmax* function, it is equivalent to adjusting the temperature of the distribution. It's important to note that despite the change in the shape of the distribution, the rank of candidate classes remains unchanged. In essence, the predicted class by the network $\mathcal{N}$ remains invariant under scalar multiplication.

$$\underset{c}{\operatorname{argmax}} \frac{e^{s\mathcal{O}(X)_c}}{\sum_{c \in \mathcal{C}} e^{s\mathcal{O}(X)_c}} = \underset{c}{\operatorname{argmax}} \frac{e^{\mathcal{O}(X)_c}}{\sum_{c \in \mathcal{C}} e^{\mathcal{O}(X)_c}} \tag{5.6}$$

Put together with the scalar associative property of the output function $\mathcal{O}(\cdot)$, we have a scalar invariant neural network:

$$\mathcal{N}(sX) = \underset{c}{\operatorname{argmax}} \{\mathcal{A} \circ \mathcal{O}(sX)\} = \underset{c}{\operatorname{argmax}} \{\mathcal{A} \circ \mathcal{O}(X)\} = \mathcal{N}(X) \tag{5.7}$$

The notion of scalar invariance in neural networks extends beyond convolutional neural networks. In essence, as long as the hidden layers execute scalar associative (and commutative) transformations, and the final activation function maintains the highest probable candidate under scalar multiplication, the neural network retains scalar invari-

ance. Given that an image input X represents a direction in the input space, and we have demonstrated that zero-bias neural networks can yield identical predictions along that direction, we may reinterpret this property as the directional robustness property.

Lemma 1 (Directional robustness property)**.** For any input X to a zero-bias neural network $\mathcal{N}$, the prediction remains the same when X is multiplied by any positive scalar s. Formally, we have $\mathcal{N}(sX) = \mathcal{N}(X) \ \ \forall s \in \mathbb{R}^+$.

5.2.4 Scalar invariant ResNet

We briefly discussed the most simple architecture of convolutional neural networks in the previous section. However, in addition to those basic layers we mentioned before, modern powerful CNNs also employ extra layers and techniques to address over-fitting and gradient exploding/vanishing issues. For example, ResNet [17] adopts *Dropout* [32], *Additive Skip Connection* [17] and *Batch Normalization* [19] which contributes enormously to its success. First, as dropout layers are disabled during the inference phase, it has no impact on the scalar invariant property. Second, it is trivial to show skip connection is also scalar multiplication associative if the corresponding residual branch $\mathcal{G}$ is also scalar multiplication associative.

$$sX + \mathcal{G}(sX) = s(X + \mathcal{G}(X)) \ \ \forall s \in \mathbb{R}^+ \tag{5.8}$$

Lastly, we consider Batch Normalization, which is performed through a normalization transformation that fixes the means and variances of inputs to each layer. Let us use $X_{\mathcal{B}}$ to denote a mini-batch of the entire training set. Then we have the batch normalization transformation as follows:

$$\mathcal{BN}(X_{\mathcal{B}}) = \gamma \hat{X}_{\mathcal{B}} + \beta \tag{5.9}$$

where γ and β are learnable parameters, and $\hat{X}_{\mathcal{B}}$ is the normalized input, represented by $\hat{X}_{\mathcal{B}} = \frac{X_{\mathcal{B}} - \mu_{\mathcal{B}}}{\sqrt{(\sigma_{\mathcal{B}})^2 + \epsilon}}$, ϵ is an arbitrarily small constant. Clearly, we observe that the scalar associative/invariant property doesn't hold for the normalization step, because:

$$\gamma(sX) + \beta = \gamma \frac{(sX) - \mu_{\mathcal{B}}}{\sqrt{(\sigma_{\mathcal{B}})^2 + \epsilon}} + \beta \neq s(\gamma X + \beta) \tag{5.10}$$

Thus, in order to achieve scalar invariance, we can adopt two approaches. Firstly, for small neural networks that do not have severe gradient explosion/vanishing issues, we can drop $\mathcal{BN}$ layers. Secondly, for larger neural networks, we can consider some alternatives to batch normalization. There exists a line of work on exploring efficient residual learning without normalization such as Instance Normalization [36], Fixup [45], $\mathcal{X}$-DNNs [18], and NFNets [3,4]. We chose the Fixup network as an alternative option to construct our networks to achieve scale invariance.

We have discussed the modifications made by Zhang et al. [45] to the original ResNet to get a Fixup ResNet in Chapter 2.4. To construct a scale-invariant Fixup network, we need to remove the bias terms. Therefore, our modification is applied solely to rule 3. After our adjustments, the three rules will be:

1. Initialize the classification layer and the last layer of each residual branch to 0.

2. Initialize every other layer using a standard method (e.g., the methods used by He et al. [17]), and scale only the weight layers inside residual branches by $L^{-\frac{1}{2m-2}}$, where L is the number of residual branches the model has, and m is number of layers inside a single residual branch.

3. *Add a scalar multiplier (initialized at 1) in every branch.*

The first and second rules exclusively impact the training phase without altering the structure of the ResNet model, thereby not influencing the scalar associative property of the Fixup-ResNet model. Conversely, the third rule describes the modifications introduced by the Fixup method to the model. Referring to Figure 2.1 for a visualization of these changes, the method replaces batch normalization layers with multiplier and bias layers. To construct the corresponding zero-bias version of the model, we simply omit the bias layers, as depicted by the middle structure in Figure 2.1. Notably, the multiplier layer maintains scalar associativity, thereby rendering the zero-bias Fixup-ResNet a scalar invariant network.

5.3 Evaluations

In preceding sections, we have demonstrated theoretically that by eliminating bias terms and adopting alternative structures for specific layers unable to achieve scalar invariance, we can construct scalar invariant zero-bias neural networks. In this section, we transition from theory to experimentation, validating the scalar invariance and comparing the performance of these models with their with-bias counterparts.

In our experimental setup, we opted for four neural network architectures: a FNN, a CNN, as well as ResNet18 and ResNet50. Notably, for the ResNet models chosen, we applied alternative techniques Fixup as we mentioned in the previous part to ensure scalar invariance in their zero-bias variants. Across each neural network type, we conducted training sessions for both with-bias and zero-bias models under the exactly the same configuration. These experiments were conducted across several image classification benchmarks, including MNIST [7], Fashion-MNIST [42], CIFAR-100 [22], and ImageNet [6].

Using the trained models, we selected a range of positive scalars from 1 to 0.0001 and augmented the dataset with these scalars to adjust the contrast of the images. Subsequently, we evaluated the performance of both with-bias and zero-bias models for each neural network selected and presented a comparative analysis of their performance under these varied positive scalars in Table 5.1.

Table 5.1: As expected, zero-bias neural networks achieve perfect scalar invariance on testing accuracies, while normal neural networks are generally not robust against decreasing the contrast of the input image. Results are replicated thrice and averaged to reduce stochasticity effects, with all variances being below 0.5.

			Scalar multiplier										
			1	0.25	0.15	0.125	0.1	0.075	0.05	0.025	0.01	0.001	0.0001
MNIST	FCN	w/ bias	88.12	87.07	84.46	82.57	79.52	74.76	65.82	42.84	16.34	10.28	10.28
		w/o bias	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**	**88.27**
Fashion-MNIST	CNN	w/ bias	**89.10**	67.10	40.12	32.52	24.16	17.91	12.46	10.12	10.00	10.00	10.00
		w/o bias	89.02	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**	**89.02**
CIFAR-100	ResNet18	w/ bias	**67.62**	19.86	8.20	6.11	4.16	2.58	1.69	1.06	1.01	1.01	1.01
		w/o bias	67.33	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**	**67.33**
ImageNet [6]	ResNet50	w/ bias	**75.37**	66.72	57.84	53.62	47.27	37.61	21.81	3.39	0.21	0.10	0.10
		w/o bias	73.82	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**	**73.82**

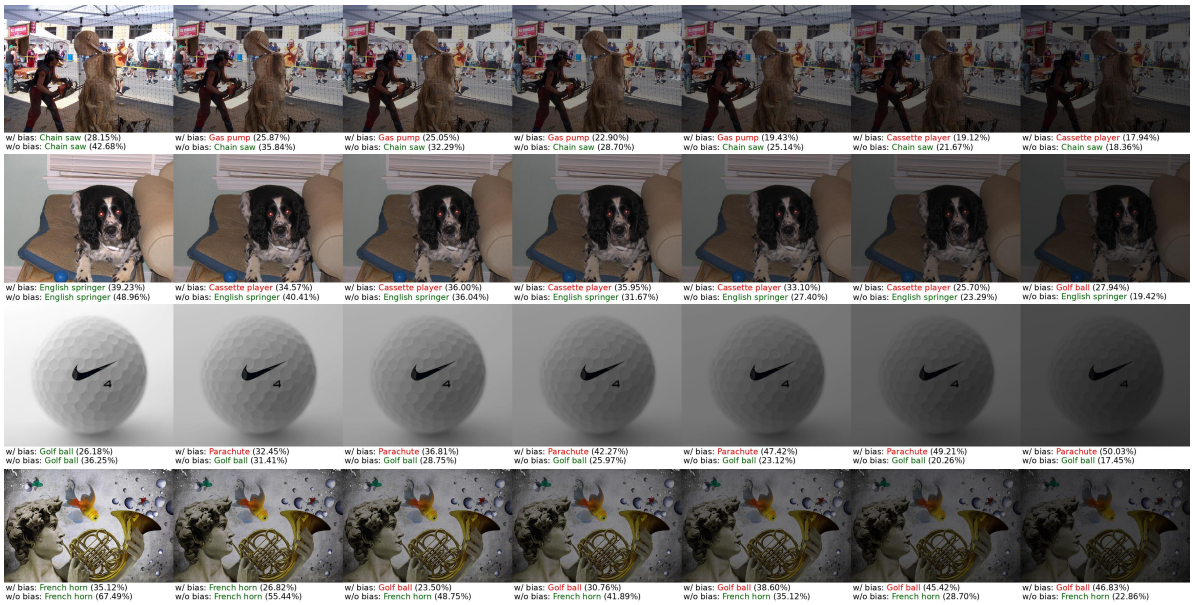
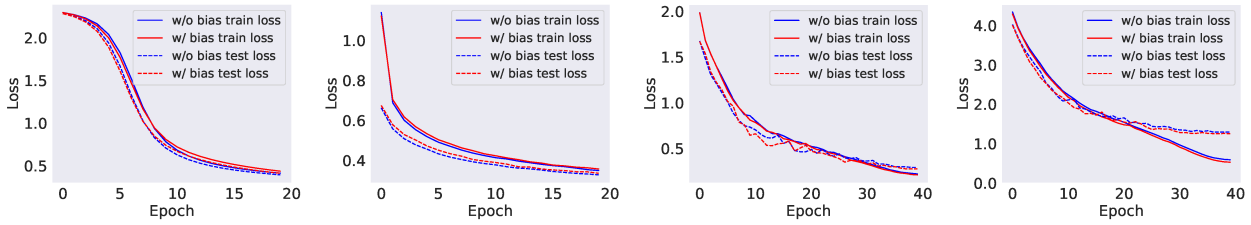


Figure 5.2: *W/ bias* and *w/o bias* stand for the prediction of normal and scalar invariant models respectively. Prediction of normally trained neural networks changes constantly as the scalar decreases, whereas that of scalar invariant networks remains unchanged. Despite the probability of the corresponding class diminishing. models inherit scalar invariance from removing bias.

Upon observation of the table, it is evident that even after bias removal, the zero-bias models exhibit comparable performance to their with-bias counterparts on the original test set (i.e., when the scalar is 1). While some networks may show slightly superior performance with with-bias models, the differences in accuracies between the two are not substantial, indicating the potential utility of zero-bias models for image classification tasks. However, as the scalar of the input image decreases from 1 to 0.0001, with-bias models demonstrate a lack of robustness, as evidenced by varying declines in their accuracies. In contrast, zero-bias models demonstrate scalar invariance, as anticipated from the proofs presented in previous sections, maintaining consistent performance irrespective of the varying contrast of input images.

In addition to the tabulated results, we provide visualizations of images with varying contrast and a comparison of predictions made by the normal with-bias model and the zero-bias model in Figure 5.2. In each row of images, the left-most image represents the original image, while subsequent images depict decreasing contrast levels from left

to right. It is noteworthy that as the contrast changes, the zero-bias model maintains consistent performance, while the predictions of the with-bias model become increasingly inaccurate. For instance, consider the image of the golf ball: both models correctly predict it in its original form, but as the contrast decreases, the with-bias model incorrectly predicts it as a parachute—indicating a departure from its initial classification. Even for images with significantly reduced contrast (i.e., those on the right-most side of each row), which remain clearly recognizable as golf balls from a human perspective, the with-bias model tends to misclassify them as parachutes.



(a) Loss curves of FCNs on MNIST.

(b) Loss curves of CNNs on Fashion-MNIST.

(c) Loss curves of ResNet9 on CIFAR-10.

(d) Loss curves of ResNet18 on CIFAR-100.

Figure 5.3: Loss curves of normally trained neural networks, and their scalar invariant counterparts are almost identical, which supports our argument that removing bias doesn't impact the training dynamics and generalization of models on image classification tasks.

Apart from comparing the testing performance and accuracies, we also examined the training progress of relatively simple networks on simple datasets, specifically MNIST and CIFAR10. The training loss curves are displayed in Figure 5.3. The red lines represent the loss curves of the models with biases, and the blue lines represent the loss curves of the zero-bias models. As shown, the curves highly overlap, indicating that the training dynamics and generalization of models on image classification tasks are very similar between the with-bias and zero-bias models.

5.4 Interesting robustness property

In the previous sections, we explored the theoretical bases of achieving scalar invariance by removing bias terms from the neural network while employing alternative solutions for batch normalization layers. Furthermore, we substantiated this theoretical analysis with empirical evidence, showcasing that the zero-bias network is not just a feasible concept but can also match the performance of normal with-bias neural networks. However, neural networks have been shown to lack robustness against small perturbations [1, 5]. Therefore, in this section, we will look into intriguing robustness properties observed in the zero-bias network that are not typically present in normal with-bias neural networks.

In Chapter 3, we explored NAPs and used them to observe intriguing properties of neural networks. In this chapter, we revisit NAPs due to the discovery of an interesting robustness property during our study of zero-bias neural networks' NAPs. The NAP we refer to here is defined identically to what was discussed in the previous chapter, i.e., Definition 3.1.1. Leveraging the multiplication associative property established in the preceding section, we uncovered that when mixing two images following the same NAP, the resulting mixed image should yield the same prediction as the two reference images.

Theorem 2 (Interpolation robustness property)**.** For any two inputs X_1 and X_2 that have the same prediction and NAP by network $\mathcal{N}$, i.e., $\mathcal{N}(X_1) = \mathcal{N}(X_2)$ and $\mathcal{P}_{X_1} = \mathcal{P}_{X_2}$, their linear interpolation also yield the same prediction, that is, $\mathcal{N}(\alpha X_1 + (1-\alpha)X_2) = \mathcal{N}(X_1) = \mathcal{N}(X_2)$, where $\alpha \in [0, 1]$.

Given that two samples share the same prediction and NAP, it follows that their interpolation also shares the same pattern. Our proof is provided below:

Proof. We show the interpolation robustness property holds for FNNs without bias. For more complicated neural networks such as CNN, the property also holds as long as all transformations before the output layer are scalar associative (**Lemma** 1). Consider a FCN $\mathcal{N}$ composed of J number of fully connected layers $\mathcal{L}_j$ and some *ReLU* layers $\mathcal{R}$. We think of layers and activation functions as transformations on the input X, then the output of

the network $\mathcal{O}(\lambda X_1 + (1-\lambda)X_2)$ before the *softamx* function is represented by:

$$\mathcal{O}(\lambda X_1 + (1-\lambda)X_2) = \mathcal{L}_J \circ \mathcal{R} \circ ... \circ \mathcal{R} \circ \mathcal{L}_1 \circ (\lambda X_1 + (1-\lambda)X_2) \tag{5.11}$$

For any fully connected layer $\mathcal{L}_j$, we have:

$$\begin{aligned} \mathcal{L}_j \circ (\lambda X_1 + (1-\lambda)X_2) &= (\lambda X_1 + (1-\lambda)X_2)\mathcal{W}_j^T \\ &= \lambda X W_j^T + (1-\lambda)X_2\mathcal{W}_j^T \\ &= \lambda \mathcal{L}_j \circ X_1 + (1-\lambda)\mathcal{L}_j \circ X_2 \end{aligned} \tag{5.12}$$

On the other hand, we have X and Y falling into the same NAP. Since the linear region corresponding to the NAP is convex, the interpolation of X_1 and X_2, i.e., $\lambda X_1 + (1-\lambda)X_2$, also lies in the same NAP. Furthermore, we have:

$$\begin{gathered} \mathcal{R} \circ \mathcal{L}_1 \circ (\lambda X_1 + (1-\lambda)X_2) = \lambda \mathcal{R} \circ \mathcal{L}_1 \circ X_1 + (1-\lambda)\mathcal{R} \circ \mathcal{L}_1 \circ X_2 \\ \mathcal{R} \circ \mathcal{L}_2 \circ (\lambda \mathcal{R} \circ \mathcal{L}_1 \circ X_1 + (1-\lambda)\mathcal{R} \circ \mathcal{L}_1 \circ X_2) = \lambda \mathcal{R} \circ \mathcal{L}_2 \circ \mathcal{R} \circ \mathcal{L}_1 \circ X + (1-\lambda)\mathcal{R} \circ \mathcal{L}_2 \circ \mathcal{R} \circ \mathcal{L}_1 \circ X_2 \\ ... \\ \mathcal{O}(\lambda X_1 + (1-\lambda)X_2) = \lambda \mathcal{O}(X_1) + (1-\lambda)\mathcal{O}(X_2), \text{ by } \textbf{Lemma } 1 \end{gathered} \tag{5.13}$$

Given that $\mathcal{N}(X_1) = \mathcal{N}(X_2)$, the index/class of the highest logit of $\mathcal{O}(X_1)$ and $\mathcal{O}(X_2)$ must be the same, that is:

$$\underset{c}{\operatorname{argmax}}\, \mathcal{O}(X_1)_c = \underset{c}{\operatorname{argmax}}\, \mathcal{O}(X_2)_c \tag{5.14}$$

Since multiplying a positive scalar to the operand won't change the output of the argmax operator, we have:

$$\underset{c}{\operatorname{argmax}}\, \lambda \mathcal{O}(X_1)_c = \underset{c}{\operatorname{argmax}} (1-\lambda)\mathcal{O}(X_2)_c = \underset{c}{\operatorname{argmax}}\, \mathcal{O}(X_1)_c = \underset{c}{\operatorname{argmax}}\, \mathcal{O}(X_2)_c \tag{5.15}$$

Note that the index/class of the highest logit of $\lambda \mathcal{O}(X_1)_c$ and $(1-\lambda)\mathcal{O}(X_2)_c$ are the same, the index/class of the highest logit of their addition is also the same as $\lambda \mathcal{O}(X_1)_c$

and $(1-\lambda)\mathcal{O}(X_2)_c$. Then it follows that:

$$\underset{c}{\operatorname{argmax}}\,\mathcal{O}(\lambda X_1 + (1-\lambda)X_2)_c = \underset{c}{\operatorname{argmax}}\,\mathcal{O}(X_1)_c = \underset{c}{\operatorname{argmax}}\,\mathcal{O}(X_2)_c \tag{5.16}$$

Since the *softmax* function will preserve the ranking of logits, we have:

$$\underset{c}{\operatorname{argmax}}\,\frac{e^{\mathcal{O}(\lambda X_1+(1-\lambda)X_2)_c}}{\sum_{c\in\mathcal{C}} e^{\mathcal{O}(\lambda X_1+(1-\lambda)X_2)_c}} = \underset{c}{\operatorname{argmax}}\,\frac{e^{\mathcal{O}(X_1)_c}}{\sum_{c\in\mathcal{C}} e^{\mathcal{O}(X_1)_c}} = \underset{c}{\operatorname{argmax}}\,\frac{e^{\mathcal{O}(X_2)_c}}{\sum_{c\in\mathcal{C}} e^{\mathcal{O}(X_2)_c}} \tag{5.17}$$

Finally, this can be restated as:

$$\mathcal{N}(\lambda X + (1-\lambda)X_2) = \mathcal{N}(X_1) = \mathcal{N}(X_2) \tag{5.18}$$

□

This property can be extended to the multiple inputs setting, where a convex region can provide robustness assurance.

Theorem 3 (Convex region robustness property)**.** Let $\{X_i \mid i \in \{1, 2, \ldots, n\}\}$ be a collection of inputs that have the same prediction and NAP by network $\mathcal{N}$, we denote the convex polygon formed by vertices X_i as $\mathcal{M}$. Then, for any point $\boldsymbol{m}$ that lies inside the polygon $\mathcal{M}$, $\boldsymbol{m}$ also yield the same prediction as X_i, that is, $\mathcal{N}(\boldsymbol{m}) = \mathcal{N}(X_i)\ \forall \boldsymbol{m} \in \mathcal{M}\ \forall i \in \{1, 2, ..., n\}$.

Since $\boldsymbol{m}$ represents a convex combination of vertices X_i, this convex region robustness property automatically holds as per Theorem 2. Notably, due to the absence of bias terms in our zero-bias network, it contrasts with normal with-bias neural networks that exhibit a higher combinatorial nature. In these networks, the activation status of neurons after *ReLU* activation layer corresponds to hyperplanes in the input space, determined by weights and bias terms for neuron calculation. With bias terms removed, all hyperplanes pass through the origin, reducing the number of linear regions. Consequently, achieving this robustness property is more challenging in normal with-bias neural networks. Recent research, such as [9, 10], further underscores that ignoring bias terms can enhance model robustness.

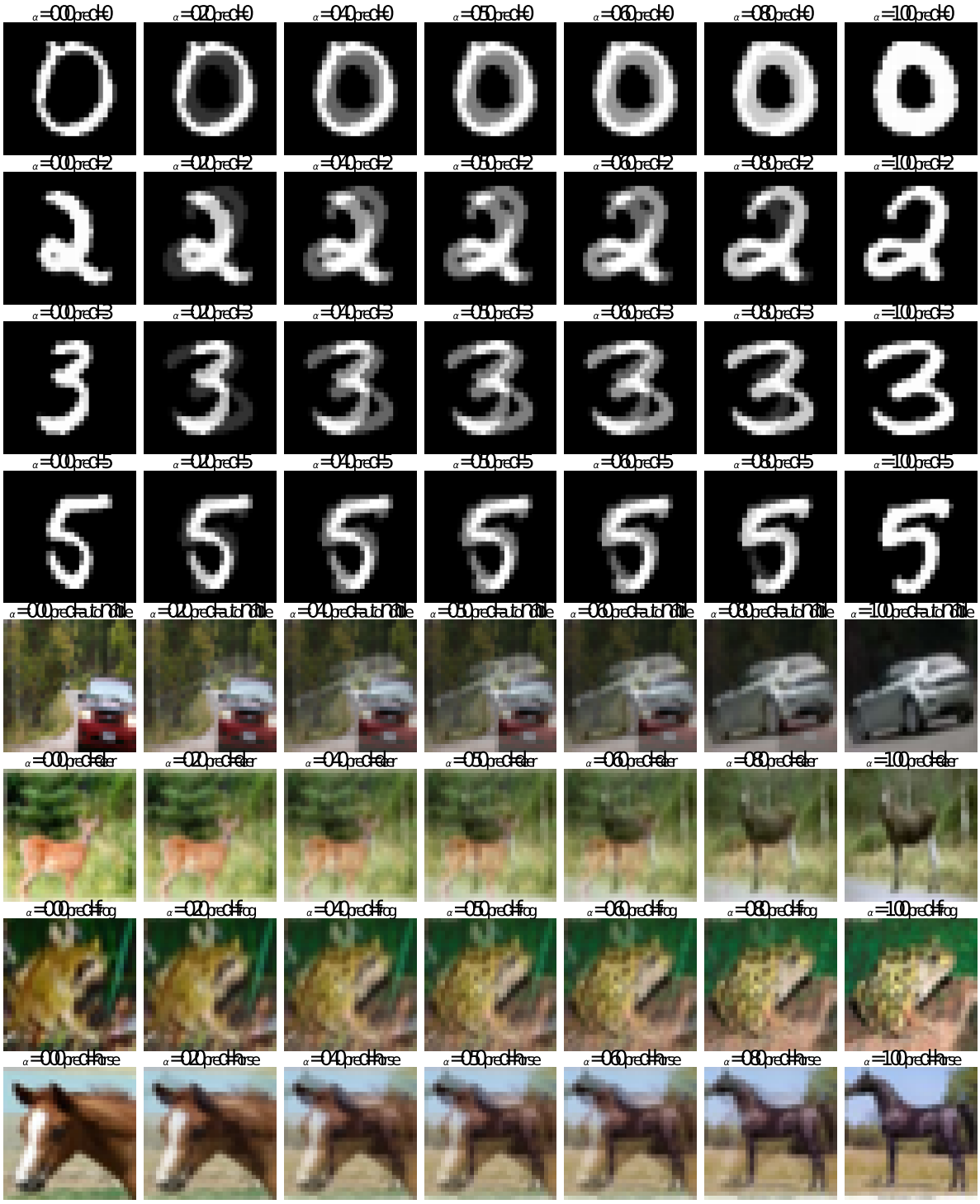


Figure 5.4: The left-most and right-most images are from the original MNIST (first five rows) and CIFAR10 (the last five rows) dataset, whereas synthesized/interpolated images are in the middle. For instance, the middle image in the first row is generated by adding $(\alpha = 0.5)$ times the left image to $(1 - \alpha)$ times the right image. The interpolated images have the predictions same as the ground truth.

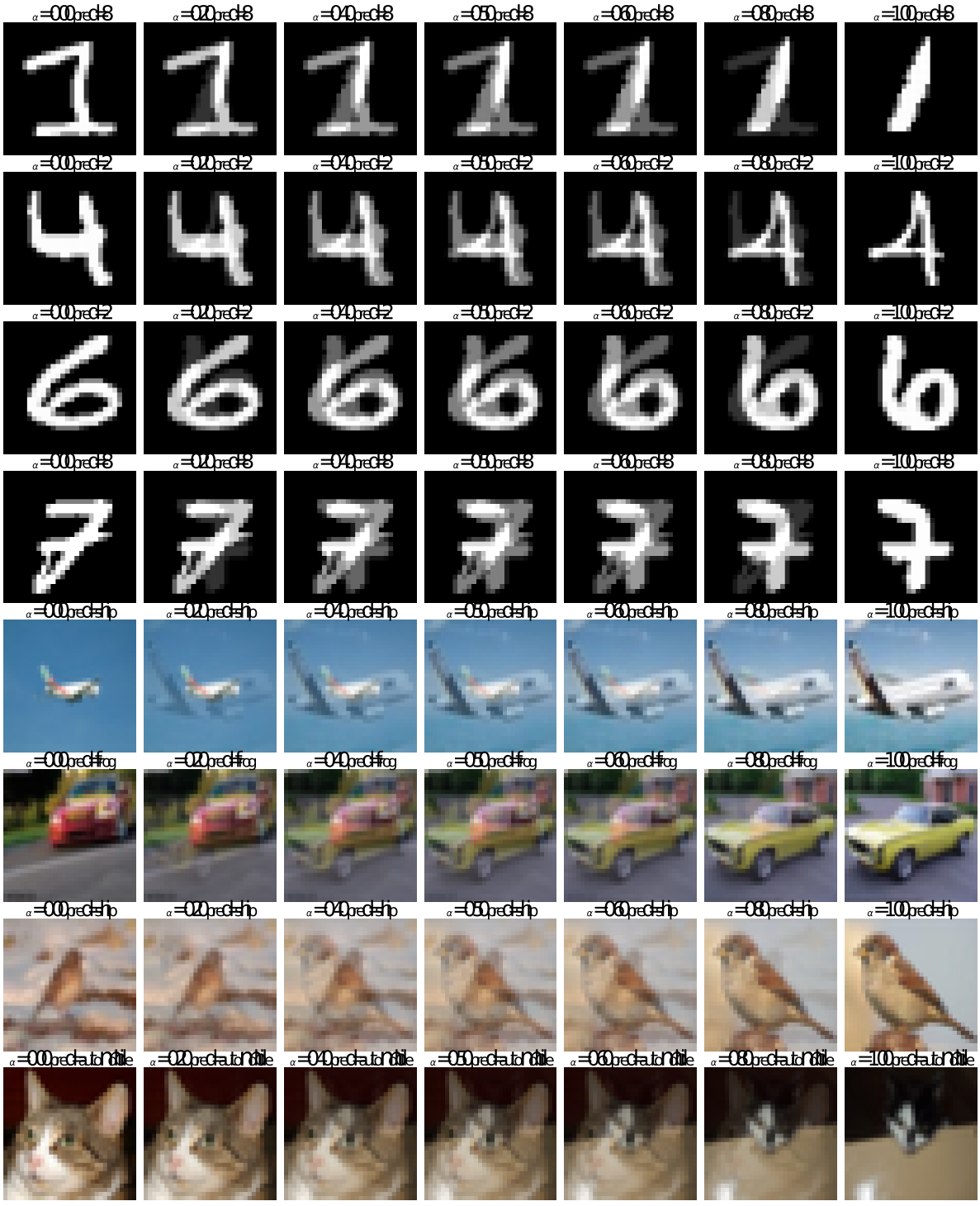


Figure 5.5: The left-most and right-most images are from the original MNIST (first five rows) and CIFAR10 (the last five rows) dataset, whereas synthesized/interpolated images are in the middle. For instance, the middle image in the first row is generated by adding $(\alpha = 0.5)$ times the left image to $(1 - \alpha)$ times the right image. The synthesized/interpolated images have the predictions different from the ground truth.

To evaluate this interpolation robustness property, we conducted experiments using samples from the MNIST and CIFAR-10 datasets. We identified pairs from these datasets that share the same NAPs and then interpolated each pair to generate 1000 images. This involved selecting 1000 α values from the interval $[0, 1]$ and performing the interpolation. We verified that all interpolations preserve the same prediction as the two images used to generate them.

In Figures 5.4 and 5.5, we showcase examples by specifically selecting 7 out of the 1000 α values, which include $\alpha = 0$ and $\alpha = 1$, representing the two reference images for interpolation. Figure 5.4 features pairs with identical NAPs and correct predictions, i.e., predictions matching the ground truth. On the other hand, Figure 5.5 includes pairs with matching NAPs and predictions that may not necessarily be correct.

However, it is important to note that to demonstrate this interpolation robustness property, we employed simple neural networks with relatively lower accuracy. Our goal was to empirically show that the theorem holds, but we cannot extend the same level of assurance to larger and more accurate neural networks. In our experiments, we used small neural networks with just 30 neurons, achieving accuracies of 32.27% and 29.6% on the MNIST and CIFAR-10 datasets, respectively. Consequently, we could easily find many qualified pairs for testing this property.

In contrast, in larger neural networks with higher accuracies, such as around 80%, we can still find some pairs demonstrating this property. However, in even larger networks with even better performance, such as 90% accuracy, we can barely find any examples of this interpolation robustness property. This is attributed to the fact that as the number of neurons increases to form the NAP, the input space becomes partitioned into more linear regions. With more neurons, it becomes increasingly challenging to find two images sharing the exact same NAP.

Although there exists a trade-off between the performance of neural networks and the interpolation robustness, we believe that there are potential ways to enhance model accuracy while maintaining these robustness guarantees. However, exploring these ways remains a topic for future work.

5.5 Discussion

In this chapter, we made a noteworthy observation regarding image data: their distribution across the input space exhibits a unique attribute known as directionality. This property implies that when images are multiplied by a positive scalar, they can preserve their original content while only adjusting the contrast or brightness levels. From a human perspective, these augmented images should still be categorized identically to the original image, resulting in the same prediction when processed through neural networks. Motivated by this observation, we delved into a novel neural network architecture known as zero-bias neural networks, involving the removal of bias terms from the network structure. Through both theoretical analysis and empirical validation, we demonstrated that these zero-bias neural networks can achieve scalar invariance, aligning with the notion that images should yield the same prediction after being multiplied by a positive scalar. Furthermore, we showed that zero-bias networks can achieve a comparable level of performance to normal with-bias neural networks, and at the same time, they also exhibit robustness guarantees that are rarely found in normal with-bias neural networks. This aspect enhances our confidence in the potential of zero-bias networks and underscores the importance of exploring them further in future research work.

The content discussed in this chapter is part of the broader work outlined in [12]. This comprehensive work includes a deeper analysis of scalar invariant zero-bias neural networks, looking into aspects such as the fairness of the zero-bias models, the training dynamics involved, and the expressiveness of these models. These aspects collectively provide evidence of the potential value inherent in zero-bias neural networks.

Chapter 6

Conclusion and Future Work

Understanding neural network properties and behaviours is crucial in modern machine learning research. As these models are increasingly deployed in critical applications, researchers are motivated by the need for improved reliability, generalization, and safety. This pursuit aims to reduce the gap between neural networks' remarkable performance and our understanding of their inner workings, ultimately leading to more robust, trustworthy, and explainable AI systems. In this thesis, we have approached the study of neural network behavior and properties from two perspectives. Firstly, we examined existing properties of neural networks in Chapters 3 and 4. Secondly, we developed novel properties inspired by observations in image domain data, as presented in Chapter 5. These explorations allow us to both deepen our understanding of established neural network characteristics and explore new avenues for enhancing their performance and interpretability.

In Chapter 3, we explored neural network behaviours using NAPs, employing a custom mining algorithm on a simple FNN with the MNIST dataset. The experiments demonstrate the trade-off between NAP specificity and test sample coverage, while showcasing NAPs' flexibility in input space coverage. This flexibility inspired potential applications in formal verification, leading to comparisons between NAP coverage and traditional ϵ-ball verification areas. The analysis reveals NAPs' broader and more flexible coverage, addressing limitations in current verification methods. Additionally, an examination of misclassified samples adhering to NAPs of wrongly predicted labels provides insights

into neural network decision-making, highlighting cases that prove challenging even from a human perspective.

Given the promising potential demonstrated by NAPs in this chapter, several areas are worthy of further exploration and improvement, offering potential future research directions. While our mining algorithm was primarily tested on relatively simple structured feedforward neural networks in this chapter, it would be intriguing to extend this analysis to neural networks with more complex layers or structures. Additionally, as highlighted in Section 3.3.3, exploring misclassified examples can provide insights into how neural networks make decisions. Furthermore, there are other scenarios we haven't covered yet, such as samples that do not adhere to NAPs from any class or samples that follow the NAP of one class but yield predictions for another class. Investigating these cases could potentially offer deeper insights into how NAPs can explain the behaviours of neural networks.

In Chapter 4, we applied *Delta Debugging* to image reduction, developing *ddImage* based on *ddmin* from [44]. Using prediction probability as the reduction criterion proved inadequate, but incorporating NAPs yielded effective minimal images, validating our Chapter 3 findings on NAPs' feature abstraction capabilities. These approaches revealed unequal pixel importance in neural network decision-making. Probability-based reduction produced unrecognizable images, exposing human-machine perception gaps and interpretability challenges. Conversely, NAP-conditioned *ddImage* preserved recognizable shapes, demonstrating NAPs' ability to capture semantically meaningful features. The contrast between methods highlights NAPs' potential for enhancing neural network interpretability. *ddImage*'s effectiveness in producing minimal yet recognizable images reinforces NAPs' role in abstracting learned features and capturing internal network representations.

Several aspects of this work hold potential for future research. The core idea of Delta Debugging—a *divide-and-conquer* process of reducing input while maintaining certain invariant properties—can be extended beyond our two cases to explore neural networks and other domains further. The varying importance of features reveals neural network vulnerabilities, while unrecognizable images raise robustness concerns. These issues sug-

gest potential applications in studying robustness against adversarial attacks, using our method to identify critical features and explore defense mechanisms. The alignment between NAP representations and human perception warrants further investigation to better understand neural networks. As discussed in Chapter 4.2.2, although different subsetting techniques did not significantly impact our results, we acknowledge the points raised by Kiss et al. [21] regarding case examination order and input format. These factors can affect subsetting details and potentially influence results. Therefore, researching optimal input subsetting methods could further enhance this work's potential.

In Chapter 5, we explored the concept of directionality in image data, highlighting their unique property where scaling by a positive scalar preserves their content while adjusting only contrast or brightness. This insight led us to investigate zero-bias neural networks, which remove bias terms to achieve scalar invariance. Through theoretical analysis and empirical validation, we confirmed that these networks maintain consistent predictions when images are scaled, demonstrating their robust scalar invariance. Furthermore, zero-bias networks showed comparable performance to traditional networks while offering improved robustness, underscoring their potential for further research and development in neural network architectures.

However, it's important to acknowledge that there are still some limitations that warrant attention in this work, paving the way for potential directions for future research. From the perspective of data augmentation, we explored the directional aspect of image data distribution by considering brightness adjustments as a means to simulate changes in environmental illumination. Multiplying a positive scalar to an image effectively alters its contrast or brightness, a concept crucial for mimicking real-world illumination shifts in scenes captured by photographs. This study holds significance in scenarios where image predictions are pivotal, such as in autonomous driving systems, where model robustness against varying illumination is paramount due to potentially costly errors. Environmental factors like time of day and weather conditions contribute to such illumination changes.

However, our approach of uniformly adjusting brightness across all pixels may not fully capture the intricacies of real-world brightness alterations. Actual changes in bright-

ness are influenced by many factors, including light angles, surface textures, colour variations, and reflections. These factors collectively influence the final appearance of an image and cannot be accurately replicated solely through uniform scalar multiplication for brightness adjustment. Hence, there is potential for future exploration aimed at enhancing model responsiveness and robustness to dynamic environmental illumination changes. This could pave the way for more effective and accurate image predictions in real-world applications.

Bibliography

[1] AKHTAR, N., AND MIAN, A. S. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access 6* (2018), 14410–14430.

[2] BÄUERLE, A., JÖNSSON, D., AND ROPINSKI, T. Neural activation patterns (naps): Visual explainability of learned concepts. *ArXiv abs/2206.10611* (2022).

[3] BROCK, A., DE, S., AND SMITH, S. L. Characterizing signal propagation to close the performance gap in unnormalized resnets. In *ICLR* (2021), OpenReview.net.

[4] BROCK, A., DE, S., SMITH, S. L., AND SIMONYAN, K. High-performance large-scale image recognition without normalization. In *ICML* (2021), vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 1059–1071.

[5] CARLINI, N., ATHALYE, A., PAPERNOT, N., BRENDEL, W., RAUBER, J., TSIPRAS, D., GOODFELLOW, I. J., MADRY, A., AND KURAKIN, A. On evaluating adversarial robustness. *CoRR abs/1902.06705* (2019).

[6] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255.

[7] DENG, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine 29*, 6 (2012), 141–142.

[8] DEVLIN, J., CHANG, M., LEE, K., AND TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR abs/1810.04805* (2018).

[9] DIFFENDERFER, J., BARTOLDSON, B. R., CHAGANTI, S., ZHANG, J., AND KAILKHURA, B. A winning hand: Compressing deep networks can improve out-of-distribution robustness. *CoRR abs/2106.09129* (2021).

[10] DIFFENDERFER, J., AND KAILKHURA, B. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning A randomly weighted network. *CoRR abs/2103.09377* (2021).

[11] GENG, C., LE, N., XU, X., WANG, Z., GURFINKEL, A., AND SI, X. Towards reliable neural specifications. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (2023), A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202 of *Proceedings of Machine Learning Research*, PMLR, pp. 11196–11212.

[12] GENG, C., XU, X., YE, H., AND SI, X. Scalar invariant networks with zero bias, 2023.

[13] GLOROT, X., AND BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010* (2010), Y. W. Teh and D. M. Titterington, Eds., vol. 9 of *JMLR Proceedings*, JMLR.org, pp. 249–256.

[14] GRAILLAT, S., JÉZÉQUEL, F., PICOT, R., FÉVOTTE, F., AND LATHUILIÈRE, B. Auto-tuning for floating-point precision with discrete stochastic arithmetic. *J. Comput. Sci. 36* (2019).

[15] GUIDOTTI, R., MONREALE, A., TURINI, F., PEDRESCHI, D., AND GIANNOTTI, F. A survey of methods for explaining black box models. *CoRR abs/1802.01933* (2018).

[16] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition, 2015.

[17] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition, 2015.

[18] HESSE, R., SCHAUB-MEYER, S., AND ROTH, S. Fast axiomatic attribution for neural networks. *CoRR abs/2111.07668* (2021).

[19] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML* (2015), vol. 37 of *JMLR Workshop and Conference Proceedings*, JMLR.org, pp. 448–456.

[20] KATZ, G., HUANG, D. A., IBELING, D., JULIAN, K., LAZARUS, C., LIM, R., SHAH, P., THAKOOR, S., WU, H., ZELJIC, A., DILL, D. L., KOCHENDERFER, M. J., AND BARRETT, C. W. The marabou framework for verification and analysis of deep neural networks. In *CAV (1)* (2019), vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 443–452.

[21] KISS, A., HODOVÁN, R., AND GYIMÓTHY, T. Hddr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (New York, NY, USA, 2018), A-TEST 2018, Association for Computing Machinery, p. 16–22.

[22] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-100 (canadian institute for advanced research).

[23] LI, Y., ZHU, C., RUBIN, J., AND CHECHIK, M. Precise semantic history slicing through dynamic delta refinement. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), pp. 495–506.

[24] MENG, M. H., BAI, G., TEO, S. G., HOU, Z., XIAO, Y., LIN, Y., AND DONG, J. S. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *CoRR abs/2206.12227* (2022).

[25] NAIR, V., AND HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel* (2010), J. Fürnkranz and T. Joachims, Eds., Omnipress, pp. 807–814.

[26] OLAH, C., MORDVINTSEV, A., AND SCHUBERT, L. Feature visualization. *Distill* (2017). https://distill.pub/2017/feature-visualization.

[27] PARNIN, C., AND ORSO, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, Association for Computing Machinery, p. 199–209.

[28] PATERSON, C., WU, H., GRESE, J., CALINESCU, R., PASAREANU, C. S., AND BARRETT, C. W. Deepcert: Verification of contextually relevant robustness for neural network image classifiers. In *Computer Safety, Reliability, and Security - 40th International Conference, SAFECOMP 2021, York, UK, September 8-10, 2021, Proceedings* (2021), I. Habli, M. Sujan, and F. Bitsch, Eds., vol. 12852 of *Lecture Notes in Computer Science*, Springer, pp. 3–17.

[29] RAJPURKAR, P., IRVIN, J., ZHU, K., YANG, B., MEHTA, H., DUAN, T., DING, D. Y., BAGUL, A., LANGLOTZ, C. P., SHPANSKAYA, K. S., LUNGREN, M. P., AND NG, A. Y. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *CoRR abs/1711.05225* (2017).

[30] RUBIO-GONZÁLEZ, C., NGUYEN, C., NGUYEN, H. D., DEMMEL, J., KAHAN, W., SEN, K., BAILEY, D. H., IANCU, C., AND HOUGH, D. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013* (2013), W. Gropp and S. Matsuoka, Eds., ACM, pp. 27:1–27:12.

[31] SAXE, A. M., MCCLELLAND, J. L., AND GANGULI, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings* (2014), Y. Bengio and Y. LeCun, Eds.

[32] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res. 15*, 1 (2014), 1929–1958.

[33] SUNEJA, S., ZHENG, Y., ZHUANG, Y., LAREDO, J. A., AND MORARI, A. Probing model signal-awareness via prediction-preserving input minimization. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021* (2021), D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds., ACM, pp. 945–955.

[34] SUNEJA, S., ZHENG, Y., ZHUANG, Y., LAREDO, J. A., AND MORARI, A. Towards reliable AI for source code understanding. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021* (2021), C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 403–411.

[35] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S. E., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. *CoRR abs/1409.4842* (2014).

[36] ULYANOV, D., VEDALDI, A., AND LEMPITSKY, V. S. Instance normalization: The missing ingredient for fast stylization. *CoRR abs/1607.08022* (2016).

[37] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. *CoRR abs/1706.03762* (2017).

[38] VNNCOMP. Vnncomp, 2021.

[39] WAMBUGU, G. M., AND MWITI, K. Automatic debugging approaches: A literature review.

[40] WANG, D., KHOSLA, A., GARGEYA, R., IRSHAD, H., AND BECK, A. H. Deep learning for identifying metastatic breast cancer, 2016.

[41] WANG, L., WANG, C., LI, Y., AND WANG, R. Explaining the behavior of neuron activations in deep neural networks. *Ad Hoc Networks 111* (2021), 102346.

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

[43] ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier Science, 2006.

[44] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.

[45] ZHANG, H., DAUPHIN, Y. N., AND MA, T. Fixup initialization: Residual learning without normalization. In *ICLR (Poster)* (2019), OpenReview.net.

[46] ZHANG, Y., TIŇO, P., LEONARDIS, A., AND TANG, K. A survey on neural network interpretability. *CoRR abs/2012.14261* (2020).